Tampere University

Matias Järvenpää

POLAR ENCODING AND PERFECT SHUFFLES IN 5G COMMUNICATION

Bachelor's Thesis
Faculty of Information Technology and Communication Sciences
Examiner: Joonas Multanen
December 2023

ABSTRACT

Matias Järvenpää: Polar Encoding and Perfect Shuffles in 5G Communication
Bachelor's Thesis
Tampere University
Bachelor's Programme in Computing and Electrical Engineering
December 2023

Digital communication is an essential part of today's society. Almost everyone has a mobile phone that is constantly wirelessly transmitting data using cellular network. To enable fast communication rates, efficient error control coding is needed. This thesis focuses on polar code, which is a linear block error-correcting code used for example in channel coding of the fifth-generation network technology, 5G.

The aim of this thesis is to present a parallel implementation of polar encoding according to the 5G New Radio (NR) specifications. To enable this implementation to be used on a Digital Signal Processor (DSP), certain instruction set extensions are needed. These custom instructions enable efficient computation, but they do come at a hardware cost, which is why the number of different custom instructions has been kept low. Implementations are proposed for frozen bit insertion, polar transform, and sub-block interleaving. All these procedures are parts of the 5G polar encoding chain.

The properties of a special permutation of bits called a perfect shuffle were found suitable for this implementation, and the permutation is used extensively in this thesis. Using a perfect shuffle to implement a butterfly-pattern based algorithm is not a new idea. In previous work it was shown that a perfect shuffle can be used for example in the implementation of Fast Fourier Transform (FFT). However, previous work using the perfect shuffle to implement polar encoding similarly as was done here was not found.

The perfect shuffle permutation used here can be compared to shuffling a pack of playing cards by dividing the deck into two halves and then releasing the two halves so that the cards in the two halves get interleaved. The properties of this perfect shuffle allow all the difficult bit-level addressing patterns found in the polar encoding process to be simplified to a regular structure easily implementable on a DSP. It is found that a single perfect shuffle permutation will allow the implementation of polar encoding for all different code block sizes defined in the 5G specification.

Keywords: polar codes, perfect shuffle, butterfly pattern, parallel polar encoding

TIIVISTELMÄ

Matias Järvenpää: Polaarienkoodaus ja täydelliset sekoitukset 5G-kommunikaatiossa
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma
Joulukuu 2023

Digitaalinen viestintä on olennainen osa nyky-yhteiskuntaa. Melkein kaikilla on älypuhelin, joka välittää jatkuvasti dataa langattomasti matkapuhelinverkon kautta. Nopean tiedonsiirron mahdollistamiseksi tarvitaan tehokasta virheenhallintakoodausta. Tämä opinnäytetyö keskittyy polaarikoodiin, joka on lineaarinen virheenkorjauskoodi, jota käytetään esimerkiksi viidennen sukupolven verkkoteknologian, 5G:n, kanavakoodauksessa.

Tämän työn tavoite on esittää 5G New Radio (NR) -spesifikaation mukainen rinnakkaistettu toteutus polaarienkoodauksesta. Jotta toteutusta voi käyttää digitaalisella signaaliprosessorilla (DSP), tarvitaan tiettyjä käskykantalaajennuksia. Nämä käskyt mahdollistavat tehokkaan laskennan, mutta niiden toteutus vaatii ylimääräistä laitteistoa. Tämän takia kustomoitujen käskyjen määrä on pyritty pitämään alhaisena. Työssä esitetään toteutukset jäädytettyjen bittien (frozen bits) lisäämiselle, polaarimuunnokselle (polar transform) ja alilohkosekoitukselle (sub-block interleaving). Kaikki nämä toimenpiteet ovat osa 5G-polaarienkoodausprosessia.

Täydelliseksi sekoitukseksi (perfect shuffle) kutsutun bittipermutaation ominaisuudet osoittautuivat sopivaksi tähän työhön, ja kyseistä permutaatiota käytetäänkin työssä jatkuvasti. Täydellisen sekoituksen käyttö perhosgraafiin (butterfly pattern) perustuvan algoritmin toteutuksessa ei ole uusi idea. Aiemmat työt näyttävät, että täydellistä sekoitusta voi käyttää mm. nopean Fouriermuunnoksen (Fast Fourier Transform, FFT) toteutuksessa. Ei kuitenkaan löytynyt aiempaa työtä, jossa täydellistä sekoitusta olisi käytetty polaarienkoodauksen toteutuksessa tämän työn tapaisesti.

Käytettyä täydellistä sekoitusta voi permutaationa verrata sellaiseen korttipakan sekoitukseen, jossa pakka jaetaan keskeltä kahtia ja sitten vapautetaan kortit niin, että pakan puolikkaat lomittuvat toisiinsa. Tämän permutaation ominaisuudet mahdollistavat polaarienkoodauksen vaatiman bittitason prosessoinnin yksinkertaistamisen säännölliseksi rakenteeksi, joka on helposti toteutettavissa prosessorilla. Huomataan, että yksi täydellinen sekoitus riittää polaarienkoodauksen toteutukseen kaikilla 5G-spesifikaation mukaisilla koodilohkon kokovaihtoehdoilla.

Avainsanat: polaarikoodit, täydellinen sekoitus, täydellinen shuffle, rinnakkaistettu polaarienkoodaus

CONTENTS

1. INTRODUCTION

In our modern society almost everyone carries in their pocket a device that for most of human history would probably have been classified as magic. Even though mobile phones have now existed for a while, their functionality is still quite magical: How is it possible that a device so small can retrieve and process information from the other side of the world in a fraction of a second?

This communication is enabled by mobile network. The network consists of base stations communicating wirelessly with mobile devices [1, p. 7]. There are different technologies for this wireless communication, but this thesis focuses on the fifth-generation mobile network technology, 5G. In particular, the focus is on error control coding built into the 5G specification. Error control coding is used to tackle the problems caused by noisy communication channels.

As an analogy, suppose that you are in a classroom during a mathematics class. Your friend has asked you what your answer to the last exercise was. However, the classroom is a bit noisy, and therefore your friend might not perfectly hear what you are saying. You don't want to talk too loudly, since you don't want to distract other students. How could you make sure that you can get your message through to your friend?

This same problem is present in 5G communication. How can a device make sure that bits it has received were not changed by random noise that a noisy communications channel might introduce? This is where error control coding is used. Error control coding can be divided into 2 parts, error detection coding and error correction coding. As the term suggests, error detection means detecting the errors that may be introduced into digital data from a noisy channel and error correction includes also the correction of those errors. [2, p. 3]

This thesis focuses on the implementation of Polar coding, which is one of the error control codes used in the 5G New Radio (NR) technical specification 38.212 "Multiplexing and channel coding" [3]. Polar codes are based on the so-called channel polarization effect introduced in 2009 by Erdal Arıkan [4]. Even though the encoding complexity of Polar codes is $O(N \log N)$, implementing the encoder efficiently on a Digital Signal Processor (DSP) is not straightforward.

This thesis proposes a method for optimizing polar encoding using a minimal amount

of instruction set extensions. The thesis is limited to polar codes as they are defined in the 5G NR specification [3]. The implementation does not cover all the procedures of the specification, but mainly the focus is on the implementation of “polar transform”, meaning the multiplication of data bits by a generator matrix, which causes the channel polarization. The hardware needed for that procedure was found to be suitable for other operations described in the specification, and therefore their implementations are considered as well. In chapter 2 the basic concepts related to channel polarization and polar encoding are explained in more detail. In chapter 3 the implementation procedures are introduced, and in chapter 4 the hardware requirements for the implementation are described.

2. BASIC CONCEPTS

There is a physical limit called channel capacity which defines how much information can be transmitted over some communication channel, and this limit sets boundaries for error control codes. Section 2.1 gives a short introduction to the information theory behind channel capacity, and section 2.2 shows how polar codes try to achieve this limit in practice.

An actual real-world application of the theory is a 5G device, for example a mobile phone transmitting data to a cell tower (uplink) or a cell tower transmitting data to a phone (downlink). The requirements of the 5G specification are shortly introduced in section 2.3.

To implement those requirements a permutation called a “perfect shuffle” is used frequently in this thesis. Perfect shuffle is a type of permutation that has been shown to be usable for example in the implementation of Fast Fourier Transform (FFT) [5]. This thesis shows how perfect shuffles can also be used to implement polar encoding. In section 2.4 perfect shuffles are defined, and their basic properties are described.

2.1 Channel Capacity

Error control techniques work by adding redundancy to the information that is to be transmitted [6, p. 3]. A simple example would be transmitting every bit twice. If you want to send “101”, you transmit “110011”. The added redundancy means that not all bit sequences are valid. The bit sequence “110011” is now a valid message, whereas for example “100011” is not, and the receiver can therefore detect an error. In the classroom analogy, this redundancy principle is also present. The languages we use have built-in redundancy. Not all combinations of sounds form a valid word [6, p. 2].

Adding redundancy has its downsides. Redundancy is a waste of capacity on communication channels. [6, p. 2] This can be seen in the repetition code: To communicate 3 bits we need to transmit 6 bits. To minimize wasting of capacity, we need to be able to minimize the amount of redundancy in our error correcting code. One might even ask: What is the minimal amount of redundancy that must be added to a message to reliably transmit it via a channel with a known noise level?

The complete information theory behind this question is out of the scope of this thesis,

but the main outline of the answer is presented.

Suppose two random variables X and Y represent the input and output of some channel. The *entropy* of X is defined:

$$H(X) = E[-\log_2 P(X)] \tag{2.1}$$

where E is the expected value of a random variable. This quantity tells something about the uncertainty of X before its measurement, or how difficult it would be to guess the value of X before measurement. [2, Ch. 1.12] If $H(X)$ is large, then guessing would be quite unsure (For example an unbiased coin toss). If $H(X)$ is small, then guessing is easier (for example a biased coin, where the other side is much more likely). It can also be thought of to tell how much information is gained when X is measured [2, Ch. 1.12].

Mutual information between X and Y is a property that can be defined as

$$I(X;Y) = H(X) - H(X \mid Y) \tag{2.2}$$

It is the uncertainty there still is about input X after measuring output Y subtracted from the original uncertainty of X. In other words, it quantifies how much information a measurement of output Y gives about the input X. [2, Ch. 1.12] As an example, assume $X \in \{0, 1\}$ and the probabilities of 0 and 1 are both $\frac{1}{2}$. The input is then transmitting information at rate 1 bit per symbol. ($H(X) = 1$) Then, assume a perfect channel: If output Y measures a value y, then that value will always tell what the value of the transmitted input was. Then $H(X \mid Y) = 0$ and $I(X;Y) = H(X) = 1 \frac{\text{bit}}{\text{symbol}}$, since there is no uncertainty about X after the measurement. Then assume a totally noisy channel: The value of Y doesn't tell anything about the value of X, meaning that even after measuring the output Y, the probabilities of both possible values of X are still $\frac{1}{2}$. Then $I(X;Y) = 0$, meaning that no information can be passed through the channel. (Similar example: [7])

The channel capacity C is then defined as

$$C = \max_{P_X(x)} I(X;Y) \tag{2.3}$$

The channel capacity is the maximum mutual information between X and Y over all input distributions. Take a Binary Erasure Channel (BEC) as an example. With a BEC it is assumed that a transmitted bit is not received correctly ("erased") with probability δ and that it is received correctly with probability $1 - \delta$. The capacity of a such channel is $C = 1 - \delta$, which is achieved when the input distribution is chosen so that the probabilities of 0 and 1 are $\frac{1}{2}$ for both. [2, Ch. 1.12]

The rate of a binary error correcting code is $R = \frac{K}{N}$, where K is the number of information bits, and N is the number of code output bits. Then $N - K$ is the amount of added

redundancy. The *channel coding theorem* then states: If $R < C$, then there exists an error correction code such that the probability of error in the decoding is less than ϵ for any arbitrarily small ϵ. Also, if $R > C$, then arbitrarily small error rate cannot be achieved. Therefore, channel capacity is the limit rate at which information can be transmitted reliably (without errors) through the channel. [2, Ch. 1.12] The channel coding theorem was introduced in 1948 in a paper “A mathematical theory of communication” [7] by Claude Shannon, who has been said to be the founder of information theory [2, Ch. 1.3].

This is the answer to the question of how much redundancy must be added to achieve reliable communication. However, the proof for the channel coding theorem is written so that the existence of such a code is proven without actually defining an example. Naturally developing a practically implementable and low-complexity error control code that would achieve the channel capacity has then been a clear goal. [4] There are multiple examples of codes that can approach the limit (for example: Low Density Parity Check code (LDPC) and Turbo-code) [2], but this thesis focuses on the implementation of Polar codes. In his paper Erdal Arıkan, the inventor of Polar codes, shows that with channel polarization we can practically construct a code that achieves channel capacity in certain channels. The encoding complexity is $O(N \log N)$, where N is the code block size. He also introduces a successive cancellation based decoder with the same $O(N \log N)$ complexity. [4]

2.2 Channel Polarization

The working principle behind polar coding is a phenomenon called “channel polarization” [4], which is a way to add redundancy to the transmitted bits so that possible errors caused by a noisy channel can be corrected. The term polarization refers to the fact that if a vector $\mathbf{u}$ of length $N = 2^n$ is used as input to polar encoding, the channel capacities of different positions in that vector are polarized, and they either approach 1 (reliable channels) or 0 (unreliable channels) [4]. The K bits carrying information are inserted to the most reliable positions, and the rest are frozen to a known value, for example 0 in case of 5G [3].

At the heart of polar coding is a matrix

$$F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{2.4}$$

The matrix is called basic polarization kernel. [8] The polarization of channels can be demonstrated with a Binary Erasure Channel (BEC), with erasure probability δ.

Assume that we have two bits u_0 and u_1 that we want to transmit. If they are sent directly, without any error control code, u_0 is erased with probability δ, and similarly u_1 is erased with probability δ. However now they are encoded using the basic polarization kernel. The

codeword $\mathbf{d} = [d_0, d_1]$ that is to be transmitted is obtained using a vector-matrix product:

$$\mathbf{d} = \mathbf{u}F\,, \tag{2.5}$$

where $\mathbf{u} = [u_0, u_1]$. [8] Using the polarization kernel defined in equation 2.4:

$$\begin{cases} d_0 = u_0 \oplus u_1 \\ d_1 = u_1 \end{cases} \tag{2.6}$$

Notice that here addition has been changed to binary XOR. This is because the operation is computed in the binary field GF(2) [4], and binary XOR is equivalent to mod-2 sum.

From equation 2.6 we can see that the receiver can now compute $\mathbf{u}$ from $\mathbf{d}$ using

$$\begin{cases} u_0 = d_0 \oplus d_1 \\ u_1 = d_1 \end{cases} \tag{2.7}$$

From this it can be seen that the receiver can correctly decode u_0 only if it has received both bits d_0 and d_1 correctly. The probability that this doesn't happen, i.e. the probability of erasure for u_0 is [8]

$$\delta_0 = 1 - (1-\delta)^2 = (2-\delta)\delta > \delta \tag{2.8}$$

Now assume that the receiver has correctly decoded u_0. Then it can decode u_1 if it has correctly received either d_0 or d_1. If d_1 has been correctly received, then

$$u_1 = d_1 \tag{2.9}$$

If d_0 has been correctly received, then

$$u_0 \oplus d_0 = d_0 \oplus d_1 \oplus d_0 = d_1 = u_1 \tag{2.10}$$

The probability that neither of the bits d_0 or d_1 is received correctly is [8]

$$\delta_1 = \delta^2 < \delta \tag{2.11}$$

From inequalities 2.8 and 2.11 it can now be seen that due to the encoding, the erasure probabilities of bits u_0 and u_1 are now polarized, meaning that $\delta_0 > \delta$ and $\delta_1 < \delta$, where δ is the original probability of erasure. [8] There was one large assumption though. When computing δ_1 it was assumed that the receiver knew the value of u_0. It is not obvious how that assumption can be justified, since u_0 is a bit that was sent out by the transmitter, and thus the receiver might not actually know its value.

So-called “frozen bits” [4] can give a partial intuitive explanation to this. In the “encoding” that was done, it can be noticed that no redundancy was actually added to the bits. We had 2 bits (u_0, u_1) that we wanted to send, and we then transmitted 2 bits (d_0, d_1). With polar codes the redundancy is added using frozen bits [4]. What that would mean in this case is that we would freeze $u_0 = 0$, for example. The receiver should know this choice. Then of course there would only be one information-carrying bit, u_1. Then despite what the receiver receives, it would know that $u_0 = 0$. Now u_0 is a frozen bit, and u_1 is a more reliable “channel” for transmitting information.

A generator matrix for larger N can be constructed from the basic polarization kernel:

$$G_N = B_N F^{\otimes n} \tag{2.12}$$

This is how the matrix is defined in the original paper that introduced polar codes [4]. B_N is a bit-reversal permutation matrix. However, the same paper states that the permutation matrix B_N can be omitted, if that is taken into account in the decoder. Then

$$G_N = F^{\otimes n} \tag{2.13}$$

This matches with the definition found in the 5G NR specification [3]. Notation $F^{\otimes n}$ means Kronecker power and is defined here as

$$\begin{aligned} F^{\otimes n} &= F \otimes F^{\otimes (n-1)} \\ F^{\otimes 0} &= \begin{bmatrix} 1 \end{bmatrix} \end{aligned} \tag{2.14}$$

for all $n \geq 0$ [4].

Now the encoding of input $\mathbf{u}$ to output $\mathbf{d}$ can be computed with

$$\mathbf{d} = \mathbf{u} G_N \tag{2.15}$$

[4] Then it can be shown that this causes channel polarization. This means that when $N \to \infty$, the channels tend towards totally unreliable channels or perfectly reliable channels [4]. Here the different positions of the vector $\mathbf{u}$ are thought of as separate communication channels. This means that bits inserted at certain positions (the reliable channels) can very likely be decoded correctly in the decoder, when the rest of the positions are frozen to a known value. The indices of the reliable positions actually depend on the properties of the channel [4], but the 5G specification defines a reliability sequence that should be used in 5G communication [3]. The ratio of information carrying bits K and the code block length N should be chosen so that the channel capacity is not exceeded [4].

Computing polar transform simply by directly computing the vector-matrix product 2.15 would be of complexity $O(N^2)$. This can be reduced to $O(N \log(N))$ by exploiting the

properties of the Kronecker-product definition of G_N:

$$G_N = F^{\otimes n} = F \otimes F^{\otimes(n-1)} = \begin{bmatrix} F^{\otimes(n-1)} & 0 \\ F^{\otimes(n-1)} & F^{\otimes(n-1)} \end{bmatrix} = \begin{bmatrix} G_{N/2} & 0 \\ G_{N/2} & G_{N/2} \end{bmatrix} \tag{2.16}$$

Then

$$\mathbf{u}G_N = \begin{bmatrix} \mathbf{u}_0 & \mathbf{u}_1 \end{bmatrix} \begin{bmatrix} G_{N/2} & 0 \\ G_{N/2} & G_{N/2} \end{bmatrix} = \begin{bmatrix} (\mathbf{u}_0 \oplus \mathbf{u}_1)G_{N/2} & \mathbf{u}_1 G_{N/2} \end{bmatrix} \tag{2.17}$$

Here $\mathbf{u}$ is divided into two halves $\mathbf{u} = \begin{bmatrix} \mathbf{u}_0 & \mathbf{u}_1 \end{bmatrix}$.

This then allows a butterfly style XOR network to be used for calculating the product with complexity $O(N \log N)$ [4]. An example where $N = 8$ is illustrated in figure 2.1.

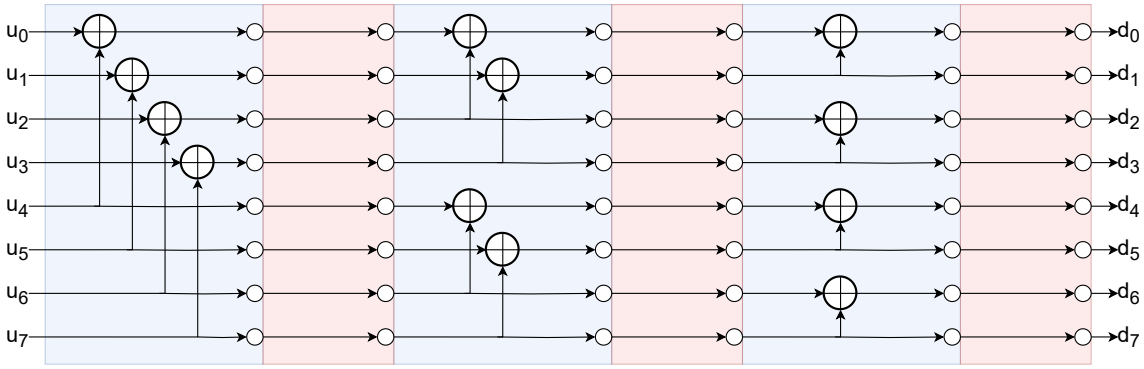

Figure 2.1. *An example of polar transform using a butterfly network*

2.3 Polar Codes in 5G NR

The vector-matrix product specified in equation 2.15 is only one part of the polar encoding process defined in the 5G NR specification. The complete process is shown in figure 2.2, along with some of the parameters defining the process. The input to the system is vector $\mathbf{a}$ of length A containing information bits, and the output is vector $\mathbf{g}$ of length G. These notations match with the ones used in the specification [3].

The first step is segmenting the bits to sections, if needed. If $I_{seg} = 1$, the bits are segmented into 2 sections. Otherwise no segmentation is done. [3]

Then a total of L Cyclic Redundancy Check -bits (CRC) are added to the info bits. Here L is the length of the generator polynomial. Therefore $K = A' + L$. [3]

If $I_{IL} = 1$, the bits are then permutated/interleaved using an interleave pattern defined in the specification. In practice $I_{IL} = 1$ in downlink operation. The idea behind this interleaving is to allow early termination in the decoding process, using the CRC bits [9].

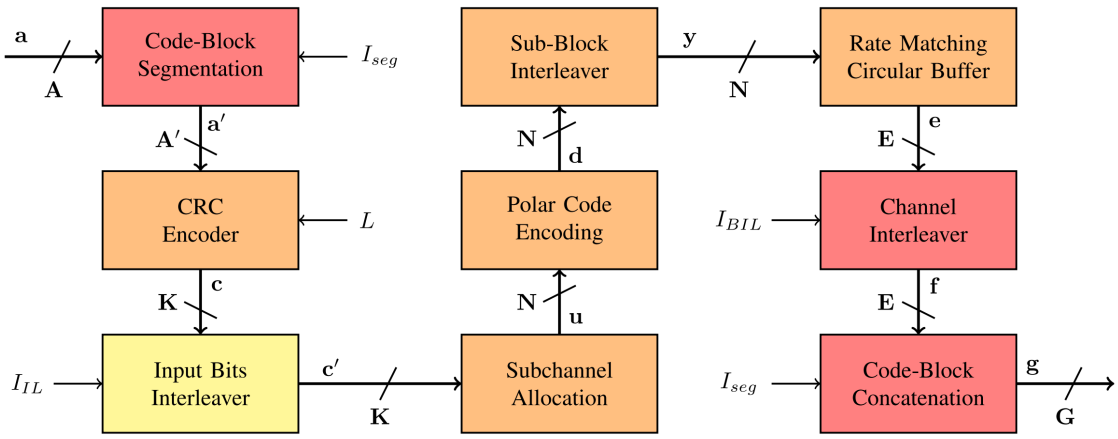

Figure 2.2. *5G polar encoding chain. Yellow indicates implementation in downlink, red uplink and orange both. [8]*

In subchannel allocation, the K info bits are inserted to the indices that correspond the most reliable channels in polar encoding, and the rest of the total of N bits are frozen to zero. This is followed by the actual polar transformation using the generator matrix of equation 2.13. The encoded bits are then again interleaved, using a sub-block interleaver consisting of 32 blocks. [3]

Then the length N of the bit sequence is adjusted to desired length E in the rate matching process. To get E bits from N, the bit sequence is either repeated, shortened or punctured [9].

If $I_{BIL} = 1$, the bits are then interleaved before the final step, where the previously segmented code blocks are concatenated back together. [3]

To limit the scope of this thesis, only implementations to subchannel allocation (also called frozen bit insertion), polar transform, and sub-block interleaving are introduced here, and only those procedures are described with more detail. Also, small code block lengths are not considered here, since they are handled as a special case in the specification [3].

The objective of frozen bit insertion is to insert K information bits to the most reliable positions of N bits, where $N \in \{32, 64, 128, 256, 512, 1024\}$. The specification table 5.3.1.2-1 gives 1024 indices of the channels sorted in ascending order of reliability. Reliability sequences for smaller code lengths are obtained as a subset of this sequence. Frozen bit insertion is then described using the table and some pseudocode. What is basically done is a bitmask, where initially all bits are zero. Then K ones are inserted to the indices specified by the reliability sequence, starting from the end of the table. (Since the table is in ascending order of reliability). Then the info bits are put to the positions indicated by ones in the bitmask, retaining the order of the bits. For small block lengths some parity bits are also added. Rate-matching specified in section 5.4.1 of the specification complicates this slightly, since depending on the rate matching output sequence length, some of the reliability sequence indices are discarded, but the general outline of

the procedure is still as described here. [3]

The polar transform is defined in the specification as the vector-matrix product of equation 2.15, where $G_N = G_2^{\otimes n}$ and $G_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ is the polarization kernel that was in Arıkan's paper [4] notated as F. [3]

The sub-block interleaver is defined in the specification section 5.4.1.1. The N output bits from polar transform are divided into 32 blocks (block size 1,2,4,8,16,32), and those blocks are then reordered according to an interleaving pattern given by specification table 5.4.1.1-1. [3]

2.4 Perfect Shuffles

Consider a set of $m = b_1 \cdot b_0$ elements. The set can be divided to blocks in 2 ways: Either b_1 blocks of size b_0 or b_0 blocks of size b_1. Therefore, the i^{th} element can be described in 2 ways: $i = i_1 b_0 + i_0 = i_1^* b_1 + i_0^*$, where $i_0, i_1^* \in \{0, 1, \ldots, b_0 - 1\}$ and $i_1, i_0^* \in \{0, 1, \ldots, b_1 - 1\}$. [10] An example could be:

$$\begin{aligned} m &= 3 \cdot 5 = 15 \\ i &= 10 = 2 \cdot 5 + 0 = 3 \cdot 3 + 1 \end{aligned} \tag{2.18}$$

This means that "10" is either the first element of the third (size 5) block, or "10" is the second element of the fourth (size 3) block. Notice here in the equations indexing starts from zero.

Now define a (b_1, b_0) perfect shuffle to be a permutation that moves element $i = i_1 b_0 + i_0$ to position $j = i_0 b_1 + i_1$ [10]. Basically, a perfect shuffle combines the 2 ways that the size $b_1 b_0$ set can be divided into blocks. In a (b_1, b_0)-shuffle the set is divided into b_1 blocks (of size b_0). Then the first element of each input block goes to the first output block (of size b_1), retaining their original order. Then the second element of each input block goes to the second output block and so on. An example of a $(3, 5)$-shuffle is illustrated in figure 2.3.

Permutations can be represented using a permutation matrix. Define S_{b_1,b_0} to be a permutation matrix so that $S_{b_1,b_0}\mathbf{v}$ performs a (b_1, b_0)-shuffle on the column vector $\mathbf{v}$.

Straight from the definition of the perfect shuffle it is fairly trivial to see that a (b_1, b_0)-shuffle is the inverse of a (b_0, b_1)-shuffle [10], meaning that $S_{b_1,b_0} = S_{b_0,b_1}^{-1}$. Also, another usable property of perfect shuffles is how they can be factorized [10]:

$$S_{b_2,b_1 b_0} = S_{b_2 b_0,b_1} S_{b_2 b_1,b_0} = S_{b_2 b_1,b_0} S_{b_2 b_0,b_1} \tag{2.19}$$

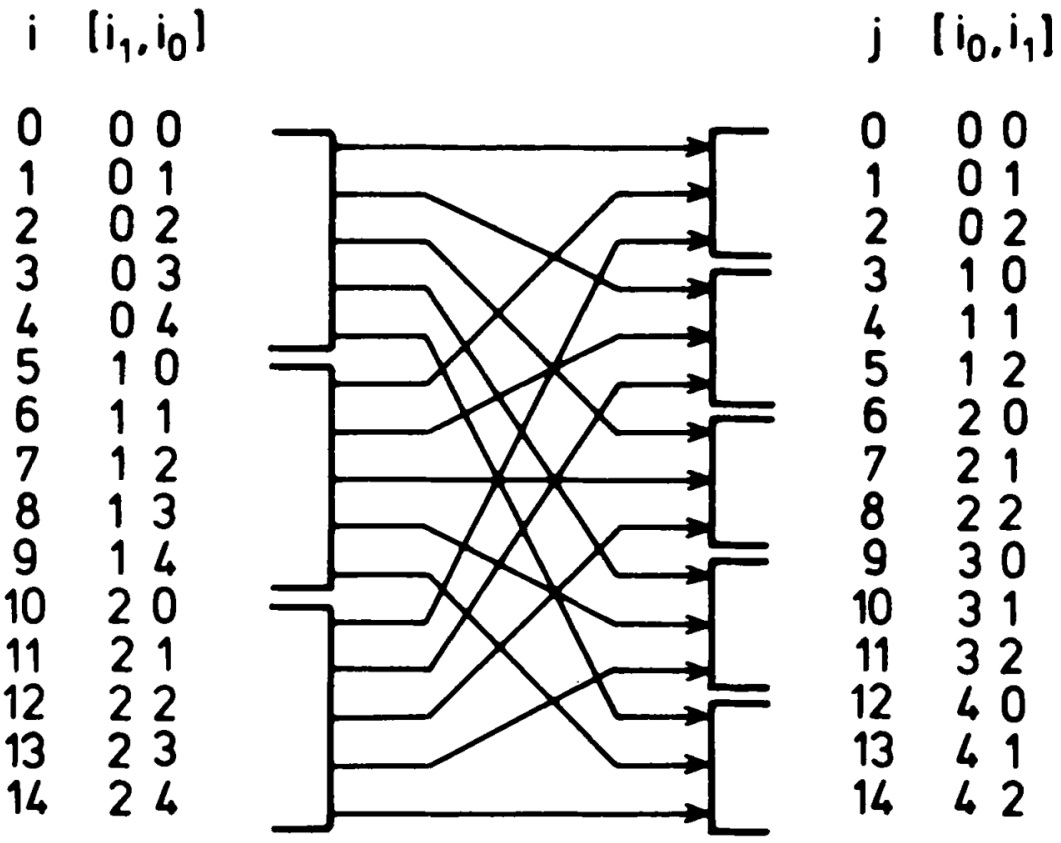

Figure 2.3. *A* $(3, 5)$ *perfect shuffle. Adapted from [10].*

The connections between Kronecker products and perfect shuffles are also very important here:

$$I_{b_2} \otimes M \otimes I_{b_0} = S_{b_0,b_2r_1} \cdot (I_{b_2b_0} \otimes M) \cdot S_{b_2c_1,b_0} \tag{2.20}$$

and

$$M \otimes I_{b_0} = S_{b_0,r_1}(I_{b_0} \otimes M)S_{c_1,b_0} \tag{2.21}$$

Here M is an $r_1 \times c_1$ -matrix [10].

3. PARALLELIZED IMPLEMENTATION

This implementation aims to minimize the number of custom instructions needed for the implementation of the operations defined by the 5G specification while keeping the operations fully parallel, meaning that bits are handled as bits. This allows the utilization of the entire “Single Instruction, Multiple Data” (SIMD) -width of the used processor. It is shown that despite the troubles of bit-level addressing, frozen bit insertion and polar transform can be implemented using only a single perfect shuffle permutation and normal bitwise logic operations. The same shuffle in addition to a special permutation defined in the specification is also sufficient for the implementation of sub-block interleaving.

3.1 Frozen Bit Insertion

As explained in section 2.3, frozen bit insertion comes down to inserting K info bits to their correct position according to an N-bit bitmask. More generally this operation can be called a “bit scatter” or “parallel deposit” operation [11]. An example of this is illustrated in figure 3.1.

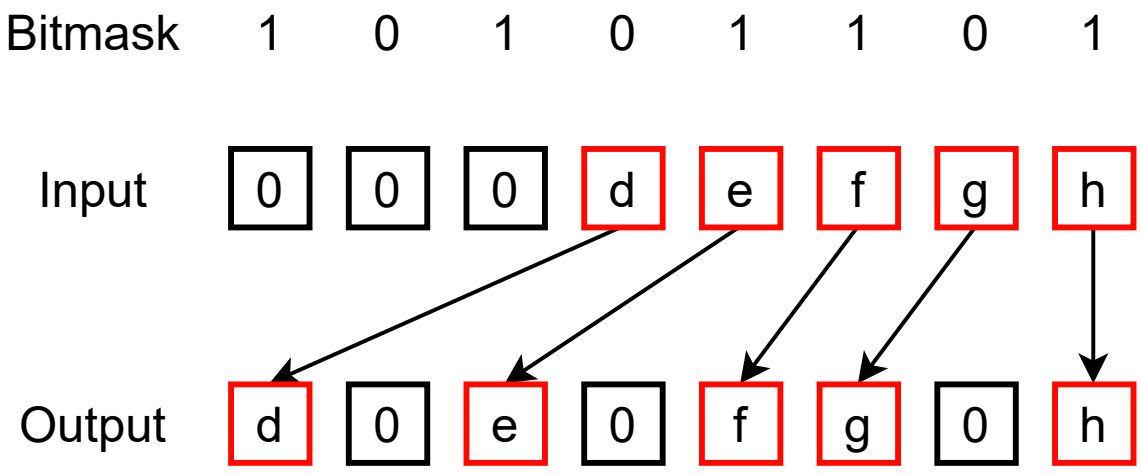

Figure 3.1. *An example of parallel deposit operation, $N = 8$, $K = 5$. Adapted from [11]*

Efficient implementation of this bit-level manipulation is not straightforward. However, it has been shown that a parallel deposit operation can be implemented in $\log_2(N) = n$ stages using a butterfly pattern [11]. In each of the stages a bit is either passed through or it is swapped with another bit, whose index is determined by the butterfly stage. The

butterfly pattern and how it can be utilized to implement the same operation as in figure 3.1 is illustrated in figure 3.2. Here control bit "1" indicates swap, which is opposite of what is used in [11].

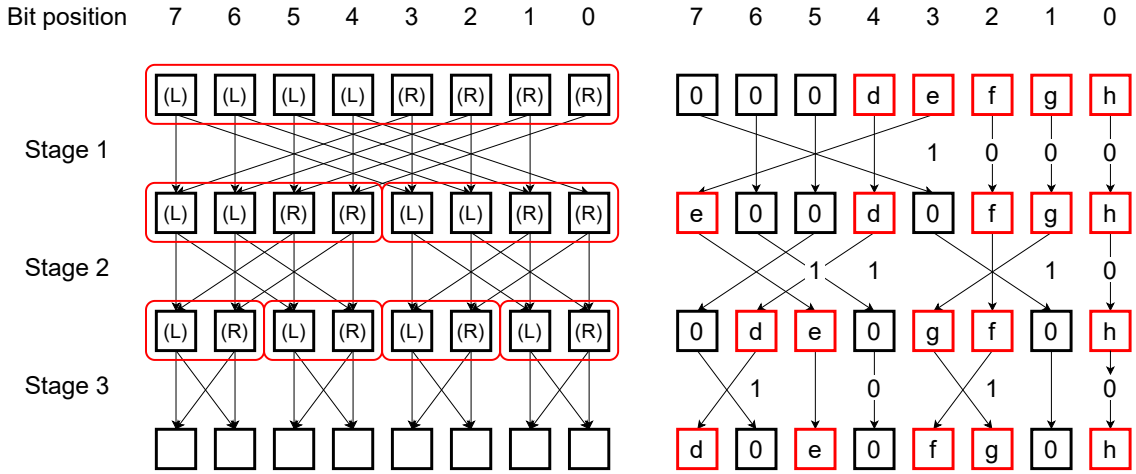

Figure 3.2. *An example of parallel deposit operation implemented using a butterfly pattern,* $N = 8$*,* $K = 5$*. Adapted from [11]*

Whether a bit is passed through or swapped is determined by precomputed control bits [11]. These control bits are independent of the actual data bits, meaning that if N and K stay constant, the control bits stay constant. However, computing the control bits in software is rather heavy compared to the actual butterfly operation. According to [11], the algorithm used to compute the control bits might take approximately 1200 cycles on an Intel Pentium-D processor ($N = 64$), while the actual butterfly pattern can be computed in 1 cycle, if the hardware is built as in their paper. The control bits can also be computed using special hardware, as is later described in chapter 4.

How the preprocessing should be implemented is left out of the scope of this thesis. The options are:

1. Precompute control bits for all possible variations of the bitmasks and store them in memory. The upside of this is that while operating, the control bits are fast to load from memory. The downside is large memory consumption.
2. Compute the control bits in software on the fly. This is quite heavy operation on software, but if the bitmask only changes rarely, the effect of this can be negligible.
3. Compute the control bits using hardware. This is fast, but obviously comes at a hardware cost.
4. Some combination of these, for example using instruction set extensions to speed up software preprocessing.

The butterfly pattern can be implemented directly on hardware as in [11], but it can also be implemented using a $(2, 2^{n-1})$-perfect shuffle, in which case its implementation will

require $\log_2(N) = n$ separate stages, and a few cycles are needed for every stage.

Based on section 2.4, a $(2, 2^{n-1})$-shuffle will move element $i = i_1 \cdot 2^{n-1} + i_0$ to position $j = i_0 \cdot 2 + i_1$. Here $i_1 \in \{0, 1\}$ and $i_0 \in \{0, 1, \ldots, 2^{n-1} - 1\}$. This means that the bit at position j will be replaced by the bit at position i. Now consider bits at even positions $j = i_0 \cdot 2$. They will be replaced by bits from position $i = i_0$, since $i_1 = 0$. Then consider the odd positions, where $i_1 = 1$. Then $j = i_0 \cdot 2 + 1$, and $i = 2^{n-1} + i_0$. These can be combined to form a function that defines the permutation:

$$\begin{aligned} &\sigma : \{0, 1, \ldots, 2^n - 1\} \to \{0, 1, \ldots, 2^n - 1\}, \\ &\sigma(j) = \begin{cases} \frac{j}{2}, & \text{if } j \text{ is even} \\ \frac{j-1}{2} + 2^{n-1}, & \text{if } j \text{ is odd} \end{cases} \end{aligned} \tag{3.1}$$

Here $\sigma(j)$ tells which bit will replace the bit at position j. Now if j is represented in base 2 using n bits, then it can be noticed that for both even and odd indices $\sigma(j)$ equals to the cyclic shift of j to the right by one position (Similarly as in [5]).

$$\sigma(j) = \text{Binary cyclic right shift of } j \text{ by one position} \tag{3.2}$$

This property of the shuffle makes it usable for the implementation of the butterfly pattern.

For this thesis's purposes it is useful to describe the butterfly pattern as follows: In stage k, a bit at position j is either passed through or swapped with a bit at position j'. Consider representing j and j' in binary. Then j' can be computed from j by flipping the $(n - k + 1)^{\text{th}}$ bit, meaning that j and j' differ only by that one bit. For example, in figure 3.2 in the first stage the bit at position 3 (binary 011) is swapped with a bit at position 7 (binary 111).

To see why this description of the butterfly pattern holds, look at figure 3.2. In stage k, the bits are divided into 2^{k-1} blocks, and the swaps happen between the left and right half inside those blocks. The position j of a bit in a right half in stage k can be written:

$$j = 2^{n-k+1} \cdot m + i\,, \tag{3.3}$$

where $m \in \{0, 1, \ldots, 2^{k-1} - 1\}$ is the index of the block and $i \in \{0, 1, \ldots, 2^{n-k} - 1\}$ is the index of the bit inside the right half of that block. Now it is obvious that the $(n - k + 1)^{\text{th}}$ bit is zero in the binary representation of j. For comparison, it is similarly obvious that the third digit of $10^3 + q$ is zero in base 10, when $q \in \{0, 1, 2, \ldots, 10^2 - 1\}$.

In stage k this bit in the right half is then swapped with bit in the left half at position $j' = j + 2^{n-k}$. Since the $(n - k + 1)^{\text{th}}$ bit is zero in the binary representation of j, this now shows that the same bit is one in j', and the binary representation of j' is otherwise

the same, and thus the behavior is as described.

This description of the butterfly pattern can be used to justify why perfect shuffles can be used here to simplify the structure of the pattern. Here it is assumed that the operation of the first stage can be implemented. There could be special hardware for this, or this can be done using normal logic operations.

An example of the possible logic operations: Assume that the input bits of the butterfly pattern are held in two vectors, $\mathbf{b}_R$ and $\mathbf{b}_L$, both of size 2^{n-1}. The control bits are in a third vector $\mathbf{c}$, also of size 2^{n-1}. If these matched the first stage of the example illustrated in figure 3.2, then

$$\mathbf{b}_R = \begin{bmatrix} h \\ g \\ f \\ e \end{bmatrix}, \mathbf{b}_L = \begin{bmatrix} d \\ 0 \\ 0 \\ 0 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.4}$$

Here in $\mathbf{c}$, "1" indicates swap. Then to perform the swaps as described by the control bits of the first stage we can compute:

$$\begin{aligned} \mathbf{x} &= (\mathbf{b}_R \oplus \mathbf{b}_L) \cdot \mathbf{c} \\ \mathbf{b}'_R &= \mathbf{b}_R \oplus \mathbf{x} \\ \mathbf{b}'_L &= \mathbf{b}_L \oplus \mathbf{x} \end{aligned} \tag{3.5}$$

Here $\mathbf{b}'_R$ and $\mathbf{b}'_L$ are the new values of $\mathbf{b}_R$ and $\mathbf{b}_L$, $\mathbf{x}$ is a temporary variable and $\oplus$ and $\cdot$ are bitwise XOR and AND. The idea is that swapping needs to happen when the bits that will be swapped are different, and the corresponding control bit is one. Then the swapping corresponds with inverting both the bits at the swap positions.

So now it is assumed that the first stage can be implemented, meaning that we can implement a function f that takes as input the 2^n input bits $\mathbf{b}_1$ and 2^{n-1} control bits, and outputs bits $\mathbf{b}'_1$, where the correct swaps have been executed. (As described earlier, swaps happen for such bits positions whose binary representations are otherwise similar, but they differ only in the n^{th} bit, as is the case in stage 1.) The same implementation can be used for the other stages, when the $(2, 2^{n-1})$-shuffle is used.

Consider performing the $(2, 2^{n-1})$ shuffle on these bits. The shuffle is described by the function $\sigma(j)$. Previously it was established that $\sigma(j)$ is the cyclic n-bit binary right shift of j. Therefore, after the shuffle a bit that was at position j has been replaced by the bit that was in position j', where j' is the cyclic binary right shift of j. Denote:

$$\mathbf{b}_2(j) = \mathbf{b}'_1(j'), \tag{3.6}$$

where $\mathbf{b}_2 = S_{2,2^{n-1}}\mathbf{b}'_1$. Now $\mathbf{b}_2$ is used as an input to the function f. The swaps still

happen for bits at positions j and k, where the binary representations of j and k only differ in the n^{th} bit. However, as is seen in equation 3.6, the bit at position j of $\mathbf{b}_2$ originates from position j', which is a cyclic right shift of j. Therefore, the bits that are actually being swapped originate from positions j' and k'. Since j and k differed only in the n^{th} bit, j' and k' differ only in the $(n-1)^{\text{th}}$ bit. This is exactly the behavior that was needed in the butterfly pattern. The behavior continues to the next stages, and the position of the differing bit keeps shifting to the right.

After the last stage, a total of n shuffles has been performed. n cyclic shifts of an n-bit number will return the number back to the original. Therefore, after the last stage, the bits are back in the correct order, and the swaps have been performed correctly. Figure 3.3 shows this principle in action, using the same example as in figures 3.1 and 3.2.

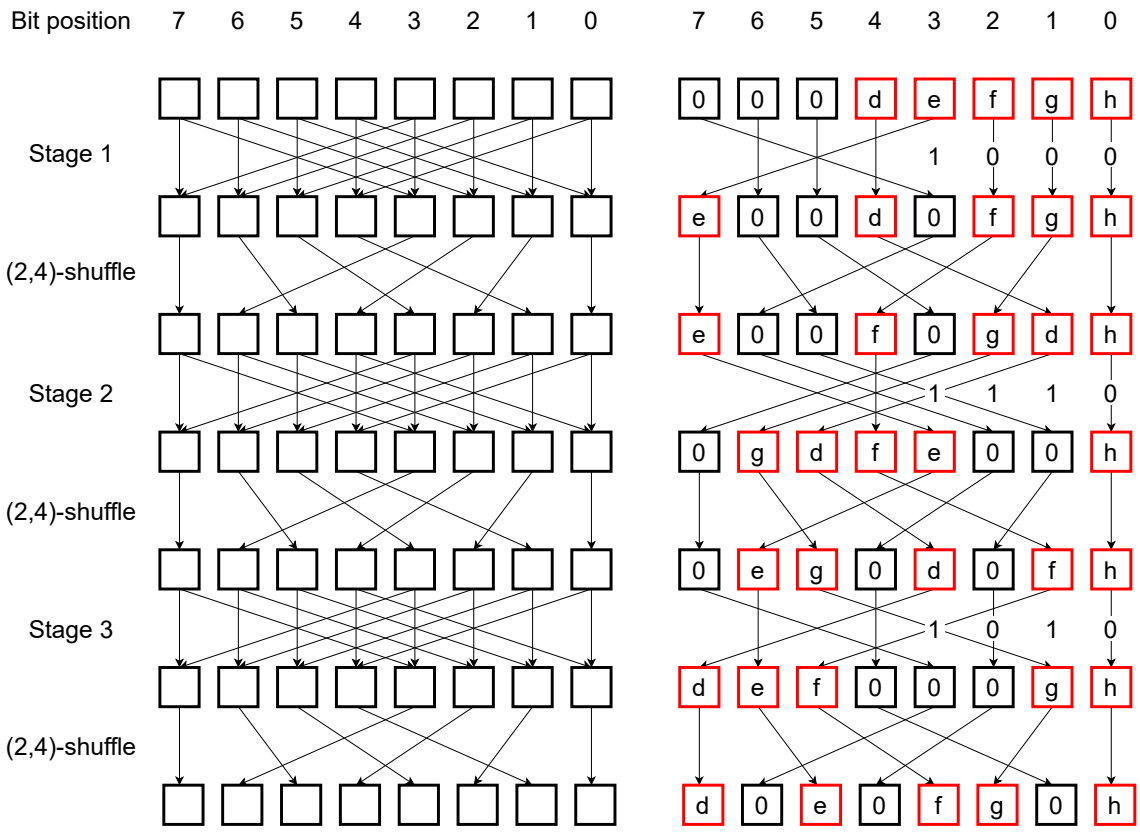

Figure 3.3. *An example of parallel deposit operation implemented using a butterfly pattern and perfect shuffles,* $N = 8$, $K = 5$.

It should be noted that to get the control bits to their correct positions, they too must be shuffled, if they are computed using the algorithm given in [11]. One way of achieving this is using the same $(2, 2^{n-1})$-shuffle. The control bits for the first stage are shuffled 0 times, the control bits for the second stage are shuffled once and so on. This shuffling can be included in the preprocessing stage.

A perfect shuffle $(2, 2^{n-1})$ that implements the frozen bit insertion for some $N = 2^n$ can also be used for smaller values of N. The smaller input vectors and bitmasks should just

be padded with zeros to make them correct size. This enables the use of the same shuffle for all possible values of N given in the 5G specification.

3.2 Polar Transform

The vector-matrix product $\mathbf{d} = \mathbf{u}G_N$ can be computed using only a bitwise XOR-operation and the same $(2, 2^{n-1})$-shuffle that was used in frozen bit insertion. This could be justified similarly as was done with the frozen bit insertion, but below this is shown using the Kronecker-power definition (equation 2.13) of the generator matrix G_N.

So far a row-vector $\mathbf{u}$ has been used for the info bits, since that is used in the source [4]. In this section however a column vector $\mathbf{v} = \mathbf{u}^T$ will be used, since that is used in [10]. Then

$$\mathbf{d}^T = (\mathbf{u}G_N)^T = G_N^T\mathbf{u}^T = (F^{\otimes n})^T\mathbf{v} = (F^T)^{\otimes n}\mathbf{v} \tag{3.7}$$

In general, a matrix $M = M_{n-1} \otimes M_{n-2} \otimes \ldots \otimes M_0$ can be factorized to an ordinary matrix product consisting of perfect shuffles and matrices M_i: [10]

$$\bigotimes_{i=n-1}^{0} M_i = \prod_{\substack{j=n-1 \\ j=i_\chi}}^{0} (I_{\alpha^i_{n-1}\ldots\alpha^i_{i+1}} \otimes M_i \otimes I_{\alpha^i_{i-1}\ldots\alpha^i_0}) \tag{3.8}$$

Here χ represents a permutation of $\{0, 1, \ldots, n-1\}$, meaning that the order at which the product is computed can be varied. Matrices M_i are of size $r_i \times c_i$ and $\alpha^i_k = r_k$ or $\alpha^i_k = c_k$ depending on certain conditions. [10]

The expression 3.8 can be simplified when it is assumed that all M_i are identical square matrices of size 2×2 and χ is selected to be the identity mapping on $\{0, 1, \ldots, n-1\}$. Using these assumptions we get [10]:

$$M^{\otimes n} = [S_{2^{n-1},2} \cdot (I_{2^{n-1}} \otimes M)]^n \tag{3.9}$$

For example, when $N = 8$ this leads to the interconnections seen in figure 3.4.

While this factorization could be used to implement polar transform, this is still not quite optimal. Imagine dividing the operation to stages as in figure 3.4. In each stage matrix multiplication must be performed on two adjacent bits. In the case of polar transform, where $M = F^T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, this would mean performing XOR-operations on adjacent bits. While not impossible, this is still not quite how an XOR-operation would by default be done on a computer. So instead of having $I_{2^{n-1}} \otimes M$ in the expression 3.9, we will modify the expression so that it will have the expression $M \otimes I_{2^{n-1}}$. When that expression

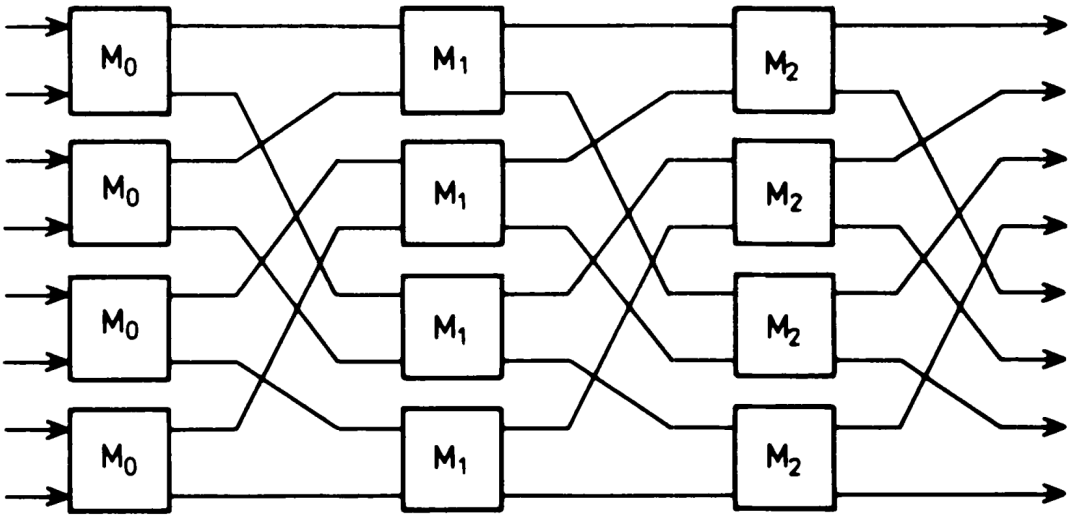

Figure 3.4. *An example of Kronecker matrix factorization, N = 8 [10]*

is used, the matrix-vector multiplication with 2^n bits stored in two variables $\mathbf{v}_0$ and $\mathbf{v}_1$ becomes simple to implement on a computer:

$$\begin{aligned}(M \otimes I_{2^{n-1}}) \begin{bmatrix}\mathbf{v}_0 \\ \mathbf{v}_1\end{bmatrix} &= (\begin{bmatrix}1 & 1 \\ 0 & 1\end{bmatrix} \otimes I_{2^{n-1}}) \begin{bmatrix}\mathbf{v}_0 \\ \mathbf{v}_1\end{bmatrix} \\ = \begin{bmatrix}I_{2^{n-1}} & I_{2^{n-1}} \\ 0 & I_{2^{n-1}}\end{bmatrix} \begin{bmatrix}\mathbf{v}_0 \\ \mathbf{v}_1\end{bmatrix} &= \begin{bmatrix}\mathbf{v}_0 \oplus \mathbf{v}_1 \\ \mathbf{v}_1\end{bmatrix}\end{aligned} \tag{3.10}$$

Now on the last line $\mathbf{v}_0 \oplus \mathbf{v}_1$ is simply the bitwise XOR-operation between two variables $\mathbf{v}_0$ and $\mathbf{v}_1$.

In addition to the beforementioned change, the shuffle pattern $S_{2^{n-1},2}$ will be changed to $S_{2,2^{n-1}}$. This is because that pattern was also used in the frozen bit insertion.

The paper [10] provides multiple ways to modify the expression. Begin from equation 3.8. That expression can be modified, since here $M_i = F^T$ for all i and F^T is a 2×2 matrix. Therefore $\alpha_k^i = 2$ for all i and k. For the order of the product i is chosen to go from 0 to $n-1$. (Opposite of what was chosen in the paper to get expression 3.9) Now we have:

$$(F^T)^{\otimes n} = \prod_{i=0}^{n-1} (I_{2^{n-i-1}} \otimes F^T \otimes I_{2^i}) \tag{3.11}$$

Use 2.20 in equation 3.11:

$$(F^T)^{\otimes n} = \prod_{i=0}^{n-1} S_{2^i,2^{n-i}} (I_{2^{n-1}} \otimes F^T) S_{2^{n-i},2^i} \tag{3.12}$$

Now consider actually computing the product. In the unwrapped product after the term $S_{2^{n-i},2^i}$ there is $S_{2^{i+1},2^{n-(i+1)}}$. Therefore, between each $(I_{2^{n-1}} \otimes F^T)$ we have

$$S_{2^{n-i},2^i} S_{2^{i+1},2^{n-(i+1)}} = S_{2^{n-i},2^i} S_{2^{i+1},2^{n-i-1}} \tag{3.13}$$

This can be simplified using equation 2.19.

$$S_{2^{n-i},2^i} S_{2^{i+1},2^{n-i-1}} = S_{2,2^i.2^{n-i-1}} = S_{2,2^{n-1}} \tag{3.14}$$

Applying that to equation 3.12:

$$(F^T)^{\otimes n} = \prod_{i=0}^{n-1} (I_{2^{n-1}} \otimes F^T) S_{2,2^{n-1}} \tag{3.15}$$

Here it is taken into account that $S_{2^i,2^{n-i}} = I$ when $i = 0$ and $S_{2^{n-i},2^i} = S_{2,2^{n-1}}$ when $i = n - 1$.

Now to get to the wanted expression, equation 2.21 can be used in addition to the fact that a (b_1, b_0)-shuffle is the inverse of (b_0, b_1)-shuffle [10]:

$$(I_{2^{n-1}} \otimes F^T) = S^{-1}_{2^{n-1},2} (F^T \otimes I_{2^{n-1}}) S^{-1}_{2,2^{n-1}} = S_{2,2^{n-1}} (F^T \otimes I_{2^{n-1}}) S^{-1}_{2,2^{n-1}} \tag{3.16}$$

Finally plugging this into 3.15:

$$(F^T)^{\otimes n} = \prod_{i=0}^{n-1} S_{2,2^{n-1}} (F^T \otimes I_{2^{n-1}}) S^{-1}_{2,2^{n-1}} S_{2,2^{n-1}} = \prod_{i=0}^{n-1} S_{2,2^{n-1}} (F^T \otimes I_{2^{n-1}}) \tag{3.17}$$

Now the final form on equation 3.17 can be used to compute polar transform on $N = 2^n$ bits. It can be seen that computing $\mathbf{d}^T = (F^T)^{\otimes n} \mathbf{v}$ consists of $n = \log_2(N)$ stages, and each stage has a similar structure. First $(F^T \otimes I_{2^{n-1}})\mathbf{v}$ is computed, and, as can be seen from equation 3.10, this can be implemented with an XOR-operation. That result is then multiplied by the permutation matrix $S_{2,2^{n-1}}$. This can be implemented with a custom instruction. An example of this process is given in figure 3.5.

In 5G NR, $N \in \{32, 64, 128, 256, 512, 1024\}$ [9]. Since N might not be constant, that would imply the need for many shuffle instructions, since the permutation $S_{2,2^{n-1}}$ depends on $n = \log_2 N$. However, that is not mandatory. Assume that the permutation instruction that implements $S_{2,2^{n-1}}$ exists for the largest possible n. (In this case $n_{\max} = \log_2 N_{\max} = \log_2 1024 = 10$.) Then $\mathbf{d} = \mathbf{u} G_{N_{\max}}$ can be computed in n regular stages, as described earlier.

The same hardware can be used to compute $\mathbf{d}_N = \mathbf{u}_N G_N$ for smaller N. The N bits $\mathbf{u}_N$

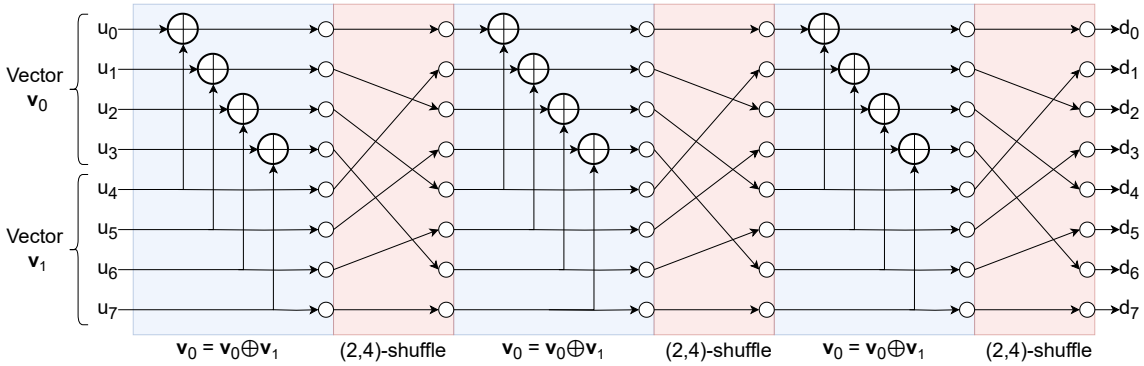

Figure 3.5. *Example of polar transform using a (2,4)-shuffle*

are loaded into the beginning of the $N_{\max}$-bit vector $\mathbf{u}$. The rest of the bits are set to zero. Then

$$\begin{aligned}
\mathbf{d} = \mathbf{u}G_{N_{\max}} &= \begin{bmatrix}\mathbf{u}_N & 0 & \dots & 0 & \dots & 0\end{bmatrix} \begin{bmatrix} G_{N_{\max}/2} & 0 \\ G_{N_{\max}/2} & G_{N_{\max}/2}\end{bmatrix} \\
&= \begin{bmatrix}\begin{bmatrix}\mathbf{u}_N & 0 & \dots & 0\end{bmatrix} G_{N_{\max}/2} & 0\end{bmatrix} \\
&= \begin{bmatrix}\begin{bmatrix}\mathbf{u}_N & 0 & \dots & 0\end{bmatrix} \begin{bmatrix} G_{N_{\max}/4} & 0 \\ G_{N_{\max}/4} & G_{N_{\max}/4}\end{bmatrix} & 0\end{bmatrix} \\
\dots &= \begin{bmatrix}\mathbf{u}_N G_N & 0\end{bmatrix}
\end{aligned} \tag{3.18}$$

So, even though generator matrix $G_{N_{\max}}$ is used, the result $\mathbf{d}$ contains the correct polar transform $\mathbf{u}_N G_N$ for any $N \leq N_{\max}$, followed by zeros. Now since the product $\mathbf{u}G_{N_{\max}}$ can be implemented as described earlier, this means that computation of $\mathbf{u}_N G_N$ for any $N \leq N_{\max}$ can be implemented using only the same instruction set extension that is needed for case $N = N_{\max}$.

3.3 Sub-Block Interleaving

Suppose that P_0 is the 32×32 permutation matrix that permutates a size 32 column vector according to the sub-block interleaving pattern given in [3]. Then it can be seen that interleaving N bits in the column vector $\mathbf{d}_N^T$ is performed by

$$\mathbf{y}_N^T = (P_0 \otimes I_{2^{n-5}})\mathbf{d}_N^T \tag{3.19}$$

This is because the block size is now $\frac{N}{32} = 2^{n-5}$, and multiplication by $(P_0 \otimes I_{2^{n-5}})$ performs the same permutation as P_0, but for blocks of size 2^{n-5}.

Define

$$P = I_{2^{n_{\max}-n}} \otimes P_0 \otimes I_{2^{n-5}} \tag{3.20}$$

to be the transformation matrix that performs sub-block interleaving. It can be seen that for $n = n_{\max}$ this definition is equivalent to 3.19. For smaller n multiplication by this matrix will still also correctly perform sub-block interleaving:

$$\begin{aligned}\mathbf{y}^T = P\mathbf{d}^T = P\begin{bmatrix}\mathbf{d}_N^T\\0\\\vdots\\0\end{bmatrix} &= \begin{bmatrix}P_0 \otimes I_{2^{n-5}} & 0 & \dots & 0\\0 & P_0 \otimes I_{2^{n-5}} & \dots & 0\\\vdots & \vdots & \ddots & \vdots\\0 & 0 & \dots & P_0 \otimes I_{2^{n-5}}\end{bmatrix}\begin{bmatrix}\mathbf{d}_N^T\\0\\\vdots\\0\end{bmatrix}\\ &= \begin{bmatrix}(P_0 \otimes I_{2^{n-5}})\mathbf{d}_N^T\\0\\\vdots\\0\end{bmatrix}\end{aligned} \tag{3.21}$$

Matrix P can be expressed using equation 2.20.

$$P = S_{2^{n-5},2^{n_{\max}-n+5}}(I_{2^{n_{\max}-5}} \otimes P_0)S_{2^{n_{\max}-n+5},2^{n-5}} \tag{3.22}$$

Using 2.19 we can derive an expression that can be used to factorize the permutation matrices in the equation above:

$$\begin{aligned}S_{2^k,2^{m-k}} &= S_{2^k,2^{m-k-1}\cdot 2} = S_{2^k\cdot 2^{m-k-1},2} \cdot S_{2^k\cdot 2,2^{m-k-1}} = (S_{2^{m-1},2}) \cdot S_{2^{k+1},2^{m-k-1}}\\ &= (S_{2^{m-1},2}) \cdot S_{2^{k+1},2^{m-k-2}\cdot 2} = (S_{2^{m-1},2}) \cdot S_{2^{k+1}\cdot 2^{m-k-2},2} \cdot S_{2^{k+1}\cdot 2,2^{m-k-2}}\\ &= (S_{2^{m-1},2})^2 \cdot S_{2^{k+2},2^{m-k-2}}\\ &\dots\\ &= (S_{2^{m-1},2})^{m-k}\end{aligned} \tag{3.23}$$

Now use that in equation 3.22:

$$P = (S_{2^{n_{\max}-1},2})^{n_{\max}-n+5}(I_{2^{n_{\max}-5}} \otimes P_0)(S_{2^{n_{\max}-1},2})^{n-5} \tag{3.24}$$

Finally, to change these shuffles to be the same shuffle used in polar transform, the cyclic property of perfect shuffles can be used. This nature of the shuffles can be seen by inserting $k = 0$ into equation 3.23:

$$S_{1,2^m} = I_{2^m} = (S_{2^{m-1},2})^m \tag{3.25}$$

Using that and the fact that a (b_1, b_0)-shuffle is the inverse of (b_0, b_1)-shuffle [10] we get:

$$(S_{2,2^{m-1}})^k = (S_{2^{m-1},2})^{m-k} \tag{3.26}$$

Now use that in 3.24 to get the final form for P:

$$P = (S_{2,2^{n_{\max}-1}})^{n-5}(I_{2^{n_{\max}-5}} \otimes P_0)(S_{2,2^{n_{\max}-1}})^{n_{\max}-n+5} \tag{3.27}$$

As stated earlier, sub-block interleaving for $N = 2^n$ bits can be done by computing $\mathbf{y}^T = P\mathbf{d}^T$, where $\mathbf{d}^T$ is an $N_{\max}$-bit vector and the information bits are located in the beginning of the vector. Now from equation 3.27 it can be seen that for all N, computing this vector-matrix product only consists of $I_{2^{n_{\max}-5}} \otimes P_0$ permutation and $n_{\max}$ iterations of the $S_{2,2^{n_{\max}-1}}$-permutation. The first permutation can be implemented as a custom instruction. The second permutation is the same that was used earlier in frozen bit insertion and in polar transform.

3.4 Comparison to Previous Work

A similar DSP-based implementation with some instruction set extensions is proposed in [12] by Guo *et al.* They introduce implementations on frozen bit insertion and polar transform.

The frozen bit insertion proposed in [12] divides the at max 1024-bit insertion operation to several parallel 8-bit parts. This division into multiple smaller groups is the largest difference between their implementation and the one presented in this thesis. Their algorithm for this includes preprocessing and “Longitudinal Concentration”, “Concatenation and Right Shifting” and “Insertion in Byte”. Longitudinal concentration uses a custom shuffle unit also used in their implementation of polar transform. The other two operations use both a custom instruction. Therefore, they need more custom instructions for their implementation compared to this thesis. Common to both implementations is the need for preprocessing. [12]

The problem with implementing the XOR pattern seen in figure 2.1 is that its bit-level addressing is difficult to implement with a DSP [12]. Both implementations (this thesis and [12]) solve this by dividing the XOR pattern to $n = \log_2(N)$ stages and having an interleaving pattern between the stages. The division into stages is illustrated with blue colors in the figure 2.1.

A major difference between the two implementations is that their implementation uses different block sizes for each interleave stage. That means that they need a special shuffle unit, and special handling for cases where the block size is smaller than a byte [12],

whereas the solution described in this thesis uses the same interleave pattern for each of the stages, which enables the interleaving to be implemented with a single instruction that consists only of wiring bits to their correct places, and no logic is necessarily needed for the instruction itself.

4. HARDWARE CONSIDERATIONS

In chapter 3 it was shown that polar encoding can be efficiently implemented using 2 bit-level permutations and basic logic operations. This chapter focuses on how these instructions can be implemented, and what is required of the DSP that performs these.

4.1 SIMD Width

The permutations that need implementing are $S_{2,2^{n_{\max}-1}}$ and $I_{2^{n_{\max}-5}} \otimes P_0$. It can be noticed that both of these matrices are of size $2^{n_{\max}} \times 2^{n_{\max}}$, which in this case means size 1024×1024. This means that the instruction implementing these would need to take 1024 bits as input and would output 1024 bits. This might not be possible on every processor. However, with some assumptions there is a way to get around this problem.

Assume that a DSP has a SIMD width of 2^{n-1} bits, meaning that it can handle variables of length 2^{n-1}. Then assume that this DSP is capable of having an instruction that takes two 2^{n-1}-bit variables as input, and outputs also two variables. Then implementing an instruction that can perform an $S_{2,2^{n-1}}$ -shuffle is possible. Use another property of perfect shuffles:

$$S_{b_2,b_1b_0} = (I_{b_1} \otimes S_{b_2,b_0})(S_{b_2,b_1} \otimes I_{b_0}) \tag{4.1}$$

This equation is similar to equation 39 in [10], and the proof is similar. Using this:

$$S_{2,2^{n_{\max}-1}} = S_{2,2^{n_{\max}-n}.2^{n-1}} = (I_{2^{n_{\max}-n}} \otimes S_{2,2^{n-1}})(S_{2,2^{n_{\max}-n}} \otimes I_{2^{n-1}}) \tag{4.2}$$

This equation shows that permutation $S_{2,2^{n_{\max}-1}}$ can be implemented using the assumed DSP in two steps: First permutation $S_{2,2^{n_{\max}-n}}$ is done in SIMD-width granularity. Then $S_{2,2^{n-1}}$ is done separately on multiple blocks.

Permutation $I_{2^{n_{\max}-5}} \otimes P_0$ is simple to implement on any machine where the SIMD width is larger than 32, the size of P_0. This is because permutation $I_{2^{n_{\max}-5}} \otimes P_0$ means basically just performing P_0 separately on each 32-bit block of the input, and therefore this permutation is trivial to divide to appropriate SIMD width sized parts.

As a sidenote, instead of implementing these bit permutations directly on hardware, it could also be possible to implement a configurable bit permutation instruction capable of

doing any bit permutation using the same instruction [13].

4.2 Parallel Deposit Preprocessing

In section 3.1 it was mentioned that the preprocessing required to compute the control bits for a parallel deposit operation is quite heavy to implement directly on software. This is not an issue if the bitmask used in parallel deposit only changes rarely, since the control bits are independent of the actual info data. For dynamic bitmasks some special hardware might however be needed. The hardware that is proposed in [11] consists of operations they call “population count” and “Left Rotate and Complement upon wrap around” (LROTC).

The first hardware stage they propose computes the population counts of the bitmask all in parallel. Here population count simply means the number of ones in the bitmask in a certain range. The counts are computed from mask index 0 to i for all $i \in \{1, 2, \ldots, N-2\}$. To implement this, they use a network of full adders.

The outputs of the population counts are directed to control different size LROTC-circuits. As the name suggests, those circuits cyclically rotate bits and complement them upon wraparound. They use a modified barrel shifter to implement this. Major simplifications are possible, since the input bits to all the LROTC circuits are all zeros. (The population counts control the shift amounts.) An example of the complete hardware is shown in figure 4.1.

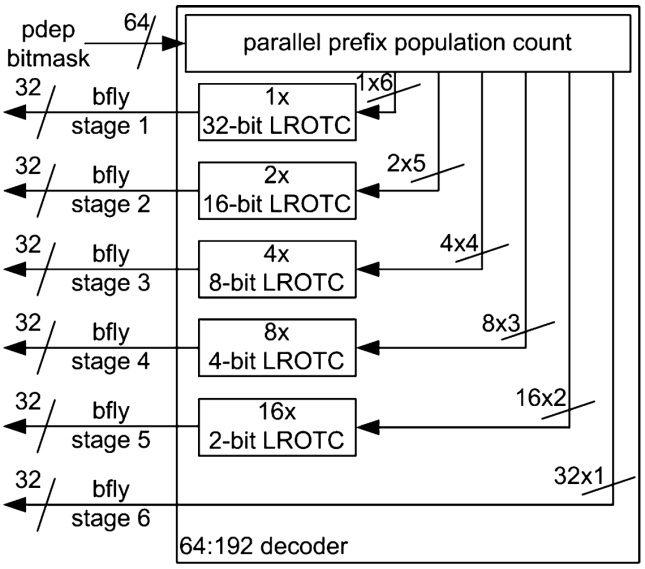

Figure 4.1. *Hardware for computing control bits for parallel deposit operations, N = 64 [11]*

Their synthesis of this hardware showed that in terms of cycle time this unit will be about 16% slower than an ALU of the same size and 2.25 times larger in area. [11]

5. CONCLUSIONS

In this thesis it was shown that polar transform can be computed fully parallel using a DSP and one perfect shuffle permutation that can be implemented as an instruction set extension. The same perfect shuffle was shown to be usable for frozen bit insertion and sub-block interleaving. All the procedures are implemented in $\log_2 N$ regular stages, where N is the code block size. One perfect shuffle instruction was shown to be usable for all possible code block lengths.

It was possible to modify the perfect-shuffle expressions as was needed for this thesis's purposes using tools given in previous work. It should be noted that the final forms given in equations 3.17 and 3.27 are not the only possible factorizations for these expressions. This is just one possible solution that enables the use of the same perfect shuffle in all the procedures.

This thesis only introduces ideas for the implementation of polar coding on a DSP, but an actual implementation analysis is not included here. Before the writing process of this thesis a DSP implementation of the ideas presented here was actually done as a proof of concept. To properly compare this thesis with previous work, that implementation should be studied thoroughly in terms of performance and area. It would be logical to also study other procedures of the encoding chain shown in figure 2.2 and include those in the DSP implementation.

The preprocessing of frozen bit insertion should also be studied further. It is left out of the scope of this thesis whether the hardware mentioned in section 4.2 is needed in 5G context, or whether a software solution possibly combined with some instruction set extensions would suffice. The control bits for different bitmasks could also be stored in memory, and used later, saving preprocessing time.

REFERENCES

[1] G. Miao, J. Zander, K. W. Sung, and S. B. Slimane, *Fundamentals of mobile data networks*. Cambridge University Press, 2016.

[2] T. K. Moon, *Error correction coding: Mathematical methods and algorithms*, eng. Newark: John Wiley & Sons, Incorporated, 2020, ISBN: 9781119567479.

[3] 3GPP, "5G;NR;Multiplexing and channel coding", 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 38.212, Mar. 2023, Version 17.5.0. [Online]. Available: https : / / portal . 3gpp . org / desktopmodules / Specifications / SpecificationDetails.aspx?specificationId=3214.

[4] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels", eng, *IEEE transactions on information theory*, vol. 55, no. 7, pp. 3051–3073, 2009, ISSN: 0018-9448.

[5] H. Stone, "Parallel processing with the perfect shuffle", eng, *IEEE transactions on computers*, vol. C-20, no. 2, pp. 153–161, 1971, ISSN: 0018-9340.

[6] M. Purser, *Introduction to error-correcting codes*. Boston. London, 1995.

[7] C. E. Shannon, "A mathematical theory of communication", eng, *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948, ISSN: 0005-8580.

[8] V. Bioglio, C. Condo, and I. Land, "Design of polar codes in 5G New Radio", eng, *IEEE Communications surveys and tutorials*, vol. 23, no. 1, pp. 29–40, 2021, ISSN: 1553-877X.

[9] Z. B. Kaykac Egilmez, L. Xiang, R. G. Maunder, and L. Hanzo, "The development, operation and performance of the 5G polar codes", eng, *IEEE Communications surveys and tutorials*, vol. 22, no. 1, pp. 96–122, 2020, ISSN: 1553-877X.

[10] M. Davio, "Kronecker products and shuffle algebra", eng, *IEEE transactions on computers*, vol. C-30, no. 2, pp. 116–125, 1981, ISSN: 0018-9340.

[11] Y. Hilewitz and R. B. Lee, "Fast bit gather, bit scatter and bit permutation instructions for commodity microprocessors", eng, *Journal of signal processing systems*, vol. 53, no. 1-2, pp. 145–169, 2008, ISSN: 1939-8018.

[12] Y. Guo, S. Xie, Z. Liu, L. Yang, and D. Wang, "Parallel polar encoding in 5G communication", eng, in *2018 IEEE Symposium on Computers and Communications (ISCC)*, IEEE, 2018, pp. 00 064–00 069, ISBN: 9781538669501.

[13] Y. Hilewitz, Z. Shi, and R. Lee, "Comparing fast implementations of bit permutation instructions", eng, in *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004*, vol. 2, Piscataway NJ: IEEE, 2004, 1856–1863 Vol.2, ISBN: 0780386221.