AREA EFFICIENT DECODING OF QUASI-CYCLIC LOW DENSITY PARITY CHECK CODES

Zhongfeng Wang

School of EECS
Oregon State University
Corvallis, OR 97331 USA

Yanni Chen, and Keshab K. Parhi

Dept. of Electrical & Computer Engineering
University of Minnesota
Minneapolis, MN 55455 USA

ABSTRACT

This paper exploits the similarity between the two stages of belief propagation decoding algorithm for low density parity check codes to derive an area efficient design that re-maps the check node functional units and variable node functional units into the same hardware. Consequently, the novel approach could reduce the logic core size by approximately 21% without any performance degradation. In addition, the proposed approach improves the hardware utilization efficiency as well.

1. INTRODUCTION

Recently, as opposed to random construction of low density parity check (LDPC) codes, regular quasi-cyclic LDPC codes [1] have been proposed that can achieve comparable performance to random codes. Furthermore, their hardware implementations lead to significantly simpler memory address generation and wire interconnections. A similar code construction method has also been extended to irregular quasi-cyclic LDPC codes to further improve the decoding performance [2].

Belief propagation (BP) algorithm [3] is usually employed to decode the LDPC codes, where two types of messages are exchanged iteratively along the corresponding Tanner graph, namely, the variable-to-check messages updated by the variable node functional units (VNFU) and the check-to-variable messages updated by the check node functional units (CNFU). In terms of decoder architectures, lower complexity partly parallel decoder [4] is more practical compared to fully parallel design [5]. In both approaches, the two types of functional units have been implemented with separate hardware.

In this paper, by observing the similarity between the CNFUs and VNFUs, a new scheme is proposed to re-map the CNFUs and VNFUs for the partly parallel decoder into the same hardware to reduce the overall complexity. For the considered example (155, 64, 31) code, the area of logic core is decreased without introducing any performance degradation.

2. BELIEF PROPAGATION DECODING ALGORITHM

Following the original BP algorithm, the check-to-variable message R_{cv} for the check node c and variable node v using the incoming variable-to-check messages L_{cn} is computed by CNFU as follows.

$$S_{cv} = \Pi_{n \in N(c), n \neq v} \, sign(L_{cn}) \tag{1}$$

$$R_{cv} = -S_{cv} \, \Psi \left(\sum_{n \in N(c), n \neq v} \Psi(L_{cn}) \right) \tag{2}$$

where S_{cv} is the sign part of R_{cv} and $N(c)$ denotes the set of variable nodes connected to the check node c. The function $\Psi(x) = log(tanh(|\frac{x}{2}|))$ can be implemented with look-up-table (LUT) operations. On the other hand, the variable-to-check message L_{cv} for the check node c and variable node v using the incoming check-to-variable messages R_{mv} and received channel information r_v is computed by VNFU,

$$L_v = \sum_{m \in M(v)} R_{mv} - \frac{2r_v}{\sigma^2} \tag{3}$$

$$L_{cv} = L_v - R_{mv} \tag{4}$$

where $M(v)$ is the set of check nodes connected to variable node v and $\frac{2r_v}{\sigma^2}$ is the intrinsic information while σ stands for the estimated standard deviation of AWGN channel. The soft output L_v for the variable node v is later sliced to check whether the decoded output is a codeword or not.

According to the above algorithm, the CNFU and VNFU can be implemented as illustrated in Fig. 1 and Fig. 2, respectively. For the sake of clarity, the parity checking part

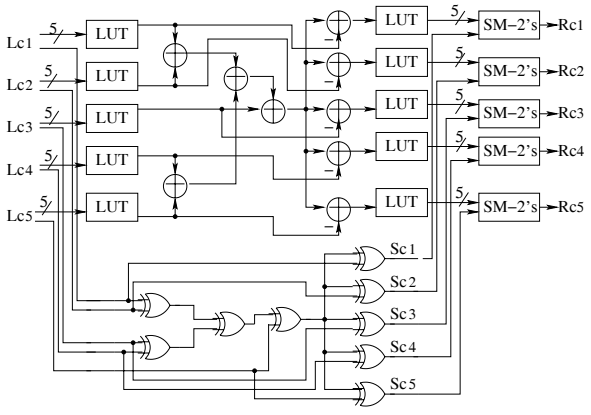

Fig. 1. Original architecture for check node functional unit

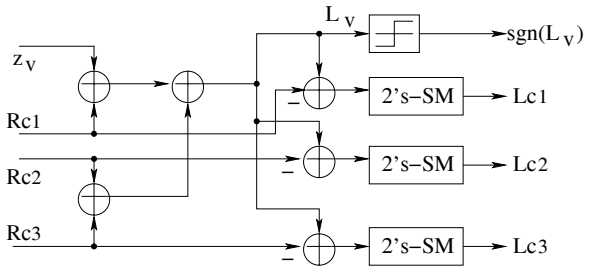

Fig. 2. Original architecture for variable node functional unit

is not shown in Fig. 1 and the intrinsic information $\frac{2r_v}{\sigma^2}$ is represented by z_v in Fig. 2.

Obviously, CNFUs are more complicated than VNFUs. It is also worth noting that data format transformation block, either from sign-magnitude (SM) to two's complement (2's) format or *vice versa*, exists in both types of functional units. The major advantage of using sign-magnitude format for LUT operations is that each LUT size can be reduced by half by making use of the symmetry properties of $\Psi(x)$ function. However, it is still more convenient to use two's complement format in VNFU computations.

3. PARTLY PARALLEL DECODER ARCHITECTURE FOR QUASI-CYCLIC LDPC CODES

The parity check matrix H for quasi-cyclic LDPC codes [1] can be constructed as follows: for a desired (j, k) code, first construct the block all-one matrix H' with size of $j \times k$, then replace each element in H' by a $m \times m$ cyclically shifted identity matrix with certain shift offsets, where m is a prime number and j, k are among the prime factors of $m-1$. The shift offsets are given by $b^{s-1} \times a^{t-1}$ modulo m, where $1 \leq s \leq j$, $1 \leq t \leq k$ and a, b have multiplicative orders of k, j, respectively. As a result, the obtained (j, k) code has parity check matrix H with size of $jm \times km$ and code rate $R \geq 1 - (j/k)$ (there are at least j - 1 dependent rows in H). This code is regular code since both the variable node degrees and check node degrees are constants. To achieve better decoding performance, irregular codes could also be constructed by optimizing the node degrees [2].

Consider the (3, 5) regular quasi-cyclic LDPC code as an example. In terms of partly decoder architecture, one straightforward approach similar to [4] is to use three 5-input CNFUs, five 3-input VNFUs and 15 message memories $m_{s,t}$ with each memory containing m memory words as shown in Fig. 3.

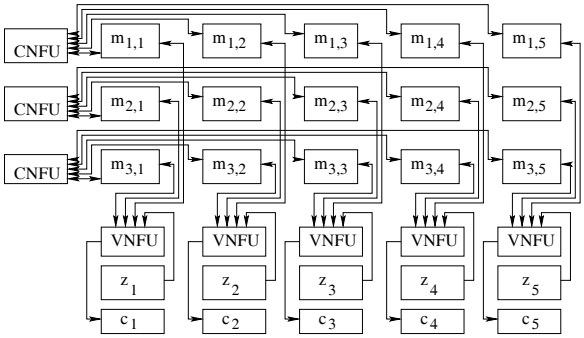

Fig. 3. Partly parallel decoder for (3, 5) quasi-cyclic LDPC code

The intrinsic information are retrieved from z memories and c memories store the hard decisions of the soft outputs $sgn(L_v)$. Consequently, the decoding process could be carried out as follows:

- *Initialization*: flush the received intrinsic information to both the z memories and the corresponding $j \times k$ extrinsic information memories $m_{s,t}$. The data are stored column-wise in the z memories and row-wise in $m_{s,t}$ memories.

- *Check node update*: in each subsequent iteration, the updated variable-to-check messages are simultaneously read from all the $m_{s,t}$ memories by all the CNFUs, each CNFU reads k memories in a *block row*, i.e., those m rows sharing the same s value. Then after CNFU computation the updated check-to-variable messages are written back to the *same* address. Consequently, in one iteration totally m

clock cycles are required to complete the updating process of all the $j \times m$ rows.

- *Variable node update*: similarly, in the same iteration, the updated check-to-variable messages are simultaneously read from all the $m_{s,t}$ memories by all the VNFUs, each VNFU reads j memories in a *block column*, i.e., those m columns sharing the same t value. Then after VNFU computation the updated variable-to-check messages are written back to the *same* address as read operations. Consequently, in one iteration totally m clock cycles are required to complete the updating process of all the $k \times m$ columns.

- *Parity checking*: at the end of every iteration, all the soft outputs are sliced to check all the parity equations. The iterative process will be terminated when either one codeword x satisfying $Hx = 0$ is found, or the pre-assigned maximum number of iterations is reached.

The decoder structure in Fig. 3 has the obvious advantage of memory requirement. In addition, it possesses some other nice features including straightforward memory address generation, localized memory access and simple routing. From the decoding process outlined above, we know that in one iteration both *check node update* and *variable node update* operations have to be performed, one after another. This leads to the merely 50% hardware utilization efficiency (HUE) of the logic core in Fig. 3 because all the VNFUs are idle when CNFUs are busy during the *check node update* and *vice versa* during the *variable node update*. To improve the HUE of logic core, the approach we considered is to re-map the VNFUs and CNFUs into the same hardware by making use of similarity between the two stages of BP algorithm to get smaller area design.

4. AREA EFFICIENT BP DECODING

By re-distributing the computation load between CNFUs and VNFUs, the two stages of BP algorithm could be equivalently reformulated as follows. Then for CNFUs,

$$S_{cv} = \Pi_{n \in N(c), n \neq v} \, sign(L_{cn}) \tag{5}$$

$$R_{cv} = -S_{cv} \sum_{n \in N(c), n \neq v} \Psi(L_{cn}) \tag{6}$$

and for VNFUs,

$$L_v = \sum_{m \in M(v)} (-sign(R_{mv})\Psi(R_{mv})) - \frac{2r_v}{\sigma^2} \tag{7}$$

$$L_{cv} = L_v + sign(R_{mv})\Psi(R_{mv}) \tag{8}$$

where all the notations remain the same as in Section 2. As a result, their corresponding architectures are depicted in Fig. 4 and Fig. 5, respectively.

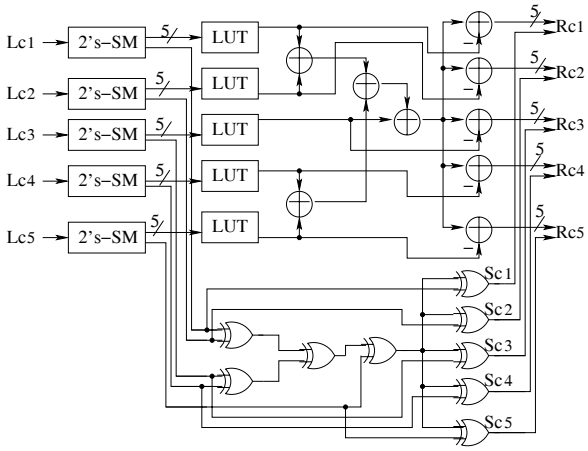

Fig. 4. Reformulated architecture for check node functional unit

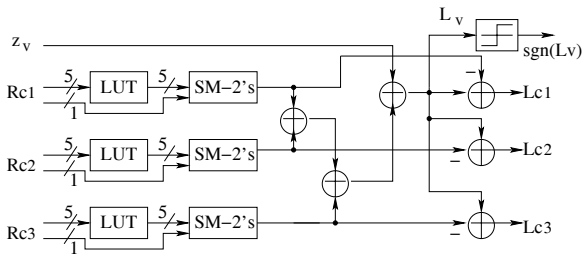

Fig. 5. Reformulated architecture for variable node functional unit

In this way, the two stages of BP decoding are better balanced since each CNFU or VNFU has one LUT operation in the critical path as opposed to two LUTs in CNFU and none in VNFU in the original architectures. However, the HUE is still only 50%. To improve the HUE, an alternative approach compared to [6] is considered in this paper. The main idea is that the entire set of both CNFUs and VNFUs can be re-mapped into the same hardware as illustrated in Fig. 6 by realizing that the numbers of inputs processed by CNFUs and VNFUs per clock cycle are the exactly same, which are equal to the total number of non-zero entries in the block matrix H'. In Fig. 6, there are totally 15 inputs and 15 outputs denoted as In_p and Out_p, respectively, where $1 \leq p \leq 15$. The new re-mapped hardware performs CNFU operations when the

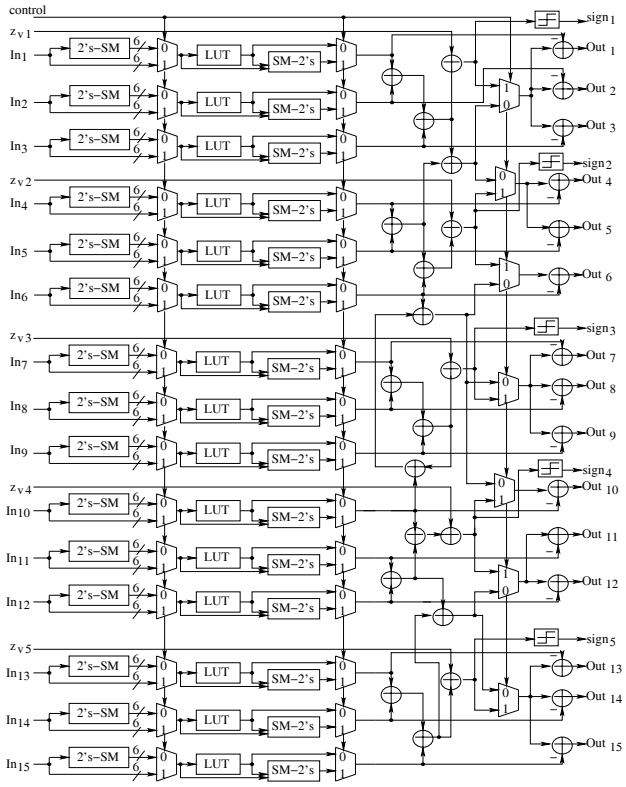

Fig. 6. New remapped hardware performing both CNFUs and VNFUs operations

control signal is $'0'$, thus the inputs are variable-to-check messages and the outputs are check-to-variable messages. On the other hand, when the *control* signal switches to $'1'$ VNFU operations are performed, where the inputs are check-to-variable messages and the outputs are variable-to-check messages. Therefore, both VNFUs and CNFUs operations could be performed by the same piece of hardware, which is always busy in every iteration and thus the HUE is increased to 100%.

To compare the area of the new remapped hardware with the original approach using separate VNFUs and CNFUs, both were described in VHDL, simulated and synthesized by $Synopsis^{TM}$ tools. Some results are shown in Table 1.

From Table 1, it is easily seen that the area of the proposed architecture is reduced by 21% compared to the original design. Furthermore, no performance degradation is introduced. Here the example we considered is (3, 5) code with relatively short length. However, the hardware saved would remain the same for a longer code obtained by choosing a larger m since the changes are only in VNFUs and CNFUs that are independent of the length.

Table 1. Area comparison between original and proposed architectures

	number of 6–bit adders	number of 32–entry LUTs	number of 2–to–1 MUXs	Area number of gates
CNFU	9	10	none	1603
VNFU	6	none	none	565
Original set	9*3 + 6*5 = 57	10*3 = 30	none	7634
Remapped set	34	15	37	6041

5. CONCLUSIONS

In this paper, for a class of quasi-cyclic LDPC codes, a scheme using common hardware to compute both bit nodes and check nodes is described, which leads to about 21% area reduction of functional units part for our considered $(3, 5)$ code. In fact, for any (j, k) quasi-cyclic LDPC code, if the row weight k is a multiple of column weight j, namely, the number of inputs processed by CNFUs is the multiple of the number of inputs processed by VNFUs, the remapping process between VNFUs and CNFUs is certainly more straightforward and simpler. Moreover, as the numbers of CNFUs and VNFUs are increased more area saving is expected due to the larger percentage of logic sharing.

6. REFERENCES

[1] D. Sridhara, T. Fuja, and R. M. Tanner, "Low density parity check codes from permutation matrices," *Conf. on Info. Sciences and Systems*, The John Hopkins University, March 2001.

[2] D. Hocevar, "LDPC code construction with flexible hardware implementation," *in Proc. ICC*, 2003.

[3] D. J. C. Mackay, "Good error correcting codes based on very sparse matrices," *IEEE Trans. Inform. Theory*, vol. 45, pp. 399-431, March 1999.

[4] T. Zhang and K. K. Parhi, "A 54 MBPS (3, 6)-regular FPGA LDPC decoder," *in Proc. IEEE SIPS*, pp. 127-132, 2002.

[5] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *IEEE J. Solid-State Circuits*, vol. 37, pp. 404-412, 2002.

[6] Y. Chen and K. K. Parhi, "High throughput overlapped message passing for low density parity check codes," *in Proc. IEEE/ACM GLSVLSI*, pp. 245-248, 2003.