SPIHT implemented in a XC4000 device

Jörg Ritter, Görschwin Fey, Paul Molitor

Abstract— **In this paper we present an efficient FPGA implementation of the 'Set Partitioning in Hierarchical Trees' (SPIHT) algorithm of Said and Pearlman [1] in combination with an arithmetic coder. The FPGA implementation is applied within a partitioned approach for wavelet-based lossy image compression [2]. The basic SPIHT algorithm uses dynamic data structures that make a hardware realization difficult. We illustrate in detail how these dynamic data structures can be implemented in the FPGA without the use of external memory. We present a hardware realization which can be run with a frequency of 40 MHz in a Xilinx XC4000 device. The design requires 23% less internal memory as the recently published algorithm 'SPIHT Image Compression without Lists' of Wheeler and Pearlman [5].**

I. Introduction

Discrete wavelet transformation (DWT) followed by embedded zero-tree encoding is a very efficient technique for image compression [4], [1]. DWT of an image results in a multi-scale representation of the image. Normally, the coefficients of the lowest frequency bands concentrate almost all of the wavelet transformed images energy and the high frequency bands of different scales and orientations show a strong self similarity. These properties of DWT images can be exploited for efficient coding. A coding algorithm specially developed for DWT transformed images is the **s**et **p**artitioning **i**n **h**ierarchical **t**rees algorithm (*SPIHT*) which has been presented by Said and Pearlman in 1996 [1]. The basic *SPIHT* algorithm as it has been presented by Said and Pearlman makes intensive use of dynamic data structures to exploit the self similarities just mentioned. This makes a hardware implementation of the basic *SPIHT* algorithm very difficult. Here we meet the challenge to present an efficient FPGA implementation with negligible memory requirements. Compared to the recently published version of *SPIHT* without lists [5] we need 23 percent less internal memory, even without the partitioned approach.

The paper is structured as follows. In Section II, we give a brief description of the basic *SPIHT* algorithm and introduce all notations and terms necessary to understand our approach, which we discuss in Section II-B. The modified *SPIHT* algorithm is compared to the basic one in Section II-C. The hardware implementation in a Xilinx XC4000 device is presented in Section III. Our choice of the probability models and our implementation of an arithmetic coder is explained in Section III-B.

II. The SPIHT algorithm and our modifications

The *SPIHT* algorithm can be applied to grey-scale and colored images. It is applicable for lossless and lossy compression. The visual quality and the compression results are excellent. Furthermore, the algorithm produces an embedded stream, i.e., the most important information to restore an image comes first. This is an important property for compression algorithms to be effective in transmissions of compressed images over networks because it makes previews of different qualities available. The SPIHT algorithm is applied to wavelet-transformed images. The transformation reduces the correlation between neighboring pixels. The energy of the original image is concentrated in the lowest frequency band of the transformed image. Additionally self similarities between different scales which result from the recursive application of the wavelet transformation steps to the low frequency band can be observed. Consequently, based upon these facts good compression performance can be achieved if those coefficients are transmitted first which represent most of the image energy.

A. Basic idea of SPIHT

In order to exploit the addressed self similarities during the coding process, oriented trees of outdegree four are taken for the representation of a wavelet transformed image. Each node of the trees represents a coefficient of the transformed image. The levels of the trees consist of coefficients at the same scale. The trees are rooted at the highest scale of the representation. The *SPIHT*-algorithm assumes that each coefficient c_{ij} is a good predictor of the coefficients which are represented by the subtree rooted by c_{ij}. The overall procedure is controlled by an attribute, which gives information on the significance of the coefficients. More formally, a coefficient of the wavelet transformed image is *insignificant with respect to a threshold* t if its magnitude is smaller than 2^t. Otherwise it is called *significant with respect to the threshold* t. A set $\mathcal{T}$ of coefficients or a whole tree $\mathcal{T}$ is called *insignificant with respect to* t if all its elements or nodes, respectively, are insignificant with respect to t. Thus, function S_t defined by $S_t(\mathcal{T}) = 1 \Longleftrightarrow \max_{(i,j)\in\mathcal{T}} |c_{i,j}| \geq 2^t$ characterizes the significance of $\mathcal{T}$.

In the basic *SPIHT* algorithm, the coefficients of a wavelet-transformed image are classified in three sets, namely the **l**ist *LIP* of **i**nsignificant **p**ixels which contains the coordinates of those coefficients which are insignificant with respect to the current threshold t, the **l**ist *LSP* of **s**ignificant **p**ixels which contains the coordinates of those coefficients which are significant with respect to t, and the **l**ist *LIS* of **i**nsignificant **s**ets which contains the coordinates of the roots of insignificant subtrees. During the compression procedure, the sets of coefficients in *LIS* are refined and if coefficients become significant they are moved from *LIP* to *LSP*.

We consider an image with N rows and N columns ($N = 2^r, r \in \mathcal{N}$) and assume a d-bit grey-scale resolution. For all m with $0 < m \leq \log_2 N$, let $\mathcal{H}$ $(= \mathcal{H}(m))$ be the set of the coordinates of the tree roots after m wavelet transformation steps, i.e., $\mathcal{H} := \{(i,j) \mid 0 \leq i,j < \frac{N}{2^{m-1}}\}$. Thus, after $\log_2 N$ transformation steps, $\mathcal{H} = \{(0,0),(0,1),(1,0),(1,1)\}$ always holds. However, note that the coefficient at coordinate $(0,0)$ has no children, as this coefficient represents the lowest frequency band. Let $\mathcal{O}(i,j) := \{(2i,2j),(2i,2j+1),(2i+1,2j),(2i+1,2j+1)\}$ be the set of the coordinates of the children of node

Institut of Computer Science, Martin-Luther-University Halle-Wittenberg, Germany, email: ritter@informatik.uni-halle.de

(i,j) (in the case that node (i,j) has children), $\mathcal{D}(i,j)$ be the set of the descendants of node (i,j), and $\mathcal{L}(i,j) := \mathcal{D}(i,j)\backslash\mathcal{O}(i,j)$ be the set of the descendants excluding the four children. Each element of *LIS* will have as attribute either A or B. If the type of an entry $(i,j) \in$ *LIS* is A, then the entry (i,j) represents set $\mathcal{D}(i,j)$. If the type is B, then it represents $\mathcal{L}(i,j)$.

The decoder duplicates the execution path of the encoder. To ensure this behavior, the coder sends the result of a binary decision to the decoder before a branch is taken in the algorithm. Thus, all decisions of the decoder are based on the received binaries. The pseudo-code of the basic *SPIHT*-algorithm is shown in Figure 1. It can be divided into three parts. The first part initializes the lists and computes the logarithm of the maximal coefficient value k_{max}. It is easy to see that the bit planes $k_{max}+1,\ldots,d-1$ need not be processed. The sorting phase and the refinement phase are executed k_{max}+1 times starting with bit plane k_{max} down to bit plane 0. In pass k, the sorting phase first outputs the kth bits of the coefficients of *LIP*. If a coefficient $c_{i,j}$ becomes significant, the sign has to be output and $c_{i,j}$ is moved to *LSP*. After this step, the elements of *LIS* are processed in an analogous manner. The refinement phase outputs the bits of the elements of list *LSP*, respectively. For more details we refer to [1].

B. Modified SPIHT

In this section we present the modifications necessary to obtain an efficient hardware implementation of the *SPIHT* compressor. At first, we have to notice that the codec cannot be applied to the whole image at once. A partitioned approach in which each partition is wavelet transformed and compressed, separately, has to be used. Here we use a method based on the partitioned approach to wavelet transform an image which has been recently presented in [2]. Furthermore we exchange the sorting phase with the refinement phase to save memory for status information. However, the greatest challenge is the hardware implementation of the three lists *LIP*, *LSP*, and *LIS*.

In contrast to the usually performed wavelet-based compression we use a partitioned approach to wavelet transform images [2] before the algorithm of Said and Pearlman is applied. This results in a design in which only small 16 × 16 bit sub-images are stored in FPGA-memory, massive parallelism is introduced, and the dynamic range of the data structures is reduced in a significant manner. The transformation of a $q \times q$ sub-image with the used CDF(2,2) wavelet [4] which is a integer-to-integer wavelet induces growing bit widths at higher scales. The bit width encountered during the transformation of a 16 × 16 sub-image with 8 bit grey-scale resolution is at most 11 bit.

In the basic *SPIHT*-algorithm status information has to be stored for the elements of *LSP* specifying whether the corresponding coefficient has been added to *LSP* in the current iteration of the sorting phase (see Line 20 in Figure 1). In the worst case all coefficients become significant in the same iteration. Consequently, we have to provide a memory capacity of q^2 bits to store this information. However, if we exchange the sorting and the refinement phase, we do not need to consider this information anymore. The compressed data stream is still decodable and it is not increased in size. Of course, we have to consider that there is a reordering of the transmitted bits during an iteration. We will take a closer look at this problem in Section II-C.

```
(0) Initialization
    compute and output k = ⌊log_2 max_{(i,j)|0≤i,j<N} |c_{i,j}|⌋; set
    LSP := ∅ and LIP := H;
    add those elements (i,j) ∈ H with D(i,j) ≠ ∅ to LIS
    as type-A entries
(1) Sorting phase
(2)   foreach (i,j) ∈ LIP
(3)     output S_k(i,j);
(4)     if S_k(i,j) = 1 then
          move (i,j) to LSP and
          output the sign of c_{i,j}
(5)   foreach (i,j) ∈ LIS
(6)     if (i,j) is of type A then
(7)       output S_k(D(i,j));
(8)       if S_k(D(i,j)) = 1 then
(9)         foreach (e,f) ∈ O(i,j)
(10)          output S_k(e,f);
(11)          if S_k(e,f) = 1 then
                add (e,f) to LSP and
                output the sign of c_{e,f}
(12)          else
                append (e,f) to LIP;
(13)        if L(i,j) ≠ ∅ then
(14)          move (i,j) to the end of LIS as an entry
              of type B and go to step (5)
(15)        else remove (i,j) from LIS
(16)    if (i,j) is of type B then
(17)      output S_k(L(i,j));
(18)      if S_k(L(i,j)) = 1 then
(19)        append each (e,f) ∈ O(i,j) to LIS as an entry
            of type A and remove (i,j) from LIS;
(20) Refinement phase
    foreach (i,j) ∈ LSP except those included in the last
    sorting pass, output the k-th bit of |c_{i,j}|;
(21) if k=0 then end; else decrement k; goto step (1)
```

Fig. 1. The basic SPIHT-algorithm

To obtain an efficient realization of the lists *LIP*, *LSP* and *LIS*, we first have to specify the operations that take place on these lists and deduce worst case space requirements in a software implementation.

For *LIS*, we have to provide the operations "initialize as empty list" (Line 0), "append an element" (Line 0 and 19), "sequentially iterate the elements" (Line 5), "delete the element under consideration" (Line 15 and 19), and "move the element under consideration to the end of the list and change its type" (Line 14). Note that the initialization phase cannot be ignored because there are several partitions to compress. Dependent on the realization chosen the costs of the initialization phase can be linear in the number of elements. The estimation of the space requirement for *LIS* is quite easy because *LIS* is an independent set. This can be easily proved by complete induction [3]. Note that if the list is longest, then all its elements are of type A. Consequently, the number of elements of *LIS* is smaller than or equal to the number of nodes in the next to the last tree level. Thus, *LIS* contains at most $\left(\frac{q}{2}\right)^2 - \left(\frac{q}{4}\right)^2 = \frac{3}{16}q^2$ elements. This results in an overall space requirement of $\frac{3}{16}q^2 \cdot (2\log_2(\frac{q}{2})+1)$ bits.

We consider both the list *LIP* and the list *LSP* together because they can be implemented by one data structure. Again, we start with the operations applied to both lists. These are "initialize as empty list" (Line 0), "append an element" (Line 0, 4, 11, and 12), "sequentially iterate the elements" (Line 20), and "delete the element under consideration from *LIP*" (Line 4). The maximum size of both lists is at most q^2. Furthermore, it holds, that *LIP* $\cap$ *LSP* $= \emptyset$. Thus, $|LIP| + |LSP| \leq q^2$ also

holds. q^2 is a tight upper bound for $|LIP| + |LSP|$. An example, where the worst case is attained, is given in [3]. Thus, the overall space requirement for both lists is $q^2 \cdot 2\log_2 q$ bits. To reduce the memory requirement for the list data structures in the worst case, we implement the lists as bitmaps. Note that by this approach we loose the ordering information within the lists *LIP*, *LSP*, and *LIS*. The effect of this reordering with respect to the decoded image will be discussed in detail in Section II-C.

(0) **Initialization**
  compute and output $k := \lfloor \log_2 \max_{\{(i,j)|0 \le i,j < N\}} |c_{i,j}| \rfloor$;
  foreach $0 \le i,j \le N-1$ set
   $LSP(i,j) = 0$
   $LIP(i,j) = \begin{cases} 1, & \text{if } (i,j) \in \mathcal{H} \\ 0, & \text{else} \end{cases}$
  foreach $0 \le i,j \le \frac{N}{2}-1$ set
   $LIS(i,j) = \begin{cases} A, & \text{if } (i,j) \in \mathcal{H} \text{ and } \mathcal{O}(i,j) \neq \emptyset \\ 0, & \text{else} \end{cases}$

(1) **Refinement and sorting phase for list LIP**
(2) for $i = 0 \ldots N-1$
(3)   for $j = 0 \ldots N-1$
(4)   if $LSP(i,j) = 1$ then
     output the k-th bit of $|c_{i,j}|$;
(5)   if $LIP(i,j) = 1$ then
(6)   output $S_k(i,j)$;
(7)   if $S_k(i,j) = 1$ then
      $LSP(i,j) = 1$; $LIP(i,j) = 0$;
      output the sign of $c_{i,j}$;

(8) **Sorting phase for list LIS**
(9) for $i = 0 \ldots \frac{N}{2}-1$
(10)   for $j = 0 \ldots \frac{N}{2}-1$
(11)   if $LIS(i,j) = A$ then
(12)   output $S_k(\mathcal{D}(i,j))$;
(13)   if $S_k(\mathcal{D}(i,j)) = 1$ then
(14)   foreach $(e,f) \in \mathcal{O}(i,j)$
(15)   output $S_k(e,f)$;
(16)   if $S_k(e,f) = 1$ then
        $LSP(e,f) = 1$;
        output the sign of $c_{e,f}$;
(17)   else $LIP(e,f) = 1$;
(18)   if $\mathcal{L}(i,j) \neq \emptyset$ then
(19) $LIS(i,j) = B$;
(20)   else $LIS(i,j) = 0$;
(21)   if $LIS(i,j) = B$ then
(22)   output $S_k(\mathcal{L}(i,j))$;
(23)   if $S_k(\mathcal{L}(i,j)) = 1$ then
(24)   (foreach $(e,f) \in \mathcal{O}(i,j)$) $LIS(e,f) = A$;
(25) $LIS(i,j) = 0$;
(26) if k=0 end; else decrement k; goto step (1);

Fig. 2. The modified SPIHT-algorithm

We provide two synchronous single-port RAM modules for the three lists. The lists *LIP* and *LSP* are realized in one RAM module, the list *LIS* in the other one. The RAM module which realizes *LIP* and *LSP* has a configuration of $q \times q$ entries of bit length 2 as for each pixel $c_{i,j}$ of the $q \times q$ image either $(i,j) \in$ *LIP* or $(i,j) \in$ *LSP* or $(i,j) \notin$ *LIP* $\cup$ LSP holds. The second RAM module implements *LIS* and is realized by a bitmap with a configuration of $\frac{q}{2} \times \frac{q}{2}$ entries of bit length 1 and 2 (for a subset of $\frac{q}{4} \times \frac{q}{4}$). Of course, with logarithmic coding we could further reduce the RAM size but at the cost of a much more complex control mechanism. To sum it up, the list *LSP* $\cup$ *LIP* and the list *LIS* have size 512 bit and 80 bit, respectively, for partition size 16, i.e., $q = 16$.

C. Comparison of the algorithms

The modifications made ensure that the compressed stream can still be decoded and that exactly the same number of bits are produced. However, the reordering of the embedded stream which is due to the realization of the lists *LIP*, *LSP*, and *LIS* by bitmaps can have an effect on the visual quality of the reconstructed images.

Let $\mathcal{I}$ and $\tilde{\mathcal{I}}$ denote the original and the reconstructed image, respectively. The basic *SPIHT* algorithm and our modified one can be compared by — I $E(\mathcal{I}, \tilde{\mathcal{I}}, b)$, the difference between the *SPIHT* algorithm and the modified algorithm after coding b bits. There are two main cases to consider. Figure II-C illustrates the

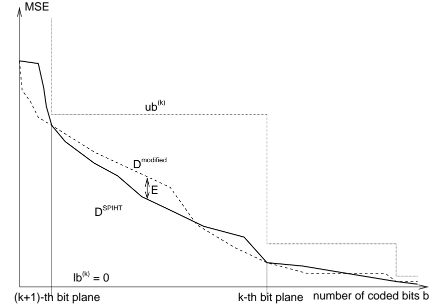

Fig. 3. Idealized illustration, lb, up are the lower/upper bound

following considerations. First, we have to compare the original and the reconstructed image after a complete bit plane is coded. Because the same information is produced by both algorithms (even though in different order), $E(\mathcal{I}, \tilde{\mathcal{I}}, b) = 0$ holds. However, the more interesting case is the difference of the two algorithms during the coding of a bit plane. The maximum difference during the coding of the kth bit plane can be estimated with lower and upper bounds $lb^{(k)}$ and $up^{(k)}$. Obviously, the lower bound $lb^{(k)}$ is zero. This is a tight lower bound. Fey [3] proves $up^{(k)} = 4^k$. Note that this is a rough estimate because it is independent of the image under consideration. Even if this upper bound is rather high, it is very difficult to distinguish the reconstructed images with the naked eye, in general.

III. FPGA-Implementation

In this section we present the most important details of the FPGA implementation. The design was written in VHDL. We have used a PCI card (microEnable, Silicon Software GmbH [6]) equipped with a Xilinx FPGA XC4085XLA device as prototyping platform. We shortly explain the overall functionality before we present some of the modules in detail. Each partition of the wavelet-transformed image is transfered once to an internal memory module. At first, the initialization of the bitmaps representing *LIP*, *LSP* and *LIS* and the computation of the significances is done in parallel. The significances of sets/trees are computed for all thresholds $t \le k_{max}$ at once and are stored in modules named 'SL' and 'SD', respectively (see Section III-A). With this information the compression can be started with bit plane k_{max}. The different finite state machines control the overall procedure. The compressed data is transfered to the local SRAM on the PCI card or directly to the PC memory. Additionally, an arithmetic coder can be configured into the design to further reduce the compression ratio (see Section III-B).

A. Efficient computation of significances

In the following we will concentrate on the module which computes the significances. The significance of an individual coef-

ficient is trivial to compute. Just select the kth bit of $|c_{i,j}|$ in order to obtain $S_k(i,j)$. However, it is much more difficult to efficiently compute the significance of sets for all thresholds in parallel. In the following we use the notation $S^*(\mathcal{T})$ which gives the maximum threshold for which some coefficient in set $\mathcal{T}$ becomes significant. Once $S^*(\mathcal{T})$ is computed for all sets $\mathcal{L}$ and $\mathcal{D}$, we have preprocessed the significances of sets for all thresholds. In order to do this, we use the two RAM modules SL and SD. $\mathcal{L}$ and $\mathcal{D}$ are organized as $\frac{N}{4} \times \frac{N}{4} \times \lceil \log_2 k_{max} + 1 \rceil$ and $\frac{N}{2} \times \frac{N}{2} \times \lceil \log_2 k_{max} + 1 \rceil$ memory, respectively. The computation is done bottom up in the hierarchy defined by the spatial oriented trees. The entries of both RAMs are initialized with zero. Now, let (e,f) be a coordinate with $0 < e,f < N$ just handled by the bottom up process and let $(i,j) = (\lfloor \frac{e}{2} \rfloor, \lfloor \frac{f}{2} \rfloor)$ be the parent of (e,f) if it exists. Then SD and SL have to be updated by the following process:

1) if $e < \frac{N}{2}, f < \frac{N}{2}$ then $SL(i,j) := \max\{SL(i,j), SD(e,f)\}$
2) $SD(i,j) := \max\{SD(i,j), S^*(e,f)\}$
3) if $e < \frac{N}{2}, f < \frac{N}{2}$ then $SD(i,j) := \max\{SD(i,j), SD(e,f)\}$

The bottom up process can be done in linear time using a simple address counter.

B. Arithmetic Coding

Our design can optionally be configured with an arithmetic coder that processes the data which is output by the *SPIHT* algorithm. We decided to utilize the coder of Feygin, Gulak and Chow [7]. It is very suitable for FPGA design because it does not contain multiplication operations and is not based on floating point arithmetic. The implementation is straightforward.

In this context, the most interesting part was the development of probability models adapted to the *SPIHT* compressor. To develop accurate probability models, we have investigated the cumulative percentage of the different output positions in the algorithm in detail. Note that in our modified algorithm (see Figure 2) there are exactly eight lines (Line 0, 4, 6, 7, 12, 15, 16, 22) which output some data. For each of these output positions we provide one probability model and call them $\mathcal{A}_1, \ldots \mathcal{A}_8$ (e.g. model $\mathcal{A}_4$ correspond to Line 7). In order to obtain meaningful models, we have counted the number of ones and zeros with respect to their output position while compressing our set of benchmarks images. Then we have expressed these cumulative percentages using conditional probabilities, with a history of maximum size four.

The VHDL implementation of these probability models have been specified by finite state machines, too. The compressed stream of the *SPIHT* algorithm is first stored in a circular buffer of length 4. A signal *type*, which determines the model, is provided from outside. It selects the corresponding probability model. A module named 'carry chain' implements the so called *bit stuffing* for the potential carry overs in the arithmetic coder [8]. Furthermore, there exist signals to insert escape sequences and to stop the compression at the specified bit rate.

C. Experimental results

We have implemented two versions of our design, one without and one with the arithmetic coder. We have achieved clock rates of 40MHz for both implementations. Note that our VHDL designs were synthesized without manual optimizations. All basic memory and arithmetic modules were generated with Xilinx tools. Our implementations take 743 and 1425 logic blocks of the Xilinx device, respectively.

In order to compare the software implementation with our presented FPGA design we measured the execution time of both. To obtain a faithful measurement of the hardware execution time we have included a counter (resolution 25ns) into the design. We could improve the compression time of $512 \times 512 \times 8$ bit greyscale images by a factor of 10 in comparison to an AMD 1GHz Athlon processor.

The effect of the arithmetic coder upon the compression ratio is shown in Table I. The coder compresses the *SPIHT* output by further 2 to 4 percent. It is remarkable, that the compression ratio is always improved, in the lossless as well as in the lossy case.

TABLE I

INFLUENCE OF THE ARITHMETIC CODER

image	baboon	barbara	goldhill	lena	peppers
0.5 bpp	95.9	98.6	97.4	97.3	95.7
lossless	97.8	97.3	96.8	96.0	96.1

IV. CONCLUSIONS

We have presented an efficient FPGA design with negligible memory requirements. In comparison to [5] we could reduce the internal memory from $(4 + \frac{5}{16}d')N^2$ to $(\frac{37}{16} + \frac{5}{16}d')N^2$ bit. It is remarkable, that in conjunction with the partitioned approach we need $(\frac{37}{16} + \frac{5}{16} \cdot 11) \cdot q^2 = 0.184$ kbytes only compared to $(4 + \frac{5}{16} \cdot 11) \cdot N^2 = 238$ kbyte of the algorithm 'SPIHT Image compression without Lists' on the whole image (bit-width $d' = 11$ bit, $N = 512, q = 16$). Thus the design should fit many times in a newer FPGA device (especially if block RAM is available) to achieve high data throughput rates.

REFERENCES

[1] A. Said and W. A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. In *Trans. Signal Processing*, volume 5, no.9, pages 1303–1310. IEEE, 1996.
[2] J. Ritter and P. Molitor. A partitioned wavelet-based approach for image compression using FPGA's. In *Proceedings of the 2000 Custom Integrated Circuits Conference*, pages 547–550. IEEE, 2000.
[3] G. Fey. Set partitioning in hierarchical trees: A FPGA - implementation. Master thesis (in German), Martin-Luther-University Halle, Germany, 2001.
[4] I. Daubechies. *Ten lectures of wavelets*. SIAM, Philadelphia PA, 1992.
[5] F. W. Wheeler, W. A. Pearlman. SPIHT Image Compression without Lists In *Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP 2000)*, IEEE, 2000
[6] Silicon Software GmbH, microEnable, PCI Prototyping Card, User Guide, 1999
[7] G. Feygin, P. G. Gulak, P. Chow. Minimizing Error and VLSI Complexity in the Multiplication Free Approximation of Arithmetic Coding In *Transactions on Signal Processing*, IEEE, 1993
[8] J. Rissanen, G. G. Langdon. Universal Modeling and Coding In *Transactions on Information Theory, Vol. II 27*, IEEE, 1981