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—The capacity-achieving property of polar codes has garnered much recent research attention resulting in low-complexity and high-throughput hardware and software decoders. It would be desirable to implement flexible hardware for polar encoders and decoders that can implement polar codes of different lengths and rates, however this topic has not been studied in depth yet. Flexibility is of significant importance as it enables the communication system to adapt to varying channel conditions and meet different transmission latency requirements and is mandated in most communication standards. 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 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]. In this work, we demonstrate that encoders and decoders for polar codes can be made length and rate flexible while maintaining low-complexity implementations and high speed. There has been much recent interest in polar codes, which achieve the symmetric capacity of memoryless channels with an explicit construction and are decoded with the low-complexity successive-cancellation decoding algorithm [3]. Several implementations of polar decoders were presented in literature, some of which are capable of decoding polar codes of different rates given a fixed code length [4], [5]. We aim to extend this flexibility so that the decoders, and encoders, can operate on any polar code of length \( n \leq n_{\text{max}} \).

Polar codes were initially introduced as non-systematic block codes [3]. Later, systematic polar coding was described in [6] as a method to ease information extraction and improve bit-error rate without affecting the frame-error rate. In addition to the error-rate improvement, systematic polar codes were shown to be well suited for use with the fast decoding algorithm introduced in [4].

The systematic encoding scheme originally proposed in [6] 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) places a speed limit on the encoding process which gets worse with increasing code length. In contrast, the non-systematic encoder presented in [3] is parallel by nature, and is amenable to very fast hardware implementations [7]. To address this, a new systematic encoding algorithm that is easy to parallelize was first described in [4]. This new encoding algorithm offers the best of both worlds: on one hand, it is systematic, and thus gains all the advantages described above. On the other hand, it is essentially equivalent to running the non-systematic encoder twice. Thus, the prior art (and future advances) used to implement fast non-systematic encoders can be used as is to implement a fast systematic encoder. We further highlight that the systematic encoder in [4] is very flexible: it can encode any polar code of a given length by simply updating bit masks stored in memory, without any other modifications to the implementation.

A polar code is defined by the set of its so-called frozen indices (we assume the freezing is always to '0'). For a given underlying channel and a specified rate, one choice of the frozen index set can result in a very good code while another choice can result in a very bad code. For certain problematic frozen index sets, the systematic encoder presented in [4] is known not to work (i.e. not produce valid polar codewords). However, it was observed in [4] that for all the instances considered, such problematic frozen sets did not occur in practice. That is, when one chose the frozen index set with the aim of obtaining the polar code with the best error-correction performance, it was speculated in [4] that this phenomenon is true in general, but no proof was given. A key result of this paper is such a proof. We show that if ties are broken in a specific way during the polar code construction phase, then the systematic encoder in [4] will always be correct and produce valid codewords. The result follows from Theorems 1 and 6.
Theorem 1. This is given as [9, Proposition 3].

The closest analog in our paper to these is what one can deduce 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 \( II \) 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 [4] and [6] 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 [6]

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

\[
F = \begin{bmatrix}
1 & 0 \\
1 & 1
\end{bmatrix}.
\]

The \( m \)-th Kronecker product of \( F \) is denoted \( F^\otimes m \) and is defined recursively as

\[
F^\otimes m = \begin{bmatrix}
F^\otimes (m-1) & 0 \\
F^\otimes (m-1) & F^\otimes (m-1)
\end{bmatrix},
\]

where \( F^\otimes 1 = F \).

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 [6] 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

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

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 $\Pi$ 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 [6], which results in an encoding algorithm running in time $O(n \cdot \log n)$. Let us recall our goal, we must find a codeword $x = (x_0, x_1, \ldots, x_{n-1})$ such that, using the notation in (5), we have that $x_{\alpha_i}$ equals $u_i$. Since $x$ 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 $x$ as described above such that $x = v \cdot F^{\otimes m}$, where $v = (v_0, v_1, \ldots, v_{n-1})$ is such that $v_i = 0$ whenever $i \notin A$.

We will now show a recursive implementation to the encoding function $\text{Encode}_m(u, A)$. Let us start by considering the stopping condition, $m = 0$. As a preliminary step, define $F^{\otimes 0} = 1$, a 1 x 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, we are forced to have $v = (0)$, and thus $\text{Encode}_0(u, A)$ returns $x = (0)$, the all-zero codeword. Otherwise, $A = \{0\}$ and we simply take $v = (u_0)$ and return $x = (u_0)$, as prescribed.

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

We will find $x$ by first finding $x''$ and then finding $x'$. 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 $x''$ is a straightforward recursive process. Namely, by (6) if we define
\[ u'' = (u_{i+n/2})_{i \in A''}, \]

then $x'' = \text{Encode}_{m-1}(u', A'')$. Now, with $x''$ calculated, we can find $x'$. Namely, considering (7), we need a $v'$ for which entry $\alpha$ of $v' \cdot F^{\otimes (m-1)}$ equals $u_i + x_{\alpha}$. Thus, defining $u' = (u_i + x_{\alpha})_{i \in A'}$, we have that $x' = \text{Encode}_{m-1}(u', A')$.

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

B. The systematic encoder [4]

We now give a high-level description of the encoder in [4]. As before, we consider a non-reversed setting. Recall that $A$ in (5) is the set of active row indices.

1) We first expand $u = (u_0, u_1, \ldots, u_{k-1})$ into a vector $v_1$ of length $n$ as follows: for all $0 \leq i < k$ we set entry $\alpha_i$ of $v_1$ equal to $u_i$. The remaining $n - k$ entries of $v_1$ are set to 0.

2) We calculate $v_{11} = v_1 \cdot F^{\otimes m}$.

3) The vector $v_{11}$ is gotten from $v_{11}$ by setting all entries not in $A$ to zero.

4) We return $x = v_{11} \cdot F^{\otimes m}$.

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

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

We also note that the systematic encoder in [4] 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 [6] as well as the one presented in [4] 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 $k$. On the other hand, by definition, both encoders produce the same output when restricted to the $k$ indexes $A$ of the codeword. 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
\[ R = (R_{i,j})_{i=0, j=0}^{n-1, k-1}, \quad \text{where} \quad R_{i,j} = \begin{cases} 1 & \text{if } i = s_j, \\ 0 & \text{otherwise}. \end{cases} \]

With this definition at hand, we require that a systematic encoder satisfy $E(u) \cdot R = u$, or equivalently that
\[ \Pi \cdot G \cdot R = I, \]
Fig. 1. The systematic encoder of [4] for an (8, 5) polar code.
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 $j$th smallest element of $A$ is denoted $\alpha_j$. For this case, recall that we define $S$ as equal to $A$ and $s_j$ as equal to $\alpha_j$.

Define the matrix $E$ as

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

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

$$G_{nrv} = E \cdot F^{\otimes m}, \quad (11)$$

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 $u$ of length $k$, we manufacture a vector $v_1$ such that the entries indexed by $\alpha_j$ equal $u_j$ and all other entries equal zero. That is,

$$v_1 = u \cdot E,$$

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

$$v_{III} = v_{II} \cdot E^T \cdot E.$$ 

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,

$$\mathcal{E}_{nrv}(u) = u \cdot E \cdot F^{\otimes m} \cdot G_{nrv} \cdot E^T \cdot E. \quad (12)$$

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

$$E \cdot F^{\otimes m} \cdot E^T \cdot E \cdot F^{\otimes m} \cdot E^T = I. \quad (13)$$

Showing this will further imply that the corresponding $\Pi$ 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

$$\langle i \rangle_2 = (i_0, i_1, \ldots, i_{m-1}),$$

where $i = \sum_{j=0}^{m-1} i_j 2^j$ and $i_j \in \{0, 1\}$. \quad (14)

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

$$\langle \tilde{i} \rangle_2 = (i_{m-1}, i_{m-2}, \ldots, i_0), \quad \tilde{i} = \sum_{j=0}^{m-1} i_j 2^{m-1-j}. \quad (15)$$

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) $\tilde{i}$ 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

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

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

$$G_{rv} = E \cdot B_n \cdot F^{\otimes m}. \quad (16)$$

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, $A$, gotten from the set of active rows $A$ by applying the bit-reverse operation on each element $\alpha_i$. As before, we order the elements of $\tilde{A}$ in increasing order and denote

$$\tilde{A} = \{\beta_j\}_{j=0}^{k-1}, \quad 0 \leq \beta_0 < \beta_1 < \cdots < \beta_{k-1} \leq n-1. \quad (17)$$

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

$$\tilde{E} = (\tilde{E}_{i,j})_{i=0}^{k-1} \quad \text{where} \quad \tilde{E}_{i,j} = \begin{cases} 1 & \text{if } j = \beta_i, \\ 0 & \text{otherwise} \end{cases}. \quad (18)$$

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

$$G_{rv} = \tilde{E} \cdot F^{\otimes m} \quad (19)$$

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

$$\mathcal{E}_{rv}(u) = u \cdot \tilde{E} \cdot F^{\otimes m} \cdot (\tilde{E})^T \cdot E \cdot F^{\otimes m} \cdot \tilde{G}_{rv}. \quad (19)$$
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 $\bar{A}$, 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 $\bar{A}$ in place of $A$. Specifically, note 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
\[ \bar{E} \cdot F^\otimes m \cdot (\bar{E})^T \cdot \bar{E} \cdot F^\otimes m \cdot (\bar{E})^T = I. \]  

(20)

IV. DOMINATION CONTIGUITY IMPLIES INVOLVATION

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 $\bar{E} \cdot F^\otimes m \cdot (\bar{E})^T$ are involutions.

Interestingly, and in contrast with the original systematic encoder presented in [6], the proof of correctness centers on the structure of $A$. That is, in [6], 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,
\[
E = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}, \quad E^T = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}, \quad F^\otimes 2 = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Thus,
\[
E \cdot F^\otimes 2 \cdot E^T = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{bmatrix}, \quad (E \cdot F^\otimes 2 \cdot E^T) \cdot (E \cdot F^\otimes 2 \cdot E^T) = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{bmatrix}.
\]

Note that the rightmost matrix above is not an identity matrix. A similar calculation shows that $\bar{E} \cdot F^\otimes 2 \cdot (\bar{E})^T$ is not an involution either. The apparent contradiction to the correctness of our algorithms is rectified by noting that $A = \{0, 1, 3\}$ cannot correspond to a polar code (as will be formalized shortly). Specifically, the above $A$ implies that $W^+$ is frozen while $W^-$ is unfrozen. As we will see, this cannot correspond to a valid polar code, since $W^+$ is upgraded (defined shortly) 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 $\geq$, as follows.

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

Namely, $i \geq j$ 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 $h \geq i$ and $i \geq j$, it holds that $i \in A$. For easy reference:

\[ (h, j \in A \text{ and } h \geq i \geq j) \implies i \in A. \]

(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 $\bar{E}$ be defined according to (5), (10), (16), and (17). Then, $E \cdot F^\otimes m \cdot E^T$ and $\bar{E} \cdot F^\otimes m \cdot (\bar{E})^T$ are involutions. That is, (13) and (20) hold.

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

We start by noting a simple characterization of $F^\otimes m$ when $F$ satisfies the polarization conditions set in [10], including being not upper triangular. Namely, the entry at row $i$ and column $j$ of $F^\otimes m$ is easily calculated:

\[ (F^\otimes m)_{i,j} = \begin{cases} 1 & i \geq j \\ 0 & \text{otherwise} \end{cases}. \]

(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 $m-1$ (the most-significant bit of $i$) equals 0 and $j \mod 2 = 1$. Next, consider the other three blocks and note that for them, $i \mod 2 = m$ iff $i \mod 2 = 1 = i \mod 2 = 1$. Thus, we continue recursively with $i \mod 2 = 1$ and $j \mod 2 = 1$. 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

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

Thus,

\[ \left( (E \cdot F^\otimes m \cdot E^T) \cdot (E \cdot F^\otimes m \cdot E^T) \right)_{p,r} = \sum_{q=0}^{k-1} (E \cdot F^\otimes m \cdot E^T)_{p,q} \cdot (E \cdot F^\otimes m \cdot E^T)_{q,r} = \sum_{i \in A} (F^\otimes m)_{h,i} \cdot (F^\otimes m)_{i,j}. \]

(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 such $i$ in $A$ for which

\[ h \geq i \text{ and } i \geq j, \]

(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 \( h \neq j \), 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 \( h \geq j \), 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 \( h \geq j \), 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 \( \Phi \) be defined according to (5), (10), (16), and (17), where \( n = 2^m \). Then, \( E \cdot F^{\otimes m} \cdot E^T \) and \( \Phi \cdot F^{\otimes m} \cdot (E^T)^T \) 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 \( \mathcal{X} = \{0, 1\} \). 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 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 \( \Phi \) 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,

\[
W^-(y_0, y_1|x_0) = \frac{1}{2} \sum_{u_1 \in \{0,1\}} W(y_0|x_0 + u_1) \cdot W(y_1|x_1) ,
\]

\[
W^+(y_0, y_1|x_0) = \frac{1}{2} W(y_0|x_0 + u_1) \cdot W(y_1|x_1). 
\]

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 \( W^- \),

\[
W^+ \succeq W^- .
\]

**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 \( \pi \). We also denote by \( \delta() \) a function taking as an argument a condition. The function \( \delta \) equals 1 if the condition is satisfied and 0 otherwise. With these definitions at hand, we take

\[
\Phi(y_0, y_1|y) = \frac{1}{2} [W(y_1|0) \cdot \delta(y_0 = y) + W(y_1|1) \cdot \delta(y_0 = \pi(y))] .
\]

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 [11]. 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 [12, 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,

\[
W^- \succeq Q^- \quad \text{and} \quad W^+ \succeq Q^+ .
\]

For a BMS channel \( W \) and \( 0 \leq i < n \), denote by \( W_i^{(m)} \) the channel which is denoted “\( W_i^{(m+1)} \)” 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,

\[
i \geq j \implies W_i^{(m)} \succeq W_j^{(m)} .
\]

**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

$$W^{(m-1)}_{[i/2]} \succeq W^{(m-1)}_{(j/2)}. $$

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 $m$ 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

$$W \succeq Q \quad \implies \quad Z(W) \leq Z(Q). \quad (28)$$

For a proof of this fact, see [13].

We deduce from (27) and (28) that if $i \succeq j$, then $Z(W_i^{(m)}) \leq Z(W_j^{(m)})$. Assume for a moment that the inequality is always strict when $i \succeq j$ and $i \not\succeq j$. 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 $i \not\succeq j$, $i \succeq j$, but $Z(W_i^{(m)})$ and $Z(W_j^{(m)})$ 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 $(t)_2$ 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 domination 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 [14, 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 [15].

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 $P$ bit/Hz, where $P$ corresponds to the level of parallelism, was presented in [7]. 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 order and has the same $P$-bit/Hz throughput and $n/P$-cycle latency as [7]. 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 $P = 4$, 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 $\in [1, \log P]$ are available in the same $P$-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 $i > \log P$ 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 $P$-bit processing elements operating in parallel, each of which has $2^{i-\log P}$ delay elements. A processing element $l$ contains a multiplexer that alternates its output between $\beta_{i-1}[Pt + l - 2^{i-1}] \oplus \beta_{i-1}[Pt + l]$ and $\beta_{i-1}[Pt + l]$ every $2^{i-\log P}$ clock cycles, where $t$ is the current cycle index.

The resulting encoder has a throughput of $P$ bit/s, a latency of $n/P$ cycles, and a critical path that passes from $u[4t + 4]$ to $x[4t]$, similar to the encoder of [7]. 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 [7] in that it can be used to implement a fully serial encoder with $P = 1$, whereas that of [7] can only scale down to $P = 2$.

B. Flexible Non-Systematic Encoder

Since the input $u$ 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.
Adapting this architecture to encode any polar code of length \( n \leq n_{\text{max}} \) 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 \( P \) instances of a \( \log n_{\text{max}} \times 1 \) multiplexer.

The width of the multiplexer can be reduced to \( \log n_{\text{max}} - \log P \) without affecting decoding latency by exploiting the combinational nature of stages \( S_{[1, S_{\log P}]} \) 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 \( n < P \). The first AND gate, \( \&_{0} \), will always have its 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 [7] 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 \( n_{\text{max}} = 16384 \) and \( P = 32 \). 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 \( n_{\text{max}} / P \) and increasing \( P \) 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.

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

Table I

<table>
<thead>
<tr>
<th>Decoder</th>
<th>LUTs</th>
<th>FF</th>
<th>RAM (bits)</th>
<th>( f ) (MHz)</th>
<th>Lat. (µs)</th>
<th>T/P</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7]</td>
<td>769</td>
<td>1,392</td>
<td>12,160</td>
<td>354</td>
<td>1.4</td>
<td>11.3</td>
</tr>
<tr>
<td>( R )-flexible</td>
<td>649</td>
<td>1,240</td>
<td>12,160</td>
<td>394</td>
<td>1.3</td>
<td>12.6</td>
</tr>
<tr>
<td>( Rn )-flexible</td>
<td>838</td>
<td>1,293</td>
<td>12,160</td>
<td>360</td>
<td>1.4</td>
<td>11.5</td>
</tr>
</tbody>
</table>

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 \( n_{\text{max}} / P \times P \)-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 \( P \)-bit/Hz throughput of the component non-systematic encoder with a latency \( L = 2L_{\text{NS}} + 2 \) cycles, where \( L_{\text{NS}} \) 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 [6] 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 [6] is implemented to be rate-flexible and low-complexity, it has a latency of $n \log n$ clock cycles; compared to the $2n/P + 2$ cycle latency of the proposed architecture.

### E. Systematic Encoder Implementation

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

### F. On Code Shortening

The work in [16] describes a shortening scheme for systematic 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 systematic codeword before transmission. The result is that $n_s$ bits containing $k_s$ information bits are transmitted; where $k_s = k - (n - n_s)$. This scheme is 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 constraints on the code length or rate.

To adapt our proposed systematic encoder and enable shortening, the second input, $e_{ni}$, to the AND gates $&_i$ becomes

$$
e_{ni} = \begin{cases} 1 & \text{when } n \geq \lfloor (Pt + i)/2 \rfloor, \\ 0 & \text{otherwise.} \end{cases}$$

Adding code shortening ability has a minor effect on the resource utilization of the $R_n$-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 $P$, e.g. $P = 8$ 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:

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

This operation is applied directly to $P$ bits simultaneously in stage $S_i$ if $2^{i-1} \geq P$. However, since we can only read and write data in groups of $P$ 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:
\[
m_i[t] = \begin{cases} 0 & \text{when } [t/2^{i-1}] \text{ is even,} \\ 1 & \text{otherwise.} \end{cases}
\]
The output for these stages is calculated using:
\[
\beta_i[t : t + \mathcal{P} - 1] = \beta_{i-1}[t : t + \mathcal{P} - 1] \oplus ((\beta_{i-1}[t : t + \mathcal{P} - 1] \& m_i) \gg 2^{i-1}).
\]
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 \( \gg \) 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 \text{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 encoded throughput values for the proposed software systematic encoder running on a 3.4 GHz Intel Core i7-2600 are shown in Table V for \( n = 32, 768 \).

<table>
<thead>
<tr>
<th>( \mathcal{P} )</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency (( \mu )s)</td>
<td>64.1</td>
<td>30.1</td>
<td>14.3</td>
<td>7.7</td>
<td>4.1</td>
<td>3.3</td>
</tr>
<tr>
<td>( \text{T/P (Gbps)} )</td>
<td>0.5</td>
<td>1.1</td>
<td>2.3</td>
<td>4.2</td>
<td>8.0</td>
<td>10.0</td>
</tr>
</tbody>
</table>

As discussed in [4], 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 [4]. In this section, we review the Fast-SSC algorithm, describe the architectural modifications necessary to decode any polar code up to a maximum length \( n_{\text{max}} \) and analyze the resulting implementation.

### A. The Fast-SSC Decoding Algorithm

Our proposed flexible decoders utilize the Fast-SSC presented in [4]. 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 \( S_{9} \), 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 [17], 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.

### B. Stage Indices and Sizes

The Fast-SSC decoder in [4] starts decoding a polar code of length \( n_{\text{max}} \) at stage \( S_{\log n_{\text{max}}} \); 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 \( n \leq n_{\text{max}} \) 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 [4] 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 \( n_{\text{max}} \):

\[
n_v(S_i) = 2^i \frac{n}{n_{\text{max}}}.
\]  

(30)

The memory allocated for a stage \( S_i \) remains the same regardless of \( n \) and is always calculated assuming \( n = n_{\text{max}} \). The flexible decoder always starts from \( S_{\log n_{\text{max}}} \), corresponding to a polar code of length \( n \leq n_{\text{max}} \), performing operations on \( n_v(S_i)/(2P) \) 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.

C. Implementation Results

Since memory is accessed as words containing multiple \( 2P \) 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 [4]. 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 \( (n_{\text{max}} = 32,768) \) with the Fast-SSC decoder of [4] \( (n = 32,768) \) when both are implemented using the Altera Stratix IV EP4SGX530KH40C2 FPGA. The resource requirements are also provided for \( n_{\text{max}} = n = 2048 \). 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 \( n < n_{\text{max}} \), 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).

The implementation results for \( Rn\)-flexible decoder supporting code shortening are also included in Table VI. The main change is the requirement for \( n_{\text{max}} \) 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 \( n_{\text{max}} \)-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 [18]. 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 \( n_{\text{max}} \) bits enables the decoder to only perform the binary addition and store its results, eliminating superfluous copy operations and decreasing latency [18]. 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 [18] 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 \( n_{\text{max}} \). 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 [18] 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 minimum 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.

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.

### Table VII

<table>
<thead>
<tr>
<th>Decoder</th>
<th>Latency ($\mu$s)</th>
<th>Info. Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar Fast-SSC</td>
<td>256</td>
<td>115</td>
</tr>
<tr>
<td>Unrolled Fast-SSC [18]</td>
<td>109</td>
<td>270</td>
</tr>
<tr>
<td>Proposed Fast-SSC</td>
<td>149</td>
<td>198</td>
</tr>
</tbody>
</table>

### Table VIII

<table>
<thead>
<tr>
<th>Decoder</th>
<th>Latency ($\mu$s)</th>
<th>Info. Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar Fast-SSC</td>
<td>16.2</td>
<td>106</td>
</tr>
<tr>
<td>Unrolled Fast-SSC [18]</td>
<td>4.0</td>
<td>430</td>
</tr>
<tr>
<td>Proposed Fast-SSC</td>
<td>11.1</td>
<td>155</td>
</tr>
</tbody>
</table>

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, parallelizable 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 matching 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 unrolled, code-specific decoders.

### References


