FPGA Based Implementation of Decoder for Array Low-Density Parity-Check Codes

Pankaj Bhagawat
Texas A&M University

Momin Uppal
Texas A&M University

Gwan Choi, Member IEEE
Texas A&M University

ABSTRACT

In the past few years, the Low Density Parity Check (LDPC) codes have received lot of attention for their excellent performance, and inherent parallelism involved in decoding them. In this work we consider a type of structured binary LDPC codes known as array LDPC codes, which have low encoding complexity and good performance, for implementation on Xilinx Field Programmable Gate Array (FPGA) device.

1. INTRODUCTION

A basic communication system is composed of three parts: a transmitter, channel, and receiver. Transmitted information becomes altered due to noise corruption and channel distortion. To account for these errors, redundancy is intentionally introduced, and the receiver uses a decoder to make corrections. In error-correcting control (ECC), it is important to have a high code rate while maintaining low complexity.

It's a well-known fact that the LDPC codes can achieve information rates very close to the Shannon limit when iteratively decoded [1]. LDPC decoders are also known to have arithmetic computations requirement that is an order of magnitude less than Turbo decoders [2] that provide similar bit-error rate (BER) performance. Furthermore, algorithms for decoding LDPC codes also have the advantage of being inherently parallel. In principle, this permits exploiting the common approach of using multiple parallel processing elements to increase the throughput of the decoder. As far as hardware implementation of the decoder is concerned, however, the exploitation of parallelism is a serious challenge. This is due to the immense complexity of the interconnects between the processing elements.

This paper discusses the implementation of array based LDPC codes (ALDPC). We successfully show that the ALDPC codes can be efficiently implemented using a partly parallel architecture [4]. Beyond just bit-error rate (BER) performance the objective is to demonstrate simplicity, and high efficiency of the hardware due to the structure imposed on the code.

2. REVIEW OF LDPC CODES

The LDPC code is a class of parity check codes, which can be fully defined by a parity–check matrix **H**. To be specific, it is defined as the null space of a very sparse *M*x*N* parity check matrix **H**, an LDPC code is represented by a bipartite graph, called Tanner graph (see fig. 1), in which one set of *N bit* or *variable* nodes corresponds to the set of codeword, another set of *M check* nodes corresponds to the set of parity check constraints and each edge corresponds to a non-zero entry in the parity check matrix **H.**

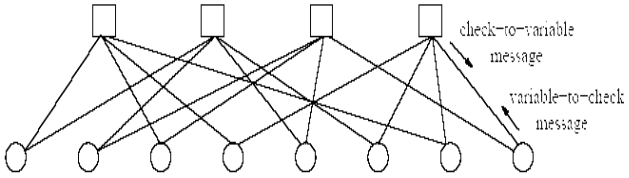

Figure 1. Tanner graph representation of LDPC code.

Effective decoding of LDPC codes can be done by an iterative algorithm called belief-propagation (BP) (also known as sum-product). The structure of BP decoding algorithm directly matches the Tanner graph, and in principle, enables us to make use of the parallelism. Unfortunately, randomly constructed LDPC codes entail a random layout of the interconnects between variable and check nodes. This makes the exploitation of parallelism very difficult because of limited routing resources available. For instance, a decoder based on direct mapping of BP algorithm is presented in [3], for a relatively short code this design uses up 50% of the chip area just for routing the messages. This is only going to get worse as the code length increases.

In this paper we make use of the partly parallel architecture for a special class of LDPC codes called ALDPC. The structure of ALDPC codes is one of the best suited for partly parallel architecture. Using ALDPC codes greatly simplifies the routing complexity, thus making is suitable for implementation on an FPGA device.

3. ARRAY BASED LOW DENSITY PARITY CHECK CODES

The **H** matrix based on array codes is different from the usual **H** matrices. It can be represented as follows [5].

$$H = \begin{bmatrix} I & I & I & \cdots & I \\ I & \beta & \beta^2 & \cdots & \beta^{k-1} \\ I & \beta^2 & \beta^4 & \cdots & \beta^{2(k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ I & \beta^{j-1} & \beta^{2(j-1)} & \cdots & \beta^{(j-1)(k-1)} \end{bmatrix}$$

Where I is a p x p identity matrix with a prime number p. The dimensions of the matrix are N x K = jp x kp with integers $j, k \le p$. β is a permutation matrix with a single cyclic right shift or left shift. For p = 5 resultant β is displayed below.

$$\beta = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Efficient encoding, minimum distance properties and good performance of ALDPC codes have been shown in [5]. Array codes have almost the same BER performance as randomly generated codes when an interleaver is used between the encoder and the channel. Array codes have a quasi-cyclic structure that can be exploited to greatly reduce the hardware complexity of the decoder maintaining comparable error rate and throughput at the same time. This makes it a very attractive candidate for high-density storage applications. In addition, by dividing the **H** matrix into multiple identity/permutation matrices the interconnections become localized thus lowering the routing complexity.

4. DECODING THE LDPC CODES

Belief Propagation algorithm has been described in numerous articles before. Here we merely represent the BP algorithm [4]. Following steps summarize the BP algorithm:

1. Initialize each variable node with the intrinsic (or channel) information, and from this compute the variable-to-check message.
2. Pass the variable-to-check message from variable nodes to check nodes along the edges of Tanner graph.
3. At each check node generate the check-to-variable message using all incoming messages from the incident variable nodes.
4. Pass the check-to-variable message from check nodes to variable nodes along the edges of Tanner graph.
5. At each variable node update the estimate of the corresponding bit (using the incoming message and intrinsic information) and compute the outgoing variable to check message.
6. Repeat steps 2-5 until, either a valid codeword is reached or a fixed number of iterations have been performed.

It's easier to perform arithmetic on hardware when messages are in log domain. All the messages are in fact log likelihood ratios (LLR). In our work all the messages have 4-bit precision.

5. DECODER ARCHITECTURE

The basic architecture of this design is shown in Fig.2. (In the figure **B** represents bit node, and **C** represents check node).

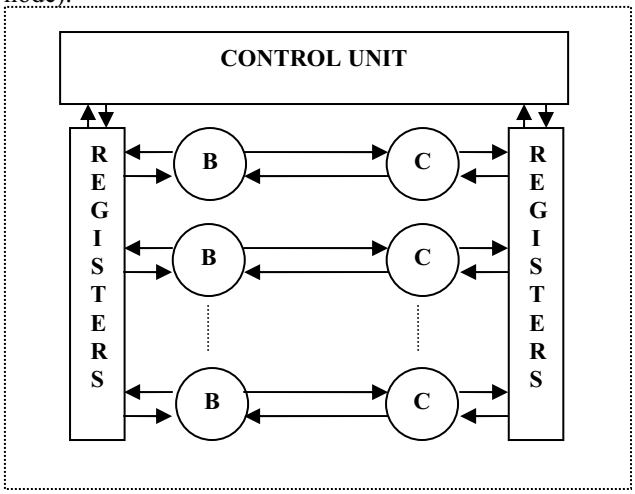

Figure 2. Basic Architecture of the decoder

The implemented decoder consists of three basic units shown in Figs. 3 to 5. We call them the Bit node (Variable node), Check node, and the Temporary check node units. From here on, these blocks will be referred to as BNU, CNU and CTU, respectively.

Our design relies on the structure of the H matrix. Since the H matrix can be generated using permutations (circular shifts) of the identity matrix, we can use circular shifts in the design to implement the changing pattern of edges from the bit nodes to the check nodes. This scheme, as would become clearer later, helps avoid the routing problems associated with LDPC decoder design. Our design, instead of handling all the edges of a bit / check node in a single clock cycle, processes each edge serially. The scalable design presented in [6] relies on processing each edge of the M parallel set of check nodes in serial,

and each edge of M parallel bit nodes in parallel. One of the differences in our design is that it processes each edge of the parallel bit nodes in a serial instead of a parallel fashion.

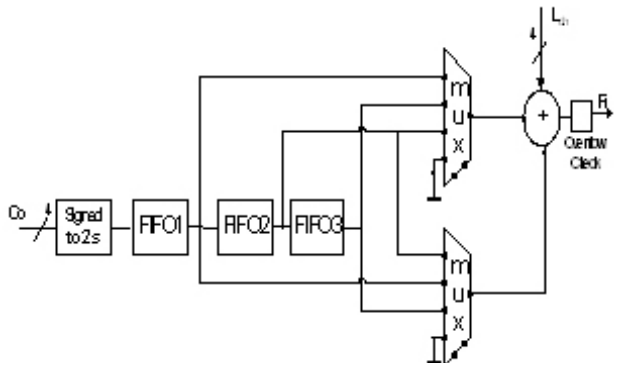

Figure 3. A single bit node-processing unit (BNU).

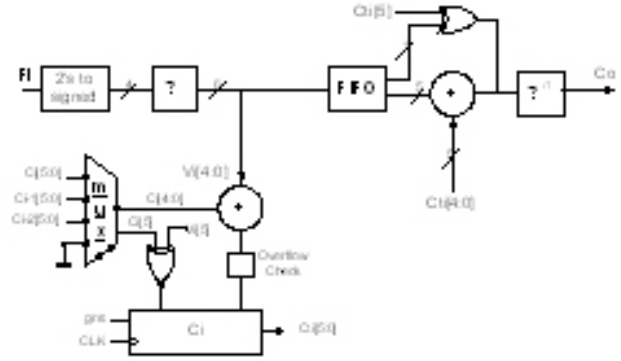

Figure 4. A single check node-processing unit (CNU).

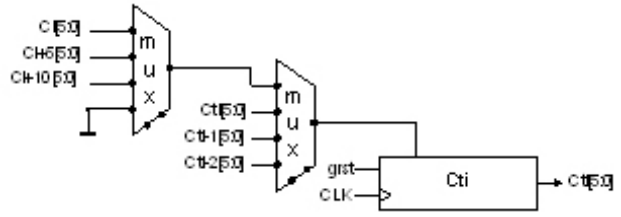

Figure 5. A single temporary check node-processing unit (CTU)

In its present state the design implements a (3,6) rate half code with p=173. It can be easily seen that the dimension of the **H** matrix is N = 173 * 6 = 1038 and K = 173 * 3 = 519. Without loss of generality, we present the design with the parameters given above. The design can be easily changed to accommodate a different set of parameters.

Next we discuss the data flow through the architecture. Each one of the FIFOs is 6 locations deep. FIFO1, FIFO2 and FIFO3 store the edge information for the 1st, 2nd and 3rd edges of bit node respectively. The output of the ψ^{-1} lookup table represents the message from the check node to the bit node. Looking at the **H** matrix, the processing flow is indicated by Fig. 6. The matrix is processed row by row. For the first row (corresponding to 1st through K/3th check node, connected to the first edges of all bit nodes) all the LLR values are summed and stored in the CNU. Note that the sub matrices corresponding to this first K/3 group of check nodes are all identity matrices. Hence for this case, we have no shifts. Once the sum is computed, it is copied to the CTU, and the messages from the check nodes along each one of the six edges to the corresponding bit nodes are calculated and stored. This final result is the check to bit message along the 1st edge of the bit nodes. Similarly 2nd and 3rd row is processed.

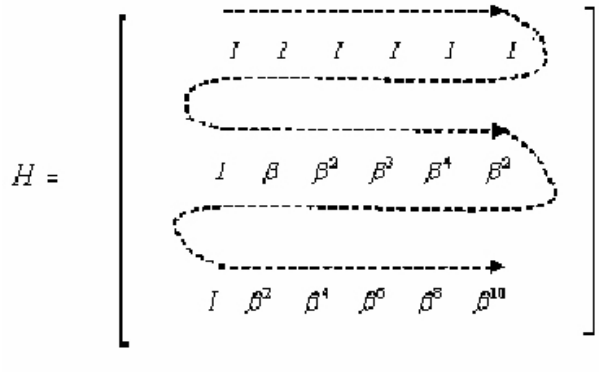

Figure 6. Process flow through the H matrix.

The number of clock cycles required for the completion of a single iteration is calculated as below.

Processing of each p rows of H =	6
Copying CNU to CTU =	1
Total per row (t) =	7
Number of rows (Nr) =	3
Total clock cycles (k) = Nr * t =	21

The clock cycle associated with the copying of CNU to CTU can be avoided at the cost of a longer critical path. If the critical path is not a concern, the total clock cycles for single iteration can be reduced to 18.

6. RESULTS

We coded all our modules using Verilog HDL. We then verified RTL model of our decoder using HDL simulators. Also we obtained BER performance through this RTL model and it corresponded very closely to the C simulations. We were able to verify our FPGA implementation using the parallel port interface with a PC.

The array code we chose for implementation had

p = 173, row weight = 6, and column weight = 3. Thus, ours was a (1038, 519) regular (3, 6) code. Following is a table that gives the amount of resources used on Xilinx VirtexE XCV2000E FPGA.

	Total Available	Used	% Usage
Slices	19,200	10,883	56
Slice Flip Flops	3,598	38,400	9
4-input LUTs	19,419	38,400	50
BRAMs	120	160	79

With the introduction of a pipeline stage at the beginning of the BNU, the maximum clock frequency allowable after the synthesis-timing estimate turned out to be 43 MHz. However, this dropped to 26.3 MHz (38 ns) after a complete placement and routing of the design. The logic delay in this critical path was only 19%, while routing took up rest of the 81%. These numbers indicate the extent to which routing can become a problem in an FPGA design environment even with a relatively regular design as ours. Nevertheless even at a clock speed of 26.3 MHz, we achieved a throughput of 72 Mbps for a total of 18 iterations. This result is a modest estimate, since a higher throughput can be obtained with some additional optimizations in the FPGA implementation. The design in [4] was able to achieve a throughput of 54 Mbps at 54MHz. Our design is at least 130% faster in terms of the throughput, and though we did not measure the power consumed by the decoder, we estimate that our design would consume 50% less power, since we run the clock at 26 MHz.

7. REFERENCES

[1] D. J. C. Mackay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *IEE Electronics Letters*, vol.33, no.6, pp.457-458, Mar. 1997.

[2] E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam, "VLSI architectures for iterative decoders in magnetic recording channels," *IEEE Trans. Magnetics*, vol.37, no.2, pp. 748-755, Mar. 2001.

[3] 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, 2002, pp. 404-412.

[4] T. Zhang and K. K. Parhi, "An FPGA Implementation of (*3,6*)-Regular Low-Density Parity-Check Code Decoder", *EURASIP Journal on Applied Signal Processing, special issue on Rapid Prototyping of DSP Systems,* May 2003 vol. 2003, no. 6, pp. 530-542.

[5] E. Eleftheriou and S. Olcer, "Low density parity-check codes for digital subscriber lines", Proc. ICC'2002, New York, pp.1752-1757(2002).

[6] A. Selvarathinam, G. Choi, A. Prabhakar, K. Narayanan, and E. Kim, "A massively scaleable decoder architecture for low-density parity-check codes", Proceedings of International Symposium on Circuits and Systems, May 2003, vol.2, pp.61-64