Flexible and Low-Complexity Encoding and Decoding of Systematic Polar Codes

Gabi Sarkis, Ido Tal, *Member, IEEE*, Pascal Giard, *Student Member, IEEE*, Alexander Vardy, *Fellow, IEEE*, Claude Thibeault, *Senior Member, IEEE*, and Warren J. Gross, *Senior Member, IEEE*

***Abstract*—In this work, we present hardware and software implementations of flexible polar systematic encoders and decoders. The proposed implementations operate on polar codes of any length less than a maximum and of any rate. We describe the low-complexity, highly parallel, and flexible systematic-encoding algorithm that we use and prove its correctness. Our hardware implementation results show that the overhead of adding code rate and length flexibility is little, and the impact on operation latency minor compared to code-specific versions. Finally, the flexible software encoder and decoder implementations are also shown to be able to maintain high throughput and low latency.**

***Index Terms*—polar codes, systematic encoding, multi-code encoders, multi-code decoders.**

I. Introduction

Modern communication systems must cope with varying channel conditions and differing throughput and transmission latency constraints. The 802.11-2012 wireless communication standard, for example, requires more than twelve error-correction configurations, increasing implementation complexity [1], [2]. Such a requirement necessitates encoder and decoder implementations that are flexible in code rate and length.

Polar codes achieve the symmetric capacity of memoryless channels with an explicit construction and are decoded with the low-complexity successive-cancellation decoding algorithm [3]. In this paper, we show that apart from the above favorable properties, polar codes are highly amenable to flexible encoding and decoding. That is, their regular structure enables encoder and decoder implementations that support any polar code of any rate and length, under the constraint of a maximal codeword length.

Systematic polar coding was described in [4] as a method to ease information extraction and improve bit-error rate without affecting the frame-error rate. The systematic encoding scheme originally proposed in [4] is serial by nature, and seems non-trivial to parallelize, unless restricted to a single polar code of fixed rate and length. The serial nature of this encoding ($O(n \cdot \log n)$ time-complexity, where n is the code length) places a speed limit on the encoding process which gets worse with increasing code length. To address this, a new systematic encoding algorithm that is easy to parallelize was first described in [5]. This algorithm is both parallel and flexible in code rate. In this work, we extend the flexibility to code length as well and provide hardware and software implementations that achieve throughput values of 29 Gbps and 10 Gbps, respectively.

We dedicate a portion of this work to proving the correctness of the systematic encoding algorithm presented in [5]. We prove that it results in valid systematic polar codewords when the sub-matrix of the encoding matrix with rows and columns corresponding to information bit indices is an involution. We prove that this condition is satisfied for both polar and Reed-Muller codes since they both satisfy a property we call *domination contiguity*, which we prove is a sufficient condition for the involution to be true.

This paper is organized into two parts addressing flexible encoding and decoding, respectively. The first part starts with Section II where we define some preliminary notation and contrast the implementation of the original systematic encoder presented in [4] with that of [5]. Note that reading [5] or [4] *is not* a prerequisite to reading the current paper, since we summarize the key points needed from those papers in Section II. Section III is mainly about setting notation and casting the various operations needed in matrix form. In Section IV, we define the property of domination contiguity, and prove that our algorithm works—in both natural and bit-reversed modes—if this property is satisfied. The fact that domination contiguity indeed holds for polar codes is proved in Section V. With correctness of the algorithm proved, flexible hardware and software systematic encoder implementations are presented in Sections VI and VII.

The second part of this paper deals with flexibility of decoders with respect to codeword length. Sections VIII and IX discuss such flexibility with respect to hardware and software implementations of the state-of-the-art fast simplified successive-cancellation (Fast-SSC) decoding algorithm, respectively. The rate and length flexible hardware implementations we present have the same latency and throughput as their rate-only flexible counterparts and incur only a minor increase in complexity. The proposed flexible software decoders can achieve 73% the throughput of the code-specific decoders.

We would like to mention that some of the proofs presented in this paper were arrived at independently by Li et al. in [6] and [7]. Specifically, the result that is most relevant to

G. Sarkis, P. Giard, and W. J. Gross are with the Department of Electrical and Computer Engineering, McGill University, Montréal, QC H3A 0E9, Canada (e-mails: {gabi.sarkis, pascal.giard}@mail.mcgill.ca, warren.gross@mcgill.ca).

I. Tal is with the Technion—Israel Institute of Technology, Haifa 32000, Israel. (e-mail: idotal@ee.technion.ac.il).

A. Vardy is with the Department of Electrical and Computer Engineering and the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093, USA (e-mail: avardy@ucsd.edu).

C. Thibeault is with the Department of Electrical Engineering, École de technologie supérieure, Montréal, QC H3C 1K3, Canada (e-mail: claude.thibeault@etsmtl.ca).

our setting in [6] is Theorem 1 thereof as well as the two corollaries that follow. The closest analog in our paper to these results is what one can deduce by combining equations (22), (27), and (28). However, in contrast to [6], our proof is more general since we do not limit ourselves to constructing the polar code via the Bhattacharyya parameter. Also, the results of [6] are *not* used in that paper for efficient systematic encoding. A systematic encoder based on these results was given later in [7], although that encoder is not as amenable to flexible parallel implementation as the encoder proposed in [5] and this paper. We also note that Proposition 3 of [7] is analogous to our Theorem 1, although the proofs are different.

II. Background

We start by defining what we mean by a "systematic encoder", with respect to a general linear code. For integers $0 < k \leq n$, let $G = G_{k \times n}$ denote a $k \times n$ binary matrix with rank k. The notation G is used to denote a generator matrix. Namely, the code under consideration is

$$\operatorname{span}(G) = \left\{ \mathbf{v} \cdot G \mid \mathbf{v} \in \mathrm{GF}(2)^k \right\} .$$

An encoder

$$\mathcal{E} \colon \mathrm{GF}(2)^k \to \operatorname{span}(G)$$

is a one-to-one function mapping an *information bit vector*

$$\mathbf{u} = (u_0, u_1, \ldots, u_{k-1}) \in \mathrm{GF}(2)^k$$

to a *codeword*

$$\mathbf{x} = (x_0, x_1, \ldots, x_{n-1}) \in \operatorname{span}(G) .$$

All the encoders discussed in this paper are linear. Namely, all can be written in the form

$$\mathcal{E}(\mathbf{u}) = \mathbf{u} \cdot \Pi \cdot G , \tag{1}$$

where $\Pi = \Pi_{k \times k}$ is an invertible matrix defined over GF(2).

The encoder $\mathcal{E}$ is systematic if there exists a set of k *systematic indices*

$$S = \{s_j\}_{j=0}^{k-1} , \quad 0 \leq s_0 < s_1 < \cdots < s_{k-1} \leq n-1 , \tag{2}$$

such that restricting $\mathcal{E}(\mathbf{u})$ to the indices S yields $\mathbf{u}$. Specifically, position s_i of $\mathbf{x}$ must contain u_i. Note that our definition of "systematic" is stronger than some definitions. That is, apart from requiring that the information bits be embedded in the codeword, we further require that the embedding is in the natural order: u_i appears before u_j if $i < j$.

Since G has rank k, there exist k linearly independent columns in G. Thus, we might naively take Π as the inverse of these columns, take S as the indices corresponding to these columns, and state that we are done. Of course, the point of [5] and [4] is to show that the calculations involved can be carried out efficiently with respect to the computational model considered. We now briefly present and discuss these two solutions.

A. The Arıkan systematic encoder [4]

Recall [3] that a generator matrix of a polar code is obtained as follows. We define the Arıkan kernel matrix as

$$F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} . \tag{3}$$

The m-th Kronecker product of F is denoted $F^{\otimes m}$ and is defined recursively as

$$F^{\otimes m} = \begin{bmatrix} F^{\otimes(m-1)} & 0 \\ F^{\otimes(m-1)} & F^{\otimes(m-1)} \end{bmatrix} , \quad \text{where } F^{\otimes 1} = F . \tag{4}$$

From this point forward, we adopt the shorthand

$$m \triangleq \log_2 n .$$

In order to construct a polar code of length $n = 2^m$, we apply a bit-reversing operation [3] to the columns of $F^{\otimes m}$. From the resulting matrix, we erase the $n - k$ rows corresponding to the frozen indices. The resulting $k \times n$ matrix is the generator matrix.

A closely related variant is a code for which the column bit-reversing operation is not carried out. We follow [4] and present the encoder there in the context of a non-reversed polar code. Let the complement of the frozen index set be denoted by

$$A = \{\alpha_j\}_{j=0}^{k-1} , \quad 0 \leq \alpha_0 < \alpha_1 < \cdots < \alpha_{k-1} \leq n-1 . \tag{5}$$

The set A is termed the set of *active rows*.

A simple observation which is key is that the matrix $F^{\otimes m}$ is lower triangular with all diagonal entries equal to 1. This is easily proved by induction using the definition of F and (4). An immediate corollary is the following. Suppose we start with $F^{\otimes m}$ and keep only the rows indexed by A (thus obtaining the generator matrix G). From this matrix, we keep only the k columns indexed by A. We are left with a $k \times k$ lower triangular matrix with all diagonal entries equal to 1. Specifically, we are left with an invertible matrix. In the setting of (1) and (2) we have that Π is the inverse of this matrix and the set S of systematic indices simply equals A.

As previously mentioned, the above description is not enough: we must show an efficient implementation. We now briefly outline the implementation in [4], which results in an encoding algorithm running in time $O(n \cdot \log n)$. Let us recall our goal, we must find a codeword $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ such that, using the notation in (5), we have that x_{α_i} equals u_i. Since $\mathbf{x}$ is a codeword, it is the result of multiplying the generator matrix G by some length-k vector from the left. As mentioned, G is obtained by removing from $F^{\otimes m}$ the rows whose index is not contained in A. Thus, we can alternatively state our goal as follows. We must find a codeword $\mathbf{x}$ as described above such that $\mathbf{x} = \mathbf{v} \cdot F^{\otimes m}$, where $\mathbf{v} = (v_0, v_1, \ldots, v_{n-1})$ is such that $v_i = 0$ whenever $i \notin A$.

We will now show a recursive implementation to the systematic encoding function $\mathsf{Encode}_m(\mathbf{u}, A)$. Let us start by considering the stopping condition, $m = 0$. As a preliminary step, define $F^{\otimes 0} = 1$, a 1×1 matrix. Note that this definition is consistent with (3) and (4) for $m = 1$. Next, note that if $m = 0$, then the problem is trivial: if A is empty than we are forced to have $\mathbf{v} = (0)$, and thus $\mathsf{Encode}_0(\mathbf{u}, A)$ returns

$\mathbf{x} = (0)$, the all-zero codeword. Otherwise, $A = \{0\}$ and we simply take $\mathbf{v} = (u_0)$ and return $\mathbf{x} = (u_0)$, as prescribed.

Let us now consider the recursion itself. Assume $m \geq 1$ and write $\mathbf{v} = (\mathbf{v}', \mathbf{v}'')$, where $\mathbf{v}'$ consists of the first $n/2$ entries of $\mathbf{v}$ and $\mathbf{v}''$ consists of the last $n/2$ entries of $\mathbf{v}$. Let $\mathbf{x} = (\mathbf{x}', \mathbf{x}'')$ be defined similarly. By the block structure in (4), we have that

$$\mathbf{x}'' = \mathbf{v}'' \cdot F^{\otimes(m-1)} \tag{6}$$

$$\mathbf{x}' = \mathbf{v}' \cdot F^{\otimes(m-1)} + \mathbf{x}'' \; . \tag{7}$$

We will find $\mathbf{x}$ by first finding $\mathbf{x}''$ and then finding $\mathbf{x}'$. Towards that end, let

$$A' = \{\alpha : \alpha \in A \text{ and } \alpha < n/2\},$$
$$A'' = \{\alpha - n/2 : \alpha \in A \text{ and } \alpha \geq n/2\}.$$

Finding $\mathbf{x}''$ is a straightforward recursive process. Namely, by (6) if we define $\mathbf{u}'' = (u_{i+n/2})_{i \in A''}$, then $\mathbf{x}'' = \mathsf{Encode}_{m-1}(\mathbf{u}'', A'')$. Now, with $\mathbf{x}''$ calculated, we can find $\mathbf{x}'$. Namely, considering (7), we need a $\mathbf{v}'$ for which entry α_i of $\mathbf{v}' \cdot F^{\otimes(m-1)}$ equals $u_i + x''_{\alpha_i}$. Thus, defining $\mathbf{u}' = (u_i + x''_{\alpha_i})_{i \in A'}$, we have that $\mathbf{x}' = \mathsf{Encode}_{m-1}(\mathbf{u}', A')$.

The main point we want to stress about the above encoder is the *serial* nature of it: *first* calculate $\mathbf{x}''$ and *only after* that is done, calculate $\mathbf{x}'$.

A parallel, higher complexity implementation of the algorithm in [4], when the frozen bits are set to 0, can calculate the parity bits directly using matrix multiplication:

$$\mathbf{x}_{A^c} = \mathbf{u}(F^{\otimes m}_{AA})^{-1} F^{\otimes m}_{AA^c},$$

where $F^{\otimes m}_{AA}$ is a sub-matrix containing rows and columns of $F^{\otimes m}$ corresponding to information-bit indices. Similarly, $F^{\otimes m}_{AA^c}$ contains the rows and columns of $F^{\otimes m}$ that correspond to information and frozen bit indices, respectively. The dimensions of $F^{\otimes m}_{AA}$ and $F^{\otimes m}_{AA^c}$ change with code rate, in contrast to our encoder which always uses the fixed $F^{\otimes m}$. A parallel multiplier that can accommodate matrices of varying dimension leads to a significant increase in implementation complexity.

B. The systematic encoder [5]

We now give a high-level description of the encoder in [5]. As before, we consider a non-reversed setting. Recall that A in (5) is the set of active row indices.

1) We first expand $\mathbf{u} = (u_0, u_1, \ldots, u_{k-1})$ into a vector $\mathbf{v}_\mathrm{I}$ of length n as follows: for all $0 \leq i < k$ we set entry α_i of $\mathbf{v}_\mathrm{I}$ equal to u_i. The remaining $n - k$ entries of $\mathbf{v}_\mathrm{I}$ are set to 0.
2) We calculate $\mathbf{v}_\mathrm{II} = \mathbf{v}_\mathrm{I} \cdot F^{\otimes m}$.
3) The vector $\mathbf{v}_\mathrm{III}$ is gotten from $\mathbf{v}_\mathrm{II}$ by setting all entries not in A to zero.
4) We return $\mathbf{x} = \mathbf{v}_\mathrm{III} \cdot F^{\otimes m}$.

Clearly, steps 1 and 3 can be implemented very efficiently in any computational model. The interesting part is calculations of the form $\mathbf{v} \cdot F^{\otimes m}$, for a vector $\mathbf{v}$ of length n.

As we will expand on later, the main merit of [5] is that the computation of $\mathbf{v} \cdot F^{\otimes m}$ can be done *in parallel*. Namely, if $\mathbf{v} = (\mathbf{v}', \mathbf{v}'')$, where $\mathbf{v}'$ (respectively, $\mathbf{v}''$) equals the first (respectively, last) $n/2$ entries of $\mathbf{v}$, then one can calculate $\mathbf{v}' \cdot F^{\otimes(m-1)}$ and $\mathbf{v}'' \cdot F^{\otimes(m-1)}$ *concurrently* and then, by (4), combine the results to get

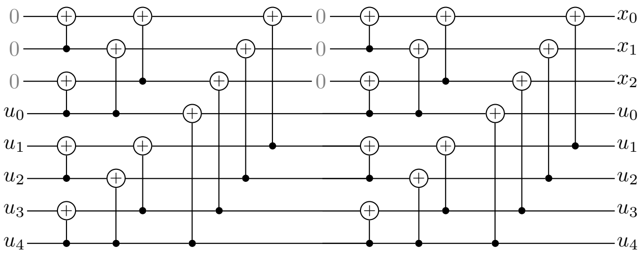


Fig. 1. The systematic encoder of [5] for an (8, 5) polar code.

$$\mathbf{v} \cdot F^{\otimes m} = ([\mathbf{v}' \cdot F^{\otimes(m-1)}] + [\mathbf{v}'' \cdot F^{\otimes(m-1)}], [\mathbf{v}'' \cdot F^{\otimes(m-1)}]) \; .$$

We also note that the systematic encoder in [5] is easily described as two applications of a non-systematic encoder, with a zeroing operation applied in-between. Thus, any advances made with respect to non-systematic encoding of polar codes immediately yield advances in systematic encoding.

Lastly, we state that both the encoder presented in [4] as well as the one presented in [5] produce the same codeword when given the same information vector. To see this, note that on the one hand, both encoders operate with respect to the same code of dimension k. That is, with respect to the same generator matrix G described above. On the other hand, by definition, both encoders produce the same output when restricted to the k systematic indexes A of the codeword. That is, to k indexes such that restricting the generator matrix G to them results in a $k \times k$ invertible matrix, as previously explained. Thus, the error-correction performance of a system utilizing the same decoder with either encoder remains the same

III. Systematic, reversed, and non-reversed codes

This section is devoted to recasting the concepts and operation presented in the previous section into matrix terminology. We start by discussing a general linear code, and then specialize to both non-reversed and reversed polar codes. Recalling the definition of S as the set of systematic indices, define the *restriction matrix* $R = R_{n \times k}$ corresponding to S as

$$R = (R_{i,j})_{i=0}^{n-1}{}_{j=0}^{k-1} \; , \quad \text{where} \quad R_{i,j} = \begin{cases} 1 & \text{if } i = s_j \; , \\ 0 & \text{otherwise} \; . \end{cases} \tag{8}$$

With this definition at hand, we require that a systematic encoder satisfy $\mathcal{E}(\mathbf{u}) \cdot R = \mathbf{u}$, or equivalently that

$$\Pi \cdot G \cdot R = I \; , \tag{9}$$

where I above denotes the $k \times k$ identity matrix. Our proofs will center on showing that (9) holds.

A. Non-reversed polar codes

In this subsection, we consider a non-reversed polar code. Recall the definition in (5) of A being the set of active rows, where the jth smallest element of A is denoted α_j. For this case, recall that we define S as equal to A and s_j as equal to α_j.

Define the matrix E as

$$E = (E_{i,j})_{i=0}^{k-1}{}_{j=0}^{n-1} \ , \quad \text{where} \quad E_{i,j} = \begin{cases} 1 & \text{if } j = \alpha_i \ , \\ 0 & \text{otherwise} \ . \end{cases} \tag{10}$$

The matrix E will be useful in several respects. First, note by the above that applying E to the left of a matrix with n rows results in a submatrix containing only the rows indexed by A. Thus, we have that

$$G_{\mathrm{nrv}} = E \cdot F^{\otimes m} \ , \tag{11}$$

where G_{nrv} is the generator matrix of our code, and "nrv" is short for "non-reversed".

Next, note that by applying E to the right of a vector $\mathbf{u}$ of length k, we manufacture a vector $\mathbf{v}_{\mathrm{I}}$ such that the entries indexed by α_j equal u_j and all other entries equal zero. That is,

$$\mathbf{v}_{\mathrm{I}} = \mathbf{u} \cdot E \ ,$$

as per step 1 of the algorithm described in Subsection II-B. Because of this property, we refer to E as the *expanding matrix*.

Let us move on to step 3 of the algorithm. Simple algebra yields that

$$\mathbf{v}_{\mathrm{III}} = \mathbf{v}_{\mathrm{II}} \cdot E^T \cdot E \ .$$

That is, multiplying a vector of length n from the right by $E^T \cdot E$ results in a vector in which the entries indexed by A remain the same while the entries not indexed by A are set to zero.

The above equations yield a succinct description of our algorithm,

$$\mathcal{E}_{\mathrm{nrv}}(\mathbf{u}) = \mathbf{u} \cdot \underbrace{E \cdot F^{\otimes m} \cdot E^T}_{\Pi} \cdot \underbrace{E \cdot F^{\otimes m}}_{G_{\mathrm{nrv}}} \ . \tag{12}$$

We end this section by noting that by (8) and (10), we have that

$$E^T = R \ .$$

Thus, recalling (9), our aim is to prove that

$$E \cdot F^{\otimes m} \cdot E^T \cdot E \cdot F^{\otimes m} \cdot E^T = I \ . \tag{13}$$

Showing this will further imply that the corresponding Π in (12) is indeed invertible.

B. Reversed polar codes

As explained, we will consider bit-reversed as well as non-bit-reversed polar codes. Let us introduce corresponding notation. For an integer $0 \leq i < n$, denote the binary representation of i as

$$\langle i \rangle_2 = (i_0, i_1, \ldots, i_{m-1}) \ , \qquad \text{where } i = \sum_{j=0}^{m-1} i_j 2^j \text{ and } i_j \in \{0,1\} \ . \tag{14}$$

For i as above, we define $\overleftarrow{i}$ as the integer with reversed binary representation. That is,

$$\langle \overleftarrow{i} \rangle_2 = (i_{m-1}, i_{m-2}, \ldots, i_0) \ , \quad \overleftarrow{i} = \sum_{j=0}^{m-1} i_j 2^{m-1-j} \ .$$

As in [3], we denote the $n \times n$ bit reversal matrix as B_n. Recall that B_n is a permutation matrix. Specifically, multiplying a matrix from the left (right) by B_n results in a matrix in which row (column) i equals row (column) $\overleftarrow{i}$ of the original matrix.

Recall that we have denoted by A the set of active rows. We stress that this notation holds for *both the reversed as well as the non-reversed setting*. Thus, recalling (10), we have analogously to (11) that

$$G_{\mathrm{rv}} = E \cdot F^{\otimes m} \cdot B_n$$

where G_{rv} is the generator matrix of our code, and "rv" is short for "reversed". By [3, Proposition 16], we know that $B_n \cdot F^{\otimes m} = F^{\otimes m} \cdot B_n$. Thus, it also holds that

$$G_{\mathrm{rv}} = E \cdot B_n \cdot F^{\otimes m} \ . \tag{15}$$

In the interest of a lighter notation later on, we now "fold" the bit-reversing operation into the set A. Thus, define the set of bit-reversed active rows, $\overleftarrow{A}$, gotten from the set of active rows A by applying the bit-reverse operation on each element α_i. As before, we order the elements of $\overleftarrow{A}$ in increasing order and denote

$$\overleftarrow{A} = \{\beta_j\}_{j=0}^{k-1} \ , \quad 0 \leq \beta_0 < \beta_1 < \cdots < \beta_{k-1} \leq n-1 \ . \tag{16}$$

Recall that the expansion matrix E was defined using A. We now define $\overleftarrow{E} = \overleftarrow{E}_{k \times n}$ according to $\overleftarrow{A}$ in exactly the same way. That is,

$$\overleftarrow{E} = (\overleftarrow{E}_{i,j})_{i=0}^{k-1}{}_{j=0}^{n-1} \ , \quad \text{where} \quad \overleftarrow{E}_{i,j} = \begin{cases} 1 & \text{if } j = \beta_i \ , \\ 0 & \text{otherwise} \ . \end{cases} \tag{17}$$

Note that $E \cdot B$ and $\overleftarrow{E}$ are the same, up to a permutation of rows (for i fixed, the reverse of α_i does not generally equal β_i, hence the need for a permutation). Thus, by (15),

$$G'_{\mathrm{rv}} = \overleftarrow{E} \cdot F^{\otimes m} \tag{18}$$

is a generator matrix spanning the same code as G_{rv}. Analogously to (12), our encoder for the reversed code is given by

$$\mathcal{E}_{\mathrm{rv}}(\mathbf{u}) = \mathbf{u} \cdot \underbrace{\overleftarrow{E} \cdot F^{\otimes m} \cdot (\overleftarrow{E})^T}_{\Pi} \cdot \underbrace{\overleftarrow{E} \cdot F^{\otimes m}}_{G'_{\mathrm{rv}}} \ . \tag{19}$$

We now highlight the similarities and differences with respect to the non-reversed encoder. First, note that for the reversed encoder, the set of systematic indices is $\overleftarrow{A}$, as opposed to A for the non-reversed encoder. Apart from that, everything remains the same. Namely, conceptually, we are simply operating the non-reversed encoder with $\overleftarrow{A}$ in place of A. Specifically, note that as in the non-reversed case, the encoder produces a codeword such that the information bits are embedded in the natural order.

Analogously to (13), our aim is to prove that

$$\overleftarrow{E} \cdot F^{\otimes m} \cdot (\overleftarrow{E})^T \cdot \overleftarrow{E} \cdot F^{\otimes m} \cdot (\overleftarrow{E})^T = I \ . \tag{20}$$

IV. Domination contiguity implies involution

In this section we prove that our encoders are valid by proving that (13) and (20) indeed hold. A square matrix is called an *involution* if multiplying the matrix by itself yields the identity matrix. With this terminology at hand, we must prove that both $E \cdot F^{\otimes m} \cdot E^T$ and $\overleftarrow{E} \cdot F^{\otimes m} \cdot (\overleftarrow{E})^T$ are involutions.

Interestingly, and in contrast with the original systematic encoder presented in [4], the proof of correctness centers on the structure of A. That is, in [4], any set of k active (non-frozen) channels has a corresponding systematic encoder. In contrast, consider as an example the case in which $n = 4$ and $A = \{0, 1, 3\}$. By our definitions,

$$E = \left[\begin{smallmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \end{smallmatrix}\right] \ , \quad E^T = \left[\begin{smallmatrix} 1&0&0 \\ 0&1&0 \\ 0&0&0 \\ 0&0&1 \end{smallmatrix}\right] \ , \quad \text{and} \quad F^{\otimes 2} = \left[\begin{smallmatrix} 1&0&0&0 \\ 1&1&0&0 \\ 1&0&1&0 \\ 1&1&1&1 \end{smallmatrix}\right] \ .$$

Thus,

$$E \cdot F^{\otimes 2} \cdot E^T = \left[\begin{smallmatrix} 1&0&0 \\ 1&1&0 \\ 1&1&1 \end{smallmatrix}\right] \ , \quad \text{and}$$
$$(E \cdot F^{\otimes 2} \cdot E^T) \cdot (E \cdot F^{\otimes 2} \cdot E^T) = \left[\begin{smallmatrix} 1&0&0 \\ 0&1&0 \\ 1&0&1 \end{smallmatrix}\right] \ .$$

Note that the rightmost matrix above is *not* an identity matrix. A similar calculation shows that $\overleftarrow{E} \cdot F^{\otimes 2} \cdot (\overleftarrow{E})^T$ is not an involution either.

The apparent contradiction to the correctness of our algorithms will be rectified in the next section. In brief, using terminology defined in Section V, the fact that $A = \{0, 1, 3\}$ implies that W^{+-} is frozen while W^{--} is unfrozen. However, this cannot correspond to a valid polar code since W^{+-} is upgraded with respect to W^{--}.

We now characterize the A for which (13) and (20) hold. Recall our notation for binary representation given in (14). For $0 \leq i, j \leq n$, denote

$$\langle i \rangle_2 = (i_0, i_1, \ldots, i_{m-1}) \ , \quad \langle j \rangle_2 = (j_0, j_1, \ldots, j_{m-1}) \ .$$

We define the *binary domination* relation, denoted $\succeq$, as follows.

$$i \succeq j \quad \text{iff for all } 0 \leq t < m \text{, we have } i_t \geq j_t \ .$$

Namely, $i \succeq j$ iff the support of $\langle i \rangle_2$ (the indices t for which $i_t = 1$) contains the support of $\langle j \rangle_2$.

We say that a set of indices $A \subseteq \{0, 1, \ldots, n-1\}$ is *domination contiguous* if for all $h, j \in A$ and for all $0 \leq i < n$ such that $h \succeq i$ and $i \succeq j$, it holds that $i \in A$. For easy reference:

$$(h, j \in A \quad \text{and} \quad h \succeq i \succeq j) \Longrightarrow i \in A \ . \tag{21}$$

Theorem 1. *Let the active rows set $A \subseteq \{0, 1, \ldots, n-1\}$ be domination contiguous, as defined in (21). Let E and $\overleftarrow{E}$ be defined according to (5), (10), (16), and (17). Then, $E \cdot F^{\otimes m} \cdot E^T$ and $\overleftarrow{E} \cdot F^{\otimes m} \cdot (\overleftarrow{E})^T$ are involutions. That is, (13) and (20) hold.*

Proof. We first note that for $0 \leq i, j < n$, we have that $i \succeq j$ iff $\overleftarrow{i} \succeq \overleftarrow{j}$. Thus, if A is domination contiguous then so is $\overleftarrow{A}$. As a consequence, proving that $E \cdot F^{\otimes m} \cdot E^T$ is an involution will immediately imply that $\overleftarrow{E} \cdot F^{\otimes m} \cdot (\overleftarrow{E})^T$ is an involution as well. Let us prove the former—that is, let us prove (13).

We start by noting a simple characterization of $F^{\otimes m}$, where F is defined as in (3). Namely, the entry at row i and column j of $F^{\otimes m}$ is easily calculated:

$$(F^{\otimes m})_{i,j} = \begin{cases} 1 & i \succeq j \ , \\ 0 & \text{otherwise} \ . \end{cases} \tag{22}$$

To see this, consider the recursive definition of $F^{\otimes m}$ given in (4). Obviously, $(F^{\otimes m})_{i,j}$ equals 0 if we are at the upper right $(n/2) \times (n/2)$ block. That is, if i_{m-1} (the most-significant bit of i) equals 0 and j_{m-1} equals 1. Next, consider the other three blocks and note that for them, $i \succeq j$ iff $i \mod 2^{m-1}$ dominates $j \mod 2^{m-1}$. Since the remaining blocks all contain the same matrix, it suffices to prove the claim for the lower left block. Thus, we continue recursively with $i \mod 2^{m-1}$ and $j \mod 2^{m-1}$.

Recalling (5) and the fact that $|A| = k$, we adopt the following shorthand: for $0 \leq p, q, r < k$ given, let

$$h = \alpha_p \ , \quad i = \alpha_q \ , \quad j = \alpha_r \ .$$

By the above, a straightforward derivation yields that

$$(E \cdot F^{\otimes m} \cdot E^T)_{p,q} = (F^{\otimes m})_{h,i}$$
$$\text{and} \quad (E \cdot F^{\otimes m} \cdot E^T)_{q,r} = (F^{\otimes m})_{i,j} \ .$$

Thus,

$$\begin{aligned} &\Big((E \cdot F^{\otimes m} \cdot E^T) \cdot (E \cdot F^{\otimes m} \cdot E^T) \Big)_{p,r} \\ &\quad = \sum_{q=0}^{k-1} (E \cdot F^{\otimes m} \cdot E^T)_{p,q} \cdot (E \cdot F^{\otimes m} \cdot E^T)_{q,r} \\ &\quad = \sum_{i \in A} (F^{\otimes m})_{h,i} \cdot (F^{\otimes m})_{i,j} \ . \end{aligned} \tag{23}$$

Proving (13) is now equivalent to proving that the right-hand side of (23) equals 1 iff h equals j. Recalling (22), this is equivalent to showing that if $h \neq j$, then there is an even number of $i \in A$ for which

$$h \succeq i \quad \text{and} \quad i \succeq j \ , \tag{24}$$

while if $h = j$, then there is an odd number of such i.

We distinguish between 3 cases.

1) If $h = j$, then there is a single $0 \leq i < n$ for which (24) holds. Namely, $i = h = j$. Since $h, j \in A$, we have that $i \in A$ as well. Since 1 is odd, we are finished with the first case.

2) If $h \neq j$ and $h \not\succeq j$, then there can be no i for which (24) holds. Since 0 is an even integer, we are done with this case as well.
3) If $h \neq j$ and $h \succeq j$, then the support of the binary vector $\langle j \rangle_2 = (j_0, j_1, \ldots, j_{m-1})$ is contained in and distinct from the support of the binary vector $\langle h \rangle_2 = (h_0, h_1, \ldots, h_{m-1})$. A moment of thought reveals that the number of $0 \leq i < n$ for which (24) holds is equal to $2^{w(h)-w(j)}$, where $w(h)$ and $w(j)$ represent the support size of $\langle h \rangle_2$ and $\langle j \rangle_2$, respectively. Since $h \neq j$ and $h \succeq j$, we have that $w(h) - w(j) > 0$. Thus, $2^{w(h)-w(j)}$ is even. Since $h, j \in A$ and A is domination contiguous, all of the above mentioned i are members of A. To sum up, an even number of $i \in A$ satisfy (24), as required. ■

Recall [3, Section X] that an (r, m) Reed-Muller code has length $n = 2^m$ and is formed by taking the set A to contain all indices i such that the support of $\langle i \rangle_2$ has size at least r. Clearly, such an A is domination contiguous, as defined in (21). Hence, the following is an immediate corollary of Theorem 1, and states that our encoders are valid for Reed-Muller codes.

Corollary 2. *Let the active row set A correspond to an (r, m) Reed-Muller code. Let E and $\overleftarrow{E}$ be defined according to (5), (10), (16), and (17), where $n = 2^m$. Then, $E \cdot F^{\otimes m} \cdot E^T$ and $\overleftarrow{E} \cdot F^{\otimes m} \cdot (\overleftarrow{E})^T$ are involutions. That is, (13) and (20) hold and thus our two encoders are valid.*

V. Polar codes satisfy domination contiguity

The previous section concluded with proving that our encoders are valid for Reed-Muller codes. Our aim in this section is to prove that our encoders are valid for polar codes. In order to do so, we first define the concept of a (stochastically) upgraded channel.

A channel W with input alphabet $\mathcal{X}$ and output alphabet $\mathcal{Y}$ is denoted $W : \mathcal{X} \to \mathcal{Y}$. The probability of receiving $y \in \mathcal{Y}$ given that $x \in \mathcal{X}$ was transmitted is denoted $W(y|x)$. Our channels will be binary input, memoryless, and output symmetric (BMS). Binary: the channel input alphabet will be denoted as $\mathcal{X} = \{0, 1\}$. Memoryless: the probability of receiving the vector $(y_i)_{i=0}^{n-1}$ given that the vector $(x_i)_{i=0}^{n-1}$ was transmitted is $\prod_{i=0}^{n-1} W(y_i|x_i)$. Symmetric: there exists a permutation $\pi : \mathcal{Y} \to \mathcal{Y}$ such that that for all $y \in \mathcal{Y}$, $\pi(\pi(y)) = y$ and $W(y|0) = W(\pi(y)|1)$.

We say that a channel $W : \mathcal{X} \to \mathcal{Y}$ is upgraded with respect to a channel $Q : \mathcal{X} \to \mathcal{Z}$ if there exists a channel $\Phi : \mathcal{Y} \to \mathcal{Z}$ such that concatenating Φ to W results in Q. Formally, for all $x \in \mathcal{X}$ and $z \in \mathcal{Z}$,

$$Q(z|x) = \sum_{y \in \mathcal{Y}} W(y|x) \cdot \Phi(z|y) \ .$$

We denote W being upgraded with respect to Q as $W \succeq Q$. As we will soon see, using the same notation for upgraded channels and binary domination is helpful.

Let $W : \mathcal{X} \to \mathcal{Y}$ be a BMS channel. Let $W^- : \mathcal{X} \to \mathcal{Y}^2$ and $W^+ : \mathcal{X} \to \mathcal{Y}^2 \times \mathcal{X}$ be the "minus" and "plus" transform as defined in [3]. That is,

$$W^-(y_0, y_1|u_0) = \frac{1}{2} \sum_{u_1 \in \{0,1\}} W(y_0|u_0 + u_1) \cdot W(y_1|u_1) \ ,$$

$$W^+(y_0, y_1, u_0|u_1) = \frac{1}{2} W(y_0|u_0 + u_1) \cdot W(y_1|u_1) \ .$$

The claim in the following lemma seems to be well known in the community, and is very easy to prove. Still, since we have not found a place in which the proof is stated explicitly, we supply it as well.

Lemma 3. *Let $W : \mathcal{X} \to \mathcal{Y}$ be a BMS channel. Then, W^+ is upgraded with respect to W^-,*

$$W^+ \succeq W^- \ . \tag{25}$$

Proof. We prove that $W^+ \succeq W$ and $W \succeq W^-$. Since "$\succeq$" is easily seen to be a transitive relation, the proof follows. To show that $W^+ \succeq W$, take $\Phi : \mathcal{Y}^2 \times \mathcal{X} \to \mathcal{Y}$ as the channel which maps (y_0, y_1, u_0) to y_1 with probability 1. We now show that $W \succeq W^-$. Recalling that W is a BMS, we denote the corresponding permutation as π. We also denote by $\delta()$ a function taking as an argument a condition. The function δ equals 1 if the condition is satisfied and 0 otherwise. With these definitions at hand, we take

$$\begin{aligned} &\Phi(y_0, y_1|y) \\ &= \frac{1}{2}\big[W(y_1|0) \cdot \delta(y_0 = y) + W(y_1|1) \cdot \delta(y_0 = \pi(y))\big] \ . \end{aligned}$$ ■

This is a good place to note that our algorithm is applicable to a slightly more general setting. Namely, the setting of compound polar codes as presented in [8]. The slight alterations needed are left to the reader.

The following lemma claims that both polar transformations preserve the upgradation relation. It is a restatement of [9, Lemma 4.7].

Lemma 4. *Let $W : \mathcal{X} \to \mathcal{Y}$ and $Q : \mathcal{X} \to \mathcal{Z}$ be two BMS channels such that $W \succeq Q$. Then,*

$$W^- \succeq Q^- \qquad \text{and} \qquad W^+ \succeq Q^+ \tag{26}$$

For a BMS channel W and $0 \leq i < n$, denote by $W_i^{(m)}$ the channel which is denoted "$W_n^{(i+1)}$" in [3]. By [3, Proposition 13], the channel $W_i^{(m)}$ is symmetric. The following lemma ties the two definitions of the $\succeq$ relation.

Lemma 5. *Let $W : \mathcal{X} \to \mathcal{Y}$ be a BMS channel. Let the indices $0 \leq i, j < n$ be given. Then, binary domination implies upgradation. That is,*

$$i \succeq j \quad \Longrightarrow \quad W_i^{(m)} \succeq W_j^{(m)} \ . \tag{27}$$

Proof. We prove the claim by induction on m. For $m = 1$, the claim follows from either (25), or the fact that a channel is upgraded with respect to itself, depending on the case. For $m > 1$, we have by induction that

$$W_{\lfloor i/2 \rfloor}^{(m-1)} \succeq W_{\lfloor j/2 \rfloor}^{(m-1)} \ .$$

Now, if the least significant bits of i and j are the same we use (26), while if they differ we use (25) and the transitivity of the "$\succeq$" relation. ■

We are now ready to prove our second main result.

Theorem 6. *Let A be the active rows set corresponding to a polar code. Then, A is domination contiguous.*

Proof. We must first state exactly what we mean by a "polar code". Let the code dimension k be specified. In [3], A equals the indices corresponding to the k channels $W_i^{(m)}$ with smallest Bhattacharyya parameter, where $0 \leq i < n$. Other definitions are possible and will be discussed shortly. However, for now, let us use the above definition.

Denote the Bhattacharyya parameter of a channel W by $Z(W)$. As is well known, if W and Q are two BMS channels, then

$$W \succeq Q \quad \Longrightarrow \quad Z(W) \leq Z(Q) \; . \tag{28}$$

For a proof of this fact, see [10].

We deduce from (27) and (28) that if $i \succeq j$, then $Z(W_i^{(m)}) \leq Z(W_i^{(m)})$. Assume for a moment that the inequality is always strict when $i \succeq j$ and $i \neq j$. Under this assumption, $j \in A$ must imply $i \in A$. This is a stronger claim then (21), which is the definition of A being domination contiguous. Thus, under this assumption we are done.

The previous assumption is in fact true for all relevant cases, but somewhat misleading: The set A is constructed by algorithms calculating with finite precision. It could be the case that $i \neq j$, $i \succeq j$, but $Z(W_i^{(m)})$ and $Z(W_i^{(m)})$ are approximated by the same number (a tie), or by two close numbers, but in the wrong order. Thus, it might conceptually be the case that j is a member of A while i is not (in practice, we have never observed this to happen). These cases are easy to check and fix, simply by removing j from A and inserting i instead. Note that each such operation enlarges the total Hamming weight of the vectors $\langle t \rangle_2$ corresponding to elements t of A. Thus, such a swap operation will terminate in at most a finite number of steps. When the process terminates, we have by definition that if $j \in A$ and $i \succeq j$, then $i \in A$. Thus, A is dominations contiguous.

Instead of taking the Bhattacharyya parameter as the figure of merit, we could have instead used the (more natural) channel misdecoding probability. That is, the probability of an incorrect maximum-likelihood estimation of the input to the channel given the channel output, assuming a uniform input distribution. Yet another figure of merit we could have taken is the channel capacity. The important point in the proof was that an upgraded channel has a figure of merit value that is no worse. This holds true for the other two options discussed in this paragraph. See [11, Lemma 3] for details and references. We note that although these are natural ways of defining polar codes, they are in some cases sub-optimal. See for example [12] (it is easily proved that our encoder is valid for the scheme in [12] as well). ■

A. Application to Shortened Codes

We end this section by discussing two shortening procedures, [13] and [14], which are compatible with our systematic encoder. Recall that shortening a code at positions $\Gamma \subseteq \{0, 1, \ldots, n-1\}$ means that only codewords $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ for which $\gamma \in \Gamma$ implies $x_\gamma = 0$ are part of the newly created code. Since the value of $\mathbf{x}$ at positions Γ is known to be 0, these codeword positions are not transmitted. Hence, a word of length $n - |\Gamma|$ is transmitted over the channel.

For our purposes, the polar shortening schemes [13] and [14] are very similar. In [13], the set Γ is defined as

$$\Gamma = \{\gamma : \gamma_0 \leq \gamma < n\} \; ,$$

for a given γ_0. The encoding in [13] is of the non-bit-reversed type. In contrast, the set Γ in [14] is obtained from the set Γ above by applying a bit-reversing operation. The encoding in [14] is bit-reversed as well.

An important consequence of the above definition of Γ is the following. In both settings, the shortening is accomplished by freezing the corresponding indices in the information vector. That is, $\gamma \in \Gamma$ implies that $\gamma \notin A$. Also, in both settings, the "channel" corresponding to a position $\gamma \in \Gamma$ (which is not transmitted) is taken as the noiseless channel when constructing the polar code. The rational is that we know with certainty that the value of x_γ is 0.

The applicability of our method to the above follows by two simple observations, which we now state without proof. Firstly, Lemma 3 and its derivatives continue to hold in the setting in which the underlying channels may be of a different type. Specifically, note that at the lowest level, a plus or minus operation may involve a "real" channel and a "noiseless" channel. However, the natural analog of (25) continues to hold. Namely, a plus operation is still upgraded with respect to a minus operation.

The second observation is that $i \succeq j$ as well as $\overleftarrow{i} \succeq \overleftarrow{j}$ imply that $i \geq j$. Thus, in this setting as well, domination contiguity continues to hold. Indeed, consider for concreteness the non-reversed case and suppose to the contrary that (21) does not hold. Namely, we have found $h \succeq i \succeq j$ such that $h, j \in A$ but $i \notin A$. By our first observation, $i \notin A$ must be the result of i being a shortened index, $i \in \Gamma$. But if i is a shortened index, we have by our second observation that h is a shortened index as well. Hence, $h \in \Gamma$ which implies that $h \notin A$, contradiction.

VI. Flexible Hardware Encoders

The encoder discussed in the previous sections uses two instances—or two passes—of a non-systematic polar encoder to calculate the systematic codeword. Therefore it is important to have a suitable non-systematic encoder that provides its output in natural or bit-reversed order.

A semi-parallel non-systematic polar encoder design with a throughput of $\mathcal{P}$ bit/cycle, where $\mathcal{P}$ corresponds to the level of parallelism, was presented in [15]. However, it presents its output in pair bit-reversed order—the output is in bit-reversed order if a pair of consecutive bits is viewed as a single entity,—rendering it unsuitable for use with our systematic encoder. This also poses a problem for parallel and semi-parallel decoders, which expect their input either in natural or in bit-reversed order.

We start this section by presenting the architecture for a new non-systematic encoder that presents its output in natural

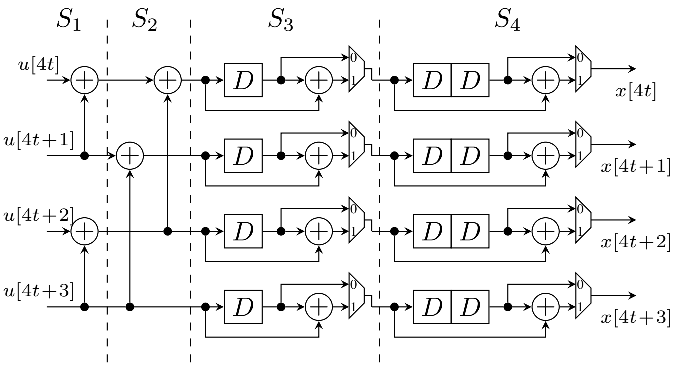


Fig. 2. Architecture of the proposed semi-parallel non-systematic polar encoder with $n = 16$ and $\mathcal{P} = 4$.

order and has the same $\mathcal{P}$-bit/cycle throughput and $n/\mathcal{P}$-cycle latency as [15]. We show the impact of adding length flexibility support, and then utilize it as the core component of a flexible systematic encoder according to the algorithm discussed in this work.

A. Non-Systematic Encoder Architecture

Fig. 2 shows the proposed architecture for a non-systematic encoder with $n = 16$ and $\mathcal{P} = 4$, where stage boundaries are indicated using dashed lines. Each stage S_i, with index i, applies the basic polar transformation to two input bits, $\beta_{i-1}[j]$ and $\beta_{i-1}[j + 2^{i-1}]$ that are 2^{i-1} bits apart in the polar code graph. Since the input pairs to stages with indices $\in [1, \log \mathcal{P}]$ are available in the same $\mathcal{P}$-bit input and the same clock cycle, these stages are implemented using combinational logic only, as shown for S_1 and S_2 in the figure. On the other hand, the two bits processed simultaneously by a stage with an index $i > \log \mathcal{P}$ are not available in the same clock cycle, necessitating the use of delay elements, denoted D in Fig. 2. Such a stage is implemented using $\mathcal{P}$ 1-bit processing elements operating in parallel, each of which has $2^{i-\log \mathcal{P}-1}$ delay elements. A processing element l contains a multiplexer that alternates its output between $\beta_{i-1}[\mathcal{P}t+l-2^{i-1}] \oplus \beta_{i-1}[\mathcal{P}t+l]$ and $\beta_{i-1}[\mathcal{P}t+l]$ every $2^{i-\log \mathcal{P}-1}$ clock cycles, where t is the current cycle index.

The resulting encoder has a throughput of $\mathcal{P}$ bit/s, a latency of $n/\mathcal{P}$ cycles, and a critical path that passes from $u[4t+4]$ to $x[4t]$, similar to the encoder of [15]. The critical path can be shortened by inserting pipeline registers at stage boundaries, increasing latency in terms of cycles, but leaving throughput per cycle unaffected. In addition to the output order, the proposed architecture has another advantage over [15] in that it can be used to implement a fully serial encoder with $\mathcal{P} = 1$, whereas that of [15] can only scale down to $\mathcal{P} = 2$. We note that throughout this work, encoding latency is measured from first data-in to first data-out and all encoders start their operation as soon as the first $\mathcal{P}$ input bits are available.

B. Flexible Non-Systematic Encoder

Since the input $\mathbf{u}$ is assumed to contain '0's in the frozen bit locations, the proposed encoder is rate flexible as the input preprocessor can change the location and number of frozen bits without affecting the encoder architecture.

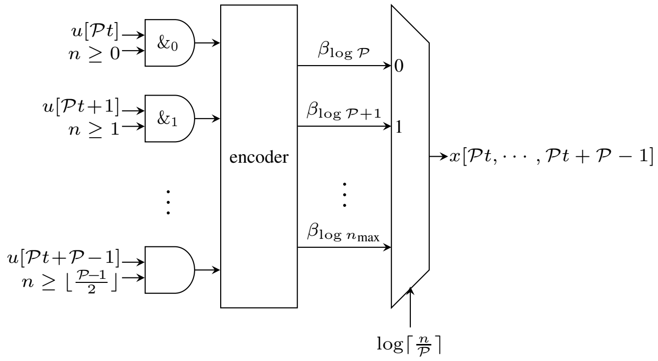

$u[\mathcal{P}t]$
$n \geq 0$
$\&_0$
$u[\mathcal{P}t+1]$
$n \geq 1$
$\&_1$
$u[\mathcal{P}t+\mathcal{P}-1]$
$n \geq \lfloor \frac{\mathcal{P}-1}{2} \rfloor$
encoder
$\beta_{\log \mathcal{P}}$
$\beta_{\log \mathcal{P}+1}$
$\beta_{\log n_{\max}}$
0
1
$x[\mathcal{P}t, \cdots, \mathcal{P}t + \mathcal{P} - 1]$
$\log\lceil \frac{n}{\mathcal{P}} \rceil$

Fig. 3. Flexible encoder with maximum code length $n_{\max}$ and parallelism $\mathcal{P}$.

Adapting this architecture to encode any polar code of length $n \leq n_{\max}$ requires extracting data from different stage outputs—indicated using the dashed lines in Fig. 2—in the encoder. The output for a code of length n can be extracted from location $S_{\log n}$ using $\mathcal{P}$ instances of a $\log n_{\max} \times 1$ multiplexer.

The width of the multiplexer can be reduced to $\log n_{\max} - \log \mathcal{P}$ without affecting decoding latency by exploiting the combinational nature of stages $\in [S_1, S_{\log \mathcal{P}}]$ and setting inputs with indices $i > n-1$ to '0'. The modified encoder architecture is illustrated in Fig. 3, where the block labeled 'encoder' is the non-systematic encoder shown in Fig. 2. The AND gates are used to mask inputs when $n < \mathcal{P}$. The first AND gate, $\&_0$, will always have it second input set to '1' in this case and will be optimized away by the synthesis tool. It is shown in the figure because it will be used to implement code shortening as described in Section VI-F.

C. Non-Systematic Encoder Implementation

Both the rate-flexible and rate and length-flexible versions of the proposed non-systematic encoder were implemented on the Altera Stratix IV EP4SGX530KH40C2 FPGA. We also implemented the encoder of [15] on the same FPGA for comparison even though its output order is not suitable for implementing the proposed systematic encoder. All decoders have a latency of 512 cycles and coded throughput of 32 bits/cycle for $n_{\max} = 16384$ and $\mathcal{P} = 32$. Table I presents these implementation results, where the proposed rate-flexible and the rate and length-flexible encoders are denoted R-flexible and Rn-flexible, respectively. From the table it can be observed that including length flexibility increases the logic requirements of the design by 27% due to the extra routing required. It also decreases the maximum achievable frequency, and in turn throughput, by 14%. The latency of the decoders is bounded by $n_{\max}/\mathcal{P}$ and increasing $\mathcal{P}$ to 64 reduced it to 0.74 and 0.82 μs, increasing the throughput to 22 and 20 Gbps, for the R- and Rn-flexible encoders, respectively. The maximum achievable frequency decreased to 344 and 313 MHz for the two encoders.

TABLE I
IMPLEMENTATION OF THE PROPOSED R-FLEXIBLE AND Rn-FLEXIBLE NON-SYSTEMATIC ENCODERS COMPARED WITH THAT OF [15] FOR $n_{\max} = 16384$ AND $\mathcal{P} = 32$ ON THE ALTERA STRATIX IV EP4SGX530KH40C2.

Decoder	LUTs	FF	RAM (bits)	f (MHz)	Lat. (μs)	T/P (Gbps)
[15]	769	1,392	12,160	354	1.4	11.3
R-flexible	649	1,240	12,160	394	1.3	12.6
Rn-flexible	838	1,293	12,160	360	1.4	11.5

The results were obtained using Altera Quartus II 15.0 and verified using both RTL and gate-level simulation with randomized testbenches.

D. Systematic Encoder Architecture

With a non-systematic encoder providing its output in a suitable order, we now present the architecture and implementation results for the proposed systematic encoder. As proved in Sections III and IV, the proposed systematic encoder can present its output with parity bits in bit-reversed or natural order locations—even with the non-systematic encoder providing its output in natural order—by changing the location of the frozen bits. We therefore use an $n_{\text{max}}/\mathcal{P} \times \mathcal{P}$-bit memory to store the frozen-bit mask, enabling the encoder to support both parity-bit locations, in addition to rate flexibility.

As mentioned in Section II-B, the systematic-encoding process performs two non-systematic encoding passes on the data. These passes can be implemented using two instances of the proposed non-systematic encoder. The output of the first is stored in registers and then masked according the content of the mask memory before being passed to another level of pipeline registers to limit the critical path length. The output of the registers is then passed to the second non-systematic encoder instance, whose output forms the systematic codeword. Such an architecture has the same $\mathcal{P}$-bit/cycle throughput of the component non-systematic encoder with a latency $\mathcal{L} = 2\mathcal{L}_{\text{NS}} + 2$ cycles, where $\mathcal{L}_{\text{NS}}$ is the latency of the non-systematic encoder.

Alternatively, to save implementation resources at the cost of halving the throughput, one instance of the component encoder can be used for both passes. The output of the non-systematic encoder is stored in registers after the first pass and is routed back to the input of the encoder. The systematic codeword becomes available after the second pass.

The systematic encoder of [4] can be used in a configuration similar to the proposed high-throughput one. However, it requires multiplication by matrices that change when the frozen bits are changed. Therefore, its implementation requires a configurable parallel matrix multiplier that is significantly more complex than the component non-systematic encoder used in this work. When the encoder of [4] is implemented to be rate-flexible and low-complexity, it has a latency of at least n clock cycles; compared to the $2n/\mathcal{P} + 2$ cycle latency of the proposed architecture.

TABLE II
IMPLEMENTATION OF THE PROPOSED R-FLEXIBLE AND Rn-FLEXIBLE SYSTEMATIC ENCODERS FOR $n_{\max} = 16384$ AND $\mathcal{P} = 32$ ON THE ALTERA STRATIX IV EP4SGX530KH40C2.

Decoder	LUTs	FF	RAM (bits)	f (MHz)	Lat. (μs)	T/P (Gbps)
Non-Pipelined						
R-flexible	1,442	2,320	36,924	206	5.0	6.6
Rn-flexible	1,782	2,381	36,924	180	5.7	5.7
Pipelined						
R-flexible	1,397	2,639	36,924	282	3.6	9.0
Rn-flexible	1,606	2,742	36,924	264	3.9	8.4

TABLE III
IMPLEMENTATION OF THE PROPOSED Rn-FLEXIBLE SYSTEMATIC ENCODER FOR DIFFERENT $n_{\max}$ AND $\mathcal{P}$ VALUES ON THE ALTERA STRATIX IV EP4SGX530KH40C2.

n_{max}	$\mathcal{P}$	LUTs	FF	RAM (bits)	f (MHz)	Lat. (μs)	T/P (Gbps)
16,384	32	1,606	2,742	36,924	264	3.9	8.4
16,384	64	2,872	5,287	16,384	235	2.2	15.0
16,384	128	4,404	8,304	16,384	272	0.9	34.8
32,768	32	1,971	2,997	85,948	258	7.9	8.2
32,768	64	3,390	5,601	64,200	265	3.9	16.9
32,768	128	5,550	10,024	37,304	234	2.2	29.9

E. Systematic Encoder Implementation

Implementation results of the throughput oriented R-flexible and Rn-flexible encoders are presented in Table II both with and without pipeline registers in between the two non-systematic encoder instances. In the pipelined version two levels were used: one before and one after the masking operation, since memory access incurred a comparatively long delay. The results show that the pipelined version performs significantly faster than the non-pipelined version, where the clock frequency was increased by 80 MHz for both pipelined encoders and was limited by clock and asynchronous reset distribution. The pipelining yielded throughput values of 9 and 8.4 Gbps for the R-flexible and Rn-flexible encoders, respectively. The reported amount of RAM included the mask memory, in addition to operations that were converted automatically by the synthesis and mapping tools.

As in the case of the non-systematic encoder, the throughput is proportional to $\mathcal{P}$ and the latency to $n/\mathcal{P}$. Table III explores the effect of different n_{max} and $\mathcal{P}$ values on the pipelined Rn-flexible encoder. Throughput in excess of 10 Gbps is achievable by the encoder when $\mathcal{P} > 32$. When $n < n_{\text{max}}$, throughput remains unchanged and latency decreases to n/n_{max} of its original value. For example, when the encoder with $n_{\text{max}} = 16384$ and $\mathcal{P} = 64$ encodes a code with $n = 2048$, throughput remains 15 Gbps and latency decreases to 281 ns.

F. On Code Shortening

As discussed in Subsection V-A, the works in [13] and [14] describe shortening schemes for polar codes in which the last $n - n_s$ information bits in a polar code of length n are replaced with '0's. Those bits are discarded from the

TABLE IV
IMPLEMENTATION OF THE PROPOSED Rn-FLEXIBLE SYSTEMATIC ENCODER WITH SHORTENING ON THE ALTERA STRATIX IV EP4SGX530KH40C2.

n_{max}	$\mathcal{P}$	LUTs	FF	RAM (bits)	f (MHz)	Lat. (μs)	T/P (Gbps)
16,384	128	4,518	8,667	16,384	272	0.9	34.8

TABLE V
LATENCY AND CODED THROUGHPUT OF A SOFTWARE SYSTEMATIC ENCODER WITH $n = 32,768$ AND DIFFERENT $\mathcal{P}$ VALUES RUNNING ON AN INTEL CORE I7-2600.

$\mathcal{P}$	8	16	32	64	128	256
Latency (μs)	64.1	30.1	14.3	7.7	4.1	3.3
T/P (Gbps)	0.5	1.1	2.3	4.2	8.0	10.0

systematic codeword before transmission. The result is that n_s bits containing k_s information bits are transmitted; where $k_s = k - (n - n_s)$. These schemes are suitable for use with the proposed systematic encoder, yielding a system that can encode normal and shortened polar codes of any length $n \in [2, n_{\text{max}}]$ without any other constraints on the code length or rate.

To adapt our proposed systematic encoder and enable shortening, the second input, en$_i$, to the AND gates $\&_i$ becomes

$$\text{en}_i = \begin{cases} 1 & \text{when } n \geq \lfloor (\mathcal{P}t + i)/2 \rfloor, \\ 0 & \text{otherwise.} \end{cases}$$

Adding code shortening ability has a minor effect on the resource utilization of the Rn-flexible encoder as can be observed in Table IV.

VII. FLEXIBLE SOFTWARE ENCODERS

In this section, we present a software implementation of our systematic encoder using single-instruction multiple-data (SIMD) operations. We use both AVX (256-bit) and SSE (128-bit) SIMD extensions, in addition to the built-in types `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t` to operate on multiple bits simultaneously. The width of the selected type determines the encoder parallelism parameter $\mathcal{P}$, e.g. $\mathcal{P} = 8$ for `uint8_t`.

The component non-systematic encoder progresses from stage S_1 to $S_{\log n}$ and presents its output in natural order. The input to S_1 is a packed vector where bits corresponding to frozen locations are set to '0' and information bits are stored in the other locations. The bit with index t at the output of a stage S_i is calculated according to:

$$\beta_i[t] = \begin{cases} \beta_{i-1}[t - 2^{i-1}] \oplus \beta_{i-1}[t] & \text{when } \lfloor t/2^{i-1} \rfloor \text{ is even,} \\ \beta_{i-1}[t] & \text{otherwise.} \end{cases} \tag{29}$$

This operation is applied directly to $\mathcal{P}$ bits simultaneously in stage S_i if $2^{i-1} \geq \mathcal{P}$. However, since we can only read and write data in groups of $\mathcal{P}$ bits whose addresses are aligned to $\mathcal{P}$-bit boundaries, operations in stages S_i with $2^{i-1} < \mathcal{P}$ are performed using a mask-shift-XOR procedure. A $\mathcal{P}$-bit mask m_i is generated for each stage $\in [S_1, S_{\log n}]$ so that:

$$m_i[t] = \begin{cases} 0 & \text{when } \lfloor t/2^{i-1} \rfloor \text{ is even,} \\ 1 & \text{otherwise.} \end{cases}$$

The output for these stages is calculated using:

$$\begin{aligned} \beta_i[t : t + \mathcal{P} - 1] =& \beta_{i-1}[t : t + \mathcal{P} - 1] \oplus \\ & ((\beta_{i-1}[t : t + \mathcal{P} - 1] \mathbin{\&} m_i) >> 2^{i-1}). \end{aligned}$$

The index t starts at 0 and is incremented by $\mathcal{P}$ with a final value of $N - \mathcal{P}$. The group of $\mathcal{P}$ bits with indices $\in [t, t + \mathcal{P} - 1]$ is denoted $t : t + \mathcal{P} - 1$. The symbol & is the bit-wise binary AND operation, and >> is the logical bit right shift operator.

Since SSE operations lack bit shift operations, but include byte shifts, operations for stages S_1, S_2, S_3 are performed using the `uint64_t` native type in the proposed software encoder. AVX version 1 does not provide any shift operations, and version 2 can only perform byte-shifts within 128-bit lanes. Therefore, we use SSE instructions until stage S_9, where the encoder switches to using AVX operations. The masking operation between the two non-systematic encoding passes is applied using $\mathcal{P}$-bit operations and masks.

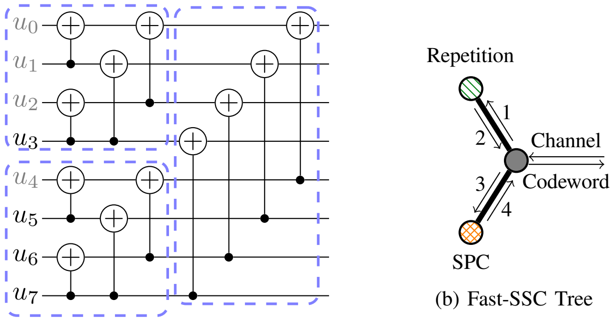
The resulting software systematic encoder operates on data in-place and requires n bits of additional memory to store the frozen-bit mask, and another $\mathcal{P} \log \mathcal{P}$ bits to store the stage masks. The latency and coded throughput values for the proposed software systematic encoder running on a 3.4 GHz Intel Core i7-2600 are shown in Table V for $n = 32,768$. $\mathcal{P}$ was varied between 8 and 256. It can be seen that the latency decreases linearly with increasing $\mathcal{P}$ until $\mathcal{P} = 128$. The latency only decreases by 20% between the SSE (128-bit) and AVX (256-bit) encoders for two reasons: the use of SSE for stages to up S_9 in the AVX encoder, and the overhead of loops and conditionals in the encoder. As a result, an encoder specialized for a given n value is expected to be faster.

The speed results indicate that even an embedded 8-bit micro-processor running 1000 times slower would still be capable of transmissions at 500 kbps, eliminating the need for a dedicated hardware encoder for many applications such as remote sensors and some internet of things devices.

VIII. FLEXIBLE HARDWARE DECODERS

We complete the flexible hardware polar coding system in this section by presenting flexible, systematic hardware decoder, which can decode channel messages based on the codewords generated by the encoder presented in Section VI. As discussed in [5], it is important that the parity bits be in bit-reversed locations to reduce routing complexity and simplify memory accesses.

The original Fast-SSC decoder was capable of decoding all polar codes of a given length: it resembled a processor where the polar code is loaded as a set of instructions [5]. In this section, we review the Fast-SSC algorithm, describe the architectural modifications necessary to decode any polar code up to a maximum length n_{max} and analyze the resulting implementation.



(a) Code Graph

(b) Fast-SSC Tree

Fig. 4. The graph of an (8, 4) polar code and its corresponding Fast-SSC tree representation.

A. The Fast-SSC Decoding Algorithm

Our proposed flexible decoders utilize the Fast-SSC presented in [5]. The polar code is viewed as a tree that corresponds to the recursive nature of polar code construction: a polar code of length n is the concatenation of two polar codes of length $n/2$. In the successive cancellation decoding, the tree is traversed depth first starting from stage $S_{\log n}$ until leaf node in stage S_0, corresponding to a constituent code of length $n = 1$ is reached. At that point, the output of is '0' if the leaf node corresponds to a frozen bit. Otherwise, it is calculated from the input log likelihood ratio (LLR) based on threshold detection. The SSC decoding algorithm, [16], directly decodes constituent codes of any length that are of rate 0 or rate 1 without traversing their sub-trees. The Fast-SSC algorithm directly decodes single parity check (SPC) and repetition codes, in addition to rate-0 and rate-1 codes, of any length. Fig. 4 shows the Fast-SSC tree of an (8, 4) polar code, where the flow of messages is indicated by the arrows.

B. Stage Indices and Sizes

The Fast-SSC decoder in [5] starts decoding a polar code of length $n_{\max}$ at stage $S_{\log n_{\max}}$; where a stage S_i corresponds to a constituent polar code of length 2^i, as discussed in Section VIII-A. Since the decoder uses a semi-parallel architecture, the length of the constituent code is used to determine the number of memory words associated with a stage. The simplest method for that decoder to decode a code of length $n \leq n_{\max}$ is to store the channel LLRs in the memory associated with stage S_n and start the decoding process from there. This, however, requires significant routing resources since the architecture presented in [5] separates the channel and internal LLRs into different memories for performance reasons.

In the proposed flexible decoder, we calculate the length, n_v, of the constituent code associated with a stage S_i as function of i, n, and $n_{\max}$:

$$n_v(S_i) = 2^i \frac{n}{n_{\max}}. \tag{30}$$

The memory allocated for a stage S_i remains the same regardless of n and is always calculated assuming $n = n_{\max}$. The flexible decoder always starts from $S_{\log n_{\max}}$, corresponding to a polar code of length $n \leq n_{\max}$, performing operations on $n_v(S_i)/(2\mathcal{P})$ LLR values at a stage S_i, and proceeds until it encounters a constituent code whose output can be directly estimated according to the rules of the Fast-SSC algorithm.

TABLE VI
IMPLEMENTATION OF THE Rn-FLEXIBLE POLAR DECODER COMPARED TO THE R-FLEXIBLE DECODER OF [5] ON THE ALTERA STRATIX IV EP4SGX530KH40C2.

Decoder	n	$\mathcal{P}$	LUTs	FF	RAM (bits)	f (MHz)
[5]	2048	64	6315	1608	50,072	102
Proposed	2048	64	6507	1600	50,072	102
w/ Shortening	2048	64	6451	1613	52,120	102
[5]	32,768	256	24,066	7,231	536,136	102
Proposed	32,768	256	23,583	7,207	536,136	102
w/ Shortening	32,768	256	23,593	7,219	568,904	102

C. Implementation Results

Since memory is accessed as words containing multiple $2\mathcal{P}$ LLR or bit-estimate values, the limits used to determine the number of memory words per stage must be changed to accommodate the new n value. The rest of the decoder implementation remains unchanged from [5]. These limits are now calculated according to (30) and using the n value provided to the decoder as an input.

Table VI compares the proposed flexible decoder ($n_{\max} = 32,768$) with the Fast-SSC decoder of [5] ($n = 32,768$) when both are implemented using the Altera Stratix IV EP4SGX530KH40C2 FPGA. The resource requirements are also provided for $n_{\max} = n = 2048$. It can be observed that the change in resource utilization is negligible as a result of the localized change in limit calculations.

When decoding a code of length $n < n_{\max}$, the flexible decoder has the same latency (in clock cycles) as the Fast-SSC decoder for a code of length n. Since our Rn-flexible decoder implementation has the same operating clock frequency as the R-flexible decoder, it also has the same throughput and latency (in time). We note that the decoders presented in this work contain an additional input buffer to store an incoming channel vector while one is being decoded. This is done to enable loading-while-decoding and allows the decoder to sustain its throughput.

The implementation results for Rn-flexible decoder supporting code shortening are also included in Table VI. The main change is the requirement for $n_{\max}$ more bit of RAM. This is a consequence of shortening being implemented using masking where the LLRs corresponding to shortened bits are replaced with the maximum LLR value based on an $n_{\max}$-bit mask that is stored in said memory.

IX. Flexible Software Decoders

High-throughput software decoders require vectorization using SIMD instructions in addition to a reduction in the number of branches. However, these two considerations significantly limit the flexibility of the decoder to the point where the lowest latency decoders in literature are compiled for a single polar code [17]. In this section, we present a software

Fast-SSC decoder balancing flexibility and decoding latency. The proposed decoder has 37% higher latency than a fully specialized decoder, but can decode any polar code of length $n \leq n_{\text{max}}$. As will be discussed later in this section, there are two additional advantages to the proposed flexible software decoder: the resulting executable size is an order of magnitude smaller, and it can be used to decode very long polar codes for which an unrolled decoder cannot be compiled. Since SIMD instructions operate mostly ‘vertically’ on data stored in different vectors, natural indexing is preferable to reversed one; in contrast to the hardware decoders.

A. Memory

Unlike in hardware decoders, it is simple to access an arbitrary memory location in software decoders. The LLR memory in the proposed software decoder is arranged into stages according to constituent code sizes. When a code of length $n \leq n_{\text{max}}$ is to be decoded, the channel LLRs are loaded into stage $S_{\log n}$, bypassing any stages with a larger index.

When backtracking through the code tree towards stages with high indices, the decoder performs the same operations on bit-estimates as the encoder—namely, binary addition and copying, depending on the index of the output bit in question. Storing the bit-estimates in a one-dimensional array of length n_{max} bits enables the decoder to only perform the binary addition and store its results, eliminating superfluous copy operations and decreasing latency [17]. For a code of length $n \leq n_{\text{max}}$, the decoder writes its estimates starting from bit index 0. Once decoding is completed, the estimated codeword will occupy the first n bits of the bit-estimate memory, which are provided as the decoder output.

B. Vectorization

The unrolled software decoder [17] specifies input sizes for each command at compile time. This enables SIMD vectorization without any loops, but limits the decoder to a specific polar code. To efficiently utilize SIMD instructions while minimizing the number of loops and conditionals, we employ dynamic dispatch in the proposed decoder. Each decoder operation in implemented, using SIMD instructions and C++ templates, for all stage sizes up to n_{max}. These differently sized implementations are stored in array indexed by the logarithm of the stage size. Therefore two branch operations are used: the first to look up the decoding operation, and the second to look up the correct size of that operation. This is significantly more efficient than using loops over the SIMD word size.

C. Results

We compare the latency of the proposed vectorized flexible decoder with a non-vectorized version and with the fully unrolled decoder of [17] using floating-point values.

Table VII compares the proposed flexible, vectorized decoder with a flexible, non-explicitly-vectorized decoder (denoted ‘scalar’) and a fully unrolled (denoted ‘unrolled’) one running on an Intel Core i7-2600 with AVX extensions. All decoders were decoding a (32768, 29492) polar code using the Fast-SSC algorithm, floating-point values, and the min-sum approximation. The flexible decoders had $n_{\text{max}} = 32,768$. From the results in the table, it can be seen that the vectorized decoder has 42.6% the latency (or 2.3 times the throughput) of the scalar version. Compared to the code-specific unrolled decoder, the proposed decoder has 137% the latency (or 73% the throughput). In addition to the two layers of indirection in the proposed decoder, the lack of inlining contributes to this increase in latency. In the unrolled decoder, the entire decoding flow is known at compile time, allowing the compiler to inline function calls, especially those related to smaller stages. This information is not available to the flexible decoder.

TABLE VII
SPEED OF THE PROPOSED VECTORIZED DECODER COMPARED WITH THAT OF NON-VECTORIZED AND FULLY-UNROLLED DECODERS WHEN $n = n_{\text{MAX}} = 32768$ AND $k = 29492$.

Decoder	Latency (μs)	Info. Throughput (Mbps)
Scalar Fast-SSC	256	115
Unrolled Fast-SSC [17]	109	270
Proposed Fast-SSC	149	198

TABLE VIII
SPEED OF THE PROPOSED VECTORIZED DECODER COMPARED WITH THAT OF NON-VECTORIZED AND FULLY-UNROLLED DECODERS FOR A (2048, 1723) CODE AND $n_{\text{MAX}} = 32768$.

Decoder	Latency (μs)	Info. Throughput (Mbps)
Scalar Fast-SSC	16.2	106
Unrolled Fast-SSC [17]	4.0	430
Proposed Fast-SSC	11.1	155

Results for $n < n_{\text{max}}$ are shown in Table VIII where $n_{\text{max}} = 32,768$ for the flexible decoders and the code used was a (2048, 1723) polar code. The advantage the vectorized decoder has 68% the latency of the non-vectorized decoder. The gap between the proposed decoder and the unrolled one increases to 2.8 times the latency. These decrease in relative performance of the proposed decoder is a result of using a shorter code where a smaller proportion of stage operations are inlined.

In addition to decoding different codes, the proposed flexible decoder has an advantage over the fully unrolled one in terms of resulting executable size and the maximum length of the polar code to be decoded. The size of the executable corresponding to the proposed decoder with $n_{\text{max}} = 32,768$ was 0.44 MB with 3 KB to store the polar code instructions in an uncompressed textual representation; whereas that of the unrolled decoder was 3 MB. In terms of polar code length, the GNU C++ compiler was unable to compile an unrolled decoder for a code of length 2^{24} even with 32 GB of RAM; while the proposed decoder did not exhibit any such issues.

X. CONCLUSION

In this work, we studied the flexibility in code rate and length of polar encoders and decoders. We proved the correctness of a flexible, parallelizeable systematic polar encoding algorithm and used it to implement high-speed, low-complexity hardware encoders with throughput up to 29 Gbps on FPGA.

The proof of correctness was provided not only for polar, but also for Reed-Muller codes. Software encoders were also presented and shown to achieve throughput up to 10 Gbps. We demonstrated rate and length flexible hardware decoders that had similar implementation complexity and the same speed as their rate-only flexible counterparts. Finally, we introduced software decoders that are flexible and able to achieve 73% the throughput of their unrolled, code-specific counterparts.

References

[1] J.-Y. Lee and H.-J. Ryu, “A 1-Gb/s flexible LDPC decoder supporting multiple code rates and block lengths,” *IEEE Trans. Consum. Electron.*, vol. 54, no. 2, pp. 417–424, May 2008.

[2] C. Condo, M. Martina, and G. Masera, “VLSI implementation of a multi-mode turbo/LDPC decoder architecture,” *IEEE Trans. Circuits Syst. I*, vol. 60, no. 6, pp. 1441–1454, June 2013.

[3] E. Arıkan, “Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels,” *IEEE Trans. Inf. Theory*, vol. 55, no. 7, pp. 3051–3073, 2009.

[4] ——, “Systematic polar coding,” *IEEE Commun. Lett.*, vol. 15, no. 8, pp. 860–862, 2011.

[5] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, “Fast polar decoders: Algorithm and implementation,” *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 946–957, May 2014.

[6] L. Li, W. Zhang, and Y. Hu, “On the error performance of systematic polar codes,” *CoRR*, vol. abs/1504.04133, 2015. [Online]. Available: http://arxiv.org/abs/1504.04133

[7] L. Li and W. Zhang, “On the encoding complexity of systematic polar codes,” in *Int. System-on-Chip Conf. (IEEE-SOCC)*, September 2015.

[8] H. Mahdavifar, M. El-Khamy, J. Lee, and I. Kang, “Compound polar codes,” in *Information Theory and Applications Workshop (ITA), 2013*, Feb 2013, pp. 1–6.

[9] S. B. Korada, “Polar codes for channel and source coding,” Ph.D. dissertation, EPFL, 2009.

[10] T. Kailath, “The divergence and Bhattacharyya distance measures in signal selection,” *IEEE Trans. Commun. Technol.*, vol. 15, no. 1, pp. 52–60, February 1967.

[11] I. Tal and A. Vardy, “How to construct polar codes,” *IEEE Trans. Inf. Theory*, vol. 59, no. 10, pp. 6562–6582, Oct 2013.

[12] M. Mondelli, S. Hassani, and R. Urbanke, “From polar to Reed-Muller codes: A technique to improve the finite-length performance,” *IEEE Trans. Commun.*, vol. 62, no. 9, pp. 3084–3091, Sept 2014.

[13] Y. Li, H. Alhussien, E. Haratsch, and A. Jiang, “A study of polar codes for MLC NAND flash memories,” in *Int. Conf. on Comput., Netw. and Commun. (ICNC)*, Feb 2015, pp. 608–612.

[14] R. Wang and R. Liu, “A novel puncturing scheme for polar codes,” *Communications Letters, IEEE*, vol. 18, no. 12, pp. 2081–2084, Dec 2014.

[15] H. Yoo and I.-C. Park, “Partially parallel encoder architecture for long polar codes,” *IEEE Trans. Circuits Syst. II*, vol. 62, no. 3, pp. 306–310, March 2015.

[16] A. Alamdar-Yazdi and F. R. Kschischang, “A simplified successive-cancellation decoder for polar codes,” *IEEE Commun. Lett.*, vol. 15, no. 12, pp. 1378–1380, 2011.

[17] P. Giard, G. Sarkis, C. Leroux, C. Thibeault, and W. J. Gross, “Low-latency software polar decoders,” *CoRR*, vol. abs/1504.00353, 2015. [Online]. Available: http://arxiv.org/abs/1504.00353