International Journal for Innovative Engineering and Management Research
A Peer Revieved Open Access International Journal
www.ijiemr.org

Efficient Implementation of Non-binary Low Density Parity Check (LDPC) Decoder in VLSI

*S.Sahaja **Mr.B.Ranjith Kumar ***Mr.M.Devadas

*M.Tech Dept of E.C.E, Vaagdevi College of Engineering
**Assistant. Prof Dept of E.C.E, Vaagdevi College of Engineering
***Assistant. Prof Dept of E.C.E, Vaagdevi College of Engineering

***Abstract*—**The best error-correcting performance can be achieved by using non-binary low-density parity check (NB-LDPC) codes. This can be of reduced decoding complexity with high cost efficiency and is mostly preferable than binary low density parity check codes. The proposed scheme not only reduces the computation complexity, but also eliminates the memory requirement for storing the intermediate messages generated from the forward and backward processes. A novel scheme and corresponding architecture are developed to implement the elementary step of the check node processing. In the design, layered decoding is applied and only nm<q messages are kept on each edge of the associated Tanner graph. The computation units and the scheduling of the computations are optimized in the context of layered decoding to reduce the area requirement and increase the speed. There by using this forward and backward process the memory requirement of the overall decoder can be substantially reduced. Thus to conclude that the above scheme leads to significant memory and complexity reduction inspired by the proposed check node processing scheme In addition, efficient architecture have been designed for the sorter and path constructor, and the computational scheduling has been optimized to further reduce the overall area and latency.

Keywords-Low-density parity check codes(LDPC),latency, Tanner graph, check node processing, sorter, path constructor

I. INTRODUCTION

Error-correcting coding has become one integral part in nearly all the modern data transmission and storage systems. Low-density parity-check (LDPC) codes are a class of linear codes which provide near-capacity performance on a large collection of data transmission and storage channel while still maintaining implementable decoders. Due to the powerful error-correcting capability, LDPC codes have been used as error-correcting codes with applications in wireless communications, magnetic and optical recording, and digital television.[2]

Since the first proposition of LDPC codes by Gallager in his 1960 doctoral dissertation, LDPC codes were mostly ignored for the following three decades. One notable exception is the work of Tanner in which a graphical representation of LDPC codes was introduced, now widely called Tanner graph. In mid-nineties, LDPC codes were re-discovered by MacKay, Luby, Wiberg and others. In their works, the sparsity of the parity check matrix of LDPC codes was utilized and belief propagation method proposed by was adopted to significantly reduce the decoding complexity [1]. After that, numerous research efforts have been done and various decoding algorithms are proposed.

When the code length is moderate, non-binary low-density parity-check (NB-LDPC) codes can achieve better error-correcting performance than binary LDPC codes at the cost of higher decoding complexity.[2].However, the complicated computations in the check node processing and the large memory requirement put obstacles to efficient hardware implementations. VLSI architecture design connects advanced error-correcting theories and their efficient hardware implementations through algorithmic and architectural level optimizations. Such optimizations are crucial to meet all kinds of implementation requirements set up by hardware applications. In this paper, efficient VLSI architectures design targeting at non-binary (NB) LDPC decoder are presented in this paper. The designs include two decoders based on two different check node unit

(CNU) processing, which are forward-backward and trellis based CNU processing, respectively.

II. TRELLIS-BASED CHECK NODE PROCESSING

In this section, a novel check node processing method and corresponding efficient architectures for both Min-max and EMS NB-LDPC decoding are proposed. Employing either the Min-max or EMS algorithm, each c-to-v probability from a check node can be derived from a limited number of the smallest LLRs in the input v-to-c message vectors. Based on this observation, a limited number of the most reliable v-to-c messages are first sorted out. These messages can be considered as nodes in a trellis. Then the c-to-v messages to all connected variable nodes are generated independently using a path construction scheme without storing other intermediate values. As a result, both the memory requirement and logic gate count can be reduced significantly without sacrificing the speed

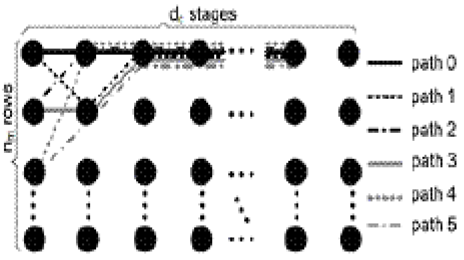


Figure 1. Trellis of variable-to-check messages.

For the decoder based on Min-max algorithm, compared to the forward backward check node unit (CNU) architecture, the proposed architecture can reduce the memory and area requirements to 19.5% and 22.5%, respectively, when applied to a (837, 726) NB-LDPC code over *GF*(25). Inspired by this novel c-to-v message computation method, and the fact that the path construction can be implemented efficiently, we propose to store the most reliable v-to-c messages as 'compressed' c-to-v messages. The c-to-v messages will be recovered from the compressed format when needed. Accordingly, the memory requirement of the overall decoder can be substantially reduced. In addition, an optimized scheduling scheme has been developed for the involved computations to further reduce the area and latency. Compared to the previous Min-max decoder architecture, the proposed design for a (837, 726) code can achieve 54% area reduction with the same throughput. For the decoder based on EMS algorithm, compared to the CNU architecture, our architecture only requires 64% of the area without sacrificing both the throughput and error-correcting performance. This work extended the idea of trellis-based Min-max check node processing. However, the path construction architecture design is fundamentally different and much more challenging due to the involved 'sum' instead of 'max' computations[3]. The overall EMS NB-LDPC decoder is the same to the Min-max decoder except that a different path constructor is employed.

III. LDPC ENCODER

LDPC codes have easily parallelizable encoding and decoding algorithms. The parallelizability is 'adjustable' providing the user an option to choose between throughput and complexity. The function of the encoder is to add extra redundant data for given decoded data. This extra redundant data, called as parity data is useful in detecting the errors that are introduced during the data transmission through a channel.

The functional block diagram of an encoder is given below with their functional specifications. The LDPC encoder receives code rate, frame size long with the data that need to encode as inputs. LDPC Encoder has three main functional blocks. Rate dependent variable generator, Parity addresses generator and parity generator. For a given code rate and frame size, rate dependent variable generator block generates the values information bit length, parity bit length and repeatability rate which are used in encoding[4]. Parity address generator contains the address of information bits that need to be ex-ored for generating a given parity bit. Parity generator receives the information bits from input and address values from parity address generator block to generate the parity data.

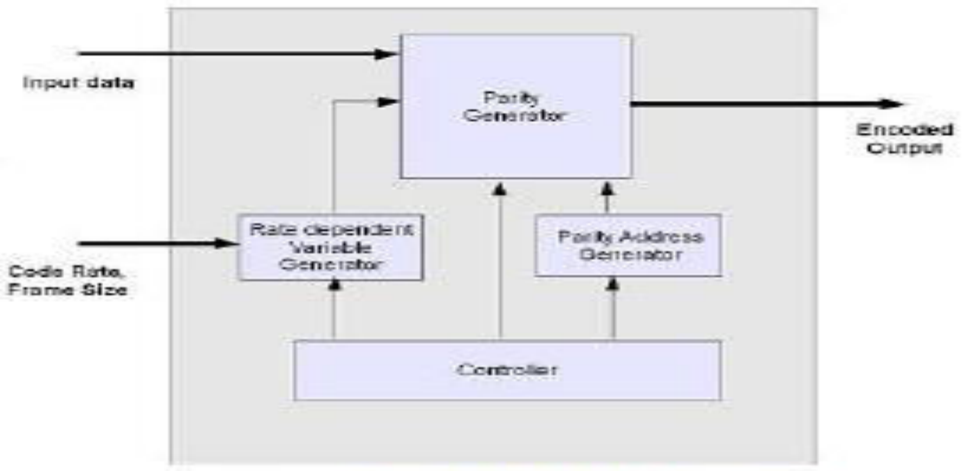


Figure 2. Functional diagram of LDPC Encoder

IV. SIMULATION RESULTS OF AN LDPC ENCODER

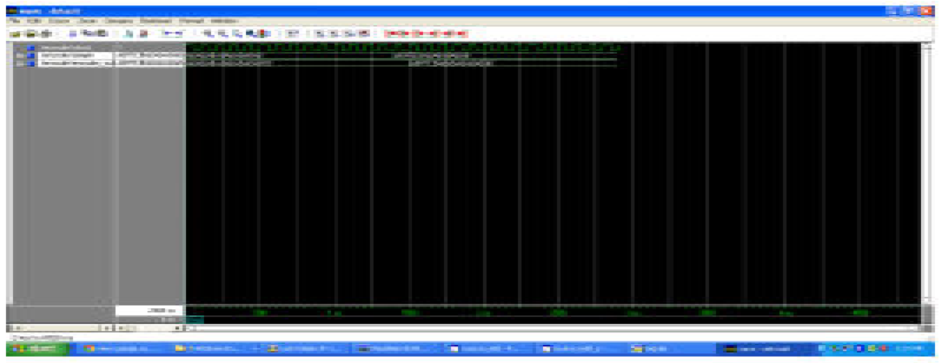


Figure 3. Hexadecimal output

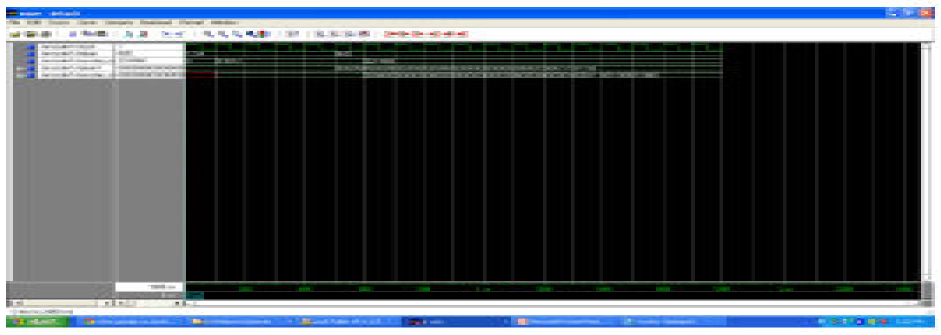
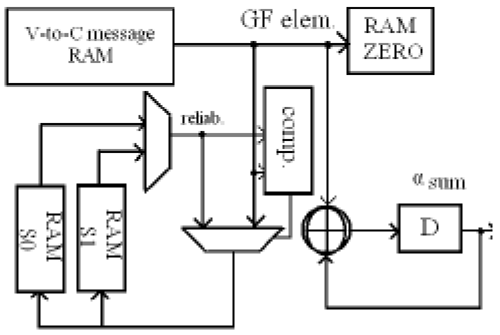


Figure 4. Integer output

V. REDUCED-COMPLEXITY CNU ARCHITECTURE

Using the proposed scheme, the CNU architecture consists of two parts: a sorter that sorts out the 1.5*nm* incoming v-to-c messages with the smallest nonzero LLRs, and a path constructor that generates the c-to-v messages from the sorting results. In the following, the architectures for these two parts are presented.

A. V-to-C message Sorter



The figure 5 shows the architecture for the sorter. The shaded blocks denote RAM blocks. A pair of RAM S blocks, denoted by RAM S0 and S1, is used to store the sorting results in a ping-pong manner. Each RAM S can record 1.5*nm* messages and the indices of the variable nodes they belong to. Hence, the size of each RAM S is 1.5*nm*(*w* + *p* + log2 *dc*) bits[5]. The finite field elements associated with zero LLRs are stored into RAM Zero when they are read out from the v-to-c message RAM. Accordingly, RAM Zero is of size *dc*×*p* bits. In addition, these field elements are added up to compute *αsum* by the adder-register loop in the bottom right corner of the above figure 5.

The sorting is carried out iteratively in *dc* rounds. In the first round, the *nm* − 1 v-to-c messages with nonzero LLRs of the first variable node are copied into RAM S0. In addition, '0' is written to RAM S0 as the variable node index.[6] In the second round, comparisons are carried out on the messages stored in RAM S0 and the v-to-c messages of the second variable node to find the 1.5*nm* messages with the smallest nonzero LLRs. The comparison output vector is written into RAM S1. Since the entries in each v-to-c message vector are stored in the order of increasing LLR, the comparison can start with the first nonzero-LLR entries in the two input vectors. If the LLR from RAM S0 is smaller, the corresponding entry will be stored into RAM S1. In addition, the next entry of RAM S0 will be read out and sent to the comparator in the next clock cycle. Otherwise, the v-to-c message is stored into RAM S1 together with the current variable node index.

Similarly, the next entry in the v-to-c message vector will be read out and sent to the comparator in the next clock cycle. Such comparisons will be repeated until 1.5nm entries are derived for the output vector. In each of the third and later rounds, comparisons are carried out on the output vector from the previous round and the v-to-c message vector of another variable node. RAM S0 and S1 are used in a ping-pong manner to store comparison input and output vectors. Since 1.5nm entries are kept in the output vector, 1.5nm clock cycles are required for the comparisons in each of the second and later rounds of the sorting.[7] In addition, one clock cycle is spent on reading the zero-LLR entry from the v-to-c message RAM in each round. Hence the sorting takes $nm + (dc - 1)(1.5nm + 1)$ clock cycles.

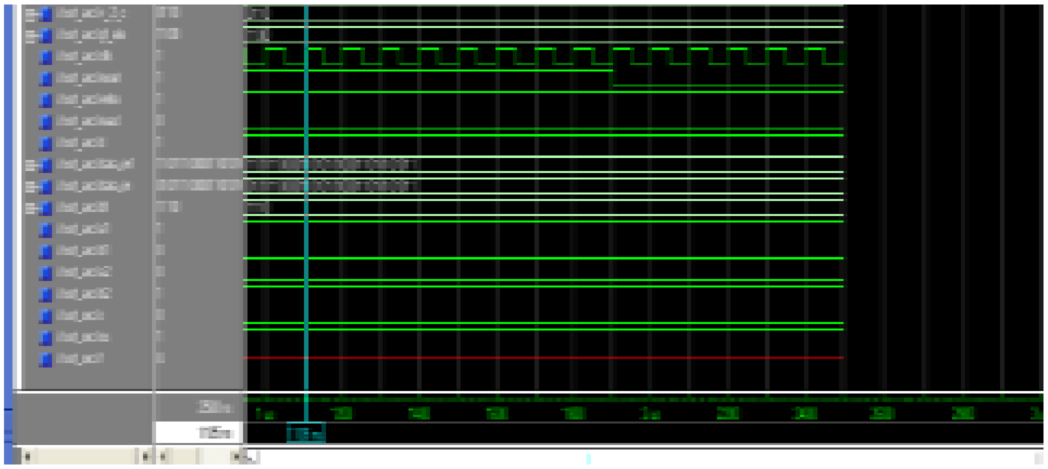


Figure 6. Sorter Output

B. Path constructor

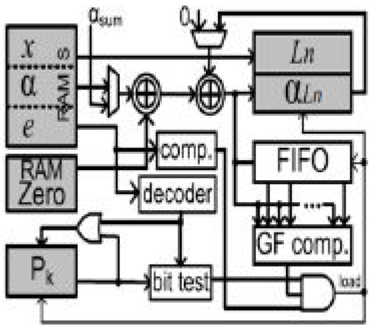


Figure 7. Path Construction Architecture

The above figure illustrates the path construction architecture that implements Algorithm B. The 1.5*nm* sorted v-to-c messages are read from RAM S one at a time, starting from the first one, which has the smallest nonzero LLR. During the path construction for computing *vm,n*, *e*(*i*), the index of the variable node that the message belongs to, is first compared with *n* in the comparator block[8]. This block can be implemented by log2 *dc* XOR gates and a log2 *dc*-input OR gate. If it outputs '0', the message read out is from variable node *n*, and thus should not be included in the path construction. Otherwise, *e*(*i*) is passed to the decoder to generate a *dc*-bit binary vector, in which only the *e*(*i*)*th* bit is '1'. The bit test block in Fig. 4 carries out bit-wise AND on this vector and *Pk*. Then the outputs of the AND gates are passed to a *dc*-input NOR gate.

Hence, the bit test block outputs '1' when *Pk*(*e*(*i*)) ≠1. The *α* in Algorithm B is computed by the two finite field adders in Fig. 4.4. The two multiplexors are added to enable the computation of *αLn*(0) at the initialization. In addition, when *α* is inserted into the *αLn* vector, it is also copied to the first-in-first-out (FIFO) buffer consisting of serially concatenated registers. In this way, each element in *αLn* can be

simultaneously compared with the newly computed *α* in the GF comparator to test if *α* *αLn*.

The GF comparator outputs '1' when *α* equals none of the elements in the FIFO. When the comparator, bit test and GF comparator all output '1', the load signal at the output of the AND gate in the bottom right corner of Fig. 4 is asserted. Accordingly, *α* and the *x*(*i*) read from RAM S are loaded into the *Ln* and *αLn* memories, respectively. In addition, the vector for the new path is generated by the bit-wise OR gates, and loaded into the *Pk* memory.

Using the proposed CNU architecture, only 1.5*nm* sorted v-to-c messages need to be stored for each check node. Compared to storing *dcnm* intermediate messages for each of the forward and backward processes, the memory requirement has been substantially reduced. In addition, this architecture also requires much less logic gates. As it will be shown in Section VI, the proposed CNU architecture for a (837, 726) NB-LDPC code with *dc* = 27 only requires 22.5% of the area of the forward-backward-based CNU architecture.

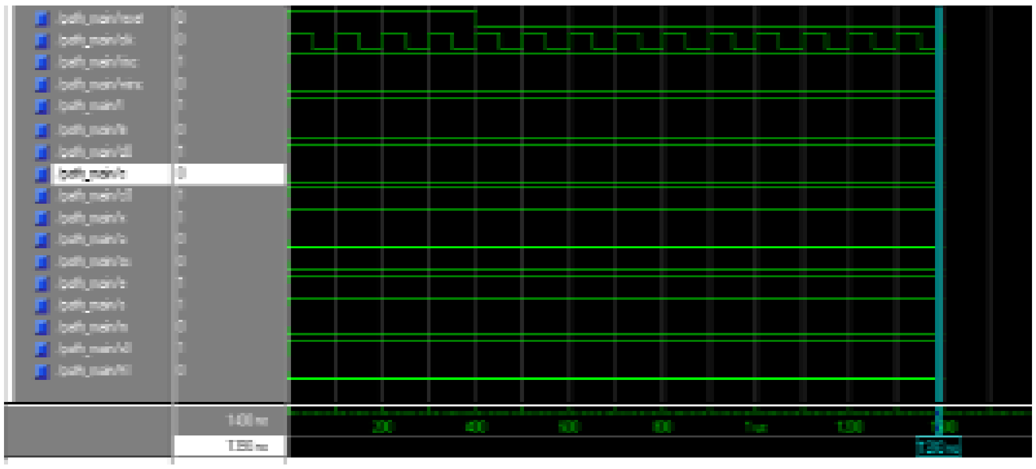


Figure 8. Path Constructor Output

VI. TRELLIS-BASED NB-LDPC DECODER ARCHITECTURE

This section considers the decoder design for QCNB-LDPC codes, whose *H* matrix can be divided into sub-matrices. Each sub-matrix can be a zero matrix or shifted identity matrix with nonzero entries replaced

by elements of *GF*(*q*) [25]. QCNB-LDPC codes can achieve very good error-correcting performance. In addition, they are more suitable for efficient high-speed parallel decoding than other NB-LDPC codes due to the regularity in the *H* matrix [9]. As it was mentioned previously, large memory requirement is a bottleneck of NB-LDPC decoder implementation since message vectors instead of single values are passed between check and variable nodes. In fact, the memory accounts for 84% of the overall area of the (837,726) partial-parallel

QCNB-LDPC decoder. Inspired by the novel check node processing scheme presented in the previous section, we propose to store the c-to-v messages in a 'compressed' format. For each check node, only the 1.5nm sorted nonzero-LLR messages and the field elements for zero-LLR messages from the connected variable nodes are stored. c-to-v messages will be computed from them when needed. Although more copies of the path constructors will be needed, they occupy much smaller area than the memory required fo storing the c-to-v messages. Accordingly, the overall decoder area can be reduced further by a large factor.

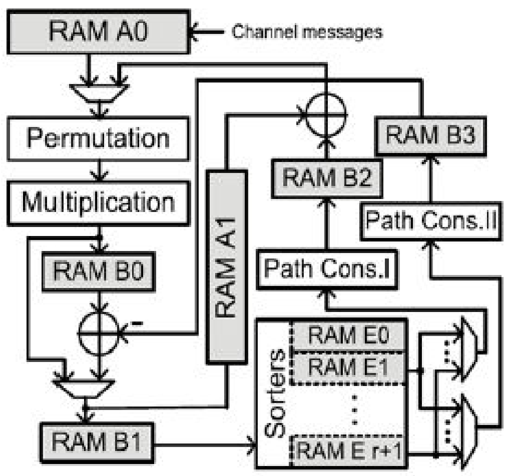


Figure 9. Architecture of Partial-Parallel Reduced-Complexity QCNB-LDPC Decoder.

In our design, layered decoding is employed to reduce the memory requirement and increase the decoding convergence speed. In layered decoding, the parity check matrix H is divided into block rows, also called layers. The c-to-v messages derived from the decoding of one layer are used right away to update the v-to-c messages of the next layer. Using the construction methods proposed in [5], the H matrix of a QCNB-LDPC code over $GF(2p)$ can be divided into $r \times t$ sub-matrices of dimension $(2p - 1) \times (2p - 1)$. Accordingly, it can be divided into r layers and the computation for $2p - 1$ rows of H are carried out at a time.

The top level architecture of our proposed partial-parallel QCNB-LDPC decoder is shown in figure 9. Three types of RAM blocks are used in this architecture. Each copy of RAM A is capable of storing nm messages for each of the $t(2p - 1)$ variable nodes. Hence its size is $nm \times t \times (2p - 1) \times (p + w)$ bits. It consists of two parts: one for LLRs and one for corresponding finite field elements. In our design, the computations for one block column ($2p - 1$ column) of H are carried out at a time. Accordingly, each part of RAM A is divided into $2p - 1$

individual RAMs to enable simultaneous access of necessary messages. Each RAM B has two sub-blocks. Each sub-block is of similar architecture to RAM A, except that it can only store the messages for a single block column of *H*. Therefore, the size of a RAM B is $2nm\times(2p-1)\times(p+w)$ bits. The RAM E blocks inside the sorters serve the same purpose as the RAM S blocks in figure 9. $2p - 1$ copies of the sorters are employed in the decoder to process one layer of *H* at a time. Hence, each RAM E consists of $2p - 1$ copies of RAM S and its size is $1.5nm \times (2p - 1) \times (w + p + log2dc)$ bits. The size, data width, memory depth and memory block number for each type of RAMs used in our decoder.

At the beginning of the decoding, the channel information is loaded into RAM A0, and is used as the v-to-c messages for the first layer in the first decoding iteration.The permutation block in the above Figure is composed of barrel shifters. It routes messages for check node processing according to the locations of the nonzero entries of *H*. In addition, the multiplications of the finite field elements of the messages by the corresponding nonzero entries of *H* are carried out in the multiplication block. After that, the messages are buffered by RAM B1. One sub-block of RAM B1 serves as the v-to-c message RAM to the sorters, while the other stores the messages for the next block column of *H* computed by the previous decoder unit. Such buffering is necessary since the messages are not read out one in each clock cycle by the sorters as discussed previously.

During the sorting process, two RAM E blocks are used in a ping-pong manner as the RAM S0 and RAM S1 in Fig. 4. Then the c-to-v messages can be recovered from the sorting results using Path Constructor-I, which consists of $2p - 1$ copies of the architecture in Fig. 4.6. For QCNB-LDPC codes, each column of *H* has at most one nonzero entry in each layer. Denote the v-to-c LLRs of layer *l* in the *jth* decoding iteration by $u\ l\ (j)$. Represent the c-to-v LLRs computed from layer *l* in the *jth* iteration by v l

$$u_{l+1}^{(j)} = (u_l^{(j)} + v_l^{(j)}) - v_{l+1}^{(j-1)}. \quad (j).$$

It can be derived that

Hence, to compute the v-to-c messages for the next layer, the outputs of path Constructor-I need to be added to the inputs of the sorters to generate the a posteriori messages (u l (j) + v l (j)). To enable this computation, the inputs to the sorters are also stored into RAM A1 while they are buffered by RAM B1. After the outputs of Path Constructor-I are buffered by RAM B2, the a posteriori messages are computed by the adder in the top right corner of Fig. 4. Division and reverse permutation should be applied to the a posteriori messages to reverse the effects of multiplication and permutation, respectively. However, the a posteriori messages can be used to compute the v-to-c messages of the same block column of H for the next layer right after. Hence, the division for a column in layer l can be combined with the multiplication for the same column in layer l+1 as a single constant multiplication, since the entries of H are known[10]. Similarly, the reverse permutation for layer l can be incorporated with the permutation of the same block column for layer l + 1. Hence, there are no division and reverse permutation blocks in our decoder. v l (−1) = 0 in the decoding of later layers in the first iteration. Hence, for these layers, the a posteriori messages computed from the previous layer are the v-to-c messages of the current layer. Starting from the second decoding iteration, the c-to-v messages of the same layer from the previous iteration need to be subtracted from the a posteriori messages to generate the v-to-c messages storing the c-to-v messages of each layer requires a RAM A, which is significantly larger than a RAM E when dc is not small. A RAM A also occupies substantially more area than path constructors. Accordingly, we propose to store the sorted v-to-c messages instead of c-to-v messages. When c-to-v messages are needed in the decoding of the same layer in the next decoding iteration, they are recovered from the sorting results using Path Constructor-II, which also has $2p - 1$ copies of the architecture in figure 9. $r + 2$ RAM E blocks are required to store the results and intermediate data of the sorting. The allocation of these RAM E blocks will be detailed in the next subsection. The two multiplexors in the bottom right corner of figure 9 are used to pass proper sorted messages to the two sets of path constructors.

It should be noted that the adder and subtractor in Figure are not component-wise vector adder and subtractor. Since nm < q messages are kept for each vector, it is possible that for a message in one vector, there is no message with the same finite field element in the other vector. Taking this into account, the addition/subtraction is carried out in two rounds. Denote the two input vectors to the adder/subtractor by row and column vectors[3]. In the first round, one entry in the row vector is read out in each clock cycle. If there is an entry in the column vector with matching finite field element, then the corresponding LLR is added/subtracted by that from the row vector. In addition, a flag is set for the entry in the column vector. If there is no entry with matching field element in the column vector, compensation LLR is used for the column vector. It has been shown that setting the compensation LLR to the largest LLR in the vector does not lead to noticeable performance loss.

In the second round, one entry is read out from the column vector at a time. If the corresponding flag is not set, its LLR is added up/subtracted by the compensation LLR of the row vector. The output vector also needs to be kept sorted according to increasing LLR. Hence, the sums/differences from the two rounds are sent to a parallel sorter, which has nm comparators, nm registers and $nm-1$ multiplexors. This parallel sorter can insert a number into a sorted sequence of length nm in one clock cycle. As a result, the addition/subtraction of two vectors can be completed in $2nm$ clock cycles, after which the output vector can be found at the registers of the parallel sorter.

VII. CONCLUSION

This paper focuses on the design of VLSI architectures for NB-LDPC decoders. The complexity of the check node processing is further reduced in the Min-max algorithm with slightly lower coding gain.More over, layered decoding can be adopted to reduce the memory requirement and increase the convergence speed of binary LDPC decoding. Novel forward-backward partial-parallel layered decoder architecture for QCNB-LDPC codes based on the Min-max algorithm is presented. The proposed design can achieve significantly higher efficiency than prior efforts, especially when the check node degree is not small. When combined with layered decoding, the overlapped scheme also leads to the elimination of the division and reverse permutation blocks. A novel check node processing schemes for both Min-max and EMS NBLDPC decoding algorithm is also proposed. The path construction scheme for EMS algorithm is more complex than that of the Min-max algorithm. Compared to the forward-backward check node processing, the proposed trellis based check node processing scheme leads to significant memory and complexity reduction. Inspired by the proposed check node processing scheme, the sorted v-to-c messages are stored in a compressed version, which will be recovered when needed. As a result, substantial memory saving can be achieved for the overall decoder. In addition, efficient architectures have been designed for the sorter and path constructor, and the computation scheduling has been optimized to further reduce the overall area and latency

REFERENCES

[1] Ali E.Pusane, Pascal 0.Vontobel and Daniel J.Costello ,"Deriving Good LDPC Convolutional Codes From LDPC Block Codes" ,IEEE Trans. IT, vol.57,PP.835-857, 2011.

[2] D. MacKay,"Good error correcting codes based on very sparse matrices", IEEE Trans.IT, vol.23, pp. 399–431,1999.

[3] J. Lin, J. Sha, Z. Wang, and L. Li, "An efficient vlsi architecture for nonbinary ldpc decoders" ,IEEE Trans. on Circuits and Systems-II ,vol. 7, pp. 532-542, 2010.

[4] Lucile Sassatelli and David Declereq, "Non Binary Hybrid LDPC Codes'",IEEE Trans.IT,vol.56,pp.5314-5334, 2010.

[5] Mohammad M. Mansour and Naresh R. Shanbhag ,"High-Throughput LDPC Decoders", IEEE Trans.IT, Vol. 11, pp.976-996 ,2003.

[6] M. Luby and et. Al, "Improved low-denstiy parity check codes using irregular graphs" , IEEE Trans.IT, vol.21, pp. 585–598,2001.

[7] M. Davey and D. J. MacKay "Low density parity check codes over gf(q)", IEEE Commun. Letter ,vol. 2, pp. 165–167, 1998.

[8] Nissim Halabi and Guyeven (2011), "LP Decoding of Regular LDPC Codes in Memoryless channels" IEEE Trans.IT,vol.57,pp.887-897, 2011.

[9] R. M. Tanner, "A recursive aproach to low complexity codes," IEEE Trans.on Information Theory., no. 9, pp. 533–547, 1981.

[10] Sangmin kim ,Gerald E.Sobel and Hanlo Lee, "A Reduced-Complexity Architecture For LDPC Layered Decoding Schemes" IEEE Trans.VLSI,vol.19,pp.1099-1103, 2011.

AUTHOR 1:-

S.Sahaja completed her B-tech in Sumathi Reddy Institute of Technology For Women. in 2014 and completed M-Tech in Vaagdevi college of Engineering.

AUTHOR 2:-

Mr.B.Ranjith Kumar is working as Assistant. professor in Dept of ECE, Vaagdevi College of Engineering.

AUTHOR 3:-

Mr.M.Devadas is working as Assistant.professor in Dept of ECE, Vaagdevi College of Engineering.