High Throughput FPGA Implementation for regular Non-Surjective Finite Alphabet Iterative Decoders

Thien Truong Nguyen-Ly, Valentin Savin, Xavier Popon, David Declercq

High Throughput FPGA Implementation for Regular Non-Surjective Finite Alphabet Iterative Decoders

Thien Truong Nguyen-Ly*†, Valentin Savin*, Xavier Popon* and David Declercq†

*CEA-LETI, MINATEC Campus, Grenoble, France, {thientruong.nguyen-ly, valentin.savin, xavier.popon}@cea.fr
†ETIS, ENSEA / CNRS UMR-8051 / University of Cergy-Pontoise, France, declercq@ensea.fr

***Abstract*—This paper deals with the recently introduced class of Non-Surjective Finite Alphabet Iterative Decoders (NS-FAIDs). First, optimization results for an extended class of regular NS-FAIDs are presented. They reveal different possible trade-offs between decoding performance and hardware implementation efficiency. To validate the promises of optimized NS-FAIDs in terms of hardware implementation benefits, we propose two high-throughput hardware architectures, integrating NS-FAIDs decoding kernels. Implementation results show that NS-FAIDs allow significant improvements in terms of both throughput and hardware resources consumption, as compared to a baseline Min-Sum decoder, with even better or only slightly degraded decoding performance.**

I. Introduction

The increasing demand of massive data rates in wireless communication systems will require significantly higher processing speed of the baseband signal, as compared to conventional solutions. This is especially challenging for Forward Error Correction (FEC) mechanisms, since FEC decoding is one of the most computationally intensive baseband processing tasks, consuming a large amount of hardware resources and energy. The use of very large bandwidths will also result in stringent, application-specific, requirements in terms of both throughput and latency. The conventional approach to increase throughput is to use massively parallel architectures. In this context, Low-Density Parity-Check (LDPC) codes are recognized as the foremost solution, due to the intrinsic capacity of their decoders to accommodate various degrees of parallelism. They have found extensive applications in modern communication systems, due to their excellent decoding performance, high throughput capabilities [1]–[4], and power efficiency [5], [6], and have been adopted in several recent communication standards.

In a recent work [7], we have introduced a new theoretical framework, referred to as Non-Surjective Finite Alphabet Iterative Decoders (NS-FAIDs), aimed at exploring the use of imprecise message storage mechanisms in Min-Sum (MS)-based LDPC decoders. It allows storing the exchanged messages using a lower precision than the one used by the processing units, thus facilitating significant reductions of the memory and interconnect blocks, with no or only slight degradation of the error correction performance, as compared to the baseline MS decoder. Since these blocks usually dominate the overall performance of the hardware implementation, NS-FAIDs emerge as a promising approach to further optimizations of cost-effective, high-throughput designs.

To validate the promises of the NS-FAID approach, in this paper we propose two different hardware architectures for regular Quasi-Cyclic (QC)-LDPC decoders, with both MS and NS-FAID decoding kernels. The proposed architectures target high-throughput and efficient use of the hardware resources. Both architectures implement layered scheduled decoding with fully parallel processing units. The first architecture is pipelined, so as to increase throughput and ensure an efficient use of the hardware resources, which in turn imposes specific constraints on the decoding layers[1], in order to ensure proper execution of the layered decoding process. The second architecture does not make use of pipelining, but allows maximum parallelism to be exploited through the use of *full decoding layers*[2], thus resulting in significant increase in throughput. Both MS and NS-FAID decoding kernels are integrated to each of the two proposed architectures, and compared in terms of throughput and resource consumption. Implementation (post place and route) results on Xilinx Zynq-7000 FPGA device are provided, showing a Hardware Utilization Efficiency (HUE) increase by up to 57.78%, when the NS-FAID kernel is used.

The rest of the paper is organized as follows. In Section II, a brief description of NS-FAIDs is first provided, and then new results on the optimization of regular NS-FAIDs are presented. The proposed hardware architectures are presented in Section III. Implementation results are provided in Section IV, and Section V concludes the paper.

II. Non-Surjective Finite Alphabet Iterative Decoders

A. Preliminaries

This section briefly presents the definition of NS-FAIDs; for more details, we refer to [7]. We consider an LDPC code defined by a bipartite graph with N variable-nodes (VNs) and M check-nodes (CNs). The *quantized* Log-Likelihood Ratios (LLRs) supplied to the decoder are denoted by γ_n, while VN and CN messages are denoted respectively by $\alpha_{m,n}$ and $\beta_{m,n}$. All of them are assumed to belong to a finite alphabet denoted by $\mathcal{Q} = \{-Q, \ldots, -1, 0, +1, \ldots, +Q\}$, where $Q = 2^{q-1} - 1$

[1]A decoding layer may consist of one or several rows of the base matrix of the QC-LDPC code, assuming that they do not overlap.

[2]A decoding layer is said to be full if each column of the base matrix has one non-negative entry is one of the rows composing the layer.

with $q > 0$. The A Posteriori (AP-) LLRs are denoted by $\tilde{\gamma}_n$ and, as usual, they are assumed to belong to an extended alphabet $\tilde{\mathcal{Q}} \supset \mathcal{Q}$.

NS-FAIDs are based on MS decoding update rules, but further perform a *framing operation* on VN-messages, as explained below. Thus, CN-messages are updated by using the same update rule as for MS decoding. Precisely, for a CN m, whose set of neighbor VNs is denoted by $\mathcal{N}(m)$, the outgoing $\beta_{m,n}$ messages (with $n \in \mathcal{N}(m)$) are given by:

$$\beta_{m,n} = \left(\prod_{n' \in \mathcal{N}(m) \setminus n} \text{sgn}(\alpha_{m,n'}) \right) \min_{n' \in \mathcal{N}(m) \setminus n} |\alpha_{m,n'}| \quad (1)$$

For a VN n, whose set of neighbor CNs is denoted by $\mathcal{M}(n)$, the outgoing $\alpha_{m,n}$ (with $m \in \mathcal{M}(n)$) messages are given by:

$$\alpha_{m,n} = F\left(\left[\gamma_n + \sum_{m' \in \mathcal{M}(n) \setminus m} \beta_{m',n} \right]_{\mathcal{Q}} \right) \quad (2)$$

where:

- the sum $\gamma_n + \sum_{m' \in \mathcal{M}(n) \setminus m} \beta_{m',n}$ is assumed to be saturated to $\mathcal{Q}$, prior to applying F on it, which is indicated by the notation $[\,]_{\mathcal{Q}}$.
- $F : \mathcal{Q} \to \mathcal{Q}$ is a non-surjective function (*i.e.* the image set of F is a strict subset of $\mathcal{Q}$), verifying:
  (i) F is an odd function, *i.e.*, $F(-x) = -F(x), \forall x \in \mathbb{Z}$
  (ii) F is a non-decreasing function, *i.e.*, $F(x) \leq F(y)$ for any $x < y$.

We note that the above properties also imply that $F(0) = 0$ and $F(x) \geq 0, \forall x > 0$. In this paper, we further extend the definition of NS-FAIDs by allowing $F(0)$ to take on non-zero values. To ensure symmetry of the decoder, we shall write $F(0) = \pm\lambda$, with $\lambda \geq 0$, meaning that $F(0)$ takes on either $-\lambda$ or $+\lambda$ with equal probability. In the following, using the terminology from [7], F will be referred to as *framing function*. Note that F is completely determined by the vector $[|F(0)|, F(1), ..., F(Q)]$, which satisfies the following inequalities:

$$0 \leq |F(0)| \leq F(1) \leq \cdots \leq F(Q) \leq Q \quad (3)$$

The weight of F, denoted by W, is the number of distinct entries in the vector $[|F(0)|, F(1), ..., F(Q)]$. It follows that $1 \leq W < Q + 1$. By a slight abuse of terminology, we shall also refer to W as the weight of the NS-FAID. As shown in [7], for a NS-FAID of weight W, the exchanged messages can be represented by using only $w = \lceil \log_2(W) \rceil + 1$ bits (including 1 bit for the sign). The w value is referred to as the *framing bit-length*. As a consequence of the message size reduction, the size of the memory and the interconnect network that carries the messages from the memory to the processing units are also reduced.

Table I provides two examples of $q = 4$-bit NS-FAIDs (hence $Q = 7$), both of which are of weight $W = 4$. Note that F_1 maps 0 to 0, while F_2 maps 0 to ± 1. The image sets of F_1 and F_2 are $\text{Im}(F_1) = \{0, \pm 1, \pm 3, \pm 7\}$ and $\text{Im}(F_2) = \{\pm 1, \pm 3, \pm 5, \pm 6\}$.

Table I
EXAMPLES OF 4-BIT FRAMING FUNCTIONS OF WEIGHT $W = 4$

m	0	1	2	3	4	5	6	7
$F_1(m)$	0	1	1	3	3	7	7	7
$F_2(m)$	± 1	1	3	3	5	5	5	6

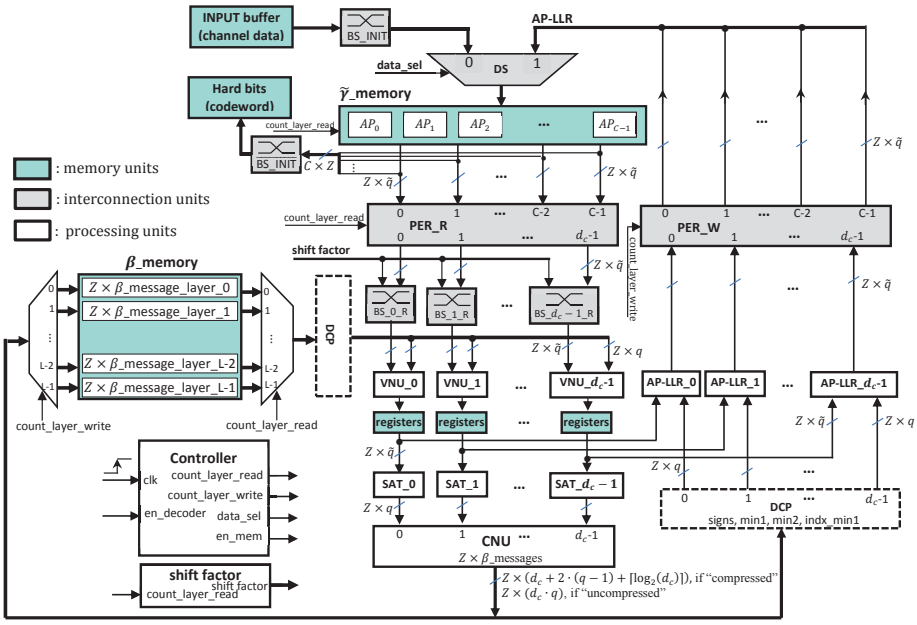
Table II
BEST NS-FAIDS FOR $(3, 6)$-REGULAR LDPC CODES

		F	SNR-thres (dB)
$w = 4$	MS	$[0, 1, 2, 3, 4, 5, 6, 7]$	1.643 ($\mu = 5.6$)
$w = 3$	$F(0) = 0$	$[\mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{3}, \mathbf{3}, \mathbf{3}, \mathbf{7}, \mathbf{7}]$	$\mathbf{1.409}$ ($\mu = \mathbf{3.8}$)
	$F(0) = \pm 1$	$[\pm 1, 1, 1, 3, 3, 4, 4, 7]$	1.412 ($\mu = 5.1$)
	$F(0) = \pm 2$	$[\pm 2, 2, 2, 3, 3, 3, 4, 7]$	1.712 ($\mu = 7.1$)
	$F(0) = \pm 3$	$[\pm 3, 3, 3, 3, 3, 4, 5, 7]$	2.227 ($\mu = 10.0$)
$w = 2$	$F(0) = 0$	$[0, 0, 0, 0, 0, 6, 6, 6]$	2.251 ($\mu = 8.6$)
	$F(0) = \pm 1$	$[\pm\mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{6}, \mathbf{6}, \mathbf{6}]$	$\mathbf{1.834}$ ($\mu = \mathbf{6.4}$)
	$F(0) = \pm 2$	$[\pm 2, 2, 2, 2, 2, 2, 2, 7]$	1.911 ($\mu = 8.3$)
	$F(0) = \pm 3$	$[\pm 3, 3, 3, 3, 3, 3, 3, 7]$	2.014 ($\mu = 9.4$)

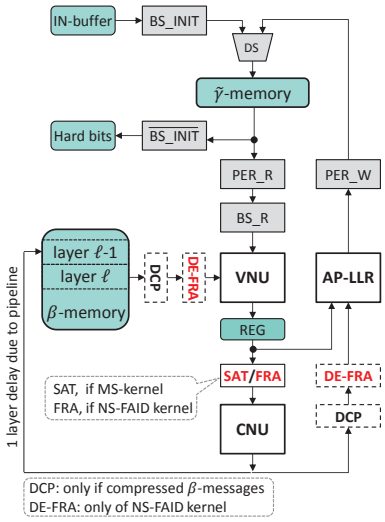
B. Optimization of Regular NS-FAIDs

In this section, we consider the optimization of regular NS-FAIDs for $(d_v = 3, d_c = 6)$-regular LDPC codes. To illustrate the trade-off between hardware complexity and decoding performance, we consider $q = 4$-bit NS-FAIDs (hence, $Q = 7$), with framing bit-length parameter $w \in \{2, 3\}$. It can be easily verified that the total number of regular NS-FAIDs is given by $N_{\text{NS-FAID}}(w = 3) = 2450$ and $N_{\text{NS-FAID}}(w = 2) = 196$. All regular NS-FAIDs have been evaluated by using the density evolution (DE) technique. As explained in [7], the DE threshold computation also encompasses the optimization of the *gain factor* μ, used for input LLR quantization: quantized LLR is obtained by first scaling the soft LLR value by μ, and then rounding the scaled value to the closest integer in $\mathcal{Q}$. Such a quantization method, with μ optimized by DE, has been shown to provide optimal performance in [8].

Table II summarizes the best NS-FAIDs according to w and $F(0)$ values (for $0 \leq |F(0)| \leq 3$); DE thresholds and corresponding gain factors (μ) are also reported. Best NS-FAIDs for $w = 2$ and $w = 3$ are emphasized in bold. For comparison purposes, the DE threshold of the baseline $q = 4$-bit MS decoder is also reported: MS threshold is equal to 1.643 dB, for $\mu = 5.6$. For $w = 3$, it can be observed that best NS-FAIDs with $F(0) = 0$ or $F(0) = \pm 1$ have better DE thresholds than the MS decoder. The best NS-FAID is given by the framing function $F = [0, 1, 1, 3, 3, 3, 7, 7]$ and its DE threshold is equal to 1.409 dB ($\mu = 3.8$), representing a gain of 0.23 dB compared to MS. For $w = 2$, the best NS-FAID is given by the framing function $F = [\pm 1, 1, 1, 1, 1, 6, 6, 6]$ and its DE threshold is equal to 1.834 dB ($\mu = 6.4$), which represents a performance loss of only 0.19 dB compared to MS. To emphasize the benefits of the proposed NS-FAIDs extension, we note that that for $w = 2$, best NS-FAIDs with $F(0) = \pm 1$, $F(0) = \pm 2$ or $F(0) = \pm 3$ have better DE thresholds than the best NS-FAIDs with $F(0) = 0$. The latter is given by the framing function $F = [0, 0, 0, 0, 0, 6, 6, 6]$ and its DE threshold is equal to 2.251 dB ($\mu = 8.6$), thus resulting in a performance loss of 0.61 dB compared to MS.



(a) Detailed HW Architecture with MS-kernel



(b) High-Level Description of the HW Architecture with MS and NS-FAID kernels

Figure 1. Block Diagram of the Proposed Pipelined Architecture

III. Hardware Architectures

We consider a QC-LDPC code dened by a base matrix B of size $R \times C$, and expansion factor z, corresponding to a parity check matrix H of size $M \times N$, with $M = zR$ and $N = zC$. A *decoding layer* consists of one or several consecutive rows of B, assuming that they do not overlap (*i.e.* each column has at most one non-negative entry within each layer). It is assumed that the same number of rows of B participate in each decoding layer, which is denoted by RPL (rows per layer). Hence, the number of decoding layers is given by $L = R/\text{RPL}$. We further define $Z = z \times \text{RPL}$, corresponding to the number of parity checks (rows of H) within one decoding layer, and referred to as the parallelism degree (of the hardware architecture).

Both hardware architectures proposed in this section assume that all CNs have the same degree, denoted by d_c (although they can be easily modified to accommodate irregular CN degrees). No assumptions are made concerning VN degrees. We present each architecture assuming the MS decoding kernel is being implemented, then we discuss the required changes in order to integrate the NS-FAID decoding kernel.

A. Pipelined architecture

Proposed architecture with MS decoding kernel is detailed in Figure 1(a). A high-level representation is also shown in Figure 1(b), for both MS and NS-FAID decoding kernels. For the sake of simplicity, in Figure 1(a) we assume that a decoding layer corresponds to a row of the base matrix (hence $\text{RPL} = 1$ and $Z = z$).

Input/Output buffers. The input buffer, implemented as a number of Serial Input Parallel Output (SIPO) shift registers, is used to store the input LLR values (γ_n) received by the decoder. The output buffer, is used to store the hard bit estimates of the decoded word. Input/output buffers allow data load/offload operations to take place during the decoding of the previous/following codeword.

Memory blocks. Two memory blocks are used, one for AP-LLR values ($\tilde{\gamma}$_memory) and one for CN-messages (β_memory). $\tilde{\gamma}_n$ values are quantized on $\tilde{q}$ bits, while $\beta_{m,n}$ messages are quantized on q bits, with $\tilde{q} > q$. $\tilde{\gamma}$_memory is implemented by registers, in order to allow massively parallel read or write operations. It is organized in C blocks, denoted by $AP_i (i = 0, \cdots, C-1)$ corresponding to the columns of base matrix, each one consisting of $z \times \tilde{q}$ bits. Data are read from/write to blocks corresponding to non-negative entries in the decoding layer being processed. β_memory is implemented as a dual port Random Access Memory (RAM), in order to support pipelining, as explained below. Each memory word consists of $Z \times \beta$_messages, corresponding to one decoding layer. Depending on the Check Node Unit (CNU) implementation, β_messages can be either "uncompressed" (*i.e.*, for a check-node m, the corresponding β_message is given by the d_c values $[\beta_{m,n_1}, \ldots, \beta_{m,n_{d_c}}]$, where $n_1, \ldots, n_{d_c}$ denote the variable nodes connected to m) or "compressed" (*i.e.*, for a check-node m, the corresponding β_message is given by the signs of the above β_{m,n_i} messages, their first and second minimum, denoted by min1 and min2, and the index of the first minimum, denoted by indx_min1) [9].

Read and Write Permutations (PER_R, PER_W). PER_R permutation is used to rearrange the data read from $\tilde{\gamma}$_memory, according to the processed layer, so as to ensure processing by the proper VNU/CNU. PER_W block operates

oppositely to PER_R.

Barrel Shifters (BS_INIT, BS_R). Barrel shifters are used to implement the cyclic (shift) permutations, according to the non-negative entries of the base matrix. The $\tilde{\gamma}$_memory is initialized from the input LLR values stored in the input buffer. However, input LLR values are shifted by BS_INIT block before being written to the $\tilde{\gamma}$_memory, according to the *last* non-negative shift factor on the corresponding base matrix column. BS_R blocks are then used to shift the LLR values read from the $\tilde{\gamma}$_memory, such that to properly align them with the right VNU. Note that there are d_c BS_R blocks, corresponding to the d_c columns with non-negative entries in the current layer ℓ. Hence, the cyclic shift implemented by each BS_R block is given by $-b_{\ell'} + b_\ell$, where $b_{\ell'}$ and b_ℓ denote respectively the *previous* and the *current* non-negative entries of the corresponding column (*previous* means previous layer with non-negative entry on the corresponding column). These values are computed offline for each layer ℓ. This avoids the use of barrel shifters when the data is written back to the $\tilde{\gamma}$_memory, thus reducing the critical path of the design. Finally, the $\overline{\text{BS\_INIT}}$ block operates oppositely to BS_INIT, and is used to shift back the hard decision bits into appropriate positions.

Variable Node Units (VNUs) and AP-LLR Units. These units compute VN-messages ($\alpha_{m,n}$) and AP-LLR values ($\tilde{\gamma}_n$).

Saturators (SATs). Prior to CNU processing, $\alpha_{m,n}$ values are saturated to q bits.

Check Node Units (CNUs). These processing units compute the CN-messages ($\beta_{m,n}$). For simplicity, Figure 1(a) shows one CNU block with d_c inputs, each one of size $Z \times q$ bits. Thus, this block actually includes Z computing units, used to process in parallel the Z check-nodes within one layer. The CNU is implemented by using either: (i) the high-speed low-cost tree-structure (TS) approach proposed in [10] for "compressed" CN-messages, or (ii) comparator trees for "uncompressed" CN-messages.

Decompress (DCP). This block is only used in case that the CN-messages are in compressed format (signs, min1, min2, indx_min1). It converts the β_messages from compressed to the uncompressed format.

Pipelining. To increase the operating frequency, the data path is pipelined by adding a set of registers after the VNU-blocks. Hence, processing one layer takes 2 clock cycles, but at each clock cycle the two pipeline stages work on two consecutive layers of the base matrix. This imposes specific constraints on the base matrix, as consecutive layers must not overlap, in order to avoid $\tilde{\gamma}$_memory conflicts (note that memory stall cycles would cancel the pipelining effect). An example of $d_c = 6$ regular base matrix with this property is given in Figure 2 (assuming that each decoding layer corresponds to a row of the base matrix).

NS-FAID decoding kernel. The changes required to integrate the NS-FAID decoding kernel are shown in Figure 1(b). First, the Saturation (SAT) block used within the MS-decoding kernel is replaced by a Framing (FRA) block. Note that the output of the VNU consists of $\tilde{q}$-bit (unsaturated) VN-messages. Hence, the FRA block actually implements the concatenation of the following operations:

$$[-\tilde{Q}, \ldots, \tilde{Q}] \xrightarrow{\text{sat}} [-Q, \ldots, Q] \xrightarrow{F} \text{Im}(F) \xrightarrow{\sim} [-W, \ldots, W],$$

where $[-\tilde{Q}, \ldots, \tilde{Q}]$ is the alphabet of unsaturated messages ($\tilde{Q} = 2^{\tilde{q}-1}-1$), F is the framing function being used, $\text{Im}(F)$ is the image of F (which is a subset of $[-Q, \ldots, Q]$ according to the framing function definition), and the last operation consists of a re-quantization of the $\text{Im}(F)$ values on a number of w-bits, where $w = \lceil \log_2(W) \rceil + 1$ is the framing bit-length. The De-framing (DE-FRA) block simply converts back from w-bit to q-bit values ($[-W, \ldots, W] \xrightarrow{\sim} \text{Im}(F) \subset [-Q, \ldots, Q]$), *i.e.* it inverts the re-quantization operation above. Although we have to add the de-framing blocks, the reduction of the CN-messages size may still save significant hardware resources, as compared to MS decoding. This will be discussed in more details in Section IV.

49	-1	-1	-1	-1	43	-1	-1	-1	-1	50	-1	-1	-1	-1	2	-1	27	-1	-1	-1	-1	49
-1	-1	-1	10	41	-1	-1	-1	-1	52	-1	-1	32	-1	-1	-1	-1	-1	50	-1	50	-1	-1	-1
-1	-1	20	-1	-1	-1	20	-1	-1	-1	51	-1	10	-1	-1	47	-1	-1	-1	-1	-1	33	-1
-1	24	-1	-1	-1	-1	22	-1	53	-1	-1	-1	-1	-1	31	-1	-1	-1	-1	18	-1	47	-1	-1
10	-1	-1	-1	15	-1	-1	-1	-1	-1	2	-1	-1	-1	-1	50	-1	13	-1	-1	-1	-1	-1	53
-1	-1	44	-1	-1	6	-1	-1	-1	-1	29	-1	40	-1	-1	16	-1	-1	-1	-1	13	-1	-1	-1
-1	2	-1	-1	-1	-1	-1	13	41	-1	-1	-1	-1	-1	42	-1	-1	-1	-1	48	-1	49	-1	-1
-1	-1	-1	36	-1	-1	24	-1	-1	50	-1	-1	12	-1	-1	-1	-1	-1	-1	10	-1	-1	-1	48	-1
-1	-1	47	-1	50	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	9	-1	7	-1	-1	-1	-1	-1	28
6	-1	-1	-1	-1	-1	5	-1	-1	-1	-1	13	-1	3	-1	-1	29	-1	-1	-1	16	-1	-1	-1
-1	-1	-1	35	-1	16	-1	-1	37	-1	-1	-1	4	-1	-1	-1	-1	-1	24	-1	-1	-1	29	-1
-1	24	-1	-1	-1	-1	-1	51	-1	38	-1	-1	-1	-1	6	-1	-1	-1	-1	23	-1	16	-1	-1

Figure 2. Base matrix of the $(3, 6)$-regular QC-LDPC code

B. Full layers architecture

A different possibility to increase throughput is to increase the hardware parallelism, by including several non-overlapping rows of the base matrix in one decoding layer. For instance, for the base matrix in Figure 2, we may consider $\text{RPL} = 4$ consecutive rows per decoding layer, thus the number of decoding layers is $L = 3$. In this case, each column of the base matrix has one (and only one) non-zero entry in each decoding layer; such a decoding layer is referred to as being *full*. Full layers correspond to the maximum hardware parallelism that can be exploited by layered architectures, but they also prevent the pipelining of the data path. The architecture proposed in this section, shown in Figure 3, is aimed at providing an effective way to benefit from the increased hardware parallelism enabled by the use of full layers. We discuss below the main changes with respect to the pipelined architecture from the previous section, consisting of the α_memory and the barrel shifters blocks (the other blocks are the same as for the pipelined architecture), as well as a complete reorganization of the data path.

α_memory. This memory is used to store the VN-messages for the current decoding layer (unlike the previous architecture, the AP-LLR values are not stored in memory). Since only one $\tilde{q}$-bit (unsaturated) VN-message is stored for each variable-node, this memory has exactly the same size as the $\tilde{\gamma}$_memory used within the previous pipelined architecture. VN-messages for current layer ℓ are read from the α_memory, then saturated or framed depending on the decoding kernel, and supplied

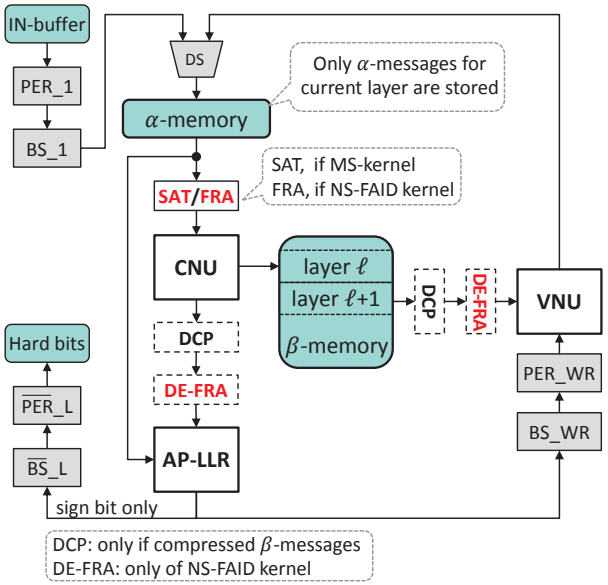


Figure 3. Proposed full layer architecture with MS and NS-FAID kernels

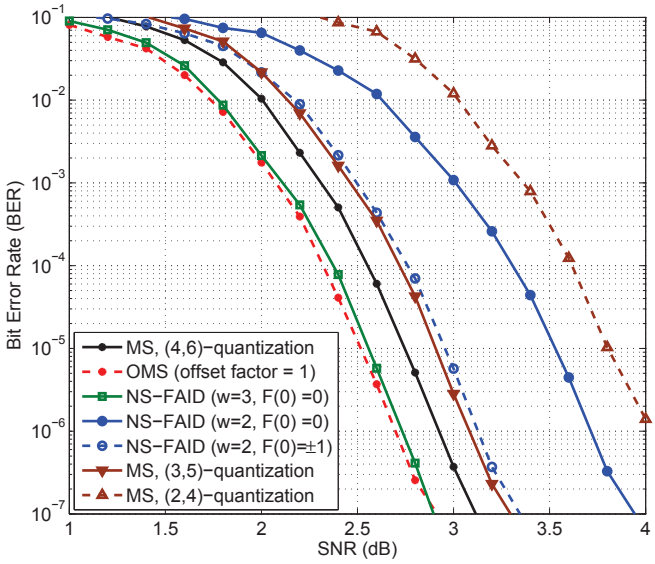


Figure 4. BER performance of optimized regular NS-FAIDs

to the corresponding CNUs. CN-messages computed by the CNUs are stored in the β_memory (location corresponding to layer ℓ), and also forwarded to the AP-LLR unit, through the DCP (decompress) and DE-FRA (de-framing) blocks, according to the CNU implementation (compressed or uncompressed) and the decoding kernel (MS or NS-FAID). The AP-LLR unit computes the sum of the incoming VN- and CN-messages, which corresponds to the AP-LLR value to be used at layer $\ell+1$ (since already updated by layer ℓ). The AP-LLR value is forwarded to the VNU, through corresponding BS and PER blocks. Eventually, the VN-message for layer $\ell+1$ is computed as the difference between the incoming AP-LLR and the corresponding layer-$(\ell+1)$ CN-message computed at the previous iteration, the latter being read from the β_memory. While the data path is completely reorganized, it can be easily verified that both architectures are logically equivalent, *i.e.*, they both implement the same decoding algorithm.

PER / BS blocks. PER_1 / BS_1 blocks permute / shift the data read from the input buffer, according to the positions / values of the non-negative entries in the first decoding layer. Similarly, the PER_WR / BS_WR blocks permute / shift the AP-LLR values, according to the *difference* between the positions / values of the current layer's (ℓ) non-negative entries and those of the next layer ($\ell+1$). This way, VN-messages stored in the α_memory are already permuted and shifted for the subsequent decoding layer.

IV. Implementation Results

Throughout this section we consider the $(3,6)$-regular QC-LDPC code, with base matrix B of size $R \times C = 12 \times 24$, shown in Figure 2. The expansion factor $z = 54$, thus the codeword length is $N = zC = 1296$ bits.

Figure 4 shows the Bit-Error Rate (BER) performance of the MS decoder with quantization parameters $(q, \tilde{q}) = (4, 6)$, as well as $q = 4$-bit NS-FAIDs with $w = 2$ and $w = 3$ (framing functions F corresponding to w and $F(0)$ values in the legend are those from Table II). Additive White Gaussian Noise (AWGN) channel model is considered, with 20 decoding iterations. Note that both decoding architectures discussed in the paper have the same decoding performance. It can be seen that the simulation results corroborate the analytic analysis from Section II-B, in terms of SNR gain / loss provided by NS-FAIDs, as compared to MS. For comparison purposes, we have further included simulations results for the Offset MS (OMS) decoder with $(4,6)$-quantization and offset factor = 1, as well as the MS decoder with $(3,5)$ and $(2,4)$-quantization.

Implementation (post place and route) results on Xilinx Zynq-7000 FPGA device are shown in Table III, for the MS$(4,6)$ decoder and the NS-FAIDs with $(w=3, F(0)=0)$ and $(w=2, F(0)=\pm 1)$, indicated in the table as NS-FAID-3 and NS-FAID-2, respectively. The first (Variant) row in Table III indicates the architecture (pipelined or full layers) and the CNU type (compressed or uncompressed). We also note that for the NS-FAID-2, the assumption that 0 is mapped to either -1 or $+1$, with equal probability, is only needed for theoretical analysis (the symmetry of the decoder allows reducing the analysis to the all-zero codeword). However, in practical situations one may always map 0 to $+1$, since random codewords are transmitted (in telecommunications systems, pseudo-randomness of the transmitted data is ensured by a scrambling mechanism).

Throughput reported in Table III is given by the formula:

$$\text{Throughput} = \begin{cases} \frac{N \times f_{\max}}{L \times n_{\text{iter}}}, & \text{full layers architecture} \\ \frac{N \times f_{\max}}{1 + L \times n_{\text{iter}}}, & \text{pipelined architecture} \end{cases} \quad (4)$$

where N is the codeword length, $f_{\max}$ is the maximum operating frequency (post-place and route), L is the number of decoding layers and n_{iter} is the number of decoding iterations (set to 20). To keep the throughput comparison on an equal

Table III
FPGA POST-PAR IMPLEMENTATION RESULTS ON ZYNQ-7000 (XC7Z045FFG900-1) – 2016.2

Variant	**pipeline.uncompressed**			**pipeline.compressed**			**full_layers.uncompressed**			**full_layers.compressed**		
Decoder	MS(4,6)	NS-FAID-3	NS-FAID-2	MS(4,6)	NS-FAID-3	NS-FAID-2	MS(4,6)	NS-FAID-3	NS-FAID-2	MS(4,6)	NS-FAID-3	NS-FAID-2
No. occupied slices	9776	9204	8844	10163	10091	9617	17578	16812	15018	21317	18313	17712
(% utilization)	(17.89)	(16.84)	(16.18)	(18.60)	(18.46)	(17.60)	(32.16)	(30.76)	(27.48)	(39.01)	(33.51)	(32.41)
Max. freq (MHz)	98	108	138	95	102	125	69	75	80	58	71	76
No. layers (L)	12	12	12	12	12	12	3	3	3	3	3	3
Throughput (Mbps)	527	580	742	510	548	672	1490	1620	1728	1252	1533	1641
HUE (Mbps)	2945	3444	4585	2741	2968	3818	4633	5266	6288	3209	4574	5063
$\pm$% w.r.t. MS(4,6)	0%	+16.94%	+55.69%	0%	+8.28%	+39.29%	0%	+13.66%	+35.72%	0%	+42.53%	+57.78%

Table IV
COMPARISON OF FPGA IMPLEMENTATIONS FOR $(3, 6)$-REGULAR LDPC CODES

Decoders	**Karkooti'04 [1]**	**Chen'11 [2]**	**Vikram'15 [3]**	**This work NS-FAID-3**
Device	Virtex 2	Virtex 2	Virtex 5	Zynq-7000
Codeword length	1536	1536	2304	1296
No. occupied slices	11352	6102	8430	16812
No. BRAMs	66	24	232	0
No. iterations	20	3 (avg)	8 (avg)	20
Max. freq (MHz)	121	149.8	114	75
Throughput (Mbps)	127	830.6 (avg)	1096 (avg)	1620

basis, we further define the *Hardware Usage Efficiency* (HUE) metric, as the throughput corresponding to 100% utilization of the hardware resources:

$$\text{HUE} = \frac{\text{Throughput}}{\%\,\text{Number of occupied slices}} \tag{5}$$

While the NS-FAID-3 decoder outperforms the baseline MS$(4, 6)$ decoder by $0.19\,$dB at BER $= 10^{-5}$ (Figure 4), it can be seen from Table III that is also exhibits a HUE improvement between 8.28% and 42.53%, depending on the hardware architecture and CNU type. As predicted, the NS-FAID-2 decoder exhibit a performance loss of $0.21\,$dB compared to MS$(4, 6)$, but yields a significant HUE improvement, by 35.72% to 57.78%.

To further emphasize the high-throughput characteristic of the proposed architectures, Table IV provides a comparison between state of the art FPGA implementations of $(3, 6)$-regular LDPC decoders and the uncompressed full layers architecture with NS-FAID-3 decoding kernel, presented in this work. Our implementation achieves a significantly increased throughput, which has also to be reported to the number of decoding iterations.

V. CONCLUSIONS

This paper first extended the previous definition of NS-FAIDs, and presented DE-based optimization results for regular NS-FAIDs. The proposed extension (allowing framing functions with $F(0) = \pm\lambda$) proved to be particularly useful for NS-FAIDs with framing bit-length $w = 2$. Then, two hardware architectures have been presented, making use of either pipelining or increased hardware parallelism in order to increase throughput. Both hardware architectures have been implemented in FPGA, using MS and NS-FAID decoding kernels. Implementation results revealed that NS-FAIDs allow significant improvements in terms of hardware usage efficiency as compared to the baseline MS decoder, with even better or slightly degraded decoding performance.

ACKNOWLEDGMENT

This work was supported by the Franco-Romanian (ANR-UEFISCDI) Joint Research Programme "Blanc-2013", project DIAMOND, and the European H2020 Work Programme, project Flex5Gware.

REFERENCES

[1] M. Karkooti and J. R. Cavallaro, "Semi-parallel reconfigurable architectures for real-time LDPC decoding," in *Proc. of Int. Conf. on Inf. Technology: Coding and Computing (ITCC)*, vol. 1, 2004, pp. 579–585.

[2] X. Chen, J. Kang, S. Lin, and V. Akella, "Memory system optimization for FPGA-based implementation of quasi-cyclic LDPC codes decoders," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 1, pp. 98–111, 2011.

[3] V. A. Chandrasetty and S. M. Aziz, "Resource efficient LDPC decoders for multimedia communication," *INTEGRATION, the VLSI journal*, vol. 48, pp. 213–220, 2015.

[4] K. Zhang, X. Huang, and Z. Wang, "High-throughput layered decoder implementation for quasi-cyclic LDPC codes," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 6, pp. 985–994, 2009.

[5] X. Peng, Z. Chen, X. Zhao, D. Zhou, and S. Goto, "A 115mW 1Gbps QC-LDPC decoder ASIC for WiMAX in 65nm CMOS," in *IEEE Asian Solid State Circuits Conference (A-SSCC)*, 2011, pp. 317–320.

[6] B. Xiang and X. Zeng, "A 4.84 mm^2 847–955 Mb/s 397 mW dual-path fully-overlapped QC-LDPC decoder for the WiMAX system in 0.13 μm CMOS," in *IEEE Symp. on VLSI Circuits (VLSIC)*, 2010, pp. 211–212.

[7] T. T. Nguyen-Ly, K. Le, V. Savin, D. Declercq, F. Ghaffari, and O. Boncalo, "Non-surjective finite alphabet iterative decoders," in *IEEE International Conference on Communications (ICC)*, 2016, pp. 1–6.

[8] Z. Mheich, T. Nguyen-Ly, V. Savin, and D. Declercq, "Code-aware quantizer design for finite-precision min-sum decoders," in *IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, Varna, Bulgaria, June 2016.

[9] Z. Wang and Z. Cui, "A memory efficient partially parallel decoder architecture for quasi-cyclic LDPC codes," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 4, pp. 483–488, 2007.

[10] C.-L. Wey, M.-D. Shieh, and S.-Y. Lin, "Algorithms of finding the first two minimum values and their hardware implementation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 55, no. 11, pp. 3430–3437, 2008.