An FPGA Implementation of $(3, 6)$-Regular Low-Density Parity-Check Code Decoder

Tong Zhang
ECSE Dept., RPI
Troy, NY
tzhang@ecse.rpi.edu

Keshab K. Parhi
ECE Dept., Univ. of Minnesota
Minneapolis, MN
parhi@ece.umn.edu

Abstract

Because of their excellent error-correcting performance, Low-Density Parity-Check (LDPC) codes have recently attracted a lot of attentions. In this work, we are interested in the practical LDPC code decoder hardware implementations. The direct fully parallel decoder implementation usually incurs too high hardware complexity for many real applications, thus partly parallel decoder design approaches that can achieve appropriate trade-offs between hardware complexity and decoding throughput are highly desirable. Applying a joint code and decoder design methodology, we develop a high-speed $(3, k)$-regular LDPC code partly parallel decoder architecture, based on which we implement a 9216-bit, rate-$1/2$ $(3, 6)$-regular LDPC code decoder on Xilinx FPGA device. This partly parallel decoder supports a maximum symbol throughput of 54 Mbps and achieves BER 10^{-6} at 2dB over AWGN channel while performing maximum 18 decoding iterations.

1 Introduction

In the past few years, the recently rediscovered Low-Density Parity-Check (LDPC) codes [1][2][3] have received a lot of attentions and have been widely considered as next-generation error-correcting codes for telecommunication and magnetic storage. Defined as the null space of a very sparse $M \times N$ parity check matrix $\mathbf{H}$, an LDPC code is typically represented by a bipartite graph[1], usually called Tanner graph, in which one set of N *variable* nodes corresponds to the set of codeword, another set of M *check* nodes corresponds to the set

[1] A bipartite graph is one in which the nodes can be partitioned into two sets, X and Y, so that the only edges of the graph are between the nodes in X and the nodes in Y.

of parity check constraints and each edge corresponds to a non-zero entry in the parity check matrix $\mathbf{H}$. An LDPC code is known as (j,k)-regular LDPC code if each variable node has the degree of j and each check node has the degree of k, or in its parity check matrix each column and each row have j and k non-zero entries, respectively. The code rate of a (j,k)-regular LDPC code is $1-j/k$ provided that the parity check matrix has full rank. The construction of LDPC codes is typically random. LDPC codes can be effectively decoded by the iterative belief-propagation (BP) algorithm [3] that, as illustrated in Fig. 1, directly matches the Tanner graph: decoding messages are iteratively computed on each variable node and check node and exchanged through the edges between the neighboring nodes.

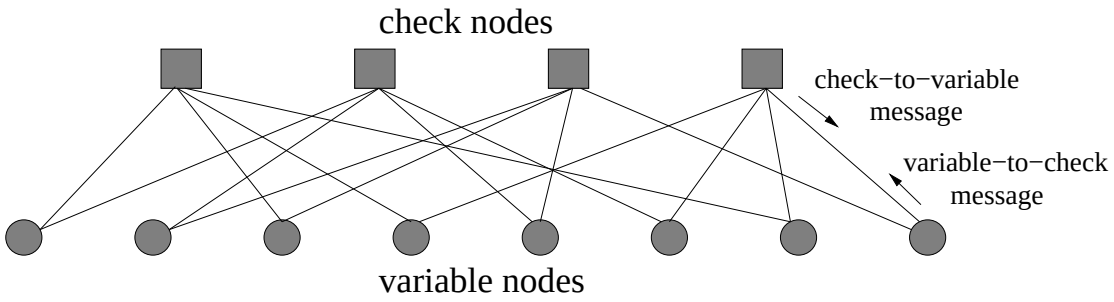

Figure 1: Tanner graph representation of an LDPC code and the decoding messages flow.

Recently, tremendous efforts have been devoted to analyze and improve the LDPC codes error-correcting capability, see [4]-[11], *etc.* Besides their powerful error-correcting capability, another important reason why LDPC codes attract so many attentions is that the iterative BP decoding algorithm is *inherently* fully parallel, thus a great potential decoding speed can be expected.

The high-speed decoder hardware implementation is obviously one of the most crucial issues determining the extent of LDPC applications in the real world. The most natural solution for the decoder architecture design is to directly instantiate the BP decoding algorithm to hardware: each variable node and check node are physically assigned their own processors and all the processors are connected through an interconnection network reflecting the Tanner graph connectivity. By completely exploiting the parallelism of the BP decoding algorithm, such fully parallel decoder can achieve very high decoding speed, *e.g.*, a 1024-bit, rate-$1/2$ LDPC code fully parallel decoder with the maximum symbol throughput of 1 Gbps has been physically implemented using ASIC technology [12]. The main disadvantage of such fully parallel design is that with the increase of code length, typically the LDPC code length is very large (at least several thousands), the incurred hardware complexity will become more and more prohibitive for many practical purposes, *e.g.*, for 1K code length the ASIC decoder implementation [12] consumes 1.7M gates. Moreover, as pointed out in [12], the routing over-

head for implementing the entire interconnection network will become quite formidable due to the large code length and randomness of the Tanner graph. Thus high-speed partly parallel decoder design approaches that achieve appropriate trade-offs between hardware complexity and decoding throughput are highly desirable.

For any given LDPC code, due to the randomness of its Tanner graph, it is nearly impossible to directly develop a high-speed partly parallel decoder architecture. To circumvent this difficulty, Boutillon *et al.* [13] proposed a *decoder-first code design* methodology: instead of trying to conceive the high-speed partly parallel decoder for any given random LDPC code, use an available high-speed partly parallel decoder to define a constrained random LDPC code. We may consider it as an application of the well-known "Think in the reverse direction" methodology. Inspired by the decoder-first code design methodology, we proposed a *joint* code and decoder design methodology in [14] for $(3, k)$-regular LDPC code partly parallel decoder design. By jointly conceiving the code construction and partly parallel decoder architecture design, we presented a $(3, k)$-regular LDPC code partly parallel decoder structure in [14], which not only defines very good $(3, k)$-regular LDPC codes but also could potentially achieve high-speed partly parallel decoding.

In this paper, applying the joint code and decoder design methodology, we develop an elaborate $(3, k)$-regular LDPC code high-speed partly parallel decoder architecture based on which we implement a 9216-bit, rate-$1/2$ $(3, 6)$-regular LDPC code decoder using Xilinx Virtex FPGA (Field Programmable Gate Array) device. In this work, we significantly modify the original decoder structure [14] to improve the decoding throughput and simplify the control logic design. To achieve good error-correcting capability, the LDPC code decoder architecture has to possess randomness to some extent, which makes the FPGA implementations more challenging since FPGA has fixed and regular hardware resources. We propose a novel scheme to realize the random connectivity by concatenating two routing networks, where all the random hardwire routings are localized and the overall routing complexity is significantly reduced. Exploiting the good minimum distance property of LDPC codes, this decoder employs parity check as the earlier decoding stopping criterion to achieve adaptive decoding for energy reduction. With the maximum 18 decoding iterations, this FPGA partly parallel decoder supports a maximum 54 Mbps symbol throughput and achieves BER 10^{-6} at 2dB over AWGN channel.

This paper begins with a brief description of the LDPC code decoding algorithm in Section 2. In section 3, we briefly describe the joint code and decoder design methodology for $(3, k)$-regular LDPC code partly parallel decoder design. In Section 4, we present the detailed high-speed partly parallel decoder architecture design. Finally, an FPGA implementation of a $(3, 6)$-regular LDPC code partly parallel decoder is discussed in Section 5.

2 Decoding Algorithm

Since the direct implementation of BP algorithm will incur too high hardware complexity due to the large number of multiplications, we introduce some logarithmic quantities to convert these complicated multiplications into additions, which leads to the Log-BP algorithm [2][15].

Before the description of Log-BP decoding algorithm, we introduce some definitions as follows: Let $\mathbf{H}$ denote the $M \times N$ sparse parity check matrix of the LDPC code and $H_{i,j}$ denote the entry of $\mathbf{H}$ at the position (i, j). We define the set of bits n that participate in parity check m as $\mathcal{N}(m) = \{n : H_{m,n} = 1\}$, and the set of parity checks m in which bit n participates as $\mathcal{M}(n) = \{m : H_{m,n} = 1\}$. We denote the set $\mathcal{N}(m)$ with bit n excluded by $\mathcal{N}(m) \setminus n$, and the set $\mathcal{M}(n)$ with parity check m excluded by $\mathcal{M}(n) \setminus m$.

Algorithm 2.1 *Iterative Log-BP Decoding Algorithm*

Input: The prior probabilities $p_n^0 = P(x_n = 0)$ *and* $p_n^1 = P(x_n = 1) = 1 - p_n^0$, $n = 1, \cdots, N$;

Output: Hard decision $\hat{\mathbf{x}} = \{\hat{x}_1, \cdots, \hat{x}_N\}$;

Procedure:

1. *Initialization: For each* n, *compute the intrinsic (or channel) message* $\gamma_n = \log \frac{p_n^0}{p_n^1}$ *and for each* $(m, n) \in \{(i, j)|H_{i,j} = 1\}$, *compute*

$$\alpha_{m,n} = sign(\gamma_n) \log\Big(\frac{1 + e^{-|\gamma_n|}}{1 - e^{-|\gamma_n|}}\Big),\ where\ \ sign(\gamma_n) = \begin{cases} +1 & \gamma_n \geq 0 \\ -1 & \gamma_n < 0 \end{cases}.$$

2. *Iterative Decoding*

   - *Horizontal (or check node computation) step: For each* $(m, n) \in \{(i, j)|H_{i,j} = 1\}$, *compute*

$$\beta_{m,n} = \log\Big(\frac{1 + e^{-\alpha}}{1 - e^{-\alpha}}\Big) \prod_{n' \in \mathcal{N}(m) \setminus n} sign(\alpha_{m,n'}), \tag{1}$$

     where $\alpha = \sum_{n' \in \mathcal{N}(m) \setminus n} |\alpha_{m,n'}|$.

   - *Vertical (or variable node computation) step: For each* $(m, n) \in \{(i, j)|H_{i,j} = 1\}$, *compute*

$$\alpha_{m,n} = sign(\gamma_{m,n}) \log\Big(\frac{1 + e^{-|\gamma_{m,n}|}}{1 - e^{-|\gamma_{m,n}|}}\Big), \tag{2}$$

     where $\gamma_{m,n} = \gamma_n + \sum_{m' \in \mathcal{M}(n) \setminus m} \beta_{m',n}$. *For each* n, *update the "pseudo-posterior log-likelihood ratio (LLR)"* λ_n *as:*

$$\lambda_n = \gamma_n + \sum_{m \in \mathcal{M}(n)} \beta_{m,n}. \tag{3}$$

- *Decision step:*

  (a) Perform hard decision on $\{\lambda_1, \cdots, \lambda_N\}$ *to obtain* $\hat{\mathbf{x}} = \{\hat{x}_1, \cdots, \hat{x}_N\}$ *such that* $\hat{x}_n = 0$ *if* $\lambda_n > 0$ *and* $\hat{x}_n = 1$ *if* $\lambda \leq 0$*;*

  (b) If $\mathbf{H} \cdot \hat{\mathbf{x}} = 0$*, then algorithm terminates, else go to Horizontal step until the pre-set maximum number of iterations have occurred.* ■

We call $\alpha_{m,n}$ and $\beta_{m,n}$ in the above algorithm as *extrinsic* messages, where $\alpha_{m,n}$ is delivered from variable node to check node and $\beta_{m,n}$ is delivered from check node to variable node.

It is clear that each decoding iteration can be performed in fully parallel by physically mapping each check node to one individual *Check Node processing Unit* (CNU) and each variable node to one individual *Variable Node processing Unit* (VNU). Moreover, by delivering the hard decision $\hat{x}_i$ from each VNU to its neighboring CNUs, the parity check $\mathbf{H} \cdot \hat{\mathbf{x}}$ can be easily performed by all the CNUs. Thanks to the good minimum distance property of LDPC code, such adaptive decoding scheme can effectively reduce the average energy consumption of the decoder without performance degradation.

In the partly parallel decoding, the operations of a certain number of check nodes or variable nodes are time-multiplexed, or folded [16], to a single CNU or VNU. For an LDPC code with M check nodes and N variable nodes, if its partly parallel decoder contains M_p CNUs and N_p VNUs, we denote $\frac{M}{M_p}$ as CNU folding factor and $\frac{N}{N_p}$ as VNU folding factor.

3 Joint Code and Decoder Design

In this section we briefly describe the joint $(3, k)$-regular LDPC code and decoder design methodology [14]. It is well known that the BP (or Log-BP) decoding algorithm works well if the underlying Tanner graph is 4-cycle free and does not contain too many short cycles. Thus the motivation of this joint design approach is to construct an LDPC code that not only fits to a high-speed partly parallel decoder but also has the average cycle length as large as possible in its 4-cycle free Tanner graph. This joint design process is outlined as follows and the corresponding schematic flow diagram is shown in Fig. 2.

1. Explicitly construct two matrices, $\mathbf{H}_1$ and $\mathbf{H}_2$, in such a way that $\hat{\mathbf{H}} = [\mathbf{H}_1^T, \mathbf{H}_2^T]^T$ defines a $(2, k)$-regular LDPC code C_2 whose Tanner graph has the girth[2] of 12;

[2]Girth is the length of a shortest cycle in a graph.

2. Develop a partly parallel decoder that is configured by a set of constrained random parameters and defines a $(3, k)$-regular LDPC code ensemble, in which each code is a sub-code of C_2 and has the parity check matrix $\mathbf{H} = [\hat{\mathbf{H}}^T, \mathbf{H}_3^T]^T$;

3. Select a good $(3, k)$-regular LDPC code from the code ensemble based on the criteria of large Tanner graph average cycle length and computer simulations. Typically the parity check matrix of the selected code has only few redundant checks, so we may assume the code rate is always $1 - 3/k$.

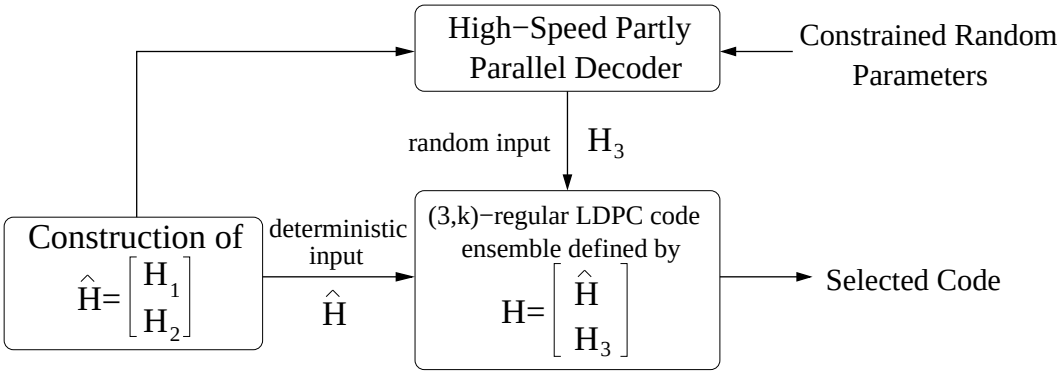

Figure 2: Joint design flow diagram.

Construction of $\hat{\mathbf{H}} = [\mathbf{H}_1^T, \mathbf{H}_2^T]^T$: The structure of $\hat{\mathbf{H}}$ is shown in Fig. 3, where both $\mathbf{H}_1$ and $\mathbf{H}_2$ are $L \cdot k$ by $L \cdot k^2$ submatrices. Each block matrix $\mathbf{I}_{x,y}$ in $\mathbf{H}_1$ is an $L \times L$ identity matrix and each block matrix $\mathbf{P}_{x,y}$ in $\mathbf{H}_2$ is obtained by a cyclic shift of an $L \times L$ identity matrix. Let T denote the right cyclic shift operator where $T^u(\mathbf{Q})$ represents right cyclic shifting matrix $\mathbf{Q}$ by u columns, then $\mathbf{P}_{x,y} = T^u(\mathbf{I})$ where $u = ((x-1) \cdot y) \bmod L$ and $\mathbf{I}$ represents the $L \times L$ identity matrix, *e.g.*, let $L = 5$, $x = 3$ and $y = 4$, we have $u = (x-1) \cdot y \bmod L = 8 \bmod 5 = 3$, then

$$\mathbf{P}_{3,4} = T^3(\mathbf{I}) = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

Notice that in both $\mathbf{H}_1$ and $\mathbf{H}_2$ each row contains k 1's and each column contains a single 1. Thus, the matrix $\hat{\mathbf{H}} = [\mathbf{H}_1^T, \mathbf{H}_2^T]^T$ defines a $(2, k)$-regular LDPC code C_2 with $L \cdot k^2$ variable nodes and $2L \cdot k$ check nodes. Let G denote the Tanner graph of C_2, we have the following theorem regarding to the girth of G:

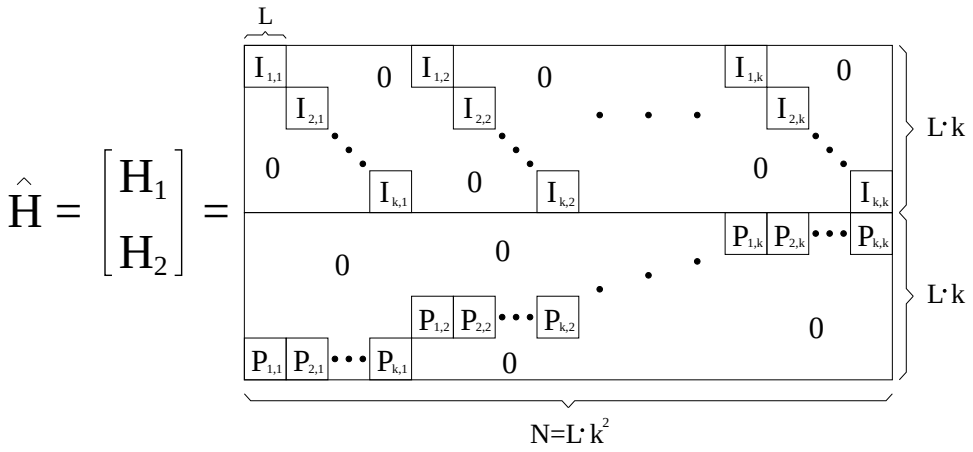

Figure 3: Structure of $\hat{\mathbf{H}} = [\mathbf{H}_1^T, \mathbf{H}_2^T]^T$

Theorem 3.1 *If L can not be factored as $L = a \cdot b$, where $a, b \in \{0, \cdots, k-1\}$, then the girth of G is 12 and there is at least one 12-cycle passing each check node.*

Partly Parallel Decoder: Based on the specific structure of $\hat{\mathbf{H}}$, a principal $(3, k)$-regular LDPC code partly parallel decoder structure was presented in [14]. This decoder is configured by a set of constrained random parameters and defines a $(3, k)$-regular LDPC code ensemble. Each code in this ensemble is essentially constructed by inserting extra $L \cdot k$ check nodes to the high-girth $(2, k)$-regular LDPC code C_2 under the constraint specified by the decoder. Therefore, it is reasonable to expect that the codes in this ensemble more likely do not contain too many short cycles and we may easily select a good code from it. For real applications, we can select a good code from this code ensemble as follows: first in the code ensemble find several codes with relatively high average cycle lengths, then select the one leading to the best result in the computer simulations.

The principal partly parallel decoder structure presented in [14] has the following properties:

- It contains k^2 memory banks, each one consists of several RAMs to store all the decoding messages associated with L variable nodes;
- Each memory bank associates with one address generator that is configured by one element in a constrained random integer set $\mathcal{R}$.
- It contains a configurable random-like one-dimensional shuffle network $\mathcal{S}$ with the routing complexity scaled by k^2;
- It contains k^2 VNUs and k CNUs so that the VNU and CNU folding factors are $L \cdot k^2/k^2 = L$ and $3L \cdot k/k = 3L$, respectively;

- Each iteration completes in $3L$ clock cycles in which only CNUs work in the first $2L$ clock cycles and both CNUs and VNUs work in the last L clock cycles.

Over all the possible $\mathcal{R}$ and $\mathcal{S}$, this decoder defines a $(3, k)$-regular LDPC code ensemble in which each code has the parity check matrix $\mathbf{H} = [\hat{\mathbf{H}}^T, \mathbf{H}_3^T]^T$, where the submatrix $\mathbf{H}_3$ is jointly specified by $\mathcal{R}$ and S.

4 Partly Parallel Decoder Architecture

In this work, applying the joint code and decoder design methodology, we develop a high-speed $(3, k)$-regular LDPC code partly parallel decoder architecture based on which a 9216-bit, rate-$1/2$ $(3, 6)$-regular LDPC code partly parallel decoder has been implemented using Xilinx Virtex FPGA device. Compared with the structure presented in [14], this partly parallel decoder architecture has the following distinct characteristics:

- It employs a novel concatenated configurable random two-dimensional shuffle network implementation scheme to realize the random-like connectivity with low routing overhead, which is especially desirable for FPGA implementations;

- To improve the decoding throughput, both the VNU folding factor and CNU folding factor are L, instead of L and $3L$ in the structure presented in [14];

- To simplify the control logic design and reduce the memory bandwidth requirement, this decoder completes each decoding iteration in $2L$ clock cycles in which CNUs and VNUs work in the 1^{st} and 2^{nd} L clock cycles, alternatively.

Following the joint design methodology, we have that this decoder should define a $(3, k)$-regular LDPC code ensemble in which each code has $L \cdot k^2$ variable nodes and $3L \cdot k$ check nodes, and as illustrated in Fig. 4, the parity check matrix of each code has the form $\mathbf{H} = [\mathbf{H}_1^T, \mathbf{H}_2^T, \mathbf{H}_3^T]^T$ where $\mathbf{H}_1$ and $\mathbf{H}_2$ have the explicit structures as shown in Fig. 3 and the random-like $\mathbf{H}_3$ is specified by certain configuration parameters of the decoder. To facilitate the description of the decoder architecture, we introduce some definitions as follows: we denote the submatrix consisting of the L consecutive columns in $\mathbf{H}$ that go through the block matrix $\mathbf{I}_{x,y}$ as $\mathbf{H}^{(x,y)}$, in which from left to right each column is labeled as $h_i^{(x,y)}$ with i increasing from 1 to L, as shown in Fig. 4. We label the variable node corresponding to column $h_i^{(x,y)}$ as $v_i^{(x,y)}$ and the L variable nodes $v_i^{(x,y)}$ for $i = 1, \cdots, L$ constitute a variable node group $\mathrm{VG}_{x,y}$. Finally, we arrange the $L \cdot k$ check nodes corresponding to all the $L \cdot k$ rows of submatrix $\mathbf{H}_i$ into check node group CG_i.

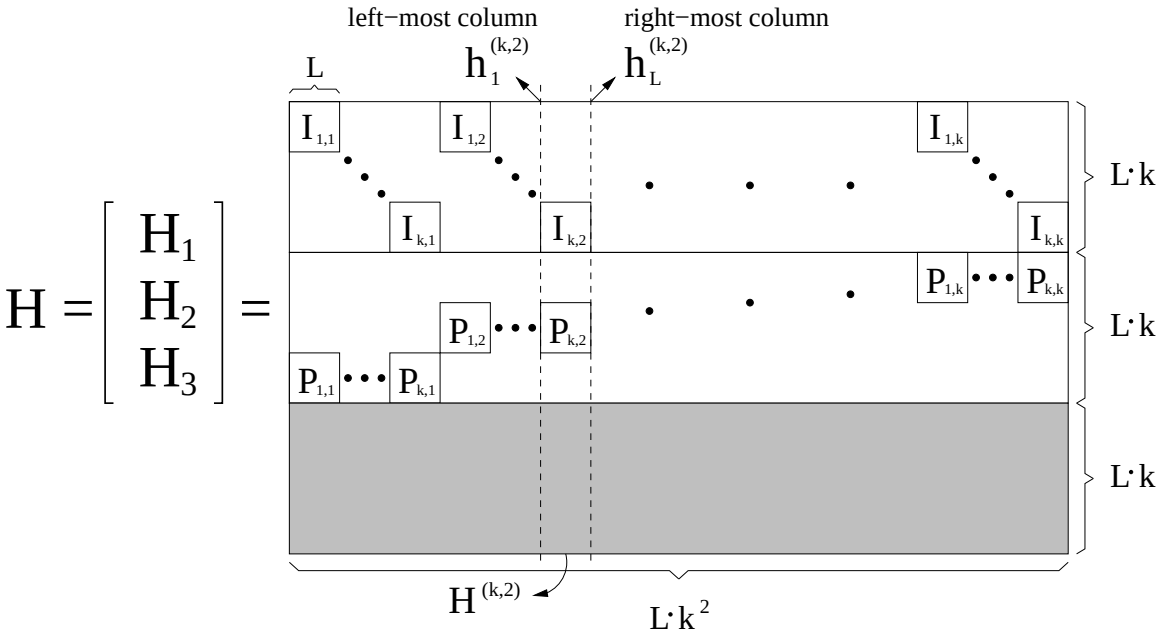

Figure 4: The parity check matrix.

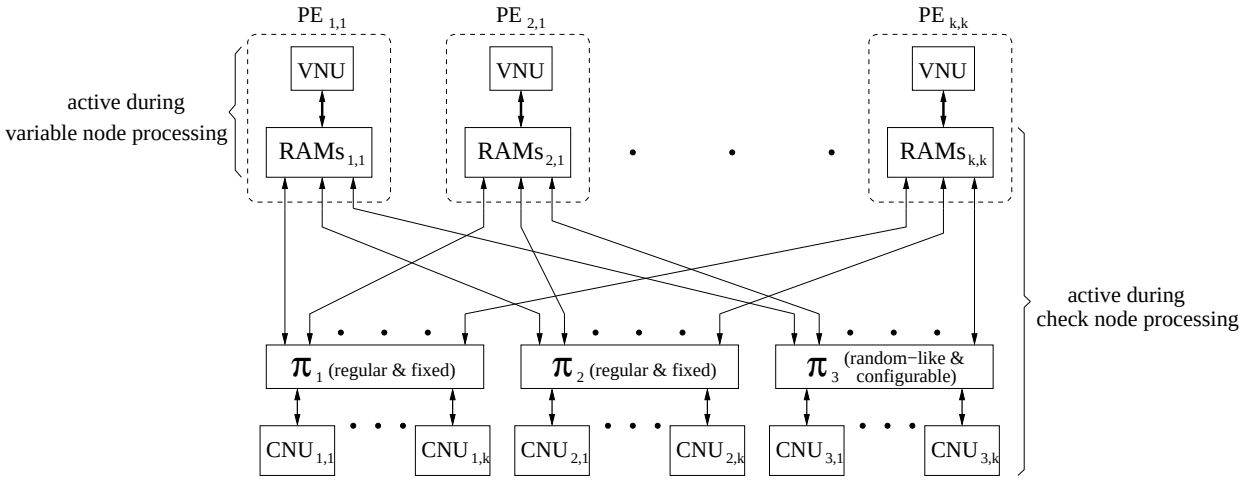

Figure 5: The principal $(3, k)$-regular LDPC code partly parallel decoder structure.

Fig. 5 shows the principal structure of this partly parallel decoder. It mainly contains k^2 PE Blocks $\text{PE}_{x,y}$ for $1 \leq x, y \leq k$, three bi-directional shuffle networks π_1, π_2 and π_3 and $3 \cdot k$ CNUs. Each $\text{PE}_{x,y}$ contains one memory bank $\text{RAMs}_{x,y}$ that stores all the decoding messages, including the intrinsic and extrinsic messages and hard decisions, associated with all the L variable nodes in the variable node group $\text{VG}_{x,y}$, and contains one VNU to perform the variable node computations for these L variable nodes. Each bi-directional shuffle network π_i realizes the extrinsic message exchange between all the $L \cdot k^2$ variable nodes and the $L \cdot k$ check nodes in CG_i. The k $\text{CNU}_{i,j}$ for $j = 1, \cdots, k$ perform the check node computations for all the $L \cdot k$ check nodes in CG_i.

This decoder completes each decoding iteration in $2L$ clock cycles, and during the 1^{st} and 2^{nd} L clock cycles, it works in *check node processing* mode and *variable node processing* mode, respectively. In the check node processing mode, the decoder not only performs the computations of all the check nodes but also completes the extrinsic message exchange between neighboring nodes. In variable node processing mode, the decoder only performs the computations of all the variable nodes.

The intrinsic and extrinsic messages are all quantized to 5 bits and the iterative decoding datapaths of this partly parallel decoder are illustrated in Fig. 6, in which the datapaths in check node processing and variable node processing are represented by solid lines and dash dot lines, respectively. As shown in Fig. 6, each PE Block $\text{PE}_{x,y}$ contains five RAM blocks: EXT_RAM_i for $i = 1, 2, 3$, INT_RAM and DEC_RAM. Each EXT_RAM_i has L memory locations and the location with the address $d - 1$ ($1 \leq d \leq L$) contains the extrinsic messages exchanged between the variable node $v_d^{(x,y)}$ in $\text{VG}_{x,y}$ and its neighboring check node in CG_i. The INT_RAM and DEC_RAM store the intrinsic message and hard decision associated with node $v_d^{(x,y)}$ at the memory location with the address $d - 1$ ($1 \leq d \leq L$). As we will see later, such decoding messages storage strategy could greatly simplify the control logic for generating the memory access address.

For the purpose of simplicity, in Fig. 6 we do not show the datapath from INT_RAM to EXT_RAM_i's for extrinsic message initialization, which can be easily realized in L clock cycles before the decoder enters the iterative decoding process.

4.1 Check node processing

During the check node processing, the decoder performs the computations of all the check nodes and realizes the extrinsic message exchange between all the neighboring nodes. At the beginning of check node processing, in each $\text{PE}_{x,y}$ the memory location with address $d - 1$ in EXT_RAM_i contains 6-bit *hybrid* data that consists of 1-bit hard decision and 5-bit variable-to-check extrinsic message associated with the variable node $v_d^{(x,y)}$ in

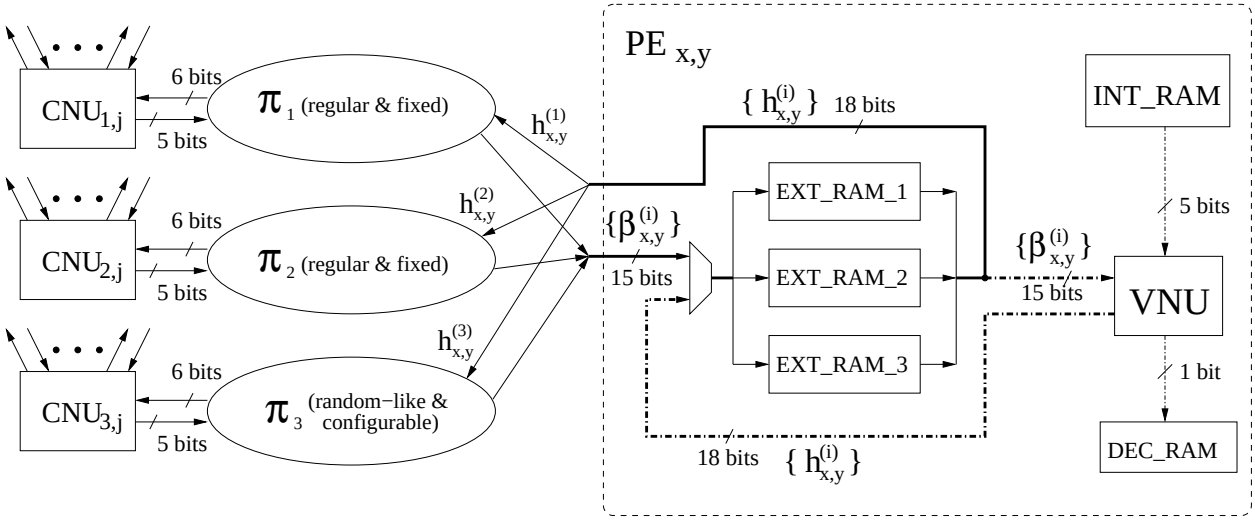

Figure 6: Iterative decoding datapaths.

$\text{VG}_{x,y}$. Each clock cycle this decoder performs the *read-shuffle-modify-unshuffle-write* operations to *convert* one variable-to-check extrinsic message in each EXT_RAM_i to its check-to-variable counterpart. As illustrated in Fig. 6, we may outline the datapath loop in check node processing as follows:

1. *Read*: One 6-bit hybrid data $h_{x,y}^{(i)}$ is read from each EXT_RAM_i in each $\text{PE}_{x,y}$;
2. *Shuffle*: Each hybrid data $h_{x,y}^{(i)}$ goes through the shuffle network π_i and arrives $\text{CNU}_{i,j}$;
3. *Modify*: Each $\text{CNU}_{i,j}$ performs the parity check on the 6 input hard decision bits and generates the 6 output 5-bit check-to-variable extrinsic messages $\beta_{x,y}^{(i)}$ based on the 6 input 5-bit variable-to-check extrinsic messages;
4. *Unshuffle*: Send each check-to-variable extrinsic message $\beta_{x,y}^{(i)}$ back to the PE Block via the same path as its variable-to-check counterpart;
5. *Write*: Write each $\beta_{x,y}^{(i)}$ to the same memory location in EXT_RAM_i as its variable-to-check counterpart.

All the CNUs deliver the parity check results to a central control block that will, at the end of check node processing, determine whether all the parity check equations specified by the parity check matrix have been satisfied, if yes, the decoding for current code frame will terminate.

To achieve higher decoding throughput, we implement the read-shuffle-modify-unshuffle-write loop operation by five-stage pipelining as shown in Fig. 7, where CNU is 1-stage pipelined. To make this pipelining scheme feasible, we realize each bi-directional I/O connection in the three shuffle networks by two distinct

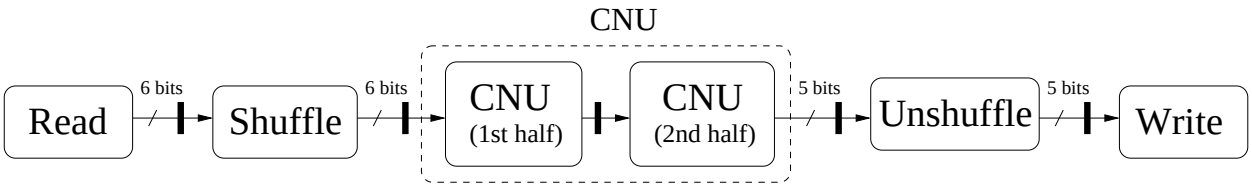

Figure 7: Five-stage pipelining of the check node processing datapath.

sets of wires with opposite directions, which means that the hybrid data from PE Blocks to CNUs and the check-to-variable extrinsic messages from CNUs to PE Blocks are carried on distinct sets of wires. Compared with sharing one set of wires in time-multiplexed fashion, this approach has higher wire routing overhead but obviates the logic gate overhead due to the realization of time-multiplex and, more importantly, make it feasible to directly pipeline the datapath loop for higher decoding throughput.

In this decoder, one address generator $\mathrm{AG}_{x,y}^{(i)}$ associates with one EXT_RAM_i in each $\mathrm{PE}_{x,y}$. In the check node processing, $\mathrm{AG}_{x,y}^{(i)}$ generates the address for reading hybrid data and, due to the five-stage pipelining of datapath loop, the address for writing back the check-to-variable message is obtained via delaying the read address by five clock cycles. It is clear that the connectivity among all the variable nodes and check nodes, or the entire parity check matrix, realized by this decoder is jointly specified by all the address generators and the three shuffle networks. Moreover, for $i = 1, 2, 3$, the connectivity among all the variable nodes and the check nodes in CG_i is completely determined by $\mathrm{AG}_{x,y}^{(i)}$ and π_i. Following the joint design methodology, we implement all the address generators and the three shuffle networks as follows.

4.1.1 Implementations of $\mathrm{AG}_{x,y}^{(1)}$ and π_1

The bi-directional shuffle network π_1 and $\mathrm{AG}_{x,y}^{(1)}$ realize the connectivity among all the variable nodes and all the check nodes in CG_1 as specified by the fixed submatrix $\mathbf{H}_1$. Recall that node $v_d^{(x,y)}$ corresponds to the column $h_i^{(x,y)}$ as illustrated in Fig. 4 and the extrinsic messages associated with node $v_d^{(x,y)}$ is always stored at address $d-1$. Exploiting the explicit structure of $\mathbf{H}_1$, we easily obtain the implementation schemes for $\mathrm{AG}_{x,y}^{(1)}$ and π_1 as follows:

- Each $\mathrm{AG}_{x,y}^{(1)}$ is realized as a $\lceil \log_2 L \rceil$-bit binary counter that is cleared to zero at the beginning of check node processing;
- The bi-directional shuffle network π_1 connects the k $\mathrm{PE}_{x,y}$ with the same x-index to the same CNU.

4.1.2 Implementations of $\mathbf{AG}_{x,y}^{(2)}$ and π_2

The bi-directional shuffle network π_2 and $\mathrm{AG}_{x,y}^{(2)}$ realize the connectivity among all the variable nodes and all the check nodes in CG_2 as specified by the fixed matrix $\mathbf{H}_2$. Similarly, exploiting the extrinsic messages storage strategy and the explicit structure of $\mathbf{H}_2$, we implement $\mathrm{AG}_{x,y}^{(2)}$ and π_2 as follows:

- Each $\mathrm{AG}_{x,y}^{(2)}$ is realized as a $\lceil \log_2 L \rceil$-bit binary counter that only counts up to the value $L-1$ and is loaded with the value of $((x-1)\cdot y) \bmod L$ at the beginning of check node processing;
- The bi-directional shuffle network π_2 connects the k $\mathrm{PE}_{x,y}$ with the same y-index to the same CNU.

Notice that the counter load value for each $\mathrm{AG}_{x,y}^{(2)}$ directly comes from the construction of each block matrix $\mathbf{P}_{x,y}$ in $\mathbf{H}_2$ as described in Section 3.

4.1.3 Implementations of $\mathbf{AG}_{x,y}^{(3)}$ and π_3

The bi-directional shuffle network π_3 and $\mathrm{AG}_{x,y}^{(3)}$ jointly define the connectivity between all the variable nodes and all the check nodes in CG_3, which is represented by $\mathbf{H}_3$ as illustrated in Fig. 4. In the above, we show that by exploiting the specific structures of $\mathbf{H}_1$ and $\mathbf{H}_2$ and the extrinsic messages storage strategy, we can directly obtain the implementations of each $\mathrm{AG}_{x,y}^{(i)}$ and π_i for $i = 1, 2$. However, the implementations of $\mathrm{AG}_{x,y}^{(3)}$ and π_3 are not easy because of the following requirements on $\mathbf{H}_3$:

1. The Tanner graph corresponding to the parity check matrix $\mathbf{H} = [\mathbf{H}_1^T, \mathbf{H}_2^T, \mathbf{H}_3^T]^T$ should be 4-cycle free;
2. To make $\mathbf{H}$ be random to some extent, $\mathbf{H}_3$ should be random-like;

As proposed in [14], to simplify the design process, we *separately* conceive $\mathrm{AG}_{x,y}^{(3)}$ and π_3 in such a way that the implementations of $\mathrm{AG}_{x,y}^{(3)}$ and π_3 accomplish the above 1^{st} and 2^{nd} requirement, respectively.

Implementations of $\mathbf{AG}_{x,y}^{(3)}$: We implement each $\mathrm{AG}_{x,y}^{(3)}$ as a $\lceil \log_2 L \rceil$-bit binary counter that counts up to the value $L-1$ and is initialized with a constant value $t_{x,y}$ at the beginning of check node processing. Each $t_{x,y}$ is selected in random under the following two constraints:

1. Given x, $t_{x,y_1} \neq t_{x,y_2}, \forall y_1, y_2 \in \{1, \cdots, k\}$;
2. Given y, $t_{x_1,y} - t_{x_2,y} \not\equiv ((x_1 - x_2)\cdot y) \bmod L, \forall x_1, x_2 \in \{1, \cdots, k\}$.

It can be proved that the above two constraints on $t_{x,y}$ are sufficient to make the entire parity check matrix $\mathbf{H}$ always correspond to a 4-cycle free Tanner graph no matter how we implement π_3.

Implementation of π_3: Since each $\mathrm{AG}_{x,y}^{(3)}$ is realized as a counter, the pattern of shuffle network π_3 can not be *fixed* otherwise the shuffle pattern of π_3 will regularly repeat in the $\mathbf{H}_3$, which means that $\mathbf{H}_3$ will always contain very regular connectivity patterns no matter how random-like the pattern of π_3 itself is. Thus we should make π_3 be configurable to some extent. In this work, we propose the following concatenated configurable random shuffle network implementation scheme for π_3.

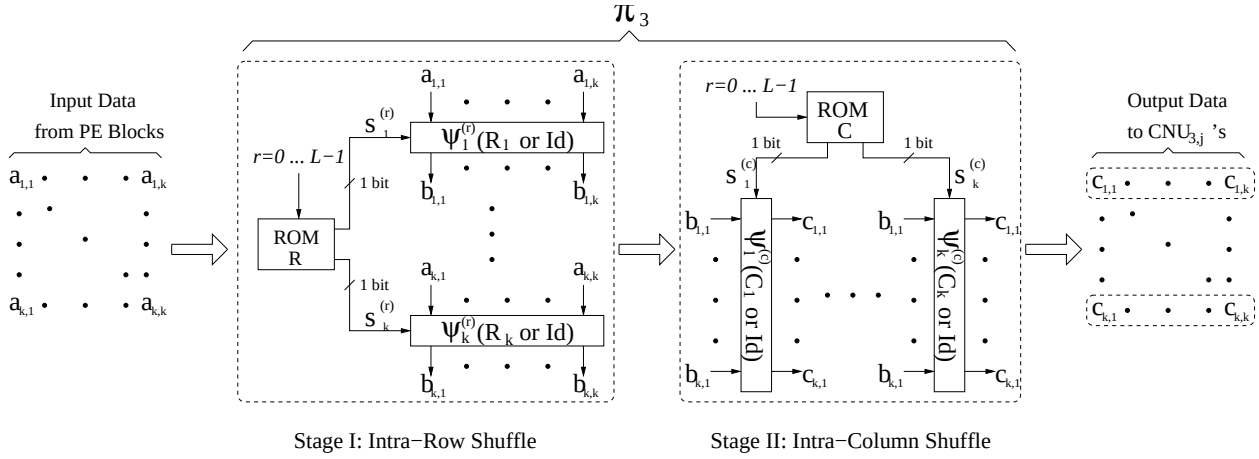

Figure 8: Forward path of π_3.

Fig. 8 shows the forward path (from $\mathrm{PE}_{x,y}$ to $\mathrm{CNU}_{3,j}$) of the bi-directional shuffle network π_3. In each clock cycle, it realizes the data shuffle from $a_{x,y}$ to $c_{x,y}$ by two concatenated stages: *intra-row* shuffle and *intra-column* shuffle. Firstly, the $a_{x,y}$ data block, where each $a_{x,y}$ comes from $\mathrm{PE}_{x,y}$, passes an *intra-row* shuffle network array in which each shuffle network $\Psi_x^{(r)}$ shuffles the k input data $a_{x,y}$ to $b_{x,y}$ for $1 \leq y \leq k$. Each $\Psi_x^{(r)}$ is configured by 1-bit control signal $s_x^{(r)}$ leading to the fixed random permutation R_x if $s_x^{(r)} = 1$, or to the identity permutation (Id) otherwise. The reason why we use the Id pattern instead of another random shuffle pattern is to minimize the routing overhead, and our simulations suggest that there is no gain on the error-correcting performance by using another random shuffle pattern instead of Id pattern. The k-bit configuration word $\mathbf{s}^{(r)}$ changes every clock cycle and all the L k-bit control words are stored in ROM R. Next, the $b_{x,y}$ data block goes through an *intra-column* shuffle network array in which each $\Psi_y^{(c)}$ shuffles the k $b_{x,y}$ to $c_{x,y}$ for $1 \leq x \leq k$. Similarly, each $\Psi_y^{(c)}$ is configured by 1-bit control signal $s_y^{(c)}$ leading to the fixed random permutation C_y if $s_y^{(c)} = 1$, or to the identity permutation (Id) otherwise. The k-bit configuration word $s_y^{(c)}$ changes every clock cycle and all the L k-bit control words are stored in ROM C. As the output of forward path, the k $c_{x,y}$ with the same x-index are delivered to the same $\mathrm{CNU}_{3,j}$. To realize the bi-directional shuffle, we only need to implement each configurable shuffle network $\Psi_x^{(r)}$ and $\Psi_y^{(c)}$ as bi-directional so that π_3 can

unshuffle the k^2 data backward from $\text{CNU}_{3,j}$ to $\text{PE}_{x,y}$ along the same route as the forward path on distinct sets of wires. Notice that, due to the pipelining on the datapath loop, the backward path control signals are obtained via delaying the forward path control signals by three clock cycles.

To make the connectivity realized by π_3 be random-like and change each clock cycle, we only need to randomly generate the control word $s_x^{(r)}$ and $s_y^{(c)}$ for each clock cycle and the fixed shuffle patterns of each R_x and C_y. Since most modern FPGA devices have multiple metal layers, the implementations of the two shuffle arrays can be overlapped from the bird's-eye view. Therefore, the above concatenated implementation scheme will confine all the routing wires to small area (in one row or one column), which will significantly reduce the possibility of routing congestion and reduce the routing overhead.

4.2 Variable node processing

Compared with the above check node processing, the operations performed in the variable node processing is quite simple since the decoder only needs to carry out all the variable node computations. Notice that at the beginning of variable node processing, the three 5-bit check-to-variable extrinsic messages associated with each variable node $v_d^{(x,y)}$ are stored at the address $d-1$ of the three EXT_RAM$_i$ in $\text{PE}_{x,y}$. The 5-bit intrinsic message associated with variable node $v_d^{(x,y)}$ is also stored at the address $d-1$ of INT_RAM in $\text{PE}_{x,y}$. In each clock cycle, this decoder performs the *read-modify-write* operations to *convert* the three check-to-variable extrinsic messages associated with the same variable node to three hybrid data consisting of variable-to-check extrinsic messages and hard decisions. As shown in Fig. 6, we may outline the datapath loop in variable node processing as follows:

1. *Read*: In each $\text{PE}_{x,y}$, three 5-bit check-to-variable extrinsic messages $\beta_{x,y}^{(i)}$ and one 5-bit intrinsic messages $\gamma_{x,y}$ associated with the same variable node are read from the three EXT_RAM$_i$ and INT_RAM at the same address;

2. *Modify*: Based on the input check-to-variable extrinsic messages and intrinsic message, each VNU generates the 1-bit hard decision $\hat{x}_{x,y}$ and three 6-bit hybrid data $h_{x,y}^{(i)}$;

3. *Write*: Each $h_{x,y}^{(i)}$ is written back to the same memory location as its check-to-variable counterpart and $\hat{x}_{x,y}$ is written to DEC_RAM.

The forward path from memory to VNU and backward path from VNU to memory are implemented by distinct sets of wires and the entire read-modify-write datapath loop is pipelined by three-stage pipelining as

illustrated in Fig. 9.

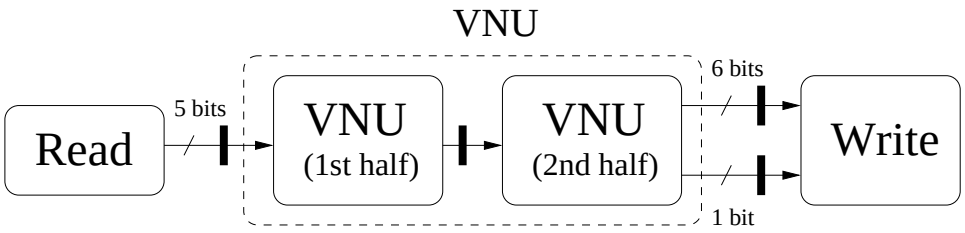

Figure 9: Three-stage pipelining of the variable node processing datapath.

Since all the extrinsic and intrinsic messages associated with the same variable node are stored at the same address in different RAM blocks, we can use only one binary counter to generate all the read address. Due to the pipelining of the datapath, the write address is obtained via delaying the read address by three clock cycles.

4.3 CNU and VNU Architectures

Each CNU carries out the operations of one check node, including the parity check and computation of check-to-variable extrinsic messages. Fig. 10 shows the CNU architecture for check node with the degree of 6. Each input $x^{(i)}$ is a 6-bit hybrid data consisting of 1-bit hard decision and 5-bit variable-to-check extrinsic message. The parity check is performed by XORing all the six 1-bit hard decisions. Each 5-bit variable-to-check extrinsic messages is represented by *sign-magnitude* format with a sign bit and four magnitude bits. The architecture for computing the check-to-variable extrinsic messages is directly obtained from (1) in Algorithm 2.1. The function $f(x) = \log(\frac{1+e^{-|x|}}{1-e^{-|x|}})$ is realized by the LUT (Look-Up Table) that is implemented as a combinational logic block in FPGA. Each output 5-bit check-to-variable extrinsic message $y^{(i)}$ is also represented by sign-magnitude format.

Each VNU generates the hard decision and all the variable-to-check extrinsic messages associated with one variable node. Fig. 11 shows the VNU architecture for variable node with the degree of 3. With the input 5-bit intrinsic message z and three 5-bit check-to-variable extrinsic messages $y^{(i)}$ associated with the same variable node, VNU generates three 5-bit variable-to-check extrinsic messages and 1-bit hard decision according to (2) and (3) in Algorithm 2.1, respectively. To enable each CNU receive the hard decisions to perform parity check as described in the above, the hard decision is combined with each 5-bit variable-to-check extrinsic message to form 6-bit hybrid data $x^{(i)}$ as shown in Fig. 11. Since each input check-to-variable extrinsic message $y^{(i)}$ is represented by sign-magnitude format, we need to convert it to *two's complement* format before performing the additions. Before going through the LUT that realizes $f(x) = \log(\frac{1+e^{-|x|}}{1-e^{-|x|}})$, each data is converted back the

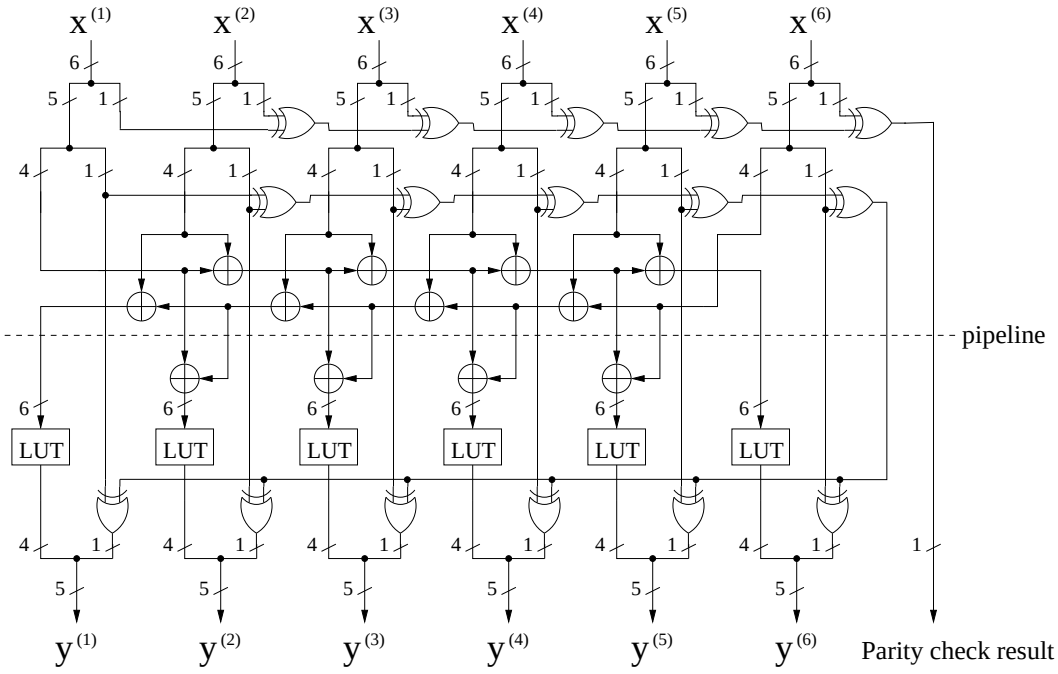

Figure 10: Architecture for check node processor unit (CNU) with $k = 6$.

sign-magnitude format.

4.4 Data Input/Output

This partly parallel decoder works simultaneously on three consecutive code frames in two-stage pipelining mode: while one frame is being iteratively decoded, the next frame is loaded into the decoder and the hard decisions of the previous frame are read out from the decoder. Thus each INT_RAM contains two RAM blocks to store the intrinsic messages of both current and next frames. Similarly, each DEC_RAM contains two RAM blocks to store the hard decisions of both current and previous frames.

The design scheme for intrinsic message input and hard decision output is heavily dependent on the floor planning of the k^2 PE Blocks. To minimize the routing overhead, we develop a square-shaped floor planning for PE Blocks as illustrated in Fig. 12 and the corresponding data input/output scheme is described in the following:

1. **Intrinsic Data Input**: The intrinsic messages of next frame is loaded 1 symbol per clock cycle. As shown in Fig. 12, the memory location of each input intrinsic data is determined by the input load address that has the width of $(\lceil \log_2 L \rceil + \lceil \log_2 k^2 \rceil)$ bits in which $\lceil \log_2 k^2 \rceil$ bits specify which PE Block (or which INT_RAM) is being accessed and the other $\lceil \log_2 L \rceil$ bits locate the memory location in the selected

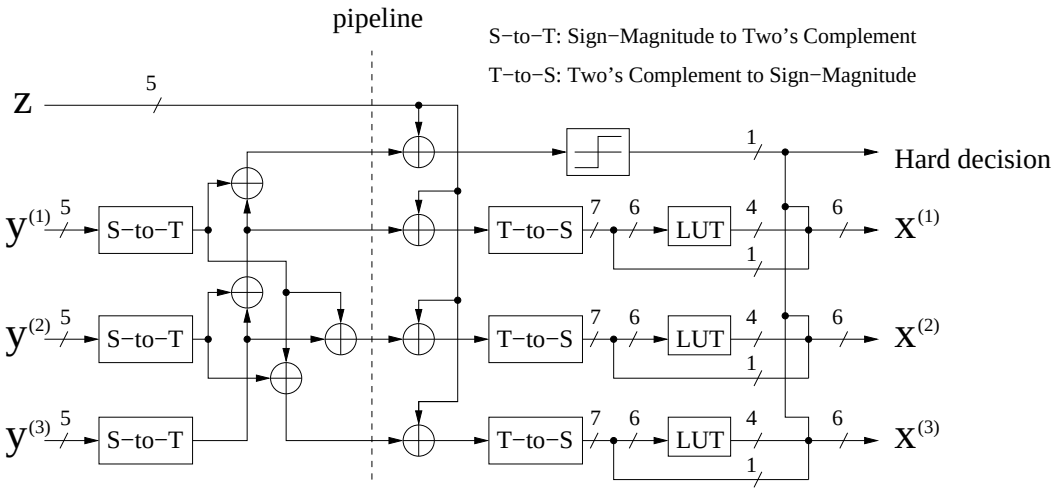

Figure 11: Architecture for variable node processor unit (VNU) with $j = 3$.

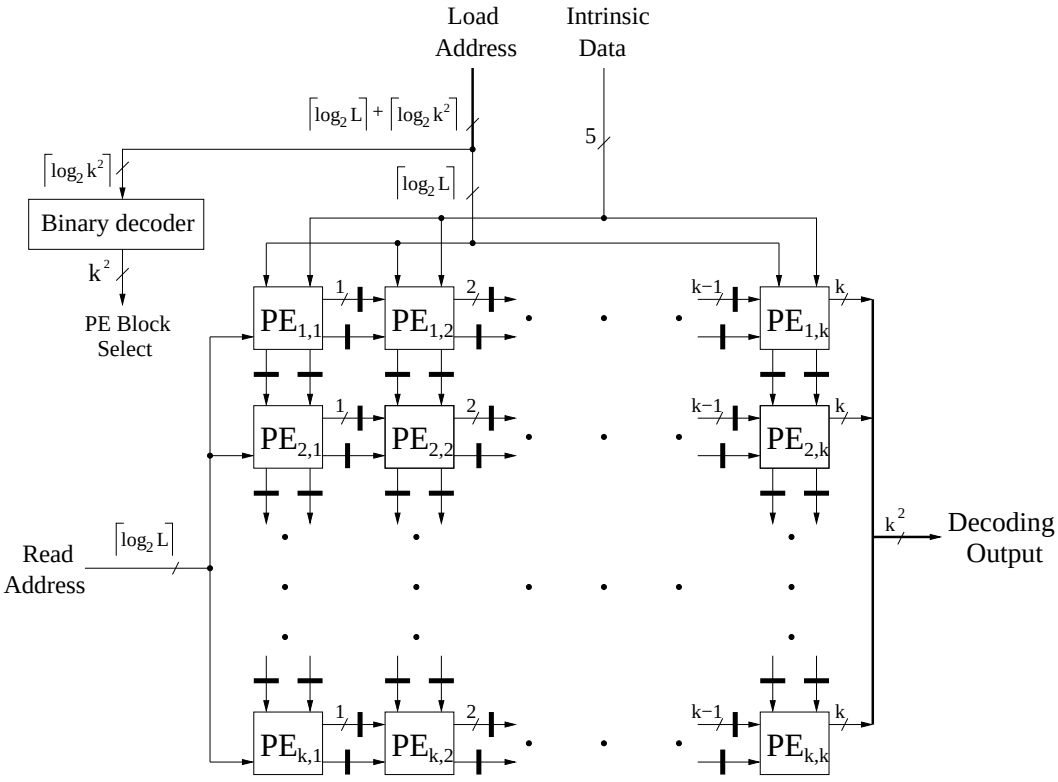

Figure 12: Data Input/Output structure.

INT_RAM. As shown in Fig. 12, the primary intrinsic data and load address input directly connect to the k PE Blocks $\text{PE}_{1,y}$ for $1 \leq y \leq k$, and from each $\text{PE}_{x,y}$ the intrinsic data and load address are delivered to the adjacent PE Block $\text{PE}_{x+1,y}$ in pipelined fashion.

2. **Decoded Data Output**: The decoded data (or hard decisions) of the previous frame is read out in pipelined fashion. As shown in Fig. 12, the primary $\lceil \log_2 L \rceil$-bit read address input directly connects to the k PE Blocks $\text{PE}_{x,1}$ for $1 \leq x \leq k$, and from each $\text{PE}_{x,y}$ the read address are delivered to the adjacent block $\text{PE}_{x,y+1}$ in pipelined fashion. Based on its input read address, each PE Block outputs 1-bit hard decision per clock cycle. Therefore, as illustrated in Fig. 12, the width of pipelined decoded data bus increases by 1 after going through one PE Block, and at the rightmost side, we obtain k k-bit decoded output that are combined together as the k^2-bit primary decoded data output.

5 FPGA Implementation

Applying the above decoder architecture, we implemented a $(3, 6)$-regular LDPC code partly parallel decoder for $L = 256$ using Xilinx Virtex-E XCV2600E device with the package FG1156. The corresponding LDPC code length is $N = L \cdot k^2 = 256 \cdot 6^2 = 9216$ and code rate is $1/2$. We obtain the constrained random parameter set for implementing π_3 and each $\text{AG}_{x,y}^{(3)}$ as follows: first generate a large number of parameter sets from which we find few sets leading to relatively high Tanner graph average cycle length, then we select one set leading to the best performance based on computer simulations.

The target XCV2600E FPGA device contains 184 large on-chip block RAMs, each one is a fully synchronous dual-port 4K-bit RAM. In this decoder implementation, we configure each dual-port 4K-bit RAM as two independent single-port 256×8-bit RAM blocks so that each EXT_RAM$_i$ can be realized by one single-port 256×8-bit RAM block. Since each INT_RAM contains two RAM blocks for storing the intrinsic messages of both current and next code frames, we use two single-port 256×8-bit RAM blocks to implement one INT_RAM. Due to the relatively small memory size requirement, the DEC_RAM is realized by distributed RAM that provides shallow RAM structures implemented in CLBs. Since this decoder contains $k^2 = 36$ PE Blocks, each one incorporates one INT_RAM and three EXT_RAM$_i$'s, we totally utilize 180 single-port 256×8-bit RAM blocks (or 90 dual-port 4K-bit RAM blocks). We manually configured the placement of each PE Block according to the floor planning scheme as shown in Fig. 12 in Section 4.4. Notice that such placement scheme exactly matches the structure of the configurable shuffle network π_3 as described in Section 4.1.3, thus the routing overhead for implementing the π_3 is also minimized in this FPGA implementation.

From the architecture description in Section 4, we know that, during each clock cycle in the iterative decoding, this decoder need to perform both read and write operations on each single-port RAM block EXT_RAMi. Therefore, suppose the primary clock frequency is W, we must generate a $2 \times W$ clock signal as the RAM control signal to achieve read-and-write operation in one clock cycle. This 2x clock signal is generated using the Delay-Locked Loop (DLL) in XCV2600E.

To facilitate the entire implementation process, we extensively utilized the highly optimized Xilinx IP cores to instantiate many function blocks, *i.e.*, all the RAM blocks, all the counters for generating addresses, and the ROMs used to store the control signals for shuffle network π_3. Moreover, all the adders in CNUs and VNUs are implemented by ripple-carry adder that is exactly suitable for Xilinx FPGA implementations thanks to the on-chip dedicated fast arithmetic carry chain.

Table 1: FPGA resources utilization statistics.

Resource	Number	Utilization rate	Resource	Number	Utilization rate
Slices	11,792	46%	Slices Registers	10,105	19%
4 input LUTs	15,933	31%	Bonded IOBs	68	8%
Block RAMs	90	48%	DLLs	1	12%

This decoder was described in the VHDL hardware description language (HDL) and SYNOPSYS FPGA Express was used to synthesize the VHDL implementation. We used the Xilinx Development System tool suite to place and route the synthesized implementation for the target XCV2600E device with the speed option -7. Table 1 shows the hardware resource utilization statistics. Notice that 74% of the total utilized slices, or 8691 slices, were used for implementing all the CNUs and VNUs. Fig. 13 shows the placed and routed design in which the placement of all the PE Blocks are constrained based on the on-chip RAM block locations.

Based on the results reported by the Xilinx static timing analysis tool, the maximum decoder clock frequency can be 56 MHz. If this decoder performs s decoding iterations for each code frame, the total clock cycle number for decoding one frame will be $2s \cdot L + L$ where the extra L clock cycles is due to the initialization process, and the maximum symbol decoding throughput will be $56 \cdot k^2 \cdot L/(2s \cdot L + L) = 56 \cdot 36/(2s + 1)$ Mbps. Here we set $s = 18$ and obtain the maximum symbol decoding throughput as 54 Mbps. Fig. 14 shows the corresponding performance over AWGN channel with $s = 18$, including the BER (Bit Error Rate), FER (Frame Error Rate) and the average iteration numbers.

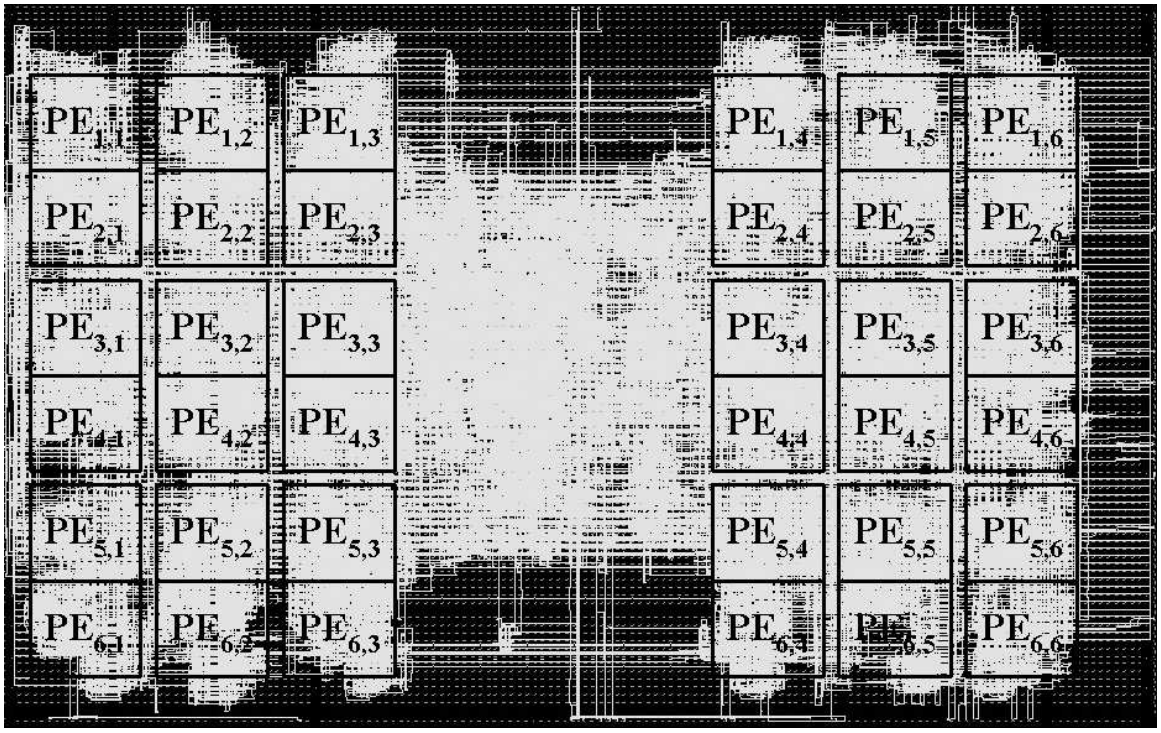

Figure 13: The placed and routed decoder implementation.

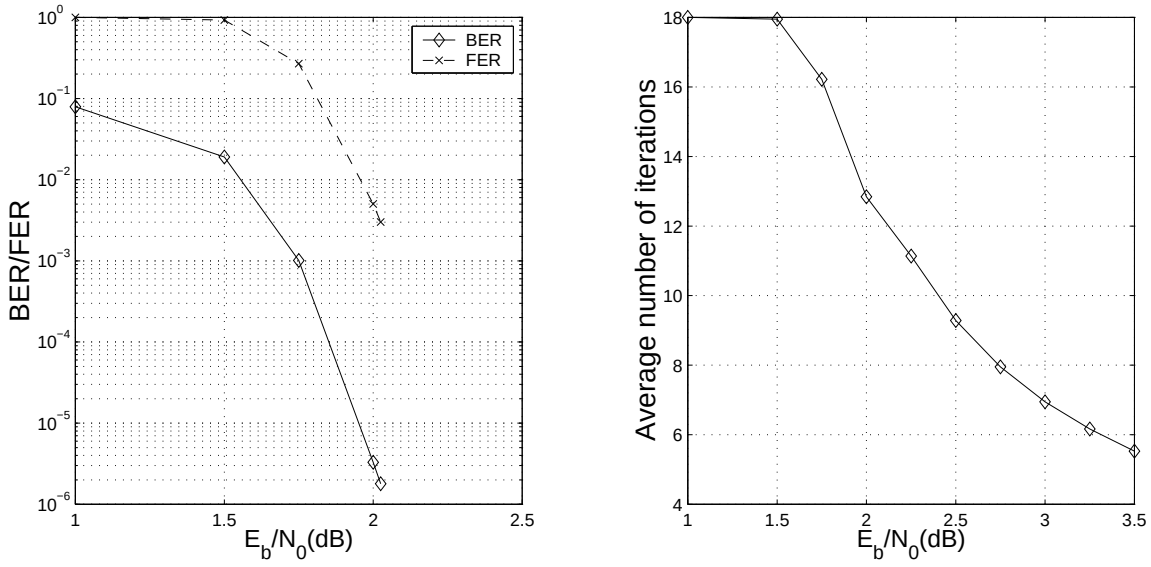

Figure 14: Simulation results on BER (Bit Error Rate), FER (Frame Error Rate) and the average iteration numbers.

6 Conclusion

Due to the unique characteristics of LDPC codes, we believe that jointly conceiving the code construction and partly parallel decoder design should be a key for practical high-speed LDPC coding system implementations. In this work, applying a joint design methodology, we developed a $(3, k)$-regular LDPC code high-speed partly parallel decoder architecture design and implemented a 9216-bit, rate-$1/2$ $(3, 6)$-regular LDPC code decoder on the Xilinx XCV2600E FPGA device. The detailed decoder architecture and floor planning scheme have been presented and a concatenated configurable random shuffle network implementation is proposed to minimize the routing overhead for the random-like shuffle network realization. With the maximum 18 decoding iterations, this decoder can achieve up to 54 Mbps symbol decoding throughput and the BER 10^{-6} at 2dB over AWGN channel. Moreover, exploiting the good minimum distance property of LDPC code, this decoder uses parity check after each iteration as earlier stopping criterion to effectively reduce the average energy consumption.

References

[1] R. G. Gallager, "Low-density parity-check codes", *IRE Transactions on Information Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

[2] R. G. Gallager, *Low-Density Parity-Check Codes*, M.I.T Press, 1963. available at http://justice.mit.edu/people/gallager.html.

[3] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices", *IEEE Transactions on Information Theory*, vol. 45, pp. 399–431, Mar. 1999.

[4] M. C. Davey and D. J. C. MacKay, "Low-density parity check codes over $GF(q)$", *IEEE Commun. Letters*, vol. 2, pp. 165–167, June 1998.

[5] M. Luby, M. Shokrolloahi, M. Mizenmacher, and D. Spielman, "Improved low-density parity-check codes using irregular graphs and belief propagation", *in Proc. 1998 IEEE Intl. Symp. Inform. Theory*, p. 117, Cambridge, MA, Aug. 1998. availabel at http://http.icsi.berkeley.edu/~luby/.

[6] T. Richardson and R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding", *IEEE Transactions on Information Theory*, vol. 47, pp. 599–618, Feb. 2001.

[7] T. Richardson, A. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes", *IEEE Transactions on Information Theory*, vol. 47, pp. 619–637, Feb. 2001.

[8] S.-Y. Chung, T. J. Richardson, and R. L. Urbanke, "Analysis of sum-product decoding of low-density parity-check codes using a Gaussian approximation", *IEEE Transactions on Information Theory*, vol. 47, pp. 657–670, Feb. 2001.

[9] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Improved low-density parity-check codes using irregular graphs", *IEEE Transactions on Information Theory*, vol. 47, pp. 585–598, Feb. 2001.

[10] S. Chung, G. D. Forney, T. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the shannon limit", *IEEE Communications Letters*, vol. 5, pp. 58–60, Feb. 2001.

[11] G. Miller and D. Burshtein, "Bounds on the maximum-likelihood decoding error probability of low-density parity-check codes", *IEEE Transactions on Information Theory*, vol. 47, pp. 2696–2710, Nov. 2001.

[12] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder", *IEEE Journal of Solid-State Circuits*, vol. 37, pp. 404–412, March 2002.

[13] E. Boutillon, J. Castura, and F. R. Kschischang, "Decoder-first code design", *in Proceedings of the 2nd International Symposium on Turbo Codes and Related Topics*, pp. 459–462, Brest, France, Sept. 2000. available at http://lester.univ-ubs.fr:8080/~boutillon/publications.html.

[14] T. Zhang and K. K. Parhi, "VLSI implementation-oriented $(3, k)$-regular low-density parity-check codes", pp. 25–36, IEEE Workshop on Signal Processing Systems (SiPS), Sept. 2001. available at http://www.ecse.rpi.edu/homepages/tzhang/.

[15] M. Chiani, A. Conti, and A. Ventura, "Evaluation of low-density parity-check codes over block fading channels", *in 2000 IEEE International Conference on Communications*, pp. 1183–1187, 2000.

[16] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, 1999.