FULLY PROGRAMMABLE LAYERED LDPC DECODER ARCHITECTURE

Christiane Beuschel and Hans-Jörg Pfleiderer

Institute of Microelectronics, University of Ulm
University of Ulm, Albert-Einstein-Allee 43, 89081 Ulm, Germany
phone: + (49) 731 5026210, fax: + (49) 731 5026222, email: christiane.beuschel@uni-ulm.de

ABSTRACT

In this article we present a fully programmable layered LDPC decoder architecture together with an optimum mapping and scheduling algorithm. In contrast to other designs proposed in the literature, we use one-phase message passing. This allows for the first time the design of a fully programmable layered decoder. The proposed mapping and scheduling algorithm exploits the full parallelism of the architecture at any time for any code, which means that the mapping algorithm achieves collision-free memory access and 100% utilization of the architecture. Compared to existing programmable designs without layered decoding we double the data throughput. The parallelism of the architecture is unconstrained and fully scalable so that hardware cost and data throughput can be exchanged with fine granularity.

1. INTRODUCTION

Low-density parity-check (LDPC) codes are known to perform very close to the Shannon limit [1]. They are proposed for error correction in many current and next generation communication standards, e.g. WiMax [2], Wifi [3], DVB-S2 [4]. Iterative decoding of LDPC codes is based on message passing between check and variable nodes. Today's state of the art LDPC decoders use partly-parallel processing schemes. Between successive updates in the nodes the messages are stored in memory banks. However, access patterns on the stored data are different during check and variable node processing. Thus a key challenge in partly-parallel decoder design is to avoid memory access collisions.

The programmable architectures presented in the literature usually involve heuristic mapping algorithms to resolve memory access collisions, which lead to stall cycles or idle processing units [5], [6]. It is widely unknown that a partly-parallel decoder architecture with collision-free memory mapping for arbitrary LDPC codes exists. However, while the architecture and the mapping presented in [7] can resolve all memory access collisions, it uses two-phase messages passing. In current LDPC decoder designs this approach is not used any more as it reduces the data throughput.

In this article, we present a fully programmable LDPC decoder architecture with one-phase message passing, collision-free memory access, and free choice for parallelism p. The identical hardware can decode any structured or unstructured LDPC code.

For LDPC code design, the error correction performance of a large set of candidate codes has to be simulated to determine the best code. Especially for low bit error rates, software simulations are very time consuming. Thus a hardware accelerator performing the decoding is an important tool to speed up the slow software simulations. Our proposed programmable decoder can be used as such a hardware accelerator as it can decode any LDPC code. Furthermore, no hardware knowledge is necessary to reconfigure the hardware accelerator for a different LDPC code.

With more and more upcoming standards using LDPC codes for error correction, another application for a fully programmable decoder is a multi-standard decoder. Most reconfigurable decoders presented in the literature are restricted to one code class [8]. With the mapping and scheduling algorithm presented in this article it is possible to implement an ASIC decoder core which can decode arbitrary LDPC codes by changing only the initialization of the control memory.

In the following, we present to the best of our knowledge the first fully programmable layered LDPC decoder architecture. The proposed mapping and scheduling algorithm guarantees collision-free memory access for any LDPC code and allows one-phase message passing. The parallelism of the architecture is fully scalable.

The structure of this article is as follows: Section 2 shortly introduces LDPC codes, in Section 3 we present the optimum mapping and scheduling algorithm, and in Section 4 we propose two fully programmable LDPC decoder architectures which match the mapping algorithm: one uses one-phase flooding schedule whereas the other one uses layered decoding. Section 5 compares the proposed architectures against other programmable decoder architectures and finally Section 6 gives the conclusion.

2. DECODING OF LDPC CODES

A binary (N, K) LDPC code is defined by a sparse M x N parity-check matrix H. Equivalently the LDPC code can be described by a bipartite graph with M check nodes corresponding to the rows and N variable nodes corresponding to the columns of H. Check node m is connected to variable node n if $h_{mn} = 1$. $M(n)$ is the set of check nodes that are connected to variable node n and $N(m)$ is the set of variable nodes that are connected to check node m. E is defined as the number edges in the bipartite graph. The vector y is the transmitted codeword with $y_i \in \{-1,+1\}$ and r is the received corrupted codeword. The degree of variable node n is given by $d_{V,n}$. The iterative belief propagation (BP) decoding algorithm can be described by the following equations:

1. Initialization, $l = 0$
   - intrinsic (channel) values for each n

$$\lambda_n = \ln \frac{P(y_n = +1|r_n)}{P(y_n = -1|r_n)}, \quad S_n^{(0)} = \lambda_n \tag{1}$$

   - for each $(m,n) \in \{(i,j)|h_{ij} = 1\}$

$$R_{m\to n}^{(l)} = 0 \tag{2}$$

2. Iteration $l = l+1$
   - check node update for each $(m,n) \in \{(i,j)|h_{ij}=1\}$

$$R_{m\to n}^{(l)} = 2 \cdot \tanh^{-1} \prod_{j\in N(m), j\neq n} \tanh\left(\frac{S_j^{(l-1)} - R_{m\to j}^{(l-1)}}{2}\right) \quad (3)$$

   - variable node update for each $n = 0 \ldots N-1$

$$S_n^{(l)} = \lambda_n + \sum_{i\in M(n)} R_{i\to n}^{(l)} \quad (4)$$

3. Decision: If the maximum number of iterations I is reached, decode for each n $q_n = \text{sign}\left(S_n^{(l)}\right)$, else continue with 2.

3. COLLISION-FREE MEMORY ACCESS

In this section, we give a universally valid solution to the memory collision problem for arbitrary LDPC codes. We present a collision-free mapping and scheduling algorithm for a fully programmable layered LDPC decoder architecture.

3.1 Mapping function

First a mapping function is defined. Given a set

$$V = \{v_0, ..., v_{E-1}\} \quad (5)$$

of E elements v_i where E can be factorized as $E = L \cdot p$, we arbitrarily define two partitions P and P' with

$$P = \{V_0, ..., V_{L-1}\} \quad \text{and} \quad P' = \{V'_0, ..., V'_{L-1}\} \quad (6)$$

where each subset V_i or V'_j contains p elements of V. A mapping function T is defined with

$$T : \{v_0, ..., v_{E-1}\} \mapsto \{0, ..., p-1\} \quad (7)$$

such that the following two conditions are fulfilled for every $k = 0, ..., L-1$ and every $i,\ j = 0, ..., E-1$ with $i \neq j$:

$$\begin{aligned} v_i, v_j \in V_k \quad &\Rightarrow \quad T(v_i) \neq T(v_j) \\ v_i, v_j \in V'_k \quad &\Rightarrow \quad T(v_i) \neq T(v_j). \end{aligned}$$

Thus any two values v_i and v_j which are in the same subset are always mapped to different values. The authors of [7] prove that such a mapping function always exists and furthermore give an algorithm to find it.

3.2 Mapping and scheduling algorithm

We now define a collision-free mapping and scheduling algorithm for one-phase message passing, which is the key for a programmable layered decoder. In contrast to [7] we use dynamic instead of static assignment of values to memory banks. Furthermore, memory access collisions are resolved for the sum values instead of for the extrinsic values.

The partly-parallel decoder architecture with parallelism p uses sequential node processing. In each clock cycle p extrinsic values $R_{m\to n}$ in (3) are updated and a partial update of p sum values S_n in (4) is performed. Each extrinsic value is updated exactly once during a decoding iteration. p extrinsic memory banks are required for the parallel update of p extrinsic values. The extrinsic values are stored in linear order in each memory bank and no memory access collisions occur.

Different from that, each sum value S_n is read and written $d_{V,n}$ times in each iteration. For quasi-cyclic (QC) decoder architectures, collision-free memory access on p sum values in parallel can be guaranteed as for each access always the *same* p sum values are needed in parallel. This is different for a fully programmable decoder architecture: the parity-check matrix is unstructured so that in each clock cycle a "random" combination of p sum values is accessed. In each of the $d_{V,n}$ accesses on sum value S_n, the value is accessed in a *different* combination with other sum values. Therefore we have to assure that each of these "random" combinations of p sum values (defined by the parity-check matrix of the code) can be accessed without memory access collisions. Thus, the p sum values for each "random" combination have to be stored in p different memory banks.

To achieve this, memory access collisions for all possible combinations of sum values for a given code have to be resolved. In the following we present a solution in three steps, which is valid for any structured or unstructured LDPC code:

1. Copy each sum value $d_{V,n}$ times. This means that E sum value copies exist, and each copy is accessed only once for read and once for write in each iteration.
2. Determine a memory bank for each copy using the mapping function from Section 3.1.
3. Remove sum value copies: interpret each copy of a sum value as the same sum value at a different point in time. This means that in the final implementation only the N original sum values are stored and that a sum value resides in different memory banks throughout one iteration.

In the following, these three steps are explained in more detail using an example with parallelism $p = 3$ and an irregular LDPC code with dimensions $N = 8$, $M = 6$ and parity-check matrix

$$H = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}. \quad (8)$$

Step 1): Fig. 1a shows $N = 8$ sum values S_n in the top line. Each S_n is copied $d_{V,n}$ times according to the corresponding variable node degree and arranged in the same way as the parity-check matrix of the code. This is illustrated by the gray squares in Fig. 1a. An overall number of E sum value copies exists. Then an access index for each sum value copy is determined. Therefore we assume that decoding starts with processing of the first $p = 3$ check nodes which correspond to lines 0 to 2 in the matrix and continues with the next p check nodes which correspond to lines 3 to 5. Each check node itself is processed sequentially as in (3). To meet the constraint that in each column each access index is used only once, the order in which the edges of one check node are processed can be modified. The resulting access indices for the sum value copies are given on the right in Fig. 1a.

Step 2): In this step, the mapping function from Section 3.1 is applied. The set V in (5) with cardinality E is defined as the set of all sum value copies. Partition P in (6)

is defined by the access indices: sum value copies with the same access index i form one subset V_i. Partition P' is defined as shown in Fig. 1b: the access indices within each column are permuted such that each index is replaced by the next smaller one and the smallest index is replaced by the biggest index in the column. Subset V_i' of partition P' consists of the sum value copies with the same permuted access index i. The cardinality of each subset V_i or V_i' is $p = 3$. Then the mapping function T from (7) is applied. The range of the function consists of $p = 3$ values which are represented by the shapes square (red), hexagon (green) and circle (blue). A possible solution for the mapping function is shown in Fig. 1c. In the decoder architecture exactly p memory banks for the sum values exist. Each shape (color) corresponds to one memory bank, and it can be seen that copies with the same access index are located in different memory banks. Thus no memory access collisions occur.

Step 3): So far all operations were performed on the E sum value copies. This was necessary to explain how the mapping of sum values to memory banks is performed. In the actual implementation, only N sum values and no copies exist. In the following we explain how all copies can be removed. The upper part of Fig. 1d shows again the sum value copies for partition P with access indices and shapes (colors) for memory banks. In the final system, all sum copies in one column of the matrix correspond to the same sum value. During initialization of the sum memory, each sum value S_n is written to the memory bank where the first read is performed from. This corresponds to the memory bank with the smallest access index in a column as indicated by the arrows in the upper part of Fig. 1d. The first read accesses the three values S_0, S_1, and S_4 according to V_0. It can be seen that different shapes (colors) are assigned to the three values which means that they are located in different memory banks. Then the values are written back to the same memory banks according to V_0'. Next the sum values S_0, S_2, and S_3 are read according to V_1. Also these three values are read from three different memory banks. Writing back is performed in permuted order, e.g. S_0 was read from the "square" memory bank but is written back to the "hexagon" bank, S_2 was read from "hexagon" and is written to "circle" and S_3 was read from "circle" and is written to the "square" bank according to V_1'. Thus after the write operation, the sum values are stored in different banks as before and are prepared for their next read access. The same process continues until all sum values are updated $d_{V,n}$ times. At the end of the iteration, all sum values are again in their original memory banks.

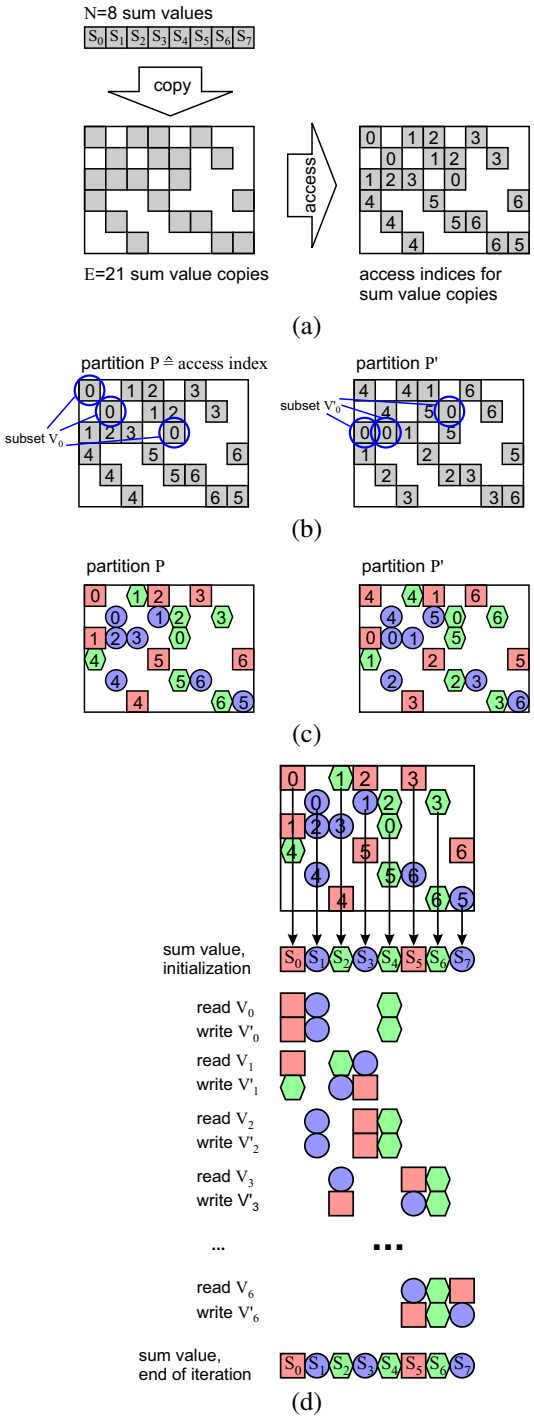

Figure 1: Mapping and scheduling algorithm: illustration of (a) step 1: copies and access indices; (b) and (c) step 2: partitions and mapping function; (d) step 3: removal of copies

4. FULLY PROGRAMMABLE DECODER ARCHITECTURES AND SCHEDULING

In this section we present a fully programmable layered decoder architecture which doubles the data throughput at no additional hardware cost compared with [7]. Using the mapping algorithm described in Section 3.2, an optimum scheduling of operations is achieved which exploits the full parallelism of the architectures at any time for any LDPC code. Before we discuss the details for the layered architecture, we present a one-phase flooding schedule architecture which updates variable nodes during check node processing and thus also doubles the data throughput of [7].

4.1 Flooding schedule decoder architecture

The architecture in Fig. 2 consists of p intrinsic memory banks IM_i for the values λ_n in (1), p extrinsic memory banks EM_i for the extrinsic messages $R_{m\to n}$ in (3), p sum memory banks SM_{Ai} and p sum memory banks SM_{Bi} to store the previous and current total sum S_n in (3), (4). Each memory bank 0 corresponds to the shape "square" in Fig. 1d, bank 1 to "hexagon" and bank 2 to "circle". All memories are dual-

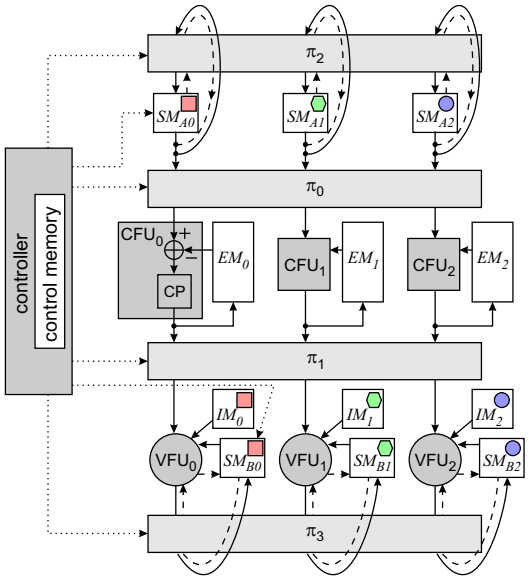

Figure 2: Fully programmable flooding schedule decoder architecture ($p = 3$): dashed lines indicate reversed data paths for even iterations, see Section 4.3

port memories allowing simultaneous and independent read and write access on two data entries. The four permutation networks π_i can be realized as Beneš networks. p check and p variable node functional units (CFU_i resp. VFU_i) perform the node computations, the core function for check node processing is implemented in the CP blocks. In the VFUs the sum of all incoming values is sent to the output. Each CFU and each VFU updates one data value in each clock cycle, thus $2p$ data values are updated in each clock cycle. A controller and a control memory are needed to configure the permutation networks and to generate the addresses for the sum memory banks. A different LDPC code can be decoded by changing the initialization of the control memory.

4.2 Layered decoder architecture

Fig. 3 shows the fully programmable layered decoder architecture. Compared with the flooding schedule decoder architecture, the hardware cost can be significantly reduced, which is consistent with results for QC decoder implementations [9]. Thus we can save the two permutation networks π_2 and π_3 and the reversed data paths as well as the p sum memory banks SM_{Bi} and the p intrinsic memory banks IM_i while we can still perform $2p$ data updates in each clock cycle. Scheduling of check nodes is performed in reverse order in every second iteration. This modified layered schedule increases the convergence speed compared to the flooding schedule.

An inherent constraint for layered decoding is to choose the order of check node processing such that the total sum values in the SM_i memory banks are not needed again before the last update was finished. The constraint can be relaxed by introducing idle states, which reduce the throughput. Simulations using simulated annealing show that in general a solution with less than 5% additional processing time can be found for $p = 16$ by rearranging the order of check node processing.

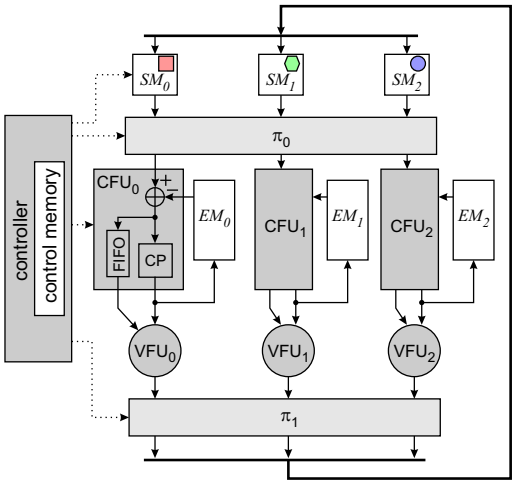

Figure 3: Fully programmable layered decoder architecture ($p = 3$)

4.3 Mapping and scheduling on decoder architecture

Mapping and scheduling are first explained in detail for the flooding schedule decoder architecture and then the results are transfered to the layered architecture. Decoding of a received corrupted codeword starts with initialization of all memories in Fig. 2 according to (1) and (2). The intrinsic value λ_n corresponds to column n in the parity-check matrix and is written to the memory bank indicated in the line "sum value, initialization" in Fig. 1d. In our example this means that λ_0 is written to the "square" banks SM_{A0} and IM_0, λ_1 to the "circle" banks SM_{A2} and IM_2. The address within each memory bank is increased by one for each value which is written to the same memory bank during initialization.

During the decoding iterations, linear addressing is used for the EM memory banks and "random" addressing for the SM banks. p total-sum values which correspond to the copies in V_k are read at time $t = k$ from the SM_{Ai}. Then the sum values are written back over π_2 to the same memory addresses but possibly different banks as indicated by V'_k. E.g. for $t = 1$ according to V_1, V'_1 the values (S_0, S_2, S_3) are read from $(SM_{A0}, SM_{A1}, SM_{A2})$ and written back in permuted order to $(SM_{A1}, SM_{A2}, SM_{A0})$. Each CFU_i sequentially updates the extrinsic messages of one check node, writes the updated messages to the EM_i bank and also sends them to the VFUs. Permutation network π_0 assigns the correct message to each CFU and π_1 performs the inverse permutation of π_0.

In the VFUs the variable node update is performed according to (4). However, the $R_{i\to n}$ values for one variable node do not arrive sequentially but in "random" order. Thus a temporary total sum is read and always one value is accumulated with $S_{n,k+1} = S_{n,k} + R_{i\to n}$ until all $d_{V,n}$ values were added. Reading and writing the total sum values in the SM_{Bi} memory banks is performed in the same order as for the SM_{Ai} banks, only a delay in time is added.

After the update of all extrinsic messages in the first decoding iteration, the second iteration is started by swapping the two sum memories SM_A and SM_B. All operations are now performed in reverse order, meaning that some data paths are inverted as indicated by the dashed lines in Fig. 2 and that the subsets are processed in reverse order starting with V'_{L-1} and ending with V_0. After the second iteration all sum values are

Table 1: Comparison of hardware cost and data throughput

	[5]	[6]	[7]	prop. arch.	
				flooding	layered
perm. net.	$1^{\dagger}$	$2^{\dagger}$	2	4	2
VFUs	p	p	p	p	p
CFUs	p	p	p	p	p
idle FUs	yes	yes	yes	no	no
mem. size, $E = 3.5N$	$N+4E$ $= 15N$	$N+E$ $= 4.5N$	$N+E$ $= 4.5N$	$3N+E$ $= 6.5N$	$N+E$ $= 4.5N$
utilization	$\approx 40\%^{\ddagger}$	$\approx 80\%^{\ddagger}$	100%	100%	100%
updates per clock	$\approx 0.4p^{\ddagger}$	$\approx 0.8p^{\ddagger}$	p	$2p$	$2p$

† significant additional number of multiplexers needed

‡ average value over a set of benchmark codes

again in the original memory banks at the original addresses. The next iterations are processed equivalently.

Processing on the layered decoder architecture can be directly derived from processing on the flooding schedule architecture. A simplification for the layered architecture is that instead of the reversed data paths, different configurations for the permutation networks π_0 and π_1 are used in even and odd iterations.

5. COMPARISON

The presented LDPC decoder architectures together with the mapping and scheduling algorithm achieve collision-free memory access and 100% utilization for arbitrary LDPC codes. Compared to [7] we double the data throughput by using one-phase message passing. Other approaches presented in the literature use heuristic mapping algorithms, which result in a significant number of idle processing units and thus lower the data throughput [5], [6].

Table 1 compares the different architectures. A fair comparison for the number of permutation networks is only possible for the last three designs as the first two designs need a significant number of additional multiplexers. The number of VFUs and CFUs is the same for all designs. However, in [5], [6] and [7] VFUs and CFUs are used alternately, while our proposed architectures use VFUs and CFUs in parallel so that no idle units exist. The memory size is counted in multiples of messages to be stored. The required memory size to store the extrinsic, intrinsic and sum values for our proposed layered architecture is as small as in [6] and [7].

The next line in the table measures the utilization of the architecture. Heuristic mapping algorithms are applied in [5], [6] which achieve significantly lower values compared with our algorithm which is optimum on the proposed architectures. For any structured or unstructured code, we reach a utilization of 100% on the flooding architecture and a utilization of 100% on the layered architecture given the code is suitable for layered decoding.

For the heuristic mapping algorithms in [5], [6] the number of data updates per clock strongly depends on the code and in general is below the optimum p. Mapping in [7] is optimum and p messages are updated in each clock cycle. However, for our proposed designs the number of updated messages is twice or even five times as high as for the other implementations. In contrast to [7] where a static mapping function is applied on the extrinsic values, our proposed architecture applies a dynamic mapping function on the sum values. Thus we are able to process check and variable nodes in parallel and double the data throughput compared with [7].

Comparing our programmable architectures with the corresponding QC decoders with flooding and layered decoding [9], our approach needs the same amount of extrinsic, intrinsic and sum memory. Instead of each barrel shifter two permutation networks are used. Storage of a random parity-check matrix inherently requires more memory than storage of a QC parity-check matrix in the control memory. However, using the same parallelism, the developed scheme for collision-free memory access on the programmable architectures achieves the same data throughput as QC architectures.

6. CONCLUSION

In this article we presented a fully programmable layered decoder architecture together with an optimum mapping and scheduling algorithm. To the best of our knowledge, this is the first time a fully programmable layered decoder architecture is presented. The proposed design enables collision-free memory access for arbitrary LDPC codes. Any structured or unstructured LDPC code can be decoded with an architecture utilization of 100% on the identical hardware. Reconfiguration for a different code is achieved by changing the initialization of the control memory. In contrast to other programmable architectures where static mapping of messages to memory locations is used, our dynamic mapping increases the data throughput by a factor two to five.

REFERENCES

[1] D.J.C. MacKay and R.M. Neal, "Near Shannon limit performance of low-density parity-check codes," Electronics Letters, vol. 33 (6), pp. 457-458, 1997.

[2] IEEE 802.16e, "Air interface for fixed and mobile broadband wireless access systems," IEEE P802.16e/D12, 2005.

[3] IEEE 802.11n, "Wireless LAN medium access control and physical layer specifications: Enhancements for higher throughput," IEEE P802.16n, 2006.

[4] European Telecommunications Standards Institute (ETSI), "Digital Video Broadcasting (DVB) Second Generation ," EN 302 307 V1.1.1.

[5] G. Masera, F. Quaglio, and F. Vacca, "Implementation of a flexible LDPC decoder," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54 (6), pp. 542–546, 2007.

[6] C. Beuschel and H.-J. Pfleiderer, "FPGA implementation of a flexible decoder for long LDPC codes", *IEEE International Conference on Field Programmable Logic and Applications, FPL*, pp. 185–190, 2008.

[7] A. Tarable, S. Benedetto, and G. Montorsi, "Mapping interleaving laws to parallel turbo and LDPC decoder architectures," *IEEE Transactions on Information Theory*, vol. 50 (9), pp. 2002–2009, 2004.

[8] K. Gunnam, G. Choi, M. Yeary, and M. Atiquzzaman, "VLSI Architectures for Layered Decoding for Irregular LDPC Codes of WiMax," *IEEE International Conference on Communications, pp. 4542–47, 2007.*

[9] 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," *DATE*, pp. 331–336, 2007.