AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Faculty of Computer Science, Electronics and Telecommunication

INSTITUTE OF ELECTRONICS

MASTER OF SCIENCE THESIS

The Hardware Codec for ANS Compression

Sprzętowy kodek dla kompresji ANS

Author: *Magdalena Pastuła*

Field of study: *Electronics and Telecommunications*

Supervisor: *Paweł Russek, Ph.D. D.Sc.*

Kraków, 2023

Contents

1. Introduction

Nowadays, information is one of the most, if not the most, valuable assets. Large data sets are used in modern AI systems for model training, and also in traditional data analysis to find new patterns of behaviour. Storage of this amount of data is problematic and costly, both in terms of hardware resources and computing power requirements. Moreover, another problem arises when a substantial part of this massive collection of data has to be transferred to another device. The more data is sent, the more time it takes, and more power is required to process it on the way.

A common solution to the problems of mentioned big data handling is the compression of data. Data can be stored and transferred in compressed form and decompressed while reading or upon receiving it. These scenarios create demand for high-throughput and efficient compressing methods and hardware to save as much space as possible and to make sure that sending compressed data is not slower than sending the original one.

Data analytics requires lossless compression algorithms in opposition to image compression for example. There are three major types of lossless compression algorithms: Huffman coding, arithmetic coding, and ANS (Asymmetric Numeral Systems). Huffman coding uses prefix codes, which means that every symbol has its unique coded equivalent and is not a prefix of any other coded symbol. This makes this algorithm relatively simple and fast and 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 [13].

The second compression algorithm, arithmetic coding, does not work with static symbols. Instead of assigning encoded value to each symbol, it uses two numbers to store the encoded data that represent a range. When a new symbol has to be encoded then these two numbers are changed based on the frequency of this symbol in data. Consequently, the result of encoding of the next symbol is dependent on the result of encoding of the previous one, thus arithmetic encoding cannot be executed concurrently. Moreover, arithmetic coding is relatively slow and more complicated to perform, but it has a better compression ratio than Huffman encoding [19].

The ANS compression algorithm, the youngest method presented here, can be perceived as a tradeoff of the former ones: it has a speed level and complexity comparable to Huffman

coding and a compressed data entropy level comparable to arithmetic coding [10]. Similarly to the arithmetic algorithm, ANS coding results in a big natural number, but some variants have also a bitstream as an additional encoding result.

1.1. Motivation

Currently, the ANS is quite an important and widespread algorithm. Thanks to the good complexity-to-efficiency ratio it is used in numerous applications, widely known file formats, like JPEG XL [8], or compressors implemented by easily recognizable companies, like Facebook's ZSTD [1, 8], Apple's LZFSE [2, 8] or Google's Draco compression library for 3D graphic [4, 8] for example. There are many more implementations and usages and what is worth noting, ANS still has not reached its peak in 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 small microcontroller-based embedded systems might be quite problematic. Although it is less complex than arithmetic encoding, it still requires a lot of processing and can consume most of a microcontroller's resources and time, especially when performed on relatively substantial size data sets with dynamic probability distributions of symbols. In that case, the hardware implementation of an algorithm becomes a solution. It requires the reduced hardware resources that fit the algorithm needs, and additionally, it off-loads the main processing unit from the computationally exhaustive tasks allowing it to focus on the more control-intensive ones. That leads to less task starvation, and lower overall power consumption, which makes it an economic and ecological approach.

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, as it increases the size of the encoded data. Moreover, this solution requires more processing units to be used at once.

Hardware implementation of the algorithm requires to 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.

1.2. Main goal of the thesis

The main purpose of this thesis is to propose and implement a hardware architecture for one of the most common and flexible variants of the ANS algorithm - Streaming-rANS. It is described in detail in Chapter 2.

A good hardware-oriented solution should avoid division and modulo operations that are performed as part of all ANS variants. We will propose a solution to this obstacle.

Furthermore, it should be examined what is the impact of the workaround on the compression ratio and if there are any other trade-offs of the proposed solution.

The design should be prepared in a hardware description language and implemented in FPGA technology. Resulting resource consumption should be presented, preferably with the awareness of the limits of the algorithm parameters, that can be implemented in available hardware resources.

The resulting hardware implementation should also be compared to other existing and available solutions in terms of used hardware resources.

The following are the main objectives of this thesis:

- designing hardware implementation of an ANS decoder,
- exploring how the size of ANS state affects compression ratio or its performance,
- measuring time performance of the designed encoder,
- comparison with other already existing solutions in terms of performance time.

Additionally, the following objectives should be considered as well:

- the proposed design should be able to process various data types, text, or images for example;
- the proposed design does not include run-length coding (RLC) - the part of the preprocessing that gets rid of counting symbols' occurrence in data. It should be either done by the main processing unit or some other hardware accelerator;
- bytes returned by the encoder should not contain any metadata, for example, the size of original data or frequencies of symbols in data. It should only contain encoded data.

1.3. Thesis structure

Chapter 2 describes briefly the most common variants of ANS - particularly the streaming-rANS which one was chosen for designing a hardware implementation.

Chapter 3 depicts the current state of the topic. Mainly, it presents other existing hardware implementations of ANS that could be found in the literature. For reference, it also gives the hardware implementations of two other common compression algorithms i.e. the Huffman encoding reported in literature.

Chapter 4 describes the process and development of the hardware codec design. It presents assumptions, that had to be made to design the codec, shows the resulting control state machine of the custom processor, and presents the implementation platform that was used for proof of concept validation.

Chapter 5 contains the experiment results and analysis of the impact of design constrains on compression ratio.

Lastly, Chapter 6 summarizes the thesis with a discussion of possible further research and development. Further modifications to the proposed design are also considered.

2. ANS family of algorithms

The ANS is an abbreviation for the Asymmetric Numeral Systems. It is an algorithm of lossless compression and it was first published by Jarosław Duda in 2009. There are many variants of this algorithm, but the idea behind them all stays the same. We create a finite-state machine whose next state depends on the previous state and a current symbol to encode. The final state of this FSM, after encoding all symbols, represents the encoded data [9].

In order to describe the most common variants of ANS, a few definitions should be presented first:

- symbol - the smallest part of data to encode. For example, if the data to encode is in the form of text, a symbol would be a representation of one letter. It will be marked as s_t in equations;

- alphabet - a set of symbols that can occur in data to encode. For example, if the data to encode is in the form of ASCII text, then its alphabet would be the whole ASCII table or its subset;

- frequency of a symbol - number of occurrences of a symbol in data to encode. It will be marked as F_{s_t} in equations;

- cumulative frequency of a symbol - a sum of frequencies of all symbols that are, in the alphabet, before the symbol for which it is calculated. It is formally described in Equation 2.1 [17]. It will be marked as C_{s_t} in equations;

$$C_{s_i} = \sum_{j=1}^{i-1} F_{s_j} \tag{2.1}$$

- size of data to encode - number of symbols in data to encode. It can be calculated as a sum of all the frequencies. It will be represented by M in equations;

- state - an integer number representing data encoded so far. After encoding of all symbols is performed, it is a result of compression;

– bitstream - an additional part of the compression result in the case of streaming-rANS and tabled-ANS variants.

Basically, the execution of every ANS variant can be divided into two phases: the initial phase when data to encode is scanned to find frequencies and cumulative frequencies of each symbol in data, and the encoding phase when consecutive symbols in data are being encoded.

The most popular variants of ANS are range-ANS, streaming-rANS, and tabled-ANS and are described in sections below.

2.1. Range-ANS

Range ANS, also called rANS, is a variant that encodes consecutive symbols in ranges, similar to arithmetic encoding [17]. When encoding a symbol, the next state is calculated with the formula

$$X_t = \left\lfloor \frac{X_{t-1}}{F_{s_t}} \right\rfloor * M + C_{s_t} + mod(X_{t-1}, F_{s_t}), \tag{2.2}$$

where operator mod stands for the modulo operation.

While decoding range ANS data, the previous symbol and the previous state can be calculated by the formulas:

$$slot = mod(X_t, M), \tag{2.3}$$

$$s_t = C_{INV}(slot), \tag{2.4}$$

$$X_{t-1} = \left\lfloor \frac{X_t}{M} \right\rfloor * F_{s_t} + slot - C_{s_t}, \tag{2.5}$$

$$C_{INV}(y) = a_i, \text{ if } C_{a_i} < y < C_{a_{i+1}}, \tag{2.6}$$

where C_{INV} stands for inverse cumulative frequency.

As one can see, every compression step of this algorithm increases the bit size needed to store the state's value. Moreover, the state's value increases approximately exponentially with the size of the data to encode, as in every step it is multiplied by a factor proportional to M. This is a major disadvantage of this version of ANS, as encoding larger data sets is problematic due to the required state register big size.

2.2. Streaming-rANS

Streaming rANS [17] is a slight modification of range-ANS. While the encoding step uses the same formula, the only difference is that before encoding the previous state is adjusted to be in an appropriate range. This range is dependent on the frequency of the symbol to encode and can be described as

$$I_{s_t} = [lF_{s_t}, 2^k lF_{s_t} - 1]. \tag{2.7}$$

This adjustment causes the state to always be in the range

$$I = [lM, 2^k lM - 1], \tag{2.8}$$

where l and k could be any integer number.

The previous state is adjusted to range given by Equation 2.7 by dividing it by 2^k, where k is any integer number. The remainders of this integer division make up the bitstream, which is a part of the result of encoding.

The value of the k parameter is equal to the number of bits of the bitstream chunk that the encoder produces or a decoder can receive at once. Usually, it is equal to 8, 16, 32, or 64, because it simplifies the process of sending the encoded data.

Algorithm 1 and 2 show the pseudocode of streaming-rANS encoding and decoding step, accordingly. Operations performed after while loops are the ones described in equations 2.2, 2.3, 2.4 and 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 $X_t \leq l * M$ **do**
  $X_t \leftarrow (X_t << k) + mod(bitstream, 2^k)$
  $bitstream \leftarrow bitstream >> k$
end while
$slot \leftarrow mod(X_t, M)$
$s_t \leftarrow C_{INV}(slot)$
$X_{t-1} \leftarrow (X_t//M) * F_{s_t} + slot - C_{s_t}$

2.3. Tabled-ANS

Because of the adjustment of the previous state to the appropriate range in the streaming-rANS, the result of encoding will always be the same for the given symbol and previous state, therefore one can calculate it in advance and put it in a table. This is the idea that lies behind

tabled ANS. This means that the initial phase of this algorithm contains one more step: calculating the next state for every value of the previous state and symbol to encode and save it in the table. Consequently, tabled ANS needs a lot of preprocessing, as every case needs to be considered, thus it is mainly used when the distribution of symbols' occurrence in data is known beforehand and does not change much for another data set or when the distribution is assumed in advance. Moreover, to store such a table, it is required to have available memory of size 2 to 16 times the size of the alphabet, when the latter is equal to 256 [10].

2.4. Variant chosen for hardware implementation

Streaming-rANS was chosen as the variant for custom hardware processor design in this work. The reason is that some hardware implementations of tabled-ANS are already reported in the literature, as described in section 3.2, but no implementations could be found for streaming-rANS. Moreover, a result of 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 [12]. This feature may play a significant role in embedded systems, for which this hardware implementation is designed.

3. Existing entropy compression hardware

3.1. Entropy encoders

There are numerous hardware implementations of the Huffman and arithmetic coding. For example, hardware implementation of 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 often it only requires adding one or adding one and shifting. This operation is performed from the symbol of the smallest width and does not require building the Huffman tree. Although there is a large number of resources needed to implement this encoder, it includes the part responsible for counting symbols' frequency in data and the part responsible for generating codes. What is also worth noting is that each field of the array containing frequencies has a 32-bit width and the size of the alphabet is assumed to be equal to 256.

Another example of a hardware implementation of an entropy encoder is the Huffman encoder in [7], based on Canonical Huffman Coding. This design builds the static Huffman tree and reads the length of each encoding. Then one of the longest encodings is 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, however, the authors did not specify for what size of the alphabet the design was implemented. The article also contains a list of resources needed for the implementation of the original Huffman encoder.

Design	Slices	4-LUT	LUT	LUTRAM	Flip Flops	BRAM	Bonded IOB	BUFG
[11]	13,853	25,224	-	-	15,791	-	123	-
[7] proposed	-	-	1,654	8	850	50	10	1
[7] original	-	-	6,266	5	464	0	10	1
[15]	7,862	13,781	-	-	6,848	-	37	-

Table 3.1: Resources used to implement found in literature entropy encoders

One more example of a hardware implementation of entropy coding is described in [15]. It consists of two parts: a modeller, a part responsible for estimating symbols' probabilities, and a coder, a part responsible for coding the symbols. The design proposed in this article has a modeller that stores symbols' frequencies as leaf nodes in a binary tree with cumulated frequencies stored in intermediate nodes. When one of the frequencies changes, a specific procedure is performed to update the binary tree. When it comes to the coding part, it is composed of binary encoders that take intermediate node and interval as arguments. The design was implemented for the alphabet size equal to 256.

Table 3.1 contains hardware resources needed for the implementation of the Huffman codecs reported in [7, 11]. Parameters that are not specified in the cited publication are marked as '-' in the table.

3.2. ANS encoders

No streaming-rANS or rANS hardware implementation could be found in literature. Moreover, only three implementations of tabled-ANS was found.

The first found implementation is described in [16]. 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, there are two encoding tables: one with the width of the bitstream and the second with the next state's value, both indexed by an address consisting of the current state's value and symbol to encode. Although this solution requires more memory, it can run on a much higher frequency.

The second implementation that was found is the one described in [6]. However, This thesis does not describe an implementation of the tabled-ANS only. It is an image compressor of the LOCO-ANS algorithm, meaning that the ANS encoding is only a part of this solution, next to the pixel decorrelator and quantizer.

The third design of the tabled-ANS encoder is the one described in [18]. This solution divides the encoding table into two parts: one with new states of equal width and the second also with new states of equal width, but greater by one than the first one. To be able to divide 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 replacing only part of the bits of the state, not all. However, this design does not have a specified number of hardware resources needed for its implementation in the article. It only specifies a throughput for four input files [18].

Table 3.2 shows resources used for the implementation of these encoders, where resources listed for [16] are for an alphabet of size equal to 256 and resources listed for [6] are for the implementation of lossless compression with a state of 7-bit width.

Design	Slices	4-LUT	LUT	Flip Flops	BRAM	Bonded IOB	DSP
[16]	192	403	-	105	3	-	-
[6]	-	-	4,572	4,373	29	-	2

Table 3.2: Resources used to implement found in literature ANS encoders

Compared to resources needed for other entropy encoders found in the literature it can be seen, that ANS encoders generally need much fewer resources. Even the design proposed in [6], where the encoder is only a part of the solution, needs fewer resources than the original Huffman encoder listed in [7]. However, there are many more hardware implementations of the Huffman coding, meaning that there might be versions that require even fewer resources.

4. Custom ANS coder development

This chapter describes the process of designing the hardware implementation of ANS encoder. Assumptions about the input data and how operations problematic in hardware implementations were dealt with are also discussed here. A description of the final solution is given. All the source code can be found in a public repository [3].

4.1. Input data-related assumptions

When analyzing the streaming-rANS algorithm, it can be noticed, that the algorithm requires the following parameters to perform a single coding step:

- symbol code,
- previous state's value,
- frequencies of all symbols from the alphabet,
- cumulative frequencies of all symbols from the alphabet,
- size of data to encode - M.

Whereas the final output of the algorithm is the final state and the bitstream. While the size of data and cumulative frequency can be calculated based on the frequencies, frequencies themselves have to be known before encoding can start. For this reason, the working of the designed hardware codec was divided into two phases: the initial phase, when the number of frequencies and frequencies themselves are transferred to the codec, and the encoding phase when the codec reads consecutive symbols of data and encodes them.

In order to design a hardware implementation of the streaming-rANS encoder, the number of register bits necessary to represent inputs, outputs, and internal variables had to be determined.

As an initial constraint, it was assumed that the alphabet's size would be not greater than 256 symbols as that allows encoding of text written in ASCII code and images. Additionally, it is possible to encode any data format this way, as long as it is read byte by byte. This assumption propagates to the width of the symbol, which should be 8-bit.

Another assumption that had to be made was the maximum size of data to encode. According to Equation 4.2, we can conclude that the maximum value of the state is equal to $2*M_{max}-1$, where M_{max} is the maximum data size. However, we want 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 this reason, the bit-width of the data size will be equal to m-1, where m is the bit-width of the state. This leads to the maximum size of the data being equal to the half of the maximum state's value. This assumption propagates to the frequency and cumulative frequency bit-width, as in the borderline case they would be equal to the size of the data.

When it comes to the state's width, it was decided that it would be parameterized, so that it would have a fixed value after implementation, no matter the data size.

Another assumption had to be made about the bitstream output and how it will be 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 will not be transferred to output bit by bit, but rather all at once. As a result, the bitstream's maximum bit-width in one step 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 would be divided by 2^{m-1}, so the bit-width of the bitstream in that step would be equal to m-1, where m is the bit-width of the state. Additionally, there will be required another output variable to know how many bits from the bitstream are valid. Its width should be 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.

Besides the aforementioned, there are also parameters l and k described in Section 2.2, which values had to be predetermined. The selection of their values is described in detail in Section 4.2.

4.2. Avoiding division and modulo operations

As mentioned in Section 1.2 the design encoder should avoid division and modulo operations that are performed in every encoding step of the Streaming-rANS.

If we look closely at the last formula, we can see that for l and k equal to one the Equations 2.7 and 2.8 simplify to the following:

$$I_{s_t} = [F_{s_t}, 2F_{s_t} - 1], \tag{4.1}$$

$$I = [M, 2M - 1]. \tag{4.2}$$

That means that Equation 2.2 can also be simplified to

$$X_t = M + C_{s_t} + X_{t-1} - F_{s_t}. \tag{4.3}$$

The reason is that $\left\lfloor \frac{X_t}{F_{s_t}} \right\rfloor$ will always be equal to one, because the previous state X_t is always aligned to range 4.1, before encoding the next symbol. Consequently, $mod(X_t, F_{s_t})$ can be calculated as $X_t - F_{s_t}$, as modulo operation can be defined as $mod(X_t, F_{s_t}) = X_t - \left\lfloor \frac{X_t}{F_{s_t}} \right\rfloor * F_{s_t}$.

For example, if the previous state X_t is equal to 12 and the frequency of the next symbol to encode is equal to 5, then the previous state will be aligned to range $I_{s_t} = [5, 9]$, by dividing it by 2, which will give 6. As a result, the $\left\lfloor \frac{X_t}{F_{s_t}} \right\rfloor$ will be equal to $\left\lfloor \frac{6}{5} \right\rfloor = 1$ and $mod(X_t, F_{s_t})$ will be equal to $X_t - F_{s_t} = 6 - 5 = 1$.

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

$$slot = X_t - M, \tag{4.4}$$

$$s_t = C_{INV}(slot), \tag{4.5}$$

$$X_{t-1} = F_{s_t} + slot - C_{s_t}. \tag{4.6}$$

Because the previous state is always aligned to the range given by Equation 4.2 before performing Equations 2.3, 2.4, and 2.5.

This observation was used when creating hardware codec meaning k and l parameters' values are fixed to one and that the next state is calculated based on Equation 4.3, and not Equation 2.2. This way the problematic division and modulo operations are replaced by one subtraction operation with no loss in compression efficiency compared to encoding with those operations and for k and l parameters' values equal to 1.

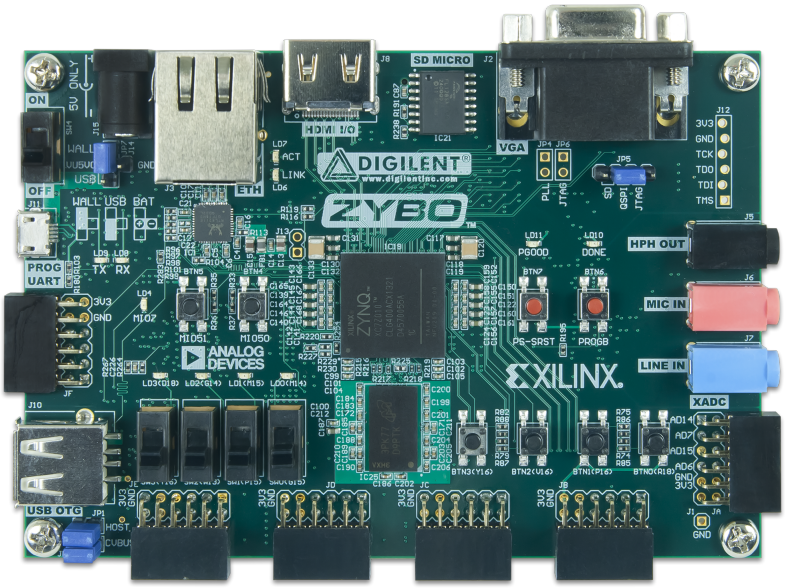

Figure 4.1: Photo of Zybo Development Board

4.3. Prototyping board

Codec was designed using a hardware description language, Verilog. Synthesis and implementation were performed for Zybo - Zynq-7000 ARM/FPGA SoC evaluation board. It was chosen because it is similar to ZedBoard Zynq-7000 Development Board which the author was already familiar with and because it was available to borrow from the thesis supervisor.

Figure 4.1 shows a photo of the Zybo Development Board.

4.4. Description of created codec

Figure 4.2 shows the state machine of the created encoder, where *freq_num* is a number of frequencies equal to the size of an alphabet and *i* variable is an iterator.

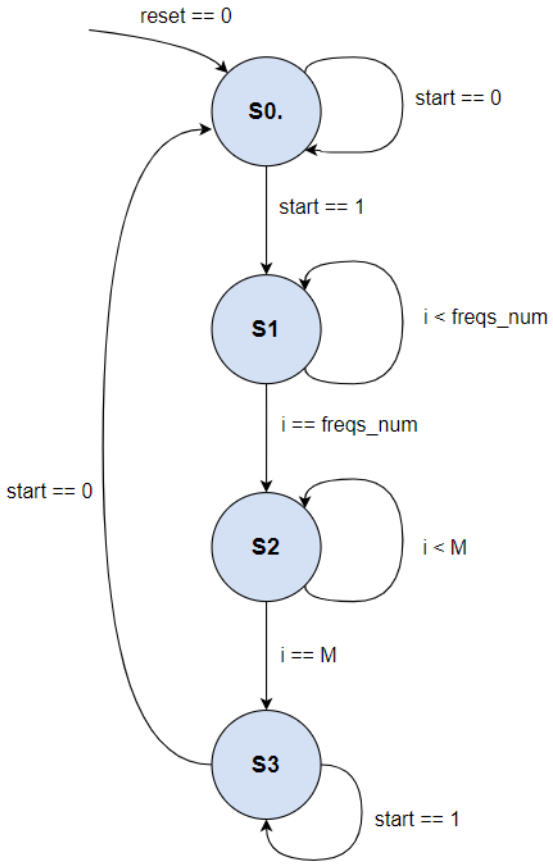

Figure 4.2: State machine graph of designed codec

The state machine of the created codec has the following states:

– S0 - initial state. In that state codec is waiting for the start signal. When it is high, a number of frequencies is read from freq input and the state is changed to S1,

– S1 - preparing state. In that state, codec reads one frequency of a symbol per clock cycle and calculates the values of M and cumulative frequencies. It changes to state S2 after number of clock cycles equal to the number of frequencies read when transitioning to S1,

– S2 - encoding state. In this state encoding is performed by reading consecutive sequence symbols every clock cycle. Also, bitstream and its width are calculated and written to output registers. It changes to S3 state after M clock cycles,

– S3 - ending state. In this state codec waits for the start signal to be equal to zero to change to state S0. Upon this transition, internal and output registers are reset to zero.

Additionally, the designed codec module has the following inputs:

– clk - clock,

– reset - reset. Active for high-level,

– start - input indicating if the encoder should start working,

– freq - frequency of occurrence of one symbol,

– symbol - next symbol from sequence to encode.

It also has the following outputs:

– output ready - output indicating if 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 - variable indicating how many bits of the bitstream are valid. Counted from the least significant bit.

Finally, it has the following internal variables:

– M - the size of data, counted as a sum of frequencies values read from freq input,

– freqs - array with frequencies values,

– C - array with cumulative frequencies 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.

In order to examine hardware resources needed for the proposed design, synthesis was run for a state's width equal to 8, 16, 32, 64, and 128-bit. The first three can encode data sets of maximum size equal to 128 bytes, 32 768 bytes, and over 2 GB, accordingly. Table 4.1 shows how many resources were used and Table 4.2 shows the percentage of used resources of ZYNQ XC7Z010 PSoC, both for the aforementioned state's widths. 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 width of frequencies and cumulative frequencies stored in tables depends on the size of the data width, which depends on the state's width, as described in 4.1. At the same time, those tables are the most hardware resource-consuming part of the encoder.

Resource	8-bit state	16-bit state	32-bit state	64-bit state	128-bit state
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
Bonded IOB	31	47	80	145	274
BUFG	1	1	1	1	1
DSP	0	0	0	0	0

Table 4.1: Resources used to implement the design for the different widths of the state according to the post-synthesis report

4.5. Implementation of the design

The proposed design's working was checked on the Zybo board by running a small program on Zynq's ARM processor that communicates with an instance of the ANS encoder uploaded to the FPGA part (PL) of the Zynq SoC.

For this purpose, the design was adjusted to communicate via AXI4 Stream and AXI Stream FIFO was used as an intermediary between the encoder and the microcontroller. Additionally, the implementation was slightly modified: the start signal was changed to clock enable signal and the output_valid output variable was changed to be set when the first bitstream is transferred to the output, not after encoding all symbols.

Resource	8-bit state	16-bit state	32-bit state	64-bit state	128-bit state
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%
Bonded IOB	31%	47%	80%	145%	274%
BUFG	3.13%	3.13%	3.13%	3.13%	3.13%
DSP	0%	0%	0%	0%	0%

Table 4.2: Utilization of ZYNQ XC7Z010 resources for the different widths of the state according to post-synthesis report

Additionally, a small program was written for the microcontroller. 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. The string is fixed in the source code and it is "aabcddaacbab", where ’a’ is represented by 0, ’b’ by 1, ’c’ by 2, and ’d’ by a number 3.

Result	Slices	LUT	LUTRAM	Flip Flops	BRAM	BUFG
ANS encoder	921	1,709	48	2,131	0	0
All	1,331	2,708	169	3,191	2	1
ANS encoder	20.93%	9.71%	0.8%	6.05%	0%	0%
All	30.25%	15.39%	2.82%	9.07%	3.33%	3.13%

Table 4.3: Post-implementation resources of the proposed design for a state of 8-bit width

Table 4.3 shows resources used for the implementation of the design, where the first row is for the ANS encoder solely, and the second is for all that was a part of the bitstream, including fifo and other modules needed to communicate with the microprocessor. The last two rows present the same data but as a percentage of resources available on the Zynq board. The implementation was only performed for an encoder with a state of 8-bit width.

Figure 4.3 shows the output received on a COM port after running the program.

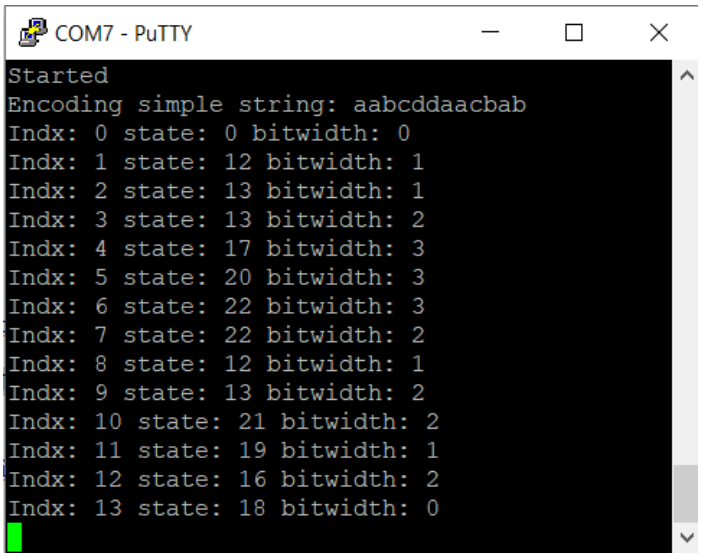

Figure 4.3: Screen of COM output from the board after running the program

4.6. Comparison to other implementations

Table 4.4 contains resources needed for implementation of encoders described in Chapter 3 as a percentage of resources needed for implementation of the proposed design only (ANS encoder) presented in Section 4.5. Moreover, the design proposed in [7] uses 50 more blocks of RAM and 1 BUFG more, the original design from [7] uses 1 BUFG more, the design given in [16] uses 3 more blocks of RAM, the design from [6] uses 29 more block of RAM, and design from [6] uses 2 more DSP units. Those values are not listed in the table, because its value was equal to zero for the proposed design. Moreover, 4-LUT usage in designs was compared to the LUT usage in the proposed design.

Design	Slices	LUT	LUTRAM	Flip Flops	Bonded IOB
[16]	20.85%	23.58%	-	4.93%	-
[7] prop	-	96.78%	16.67%	39.89%	32.26%
[6]	-	267.52%	-	205.21%	-
[7] orig	-	366.65%	10.42%	21.77%	32.26%
[15]	853.64%	806.38%	-	321.35%	119.35%
[11]	1,504.13%	1,475.95%	-	741.01%	396.77%

Table 4.4: Resources used to implement reported in literature implementations of entropy encoders as a percentage of resources used to implement the proposed design

It can be seen, that for some implementations usage of resources is lower, and for some is greater. The latter can be observed for [11], [15], and [6]. However, as mentioned in Chapter

3, those values include more than just the coding step. For example, they include also the part responsible for preprocessing the data, thus the outcome is as expected.

On the other hand, encoders that require fewer resources exist. For example, the proposed Huffman encoder from [7] and the tANS encoder from [16]. While the first one has a similar number of LUTs needed, the second one requires much fewer resources in general.

In the middle of the rank, there is an original Huffman encoder from [7]. It does take more than three times more LUTs than the proposed design, but at the same time, it takes much fewer flip-flops and input-output ports.

5. Discussion

This chapter presents a discussion of the influence of the l and k parameters on compression rate and resource uses.

5.1. Impact of the l and k parameters on the compression rate

As mentioned in Section 4.2, the l and k parameters that manipulate the position and width of ranges (equations 2.7 and 2.8) were assumed to be equal to 1 in the proposed design. However, this assumption can have an impact on compressing efficiency, thus, an analysis of the l and k parameters impact on the compression rate was conducted to examine this issue. Analysis was conducted on a text of a specified size generated to a file with a Python script. The distribution of character occurrence in the text was calculated based on *Alice in Wonderland*. Analysis was performed with the Python script created for this purpose as well.

Figures 5.1, 5.5, 5.9 show the number of bits per symbol used to compress the data as a function of k for fixed small values of l. Figures 5.2, 5.6, 5.10 show the number of bits per symbol used to compress the data as a function of k for fixed bigger values of l. All for data size equal to 128 bytes, 32,768 bytes and 2,147,483,648 bytes (over 2GB), accordingly. Those 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 4.4.

Similarly, Figures 5.3, 5.7, and 5.11 show the number of bits per symbol used to compress the data as a function of l for fixed small values of k. Figures 5.4, 5.8, 5.12 for fixed bigger values of k. Additionally, some figures for greater values of l or k have results for l or k equal to one, for better comparison.

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 parameters' values. Different behaviour can be noticed for a data set of medium size, here the size equal to 32,768 bytes (over 32 KB). The increase of the parameters' values makes the compression ratio better, but not in the entire range. Figures 5.6 and 5.8 show that for greater values of parameters, the trend is reversed and

the increase in values worsens the compressions ratio. Lastly, for a data set of a large size, here the data size over 2 GB, the relation is the opposite than for the small data set: the increase of parameter's values causes a better compression ratio.

Moreover, the best compression ratio was obtained for l equal to 1 and k equal to 2 or l equal to 2 and k equal to 1 in case of data size equal to 128 bytes, for l equal to 16 and k equal to 8 in case of data size over 32 KB, and for l and k equal to 128 in case of data size over 2 GB. The compression ratio for these parameters' values, 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%, whereas the compression ratio for l and k equal to 1 is equal to 52.05%, 58.19% and 57.16% for data size equal to 128 bytes, over 32 KB, and over 2 GB accordingly.

Overall, based on the analysis, the bigger a data set, the greater values should be assigned to the l and k parameters. As a result, the proposed design will offer the best compression ratio for smaller sets of data.

What is also interesting is that most of the obtained results are close to the entropy level of the data, which is marked as a red horizontal line on the figures. Entropy is described as the level of information in data, but, at the same time, is a lower bound of the number of bits needed to encode one symbol of data with lossless compression. Formula 5.1 for an alphabet X can be used to calculate it, where p(x) is the probability of occurrence of symbol x in data, which can be calculated as the number of occurrences divided by the size of the data [14].

$$H(X) = -\sum_{x \in X} p(x) * log_2(p(x)) \tag{5.1}$$

The difference between entropy and the minimum number of bits per symbol obtained in the analysis is equal to $6.7\mathrm{E}{-2}$, $6.1\mathrm{E}{-4}$, and $8.4\mathrm{E}{-7}$ bits per symbol resulting in a difference of $8.3\mathrm{E}{-1}$, $7.72\mathrm{E}{-3}$, and $1\mathrm{E}{-5}$ percentage points in compression ratio forsize equal to 128 bytes, over 32 KB and over 2 GB, accordingly.

Analysis was not performed for even larger data sets because of the long computing time.

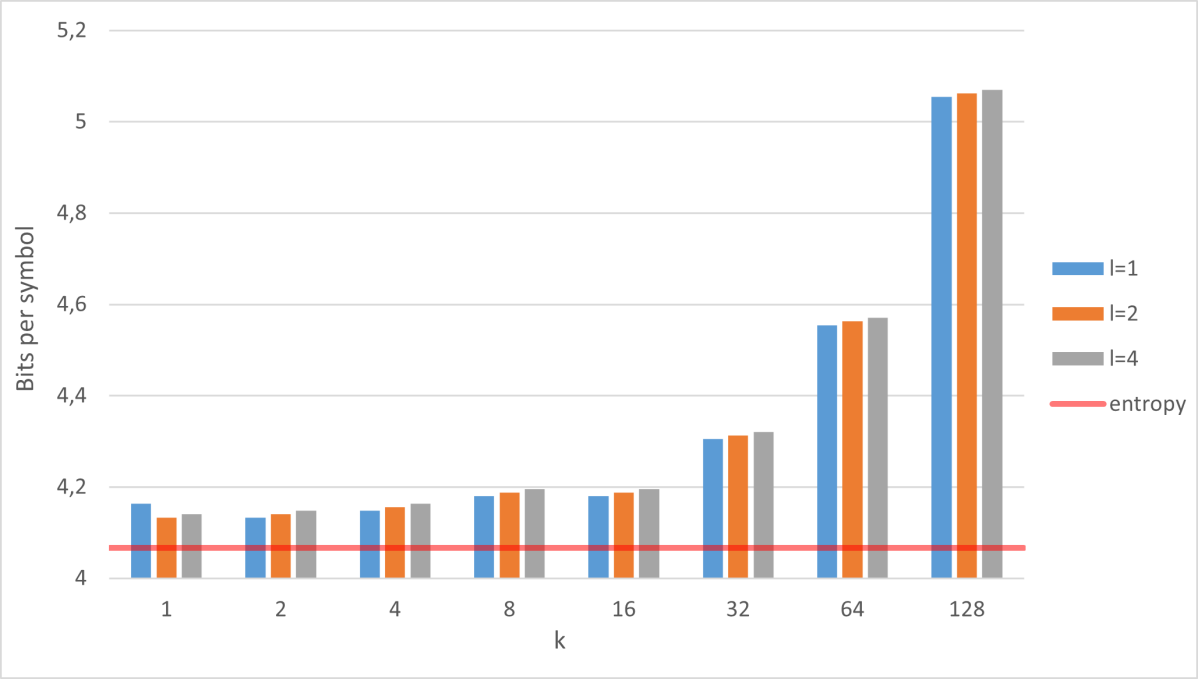

Figure 5.1: Bits per symbol used to encode 128 bytes of data for various values of k and l={1, 2, 4}

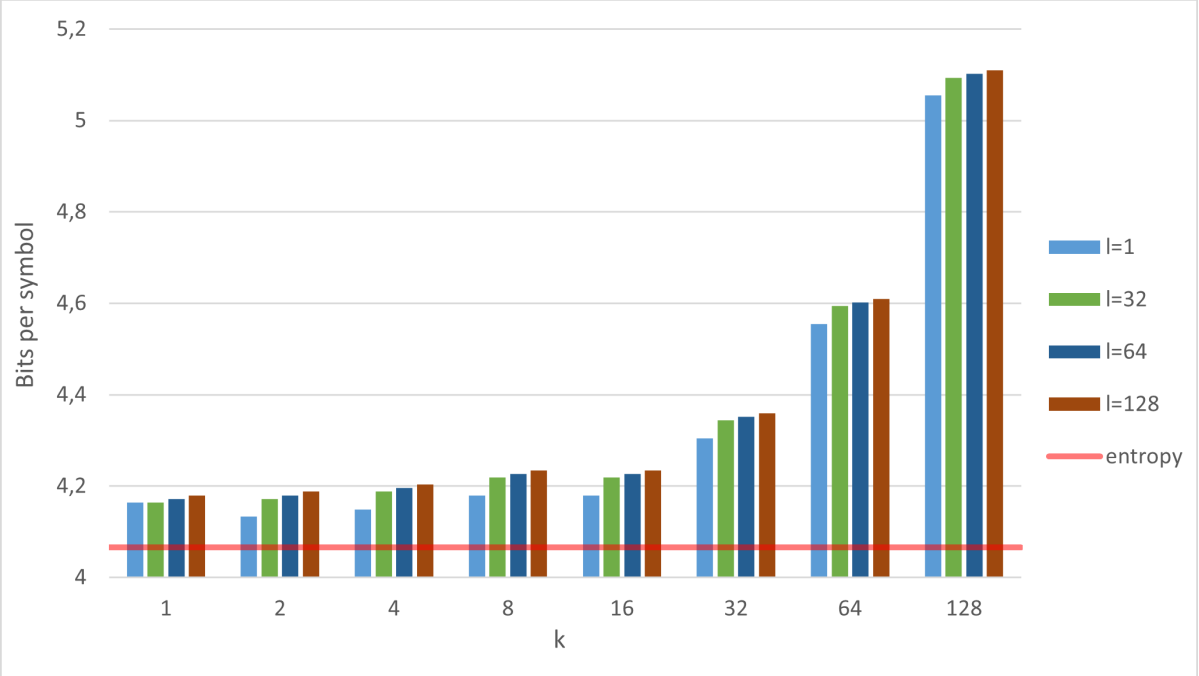

Figure 5.2: Bits per symbol used to encode 128 bytes of data for various values of k and l={1, 32, 64, 128}

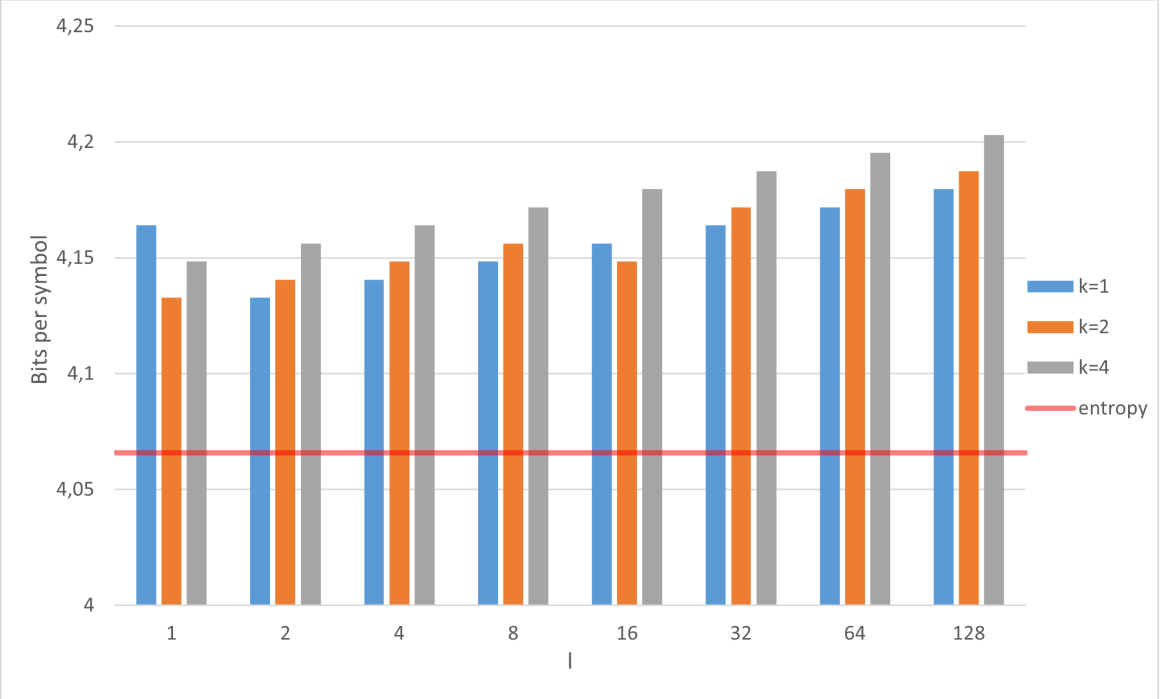

Figure 5.3: Bits per symbol used to encode 128 bytes of data for various values of l and k={1, 2, 4}

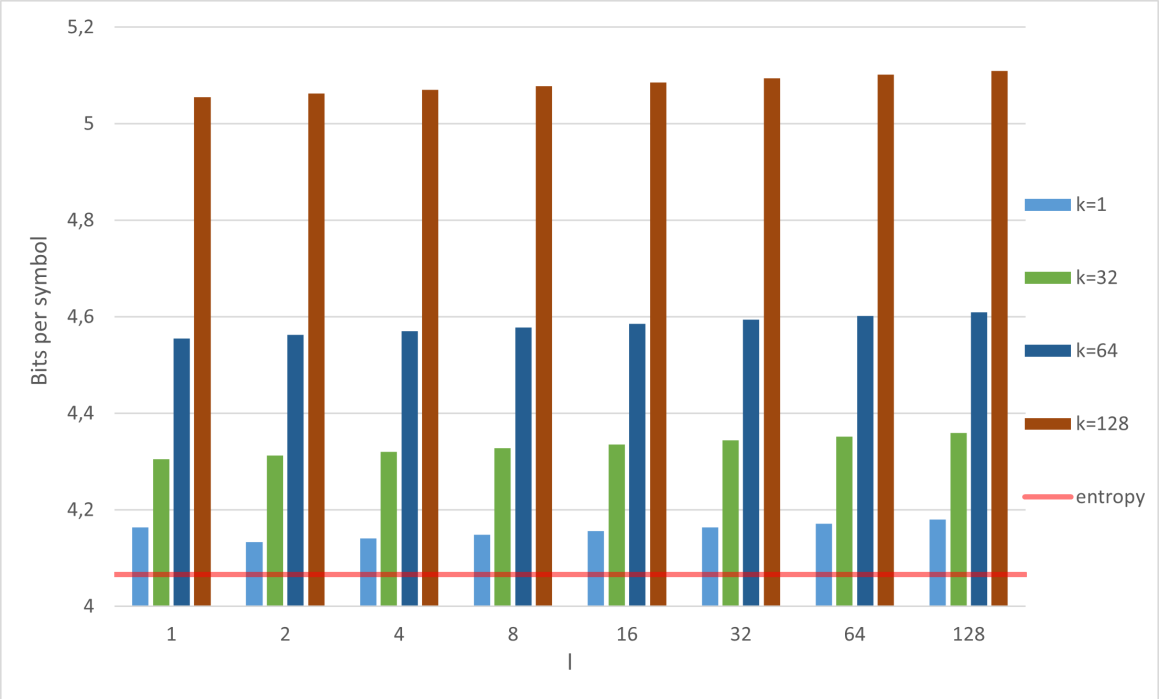

Figure 5.4: Bits per symbol used to encode 128 bytes of data for various values of l and k={1, 32, 64, 128}

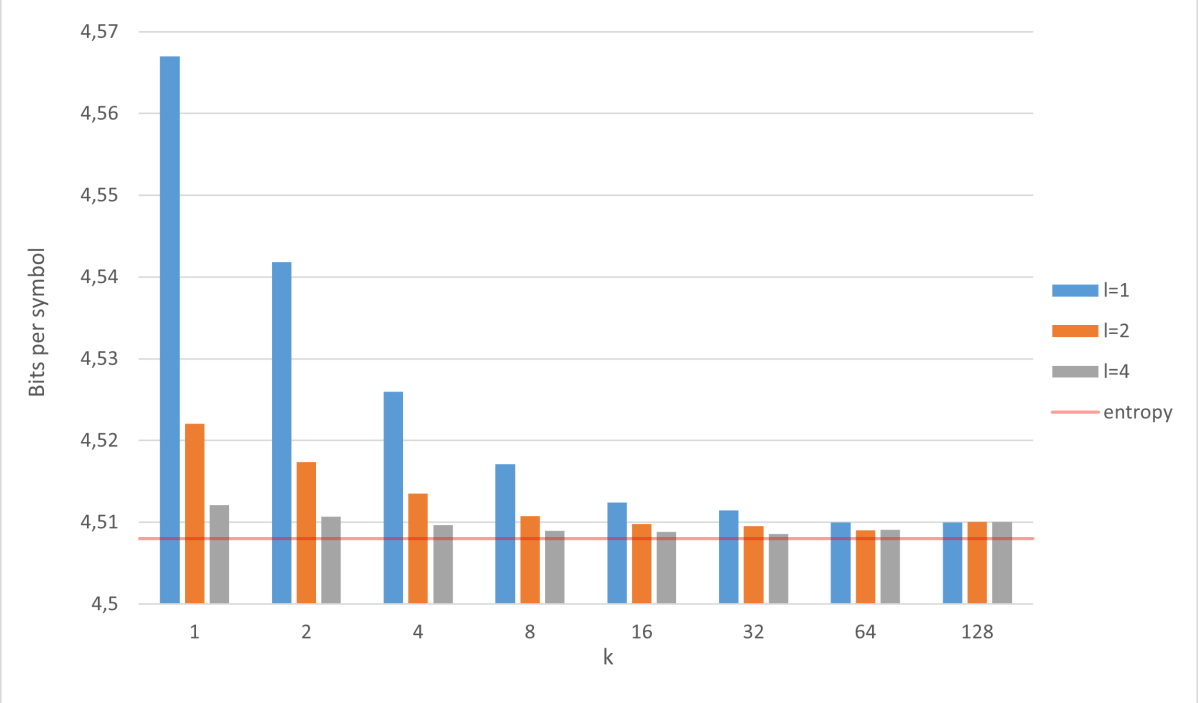

Figure 5.5: Bits per symbol used to encode over 32 KB of data for various values of k and l={1, 2, 4}

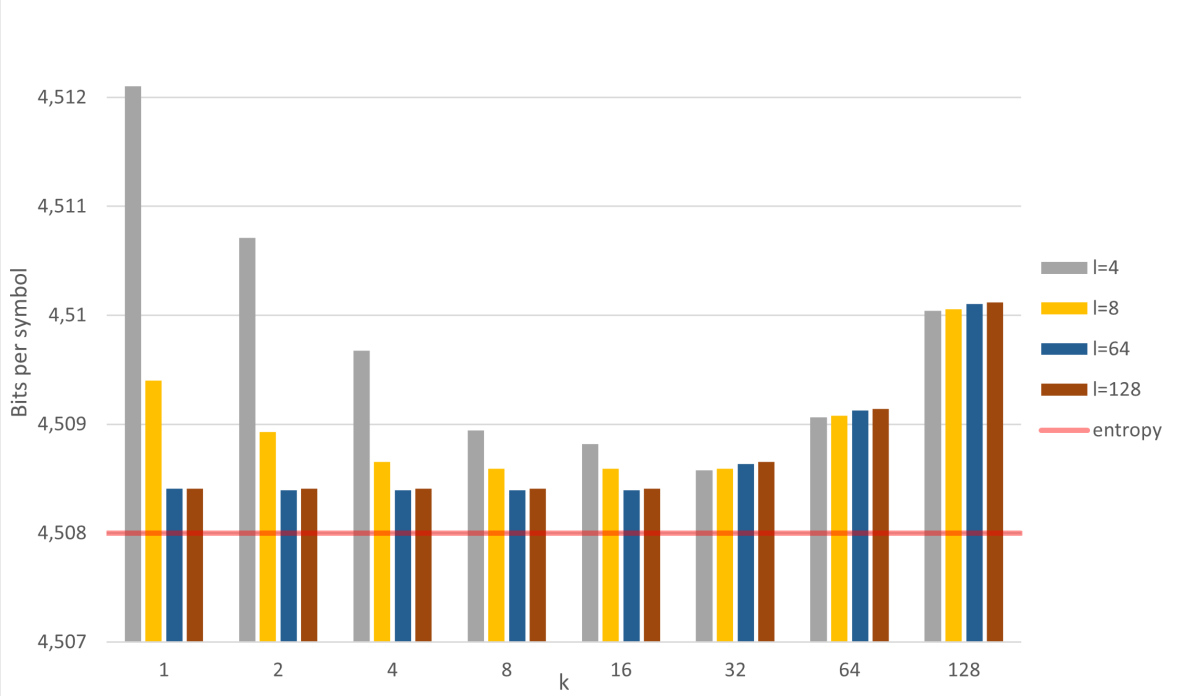

Figure 5.6: Bits per symbol used to encode over 32 KB of data for various values of k and l={4, 8, 64, 128}

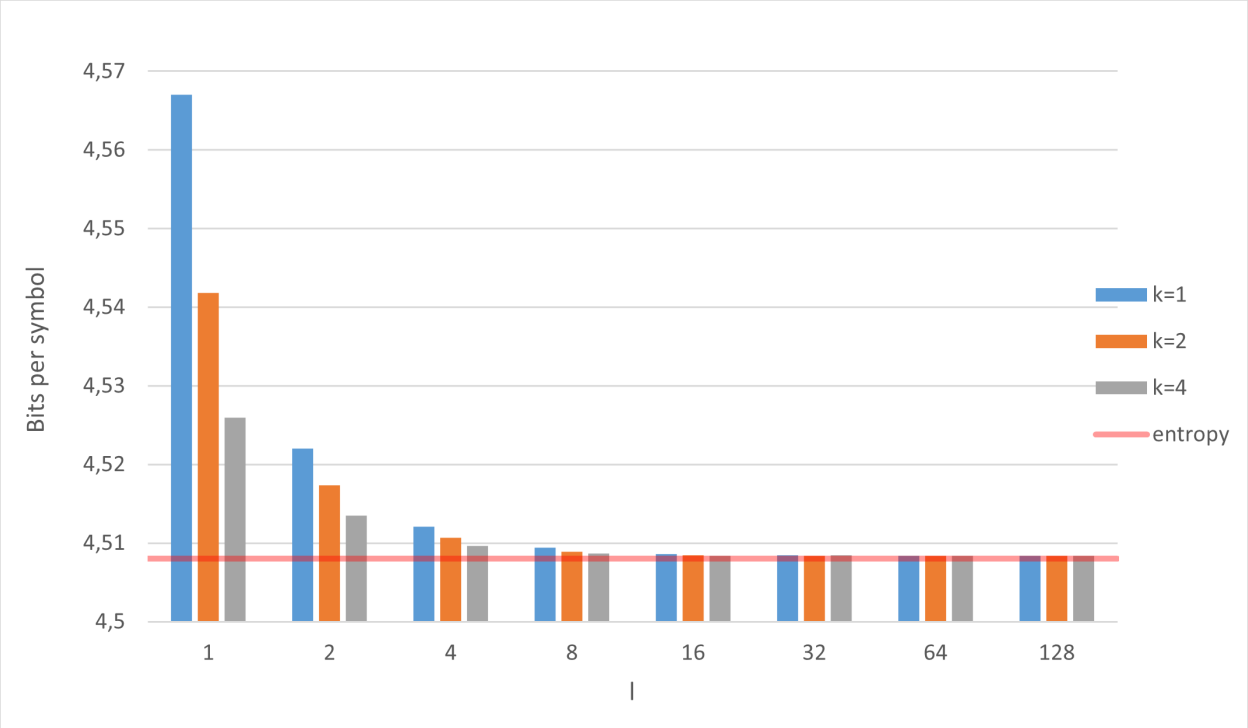

Figure 5.7: Bits per symbol used to encode over 32 KB of data for various values of l and k={1, 2, 4}

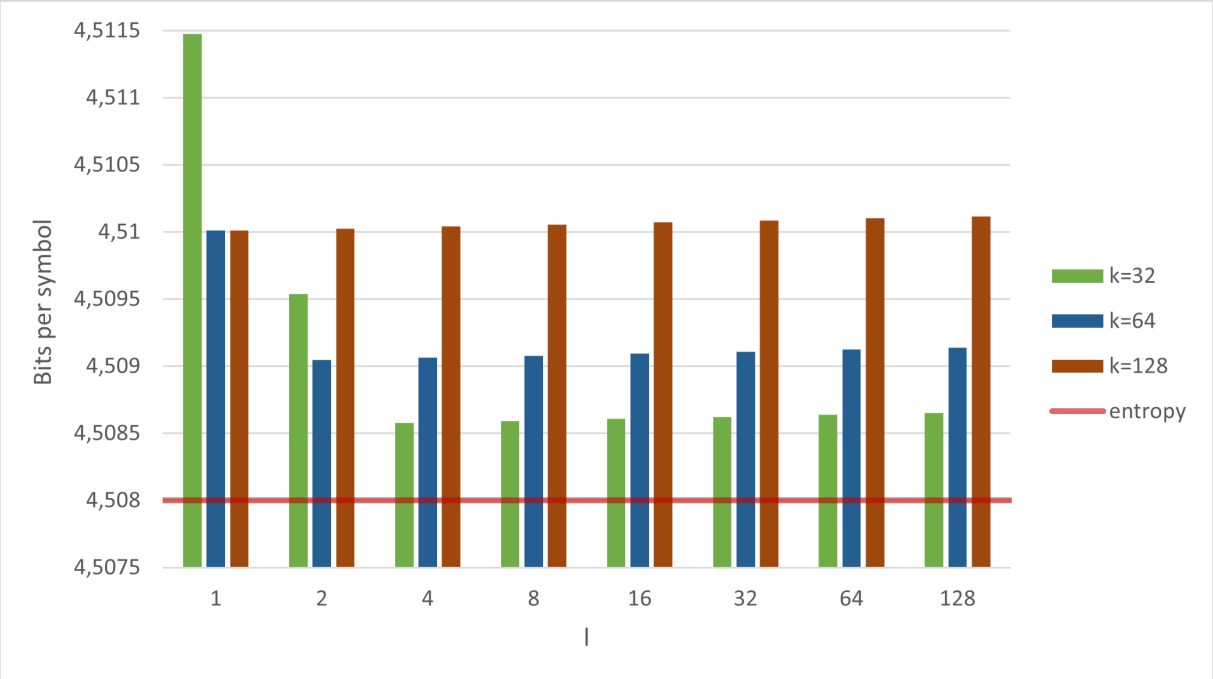

Figure 5.8: Bits per symbol used to encode over 32 KB of data for various values of l and k={32, 64, 128}

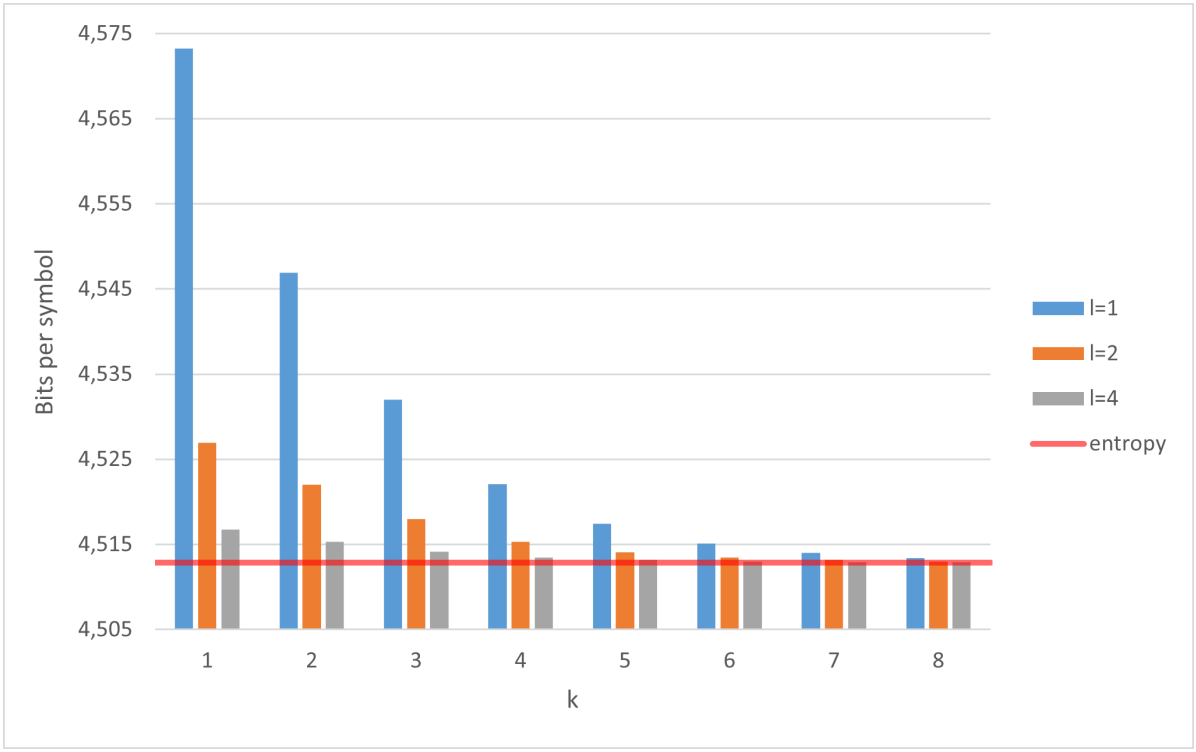

Figure 5.9: Bits per symbol used to encode over 2 GB of data for various values of k and l={1, 2, 4}

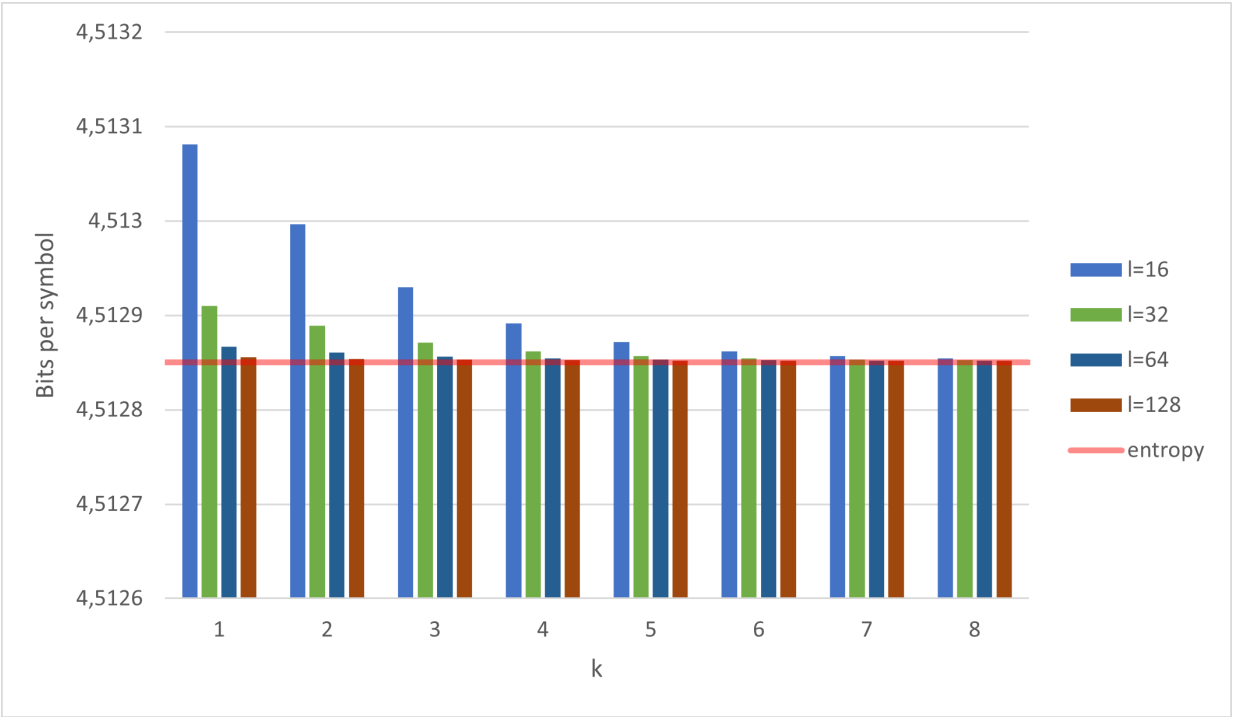

Figure 5.10: Bits per symbol used to encode over 2 GB of data for various values of k and l={16, 32, 64, 128}

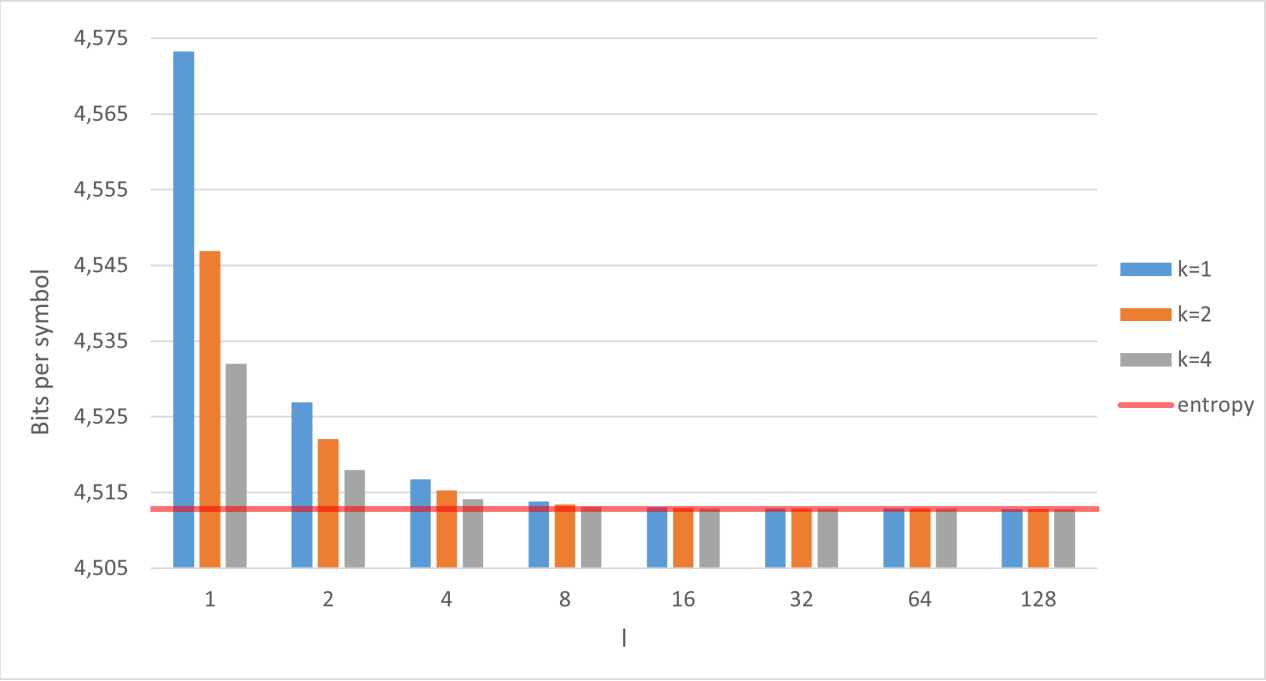

Figure 5.11: Bits per symbol used to encode over 2 GB of data for various values of l and k={1, 2, 4}

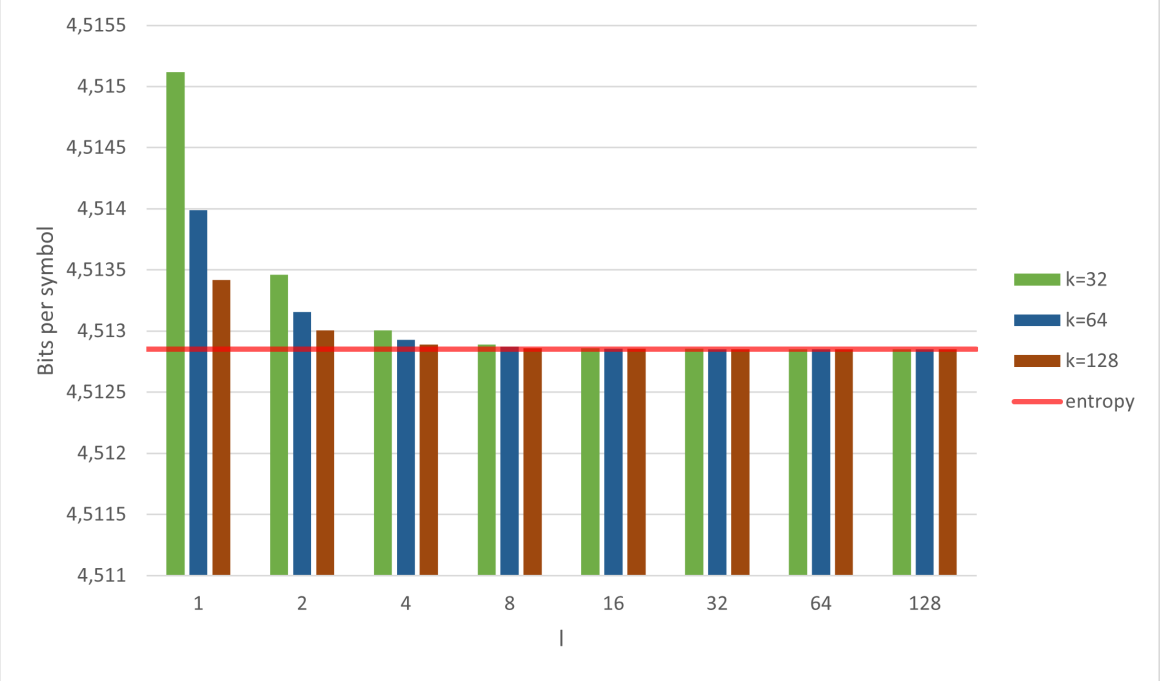

Figure 5.12: Bits per symbol used to encode over 2 GB of data for various values of l and k={32, 64, 128}

5.2. Impact of the l and k parameters on resources used

In order to recognize if the proposed l and k parameters selection truly decreases the number of resources needed for hardware codec implementation, the number of resources needed for the not-optimized encoder was examined. The difference between this not-optimised and the one proposed in this thesis is how the next state's value is calculated: the not-optimised design uses division and modulo operations that are avoided in the optimized one.

l\k	1	2	4	8	16	32	64	128
1	121.22%	122.69%	114.09%	117.27%	126.77%	146.97%	192.16%	269.79%
2	123.07%	113.83%	115.87%	118.55%	128.17%	148.18%	193.44%	270.68%
4	123.71%	114.98%	117.14%	121.86%	129.32%	149.46%	194.77%	272.28%
8	118.42%	117.14%	118.61%	123.07%	131.61%	155.32%	196.05%	273.23%
16	120.71%	118.29%	119.82%	124.41%	132.95%	154.62%	197.39%	274.70%
32	130.98%	120.71%	121.86%	126.20%	134.74%	155.90%	200.76%	275.91%
64	125.56%	121.99%	123.26%	127.66%	135.12%	157.17%	200.00%	277.50%
128	135.76%	124.22%	124.67%	128.87%	136.46%	158.44%	201.27%	278.78%

Table 5.1: Post-synthesis usage of LUT for different values of l and k parameters for not-optimised encoder with 7-bit-width data size as a percentage of the usage of resources for the optimised design

The resource usage of the not optimized design was examined for various values of l and k parameters' values. Tables 5.1, 5.2, 5.3 present accordingly the number of LUTs, flip-flops, and bonded IOB used in design for various values of l and k and 7-bit-width data size as the percentage of resources used in the design described in Section 4.4 for the same bit-width data size (8-bit-width state). Similar percentage results of these resources were obtained for 15-bit-width and 31-bit-width data sizes when compared to optimized design with corresponding data sizes (16-bit-width state and 32-bit-width-state accordingly).

Additionally, Tables 5.4, 5.5, and 5.6 show a number of DSP units used in the design with division and modulo operations for 7-bit, 15-bit and 31-bit-width data size.

The usage of other resources was the same as in the case of the optimized design.

Generally, it can be seen that the not-optimised design uses significantly more hardware resources than the one proposed in this thesis. Even if parameters are equal to one, there are still 21% more LUTs and 14% more flip-flops needed than in the proposed design and it mostly increases for greater values of parameters. Additionally, the not-optimised design also uses DSP units, which were not used at all in the optimized one. Changes in the number of DSP units used sometimes 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.

The exception to the above observation is the number of input-output ports. For parameters equal to one number of bonded IOB is the same as in the optimized design and it constantly grows with the increase of those values. The reason is that the state's width, which is one of the output variables, is dependent directly on the l and k parameters' values. The only other value that affects the number of input-output ports is the maximum size of data and it is constant in this case.

What is more, when comparing the usage of DSP 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's values, as increasing data size causes a change of the range given in equation 2.7. As a result, codec performs division and modulo operations on bigger values, which might require more resources.

l\k	1	2	4	8	16	32	64	128
1	114.16%	114.32%	114.16%	114.54%	115.41%	117.14%	120.59%	127.51%
2	114.22%	114.11%	114.27%	114.65%	115.51%	117.24%	120.70%	127.62%
4	114.11%	114.22%	114.38%	114.81%	115.62%	117.35%	120.81%	127.73%
8	114.27%	114.32%	114.49%	114.92%	115.73%	117.46%	120.92%	127.84%
16	114.38%	114.43%	114.59%	115.03%	115.84%	117.57%	121.03%	127.95%
32	114.43%	114.54%	114.70%	115.14%	115.95%	117.68%	121.14%	128.05%
64	114.59%	114.65%	114.86%	115.24%	116.05%	117.78%	121.24%	128.16%
128	115.03%	114.76%	114.97%	115.35%	116.16%	117.89%	121.35%	128.27%

Table 5.2: Post-synthesis usage of flip-flops for different values of l and k parameters for not-optimised encoder with 7-bit-width data size as a percentage of the usage of resources for the optimised design

l\k	1	2	4	8	16	32	64	128
1	100.00%	106.45%	112.90%	125.81%	154.84%	209.68%	316.13%	525.81%
2	106.45%	109.68%	116.13%	129.03%	158.06%	212.90%	319.35%	529.03%
4	109.68%	112.90%	119.35%	135.48%	161.29%	216.13%	322.58%	532.26%
8	112.90%	116.13%	122.58%	138.71%	164.52%	219.35%	325.81%	535.48%
16	116.13%	119.35%	125.81%	141.94%	167.74%	222.58%	329.03%	538.71%
32	119.35%	122.58%	129.03%	145.16%	170.97%	225.81%	332.26%	541.94%
64	122.58%	125.81%	135.48%	148.39%	174.19%	229.03%	335.48%	545.16%
128	125.81%	129.03%	138.71%	151.61%	177.42%	232.26%	338.71%	548.39%

Table 5.3: Post-synthesis usage of bonded IOB for different values of l and k parameters for not-optimised encoder with 7-bit-width data size as a percentage of the usage of resources for the optimised design

l\k	1	2	4	8	16	32	64	128
1	0	0	2	2	2	2	4	8
2	0	2	2	2	2	2	4	8
4	2	2	2	2	2	2	4	8
8	2	2	2	2	1	2	4	8
16	2	2	2	2	1	3	4	8
32	2	2	2	2	1	3	4	8
64	2	2	2	2	2	3	5	8
128	2	2	2	2	2	3	5	8

Table 5.4: Post-synthesis usage of DSP for different values of l and k parameters for not-optimised encoder with 7-bit-width data size

l\k	1	2	4	8	16	32	64	128
1	1	2	2	2	2	3	5	8
2	2	2	2	2	2	3	5	9
4	2	2	2	2	2	3	5	9
8	2	2	2	1	2	3	5	9
16	2	2	2	1	2	3	5	9
32	2	2	2	1	2	3	5	9
64	2	2	2	2	2	3	5	9
128	2	2	1	2	2	3	5	9

Table 5.5: Post-synthesis usage of DSP for different values of l and k parameters for not-optimised encoder with 15-bit-width data size

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

Table 5.6: Post-synthesis usage of DSP for different values of l and k parameters for not-optimised encoder with 31-bit-width data size

6. Summary

6.1. Conclusions

The proposed design of an ANS encoder meets the objectives listed in Section 1.2. Assigning k and l parameters' values to one allows the elimination of a division operation and replacement of a modulo operation by a subtraction, which are performed every time a new symbol is encoded. As a result, the proposed design requires significantly fewer hardware resources for implementation compared to the one using division and modulo operations. This drop in hardware resources required can vary from 11.5% up to 65%, depending on k and l parameters' values and maximum data size.

A disadvantage of this solution might be a drop in compression efficiency. However, this depends on the bit-size of the data, as the analysis performed as part of this thesis showed, that the larger the data set, the greater should be the values of l and k parameters to achieve the best compression ratio. Additionally, differences between compression ratios for parameters' values equal to one and the best compression ratios obtained in the analysis were about only 1 percentage point.

The only objective of the thesis that was not met from the ones listed in Section 1.2 was measuring the performance time of the created codec. The reason is lack of time to do so.

6.2. Further modification

There are a few modifications of the designed encoder that can be done to adjust it.

Firstly, to improve its performance time, the designed encoder could be changed to accept frequencies only when it should be changed, and not after every data set. As a result, the encoder would encode multiple data sets with the same frequency distribution. However, this solution would be beneficial only if the chosen data sets to encode would have the same or a similar distribution of symbols' frequencies, for example, if the codec was supposed to encode only text in one language.

Secondly, to reduce the number of hardware resources needed for implementation, one could modify the proposed 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.

6.3. Further research and development

Although this thesis covers a wide range of analyzed cases, some areas still could be examined.

Firstly, this thesis covers only the streaming-rANS encoding and does not examine the influence of l and k parameters' values on the streaming-rANS decoder. Based on the fact, that the state's value will always be in the range 4.2, $\left\lfloor \frac{X_t}{M} \right\rfloor$ operation form equation 2.5 will always be equal to one and equation 2.3 can be replaced by subtraction $slot = X_t - M$ as well. Consequently, it could be examined, how much more hardware resources the traditional decoder would require than the one without the division and modulo operations and what differences would be in performance time.

Secondly, for parameter k equal to one, the range given in Equation 2.7 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.

Moreover, this thesis 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, another interesting research could be to examine, if and how assigning l and k parameters' values to one would affect a tabled-ANS encoder hardware implementation. The tabled-ANS encoding algorithm has the same operations when generating an encoding table, thus this process could be reduced. Most existing implementations use approximations to do that and they could be compared to this solution.

List of tables

List of figures

Bibliography

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

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

[3] Repository with source code of ans encoder and python scripts used in this thesis. URL: https://github.com/Sharon131/masters_project, accessed: [2023-08-24].

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

[5] Verilog tutorial. URL: https://www.chipverify.com/verilog/verilog-tutorial, accessed: [2023-05-20].

[6] 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*, 10:2934, 11 2021.

[7] 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)*, pages 2212–2215, 2017.

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

[9] J. Duda. Asymmetric numeral systems, 2009.

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

[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*, pages 238–243, 2010.

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

[13] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

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

[15] S. Mahapatra and K. Singh. An fpga-based implementation of multi-alphabet arithmetic coding. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(8):1678–1686, 2007.

[16] 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)*, pages 256–259, 2015.

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

[18] 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*, 39:711 – 732, 2020.

[19] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, jun 1987.