

# Pipelined Architecture for Soft-decision Iterative Projection Aggregation Decoding for RM Codes

Marzieh Hashemipour-Nazari\*, Yuqing Ren†, Kees Goossens\*, and Alexios Balatsoukas-Stimming\*

\*Electronic Systems, Eindhoven University of Technology, The Netherlands

†Telecommunications Circuits Laboratory, École Polytechnique Fédérale de Lausanne, Switzerland

**Abstract**—The recently proposed recursive projection-aggregation (RPA) decoding algorithm for Reed-Muller codes has received significant attention as it provides near-ML decoding performance at reasonable complexity for short codes. However, its complicated structure makes it unsuitable for hardware implementation. Iterative projection-aggregation (IPA) decoding is a modified version of RPA decoding that simplifies the hardware implementation. In this work, we present a flexible hardware architecture for the IPA decoder that can be configured from fully-sequential to fully-parallel, thus making it suitable for a wide range of applications with different constraints and resource budgets. Our simulation and implementation results show that the IPA decoder has 41% lower area consumption, 44% lower latency, four times higher throughput, but currently seven times higher power consumption for a code with block length of 128 and information length of 29 compared to a state-of-the-art polar successive cancellation list (SCL) decoder with comparable decoding performance.

**Index Terms**—RPA, IPA, Reed-Muller codes, pipelined architecture.

## I. INTRODUCTION

**F**UTURE communications systems will need to enable ultra-reliable low-latency communications (URLLC) and machine-type communications (MTC) [1]. Low latency generally implies the use of very short packets. Moreover, in some MTC systems, such as Internet of Things (IoT) applications, there is not enough data to create large packets because sensors typically only transmit a small amount of data infrequently [2], [3]. Low-density parity check (LDPC) [4] and turbo [5] codes are highly regarded due to their ability to achieve significant coding gains in the moderate blocklength regime, while maintaining linear decoding complexity. However, conventional asymptotic methods used to construct LDPC and turbo-like codes, such as extrinsic information transfer (EXIT) charts, often have difficulties to generate short codes with good performance. As a result, achieving high reliability with short packets becomes challenging, as conventional error-correcting schemes typically require moderate to large blocklengths to be effective.

An alternative approach for short packets is to utilize polar codes, which are capacity-achieving codes with low encoding and decoding complexity for binary-input memoryless symmetric (BMS) channels under successive cancellation (SC) decoding [6]. However, to make polar codes effective for short blocklengths, certain modifications are necessary. Typically, this means employing the SC list (SCL) decoder with a very large list size, combined with a CRC code. Consequently, this results in a reduced effective information rate for the code and

increased decoding complexity and latency. These challenges have increased attention towards codes and decoding algorithms specifically designed for short-length packets [7], [8], aiming to enhance communication performance and achieve low latency.

Reed-Muller (RM) codes are a class of linear block error-correcting codes that are closely related to polar codes. They were first discovered and introduced by Reed [9] and Muller [10]. Reed's decoder is based on majority voting and can correct a number of errors up to half of the code's minimum distance. Several additional decoding methods were developed to improve the decoding performance [11]–[15]. More recently, there has been a renewed interest in RM codes as they can achieve the Shannon capacity on any BMS channel [16]–[20] and they were shown to outperform polar codes under maximum likelihood (ML) decoding for short codes [21], [22].

As ML decoding is generally intractable, the authors of [23] introduced a more practical near-ML decoding method for RM codes called recursive projection-aggregation (RPA) decoding. The RPA algorithm exploits the recursive structure of RM codes by projecting a received codeword with a length of  $n$  into  $n - 1$  shorter codewords and decoding the projected codewords recursively until codewords belonging to a first-order RM code are reached, which can be decoded efficiently using the fast Hadamard transform. The number of recursive calls depends on the order of the employed RM code. The complexity of RPA decoding scales as  $n^r$ , where  $r$  is the order and  $n$  is the length of the RM code. However, decoding low-order RM codes (i.e., second and third-order RM codes) with a short length is still practical with RPA decoding, making it particularly interesting for URLLC and MTC applications. Nevertheless, the complexity and recursive structure of RPA decoding are still major challenges for its efficient hardware implementation.

Some modified versions of the RPA algorithm have been proposed to reduce its algorithmic complexity. Simplified RPA [23] is a variant of RPA deploying two-dimensional subspaces for the projection step, which reduces the total number of projections. The authors of [24] proposed a collapsed projection-aggregation (CPA) decoding algorithm, which merges multiple recursion levels into a single step and has lower complexity. The results in [24] show that the CPA algorithm achieves a similar error-correcting performance to the RPA algorithm for RM codes with  $r = 3$  and  $n = 128$ . To further reduce complexity, [25] and [26] proposed different ways to exploit correlations between projection to prune CPA. Although both simplified RPA and CPA reduce the overall algorithmic complexity, they make the projection and aggregation steps

more involved as they employ more complex operations. Sparse RPA (SRPA) [27] is another modification of the RPA decoder that consists of multiple sparse RPA decoders. Each sparse RPA decoder uses only a random subset of projections. The work of [28] has lowered the average computational complexity of RPA by taking advantage of syndrome-based early stopping techniques along with a scheduling scheme.

Even though all aforementioned algorithmic complexity-reduction methods for RPA are promising, there are still challenges in their hardware implementation due to their recursive and/or complex structure. The iterative projection-aggregation (IPA) [29] algorithm transforms the recursive structure of the RPA decoder into an iterative structure, making it more straightforward for hardware implementation. The work of [29] includes a preliminary fully-parallel hardware implementation of the IPA algorithm, but only for the special case of hard-decision decoding.

*Contributions:* The main contributions of this paper are:

- We first design a flexible processing unit for soft-decision IPA decoding that can perform one level of projection, first-order decoding, and a part of the aggregation step. The proposed processing unit is configurable at runtime for performing different projections and their corresponding aggregations. Moreover, we propose hardware-friendly architectures for the projection and aggregation steps that reduce the required hardware resources and simplify the data flow.
- We design a flexible pipelined architecture based on the proposed processing units for the IPA algorithm that can be configured to be from fully-sequential to fully-parallel. As a result, the proposed architecture is very flexible in trading latency and throughput for area, such that it is applicable to a wide range of URLLC and MTC systems with different requirements and constraints. Additionally, to achieve high throughput, we design the controlling path of the architecture to support pipelining. This enables efficient data processing and optimal utilization of available resources.
- We compare the error-correcting performance of short RM codes under IPA decoding with similar 5G polar codes under SCL decoding [30], [31]. Moreover, we compare our proposed architecture for the IPA decoder and a state-of-the-art implementation of the SCL decoding [32] for the same blocklength, rate, and error-correcting performance, and we show that our IPA decoder is superior with respect to the area and latency in the short blocklength and low-rate regime. However, it currently has a higher power consumption compared to a state-of-the-art SCL decoder.

*Outline:* The remainder of this paper is organized as follows. In Section II, we review the background of RM codes as well as the RPA and IPA decoding algorithms. In Section III, we provide a detailed description of our proposed architecture for the second-order IPA decoder by explaining the structure of the processing unit, voting circuit, register array, and control unit. In Section IV, we explain how our basic second-order decoder architecture can be generalized to decode RM codes of any order. In Section V, we discuss the simulation and implementation results. We first compare the error-correcting performance of the IPA decoder to the baseline RPA decoder and then present multiple simulations to justify certain param-

eter choices for the hardware implementation in Section V-A. In addition, we compare the hardware implementation results of the IPA decoder for different RM codes with hard-decision (HD) IPA [29], and a state-of-the-art SCL decoder [32] for polar codes in Section V-B. Finally, Section VI concludes this paper.

## II. BACKGROUND

*Notation:* In this manuscript, lowercase and uppercase boldface letters denote vectors and matrices, respectively. In addition, vectors of log-likelihood ratios (LLR) are denoted by the boldface uppercase and non-italic letter  $\mathbf{L}$ . The symbols  $\mathbf{y}^i$  and  $y(j)$  represent the  $i$ -th projected vector and the  $j$ -th coordinate of the vector  $\mathbf{y}$ , respectively.

### A. Reed-Muller codes

Reed-Muller (RM) codes are linear block codes denoted by  $\text{RM}(m, r)$  with rate  $R = \frac{k}{n}$ , where  $n = 2^m$  indicates the code length,  $r$  is the order, and  $k = \sum_{i=0}^r \binom{m}{i}$  is the dimension of the code. There are several ways to define RM codes [33], [34] including a recursive approach, which is called Plotkin construction. In the Plotkin  $(\mathbf{u}, \mathbf{u} + \mathbf{v})$  construction of RM codes,  $\mathbf{u}$  and  $\mathbf{u} + \mathbf{v}$  are two subvectors of length  $2^{m-1}$  of a codeword  $\mathbf{c} = (\mathbf{u}, \mathbf{u} + \mathbf{v}) \in \text{RM}(m, r)$ , where  $\mathbf{u} \in \text{RM}(m-1, r)$  and  $\mathbf{v} \in \text{RM}(m-1, r-1)$ . Therefore, the generator matrix  $\mathbf{G}_{(m,r)} \in \mathbb{F}^{k \times n}$  for an  $\text{RM}(m, r)$  code is recursively defined based on the Plotkin construction as:

$$\mathbf{G}_{(m,r)} = \begin{bmatrix} \mathbf{G}_{(m-1,r)} & \mathbf{G}_{(m-1,r)} \\ \mathbf{0} & \mathbf{G}_{(m-1,r-1)} \end{bmatrix}, \quad \mathbf{G}_{(1,1)} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}. \quad (1)$$

In addition, the minimum Hamming distance of the  $\text{RM}(m, r)$  code is  $d = 2^{m-r}$ .

### B. First-order RM codes and decoder

$\text{RM}(m, 1)$  represents the first-order RM code with length  $n = 2^m$ , dimension  $k = m + 1$ , and minimum Hamming distance  $d = 2^{m-1}$ . The most popular optimal decoding method for first-order RM codes is the decoding algorithm based on the fast Hadamard transform (FHT) [35], [36]. The FHT-based decoding operates in the following four steps:

- 1) Calculate the FHT of the received vector  $\mathbf{y}$ , represented by its LLR values  $\mathbf{L}$  as:

$$\omega = \mathbf{H}_{2^m} \mathbf{L}, \quad (2)$$

where the Hadamard matrix  $\mathbf{H}_{2^m}$  is defined as:

$$\mathbf{H}_{2^m} = \begin{bmatrix} \mathbf{H}_{2^{m-1}} & \mathbf{H}_{2^{m-1}} \\ \mathbf{H}_{2^{m-1}} & -\mathbf{H}_{2^{m-1}} \end{bmatrix}, \quad \mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \quad (3)$$

- 2) Find the index  $\beta$  of vector  $\omega$  such that:

$$\beta = \arg \max_{i \in \{0, \dots, n-1\}} |\omega(i)| \quad (4)$$

- 3) Calculate the index  $\alpha$  of the closest codeword  $\hat{\mathbf{y}} \in \text{RM}(m, 1)$  to the received vector as:

$$\alpha = n\lambda + \beta \text{ where } \lambda = \begin{cases} 0, & \omega(\beta) \geq 0, \\ 1, & \omega(\beta) < 0. \end{cases} \quad (5)$$

- 4) The decoded codeword  $\hat{\mathbf{y}} \in \text{RM}(m, 1)$  is then given by:

$$\hat{\mathbf{y}} = \text{de2bi}(\alpha) \mathbf{G}_{(m,1)}, \quad (6)$$

where  $\text{de2bi}(\alpha)$  gives the right-MSB binary representation of the index  $\alpha$ .

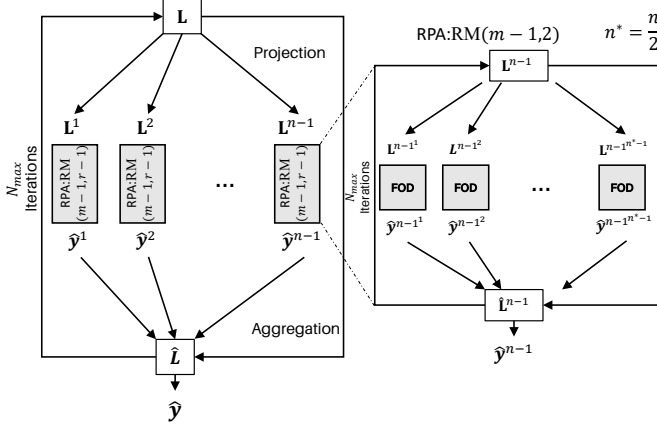


Fig. 1: RPA decoding for third-order RM codes based on [23].

### C. RPA decoding

As mentioned in Section I, a wide variety of decoding algorithms have been proposed for RM codes, some of which make use of the recursive structure and the large automorphism group of RM codes to propose projection-based methods. One of the algorithms taking advantage of the recursive structure of RM codes is RPA decoding. As shown in Fig. 1, the RPA method uses three steps, namely, projection, recursive decoding, and aggregation to decode a noisy received vector  $\mathbf{y}$  from a transmitted RM codeword  $\mathbf{c}$ .

In addition, the RPA algorithm has two flavors: 1) hard-decision decoding where  $\mathbf{y}$  is a binary vector, mostly used for binary symmetric channels (BSCs), and 2) soft-decision decoding where  $\mathbf{y}$  is a vector of LLRs, which can be used for more general communication channels like additive white Gaussian noise (AWGN) channels. The general structure of the RPA algorithm is the same for both hard- and soft-decision decoding, but the projection and aggregation steps are different. In this paper, we focus on soft-decision decoding. RPA decoding can be described by the following steps:

1) *Projection*: In this step,  $\mathbf{L}$  is transformed into  $n-1$  distinct vectors  $\mathbf{L}^1, \mathbf{L}^2, \dots, \mathbf{L}^{n-1}$  of length  $\frac{n}{2}$ . The projection rule is:

$$\mathbf{L}^i(j) = 2 \tanh^{-1} \left( \tanh \left( \frac{\mathbf{L}(j^a)}{2} \right) \tanh \left( \frac{\mathbf{L}(j^b)}{2} \right) \right), \quad (7)$$

where  $i \in \{0, \dots, n-1\}$  is the projection number,  $j \in \{0, \dots, \frac{n}{2}-1\}$ , and  $j^a$  and  $j^b$  are the coordinates of the original vector  $\mathbf{L}$  that are used to create  $\mathbf{L}^i(j)$ . The set of these coordinates for the  $i$ -th projection is given by:

$$\{(j^a, j^b) | j^a = j^b \oplus i; \forall j^b \in \{0, \dots, n-1\}\}. \quad (8)$$

We note that the above set contains  $n/2$  pairs of elements for which  $(j^a, j^b) = (j^b, j^a)$ . To avoid repetition, for each such pair, we remove  $(j^b, j^a)$  from the set. Equation (7) is often approximated by the so-called min-sum approximation [37], which is defined as:

$$\mathbf{L}^i(j) = \min \{|\mathbf{L}(j^a)|, |\mathbf{L}(j^b)|\} \text{sgn}(\mathbf{L}(j^a)) \text{sgn}(\mathbf{L}(j^b)). \quad (9)$$

2) *Recursive decoding*: In this step, each projected vector from the previous step is recursively decoded with RPA decoding for  $\text{RM}(m-1, r-1)$  until first-order codes are reached, for which the FHT-based first-order decoder (FOD) explained in Section II-B is applied.

### Algorithm 1: RevReorder

```

1 Input:  $i, n$ 
2 Output:  $\mathbf{U} = \{\mathbf{u}(j) \mid \mathbf{u}(j) = (j^a, j^b); j = 0, \dots, \frac{n}{2}-1\}$ 
3 if  $i < \frac{n}{2}$  then
4    $\mathbf{U} \leftarrow \left(0 \text{ to } \frac{n}{4}-1\right) \leftarrow \text{RevReorder}(i, \frac{n}{2})$ 
5    $\mathbf{U} \leftarrow \left(\frac{n}{4} \text{ to } \frac{n}{2}-1\right) \leftarrow \text{RevReorder}(i, \frac{n}{2}) + \frac{n}{2}$ 
6 else
7   for  $j=1 : \frac{n}{2}-1$  do
8      $\mathbf{U}(j) \leftarrow (j, j \oplus i)$ 
9   end
10   $\mathbf{U}(0) \leftarrow (0, i)$ 
11 end
12 return  $\mathbf{U}$ 
    
```

3) *Aggregation*: In this step, a per-coordinate average is taken from all the decoded codewords obtained from the recursive decoding step. Then, the hard-decoded binary vector  $\hat{\mathbf{y}}$  from the obtained LLR vector  $\hat{\mathbf{L}}$  is considered as an estimation for the transmitted codeword  $\mathbf{c}$ . This step is effectively the reverse of the projection step, so similarly to the projection step, it requires first finding the origins of each coordinate  $\hat{y}^i(j)$ ,  $j \in \{1, \dots, n/2\}$ , of the decoded codeword  $\hat{\mathbf{y}}^i$ ,  $i \in \{1, \dots, n-1\}$ . The RevReorder function given in Algorithm 1 finds the index pair  $\mathbf{U}^i(j) := (j^a, j^b)$  for each coordinate  $\hat{y}^i(j)$ , that was originally created by the  $i$ -th projection on the coordinates  $j^a$  and  $j^b$  of the input vector  $\mathbf{L}$ . Then, the average of the LLR values that were involved in creating each pair of coordinates  $\hat{y}(j^a)$  and  $\hat{y}(j^b)$  is calculated as follows:

$$\hat{\mathbf{L}}(j^a) = \frac{1}{n-1} \sum_{i=1}^{n-1} (1 - 2\hat{y}^i(j)) \mathbf{L}(j^b), \quad (10)$$

$$\hat{\mathbf{L}}(j^b) = \frac{1}{n-1} \sum_{i=1}^{n-1} (1 - 2\hat{y}^i(j)) \mathbf{L}(j^a). \quad (11)$$

Furthermore, as shown in Fig. 1, several iterations of the aforementioned steps are performed at every recursion level until either there are no changes in the decoded codeword or a pre-defined maximum number of iterations  $N_{\max}$  is reached, which the authors of [23] set to  $\lceil m/2 \rceil$ .

### D. IPA decoding and implementation

RPA decoding performs up to  $N_{\max}$  iterations at each level of the recursion, which increases the complexity and significantly complicates the RPA decoding structure. Fig. 2a shows the data flow of one iteration of the RPA algorithm to decode a codeword from an  $\text{RM}(m, 3)$  code. In this example, two levels of projection are shown with blue triangles until the first-order codes are reached. Then, the FODs, represented with green hexagons, decode the first-order RM codes. Similar to the projection step, there are two levels of aggregation represented with brown triangles. However, as shown in the highlighted region, the second projected vector requires an extra iteration on its recursive call to the RPA of  $\text{RM}(m-1, 2)$  after the first level of aggregation. This makes the other parallel branches stall. Furthermore, handling multiple iterations at

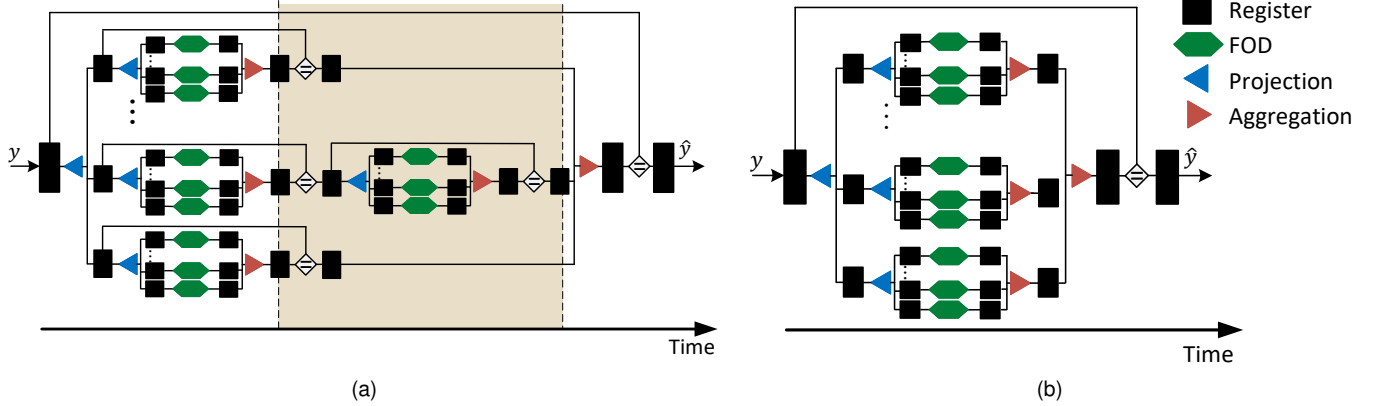


Fig. 2: Data-flow of one iteration of RPA (a) and IPA (b) decoding of an arbitrary codeword from  $RM(m, 3)$  code.

each recursion level requires complicated control circuitry and memory structure when implemented in hardware.

The work of [29] showed that it is possible to remove the iterations at internal levels of recursion with minimal degradation in error-correcting performance. This change transforms the recursive structure of RPA decoding into an iterative one that we call IPA, shown in Fig. 2b. This simplification of the RPA algorithm made it feasible to have a fully-parallel hardware implementation [29].

**Computational complexity:** The number of calls to the FOD function is commonly used as a measure to estimate the computational complexity of RPA decoding [24]–[28]. In the worst-case scenario, where the maximum number of iterations is performed for every recursive call, the total number of FOD calls can be determined as:

$$\Theta_{(m,r,N_{\max})} = N_{\max}^{r-1} \prod_{i=1}^{r-1} (2^{m-i-1} - 1). \quad (12)$$

However, with the simplification in IPA, the number of FODs will be decreased by  $N_{\max}^{r-2}$  times as there is no additional iteration for  $r - 2$  levels of recursion.

**Implementation:** The proposed architecture in [29] is comprised of three main components: *projection*, *FOD*, and *aggregation* for HD decoding of the received vector  $\mathbf{y} \in RM(m, r)$ . The *projection* component performs  $r - 1$  levels of projection. Each projection level is equipped with parallel projection units. These units are comprised of a crossbar to combine the corresponding coordinates for the desired projection of the input vector  $\mathbf{y}$ . Additionally, each unit includes an XOR circuits to apply the projection rule for HD decoding. The *FOD* component provides first-order decoding for all first-order codewords, obtained in the innermost level of projection, in parallel. Each *FOD* consists of the hardware implementation of each step in the decoding method explained in Section II-B. The *aggregation* component provides  $r - 1$  levels of aggregation. Each level includes the required parallel crossbars to expand the corresponding coordinates of the desired decoded codeword from  $RM(m - j, r - j)$  in order to estimate the codeword belonging to  $RM(m - j + 1, r - j + 1)$ , where  $j = \{1, \dots, r - 1\}$  represents the current level of aggregation.

Although the proposed architecture has very low latency, the resource utilization is extremely high due to its fully-parallel

---

#### Algorithm 2: IPA decoding for $RM(m, 2)$ codes

---

```

1 Input:  $\mathbf{L}, m, N_{\max}$ 
2 Output: Codeword  $\hat{\mathbf{y}}$ 
3  $n \leftarrow 2^m$ 
4 for  $j = 0 : N_{\max}$  do
5   for  $i = 1 : n - 1$  do
6      $\mathbf{L}^i \leftarrow \text{Projection}(\mathbf{L}, m, i)$ 
7      $\hat{\mathbf{y}}^i \leftarrow \text{FOD}(\mathbf{L}^i)$  // First-order decoder
8      $\mathbf{L}_{\text{agg}}^i \leftarrow \text{PreAggregation}(\mathbf{L}, \hat{\mathbf{y}}^i, i, m)$ 
9   end
10   $\mathbf{L} \leftarrow \text{Voting}(\mathbf{L}_{\text{agg}}^1, \dots, \mathbf{L}_{\text{agg}}^{n-1})$ 
11 end
12 for  $z = 0 : n - 1$  do
13    $\hat{\mathbf{y}} \leftarrow 1 - 1[\mathbf{L}(z) \geq 0]$  // Hard-decision
14 end
15 return  $\hat{\mathbf{y}}$ 

```

---

structure. Moreover, the decoder only supports HD decoding, which accepts binary vectors as input rather than LLR values. The projection and aggregation rules for HD decoding, which are explained in detail in [23] and [29], differ significantly from those of a soft-decision decoder.

In Section III, we present a flexible hardware implementation of the soft-decision IPA decoding for general binary-input memoryless channels. Our design can be easily configured based on the specific requirements of the application, offering a range of options from fully sequential to fully parallel configurations including partial-parallel configuration.

#### E. Polar code and SCL decoder

Polar codes [6] were ratified as the channel coding scheme of 5G enhanced mobile broadband (eMBB). However, even with highly specialized node-based SCL decoders capable of parallel bit decoding, meeting the stringent demands of high-reliability and low-latency in 5G (and beyond) is still a challenge for existing polar decoders. Recently, the authors of [32] proposed the first generalized node-based SCL decoding algorithm and presented a corresponding hardware implementation. By extending a generalized node called the *sequence repetition* (SR) node [38] to SCL decoding, this state-of-the-art polar decoder in [32]

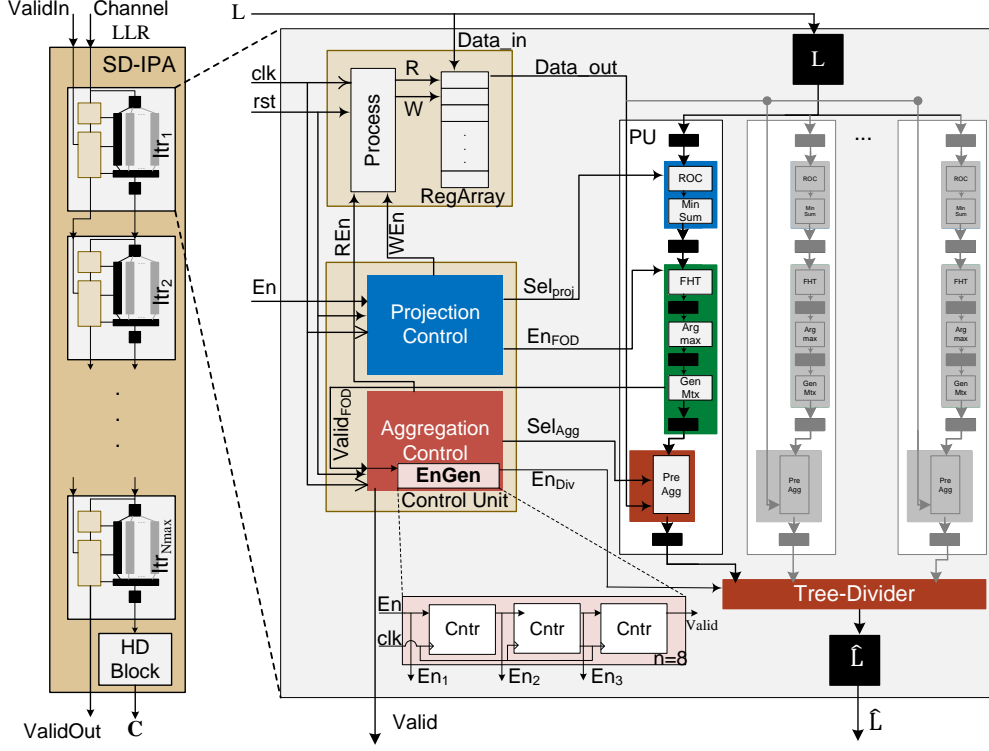


Fig. 3: Overview of our proposed second-order SIPA decoder architecture.

achieves increased decoding parallelism and more efficient utilization of computation units. This enhancement offers a competitive solution for 5G polar codes. We compare our proposed architecture against this state-of-the-art polar decoder in Section V.

### III. SECOND-ORDER SOFT-INPUT IPA DECODER ARCHITECTURE

Since first-order RM codes can be decoded optimally using the FHT [36], the lowest order for which RPA decoding is meaningful is  $r = 2$ . Therefore, we first describe a base architecture for IPA decoding for second-order RM codes and we explain how it can be extended to decode RM codes of higher order in Section IV.

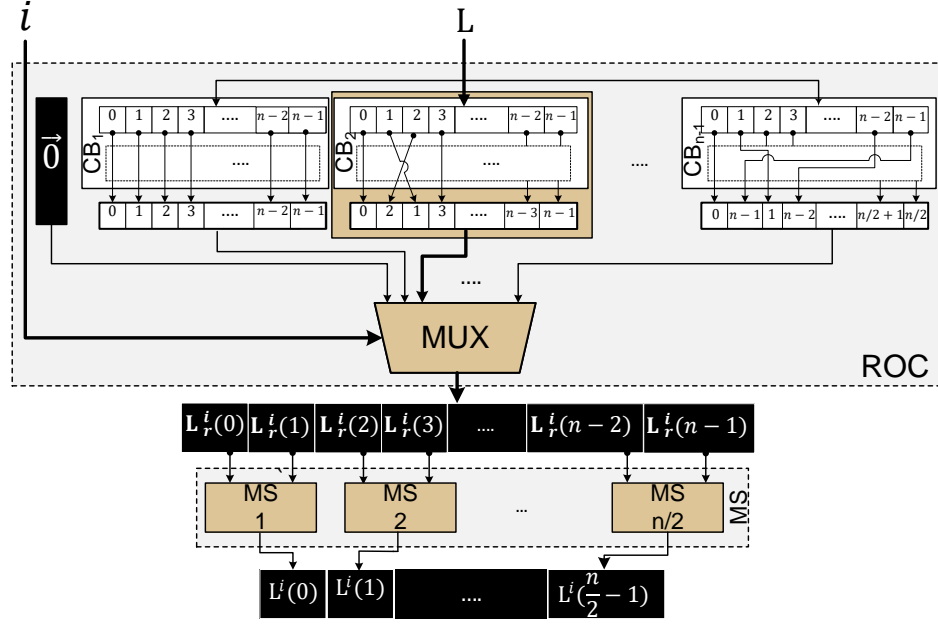
We implement soft-decision IPA decoding as described in Algorithm 2 with a pipelined architecture shown in Fig. 3 that has dedicated hardware blocks for each iteration of the outer for loop in lines 4-11 of Algorithm 2. The inputs of this architecture are a vector of channel LLRs and the *ValidIn* signal, which is high when a new vector of channel LLRs becomes available. The outputs are the decoded codeword and the *ValidOut* signal, which is high when the output is valid. Furthermore, our architecture includes processing units (PUs) that implement the projection step, first-order decoding, and a part of the aggregation step, which we call pre-aggregation. A hardware-friendly pipelined tree divider is also placed after the PU to complete the aggregation step of the IPA decoding. In the following subsections, we explain these components in detail.

#### A. Processing unit (PU)

The for loop in lines 5-9 of Algorithm 2 can be fully parallelized. Therefore, we design a PU which is pipelined and has appropriate hardware components to perform the Projection, FOD, and PreAggregation functions for any value of the loop variable  $i \in \{0, \dots, n-1\}$ , where  $i = 0$  corresponds to a dummy all-zeros vector that simplifies the implementation and that we explain in Section III-B. Thus, our second-order IPA decoder can be from fully-sequential with  $P = 1$  PUs to fully-parallel with  $P = 2^m$  PUs.

1) *Projection component*: This component includes two sub-components: Reordering (ROC) and min-sum (MS) shown in Fig. 4. The ROC contains  $n-1$  crossbars (CB) and a multiplexer that selects the crossbar for the corresponding projection. Each crossbar  $i$  is built with function  $\text{Reorder}(\mathbf{L}, i, m)$  represented in Algorithm 3 that finds the relevant coordinates of the input vector  $\mathbf{L}$ , taking the projection number  $i$  into account. Then, it reorders  $\mathbf{L}$  to put those coordinates in consecutive pairs. Consequently, the output vector  $\mathbf{L}_r^i$  is a reordered version of the input vector  $\mathbf{L}$  according to the projection number  $i$ . As shown in Fig. 4, the multiplexer selects the reordered vector  $\mathbf{L}_r^i$  corresponding to the current projection  $i$  determined with its *selector* input. For example, in Fig. 4, the highlighted crossbar shows that the second projection is currently performed.

The number of crossbars inside the ROC block depends on the number of available PUs. When  $P$  PUs are instantiated, the ROC block placed in the  $j$ -th PU,  $j \in \{0, \dots, P-1\}$ , includes the crossbars  $i$  such that  $i \bmod P = j$ . Therefore, for one PU, the ROC block contains all the crossbars, but for  $P > 1$ , each PU has a distinct circuit for its ROC sub-component.


 Fig. 4: Architecture of the *Projection* component including the ROC and *MS* sub-components.

**Algorithm 3: Reorder**

```

1 Input:  $\mathbf{L}, i, n$ 
2 Output:  $\mathbf{L}_r^i$ 
3 if  $i < \frac{n}{2}$  then
4    $\mathbf{L}_r^i(0 \text{ to } \frac{n}{2}-1) \leftarrow \text{Reorder}(\mathbf{L}(0 \text{ to } \frac{n}{2}-1), i, m-1)$ 
5    $\mathbf{L}_r^i(\frac{n}{2} \text{ to } n-1) \leftarrow \text{Reorder}(\mathbf{L}(\frac{n}{2} \text{ to } n-1), i, m-1)$ 
6 else
7   for  $j=1 : \frac{n}{2}-1$  do
8      $\mathbf{L}_r^i(2j) \leftarrow \mathbf{L}(j)$ 
9      $\mathbf{L}_r^i(2j+1) \leftarrow \mathbf{L}(j \oplus i)$ 
10  end
11   $\mathbf{L}_r^i(0) \leftarrow \mathbf{L}(0)$ 
12   $\mathbf{L}_r^i(1) \leftarrow \mathbf{L}(i)$ 
13 end
14 return  $\mathbf{L}_r^i$ 
    
```

In addition, the *MS* component shown in Fig. 4 consists of  $n/2$  blocks performing (9) on every pair of coordinates  $(\mathbf{L}_r^i(2j), \mathbf{L}_r^i(2j+1))$ ,  $j \in \{0, \dots, n/2-1\}$ , of the reordered vector  $\mathbf{L}_r^i$ .

2) *FOD component*: We use the architecture proposed in [29] modified for soft-input FOD, explained in Section II-B, to decode the vector  $\mathbf{L}^i$  corresponding to a first-order codeword obtained from the *Projection* component. As Fig. 5 demonstrates, the *FOD* component is pipelined and contains three modules: 1) the *FHT* module, which computes the fast Hadamard transform of  $\mathbf{L}^i$ , 2) the *Argmax* module to find the index of the maximum value of the output of the *FHT* module, and 3) the *GenMtx* module, which implements an encoder for an  $\text{RM}(m-1, 1)$  code. The output of the *FOD* component is the decoded codeword  $\hat{\mathbf{y}}^i$  corresponding to the projected vector  $\mathbf{L}^i$ .

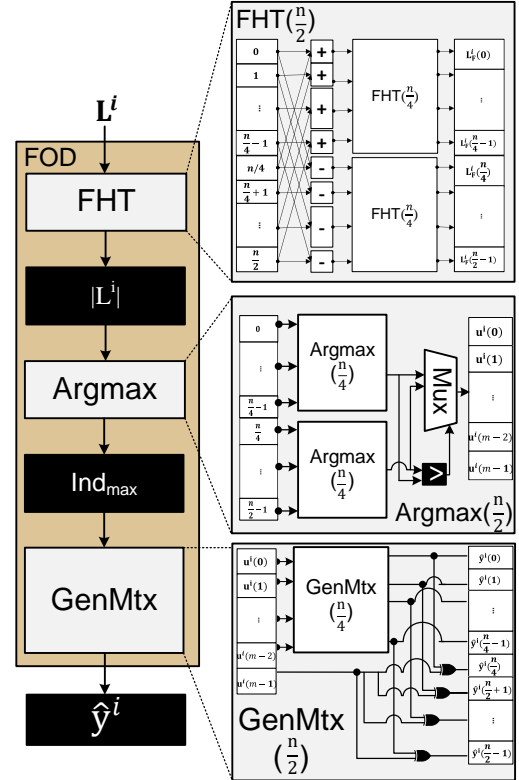


Fig. 5: Hardware implementation of FOD based on [29]

3) *PreAggregation component*: We have rewritten the aggregation step explained in Section II-C3 in Algorithm 4. The for loop on lines 4-11 of Algorithm 4 can be fully parallelized. The  $\text{PreAggregation}(\mathbf{L}, \hat{\mathbf{y}}^i, i, m)$  function in Algorithm 2 implements exactly this for loop. The *PreAggregation* function first calls *RevReorder* function corresponding



**Algorithm 4:** Aggregation

---

```

1 Input:  $\mathbf{L}, \hat{\mathbf{y}}^1, \hat{\mathbf{y}}^2, \dots, \hat{\mathbf{y}}^{n-1}, m$ 
2 Output:  $\hat{\mathbf{L}}$ 
3  $n \leftarrow 2^m$ 
4 for  $i=1 : n-1$  do
    // PreAggregation( $\mathbf{L}, \hat{\mathbf{y}}^i, i, m$ )
5    $\mathbf{U}^i \leftarrow \text{RevReorder}(i, m)$ 
6   for  $j = 0 : \frac{n}{2} - 1$  do
7      $(j^a, j^b) \leftarrow \mathbf{U}^i(j)$ 
8      $\mathbf{L}_{\text{agg}}^i(j^a) \leftarrow (1 - 2\hat{\mathbf{y}}^i(j)) \mathbf{L}(j^b)$ 
9      $\mathbf{L}_{\text{agg}}^i(j^b) \leftarrow (1 - 2\hat{\mathbf{y}}^i(j)) \mathbf{L}(j^a)$ 
10  end
11 end
    // Voting
12 for  $z = 0 : n-1$  do
13    $\hat{\mathbf{L}}(z) \leftarrow \frac{\sum_{i=1}^{n-1} \mathbf{L}_{\text{agg}}^i(z)}{n-1}$ 
14 end

```

---

to the  $i$ -th loop variable for finding the pairs of indices  $(j^a, j^b)$  of the input vector  $\mathbf{L}$  from which the coordinates  $\mathbf{L}^i(j)$ ,  $j \in \{0, \dots, n/2\}$  were originally created. Then, it applies the aggregation rule in (10), except the final averaging step. The hardware implementation of the PreAggregation function, shown in Fig. 6, has three sub-components:

- *Extension* module, which extends the length- $n/2$  decoded binary codeword  $\hat{\mathbf{y}}^i$  to the length- $n$  vector  $\hat{\mathbf{y}}_e^i$ . The extension rule is determined by the RevReorder function, which copies each coordinate  $\hat{\mathbf{y}}^i(j)$ ,  $j \in \{0, \dots, n/2\}$ , to their corresponding coordinates of  $\hat{\mathbf{y}}_e^i$  indexed by  $j^a$  and  $j^b$ .
- *ReArrangement* module, which flips the values of each pair of coordinates  $(\mathbf{L}(j^a), \mathbf{L}(j^b))$ . The output vector with flipped LLRs is  $\mathbf{L}_e^i$ . Similarly to the *Extension* module, each crossbar  $i$  in the *ReArrangement* module finds the pair  $(j^a, j^b)$  based on the RevReorder function.
- *TwosComp* module, which finds the two's complement of each coordinate  $j$ ,  $j \in \{0, \dots, n-1\}$ , of  $\mathbf{L}_e^i$  if  $\hat{\mathbf{y}}_e^i(j) = 1$ , as stated in lines 8-9 of Algorithm 4.

In addition, similarly to the *ROC* module, both the *Extension* and the *ReArrangement* modules contain a multiplexer to select the extension network or crossbar corresponding to the current decoded projected vector  $\hat{\mathbf{y}}^i$ . Moreover, the number of the extension networks and crossbars inside the *Extension* and *ReArrangement* modules is adjusted based on the number of instantiated PUs, similarly to the *ROC* module.

### B. Tree divider

The second for loop in the Aggregation function, shown in lines 12-14 of Algorithm 4, is the Voting function mentioned in Algorithm 2. More specifically, it takes the average value of the pre-aggregated vectors  $\mathbf{L}_{\text{agg}}^i$ ,  $i \in \{1, \dots, n-1\}$ , which are the outputs of PUs. As shown in line 11 of Algorithm 4, the Voting function needs an  $(n-1)$ -input adder to add up all  $\mathbf{L}_{\text{agg}}^i$ ,  $i = 1, \dots, n-1$ . In addition, it requires a divider to find the average value out of the accumulated LLRs. Implementing such adder and divider in hardware is relatively expensive, especially when  $n$  gets large. As a result,

we propose a tree divider structure to implement the Voting function.

The structure of the divider follows from the fact that the average value of two sets of  $n/2$  numbers is equal to the sum of the average values of each set divided by two:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i = \frac{\frac{1}{2} \sum_{i=1}^{n/2} x_i + \frac{1}{2} \sum_{i=n/2+1}^n x_i}{2}. \quad (13)$$

As a result, if  $n = 2^m$ , we end up with a tree structure represented in Fig. 7. In this structure, we have two-input adders with one bit extension and shifting elements that perform division by two, simply shifting the dividend one bit to the right. However, there are  $n-1$  pre-aggregated vectors instead of  $n$  required vectors for the tree divider. Consequently, we add a dummy all zero vector to the flow of our proposed decoder. This approximation affects the output of Voting function, but the effect is negligible, as we will show in Section V-A. The 0 vector is generated with  $i = 0$  in *Projection* and *PreAggregation* units as it is shown in Fig. 4 and Fig. 6.

In the case of a fully-sequential implementation with  $P = 1$  PUs, only one new divider input is available per clock cycle. Hence, we implemented the divider, depicted in Fig. 7, with  $\log(n)$  shift registers, two-input adders, and shifting elements. Each shift register contains two registers for two LLRs, data and enable input ports, as well as one data output port. The shift register shifts its internal array one position to the right and writes its current data input to the location of the shifted value. For the shift register in the first level  $l = 0$ , the data input is the output of the PU, and its enable is the valid output port of the PU. The data input for the shift register in level  $l > 0$  is the output of the shift register in level  $l-1$ , and its enable input is high when the shift register at level  $l-1$  has written two new data inputs. This signal is generated with *EnGen* module in the *Control* unit.

However, in a partially-parallel design with  $P > 1$  PUs, more than one input is ready at every clock cycle. Taking the tree divider's structure into account, we add a condition for choosing the number of PUs to keep the divider and pipeline stages as simple as possible. This constraint limits  $P = 2^p$ ,  $p \in \{0, \dots, m\}$ , so that we replace the shift registers with standard registers for the first  $p$  levels of the tree divider. Hence, we have  $2^{p-l}$  two-input standard registers at each level  $l$ ,  $l \in \{0, \dots, m-p-1\}$ . These registers are enabled with the valid output of the *PreAggregation* unit delayed by  $l-1$  clock cycles. Furthermore, we keep the sequential divider operating with shift registers for the last  $m-p$  levels of the divider. The shift register placed in the first level of the sequential part which is the  $(m-p)$ -th level is enabled with the valid signal of *PreAgg* unit delayed by  $p$  clock cycles. The enable signals for the rest of the shift registers are generated by the *Control* unit.

### C. Register array

Fig. 8 demonstrates the pipeline stages for a fully-sequential implementation of IPA, i.e.,  $P = 1$ , for RM(3,2). In this example, for simplicity each stage is assumed to require one clock cycle. However, in the hardware implementation, the length of each stage can be multiple clock cycles to balance the critical path depending on the input blocklength. As shown in the highlighted time slots in Fig. 8, the *Projection* component

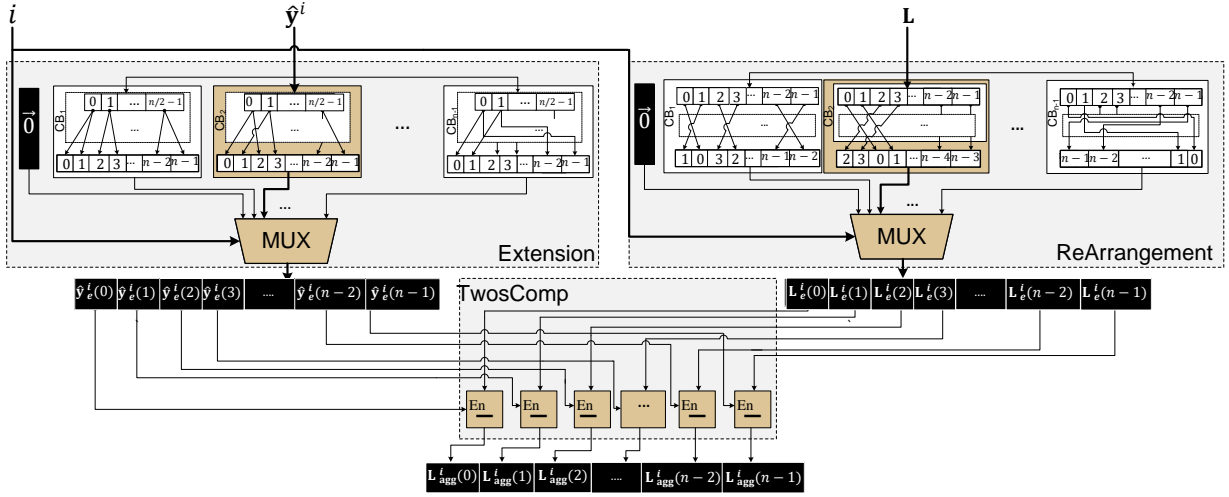
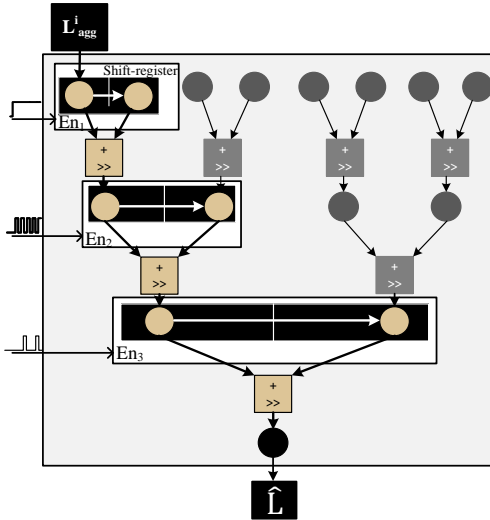


Fig. 6: Hardware implementation of the PreAggregation function.


 Fig. 7: Hardware implementation of the Voting function with a tree-like divider for an example of  $n = 8$ .

loads the new received vector while the *PreAggregation* component is still busy with the previously received vector. Therefore, we need an array of registers to store the channel LLRs for each input vector  $\mathbf{L}$  during the decoding process. The depth of this array is calculated by:

$$D_{\text{RegArr}} = \left\lceil \frac{t_{\text{agg}}}{n/P} \right\rceil + 1, \quad (14)$$

where  $t_{\text{agg}}$  is the number of clock cycles required for the pipeline stages before reaching the *PreAggregation* component,  $P$  is the number of PUs, and  $n$  is the block length. Hence, the more PUs we instantiate, the more registers we need. In the case of a fully-sequential implementation with  $P = 1$ , only two registers are required to store the channel LLRs of two input vectors as  $n > t_{\text{agg}}$  in most practical cases. On the other hand, in a fully-parallel implementation with  $P = n$  PUs, we have  $D_{\text{RegArr}} = t_{\text{agg}} + 1$ . As a result, this register array is generally relatively small. The register array is managed using a small *write* counter, which keeps track of the register that needs to be updated. Additionally, a counter is utilized to determine

the register that needs to be read when the *PreAggregation* component begins processing a new vector. The *Control* unit generates the necessary signals to facilitate the updating and reading of registers within the array.

#### D. Control unit

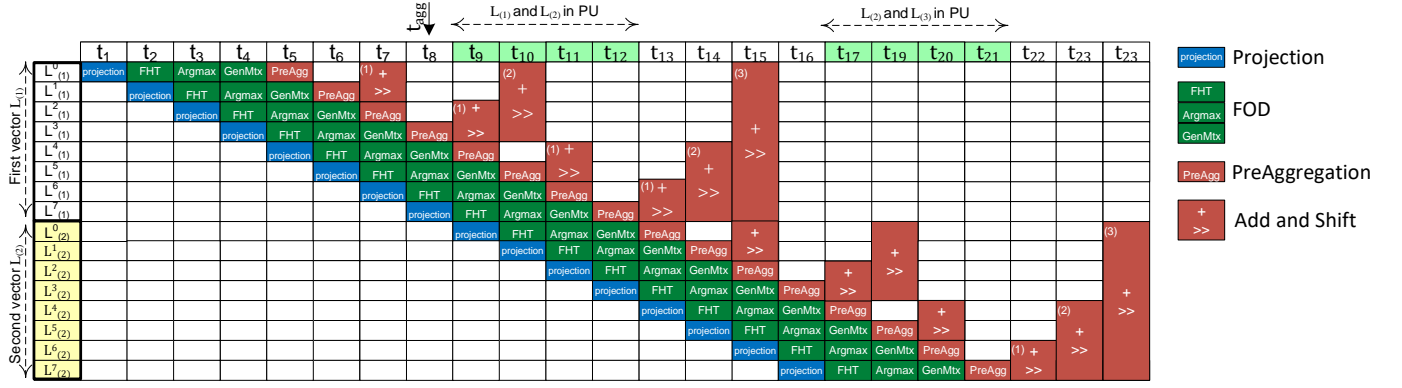
The *Control* unit generates all the controlling signals required for the pipeline structure, the PUs, the tree divider, and the memory. According to the pipeline table represented in Fig. 8 for an implementation that is not fully parallel, it is not possible to decode one codeword every clock cycle, so the *read* and *write* counters in the memory should be controlled carefully. Besides, the selectors for the multiplexers in the *Projection* and *PreAggregation* components of the PU do not have the same value, e.g., when the *PreAggregation* starts processing the first decoded projected codeword at  $t_5$ , the *Projection* starts the 5-th projection in the example shown in Fig. 8. Moreover, as mentioned earlier, the data input  $\mathbf{L}$  is not the same for the *Projection* and *PreAggregation* units in one PU as shown in the highlighted part from time instances  $t_9$  to  $t_{12}$  and from  $t_{17}$  to  $t_{21}$  in Fig. 8. Therefore, the *Control* unit runs two state machines simultaneously to generate the controlling signals for the projection and aggregation steps.

In addition, the *EnGen* component of the *Control* unit generates the enable signals for the registers used in the tree divider. It consists of  $m-p$  cascaded counters for implementing the IPA decoder for the  $\text{RM}(m, 2)$  code with  $2^p$  PUs. Each counter makes its output high for one clock cycle when it counts two high levels of its enable signal. The enable signal of the first counter is the valid output of the PU delayed by  $p$  clock cycles. Furthermore, the remaining counters in the consecutive levels are enabled with the previous level's output.

#### E. Iteration

After the aggregation step in the RPA algorithm, a comparison is made to check whether the output of the current iteration converges to its input or not. If the condition is satisfied, the algorithm stops before performing all  $N_{\text{max}}$  iterations. However, since very few iterations are generally required, we removed




 Fig. 8: Pipeline stages for the fully-sequential second-order IPA decoder for an example of  $n = 8$ .

the early stopping condition in our proposed architecture to simplify the pipeline flow and to have a constant throughput.

For a decoder with  $N_{\max}$  iterations, we cascade  $N_{\max}$  copies of the single-iteration hardware, as shown in Fig. 3, such that the *ValidIn* input port of  $i$ -th iteration is the *ValidOut* of  $(i-1)$ -th iteration. As Fig. 3 shows, there is a hard-decision maker module at the end of the flow. This module considers the most-significant bit of the estimated LLRs of the  $N_{\max}$ -th iteration as the binary decoded codeword.

#### F. Throughput and latency

As shown in Fig. 8, a new codeword with blocklength  $n$  can be processed every  $n$  clock cycles in a fully-sequential IPA decoder. However, in the partially-parallel IPA decoder with  $P$  PUs, a new codeword can be inserted into the pipeline every  $\frac{n}{P}$  clock cycles. In general, the throughput of the proposed second-order IPA decoder is:

$$\text{Thr}_{\text{Mbps}} = \frac{P \times \text{Frequency}}{n} \times n = P \times \text{Frequency}, \quad (15)$$

where the frequency is given in MHz.

As all steps of the IPA decoder are pipelined, the latency of one iteration of the second-order IPA decoder is:

$$t_{(m,2)} = (t_{\text{proj}} + t_{\text{FOD}} + t_{\text{PreAgg}}) + \left( \frac{2^m}{P} - 1 \right) + t_{\text{Divider}} + t_{\text{IO}}, \quad (16)$$

where  $t_{\text{proj}}$ ,  $t_{\text{FOD}}$ , and  $t_{\text{PreAgg}}$  are the delays of the *Projection*, *FOD*, and *PreAggregation* components, respectively, measured in clock cycles. In addition,  $t_{\text{Divider}} = m$  and  $t_{\text{IO}} = 2$  corresponding to two input and output registers. In our design, we consider  $t_{\text{proj}} = t_{\text{PreAgg}} = 1$ , and  $t_{\text{FOD}} = 3$  or  $t_{\text{FOD}} = 4$ , depending on the input blocklength. The total latency of the second-order IPA decoder with  $N_{\max}$  iterations is  $t_{(m,2)} \times N_{\max}$ .

#### IV. PROPOSED ARCHITECTURE FOR SOFT-INPUT IPA DECODER FOR RM CODES WITH $r > 2$

IPA decodes  $\text{RM}(m, r)$  codes with  $r > 2$  by producing  $2^m - 1$  projected codewords from  $\text{RM}(m-1, r-1)$ , as discussed in Section II-C. Therefore, we can generalize the architecture shown in Fig. 3 to decode RM codes with  $r > 2$  by adding *Projection* and *PreAggregation* components along with the tree divider for any level  $r > 2$ . Additionally, a dedicated memory with different blocklength and depth is required for each level

of  $r > 2$ . Finally, we add a *Control* unit for each level of  $r > 2$  to generate the control signals required for the corresponding  $r$ . Fig. 9 shows an overview of the proposed architecture for one iteration of the third-order IPA decoder.

We keep the dummy all-zero vector inserted in the base second-order IPA decoder, and the projection, first-order decoding, and aggregation steps are performed on this dummy vector. As a result, some clock cycles are lost, but it simplifies the divider and *Control* unit as mentioned in Section III. However, for the levels  $r > 2$ , we do not use the dummy all-zero vector because it results in wasting significant number of clock cycles depending on the structure of the base second-order decoder. Therefore, to keep the tree divider and the whole structure simple, we only freeze the entire decoder for one clock cycle at each level of  $r > 2$  and we insert zeros in the frozen cycle to the tree divider directly.

Similarly to the second-order IPA decoder, the proposed  $r$ -order IPA decoder is able to be adjusted from fully-sequential to fully-parallel. A fully-sequential architecture is obtained by instantiating one PU in the level  $r = 2$  and a fully-parallel implementation is obtained by instantiating all  $\left( \prod_{i=0}^{r-3} 2^{m-i} - 1 \right) \times 2^{m-r+2}$  required PUs for the level  $r = 2$ . Any other number of available PUs, following the constraint mentioned in Section III-B, results in partially-parallel architectures. It is worth mentioning that with  $P > 2^{m-r+2}$ , i.e., with more than the required PUs for a fully-parallel base second-order IPA, multiple second-order IPA decoders will be instantiated in the design.

Similar to the second-order IPA decoder, the throughput of the higher-order IPA decoder is calculated as:

$$\text{Thr}_{\text{bps}} = \frac{P \times \text{Frequency}}{\left( \prod_{i=0}^{r-3} 2^{m-i} - 1 \right) \times 2^{m-r+2}} \times n. \quad (17)$$

The latency of one iteration of the  $r$ -order IPA decoder is:

$$t_{(m,r)} = t_{\text{proj}} + t_{(m-1,r-1)} + t_{\text{PreAgg}} + \left\lceil \frac{d_{r-1} \times (2^m - 2)}{N_{\text{Dec}}} \right\rceil + m, \quad (18)$$

where  $N_{\text{Dec}}$  is the the number of instantiated decoders for the  $\text{RM}(m-1, r-1)$  code. In addition,  $d_{r-1}$  is the delay that should be considered between inserting the inputs to the  $(r-1)$ -th level of the decoder, which is dependent on the number of available PUs. The latency of the second-order base IPA decoder, i.e.,  $t_{(m,2)}$ , is calculated based on (16).

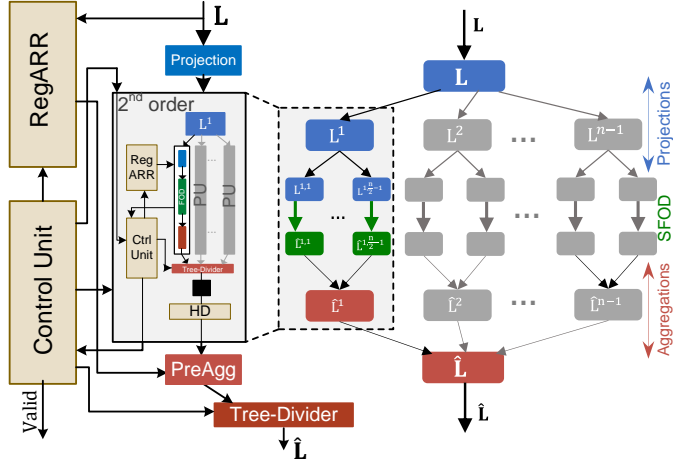


Fig. 9: Overview of third-order IPA decoder architecture for one iteration.

## V. SIMULATION AND IMPLEMENTATION RESULTS

### A. Simulation results

In this section, we present simulation results of the decoding performance of the IPA decoder for both hard- and soft-decision decoding including the approximate divider in the aggregation step. It is important to note that when we refer to our work as IPA in this section, we specifically mean soft-decision IPA. For hard-decision IPA, we use the term HD-IPA. First, we compare IPA decoding to the baseline RPA algorithm for different codes. Then, we compare IPA decoding for different numbers of iterations as well as different quantization bit-widths to justify our design choices for the hardware implementation. Furthermore, we compare IPA decoding of RM codes to SCL decoding of polar codes with the same blocklength and rate. We use the 5G-compatible SCL decoder of [39] with an 11-bit cyclic redundancy check (CRC). The target codes for simulations are RM(6, 3) and RM(7, 2) to cover different orders as well as different rates. For the comparison to polar codes, we consider polar codes with  $(n, k)$  pairs of (64, 42) and (128, 29), as they have the same rate and blocklength to the RM(6, 3) and RM(7, 2) codes, respectively. All simulations are performed over the AWGN channel.

1) *RPA decoding vs IPA decoding*: Fig. 10 shows the frame error rate (FER) for RPA and IPA decoding implemented with the exact projection rule (7) and the min-sum approximation (9). As in [23], we set the maximum number of iterations  $N_{\max} = \lceil m/2 \rceil$  for the simulations represented in Fig. 10. For second-order RM codes, the only difference between IPA and RPA is the approximate divider during aggregation. The simulation results in Fig. 10 show that the effect of this approximate divider is negligible as there is effectively no performance difference between IPA and RPA for the RM(7, 2) code. For the RM(6, 3) code, the performance difference between IPA and RPA is also negligible, even though  $\sim 567$  internal iterations are removed by the IPA algorithm. In addition, Fig. 10 shows that the degradation caused by the min-sum update rule (9) compared to the exact update rule (7) is small for both codes. In addition, we aim to compare the hardware implementation of our proposed IPA to the fully-parallel HD-IPA architecture proposed in [29]. Therefore, a comparison between the HD-

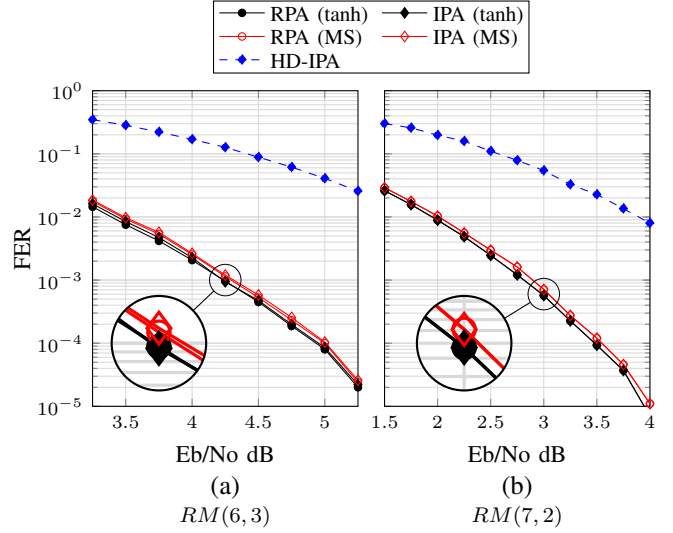


Fig. 10: Comparison of different flavours of RPA and IPA for Reed-Muller codes over AWGN channel.

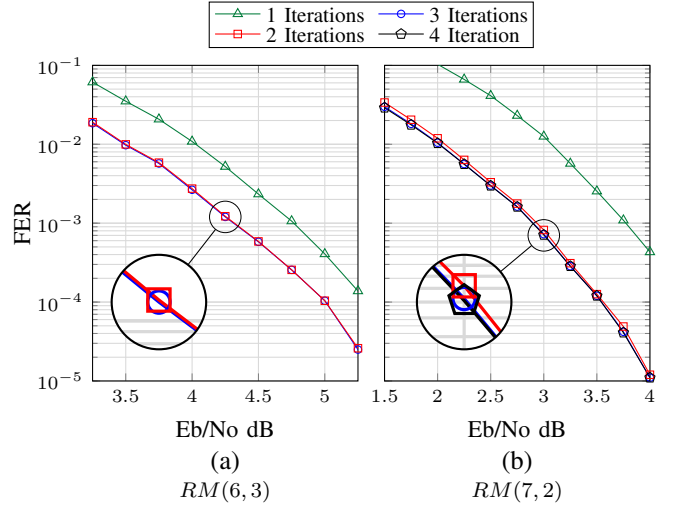


Fig. 11: Comparison of different numbers of iterations for the IPA algorithm.

IPA [29] and IPA is made in Fig. 10. As expected, the error-correcting performance of the HD-IPA shows a noticeable degradation of more than 2 dB and 1.75 dB for RM(6,3) and RM(7,3) codes, respectively, in comparison to the soft-input IPA.

2) *Number of iterations*: The hardware cost of our architecture represented in Fig. 9 scales linearly with the number of iterations. Therefore, we also explored whether the number of iterations can be reduced without degrading the error-correcting performance significantly. Fig. 11 shows the performance of IPA decoding with 1 to  $\lceil m/2 \rceil$  iterations. We observe that the performance loss with one iteration is significant compared to  $\lceil m/2 \rceil$  iterations for both examined RM codes. However, there is no performance degradation for the RM(6, 3) code and only a very small degradation for RM(7, 3) with 2 iterations, instead of  $\lceil m/2 \rceil = 3$  and  $\lceil m/2 \rceil = 4$  iterations, respectively. Therefore, we set  $N_{\max} = 2$  for the hardware implementation of the IPA decoder for both of these codes, reducing the

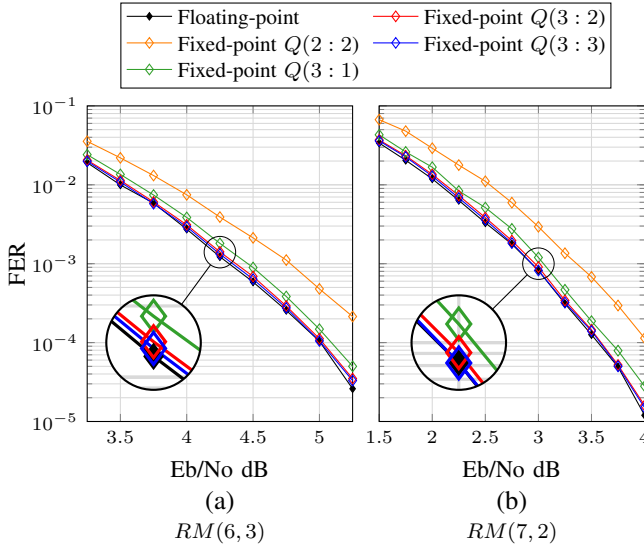


Fig. 12: FER comparison between floating-point and different fixed-point implementation of IPA decoding.

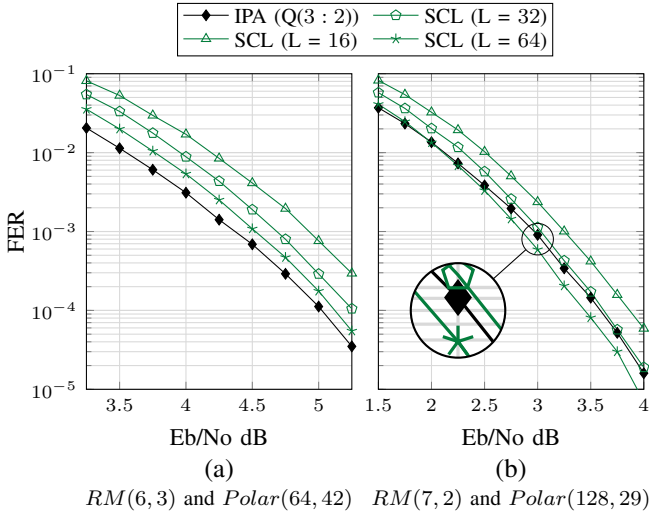


Fig. 13: FER comparison between 5-bit fixed-point IPA decoding and floating-point SCL decoding.

required hardware and the latency by 33% and 50% for each code, respectively.

3) *Quantization bit-width*: For the fixed-point implementation, we quantized all LLRs using a  $Q(q_i : q_f)$  quantization scheme, where  $q_i$  and  $q_f$  is the number of integer and fractional bits, respectively. We increase the bit-width by one at each stage of the FHT inside the FOD because it is very sensitive to saturation. The results of all other operations are always clipped to remain within the representable range. Fig. 12 shows the results for the different values of  $q_i$  and  $q_f$ . We observe that the 5-bit quantized IPA decoder with  $q_i = 3$  and  $q_f = 2$  is almost as accurate as the floating-point IPA decoder for both considered codes. Therefore, we present hardware implementation results using 5-bit LLRs.

4) *IPA decoding of RM codes vs SCL decoding of polar codes*: For the final comparison, we compare the performance of our selected 5-bit quantized IPA with 2 iterations to the floating-point SCL decoding of 5G polar codes with 11-bit

CRC. As Fig. 13 demonstrates, the quantized IPA decoder for RM(6, 3) outperforms the floating-point SCL decoder for a polar code of the same blocklength and rate with list sizes of  $L = 16$ ,  $L = 32$ , and  $L = 64$ . Moreover, the quantized IPA decoder for RM(7, 2) outperforms the floating-point SCL decoder for a polar code of the same blocklength and rate with list size of  $L = 16$  and  $L = 32$ , but has worse performance for  $L = 64$ .

### B. ASIC synthesis results

In this section, we present synthesis results for our proposed IPA decoder architecture. The IPA decoder has been implemented in VHDL and synthesized using the Cadence Genus RTL compiler with the STM 28nm FD-SOI technology in the slow-slow corner and at 25° C. To obtain accurate power measurements, we performed gate-level (GL) simulations by generating a standard delay file (SDF) through synthesis. Subsequently, we utilized the SDF to perform GL simulations in the Cadence Xcelium simulator. To ensure an accurate power estimate, we incorporated the switching activity obtained from the GL simulation for  $10^3$  frames into our analysis. We show the synthesis results for the IPA decoder for RM(7, 2) and RM(6, 3) codes. As concluded from Fig. 12, the IPA decoder has been implemented for 5-bit input LLRs, i.e.,  $Q(3 : 2)$  and two decoding iterations. Furthermore, we provide the synthesis results for different numbers of PUs employed in the IPA decoder to show the trade-off between area consumption and latency as well as throughput.

Since, to the best of our knowledge, there are no other hardware implementations of soft-decision projection-aggregation based RM decoders in the literature, we compare the area consumption, latency, and throughput of the IPA decoder against the state-of-the-art SCL decoder of [32] synthesized for polar codes with the same blocklength and information rate and with the same technology and settings. The SCL decoder has been synthesized for 6-bit quantization to ensure that there is minimal performance loss between the SCL decoder with quantized LLRs and the floating-point LLRs [32]. Additionally, we select the list sizes based on Fig. 13 and we allow the SCL decoder to have slightly worse error-correcting performance if necessary to be as conservative in our comparison as possible. We also synthesized the fully-parallel HD-IPA decoder [29] for comparison. To ensure a fair comparison with the soft-decision IPA, we also synthesized the HD-IPA decoder for 2 iterations.

Table I presents the synthesis results for soft-decision IPA decoders with various numbers of PUs and fully-parallel HD-IPA for RM(6, 3) code. It also compares the IPA decoders with the SCL decoder of [32] for the 5G polar code with  $n = 64$  and  $k = 42$ , which has the same rate and blocklength as the RM code. The second-order decoder instantiated in the IPA decoder for the RM(6, 3) code performs 32 projections, resulting in 32 PUs in case of a fully-parallel implementation. Therefore, we synthesized the IPA decoder for 16 and 32 PUs to show the effects of the partially-parallel and fully-parallel second-order decoder used in the IPA decoder for decoding a third-order code. In addition, Table I also includes synthesis results for 64 PUs, meaning that two fully-parallel second-order decoders are employed in the third-order IPA decoder. The results show that increasing the number of PUs from 16 to 32 results in a 58% increase in the area while the latency decreases by

TABLE I

SYNTHESIS RESULTS FOR OUR PROPOSED IPA DECODER FOR RM(6, 3) CODE AND FOR THE STATE-OF-THE-ART SCL DECODER OF [32] FOR A 5G POLAR CODE WITH  $n = 64$  AND  $k = 42$ .

Code Decoder	RM(6, 3)			Polar (64, 42)
	IPA	HD-IPA [29]	SCL [32]	
List size (L)	-	-	-	64
Number of PUs (P)	16	32	64	8001 <sup>a</sup>
Clock Rate (MHz)	714	714	714	500
Latency (cc)	294	168	106	26
Latency ( $\mu$ s)	0.411	0.235	0.148	0.052
Throughput (Mbps)	357	714	1428	1231
Area ( $\text{mm}^2$ )	0.38	0.61	1.21	4.08
Area Eff. (Gbps/ $\text{mm}^2$ )	0.94	1.17	1.60	0.30
Power (mW)	317	505	801	NA <sup>b</sup>
Energy (pJ/b)	888	707	561	NA

<sup>a</sup> There is no PU defined in [29], but there are  $127 \times 63$  parallel units including two levels of projection, FOD, and two levels of aggregation.

<sup>b</sup> Not available: Due to the high resource utilization, the post-synthesis power analysis failed to run.

TABLE II

SYNTHESIS RESULTS FOR OUR PROPOSED IPA DECODER FOR RM(7, 2) CODE AND FOR THE STATE-OF-THE-ART SCL DECODER OF [32] FOR A 5G POLAR CODE WITH  $n = 128$  AND  $k = 29$ .

Code Decoder	RM(7, 2)			Polar (128, 29)
	IPA	HD-IPA [29]	SCL [32]	
List size (L)	-	-	-	32
Number of PUs (P)	2	4	8	127
Clock Rate (MHz)	555	555	555	384
Latency (cc)	156	92	60	15
Latency ( $\mu$ s)	0.281	0.165	0.108	0.039
Throughput (Mbps)	1110	2220	4440	3282
Area ( $\text{mm}^2$ )	0.33	0.50	0.88	1.90
Area Eff. (Gbps/ $\text{mm}^2$ )	3.30	4.35	5.04	1.72
Power (mW)	214	364	662	237
Energy (pJ/b)	192	163	149	72

42%. However, implementing the IPA decoder with 64 PUs results in slightly higher than two times area consumption compared to 32 PUs since there are two decoders with separate control units and memories instantiated in the third-order IPA decoder. Furthermore, we observed that HD-IPA shows significantly lower latency compared to IPA due to its fully-parallel architecture. However, it has a significantly worse error-correcting performance, as shown in Fig. 10. Moreover, this remarkably low latency is achieved at the cost of using significantly more resources, making it impractical for many applications. This further emphasizes the necessity for a partial-parallel architecture, such as our proposed architecture, which offers a more viable alternative. Additionally, despite its lower latency, HD-IPA does not achieve high throughput due to its non-pipelined structure, resulting in very low area efficiency. We also see that all different configurations for the proposed IPA decoder have a significantly smaller area and a much higher throughput than the SCL decoder, resulting in an improvement in the area efficiency of one order of magnitude. At the same time, the IPA decoders have 13% and 45% lower absolute latency than the SCL decoder for  $P = 32$  and  $P = 64$ , respectively.

Table II presents the synthesis results for various soft-decision IPA decoders and HD-IPA for the RM(7, 2) code as well as an SCL decoder for the 5G polar code with  $n = 128$  and  $k = 29$ . We see that HD-IPA shows up to 7 times lower latency but requires up to 6 times more resources. Similarly to what was observed in Table I, HD-IPA is less efficient in terms of resource usage compared to IPA because it does not use a pipelined architecture. However, the resource efficiency for

RM(7, 2) is better than RM(6, 3) due to the lower number of clock cycles required for decoding each codeword. It is again important to highlight that the error-correcting performance of HD-IPA is significantly worse than that of the soft-decision IPA as it is shown in Fig. 10. Similarly to the previous results, all IPA decoders have a significantly smaller area and a much higher throughput than the SCL decoder, resulting in an improvement in the area efficiency by a factor between 6 and 9, depending on the number of PUs. At the same time, the IPA decoders have 29% and 54% lower absolute latency than the SCL decoder for  $P = 4$  and  $P = 8$ , respectively.

Table I and Table II also provide a comparison of the power consumption among the soft-decision IPA, HD-IPA, and SCL decoder. The proposed IPA architecture shows higher power consumption compared to the other two decoders. This can be attributed to its pipelined architecture, where all components remain active during each clock cycle, contrary to the SCL decoder that mostly consists of memory. Furthermore, it is evident that as the level of parallelism, indicated by the number of PUs, increases the energy consumption per bit decreases. This implies that the IPA architecture can be configured to deliver a high-throughput decoder with reasonable energy consumption per bit if such performance is required.

We have included detailed information regarding the average area per iteration and the power consumption for each iteration of any block in the IPA decoder in Table III. As expected, the PU utilizes the largest area among all blocks. Additionally, within the PU, the *FOD* block is the most area-intensive component. The area consumption of the register array, designed to fulfill pipeline requirements, is negligible compared to other blocks, as it only needs to store a small number of codewords. Furthermore, it is worth noting that the area utilization for one PU in the second-order decoder embedded within the decoder for RM(6, 3) is relatively low. This is because we employed a fully-parallel second-order decoder with 32 PUs at that level, resulting in smaller *Projection* and *PreAggregation* components. Similarly, the divider at that level appears relatively large compared to other parts, as it has 32 vectors ready at each clock cycle, requiring five levels of parallel adders and shift registers, as depicted in Fig. 7.

In terms of power consumption, we reported the power values for both iterations. The second iteration is less computationally intensive, as the majority of error correction occurs in the first iteration. Consequently, the *FOD* units exhibit lower activity levels during the second iteration. Therefore, we observe approximately 18% and 14% less power consumption for the second iteration of the IPA decoder for RM(7, 2) and RM(6, 3) codes, respectively. Additionally, we noticed that the power consumption of the control units, register array, and divider for the second iteration remains almost the same as the first iteration, as their activities are not dependent on input values.

## VI. CONCLUSION

In this work, we described a pipelined and flexible architecture for soft-decision IPA decoding of RM codes. We used several algorithmic<sup>1</sup> and architectural optimizations to reduce

<sup>1</sup>We note that IPA decoding can be further simplified using recently published methods [28], [40]. Since our results, considered as a baseline, are already highly promising even without these simplifications, we consider the quantification of these additional improvements as future work.

TABLE III

AREA UTILIZATION AND POWER CONSUMPTION OF EACH COMPONENT OF THE IPA DECODER

	RM(7, 2), P = 4			RM(6, 3), P = 32		
	Area <sup>a</sup> mm <sup>2</sup>	Power(mW) itr. 1	itr. 2	Area <sup>a</sup> mm <sup>2</sup>	Power(mW) itr. 1	itr. 2
IPA( $r = 3$ )	-	-	-	0.306	271.76	232.84
Projection( $r = 3$ )	-	-	-	0.007	5.40	4.03
IPA( $r = 2$ )	0.249	199.94	162.10	0.266	244.42	208.56
PU	0.048	45.06	36.79	0.008	7.02	05.90
Projection	0.008	5.98	4.32	0.001	0.34	0.27
FOD	0.032	27.35	21.69	0.006	5.45	4.46
PreAggregation	0.008	8.19	7.73	0.000	0.18	0.16
Divider	0.047	25.64	23.04	0.023	25.94	25.20
Register array	0.075	3.53	3.53	0.006	5.56	5.42
Control unit	0.001	0.07	0.07	0.001	0.00	0.00
PreAggregation( $r = 3$ )	-	-	-	0.005	4.39	3.22
Divider( $r = 3$ )	-	-	-	0.021	13.21	12.78
Register array ( $r = 3$ )	-	-	-	0.003	2.20	2.22
Control unit( $r = 3$ )	-	-	-	0.000	0.01	0.01

<sup>a</sup> Average area utilization for one iteration.

the hardware implementation complexity while maintaining an error-correcting performance that is very close to the original RPA decoding algorithm from which IPA is derived. Our synthesis results with an STM 28 nm technology demonstrate that the IPA decoder exhibits notable advantages in terms of area efficiency, with improvements of up to 10 times, and latency reductions of up to 54% when compared to the SCL decoder. These improvements are achieved while maintaining comparable error-correcting performance, highlighting the potential benefits of the IPA decoder for applications that require high reliability and low latency. However, the post-synthesis simulation results showcases significantly higher power consumption compared to a state-of-the-art SCL decoder for polar codes. Therefore, the proposed flexible architecture of IPA enables a wide range of trade-offs between area and power consumption on one side, and latency and throughput on the other side.

## VII. REFERENCES

- [1] N. H. Mahmood *et al.*, “White paper on critical and massive machine type communication towards 6G,” *arXiv 2004.14146*, 2020.
- [2] G. Durisi, T. Koch, and P. Popovski, “Toward massive, ultra-reliable, and low-latency wireless communication with short packets,” *Proceedings of the IEEE*, pp. 1711–1726, Sep 2016.
- [3] H. Chen *et al.*, “Ultra-reliable low latency cellular networks: Use cases, challenges and approaches,” *IEEE Communications Magazine*, vol. 56, no. 12, pp. 119–125, 2018.
- [4] R. Gallager, “Low density parity check codes,” *Cambridge*, vol. 1, pp. 1–73, 1963.
- [5] C. Berrou and A. Glavieux, “Near optimum error correcting coding and decoding: Turbo-codes,” *IEEE Transactions on communications*, vol. 44, no. 10, pp. 1261–1271, 1996.
- [6] E. Arıkan, “Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels,” *IEEE Transactions on Information Theory*, vol. 55, no. 7, pp. 3051–3073, jul 2009.
- [7] M. C. Coşkun *et al.*, “Efficient error-correcting codes in the short blocklength regime,” *Physical Communication*, vol. 34, pp. 66–79, Jun. 2019.
- [8] T. Tonnellier, M. Hashemipour-Nazari, N. Doan, W. J. Gross, and A. Balatsoukas-Stimming, “Towards practical near-maximum-likelihood decoding of error-correcting codes: An overview,” in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Jun. 2021.
- [9] I. Reed, “A class of multiple-error-correcting codes and the decoding scheme,” *Transactions of the IRE Professional Group on Information Theory*, vol. 4, no. 4, pp. 38–49, Sep 1954.
- [10] D. E. Muller, “Application of boolean algebra to switching circuit design and to error detection,” *Transactions of the IRE Professional Group on Electronic Computers*, vol. EC-3, no. 3, pp. 6–12, Sep 1954.
- [11] I. Dumer, “Recursive decoding and its performance for low-rate Reed–Muller codes,” *IEEE Transactions on Information Theory*, vol. 50, no. 5, pp. 811–823, May 2004.
- [12] B. Sakkour, “Decoding of second order Reed–Muller codes with a large number of errors,” in *IEEE Information Theory Workshop*, Aug 2005.
- [13] I. Dumer, “Soft-decision decoding of Reed–Muller codes: A simplified algorithm,” *IEEE Transactions on Information Theory*, vol. 52, no. 3, pp. 954–963, Mar 2006.
- [14] I. Dumer and K. Shabunov, “Soft-decision decoding of Reed–Muller codes: Recursive lists,” *IEEE Transactions on Information Theory*, vol. 52, no. 3, pp. 1260–1266, Mar 2006.
- [15] R. Sappharishi, A. Shpilka, and B. L. Volk, “Efficiently decoding Reed–Muller codes from random errors,” in *IEEE Transactions on Information Theory*, vol. 63, 2017, pp. 1954–1960.
- [16] D. J. Costello and G. D. Forney, “Channel coding: The road to channel capacity,” *Proceedings of the IEEE*, vol. 95, no. 6, pp. 1150–1177, Jun 2007.
- [17] E. Abbe, A. Shpilka, and A. Wigderson, “Reed–Muller codes for random erasures and errors,” in *Annual ACM Symposium on Theory of Computing*, Jun 2015.
- [18] S. Kudekar, S. Kumar, M. Mondelli, H. D. Pfister, E. Sasoglu, and R. L. Urbanke, “Reed–Muller codes achieve capacity on erasure channels,” *IEEE Transactions on Information Theory*, vol. 63, no. 7, pp. 4298–4316, Jul 2017.
- [19] E. Abbe and M. Ye, “Reed–Muller codes polarize,” *IEEE Transactions on Information Theory*, vol. 66, no. 12, pp. 7311–7332, 2020.
- [20] O. Sberlo and A. Shpilka, “On the performance of Reed–Muller codes with respect to random errors and erasures,” in *Annual ACM-SIAM Symposium on Discrete Algorithms*, 2020, pp. 1357–1376.
- [21] E. Arıkan, H. Kim, G. Markarian, Ü. Özgür, and E. Poyraz, “Performance of short polar codes under ML decoding,” in *ICT-MobileSummit Conference*, Sep. 2009, pp. 10–12.
- [22] M. Mondelli, S. H. Hassani, and R. L. Urbanke, “From polar to Reed–Muller codes: A technique to improve the finite-length performance,” *IEEE Transactions on Communications*, vol. 62, no. 9, pp. 3084–3091, Sep 2014.
- [23] M. Ye and E. Abbe, “Recursive projection-aggregation decoding of Reed–Muller codes,” *IEEE Transactions on Information Theory*, vol. 66, no. 8, pp. 4948–4965, Aug 2020.
- [24] M. Lian, C. Hager, and H. D. Pfister, “Decoding Reed–Muller codes using redundant code constraints,” in *IEEE International Symposium on Information Theory (ISIT)*, Jun 2020.
- [25] Q. Huang and B. Zhang, “Pruned collapsed projection-aggregation decoding of Reed–Muller codes,” *arXiv:2105.11878*, May 2021.
- [26] J. Li and W. J. Gross, “Optimization and simplification of PCPA decoder for Reed–Muller codes,” *IEEE Communications Letters*, vol. 26, no. 6, pp. 1206–1210, Jun 2022.
- [27] D. Fathollahi, N. Farsad, S. A. Hashemi, and M. Mondelli, “Sparse multi-decoder recursive projection aggregation for Reed–Muller codes,” in *IEEE International Symposium on Information Theory*, Jul 2021.
- [28] J. Li, S. M. Abbas, T. Tonnellier, and W. J. Gross, “Reduced complexity RPA decoder for Reed–Muller codes,” in *IEEE International Symposium on Topics in Coding*, Sep 2021.
- [29] M. Hashemipour-Nazari, K. Goossens, and A. Balatsoukas-Stimming, “Hardware implementation of iterative projection-aggregation decoding of Reed–Muller codes,” in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Jun 2021.
- [30] I. Tal and A. Vardy, “List decoding of polar codes,” *IEEE Transactions on Information Theory*, vol. 61, no. 5, pp. 2213–2226, 2015.
- [31] A. Balatsoukas-Stimming, M. Bastani Parizi, and A. Burg, “LLR-based successive cancellation list decoding of polar codes,” *IEEE Transactions on Signal Processing*, vol. 63, no. 19, pp. 5165–5179, Oct 2015.
- [32] Y. Ren, A. T. Kristensen, Y. Shen, A. Balatsoukas-Stimming, C. Zhang, and A. Burg, “A sequence repetition node-based successive cancellation list decoder for 5G polar codes: Algorithm



- and implementation,” *IEEE Transactions on Signal Processing*, vol. 70, pp. 5592–5607, 2022.
- [33] F. MacWilliams and N. Sloane, *The Theory of Error-Correcting Codes*. Amsterdam, The Netherlands: Elsevier, 1977.
  - [34] E. Abbe, A. Shpilka, and M. Ye, “Reed–Muller codes: Theory and algorithms,” *IEEE Transactions on Information Theory*, vol. 67, no. 6, pp. 3251–3277, Jun 2021.
  - [35] R. Green, “A serial orthogonal decoder,” *Jet Propulsion Laboratory (JPL) Space Programs Summary*, vol. 37, pp. 247–253, 1966.
  - [36] Y. Be’ery and J. Snyders, “Optimal soft decision block decoders based on fast Hadamard transform,” *IEEE Transactions on Information Theory*, vol. 32, no. 3, pp. 355–364, May 1986.
  - [37] M. Fossorier, M. Mihaljevic, and H. Imai, “Reduced complexity iterative decoding of low-density parity check codes based on belief propagation,” *IEEE Transactions on Communications*, vol. 47, no. 5, pp. 673–680, May 1999.
  - [38] H. Zheng *et al.*, “Threshold-based fast successive-cancellation decoding of polar codes,” *IEEE Transactions on Communications*, vol. 69, no. 6, pp. 3541–3555, 2021.
  - [39] MathWorks. (2021) 5G new radio polar coding. Accessed: 2022-09-30. [Online]. Available: <https://nl.mathworks.com/help/5g/gs/polar-coding.html>
  - [40] M. Hashemipour-Nazari, K. Goossens, and A. Balatsoukas-Stimming, “Multi-factor pruning for recursive projection-aggregation decoding of RM codes,” in *IEEE Workshop on Signal Processing Systems (SiPS)*, Nov 2022.