# A Semi-Parallel Successive-Cancellation Decoder for Polar Codes

Camille Leroux*, Alexandre J. Raymond†, Gabi Sarkis†, and Warren J. Gross†
*IMS Laboratory - Institut Polytechnique de Bordeaux, Bordeaux, France
†Department of Electrical and Computer Engineering, McGill University, Montréal, Québec, Canada
camille.leroux@ipb.fr, {alexandre.raymond, gabi.sarkis}@mail.mcgill.ca, warren.gross@mcgill.ca

***Abstract*—Polar codes are a recently-discovered family of capacity-achieving codes that are seen as a major breakthrough in coding theory. Motivated by the recent rapid progress in the theory of polar codes, we propose a semi-parallel architecture for the implementation of successive cancellation decoding. We take advantage of the recursive structure of polar codes to make efficient use of processing resources. The derived architecture has a very low processing complexity while the memory complexity remains similar to that of previous architectures. This drastic reduction in processing complexity allows very large polar code decoders to be implemented in hardware. An $N = 2^{17}$ polar code successive cancellation decoder is implemented in an FPGA. We also report synthesis results for ASIC.**

## I. Introduction

Claude Shannon [1] proved the existence of a maximum rate—called the channel capacity—at which information can be reliably transmitted over a channel, and the existence of codes which enable data transmission at that rate.

Since then, different capacity-*approaching* codes have been created [2], [3]; however, designing capacity-*achieving* codes with an explicit construction eluded researchers until Arıkan proposed polar codes [4] and proved that they asymptotically achieve the capacity of binary-input symmetric memoryless channels. Later works proved that polar codes achieve the channel capacity for any discrete memoryless channel and, by extension, for any continuous memoryless channel [5]. Moreover, Arıkan provided an explicit construction method for polar codes and showed that they can be efficiently encoded and decoded with complexity $O(N \log N)$, where $N$ is the code length. Polar codes were also shown to be good at solving other information theoretic problems in an efficient manner [6], [7], [8].

On the other hand, polar codes require large code lengths to approach the capacity of the underlying channel ($N > 2^{20}$), in which case, over 20 million computational nodes are required to decode one codeword. This presents a real challenge when it comes to implementing the associated successive cancellation (SC) decoder in hardware.

Despite the large code lengths required, polar codes have two desirable properties for hardware implementation. First, they are explicitly described in a recursive framework. This regular structure enables resource sharing and significantly simplifies the scheduling and the architecture. Additionally, unlike many common capacity-approaching codes, polar codes do not require any kind of randomness to achieve good error-correcting performance. This helps avoid memory conflicts and graph-routing problems during the implementation.

Much of the work in the literature is aimed at improving the error-correction performance of polar codes at moderate lengths. Examples of this work include: list decoding [9], non-binary polar codes [10], and using more complex construction methods [11]. In this work, we focus on implementing the standard SC decoding algorithm. Any improvement made to a SC decoding implementation can benefit more complex decoding algorithms since all polar codes are based on the same recursive construction. In addition, a low-complexity SC decoder implementation enables the use of longer polar codes.

The original description of the SC decoding algorithm [4] mapped the decoder to the factor graph of the polar code, without going into details of a specific architecture. The resulting SC decoder includes $N \log_2 N$ node processors, requires $N \log_2 N$ memory elements (ME), and takes $2N - 2$ steps to decode one codeword.

A particular scheduling of SC decoding was shown to enable beneficial resource sharing without affecting throughput [12]. An SC decoder architecture containing only $N$ node processors and $2N$ memory elements was detailed. It was also suggested to perform computations in the logarithmic domain in order to reduce the complexity of each node processor.

In this study, we apply well known implementation methods to optimize a polar code successive-cancellation decoder, such as semi-parallel architecture [13], min-sum algorithm [14], arithmetic resource sharing and RAM access sharing. Some of these strategies were proposed for other channel decoder architectures.

Similar to what was done for LDPC decoders [13], we further reduce the hardware complexity of SC decoding by introducing a semi-parallel architecture. We estimate the speed penalty induced by its processing resource reduction and show that it is negligible compared to the associated complexity reduction. Logic synthesis results on FPGA and ASIC targets confirm the higher efficiency of the proposed semi-parallel architecture.

In Section II, we briefly review the polar code encoding and decoding processes. Section III describes the scheduling of SC decoding and defines the latency and utilization rate of a SC decoder. Section IV surveys existing decoder architectures and highlights their low computational-resource utilization rate. Section V describes and analyzes the proposed semi-parallel

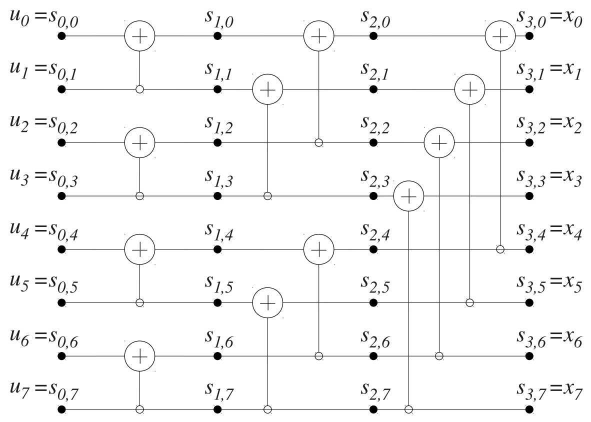

Fig. 1. Polar code encoder with $N = 8$.

SC decoder architecture. A detailed hardware architecture is given in Section VI. Finally, FPGA and ASIC implementation details are presented in Section VII.

## II. Polar Codes

### A. Polar Code Construction and Encoding

Polar codes are linear block codes of length $N = 2^n$ whose generator matrix is constructed using the $n^{\text{th}}$ Kronecker power of the matrix $F = \left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right]$. For example, for $n = 3$,

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

The equivalent graph representation of $F^{\otimes 3}$ is illustrated in Fig. 1, where $\mathbf{u} = u_0^7$ represents the information-bit vector and $\mathbf{x} = x_0^7$ is the codeword sent over the channel. We use the same notation for vectors as that of [4], namely $u_a^b$ consists of bits $u_a, ..., u_b$ of the vector $\mathbf{u}$.

When the received vectors are decoded using an SC decoder, every estimated bit $\hat{u}_i$ has a predetermined error probability—given that bits $u_0^{i-1}$ were correctly decoded—that tends towards either 0 or 0.5. Moreover, the proportion of estimated bits with a low error probability tends towards the capacity of the underlying channel, as proved in [4]. Polar codes exploit this phenomenon, called channel polarization, by using the most reliable $K$ bits to transmit information, while setting—or freezing—the remaining $N - K$ bits to a predetermined value, such as 0.

### B. Successive Cancellation Decoding

Given a received vector $\mathbf{y}$ corresponding to a transmitted codeword $\mathbf{x}$, the SC decoder successively estimates the transmitted bits $u_0$ to $u_{N-1}$. At step $i$, if $i$ is not in the frozen set, the SC decoder estimates $\hat{u}_i$ such that:

$$\hat{u}_i = \begin{cases} 0, & \text{if } \frac{\Pr(\mathbf{y}, \hat{u}_0^{i-1} | u_i = 0)}{\Pr(\mathbf{y}, \hat{u}_0^{i-1} | u_i = 1)} > 1 \\ 1, & \text{otherwise,} \end{cases} \tag{2}$$

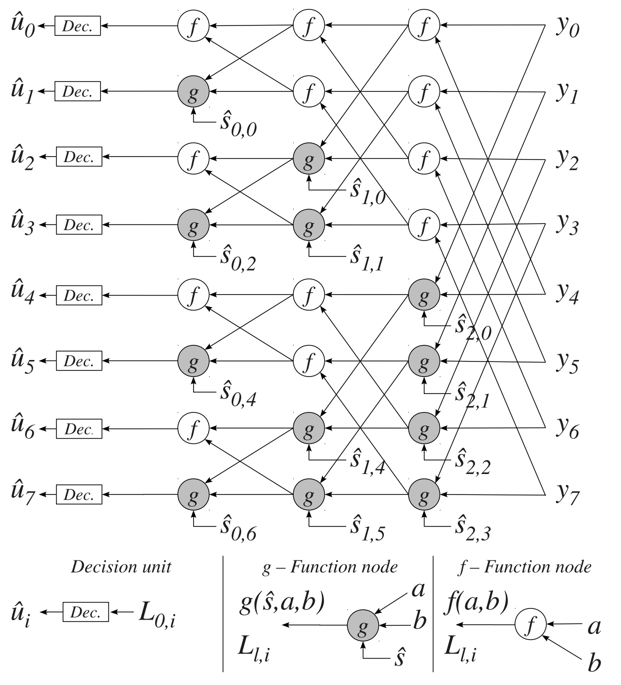

Fig. 2. Butterfly-based SC decoder with $N = 8$.

where $\Pr(\mathbf{y}, \hat{u}_0^{i-1} | u_i = b)$ is the probability that $\mathbf{y}$ was received and the previously decoded bits are $\hat{u}_0^{i-1}$, given the currently decoded bit is $b$, where $b \in \{0, 1\}$. The ratio of probabilities in (2) represents the likelihood ratio (LR) of bit $\hat{u}_i$.

## III. SC Decoding Scheduling

The SC decoding algorithm successively evaluates the LR $L_i$ of each bit $\hat{u}_i$. Arıkan showed that these LR computations can be efficiently performed in a recursive manner by using a data flow graph which resembles the structure of a fast Fourier transform. That structure, shown in Fig. 2, is named a butterfly-based decoder. Messages passed in the decoder are LR values denoted as $L_{l,i}$, where $l$ and $i$ correspond to the graph stage index and row index, respectively. In addition, $L_{0,i} = L(\hat{u}_i)$ and $L_{n,i}$ is the LR directly calculated from the channel output $y_i$. The nodes in the decoder graph calculate the messages using one of two functions:

$$L_{l,i} = \begin{cases} f\left(L_{l+1,i}; L_{l+1,i+2^l}\right) & \text{if } B(l,i) = 0 \\ g\left(\hat{s}_{l,i-2^l}; L_{l+1,i-2^l}; L_{l+1,i}\right) & \text{if } B(l,i) = 1, \end{cases} \tag{3}$$

where $\hat{s}$ is a modulo-2 partial sum of decoded bits, $B(l,i) \triangleq \frac{i}{2^l} \bmod 2$, $0 \le l < n$, and $0 \le i < N$. In the LR domain, functions $f$ and $g$ can be expressed as:

$$f(a,b) = \frac{1 + ab}{a + b} \tag{4}$$

$$g(\hat{s}, a, b) = a^{1-2\hat{s}} b. \tag{5}$$

Function $f$ can be computed as soon as $a = L_{l+1,i}$ and $b = L_{l+1,i+2^l}$ are available. On the other hand, the computation of $g$ requires knowledge of $\hat{s}$, which can be computed by using the factor graph of the code. For instance, in Fig. 1, $\hat{s}_{2,1}$ is estimated by propagating $\hat{u}_0^3$ in the factor graph: $\hat{s}_{2,1} =$

$\hat{u}_1 \oplus \hat{u}_3$. This partial sum of $\hat{u}_0^3$ is then used to compute $L_{2,5} = g(\hat{s}_{2,1}; L_{3,1}; L_{3,5})$.

The need for partial sum computations causes strong data dependencies in the SC algorithm. This constrains the order in which the LRs can be computed in the graph. Fig. 3 shows the scheduling of the decoding process for $N = 8$ using a butterfly-based SC decoder. At each clock cycle (CC), LRs are evaluated by computing function $f$ or $g$. It is assumed here that those functions are calculated as soon as the required data is available. Once the channel information $y_0^{N-1}$ is available on the right hand side of the decoder, bits $\hat{u}_i$ are successively estimated by updating the appropriate nodes of the graph, from right to left. When bit $\hat{u}_i$ is estimated, all partial sums involving $\hat{u}_i$ are updated, allowing future evaluations of function $g$ to be carried out.

One can notice that when stage $l$ is activated, a maximum of $2^l$ operations can be performed simultaneously. Moreover, only one kind of function ($f$ or $g$) is used when activating a given stage. Finally, a stage $l$ is activated $2^{n-l}$ times during the decoding process of a vector. As a consequence, assuming one clock cycle per stage activation, the total number of clock cycles required to decode a vector is:

$$\mathfrak{L}_{ref} = \sum_{l=0}^{n-1} 2^{n-l} = 2N - 2. \tag{6}$$

Despite the seemingly parallel structure of this decoder, strong data dependencies constrain the decoding process and make this decoder quite inefficient. For instance, if we define an active node as one whose inputs are ready, allowing it to perform its operations, then only a fraction of the nodes are actually active during each decoding clock cycle, as shown in Fig. 3. In order to characterize the efficiency of an SC decoder architecture, we use the utilization rate, $\alpha$, which represents the average number of active nodes per clock cycle:

$$\alpha \triangleq \frac{\text{total number of node updates}}{\text{computational resource complexity} \times \text{computation time}}. \tag{7}$$

In SC decoding, $N \log_2 N$ node updates are required to decode one vector. A butterfly-based SC decoder performs this amount of computation with $N \log_2 N$ node processors which are used during $2N-2$ clock cycles; its utilization rate is thus:

$$\alpha_{\text{ref}} = \frac{N \log_2 N}{N \log_2 N (2N-2)} \approx \frac{1}{2N}. \tag{8}$$

The utilization rate rapidly decreases towards 0 as $N$ grows. This indicates potential for a more efficient utilization of processing resources.

## IV. State-of-the-Art SC Decoder Architectures

Since polar codes are a fairly new coding scheme, very few architectures have been proposed for their implementation in hardware. This section gives an overview of existing decoder architectures and describes their main properties.

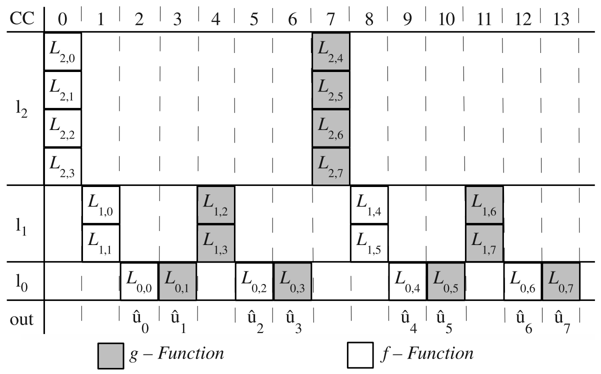

Fig. 3. Scheduling for the butterfly-based SC decoder with $N = 8$.

### A. Tree SC Decoder

Section III showed that whenever a stage $l$ is activated, only $2^l$ nodes of the graph are updated. For this reason, the $N$ nodes of each stage $l$ can be implemented using only $2^l$ processing elements (PE) as shown in [12]. A PE is a programmable functional unit able to perform either function $f$ or function $g$. This tree decoder consists of $n$ stages of $2^l$ PEs each, with $0 \leq l < n$. This resource sharing allows a latency identical to that of the butterfly-based decoder ($2N - 2$ clock cycles) with only $N - 1$ PEs associated with $2N$ memory elements (ME) required to store temporary LR calculations and channel LRs. As will be detailed in Section VI-A, resource sharing within a PE enables us to reduce the complexity of a PE: $C_{\text{PE}} \leq C_f + C_g$. Therefore, in the least favorable case ($C_{\text{PE}} = C_f + C_g$), the utilization rate of the tree decoder is:

$$\alpha_{tree} = \frac{N \log_2 N}{2(N-1)(2N-2)} \approx \frac{\log_2 N}{4N}. \tag{9}$$

This improves the utilization rate by a factor of $\frac{\log_2 N}{2}$ over the butterfly-based decoder. However, this architecture does not address the fact that only a single stage is active at any given time, leaving the remaining $n - 1$ stages idle; two architectures proposed in [12] to address this shortcoming will be reviewed in the following sections.

### B. Line SC Decoder

Fig. 3 illustrates that a maximum of $\frac{N}{2}$ computations of functions $f$ and $g$ is actually performed in one clock cycle. Furthermore, only $N - 1$ MEs are required to store partial results, in addition to the $N$ MEs needed to store the input LRs from the channel. In other words, a decoder with only $\frac{N}{2}$ processing elements and $\sim 2N$ MEs can achieve the same latency as a butterfly-based decoder. This architecture, introduced in [12] and named line decoder, consists of $\frac{N}{2}$ PEs. Its utilization rate can be computed as:

$$\alpha_{\text{line}} = \frac{N \log_2 N}{N(2N-2)} \approx \frac{\log_2 N}{2N}. \tag{10}$$

### C. Vector-Overlapped Decoder

It is also possible to use the idle stages of the tree decoder to decode multiple received vectors simultaneously. It was

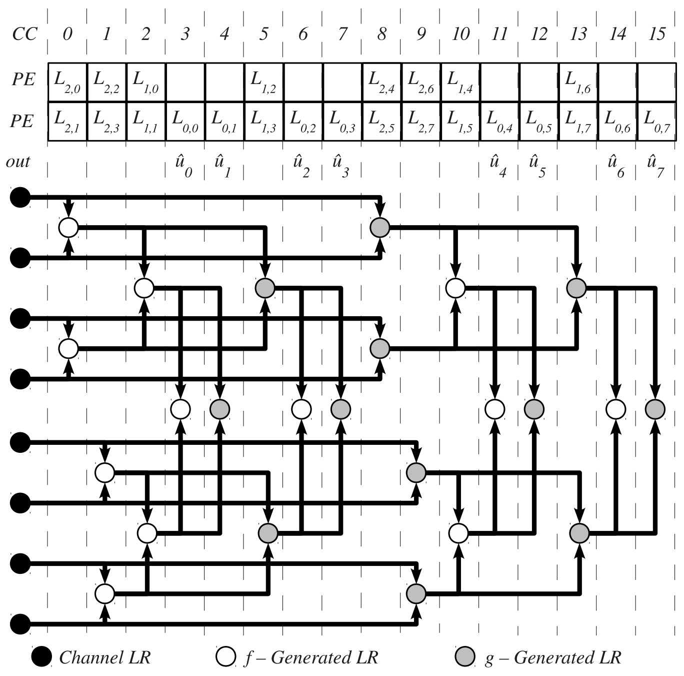
CC
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
PE
PE
out
$L_{2,0}$ $L_{2,2}$ $L_{1,0}$ $L_{1,2}$ $L_{2,4}$ $L_{2,6}$ $L_{1,4}$ $L_{1,6}$
$L_{2,1}$ $L_{2,3}$ $L_{1,1}$ $L_{0,0}$ $L_{0,1}$ $L_{1,3}$ $L_{0,2}$ $L_{0,3}$ $L_{2,5}$ $L_{2,7}$ $L_{1,5}$ $L_{0,4}$ $L_{0,5}$ $L_{1,7}$ $L_{0,6}$ $L_{0,7}$
$\hat{u}_0$ $\hat{u}_1$ $\hat{u}_2$ $\hat{u}_3$ $\hat{u}_4$ $\hat{u}_5$ $\hat{u}_6$ $\hat{u}_7$
Channel LR
f – Generated LR
g – Generated LR

Fig. 4. Scheduling and LR data flow graph of a semi-parallel SC decoder with $N = 8$ and $P = 2$.

shown in [12] that this decoding scheme yields an increased utilization rate:

$$\alpha_{\text{vo}} \approx \frac{V \log_2 N}{4N + (2V+2)\log_2(\frac{V+1}{4})}, \tag{11}$$

where $V$ represents the number of overlapped vectors processed in the decoder. However, this architecture requires that memory be duplicated $V$ times, which becomes impractical when $V \gg 1$.

## V. Semi-Parallel SC Decoder

### A. Principle and Scheduling

The architectures presented in Section IV share the common flaw that their utilization rate decreases as the code length increases. We therefore propose to improve $\alpha$ by limiting the number of PEs implemented in the decoder.

Considering that, in the line decoder, the $\frac{N}{2}$ PEs are only all activated simultaneously twice during the decoding of a vector—regardless of the code size—it follows that we can increase the utilization rate of a decoder by reducing the number of PEs without significantly impacting throughput. For example, a modified line decoder implementing only $\frac{N}{4}$ PEs will only incur a 2 clock cycle penalty compared to a full line decoder. This simplified architecture, which we name semi-parallel decoder, has a lower complexity at the expense of a small increase in latency.

This approach can be generalized to a smaller number of PEs. Let us define $P < \frac{N}{2}$ as the number of implemented PEs. Fig. 4 describes the scheduling of a semi-parallel decoder with $\{P = 2; N = 8\}$, where it can be observed that this scheduling only requires 2 additional clock cycles over the equivalent line decoder. Indeed, the computations performed during clock cycles $\{0, 1\}$ and $\{8, 9\}$ are executed in a single clock cycle in a line decoder.

In addition, Fig. 4 depicts the data flow graph of LRs generated during the decoding process for $\{N = 8; P = 2\}$. Data generated during CC $= \{0, 1\}$ is not needed after CC $= 5$ and can therefore be replaced by the data generated in CC $= \{8, 9\}$. It follows that the same memory element can be used to store the results of both computations.

In general, the memory requirements remain unchanged in comparison with the line decoder: the semi-parallel SC decoder needs $N$ MEs for the channel information $\mathbf{y}$, and $N - 1$ MEs for intermediate results. Consequently, for a code of length $N$, the memory requirements of the semi-parallel decoder remain constant regardless of the number of implemented PEs.

It should be noted that the data dependencies involving $\hat{s}$ are not represented in Fig. 4. Therefore, even though it may seem that the data generated at CC $= \{8, 9\}$ could have been produced earlier, this is not the case as the value of $\hat{u}_3$ must be known in order to compute $L_{2,4}, L_{2,5}, L_{2,6}$ and $L_{2,7}$.

### B. Complexity vs Latency

Although the reduced number of processing elements implemented in a semi-parallel SC decoder increases latency, this reduction only affects the processing of stages that require more than $P$ node updates. Starting from this general observation, we now quantify the impact of reducing the number of processing elements on latency. In order to keep some regularity in the scheduling, we assume that the number of implemented PEs, $P$, is a power of 2 and $P = 2^p$.

In a line SC decoder, every stage $l$ of the graph is updated $2^{n-l}$ times and it takes a single clock cycle to perform those updates since a sufficient number of PEs is implemented. However, in a semi-parallel decoder, a limited number of PEs is implemented and it may take several clock cycles to complete a stage update. The stages satisfying $2^l \leq P$ are not affected and their latency is unchanged. However, for stages requiring more LR computations than there are implemented PEs, it takes multiple clock cycles to complete the update. Specifically, $\frac{2^l}{P}$ clock cycles are required to update a stage $l$ with $P$ implemented PEs.

Therefore, the total latency of a semi-parallel decoder is:

$$\begin{aligned}\mathfrak{L}_{\text{SP}} &= \underbrace{\sum_{l=0}^{p} 2^{n-l}}_{\text{non-affected stages}} + \underbrace{\sum_{l=p+1}^{n-1} 2^{n-l} 2^{l-p}}_{\text{affected stages}} \\ &= 2N(1 - \frac{1}{2P}) + (n - p - 1)\frac{N}{P} \\ &= 2N + \frac{N}{P}\log_2(\frac{N}{4P}). \end{aligned} \tag{12}$$

As expected, the latency of the semi-parallel decoder increases as the number of implemented PEs decreases. However, this latency penalty is not linear with respect to $P$. In order to quantify the trade-off between the latency of the semi-parallel decoder ($\mathfrak{L}_{\text{SP}}$) and $P$, we define the relative-speed

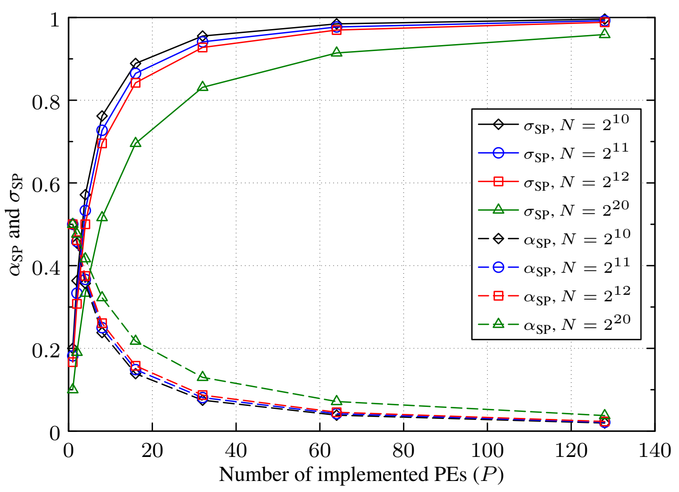

Fig. 5. Utilization rate ($\alpha_{\mathrm{SP}}$) and relative-speed factor ($\sigma_{\mathrm{SP}}$) for the semi-parallel SC decoder.

factor of a semi-parallel SC decoder as:

$$\sigma_{\mathrm{SP}} = \frac{\mathfrak{L}_{\mathrm{ref}}}{\mathfrak{L}_{\mathrm{SP}}} = \frac{2P}{2P + \log_2 \frac{N}{4P}}, \tag{13}$$

where $\mathfrak{L}_{ref}$ is defined in (6).

This metric defines the throughput attainable by the semi-parallel decoder, relative to that of the line decoder. One should notice that the definition of $\sigma_{\mathrm{SP}}$ implicitly assumes that both decoders can be clocked at the same frequency: $T_{\text{clk-line}} = T_{\text{clk-SP}}$. In Section VII, synthesis results show that due to the large number of PEs in the line decoder, we actually have $T_{\text{clk-line}} > T_{\text{clk-SP}}$. Consequently, (13) represents the least favorable case for the semi-parallel architecture.

The utilization rate of a semi-parallel decoder, on the other hand, is defined as:

$$\alpha_{\mathrm{SP}} = \frac{N \log_2 N}{2P(2N + \frac{N}{P}\log_2(\frac{N}{4P})} = \frac{\log_2 N}{4P + 2\log_2(\frac{N}{4P})}. \tag{14}$$

Fig. 5 plots $\sigma_{\mathrm{SP}}$ and $\alpha_{\mathrm{SP}}$ as $P$ is varied from 1 to 128 for code lengths $N = \{2^{10}, 2^{11}, 2^{12}, 2^{20}\}$. One should first notice that both metrics vary only marginally with respect to the code length. Moreover, these curves show that $\sigma_{\mathrm{SP}}$ is close to 1 even for small values of $P$. This means that a small number of PEs is sufficient to achieve a throughput similar to that of a line decoder. For example, the semi-parallel decoders presented in this figure can achieve >90% of the throughput of a line SC decoder using only 64 PEs. The number of PEs is reduced by a factor $\frac{N}{2P}$, which is 8192 for $N = 2^{20}$ and $P = 64$. For $P = 64$ and $N = 1024$, the utilization rate ($\alpha_{SP} = 3.5\%$) is improved by a factor 8 compared to the line decoder. This demonstrates a more efficient use of processing resource during the decoding process.

This dramatic complexity reduction makes the size of the processing resources very small in comparison to that of the memory resources required by this architecture, as discussed in Section VII.

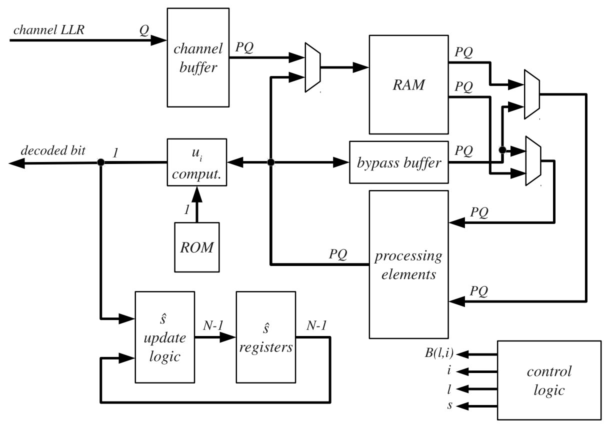

Fig. 6. Semi-parallel SC decoder architecture

## VI. Hardware Architecture

This section provides a detailed description of the various modules comprised in the semi-parallel decoder, whose top-level architecture is illustrated in Fig. 6.

### A. Processing Elements

SC polar code decoders carry out their likelihood estimations using update rules (4) and (5). However, those equations require divisions and multiplications, which makes them unsuitable for a hardware implementation. To reduce complexity, reference [12] suggested replacing these LR updates with equivalent functions in the logarithmic domain. In this paper, we denote log likelihood ratio (LLR) values by $\lambda_X = \log(X)$, where $X$ is an LR.

In the LLR domain, functions $f$ and $g$ become the sum-product algorithm (SPA) equations:

$$\lambda_f(\lambda_a, \lambda_b) = 2\tanh^{-1}\left(\tanh\left(\frac{\lambda_a}{2}\right)\tanh\left(\frac{\lambda_b}{2}\right)\right) \tag{15}$$

$$\lambda_g(\hat{s}, \lambda_a, \lambda_b) = \lambda_a(-1)^{\hat{s}} + \lambda_b. \tag{16}$$

At first glance, $\lambda_f$ appears more complex than its counterpart (4) as it involves hyperbolic functions. However, [14] showed that it could be approximated using the minimum function, yielding the simpler min-sum (MS) equations:

$$\lambda_f(\lambda_a, \lambda_b) \approx \psi^\star(\lambda_a)\psi^\star(\lambda_b)\min(|\lambda_a|, |\lambda_b|) \tag{17}$$

$$\lambda_g(\hat{s}, \lambda_a, \lambda_b) = \lambda_a(-1)^{\hat{s}} + \lambda_b, \tag{18}$$

where $|X|$ represents the magnitude of variable $X$ and $\psi^\star(X)$, its sign, defined as:

$$\psi^\star(X) = \begin{cases} 1 & \text{when } X \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{19}$$

Equations (17) and (18) indeed suggest a much simpler hardware implementation than their counterparts in the LR domain. In addition, Fig. 7 shows that although (17) involves an approximation, its impact on decoding performance is minimal.

Hardware-wise, we propose merging $\lambda_f$ and $\lambda_g$ into a single processing element using the sign and magnitude (SM)

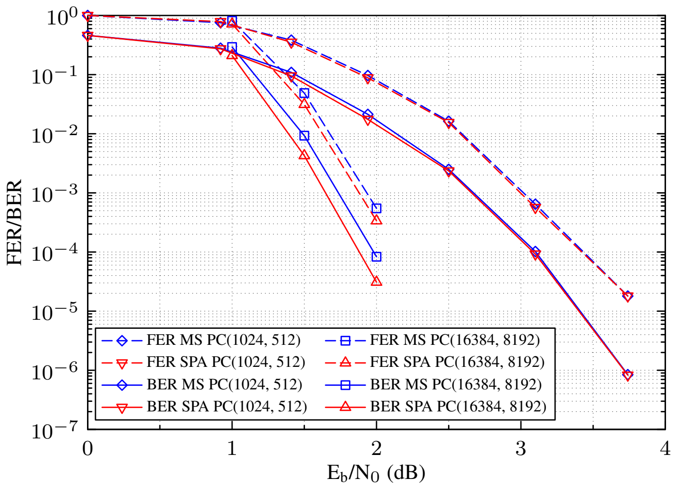

Fig. 7. BER/FER performance with and without using the min-sum approximation on an AWGN channel, for $N = 2^{10}$ and $N = 2^{14}$.

representation for LLR values as it simplifies the realization of (17):

$$\psi(\lambda_f) = \psi(\lambda_a) \oplus \psi(\lambda_a) \tag{20}$$
$$|\lambda_f| = \min(|\lambda_a|, |\lambda_b|), \tag{21}$$

where $\psi(X)$, like $\psi^\star(X)$, describes the sign of variable $X$, although in a way that is compatible with the sign and magnitude representation:

$$\psi(X) = \begin{cases} 0 & \text{when } X \geq 0 \\ 1 & \text{otherwise.} \end{cases} \tag{22}$$

These computations are implemented using a single XOR gate and a $(Q-1)$-bit compare-select (CS) operator, as shown in Fig. 8. Function $\lambda_g$, on the other hand, is implemented using an SM adder/subtractor. In SM format, $\psi(\lambda_g)$ and $|\lambda_g|$ depend not only on $\hat{s}$, $\psi(\lambda_a)$, $\psi(\lambda_b)$, $|\lambda_a|$, and $|\lambda_b|$, but also on the relation between the magnitudes $|\lambda_a|$ and $|\lambda_b|$. For instance, if $\hat{s} = 0$, $\psi(\lambda_a) = 0$, $\psi(\lambda_b) = 1$, and $|\lambda_a| > |\lambda_b|$, then $\psi(\lambda_g) = \psi(\lambda_a)$ and $|\lambda_g| = |\lambda_b| - |\lambda_a|$. This relation between $|\lambda_a|$ and $|\lambda_b|$ is represented by bit $\gamma_{ab}$, which is generated using a magnitude comparator:

$$\gamma_{ab} = \begin{cases} 1 & \text{if } |\lambda_a| > |\lambda_b| \\ 0 & \text{otherwise.} \end{cases} \tag{23}$$

The sign $\psi(\lambda_g)$ depends on four binary variables: $\psi(\lambda_a)$, $\psi(\lambda_b)$, $\hat{s}$ and $\gamma_{ab}$. By applying standard logic minimization techniques to the truth table of $\psi(\lambda_g)$, we obtain the following simplified boolean equation:

$$\psi(\lambda_g) = \overline{\gamma_{ab}} \cdot \psi(\lambda_b) + \gamma_{ab} \cdot (\hat{s} \oplus \psi(\lambda_a)), \tag{24}$$

where $\oplus$, $\cdot$ and $+$ represent binary XOR, AND and OR, respectively.

As shown in Fig. 8, the computation of $\psi(\lambda_g)$ only requires an XOR gate and a multiplexer, while $\gamma_{ab}$ is already available from the CS operator, which is shared between $\lambda_f$ and $\lambda_g$.

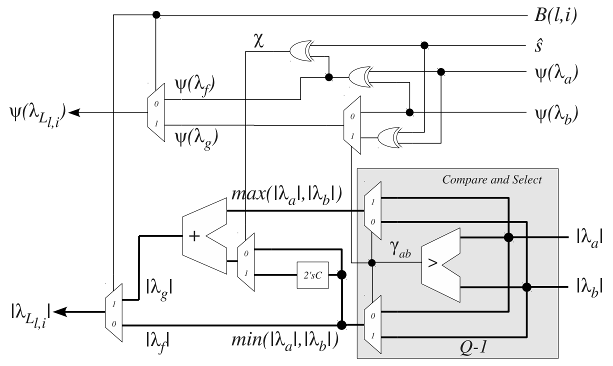

Fig. 8. Sign and magnitude processing element architecture.

On the other hand, the magnitude $|\lambda_g|$ is the addition or subtraction of $\max(|\lambda_a|, |\lambda_b|)$ and $\min(|\lambda_a|, |\lambda_b|)$:

$$|\lambda_g| = \max(|\lambda_a|, |\lambda_b|) + (-1)^\chi \min(|\lambda_a|, |\lambda_b|) \tag{25}$$
$$\chi = \hat{s} \oplus \psi(\lambda_a) \oplus \psi(\lambda_b), \tag{26}$$

where bit $\chi$ determines whether $\min(|\lambda_a|, |\lambda_b|)$ should be inverted or not. $|\lambda_g|$ is implemented using an unsigned adder, a multiplexer, and a two's complement operator—used to negate a number so that the unsigned adder can be used to perform subtraction by overflowing—in addition to the shared CS operator.

Finally, the result of the processing element is determined by bit $B(l, i)$, such that:

$$\psi(\lambda_{L_{l,i}}) = \begin{cases} \psi(\lambda_f) & \text{when } B(l,i) = 0 \\ \psi(\lambda_g) & \text{otherwise} \end{cases} \tag{27}$$

$$|\lambda_{L_{l,i}}| = \begin{cases} |\lambda_f| & \text{when } B(l,i) = 0 \\ |\lambda_g| & \text{otherwise.} \end{cases} \tag{28}$$

## *B. LLR Memory*

Throughout the decoding process, the PEs compute LLRs which are reused in subsequent steps of the process. To allow this reuse to take place, the decoder must store those intermediate estimates in a memory. It was shown in [12] that $2N - 1$ $Q$-bit memory elements are sufficient to store the received vector and keep track of all the intermediate $Q$-bit LLR estimates. This memory can be conceptually represented as a tree structure in which each level stores LLRs for a stage $l$ of the decoding graph, with $0 \leq l \leq n$. Channel LLRs are stored in the leaves of the tree whereas decoded bits are read from the root.

In order to avoid introducing additional delays in decoding, we seek to have single-clock-cycle operation of the processing elements, which requires that it be possible to simultaneously read their inputs and write their outputs in a single clock cycle. A straightforward solution would be to implement these parallel accesses using a register-based architecture, as was proposed in [12] for the line decoder. However, preliminary synthesis results demonstrated that the routing and multiplexing requirements associated with this approach were not suitable for the implementation of the very large code lengths required by polar codes. We thus instead propose to implement this parallel access using random access memory (RAM). In a polar codes decoder, the PEs consume twice as much information as they produce. Therefore, our semi-parallel architecture uses a dual-port RAM configured with a write port of width $PQ$ and a read port of width $2PQ$, and a specific placement of data in memory. Note that using RAM has the added benefit that it also significantly reduces the area per stored bit over the register-based approach.

Within each memory word, LLRs must be properly aligned such that data is presented in a coherent order to the PEs. For example, the $\{N = 8; P = 2\}$ semi-parallel SC decoder shown in Fig. 2 computes $\lambda_{L_{1,0}}$ and $\lambda_{L_{1,1}}$ by accessing a memory word containing $\{\lambda_{L_{2,0}}, \lambda_{L_{2,2}}, \lambda_{L_{2,1}}, \lambda_{L_{2,3}}\}$ in this order—which follows the bit-reversed indexing scheme of [4]—and presenting those LLRs to the PEs. Effecting this bit-reversal throughout in this figure leads to a mirrored decoding graph, as seen in Fig. 9, with bit-reversed vectors for the channel information $\mathbf{x}$ and the decoded output $\hat{\mathbf{u}}$.

This ordering is advantageous since the processing elements only access contiguous values in memory. For example, LLRs $\{\lambda_{L_{2,0}}, \lambda_{L_{2,2}}, \lambda_{L_{2,1}}, \lambda_{L_{2,3}}\}$, discussed earlier, are now located in LLR locations $\{8, 9, 10, 11\}$ in memory, after the received vector $\mathbf{x}$. This observation holds true for any nodes emulated by a PE in the mirrored graph. This means that the decoder can now feed a contiguous block—word 2 in Fig. 10, in our example—of memory directly to the PEs. Note that this means that the received vector $\mathbf{y}$ must be stored in bit-reversed order in memory, but this can easily be done by modifying the order in which the encoder sends the codeword over the channel.

In order to simplify the memory address generation, we use the particular structure and data placement illustrated in Fig. 10, where the unused values computed by the PEs in the stages satisfying $l \leq p$ are also stored in memory, to preserve a regular structure. It allows for a direct connection between the dual-port RAM and the PEs, without using complex multiplexing logic or interconnection networks. On the other hand, this layout has an overhead of $Q(2P \log_2 P{+}1)$ bits over the minimum required amount of memory. Fortunately, this overhead is constant with respect to $N$, which implies a shrinking proportion of the overall memory requirements as code length increases, for a fixed $P$. For example, our approach needs $769Q$ additional bits of RAM for $P = 64$, regardless of code size. The overhead for a $\{N = 1024; P = 64\}$ decoder can be computed to be $\sim$37.6% while that percentage shrinks to $\sim$1.17% for a $N = 32,768$ decoder with the same parameters.

## C. Bypass Buffer

When a graph stage $l$, where $l \leq p$, is processed, the data generated by the processing elements needs to be reused immediately after being produced. If we assume that the LLR RAM does not have write-through capability, then a $PQ$-bit buffer register is required to loop these generated data directly back to the input of the PEs, while preventing a combinatorial loop in the circuit.

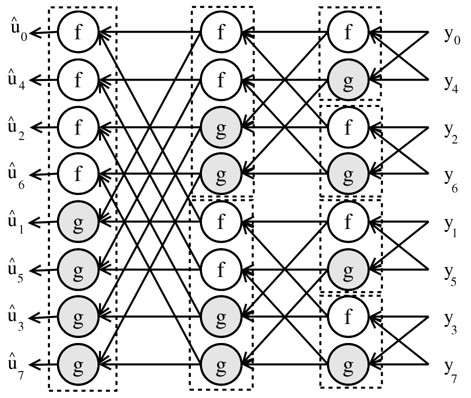

Fig. 9. Mirrored decoding graph for $N = 8$.

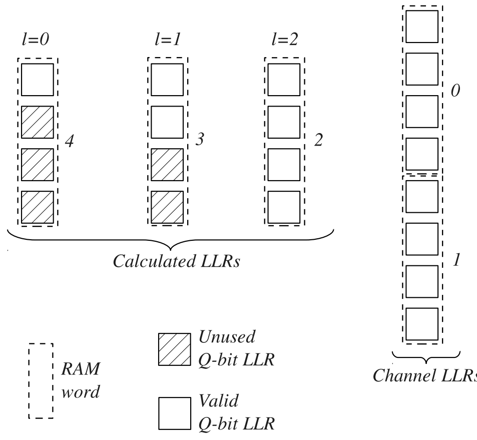

Fig. 10. Organization of the LLR memory for $N = 8$ and $P = 2$ with uniform memory block size.

## D. Channel Buffer

Since the RAM operates on $PQ$-bit words natively, a buffer was introduced at the input of the decoder to accumulate $P$ $Q$-bit LLRs from the channel before writing them to RAM as a memory word. This buffer allows the decoder to receive channel LLRs serially while keeping the interface to the RAM uniform for both the PEs and the channel inputs.

## E. Partial Sum Registers

Throughout the decoding process, the processing elements must be provided with specific partial sums as part of $\lambda_g$. Furthermore, whenever a bit $\hat{u}_i$ is estimated, many such partial sums typically require updating.

Unlike the likelihood estimates stored in the LLR memory, partial sums do not have a regular structure that would allow them to be packed in memory words. As a result, storing them in a RAM would lead to scattered memory accesses requiring multiple clock cycles. To avoid lowering the throughput of the decoder, we instead store them in registers. Each $g$ node of the decoding graph is mapped to a specific flip-flop in the partial sum register. The partial sum update logic module, described in Section VI-F, updates the values of this register each time a bit $\hat{u}_i$ is estimated.

We found that $N-1$ bits suffice to store the required partial sums if we time multiplex the use of those memory locations

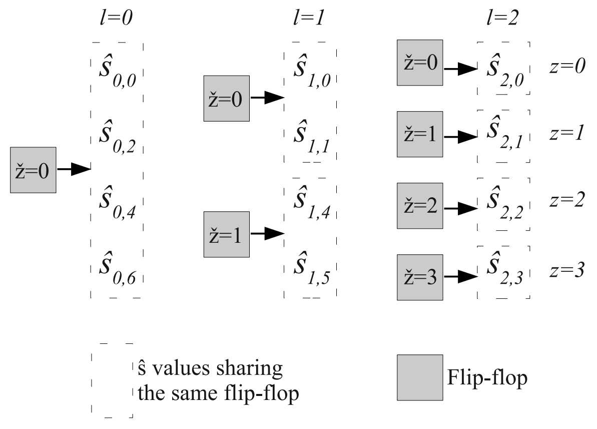

Fig. 11. Architecture of the partial sum registers with $N = 8$.

between all the nodes emulated by a given processing element. We noticed that the $g$ nodes could be grouped into $2^l$ groups in each stage $l$, each group only requiring a single bit of memory to store their partial sums, for a total of $N - 1$ memory bits. For example, looking at Fig. 2, we notice that all partial sums in stage 0 can be stored in a single bit, provided that this bit be reset at each odd clock cycle of the decoding process. Similarly, we see that the nodes of stage 1 can be grouped into 2 partial sums, provided that the first two partial sums be stored in the same location (same for the last two), and that both locations be reset at clock cycles 3 and 7. Fig. 11 shows the mapping of each partial sum to one of the $N - 1$ 1-bit flip-flops, for $N = 8$.

### *F. Partial Sum Update Logic*

Every computation of function $\lambda_g$ requires a specific input $\hat{s}_{l,z}$ corresponding to a sum of a subset of the previously estimated bits $\hat{u}_0^{N-1}$, per (5). This subset of $\hat{u}_0^{N-1}$ needed for the $g$ node with index $z$ when decoding bit $i$ in stage $l$ is determined according to the indicator function

$$I(l,i,z) = \overline{B(l,i)} \cdot \prod_{v=l}^{n-2} \left(\overline{B(v,z) \oplus B(v+1,i)}\right) \cdot \prod_{w=0}^{l-1} \left(\overline{B(w,z)} + B(w,i)\right), \qquad (29)$$

where $\cdot$ and $\prod$ are the binary AND operation, $+$ the binary OR operation, and $B(a,b) \triangleq \frac{b}{2^a} \bmod 2$. An estimated bit $\hat{u}_i$ is included in the partial sum if the corresponding indicator function value is 1. For example, the values of the indicator function when $N = 8$ and $l = 2$ are

$$I(2,i,z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

and the first four partial sums are

$$\begin{aligned} \hat{s}_{2,0} &= \hat{u}_0 \oplus \hat{u}_1 \oplus \hat{u}_2 \oplus \hat{u}_3, \\ \hat{s}_{2,1} &= \hat{u}_1 \oplus \hat{u}_3, \\ \hat{s}_{2,2} &= \hat{u}_2 \oplus \hat{u}_3, \text{ and} \\ \hat{s}_{2,3} &= \hat{u}_3. \end{aligned}$$

Using the indicator function, the general form of the partial sum update equation is

$$\hat{s}_{l,z} = \bigoplus_{i=0}^{N-1} \hat{u}_i \cdot I(l,i,z), \qquad (30)$$

where $\bigoplus$ is the binary XOR operation.

In terms of hardware implementation, since each evaluation of function $g$ requires a different partial sum $\hat{s}_{l,z}$, flip-flops are used to store all needed combination. Since the hard decisions $\hat{u}_i$ are obtained sequentially as decoding progresses, the contents of flip-flop $(l,z)$ is produced by adding $\hat{u}_i$ to the current flip-flop value if $I(l,i,z) = 1$. Otherwise, the flip-flop value remains unchanged.

Using the time multiplexing described in the previous section, the indicator function can be further simplified:

$$I'(l,i,\check{z}) = \overline{B(l,i)} \cdot \prod_{v=0}^{l-1} \left(\overline{B(l-v-1,\check{z})} + B(v,i)\right), \qquad (31)$$

where $\check{z}$ corresponds to the index of the flip-flops within a stage. Since time multiplexing is used in the partial sum registers, a flip-flop in stage $l$ effectively holds $2^{n-l-1}$ partial sums, at different points in the decoding process. Both indexing methods are illustrated in Fig. 11.

### *G. Frozen Channel ROM*

A polar code is completely defined by its code length $N$ and the indices of the frozen bits in the vector $\mathbf{u}$.

Our architecture stores the indices of those frozen bits in a 1-bit ROM of size $N$. Every generated soft output $\lambda_{L_{0,i}}$ passes through the $u_i$ computation block, which sets the output to the frozen bit value if indicated by the contents of ROM, or performs a threshold-detection-based hard decision otherwise. This ROM is addressed directly using the current decoded bit $i$.

Note that this implementation opens up the possibility of easily reprogramming the decoder for different operational configurations by replacing the contents of this ROM with a different set of frozen bits; the architecture of the decoder is decoupled from its operational parameters. Since a polar code is created for given channel conditions—e.g.: the noise variance for the AWGN channel—replacing the ROM by a RAM would allow the indices of the frozen bits to be changed, adapting to the current channel conditions.

### *H. Controller*

The controller module coordinates the various phases and components of the decoding process. To support those tasks, it must compute different control signals, such as the current

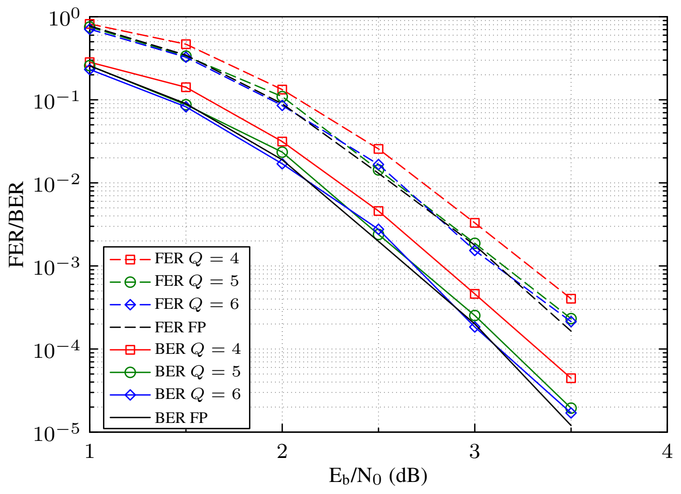

Fig. 12. Effect of quantization on error-correction performance for the (1024,512) polar code.

decoded bit $i$, the current stage $l$, and the portion $\epsilon$ of a stage being processed, $0 \leq \epsilon < \lceil \frac{2^l}{P} \rceil$.

Being sequential, the calculation of $i$ is straightforward, using a simple $n$-bit counter enabled each time the decoder reaches $l = 0$.

The stage number $l$, on the other hand, involves a computation slightly more involved—as illustrated in Fig. 4—because it relies on both $i$ and $\epsilon$. Whenever $\epsilon \geq \lceil \frac{2^l}{P} \rceil$, $i$ is updated and $l$ is modified such that it is set to the index of the first bit set in the binary representation of the updated value of $i$, or to $n-1$ when $i$ wraps around $N-1$.

Since stage $l$ needs $2^l$ calculations, it requires $\lceil \frac{2^l}{P} \rceil$ uses of the processing elements—and thus the same number of clock cycles—to perform all computations associated with this stage. Therefore, a simple counter is used to keep track of $\epsilon$, with a reset conditioned on the above condition.

The controller also controls a multiplexer located at the input of the RAM. During the input phase, this multiplexer selects the channel buffer as the input to the RAM; during the decoding, it selects the outputs of the processing elements instead.

Then, the controller selects the input of the processing elements using a set of multiplexers located at the output of the RAM, depending on the value of the read and write addresses provided to the RAM. If those addresses point to overlapping locations in memory, it selects the bypass buffer as part of the inputs to the PEs: it bypasses the low part (resp. high part) of the memory output if the least significant bit of the write address is 0 (resp. 1).

Finally, the controller selects the function ($f$ or $g$) performed by the processing elements based on $B(l, i)$, as described in Section III.

## VII. Implementation Results

We start this section by investigating the effects of implementation parameters on error-correction performance. Then, we present the implementation results for FPGA and ASIC, comparing them with other works in the literature.

TABLE I
Resource utilization of the BP and SC decoders on the Xilinx Virtex IV XC4VSX25-12 using the (1024, 512) polar code.

| Algorithm | LUT | FF | BRAM | T/P (Mbps) |
|---|---|---|---|---|
| BP [15] | 2,794 | 1,600 | 12 | 2.78 |
| SP-SC | 2,600 | 1,181 | 5 | 22.22 |

### A. *Decoding Performance*

Two factors impact the error-correction performance of the hardware decoder: the number of quantization bits $Q$ used to represent LLRs and the maximum channel symbol magnitude allowed before saturation occurs.

Fig. 12 shows the BER for the (1024, 512) code using $Q = \{4, 5, 6\}$ quantization bits, in addition to the BER of the non-quantized decoder. The quantized decoder inputs were limited to the range [-2, 2] and values outside this range were clipped. From the figure, we note that when using 4 quantization bits, the BER is degraded by less than 0.25 dB. However, increasing $Q$ to 5 results in performance almost matching that of the non-quantized decoder. Further increasing $Q$ offers no benefit for this decoder. We therefore use $Q = 5$ and saturate the decoder input to [-2, 2] for the implementation analysis.

### B. *FPGA Implementation*

To the best of our knowledge, there is only one other hardware implementation of a polar code decoder [15] in the literature. That work implemented a semi-parallel belief-propagation (BP) decoder on a Xilinx FPGA and presented throughput and error-correction performance results. The error-correction performance of this BP decoder was provided for 50 decoding iterations without early termination; it matches that of our SC decoder using 4 bits of quantization. However, the throughput results of [15] were provided for 5 iterations of the decoder; we therefore scale them down by a factor of 10 to allow a comparison of the BP and SC decoders at the same error-correction performance. From Table I, which summarizes the resources used by a BP and an SC decoder having the same error-correction performance, we see that the proposed SC decoder utilizes fewer resources—especially memory resources—while providing an information throughput an order of magnitude greater than that of the BP decoder. The BP decoder may reduce the number of iterations at high SNR to increase its throughput. However, such a strategy would increase the implementation complexity and was not reported in [15].

In addition to the $N = 1024$ code, we also implemented codes up to $N = 2^{17} = 131,072$ in length. Table II presents the synthesis results for these codes on an Altera Stratix IV FPGA. We chose the number of PEs implemented to be 64 as it was able to achieve over 90% of the throughput of a line decoder running at the same frequency, and $Q$ was set to 5. We also included the results with $P = 16$ for the $N = 1024$ code because the throughput loss due to the semi-parallel nature of the decoder is offset by a higher clock frequency. From the table, we note that the number of look-up tables (LUT), flip-flops (FF), and RAM bits grows linearly in code length $N$.

TABLE II
FPGA SYNTHESIS RESULTS ON THE ALTERA STRATIX IV EP4SGX530KH40C2 USING THE POLAR CODES OF DIFFERENT LENGTHS.

| $N$ | $P$ | LUT | FF | RAM (bits) | $f$ (MHz) | T/P (Mbps) |
|---|---|---|---|---|---|---|
| $2^{10}$ | 16 | 2,888 | 1,388 | 11,904 | 196 | $87R$ |
| $2^{10}$ | 64 | 4,130 | 1,691 | 15,104 | 173 | $85R$ |
| $2^{11}$ | 64 | 5,751 | 2,718 | 26,368 | 171 | $83R$ |
| $2^{12}$ | 64 | 8,635 | 4,769 | 48,896 | 152 | $73R$ |
| $2^{13}$ | 64 | 16,367 | 8,868 | 93,952 | 134 | $64R$ |
| $2^{14}$ | 64 | 29,897 | 17,063 | 184,064 | 113 | $53R$ |
| $2^{15}$ | 64 | 58,480 | 33,451 | 364,288 | 66 | $31R$ |
| $2^{16}$ | 64 | 114,279 | 66,223 | 724,736 | 56 | $26R$ |
| $2^{17}$ | 64 | 221,471 | 131,764 | $1,445,632$ | 10 | $4.6R$ |

TABLE III
COMPARISON WITH AN 802.16E (WIMAX) CTC DECODER [15] ON A XILINX XC5VLX85.

| Decoder | LUT | FF | BRAM | DSP | $f$ (MHz) | T/P (Mbps) |
|---|---|---|---|---|---|---|
| CTC [15] | 6,611 | 6,767 | 9 | 4 | 160 | 30 |
| PC (4096) | 7,356 | 4,293 | 6 | 0 | 94 | 39 |
| PC (8192) | 12,154 | 8,386 | 10 | 0 | 68 | 28 |

The frequency decreases almost linearly in the logarithm of the code length $n$.

Analysis revealed that the reduction in operating frequency is largely due to the partial-sum update logic. Since the implementation of the semi-parallel decoder, except the contents of the frozen-bit ROM, remains the same regardless of the code rate $R$ used, the operating clock frequency is independent of $R$. For example, the $N = 2^{16}$ code has an information-bit throughput equal to $26.35R$ Mbps. Hence, high values of $R$ can be used to reach a throughput close to 26.35 Mbps. The information throughput values as a function of the code rate $R$ for the other code lengths are also provided in Table II.

The throughput of the line decoder is given by $0.5fR$. As such, it will be faster than a semi-parallel decoder operating at the same frequency. However, as indicated by the results for the $N = 1024$ code, increasing $P$ reduces the operating frequency. As a result, a line-decoder—if it is implementable given the available hardware resources—will have a lower operating frequency and a higher complexity than a semi-parallel decoder. The largest line decoder that was successfully synthesized for this FPGA was of length $N = 2^{12}$.

To compare the resource utilization and speed of the semi-parallel polar decoder to those of other codes, we have elected to use the convolutional turbo code (CTC) decoder for the 802.11e (WiMAX) standard presented in [15]. We found that a (4096, 2048) polar offered FER within 0.2 dB of the (960, 480) CTC WiMAX code and an (8192, 4096) polar code was within 0.1 dB; therefore, we compare the throughput and synthesis results of decoders for these three codes in Table III. For the (4096, 2048) code, the implementation complexity is comparable: SP-SC requires more LUTs, but fewer flip-flops and block-RAM banks. The throughput of SP-SC was 30% higher than that of the CTC decoder, even thought it had a lower clock frequency. The (8192, 4096) SP-SC decoder had higher complexity than the CTC one and its throughput was 7% lower.

## C. ASIC Implementation

In addition to the FPGA implementation, we have also synthesized the semi-parallel SC decoder as an ASIC using *Cadence RTL Compiler* with a 65nm TSMC standard-cell library. Table IV presents these synthesis results for different code lengths with a target clock frequency set to 500 MHz. We note that the majority of the area is occupied by the circuitry required to store the LLR results of the calculations. This is in large part due to the use of registers instead of RAM as we did not have access to a RAM compiler for the 65nm technology. The partial-sum update circuitry also occupied a sizable portion of the design as a result of the large amount of multiplexing and logic required by these operations. Finally, the throughput values for these designs are also given in Table IV and are all greater than $240R$ Mbps.

## VIII. CONCLUSION

In this paper, we proposed a new semi-parallel architecture for successive cancellation decoding of polar codes. We first showed that butterfly-based and line decoder architectures have a very low utilization rate. Then, we took advantage of the very regular structure of polar codes to increase processing resource sharing further. The throughput penalty is showed to be small in comparison with the drastic reduction in processing complexity. The resulting semi-parallel decoder architecture was then detailed and implemented on FPGA and ASIC targets. FPGA implementation shows that polar code decoders are getting close to existing CTC decoder in terms of hardware resources utilization and throughput.

Future work will focus on increasing the clock frequency by simplifying the partial sum update logic and associated registers, which are now the main contributors to the critical path in successive cancellation decoding of polar codes.

## REFERENCES

[1] C.E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, Jul.-Oct. 1948.

[2] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit error-correcting coding and decoding: Turbo-codes,” in *ICC 93, Geneva*, May 1993, vol. 2, pp. 1064–1070.

[3] R. Gallager, “Low-density parity-check codes,” *IRE Trans. on Information Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962.

[4] E. Arıkan, “Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels,” *IEEE Trans. on Inform. Theory*, vol. 55, no. 7, pp. 3051–3073, Jul. 2009.

[5] E. Sasoglu, E. Telatar, and E. Arıkan, “Polarization for arbitrary discrete memoryless channels,” in *Proc. IEEE Information Theory Workshop ITW*, 2009, pp. 144–148.

[6] H. Mahdavifar and A. Vardy, “Achieving the Secrecy Capacity of Wiretap Channels Using Polar Codes,” *IEEE Trans. on Information Theory*, vol. 57, no. 10, pp. 6428–6443, Oct. 2011.

[7] N. Hussami, R. Urbanke, and S.B. Korada, “Performance of polar codes for channel and source coding,” in *IEEE ISIT 2009*, Jun. 2009.

[8] S.B. Korada and R.L. Urbanke, “Polar Codes are Optimal for Lossy Source Coding,” *IEEE Trans. on Inform. Theory*, vol. 56, no. 4, pp. 1751–1768, Apr. 2010.

[9] Ido Tal and Alexander Vardy, “List decoding of polar codes,” in *Proc. ISIT*, 2011.

[10] R. Mori and T. Tanaka, “Non-binary polar codes using Reed-Solomon codes and algebraic geometry codes,” in *Proc. IEEE Information Theory Workshop (ITW)*, 2010, pp. 1–5.

TABLE IV
ASIC SYNTHESIS RESULTS TARGETTING THE TSMC 65NM PROCESS AT 500 MHZ.

| $N$ | $P$ | $Q$ | Area (μm$^2$) | LLR Mem. (%) | Part. Sums (%) | PEs (%) | Control (%) | T/P (Mbps) |
|---|---|---|---|---|---|---|---|---|
| $2^{10}$ | 64 | 5 | 308,693 | 76.10 | 17.52 | 4.77 | 0.48 | 246.1$R$ |
| $2^{11}$ | 64 | 5 | 527,103 | 77.15 | 18.85 | 2.99 | 0.33 | 244.2$R$ |
| $2^{12}$ | 64 | 5 | 940,420 | 76.60 | 20.91 | 2.00 | 0.23 | 242.4$R$ |
| $2^{13}$ | 64 | 5 | $1,893$,835 | 79.92 | 18.78 | 0.97 | 0.15 | 240.6$R$ |

[11] S. B. Korada, E. Sasoglu, and R. Urbanke, "Polar Codes: Characterization of Exponent, Bounds, and Constructions," *IEEE Trans. on Information Theory*, vol. 56, no. 12, pp. 6253–6264, 2010.

[12] C. Leroux, I. Tal, A. Vardy, and W. J. Gross, "Hardware architectures for successive cancellation decoding of polar codes," in *Proc. IEEE Int Acoustics, Speech and Signal Processing (ICASSP) Conf*, 2011, pp. 1665–1668.

[13] Tong Zhang and K.K. Parhi, "A 54 Mbps (3,6)-regular FPGA LDPC decoder," in *IEEE Workshop on Signal Processing Systems, (SIPS '02)*, Oct. 2002, pp. 127–132.

[14] M.P.C. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Trans. on Comm.*, vol. 47, no. 5, May 1999.

[15] A. Pamuk, "An FPGA implementation architecture for decoding of polar codes," in *8th International Symposium on Wireless Communication Systems (ISWCS)*, Nov. 2011, pp. 437–441.