TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Electrical and Computer Engineering
Institute of Circuits and Systems
Chair of Highly-Parallel VLSI Systems and Neuro-Microelectronics

PROJECT WORK

SW/HW Implementation of Arithmetic Coding for Neural Signal Compression

Student:	Zhuo Wang
Enrollment number:	4985827
Born on:	03.01.1998, China
Supervisor:	Dip.-Ing. Liyuan Guo
	Dr.-Ing. Seyed Mohammad Ali Zeinolabedin
1^{st} Examiner:	Prof. Dr.-Ing. habil. Christian Mayr
2^{nd} Examiner:	Dr.-Ing. Johannes Partzsch
Submit on:	February 21, 2023

Abstract

Neural signals contain a large amount of physiological and pathological information. It plays an irreplaceable role in the clinical diagnosis of brain diseases. In practice, it is necessary to collect neural signals inside the brian from multiple channels for a long time, which will generate a large amount of data. Due to the limitation of transmission bandwidth and power consumption, it is important to compress raw data, reduce device power consumption and meet the transmission bandwidth limit under the condition that the main characteristics of intracranial recorded signals remain unchanged.

Arithmetic coding (AC) as a lossless compression method, can compress neural signals with effectiveness and versatility. With predefined distribution obtained from a training data, AC can reach theoretically near-optimal compression to data from different sources. Moreover, it is efficient to operate on digital circuits. Based on studies of entropy coding methods and previous work of adaptive arithmetic coding (AAC) and pseudo-adaptive Golomb coding (PAGC), this project proposed AC compression for neural signals and other sources, with software and hardware implementation. The main work is as follows:

1. Studies about practical one-dimensional signal compression algorithms like arithmetic coding, research about prerequisites of efficient entropy coding like decorreation with differential pulse code modulation (DPCM).

2. Software implementation of the Arithmetic coding and analysis of its compression performance on neural signals, compared to the existing AAC and PAGC.

3. Synthesizable hardware implementation of the arithmetic coding in real time logistics (RTL) and the wrapper module for the constant data rate.

4. Verification of the hardware implementation against a software reference, a proper area analysis for different operating frequencies.

Statement of Authorship

I hereby declare that I completed this thesis with the topic

SW/HW Implementation of Arithmetic Coding for Neural Signal Compression

on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this, nor a similar work, has been published or presented to an examination committee.

Dresden, February 21, 2023

Contents

List of Figures

List of Tables

List of Abbreviations

AAC	Adaptive arithmetic coding
AC	Arithmetic coding
ADC	Analog to digital converter
BMI	Brain-machine interface
CPB	cycles per byte
DCT	Discrete cosine transform
DFT	Discrete fourier transform
DPCM	Differential pulse code modulation
DSP	Digital signal processor
ECG	Electrocardiogram
FFT	Fast Fourier transform
FSM	Finite state machine
HW	Hardware
KLT	Karhunen-Loève transform
LL	Lossless
NLL	Near-lossless
PCM	Pulse code modulation
RLE	Run-length coding
RF	Radio frequency
SRAM	Static random-access memory
SSR	Space saving ratio
SW	Software

1 Introduction

1.1 Background

In the past few years, brain-related studies have become a popular topic. With the recent advancements in neuroscience research and relevant developments of neural interfacing systems such as Brain-machine interfaces (BMIs), people are looking forward to monitoring the brain's electrical signals for research and diagnosis, and utilizing the knowledge in bionics. As a part of BMI, neural recording implants are in charge of capturing and pre-processing brain signals for transmission. However, one main challenge for neural implants is power consumption limit. Although the latest technology has proved the possibility of implanting thousands of electrodes, existing radio-frequency (RF) transceivers is not capable of handling large data transmission with thousands of channel counts, since power and chip area are strictly limited. Therefore, an on-chip compression hardware accelerator is required to save power when transmitting the raw data.

Data compression also called source coding, is the process of converting an input data stream (the source stream or the original raw data) into another data stream (the output, the bitstream, or the compressed stream) that has a smaller size [1]. From the 1980s, the digital storage cost started increasing exponentially, while many data compression algorithms were developed at the same time. With better compression techniques, people can lower the cost of storage and latency of transmission. Nowadays, compression techniques are applied everywhere when computer science and digital signal processing are developing rapidly.

1.2 Basic Concepts

In general, data can be compressed by eliminating data redundancy and irrelevancy [2]. A simple but basic law for to reduce redundancy is to represent common symbols with short codes and rare symbols with long codes. Normally it is possible, since most information in computers are redundant to make it easier for processing. A more theoretical explanation is, any non-random data must possess some repeat or related structures, and these structures could be replaced by

shorter representations. True random information where all symbols are equally distributed, on the other hand, is not compressible.

In the field of data compression, several concepts may be mentioned in this thesis.

1) **Encoder/decoder**: An encoder, or compressor, compresses the raw data from the input side and creates an output stream with lower-redundancy data. A decoder, or decompressor, decompresses the input data and recovers it to its original form. Codec is the short term for encoder and decoder.

2) **Adaptive/Non − adaptive/Semi − adaptive**: A nonadaptive compression method is based on fixed process and parameters, which do not change with various inputs. An adaptive method modifies itself when processing the raw data. For a semi-adaptive method, usually the statistics of raw data are collected first and then used as compression parameters.

3) **Lossy/lossless compression**: Some compression methods sacrifice some information for better performance, which means the decoder cannot restore the original information 100%. Such methods are common in compressing videos or audio, where the loss is not significant for human senses. On the other hand, the data from text files or computer programs should not be changed after compression and decompression. In such circumstances, lossless compression is desired.

4) **Compression performance**: Several parameters are used to describe the performance of the compression method. Space saving ratio (SSR) is defined as

$$space\ saving\ ratio = 1\ -\ \frac{size\ of\ output\ stream}{size\ of\ input\ stream} \tag{1.1}$$

A value of 0.6 means that the data occupies 40% of its original size after compression. A closer to 1 value is desired for an encoder. In this thesis, this more intuitive definition is used to represent a better compression performance with a higher value.

The speed of compression can be measured in cycles per byte (CPB). It is defined as the average clock cycles it takes to compress one byte.

5) **Entropy**: As one of other terms Prof. D. Salomon mentioned in his Ten Commandments of Compression, entropy is second only to redundancy in data compression [1]. In information theory, entropy is defined to be proportional to the minimum number of yes/no questions needed to reach the answer to a question. It is the average level of information or uncertainty inherent to the variable's possible outcomes. Given a discrete random variable X with n possible values x_i, which occurs with possibility P_i, the entropy is defined as

$$H(X) = -\sum_{i=1}^{n} P_i log_2 P_i \tag{1.2}$$

We can see that the entropy of a set of n symbols depends on the individual probabilities P_i, and is largest when all probabilities are equal.

There is an inner relation between entropy and redundancy. When entropy reaches its maximum, the data is considered to have the maximum information content and thus cannot be further compressed. At this moment we define the redundancy to be zero. Due to this theory, in some compression methods, encoders compress data by assigning each symbol a variable length of the codeword, which is nearly proportional to the negative logarithm of its probability of occurrence. In this way, the most common symbol is assigned with the shortest codes, and the redundancy is reduced. We will shortly introduce three methods of entropy encoding in the following part.

- Entropy coding – Huffman coding

At the start of the history of compression methods, in 1952, David A. Huffman proposed a variable-length encoding, which is known as Huffman encoding. This encoding method uses a binary tree construction method to encode characters with the shortest average length according to the probability of occurrence of characters. it produces the best code when the probabilities of the symbols are negative powers of 2. However, Hoffman coding is not always optimal for all cases. Compared to well-known arithmetic coding, Huffman coding is faster and safer, since the bit error will not propagate. But when a better compression ratio is required, especially when the probability of the most common symbols is around 15% - 20%, the result of arithmetic coding can be 0.1 bit per symbol less than that of Huffman coding [1]. As a result of pursuing the best compression ratio, arithmetic coding and its fellow entropy coding methods are more commonly used today.

- Entropy coding – Arithmetic coding

Arithmetic coding (AC) performs better than other entropy coding methods in many terms, such as effectiveness, versatility, and simplicity. It can work efficiently in many circumstances and purposes. Arithmetic coding starts with a certain interval, reads the input stream symbol by symbol, and uses the probability of each symbol to narrow the interval. In a word, arithmetic coding uses a single codeword (usually very long) to represent the entire input file. Arithmetic coding has many desirable features [3]. When applied to independent and identically distributed (i.i.d.) sources, the compression of each symbol is proved to be optimal. For complex sources, it can simplify automatic modeling. It is effective in a wide range of situations and compression ratios. Since its basic process is arithmetic,

CPUs or digital signal processors (DSPs) could support the processing with high efficiency, which is also the reason we apply mainly this method in our research. The detailed analysis will be presented in the following chapters, where software and hardware implementations are studied.

- Golomb coding

Golomb coding is another lossless data compression method [4]. Symbols following a geometric distribution will have a Golomb code as an optimal prefix code, making Golomb coding highly suitable for situations in which the occurrence of small values in the input stream is significantly more likely than large values. Rice coding, as a subset of Golomb coding, uses a power of two numbers as its tunable parameters. This feature enables Rice coding to be easy for computer processing and produces a simpler but possibly suboptimal code.

The best performance that Rice-Golomb coding can reach for specific symbols is considered to be better than arithmetic coding's. The comparisons will be made briefly in this thesis.

1.3 Contributions and thesis Formulation

In this thesis, arithmetic coding is introduced, including the algorithm and its compression performance on neural signals. Meanwhile, other methods to help compression like differential pulse-code modulation are studied. A synthesizable hardware implementation of encoder design is brought up and verified to be trustworthy compared with software reference. Lastly, Some result analysis of the current design is made.

The thesis includes 3 main chapters. Chapter 2 introduces two algorithms applied in this study. Chapter 3 brings up the feasible software implementation of the encoder and decoder. Chapter 4 shows the results of hardware implementation with corresponding result analysis.

2 Compression Algorithms

In this chapter, two algorithms of neural signal compression will be formulated, as well as the problems with practical applications. To realize the practical performance of compression, a well-balanced data pre-processing procedure and a suitable compression algorithm are both important.

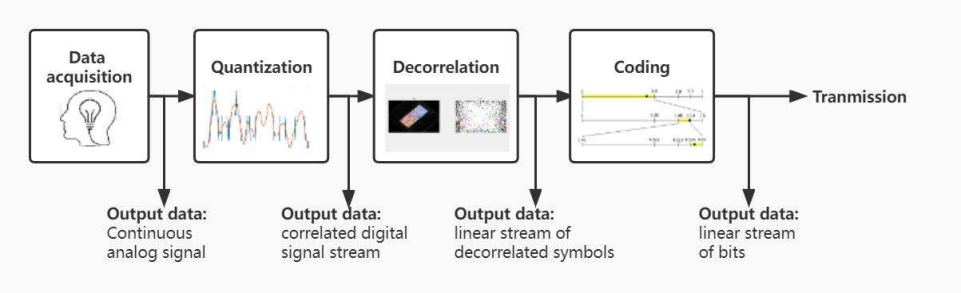


Figure 2.1: Neural signal compression flow

As shown in Figure 2.1, before transmission, neural signals collected should be digitalized by analog to digital converter (ADC) first, and then processed and compressed to reduce the transmission burden. Meanwhile, this part of circuits should also maintain minimal power and area consumption, since in a practical situation, the simultaneous operation of hundreds of channels will bring increased resource intensity. For signal channel transmission, it is recommended to perform concise but highly efficient signal processing algorithms. Therefore, DPCM and arithmetic coding are applied for decorrelation and coding process respectively.

The first part of this chapter leads to the concepts of decorrelation in the pre-processing step, and a concise but sufficient decorrelation method for single-channel neural signal compression, Differential Pulse Code Modulation (DPCM). The second part of this chapter explains the statistical compression algorithm utilized after the preprocessing, the Arithmetic Coding AC algorithm.

2.1 Preprocessing - Decorrelation

Like other analog signals captured from natural world, neural signals have strong correlations in multiple dimensions. Dependencies among neurons decide the over-

all encoding capacity of the whole network. For signals captured from single neuron, strong correlation also shows in time domain. As we discussed in the first chapter, in the field of data compression, normally correlation is considered as redundancy, which is expected to be eliminated. The goal, is to reduce the auto-correlation and cross-correlation, while preserve other aspects of signal. For this research single channel signal compression is studied, so this part is mainly focused on preprocessing the data by reducing the correlation in time domain, before the entropy compression algorithm comes in.

2.1.1 Overview of Decorrelation Methods

The decorrelation methods can be classified based on different principles. One is related to signal transform, mostly applied for lossy compression, such as image compression. Linear or nonlinear decomposition of 1D or 2D signals is performed to concentrate energy in frequency domain, for example, the famous Discrete Fourier Transform (DFT), Karhunen-Loève transform (KLT), Discrete Cosine transform (DCT), etc. These methods normally consume higher cost of computation and storage, therefore is not desired in low power applications.

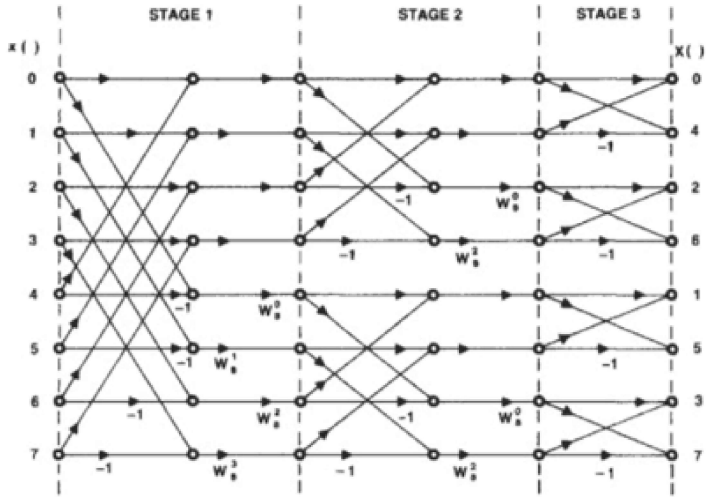


Figure 2.2: FFT "Butterfly" structure [5]

The other category is based on signal prediction. Prediction is a successful technique in signal processing. Here, N-th order (one or multi-dimensional) Markov process is performed to reduce the entropy of prediction error. A Markov process is one in which predictions about the future can be made based on the current state alone, and the predictions are as good as those made based on all past

Figure 2.3: Before Delta coding

Figure 2.4: After Delta coding [6]

states.[7] After prediction, the original signal or the current state is replaced by prediction error, which has a smaller entropy and performs better as the input of future statistical compression algorithms. Some classical predictors are based on signal processing, such as DPCM and delta coding (both are linear predictors), and non-linear predictors as well as adaptive predictors.

From the perspective of the complexity of building digital integrated circuits, DPCM can maximize practicality and functionality without utilizing much resources. The details of DPCM principles will be introduced in the next part.

2.1.2 DPCM Principles

Differential pulse code modulation or DPCM, is the extension of pulse code modulation (PCM). It was invented by C. Chapin Cutler in 1950s. By performing DPCM to digital input data stream, the temporal correlation of the signal is well reduced. In general, the reduction of waveform redundancy is achieved by replacing highly correlated signal with a less correlated estimation error signal. As shown in Figure 2.5, using prediction error to represent original signal removes temporal correlation or redundancy, meanwhile reduces the peak amplitude of the stream and saves power for transmission.

Since the estimation error stream provides same amount of information as the original data stream does, there should be zero loss during reconstruction of raw data. The major source of reconstruction error is actually the amplitude quantization error. In the field of digital signal processing, the quantization is operated before encoder, therefore DPCM coder can be lossless.

DPCM encoder performs signal prediction in either of these two processes:

1) take the values of two consecutive samples; calculate the difference between two samples; then output the difference.

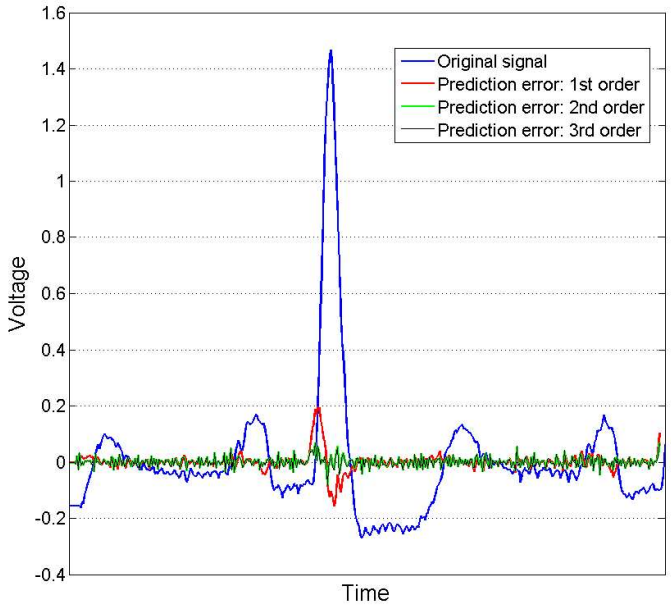


Figure 2.5: Electrocardiogram (ECG) signal and DPCM results[8]

2) take the difference between input sample and the output of a local model of the decoder process; output the difference.

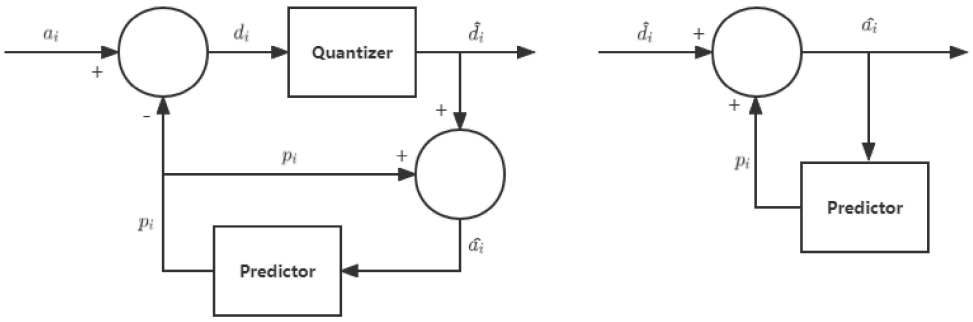


Figure 2.6: A DPCM encoder and decoder

The first process is actually the simplification of the second process. Figure 2.6 shows the diagram of a typical DPCM codec. Here, a_i is input data stream, d_i is the predictor output stream, and the quantized signals are denoted with $\hat{}$. If the quantizer block is bypassed in a sequential logic, meanwhile consider the predictor

output to be the last sample value, $p_i = a_{i-1}$, then the process is simplified to the first one, where $\hat{d}_i = d_i = a_i - p_i = a_i - a_{i-1}$. Such simplified process has no need of model training or extra memory, and zero information loss after reconstruction, thus is desired real-time and power sensitive circuits. In the following part of this research, the term DPCM refers to this simplified 1-D model.

2.1.3 DPCM model in this work

The 1st-order DPCM is defined as

$$dpcm\_1st(i) = a(i) - a(i-1) \tag{2.1}$$

where $a(i)$ denotes the raw data stream. The encoder simply uses the difference between current input sample and last input sample as the output code.

The decoder can easily generate reconstructed signal operating in the opposite direction, the sample is reconstructed by the sum of last sample reconstructed and the current DPCM code.

The DPCM encoder could have multiple stages. When multiple DPCM stages are cascaded, a higher-order DPCM encoder is obtained. For example, 2nd-order and 3rd-order DPCM are derived respectively as

$$\begin{aligned} dpcm\_2nd(i) &= dpcm\_1st(i) - dpcm\_1st(i-1) \\ &= a(i) - a(i-1) - (a(i-1) - a(i-2)) \\ &= a(i) - 2 \cdot a(i-1) + a(i-2) \end{aligned} \tag{2.2}$$

$$\begin{aligned} dpcm\_3rd(i) &= dpcm\_2nd(i) - dpcm\_2nd(i-1) \\ &= a(i) - 2 \cdot a(i-1) + a(i-2) - (a(i-1) - 2 \cdot a(i-2) + a(i-3)) \\ &= a(i) - 3 \cdot a(i-1) + 3 \cdot a(i-2) - a(i-3) \end{aligned} \tag{2.3}$$

More than three stages can be realized in a similar way, but the effect of decorrelation should be investigated. As a matter of fact, a DPCM encoder built from too many cascaded stages may even worsen the decorrelation performance, meanwhile brings extra cost on circuit elements.

2.1.4 Result Prediction and Expectation

As the result of decorrelation, most common samples at DPCM output should be distributed around zero. With high-order DPCM, the effect of decorrelation should be optimized.

The concentrated distribution of the most common symbols has one significant benefit: the entropy of the data stream is lowered. For the future entropy coding

algorithms, it is essential to have such a low entropy source as input, which helps to reduce the total bits to represent the raw sequence in a lossless way. Consequently, the performance of compression is optimized. The details of entropy coding will be discussed in the next chapter.

2.2 Entropy Coding Algorithm

The decorrelated data stream still consumes the same transfer resources as the original does. To realize real compression, proper coding algorithms must be implemented. Entropy coding is one of the lossless compression categories. These algorithms work based on the probability distribution of source symbols, trying to reach the best of information entropy. According to Shannon's source coding theorem, the optimal code length for a symbol is $-log_b(P)$, where b is the number of symbols used to make output codes and P is the probability of the input symbol. However, this aim is hardly achieved in real applications. But still many algorithms are invented to help us get as close as we can, such as Golomb coding, Huffman coding, arithmetic coding, etc.

Source coding theorem

(i) The average number of bits/symbol of any uniquely decodable source must be greater than or equal to the entropy H of the source.

(ii) If the string of symbols is sufficiently large, there exists a uniquely decodable code for the source such that the average number of bits/symbol of the code is as close to H as desired.

The specific coding method implemented in this research is arithmetic coding (AC). Its basic idea is like other entropy coding methods: assign common input symbols with short codewords, and replace each fixed-length input symbol with the corresponding variable-length output codeword. We will discuss the principles of arithmetic coding in this section, and give a brief sample of arithmetic coding model with integer inputs and outputs to help understanding.

2.2.1 Introduction of Arithmetic Coding

The basic function of arithmetic coding is to compress the input data sequence into one (normally long) code which contains all the information from the input stream. Actually, the final codeword here comes from a very small interval. To begin with, the algorithm picks a certain interval, then it reads the input symbol by symbol and narrow the interval according to the probability of each symbol. The

codeword can be any value located within the last interval. Like other entropy coding algorithms, arithmetic encoder is designed in a way that symbols with higher probability narrows the interval less that lower-probability symbols, in other words, fewer bits are used to describe a high-probability symbol.

The principle of arithmetic coding was first brought up by Peter Elias in the early 1960s [9]. In 1976, two independent researches of implementations of arithmetic coding were published by Jorma J. Rissanen [10] and by Richard C. Pasco [11], respectively. In the next year, Rissanen's work was patented. Nowadays many techniques of arithmetic coding have been public since the patent expired in 2006.

In this chapter, the original interval is specified as [0, 1), which means the range of real numbers from 0 to 1, including 0 but excluding 1. After the iteration of encoding steps, the final interval should always be within this range. For example, the final interval might be [0.972643201, 0.972643244), thus any code between 971643201 and 972643244 but excluding 972643244 could be the output codeword (0. is ignored). In this case, it is always easy to pick the left bound as the output code.

2.2.2 Basic representations of AC

Let Ω be a data source with symbols s_k coded as integers in the set 0, 1, ..., M − 1, and let $S = s_1, s_2, ..., s_N$ be a sequence of N random symbols put out by Ω. Assume the source symbols are independent and identically distributed (i.i.d.) with probability

$$p(m) = Prob\{s_k = m\}, \quad m = 0, 1, 2, ..., M-1, \quad k = 1, 2, ..., N. \tag{2.4}$$

Assume that for all symbols $p(m) \neq 0$, and define $c(m)$ to be cumulative frequency [12],

$$c(m) = \sum_{s=0}^{m-1} p(s), \quad m = 0, 1, ..., M. \tag{2.5}$$

Here, $c(0) \equiv 0$, $c(M) \equiv 1$, and

$$p(m) = c(m+1) - c(m). \tag{2.6}$$

We use bold letters to represent the vectors with all $p(m)$ and $c(m)$ values,

$$\boldsymbol{p} = [p(0) \quad p(1) \quad ... \quad p(M-1)],$$

$$\boldsymbol{c} = [c(0) \quad c(1) \quad ... \quad c(M-1) \quad c(M)].$$

2.2.3 Encoding Process

The prerequisite of normal arithmetic coding is to calculate the probabilities of each symbol. The best performance requires the calculation through the whole input file, but it would not be possible for many circumstances or realistic for universal codecs. Normally the probabilities are estimated from the frequencies of different possible input sources, and generated to be a universal distribution, which can be applied to both encoder and decoder sides.

***Encoder example* 1**

The following example explains a simple model for arithmetic coding, using a source Ω with four i.i.d. symbols ($M = 4$). Assume the distribution of the symbols are $\boldsymbol{p} = [0.4 \quad 0.3 \quad 0.1 \quad 0.2]$ and $\boldsymbol{c} = [0 \quad 0.4 \quad 0.7 \quad 0.8 \quad 1]$, and the sequence to be encoded is $S = \{0, 1, 1, 2, 3, 0, 2\}$.

Firstly, the interval [0, 1) is divided into four subintervals, each with length equal to the probability of corresponding symbol. To be specific, interval [0, 0.4) corresponds to $s_1 = 0$, interval [0.4, 0.7) corresponds to $s_1 = 1$, interval [0.7, 0.8) corresponds to $s_1 = 2$, and interval [0.8, 1) corresponds to $s_1 = 3$. The first input symbol decides which subinterval should be used for further compression, in this case symbol "0" represents the next encoding iteration should start from [0, 0.4). The second symbol "1" reduces the interval [0, 0.4) in the same way to [0.16, 0.28). Each symbol processed gives upper and lower boundaries of the next interval, which are updated by

$$\Phi_k(High) = \Phi_{k-1}(Low) + c(s_k) * (\Phi_{k-1}(High) - \Phi_{k-1}(Low)) \tag{2.7}$$

$$\Phi_k(Low) = \Phi_{k-1}(Low) + c(s_{k-1}) * (\Phi_{k-1}(High) - \Phi_{k-1}(Low)) \tag{2.8}$$

Here, Φ represents interval, and the index of current interval is denoted by k.

In the second iteration, according to equations 2.8 and 2.9, the subinterval [0.16, 0.28) are calculated from [0, 0.4) by $0+0.4\times(0.4-0) = 0.16$ and $0+0.7\times(0.4-0) = 0.28$. Step by step, the last input symbol reduces the interval to [0.2362816, 0.2363104), whose length is $1/(2.88 \times 10^5)$ of the original interval. The detailed progress is shown in Figure 2.7 and Table 2.1. After several iterations the interval becomes extremely small, so the graphically magnification is applied to some steps so that the process is more understandable.

The interval length reduction during the process of encoding each symbol has connection with the probability of the current symbol. For example, every occurrence of symbol "0" must result in a reduction of the interval length to 40% of its current length, which is equal to the probability of symbol "0" in Ω. In this way, with the progress of encoding, the distribution of code values should be closer to a uniform distribution [13].

The last task of encoder is to pick a codeword within the last interval to represent the data sequence. Any code value in $\Phi_N(S)$ can be decoded correctly, so we could

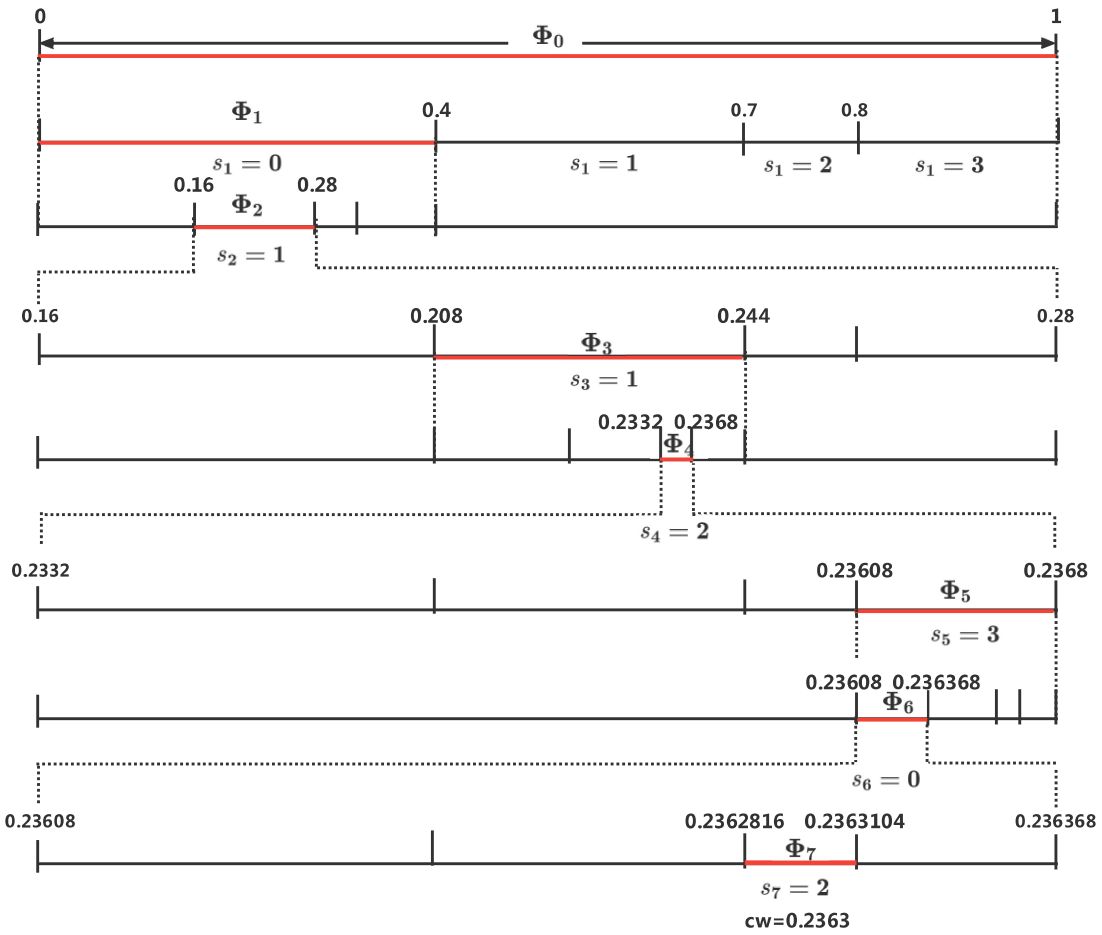


Figure 2.7: Graphical representation of the arithmetic coding process of Example 1: The interval $\Phi_0 = [0, 1)$ is divided into nested intervals according to the probability of the data symbols. Selected intervals corresponding to the data series S = {0, 1, 1, 2, 3, 0, 2} are represented by bold lines.

just pick a shorter one, e.g., 0.2363. Normally, "0" can be omitted, so the final output can be 2363. This value contains all information of the input sequence and can be transmitted through the channel to the decoder.

In case of binary representation is preferred, dichotomy is applied in the interval reduction progress. The shortest binary codeword is obtained with

$$0.001111000111111_2 = 0.236297607421875,$$

costing 15 bits to output.

When interval length is halved, the number of output bits is increased by one. According to the scaling of interval length, the minimum number of bits required

for representing $\Phi_N(S)$ is

$$B_{min} = \lceil -log_2(l_N) \rceil \quad bits. \tag{2.9}$$

The final interval in Example 1 is 0.0000288, thus $B_{min} = \lceil -log_2(0.0000288) \rceil = 16$ bits. The reason of one more bit here in calculation is we simply eliminated the tailing zero from the binary representation we chose. However, with such representations the length of codeword in binary is saved only by one bit, which is more than depressing in practice. The practical realization of arithmetic encoder with binary-outputs will be discussed in the next chapter.

Iteration k	**Input Symbol** s_k	**Low Value** $\Phi_k(Low)$	**High Value** $\Phi_k(High)$	**Range** l_k
0	-	0	1	1
1	0	0	0.4	0.4
2	1	0.16	0.28	0.12
3	1	0.208	0.244	0.036
4	2	0.2332	0.2368	0.0036
5	3	0.23608	0.2368	0.00072
6	0	0.23608	0.236368	0.000288
7	2	0.2362816	0.2363104	0.0000288

Table 2.1: Arithmetic encoding results for Example 1

2.2.4 Decoding Process

The arithmetic decoder operates in reverse. The basic workflow is starting by inputting codeword. The decoder decides which interval the code value belongs to and gives the current symbol decoded. Then the code value is updated for next iteration, according to

$$cw_k = (cw_{k-1} - c(k)) \quad / \quad p(k). \tag{2.10}$$

Table 2.2 illustrated the process of decoding the codeword obtained from Example 1. The decoding process should have finished after seven correct output symbols. Instead, the last two rows show that the decoder continues to decode and generates meaningless output symbols. Such property is not desired. However, it seems like an unsolvable problem under current operating schemes. The reason is explained below.

The arithmetic encoding actually maps intervals to set of sequences. Each subinterval itself leads to a finite code sequence, while the values within the interval

Iteration k	**L-bound** c_k	**R-bound** c_{k+1}	**Range** p_k	**Code Value** $cw_k = \frac{cw_{k-1}-c(k)}{p(k)}$	**Decoded Symbol** s_k
0	0	1	1	0.2363	-
1	0	0.4	0.4	0.59075	0
2	0.4	0.7	0.3	0.635833	1
3	0.4	0.7	0.3	0.786111	1
4	0.7	0.8	0.1	0.861111	2
5	0.8	1	0.2	0.305555	3
6	0	0.4	0.4	0.763888	0
7	0.7	0.8	0.1	0.638888	2
8	0.4	0.7	0.3	0.796196	1
9	0.7	0.8	0.1	0.962962	2
...					

Table 2.2: Arithmetic decoding process for Example 1

corresponds to sequences starting from that finite sequence. For example, $\Phi_5 = [0.23608, 0.2368)$ represents the sequence {0, 1, 1, 2, 2}, while each real number within that range corresponds to a sequence that starts as {0, 1, 1, 2, 2, ...}. As the result, the code value $cw = 0.2363$ not only provides the correct sequence of input symbols, but also continuous decoding results without meaning.

There are two solution to that problem:

(i) Provide the total number of encoded symbols (N) in the beginning of the compression result.

(ii) Define an additional symbol indicating the end of the input stream, e.g., "end-of-file" or "EOF", with a small probability compared to all official symbols.

The final benchmark of decoder is correct reconstruction of input stream. Since different sequences correspond to non-overlapping intervals, it is guaranteed that they cannot produce the same codeword.

2.2.5 Optimality Analysis

According to information theory [14] and equation (1.2), the average number of bits needed for coding each symbol from a stationary and memoryless source Ω cannot be smaller than its entropy $H(\Omega) = \sum_{m=0}^{M-1} p(m) log_2 p(m)$ bits/symbol. Now the optimality of arithmetic coding is verified below.

Assume the distribution in Example 1 still applies, $\boldsymbol{p} = [0.4 \quad 0.3 \quad 0.1 \quad 0.2]$ and $\boldsymbol{c} = [0 \quad 0.4 \quad 0.7 \quad 0.8 \quad 1]$. As discussed in 2.2.3, each occurrence of one symbol m causes the interval length to be reduced to $p(m)$ of its current length. This can be verified by a subtraction of (2.7) and (2.8):

$$\begin{aligned} l_k &= \Phi_k(High) - \Phi_k(Low) \\ &= (c(s_k) - c(s_{k-1})) * (\Phi_{k-1}(High) - \Phi_{k-1}(Low)) \\ &= p(s_k) * l_{k-1} \end{aligned} \tag{2.11}$$

The encoder reduces the interval length by a factor equal to the symbol probability in order to obtain code values uniformly distributed in the interval [0, 1). For example, if 40% of sequences start with "0", then 40% of the code values must be within the interval assigned to those sequences, which is only possible within the interval we specified for the first symbol "0" length equal to its probability, $p(0) = 0.4$. This feature shows that the necessary condition for optimality.

To verify the sufficient condition, consider the compression output with overhead of σ bits, which may contain

- information of distribution ($\boldsymbol{p}$ or $\boldsymbol{c}$);
- a number that illustrates the number of symbols compressed;
- extra bits to save the output stream with integer number of bytes.

From (2.9), the number of bits used to compress a sequence S satisfies

$$B_S \leq \frac{\sigma - log_2(l_N)}{N} \quad bits/symbol. \tag{2.12}$$

From (2.11) the final interval length can be obtained by

$$l_N = \prod_{k=1}^{N} p(s_k), \tag{2.13}$$

thus

$$B_S \leq \frac{\sigma - \sum_{k=1}^{N} log_2 p(s_k)}{N} \quad bits/symbol. \tag{2.14}$$

Define $E\{\cdot\}$ as the expected value operator, the expected number of bits to compress each symbol is

$$\begin{aligned} \bar{B} = E\{B_s\} &\leq \frac{\sigma - \sum_{k=1}^{N} E\{log_2 p(s_k)\}}{N} \\ &= \frac{\sigma - \sum_{k=1}^{N} \sum_{m=0}^{M-1} p(m) log_2 p(m)}{N} \\ &\leq H(\Omega) + \frac{\sigma}{N} \end{aligned} \tag{2.15}$$

Add the left bound according to (1.2),

$$H(\Omega) \leq \bar{B} \leq H(\Omega) + \frac{\sigma}{N}, \tag{2.16}$$

then one can anticipate that

$$\lim_{N \to \infty} \bar{B} = H(\Omega), \tag{2.17}$$

which indicates that the result of arithmetic coding does achieve optimal compression performance.

2.2.6 Properties of Arithmetic Coding

Arithmetic coding has many benefits. One is optimality, which is proved in section 2.2.5, reveals that the compression performance of arithmetic coding in practice is better than other algorithms. Practical arithmetic coding is very nearly optimal, especially when probability of some single symbol is close to 1, the compression performs much better than other methods.

Another important advantage is flexibility. Arithmetic coding separates the source modeling and the coding process, which allows conjunction with different coding methods (see Figure 2.8). Such separation provides considerable flexibility in implementations, including adaptive coding method, while on the other hand increases time and space consumption communicating between the model and the coder. We will discuss this cost with hardware implementation practice in chapter 4.

Arithmetic coding has one important feature that the encoder and the decoder make synchronized decisions. The encoder may change the parameters or distributions while encoding, and the decoder is capable of capturing and copying the same changes during its operation. This property is essential for adaptive schemes. Moreover, since arithmetic coding uses one very long code to represent the whole input file, one single flaw in encoding result could be deadly. But with synchronized behavior, even if the encoder makes approximations, the decoder still can work correctly as long as it makes identical approximations. This helps us to implement shifting and interval rescaling in practice, which will be discussed in the next section.

Except from those benefits, arithmetic coding has some disadvantages. One is the speed limitations. Each iteration in the encoding process requires at least one multiplication, for some implementations, up to two multiplications and two divisions. Moreover, the distribution lookup operations and for adaptive applications, update operations, all slow down the encoding progress. Another disadvantage is that arithmetic encoder is not compatible with parallel coding, since it does not

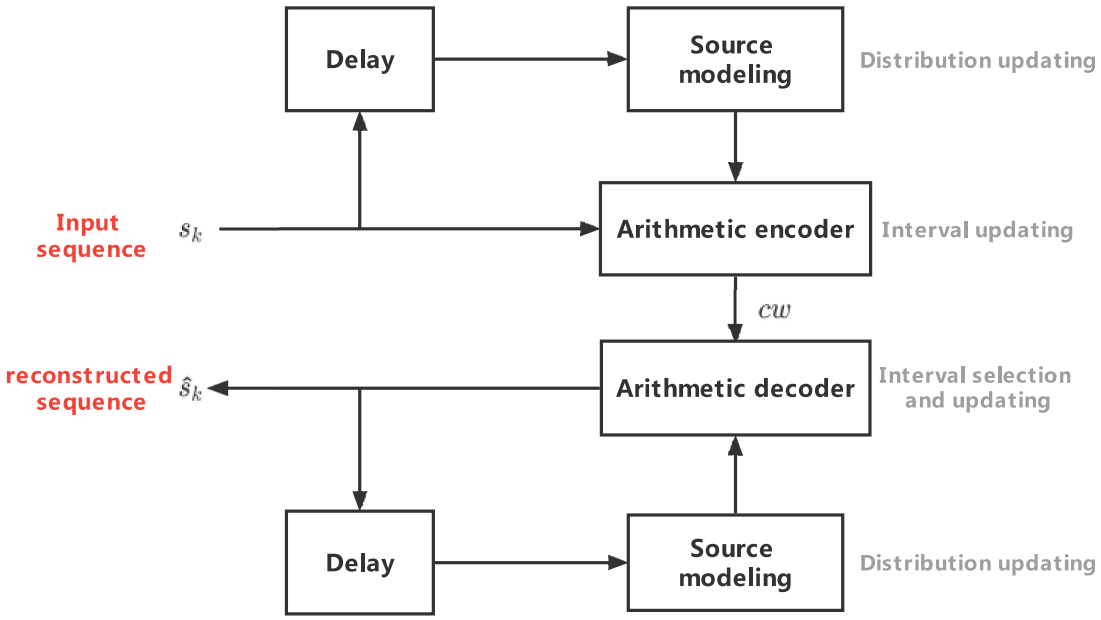


Figure 2.8: Separation of coding and source modeling

produce a prefix-free code. In addition, poor error toleration is also a concern, especially in adaptive models.

2.2.7 Implementation Techniques

In this section some techniques of practical arithmetic coding are introduced. There are two main concerns when the algorithm is implemented. One is the floating-point values, which cause lower speed in processing and loss of precision. In any practical implementations, floating points should be replaced by integers with sufficient precision. Another concern is the expanding bits required to represent the shrinking intervals. The increment of data width will soon reach the hardware limitations and cause loss of precision, so the computation and outputting should be continuous, not accumulated. Luckily, there are solutions to both problems.

(I) Interval Rescaling
Interval rescaling helps with intervals that are too narrow. According to equation (2.11), the proportion between subdivided intervals has no relation with initial or any intervals before, it only relates to the current coded data. For example, if the initial interval Φ_0 is changed to $[2, 4)$, the coding process remains the same, except all intervals are enlarged to two times and shifted by two.

Such interval rescaling can also be performed in the middle of encoding. If at some certain stage m, the interval is scaled by γ and shifted by δ, then the interval can be represented by

$$\begin{aligned} \Phi'_m(Low) &= \gamma(\Phi_m(Low) - \delta), \\ \Phi'_m(High) &= \gamma(\Phi_m(High) - \delta), \\ l'_m &= \gamma l_m, \end{aligned} \tag{2.18}$$

and the encoding can be continued from a new interval Φ'_m. Such interval rescaling can operate multiple times during the encoding process, with different values of m, δ and γ. The decoder still can reconstruct the source stream, if m, δ and γ are provided. During decoding, the interval follows the similar way of interval rescaling, and the code value cw_m needs to be rescaled as well, using

$$cw' = \gamma(\hat{cw} - \delta) \tag{2.19}$$

As an example, Figure 2.9 shows the encoding process of Example 1 with rescaling techniques.

With interval rescaling, narrow intervals are "amplified" so that exploding increase of precision can be avoided. This technique can also be implemented with integers instead of floating points, which proves to be helpful cooperating with the following technique called shifting.

(II) Shifting
The concept shifting is common in digital circuits, where it is applied for scaling within a certain data width. Here, in order to use integers as boundaries of intervals, shifting extracts the codeword step by step, meanwhile maintains the necessary information for future computation. The example process is shown as follow.

To compress the sequence in Example 1, suppose the boundaries of intervals are limited in four decimal digits. The lower bound can be obtained by $\Phi_k(Low) * 10^4$, and notice the four-digit limitation, approximations and cuttings may be necessary. The calculation of higher bound, however, requires an extra minus one to complete. This helps prevent the possible overflow of precision. A regular pattern of low value and high value is $xx00$ and $xx99$ respectively.

As the encoding progress goes, boundary values are updated, as well as the four-digit representations. Once the leftmost digits of higher and lower boundaries are the same, this digit is outputted, and the boundaries are shifted one digit to the left. In the next iteration, the interval calculation will start from the shifted boundaries. After the processing of last symbol,

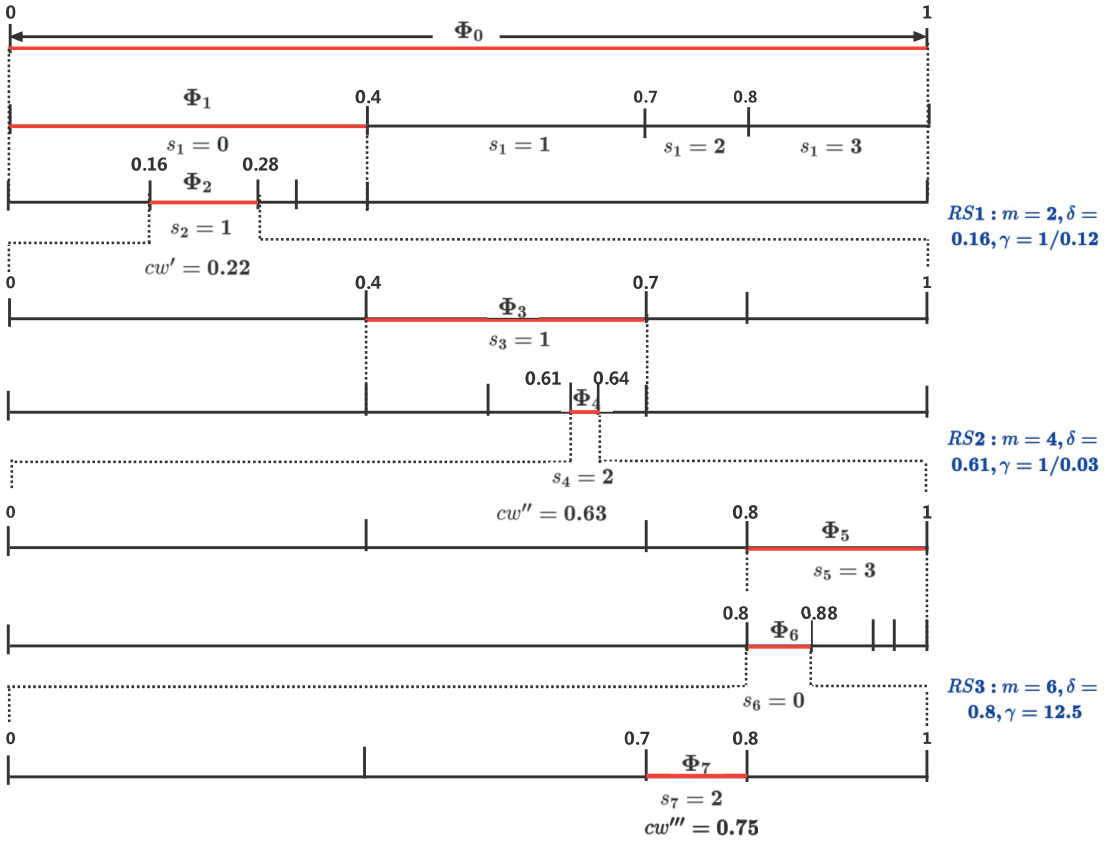


Figure 2.9: Graphical representation of the arithmetic coding process of Example 1 using numerical rescaling.

the four digits of lower boundary are outputted and the compression is finished.

The encoding process is described in Table 2.3. The final output is 2362816.

The decoding process is in the opposite direction. Note that for each iteration, code value is also limited in four digits, here denoted by *Code*. Let L and H denote four-digit representations of lower and higher boundaries respectively. Each iteration of decoding loop consists following procedure:

1 Calculate $index = [(Code - L)/(H - L + 1)]$. $[\cdot]$ is rounding operation.

2 Compare $index$ with $\boldsymbol{c}$ to select the decoded symbol m.

3 Update L and H according to

$$L_k = L_{k-1} + (H - L + 1) * c(m) \quad and \quad H_k = L_{k-1} + (H - L + 1) * c(m+1) \tag{2.20}$$

Iteration	Input Symbol	New boundaries	Integer bounds	Output digit	After shifting
0	-	L=0 H=1	0000 9999	-	0000 9999
1	0	L=0+(1-0)*0.0=0 H=0+(1-0)*0.4=0.4	0000 3999	-	0000 3999
2	1	L=0+(0.4-0)*0.4=0.16 H=0+(0.4-0)*0.7=0.28	1600 2799	-	1600 2799
3	1	L=0.16+(0.28-0.16)*0.4=0.208 H=0.16+(0.28-0.16)*0.7=0.244	2080 2439	2	0800 4399
4	2	L=0.08+(0.44-0.08)*0.7=0.332 H=0.08+(0.44-0.08)*0.8=0.368	3320 3679	3	3200 6799
5	3	L=0.32+(0.68-0.32)*0.8=0.608 H=0.32+(0.68-0.32)*1.0=0.68	6080 6799	6	0800 7999
6	0	L=0.08+(0.80-0.08)*0.0=0.08 H=0.08+(0.80-0.08)*0.4=0.368	0800 3679	-	0800 3679
7	2	L=0.08+(0.368-0.08)*0.7=0.2816 H=0.08+(0.368-0.08)*0.7=0.3104	2816 3103	-	2816 3103

Table 2.3: Arithmetic encoding process for Example 1: Shifting

4 If the leftmost digits of L and H are the same, shift *Code*, L and H one digit to the left. The padding of L and H are the same as the encoder side: add 0 to L and add 9 to H. *Code* gets the next digit from codeword sequence. If the leftmost digits are still the same, repeat step 4 again.

With the help of shifting, arithmetic coding using integers and limited precision can finally be applied for software and hardware implementations. The detailed design will be discussed in following chapters.

3 Software Implementation

In this chapter, software implementation of algorithms introduced earlier will be discussed and the simulation results will be analyzed.

In practice, single DPCM stage is not sufficient for decorrelation, while multiple-stage design is doubtful for extra cost with low profit. Meanwhile, each DPCM stage comes with the risk of overflow, leading to possible eruption of data. As for arithmetic coding, a hardware-friendly encoder design is required, and some detailed parameters are added to specify the behavior of the codec.

The platform for software simulation is MATLAB 2020a. All algorithms are developed and tested with MATLAB and some uses C as verification.

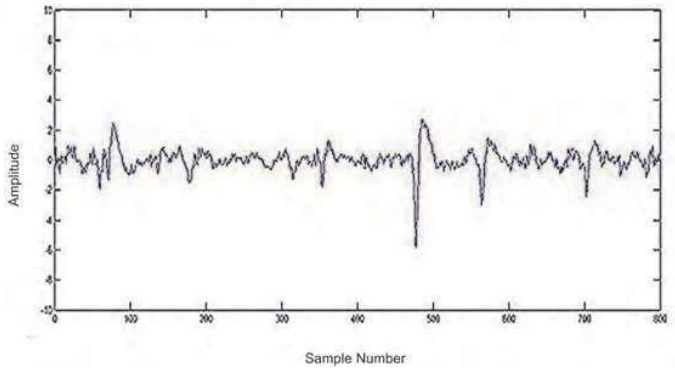
This chapter includes four sections. The first section introduces neural signals to be compressed, and the datasets used for simulation. The second section explains the realization of multi-stage DPCM and solution to possible overflow. The third section introduces the model of practical arithmetic coding. The last section proposes a complete codec design, including performance analysis.

3.1 Introduction to Datasets

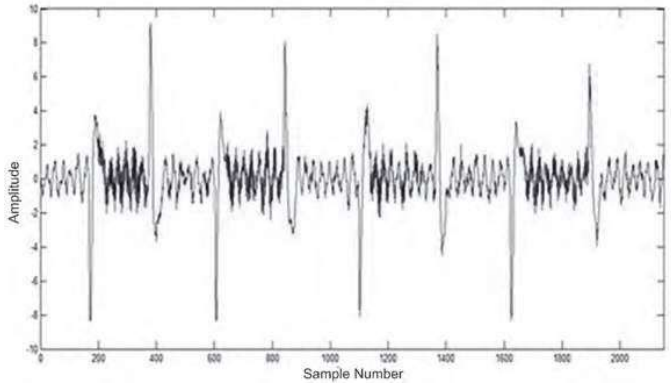
Neural signal, or specifically electroencephalogram (EEG), is the description of electrical activity of the brain. Using single electrode, the amplitude and pattern of neural signal at one position can be captured and recorded as a one-dimensional signal with respect to time. The behavior of neural signal is connected to the degree of excitation of the brain. Normally neural signals are divided into five frequency bands, and mostly higher frequencies appear when the subject is awake or in active mode. Figure 3.1 shows the different patterns of neural signals.

The similarity of these signal forms is for each signal in time domain, it consists of multiple spiking areas and resting areas. Excited brain tends to have spikes more frequently. Another property is, as a recorded analog signal, there exists strong correlation in time domain.

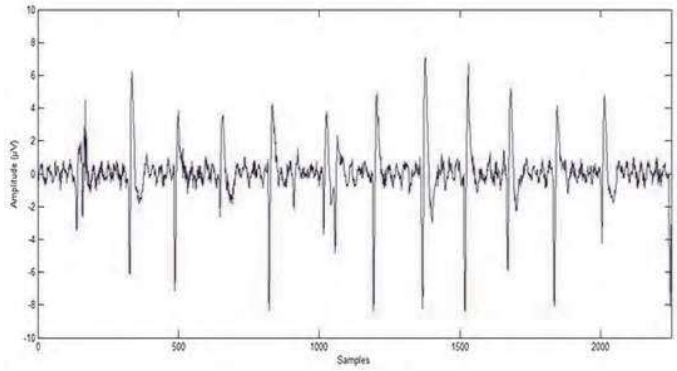
There are plenty of works about how to sample and simulate neural signals. [16] provides the method of recording extracellular signals to predict the activities inside the neuron. Authors also made the experiment datasets public. Reference [17] introduces a method to detect and classify spikes within neural signal. In this work, a method of building artificial neural signals is introduced. With the



(a) Relaxation



(b) Eyes opening and closure



(c) Eyes moving up and down

Figure 3.1: Neural signal with different activities[15]

development of machine learning, more and more algorithms and methods are brought up for neural signal classification, processing and analysis [18][19].

To help with the simulation in this work, we choose the dataset from [17] with

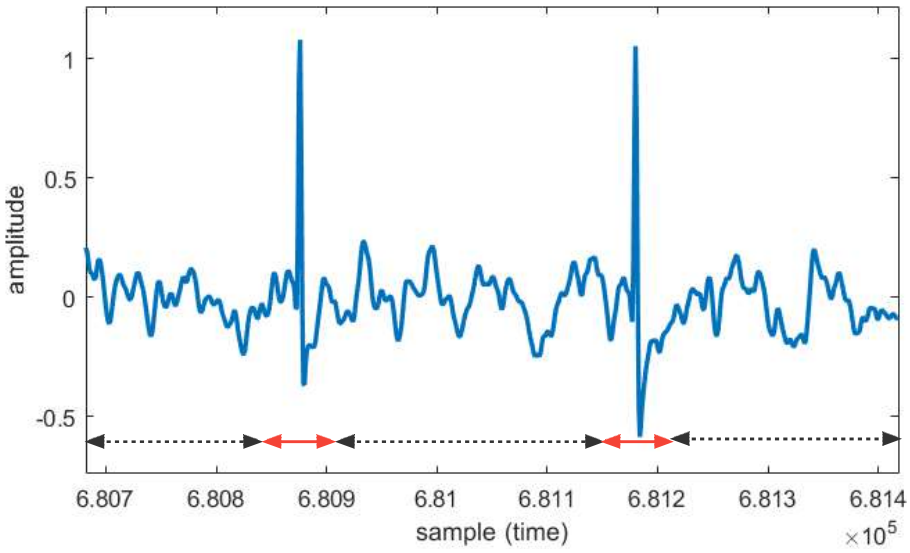


Figure 3.2: Sample data from one dataset. The noise level is 0.1. Spikes are denoted with solid line.

various configuration as the simulation source. These datasets provide neural signals with different noise levels and spike shapes. The average sampling frequency is 24 kHz. Meanwhile, we use datasets recorded from [16] as verification.

For quantization, we use 9-bit levels to sample the raw data, which means in total $2^9 = 512$ symbols.

3.2 DPCM Implementation

Section 2.1.3 has illustrated DPCM model of this work. However, in practical implementations, there is still uncertainty. First, we have to figure out which order has the best performance of decorrelation. Then the possibility of overflow when doing subtraction must be considered.

$$dpcm\_1st(i) = a(i) - a(i-1) + 256 \tag{3.1}$$

$$dpcm\_2nd(i) = a(i) - 2 \cdot a(i-1) + a(i-2) + 256 \tag{3.2}$$

$$dpcm\_3rd(i) = a(i) - 3 \cdot a(i-1) + 3 \cdot a(i-2) - a(i-3) + 256 \tag{3.3}$$

In our implementation, the symbols are indexed as unsigned integers from 0 to 511. As compensation, proper bias value (here +256) should be added to the equations (3.1) to (3.3) during calculation. This bias helps to maintain the original distribution of symbols by index. As shown in Figure 3.3, with no bias applied, the distribution is not concentrated but separated, which is not expected.

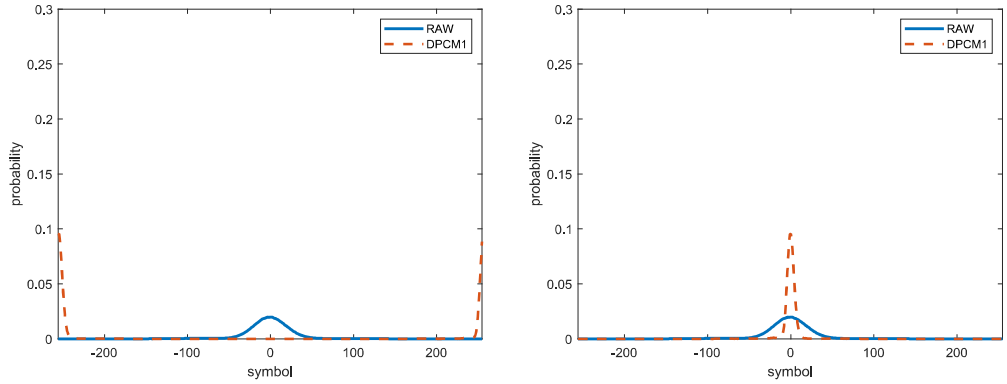


Figure 3.3: DPCM without (left) or with (right) bias. Without bias, the distribution of symbols exhibits an offset from expected "0"-centered distribution.

3.2.1 Cascade Analysis

Section 2.1.3 defines the equations for DPCM encoder, from first to third order. In practice, the order can be infinitely increased by cascading multiple 1st-order DPCM. But it is not necessary or recommended, because the profit of cascading is declining, while cost and latency is rising.

To compare their performance of decorrelation, we applied four configurations (raw data, 1st-order DPCM, 2nd-order DPCM, 3rd-order DPCM) to the same input stream from datasets with different noise levels, trying to figure out the pattern of profit-cost ratio of cascaded DPCM. The result in average is shown in Figure 3.4. The distribution of symbols is used to measure correlation: more concentrated distribution means lower correlation, which leads to lower entropy and better compression. For 1st and 2nd-order DPCM, in output streams, processed symbols are getting closer and closer to “0”, showing a more concentrated distribution. After the third stage of cascading, the distribution is not as close as before. At the same time, 3rd-order DPCM will bring extra cost on hardware and more latency. In conclusion, it is recommended to apply 2nd-order DPCM to reach the best decorrelation performance, meanwhile maintaining the simplicity of the system.

3.2.2 Overflow Analysis

Overflow occurs when two adjacent samples with large difference in their actual values. For example, suppose in DPCM input stream, $r_i = 0$ and $r_{i-1} = 500$, according to equation (3.1), the result d_i should be -244. This value is beyond the original range of index of symbols, thus it cannot be decoded. The problem

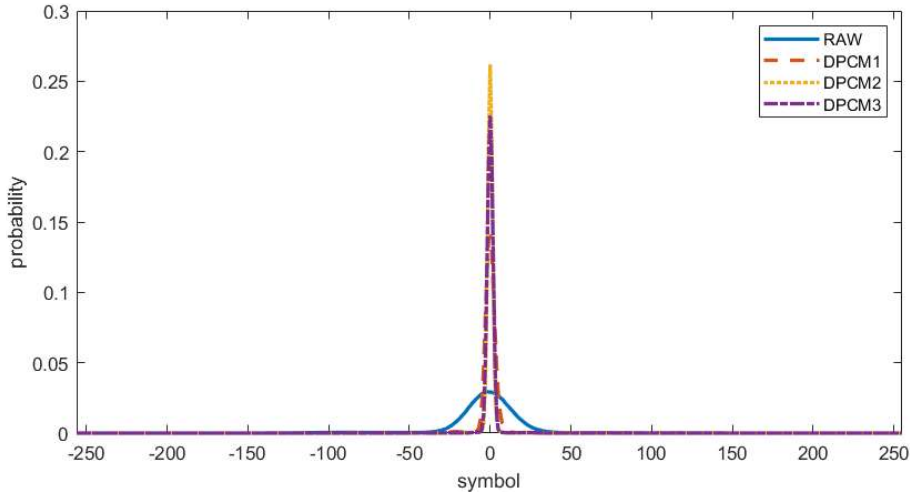


Figure 3.4: Comparison among raw data and different orders of DPCM.

is not very common when processing neural signals, but it brings trouble when cascading is applied. The symbols encoded at both ends of the stream could easily overflow.

To maintain the definition of 9-bit symbol range, rectification is applied to those overflowed results: we add a shift to the value before it is outputted. By adding or subtracting $2^9 = 512$, the range is reserved with zero information loss. The algorithm of complete 1st-order DPCM encoder is shown as psuedo code in Algorithm 1. In MATLAB, for-loop can be replaced by array arithmetic, which means each value from the sequence d_k can be calculated in parallel. The processing can be faster and the result is the same.

Algorithm 1: 1st-order DPCM encoder

Input: raw data sequence $r_k \in [0, 511], \quad k = 0, 1, ..., N-1$
Output: DPCM sequence $d_k \in [0, 511], \quad k = 0, 1, ..., N-1$

$d_0 = r_0$;
$bias = 256$;
for $1 \leq k \leq N-1$ **do**
    $d_k = r_k - r_{k-1} + bias$;
    if $d_k < 0$ **then**
        $d_k = d_k + 512$;
    else if $d_k > 511$ **then**
        $d_k = d_k - 512$;
end

The process can be cascaded to realize different order of DPCM. As proved

in Section 3.2.1, the best performance is reached when two DPCM stages are cascaded. The calibration of overflow is done at the end of each stage. Notice that the bias value (+256) can also be added only once, after the final cascading stage.

Decoding is in the opposite direction, but unlike encoding, the sequence cannot be processed in parallel. In other words, the processing of next symbol should not start until the decoding and calibration of overflow of current symbol is finished. Overflow calibration and bias is calculated in the same way. The algorithm is shown in Algorithm 2.

Algorithm 2: 1st-order DPCM decoder

Input: DPCM sequence $d_k \in [0, 511], \quad k = 0, 1, ..., N-1$

Output: reconstructed sequence $\hat{r}_k \in [0, 511], \quad k = 0, 1, ..., N-1$

```
r̂_0 = d_0;
bias = 256;
for 1 ≤ k ≤ N − 1 do
    r̂_k = d_k + d_{k−1} + bias;
    if r̂ < 0 then
        r̂ = r̂ + 512;
    else if r̂ > 511 then
        r̂ = r̂ − 512;
end
```

After applying 2nd-order DPCM to the sampled raw data, the decorrelation step shown in Figure 2.1 is finished. The stream can be sent to entropy encoders for compression.

3.3 Arithmetic Coding Implementation

To implement arithmetic coding in computers and digital circuits, apart from applying the techniques introduced in section 2.2.7, some details of the algorithm need to be modified as well. Firstly, we know that in practice, unlimited precision is not possible for storage or calculation. Fixed-length integers should replace floating point. Secondly, for the convenience of hardware implementation, the encoding result should be binary. Thirdly, mind the difference between numerical calculation in MATLAB and digital logic in circuits. For range expression, this difference could lead to precision loss.

For range expression, this difference could lead to precision loss. In this section, the practical software implementation of arithmetic coding will be explained, including the parameter settings, in-out requirements, process flow of codec, and additional functions for simulation.

3.3.1 Preparation - Training for Distribution

The input stream of arithmetic encoder is from DPCM's output. Each symbol is 9-bit long, indexed from 0 to 511. The output of the encoder is a stream of binary numbers. The result is saved in a text file, which also serves as the input source of decoder. For the decoder side, the in-out requirements are just opposite.

The preparation of distribution of the symbols is important. As explained in Chapter 2, better compression performance comes from how close the probabilities of each symbol in the input stream and the distribution of all symbols in the AC encoder are. If the distribution is calculated from the entire input file, theoretically we can reach the best compression. For the practical implementation, a static distribution is applied in both encoder and decoder. This distribution is calculated from the frequencies of symbols from all training datasets (with/without DPCM applied) at average, thus making it a non-ideal but suitable preset for all input files in simulation.

A few steps to do so:

1. Prepare the DPCM result of each training datasets.
2. Calculate the number of occurrences of each symbol for all datasets as its frequency.
3. If any symbol shows zero occurrence among all datasets, count it as one.
4. Do proportional scaling to all frequencies, until the sum reach the distribution boundary (2^{14} or 2^{30}).
5. Do approximations, maintain the frequencies integers and avoid zero for each symbol. A distribution for 512 symbols is prepared.

The same reason applied to distribution storage as well, the data width of each element from the array which stores the distribution should be finite-length integers. Here, 16-bit and 32-bit length are implemented respectively. The difference in performance will be discussed in the last section of this chapter. These values are also the width of range expressions in the codec. However, to avoid possible overlaps in the processing, not all 16 or 32 bits are employed for distribution storage. For each case, at least 2 bits must be idle so the binary codec can operate correctly. As the result, the actual data widths for distribution storage are 14 and 30 bits. This will be explained soon in Figure 3.5.

For those symbols which never showed up in the training datasets, their frequencies should be zero. Then the distribution directly calculated from them will include several null symbols which will never be encoded or decoded. This would damage the generality of the codec. To overcome this problem, we added one

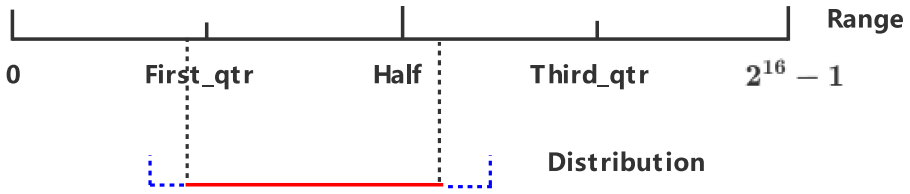


Figure 3.5: Example of possible overlapping in interval rescaling: In 16-bit module, if width of cumulative frequency is not limited below 14 bits, it is possible that one symbol (red line) occupies most of the distribution space of all symbols (blue dash), and exceeds a quarter of complete range of 2^{16} (black line), making it impossible to do interval rescaling during encoding or decoding. With the sacrifice of two bits, the output generation can be determined and unequally recognized by the decoder side. For details, see section 3.3.2.

to the frequencies of missing symbols, meanwhile maintain the total cumulative frequency (sum of all frequencies in the distribution array) within 14 or 30 bits. The compensation could be made from the higher-frequency symbols.

Normally, symbols with highest frequencies exhibit in the middle of the complete range. It becomes more evident when DPCM is applied. After processing, the distribution array can be described as cumulative frequency array:

$$\boldsymbol{c} = [0 \quad 1 \quad 2 \quad ... \quad 2^{14}-1] \quad or \quad \boldsymbol{c} = [0 \quad 1 \quad 2 \quad ... \quad 2^{30}-1] \tag{3.4}$$

The training process of multiple datasets is implemented by software. Once finished, the training results can be used as pre-defined distribution in software and hardware encoders and decoders.

3.3.2 Encoding Process

Up to now we have discussed multiple techniques for implementing arithmetic coding in practice:

- Shifting
- Interval rescaling
- Using binary outputs
- Pre-defined distributions

The following AC encoder is the combination of those techniques with some modifications.

Before the introduction of work flow, there are some terms need to be specified. **Low_value** and **High_value** define the range of current processing step. At the beginning of each iteration, a new input symbol comes in, and two operations will be performed to this range:

- Symbol assigning, meaning a shortened range is cut from the current one due to the probability of current processing symbol;
- Rescaling (normally multiple times), meaning the range is enlarged according to some rules.

The initial range is the largest range accessible. Any range generated from processing should not exceed this range, otherwise the algorithm cannot proceed. Once the data width of distribution is set, several static values can be used to separate this initial range and help with rescaling conditions. For example, in the 16-bit distribution mode, the **Low_value** and **High_value** of the initial range is 0 and $2^{16} - 1$. Three static values are defined as:

1) **First_qtr** $= 2^{(16-2)} = 2^{14}$,
2) **Half**$= 2 \times$ **First_qtr**,
3) **Third_qtr** $= 3 \times$ **First_qtr**.

As shown in Figure 3.5 these values separate the initial range in four parts, which provides boundaries for rescaling and encoding loop. The four parts are: [0,First_qtr), [First_qtr, Half), [Half, Third_qtr) and [Third_qtr, $2^{16} - 1$], respectively. Figure 3.6 shows the process of the encoding loop.

At the beginning of each iteration, symbol assigning is performed according to the probability of the current input, **Low_value** and **High_value** are relatively closer to each other, meaning that processing interval is small. Then interval rescaling and shifting cycles are performed, generating possible output bits according to the location of current processing interval in the complete range (0 to $2^{16} - 1$). Here, unlike traditional AC algorithm which narrows the interval all the time, what we called interval rescaling helps expanding the current interval step by step and generates output bits subsequently, after the interval is shrunken by a symbol. It includes 3 types of operations:

1) interval locates in lower half of complete range or **High_value** $<$ **Half**, double the bounds of current interval;
2) interval locates in upper half of complete range or **Low_value** $\geq$ **Half**, shift the current interval by **Half** to the left and double the bounds;

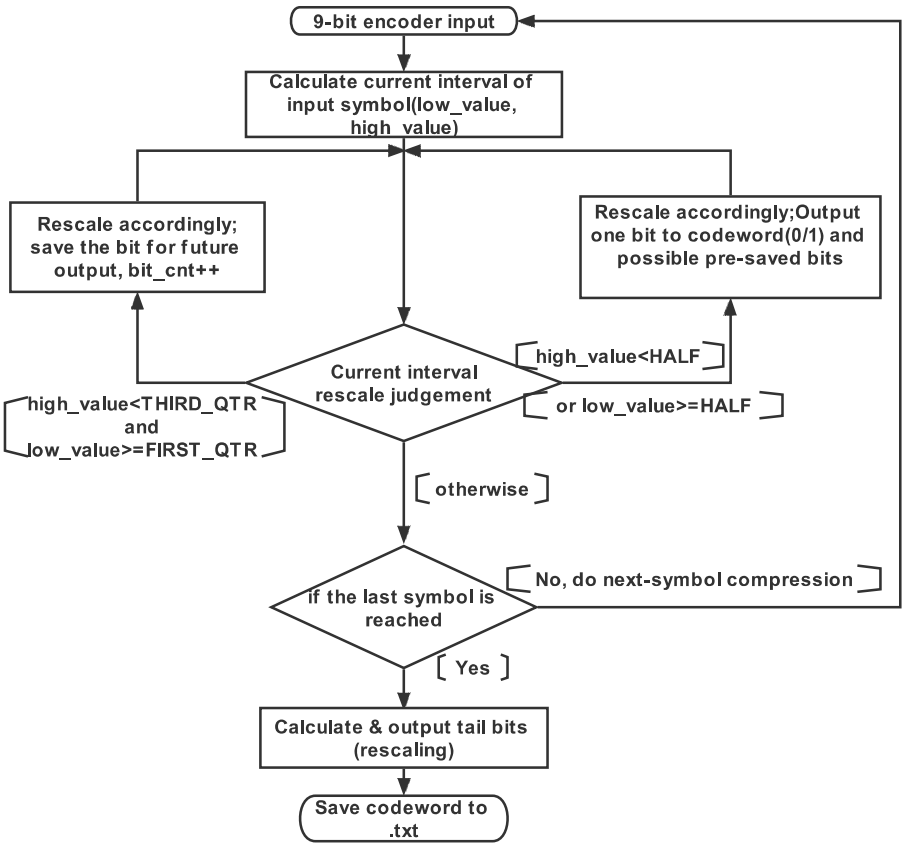


Figure 3.6: The work flow of AC encoder.

3) interval locates in the middle of complete range or **Low_value** $\geq$ **First_qtr** and **High_value** $<$ **Third_qtr**, shift the current interval by **First_qtr** to the left and double the bounds.

Note that not every cycle outputs one bit, only when the interval is completely situated within lower (bit 0) or upper half (bit 1) of the complete range. The reason is in either of these two situations one bit can decide the direction of interval rescaling. This is like someone stops at a crossing, if interval locates in lower half, turns left, and if interval locates in upper half turns right. If the interval locates in the middle, it is suggested to stay a bit longer and observe more information to decide which way to turn. If the current interval locates in the middle of the complete range, the output of this cycle will be null and this bit will be saved for future output. When output condition is satisfied in another rescaling cycle, the encoder will output all saved bits before the current output bit. In this situation multiple output bits will only be determined by the output of current cycle: if the interval is in the lower half, the multiple bits would be like

...1110; for the upper half output would be like ...0001. All pre-saved bits are the same and complementary to current output bit.

Each rescaling will enlarge the current processing interval. After several cycles, if next cycle would exceed the complete range, the rescaling will finish and the current interval will be regarded as initial range for compressing next input symbol. Until now the iteration of encoding one symbol is finished.

The encoder will repeat this process until the last input symbol is reached. Several bits will be outputted as tail to finish the loop of encoding. As the result, input symbols are completely compressed and expressed by a bit stream in a lossless way.

3.3.3 Decoding Process

The AC decoder is implemented in a reverse direction. In fact, each step in the interval rescaling loop of decoding process leads to the same result from encoding process, i.e., boundaries of the processing interval, and judgement of jumping out of the rescaling loop. Besides, shifting and binary simulation are also implemented similarly.

In decoding process, the current processing block is shifted by each input bit. This block has a fixed length of 16 or 32 bits, depending on how distribution of symbols is recorded. The value of this block represents the position of current symbol in the complete range of distribution, thus leading to decompression of the symbol. Before that, this value has to be modified to suit in the current processing interval. The decoding process is expressed in algorithm 3 below.

3.3.4 Additional Functions

The datasets include information of spikes. As discussed in section 3.1, neural signals have large redundancy in time domain, and spikes contain most of information we need. Spikes-mode in AC encoder provides compression focused on spikes and multiple surrounding samples. This mode will lift the SSR to a very high level, only losing a small percentage of information.

At the beginning of AC encoder, if distribution of symbols are not provided in advance, the encoder can be switched to semi-adaptive mode, to calculate the frequencies of all symbols and generate a temporary distribution with best possible SSR for this input. This part could also be used to generate an average distribution for an input set.

Results of encoder can be saved to a text file for further usage, i.e. as input of decoder or template for hardware results.

To verify the functions of AC encoder and decoder, results of decoder are checked if all symbols are reconstructed correctly.

Algorithm 3: AC decoder

Input: bit stream generated from encoder, distribution of symbols or $\boldsymbol{c}$
Output: reconstructed sequence of 9-bit symbols

```
value = 16-bit binary value cut from input stream;
modify value into the current processing interval;
while input stream continues do
    for m=1:num of symbols do
        if modified value ∈ [c(m),c(m + 1)) then
            decoded symbol = m;
            calculate new processing interval;
            break;
    end
    while rescale is possible do
        rescale the processing interval;
        shift the pointer at input stream to select new value;
    end
end
```

3.4 Complete Codec Design and Result Analysis

Until now all necessary blocks of encoder and decoder have been introduced. In order to analyse the performance of compression with different algorithm combinations, one can build such an all-in-one codec design that provides multiple functions, including preparing data, distribution analysis, decorrelation, compression, reconstruction, etc. Apart from theoretical analysis of each algorithm, the feasibility of complete codec needs to be guaranteed.

One more thing to note is that not only redundancy in data needs compression, also the redundancy of the system has to be minimized. This ensures the minimum spare on hardware circuit.

This section provides the design of complete codec with tunable compression conditions. In the second part a short discussion about accuracy and performance is made. In addition, different sizes of symbol width and memory width are compared.

3.4.1 The Work Flow of AC Codec

Here, a complete codec model is proposed. The work flow of this model includes 4 phases, preparation, encoding, decoding and verification. The encoder and decoder is independent, therefore it is convenient for those who only utilizes this

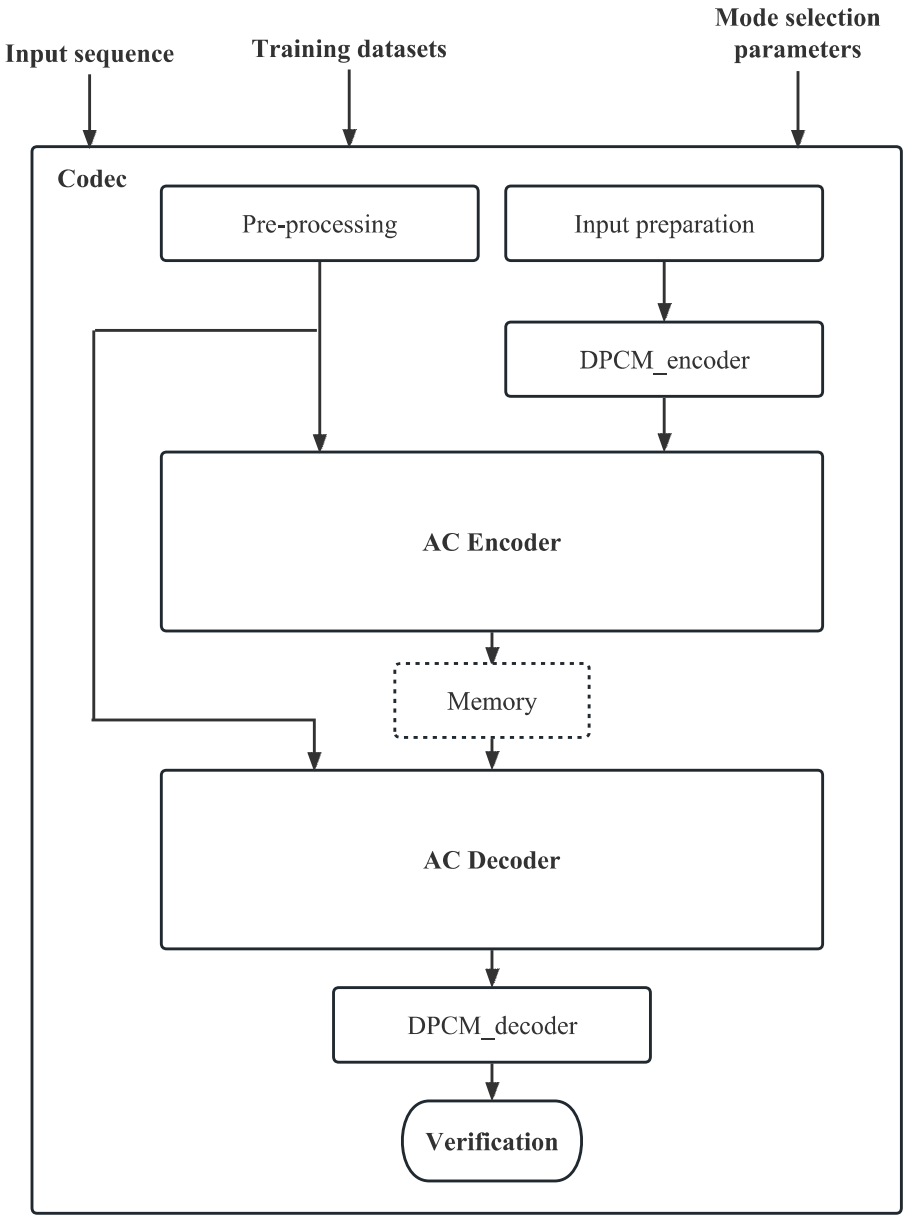


Figure 3.7: Complete software AC codec.

model as an encoder or a decoder. The codec has multiple functions, though some of them are just for analysis usage. One can choose different operating modes from mode select parameters. All parameters are listed in the following chart.

- distribution mode: use given distribution or calculated distribution by training datasets (default), or analyse current input sequence and calculate the optimal distribution from it own (semi-adaptive compression)

- input mode: compress the data with AC, the data is draw from raw sequence or outputs from different order of DPCM (1st - 3rd, 2nd order DPCM as default)
- spikes mode: compress the complete input sequence in a lossless way (default) or only the spiking parts in a near-lossless way
- preset calculation mode: use distribution calculated from training datasets (default) or given distribution from pre-saved file, which saves much time in batch processing, and provides the possibility of using existing distribution data
- batch mode: compress a single sequence(default), or compress a set of sequences in the same way
- encode save: saves the encoder output to a text file(default), or bypass to save some time in performance analysis
- bypass encoder: bypass the encoding process to decode sequences from other sources(default deactivated)

As shown in Figure 3.7, The software design can be separated into several blocks.

- Pre-processing: distribution preparation, including sampling and quantifying all training datasets, calculating distribution of each symbol from training datasets with zero to third order DPCM, and generation of executable distribution array within a certain length.
- Input preparation: including sampling and quantification process, spike extraction if near-lossless (NLL) mode is activated.
- DPCM encoder: performs DPCM with a certain order (selected by mode parameters, default 2nd order) to the input sequence after quantification, outputs decorrelated sequence
- AC encoder: performs arithmetic encoding to decorrelated sequence, outputs compressed sequence in binary mode or in a text file. Calculates the compression ratio and SSR.
- Memory: saves compression result, meanwhile works as the input of decoder blocks. To use the decoder only, one can bypass the encoder block using mode parameters, and switch the address to the file to be decompressed.
- AC decoder: performs arithmetic decoding to compressed binary sequence, outputs decorrelated symbol sequence.

- DPCM decoder: performs inverse DPCM with a certain order to the decorrelated symbol sequence, outputs quantified symbol sequence.
- Verification: compares symbol sequence from input and output sides of codec.

To illustrate how this codec works, let us assume that all default modes are activated, and input sequence is from one of the training datasets. At the beginning, the pre-processing block generates distribution according to all datasets, and input preparation block performs quantification to all samples from the input dataset. Then the input sequence is sent to DPCM encoder for decorrelation. After that, arithmetic encoder compresses the data and save it to memory. To reconstruct the data, two decoder blocks work successively, and finally the reconstructed data are compared with the raw data.

3.4.2 Result Analysis

Based on the model introduced earlier, data compression and reconstruction are simulated, and the performance of this design is evaluated by space saving ratio(SSR). In the simulation, 8 datasets with different noise levels provide training data for distribution pre-processing, and for each simulation cycle, the input sequence is some random cut from one of these datasets. The definition of noise level is their standard deviation, which varies from 0.05 to 0.35, relative to the amplitude of the spike classes. [17]

Comparison LL and NLL mode

For each input signal, both LL and NLL modes are simulated. It is obvious that the SSR of NLL mode significantly exceeds that of LL mode. Meanwhile, NLL mode is less affected by noise level changes. NLL mode has natural advantages in noise canceling because the zero-information part of neural signal are ignored.

However, NLL mode has no capability of data reconstruction. The transmission will ignore the noise parts and those cannot be restored.

Comparison DPCM orders

In Table 3.1, one can observe that under LL mode, 1st-order DPCM and 2nd-order DPCM could increase the SSR of the encoder, while 3rd-order DPCM has some opposite effects, especially when noise level is not very high. The average lifts of DPCM-1 and DPCM-2 compared to its predecessor are 113% and 17%. Compared to DPCM-2, DPCM-3 has no obvious benefits.

The result shows DPCM-2 would be the best order for operation, which is consistent with 3.2.1. This decorrelation stage has the best balance of trade-off between performance and simplicity.

Noise lvl.	LL-raw	LL-dpcm1	LL-dpcm2	LL-dpcm3	NLL-dpcm1	NLL-dpcm2
005	0,3293	0,5791	0,6022	0,5922	0,9097	0,9181
01	0,2932	0,5450	0,5895	0,5850	0,9071	0,9166
015	0,2580	0,5097	0,5733	0,5743	0,9052	0,9157
02	0,2527	0,5037	0,5702	0,5728	0,9073	0,9165
025	0,2139	0,4642	0,5520	0,5596	0,9082	0,9188
03	0,1917	0,4401	0,5406	0,5510	0,9014	0,9132
035	0,1574	0,4041	0,5235	0,5377	0,8968	0,9100
04	0,1708	0,4172	0,5304	0,5437	0,9036	0,9154

Table 3.1: Performance of 16-bit codeblock encoder: space saving ratio(SSR) with different order of DPCM, compressed by LL or NLL mode, separately.

Comparison 32-bit and 16-bit codeblocks

This part was first brought up in section 3.3.1, and the size of codeblock matches the width of distribution of each symbol. 16-bit codeblock saves much space in memory, and it will not cause a noticable drop in SSR, as shown in Table 3.1 and Table 3.2.

The simulation results show that encoder operates with 32-bit codeblock has less than one percent improvement in SSR compared to 16-bit encoder. Considering the cost of memory and doubled register expenses in hardware, 16-bit seems to be the best choice.

Noise lvl.	LL-raw	LL-dpcm1	LL-dpcm2	LL-dpcm3	NLL-dpcm1	NLL-dpcm2
005	0.3319	0.5826	0.6059	0.5960	0.9102	0.9186
01	0.2956	0.5485	0.5932	0.5888	0.9076	0.9170
015	0.2601	0.5131	0.5770	0.5781	0.9057	0.9161
02	0.2553	0.5073	0.5740	0.5767	0.9078	0.9170
025	0.2160	0.4678	0.5558	0.5634	0.9086	0.9192
03	0.1933	0.4436	0.5443	0.5549	0.9019	0.9136
035	0.1581	0.4073	0.5272	0.5416	0.8972	0.9103
04	0.1723	0.4207	0.5342	0.5476	0.9041	0.9157

Table 3.2: Performance of 32-bit codeblock encoder: SSR with different order of DPCM, compressed by LL or NLL mode, separately.

Discussion about sampling precision

In the previous discussions, DPCM-2 and 16-bit codeblock have been selected as the best platforms. To better adapt to hardware requirements, the pre-processing step is optimized to generate a distribution that fully occupies the space assigned to it. The result shows that in most cases the optimized encoder performs better.

With this hardware-friendly design, we simulated the possibility of sampling data with other previsions, e.g. 8-bit and 10-bit symbol. In Table 3.3, when noise level is low, lower sampling precision like 8-bit symbol has higher SSR. When noise level goes up, the performance of 8-bit decreases significantly, where 10-bit and 9-bit models are less affected. The reason is that decorrelation stage is not sufficient to concentrate the symbols in the case of 8-bit model, which leads to bad performance in higher noise environments. In most cases, 9-bit model performs superior to 8-bit and 10-bit model. Overall, higher precision leads to less variation by noise level changes, meanwhile the cost of computing must be considered. That is the reason the hardware design is originally based on 9-bit symbol design.

Noise lvl.	10-bit	9-bit (default)	8-bit
005	0,5978	0.6765	0.7256
01	0,5858	0.6522	0.6780
015	0,5698	0.6207	0.6137
02	0,5668	0.6133	0.5969
025	0,5484	0.5757	0.5211
03	0.5367	0.5516	0.4764
035	0.5191	0.5150	0.4133
04	0.5261	0.5277	0.4335

Table 3.3: Performance of optimized 16-bit codeblock encoder (DPCM-2): SSR under different sampling precisions.

3.5 Conclusion

During SW design, the performance of decorrelation method was firstly tested. Then, a practical AC encoding and decoding regime was brought up and simulated, reaching around 60% of SSR on average noise level. After that, with multiple functional parameters applied, a complete AC codec was built and proved to be feasible, and the best operating conditions were discussed.

4 Hardware Implementation

In this chapter, an efficient arithmetic encoder including decorrelation stages will be presented. Based on simulation results, performance of this design will be quantified.

The software simulation has revealed a proper setup for the hardware design. Specifically, the following presets are optimal for our encoder circuit:

- 9-bit symbol input: good anti-noise effect, relatively low consumption
- 16-bit codeblock: low demand on registers and SRAM storage
- Binary output: efficient coupling
- 2nd-order DPCM decorrelation: best decorrelation performance
- Distribution calculated from average of datasets: best average compression (see section 3.3.1)

The platform of hardware design is project- and design-flow management system ICPRO (v1.7).

This chapter includes three sections. The first section explains how hardware circuit is implemented. The second part shows the result of hardware synthesis. The third section briefly concludes the work.

4.1 Hardware Implementation

Considering the versatility requirements of the module, The design is shown in Figure 4.1. The top module Encoder communicates with SRAM who stores the information of distribution, and generates the compressed code. Detailed information of each block and port are listed below.

4.1.1 Input ports

- clk, rst_n: Global clock and reset signals.
- data_i (bus): 9-bit input data, with an independent valid signal.

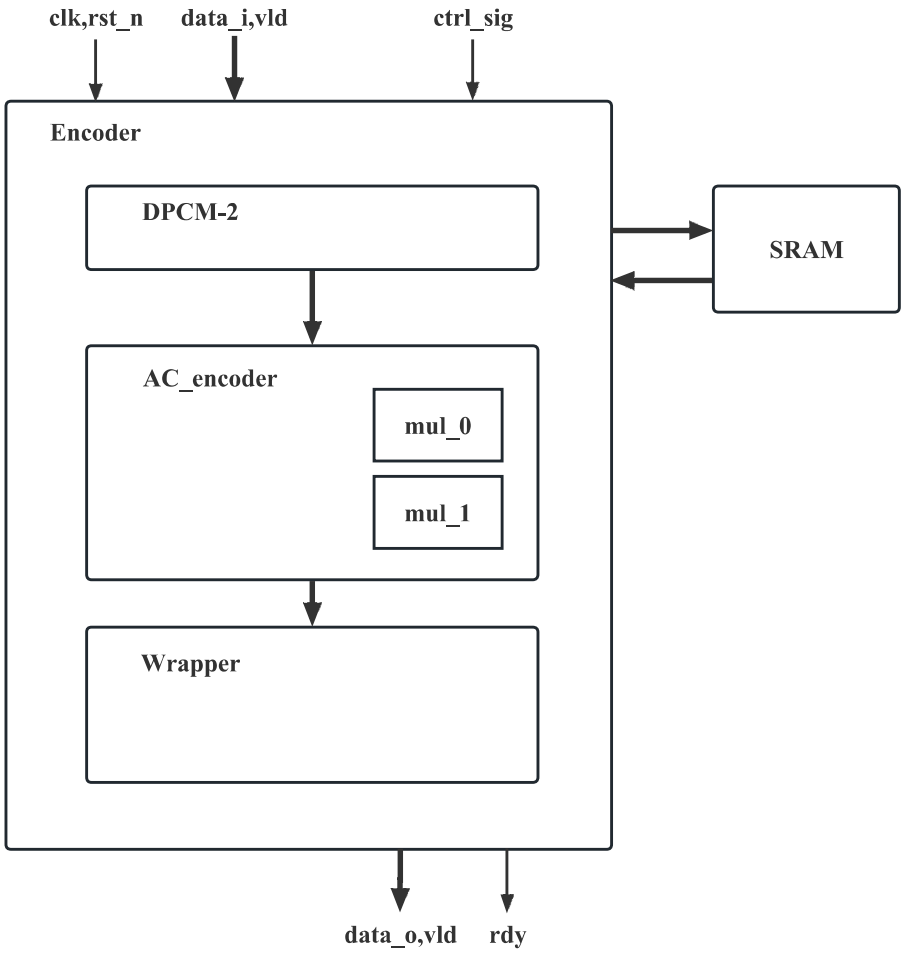


Figure 4.1: AC HW encoder module

- ctrl_sig: Encoding start and termination signals.
- Distribution information read from SRAM.

4.1.2 Output ports

- data_o: Compressed binary data, encapsulated in byte format, with an independent valid signal.
- rdy: Ready signal of the complete module.
- Requests for distribution to SRAM, including address.

4.1.3 Module composition

- DPCM-2: 2nd-order DPCM processor.
- AC_encoder: Arithmetic coding processor.
- Wrapper: A normalizer for outputs.
- SRAM: Memory block, 32-bit storage, here simulated by accessing distribution file. The memory required for AC is 1k bytes.

4.1.4 Details of Implementation

There are a few details about DPCM-2 and AC_encoder. First, for DPCM-2, cascading is replaced by one-step computation, meaning that 1st-order DPCM is saved and therefore area consumption is reduced. To realize such job a set of registers are used to store the input symbols from the past two clock cycles.

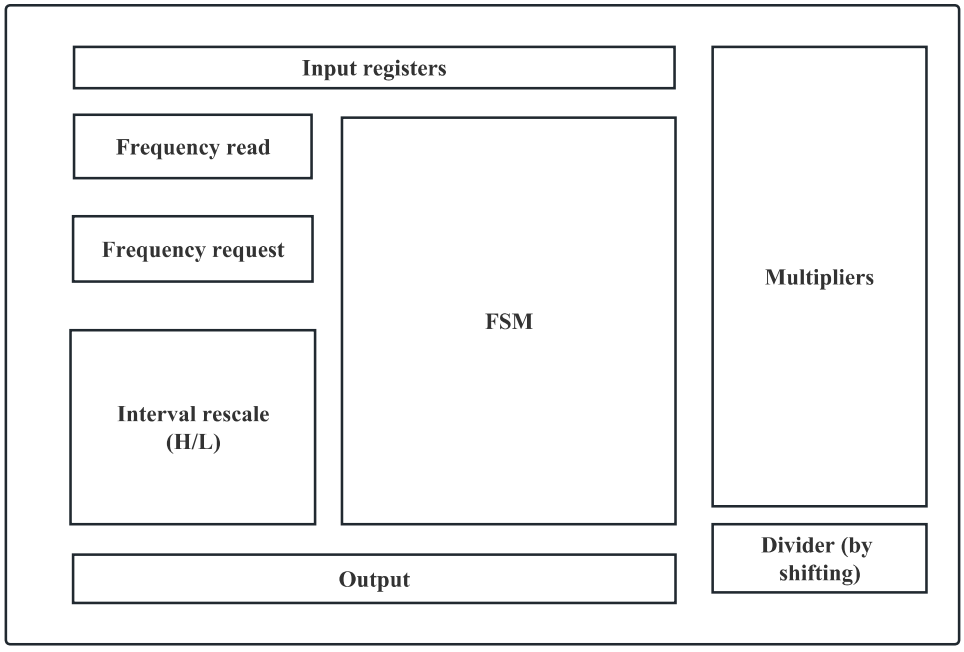


Figure 4.2: Block design in AC_encoder

For AC-encoder, Figure 4.2 exhibits main execution blocks within the module. Input registers collect data as well as control signals from the top module, meanwhile store them in corresponding registers for further processing. To communicate with SRAM, two blocks realized by sequential logic are called by FSM every time a symbol is prepared: "Request" generates corresponding address of

the signal and send out requests for frequency information stored in distribution in SRAM, while "Read" collects those frequency information from SRAM. One thing to notice is that in HW implementation, distribution in SRAM is stored in 32-bit format. Request and read operation may be repeated while current symbol requires cumulative frequencies (high and low) from two neighboring addresses.

When computing the current interval, according to SW design in 3.3.2, an interval is defined by two values, thus two multipliers are used in parallel to save a few clock cycles. Interval rescaling works in double threads as well, controlled by FSM, the interval rescaling is repeated several times during a symbol. Meanwhile, output bits with control signal are generated and sent out from registers to output ports.

One benefit of binary operation is that division can be replaced by simple shifting. In fact, that shifting has already been covered when encoder outputs a bit. That would save a lot of area for hardware design.

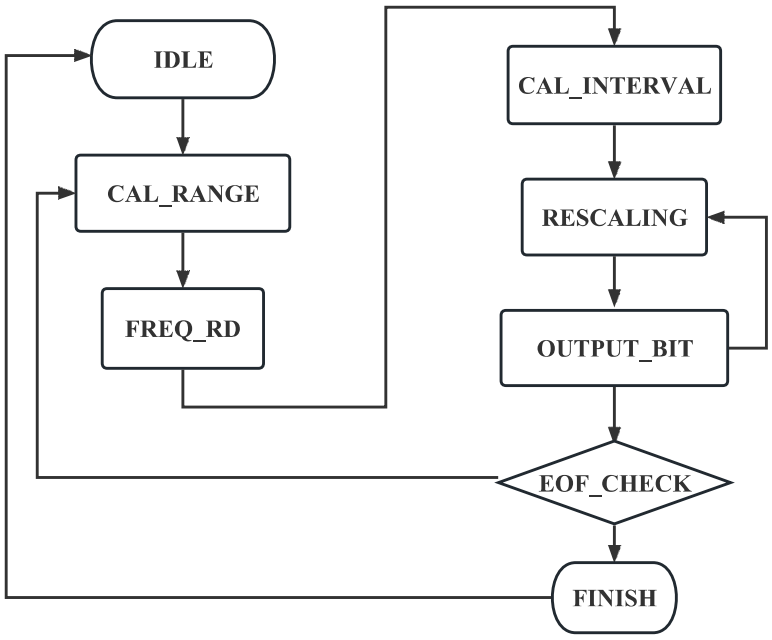


Figure 4.3: Work flow of FSM

Referring to Figure 3.6, the workflow of AC_encoder can be controlled by a finite state machine (FSM). Figure 4.3 shows the simplified graph of FSM. When a symbol comes the encoder firstly calculate the size of current interval, then read the cumulative frequencies of the symbol from SRAM. After that, multipliers are called to calculate the interval narrowed by Equation 2.20. Then interval rescaling and shifting starts, output bits are generated or stored temporally. The next mode would check if the next symbol arrives or the compression is over, and switch to

range calculation or waiting mode accordingly.

So far the design is complete. To verify the function of hardware encoder, a testcase is applied during simulation process. Details of test results will be discussed soon.

4.2 Result Analysis

4.2.1 Functionality verification

The verification can be done by two ways.

One is called parallel verification. Assume that symbol distribution are fixed and inputs are the same, the verification is done by comparing the compression result from software encoder and hardware encoder respectively. Except from mapping bits one by one, there is a simpler way for encoders based on arithmetic coding. For developers, it is possible to directly match the last operating interval in AC with the interval from software. If those intervals are the same, the hardware result should be identical to the software results.

Another is sequential verification. One can also save the hardware result in a text file and use the software decoder to reconstruct the data. Activate bypassing encoder in software codec and include the hardware compressed file, the verification is done automatically. The verification schemes are shown in Figure 4.4.

In this project, it is proved that the hardware encoder generates identical result as software encoder does, and the result can be reconstructed correctly.

4.2.2 Performance analysis

Under 28nm node, the speed and area consumption of the encoder module can be estimated.

Depending on the data to be compressed, the speed of compression varies. Using data from [17] as inputs, it takes 34 clock cycles in average to compress a symbol, which can be transferred into 38.25 CPB. However in the most rare case, compressing a symbol may cost around 120 cycles.

As shown in Figure 4.5, AC encoder occupies most of the area. This is caused by two adders called by AC, each takes 40% of total area. Owing to the elimination of dividers, the area is well controlled. And if several more clock cycles are tolerable, current parallel multipliers can be replaced by a single on, which reduces area consumption further. Note that SRAM is not included in this statistic, it is shared among all compression channels.

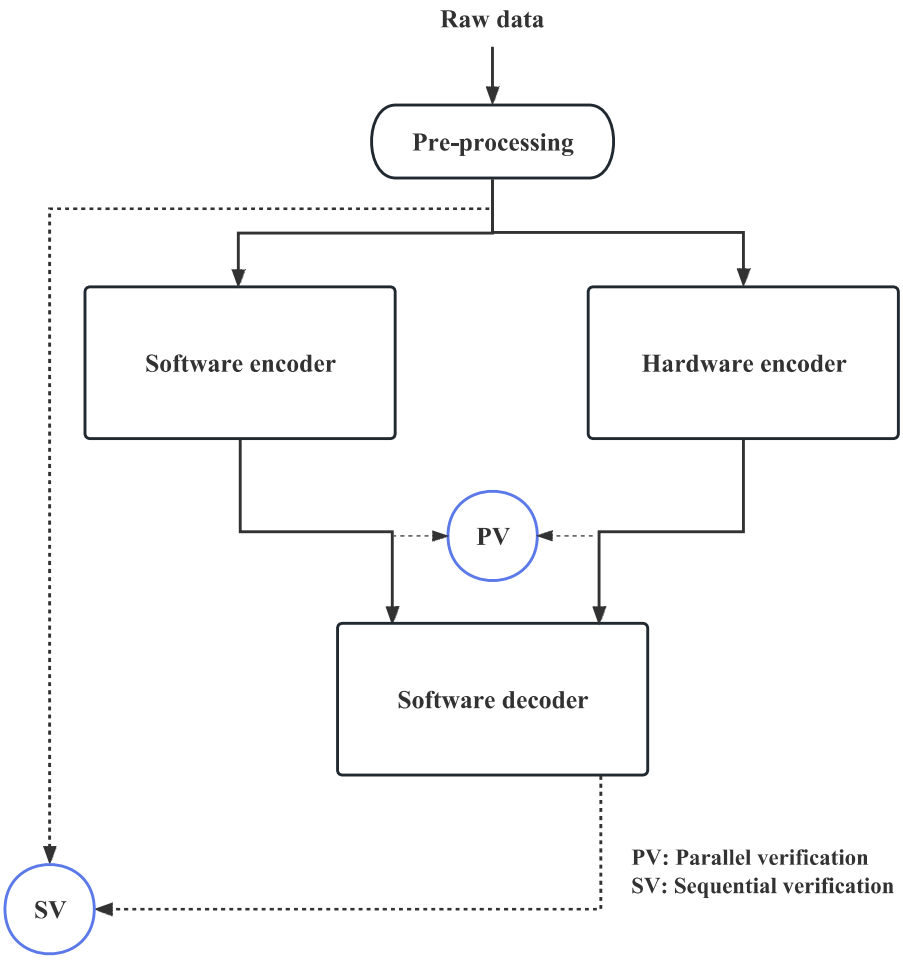


Figure 4.4: Verification schemes

The next step is to observe the area consumption under different frequencies. The area percentage hardly varies with operating frequency. In this work, the lowest frequency possible is

$$\frac{symbol\ period}{number\ of\ cycles\ per\ symbol} = 2.4MHz,$$

since the real-time neural signal is transferred every 50μs, and at least 120 clock cycles are needed to ensure compressing a symbol. In Figure 4.6, one can see that under ultra low frequencies, the area consumption varies very slow. When the frequency goes high, the area grows fast until reaching the limit at 400MHz. Over 400MHz, time violation occurs and circuit is not reliable. The difference between the largest and the lowest area consumption is 14%.

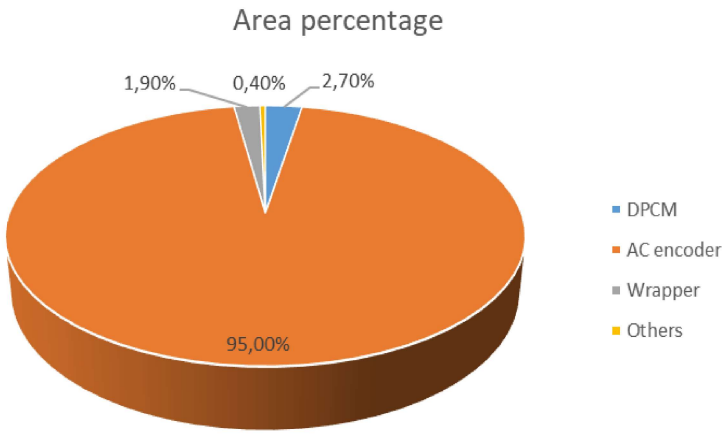


Figure 4.5: The area occupation with the module excluding the SRAM

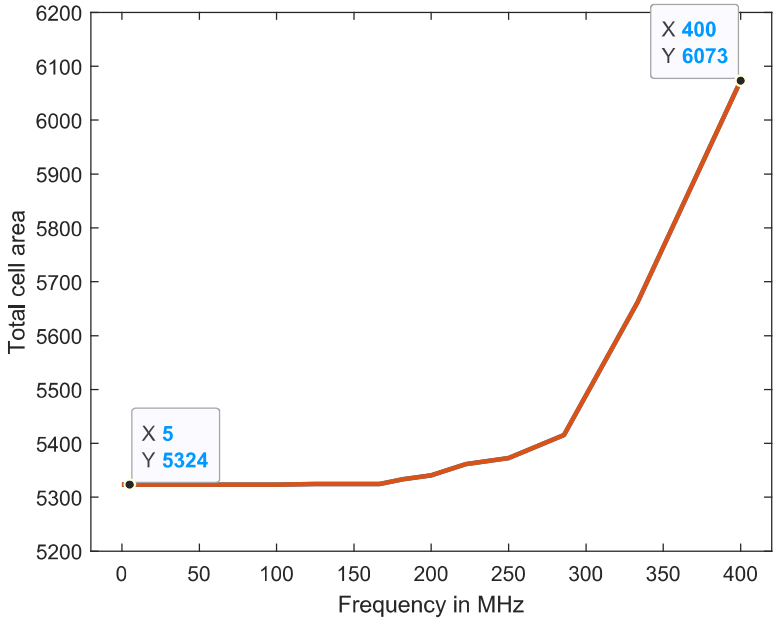


Figure 4.6: Cell area as a function of operating frequency

4.3 Application and Discussion

Application - SYNCH

The AC compression engine brought up in this chapter can be applied in many scenarios. For example, it can help building a multi-channel compression solution in SYNCH, a neural SoC by HPSN. As shown in Figure 4.7, NAC is the multi-channel arithmetic coding engine, including 8 channels and an independent SRAM

for distribution storage.

In general, the processing elements (PE) generates distribution information from data collected by analog-to-digital converters (ADCs). This process can also be done by external processors, like the software engine in Chapter 3, and transmitted back to SRAM for storage by serial peripheral interface (SPI). After that, normal arithmetic coding (NAC) developed in this thesis can start to compress the data according to the distribution. The compressed data will be sent to SPI or PE for further processing.

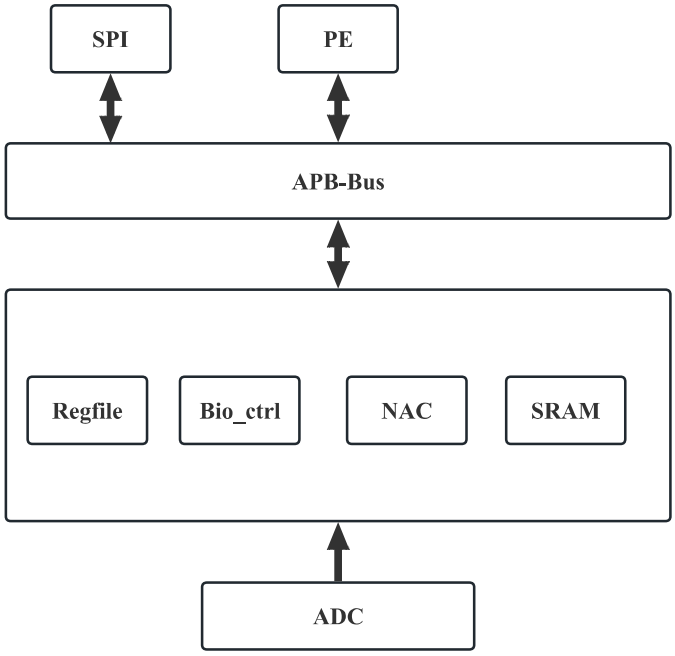


Figure 4.7: NAC implementation in SYNCH.

Discussion

In this project, a scheme for compressing neural signals is proposed and verified. It provides fair SSR under different noise levels, meanwhile maintains rather small area consumption.

Comparing with existing compression engines based on Golomb coding [20], this work can reach higher compression under low noise levels (from 0.05 to 0.15) without concerns about m value.

Comparing with adaptive arithmetic coding (AAC) solution, SSR of this work is slightly lower than that from adaptive way, while the normal AC consumes less area than AAC encoder.

Therefore, It is efficient to apply AC to compress neural data whose distribution is rarely changed. The tradeoff between area and SSR can be well balanced.

5 Summary

This thesis aims to propose a practical method of compressing neural signals in a lossless way, and implement the arithmetic coding algorithm by software and hardware.

Firstly, as a necessary pre-processing step for entropy coding algorithms, DPCM decorrelation method is introduced. After that, the theory of AC is explained in its original format, and the advantages of AC are discussed. According to analysis, AC is an optimal, flexible, synchronized solution for neural data compression.

A multi-functional AC SW codec is implemented on MATLAB platform. A process of distribution analysis and training is firstly implemented. SW implementation provides the comparisons of various DPCM stages and several tunable AC parameters. With the help of DPCM and binary-output AC encoder, the performance of such design can reach over 60% of average SSR. From the performance analysis, a set of optimal conditions and parameters are chosen and will be applied for HW design. Also a decoder is designed to verify the results from SW and HW encoders. Some additional features like NLL compression and semi-adaptive compression are discussed. Compared with PAGC and AAC implementations, the design in this thesis shows an SSR of 64.98% in lower noise levels, which is higher than PAGC's performance (61.84%). As an average for all noise levels, SSR continues descending to 59.16%, which is slightly lower than AAC's result (62.5%). Considering the area and power consumption of adaption module, AC in this thesis can operate in a more resource-friendly way.

An efficient arithmetic encoder HW design is presented. With the help of distribution information determined from SW, the HW implementation is able to compress quantified neural data and output the result in byte format. The typical area composition of the module is $5323\mu m^2$ under 5MHz. An average operating speed is presented and analysed. The module design has been verified and applied to SYNCH SoC.

Bibliography

[1] David Salomon, Giovanni Motta, and Giovanni Motta. *Handbook of data compression*. Vol. 2. Springer, 2010.

[2] Uthayakumar Jayasankar, Vengattaraman Thirumal, and Dhavachelvan Ponnurangam. "A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications". In: *Journal of King Saud University - Computer and Information Sciences* 33.2 (2021), pp. 119–140. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2018.05.006.

[3] Khalid Sayood. *Lossless compression handbook*. Elsevier, 2002.

[4] R. Rice and J. Plaunt. "Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data". In: *IEEE Transactions on Communication Technology* 19.6 (1971), pp. 889–897. DOI: 10.1109/TCOM.1971.1090789.

[5] Simeng Li et al. "A low-power 4K point FFT processor for CMMB OFDM receiver system". In: (Oct. 2009). DOI: 10.1109/ASICON.2009.5351632.

[6] Alexey Pak. *Image Data Compression De-correlation techniques*. https://ies.anthropomatik.kit.edu/ies/download/lehre/idc/IDC-05- Decorrelation.pdf. 2018.

[7] Evgeniĭ Borisovich Dynkin. *Theory of Markov processes*. Courier Corporation, 2012.

[8] M. Mohana Soundarya and S. Jayachitra. "An Efficient Code Compression Technique For Ecg Signal Application Using Xilinx Software". In: *International Journal of Scientific & Technology Research* 8 (2019), pp. 1958–1965.

[9] P Elias and N Abramson. *Information theory and coding*. 1963.

[10] Jorma J Rissanen. "Generalized Kraft inequality and arithmetic coding". In: *IBM Journal of research and development* 20.3 (1976), pp. 198–203.

[11] Richard Clark Pasco. "Source coding algorithms for fast data compression". PhD thesis. Stanford University CA, 1976.

[12] Athanasios Papoulis and S Unnikrishna Pillai. *Probability, random variables and stochastic processes*. 2002.

[13] Amir Said. "Introduction to Arithmetic Coding–Theory and Practice". In: *arXiv preprint arXiv:2302.00819* (2023).

[14] Claude E Shannon. "A mathematical theory of communication". In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[15] Wahidah Mansor, Mohd Shaifulrizal Abd Rani, and Nurfatehah Wahy. *Integrating neural signal and embedded system for controlling small motor*. InTech, 2011.

[16] Darrell A Henze et al. "Intracellular features predicted by extracellular recordings in the hippocampus in vivo". In: *Journal of neurophysiology* 84.1 (2000), pp. 390–400.

[17] R Quian Quiroga, Zoltan Nadasdy, and Yoram Ben-Shaul. "Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering". In: *Neural computation* 16.8 (2004), pp. 1661–1687.

[18] Gaowei Xu et al. "A deep transfer convolutional neural network framework for EEG signal classification". In: *IEEE Access* 7 (2019), pp. 112767–112776.

[19] Perattur Nagabushanam, S Thomas George, and Subramanyam Radha. "EEG signal classification using LSTM and improved neural network algorithms". In: *Soft Computing* 24 (2020), pp. 9981–10003.

[20] Qier Ma et al. "Ultra-low Power and Area-efficient Hardware Accelerator for Adaptive Neural Signal Compression". In: *2021 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. IEEE. 2021, pp. 1–4.