Hardware Implementation of Successive Cancellation Decoders for Polar Codes

Camille Leroux*, Alexandre J. Raymond†, Gabi Sarkis†, Ido Tal‡, Alexander Vardy‡ and Warren J. Gross†
*Institut Polytechnique de Bordeaux, CNRS IMS, UMR 5218, Bordeaux, France.
†McGill University, Montreal, QC, Canada.
‡University of California San Diego, La Jolla, CA, USA.

***Abstract*—The recently-discovered polar codes are seen as a major breakthrough in coding theory; they provably achieve the theoretical capacity of discrete memoryless channels using the low complexity successive cancellation (SC) decoding algorithm. Motivated by recent developments in polar coding theory, we propose a family of efficient hardware implementations for SC polar decoders. We show that such decoders can be implemented with $O(n)$ processing elements, $O(n)$ memory elements, and can provide a constant throughput for a given target clock frequency. Furthermore, we show that SC decoding can be implemented in the logarithm domain, thereby eliminating costly multiplication and division operations and reducing the complexity of each processing element greatly. We also present a detailed architecture for an SC decoder and provide logic synthesis results confirming the linear growth in complexity of the decoder as the code length increases. *Index Terms*—olar codes successive cancellation decoding hardware implementation VLSI.olar codes successive cancellation decoding hardware implementation VLSI.P**

I. Introduction

Polar codes [1] form a family of error correcting codes with an explicit and efficient construction [2] encoding and decoding algorithms. They achieve channel capacity—asymptotically in the code length n—when the underlying channel is memoryless and has a discrete input alphabet [3]. To date, they are the first codes to provably achieve channel capacity with tractable decoding complexity. Moreover, in some information theoretic applications, such as achieving the secrecy capacity of the wiretap channel in the general case, polar codes are the only known solution which is both explicit and efficient [4]. They are therefore seen as a major breakthrough in coding and information theory.

From a practical point of view, however, polar codes come close to achieving the channel capacity only for very large code lengths, e.g. $n \geq 2^{20}$. Recent works have therefore started to address the issue of performance at shorter code lengths. For example, it was shown in [5] that the belief propagation (BP) decoding of polar codes improved their performance compared to successive cancellation (SC) decoding without an increase in block length n. This performance gain is however obtained at the expense of an increase in decoding complexity. List decoding [6] also improves performance without an increase in code length; however, decoding complexity grows linearly in list size.

Driven by recent theoretical advances related to polar codes and the extra complexity incurred by the use of BP or list decoding, we aim to find efficient hardware architectures for SC decoding, allowing both high throughput and low area implementations of moderate length polar decoders. Starting from the general framework proposed by Arıkan [1] and described in Section II, we develop multiple decoder architectures in order of decreasing hardware complexity and show that SC decoding can actually be implemented with hardware complexity $O(n)$ using the line decoder in Section III. Finally, We address the implementation of the decoder and its computational nodes and present logic synthesis results confirming our complexity analysis in Section IV.

II. Polar Codes

A polar code is a linear block error-correcting code designed for a specific discrete input, memoryless channel. From here on, we will assume that the channel has a binary input alphabet and is symmetric as well [7]. Let $n = 2^m$ be the code length and let $\mathbf{u} = (u_0, u_1, \ldots, u_{n-1})$ and $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ denote the input bits and the corresponding codeword[1], respectively. The encoding operation has a Fast-Fourier-Transform-like butterfly structure depicted in Figure 1 for $n = 8$. Note that the ordering of the u_i bits in Figure 1 is according to the bit-reversed order: if we reverse the order of the bits in the binary representation of i, we then get the natural ordering.

After $\mathbf{u}$ is encoded into $\mathbf{c}$, the codeword $\mathbf{c}$ is sent over the underlying channel (the channel is used n times). Denote by $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})$ the corresponding channel output. We now wish to decode $\mathbf{y}$. This is done in terms of a *successive cancellation* decoder. That is, given $\mathbf{y}$, we first try to deduce the value of u_0, then that of u_1, and so forth up until u_{n-1}. We do this as follows. Assume that we are currently at bit i and have already estimated the values of $u_0, u_1, \ldots, u_{i-1}$ to be $\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{i-1}$. Next, for $b \in \{0, 1\}$, denote by $\Pr(\mathbf{y}|\hat{u}_0^{i-1}, u_i = b)$ the probability that $\mathbf{y}$ was received, given that $u_0^{i-1} = \hat{u}_0^{i-1}$, $u_i = b$, and $u_{i+1}, u_{i+2}, \ldots, u_{n-1}$ are independent random variables with Bernoulli distribution of parameter 0.5. The estimated value $\hat{u}_i$ is chosen according to:

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

As the code length, n, increases, the probability that a bit u_i is correctly decoded, given that all previous bits were

[1]Note that n input bits are encoded to a length n codeword. However, as we will see later on, not all of the n input bits carry information.

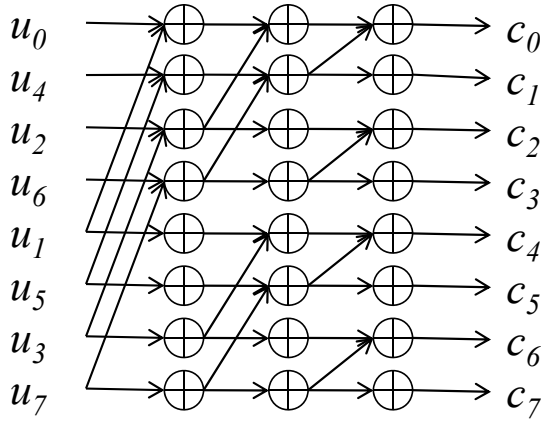

Fig. 1. Encoder architecture for $n = 8$.

correctly decoded, approaches[2] either 1 or 0.5 as proven in [1]. The fraction of bits whose probability of successful decoding approaches 1 tends towards the capacity of the underlying channel as n increases. This information regarding bit reliabilities is used to select a high reliability subset of $\mathbf{u}$ to store information bits; while the rest of $\mathbf{u}$, called the frozen bit set, is set to a fixed value, assumed to be 0 in this work. The frozen set is known at the decoder, which sets $\hat{u}_i$ to 0 if it is in the frozen set, and uses Equation (1) otherwise.

III. Successive Cancellation Decoder Architectures

A. Butterfly-based architecture

Arıkan showed that SC decoding can be efficiently implemented by the factor graph of the code, which has a structure resembling that of the Fast Fourier Transform. In the remainder of this paper, we will refer to this decoder architecture as the "butterfly-based SC decoder." Figure 2 illustrates the graph of this SC decoder for $n = 8$. Channel likelihood ratios (LRs) λ_i are assumed to be presented to the right hand side of the graph whereas the estimated bits $\hat{u}_i$ appear on the opposite end.

The SC decoder is composed of $m = \log_2 n$ stages, each containing n nodes. We refer to a specific node as $\mathcal{N}_{l,j}$ where l designates the stage index $(0 \leq l \leq m-1)$, and j, the node index within stage l $(0 \leq j \leq n-1)$. Each node updates its output according to one of the two following update rules:

$$\begin{aligned} f(a,b) &= \frac{1+ab}{a+b} \text{ or} \\ g_{\hat{u}_s}(a,b) &= a^{1-2\hat{u}_s} b. \end{aligned} \tag{2}$$

The values a and b are likelihood ratios while $\hat{u}_s$ is a bit that represents the partial modulo-2 sum of previously estimated bits. For example, in node $\mathcal{N}_{1,3}$, the partial sum is $\hat{u}_s = \hat{u}_4 \oplus \hat{u}_5$. The value of $\hat{u}_s$ determines if function g should be a multiplication or a division. These update rules are complex to implement in hardware since they involve multiplications and divisions. In Section III-D, we propose

[2]This is true for almost all i

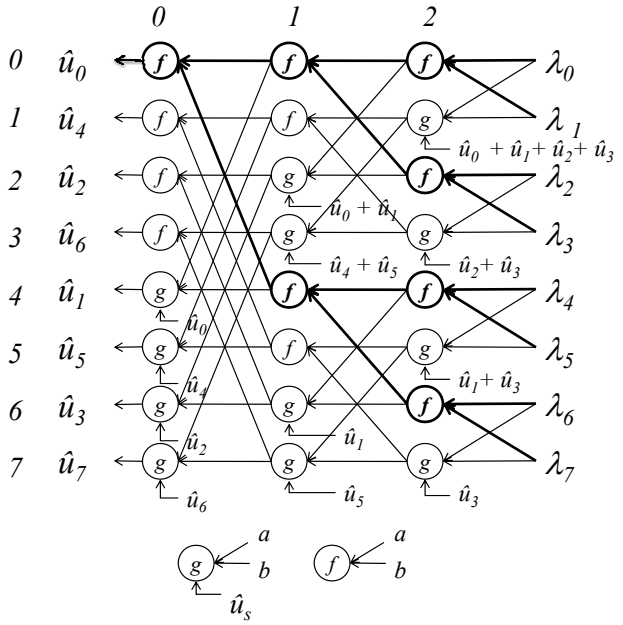

Fig. 2. The butterfly-based SC decoder architecture for $n = 8$.

to perform these operations in the logarithm domain and to apply an approximation to the function f.

The sequential nature of the algorithm introduces data dependencies to the decoding process. We notice that $\mathcal{N}_{1,2}$ cannot be updated before bit $\hat{u}_1$ is computed and a fortiori neither before $\hat{u}_0$ is known. In order to respect the data dependencies, a scheduling has to be defined. Arıkan proposed two schedulings for this decoding framework [1]. In the left-to-right scheduling, nodes recursively call their predecessors until an updated node is reached. The recursive nature of this scheduling is especially suitable for software implementation. In the alternative right-to-left scheduling, any node updates its value whenever its inputs are available which enables some nodes to update their values in parallel. Each bit $\hat{u}_i$ is successively estimated by activating the spanning tree rooted at $\mathcal{N}_{0,\pi(i)}$, where $\pi(.)$ denotes the bit-reverse mapping function. As an example, in Figure 2, the tree associated with $\hat{u}_0$ is highlighted. If we assume that memory elements are inserted between each stage or equivalently that each node processor can store its updated value, then some results can be reused. For example, in Figure 2, bit $\hat{u}_1$ can be decoded by only activating $\mathcal{N}_{0,4}$ since $\mathcal{N}_{1,0}$ and $\mathcal{N}_{1,4}$ have already been updated during the decoding of $\hat{u}_0$.

Despite this well-defined structure and scheduling of the butterfly-based decoder, in [1], Arıkan does not address the problem of resource sharing, memory management or control generation that would be required for hardware implementation. This framework however suggests that it could be implemented with $n \log_2 n$ combinational node processors together with n registers between each stage to store intermediate results. In order to store the channel information, n extra registers are included as well. The total complexity of such a decoder is

$$C_{butterfly} = (C_{\mathrm{np}} + C_{\mathrm{r}}) n \log_2 n + n C_{\mathrm{r}}, \tag{3}$$

where C_{np} and C_{r} are the hardware complexity of a node processor and a memory register, respectively. In order to decode one vector, each stage l has to be activated 2^{m-l} times. If we assume that one stage is activated at each clock cycle, then the number of clock cycles required to decode one vector is

$$NCC = \sum_{l=0}^{m-1} 2^{m-l} = 2n - 2. \tag{4}$$

The throughput in bits per second would then be

$$T = \frac{n}{NCC \times t_{\text{np}}} \approx \frac{1}{2t_{\text{np}}}, \tag{5}$$

where t_{np} is the propagation time in seconds through a node processor which also represents the clock period. It follows that every node processor is actually used once every $2n-2$ clock cycles. This motivates us to find a schedule to merge some of the nodes into a single processing element.

B. Pipelined tree architecture

Further studying of the scheduling reveals that whenever stage l is activated, only 2^l nodes are actually updated. For example, in Figure 2, when stage 0 is enabled, only one node is updated. Then the n nodes of stage 0 can be implemented using a single processing element (PE). As such, for stage l, 2^l processing elements are sufficient to update all the nodes. However, this resource sharing does not necessarily guarantee that the memories assigned to the merged nodes can also be merged. Table I shows the stage activation during the decoding of one vector $\mathbf{y}$. When stage S_l is enabled, we indicate which function (f or g) is applied to the 2^l activated nodes at stage S_l during each clock cycle (CC). Every generated variable is used twice during the decoding. For example, the four variables generated in stage 2 at CC #1 are consumed on CC #2 and CC #5 in stage 1. This means that, in stage 2, the four registers associated with the f function can be reused at CC #8 to store the four data values generated by the g function. This observation is applicable to any stage in the decoder. The resulting proposed architecture is shown in Figure 3 for $n = 8$. The channel LRs, λ_i, are stored in n registers. The rest of the decoder is composed of a pipelined tree structure that includes $n-1$ PEs, $\text{P}_{l,j}$, and $n-1$ registers, $\text{R}_{l,j}$ with $0 \le l \le m-1$ and $0 \le j \le 2^l - 1$. A decision unit generates the estimated bit $\hat{u}_i$ which is then broadcast back to every PE. A PE is a configurable element that can perform either the f or the g function. It also includes the $\hat{u}_s$ computation block that updates the $\hat{u}_s$ value with the last decoded bit $\hat{u}_i$ only if the control bit $b_{l,j} = 1$. Another control bit is used to select the f or g function. Compared to the butterfly-based structure, the pipelined tree architecture performs the same amount of computation with the same scheduling (see Table I) but with a smaller number of PEs and registers. The throughput is then the same as in (5) and the decoder has lower hardware complexity

$$C_{tree} = (n-1)(C_{PE} + C_{\text{r}}) + nC_{\text{r}}, \tag{6}$$

where C_{PE} represents the complexity of a single PE. In addition to the lower complexity, one can notice that the routing network in the decoder is much simpler in the tree architecture than in the butterfly-based structure. Connections between PEs are also local. This lowers the risk of congestion during the wire routing phase of an integrated circuit design and potentially increases the clock frequency and the throughput.

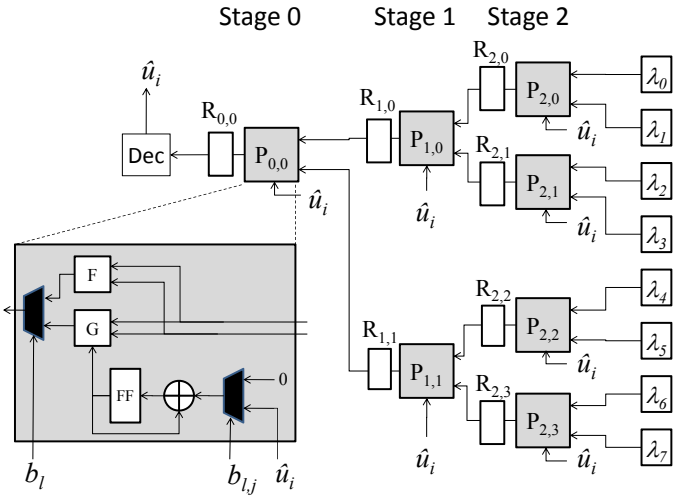

Fig. 3. Pipelined tree SC architecture for $n = 8$.

CC	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S_2	f							g						
S_1		f			g				f			g		
S_0			f	g		f	g			f	g		f	g
$\hat{u}_i$			$\hat{u}_0$	$\hat{u}_1$		$\hat{u}_2$	$\hat{u}_3$			$\hat{u}_4$	$\hat{u}_5$		$\hat{u}_6$	$\hat{u}_7$

TABLE I
SCHEDULE FOR THE BUTTERFLY-BASED AND PIPELINE TREE SC ARCHITECTURES ($n = 8$).

C. Line architecture

Despite the low complexity of the pipelined tree architecture, it is possible to further reduce the number of PEs. Looking at Table I, it appears that only one stage is activated at a time. In the worst case—stage $m-1$ is activated—$\frac{n}{2}$ PEs have to be used simultaneously. This means that the same throughput can be achieved with only $\frac{n}{2}$ PEs. The resulting architecture is shown in Figure 4 for $n = 8$. The processing elements P_j are arranged in a line; while the registers retain a tree structure emulated by a multiplexing resources connecting the two.

For example, since $\text{P}_{2,0}$ and $\text{P}_{1,0}$ (in Figure 3) are merged into P_2 (in Figure 4), P_2 should write either to $\text{R}_{2,0}$ or $\text{R}_{1,0}$; and it should also read from the channel registers or from $\text{R}_{2,0}$ and $\text{R}_{2,1}$. The $\hat{u}_s$ computation block is moved out of P_j and kept close to the associated register because $\hat{u}_s$ should also be forwarded to the PE. The overall complexity of the line SC architecture is

$$C_{line} = (n-1)(C_{\text{r}} + C_{\hat{u}_s}) + \frac{n}{2}C_{PE} + \left(\frac{n}{2} - 1\right) 3C_{\text{mux}} + nC_{\text{r}}, \tag{7}$$

where C_{mux} represents the complexity of a 2-input multiplexer and $C_{\hat{u}_s}$ is the complexity of the $\hat{u}_s$ computation block.

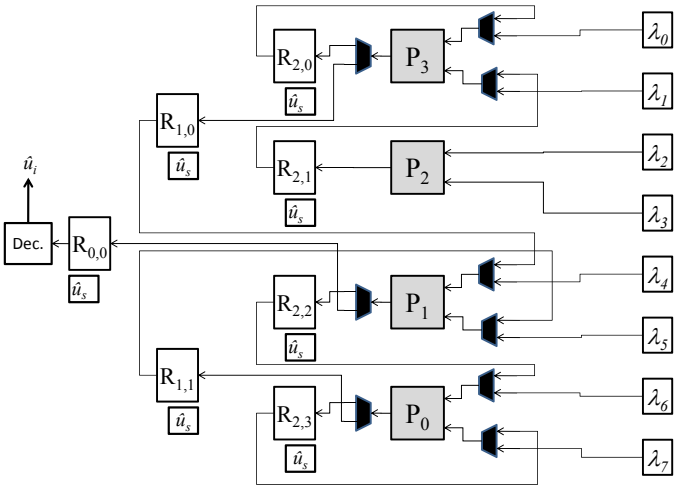

Fig. 4. Line SC architecture for $n = 8$.

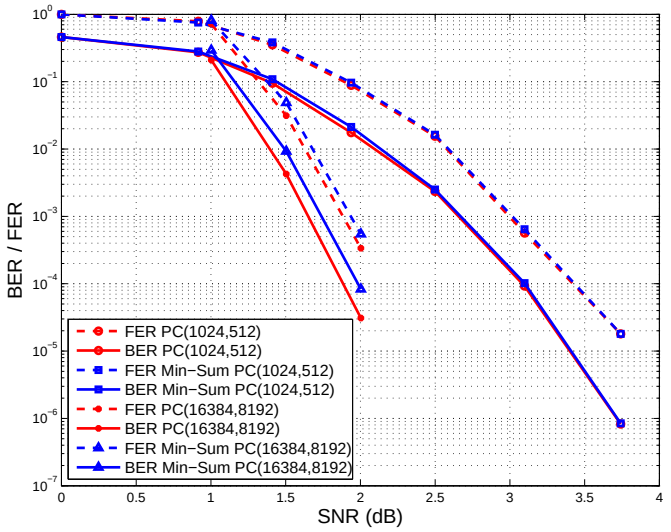

Fig. 5. The min-sum approximation error-correction performance change for PC(1024,512) and PC(16384,8192).

Despite the extra multiplexing logic required to route the data through the PE line, the savings in number of PEs makes this SC decoder less complex than the pipelined tree architecture while achieving the same throughput computed in (5). The control logic is not included in the complexity estimation since it is negligible compared to processing and memory. This will be confirmed by logic synthesis results in section IV-C.

The Line SC architecture can be seen as a tree architecture in which complexity is reduced by merging some of the PEs without affecting throughput.

D. The min-sum approximation

SC decoding was originally proposed in the likelihood ratio domain, in which the update rules f and g require multiplications and divisions. Since the cost of implementing these operations in hardware is very high, they are usually avoided in practice. We thus propose to perform SC decoding in the logarithm domain in order to reduce the complexity of the f and g computation blocks. We assume that the channel information is available as log-likelihood ratios (LLRs) L_i, which leads to the following alternative representation for equations f and g:

$$f(L_a, L_b) = 2\tanh^{-1}\left(\tanh\left(\frac{L_a}{2}\right)\tanh\left(\frac{L_b}{2}\right)\right) \text{ and}$$
$$g_{\hat{u}_s}(L_a, L_b) = L_a(-1)^{\hat{u}_s} + L_b. \tag{8}$$

In terms of hardware implementation, g can easily be mapped to an adder-subtractor controlled by the bit $\hat{u}_s$. However, f involves some transcendental functions that are complex to implement in hardware. One can notice that the f and g functions are identical to the update rules used in BP decoding of low-density parity-check (LDPC) codes. Consequently, an approximation used in LDPC decoder implementations [8] can be used to approximate f using the *minimum* function, such that

$$f(L_a, L_b) \approx \text{ sign}(L_a)\ \text{sign}(L_b)\min(|L_a|, |L_b|) \text{ and}$$
$$g_{\hat{u}_s}(L_a, L_b) = L_a(-1)^{\hat{u}_s} + L_b. \tag{9}$$

In order to estimate the performance degradation incurred by this approximation, we simulated the performance of different polar codes on an additive white-Gaussian (AWGN) channel with binary phase-shift keying (BPSK). As it can be seen in Figure 5, the performance degradation is minor for moderate code lengths and is very small (0.1dB) for longer codes.

IV. Line Decoder Hardware Implementations

Section III showed that the line architecture has a lower hardware complexity—and is thus more efficient—than its tree-based counterpart. This section presents details and synthesis results of an implementation of the line architecture.

A. Fixed-point simulations

The number of quantization bits impacts both the decoding performance of the algorithm and the hardware complexity of the decoder. Consequently, prior to implementing the line decoder, a detailed analysis was carried out on a software-based SC decoder in order to find the best tradeoff between performance and complexity. The resulting simulations revealed that fixed-point operations on a limited number of quantization bits attained a decoding performance very similar to that of a floating point algorithm. Figure 6 illustrates this phenomenon for a $\text{PC}(1024, 512)$ decoder. It shows that 5 or 6 quantization bits are sufficient to reach near-floating point performance at a saturation level of $\pm 3\sigma$, which exhibits good performance over all quantization levels. It should be noted that the channel saturation level has a high impact on the performance of low quantization ($q = 3, 4$) decoders. The selected saturation value ($\pm 3\sigma$) was chosen from further software simulations not shown here.

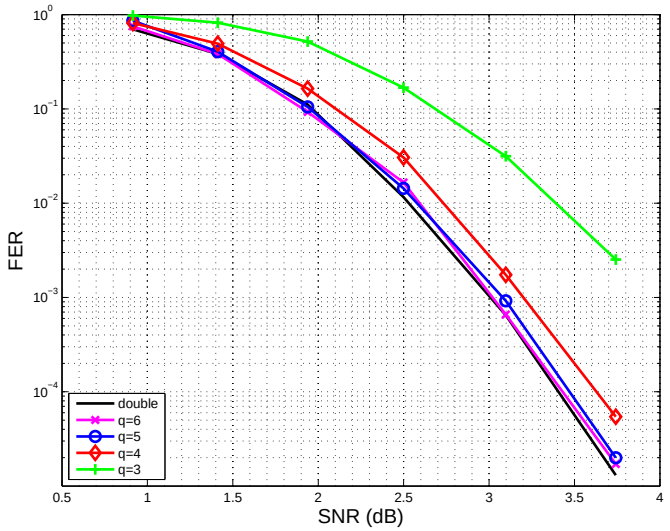

Fig. 6. Fixed-point FER simulation for PC(1024,512) on AWGN channel. Input saturation $= 3\sigma$.

B. Line decoder detailed architecture

1) Processing elements: The processing element is the main arithmetic component of the line decoder. It embodies the arithmetic logic needed to perform both f and g functions within a single logic component. This grouping, motivated by the fact that all stages of the decoding graph either perform function f or g at any given time, allowed a greater level of resource sharing. PEs also implement the min-sum approximation described in Section III-D, which allows for much simpler decoding logic, as it replaces three transcendental functions with a single comparator. Since processing elements are replicated $n/2$ times, Equation (7), this approximation has a significant impact on the overall size of the decoder.

In this work, processing elements are fully combinational and operate on quantized sign-and-magnitude (SM) coded LLRs. We initially implemented our PEs in two's complement format (TC), for its wide support in HDL languages, but logic synthesis showed a 20% area reduction when using SM instead. Indeed, Equation (9) shows that the main operations performed on LLRs are addition, subtraction, absolute value, sign retrieval, and minimum value, all of which are very low complexity operations when using SM format.

Figure 7 illustrates the overall architecture of our SM-based PE. In this figure, L_a and L_b are the two q-bit input LLRs of functions f and g; a partial sum signal $\hat{u}_s$ controls the behavior of g; the sign $s(.)$ and the magnitude $|.|$ of input LLRs are directly extracted; and the comparator is shared for the computation of $|L_f|$, $|L_g|$ and $s(L_g)$. Thick lines and thin lines represent magnitude and sign data paths, respectively.

2) Register banks: As seen in Section III-C, memory resources are needed to store partial results during the decoding process. The decoder is implemented using two separate memories: one for partial LLR calculations, and another for the partial sums $\hat{u}_s$. The line decoder memory has a tree structure and uses $(2n-1)$ q-bit memory cells to store LLRs, in addition to $(n-1)$ 1-bit cells to store the partial sums $\hat{u}_s$ used to carry out function g.

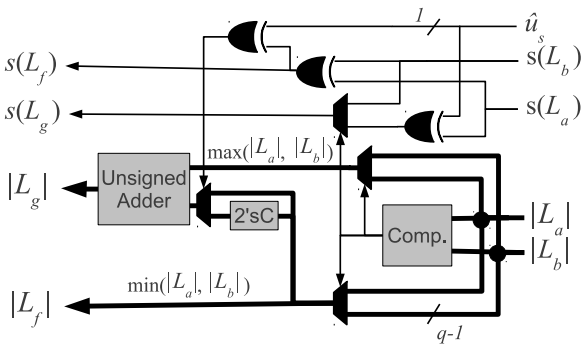

Fig. 7. Processing element architecture.

The LLRs memory can be seen as $(\log_2 n + 1)$ separate memories—one for each stage—with each stage l requiring 2^l q-bit memory cells. Stage $(\log_2 n + 1)$ is special in that it contains the received channel LLRs, requiring shift-register capabilities. Each stage produces half as much data as it consumes. This data is written into memory locations read by the subsequent stage, $l-1$.

The partial sum memory combines the $n \log_2 n$ partial sums of the decoding graph into $n-1$ memory cells by time-multiplexing each memory cell for use by multiple nodes of the graph.

3) Multiplexing: The shared nature of the processing elements used in the line architecture requires the multiplexing of their inputs and outputs. As shown in Figure 4, memory is implemented using registers, and separate networks of multiplexers and demultiplexers are used to provide them with appropriate inputs from memory and store their outputs to memory, respectively.

An alternate design for the line architecture could make use of SRAM blocks, in which case the multiplexing networks could be avoided completely, as equivalent logic would be directly embodied in the memory decoder of the SRAM modules. This would allow for a more compact memory block, although potentially increasing access time. An even more optimized design could mix both SRAM and registers: looking at table I, it appears that some of the memory elements are accessed more often than others. It would be more efficient to implement these frequently accessed memory elements into registers while keeping the SRAM blocks for less frequently accessed data. In this work, since we target moderate length codes, we choose to use registers only.

4) General control: The line decoder is a multi-stage design which sequentially decodes each codeword. It uses specific control signals [3] to orchestrate the decoding.

Those control signals are combinational functions of i, the current decoded bit number, and l, the current stage. These two signals are in turn generated using counters, and some extra logic. The underlying understanding is that up to $\log_2(n)$ stages must be activated in sequence to decode each bit $\hat{u}_i$. Once it has been decoded, this bit is stored in specific partial

[3]Binary representations of integers are assumed to be stored in little-endian format

sums $\hat{u}_s$ for the decoding of subsequent bits, according to the data dependencies highlighted previously.

Both i and l can be viewed as counters, where i counts up from 0 to $n-1$, for each decoded bit; while l counts down to 0, from a value between 1 and $(\log_2(n)-1)$, for each stage. The decoding of a codeword takes $2n-2$ clock cycles overall, as demonstrated in Equation (4).

Counter l, unlike i, is not reset to a fixed value. By making use of the partial computations stored in the LLR memory, it can be reset to the result of a find-first-bit-set (`ffs`) operation on i, corresponding to a modified priority encoder. Specifically, it is reset to $\texttt{ffs}^\star(i+1)$ upon reaching 0, according to Equation (10).

$$\texttt{ffs}^\star(x_{m-1}\dots x_1x_0) = \begin{cases} \min(i) : x_i = 1 & \text{if } x > 0 \\ m-1 & \text{if } x = 0 \end{cases} \tag{10}$$

Another control signal deals with the function that the processing elements must perform on behalf of a specific stage. Since the nodes of a given stage all perform the same function at any given time, this signal can be used to control all the PEs of the line. The function selection is performed using Equation (11).

$$\texttt{selector}_{f,g}(i_m\dots i_1i_0,\ l) = \begin{cases} \texttt{f} & \text{if } i_l = 0 \\ \texttt{g} & \text{if } i_l = 1 \end{cases} \tag{11}$$

5) Memory control: Both the LLR and the partial sum memory require significant multiplexing in order to route the proper values from both memories to the PEs, and vice versa.

The multiplexer network mapping the inputs of the processing elements to the LLR memory use the mapping shown in Equation (12),

$$\texttt{MAP}_{\texttt{LLR}}^{\texttt{MEM}\to\texttt{PE}}(l,p) = \begin{cases} \texttt{MEM}_{\texttt{LLR}}(2n-2^{l+2}+2p) & \text{for } L_a \\ \texttt{MEM}_{\texttt{LLR}}(2n-2^{l+2}+2p+1) & \text{for } L_b \end{cases} \tag{12}$$

where $0 \le p \le (\frac{n}{2}-1)$ is the index of the PE in the line. This mapping assumes that the original codeword is stored in memory $\texttt{MEM}_{\texttt{LLR}}(0:n-1)$.

The resulting computation is then stored according to the mapping shown in Equation (13), noting that only the first 2^l PEs of the line are active in stage l.

$$\texttt{MAP}_{\texttt{LLR}}^{\texttt{PE}\to\texttt{MEM}}(l,p) = \texttt{MEM}_{\texttt{LLR}}(2n-2^{l+1}+p) \tag{13}$$

Once stage 0 has been activated, the output of $\texttt{PE}_0$ contains the LLR of the decoded bit i, and a hard decision $\hat{u}_i$ can be obtained from this soft output using Equation (1); in other words, if $\texttt{sign}(\texttt{LLR}) = 0$. At this point, if bit i is known to be a frozen bit, the output of the decoder is forced to $\hat{u}_i = 0$.

Once bit $\hat{u}_i$ has been decoded, this value must be reflected in the partial (modulo-2) sums $\hat{u}_s$ of the decoding graph. Algorithm 1 determines, for each g node with index z in stage l, whether it must be updated.

One can note that the original decoding graph contains $\frac{n}{2}\log_2(n)$ such partial sums, but that a maximum of $n-1$ of them are used for the decoding of any given bit. With some careful time-multiplexing, it is thus possible to reduce the number of memory cells used to hold the partial sums to $n-1$, a clear reduction in complexity. This is the approach taken in this paper.

Algorithm 1 Partial sums updating algorithm

```
z* ← bitreverse(z)
if l_i = 0 then
  if l = m − 1 or i_{(m−1):(l+1)} = z*_{(m−2):l} then
    if (l = 0) or ((not(i_{l:0}) and z*_{l:0}) = 0) then
      Node z updates its partial sum with û_i
    end if
  end if
end if
```

Finally, the mapping shown in Equation (14) connects the partial sum input of $\texttt{PE}_p$ to the partial sums memory.

$$\texttt{MAP}_{\hat{u}_s}^{\texttt{MEM}_{\hat{u}_s}\to\texttt{PE}_p}(l,p) = \texttt{MEM}_{\hat{u}_s}(n-2^{l+1}+p) \tag{14}$$

All of these mapping equations, together with Algorithm 1, are efficiently implemented with combinational logic.

C. ASIC synthesis results

In order to evaluate the silicon footprint of the line decoder, a generic RTL description of the architecture was designed, and synthesized using a standard cell library.

This generic description enabled us to generate specific line decoder instances for any code length n, code rate R, target signal-to-noise ratio SNR, and quantization level q. Syntheses were carried out to measure the impact of these parameters on area, using *Cadence RTL Compiler v9.1* and the TSMC 65nm worst case [4] CMOS standard cell library. Synthesis was driven by Physical Layout Estimators (PLE), which allow a more accurate estimation of interconnection delays and area, compared to the classical wire-load model. The target frequency was set to 500MHz.

A first set of decoders was generated for $8 \le n \le 1024$ and $4 \le q \le 6$. Figure IV-C shows the evolution of area as code size and quantization increase. As expected, area grows linearly with n and q. The linear hardware complexity of the line decoder validates Equation (7).

Then, a second set of decoders was generated and synthesized for $n = 1024$, for different codes rates. Synthesis results confirmed that the code rate does not impact hardware complexity. This was expected because the frozen bits are stored in a ROM, whose size is constant; only its contents changes, according to the code rate and target SNR.

Finally, a set of decoders was generated for $8 \le n \le 1024$ and $q = 5$. The area of each component block was extracted, in order to estimate their relative complexity share inside the decoder. Results are shown in Figure V. Memory resources (register banks), processing logic, and multiplexing, represent 38%, 36%, and 26% of the total area, respectively. The control logic is negligible ($< 1\%$), which is expected as it grows logarithmically in n.

[4] Nominal supply voltage and temperature are $V_{dd} = 0.9V$ and $T = 125^oC$, respectively

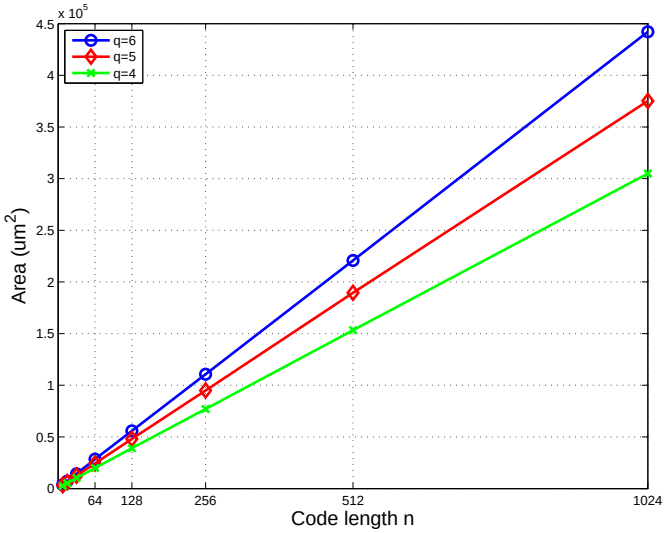

Fig. 8. Line decoder area for different quantization and code lengths, TSMC 65nm, f=500MHz

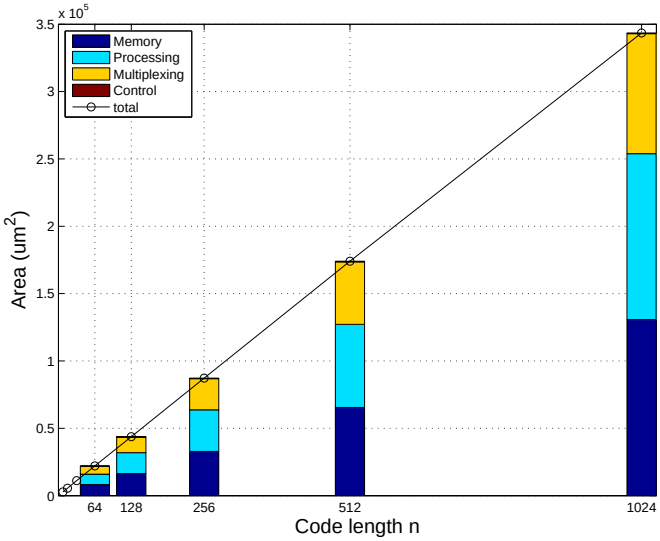

Fig. 9. Line decoder area repartition for different code lengths and $q = 6$, TSMC 65nm, f=500MHz

D. Verification

Verification of the hardware design was carried out by means of functional simulation. Specifically, a testbench was written to exercise the decoder using 10^3 to 10^6 randomly-generated noisy input vectors. The output of the simulated hardware decoder was then compared to its software counterpart, whose error-correction capabilities had previously been verified experimentally. This validation was repeated for various combinations of SNR and code lengths to ensure good test coverage.

V. Conclusion

Polar codes have recently generated great interest from a theoretical point of view. In this paper, we explore the hardware implementation of polar code decoders; we propose two SC decoders architectures with linear complexity. Software simulations allowed us to validate the proposed min-sum approximation, and to determine implementation parameters, such as the quantization level. For the most efficient decoder—the line-decoder—we provided a detailed description of each component block. Logic synthesis using a standard cell library confirmed the linear evolution of hardware complexity with respect to the code length.

References

[1] E. Arikan, "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.

[2] I. Tal and A. Vardy, "How to construct polar codes," *submitted to IEEE Trans. Inform. Theory, available online as* `arXiv:1105.6164v2`, 2011.

[3] E. Sasoglu, E. Telatar, and E. Arikan, "Polarization for arbitrary discrete memoryless channels," in *Proc. IEEE Information Theory Workshop ITW 2009*, 2009, pp. 144–148.

[4] H. Mahdavifar and A. Vardy, "Achieving the secrecy capacity of wiretap channels using polar codes," in *IEEE ISIT 2010*, Jun. 2010, pp. 913 –917.

[5] N. Hussami, R. Urbanke, and S.B. Korada, "Performance of polar codes for channel and source coding," in *IEEE ISIT 2009*, Jun. 2009, pp. 1488 –1492.

[6] Ido Tal and Alexander Vardy, "List decoding of polar codes," in *Proc. (ISIT) Symp. IEEE Int Information Theory*, 2011.

[7] R. Gallager, *Information Theory and Reliable Communications*, John Wiley, New York, 1968.

[8] 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, pp. 673 –680, May. 1999.