14th European Signal Processing Conference (EUSIPCO 2006), Florence, Italy, September 4-8, 2006, copyright by EURASIP

SERIAL LDPC DECODING ON A SIMD DSP USING HORIZONTAL SCHEDULING

Marco Gomes, Vitor Silva, Cláudio Neves and Ricardo Marques

Institute of Telecommunications - Department of Electrical and Computer Engineering
University of Coimbra, P-3030-290 Coimbra, Portugal
phone: + 351 239 79 62 36, fax: + 351 239 79 62 93
email: marco@co.it.pt, vitor@co.it.pt, claudio.neves@co.it.pt, ricky@co.it.pt

ABSTRACT

In this paper we propose an efficient vectorized low density parity check (LDPC) decoding scheme based on the min-sum algorithm and the horizontal scheduling method. Also, the well known forward-backward algorithm, used in the check-node messages update, is improved.

Results are presented for 32 and 16 bits logarithm likelihood ratio messages representation on a high performance and modern fixed point DSP. The single instruction multiple data (SIMD) feature was explored in the 16 bits case. Both regular and irregular codes are considered.

1. INTRODUCTION

Owing to their outstanding performance, low-density parity check codes (LDPC) are recognized to be one of the best classes of codes which approach the Shannon limit efficiently.They were first proposed by Robert Gallager [1], [2], in the 1960′s, based on sparse binary matrices and efficient message-passing decoding algorithms (belief propagation and bit-flipping).Their rediscover by MacKay and Neal [3] led to a growing interest on research and development of efficient LDPC coding solutions for digital communication and storage systems [4], [5].

A LDPC code is characterized by a parity check matrix, $\mathbf{H}$, with a low density of 1's. As any linear block code, a LDPC code can also be represented by a Tanner graph (TG) [6]. This bipartite graph is formed by two types of nodes. The check nodes (CN), one per each code constraint, and the bit nodes (BN), one per each bit of the codeword, with the connections between them given by $\mathbf{H}$. The $\mathbf{H}$ matrix of regular LDPC codes presents a uniform row and column weight structure, which can represent an advantage in the design of an iterative decoder. Yet, irregular LDPC codes are shown to have a better performance [7].

The sum product decoding algorithm (SPA) [1] applied to LDPC codes is known to have the best performance. The aim of SPA algorithm is to compute iteratively the *a posteriori* probability (APP) of each codeword bit to be 1 (or 0) considering that all parity check constraints are satisfied . The iterative procedure is based on an exchange of messages between the BN's and CN's of the Tanner graph, containing believes about the value of each codeword bit. These messages (probabilities) can be represented rigorously in their domain or, more compactly, using logarithm likelihood ratios. Due to the high computational complexity, the log-likelihood ratio form of SPA and its simplifications are preferred, namely, the *min-sum* algorithm [8]. We will follow this approach.

Usually, the message-passing is based on the so-called "flooding schedule" i.e., the messages sent by BN's are updated all together (in a serial or parallel manner) before CN's messages could be updated and, vice versa. Recently, and motivated by the question of whether it would be possible to improve the velocity of convergence and the performance of the iterative decoding algorithms, new message-passing schedules have been proposed [9], [10], [11]. One of the novel approaches is horizontal scheduling [10]. The main idea is to use the fresh calculated information as soon as it is available. So, in horizontal scheduling the information sent to a CN under process takes in account not only the information of the previous iteration, but also, all the information from the current iteration that has been updated by the previous processed CN's. Since each CN receives all up-to-date information, the new messages sent by that CN tend to be more reliable which accelerates the convergence of the algorithm. The parallel nature of the flooding schedule is not very important when we use a digital processor. Thus, a serial and faster message scheduling is preferable.

The simplicity of the operations involved in the *min-sum* algorithm (sum's and min's), is suitable for a fast fixed point digital signal processor (DSP), which constitutes, for medium length LDPC codes, a good and flexible decoding alternative to the expensive and specialized hardware decoding solutions. Furthermore, experimental results have shown that the *min-sum* loses no performance under message quantization with more than 4 bits [12].

Last generation of fixed point digital signal processors (DSP) contains a powerful set of single instruction multiple data (SIMD) operations which allow the processing of multiple data in a single instruction. In order to take full advantage of all the capacities of modern DPS's, efficient vectorized decoding approaches have to be developed. In this paper, we propose, 32 and 16 bits, vectorized solutions for LDPC decoding using the *min-sum* algorithm and horizontal scheduling.

In section 2, we present a brief review of the *min-sum* algorithm following the traditional flooding schedule approach. The advantages of the serial check node processing (horizontal scheduling) are discussed and the method is presented. Next, an improved forward-backward method is developed in section 3. In section 4 efficient and generic vectorized LDPC decoding solutions (16 and 32 bits) are developed for a modern fixed point SIMD DSP. Finally, in section 5, the experimental results are reported.

2. THE MIN-SUM ALGORITHM

The *min-sum* algorithm [8] is a simplified version of the SPA [1], [2] in the logarithmic domain, where multiplications are replaced by additions (less demanding). Consequently, the processing load in check nodes decreases significantly with just a small loss in performance.

Given a (N,K) LDPC code, we assume BPSK modulation which maps a codeword, $\mathbf{c} = (c_1, c_2, \cdots, c_N)$, onto the sequence, $\mathbf{x} = (x_1, x_2, \cdots, x_N)$, according to $x_i = (-1)^{c_i}$. Then, the modulated vector $\mathbf{x}$ is transmitted through an additive white Gaussian noise (AWGN) channel. The received sequence is $\mathbf{y} = (y_1, y_2, \cdots, y_N)$, with $y_i = x_i + n_i$, where n_i is a random gaussian variable with zero mean and variance, $N_0/2$. We denote the set of bits that participate in check m by $N(m)$ and, similarly, we define the set of checks in which bit n participates as $M(n)$. We also denote $N(m)\backslash n$ as the set $N(m)$ with bit n excluded and, $M(n)\backslash m$ as the set $M(n)$ with check m excluded.

Denoting the log-likelihood ratio (LLR) of a random variable as, $L(x) = \ln(p(x=0)/p(x=1))$, we designate:

LP_n - The *a priori* LLR of BN$_n$, derived from the received value y_n;

Lr_{mn} - The message that is sent from CN$_m$ to BN$_n$, computed based on all received messages from BN's $N(m)\backslash n$. It is the LLR of BN$_n$, assuming that the CN$_m$ restriction is satisfied;

Lq_{mn} - The LLR of BN$_n$, which is sent to CN$_m$, and is calculated, based on all received messages from CN's $M(n)\backslash m$ and the channel information, LP_n ;

LQ_n - The *a posteriori* LLR of BN$_n$.

2.1 Traditional flooding-schedule

The *min-sum* algorithm following the so-called "flooding schedule" is carried out as follows:

For each node pair (BN$_n$, CN$_m$), corresponding to $h_{mn} = 1$ in the parity check matrix $\mathbf{H}$ of the code do:

<u>*Initialization:*</u>

$$Lq_{nm} = LP_n = \frac{2y_n}{\sigma^2}. \qquad (1)$$

The factor $2/\sigma^2$ can be omitted without any performance loss [8], thus no *a priori* information about the AWGN channel is required, i.e., $Lq_{nm} = LP_n = y_n$.

<u>*Iterative body:*</u>

I. Calculate the log-likelihood ratio of message sent from CN$_m$ to BN$_n$:

$$Lr_{mn} = \prod_{n' \in N(m)\backslash n} \alpha_{n'm} \min_{n' \in N(m)\backslash n} \beta_{n'm}, \qquad (2)$$

where

$$\alpha_{nm} \triangleq sign(Lq_{nm}), \qquad (3)$$

and

$$\beta_{nm} \triangleq |Lq_{nm}|. \qquad (4)$$

II. Calculate the log-likelihood ratio of message sent from BN$_n$ to CN$_m$:

$$Lq_{nm} = LP_n + \sum_{m' \in M(n)\backslash m} Lr_{m'n}. \qquad (5)$$

III. Compute the a posteriori pseudo-probabilities and perform hard decoding:

$$LQ_n = LP_n + \sum_{m' \in M(n)} Lr_{m'n}, \qquad (6)$$

$$\forall n,\ \hat{c}_n = \begin{cases} 1 & \Leftarrow \quad LQ_n < 0 \\ 0 & \Leftarrow \quad LQ_n > 0 \end{cases}. \qquad (7)$$

The iterative procedure is stopped if the decoded word verifies all parity check equations of the code ($\hat{\mathbf{c}}\mathbf{H}^T = \mathbf{0}$) or the maximum number of iterations is reached.

2.2 Horizontal scheduling

It is well known that SPA and *min-sum*, following the traditional flooding-schedule message updating rule, are optimum APP decoding methods when applied to codes described by TG's without cycles [14]. However, good codes always have cycles and the short ones tend to degrade the performance of the iterative message passing algorithms (results far from optimal). Motivated by the referred problem and the speed up convergence goal, new message-passing schedules have been proposed [9], [10], [11].

Considering flooding-schedule, the messages sent by BN's are updated all together (in a serial or parallel manner) before CN's messages could be updated and, vice versa. At each step, the messages used in the computation of a new message, are all from the previous iteration. A different approach is to use new information as soon as it is available, so that, the next node to be updated could use more up-to-date (fresh) information. For example, this can be done following two different strategies know by horizontal and vertical scheduling with a considerable processing gain in the number of iterations to reach a valid codeword [10].

Vertical-schedule operates along the BN's while horizontal schedule operates along CN's. Both schedules are serial, i.e., the nodes are processed sequentially, one after the other. The computational complexity of the flooding and serial schedules is similar. However, from a DSP point of view, horizontal-schedule is preferable to vertical-schedule, as it will be described next.

Observing equations (5) and (6) we note that the message sent from BN$_n$ to CN$_m$, can easily be obtained by

$$Lq_{nm} = LQ_n - Lr_{mn}. \qquad (8)$$

Consequently, we only have to keep in memory LQ_n and Lr_{mn} messages, because according with (2), the values of $Lq_{nm}^{(i-1)}$, with $n \in N(m)$, needed to compute the new $Lr_{mn}^{(i)}$ messages, sent by CN$_m$ at iteration i, can be obtained from values, $LQ_n^{(i-1)}$ and $Lr_{mn}^{(i-1)}$, of the previous iteration, as shown in (8). The old $Lr_{mn}^{(i-1)}$ values are replaced by the new ones, $Lr_{mn}^{(i)}$, at the end of CN$_m$ updating process.

Processing BN's in a serial manner, following the so called vertical-schedule, means that after processing each BN, says BN$_n$, all CN's connected to it, i.e. CN's $m \in M(n)$, must reflect the current BN$_n$ actualization. Thus, for each CN $m \in M(n)$, all messages $Lr_{mn'}$, with $n' \in N(m)\backslash n$, must be updated according with the new received message Lq_{nm} from BN$_n$. This represents a higher processing burden in comparison to the horizontal-schedule approach, where the updating problem is solved by just actualizing the *a posteriori* LLR's, LQ_n, according to (8).

Memory addressing problems are also simplified when using horizontal-schedule since after processing a CN, says CN$_m$, only one value LQ_n per BN $n \in N(m)$, must be read (old LQ_n) from memory and written (new updated LQ_n)

back to the same place. Considering vertical-schedule, after processing a BN, says BN_n, a set of values $Lr_{mn'}$, with $n' \in N(m)\backslash n$, for each CN $m \in M(n)$, must be read, recalculated and written back to the same memory positions.

Finally, the *min-sum* algorithm with horizontal-schedule is carried out as follows:

For each node pair (BN_n, CN_m), corresponding to $h_{mn} = 1$ in the parity check matrix $\mathbf{H}$ of the code do:

Initialization:

$$LQ_n = LP_n = \frac{2y_n}{\sigma^2}, \tag{9}$$

$$Lr_{mn} = 0. \tag{10}$$

Iterative body:

I. Process each CN one after the other:

Step 1: For the CN_m, calculate the log-likelihood ratio of messages sent to BN $n \in N(m)$:

$$Lr'_{mn} = \prod_{n' \in N(m)\backslash n} \alpha_{n'm} \min_{n' \in N(m)\backslash n} \beta_{n'm}, \tag{11}$$

where

$$\alpha_{nm} \triangleq sign\left(Lq_{nm}\right) = sign\left(LQ_n - Lr_{mn}\right), \tag{12}$$

and

$$\beta_{nm} \triangleq \left|Lq_{nm}\right| = \left|LQ_n - Lr_{mn}\right|, \tag{13}$$

with Lr'_{mn} being the new calculated message to be sent from CN_m to BN_n, and Lr_{mn} referring to the old message that was sent.

Step 2: Update the *a posteriori* LLR of BN_n.

$$LQ'_n = LQ_n - Lr_{mn} + Lr'_{mn}. \tag{14}$$

II. Perform hard decoding as in (7) and stopping if the decoded word $\hat{\mathbf{c}}$ verifies all parity check equations of the code ($\hat{\mathbf{c}}\,\mathbf{H}^T = \mathbf{0}$) or the maximum number of iterations is reached.

3. IMPROVED FORWARD BACKWARD METHOD

The serial CN updating process (for the various message-passing LDPC decoding algorithms) is normally carried out using an efficient forward backward scheme [9], [15]. The procedure avoids the re-computation of common values to the various messages sent by one CN to all the BN's connected to it.

Consider a CN_m of weight k, and denote by n_i, the i-th BN connected to it. The CN updating rule can be written as:

$$Lr_{mn} = \underset{n' \in N(m)\backslash n}{\boxplus} Lq_{n'm} \tag{15}$$

where the operation $\boxplus$ is algorithm dependent (for *min-sum*, $a \boxplus b \triangleq sign(a)\, sign(b) \min(|a|, |b|)$).

Traditionally, the forward backward method is seen as a three step iterative procedure:

Step 1 (*move forward*):

$$\begin{aligned} A_1 &= Lq_{n_1 m} \\ A_i &= A_{i-1} \boxplus Lq_{n_i} m, \quad i = 2, ..., k-1 \end{aligned} \tag{16}$$

Step 2 (*move backward*):

$$\begin{aligned} B_k &= Lq_{n_k m} \\ B_i &= B_{i+1} \boxplus Lq_{n_i} m, \quad i = k-1, ..., 2 \end{aligned} \tag{17}$$

Step 3 (*updating*):

$$\begin{aligned} Lr_{mn_1} &= B_2 \\ Lr_{mn_i} &= A_{i-1} \boxplus B_{i+1}, \quad i = 2, ..., k-1 \\ Lr_{mn_k} &= A_{k-1} \end{aligned} \tag{18}$$

Now, we will show that is possible to merge step 2 and 3 in just one iterative cycle, decreasing for a half their processing time.

It is important to note that the Lq_{nm} values are stored in memory (unless the CN's weights are very small and it is possible to keep them in CPU registers). Thus, they must be read in order to calculate A_i according to (16). After that, step 2 and 3 can be done jointly by simply reversing the Lr_{mn_i} updating order. Consequently, the new algorithm is defined as follows (with $i = k, ..., 2$):

- ***Iteration*** k**:** At the begin of the backward pass it is already known A_{k-1} and it was received the last BN message, $Lq_{n_k m}$, and so, calculate:

$$\begin{aligned} Lr_{mn_k} &= A_{k-1} \\ B_k &= Lq_{n_k m} \end{aligned} \tag{19}$$

- ***Iteration*** $k-1$**:** Since B_k is already known from the previous iteration, calculate:

$$\begin{aligned} Lr_{mn_{k-1}} &= A_{k-2} \boxplus B_k \\ B_{k-1} &= B_k \boxplus Lq_{n_{k-1} m} \end{aligned} \tag{20}$$

$$\vdots$$

- ***Iteration*** 2**:** Since B_3 is already known from the previous iteration, calculate:

$$\begin{aligned} Lr_{mn_2} &= A_1 \boxplus B_3 \\ Lr_{mn_1} &= B_2 = B_3 \boxplus Lq_{n_2 m} \end{aligned} \tag{21}$$

4. VECTORIZED LDPC DECODING

We developed useful 32 and 16 bits vectorized data structures for effective LDPC decoding on a DSP using the *min-sum* algorithm and horizontal scheduling. We choose the fixed point TMS320C6416 [13], from Texas Instruments as the target device. However, the structure design is almost generic which means minor changes when implemented in a different processor. In order to improve significantly the decoding speed without any performance loss, the 16 bit version is carefully developed to use SIMD instructions.

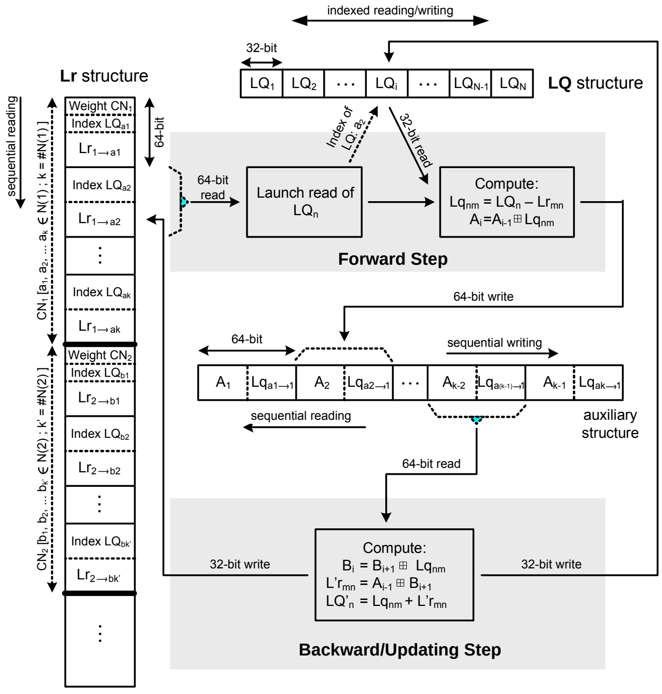

Figure 1: Vectorized LDPC decoding using horizontal scheduling and the forward backward method. Data structures layout and processing flow for the 32 bit version.

4.1 32-bit version

Based on the Tanner graph of the LDPC code, and the processing flow of the *min-sum* algorithm with horizontal scheduling, we created dynamic and vectorized data structures to represent all exchanged messages between connected nodes (BN_n, CN_m) which allow us to reduce the required memory space and processing time, improving the computational efficiency of the implemented decoding algorithms.

Therefore, we created two data vector structures named by **LQ** and **Lr** (see fig. 1), respectively, to store all LQ_n values, and all Lr_{mn} messages sent from each CN_m to its connected BN's. In order to compute the Lq_{nm} messages according with (8) and, simultaneously, to reduce the number of memory reads by sequential data accessing (avoiding this way addressing overheads), we pack consecutively in **Lr** memory pairs of values Lr_{mn} and the corresponding LQ_n indexes (pointers to **LQ** memory).

In fig. 1 is also possible to observe the mechanism of the improved two step forward and backward/updating algorithm. Both steps are iterative procedures. By using the CPU parallelism and proper programming techniques (namely, software pipelining), we reduced both iterative blocks to minimum admissible length, i.e., 6 CPU cycles.

An auxiliary structure is used to keep the temporary A_i and Lq_{nm} values, computed during the forward step. Although in the fig. 1, we may think, that all computed values must be stored before backward/updating step could take place, that isn't true, because it is possible to merge the *epilog* of the forward step, with the *prolog* of backward/updating step, by just keeping the last computed A_i and Lq_{nm} values in CPU registers.

Hard decoding is performed by simply extracting the sign of each value of the **LQ** structure. Sequential data access is taken using 64-bits reads.

The process of verifying if all parity checks equations are satisfied is also optimized, by just stopping the testing procedure when a code restriction fail.

4.2 16-bit version

The TMS320C6416 is able, for example, to sum two pairs of 16 bits data in a single instruction (the same is true for the min operation). In order to take full advantage of SIMD instruction set, a different packing order was adopted in the definition of both **Lr** and the auxiliary structure. As can be observed in fig. 2, at each iterative cycle of both forward and backward/updating steps, two pairs of (Lr_{mn}, LQ_n) values are processed (against one pair in 32-bit version).

The computational complexity of the forward step remained unchanged, meaning that we were able to process the same number of data, twice as fast. In the backward/updating case, there was an increase to 9 CPU cycles, meaning that we were able to perform 1.3 times faster. Hard decoding is also accomplished two times faster. Obviously, this approach could be easily generalized to an 8 bit version.

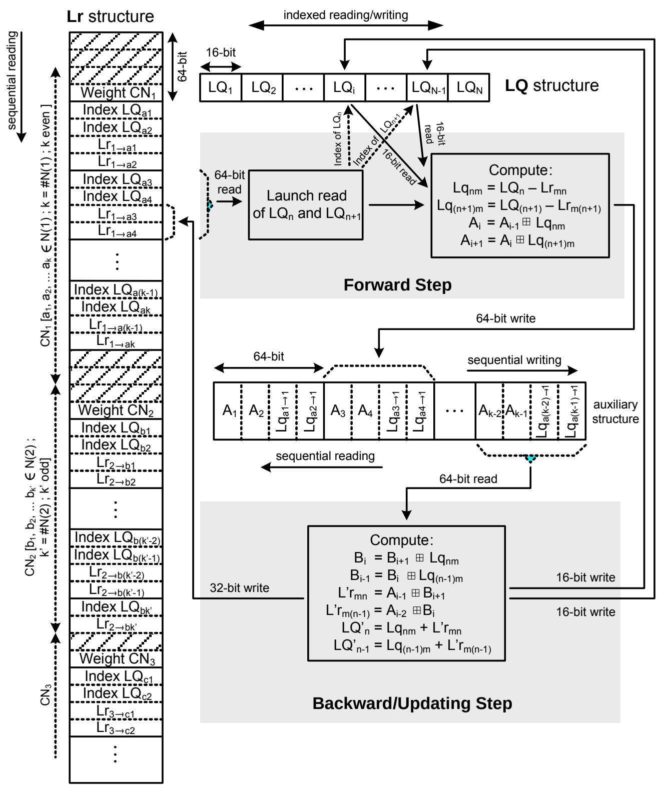

Figure 2: Vectorized LDPC decoding using horizontal scheduling and the forward backward method. Data structures layout and processing flow for the 16 bit version.

5. EXPERIMENTAL RESULTS

Decoding results were obtained using medium length LDPC codes, namely, two regular $(504, 252)$ and $(1908, 1696)$ codes, with CN's of weight, respectively, 6 and 36, and one irregular $(1057, 813)$ with CN's of weight 17 and 18 [16].

The processing time required to perform the forward, backward/updating and hard decoding steps is independent of the channel SNR. The measured time to carry out the mentioned steps using hand optimized machine code is shown in table 1 in terms of the number of CPU cycles per iteration. In order to perform a fair comparison between the different codes, it is also presented the number of CPU cycles per iteration and per edge (a Tanner graph connection between a CN and a BN) which represents the computation burden of each (Lr_{mn}, LQ_n) pair. By comparison of these results, we observe that independently of the code characteristics (length, rate, regularity and CN's weights), the obtained values are almost constants which shows the effectiveness of the proposed approaches.

For the $(504, 252)$ code the 16-bit version improvement is almost negligible due to the short weight of the CN's, which means a low pipelining programming gain. Additionally, we can conclude by comparison with the 32-bit register version (auxiliary data structure stored in the CPU registers) that the proposed generic decoding structure performs almost as fast as the dedicated solution.

For the $(1057, 813)$ and $(1908, 1696)$ codes, we can recognize a significantly decrease in the number of CPU cycles per iteration in the 16-bit (SIMD) version case, i.e., a gain of approximately, 1.3 and 1.4, respectively. For heavy CN codes the gain is almost constant. However, for odd weight CN's codes, it may occur a little gain loss, since the execution of the *prolog* and *epilog* of both forward and backward/updating steps is more complex (see in fig. 2 the odd weight CN case where the last pair (Lr_{mn}, LQ_n) must be processed alone).

		(504,252)	(1057,813)	(1908,1696)
Number of code edges		1 512	4 228	7 632
CPU cycles / Iteration	32-bit	22 698	68 607	125 426
	16-bit	21 147	53 041	91 988
	Reg.	18 009		
CPU cycles /Iteration/Edge	32-bit	15.01	16.23	16.43
	16-bit	13.99	12.55	12.05

Table 1: Full decoding (except syndrome computation) processing time in CPU cycles per iteration for 32-bit, 16-bit and register versions.

Considering a maximum number of 10 iterations, it is shown in Table 2 the average number of CPU cycles per iteration to perform the whole proposed algorithm (vectorized *min-sum* with horizontal scheduling) for the 32-bit version.

	(504,252)	(1057,813)	(1908,1696)
2dB	25786	69314	126824
3dB	28664	74553	126906
4dB	30771	88192	139182

Table 2: The average processing time in CPU cycles per iteration for the 32-bit version.

Definitely, there is a significantly increase in the number of CPU cycles per iteration for high SNR's. It means that the developed process to verify the syndrome vector works well. For small SNR values, there is a high probability to find an early failed syndrome equation, which means that $(\hat{\mathbf{c}}\mathbf{H}^T = \mathbf{0})$ verification will be stopped almost at the beginning. In opposition, for high SNR values, $(\hat{\mathbf{c}}\mathbf{H}^T = \mathbf{0})$ verification takes an important portion of the overall decoding time.

6. CONCLUSION

In this work, we have proposed efficient vectorized LDPC decoding solutions (16 and 32 bits) based on the serial check node processing (horizontal scheduling) and the *min-sum* algorithm. The target device was a modern fixed point SIMD DSP. Also, an improved forward-backward method was developed. The obtained experimental results have shown the applicability of the proposed approaches in real time decoding of medium length LDPC codes included in high bit rate transmission/storage data systems.

REFERENCES

[1] R. G. Gallager, "Low-Density Parity-Check Codes", IRE Transactions on Information Theory, vol. IT-8, pp.21-28, Jan. 1962.

[2] R. G. Gallager, Low-Density Parity-Check Codes, Cambridge, MIT Press, 1963.

[3] D. J. C. MacKay and R. M. Neal, "Near Shannon Limit Performance of Low Density Parity Check Codes", IEEE Electronics Letters, vol. 32, no. 18, pp. 1645-1646, Aug. 1996.

[4] ETSI EN 302 307 V1.1.1, Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications, March 2005.

[5] Tom Richardson, "The Renaissance of Gallager's Low Density Parity Check Codes", IEEE Commun. Magazine, vol. 41, no. 8, pp. 126-131, Aug. 2003.

[6] R. Tanner, "A Recursive Approach to Low Complexity Codes", IEEE Trans. Inform. Theory, vol. 27, no.5, pp. 533-547, Sept. 1981.

[7] T. Richardson and R. Urbanke, "The Capacity of Low-Density Parity-Check Codes Under Message-Passing Decoding", IEEE Transactions on Information Theory, vol. 47, no. 2, pp. 599-618, Feb. 2001.

[8] J. Chen and M. Fossorier, "Near Optimum Universal Belief Propagation Based Decoding of Low-Density Parity Check Codes", IEEE Transactions on Communications, vol. 50, no. 3, pp. 406-414, March 2002.

[9] J. Zhang and M. Fossorier, "Shuffled Belief Propagation Decoding", Thirty-Sixth Asilomar Conference on Signals, Systems and Computers 2002, vol. 1, pp. 8-15, Nov. 2002.

[10] E. Sharon, S. Litsyn and J. Goldberger, "An efficient message-passing schedule for LDPC decoding", 23rd IEEE Convention of Electrical and Electronics Engineers in Israel, pp. 223-226, Sep. 2004.

[11] H. Xiao and A. M. Banihashemi, "Graph-Based Message Passing Schedules for Decoding LDPC Codes", IEEE Transactions on Communications, vol. 52, no. 12, pp. 2098-2105, Dec. 2004.

[12] F. Zarkeshvari and A. H. Banihashemi, "On implementation of Min-Sum Algorithm for Decoding Low Density Parity Check (LDPC) Codes", IEEE Globecom 2002, vol. 2, pp.1349-1353, Nov. 2002.

[13] TMS320C6000 CPU and Instr. Set Ref. Guide, Texas Inst., Oct. 2000.

[14] F. R. Kschischang, B. J. Frey and H. Loeliger, "Factor Graphs and the Sum-Product Algorithm", IEEE Transactions on Information Theory, vol. 47, no. 2, pp. 498-519, Feb. 2001.

[15] X.-Y. Hu, E. Eleftheriou, D.-M. Arnold and A. Dholakia, "Efficient Implementations of the Sum-Product Algorithm for Decoding LDPC Codes", IEEE GLOBECOM '01, vol. 2, pp. 1036-1036E, Nov. 2001.

[16] http://www.inference.phy.cam.ac.uk/mackay/codes