Efficient Algorithms for Systematic Polar Encoding

Harish Vangala, Yi Hong, and Emanuele Viterbo

***Abstract*—Arıkan has laid the foundations of systematic polar codes and has also indicated that the computational complexity order of the systematic polar encoder (SPE) can be the same of a nonsystematic polar encoder (NSPE) i.e., $\Theta(N \log N)$. In this letter, we propose three efficient encoders along with their full pseudocode implementations, all with $\Theta(N \log N)$ complexity. These encoders work for any arbitrary choice of frozen bit indices, and they allow a tradeoff between the number of XOR computations and the number of bits of memory required by the encoder. We show that our best encoder requires exactly the same number of XORs as that of NSPE.**

***Index Terms*—Encoder, low complexity, systematic polar codes.**

I. Introduction

SYSTEMATIC polar codes (SPC) were proposed by Arıkan and are known for their improved bit error rate (BER) performance compared to the original non-systematic polar codes [1], [2]. In [1], a systematic polar encoder (SPE) was proposed based on a successive cancellation decoder (SCD) over a binary erasure channel (BEC). In [3], another encoder was proposed using a cascade of two non-systematic polar encoder (NSPE) circuits. It was recently proved in [4] that such encoder works only under certain conditions. They also prove that if a given polar code does not satisfy these conditions they can transform it into a new polar code that performs at least as well as the original one. In [5, Section 3], following Arıkan's recursion, a systematic encoder is proposed for general $\ell \times \ell$ triangular kernels with a complexity $\Theta(N \log N)$ and a memory of $\Theta(N)$ bits. We note that all the above SPEs have complexity, memory, and latency higher than an NSPE.

In this letter, we measure the efficiency of an encoder as the exact number of XOR operations and the exact memory required in number of bits for the computations (excluding the input/output). In particular,

1) we present three efficient systematic encoders for SPC, which work for codes with any choice of frozen bit indices.
2) we demonstrate a trade-off between the number of XOR operations and the number of bits required among our three encoders. The least number of XORs required by our non-recursive SPE is exactly equal to that of an NSPE.
3) we revisit [1] and present explicit and efficient *recursive* encoders with exact memory and XOR counts.

Manuscript received July 17, 2015; revised October 26, 2015; accepted October 27, 2015. Date of publication November 2, 2015; date of current version January 7, 2016. This work is supported by the NPRP grant #NPRP5-597-2-241 from the Qatar National Research Fund (a member of Qatar Foundation). The associate editor coordinating the review of this paper and approving it for publication was Z. Ma.

The authors are with the Department of Electrical and Computer Systems Engineering, Monash University, Melbourne, Vic. 3800, Australia (e-mail: harish.vangala@monash.edu; emanuele.viterbo@monash.edu; yi.hong@monash.edu).

Digital Object Identifier 10.1109/LCOMM.2015.2497220

An open access simulation platform, including the encoders in this letter is available in [6].

II. System Model

Let $\mathcal{N} \triangleq \{1, 2, \ldots, N\}$. Boldface lower and upper case letters represent vectors, and matrices respectively. All the vectors in this letter are row-vectors. Given a vector $\mathbf{z} = [z_1, z_2, \ldots]$ and a subset of indices $\mathcal{J}$, $\mathbf{z}_\mathcal{J}$ denotes the sub-vector formed by the elements with the indices in $\mathcal{J}$. Similarly, given a matrix $\mathbf{A}$, $\mathbf{A}_\mathcal{J}$ denotes the sub-matrix formed by the rows at the indices $\mathcal{J}$. Let $\mathbf{F} \triangleq \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$ be the standard kernel used to define polar codes and $\mathbf{F}^{\otimes n}$ denote the n-fold Kronecker product of n copies of $\mathbf{F}$.

A. The Non-Systematic Polar Encoder (NSPE)

Binary polar codes are completely specified by a triple $(N, K, \mathcal{I})$, where N is the code length, K is the length of the message, and $\mathcal{I} \subseteq \mathcal{N}$, $|\mathcal{I}| = K$ is the set of indices known as *information bit indices*. The remaining $N - K$ indices are called as *frozen bit indices*. Here, N is a power of 2 and we define $n \triangleq \log_2(N)$.

For a $(N, K, \mathcal{I})$ polar code, the generator matrix is $\mathbf{G} = \left(\mathbf{F}^{\otimes n}\right)_\mathcal{I}$. Therefore, given a message vector $\mathbf{u}$ of K information bits, a codeword $\mathbf{x}$ is generated as:

$$\mathbf{x} = \mathbf{u} \cdot \mathbf{G} = \mathbf{d} \cdot \mathbf{F}^{\otimes n} \tag{1}$$

where $\mathbf{d}$ is a vector of N bits including information bits such that $\mathbf{d}_\mathcal{I} = \mathbf{u}$, $\mathbf{d}_{\mathcal{I}^c} = \mathbf{0}$, and $\mathcal{I}^c \triangleq \mathcal{N} \backslash \mathcal{I}$. The bits $\mathbf{d}_{\mathcal{I}^c}$ are called as *frozen bits*, and are set to zero.

Due to the recursive construction of the matrix $\mathbf{F}^{\otimes n}$, we can perform this matrix-vector multiplication in $\Theta(N \log N)$ only. This NSPE is illustrated in Fig. 1 and it requires exactly $\left(\frac{N}{2} \log_2 N\right)$ XORs and $2N$ bits (i.e. for the extreme nodes). We will use these values to benchmark our new encoders.

B. The Systematic Polar Encoding

A systematic polar code may be described as equivalent to the original polar code in (1), except that the message vectors are mapped to codewords, such that the message-bits are explicitly visible [1]. Consider the indices of K bits in a codeword $\mathbf{x}$, where the message bits appear explicitly. It was shown in [1] that this set can be chosen equal to the set of *information bit indices* $\mathcal{I}$. Note that, it slightly differs from what is usually considered as a systematic linear block code, where message bits appear as *first* K bits. This motivates our interpretation of the SPE below.

The SPE of an information $\mathbf{u}$ of K bits, is the solution of:

$$\mathbf{x} = \mathbf{y} \cdot \mathbf{F}^{\otimes n} \tag{2}$$

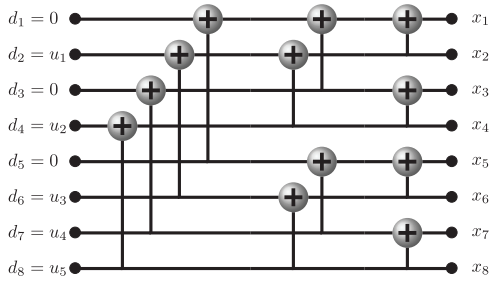

Fig. 1. The non-systematic polar encoder (NSPE) with an exact complexity of $\left(\frac{N}{2}\log_2 N\right)$ XORs, when $(N, K, \mathcal{I}) = (8, 5, \{2, 4, 6, 7, 8\})$.

TABLE I
SUMMARY OF SYSTEMATIC POLAR ENCODERS

Algorithm	Recursion	# bits (excl. I/O)	# XORs
EncoderA	No	$N(1+\log_2 N)$	$\frac{N}{2}\log_2 N$
EncoderB	Yes	$2N-1$	$N(1+\log_2 N)$
EncoderC	Yes	N	$N(1+2\log_2 N)$
NSPE	Yes/No	$2N$	$\frac{N}{2}\log_2 N$

where, $\mathbf{y}_{\mathcal{I}}$ and $\mathbf{x}_{\mathcal{I}^c}$ are the unknowns. Within the systematic codeword $\mathbf{x}$, $\mathbf{x}_{\mathcal{I}} = \mathbf{u}$ are information bits and $\mathbf{x}_{\mathcal{I}^c}$ are parity bits. Frozen bits are given by $\mathbf{y}_{\mathcal{I}^c} = 0$. For example, if $(N, K, \mathcal{I}) = (4, 2, \{1, 3\})$ and $\mathbf{u} = [10]$, then (2) becomes:

$$\left[\, 1, x_2, 0, x_4 \,\right] = \left[\, y_1, 0, y_3, 0 \,\right] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

where y_1, y_3 and x_2, x_4 are the unknowns. Note that there are exactly N unknowns, shared between $\mathbf{x}$ and $\mathbf{y}$. It is easy to see that (2) is a system of linear equations, up to a re-arrangement.

Since $\mathbf{F}^{\otimes n}$ is an upper triangular matrix, a straightforward *Gaussian elimination* (GE) could be used to solve the equations. Then the number of XORs required to perform GE would be approximately equal to the number of ones in the matrix: $3^{\log_2 N} = N^{\log_2 3} \approx N^{1.585}$. This is much higher than the achievable complexity of $\Theta(N \log N)$ XORs, as indicated by Arıkan. In fact, devising efficient encoders achieving this complexity bound forms the core objective of this letter.

III. NEW SYSTEMATIC POLAR ENCODERS

In this section, we propose three novel systematic polar encoders based on efficient methods of solving (2). Our first SPE requires the least number of XORs, exactly equal to that of an NSPE. Later, using the recursion in [1] we show two explicit SPEs, which recursively split the set of encoder equations in (2). For each of our encoders, we compute the exact number of XORs and memory required in bits (see Table I).

The pseudocodes of our three encoders are provided at the end as routines **EncoderA**$(\mathbf{y}, \mathbf{x})$, **EncoderB**$(i, j, \mathbf{y}, \mathbf{x}, \mathbf{r})$, and **EncoderC**$(i, j, \mathbf{y}, \mathbf{x}, \mathbf{r})$. To perform the encoding of message vector $\mathbf{u}$, we need to initialize the arguments as: $i = 1$, $j = N$, $\mathbf{x}_{\mathcal{I}} = \mathbf{u}$, $\mathbf{x}_{\mathcal{I}^c}$ = undef, $\mathbf{y}_{\mathcal{I}}$ = undef, $\mathbf{y}_{\mathcal{I}^c} = \mathbf{0}$ and $\mathbf{r} = \mathbf{0}_{N\times 1}$. The values of the triple $(N, K, \mathcal{I})$ are implicitly available to all routines. The solutions are reflected in the values of $\mathbf{x}$ and $\mathbf{y}$ (pass-by-reference) after each function call, according to (2).

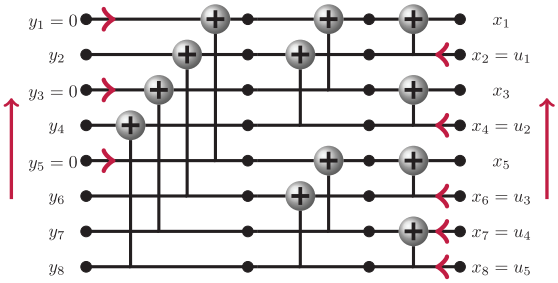

Fig. 2. The flow-of-calculations required for SPE on the encoder graph.

A. *EncoderA: Non-Recursive, With the Least XORs*

In the circuit implementation of NSPE in Fig 1, bits evolve from left to right in $n = \log_2 N$ stages. Our **EncoderA** considers exactly the same circuit, except for a different order of computations and some intermediate memory elements.

We first see that in an SPE based on (2), all the N known bits and N unknown bits of (2) are distributed on the two extremes of the circuit shown in Fig. 2. The circuit has $N(1+\log_2 N)$ nodes, each storing a bit. Each horizontal connection holds $n+1$ nodes. Further, on any horizontal connection, only 1 out of $n+1$ nodes is known at one of the two extremes. Therefore we start calculations at the known node and move gradually towards the other extreme, calculating the nodes one by one. An illustration of such order of computations is shown in Fig. 2. The key observation that justifies this computation is as follows. The computation of any of the $(n+1)$ nodes on any given horizontal connection involves only the nodes from the same connection and the nodes below it. Also note that, the order of evaluation of the horizontal connections must be bottom-up. This is reminiscent of a backward-substitution.

This procedure requires exactly the same number $\frac{N}{2}\log_2 N$ of XOR computations as an NSPE, and $N(1+\log_2 N)$ bits (see the pseudocode of **EncoderA**( ) and Table I).

B. *Recursive SPEs Based on Arıkan's Recursion [1]*

We revisit Arıkan's recursive idea based on a divide-and-conquer iteration, which splits the set of equations (2) into two at each step. This enables us to design efficient encoding algorithms that trade the number of XORs with memory.

Consider a vector $\mathbf{z} = [z_1, \ldots, z_N]$ of N elements. We define a *level-l partition* of $\mathbf{z}$ as,

$$\mathbf{z} = [\mathbf{z}_{2^l}^{(l)}, \ldots, \mathbf{z}_2^{(l)}, \mathbf{z}_1^{(l)}],\ 0 \le l \le n, \tag{3}$$

where, $\mathbf{z}_i^{(l)}$ denotes a sub-vector of $\mathbf{z}$ of length 2^{n-l}, which can be expanded as $\mathbf{z}_i^{(l)} = [z_{a_i}, \ldots, z_{a_i+2^{n-l}-1}]$, where $a_i = (2^l - i)\cdot 2^{n-l} + 1$, $1 \le i \le 2^l$. In other words, $\mathbf{z}_i^{(l)}$ is simply a pointer to the above sub-array of $\mathbf{z}$ (as in a programming language like C). The partition indices are given in reverse order so that they follow the order of evaluation in our encoders. Such indices also simplify the traversal of a tree structure that we use later. Finally, note that $\mathbf{z}_i^{(n)} = z_{N-i+1}$ and $\mathbf{z}_1^{(0)} = \mathbf{z}$.

Let $\{\mathbf{r}_{l,i} : 0 \le l \le n, 1 \le i \le 2^l\}$ denote a collection of $2N-1$ different length binary vectors corresponding to the $2N-1$ nodes of a binary tree shown in Fig. 4. Let l be the level in the tree (or recursion index) and i be the sub-vector

index, for the nodes at the same level in the tree. All the 2^l vectors at a given level l have the same length 2^{n-l} and can be concatenated as a single length N vector. Although all these vectors add up to a total of $N(1+\log_2 N)$ bits, we will later show how to significantly reduce such memory requirement.

Recursion at level $l = 1$ — Consider the level-1 partitioning: $\mathbf{x} = [\mathbf{x}_2^{(1)}, \mathbf{x}_1^{(1)}]$ and $\mathbf{y} = [\mathbf{y}_2^{(1)}, \mathbf{y}_1^{(1)}]$. Using the block decomposition $\mathbf{F}^{\otimes n}$, (2) can be rewritten as:

$$\mathbf{x}_1^{(1)} = \mathbf{y}_1^{(1)}\mathbf{F}^{\otimes n-1} \text{ and} \tag{4}$$

$$\mathbf{x}_2^{(1)} = \mathbf{y}_2^{(1)}\mathbf{F}^{\otimes n-1} + \mathbf{x}_1^{(1)} \tag{5}$$

Clearly, the set of equations in (4) can be independently solved (to obtain $\mathbf{x}_1^{(1)}$) and substituted in (5) to save one sub-matrix multiplication. Note that solving (5) is different from solving (4) due to the presence of a *binary offset* vector $\mathbf{x}_1^{(1)}$. We initialize these offset vectors as: $\mathbf{r}_{1,1} = \mathbf{0}$ and $\mathbf{r}_{1,2} = \mathbf{x}_1^{(1)}$.

Recursion at levels $2 \le l \le n$ — In general, at level $(l-1)$, we find 2^{l-1} equations, relating the level $l-1$ partition of vectors of $\mathbf{x}$ and $\mathbf{y}$ as below.

$$\mathbf{x}_i^{(l-1)} = \mathbf{y}_i^{(l-1)}\mathbf{F}^{\otimes n-l+1} + \mathbf{r}_{l-1,i} \quad 1 \le i \le 2^{l-1}. \tag{6}$$

Then, at level $2 \le l \le n$, each of the above equations splits similarly to (4) and (5) as follows

$$\mathbf{x}_{2i-1}^{(l)} = \mathbf{y}_{2i-1}^{(l)}\mathbf{F}^{\otimes n-l} + \mathbf{r}_{l,2i-1} \text{ and} \tag{7}$$

$$\mathbf{x}_{2i}^{(l)} = \mathbf{y}_{2i}^{(l)}\mathbf{F}^{\otimes n-l} + \mathbf{r}_{l,2i}, \tag{8}$$

where the new binary offsets $\{\mathbf{r}_{l,2i-1}, \mathbf{r}_{l,2i}\}$ at level l are computed as

$$\mathbf{r}_{l,2i-1} := (\mathbf{r}_{l-1,i})_1^{(1)} \tag{9}$$

$$\mathbf{r}_{l,2i} := (\mathbf{r}_{l-1,i})_1^{(1)} + (\mathbf{r}_{l-1,i})_2^{(1)} + \mathbf{x}_{2i-1}^{(l)}, \tag{10}$$

where the two halves of $\mathbf{r}_{l-1,i}$ are denoted by $(\mathbf{r}_{l-1,i})_1^{(1)}$ and $(\mathbf{r}_{l-1,i})_2^{(1)}$, using a similar notation to (3). Note that (10) can be applied only after solving (7), since the solution $\mathbf{x}_{2i-1}^{(l)}$ is required. We will next see a more convenient visualization of these recursive computations. Much different from [5, Section 3], our recursion operates on the binary offset vectors and does not involve any bit-reversal permutation.

Binary tree representation — Consider a perfect binary tree with $n+1$ levels $l = 0, \ldots, n$, with root node at level-0. The 2^{l-1} nodes at a level $l-1$ are associated with the 2^{l-1} set of 2^{n-l+1} equations from (6). Each node at level $l-1$ splits into two at level l as shown in Fig. 3. The solution of (2) is obtained by an *in-order* traversal of the tree. Each time we visit a leaf node (i.e., at $l = n$), we obtain a simple equation of the form: $\mathbf{x}_i^{(n)} = \mathbf{y}_i^{(n)} + \mathbf{r}_{n,i}$, where either $\mathbf{x}_i^{(n)}$ or $\mathbf{y}_i^{(n)}$ is the unknown to solve. Computing $\mathbf{r}_{n,i}$ using the successive updates of (9) and (10), is therefore the critical objective of this traversal. The binary offsets $\mathbf{r}_{l,i}$ are stored in a tree structure in Fig. 4 as explained before.

As a part of the in-order traversal of the tree in Fig. 3 (or Fig. 4), we visit each node *three* times, except for the leaf nodes which are visited only once.

Let us consider the sequence of visits and respective operations for a node at level $l-1$ in the tree as shown in Figs. 3 and 4.

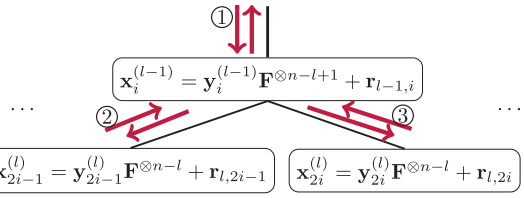

Fig. 3. The tree of recursive computations.

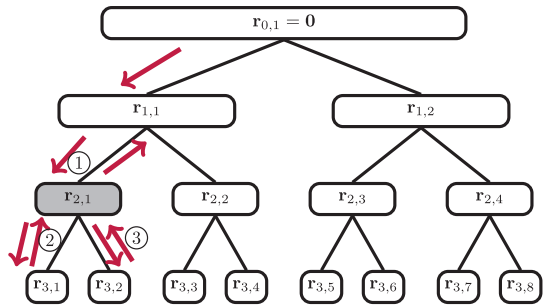

Fig. 4. Traversal of nodes in recursion, at $N = 8$.

1) *First Visit* ① — The required offsets $\mathbf{r}_{l,2i-1}$ are simply copied using (9). Then a recursive call down the left branch is made which, upon return, gives the solution (7).
2) *Second Visit* ② — Returning from the left sub-tree we have full knowledge of $\mathbf{x}_{2i-1}^{(l)}$, so that $\mathbf{r}_{l,2i}$ can be updated by using (10). Then a recursive call down the right branch is made which, upon return, gives the solution of (8).
3) *Third Visit* ③ — By now we have visited the entire sub-tree and we have the all solutions of (6). Then we return to the parent node.

The exact XOR count — Let $f(N)$ denote the number of XORs required to solve (2). From the recursive equations (7)–(10) we have:

$$f(N) = 2 \cdot f(\tfrac{N}{2}) + \beta\tfrac{N}{2} \text{ and } f(1) = 1. \tag{11}$$

The first term on the right hand side accounts for solving recursively (7) and (8), while the second term $\beta\frac{N}{2}$ accounts for the computations required by (9) and (10). In the case where we have allocated memory for each node of the tree in Fig. 4 (namely, **EncoderB**), we have $\beta = 2$. A different implementation (namely, **EncoderC**), saving on the memory to store the intermediate values of the $\mathbf{r}_{l,i}$ offset vectors will have $\beta = 4$. The closed form solution of (11) yields $f(N) = N + \frac{\beta N}{2}\log_2 N$.

Note that we ignore all the index computations to address the subvectors and any operation to identify frozen bits that are implementation dependent and can be avoided by hardwiring.

EncoderB requires exactly $N(1+\log_2 N)$ XORs, which is approximately the same number of XORs required by the encoder proposed in [3]. However, our **EncoderB** works without any restrictions on code rate and frozen bit indices. Our pseudocode **EncoderB( )** implements the storage of offset vectors as a local variable **v** allocated upon each recursive call and released on return. At any point in the tree traversal, we have a chain of at most $n+1$ recursive function calls being active (a path from the root node to a leaf node in Fig. 4).

Function EncoderA(y, x)

INPUT : **y**, **x** with unfilled bits (variables of (2));
OUTPUT: Updated vectors **y** and **x**, with solutions of (2) filled in.

$n \triangleq \log_2(N)$ and $\mathbf{X}$ is an $N \times (n+1)$ binary matrix // $\Theta(N \log N)$ bits
Set: $\mathbf{X}[:][1] = \mathbf{y}$ and $\mathbf{X}[:][n+1] = \mathbf{x}$ // First & last columns
for $i = N, N-1 \ldots, 1$ **do**
  if $i \in \mathcal{I}$ **then** $s = n+1$; $\delta = -1$;
  else $s = 1$; $\delta = 1$; **end**
  Let the binary representation: $(i-1) \equiv b_1 b_2 \ldots b_n$, with b_1 as MSB
  for $j = 1, 2, \ldots, n$ **do**
    $t = s + j\delta$; $l = \min(t, t-\delta)$; $\kappa = 2^{n-l}$
    if $b_l = 0$ **then**
      $\mathbf{X}[i][t] = \mathbf{X}[i][t-\delta] \oplus \mathbf{X}[i+\kappa][t-\delta]$
    else
      $\mathbf{X}[i][t] = \mathbf{X}[i][t-\delta]$
    end
  end
end
$\mathbf{y} = \mathbf{X}[:][1]$ & $\mathbf{x} = \mathbf{X}[:][n+1]$ // Solutions in first & last columns

Therefore, the maximum size of allocated memory by these function calls is exactly $N + \frac{N}{2} + \ldots + 1 = 2N - 1$ bits. Note that the same memory allocation/deallocation process can be implemented by simply addressing an array of $2N-1$ bits.

EncoderC provides a memory-efficient implementation by using a single N-bit vector, since at any level l of Fig. 4, we need to store N bits of binary offsets $\mathbf{r}_{l,i}$. However, this slightly increases the number of XORs due to the following reason. We update the offsets every time we visit a non-leaf node for the second time. Therefore when we visit it for the third time, we should restore the original offset to enable computations at the parent node (see Fig. 4). This requires using twice equation (10). Hence doubling the number of XORs yields $\beta = 4$, and leads to an overall $N(1 + 2\log_2 N)$ XOR count, with only N bits memory (excl. I/O). **EncoderC**( ) provides its pseudocode.

Function v = EncoderB(i, j, y, x, r)

INPUT : Sub-vectors of **y**, **x** formed by bits from index i to j and offset bits from a vector **r** of $L = j - i + 1$ elements;
OUTPUT: Updated **y**, **x** and a returned vector **v** of same size as **r** represents a useful intermediate computation

$m \triangleq \lfloor \frac{i+j}{2} \rfloor$ and $L \triangleq (j - i + 1)$
Initialize: **v** — an $L \times 1$ vector // $2N-1$ bits in total
if $L = 1$ **then**
  if $i \in \mathcal{I}$ **then:** $\mathbf{y}[i] = \mathbf{x}[i] \oplus \mathbf{r}[1]$ and $\mathbf{v}[1] = \mathbf{y}[i]$
  else: $\mathbf{x}[i] = \mathbf{y}[i] \oplus \mathbf{r}[1]$ and $\mathbf{v}[1] = \mathbf{y}[i]$
else
  Denote the half-partitions: $\mathbf{v} \triangleq \left[\begin{smallmatrix}\mathbf{v}_2\\ \mathbf{v}_1\end{smallmatrix}\right]$ and $\mathbf{r} \triangleq \left[\begin{smallmatrix}\mathbf{r}_2\\ \mathbf{r}_1\end{smallmatrix}\right]$
  $\mathbf{v_1} = \mathbf{EncoderB}(m+1, j, \mathbf{y}, \mathbf{x}, \mathbf{r_1})$ // (7) and (9)
  $\mathbf{v_2} = \mathbf{r_2} \oplus \mathbf{v_1}$ // (10) $\boxed{\frac{L}{2}\ \text{XORs}}$
  $\mathbf{v_2} = \mathbf{EncoderB}(i, m, \mathbf{y}, \mathbf{x}, \mathbf{v_2})$ // (8)
  $\mathbf{v_2} = \mathbf{v_2} \oplus \mathbf{v_1}$ $\boxed{\frac{L}{2}\ \text{XORs}}$
end
return **v** // An intermediate result: $(\mathbf{x}_i^{(l-1)} + \mathbf{r}_{l-1,i}$ from (6))

Function EncoderC(i, j, y, x, r)

INPUT : Sub-vectors of $N \times 1$ size vectors y, x and r formed by the consecutive $L = j - i + 1$ bits from i to j;
OUTPUT: Updated **y**, **x** and **r**

if $i = j$ **then**
  if $i \in \mathcal{I}$ **then** $\mathbf{y}[i] = \mathbf{x}[i] \oplus \mathbf{r}[i]$
  else: $\mathbf{x}[i] = \mathbf{y}[i] \oplus \mathbf{r}[i]$
else
  $m \triangleq \lfloor \frac{i+j}{2} \rfloor$
  EncoderC$(m+1, j, \mathbf{y}, \mathbf{x}, \mathbf{r})$ // (7) & (9)
  $\mathbf{r}[i:m] = \mathbf{r}[i:m] \oplus \mathbf{r}[m+1:j] \oplus \mathbf{x}[m+1:j]$ // (10) $\boxed{\text{L XORs}}$
  EncoderC$(i, m, \mathbf{y}, \mathbf{x}, \mathbf{r})$ // (8)
  $\mathbf{r}[i:m] = \mathbf{r}[i:m] \oplus \mathbf{r}[m+1:j] \oplus \mathbf{x}[m+1:j]$ // restore **r** $\boxed{\text{L XORs}}$
end

IV. Conclusions

We have designed three efficient, low complexity algorithms to perform systematic polar encoding, at the same complexity order of $\Theta(N \log N)$. All the three algorithms work for any arbitrary choice of frozen bit indices (not necessarily polar). They further illustrate a trade-off between the required number of XORs and the memory required for an SPE. We benchmark our encoders with a standard NSPE as detailed in Table I. Our first encoder is a non-recursive encoder that requires the least number of XORs, exactly equal to that of an NSPE. The remaining two encoders are recursive, that require approximately twice and four times the number of XORs as an NSPE, respectively. Further improvements to the encoders such as higher parallelization and lower memory are relegated to our future work.

References

[1] E. Arikan, "Systematic polar coding," *IEEE Commun. Lett.*, vol. 15, no. 8, pp. 860–862, Aug. 2011.
[2] E. Arikan, "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, Jul. 2009.
[3] 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.
[4] G. Sarkis, I. Tal, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Flexible and low-complexity encoding and decoding of systematic polar codes," 2015, pp. 1–9, arXiv:1507.03614 [cs.IT].
[5] N. Presman and S. Litsyn, "Recursive descriptions of polar codes," 2012, pp. 1–60, arXiv:1209.4818v3 [cs.IT].
[6] H. Vangala, E. Viterbo, and Y. Hong. *Polar Coding Algorithms in MATLAB*, 2015 [Online]. Available: http://www.ecse.monash.edu.au/staff/eviterbo/polarcodes.html