Memory System Optimization for FPGA-Based Implementation of Quasi-Cyclic LDPC Codes Decoders

Xiaoheng Chen, Jingyu Kang, Shu Lin, *Life Fellow, IEEE*, and Venkatesh Akella

***Abstract*—Designers are increasingly relying on field-programmable gate array (FPGA)-based emulation to evaluate the performance of low-density parity-check (LDPC) codes empirically down to bit-error rates of 10^{-12} and below. This requires decoding architectures that can take advantage of the unique characteristics of a modern FPGA to maximize the decoding throughput. This paper presents two specific optimizations called *vectorization* and *folding* to take advantage of the configurable data-width and depth of embedded memory in an FPGA to improve the throughput of a decoder for quasi-cyclic LDPC codes. With folding it is shown that quasi-cyclic LDPC codes with a very large number of circulants can be implemented on FPGAs with a small number of embedded memory blocks. A synthesis tool called QCSyn is described, which takes the H matrix of a quasi-cyclic LDPC code and the resource characteristics of an FPGA and automatically synthesizes a vector or folded architecture that maximizes the decoding throughput for the code on the given FPGA by selecting the appropriate degree of folding and/or vectorization. This helps not only in reducing the design time to create a decoder but also in quickly retargeting the implementation to a different (perhaps new) FPGA or a different emulation board.**

***Index Terms*—Alignment, field programmable logic array (FPGA), folding, low-density parity-check (LDPC) decoder, memory system optimization, normalized min-sum algorithm, quasi-cyclic low-density parity-check (QC-LDPC) codes, very large scale integration (VLSI) implementation.**

I. Introduction

LOW-DENSITY parity-check (LDPC) codes, discovered by Gallager in 1962 [1], were rediscovered and shown to approach Shannon capacity in the late 1990s. Today LDPC codes are being considered for a wide variety of emerging applications such as high-density flash memory, satellite broadcasting, WiFi and mobile WiMAX. Rapid performance evaluation of LDPC codes is very desirable to design better codes for these applications. LDPC codes are decoded iteratively using the sum-product algorithm [1], [2] and its variations such as the normalized min-sum algorithm (NMSA) [3]. Memory bandwidth is the key performance limiting factor in the hardware realization of a LDPC decoder. For example, consider the implementation of the (8176,7156) LDPC code [4], [5] used in NASA LANDSAT and cruise exploration shuttle missions. The underlying Tanner graph of this code has 32 704 edges, which means approximately $(32\,704 \times 4 = 131\,086)$ messages have to be read and written per iteration to decode a codeword (the factor 4 is needed because each iteration consists of two phases called check node and variable node processing, and each phase requires reading and writing messages corresponding to each edge). Clearly, the decoding throughput of a LDPC decoder is limited by this memory bandwidth requirement. Modern field-programmable gate arrays (FPGAs) from Xilinx and Altera have a large number of embedded memory blocks, typically in the range of a few hundreds. In a Xilinx FPGA, these embedded memory blocks are known as block RAMs and in Altera FPGA they are called embedded array blocks. We will focus on a Xilinx FPGA in the rest of the paper, though the ideas described in this paper can easily be migrated to an Altera FPGA. Block RAMs are dual-ported and can be accessed independently in a single cycle (typically 400 to 500 MHz in a Xilinx Virtex FPGA), that results in an enormous internal memory bandwidth, which if exploited properly can result in very high decoding throughput. Consequently, in the past several years, there has been a great deal of interest in developing FPGA-based architectures for decoding LDPC codes [6]–[11].

However, most of the FPGA-oriented implementations reported in literature fail to take full advantage of the fact that the aspect ratio of the block RAMs is *configurable*—i.e., both the word size and the number of locations in the memory or depth of the memory can be selected by the designer. For example, each 18 Kb block RAM in a Virtex-4 FPGA can be configured to operate as a 512x36, 1Kx18, 2Kx9, 4Kx4, 8Kx2, or 16Kx1 memory block (in a 512x36 configuration, 36 bits can be read or written simultaneously, whereas in a 1Kx18 configuration 18 bits can be read or written simultaneously). In fixed-point implementation of LDPC codes, the messages are typically 6 to 8 bits [11]. Most FPGA implementations store one message per block RAM word. So, typically they utilize 8/36 or roughly 22% of the available memory bandwidth, or in other words, roughly 78% of the available memory bandwidth is not being used.

We propose two architectural techniques to increase the throughput of an FPGA-based implementation of a LDPC decoder. The first technique, called *vectorization*, exploits the configurable width of the block RAM; and the second, called *folding*, takes advantage of the configurable depth of the block RAM. Vectorization allows the packing of multiple messages

Manuscript received September 14, 2009; revised February 17, 2010; accepted June 11, 2010. Date of publication August 03, 2010; date of current version December 30, 2010. This work was supported in part by the National Science Foundation under Grant CCF-0727478, in part by NASA under Grants NNX07AK50G and NNX09AI21G, and in part by gift grants from Intel and Northrop Grumman Space Technology. This paper was recommended by Associate Editor L. He.

The authors are with the Department of Electrical and Computer Engineering, University of California, Davis, CA 95616 USA (e-mail: xhchen@ucdavis.edu; jykang@ucdavis.edu; shulin@ucdavis.edu; akella@ucdavis.edu).

Digital Object Identifier 10.1109/TCSI.2010.2055250

into the same physical word by exploiting the wider word configuration of a block RAM. However, it introduces some key challenges. Given that each memory access delivers multiple messages, duplication of the functional units to process the messages concurrently, and data alignment hardware to route the messages to the appropriate functional unit is required. In an FPGA the total number of resources (flip-flops, lookup tables, and interconnect) are fixed. So vectorization could result in an implementation that either might not fit on a given FPGA or could result in a lower clock frequency due to the alignment logic and interconnect complexity. As a result, careless vectorization might result in performance that is worse than a scalar implementation. The resource utilization and the resultant clock frequency depend on the structure of the code being implemented. To address this challenge, we developed a semiautomated tool called QCSyn that automatically synthesizes a vector architecture for a given quasi-cyclic code. This allows the designer to pick the correct degree of vectorization and pipeline depth for functional units, for a target FPGA and throughput requirements.

Though the idea of packing multiple messages in a single memory word has been reported in literature [12], [13], the key contribution of this paper is a configurable vector decoder architecture for quasi-cyclic LDPC codes that can be customized to a given code and a given FPGA (which represents a set of resource constraints) by choosing the appropriate degree of pipelining of the functional units and automating the generation of the data-alignment logic. As a result, the proposed approach reduces the *time to design* a custom vector architecture for a given code and given FPGA platform. We also extend the overlapped message passing algorithm [8] to a vector architecture to further improve the performance of an FPGA-based vector decoder, which has not been addressed by prior research.

Folding is an optimization that allows implementing large (complex) LDPC codes on FPGAs with limited number of block RAMs. Even the largest FPGA that is commercially available (a Virtex-5) has only around hundreds of block RAMs, so approaches such as the partially parallel implementation described in [9]–[11] that maps each circulant to a separate block RAM will not work for codes with a very large number of circulants. In other words, existing approaches such as [9]–[11] implicitly assume that the number of block RAMs is always greater than the number of submatrices (circulants) in the LDPC code, which is not always possible, especially with new applications such as DVB-S2 etc which require codes with a very large number of circulants. Folding is essentially a memory virtualization technique that maps messages corresponding to multiple submatrices in the same physical block RAM. This allows us to implement codes with a very large number of edges in their underlying Tanner graph on an FPGA using the partially parallel architectural template.

In summary, the specific contributions of this paper are threefold.

1) We propose an extension to the partially parallel decoder architecture to incorporate vector processing to take advantage of the configurable width of the block RAMs. The words in the block RAM are treated as short vectors and suitable functional units and data alignment structures are created to implement a customized vector processor for a given quasi-cyclic code. The overlapped message passing algorithm [8], [9] is extended to handle vector processing.
2) A memory virtualization technique called folding is developed that allows messages from different submatrices of quasi-cyclic code to share the physical block RAM. This allows us to implement very large codes, i.e., codes with a very large number of submatrices (and hence edges) efficiently on a given FPGA.
3) A semiautomated tool called QCSyn is developed to explore the design space of vectorization and folding to meet the resource and throughput targets for a given code and application on a given FPGA. An emulation platform has been realized using the DN8000K10PSX board from Dini Group and implementation results from the actual hardware are reported.

The rest of the paper is organized as follows. Section II provides the necessary background on LDPC codes and the message passing algorithm to decode LDPC codes. Section III describes the vector decoder architecture and the vector overlapped message passing algorithm. Section IV presents the folding technique and examples. The results and discussion for each method are presented in the corresponding section. Section V describes the QCSyn tool and the FPGA emulation platform that was developed and Section VI presents the comparison of the proposed methods with related approaches in literature.

II. Low-Density Parity-Check Codes

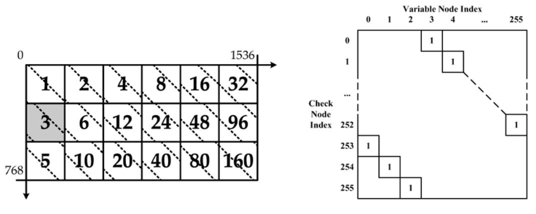
In this section we provide the background to understand the techniques proposed in this paper. We start with an overview of LDPC codes and the definition of quasi-cyclic LDPC codes. Next, we describe an iterative message passing algorithm called normalized min-sum algorithm (NMSA) [3], which is the algorithm realized by the various architectures proposed in this paper. We will end the section with a brief overview of the partially parallel decoder architecture that serves as the baseline for the implementations reported in this paper.

A. Classification of LDPC Codes

A binary LDPC code $\mathcal{C}$ of length n is given by the null space of an $J \times n$ sparse parity-check matrix $\mathbf{H} = [\mathbf{h}_{i,j}]$ over GF(2). If each column has constant weight γ (the number of 1-entries in a column) and each row has constant weight ρ (the number of 1-entries in a row), then the LDPC code C is referred to as a (γ, ρ)-regular LDPC code. If the columns and/or rows of the parity-check matrix $\mathbf{H}$ have multiple weights, then the null space of $\mathbf{H}$ gives an irregular LDPC code. If $\mathbf{H}$ is an array of sparse circulants of the same size over GF(2), then the null space of $\mathbf{H}$ gives a quasi-cyclic (QC)-LDPC codes. If $\mathbf{H}$ consists of a single sparse circulant or a column of sparse circulants, then the null space of $\mathbf{H}$ gives a cyclic LDPC code.

A binary n-tuple $\mathbf{v} = (v_0, v_1, \ldots, v_{n-1})$ is a codeword in $\mathcal{C}$ if and only if $\mathbf{v}\mathbf{H}^T = 0$. For $0 \le i < J$, let $\mathbf{h}_i = (h_{i,0}, h_{i,1}, \ldots, h_{i,n-1})$ be the ith row of $\mathbf{H}$. Then the condition $\mathbf{v}\mathbf{H}^T = 0$ gives the following set of m constraints on the n code bits of a codeword $\mathbf{v}$ in $\mathcal{C}$:

$$v_0 h_{i,0} + v_1 h_{i,1} + \cdots + v_{n-1} h_{i,n-1} = 0 \qquad (1)$$



(a) (b)

Fig. 1. Parity check Matrix for a (3,6)-regular QC-LDPC code. There are 18 circulant permutation matrices (or CPMs) marked 1, 2, 4, 8, 16, 32, 3, 6, 12 ... 80, 160. The number denotes the offset for the CPM. For example, the circulant marked labeled 3 is shown in more detail in (b). Each circulant is a 256×256 matrix. The offset is the position of the nonzero entry in the first row of the circulant. (a) A (3,6)-regular QC-LDPC code $(m = 256)$. (b) A CPM with $m = 256$, offset $\Delta = 3$.

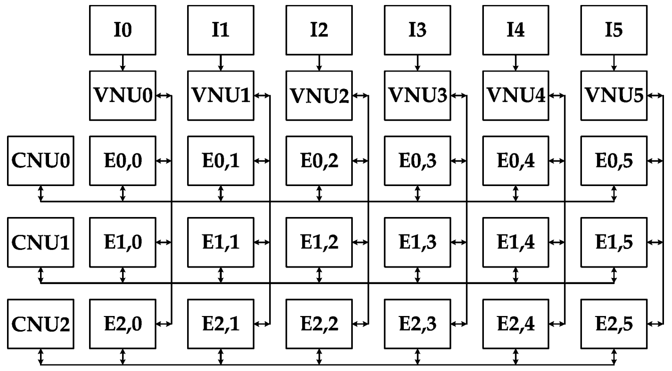


Fig. 2. Partially parallel decoder architecture for (3,6)-regular QC-LDPC codes with $m = 256$, adapted from [9], $\mathbf{I}_j (0 \le j < 6)$ denotes the jth IMEM, $\mathbf{E}_{i,j} (0 \le i < 3, 0 \le j < 6)$ denotes the EMEM on the ith row and on the jth column.

for $0 \le i < m$, where the operations in the sum are modulo-2 operations. Each sum is called a check-sum (check constraint).

Often an LDPC code $\mathcal{C}$ is represented graphically by a bipartite graph which consists of two disjoint sets of nodes. Nodes in one set represent the code bits and are called variable nodes (VNs), and the nodes in the other set represent the check-sums that the code bits must satisfy and are called check nodes (CNs). Label the VNs from 0 to $n-1$ and the CNs from 0 to $J-1$. The ith CN is connected to the jth VN by an edge if and only if $h_{i,j} = 1$. The VNs connected to the ith CN simply correspond to the code bits that are contained in the ith check-sum. The CNs connected to the jth VN simply correspond to the check sums that contain the jth VN.

In terms of encoding and decoding implementation, the most advantageous structure of an LDPC code is the QC structure. Commonly, in most of the proposed constructions of QC-LDPC codes, the parity-check matrix of a QC-LDPC code is given as a $\gamma \times \rho$ array (or block) of circulants or circulant permutation matrices (CPMs) and/or zero matrices of the same size, say $m \times m$, as follows:

$$\mathbf{H} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \cdots & \mathbf{A}_{0,\rho-1} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \cdots & \mathbf{A}_{1,\rho-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{\gamma-1,0} & \mathbf{A}_{\gamma-1,1} & \cdots & \mathbf{A}_{\gamma-1,\rho-1} \end{bmatrix}. \quad (2)$$

Then $\mathbf{H}$ is a $\gamma m \times \rho m$ matrix over GF(2). The QC-LDPC code given by the null space of the $\mathbf{H}$ matrix has length ρm and rate at least $1 - (\gamma/\rho)$. Many standard codes for communication systems are QC-LDPC codes.

Fig. 1(a) shows the code structure of a (3,6) regular QC-LDPC codes. We denote this code as $\mathcal{C}_1$. The $\mathbf{H}$ matrix has 3 block rows and 6 block columns for a total of 18 circulant permutation matrices. Each circulant permutation matrix (CPM) is 256×256 with a certain offset, which denotes the position of the nonzero entry in the first row of the matrix. Details of a CPM with offset equal to 3 is shown in Fig. 1(b).

B. Normalized Min-Sum Algorithm

The NMSA [3] is a reduced-complexity approximation of the SPA. With carefully chosen normalization factor, the NMSA performs as good as SPA. Let $\mathbf{v}$ be transmitted over the binary-input AWGN channel with two-sided power spectral density $N_0/2$. Assume transmission using BPSK signaling with unit energy per signal. Then the codeword $\mathbf{v}$ is mapped into a sequence of BPSK signals $(1-2v_0, 1-2v_1, \ldots, 1-2v_{n-1})$ for transmission. Suppose $\mathbf{v}$ is transmitted. Let $\mathbf{L} = (L_0, L_1, \ldots, L_{n-1})$ be the received *soft-decision* sequence, where $L_j = (1-2v_j) + x_j$ is the intrinsic channel reliability value of the jth code bit, x_j is a Gaussian random variable with zero-mean and variance $N_0/2$.

For $0 \le i < J$ and $0 \le j < n$, we define $N_i = \{j : 0 \le j < n, h_{i,j} = 1\}$, and $J_j = \{i : 0 \le i < J, h_{i,j} = 1\}$. Let $K_{\max}$ be the maximum number of iterations to be performed. For $0 \le k \le K_{\max}$, let $\mathbf{z}^{(k)} = (z_0^{(k)}, z_1^{(k)}, \ldots, z_{n-1}^{(k)})$ be the hard decision vector generated in the kth decoding iteration, $L_{i \to j}^{(k)}$ be extrinsic message passed from the ith CN to the jth VN, $L_{i \leftarrow j}^{(k)}$ be the extrinsic message passed from the jth VN to the ith CN, and $L_j^{(k)}$ be the reliability value of the jth code bit. The NMSA can be formulated as follows:

1) Initialization: Set $k = 0$, $\mathbf{z}^{(0)} = \mathbf{z}$ and the maximum number of iterations to $K_{\max}$. For all j, set $L_j^{(0)} = L_j$, set $L_{j \to i}^{(0)} = L_j$ when $h_{ij} = 1$.

Step 1): Parity check: Compute the syndrome $\mathbf{z}^{(k)}\mathbf{H}^T$ of $\mathbf{z}^{(k)}$. If $\mathbf{z}^{(k)}\mathbf{H}^T = \mathbf{0}$, stop decoding and output $\mathbf{z}^{(k)}$ as the decoded codeword; otherwise go to Step 2).

Step 2): If $k = K_{\max}$, stop decoding and declare a decoding failure; otherwise, go to Step 3).

Step 3): CNs update: Compute the message

$$L_{i \to j}^{(k)} = \alpha \left(\prod_{j' \in N_i \setminus j} \operatorname{sign}\left(L_{i \leftarrow j'}^{(k)}\right) \right) \left(\min_{j' \in N_i \setminus j} \left| L_{i \leftarrow j'}^{(k)} \right| \right) \quad (3)$$

where $0 < \alpha < 1$ is the normalization factor. Pass messages from CNs to VNs.

Step 4): VNs update: $k \leftarrow k + 1$. Compute the message

$$L_{i \leftarrow j}^{(k)} = L_j + \sum_{i' \in J_j \setminus j} L_{i' \to j}^{(k-1)} \quad (4)$$

and update the reliability of each received bit by

$$L_j^{(k)} = L_j + \sum_{i' \in J_j} L_{i' \to j}^{(k-1)}. \quad (5)$$

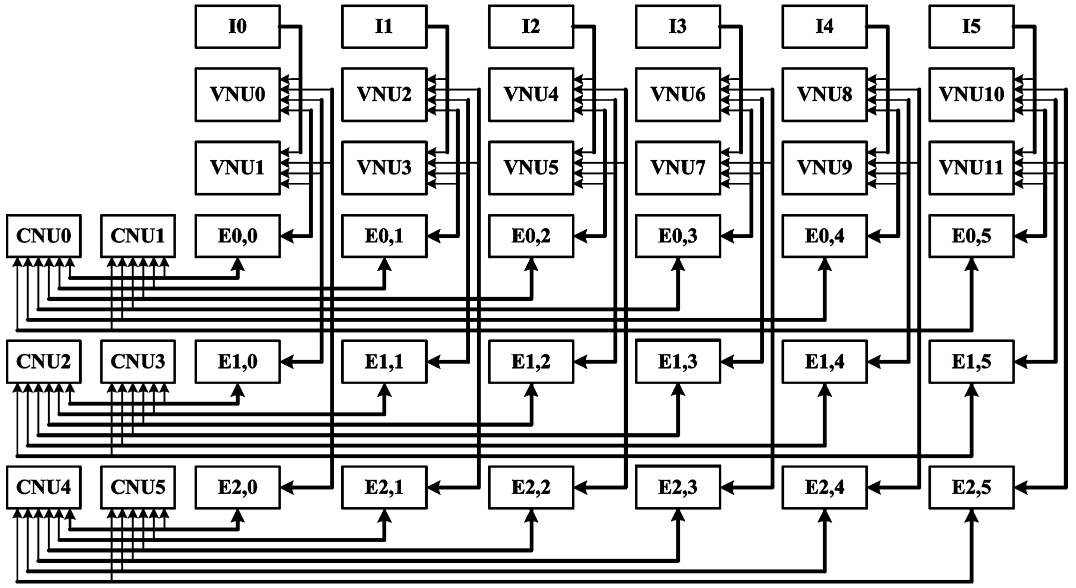


Fig. 3. A vector decoder for code $\mathcal{C}_1$ when $K = 2, \mathbf{I}_j (0 \le j < 6)$ denotes the jth IMEM, $\mathbf{E}_{i,j} (0 \le i < 3, 0 \le j < 6)$ denotes the EMEM on the ith row and on the jth column.

For $0 \le j < n$, make the following hard-decision: 1) $z_j^{(k)} = 0$, if $L_j^{(k)} \ge 0$; 2) $z_j^{(k)} = 1$, if $R_j^{(k)} < 0$. Form a new received vector $\mathbf{z}^{(k)} = \left(z_0^{(k)}, z_1^{(k)}, \ldots, z_{n-1}^{(k)}\right)$. Go to Step 1).

The partially parallel architecture proposed by Chen and Parhi in [9] is an elegant method to realize quasi-cyclic LDPC codes. For (γ, ρ)-regular QC-LDPC code, this approach uses γ ρ-input CNUs (check node units), γ ρ-input VNUs (variable node units), ρ intrinsic message memories (IMEM) with each storing m intrinsic messages, and $\gamma\rho$ extrinsic message memories (EMEM) with each storing m extrinsic messages and m hard decision bits. The partially parallel decoder for $\mathcal{C}_1$ is shown in Fig. 2.

Functional unit CNU_i, $0 \le i < \gamma$ performs the CN update and tentative decoding [Step 1)–3) in NMSA] for the ith block row, while functional unit VNU_j, $0 \le j < \rho$ performs the VN update [Step 4) in NMSA]. $\mathbf{I}_j (0 \le j < \rho)$ denotes the IMEM of the jth block column, which stores the received intrinsic message. $\mathbf{E}_{i,j} (0 \le i < \gamma, 0 \le j < \rho)$ denotes the EMEM which stores the messages passed between the ith CNs and the jth VNs and the hard decision bits corresponded to the jth VNs. The hard decision bit is computed by VNUs and appended at the head of the variable-to-check messages. There are ρ IMEMs for a (γ, ρ)-regular QC-LDPC codes. Every message is stored in one memory word of the block RAM. Thus we denote the conventional partially parallel decoder architecture as a *scalar* decoder to contrast it with the vector decoder proposed in Section III. In [9], the authors propose a simple method to take advantage the quasi-cyclic nature of the codes to improve the decoding throughput by *overlapping* Step 1)–3) and Step 4) in the NMSA (called overlapped message passing) as shown in Fig. 6(c). This requires the computation of a critical parameter called the *waiting time* (see [8] for a detailed explanation and algorithm), which determines the exact time when the overlapping message passing can begin. In the limit, overlapped message passing can increase the throughput of the decoder over the baseline approach by a factor of 2.

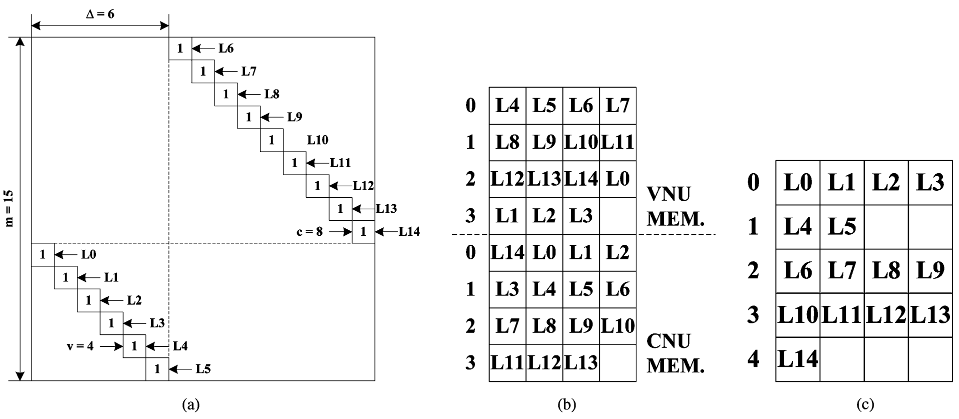
III. Vector Decoder Architecture for QC-LDPC Codes

Vector decoder architecture overcomes the limitation of the scalar decoder (described in the previous section) by packing multiple messages in the same memory word. As noted before, this is possible because, block RAMs can be configured into different aspect rations. For the NMSA, the intrinsic and the extrinsic messages are usually 6–8 bit wide, thus up to six 6-bit messages can be packed in one memory word in the 512x36 block RAM configuration. We define the number of messages packed into one memory word as K. The vector decoder for a (γ, ρ)-regular code requires $K\gamma$ CNUs, $K\rho$ VNUs, and $(\gamma + 1) \times \rho$ block RAMs for intrinsic and extrinsic memories. Fig. 3 shows a vector decoder for the code $\mathcal{C}_1$ when $K = 2$.

Potentially, the throughput of a vector decoder can be K times that of a scalar decoder, given that there are K times more functional units operating simultaneously. However, without proper data packing scheme, memory access conflicts will be caused since multiple messages are accesses per cycle. Besides, efficient message alignment units are required so that the additional logic incurred would be reduced. The techniques to overcome these challenges is described next.

A. Message Packing and Alignment

In a scalar decoder, one full block RAM memory word is used to store a message. Messages are packed in column major order, i.e., the message in the ith CPM column is packed in $L[i]$, where $L[0 \ldots m-1]$ is a one-dimensional array that represents the storage of the CPM in the block RAM. $L[i]$ stores CN update message during Step 3) of the execution and VN update message during Step 4), i.e., the extrinsic messages are updated in place. For example, in the scalar implementation of the partially parallel decoder proposed in [9] (and shown in Fig. 2), the CNU starts from the cth CPM row, i.e., in the order of $L[(c+\Delta) \bmod m], L[(c+\Delta+1) \bmod m], \ldots, L[m-1], \ldots, L[(c+\Delta-1) \bmod m]$. The VNU starts from the vth CPM column, i.e., in the order of $L[v], L[(v+1) \bmod m], \ldots, L[m-$



0	L4	L5	L6	L7
1	L8	L9	L10	L11
2	L12	L13	L14	L0
3	L1	L2	L3	
0	L14	L0	L1	L2
1	L3	L4	L5	L6
2	L7	L8	L9	L10
3	L11	L12	L13	

0	L0	L1	L2	L3
1	L4	L5		
2	L6	L7	L8	L9
3	L10	L11	L12	L13
4	L14			

Fig. 4. Illustration of message packing for a CPM with $m = 15$ and $\Delta = 6$ when vector size $K = 4$. (a) Example CPM (b) Packing scheme proposed in this paper (c) Wang's [12] Packing Scheme.

$1], \ldots, L[(v-1) \bmod m]$. In overlapped message passing, c and v could be non zero, as the processing could start at any arbitrary message.

In a vector decoder, each block RAM location holds multiple messages. Memory conflicts could arise if the CNU and VNU try to access the same location simultaneously. The key challenges with vector decoding is to reduce the potential for such conflicts and to ensure that the overhead of resolving these conflicts through alignment units does not increase the complexity of the decoder and limit scalability and clock frequency. We address this challenge with a combination of three techniques that are described next. *First*, we use double buffering, i.e., the messages are replicated for CNU and VNU access, so that they are stored in different ways to match the access pattern of the CNU and VNU processing. Though it doubles the amount of memory needed for the storage, it does not increase the number of block RAMs necessary, because we use the same block RAM to store both CNU and VNU memory. This works because typically the CPM sizes are much smaller than the depth of the block RAMs in an FPGA. *Second*, we develop a new packing strategy that not only uses the block RAMs efficiently but also reduces the potential for conflicts. *Third* we propose a sequential alignment unit and its implementation to demonstrate that the alignment task can be achieved with relatively low complexity, which makes the scheme scalable.

1) Proposed Packing Scheme: The CNU and VNU memory can be modeled as two dimensional arrays, denoted by $\mathbf{L}_c$ and $\mathbf{L}_v$. The variable-to-check messages are stored in the CNU memory by the CNU access order, i.e., the message $L[(c + \Delta) \bmod m]$ is packed as $L_c[0][0]$. The check-to-variable messages are packed in the VNU memory by the VNU access order, i.e., the message $L[v]$ is packed as $L_v[0][0]$. In general, message $L[k]$ is packed to the location $L_c[\lfloor((k-c-\Delta) \bmod m)/K\rfloor][((k-c-\Delta) \bmod m) \bmod K]$ in the CNU memory and to $L_v[\lfloor((k - v) \bmod m)/K\rfloor][((k - v) \bmod m) \bmod K]$ in the VNU memory. We will illustrate this with an example. Fig. 4(a) shows a circulant permutation matrix with size $m = 15$ and offset $\Delta = 6$. The starting row for CNU processing, $c = 8$ and the starting column for VNU processing is $v = 4$. Fig. 4(b) shows how the messages $L[0]$, $L[1], \ldots, L[15]$ corresponding to the nonzero entries in the CPM are stored in the block RAM. Note, that each message appears in two different locations, because of the double buffering described above. Each block RAM is partitioned logically into a VNU memory and CNU memory and the messages are stored in different order to facilitate conflict free access by CN and VN processing units. We compare our method with the scheme presented in Wang [12], which is shown in Fig. 4(c). There are two advantages of our scheme over the scheme proposed in [12]. First, our scheme does not require any read alignment units (due to double buffering), where as the packing shown in Fig. 4(c) requires two read alignment units in addition to two write alignment units (which we also require). Second, our method works for any value of c and v which is essential to support overlapped message passing, whereas the method in [12] works well for nonoverlapped message passing, i.e., c and v are implicity assumed to be 0. When c and v are nonzero as in this example, the packing scheme in [12] becomes inefficient. For example, consider the updating of the messages in the third word in the VNU memory, $L[12]$, $L[13]$, $L[14]$, $L[0]$. In Fig. 4(c), one can see that these messages are spread across three different memory words, which would entail three reads and a very complex alignment circuitry with the concomitant increase in latency.

2) Sequential Write Alignment Unit: Given that a given message $L[k]$ is mapped to different locations in the CNU and VNU memory, except when $\Delta = 0$, the messages need to be aligned before they are written to the memory. Given the double buffered efficient packing scheme described above, the alignment task is greatly simplified. We need just the current word and the previous word to reconstruct the input order for the CN and VN processing units. Table I shows the state transition table for the example shown in Fig. 4(b), which represents the case when vector length $K = 4$. The alignment unit takes four inputs $I0$, $I1$, $I2$, $I3$ and produces four outputs $O1$, $O2$, $O3$, $O4$ every cycle using the hardware circuitry shown in Fig. 5. The messages are assumed to be quantized to q bits.

The hardware requirements are relatively modest. Let $s = (v - c - \Delta) \bmod m$ for VNU alignment and $s = (c + \Delta - v) \bmod m$ for CNU alignment. For vector length K and CPM size m, 1) when $s = 0$, or when $m \bmod K = 0$, $s \bmod K = 0$, no alignment unit circuitry is needed; 2) when $m \bmod K = 0$,

TABLE I
TRANSITION TABLE FOR THE ALIGNMENT UNIT IN FIG. 5

write addr.	input @ cycle $t-1$				input @ cycle t				output @ cycle t			
	I0	I1	I2	I3	I0	I1	I2	I3	O0	O1	O2	O3
-	-	-	-	-	L4	L5	L6	L7	-	-	-	-
2	L4	L5	L6	L7	L8	L9	L10	L11	L7	L8	L9	L10
3	L8	L9	L10	L11	L12	L13	L14	L0	L11	L12	L13	L14
0	L12	L13	L14	L0	L1	L2	L3	-	L14	L0	L1	L2
1	L1	L2	L3	-	L4	L5	L6	L7	L3	L4	L5	L6

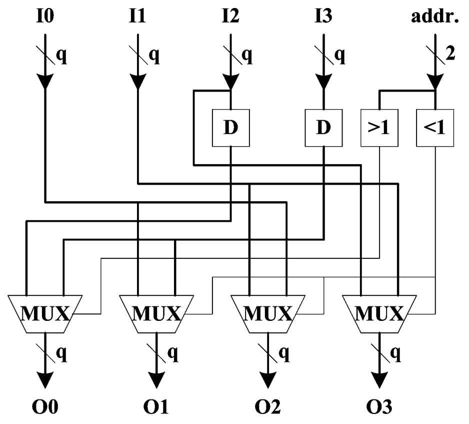


Fig. 5. Hardware implementation of the sequential alignment unit for the example in Fig. 4(b).

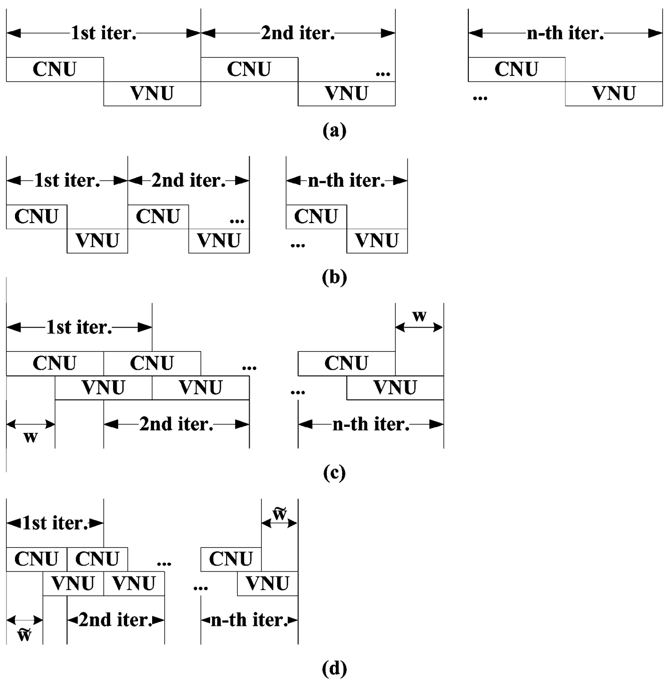


Fig. 6. Timing diagrams of different scheduling algorithms. (a) Original message passing. (b) Vector message passing. (c) Overlapped message passing. (d) Vector overlapped message passing.

$s \bmod K \neq 0$, $s \bmod Kq$-bit registers are required; 3) when $m \bmod K \neq 0$ (the worst case), our implementation requires $K + (s \bmod K) - (m \bmod K)$ q-bit registers and K two to one multiplexors and two comparators. The latency is one cycle for any value of c and v.

B. Vector Overlapped Message Passing

As described in Section II, overlapped message passing (OMP) method was proposed by [9] to improve the throughput of the partially parallel decoder. Fig. 6(a) shows the timing diagram of the baseline message passing algorithm and Fig. 6(c) shows the benefits of overlapped message passing. Fig. 6(b) shows how vector processing (described in the previous section) helps in improving the performance by reducing the CNU and VNU processing time. Vector processing can be combined with overlapped message passing to further improve the throughput as shown by the timing diagram in Fig. 6(d). We call this VOMP or vector overlapped message passing which is described next.

Let $\tilde{w}$ denote the number of waiting clock cycles between CNU update and VNU update of the same iteration. The optimal value of $\tilde{w}$ is computed as follows.

Step 1): Let $\mathbf{c} = \{c_0, c_1, \ldots, c_{\gamma-1}\}$ denote the starting CPM rows for CNUs. Let $\mathbf{v} = \{v_0, v_1, \ldots, v_{\rho-1}\}$ denote the starting CPM columns for VNUs. Let w denote the waiting time between intraiteration CNU and VNU computations. Apply the OMP method described in [8] to get $\mathbf{c}$, $\mathbf{v}$, and w.

Step 2): Compute the starting read and write word address at the 1st iteration. The starting CNU or VNU read address is 0. The starting write message for CNU is $L[(c_i + \Delta) \bmod m]$, which is packed in $L_v[x_c(i,j)][y_c(i,j)]$ in the VNU memory, where

$$x_c(i,j) = \left\lfloor \frac{((c_i - v_j + \Delta) \bmod m)}{K} \right\rfloor,$$
$$y_c(i,j) = ((c_i - v_j + \Delta) \bmod m) \bmod K.$$

Likewise, the starting write message for VNU is $L[v_j]$, which is packed in $L_c[x_v(i,j)][y_v(i,j)]$ in the CNU memory, where

$$x_v(i,j) = \left\lfloor \frac{((v_j - c_i - \Delta) \bmod m)}{K} \right\rfloor,$$
$$y_v(i,j) = ((v_j - c_i - \Delta) \bmod m) \bmod K.$$

Step 3): Compute the minimum waiting time for VOMP by

$$\tilde{w} = \max_{0 \le i < \gamma, 0 \le j < \rho} \left\{ \left\lceil \frac{m}{K} \right\rceil - x_c(i,j), \left\lceil \frac{m}{K} \right\rceil - x_v(i,j) \right\} + 1.$$

Step 4): Compute the number of clock cycles for subiteration time by

$$\tilde{m} = \begin{cases} \left\lceil \frac{m}{K} \right\rceil & \text{if} \sum_{i=0}^{\gamma} \sum_{j=0}^{\rho} (y_c(i,j) + y_v(i,j)) = 0, \\ \left\lceil \frac{m}{K} \right\rceil + 1 & \text{otherwise.} \end{cases}$$

The starting read and write address for CNUs and VNUs increases cyclically with the increment of $\tilde{m} - 1$ for every new iteration for data dependency. In VOMP scheduling scheme, CNUs and VNUs may read and write the extrinsic memory at the same

TABLE II
IMPLEMENTATION RESULTS FOR $\mathcal{C}_1$

	Dai [8]	K=1	K=2	K=3	K=4
Slices	1616 (4%)	1472 (4%)	2792 (8%)	5711 (16%)	6102 (18%)
Slice Flip Flops	1073 (2%)	2178 (3%)	4261 (6%)	8496 (12%)	9263 (13%)
4 input LUTs	2887 (3%)	2449 (3%)	4554 (6%)	8565 (12%)	9698 (14%)
block RAMs	36 (26%)	24 (17%)	24 (17%)	24 (17%)	24 (17%)
f_{CLK} (MHz)	148.7	162	161.9	150.5	149.8
N_{iter} at 4.5 dB	3	3	3	3	3
W	63	63	33	22	17
M	256	256	129	87	65
T_f (Mbps)	99.1	228.9	453	624.8	830.6
T_f Speedup	1	2.3	4.6	6.3	8.4

time, thus memory modules with four ports are needed. We emulate quad-ported memory by clocking the block RAMs at twice the clock frequency and time-multiplexing the read and write ports. This is possible because the critical path in the vector decoder architecture, is in the check-node functional units, and is more than twice the access time for the block RAMs.

Using the method described in [8], we derive a general formula for computing the throughput of the vector decoder as follows:

$$T_f = \begin{cases} \frac{nf_{\text{CLK}}}{M+(M\times N_{iter}+W)} & \text{if } W \le \lfloor \frac{M}{2} \rfloor, \\ \frac{nf_{\text{CLK}}}{M+(W\times(2N_{iter}-1)+M)} & \text{if } W > \lfloor \frac{M}{2} \rfloor \end{cases}$$

where f_{CLK} is the clock frequency, N_{iter} is the average iteration number, W is the number of clock cycles for intraiteration wait, and M is the number of clock cycles for loading or updating messages. The denominator denotes the number of clock cycles to decode a codeword, and contains two parts: overlapped load/store time M, and iteration time. Note that this formula can predict the throughput for all the four cases shown in Fig. 6. For example, for the baseline scalar decoder [Fig. 6(a)] $W = M = m$, while for scalar decoder with OMP scheduling [Fig. 6(c)], $W = w$, $M = m$. For a basic vector decoder [Fig. 6(b)] $W = M = \tilde{m}$ and for vector decoder with OMP scheduling [Fig. 6(d)], $W = \tilde{w}$, $M = \tilde{m}$.

C. Performance, Results and Discussion

First, we compare our implementation with the best known partially parallel decoder architecture implementation [8]. We use the same experimental conditions as in [8]—the same code $\mathcal{C}_1$ [shown in Fig. 1(a)], the same Xilinx Virtex 2 XC2V6000-5 FPGA with 8-bit quantization scheme and min-sum algorithm. For CNU and VNU implementation, we use the method introduced in [14]. Table II shows the results of the comparison with various values of K. The proposed vector architecture with $K = 4$ results in 8.4X improvement in throughput (99.1 Mbps versus 830.6 Mbps). Furthermore, our implementation uses only 24 block RAMs instead of 36. This is because of our optimization of embedding hard decision bit in the extrinsic messages. Note that, when $K = 1$ which is same as scalar decoder, our implementation still outperforms the implementation in [8] because of the double buffering and effective emulation of quad-ported memory, which the implementation in [8] method does not employ.

Next we evaluate the performance of our implementation on two large QC-LDPC codes. The first code is a (8176,7156) (4,32)-regular QC-LDPC code [4], [5] which has been adopted for NASA's LANDSAT (near-earth high-speed satellite communications) and other missions including TDRSS high-rate 1.0 and 1.5 Gbps return link service. The other code is a (3969,3213) irregular code [15].

The (8176,7156) QC-LDPC code is constructed based on the 3-dimensional Euclidean geometry $\text{EG}(3,2)^3$ over the finite field $\text{GF}(2)^3$. Based on the lines of $\text{EG}(3,2)^3$ not passing through the origin, nine circulants of size 511×511 over GF(2) can be constructed, each circulant having both column and row weights 8. To construct the (8176,7156) code, 8 circulants are taken and arranged in a row $\mathbf{G} = [\mathbf{G}_0, \mathbf{G}_1, \ldots, \mathbf{G}_7]$. Then, each circulant $\mathbf{G}_i$, $0 \le i < 8$, is decomposed into a 2×2 array $\mathbf{M}_i$ of 511×511 constituent circulants by column and row decompositions. The decomposition of each circulant of $\mathbf{G}_i$ results in a 2×16 array of constituent circulants of size 511×511. This array is a 2044×8176 matrix $\mathbf{H}$ over GF(2) with column and row weights 4 and 32, respectively. The null space of $\mathbf{H}$ gives the (8176,7156) QC-LDPC code with rate 0.8752. The bit-error performance down to 10^{-8} simulated by computer is shown in Fig. 7.

A VLSI decoder for this code has been built by NASA. Using this decoder, bit-error rate (BER) down to 10^{-14} has been computed, which is shown by the curve labeled "NASA" in Fig. 7. Above BER of 10^{-8}, the performance computed with the VLSI decoder agrees with the one simulated by computer [16]. From Fig. 7, we also see that there is no error floor down to the BER of 10^{-14}. An FPGA decoder for this code is also reported in [12], which packs two messages per memory word. The performance of the code simulated by the FPGA decoder in [12] is 0.1 dB worse than the performance shown in Fig. 7 and furthermore, it has an error-floor at the BER of 10^{-10}.

FPGA implementation of (8176, 7156) code on a Xilinx Virtex 4 XC4VLX-160 FPGA is shown in Table III. We use NMSA $(\alpha = 0.75)$ and 6-bit uniform quantization scheme to design the decoder for different values of the vectorization, i.e., $K = 1, 2, 3, 4$. For comparison we also report the results from [12] on the same code, though it uses fixed-point SPA algorithm as opposed to the min-sum algorithm. Our implementation uses fewer resources and produces a higher decoding throughput. Also, we can see that the throughput does not decrease dramatically, so that the alignment and vector factor up to 4 does not reduce routing efficiency. The logical power increase in a

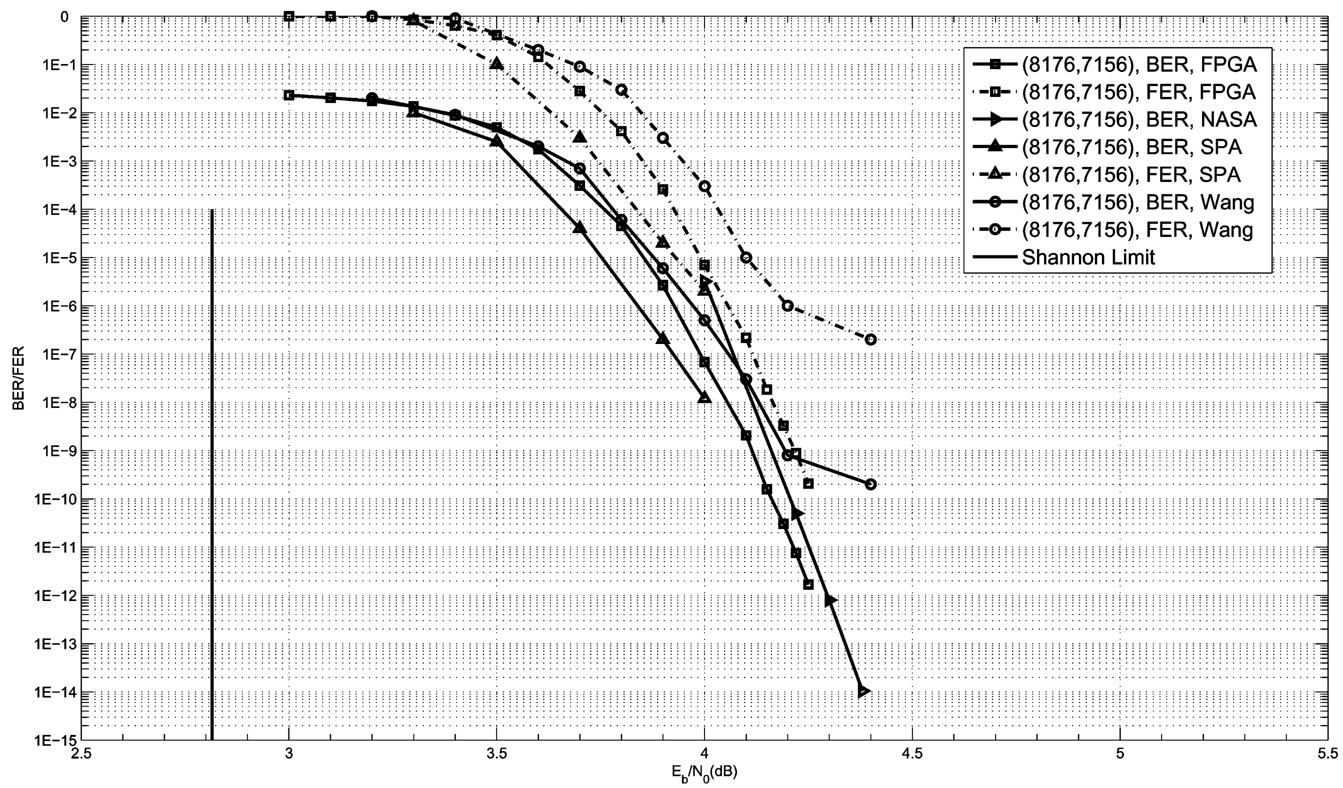


Fig. 7. Performance of the (8176,7156) Code. The curve labeled FPGA denotes the performance measured from our FPGA implementation. The curve labeled NASA is the performance reported from the VLSI chip designed by NASA and the curve labeled Wang is from [12].

TABLE III
FPGA IMPLEMENTATION RESULTS FOR (8176,7156) CODE

	Wang [12] (K=2)	K			
		1	2	3	4
Slices	23052	4021 (4%)	9085 (13%)	14769 (21%)	17857 (26%)
Slice Flip Flops	26926	5907 (3%)	13911 (10%)	21786 (16%)	27210 (20%)
4 input LUTs	28229	7385 (4%)	13720 (10%)	21943 (16%)	27046 (20%)
block RAMs	128	80 (28%)	80 (28%)	80 (28%)	80 (28%)
f_{CLK} (MHz)	193.4	228.7	227.6	214.8	212.2
N_{iter}	15	15	15	15	15
M	256	511	256	171	128
W	256	295	149	100	75
T_f (Mbps)	199.2	195.2	385	541.7	713.8
T_f Speedup	1	1	1.9	2.7	3.6
Logic Power (mW)		199	402	601	736
BRAM Power (mW)		370	381	371	367
Total Power (mW)		2003	2436	2794	3033
Energy per bit (nJ)		10.26	6.33	5.1	4.25

linearly way. The BRAM width influence the power a little bit, and looks similar.

The (3969,3213) irregular QC-LDPC code was constructed using method 1 in [15] as follows. First, a 63×63 array $\mathbf{B}$ of CPMs of size 63×63 is constructed based on the finite field $\text{GF}(2)^6$. Take a 12×63 subarray $\mathbf{B}(12, 63)$ of CPMs from $\mathbf{B}$. Then replace some of the CPMs in $\mathbf{B}(12, 63)$ with zero matrices of size 63×63. This replacement is called masking. The masking is designed based on the degree distributions of VNs and CNs derived using density evolution. After masking, an array $\mathbf{H}$ of CPMs and zero matrices is obtained. The distributions of CPMs in columns and rows of $\mathbf{H}$ are given in Table IV. $\mathbf{H}$ is a 756×3969 matrix with varying column and row weights.

TABLE IV
COLUMN AND WEIGHT DISTRIBUTIONS OF THE (3969,3213) CODE

Column CPM Distribution		Row CPM Distribution	
Column CPM	No. of columns	Row CPM	No. of rows
3	26	20	1
4	25	21	2
7	8	22	2
8	3	23	7
9	1		

The null space of $\mathbf{H}$ gives the (3969,3213) irregular QC-LDPC code with rate 0.81.

TABLE V
VECTOR DECODER IMPLEMENTATION FOR IRREGULAR (3969,3213) CODE

K	1	2	3	4
Slices	14636 (16%)	36708 (41%)	43514 (48%)	62362 (70%)
Flip Flops	21509 (12%)	56507 (31%)	66964 (37%)	98003 (55%)
4 input LUTs	25836 (14%)	55968 (31%)	68485 (38%)	111035 (62%)
block RAMs	330 (98%)	330 (98%)	330 (98%)	330 (98%)
f_{CLK} (MHz)	226.4	204	199.7	195.7
N_{iter}	15	15	15	15
M	63	33	22	17
W	63	33	22	17
T_f (Mbps)	460.1	791	1162.2	1474
T_f Speedup	1	1.7	2.5	3.2
Logic Power (mW)	698	1466	1733	2649
BRAM Power (mW)	1510	1409	1424	1395
Total Power (mW)	4336	5772	6166	7632
Energy per bit (nJ)	9.42	7.3	5.3	5.18

We use the NMSA ($\alpha = 0.75$) and 6-bit uniform quantization scheme to implement the decoder when $K = 1, 2, 3, 4$. The implementation results of the (3369,3213) code on a Xilinx Virtex 4 XC4VLX-200 FPGA is listed in Table V. The results indicate that the decoding performance increases almost linearly with vector length (K). The logic resources increase proportionally to support the additional functional units and alignment logic. However, with each successive generation of FPGAs, the number of resources in terms of slices, flip-flops, and LUTs is increased significantly, so the proposed technique can be used to scale the decoding performance as bigger FPGAs become available.

D. Scalability and Power Analysis of Vector Decoders

Table III and Table V show the impact of increasing the vector length on the power, performance, and resource requirements. It is clear that the proposed implementation is scalable to $K = 4$—the clock frequency drops slightly but the overall throughput increase almost linearly. As expected the number of resources in terms of flip-flops and LUTs increases because of the alignment units but the overall utilization of the slices, flip-flops and LUTs is less than 70%, which is very reasonable. For the (8176,7156) code the resource utilization is under 26%. We believe that scalability beyond vector length of 4 is not necessary for this particular application because of the following reasons. First, the typical word length of the messages is around 8 bits, so it is not possible to pack more than 4 messages in each block RAM word which is around 36 bits. Second, for complex codes, the limitation is the number of block RAMs available in a given FPGA—as high-end FPGAs typically have a very large number of flip-flops and LUTs. So, one is likely to run out block RAMs before flip-flops or LUTs, as results from the (3969,3213) seem to indicate.

The power analysis of the vector decoder shows that the vector decoders improve the energy efficiency of LDPC decoding due to enhanced parallelism and pipelining. For the (8176,7156) code the energy efficiency increases from 10.26 nJ/bit to 4.25 nJ/bit as we increasing the vector length from 1 to 4 and for the more complex code, the energy efficiency increases from 9.42 nJ/bit to 5.18 nJ/bit. The reason being vector decoding makes better use of the FPGA interconnect and block RAM resources through enhanced parallelism and pipelining. Whether you store one message per word or 4 messages per word, the power consumption due to block RAM access is about the same, so vectorization can increase the throughput without increasing the power consumption.

IV. FOLDING: A TECHNIQUE TO VIRTUALIZE BLOCK RAMS FOR QC-LDPC DECODER IMPLEMENTATION

A. Motivation

Many practical codes are large. For example, the $\mathbf{H}$ matrix of the (64800,32400) LDPC code for the DVB-S2 standard can be expressed as quasi-cyclic code with 630 circulants, where each circulant is 360×360 matrix. Even the largest FPGA commercially available (a Virtex-5) has only around hundreds of block RAMs, so a partially parallel implementation show in Fig. 2 that maps each circulant to a separate block RAM will not work. We need to map messages corresponding to multiple circulants to the same physical block RAM. Irregular codes make this problem a little bit more challenging, as instead of circulant permutation matrices we have zero matrices in some places.

We propose a simple (greedy) heuristic to *fold* multiple circulant permutation matrices to the same block RAM. The goal of our heuristic is to reduce the hardware complexity, which includes resources required to route the data from the memory to the CNUs and VNUs and the complexity of the VNUs and CNUs themselves, especially in the case of an irregular code. For a irregular code all the VNUs and CNU will not have the same number of input ports, so mapping CPMs with the same number of inputs to the same block RAM will be advantageous. For simplicity, we make the following additional assumptions. Each message will occupy a single memory word, i.e., we will not attempt to vectorize the decoder in the current exposition; vectorization is orthogonal to folding, so a folded architecture can always be vectorized by using the techniques proposed in Section III. We also assume that CNU and VNU processing is not overlapped. Given that multiple CPMs will be mapped to the same block RAM, and the number of ports to the memory are fixed, VNU and CNU processing will require multiple clock cycles.

B. Folding Transformation for Decoding of QC-LDPC Codes

Let F be an integer that denotes the folding factor, i.e., the number of CPMs mapped to a single block RAM. For example, $F = 2$, means two CPMs will be mapped to the same block

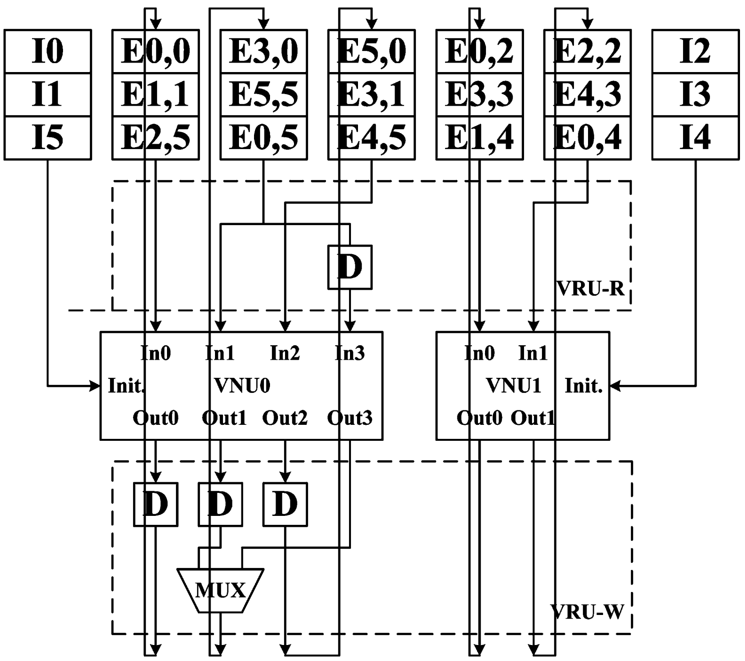


Fig. 8. Variable node reorder unit (VRU) for the (6, 6)-irregular QC-LDPC code decoder: routes messages from/to the EMEMs to/from the VNUs.

RAM. Let B be the number of block RAMs available in the targeted FPGA. For example, in a Virtex-4 LX160 that is used in this paper, B is 288. Let D be the depth of the block RAM such that the word size is sufficient to fit one message.

Consider a (γ, ρ) $\mathbf{H}$-matrix which has γ block rows and ρ block columns, for total of $\gamma \times \rho$ CPMs where each CPM is a $m \times m$ circulant permutation matrix. Let u be the number of nonzero CPMs in $\mathbf{H}$. For a regular code u will be $\gamma \times \rho$, while in a irregular code u will be less than $\gamma \times \rho$ because some CPMs are replaced by all-zero matrices.

Step 1): Find the smallest F and D such that $\lceil(\rho/F)\rceil + \lceil(u/F)\rceil \leq B$ and $F \times m \leq D$. For the scalar decoder, ρ and u block RAMs are needed for IMEMs and EMEMs respectively. Thus, the memory usage are reduced to almost $1/F$.

Step 2): The partially parallel architecture shown Fig. 2 is modified as follows. Use $\lceil\gamma/F\rceil$ CNUs and $\lceil\rho/F\rceil$ VNUs instead of ρ CNUs and γ VNUs.

Step 3): Partition the ρ block columns into $\lceil\rho/F\rceil$ groups such that number of total number of inputs ports of the VNUs is minimized. We use a simple heuristic to do this. Sort the block columns of $\mathbf{H}$ matrix in descending order of the block column weight. Group each successive F block columns together starting with the smallest.

Step 4): Partition the block rows into $\lceil\gamma/F\rceil$ sets in the same way as Step 3).

C. Example: Folding on a Irregular QC-LDPC Code

Consider the $\mathbf{H}$ matrix of an arbitrary irregular QC-LDPC code shown below. The $\mathbf{H}$ matrix of the (6,6)-irregular QC-LDPC code has six block columns $\{\mathbf{C}_0, \mathbf{C}_1, \mathbf{C}_2, \mathbf{C}_3, \mathbf{C}_4, \mathbf{C}_5\}$, whose weights are 4, 2, 2, 3, 2, and 2, respectively. We illustrate the implementation of the code on an FPGA with only 7 block RAMs using the folding transformation described above

$$\mathbf{H} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{O} & \mathbf{A}_{0,2} & \mathbf{O} & \mathbf{A}_{0,4} & \mathbf{A}_{0,5} \\ \mathbf{O} & \mathbf{A}_{1,1} & \mathbf{O} & \mathbf{O} & \mathbf{A}_{1,4} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} & \mathbf{A}_{2,2} & \mathbf{O} & \mathbf{O} & \mathbf{A}_{2,5} \\ \mathbf{A}_{3,0} & \mathbf{A}_{3,1} & \mathbf{O} & \mathbf{A}_{3,3} & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{A}_{4,3} & \mathbf{O} & \mathbf{A}_{4,5} \\ \mathbf{A}_{5,0} & \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{A}_{5,5} \end{bmatrix}.$$

Using Step 1), the optimum folding factor F is found to 3. Following Step 3), the block columns are sorted in descending order, $\{\mathbf{C}_5, \mathbf{C}_0, \mathbf{C}_1, \mathbf{C}_2, \mathbf{C}_3, \mathbf{C}_4\}$ and partitioned into two sets: $\{\mathbf{C}_1, \mathbf{C}_3, \mathbf{C}_0\}$ and $\{\mathbf{C}_1, \mathbf{C}_3, \mathbf{C}_4\}$. We need a 4-input VNU to process the first set and a 2-input VNU to process the second set, thus a total number of six ports are required for incoming extrinsic messages. On the other hand, if we had not sorted the columns but just grouped the columns in the order in which they appear in the $\mathbf{H}$ matrix, i.e., choose $\{\mathbf{C}_0, \mathbf{C}_1, \mathbf{C}_2\}$ as one set and $\{\mathbf{C}_3, \mathbf{C}_4, \mathbf{C}_5\}$ as another, we would need a VNU with 4 inputs for the first set and a VNU with 3 inputs for the second set, and thus a total number of seven ports are used. Similarly, the block rows are also partitioned into two sets $\{\mathbf{R}_2, \mathbf{R}_4, \mathbf{R}_5\}$ and $\{\mathbf{R}_0, \mathbf{R}_1, \mathbf{R}_3\}$ along the lines of Step 4). Without folding, extrinsic messages of block columns $\mathbf{C}_i$, $i \in \{0, 1, 2, 3, 4, 5\}$ are updated in one clock cycle; with folding, extrinsic messages are updated in $F = 3$ clock cycles as follows—messages of $\mathbf{C}_0$ and $\mathbf{C}_2$ are updated in the first cycle, messages of $\mathbf{C}_1$ and $\mathbf{C}_3$ are updated in the next cycle and messages of $\mathbf{C}_5$ and $\mathbf{C}_4$ are updated in the last cycle.

Fig. 8 shows the routing of the memory modules to/from the VNUs of the folded partially parallel decoder architecture for the example irregular QC-LDPC code. The majority of the inputs of the VNUs are directly connected to the outputs of the

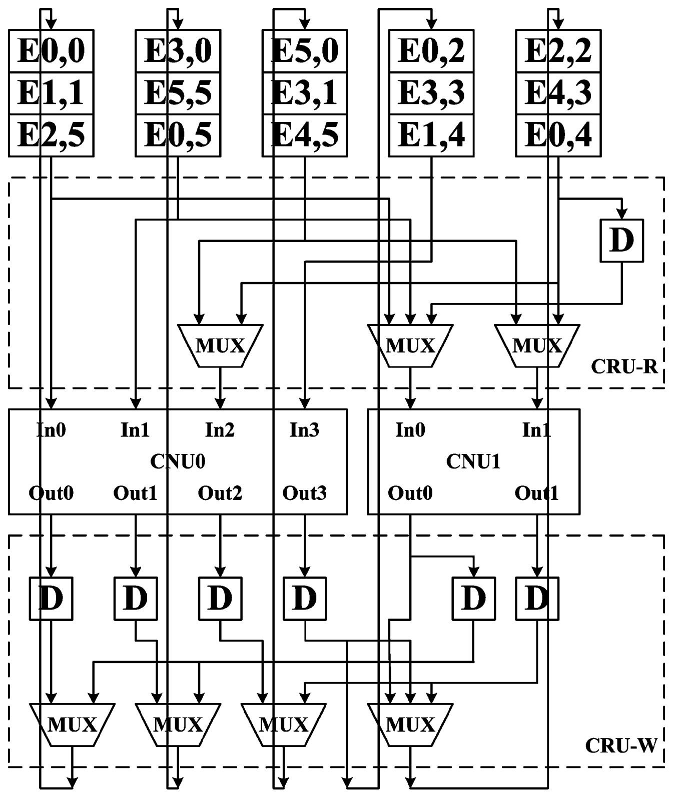


Fig. 9. Check node reorder unit (CRU) for the (6,6)-irregular QC-LDPC code decoder: routes messages from the EMEMs to the CNUs.

TABLE VI
IMPLEMENTATION RESULTS FOR FOLDED DECODER FOR (8176,7156) CODE, NON-OMP SCHEDULING IS USED

	$K=1$	$F=2$	$F=3$	$F=4$
Slices	3347 (4%)	2938 (4%)	2196 (3%)	1579 (2%)
Flip Flops	4563 (3%)	3691 (2%)	2778 (2%)	1938 (1%)
4-input LUTs	6041 (4%)	5421 (4%)	4065 (3%)	2908 (2%)
block RAMs	80 (24%)	40 (12%)	30 (9%)	20 (6%)
f_{CLK} (MHz)	220.5	200.7	200.5	196.2
N_{iter}	15	15	15	15
M	511	1022	1533	2044
W	511	1022	1533	2044
T_f (Mbps)	113.8	51.8	34.5	25.3

TABLE VII
IMPLEMENTATION RESULTS FOR FOLDED DECODER FOR (3969,3213) CODE $K = 1$ CASE IS ON A XILINX XC4VLX-200 FPGA

	$K=1$	$F=2$	$F=3$	$F=4$
Slices	14636 (16%)	12604 (18%)	9830 (14%)	6679 (7%)
Flip Flops	21509 (12%)	18392 (13%)	14016 (10%)	10029 (5%)
4-input LUTs	25836 (14%)	18208 (13%)	13147 (9%)	9303 (6%)
block RAMs	330 (98%)	166 (49%)	110 (33%)	83 (25%)
f_{CLK} (MHz)	226.4	203.8	200.5	200.4
N_{iter}	15	15	15	15
M	63	126	189	252
W	63	126	189	252
T_f (Mbps)	460	207	135.8	101.8

block RAMs, while others are generated via a simple reordering network. The logical memory $\mathbf{I}_l$ and $\mathbf{E}_{j,l}$ are packed in nonoverlapped segments of physical memory based on the partition results in Step 3) of the folding transformation. In this way, the probability that two logical memory of the same columns will reside in the same physical memory will be lowered. For example, the messages of $\{\mathbf{C}_1, \mathbf{C}_3, \mathbf{C}_4\}$ are mapped to three physical block RAMs. Also, we make sure that the messages of the same rows do not map to the same block RAMs. In this way, we can reduce the latency of the processing VNU reorder units and CNU reorder units. As the memory is packed for the preference of VNU update, we need a reorder unit for CNU to read (labeled CRU-R in Fig. 9) that routes the messages from the EMEMs to the CNUs, and also routes the messages from the CNUs back (labeled CRU-W) to the EMEMs, as shown in Fig. 9. Also, there is a reorder unit (labeled VRU in Fig. 8) that routes the residue messages between EMEMs and VNUs.

It should be noted that due to the differences in the block row weight or block column weight, in certain clock cycles, some inputs might not be useful. For example, for VNU update, $\mathbf{C}_4$ and $\mathbf{C}_5$ share the same 4-input VNU. $\mathbf{C}_4$ has a weight of 2. Thus, we should set the corresponding input to 0 since it does not influence the result. Similarly, for CNU update, $\mathbf{R}_0$ and $\mathbf{R}_5$ share the same 4-input CNU, so with min-sum algorithm the unused inputs are set to the maximum allowable positive value and when sum-product algorithm is used the unused inputs are set to 0. This ensures that the computation does not change when a VNU or CNU with a larger number of inputs are used.

D. Results and Discussion

In this section we will present the results for the folded FPGA implementation for the same two codes discussed in Section III. We use NMSA with $\alpha = 0.75$ and 6-bit uniform quantization scheme and Xilinx Virtex 4 XC4VLX-160 FPGA. Table VI shows the results for the (8176,7156) code while Table VII shows the results for the (3969,3213) code. To show the impact of folding we present the results for different folding factors, i.e., $F = 2, 3, 4$. We also show results for $K = 1$ which is our baseline partially parallel decoder with degree of vectorization equal to 1 (a scalar decoder). To make a fair comparison, nonoverlapped message passing is used for $K = 1$. The key point to note is that even complex codes such as the (8176,7156) code can be realized with as few as 20 block RAMs (with $F = 4$), of course, with a significantly lower performance because the memory is effectively shared by different functional units. It should be noted that the OMP scheduling is not applied to the folded decoder.

Fig. 10 shows how folding can be combined with vectorization described in the previous section. Both folding and vectorization exploit the unused depth of the block RAMs—given that most CPMs are much smaller than the number of locations in a block RAM. Vectorization uses the block RAM depth for double buffering as explained in the previous section, while folding uses the block RAM depth to map multiple CPMs into the same block RAM. So, the folding factor F will be constrained by the available depth of the block RAM. For example, when vector decoding and folded decoding are applied to the (8176,7156) code, F can only be 2, since $2 \times 511 \times 2$ messages will be used.

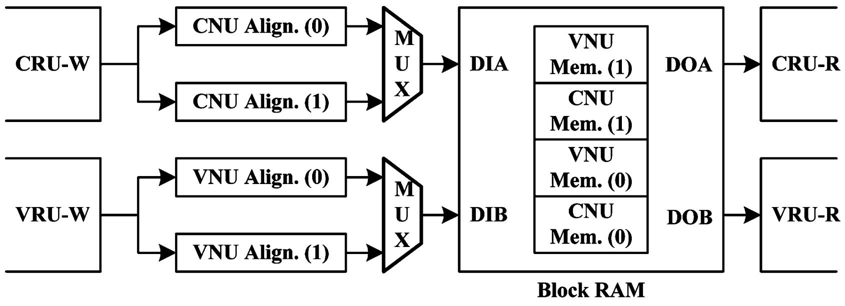


Fig. 10. Vectorized folded decoder.

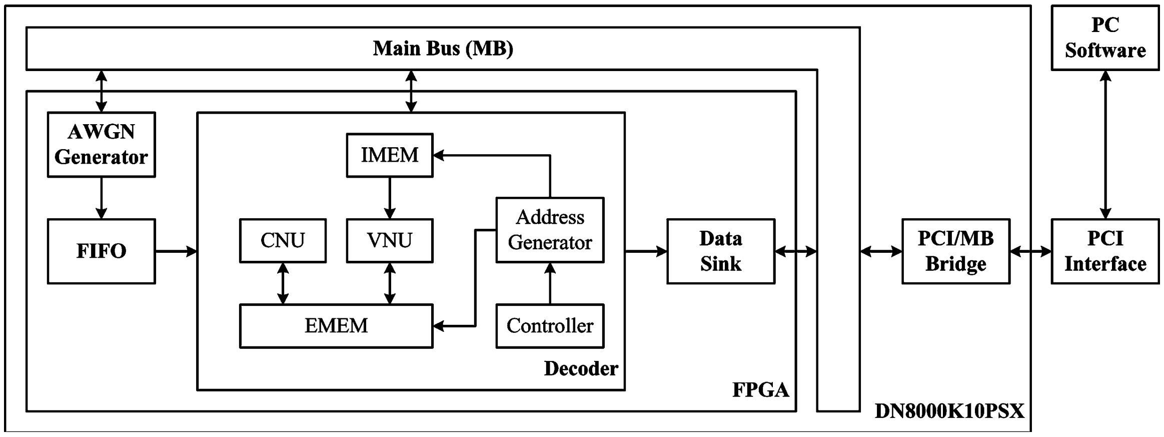


Fig. 11. FPGA-based simulation platform for QC-LDPC codes decoder.

V. QCSyn and Emulation Platform

The goal of this project is to create an FPGA-based emulation platform to accelerate the design space exploration of quasi-cyclic LDPC codes. The design environment consists of the DN8000K10PSX board from the Dini Group and an architectural synthesis tool called QCSyn. The implementation results reported in this paper were generated using the design environment described here.

QCSyn is a tool written in Python that takes the description of the QC-LDPC code and the target FPGA and generates the Verilog code a folded or vector architecture. The Verilog code can be simulated with Modelsim and synthesized using Xilinx synthesis tools. The goal of QCSyn is to reduce the time required to generate a customized folded or vectorized architecture for a given LDPC code from months to days. The first step involves reading the FPGA block RAM resource specification file (which describes the number of block RAMs (block RAM budget) and the aspect ratios) and the QC-LDPC code structure along with the design parameters listed such as data quantization format for message representation, number of pipeline stages for CNU, number of pipeline stages for VNU, CPM offsets, CPM size m, distribution of CN degree and distribution of VN degree. One of the major parameters for choosing architectures is the number of block RAMs. When it is smaller than the requirement of scalar decoder, fold decoder is used.

Next, depending on whether the designer wants a vector decoder or a folded decoder, the architecture specific parameters are determined. This includes resource allocation (number and types of CNUs and VNUs) and scheduling. This is followed by generation of all the logic required for alignment, message reordering, and interconnection. The last steps involves instantiating the CNU and VNU components from the prebuilt library and integration with the testbench. The generated Verilog can be automatically verified with the bit-accurate C model and synthesized by the Xilinx tools. The tasks that have to be performed manually include generating the CNU or VNU of a specific number of inputs (if it is not already in the library) and some logic to interface to the FIFO and noise generator on the FPGA board.

Our FPGA-based simulation platform shown in Fig. 11 is based on the DN8000K10PSX board from Dini Group, which has two Xilinx Virtex-4 LX160-10's and is hosted in a 64-bit PCI slot (66MHz). The *Main Bus*, *PCI/Main Bus Bridge*, *PCI Interface* are provided by the vendor, the rest of the modules were developed by us. The main components include the **AWGN generator**, based on the Wallace method [17] that generates the erroneous all-zero codeword for the decoder, a FIFO to synchronize data across different clock boundaries, the **data sink** which gathers the decoder statistics such as iteration count, status of the codeword, etc., and the **PC software** that initializes the AWGN generator and interfaces with the user.

VI. Related Work

Fully parallel decoder architectures such as [18] that employ a separate processing unit for each node of the Tanner graph can exhibit the highest performance but are impractical for an FPGA-based implementation because FPGAs have limited resources. Thus, partially parallel decoders are more preferable

[19]–[23]. The details of a partially parallel decoder and its optimization using overlapped message passing were described in Section II illustrated in Figs. 2 and 6. In [12], Wang and Cui propose an enhanced partially parallel decoder architecture for QC-LDPC codes, which is somewhat similar to the vector decoding architecture described in Section III. However, there are some key differences. First, the memory packing method described in [12] is not amenable to overlapped message passing and the data alignment scheme is inefficient. As a result, the decoding throughput is significantly lower than what is achieved by our approach, as shown in Fig. 3. With $K = 4$ we outperform them by a factor of 3.6. Furthermore, the bit error performance on the (8176,7156) code reported in the paper has some discrepancy from NASA and our implementation, as shown by the error-floor in Fig. 7. Preliminary results from the vector processing aspects of this work were reported in [24]. However, [24] does not present bit-error performance curves and does not show how the proposed vector implementation compares in terms of resource utilization and decoding throughput with other architectures that use message packing. Furthermore, [24] does not address the folding transformation to take advantage of configurable depth of the block RAMs and the QCSyn tool that facilitates design space exploration of FPGA implementation of decoders for quasi-cyclic LDPC codes.

In [25], Liu proposes a register-based decoder for structured LDPC codes which is composed of circulants (not necessarily circulant permutation matrices). For an LDPC code whose parity check matrix is $J \times n$, the variable-to-check messages are grouped by equal size of p. For the (γ, ρ)-regular code, the decoder has J ρ/p-input CNUs and n/p γ-input VNUs. The decoder breaks the sequential tie between CNU update and VNU update. The two half-iteration can be overlapped with the waiting clock cycle of 1. The decoder is a tradeoff between memory-shared partially parallel decoder and fully parallel decoder. The proposed ASIC implementation of a (2048,1723) (6,32)-regular structured LDPC code (10GBaseT ethernet standard) achieved a maximum decoding throughput of 5.3 Gb/s at 16 iterations.

The idea of mapping messages from multiple CPMs to the the same physical memory has been explored by some researchers including [6], [13], [26]–[28]. The common theme underlying these approaches is to pack messages of the same CPM into one *long* word (realized by a concatenation of several block RAMs) or several consecutive (say 4) memory words and use a long barrel shifter to reorder the message so that they can be aligned to be processed. Typically, one-input one-output processing units are used to process the messages serially. The barrel shifter [29] is quite expensive and researchers [29], [30] have explored efficient implementations of the barrel shifter using Benes network structure. By mapping messages in this way, it is very efficient to implement highly flexible decoder for multiple standards [31], [32].

VII. Conclusions and Future Work

We described two specific optimizations called *vectorization* and *folding* to take advantage of the configurable data-width and depth of embedded memory in an FPGA to improve the throughput of a decoder for quasi-cyclic LDPC codes. We demonstrate a decoding throughput of 1.162 gigabits per second for the (3969,3213) code on a Xilinx Virtex4 XC4VLX-200 FPGA and 713.8 Mbps for the (8176,7156) code on a Xilinx Virtex4 XC4VLX-160 FPGA using the vectorization techniques, which is significantly better than the best known FPGA implementation for the same code in research literature [12]. With folding we showed that quasi-cyclic LDPC codes with a very large number of circulants can be implemented on FPGAs with a small number of embedded memory blocks. This is useful for developing decoders for the codes used in DVB-S2 for example. We also described a synthesis tool called QCSyn that takes the $\mathbf{H}$ matrix of a quasi-cyclic LDPC code and the resource characteristics of an FPGA and automatically synthesizes a folded or vector architecture that maximizes the decoding throughput for the code on the given FPGA by selecting the appropriate degree folding and/or vectorization. We plan to extend the methods described in this paper and the QCSyn tool to develop an universal decoder architecture for quasi-cyclic LDPC codes, basically, a single decoder that works for a family of quasi-cyclic codes.

References

[1] R. Gallager, "Low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. IT-8, no. 1, pp. 21–28, Jan. 1962.

[2] D. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inf. Theory*, vol. 45, no. 2, pp. 399–431, Mar. 1999.

[3] M. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Trans. Commun.*, vol. 47, no. 5, pp. 673–680, May 1999.

[4] L. Chen, J. Xu, I. Djurdjevic, and S. Lin, "Near-Shannon-limit quasi-cyclic low-density parity-check codes," *IEEE Trans. Commun.*, vol. 52, no. 7, pp. 1038–1042, Jul. 2004.

[5] Z. Li, L. Chen, L. Zeng, S. Lin, and W. Fong, "Efficient encoding of quasi-cyclic low-density parity-check codes," *IEEE Trans. Commun.*, vol. 53, no. 11, p. 1973, Nov. 2005.

[6] Z. Zhang, L. Dolecek, B. Nikolic, V. Anantharam, and M. Wainwright, "Investigation of error floors of structured low-density parity-check codes by hardware emulation," in *Proc. IEEE Global Telecommun. Conf.*, Dec. 2006, pp. 1–6.

[7] E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam, "High throughput low-density parity-check decoder architectures," in *Proc. IEEE Global Telecommun. Conf.*, 2001, vol. 5, pp. 3019–3024.

[8] Y. Dai, Z. Yan, and N. Chen, "Optimal overlapped message passing decoding of quasi-cyclic LDPC codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 5, pp. 565–578, May 2008.

[9] Y. Chen and K. Parhi, "Overlapped message passing for quasi-cyclic low-density parity check codes," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 51, no. 6, pp. 1106–1113, Jun. 2004.

[10] T. Zhang and K. Parhi, "A 54 Mbps (3,6)-regular FPGA LDPC decoder," in *Proc. IEEE Workshop Signal Process. Syst.*, Oct. 2002, pp. 127–132.

[11] T. Zhang, "Efficient VLSI Architectures for Error-Correcting Coding," Ph.D., Univ. Minnesota, Minneapolis, 2002.

[12] Z. Wang and Z. Cui, "Low-complexity high-speed decoder design for quasi-cyclic LDPC codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 1, pp. 104–114, Jan. 2007.

[13] M. Gomes, G. Falcao, V. Silva, V. Ferreira, A. Sengo, and M. Falcao, "Flexible parallel architecture for DVB-S2 LDPC decoders," in *Proc. IEEE Global Telecommun. Conf.*, Nov. 2007, pp. 3265–3269.

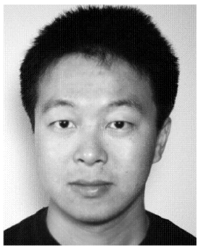
[14] C. Wey, M. Shieh, and S. Lin, "Algorithms of finding the first two minimum values and their hardware implementation," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 55, no. 11, pp. 3430–3437, Dec. 2008.

[15] L. Lan, L. Zeng, Y. Tai, L. Chen, S. Lin, and K. Abdel-Ghaffar, "Construction of quasi-cyclic LDPC codes for AWGN and binary erasure channels: A finite field approach," *IEEE Trans. Inf. Theory*, vol. 53, no. 7, pp. 2429–2458, Jul. 2007.

[16] W. Ryan and S. Lin, *Channel Codes: Classical and Modern*. Cambridge, U.K.: Cambridge Univ. Press, 2009.

[17] D.-U. Lee, W. Luk, J. Villasenor, G. Zhang, and P. Leong, "A hardware Gaussian noise generator using the Wallace method," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 13, no. 8, pp. 911–920, Aug. 2005.
[18] A. Blanksby and C. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *IEEE J. Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, Mar. 2002.
[19] S.-H. Kang and I.-C. Park, "Loosely coupled memory-based decoding architecture for low density parity check codes," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 53, no. 5, pp. 1045–1056, May 2006.
[20] L. Yang, H. Liu, and C.-J. Shi, "Code construction and FPGA implementation of a low-error-floor multi-rate low-density parity-check code decoder," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 53, no. 4, pp. 892–904, Apr. 2006.
[21] G. Masera, F. Quaglio, and F. Vacca, "Implementation of a flexible LDPC decoder," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 54, no. 6, pp. 542–546, Jun. 2007.
[22] T. Zhang and K. Parhi, "Joint (3,k)-Regular LDPC code and decoder/encoder design," *IEEE Trans. Signal Process.*, vol. 52, no. 4, pp. 1065–1079, Apr. 2004.
[23] H. Zhong and T. Zhang, "Block-LDPC: A practical LDPC coding system design approach," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 52, no. 4, pp. 766–775, Apr. 2005.
[24] X. Chen, J. Kang, S. Lin, and V. Akella, "Accelerating FPGA-based emulation of quasi-cyclic LDPC codes with vector processing," presented at the Des., Autom. Test Eur., Nice, France, Apr. 2009.
[25] L. Liu and C.-J. Shi, "Sliced message passing: High throughput overlapped decoding of high-rate low-density parity-check codes," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 55, no. 11, pp. 3697–3710, Dec. 2008.
[26] F. Kienle, T. Brack, and N. Wehn, "A synthesizable IP core for DVB-S2 LDPC code decoding," in *Design, Automation and Test in Europe*, Mar. 2005, vol. 3, pp. 100–105.
[27] J. Dielissen, A. Hekstra, and V. Berg, "Low cost LDPC decoder for DVB-S2," in *Proc. Des., Autom. Test Eur.*, Mar. 2006, vol. 2, pp. 1–6.
[28] T. Brack, M. Alles, T. Lehnigk-Emden, F. Kienle, N. Wehn, N. L'Insalata, F. Rossi, M. Rovini, and L. Fanucci, "Low complexity LDPC code decoders for next generation standards," in *Proc. Des., Autom. Test Eur.*, Apr. 2007, pp. 1–6.
[29] J. Lin, Z. Wang, L. Li, J. Sha, and M. Gao, "Efficient shuffle network architecture and application for WiMAX LDPC decoders," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 56, no. 3, pp. 215–219, Mar. 2009.
[30] Z. Wang and Z. Cui, "A memory efficient partially parallel decoder architecture for quasi-cyclic LDPC codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 4, pp. 483–488, Apr. 2007.
[31] D. Bao, B. Xiang, R. Shen, A. Pan, Y. Chen, and X. Zeng, "Programmable architecture for flexi-mode QC-LDPC decoder supporting wireless LAN/MAN applications and beyond," *IEEE Trans. Circuits Syst. I, Reg. Papers*, to be published.
[32] C. Zhang, Z. Wang, J. Sha, L. Li, and J. Lin, "Flexible LDPC decoder design for multi-Gb/s applications," *IEEE Trans. Circuits Syst. I, Reg. Papers*, to be published.

Xiaoheng Chen received the B.S. and M.S. degrees from the Zhejiang University, Hangzhou, China, in 2005 and 2007, respectively. He is currently working toward the Ph.D. degree in the Department of Electrical and Computer Engineering, University of California, Davis. He has done a lot of work on field-programmable gate arrays and digital application-specific integrated circuit designs. His general interests include channel coding theory, very large scale integration (VLSI) design, communication theory, multimedia coding theory, VLSI system design, and VLSI design methodology. Main current interests include VLSI design for low-power and high-performance communication system, VLSI design for ultrahigh-throughput channel coding system, and VLSI design for video and audio technology.



Jingyu Kang received the B.S. and M.S. degrees in electronic engineering from Tsinghua University, Beijing, China, in 2002 and 2005, respectively, and the Ph.D. degree in electrical and computer engineering from the University of California, Davis in 2009.

He is currently with Augusta Technology USA, Inc., Santa Clara, CA, as a System Engineer. His research interests include error control coding and signal processing for data storage and communication systems.

Shu Lin (S'62-M'65-SM'78-F'80-LF'00) received the B.S.E.E. degree from the National Taiwan University, Taipei, Taiwan, in 1959, and the M.S. and Ph.D. degrees in electrical engineering from Rice University, Houston, TX, in 1964 and 1965, respectively.

In 1965, he joined the Faculty of the University of Hawaii, Honolulu, as an Assistant Professor of Electrical Engineering. He became an Associate Professor in 1969 and a Professor in 1973. In 1986, he joined Texas A&M University, College Station, as the Irma Runyon Chair Professor of Electrical Engineering. In 1987, he returned to the University of Hawaii. From 1978 to 1979, he was a Visiting Scientist at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, where he worked on error control protocols for data communication systems. He spent the academic year of 1996–1997 as a Visiting Professor at the Technical University of Munich, Munich, Germany. He retired from University of Hawaii in 1999 and he is currently an Adjunct Professor at University of California, Davis. He has published numerous technical papers in IEEE Transactions and other refereed journals. He is the author of the book *An Introduction to Error-Correcting Codes* (Englewood Cliffs, NJ: Prentice-Hall, 1970). He also coauthored (with D. J. Costello) the book *Error Control Coding: Fundamentals and Applications* (Upper Saddle River, NJ: Prentice-Hall, 1st ed., 1982, 2nd ed., 2004), and (with T. Kasami, T. Fujiwara, and M. Fossorier) the book *Trellises and Trellis-Based Decoding Algorithms*, (Boston, MA: Kluwer Academic, 1998). He has served as the Principal Investigator on 36 research grants. His current research areas include algebraic coding theory, coded modulation, error control systems, and satellite communications.

Dr. Lin is a Member of the IEEE Information Theory Society and the Communication Society. He served as the Associate Editor for Algebraic Coding Theory for the IEEE TRANSACTIONS ON INFORMATION THEORY from 1976 to 1978, the Program Cochair of the IEEE International Symposium of Information Theory held in Kobe, Japan, in June 1988, and a Cochair of the IEEE Information Theory Workshop held in Chengdu, China, October 2006. He was the President of the IEEE Information Theory Society in 1991. In 1996, he was a recipient of the Alexander von Humboldt Research Prize for U.S. Senior Scientists, a recipient of the IEEE Third-Millennium Medal, 2000, and a recipient of the IEEE Communications Society 2007 Stephen O. Rice Prize in the Field of Communication Theory.

Venkatesh Akella received the Ph.D. degree in computer science from the University of Utah, Salt Lake City.

He is a Professor of Electrical and Computer Engineering at University of California, Davis. His current research encompasses computer architecture and embedded systems.

Prof. Akella is a Member of ACM.