Low cost LDPC decoder for DVB-S2

John Dielissen*, Andries Hekstra*, Vincent Berg+

* Philips Research, High Tech Campus 5, 5656 AE Eindhoven, The Netherlands
+ Philips Semiconductors, 2, rue de la Girafe, BP. 5120, 14079 Caen, France
E-mail: john.dielissen@philips.com

Abstract

Because of its excellent bit-error-rate performance, the Low-Density Parity-Check (LDPC) algorithm is gaining increased attention in communication standards and literature. The new Digital Video Broadcast via Satellite standard (DVB-S2) is the first broadcast standard to include a LDPC-code, and the first implementations are available. In our investigation of generic LDPC-implementations we found that scalable sub-block parallelism enables efficient implementations for a wide range of applications. For the DVB-S2 case, using sub-block parallelism we obtain half the chip-size of known solutions. For the required performance in the normative configurations for the broadcast service (90 Mbps), the area is even $\frac{1}{3}$ compared to the smallest published decoder.

1. Introduction

Since the rediscovery of Low-Density Parity-Check (LDPC) codes [7] in 1996, many publications of their implementations did appear. LDPC codes are included in the DVB-S2[1] standard, which is a digital satellite video broadcast standard. For this standard, implementations have been suggested in [5] and [8]. Because of the excellent bit-error-rate (BER) performance of the algorithm, LDPC-codes are expected to be part of many future standards as well.

In the DVB-S2 standard, the interconnection between computation kernels is grouped in blocks of 360. When fully exploiting this block-level parallelism, it results in an implementation with 360 computation kernels. Such an implementation achieves a throughput rate of more than 800 Mbps. It is obvious that such an architecture is too expensive when only 90 Mbps is required. However scaling down the architecture to achieve lower performance is non-trivial. First of all, the algorithm should be re-structured such that downscaling is possible, and second, during the mapping, freedom inside iterations should be recognized and exploited.

In this paper we first discuss the state-of-the-art work. In Section 3, we briefly explain the LDPC algorithm. We explain the selected mixture of known technologies which are used in the new LDPC-decoder. Although the technology explained in this paper can be applied to a wider range of applications, the results and the benchmarks are presented primarily for the DVB-S2 standard, which is explained in Section 4. In this section, we also discuss the set of LDPC-codes to which this solution applies. In Section 5 we present the scalable sub-block architecture, and in Section 6, the efficiency of the architecture is explained. We end this paper with conclusions and recommendations for future work.

2. State-of-the-art

One of the most common measures to compare LDPC decoders, is the level of parallelism. This ranges from fully parallel [2], via block-level parallel[5, 8], grouped sequentially [4], to fully sequentially. In a fully parallel implementation, all symbol and check-node calculations (see Section 3), are directly realised in hardware. All units are interconnected via many wires, leading to congestion in the layout. A fully parallel implementation of the DVB-S2 decoder is impractical. For DVB-S2, even at 1 MHz, this solution yields a throughput of 1Gbps, calculating 10^{12} operations/second. In a block-level parallel implementation, the code needs the structure that the interconnect between the symbol- and check-node calculations is grouped. Both known implementations ([5] and [8]) of an LDPC-DVB-S2 decoder are examples of this block-level parallelism. To our knowledge, no implementations of sub-block-level parallelism have been published yet, which is the category in which this decoder falls. In a grouped sequential solution [4], the al-

3-9810801-0-6/DATE06 © 2006 EDAA

gorithm loops sequentially through all symbol-nodes, executing the connected check-node calculations simultaneously. This solution requires as many memory accesses as there are connected check-nodes. Since in DVB-S2 the number of connected check-nodes varies from 1 to 13, this is not an efficient solution. Moreover, the solution would lead to a throughput of 10 Mbps (@300 MHz), e.g. too low for DVB-S2. Fully serial solutions, traversing symbol-nodes and check-nodes consecutively, would result in an even lower throughput, hence this solution is also out of scope for an efficient implementation for DVB-S2.

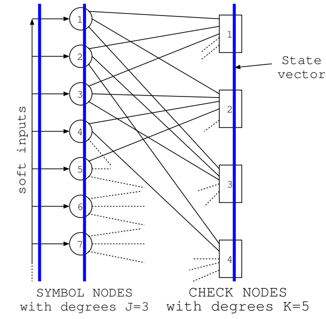

Figure 1. Tanner graph of LDPC code

3. LDPC decoding algorithm

In this paper we use the *uniformly-most-powerful belief propagation* UMP-BP version, of the LDPC-algorithm, which is described in [3, 6]. This algorithm achieves a performance level very close to, or sometimes even out performing that of BP decoding, while offering significant hardware advantages. Let N be the codeword length and M be the number of parity check equations. The parity check matrix H consists of M rows and N columns with elements "0" or "1". The rows in the matrix are the parity check equations, and the set of elements which have a '1' in a row are the arguments of the equation. For a parity check equation with index m, $0 \leq m < M$, define the set $N(m)$ of codeword symbol positions that it checks,

$$N(m) = \{n | n = 0, 1, \ldots, N-1; H_{mn} \neq 0\}.$$

The number of elements in $N(m)$ is refered to as K_m. Similarly, for a codeword symbol position n, $0 \leq n < N$, define the set $M(n)$ of indices of parity check equations that check the symbol position n,

$$M(n) = \{m | m = 0, 1, \ldots, M-1; H_{mn} \neq 0\}.$$

The number of elements in $M(n)$ is refered to as J_n. The parity check matrix can be associated with a bipartite graph (V, E) called the Tanner graph, shown in Figure 1. The set of vertices (V) is the union of the set of N symbol-nodes and the set of M parity check-nodes. The set of edges (E) consisting of all edges (m, n) for which $H_{mn} = 1$. Classical iterations of the LDPC algorithm consist of information send from symbol-nodes (N) via the edges (E) to the check-nodes (M), and back.

For a given iteration of the UMP-BP algorithm, we define the following variables:

- L_n - The x bit, signed input message into symbol-node n.

$$L_n = \frac{2y_n}{\sigma^2} \tag{1}$$

  y_n being the received BPSK symbol value, and σ^2 being the noise variance.

- λ_{nm}^i - The message sent from symbol-node n to check-node m in the ith iteration.

$$\lambda_{nm}^i = L_n + \sum_{m' \in M(n) \backslash m} \Lambda_{m'n}^{i-1} \tag{2}$$

- Λ_{mn}^i - The message sent from to check-node m to symbol-node n in the ith iteration.

$$\begin{aligned} \Lambda_{mn}^0 &= 0, \\ \Lambda_{mn}^i &= \underset{n' \in N(m) \backslash n}{XOR} \{sign(\lambda_{n'm}^i)\} \\ & \quad * \underset{n' \in N(m) \backslash n}{MIN} \{|\lambda_{n'm}^i|\} \end{aligned} \tag{3}$$

  XOR is defined as sign equivalent of the boolean xor function, e.i. $XOR(-, -) = +$

- λ_n - The decoder output messages. Unlike the λ_{nm}'s, the decoder output message λ_n uses all available information available in a symbol-node n, and is only necessary in the last iteration I.

$$\lambda_n = L_n + \sum_{m \in M(n)} \Lambda_{mn}^I \tag{4}$$

A straightforward implementation of the UMP-BP algorithm stores the received input message (all L_n's) in the decoder input memory, shown as the left state vector in Figure 1. The set of input symbols Λ_{mn} to the symbol-nodes constitute the middle state vector (through symbol-nodes),

The set of input symbols λ_{nm} to the check-nodes constitutes the right state vector (through check-nodes). During the first half-iteration, all messages λ_{nm} are sent from all symbol-nodes to the check-nodes. During the second half-iteration, all messages Λ_{mn} are sent from all check-nodes to the symbol-nodes.

One of the key elements in LDPC decoding is the iterative nature. In practical decoders, the worst case number of iterations ranges from 30 to 50, depending on the required performance. In 2001, Yeo at.al. [9], started a technology called "staggered decoding": intermediate results are used within an iteration, and in fact it is similar to the step from Jacobi iteration to Gauss-Seidel iteration in numerical mathematics. In other publications the technique is also known under "turbo decoding LDPC", "shuffled decoding", and "layered decoding". In the remainder of this article we refer to this technology as Gauss-Seidel iterations. In the traditional implementations, the messages calculated during the previous iteration must be stored until its last access, and concequently new messages must be stored in other locations. However in Gauss-Seidel iterations the messages can be overwritten immediately, and this technology therefore leads to half the message memory requirements. The second, and most important benefit of Gauss-Seidel iteration is the factor two reduction in the number of iterations required to meet a certain BER performance. Also in the implementations published in [5], and [8], this technique is used.

The complexity of calculating check-nodes (3) is reduced when its arguments arrive consecutively [6]. For the first part of (3) the property that the XOR of "all but one", is equal to the XOR of "all and the one" can be used. This results in storing individual signs, and administrating a running XOR-value. For the second part of (3), it can be observed that the magnitudes leaving a check-node have only two values: either the overall minimum or the overall one-but minimum in the case the corresponding input delivered the minimum. Magnitude calculations thus result in tracking a running minimum, a running one-but-minimum, and the index of the incoming edge providing the minimum. Instead of storing the K_m outgoing check-node messages, this set of data can be stored as one compressed vector[4] which entails the magnitude of the minimum, the one-but-minimum, the index of the minimum, and K_m signs. For 5-bit magnitudes this relates to $\frac{2*5+log_2(K_m)+K_m}{K_m}$ bits, against $(1+5)$ bits per message otherwise. For DVB-S2, the lowest K_m equals 4, resulting in only marginal reduction. However, since the number of messages is larger for some of the higher rates in the code, a message memory compression of approximately a factor 2 is achieved.

In [9], Yeo et.al. suggests to store the sum of all, e.g. λ_n, and to use the next calculations:

$$\lambda_{nm}^{i} = \lambda_{n}^{i-1} - \Lambda_{mn}^{i-1} \tag{5}$$

This type of calculations is very beneficial compared to equation (2), when $M(n)$ consists of many elements, as is the case in DVB-S2. Note that in "staggered decoding" [9] ommites the second term in equation (5), which is a step that we do not do because of the BER performance degradation. The novel calculation form used in our solution is to store λ_n in the symbol memory, and to store all Λ_{mn} for one m in a compressed way (noted as Λ_m) in the memory. The calculations are organized check-node centric: the algorithm loops through all check-nodes, and for each check-node, the K_m corresponding λ_n's are retrieved in consecutive cycles. By on-the-fly decompression of Λ_m, the $K_m\lambda_{nm}$'s are reconstructed. After K_m cycles, the new Λ_m is known, and the λ_n's in the memory can be updated. Note that by these intermediate updates of λ_n, the next check-node uses a newer version of the symbol-node within an iteration, and Gauss-Seidel iteration are conducted. With these Gauss-Seidel iterations both the memory reduction and the halving of the number of iterations, as shown in [9], are established.

Where in the check-node equation (3), only the minima of all except its corresponding input, [10] has shown several different correction techniques which can be applied to increase the BER performance, even beyond the ideal BP algorithm. Also the application of a fudge-factor, e.g. a constant, with which the output of the minimum function is multiplied, improves the performance. In fact, with normalisation the performance of the UMP-BP algorithm becomes similar to the (more difficult to calculate) BP algorithm [3].

4. LDPC code for DVB-S2

The specification of a code can in general be done by presenting its H-matrix. The rows in the matrix are the parity check equations, and the set of elements which have a '1' in a row are the arguments in that equation. If e.g. there is a '1' in the second column of a row, and a '1' in the column 365, the symbols '2', and '365' participate in one equation. The structure within the DVB-S2 code, revealed in the H-matrix, can be observed in Figure 2. Note that the code consists of interconnect blocks of $360 * 360$. There are $360 * Q$ parity symbols, and also $360 * Q$ parity equations. The block length for all specified rates is 64800 symbols. The diagonals should be seen as the line of '1's in the matrix. These lines have a clear div-mod structure.

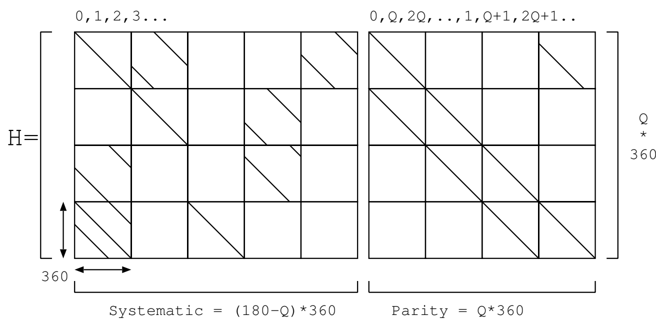

Figure 2. H-matrix of DVB-S2 LDPC code

The code is specified for rates $\frac{1}{4}$ to $\frac{9}{10}$, (e.g. Q=135..18). K_m, being the number of symbol-nodes to which a check-node is connected, differs for each rate, but is constant for all equations within a rate [1]. J_n, being the number of check-nodes to which a symbol-node is connected differs for all rates, and differs for symbol-nodes within a rate. J_n can vary between 13 and 1.

The architecture presented in this paper can handle all codes where the parity matrix has the (scrambled) quasi cyclic structure shown in Figure 2.

5. Proposed LDPC decoding architecture

The center of the architecture is formed by the data path, shown in Figure 3. As explained in Section 3, the $K_m\lambda_{nm}$'s are formed by subtracting the decompressed Λ_{mn} from the sequentially arriving λ_n's. These λ_{nm}'s are stored, and are simultaneously used to calculate the new vector (running minimum, one-but minimum, index, and xor). During the next K_m cycles the mentioned operations are repeated for the next parity check equation, while for the current parity check equation, the Λ_{mn} are calculated by decompressing the new vector. These Λ_{mn} are added to the λ_{nm} from the storage, which results in the new λ_n. The data path thus produces the $K_m\lambda_n$'s of one parity equation, while receiving the $K_m\lambda_n$'s for the next equation. To avoid conflicts, the order of the parity equations must be statically scheduled such that two consecutive equations have no symbol-nodes in common. When achieving this, a code with a constant K_m can conduct one data path calculation per clock cycle. In other cases, no-operations need to be inserted.

Although the "running vector calculator" suggest that only the minimum, one-but minimum, index and signs are

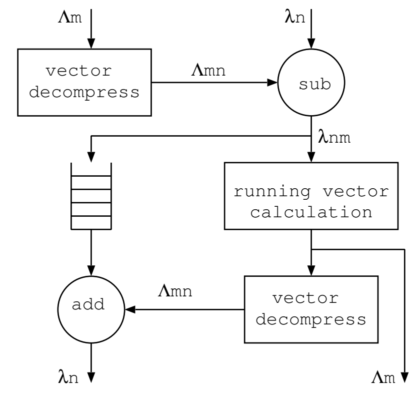

Figure 3. data path of architecture

calculated, more complex calculations, as suggested in Section 3 can be incorporated without much effort and at low cost. The proposed data-path can handle one connection between a check-node and a symbol-node per clock cycle. For code-rate $\frac{3}{5}$, there is an averate of 4.4 connections per symbol-node, and for 30 Gauss-Seidel iterations, running at 300 MHz, one data path, can handle a throughput of 2.27 Mbps. This implies that for the targeted 90 Mbps, $D = 40$ data paths are required. To allow some input/output overhead, $D = 45$ data paths are required. The scalability of our solution is in the choice of D: Depending on the clock frequency, number of iteration, and throughput requirements, a D can be chosen such that the area is as small as possible. Note that D must be a divider of block-level parallelism inside the code. The top-level architecture, which includes these D data paths is shown in Figure 4.

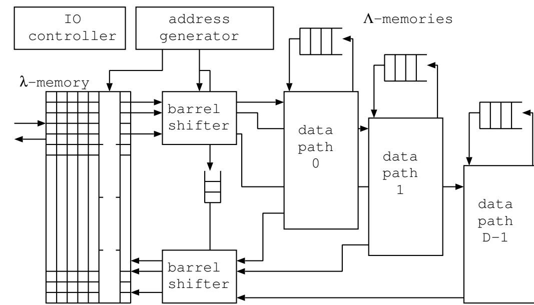

Figure 4. top-level architecture

The architecture shows that D words λ_n are packed into one word in the λ-memory. These λ_n's are rotated over a certain angle by the barrel shifter [5], and passed on to the D data paths. After K_m clock cycles the

1 there is one equation which is connected to one symbol-node less

λ_n are returned, and are rotated back over the same angle. In this way the λ_n always have the same position in the λ-memory. Note that this allows the decoder to stop anywhere in an iteration, and produce the output. There is always a valid λ_n in a predetermined place in the memory.

Although the architecture suggests D different FIFOs for the Λ-memories, in practice it consists of a few single port memories in which the vectors are read, and written efficiently. The λ-memory is D*8 bits wide dual-port memory of $180 * \frac{360}{D}$ words.

The main difference between our architecture, and the architectures proposed by [5, 8], is that here only $D = 45$ data paths are necessary, whereas [5, 8] need 360. Furthermore, our shuffling network only rotates over D elements, while the state-of-the art solution rotates over 360. Applying our data path 360 times would lead to a throughput of more than 800 Mbps. The downscaling proposed in [5, 8], consist of separate symbol-node and check-node calculations (factor 2), and multiple clock cycles per calculation. This however does not lead to a smaller barrel shifter, or to significant smaller data path logic. The solution chosen in our approach is to use high speed, high throughput data paths, and to instantiate less of these data paths. This approach leads to a linear scaling of data path area with throughput, and a quadratic reduction in complexity of the barrel shifters. The number of data paths chosen must however be a divider of the quasi-cyclic sub-block size.

The remaining question to be answered in this section is how the λ-memory is partitioned in order to enable this scalable architecture. In Section 4 we showed that the DVB-S2 parity check matrix H consists of blocks of $360 * 360$. All symbol-nodes connect to all parity check equation once, according to the diagonal shown inside. The group of 360 λ_n's are distributed over $\frac{360}{D}$ words containing D λ_n's each. The λ_n's are grouped such that for $D = 45$ symbol-node 0, 8, 16, ..., 352 are in one word, 1, 9, 17, ..., 353, are in one word, etc. In this way, it is assured that no matter where the diagonal starts, the 45 corresponding symbol-nodes belonging to the 45 parity equations at distance 8, are in the same word.

In Figure 5, this structure is explained with two blocks of 15. In the first block, the diagonal starts at 0, while in the second block the diagonal starts at 30 + 9 (this is the third block, starting at 30). This implies that for one parity check equation, the symbols 0 and 39 are required, for the next 1, and 40, and for the last in this group 14, and 38. When applying a $D = 3$, three λ_n's are stored in one word. The 3 check-nodes calculated in parallel are highlighted horizontally in Figure 5. For these calculations, first symbol-nodes 0, 5, and 10 are required. Since they belong to one word (first block, first word), they are retrieved in one clock cycle. In the next clock cycle, symbol-nodes 39, 44, and 34 of the next block are required, and they also belong to one word (third block, fourth word). They have to be rotated left over one position in order to match the same order as the previous retrieved word.

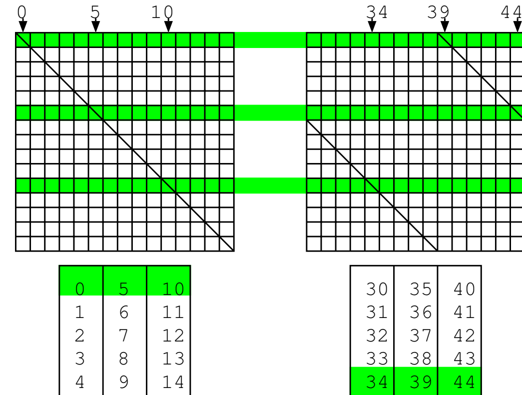

Figure 5. memory partitioning

For the next set of 3 parity check equations, first symbol-nodes 1, 6, and 11 are required, and during the next clock cycle, the symbol-nodes 40, 30, and 35 of the next block are required, which belong again to one word, left rotated over two positions. In fact, this rotation by one position more compared to the previous set, can be accounted for by the overflow in rows: we went from row 4 (containing 34, 39, 44) to row 0 (containing 30, 35, 40). It is the task of the barrel shifters, presented in Figure 4 and mentioned earlier in this section, to facilitate these rotations.

6. Results

In this section we compare the area results of two known solutions ([5], and [8]) with the architecture proposed here. Since in [5] only total area figures are given, the decomposition of area is done based on the ratios in the chip-photos. This area includes test-logic and utilization overhead. From [8] the area figures are taken directly, and are scaled by a factor of 2 in order to resemble 90 nm CMOS results. Our own results are obtained by means of logic synthesis towards a 90 nm CMOS technology. Both our area results and the one presented in [8] exclude test overhead and layout utilization factors. The fig-

ures are shown in Table 1. Although our input memory is bigger, the total memory area is approximately half compared to [5] and [8]. This is mainly due to the compression of messages. The memory required for the application is independent [2] of the parallelism factor chosen. All three solutions have 6 bit inputs values, and 6 bit message values.

90 nm CMOS	[5]	[8]	New
area in mm^2	135Mbps @ 41 it	255Mbps @ 30 it	90Mbps @ 30 it
input mem	1.4	1.0	1.8
message mem	3.8	4.5	1.2
ctrl	-	0.04	0.1
func. logic	8.2	5.4	0.8
ctrl	-	0.1	0.1
shuffling BCH+encoders	0.8 (1.6)	0.3 NA	0.2 NA
total	15.8 (14.2)	11.34	4.1

Table 1. Area cost of LDPC decoder

The decoders in [5] and [8] differ 35% in area, which can be accounted to the difference between making chip and conducting synthesis. This leads to the observation that both solutions have approximately the same area. The proposed architecture however has an area which is a factor 2.7 smaller compared to [5, 8]. For this solution, 8 memories are necessary (against 65 in the [5] solution). Furthermore, the solution replaces a (difficult to layout) 360x barrel shifter, with a 45x barrel shifter. The throughput difference between [8] and our solution is a factor 2.8. When however upscaling the throughput of our solution to the [8] solution, we need to increase the number of data paths to $\frac{255}{2.27} = 112$, which will map to 120 data paths, e.g. an increase of $2\frac{2}{3}$. This will roughly let the area increase to 6 mm^2, approximately half compared to the [8] numbers.

Due to the long simulation times (months) it is time-consuming to evaluate the BER performances of word-width's and techniques discussed in Section 3 downto the required level of $< 10^{-9}$. To reduce this simulation time to days, the proposed architecture has also been mapped to an FPGA, and performances evaluations are ongoing.

7. Conclusions

In this paper we have shown that scalable sub-block parallelism enables efficient implementations. For the investigated DVB-S2 case, this sub-block parallelism leads to a design which is half the size of known solutions. For the required performance in the normative configurations in the broadcast service (90 Mbps), the area is $\frac{1}{3}$ compared to the smallest published decoder. The described technique can be adapted to any block LDPC-code giving it a wider application scope than DVB-S2.

Recommendations for future work include finding more efficient solutions for codes having varying numbers of symbol-nodes connected to check-nodes, and replacing the dual-port symbol-node memories with single port equivalents.

2 different parallelism factors lead to different ratios, which might have different utilizations

References

[1] Digital video broadcasting (dvb); second generation. In *ETSI EN 302 307 v1.1.1*, 2005.

[2] A.J. Blanksby and C.J. Howland. A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder. In *IEEE Journal of Solid-State Circuits*, volume 37, pages 404 – 412, 3 2002.

[3] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X Hu. Reduced-complexity decoding of LDPC codes. In *IEEE Transactions on Communications*, volume 53, pages 1288–1299, 2005.

[4] Mauro Cocco, John Dielissen, Marc Heijligers, Andries Hekstra, and Jos Huisken. A scalable architecture for LDPC decoding. In *IEEE Proceeding of DATE*, 2004.

[5] P. Urard et.al. A 135Mb/s DVB-S2 compliant codec based on 64800b LDPC and BCH codes. In *IEEE Solid-state Circuits Conference (ISSCC)*, 2005.

[6] M. Fossorier and Jinghu Chen. Near optimum universal belief propagation based decoding of low-density parity check codes. In *IEEE Transactions on Communications*, volume 50, pages 406–414, 2002.

[7] R.G. Gallager. Low density parity check codes. In *IRE Transations on Information Theory*, volume 8, pages 21–28, 1962.

[8] F. Kienle, T. Brack, and N. Wehn. A synthesizable IP core for DVB-S2 LDPC code decoding. In *IEEE Conference on Design Automation and Test in Europe (DATE)*, 2005.

[9] E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam. High throughput low-density parity-check decoder architectures. In *IEEE proceedings of GLOBECOM*, volume 5, pages 3019–3024, 2001.

[10] J. Zhao, F. Zarkeshvari, and A. Banihashemi. On implementation on min-sum algorithm and its modifications of decoding low-density parity-check (LDPC) codes. In *IEEE transactions on communications*, volume 53, pages 549–554, 2005.