

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

Tong Zhang

Department of Electrical, Computer, and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY 12180, USA
Email: tzhang@ecse.rpi.edu

Keshab K. Parhi

Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455, USA
Email: parhi@ece.umn.edu

Received 28 February 2002 and in revised form 6 December 2002

Because of their excellent error-correcting performance, low-density parity-check (LDPC) codes have recently attracted a lot of attention. In this paper, 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 2 dB over AWGN channel while performing maximum 18 decoding iterations.

Keywords and phrases: low-density parity-check codes, error-correcting coding, decoder, FPGA.

1. INTRODUCTION

In the past few years, the recently rediscovered low-density parity-check (LDPC) codes [1, 2, 3] have received a lot of attention 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, usually called Tanner graph, in which one set of N variable nodes corresponds to the set of code-word, another set of M check nodes corresponds to the set of parity-check constraints and each edge corresponds to a nonzero entry in the parity-check matrix \mathbf{H} . (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 .) 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 nonzero 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 Figure 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.

Recently, tremendous efforts have been devoted to analyze and improve the LDPC codes error-correcting capability, see [4, 5, 6, 7, 8, 9, 10, 11] and so forth. Besides their powerful error-correcting capability, another important reason why LDPC codes attract so many attention 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, for example, 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

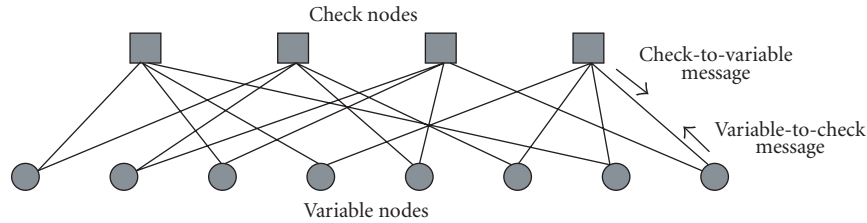


FIGURE 1: Tanner graph representation of an LDPC code and the decoding messages flow.

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, for example, for 1-K code length, the ASIC decoder implementation [12] consumes 1.7M gates. Moreover, as pointed out in [12], the routing overhead 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 hardware 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 of 54 Mbps symbol throughput and achieves BER (bit error rate) 10^{-6} at 2 dB 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 lead 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 1 (Iterative Log-BP Decoding Algorithm).

Input

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

Output

Hard decision $\hat{\mathbf{x}} = \{\hat{x}_1, \dots, \hat{x}_N\}$;

Procedure

- (1) *Initialization:* For each n , compute the intrinsic (or channel) message $\gamma_n = \log p_n^0/p_n^1$ and for each $(m, n) \in$

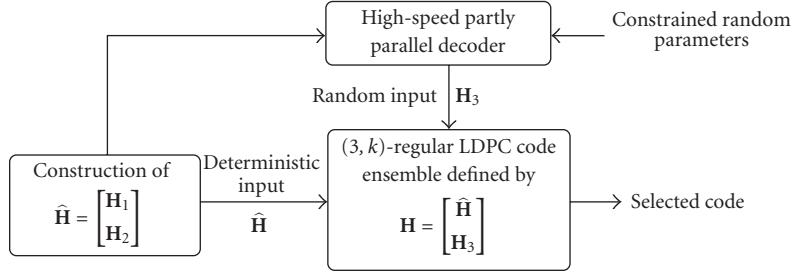


FIGURE 2: Joint design flow diagram.

$\{(i, j) \mid H_{i,j} = 1\}$, compute

$$\alpha_{m,n} = \text{sign}(\gamma_n) \log \left(\frac{1 + e^{-|\gamma_n|}}{1 - e^{-|\gamma_n|}} \right), \quad (1)$$

where

$$\text{sign}(\gamma_n) = \begin{cases} +1, & \gamma_n \geq 0, \\ -1, & \gamma_n < 0. \end{cases} \quad (2)$$

(2) Iterative decoding

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

$$\beta_{m,n} = \log \left(\frac{1 + e^{-\alpha}}{1 - e^{-\alpha}} \right) \prod_{n' \in \mathcal{N}(m) \setminus n} \text{sign}(\alpha_{m,n'}), \quad (3)$$

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

(ii) Vertical (or variable node computation) step: for each $(m, n) \in \{(i, j) \mid H_{i,j} = 1\}$, compute

$$\alpha_{m,n} = \text{sign}(\gamma_{m,n}) \log \left(\frac{1 + e^{-|\gamma_{m,n}|}}{1 - e^{-|\gamma_{m,n}|}} \right), \quad (4)$$

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

$$\lambda_n = \gamma_n + \sum_{m \in \mathcal{M}(n)} \beta_{m,n}. \quad (5)$$

(iii) Decision step:

- (a) perform hard decision on $\{\lambda_1, \dots, \lambda_N\}$ to obtain $\hat{\mathbf{x}} = \{\hat{x}_1, \dots, \hat{x}_N\}$ such that $\hat{x}_n = 0$ if $\lambda_n > 0$ and $\hat{x}_n = 1$ if $\lambda_n \leq 0$;
- (b) if $\mathbf{H} \cdot \hat{\mathbf{x}} = 0$, then algorithm terminates, else go to horizontal step until the preset maximum number of iterations have occurred.

We call $\alpha_{m,n}$ and $\beta_{m,n}$ in the above algorithm *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.

Each decoding iteration can be performed in fully parallel fashion 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.

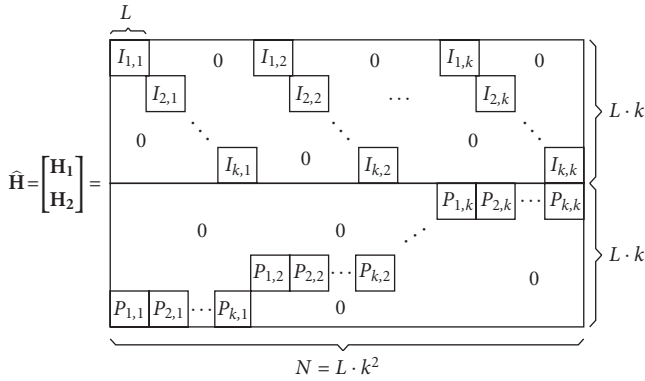
In the partly parallel decoding, the operations of a certain number of check nodes or variable nodes are time-multiplexed, or folded [16], to a single CNU or VNU. For an LDPC code with M check nodes and N variable nodes, if its partly parallel decoder contains M_p CNUs and N_p VNUs, we denote M/M_p as CNU folding factor and 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 Figure 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 girth¹ of 12.
- (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 subcode 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 that the code rate is always $1 - 3/k$.

¹Girth is the length of a shortest cycle in a graph.


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

Construction of $\hat{\mathbf{H}} = [\mathbf{H}_1^T, \mathbf{H}_2^T]^T$

The structure of $\hat{\mathbf{H}}$ is shown in Figure 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, for example, if $L = 5$, $x = 3$, and $y = 4$, we have $u = (x-1) \cdot y \bmod L = 8 \bmod 5 = 3$, then

$$\mathbf{P}_{3,4} = T^3(\mathbf{I}) = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}. \quad (6)$$

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 .

Theorem 1. *If L cannot be factored as $L = a \cdot b$, where $a, b \in \{0, \dots, k-1\}$, 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.

- (i) It contains k^2 memory banks, each one consists of several RAMs to store all the decoding messages associated with L variable nodes.
- (ii) Each memory bank associates with one address generator that is configured by one element in a constrained random integer set \mathcal{R} .
- (iii) It contains a configurable random-like one-dimensional shuffle network \mathcal{S} with the routing complexity scaled by k^2 .
- (iv) 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.
- (v) 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 \mathcal{S} .

4. PARTLY PARALLEL DECODER ARCHITECTURE

In this paper, 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.

- (i) 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.
- (ii) 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].
- (iii) 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 1st and 2nd 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 Figure 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 Figure 3 and the random-like \mathbf{H}_3 is specified by certain configuration parameters of the decoder. To facilitate the description of the

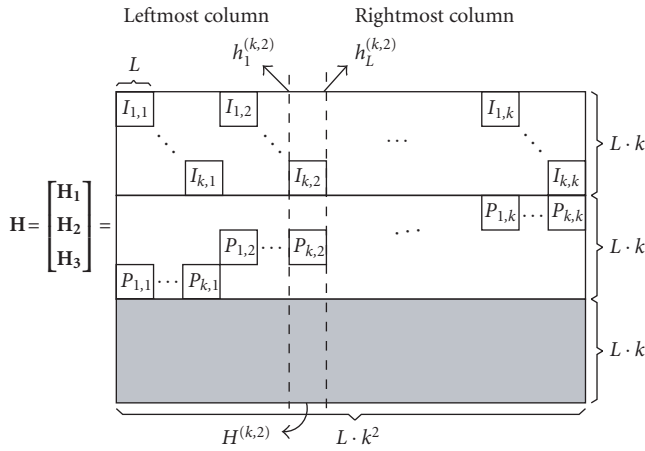


FIGURE 4: The parity-check matrix.

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 $\mathbf{H}_i^{(x,y)}$ 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 Figure 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 $i = 1, \dots, L$ constitute a variable node group $\text{VG}_{x,y}$. 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 5 shows the principal structure of this partly parallel decoder. It mainly contains k^2 PE blocks $\text{PE}_{x,y}$, for $1 \leq x$ and $y \leq k$, three bidirectional shuffle networks π_1 , π_2 , and π_3 , and $3 \cdot k$ CNUs. Each $\text{PE}_{x,y}$ contains one memory bank $\text{RAM}_{s,x,y}$ 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 bidirectional 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 CNU $_{i,j}$, for $j = 1, \dots, k$, 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 first and second 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 five bits and the iterative decoding datapaths of this partly parallel decoder are illustrated in Figure 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 Figure 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 Figure 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 $\text{VG}_{x,y}$. In 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 Figure 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 at CNU $_{i,j}$;
- (3) *modify*: each 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 Figure 7, where CNU is one-stage pipelined. To make this pipelining scheme feasible, we realize each bidirectional I/O connection in the three

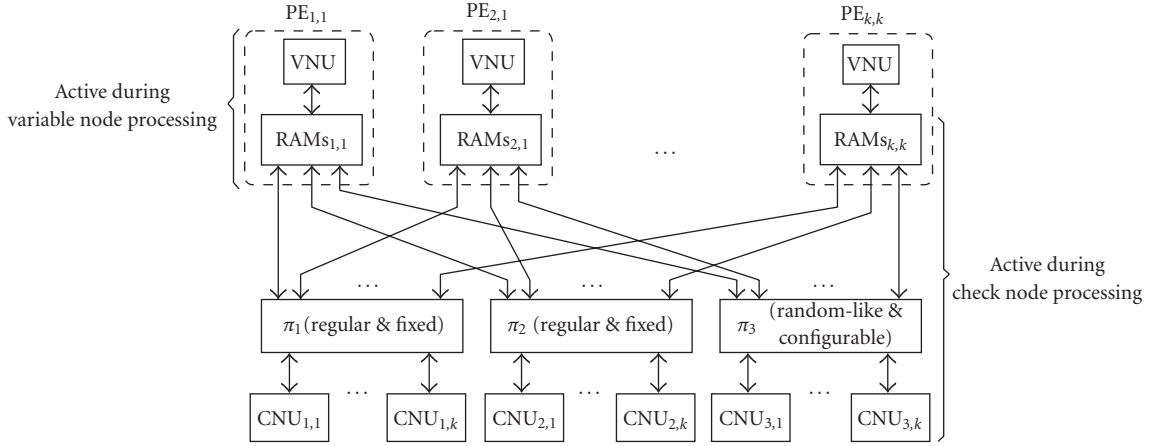


FIGURE 5: The principal (3, k)-regular LDPC code partly parallel decoder structure.

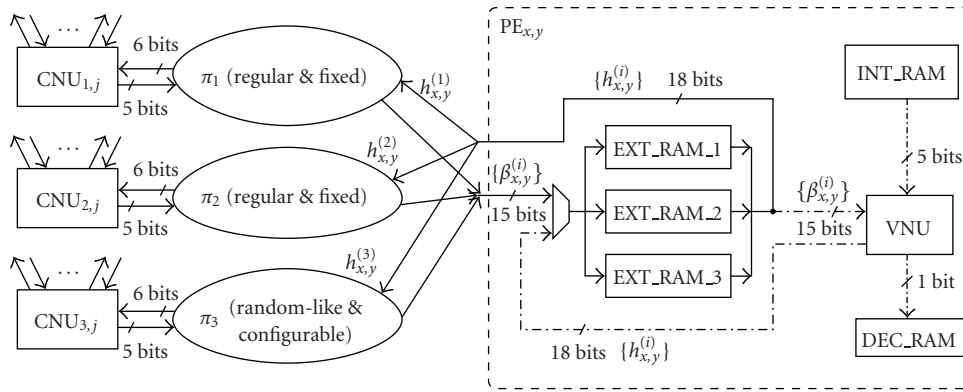


FIGURE 6: Iterative decoding datapaths.

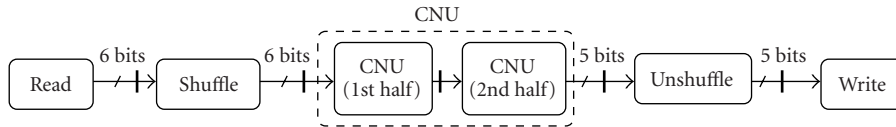


FIGURE 7: Five-stage pipelining of the check node processing datapath.

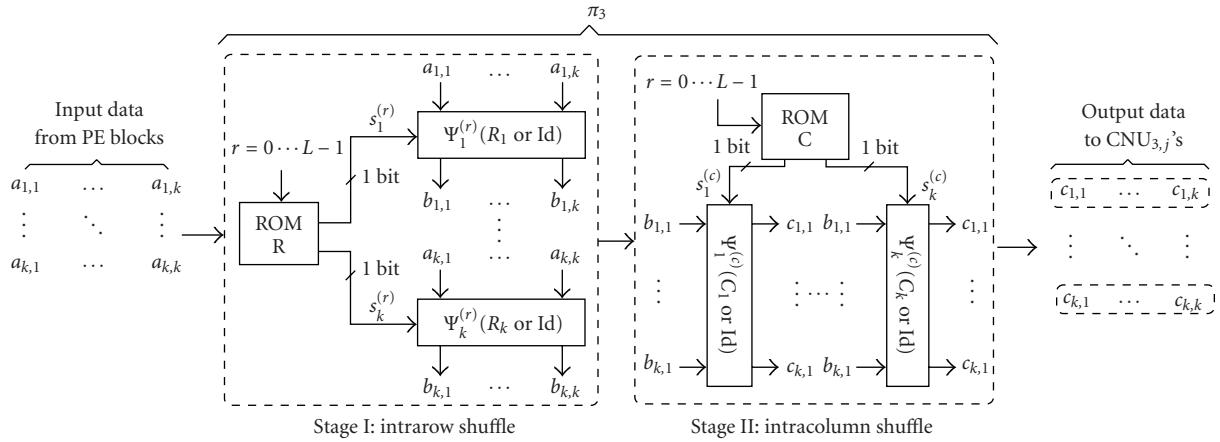
shuffle networks by two distinct sets of wires with opposite directions, which means that the hybrid data from PE blocks to CNUs and the check-to-variable extrinsic messages from CNUs to PE blocks are carried on distinct sets of wires. Compared with sharing one set of wires in time-multiplexed fashion, 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 $AG_{x,y}^{(i)}$ associates with one EXT_RAM_i in each $PE_{x,y}$. In the check node processing, $AG_{x,y}^{(i)}$ 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 ob-

tained 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 $AG_{x,y}^{(i)}$ 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 $AG_{x,y}^{(1)}$ and π_1

The bidirectional shuffle network π_1 and $AG_{x,y}^{(1)}$ 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

FIGURE 8: Forward path of π_3 .

that node $v_d^{(x,y)}$ corresponds to the column $h_i^{(x,y)}$ as illustrated in Figure 4 and the extrinsic messages associated with node $v_d^{(x,y)}$ are always stored at address $d - 1$. Exploiting the explicit structure of \mathbf{H}_1 , we easily obtain the implementation schemes for $AG_{x,y}^{(1)}$ and π_1 as follows:

- (i) each $AG_{x,y}^{(1)}$ is realized as a $\lceil \log_2 L \rceil$ -bit binary counter that is cleared to zero at the beginning of check node processing;
- (ii) the bidirectional shuffle network π_1 connects the k $PE_{x,y}$ with the same x -index to the same CNU.

4.1.2 Implementations of $AG_{x,y}^{(2)}$ and π_2

The bidirectional shuffle network π_2 and $AG_{x,y}^{(2)}$ 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 $AG_{x,y}^{(2)}$ and π_2 as follows:

- (i) each $AG_{x,y}^{(2)}$ 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;
- (ii) the bidirectional shuffle network π_2 connects the k $PE_{x,y}$ with the same y -index to the same CNU.

Notice that the counter load value for each $AG_{x,y}^{(2)}$ 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 $AG_{x,y}^{(3)}$ and π_3

The bidirectional shuffle network π_3 and $AG_{x,y}^{(3)}$ jointly define the connectivity among all the variable nodes and all the check nodes in CG_3 , which is represented by \mathbf{H}_3 as illustrated in Figure 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 $AG_{x,y}^{(i)}$ and π_i , for $i = 1, 2$. However, the implementa-

tions of $AG_{x,y}^{(3)}$ 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} random to some extent, \mathbf{H}_3 should be random-like.

As proposed in [14], to simplify the design process, we separately conceive $AG_{x,y}^{(3)}$ and π_3 in such a way that the implementations of $AG_{x,y}^{(3)}$ and π_3 accomplish the above first and second requirements, respectively.

Implementations of $AG_{x,y}^{(3)}$

We implement each $AG_{x,y}^{(3)}$ 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 , $t_{x,y_1} \neq t_{x,y_2}$, for all $y_1, y_2 \in \{1, \dots, k\}$;
- (2) given y , $t_{x_1,y} - t_{x_2,y} \neq ((x_1 - x_2) \cdot y) \bmod L$, for all $x_1, x_2 \in \{1, \dots, k\}$.

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 $AG_{x,y}^{(3)}$ is realized as a counter, the pattern of shuffle network π_3 cannot be fixed, otherwise the shuffle pattern of π_3 will be regularly repeated 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 configurable to some extent. In this paper, we propose the following concatenated configurable random shuffle network implementation scheme for π_3 .

Figure 8 shows the forward path (from $PE_{x,y}$ to $CNU_{3,j}$) of the bidirectional 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: *intrarow* shuffle and *intracolumn* shuffle. Firstly, the $a_{x,y}$ data block, where each $a_{x,y}$ comes from $PE_{x,y}$, passes an *intrarow* 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 $s^{(r)}$ 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 *intracolumn* 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 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 $CNU_{3,j}$. To realize the bidirectional shuffle, we only need to implement each configurable shuffle network $\Psi_x^{(r)}$ and $\Psi_y^{(c)}$ as bidirectional so that π_3 can *unshuffle* the k^2 data backward from $CNU_{3,j}$ to $PE_{x,y}$ 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 random-like and change each clock cycle, we only need to randomly generate the control words $s_x^{(r)}$ 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 $PE_{x,y}$. 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 $PE_{x,y}$. 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

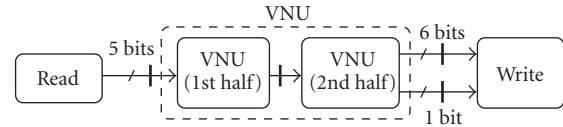


FIGURE 9: Three-stage pipelining of the variable node processing datapath.

hard decisions. As shown in Figure 6, we may outline the datapath loop in variable node processing as follows:

- (1) *read*: in each $PE_{x,y}$, 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 .

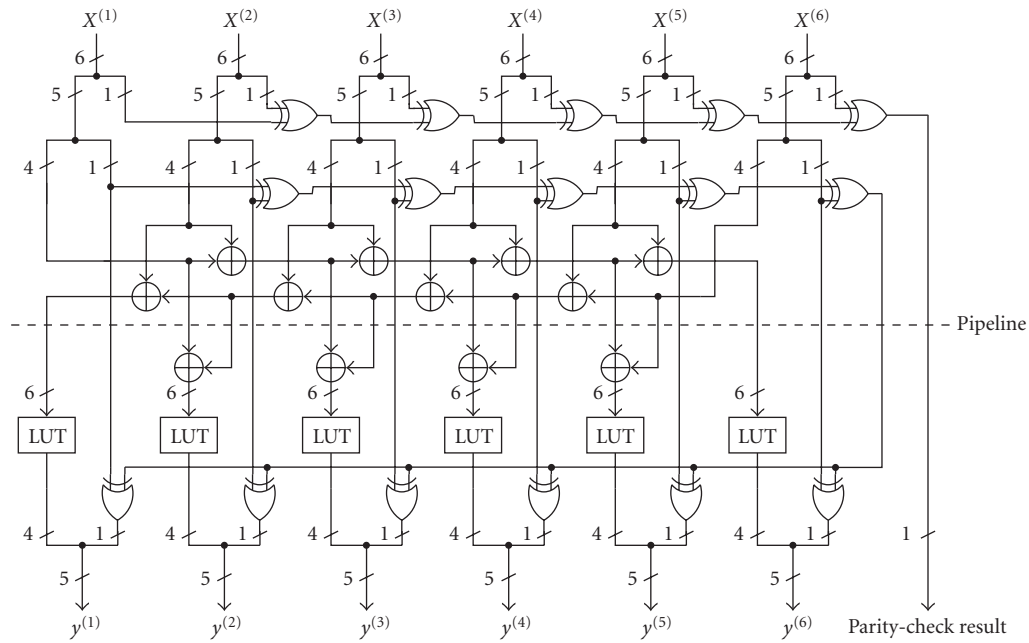
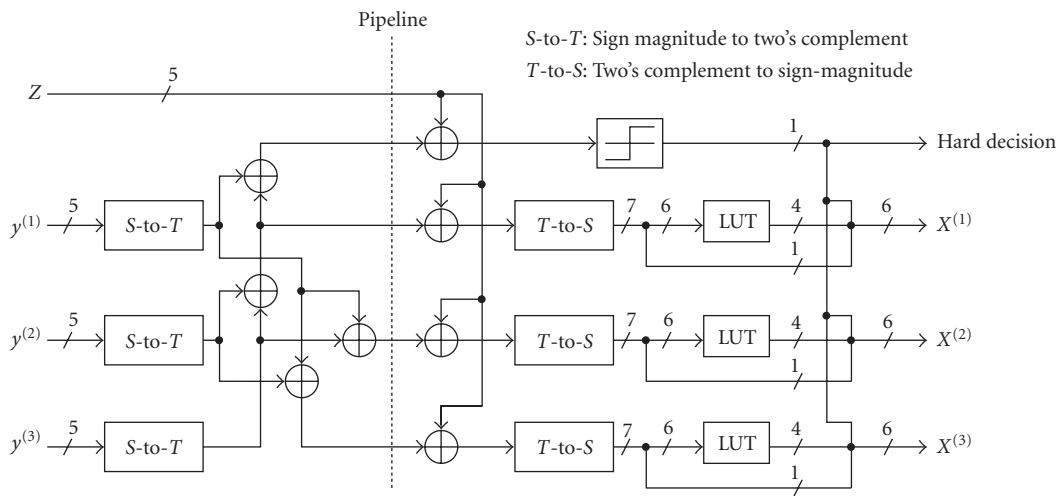
The forward path from memory to VNU and backward path from VNU to memory are implemented by distinct sets of wires and the entire read-modify-write datapath loop is pipelined by three-stage pipelining as illustrated in Figure 9.

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. Figure 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 (3). The function $f(x) = \log((1 + e^{-|x|})/(1 - e^{-|x|}))$ is realized by the LUT (lookup 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. Figure 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

FIGURE 10: Architecture for CNU with $k = 6$.FIGURE 11: Architecture for VNU with $j = 3$.

generates three 5-bit variable-to-check extrinsic messages and 1-bit hard decision according to (4) and (5), respectively. To enable each CNU to receive the hard decisions to perform parity check as described 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 Figure 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 $f(x) = \log((1 + e^{-|x|})/(1 - e^{-|x|}))$, each data is converted back to the 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

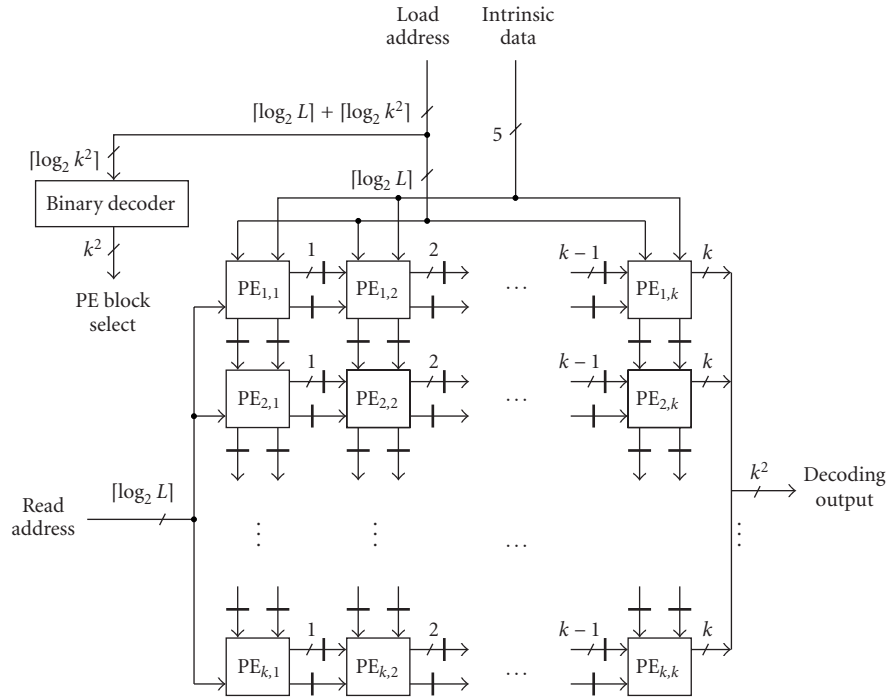


FIGURE 12: Data input/output structure.

the k^2 PE blocks. To minimize the routing overhead, we develop a square-shaped floor planning for PE blocks as illustrated in Figure 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 Figure 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 INT_RAM. As shown in Figure 12, the primary intrinsic data and load address input directly connect to the k PE blocks $PE_{1,y}$ for $1 \leq y \leq k$, and from each $PE_{x,y}$ the intrinsic data and load address are delivered to the adjacent PE block $PE_{x+1,y}$ 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 Figure 12, the primary $\lceil \log_2 L \rceil$ -bit read address input directly connects to the k PE blocks $PE_{x,1}$ for $1 \leq x \leq k$, and from each $PE_{x,y}$ the read address are delivered to the adjacent block $PE_{x,y+1}$ in pipelined fashion. Based on its input read address, each PE block outputs 1-bit hard decision per clock cycle. Therefore, as illustrated in Figure 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 $AG_{x,y}^{(3)}$ 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 Figure 12. Notice that such placement

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%

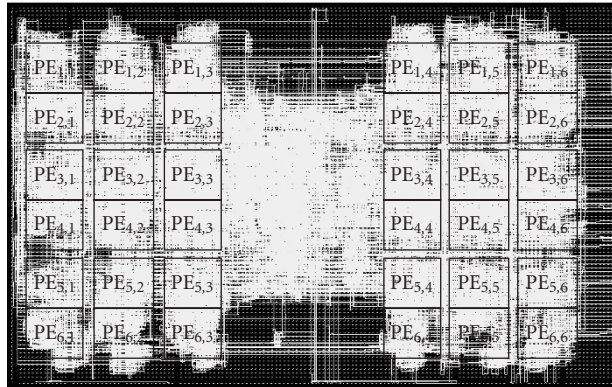


FIGURE 13: The placed and routed decoder implementation.

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_RAM_i . 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 $2 \times W$ 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, that is, 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.

This decoder was described in the VHDL (hardware description language) and SYNOPSIS 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. Figure 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. Figure 14 shows the corresponding performance over AWGN channel with $s = 18$, including the BER, 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 paper, 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 2 dB 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.

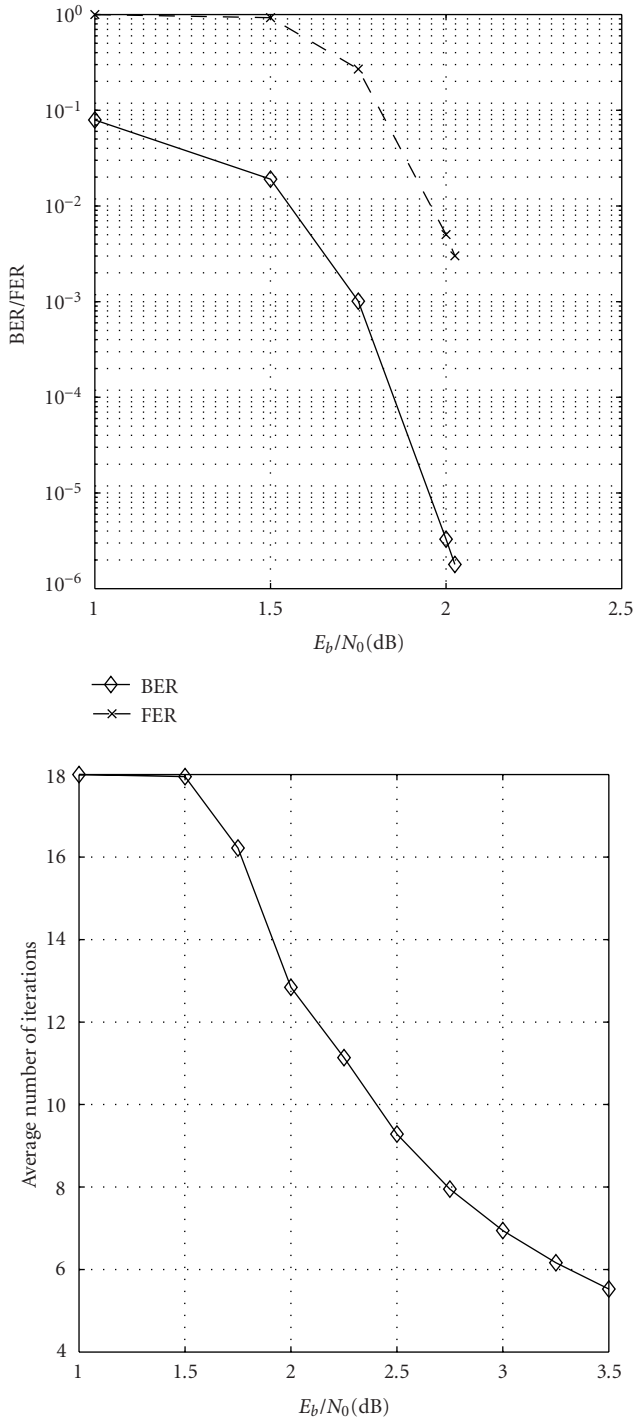


FIGURE 14: Simulation results on BER, FER and the average iteration numbers.

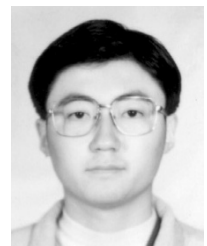
REFERENCES

- [1] R. G. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. IT-8, no. 1, pp. 21–28, 1962.
- [2] R. G. Gallager, *Low-Density Parity-Check Codes*, MIT Press, Cambridge, Mass, USA, 1963.
- [3] D. J. C. MacKay, "Good error-correcting codes based on very

sparse matrices," *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, 1999.

- [4] M. C. Davey and D. J. C. MacKay, "Low-density parity check codes over $GF(q)$," *IEEE Communications Letters*, vol. 2, no. 6, pp. 165–167, 1998.
- [5] M. Luby, M. Mitzenmacher, M. Shokrollahi, and D. Spielman, "Improved low-density parity-check codes using irregular graphs and belief propagation," in *Proc. IEEE International Symposium on Information Theory*, p. 117, Cambridge, Mass, USA, August 1998.
- [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, no. 2, pp. 599–618, 2001.
- [7] T. Richardson, M. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 619–637, 2001.
- [8] S.-Y. Chung, T. Richardson, and R. Urbanke, "Analysis of sum-product decoding of low-density parity-check codes using a Gaussian approximation," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 657–670, 2001.
- [9] M. Luby, M. Mitzenmacher, M. Shokrollahi, and D. A. Spielman, "Improved low-density parity-check codes using irregular graphs," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 585–598, 2001.
- [10] S.-Y. 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, no. 2, pp. 58–60, 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, no. 7, pp. 2696–2710, 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, no. 3, pp. 404–412, 2002.
- [13] E. Boutillon, J. Castura, and F. R. Kschischang, "Decoder-first code design," in *Proc. 2nd International Symposium on Turbo Codes and Related Topics*, pp. 459–462, Brest, France, September 2000.
- [14] T. Zhang and K. K. Parhi, "VLSI implementation-oriented (3, k)-regular low-density parity-check codes," in *IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 25–36, Antwerp, Belgium, September 2001.
- [15] M. Chiani, A. Conti, and A. Ventura, "Evaluation of low-density parity-check codes over block fading channels," in *Proc. IEEE International Conference on Communications*, pp. 1183–1187, New Orleans, La, USA, June 2000.
- [16] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, New York, USA, 1999.

Tong Zhang received his B.S. and M.S. degrees in electrical engineering from the Xian Jiaotong University, Xian, China, in 1995 and 1998, respectively. He received the Ph.D. degree in electrical engineering from the University of Minnesota in 2002. Currently, he is an Assistant Professor in Electrical, Computer, and Systems Engineering Department at Rensselaer Polytechnic Institute. His current research interests include design of VLSI architectures and circuits for digital signal processing and communication systems, with the emphasis on error-correcting coding and multimedia processing.



Keshab K. Parhi is a Distinguished McKnight University Professor in the Department of Electrical and Computer Engineering at the University of Minnesota, Minneapolis. He was a Visiting Professor at Delft University and Lund University, a Visiting Researcher at NEC Corporation, Japan, (as a National Science Foundation Japan Fellow), and a Technical Director DSP Systems at Broadcom Corp. Dr. Parhi's



research interests have spanned the areas of VLSI architectures for digital signal and image processing, adaptive digital filters and equalizers, error control coders, cryptography architectures, high-level architecture transformations and synthesis, low-power digital systems, and computer arithmetic. He has published over 350 papers in these areas, authored the widely used textbook *VLSI Digital Signal Processing Systems* (Wiley, 1999) and coedited the reference book *Digital Signal Processing for Multimedia Digital Signal Processing Systems* (Wiley, 1999). He has received numerous best paper awards including the most recent 2001 IEEE WRG Baker Prize Paper Award. He is a Fellow of IEEE, and the recipient of a Golden Jubilee medal from the IEEE Circuits and Systems Society in 1999. He is the recipient of the 2003 IEEE Kiyo Tomiyasu Technical Field Award.