INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2024, VOL. 70, NO. 1, PP. 219–228
Manuscript received October 1, 2023; revised March, 2024. DOI: 10.24425/ijet.2024.149534

Low-cost ANS encoder for lossless data compression in FPGAs

Magdalena Pastuła, Paweł Russek, and Kazimierz Wiatr

***Abstract*—We present the implementation of the hardware ANS compressor in FPGAs. The main goal of the design was to propose a solution suitable to low-cost, low-energy embedded systems. We propose the streaming-rANS algorithm of the ANS family as a target for the implementation. Also, we propose a set of algorithm parameters that substantially reduce the use of FPGA resources, and we examine what is the influence of the chosen parameters on compression performance. Further, we compare our design to the lossless codecs found in literature, and to the streaming-rANS codecs with arbitrary parameters.**

***Keywords*—Lossless compression; Asymmetric Numeral Systems (ANS); Hardware codecs; FPGA**

I. Introduction

NOWADAYS, information is increasingly valuable asset. Everyone is aware that large data sets are used by AI systems for new model development, but huge data volumes are also analysed in the traditional way to find new patterns of behaviour. Storage of this amount of data is expensive both in terms of hardware resources and energy consumption. Moreover, transferring such a massive collection of data between devices requires sufficient bandwidth available. In short, the more data is sent, the more time it takes, and more power (electrics and computing) is required to process it on the way.

The common solution to mitigate the big data handling problem is data compression of course. Data can be stored and transferred in compressed form and decompressed when analysis is required. These scenarios create demand for efficient, cheap, and fast compressing methods. Simultaneously, the hardware codecs can be necessary to make sure that compression and decompression are quick enough to pace up with the data read/write bandwidth.

Unlike image compression, data analytics requires lossless data compression algorithms. There are three main types of lossless compression methods: Huffman coding, arithmetic coding, and Asymmetric Numeral Systems (ANS).

Huffman coding uses prefix codes, i.e. every symbol has its unique code that is not a prefix of any other symbol code. This makes this algorithm relatively simple and fast. It can be executed in parallel, as symbols can be encoded simultaneously. However, Huffman coding is inefficient, as we cannot use the fractional number of bits per symbol and the average width of the encoded symbol grows drastically with the increase of the number of unique symbols [1].

Authors are with AGH University of Krakow, Krakow, Poland (e-mail: pastulamagdalena@gmail.com, russek@agh.edu.pl, wiatr@agh.edu.pl).

Arithmetic coding does not work with static symbols. It uses two numbers that represent a range to store the encoded data. When a new symbol has to be encoded, these two numbers are changed based on the probability of this symbol in the data. Consequently, the encoding of the next symbol is dependent on the result of the encoding of the previous ones. Thus, arithmetic encoding cannot be executed concurrently. Moreover, arithmetic coding is relatively complex and slow, but it offers a better compression ratio than Huffman encoding [2].

The method examined in our work, i.e. the ANS compression algorithm is a trade-off of the former ones. It has a speed and simplicity comparable to Huffman coding, and it offers compression performance comparable to arithmetic coding [3]. Similarly to the arithmetic algorithm, ANS outputs a big natural number. Additionally, some ANS variants produce a bitstream as an additional encoding result.

II. Motivation and Goal

Currently, the ANS is an important and widely used algorithm. Thanks to the good complexity-to-efficiency ratio, it is used in numerous applications: in JPEG XL image file format [4], or compression software used by companies: Facebook's ZSTD [4], [5], Apple's LZFSE [4], [6], or Google's Draco compression library for 3D graphic [4], [7] for example. There are many more implementations and usages and what is worth noting, ANS still has not reached its peak popularity yet. It is a subject of a growing number of articles and its new applications are still being developed.

However, implementing the ANS algorithm in energy-optimised embedded systems might be quite problematic. Although it is less complex than arithmetic encoding, it still requires processing that can use most of the resources of a small microcontroller, especially when working with substantial data size with dynamic probability distributions of symbols. In that case, the hardware codec that is customised for the algorithm is a solution. It requires reduced hardware resources that fit the algorithm needs, consumes less energy, and additionally, off-loads the CPU from the computationally exhaustive tasks. Custom codecs for the algorithm requires the design of dedicated hardware that is customized to execute such an algorithm. Usually, hardware implementations of an algorithm operate at lower clock frequencies than the main processing unit, but they are optimized to complete their task within as few clock cycles as possible, which leads to a lower overall execution time.

© The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0, https://creativecommons.org/licenses/by/4.0/), which permits use, distribution, and reproduction in any medium, provided that the Article is properly cited.

Another motivation to focus on the algorithm's hardware implementation is the better final performance of the system. As was also mentioned at the beginning, in the case of data transfer, the compression process should be fast enough to pace up with transmission speed, so that sending compressed data does not take longer than sending the original data. To ensure that the compressing process takes as little time as possible, one can accelerate its execution by running it in parallel. However, the ANS algorithm cannot be executed concurrently, because the encoding of the next symbol depends on the encoding of the previous symbol. Although data can be divided into several parts which can be encoded separately, it is less effective in terms of compression ratio.

The main purpose of this thesis is to propose and implement an efficient hardware architecture for one of the most common variants of the ANS algorithm i.e. Streaming-rANS. Streaming-rANS is introduced in Section III-B. As a good hardware-oriented algorithm should avoid certain operations that are unfortunately a part ANS (i.e. division and modulo operations), we will propose some constraints that are a solution to this obstacle. Consequently, it will be examined what is the impact of the constraints on the compression ratio and whether there are any other trade-offs of the proposed solution.

The design will be prepared in a hardware description language and validated in an FPGA-based Programmable SoC (AMD's ZYNQ PSoC). Required FPGA resources use will be presented for the different algorithm parameters, and they will be compared to other existing solutions reported in literature.

The paper structure is as follows. Section III describes briefly the most common variants of ANS - particularly the streaming-rANS which is chosen for designing a hardware implementation. Section IV reviews the found articles and depicts the current state of the topic. Mainly, it presents other existing hardware implementations of ANS. Section V describes the development of the hardware and presents assumptions that were made to design the codec. Section VI contains the experiment results and analysis of the impact of the design constraints on compression ratio. Lastly, Section VII summarizes the paper with a discussion of possible further research and development. Further modifications to the proposed design are also considered.

III. ANS FAMILY OF ALGORITHMS

The Asymmetric Numeral Systems (ANS) is an algorithm of lossless compression, and it was first published by Jarosław Duda in 2009 [8]. There are a few variants of this algorithm, but the common idea behind them is to create a finite-state machine whose next state depends on the previous state and a current symbol to encode. The states are enumerated by big positive integer numbers, and the final state of the FSM, after encoding all symbols, represents the encoded data.

The definitions should be presented first to formally introduce ANS coding.

- Symbol – the smallest element of data to encode. For text, a symbol is a letter for example. It is denoted as s_t.
- Alphabet – a set of all symbols that occur in data to encode. For text, it is the ASCII character set for example.
- Frequency of a symbol – number of occurrences of a symbol in data to encode. It is marked as F_{s_t} in equations.
- Cumulative frequency for a symbol – a sum of frequencies of all symbols that preceded s_t in the alphabet [9]. It is denoted as C_{s_t} and described by equation

$$C_{s_i} = \sum_{j=1}^{i-1} F_{s_j}. \tag{1}$$

- Size of data to encode – number of symbols in data to encode. It can be calculated as a sum of all frequencies. It will be represented by M in equations.
- State – a big integer number representing data encoded so far. After encoding of all symbols, it is a result of compression.
- Bitstream – an additional part of the compression result for the streaming-rANS and tabled-ANS variants of ANS.

The execution of each ANS variant consist of two phases: the initial phase when the symbol frequencies (F_{s_t} and C_{s_t}) are calculated, and the encoding phase when consecutive symbols of data are encoded.

The most popular variants of ANS are range-ANS, streaming-rANS, and tabled-ANS.

A. Range-ANS

Range ANS (rANS) encodes consecutive symbols in ranges, which is similar to arithmetic coding [9]. When encoding a symbol, the next state is calculated with the formula

$$X_t = \left\lfloor \frac{X_{t-1}}{F_{s_t}} \right\rfloor * M + C_{s_t} + mod(X_{t-1}, F_{s_t}), \tag{2}$$

where $mod()$ stands for the modulo operation.

To decode, the previous symbol and state can be calculated thanks to the formulas:

$$X_{t-1} = \left\lfloor \frac{X_t}{M} \right\rfloor * F_{s_t} + slot - C_{s_t}, \tag{3}$$

$$s_t = C_{INV}(slot), \tag{4}$$

$$slot = mod(X_t, M), \tag{5}$$

where

$$C_{INV}(y) = a_i, \text{ if } C_{a_i} < y < C_{a_{i+1}} \tag{6}$$

stands for inverse cumulative frequency.

As one can see, every compression step increases the bit size of the state's value. Unfortunately, the state's value increases exponentially with the size of the data to encode, as in every step it is multiplied by a factor proportional to M. This is the major disadvantage of rANS, as coding of big data sets requires a large bit-width of registers and operators to execute the algorithm and store the state value.

B. Streaming-rANS

Streaming rANS [9] is a modification of range-ANS. The encoding step uses the same formula, but before encoding the previous state is adjusted to lay in an appropriate range. This range is dependent on the frequency of the symbol to encode, and it is described as

$$I_{s_t} = [lF_{s_t}, 2^k lF_{s_t} - 1]. \tag{7}$$

This adjustment causes the state to stay in the range

$$I = [lM, 2^k lM - 1], \tag{8}$$

where l and k can be any positive integer numbers.

The previous state is adjusted to the range given by Equation 7 by dividing it by 2^k. The remainders of this integer division make up the bitstream, which is a part of the encoding result. Consequently, k is equal to the number of bits of the bitstream chunk that the encoder produces at once. Usually, k is chosen to be 8, 16, 32, or 64, as it simplifies the process of sending the encoded data.

Algorithms 1 and 2 show the pseudocode of streaming-rANS for the encoding and decoding steps accordingly. The while loops represent the state adjustment, and the remaining are the rANS operations described by equations {2–5}.

Algorithm 1 Streaming-rANS encoding step pseudocode

while $X_{t-1} \geq 2^k * l * F_{s_t}$ **do**
  $bitstream \leftarrow (bitstream << k) + mod(X_{t-1}, 2^k)$
  $X_{t-1} \leftarrow X_{t-1} >> k$
end while
$X_t \leftarrow (X_{t-1} // F_{s_t}) * M + C_{s_t} + mod(X_{t-1}, F_{s_t})$

Algorithm 2 Streaming-rANS decoding step pseudocode

while $X_t < l * M$ **do**
  $X_t \leftarrow (X_t << k) + mod(bitstream, 2^k)$
  $bitstream \leftarrow bitstream >> k$
end while
$slot \leftarrow mod(X_t, M)$
$s_t \leftarrow C_{INV}(slot)$
$X_{t-1} \leftarrow (X_t // M) * F_{s_t} + slot - C_{s_t}$

C. Tabled-ANS

The result of encoding is always the same for the given symbol and previous state. Thanks to the adjustment of the previous state to the appropriate range the number of states is limited and reasonably big in the streaming-rANS. Therefore, the next state can be pre-calculated and put into a lookup table. This is the idea that lies behind tabled-ANS. The initial phase of this algorithm includes calculating the next state for every previous state and symbol and putting it in the lookup table. Consequently, tabled-ANS requires a lot of pre-processing for every distribution of symbols in data. It makes sense if the distribution is known beforehand and does not change much for big volumes of compressed data. Moreover, a memory of size 2 to 16 times the size of the alphabet is required to store the lookup table for the size of the alphabet equal to 256 [3].

Streaming-rANS was chosen for a custom hardware processor examined in this work. The reason is that some hardware implementations of tabled-ANS are already reported in the literature (see Section IV-B), but no implementations could be found for streaming-rANS. There is also another advantage of streaming-rANS for embedded systems. The streaming-rANS and tabled-ANS encoding can be treated as an encryption algorithm, where bitstream is the data being encrypted and state is the key needed to decrypt the data [10]. This additional feature can be an important advantage in small embedded systems, as the same hardware can be used for data compression and encryption.

IV. Encoders Reported in Literature

A. Entropy encoders

Numerous hardware implementations of the Huffman and arithmetic coding are reported in the literature. For example, hardware implementation of the tree-less Huffman encoding is described in [11]. The main idea is to generate codes for some symbols based on codes for other ones, as it is often sufficient to add one or add one and shift only. This operation is performed from the symbol of the smallest width and does not require building the Huffman tree. However, a large number of resources is necessary to implement this solution, as it requires extra modules: one for counting symbol frequencies and one for code generation. What is also worth noting, each field of the frequencies array has a 32-bit width and the size of the alphabet is assumed to be 256.

Another hardware implementation of the Huffman encoder is given in [12]. This design, based on Canonical Huffman Coding, builds the static Huffman tree and reads the length of each encoding. Then, the longest encodings are changed to zeros and the next encodings are calculated by adding one to the previous ones. When encoding length changes to smaller, additional shifting is performed. This design does not include frequency counting. Additionally to the proposed codec resources, the article reports resources needed for the implementation of the original Huffman encoder. Unfortunately, the authors did not specify for what size of the alphabet the system was implemented.

TABLE I
Resources of the Encoders Reported in Literature

Design	[11]	[12]-oryg.	[12]-prop.	[13]
Slices	13,853	-	-	7,862
4-LUTs	25,224	-	-	13,781
LUTs	-	6,266	1,654	-
LUTRAMs	-	5	8	-
Flip-Flops	15,791	464	850	6,848
BRAMs	-	0	50	-

Implementation of arithmetic coding is described in [13]. It consists of two parts: a modeller – a part responsible for estimating symbols' probabilities, and a coder – a part

responsible for symbols coding. The proposed design features a modeller that stores symbols' frequencies as leaf nodes in a binary tree, and cumulated frequencies stored in the intermediate nodes. When any of the frequencies change, a specific procedure is performed to update the binary tree. The coding part is composed of binary encoders that take intermediate node and interval as arguments. The design was implemented for the alphabet size of 256.

Table I contains hardware resources needed for the implementation of the codecs reported in [11]–[13]. Parameters that are not specified in the publications are marked as '-' in the table.

B. ANS encoders

The authors did not find any paper related to streaming-rANS or rANS hardware codec implementation. However, three implementations of tabled-ANS were found.

The first implementation is described in [14]. The method proposed in this article is to calculate the width of the bitstream in the initial phase, when the encoding table is generated, instead of calculating it when encoding a symbol. As a result, two encoding tables are in use: the width of the bitstream table and next state's value table. Both tables are indexed by the address created of the current state and symbol to encode. Such a solution requires bigger memory but it runs with a higher frequency clock.

The second implementation is given in [15]. However, the ANS coding is only a part of this work. It regards an image compressor of the LOCO-ANS algorithm, meaning that the ANS encoding is accompanied by the pixel decorrelator and quantizer modules.

The third design of the tabled-ANS encoder is described in [16]. This solution splits the new state table into two parts: the first table of equal width and the second table of equal width but greater by one than the first one. To be able to split the encoding table this way, it is required to renormalize the number of occurrences of symbols so that their sum is a power of two. After this encoding process is changed to replace only part of the bits of the state, not all. Unfortunately, this article does not specify used hardware resources. It specifies a throughput for four input files only.

Table II shows resources used for the implementation of encoders given in [14], [15]. Resources given for [14] are for the alphabet of size 256, and resources given for [15] are for the implementation of the state's bit-width of 7 bits.

TABLE II
THE ANS CODERS' RESOURCES REPORTED IN LITERATURE

Design	**[14]**	**[15]**
Slices	192	-
4-LUTs	403	-
LUTs	-	4,572
Flip-Flops	105	4,373
BRAMs	3	29
DSP Blocks	-	2

Compared to other entropy encoders, it can be seen that ANS encoders generally need much fewer resources. Even the design proposed in [15], where the encoder is only a part of the solution, needs fewer resources than the original Huffman encoder presented in [12]. However, there are many hardware implementations of the Huffman coding, meaning that versions that require fewer resources can exist.

V. Custom ANS coder development

For any processor design, design constraints must be considered before the system architecture is created. For example, assumptions about the codec's input data representation, volume and possible statistical distribution have to be taken. But first and foremost, the designer should consider how to deal with operations problematic in hardware implementations.

A. Avoiding division and modulo operations

The energy-efficient and fast custom processors should avoid computationally expensive operations. In the case of streaming-rANS, division and modulo are undoubtedly the most resource-consuming and execution-time-hungry operations. Their optimisation was requested and possible. Looking closely at the formulas 7 and 8, we can see that for l and k equal to one the equations simplify as follows:

$$I_{s_t} = [F_{s_t}, 2F_{s_t} - 1], \tag{9}$$

$$I = [M, 2M - 1], \tag{10}$$

and Equation 2 can be simplified to

$$X_t = M + C_{s_t} + X_{t-1} - F_{s_t}. \tag{11}$$

The reason for that is because $\left\lfloor \frac{X_t}{F_{s_t}} \right\rfloor$ is always one, as the previous state X_t is always in the range given in Equation 9. Consequently, $mod(X_t, F_{s_t})$ can be calculated as $X_t - F_{s_t}$, as modulo operation can be defined as $mod(X_t, F_{s_t}) = X_t - \left\lfloor \frac{X_t}{F_{s_t}} \right\rfloor * F_{s_t}$.

Similarly to the encoding, the decoding formulas can be simplified to formulas:

$$slot = X_t - M, \tag{12}$$

$$s_t = C_{INV}(slot), \tag{13}$$

$$X_{t-1} = F_{s_t} + slot - C_{s_t}, \tag{14}$$

because the previous state is always within the range given by Equation 10 before executing equations 3, 4, and 5.

Those observations were used to create efficient hardware for streaming-rANS. Parameters k and l are fixed to one in our design, and the expensive division and modulo operations could be replaced by the cheap subtraction operations.

B. Input data-related assumptions

Looking at pseudocodes given in listings 1 and 2, the single step of the streaming-rANS algorithm requires five input variables:

- symbol code,
- previous state's value,
- frequency of the symbol,
- cumulative frequency of the symbol,
- size of data to encode – M.

The resulting output of the algorithm is the last state value and the bitstream.

Symbols' frequencies F_{s_t} have to be known before encoding starts, but the size of data M and cumulative frequency C_{s_t} can be calculated based on them. For this reason, the operation of the designed codec was divided into two phases: the initial phase – when the frequencies are transferred to the codec and cumulative frequencies are calculated, and the encoding phase – when the codec reads and encodes consecutive symbols of data.

In the process of the codec processor design, the number and size of the internal registers have to be determined, along with the bit-width of all the processor's input and output ports. It was assumed that the alphabet size is smaller than 256, and it is sufficient for ASCII text encoding as well as for selected image compression algorithms. In practice, data of any format can be compressed, as long as it can be read byte by byte. The bit-width of the symbol is set to 8 bits.

An important constraint to be made was the maximum size of data M_{max} to encode. According to Equation 10, the maximum value of the state is equal to $2 * M_{max} - 1$, as the maximum size of the data is equal to half of the maximum state's value. However, we prefer to control the bit-width of the state directly, not through the size of the data, to simplify the transfer or storage of encoded data – for example, internal system bus size can be a constraint. For this reason, we set up an internal bit-width of the state first, and the size of the data is its derivative in our design. Let the bit-width of the state be m, and the data size bit-width is m-1 accordingly.

The assumption about the state register bit-width propagates to the frequency and cumulative frequency bit-widths – as in the borderline case they are equal to the size of the data.

When it comes to the state bit-width, it was decided that it is a constant parameter in the design, so that it is fixed for the given hardware codec architecture, no matter the data size.

Another assumption had to be made about the bitstream output and how it is transferred to the output of the hardware codec. Because the number of steps while adjusting the previous state to a new range can vary, it was decided that new bits of bitstream are not transferred to output bit-by-bit, but rather all at once. As a result, the maximum bit-width of bitstream output is also dependent on the bit-width of the state. In the worst-case scenario, the previous state would have to be aligned to the range $[1, 2]$. In that case, it has to be divided by 2^{m-1}, so the bit-width of the bitstream output has to be equal to m-1, where m is the bit-width of the state. Additionally, another output port is required to keep the number of valid bits on the bitstream output port. Its bit-width is equal to $\lceil log_2(state_width) \rceil$.

Although transferring all new bits of bitstream at once requires additional input and output ports, it will allow to encode one symbol per clock cycle.

C. Implementation

A Verilog description of the final solution was created. All the source code can be found in a public repository [17]. The codec was validated using a hardware description language, Verilog. Synthesis and implementation were performed for the Zybo development board that hosts the XC7Z020 chip from AMD's Zynq-7000 PSoC family.

The designed codec module has the following inputs:

- clk – codec's clock,
- reset – active high reset signal,
- start – flag indicating the encoder start working,
- freq – frequency of occurrence of one symbol,
- symbol – next symbol from sequence to encode.

It also has the following outputs:

- output ready – flag indicating the output state is valid,
- state – during the encoding bitstream bits for given input symbol, after the encoding the final value of ANS state,
- bitstream width – indicating how many bits of the bit-stream are valid. Counted from the least significant bit.

Finally, it has the following internal registers:

- M – the size of data, counted as a sum of frequencies values read from freq input,
- freqs – array with frequency values,
- C – array with cumulative frequency values,
- alphabet size – the size of the alphabet,
- ans state – state of the ANS algorithm that holds data encoded so far. Its initial value is equal to M,
- state – variable holding encoder's state number,
- index – index for loop operations.

For the purpose of verification, the working was run on the Zybo board by running a small program on Zynq's ARM processor that communicates with the ANS encoder uploaded to the FPGA part of the Zynq SoC.

For this purpose, the design was adjusted to communicate via AXI4 Stream. The AXI Stream FIFO was used as an interface between the encoder and the microprocessor. Additionally, a small program was written for the ARM Cortex-A9. The program initializes the encoder by resetting it and sends information about a string to encode, which is the size of the alphabet and frequencies of symbols, and then the string itself.

Table III shows resources used for the implementation of the validation design, where the first row is for the ANS encoder solely, and the second is for all modules that were a part of the FPGA programming configuration, including fifo and other modules needed to communicate with the microprocessor. The validation implementation was performed for an encoder with a state of 8-bit width.

D. Estimation of hardware resources

To better examine the hardware resources needed for the proposed design, synthesis was run for state bit-widths of

TABLE III
POST-IMPLEMENTATION RESOURCES OF THE PROPOSED DESIGN FOR A STATE OF 8-BIT WIDTH

Resource	Use	Percent
Slices	921	20.9%
LUTs	1,709	9.7%
LUTRAMs	48	0.8%
Flip-Flops	2,131	6.05%
BRAMs	0	0 %

8, 16, 32, and 64 bits. The first three can encode data sets of maximum size of 128 bytes, 32 768 bytes, and 2 GB, accordingly. Table IV shows how many resources were used and Table V shows the percentage of used resources of ZYNQ XC7Z010 PSoC. The numbers presented in these tables are the ones obtained after the synthesis of the codec. As it can be seen, along with increasing the width of the state and therefore the maximum size of data to encode, the number of hardware resources needed increases as well. This happens mainly because the bit-width of frequencies and cumulative frequencies stored in the tables depend on the bit-width of the data size. At the same time, those tables are the most hardware resource-consuming part of the encoder.

TABLE IV
RESOURCES USED TO IMPLEMENT THE DESIGN FOR THE DIFFERENT BIT-WIDTHS OF THE STATE ACCORDING TO THE POST-SYNTHESIS REPORT

Resource	8-bit	16-bit	32-bit	64-bit	128-bit
LUT	1,569	2,934	6,178	14,273	36,471
LUTRAM	40	80	168	336	680
Flip Flops	1,850	3,942	8,115	16,483	33,016
BRAM	0	0	0	0	0
DSP	0	0	0	0	0

TABLE V
UTILIZATION OF ZYNQ XC7Z010 RESOURCES FOR THE DIFFERENT WIDTHS OF THE STATE ACCORDING TO POST-SYNTHESIS REPORT

Resource	8-bit	16-bit	32-bit	64-bit	128-bit
LUT	8.91%	16.67%	35.10%	81.10%	207.22%
LUTRAM	0.67%	1.33%	2.80%	5.60%	11.33%
Flip Flops	5.26%	11.20%	23.05%	46.83%	93.80%
BRAM	0%	0%	0%	0%	0%
DSP	0%	0%	0%	0%	0%

E. Comparison to other implementations

Table VI contains resources needed for the implementation of encoders described in Section IV as a percentage of resources needed for the implementation of the proposed design. Additionally, what is not given in the table, the design proposed in [12] uses 50 more blocks of RAM, the design given in [14] uses 3 more blocks of RAM, the design from [15] uses 29 more blocks of RAM, and design from [15] uses two more DSP units. The number of RAM blocks is not listed in Table VI, because RAM blocks are not used by the proposed design. Also, 4-LUT usage in the other designs was compared to the 6-LUT usage in the proposed design.

TABLE VI
RESOURCES USED TO IMPLEMENT REPORTED IN LITERATURE IMPLEMENTATIONS OF ENTROPY ENCODERS AS A PERCENTAGE OF RESOURCES USED TO IMPLEMENT THE PROPOSED DESIGN

Design	Slices	LUT	LUTRAM	Flip Flops
[11]	1,504.13%	1,475.95%	-	741.01%
[12] prop	-	96.78%	16.67%	39.89%
[12] orig	-	366.65%	10.42%	21.77%
[13]	853.64%	806.38%	-	321.35%
[14]	20.85%	23.58%	-	4.93%
[15]	-	267.52%	-	205.21%

The use of resources varies between the implementations. The higher use is observed in [11], [13], and [15]. However, as mentioned in Section IV, those values include more functionalities than just compression – they include the part responsible for data pre-processing also, thus, the outcome is as expected.

On the other hand, The Huffman encoder proposed in [12] and the tANS encoder from [14] require fewer resources. The first one needs a similar number of LUTs, the second one requires much fewer resources in general.

The original Huffman encoder from [12] is positioned in the middle of the rank. It does take more than three times more LUTs than the proposed design, but at the same time, it takes much fewer flip-flops.

VI. CODEC PERFORMANCE

This section presents an important discussion of the influence of the hardware design-related assumptions that regard l and k values on compression rate and resource uses. Basically, we want to know what is the trade-off between the two. It is worth mentioning, that the provided repository [17] includes also the implementation codes for the arbitrary l and k values.

A. Compression rate

As mentioned in Section V-A, the l and k were assumed to be equal to one in the proposed design. However, this assumption can impact coding efficiency, thus, the l and k impact on the compression rate was examined. The analysis was conducted on artificially generated text data of various sizes: 128 bytes, 32,768 bytes (32 kB), and 2,147,483,648 bytes (2 GB). The data size values were chosen to match the maximum data size to encode codec state with 8-bit, 16-bit and 32-bit registers accordingly, as described in Section V-B. The probability distribution of symbols in the text was based on statistics of the *Alice in Wonderland* book.

Figures 1, 3, 5 show the number of bits per symbol (BPS) used to compress the data as a function of k for fixed values of l=1, 2, 4. Figures 2, 4, 6 show the number of bits per symbol used to compress the data as a function of k for values of l=1, 32, 64, 128. The examination was performed for 128 bytes

(Figs. 1 and 2), 32 kB (Figs. 3 and 4), and 2 GB (Figs. 5 and 6). For better comparison, some figures for greater values of l have results for l equal to one also.

Similarly, Figures 7, 9, and 11 show the number of bits per symbol used to compress the data as a function of l for k=1, 2, 4; and Figures 8, 10, and 12 for k=32, 64, 128.

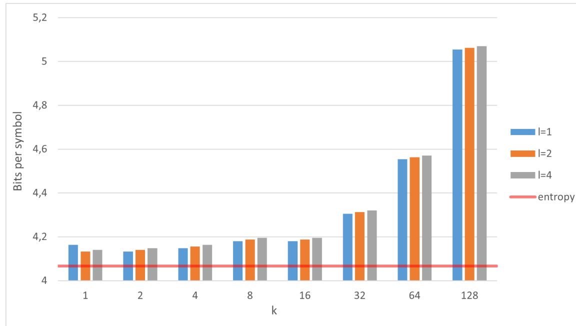

Fig. 1. BPS to encode 128 bytes for k and l={1, 2, 4}

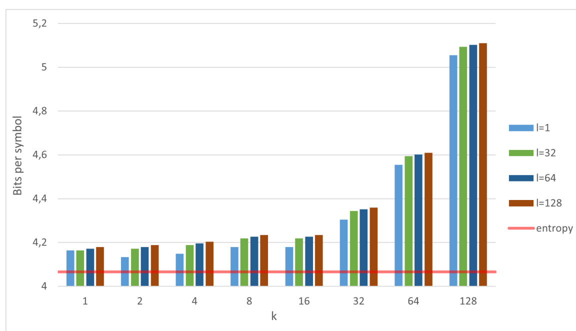

Fig. 2. BPS to encode 128 bytes for k and l={1, 32, 64, 128}

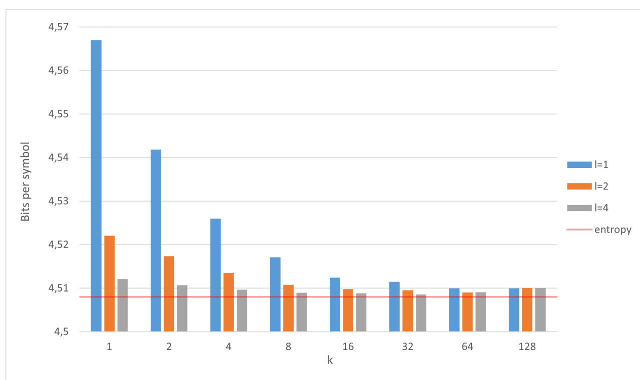

Fig. 3. BPS to encode 32 KB for k and l={1, 2, 4}

In general, one can notice, that for a small data set, here data size equal to 128 bytes, the compression ratio generally worsens with the increase of l and k values. Opposite can be noticed for a data set of size 32 kB. The increase of the parameters makes the compression ratio better. However, Figures 4 and 10 show that for greater values of parameters, the trend has its minimum for average parameters value. Lastly, for a data set of 2 GB, the relation is the opposite

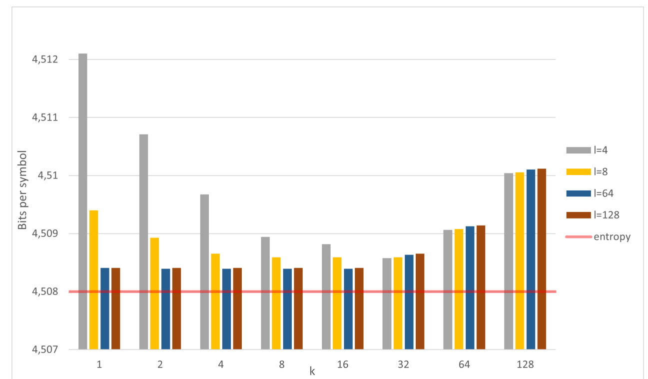

Fig. 4. BPS to encode 32 KB for k and l={4, 8, 64, 128}

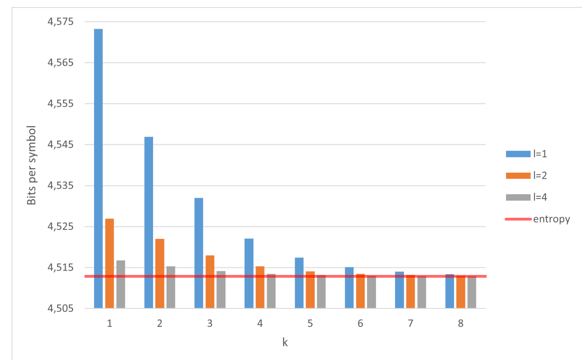

Fig. 5. BPS to encode 2 GB for k and l={1, 2, 4}

than for the 128 bytes of data set: the increase of parameter values causes a better compression ratio.

For data size equal to 128 bytes, the best compression ratio was obtained for l= to 1 and k equal to 2, or l=2 and k=1. For data size over 32 KB, l=16 and k=8 give the best ratio. For data size of 2 GB, l and k should be equal to 128 for optimal compression results. The compression ratio for these optimal parameters, calculated as a ratio of the size of compressed data to the size of uncompressed data, is equal to 51.66%, 56.43%, and 56.41% accordingly, whereas the compression ratios for l and k equal to 1 are 52.05%, 58.19%, and 57.16% for data size equal to 128 bytes, over 32 KB, and over 2 GB accordingly.

This means that one degrades the compression ratio no more than 2% if, for the hardware simplification reason, sets l and k to ones.

Based on the analysis, the bigger the data size, the greater values should be assigned to the l and k parameters. The reason for this lies in the overhead that is introduced by streaming-rANS compared to the original ANS. In streaming-rANS, the compression output consists of both the bitstream and the final state value. The coding overhead of the final state value is higher for bigger l and k values, however, the bigger l and k offer better alignment of the bitstream with entropy (bitstream bits are output more rarely). For smaller datasets, the final coding state prevails in the coding rate, and for bigger datasets the bitstream prevails. Thus, bigger datasets prefer bigger l and k, and the smaller the opposite.

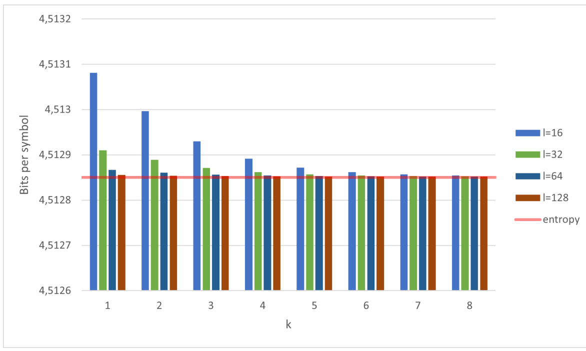

Fig. 6. BPS to encode 2 GB for k and l={16, 32, 64, 128}

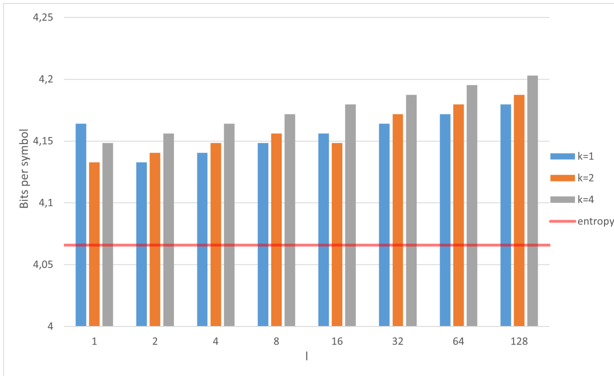

Fig. 7. BPS to encode 128 bytes for l and k={1, 2, 4}

As a result, the proposed design will offer the best compression ratio for smaller sets of data.

What is also interesting, the most of the obtained results are very close to the entropy level of the data, which is marked as a red horizontal line on the figures. Entropy is a lower bound of the number of bits needed to encode one symbol of data with lossless compression. It is calculated as

$$H(X) = -\sum_{x \in X} p(x) * log_2(p(x)), \quad (15)$$

where p(x) is the probability of occurrence of symbol x from alphabet X in data, which can be calculated as the number of symbol occurrences divided by the size of the data [18].

The difference between entropy and BPS obtained is 6.7E−2, 6.1E−4, and 8.4E−7 bits per symbol, resulting in a 8.3E−1%, 7.72E−3%, and 1E−5% difference for 128 bytes, 32 KB, and 2 GB data sizes, accordingly.

B. Resources used

In order to recognize if the proposed l=1 and k=1 values truly decrease the number of resources for codec implementation, the number of resources needed for the non-optimized encoder was examined. The main difference between this non-optimised and the one proposed in this paper is how the next state is calculated: the non-optimised design uses division and modulo operations, which are avoided in the optimized one. The results are given in Tables VII and VIII. Resources of the optimised design are given in Table III, for the same 7-bit-width data size (8-bit-width state).

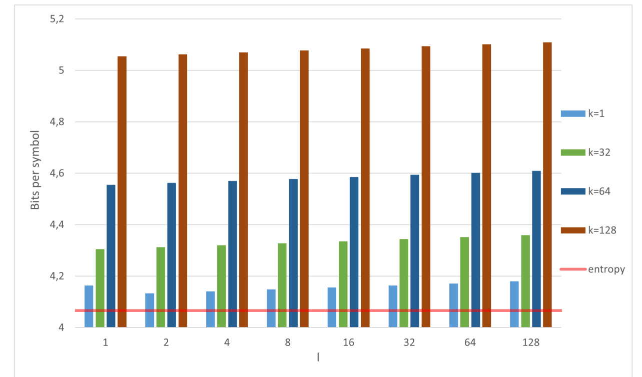

Fig. 8. BPS to encode 128 bytes for l and k={1, 32, 64, 128}

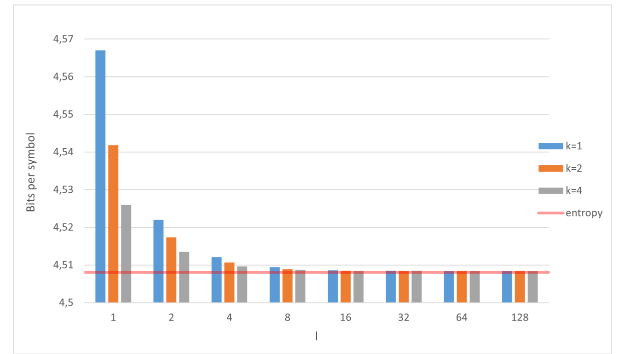

Fig. 9. BPS to encode 32 KB for l and k={1, 2, 4}

TABLE VII
USE OF LUTS BY THE NON-OPTIMISED 7-BIT-DATA-WIDTH ENCODER COMPARED TO THE OPTIMISED DESIGN FOR THE VALUES OF l AND k

l\k	1	2	4	8	16	32	64	128
1	121%	123%	114%	117%	127%	147%	192%	270%
2	123%	114%	116%	119%	128%	148%	193%	271%
4	124%	115%	117%	122%	129%	149%	195%	272%
8	118%	117%	119%	123%	132%	155%	196%	273%
16	121%	118%	120%	124%	133%	155%	197%	275%
32	131%	121%	122%	126%	135%	156%	201%	276%
64	126%	122%	123%	128%	135%	157%	200%	278%
128	136%	124%	125%	129%	136%	158%	201%	279%

TABLE VIII
USE OF FFS BY THE NON-OPTIMISED 7-BIT-DATA-WIDTH ENCODER COMPARED TO THE OPTIMISED DESIGN FOR THE VALUES OF l AND k

l\k	1	2	4	8	16	32	64	128
1	114%	114%	114%	115%	115%	117%	121%	128%
2	114%	114%	114%	115%	116%	117%	121%	128%
4	114%	114%	114%	115%	116%	117%	121%	128%
8	114%	114%	114%	115%	116%	117%	121%	128%
16	114%	114%	115%	115%	116%	118%	121%	128%
32	114%	115%	115%	115%	116%	118%	121%	128%
64	115%	115%	115%	115%	116%	118%	121%	128%
128	115%	115%	115%	115%	116%	118%	121%	128%

Additional examinations were performed for the use of DSP blocks. They were obtained for 7-bit-width, 15-bit-width and 31-bit-width data sizes. As it can be seen in Table IX, the use of the DSP blocks is not a zero, in difference to the optimised design.

For the non-optimised 15-bit-width and 31-bit-width data sizes, the usage of LUTs and flip-flops is roughly the same as for the optimized design.

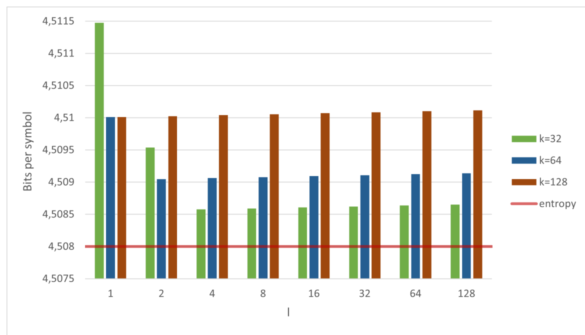

Fig. 10. BPS to encode 32 KB for various values of l and k={32, 64, 128}

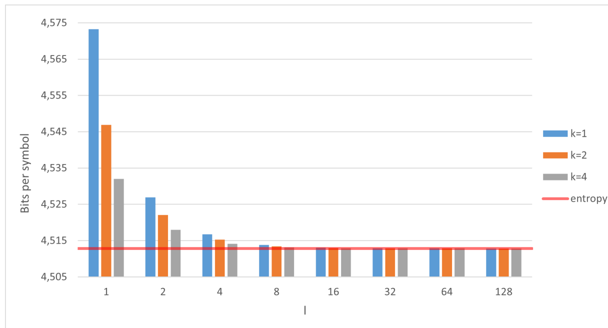

Fig. 11. BPS to encode 2 GB for l and k={1, 2, 4}

TABLE IX
POST-SYNTHESIS USE OF DSP BLOCKS FOR VALUES OF L AND K FOR NON-OPTIMISED ENCODER WITH 7-BIT-WIDTH/15-BIT-WIDTH/31-BIT-WIDTH DATA SIZE

l\k	1	2	4	8	16	32	64	128
1	0/1/3	0/2/3	2/2/4	2/2/4	2/2/5	2/3/7	4/5/11	8/8/18
2	0/2/3	2/2/3	2/2/4	2/2/4	2/2/5	2/3/7	4/5/11	8/9/18
4	2/2/3	2/2/4	2/2/4	2/2/4	2/2/5	2/3/7	4/5/11	8/9/19
8	2/2/4	2/2/4	2/2/4	2/1/5	1/2/5	2/3/7	4/5/11	8/9/19
16	2/2/4	2/2/4	2/2/4	2/1/5	1/2/5	3/3/7	4/5/11	8/9/19
32	2/2/4	2/2/4	2/2/4	2/1/5	1/2/6	3/3/7	4/5/11	8/9/19
64	2/2/4	2/2/4	2/2/4	2/2/5	2/2/6	3/3/8	5/5/11	8/9/19
128	2/2/4	2/2/4	2/1/5	2/2/5	2/2/6	3/3/8	5/5/11	8/9/19

Generally, it can be seen that the non-optimised design uses significantly more hardware resources than the proposed one. Even if l and k are ones, there are still 21% more LUTs and 14% more flip-flops needed. Additionally, the non-optimised design uses DSP units, which were not used at all in the optimized one. Changes in the number of DSP units used cause a decrease in the number of LUTs and flip-flops used. However, the number of LUTs and flip-flops utilized is not smaller than 113% of those used in the optimized design.

When comparing the use of DSPs between different data sizes, it can be seen, that with the increase of the data size, usage of DSP units increases as well. It is most probably caused by the increase of the state register size, as increasing data size causes a change of the range given in equation 7. As a result, the codec performs division and modulo operations on bigger values, which requires more resources.

VII. CONCLUSIONS AND FURTHER WORK

The proposed design of an ANS encoder meets the objectives listed in Section II. Setting k and l parameters to

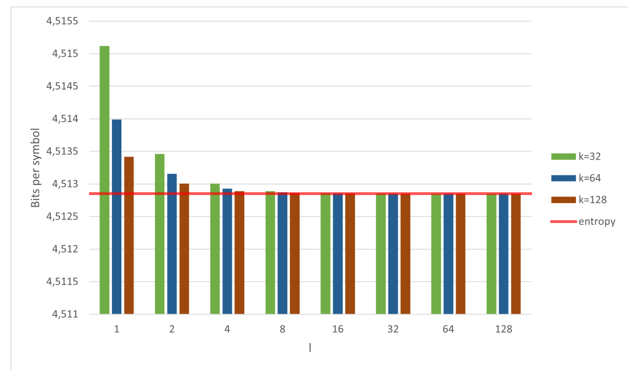

Fig. 12. BPS to encode 2 GB for l and k={32, 64, 128}

one allows a hardware designer to eliminate division and modulo operations. As a result, the proposed design requires significantly fewer hardware resources. The drop in hardware resources varies between 11.5% and 65%, depending on the k and l parameters and required data size.

A disadvantage is a drop in compression efficiency. However, this depends on the data size, as the analysis performed as part of this thesis showed. The drop happens for large data sizes, but the degradation of algorithm performance is about 1% only.

There are a few modifications of the encoder that can be implemented. Firstly, to improve it performance, the encoder should request the frequency table only when its values have to be changed, not for every new data set. As a result, the encoder would encode multiple data sets with the same frequency distribution immediately. However, this solution would be beneficial only if the chosen data sets have the same or a similar symbol probability distribution, for example, when the texts in the same language are compressed. Secondly, to reduce the number of hardware resources, it is possible to modify the design to calculate the bitstream in a loop and transfer it bit-by-bit to the output. However, with this modification, compression would take much more time, and encoding of one symbol would take more than one clock cycle, especially for greater state width.

Although this paper covers a range of analyses, some areas are still not covered.

Firstly, this paper covers the streaming-rANS encoding and does not examine the influence of l and k parameters on the decoder. Based on the fact, that the state's value is always in the range given by Equation 10, the $\lfloor \frac{X_t}{M} \rfloor$ operation in Equation 3 always results in one, and Equation 5 can be replaced by subtraction $slot = X_t - M$ in the decoder as well. Consequently, it could be examined, how much more hardware resources the traditional decoder requires than the one without the division and modulo operations, and what differences would be in performance time.

Secondly, for parameter k=1, the range given in Equation 7 can be written as $I = [F_{s_t}, 2lF_{s_t} - 1]$. This means, that the result of $\left\lfloor \frac{X_t}{F_{s_t}} \right\rfloor$ division will always be in range $[1, l-1]$. For small values of l, one can calculate the result of this division by iterative addition or subtraction. Then, the modulo operation

could be replaced by $X_t - l_{s_t} * F_{s_t}$, where l_{s_t} is the result of the division. It would be interesting to see, how it affects the number of hardware resources needed for the implementation of such an encoder, and how it affects its execution time.

Also, the work does not examine how performance time is affected by the maximum size of the data to encode, which could be an important observation. It could be helpful to decide how big could be the maximum size of the data that would still allow to have an acceptable encoding time.

Lastly, an interesting aspect to examine is how the assignment l and k values to one would affect a tabled-ANS encoder hardware implementation. The table generation time can be reduced for the tabled-ANS if the ideas presented here are incorporated. Most existing implementations of the tabled-ANS use approximations to do that and they could be compared to our solution.

REFERENCES

[1] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, September 1952.

[2] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, p. 520–540, jun 1987. [Online]. Available: doi:10.1145/214762.214771

[3] J. Duda, "Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding," 2014.

[4] ——, "List of asymmetric numeral systems implementations," URL: https://encode.su/threads/2078-List-of-Asymmetric-Numeral-Systems-implementations, accessed: [2023-08-12].

[5] "Documentation of Facebook's ZSTD," URL: https://github.com/facebook/zstd/blob/master/doc/zstd_compression_format.md#entropy-encoding, accessed: [2023-08-13].

[6] "Repository of Finite State Entropy," URL: https://github.com/Cyan4973/FiniteStateEntropy, accessed: [2023-08-13].

[7] "Specification of draco - google's 3d graphic compressor," URL: https://google.github.io/draco/spec/, accessed: [2023-08-12].

[8] J. Duda, "Asymmetric numeral systems," 2009.

[9] K. Tatwawadi, "What is asymmetric numeral systems? understanding the new entropy coder family," URL: https://kedartatwawadi.github.io/post--ANS/, accessed: [2023-06-05].

[10] P. A. Hsieh and J.-L. Wu, "A review of the asymmetric numeral system and its applications to digital images," *entropy*.

[11] M. A. S. Hernández, O. Alvarado-Nava, and F. J. Z. Martínez, "Huffman coding-based compression unit for embedded systems," in *2010 International Conference on Reconfigurable Computing and FPGAs*, 2010, pp. 238–243. [Online]. Available: doi:10.1109/ReConFig.2010.65

[12] Y. Chen, G. C. Wan, Z. W. Xia, and M. S. Tong, "A hardware design method for canonical huffman code," in *2017 Progress in Electromagnetics Research Symposium - Fall (PIERS - FALL)*, 2017, pp. 2212–2215. [Online]. Available: doi:10.1109/PIERS-FALL.2017.8293507

[13] S. Mahapatra and K. Singh, "An fpga-based implementation of multi-alphabet arithmetic coding," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 8, pp. 1678–1686, 2007. [Online]. Available: doi:10.1109/TCSI.2007.902527

[14] S. M. Najmabadi, Z. Wang, Y. Baroud, and S. Simon, "High throughput hardware architectures for asymmetric numeral systems entropy coding," in *2015 9th International Symposium on Image and Signal Processing and Analysis (ISPA)*, 2015, pp. 256–259. [Online]. Available: doi:10.1109/ISPA.2015.7306068

[15] T. Alonso, G. Sutter, and J. López de Vergara Méndez, "An fpga-based loco-ans implementation for lossless and near-lossless image compression using high-level synthesis," *Electronics*, vol. 10, p. 2934, 11 2021. [Online]. Available: doi:10.3390/electronics10232934

[16] N. Wang, C. Wang, and S.-J. Lin, "A simplified variant of tabled asymmetric numeral systems with a smaller look-up table," *Distributed and Parallel Databases*, vol. 39, pp. 711 – 732, 2020.

[17] "Repository with source code of ans encoder and python scripts," URL: https://github.com/Sharon131/masters_project, accessed: [2023-08-24].

[18] L. Kozlowski, "Shannon entropy calculator," URL: https://www.shannonentropy.netmark.pl, accessed: [2023-08-02].