Loosely Coupled Memory-Based Decoding Architecture for Low Density Parity Check Codes

Se-Hyeon Kang, *Student Member, IEEE,* and In-Cheol Park, *Senior Member, IEEE*

***Abstract*—Parallel decoding is required for low density parity check (LDPC) codes to achieve high decoding throughput, but it suffers from a large set of registers and complex interconnections due to randomly located 1's in the sparse parity check matrix. This paper proposes a new LDPC decoding architecture to reduce registers and alleviate complex interconnections. To reduce the number of messages to be exchanged among processing units (PUs), two data flows that can be loosely coupled are developed by allowing duplicated operations. In addition, intermediate values are grouped and stored into local storages each of which is accessed by only one PU. In order to save area, local storages are implemented using memories instead of registers. A partially parallel architecture is proposed to promote the memory usage and an efficient algorithm that schedules the processing order of the partially parallel architecture is also proposed to reduce the overall processing time by overlapping operations. To verify the proposed architecture, a 1024 bit rate-1/2 LDPC decoder is implemented using a 0.18-μm CMOS process. The decoder runs correctly at the frequency of 200 MHz, which enables almost 1 Gbps decoding throughput. Since the proposed decoder occupies an area of 10.08 mm^2, it is less than one fifth of area compared to the previous architecture.**

***Index Terms*—Channel coding, decoder, factor graph, low density parity check (LDPC) code, matrix permutation, scheduling.**

I. Introduction

Low density parity check (LDPC) codes are originally devised to exploit low decoding complexity by constructing sparse parity check matrices. Though the LDPC code does not have a maximized minimum distance due to the randomly generated sparse parity check matrix, the typical minimum distance increases linearly as the block length increases. Moreover, the error probability decreases exponentially as signal-to-noise ratio (SNR) increases when the code length is sufficiently long, whereas the decoding complexity is linearly proportional to the code length. Exploiting this benefit, the LDPC code is applied to several applications such as data storages, digital subscriber lines [6], DVB-S2 [7], and CDMA [8], especially to space-time coded OFDM systems due to its scalability [9]. Since it is not difficult to design competitive LDPC codes that deal with any block-length and any code rate, the LDPC codes can be easily combined with space time coded systems that scale the number of antennas or information rate according to the environment change. Recent simulation results show that the LDPC code can achieve a performance that is within 0.04 dB of Shannon limit [2] and the performance of the LDPC code is close to that of the turbo code if the block length is larger than 1000 bits [3].

Despite these advantages, when the LDPC code was first introduced, it made little impact on the information theory community because enormous storage is required in encoding and the large computational complexity in decoding. Whereas the iterative decoding can be explained in several different points of view, the main point to understand the complexity of the decoding algorithm is that check equations and received bits affect each other iteratively. The check equations indicate probable error positions. According to the results of the check equations, the received bits are modified and then fed again into the check equations. This process is repeated until all the check equations are satisfied or the predefined maximum number of iterations is reached. In the message passing algorithm, which this decoding process is generally expressed by, the check equations and received bits are mapped into check nodes and variable nodes, respectively, and the error probabilities are calculated and sent as messages along the edges connecting two nodes. Though there are a few edges for each variable node, the number of edges usually exceeds several thousands to get a reasonable coding gain. Since the graph is irregular, how to make interconnections for the large number of edges is a main problem to be solved in hardware implementation.

Modern VLSI technology is so advanced that it enables parallel architectures exploiting the benefit of inherently parallel LDPC decoding algorithms. Blanksby *et al.* implemented an 1-Gb/s fully parallel decoder in which the message passing algorithm is directly mapped [4]. This architecture, however, requires a large number of complex routings between concurrent processing units (PUs) each of which corresponds to a node of the Tanner graph of the code, leading to the average net length of 3 mm and the total die size of 52.5 mm^2. On the other side, Yeo *et al.* proposed an area-efficient architecture that serializes the computations by sharing PUs [5]. Consequently, one iteration takes about ten-thousand cycles to decode a codeword and wide-input multiplexers are required to select one out of several thousand intermediate values to be fed into the shared PUs. These two counter examples show that high-throughput LDPC decoding architectures should exploit the benefit of parallel decoding algorithms while reducing the interconnection complexity.

This paper proposes a new architecture to reduce registers and alleviate complex interconnections. The proposed architecture consists of two data flows to minimize the number of messages

Manuscript received September 13, 2004; revised April 20, 2005. This work was supported in part by the Institute of Information Technology Assessment through the ITRC, by the Korea Science and Engineering Foundation through MICROS center and by IC Design Education Center. This paper was recommended by Associate Editor Z. Wang.

The authors are with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea (e-mail: shkang@ics.kaist.ac.kr; icpark@ee.kaist.ac.kr).

Digital Object Identifier 10.1109/TCSI.2005.862181

to be exchanged, leading to an area-efficient decoder. Instead of sending all the messages, only the minimal information is exchanged between the two data flows. Then, each data flow reconstructs the original messages using the minimal information. Furthermore, the intermediate values are stored into local memories each of which is accessed by only one PU.

Since the parity check matrix does not have any property except the row and column regularity constraints, several LDPC codes can be constructed by pseudorandom permutations of sub-matrices. Motivated by this fact, there were some research efforts on architecture-aware LDPC code construction [14], [15]. These approaches put some constraints on the pseudorandom permutation to enhance the regularity of the parity check matrix while maintaining the code performance. This regularity helps alleviate the interconnection complexity because large-sized multiplexers and de-multiplexers, which are needed to read and write the messages due to randomly generated 1's, can be reduced into small-sized multiplexers. In this paper, however, we focus on reducing the interconnection complexity for a given LDPC code rather than the code construction.

The rest of this paper is organized as follows. Section II briefly introduces the LDPC code and the decoding algorithm. Section III proposes a new LDPC decoding architecture based on loosely coupled two data flows. Section IV extends the proposed architecture to the partially parallel architecture and proposes an efficient scheduling algorithm. A prototype LDPC decoder is presented in Section V and its performance is summarized in Section VI. Finally, Section VII addresses some concluding remarks.

II. Low Density Parity Check Codes

The LDPC code, which was first introduced by Gallager in 1962 [1], is a kind of binary linear block codes. A (n, γ, ρ) LDPC code means that its block length is n and the column and row weight of its parity check matrix are γ and ρ, which represent the number of 1's in a column and a row, respectively. The column and row weights are much smaller than n to achieve a sparse matrix $\mathbf{H}$. For a (n, γ, ρ) LDPC code, the number of parity check equations is as many as the number of rows of the parity check matrix: i.e., $J(= n \times \gamma/\rho)$ parity check bits are attached to the message. Since the total number of 1's in the parity check matrix is the same whether it is counted in the column direction or in the row direction, the equation, $n \times \gamma = J \times \rho$, holds for the code parameters and the code rate can be calculated as $R = 1 - \gamma/\rho$. Fig. 1 shows an example matrix $\mathbf{H}$ of a (12, 3, 6) LDPC code. An LDPC code associated with fixed row and column weights is called a regular LDPC code, and an irregular LDPC code allows some variations in row and column weights.

A. Factor Graph

The LDPC code is often represented by a factor graph to make it easy to understand the message passing decoding algorithm, which is a bipartite graph that expresses how a global function of many variables is factored into a product of local functions [11]–[13].

$$\mathrm{H} = \left[\begin{array}{cccccccccccc}
1&1&0&0&0&1&1&1&0&0&1&0\\
0&1&1&1&1&0&1&0&1&0&0&0\\
1&1&1&0&0&1&0&0&1&0&0&1\\
0&0&0&1&0&0&1&1&0&1&1&1\\
0&0&0&0&1&1&0&1&1&1&1&0\\
1&0&1&1&1&0&0&0&0&1&0&1
\end{array}\right]
\begin{array}{l}\text{row weight } \rho = 6\\ \\ \text{rows } J = 6\\ \\ \\ \\ \end{array}$$

$$\text{column weight } \gamma = 3 \qquad \text{columns } n = 12$$

Fig. 1. Example matrix $\mathbf{H}$ of a (12, 3, 6) LDPC code.

Fig. 2 shows the factor graph of the (12, 3, 6) LDPC code, which consists of two sets of nodes: i.e., variable nodes, $\{v_j\}$, and check nodes, $\{c_i\}$. A column of the parity check matrix corresponds to a variable node represented as a circle in the left side of the factor graph and a row corresponds to a check node represented as a square. The edge between a variable node v_j and a check node c_i is constructed if there is 1 at (i, j) in the parity check matrix. Therefore, each check node represents a check equation used in generating parity check bits and each variable node represents one bit in the codeword. The variable nodes that are connected to a check node are called the neighbor variable nodes of the check node. For a variable node, the neighbor check nodes can be defined similarly.

The message passing algorithm can be expressed easily based on the factor graph. For all the bits in a received vector, *a posteriori* probabilities are calculated in the variable nodes and delivered to the neighbor check nodes through the edges. Parity check operations are executed with these probabilities in all the check nodes and the results are passed again to the neighbor variable nodes. Therefore, a fully parallel decoding architecture can be directly derived from the factor graph by implementing each node as a PU.

B. Message Passing Algorithm

General parity check codes whose minimum distance is $d_{\min}$ can be optimally decoded if there are $\lfloor (d_{\min} - 1)/2 \rfloor$ or fewer errors, whereas the LDPC codes cannot be optimally decoded. Therefore, the message passing algorithm to be described below is widely used instead, although it is optimal only when there is no cycle in the factor graph. In the following equations, an antipodal bit-to-symbol mapping is assumed; i.e., 0 and 1 are mapped to -1 and 1, respectively.

1) Initialize each variable node v_j to the probability ratio of the corresponding received bit

$$\Delta_j = \frac{p(r_j | x_j = +1)}{P(r_j | x_j = -1)} = \exp\left(\frac{2r_j}{\sigma^2}\right). \tag{1}$$

2) Variable-to-check (VTC) step: Each check node c_i computes the likelihood ratios Λ_{ij} by using the following equation and sends it to variable node v_j

$$\Lambda_{ij} = \prod_{j' \in N(i) \backslash j} \frac{1 - \Delta_{j'i}}{1 + \Delta_{j'i}}. \tag{2}$$

$$
\mathrm{H} = \begin{bmatrix}
1\,1\,0\,0\,0\,1\,1\,1\,0\,0\,1\,0 \\
0\,1\,1\,1\,1\,0\,1\,0\,1\,0\,0\,0 \\
1\,1\,1\,0\,0\,1\,0\,0\,1\,0\,0\,1 \\
0\,0\,0\,1\,0\,0\,1\,1\,0\,1\,1\,1 \\
0\,0\,0\,0\,1\,1\,0\,1\,1\,1\,1\,0 \\
1\,0\,1\,1\,1\,0\,0\,0\,0\,1\,0\,1
\end{bmatrix}
$$

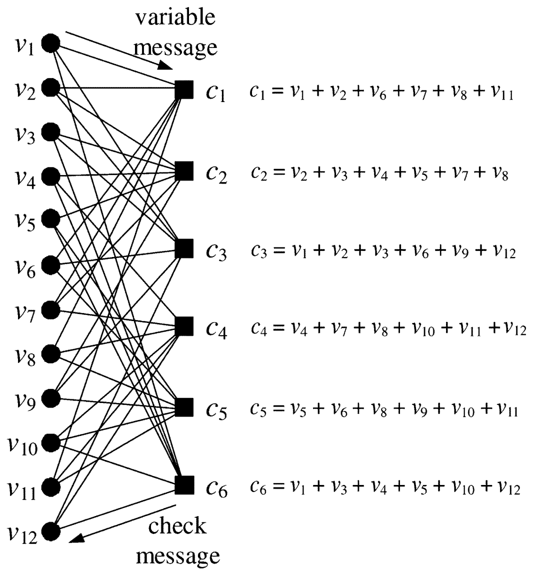

variable message
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
v_9
v_{10}
v_{11}
v_{12}
c_1
c_2
c_3
c_4
c_5
c_6
$c_1 = v_1 + v_2 + v_6 + v_7 + v_8 + v_{11}$
$c_2 = v_2 + v_3 + v_4 + v_5 + v_7 + v_8$
$c_3 = v_1 + v_2 + v_3 + v_6 + v_9 + v_{12}$
$c_4 = v_4 + v_7 + v_8 + v_{10} + v_{11} + v_{12}$
$c_5 = v_5 + v_6 + v_8 + v_9 + v_{10} + v_{11}$
$c_6 = v_1 + v_3 + v_4 + v_5 + v_{10} + v_{12}$
check message

Fig. 2. Factor graph for a (12, 3, 6) LDPC code.

3) Check-to-variable (CTV) step: Each variable node v_j computes probability ratios, denoted by Δ_{ji}, by using the following equation and sends it to check node c_i

$$\Delta_{ji} = \frac{p(r_j|+1)}{P(r_j|-1)} \prod_{i' \in M(j)\backslash i} \frac{1-\Lambda_{i'j}}{1+\Lambda_{i'j}}. \tag{3}$$

4) For each variable node, calculate a tentative bit-by-bit decoding $\hat{x}_j$ using the pseudo-posteriori probability Δ_j

$$\Delta_j = \frac{p(r_j|+1)}{P(r_j|-1)} \prod_{i \in M(j)} \frac{1-\Lambda_{ij}}{1+\Lambda_{ij}} \tag{4}$$

$$\hat{x}_j = \begin{cases} 0, & \text{when } \Delta_j \le 1 \\ 1, & \text{when } \Delta_j > 1. \end{cases} \tag{5}$$

5) Check if $\hat{\mathbf{x}} \times \mathbf{H}^T = 0$ is satisfied. If it is satisfied or the maximum number of iterations is reached, the decoding algorithm finishes. Otherwise, the algorithm repeats from step 2).

The symbols Δ and Λ correspond to the outgoing messages of the variable and check nodes, respectively, and $N(i)$ and $M(j)$ represent the set of neighbor nodes of check node c_i and variable node v_j, respectively. The not-notation (') marked as superscript means that the product is calculated over the indexes excluding its own index, and it is referred to as "self-exclusive" in this paper. A detailed explanation of the algorithm can be found in [10].

To reduce the hardware cost, two transformations are usually employed in practical implementations. First, the product operations are transformed into summations by using log-likelihood ratio, and the probability ratio is expressed in terms of tanh and $\tanh^{-l}$ functions by applying

$$\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}},$$
$$\tanh^{-1}(x) = -\frac{1}{2}\log\left(\frac{1+x}{1-x}\right). \tag{6}$$

Therefore, (2) and (3) are transformed into the following (7) and (8). In these equations, Δ' and Λ' represents log probability ratio; i.e., $\Delta' = \log \Delta$ and $\Lambda' = \log \Lambda$. For the sake of clear representation, we redefine and use Δ and Λ as log probability ratios hereafter. The *a posteriori* probability expressed in log domain is robust and can be represented with a small number of bits while maintaining coding gain, leading to an area-efficient implementation. Detailed explanation about the *a posteriori* log-likelihood ratio can be found in [20]

$$\Lambda'_{ij} = \sum_{j' \in N(i)\backslash j} \log\left(\tanh\left(-\frac{\Delta'_{j'i}}{2}\right)\right) \tag{7}$$

$$\Delta'_{ji} = \log\left(\frac{p(r_j|+1)}{p(r_j|-1)}\right) - \sum_{i' \in M(j)\backslash i} 2\cdot\tanh^{-1}(\exp(\Lambda'_{i'j})). \tag{8}$$

Second, a close look at the iterative decoding algorithm tells that the VTC (CTV) operation can be broken to a summation of all the elements in the same row (column) and a subtraction of its own value from the summation as expressed below. Therefore, (7) is divided into the row summation in (9) and the subtraction in (10). Similarly, (8) is divided into (11) and (12)

$$\Lambda_i = \sum_{j \in N(i)} \log\left(\tanh\left(-\frac{\Delta_{ji}}{2}\right)\right) \tag{9}$$

$$\Lambda_{ij} = \Lambda_i - \log\left(\tanh\left(-\frac{\Delta_{ji}}{2}\right)\right) \tag{10}$$

$$\Delta_j = \log\left(\frac{p(r_j|+1)}{p(r_j|-1)}\right) - \sum_{i \in M(j)} 2\cdot\tanh^{-1}(\exp(\Lambda_{ij})) \tag{11}$$

$$\Delta_{ji} = \Delta_j + 2\cdot\tanh^{-1}(\exp(\Lambda_{ij})). \tag{12}$$

III. Proposed LDPC Decoding Architecture

Since the functions of variable nodes and check nodes are not complex compared to other elaborate codes such as turbo

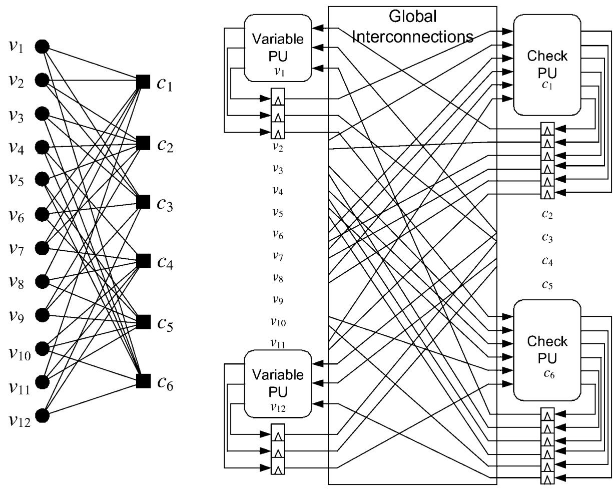


Fig. 3. Fully parallel LDPC decoding architecture.

codes, the main issue in implementation is how to exchange the large number of messages with other nodes. As opposed to the previous implementations that solve the problem by providing complex wire interconnections, this paper proposes a new architecture based on loosely coupled data flows to reduce the interconnection complexity.

A. Previous Architectures

Fig. 3 shows a fully parallel LDPC decoding architecture for the (12, 3, 6) LDPC code. The fully parallel architecture is inherited from the message passing algorithm and all the operations required in each node are implemented as a PU. The number of PUs is as many as the number of the variable and check nodes. For the sake of clarity, only two variable PUs and two check PUs are depicted in Fig. 3, where a check PU at the right-hand side denotes a check node that computes 6 Λ_{ij} messages using 6 Δ_{ji} messages transferred from variable nodes, and a variable PU at the left-hand side corresponds to a variable node computing three Δ_{ji} messages in a similar way. Computed messages are stored into flip-flops located under the PUs for the next computation.

The PUs calculate (9) and (10), or (11) and (12) using look-up tables and self-exclusive adder trees as depicted in Fig. 4, where the functions of LT and AE are $-\log(\tanh(x/2))$ and $2\cdot\tanh^{-1}$ (exp $(-x)$), respectively.

Although the fully parallel implementation of the message passing algorithm is straightforward and results in a high throughput decoder, it faces with complex interconnections caused by a quite large number of irregular edges. Complex interconnections are required to sum up the Λ_{ij} and Δ_{ji} messages that are calculated in the PUs spread over the chip area. The complexity of the interconnection can be easily inferred from the number of edges in Fig. 3 by expanding the factor graph for more than one thousand nodes. Consequently the global interconnections, denoted by a white box in the middle of Fig. 3, become complex and long.

The interconnection complexity can be reduced by employing the serial architecture in which a shared PU computes all the rows or columns one after another as shown in Fig. 5.

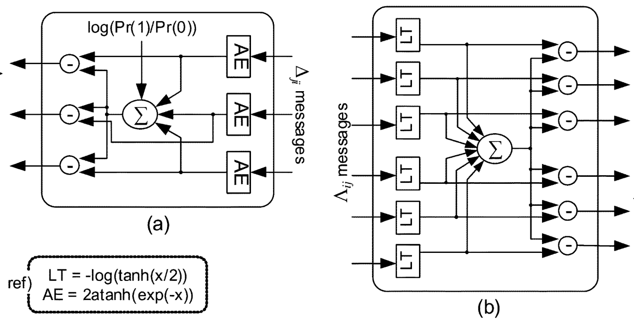


Fig. 4. PUs. (a) Variable PU. (b) Check PU.

Whereas the fully parallel architecture computes all the messages simultaneously, the serial architecture computes messages row-by-row or column-by-column because there is only one PU for each step. Therefore, variable messages calculated by a variable PU is stored into a memory and accessed later by a check PU, and vice versa. In this architecture, the Λ_{ij} or Δ_{ji} messages are stored in a memory, but aligned in different ways. The read operation is relatively simple because a PU can read a corresponding row or column through multiplexers. The write operation, however, is not so simple because computed messages are written to the other memory aligned in a different manner. Many multiplexers are required for the write access, which are as many as the number of messages computed at a time. Thus, the complex interconnection problem is transformed to how to read and write the messages aligned in different ways, requiring complex index generation to access the storage elements.

B. Proposed Loosely Coupled Processing

To resolve the complex interconnection problem, the proposed architecture does not exchange the variable and check messages, Δ_{ji} and Λ_{ij}, between check and variable nodes. Instead, it delivers only the row and column summation values, Δ_j and Λ_i defined in (9) and (11), to the neighbor nodes as shown in Fig. 6.

To derive the individual messages from the summation value, a PU has to include additional operations computed in the neighbor PUs in the previous architecture. Equations (9) to (12) are restructured to reflect the additional operations. Given the column summation values, Δ_j, a check PU recovers individual messages and generates the next row summation value Λ_i to be delivered to the variable PUs. And then it computes the intermediate values, δ_{ij}, to be used to recover the individual messages from the next column summation values and stores them into the local flip-flops. The following equations stand for the detailed operation of the check PU

$$\Delta_{ji} = \Delta_j - \delta_{ij} \tag{13}$$

$$\Lambda_i = \sum_{j\in N(i)} \log\left(\tanh\left(-\frac{\Delta_{ji}}{2}\right)\right) \tag{14}$$

$$\Lambda_{ij} = \Lambda_i - \log\left(\tanh\left(-\frac{\Delta_{ji}}{2}\right)\right) \tag{15}$$

$$\delta_{ij} = -2\cdot\tanh^{-1}(\exp(\Lambda_{ij})). \tag{16}$$

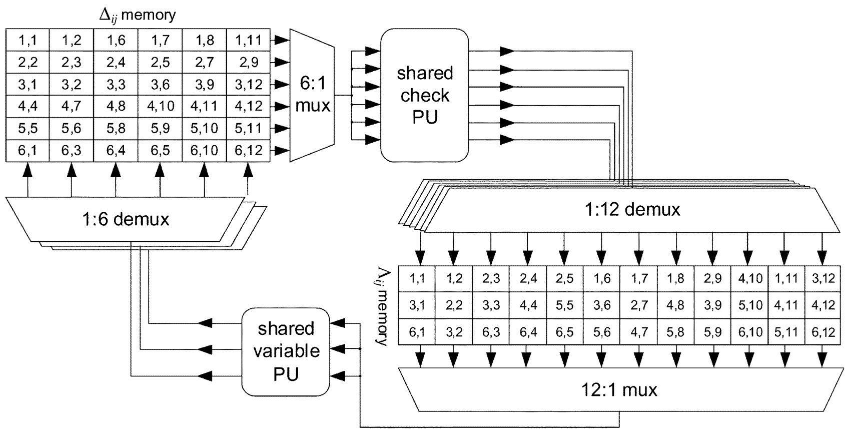


Fig. 5. Serial LDPC decoding architecture.

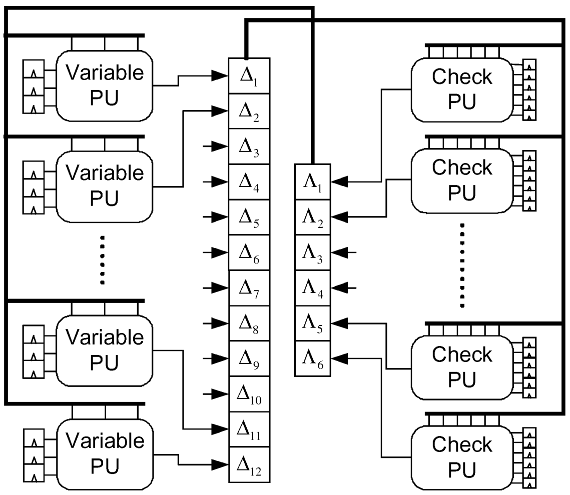


Fig. 6. Proposed architecture based on loosely coupled data flows.

Similarly, the variable PU recovers individual messages and generates column summation value Λ_i to be delivered to the check PUs. In this case, λ_{ji} is the intermediate value

$$\Lambda_{ij} = \Lambda_i - \lambda_{ji} \tag{17}$$

$$\Delta_j = \log\left(\frac{p(r_j| + 1)}{p(r_j| - 1)}\right) - \sum_{i\in M(j)} 2\cdot\tanh^{-1}(\exp(\Lambda_{ij})) \tag{18}$$

$$\Delta_{ji} = \Delta_j + 2\cdot\tanh^{-1}(\exp(\Lambda_{ij})) \tag{19}$$

$$\lambda_{ji} = \log\left(\tanh\left(-\frac{\Delta_{ji}}{2}\right)\right). \tag{20}$$

Note that the intermediate values are not Δ_{ji} and Λ_{ij}, but δ_{ij} and λ_{ji} defined in (16) and (20). Detailed block diagrams of the variable PU and check PU are depicted in Fig. 7. Compared to the previous architecture, the PUs in the proposed architecture have both LT and AE look-up tables and additional subtractions for the added (13), (16), (17) and (20). For the (12, 3, 6) LDPC code, a variable PU reads 3 λ_{ji}'s from the local storage and gets the corresponding row summation values to compute a column summation value Δ_j to be delivered to the check nodes, and then it computes 3 new λ_{ji}'s for the next iteration. The check PU does the similar processing for 6 δ_{ij}'s. As δ_{ij}, and λ_{ji} are not delivered to other type of PUs, the number of interconnections is reduced significantly.

In the proposed architecture, (13) to (16) consist of a data flow of the check node and (17) to (20) consist of another data flow of the variable node. The two data flows are symmetrical and each has whole processing operations except the row or column summation. As only the row and column summation values, Δ_j and Λ_i, are exchanged, the dependence between the two data flows become loose at the cost of duplicated operations. This loosely coupled architecture alleviates the interconnections required for exchanging Λ_{ij} and Δ_{ji} messages aligned in different manners. The duplicated operations increase the hardware complexity of PUs. This overhead is inevitable to reduce the interconnection complexity that is more serious than the logic complexity in today's deep submicron technology. To reduce the overhead, a partially parallel architecture is presented in the next section.

IV. Area-Efficient Partially Parallel Decoding Architecture

As the intermediate values, δ_{ij} and λ_{ji}, are stored in the local storage, the proposed architecture can exploit the partially parallel architecture in which each PU takes in charge of several number of rows or columns. As a PU is shared for a number of rows or columns, the number of PUs becomes much smaller than that of the fully parallel architecture, as shown in Fig. 8. In addition, since a PU processes a row or column at a time, the intermediate values, δ_{ij} and λ_{ji}, processed by a PU can be grouped and stored into a local memory instead of flip-flops to save area. This architecture can reduce the number of interconnections further by sharing the multiplexers required to access the row and column summation values.

Though the partially parallel architecture looks similar to the previous serial architecture, its write interface is much simpler than that of the serial architecture. Since the intermediate values δ_{ij} and λ_{ji} stored in a local memory are accessed by only a PU, they can be calculated regardless of their alignment in the local memory. Moreover the resulting intermediate values are written back to the positions they are read from. Thus, the index generation required for writing the results to the other memory in the serial architecture is no longer necessary in the partially parallel architecture.

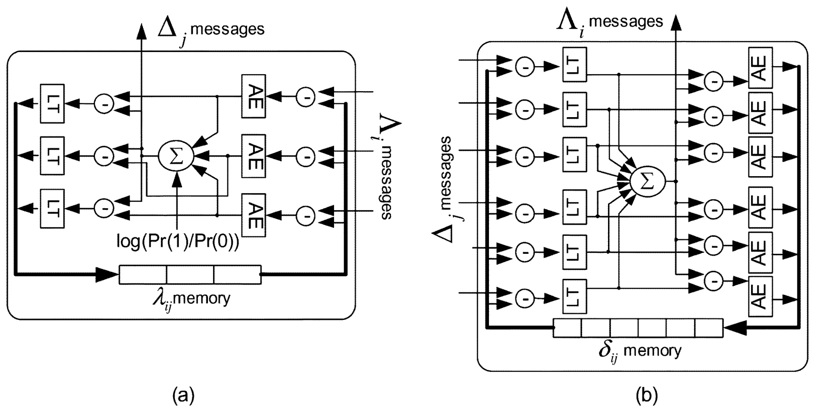


Fig. 7. PUs for the proposed LDPC decoder. (a) Variable PU. (b) Check PU.

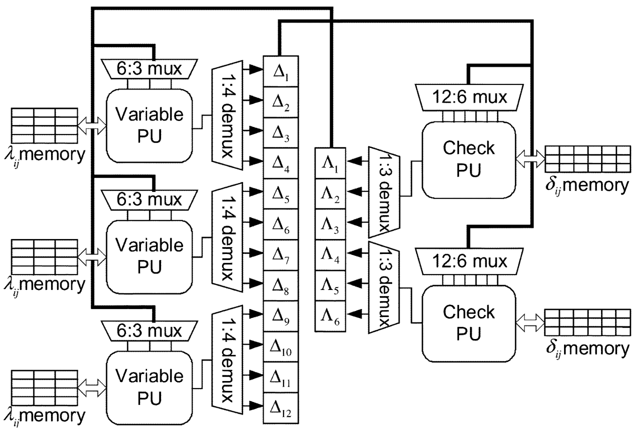


Fig. 8. Partially parallel LDPC decoding architecture.

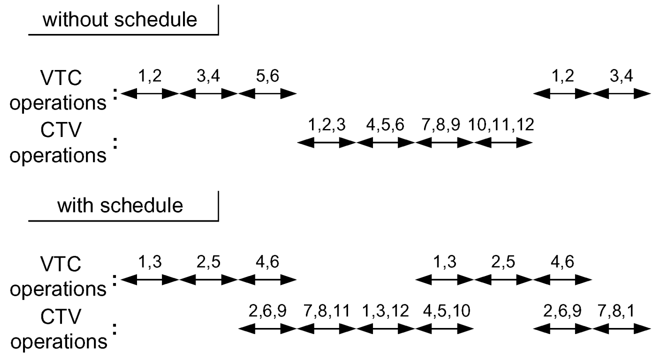


Fig. 9. Example of overlapped processing.

A. Overlapped Processing

As described in Section II, the VTC and CTV operations are parallel in nature and thus a PU can process any rows or columns regardless of their order in matrix $\mathbf{H}$. Since a PU takes in charge of several rows or columns in the partially parallel architecture, we have to determine which rows or columns are processed in the PU. In the grouping, the dependencies between rows and columns should be considered to minimize overall cycles by overlapping the VTC and CTV operations. For example, the CTV operation for v_1 in the (12, 3, 6) LDPC code requires the VTC operations for c_1, c_3 and c_6 to be completed beforehand. In addition, it is important to determine the processing order of rows or columns for a PU, because the dependence of the δ_{ij} and λ_{ji} can hinder the overlapped processing.

Traditionally, the CTV operations start only after the entire VTC step finishes completely, and vice versa. A well-scheduled sequence of the message calculations can reduce the latency, because the CTV step can start in the course of the VTC step if the corresponding row summation values are available, and vice versa. The overlapped processing of the VTC and CTV steps results in a reduced number of cycles.

Fig. 9 shows an example scheduling for the (12, 3, 6) LDPC code, assuming that the numbers of check PUs and variable PUs are two and three, respectively. Each arrow denotes a clock cycle and the numbers above an arrow indicate the row or column indexes processed at that cycle. If the VTC and CTV steps are performed according to the increasing order of indexes without scheduling, no overlapped processing is possible. Though two CTV operations for variable nodes of index 2 and 7 can start at the last cycle of the VTC step, CTV step does not start until three CTV operations are enabled for easy control design. If the VTC operations are scheduled as shown in Fig. 10(b), three CTV operations for variable nodes of index 2, 6 and 9 can start processing at the last cycle of the VTC step. Furthermore, the CTV operations can be scheduled to enable the VTC operations of the next iteration to start earlier. In this example, the scheduling of the VTC and CTV operations saves two cycles.

The scheduling process can be viewed as the row and column permutation of the parity check matrix $\mathbf{H}$. Fig. 10(a) is the original parity check matrix $\mathbf{H}$ whose size is $M \times N$. In this case, the size of $\mathbf{H}$ is 6×12 and we assume 2 check PUs and 3 variable PUs. If there are m check PUs, m rows at the top of the matrix are processed in the first cycle, the next m rows in the second cycle, and so on. Therefore, the VTC step takes $\lceil M/m \rceil$ cycles where $\lceil x \rceil$ means the smallest integer that is greater than or equal to x. Similarly, the CTV step takes $\lceil N/n \rceil$ cycles if there are n variable PUs. The permutation changes the sequence of the VTC and CTV operations to enlarge the empty slots at the lower left and upper right corners as shaded in Fig. 10(b).

The empty slots in a column mean that the processing of the column can start as long as the rows above the empty slots are processed beforehand. If there are k empty slots in a column, the column processing can start $\lfloor k/m \rfloor$ cycles earlier, and if the first n columns have more than m empty slots, all the variable PUs can start their processing one cycle earlier. The simultaneous

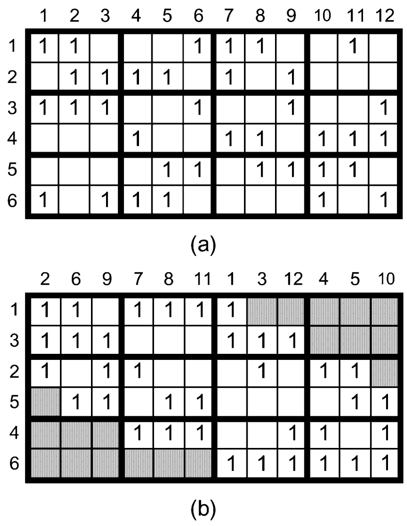


Fig. 10. Scheduling by the permutation of matrix H. (a) Matrix H. (b) Permuted matrix.

early starting is possible if the $m \times n$ slots that are at the lower left part surrounded by thick lines are all empty, and it is required to achieve an easy control design. For example, as three columns have empty slots larger than two in Fig. 10(b), they can start after two cycles of the VTC step. Similarly, the empty slots in a row mean that the processing of the row can start as long as the columns at the left-hand side of the empty slots are processed beforehand, leading to two rows that can start after three cycles of the CTV step.

In fact, the earliest cycle found by the permutation is not the real starting time of the CTV (VTC) step. For the sake of easy control design, we assume that the CTV (VTC) step performs continuously without skipping cycles once it starts. If there is a cycle in which any column (row) cannot be processed during the CTV (VTC) step, the former operations are delayed to achieve continuous CTV (VTC) operations.

B. Scheduling Algorithm

It takes a considerable amount of time to find an optimum schedule that minimizes the overall cycles because of the large number of rows and columns. There has been a heuristic scheduling algorithm proposed for quasi-cyclic LDPC codes [19], but it cannot be applied to general LDPC codes. We describe in this section a new scheduling algorithm developed for the partially parallel LDPC decoding architecture. The proposed algorithm is based on the concept of the matrix permutation. The algorithm makes the row sequence and column sequence that can result in empty spaces in the lower left and upper right corners of the permuted matrix when the matrix $\mathbf{H}$ is rearranged according to the sequences. The row sequence is made by selecting rows one by one in the proposed algorithm. Before describing the proposed algorithm, there are two terms to be defined first.

Definition I: Common columns (CC) of an unselected row means the number of 1's that are common with the selected rows.

If we assume that the first and third rows in the parity check matrix for the (12, 3, 6) LDPC codes are selected, the CC of the fifth row is four, because the row has four 1's common with the selected rows, the second, eighth, ninth, and eleventh columns.

Definition II: Columns to be completed (CTC) of an unselected row means the number of columns whose $(\gamma - 1)$ 1's are located in the selected rows. This means that all the 1's in those columns are covered if the row is selected.

If all the 1's in a column are included in the already scheduled rows, it can be thought that all the VTC operations which have dependencies with the column are completed. In Fig. 10(b), if we assume that the first, third and second rows are already selected, the CTC of the fifth row is two since all the 1's in the 6th and ninth columns are included in the scheduled rows by appending the fifth row to the set of scheduled rows.

The proposed scheduling algorithm is as follows.

Step 1) Row sequence = null, column sequence = null.
Step 2) Order row indexes randomly and pick the first one. Then go to Step 4.
Step 3) Count CC and CTC for unselected rows. Then pick a row r_i that has the maximal CTC. If there are more than two rows associated with the maximal CTC, select the one that has the largest CC.
Step 4) Append columns which have 1 at the just selected row but not selected yet to the column sequence.
Step 5) If there are columns that are newly completed by selecting the rows, move them right after the already completed columns. Uncompleted columns are shifted right by the number of the newly completed columns.
Step 6) If there are unselected rows go to step 3. Otherwise go to Step 7.
Step 7) If there are m check PUs, assign the first m rows at the top of the permuted matrix to the first cycle of the VTC step, the next m rows to the second cycle, and so on. Similarly, for n variable PUs, assign n columns from the left of the permuted matrix to each cycle of the CTV step.
Step 8) Calculate the start point of the CTV and VTC operations for easy control design.

The random ordering of row indexes in Step 2 is to explore more search space by starting from a different row and giving priority to a different row when there are many rows that have the same CTC and CC. Steps 3 to 5 are the main loop of matrix permutation and Steps 7 to 8 assign rows and columns to PUs considering easy control design. In the algorithm, rows are selected to make the upper right part of the permuted matrix as empty as possible and completed columns are moved to the left in order to make enlarged empty space in the lower left part. An example of the algorithm is illustrated in Fig. 11. By applying the proposed scheduling algorithm to various LDPC codes, we can save the number of processing cycles per iteration as summarized in Table I, where parallelism (m, n) stands for an implementation associated with m check PUs and n variable PUs.

In Table I, there are three types of LDPC codes. One is from Mackay's Encyclopedia of Sparse Graph Codes [16]. They are regular LDPC codes constructed systematically for the binary symmetric channel [18]. Another is constructed using Neal's software [17]. Basically, it tries to locate 1's randomly on the parity check matrix, maintaining the number of checks per row approximately uniform. Resulted LDPC codes are irregular. The

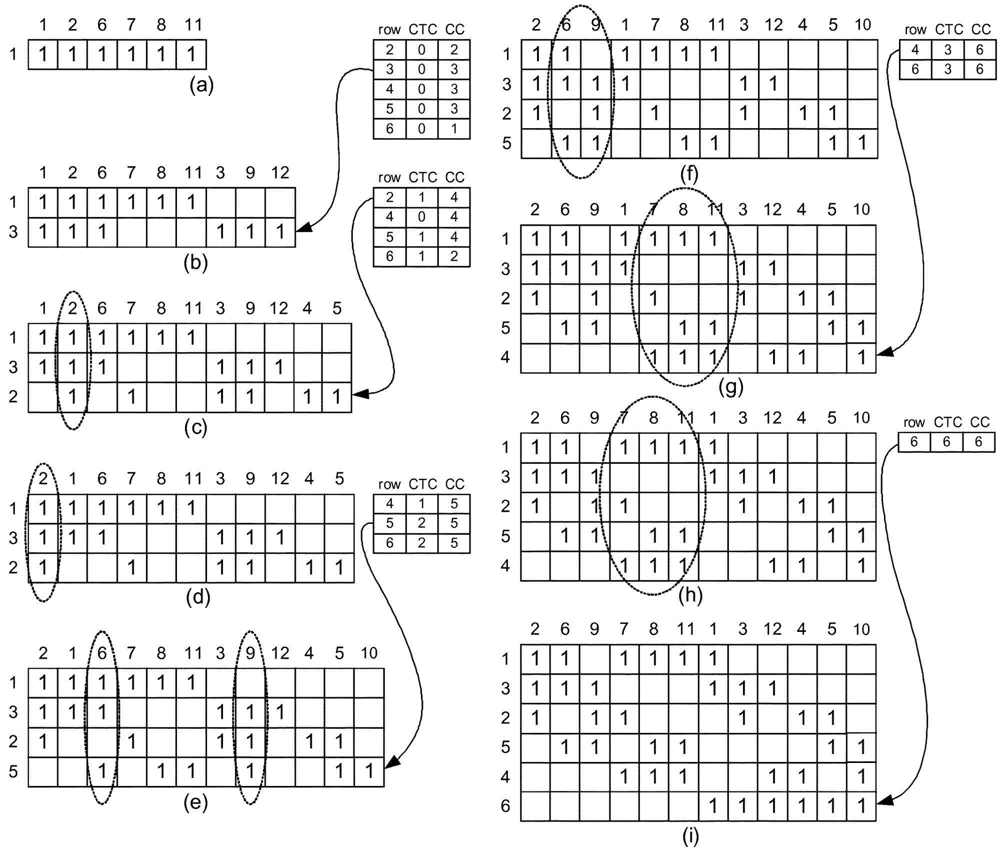


Fig. 11. Illustrative example of the proposed scheduling. (a) Select r_1. (b) Append r_3. (c) Append r_2. (d) Move c_2. (e) Append r_5. (f) Move c_6, c_9. (g) Append r_4. (h) Move c_7, c_8, c_{11}. (i) Append r_6.

other is also a kind of randomly constructed LDPC codes. The parity check matrix is extended by appending a column whose 1's locations are randomly generated under the constraints of the given row and column weights and no 4-cycle. For three types of LDPC codes, we generate several codes by varying code length, column and row weights and code rate.

For these codes, the proposed scheduling algorithm reduces the number of cycles per iteration by 36.7% on the average. The third and forth columns represent the number of cycles per iteration obtained without and with overlapped processing. The fifth and sixth columns indicate the number of reduced cycles resulted from the proposed algorithm and its percentage to the cycles without scheduling. The proposed algorithm works well for all the three types of LDPC codes and the percentage of the saved cycles is almost uniform if the code length and row and column weight are given regardless of their construction methods. Table II shows the maximum number of saved cycles that is achieved by 100000 random schedulings which can be used as a performance reference of the proposed scheduling algorithm.

V. Prototype Implementation

There are two things to be considered in implementing an LDPC decoder based on the proposed architecture. First, we have to determine the number of check and variable PUs. The number of PUs to be employed in implementation should be determined by exploration, because the number of overlapped cycles and the hardware complexity are not linearly proportional to the number of PUs as shown in Table I. Therefore, a tradeoff between the area and the processing time has to be investigated for a number of parallelisms.

Fig. 12 shows experimental results for a (1024, 3, 6) LDPC code. The first bar in the graph represents the total gate count including both the PUs and the δ and λ memories implemented with dual port SRAMs. The second bar represents the total processing cycles considering the dependence between the row and column summation values. The solid line represents the throughput per gate, which is a measure of the tradeoff between throughput and hardware cost.

The results show that more parallelism delivers higher throughput because the second bar becomes shorter as the parallelism increases. The fully parallel architecture ensures the highest throughput if the area is not restricted. In the dual port SRAM used in this experiment, there is a restriction that the minimum number of words is 32. This constraint prevents higher parallelism because the number of nodes computed by a PU decreases as the number of PUs increases. In other words, the height of SRAM becomes shorter. Due to the minimum number of memory words of the SRAM and the peripheral circuit overhead such as address decoders and sense amplifiers, the gate count increases exponentially for more than parallelism (16, 32). Since on-chip memory is fast enough to be accessed twice during one cycle, two local memories can be merged into a large-sized local memory being accessed by two PUs. With this technique, more parallelism can be exploited. For more than parallelism (32, 64), however, some part of the SRAM starts to be vacant due to the minimum height. Consequently,

TABLE I
CYCLES SAVED BY SCHEDULING PER ITERATION

Code	Parallelism	Cycles w/o scheduling	Cycles w/ scheduling	Saved cycles	Saved cycles (%)
816.3.6.MK	(17, 34)	48	29	19	39.6
	(34, 51)	28	18	10	35.7
	(34, 68)	24	15	9	37.5
816.4.6.MK	(17, 34)	56	35	21	37.5
	(34, 51)	40	27	13	32.5
	(34, 68)	28	18	10	35.7
1008.3.6.MK	(24, 56)	39	24	15	38.5
	(28, 56)	36	22	14	38.9
	(36, 63)	30	19	11	36.7
816.3.6.N	(17, 34)	48	28	20	41.7
	(34, 51)	28	18	10	35.7
	(34, 68)	24	15	9	37.5
1024.3.6.N	(16, 32)	64	38	26	40.6
	(32, 32)	48	33	15	31.3
	(32, 64)	32	20	12	37.5
1024.4.8.N	(16, 32)	64	41	23	35.9
	(32, 32)	48	34	14	29.2
	(32, 64)	32	21	11	34.4
816.3.6.r4	(17, 34)	48	28	20	41.7
	(34, 51)	28	18	10	35.7
	(34, 68)	24	15	9	37.5
1024.3.6.r4	(16, 32)	64	38	26	40.6
	(32, 32)	48	32	16	33.3
	(32, 64)	32	20	12	37.5
2048.3.6.r4	(32, 64)	64	39	25	39.1
	(64, 64)	48	33	15	31.3
	(64, 128)	32	20	12	37.5
Average	-	-	-	-	36.7

TABLE II
NUMBER OF CYCLES SAVED BY RANDOM SCHEDULING

Code	Parallelism	Saved cycles	
		Max.	Min.
1008.3.6.MK	(24, 56)	2	0
	(28, 56)	2	0
	(36, 63)	1	0
1024.3.6.N	(16, 32)	4	0
	(32, 32)	2	0
	(32, 64)	1	0
1024.3.6.r4	(16, 32)	3	0
	(32, 32)	2	0
	(32, 64)	1	0

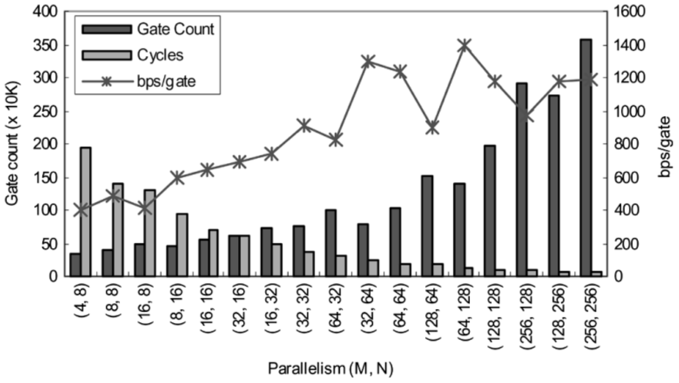


Fig. 12. Hardware cost versus processing cycles.

excessive parallelism does not provide the optimal design in terms of area.

The second thing to be considered is how to implement look-up tables. Since for a PU the number of look-up tables required is two times as many as the row or column weight, it is important to make the table efficient in the sense of area. As mentioned in Section III, the table functions are $-\log(\tanh(x/2))$ and $2 \cdot \tanh^{-1}$ (exp $(-x)$). The two functions are identical if x is greater than 0 and can be implemented with memories or simple combinatorial logics because the log-domain probability needs an arithmetic precision of just 3–5 bits for good coding performance. As $f(x) = -\log(\tanh(x/2)) = 2 \cdot \tanh^{-1}$ $(\exp(-x))$ is positive for a positive input x and $f(-x) = -f(x)$, the signed magnitude representation is more efficient than the 2's complement representation. On the other hand, as the 2's complement representation is efficient for the self-exclusive adder trees and the subtraction required for calculating the individual messages,

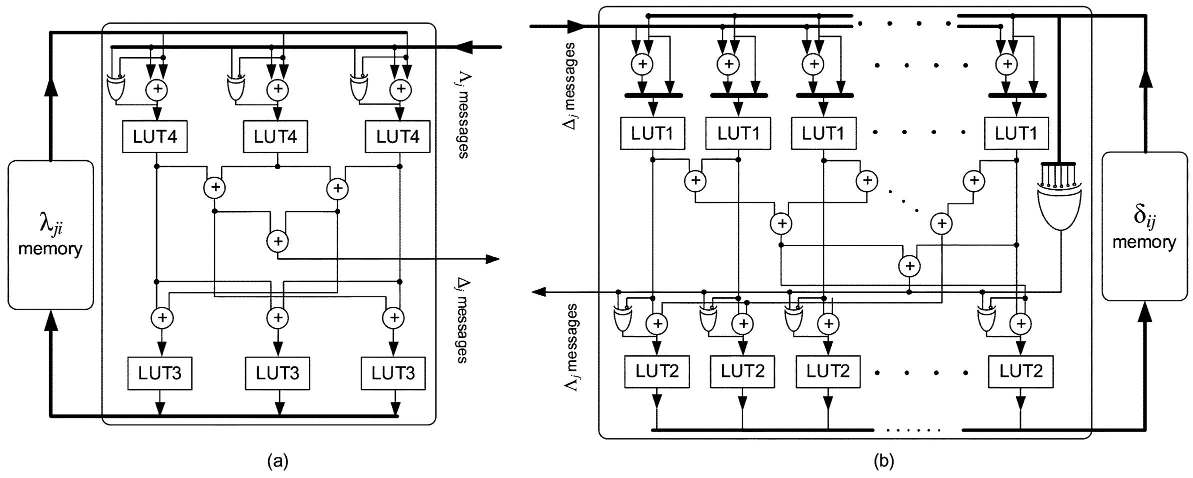


Fig. 13. Detailed PUs of the proposed LDPC decoder. (a) Variable PU. (b) Check PU.

TABLE III
TYPES OF LOOK-UP TABLES

Name	Function	Input		Output	
		Size	Representation	Size	Representation
LUT1	*LT*	n bits	2's complement	n-1 bits	Signed magnitude
LUT2	*AE*	n bits	Signed magnitude	n bits	-(2's complement)
LUT3	*LT*	n bits	2's complement	n bits	-(2's complement)
LUT4	*AE*	n bits	2's complement	n bits	2's complement

we have to convert the signed magnitude representation to the 2's complement representation. To minimize the area and delay, these conversion logics can be merged with the function $f(x)$ while maintaining the look-up table for the positive input. Therefore, four different types of look-up tables are implemented to take into account the operations performed before and after the function and the sign bits are processed separately, as described in Fig. 13. The functions and input/output representations of look-up tables are summarized in Table III.

In the check PU, LUT1 gets an n-bit 2's complement input and generates an $(n-1)$-bit output that is the magnitude of the n-bit signed magnitude representation, because $f(x) = -\log(\tanh(|x|/2)$ is always positive and the sign that is separately processed is combined before accessing LUT2 whose function is $2 \cdot \tanh^{-1}(\exp(-|x|))$. LUT2 gets an n-bit signed magnitude input and generates an n-bit 2's complement output whose value is negated for the subtraction located before LUT1. The adder in Fig. 13(b) is to subtract δ_{ij} stored in the local memory from the column summation values. Thus, LUT2 convert the output in advance considering its sign bit. The negation is to enhance the processing speed, as addition is usually faster than subtraction. The input and output representation of LUT3 and LUT4 are determined similarly.

VI. EXPERIMENTAL RESULTS

We designed a (1024, 3, 6) LDPC decoder based on the proposed partially parallel architecture. As plotted in Fig. 14, the overall architecture is divided into two parts according to δ and λ memory. The intermediate values calculated in a PU are stored in the corresponding local memory, while row and column summation values are stored in two one-dimensional arrays of D flip-flops for instant and simultaneous access. To feed the summation values to other type of PUs, multiplexers are used. Since δ_{ij} and λ_{ji} are stored in the local memory in a scheduled sequence, PUs can read and write the intermediate values to calculate the summation by simply incrementing the address of the local memory. Each column summation value is bit-by-bit decoded according to its sign bit to determine the final output.

For parallelism (16, 32) and (32, 64), we designed two (1024, 3, 6) LDPC decoders using a 0.18-μm 4-Metal CMOS process. The performances of the decoders are summarized in Table IV. The first decoder corresponding to parallelism (16, 32) has a synthesized area of 6.29 mm^2 and is simulated correctly at the frequency of 200 MHz, resulting in more than 500 Mbps decoding throughput. The second decoder corresponding to parallelism (32, 64) occupies an area of 10.08 mm^2 and provides almost 1 Gbps decoding throughput at the same frequency. The performance of the second decoder is comparable to the fully parallel architecture proposed by Blanksby *et al.*, but the proposed decoders achieve significant area reduction. Fig. 15 shows the die photo of the second decoder whose parallelism is (32, 64). Although the iteration number in Blanksby's implementation is fixed to 64 due to the scan-chain like I/O mechanism, there is no restriction in the proposed architecture. We chose eight iterations to provide sufficient bit-error rate (BER) performance.

VII. CONCLUSION

This paper has presented a new LDPC decoding architecture proposed to relax the interconnection complexity and reduce

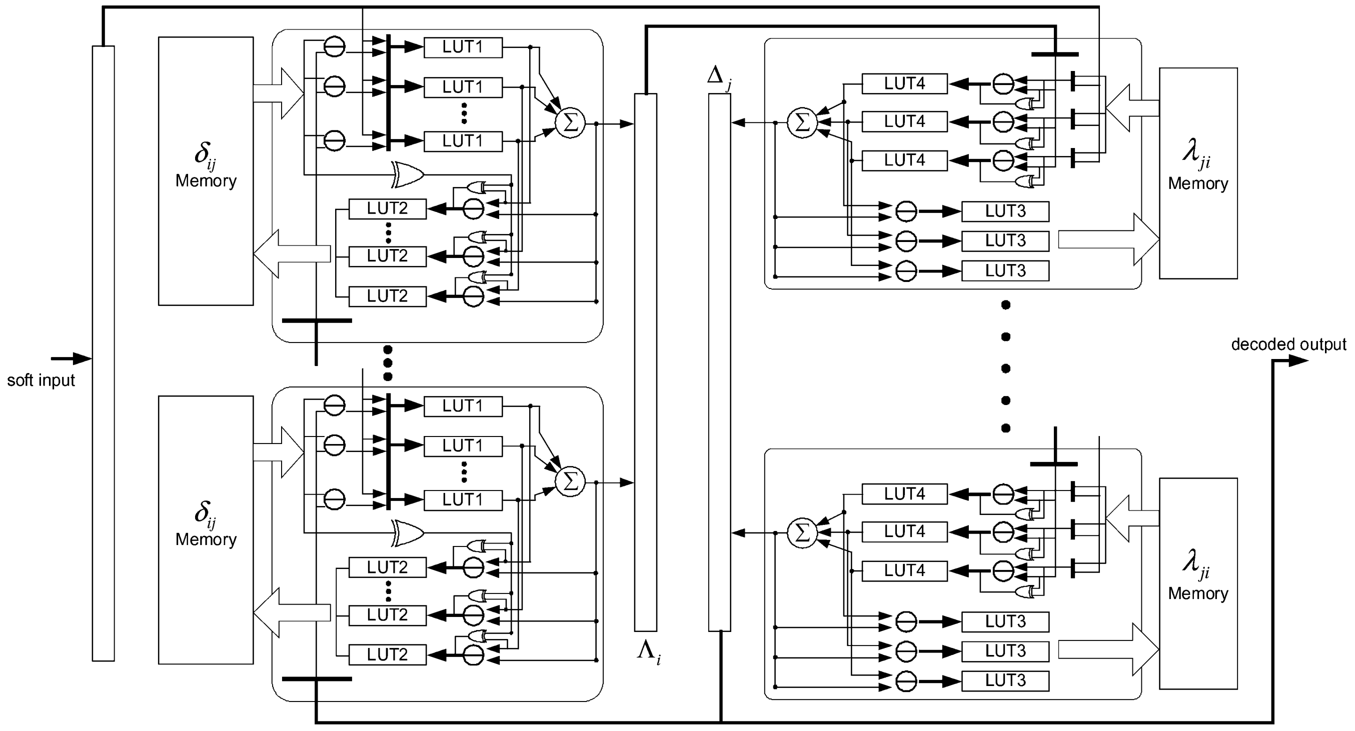


Fig. 14. Block diagram of the proposed LDPC decoder.

TABLE IV
COMPARISION OF LDPC DECODERS

	Blanksby [4]	Proposed	
		Parallelism (16,32)	Parallelism (32,64)
Technology	0.16 um	0.18 um	0.18 um
Bit rate	1 Gbps	582 Mbps	985 Mbps
Frequency	64MHz	200MHz	200MHz
Gate counts	1750K	457K	543K
Area	52.5 mm^2	6.29 mm^2	10.08 mm^2

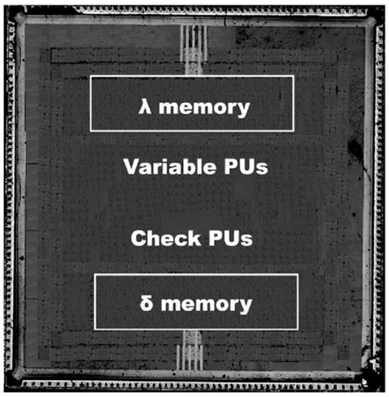


Fig. 15. Die photo of the proposed LDPC decoder chip.

area. In the proposed architecture, only the row and column summation values are exchanged among check and variable PUs to reduce the interconnection complexity. To recover the individual messages from the summation values, the function of a PU is restructured to include some operations that are traditionally performed in the neighbor nodes. In addition, intermediate values accessed by a PU are grouped and stored into a local memory instead of registers, since other PUs do not access them. The memory-based architecture is area-efficient in the sense that the memory takes much less area compared to the register if both have the same size. We have extended the proposed architecture for the partially parallel architecture in order to further reduce area by increasing memory usage, and have proposed an efficient algorithm that schedules the processing order of the partially parallel architecture to reduce the overall processing time by overlapping CTV and VTC steps. It saves the number of cycles per iteration by 36.7% on the average and is applicable to both regular and irregular LDPC codes.

To verify the proposed architecture, a (1024, 3, 6) LDPC decoder is implemented using a 0.18-μm CMOS process. The decoder occupies an area of 10.08 mm^2 and runs correctly at the frequency of 200 MHz, resulting in almost 1 Gbps decoding throughput. Compared to the previous architecture, the proposed decoder occupies almost one fifth of area. Since the proposed architecture is highly regular, it is easy to scale up and down the parallelism according to the system specification.

REFERENCES

[1] R. G. Gallager, "Low density parity check codes," *IRE Trans. Inf. Theory*, vol. IT-8, pp. 533–547, Jan. 1962.

[2] S. Chung, 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 Comm. Lett.*, vol. 5, pp. 58–60, Feb. 2001.

[3] T. Richardson, M. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 619–637, Feb. 2001.

[4] A. Blanksby and C. Howland, "A 690-mW 1-Gb/s, rate-1/2 low-density parity-check code decoder," *IEEE J. Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, Mar. 2002.

[5] E. Yeo, P. Pakzad, B. Nikolić, and V. Anantharam, "VLSI architectures for iterative decoders in magnetic recording channels," *IEEE Trans. Magn.*, vol. 37, pp. 748–755, Mar. 2001.

[6] E. Eleftheriou and S. Ölçer, "Low-density parity-check codes for digital subscriber lines," in *Proc. IEEE Int. Conf. Commun.*, May 2002, pp. 1752–1757.

[7] P. Urard *et al.*, "A 135 Mb/sDVB-S2 compliant codec based on 64 800 b LDPC and BCH codes," in *Proc. IEEE Int. Solid-State Circuits Conf.*, Feb. 2005, pp. 446–447.

[8] V. Sorokine, F. R. Kschischang, and S. Pasupathy, "Gallager codes for CDMA applications—Part I: Generalizations, constructions and performance bounds," *IEEE Trans. Commun.*, vol. 48, no. 10, pp. 1660–1668, Oct. 2000.

[9] B. Lu, X. Wang, and K. R. Narayanan, "LDPC-based space-time coded OFDM systems over correlated fading channels: Performance analysis and receiver design," *IEEE Trans. Commun.*, vol. 50, no. 1, pp. 74–88, Jan. 2002.

[10] K. Lo, "Layered space–time structures with low density parity check and convolutional codes," M.S. thesis, School of EIE, Univ. of Sydney, Sydney, Australia, 2001.

[11] R. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inf. Theory*, vol. IT-27, pp. 533–547, Sep. 1981.

[12] F. R. Kschischang, B. J. Frey, and H. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 498–519, Feb. 2001.

[13] F. R. Kschischang, "Iterative decoding of compound codes by probability propagation in graphical models," *IEEE J. Sel. Areas Commun.*, vol. 16, pp. 219–230, Feb. 1998.

[14] M. M. Mansour and N. R. Shanbhag, "On the architecture-aware structure of LDPC codes from generalized Ramanujan graphs and their decoder architecture," in *Proc. 37th Annu. Conf. Inf. Sci. Syst.*, Mar. 2003, pp. 215–220.

[15] ——, "High-throughput LDPC decoders," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, pp. 976–996, Dec. 2003.

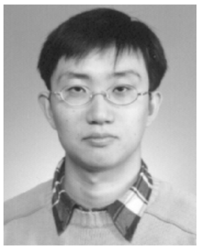
[16] Encyclopedia of sparse graph codes, D. Mackay. [Online]. Available: http://www.inference.phy.cam.ac.uk/mackay/codes/data.html

[17] Software for low density parity check (LDPC) codes, R. M. Neal. [Online]. Available: http://www.cs.utoronto.ca/~radford/ldpc.software.html

[18] D. Mackay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inf. Theory*, vol. 45, no. 3, pp. 399–431, Mar. 1999.

[19] Y. Chen and K. 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.

[20] J. Barry, "Low-density parity-check codes," Georgia Inst. Technology, Atlanta, Tech. Rep., 2001.



Se-Hyeon Kang (S'00) received the B. Tech. degree in electrical engineering from Pohang University of Science and Technology (POSTECH), Pohang, Korea, in 2000, the M.S. degree in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejon, Korea, in 2002. Currently, he is working toward the Ph.D. degree in the Department of Electrical Engineering and Computer Science at KAIST.

His research interests include VLSI design for communication and signal processing.

In-Cheol Park (S'88–M'92–SM'02) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1986, the M.S. and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejon, Korea, in 1988 and 1992, respectively.

Since June 1996, he has been an Assistant Professor and is now a Professor in the Department of Electrical Engineering and Computer Science at KAIST. Prior to joining KAIST, he was with IBM T.J. Watson Research Center, Yorktown, NY, from May 1995 to May 1996, where he researched on high-speed circuit design. His current research interest includes computer-aided design (CAD) algorithms for high-level synthesis and VLSI architectures for general-purpose microprocessors.

Prof. Park received the Best Paper award at ICCD in 1999, and the best design award at ASP-DAC in 1997.