HIGH-THROUGHPUT FPGA QC-LDPC DECODER ARCHITECTURE FOR 5G WIRELESS

BY SWAPNIL MHASKE

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Predrag Spasojevic

and approved by

_____

_____

_____

New Brunswick, New Jersey

October, 2015

ABSTRACT OF THE THESIS

High-Throughput FPGA QC-LDPC Decoder Architecture for 5G Wireless

by Swapnil Mhaske

Thesis Director: Professor Predrag Spasojevic

Wireless data traffic is expected to increase by a 1000 fold by the year 2020 with more than 50 billion devices connected to these wireless networks with peak data rates upto 10 Gb/s . The next generation of wireless cellular technology (being collectively termed as 5G) is slated to operate in the mm-wave (30-300GHz) spectrum which comes with challenges such as, reliance on line of sight (LOS) communication, short range of communication, increased shadowing and, rapid fading in time. This will necessitate additional signal processing techniques such as large antenna arrays and beamsteering which will further reduce the processing budget available to the channel coding system.

In an effort to design and develop a channel coding solution suitable to such systems, in this thesis we propose strategies to achieve a high-throughput FPGA-based decoder architecture for a QC-LDPC code based on circulant-1 identity matrix construction. We present a novel representation of the parity-check matrix (PCM) providing a multi-fold throughput gain. Splitting of the node processing algorithm enables us to achieve pipelining of blocks and hence layers. By partitioning the PCM into not only layers but superlayers, we derive an upper bound on the pipelining depth with respect to the size of the superlayer for the compact representation. To validate the architecture, a decoder for the *IEEE 802.11n (2012)* QC-LDPC is implemented on the *Xilinx Kintex-7* FPGA

with the help of the *FPGA IP* compiler available in the *NI LabVIEW^{TM} Communication System Design Suite (CSDS^{TM})*. It offers an automated and systematic compilation flow. An optimized hardware implementation from the decoder algorithm was generated in approximately 3 minutes, achieving an overall throughput of 608Mb/s (at 260MHz). With little or no modifications, the proposed decoder architecture caters to a wide range of circulant-1 identity matrix construction based QC-LDPC codes widely accepted in several communication and data storage standards.

Acknowledgements

I would like to thank my advisor Prof. Predrag Spasojevic for having faith in me and guiding me throughout the course of this research work, the importance of his advice cannot be overstated. Special thanks to Prof. Roy Yates and Prof. Zoran Gajic for being a part of my thesis committee.

I would also like to thank my colleagues Hojin Kee and Tai Ly in the LabVIEW FPGA R&D, National Instruments, Austin for their valuable feedback and guidance. I am especially grateful to Ahsan Aziz and the Advanced Wireless Research team in National Instruments, Austin for their support. I am thankful to Christopher Mueller-Smith for his help and contribution toward developing the BCJR decoding algorithm for the 3GPP turbo decoder during the very initial phase of this project. Many thanks go out to the Department of Electrical & Computer Engineering, Rutgers University for their continual support for this research work.

Finally, I thank my parents and my dear friends Chaitral and Manasa, without their love and encouragement this would not have been possible.

Dedication

To Aai, Baba, Tai, Ajay & Adu

Table of Contents

Abstract ii

Acknowledgements iv

Dedication v

List of Tables viii

List of Figures ix

1. Introduction 1

2. Quasi-Cyclic LDPC Codes and Decoding 3

2.1. Quasi-Cyclic LDPC Codes 3

2.2. Scaled Min-Sum Algorithm (MSA) for Decoding QC-LDPC Codes . . . 6

3. Strategies to Achieve High-throughput 9

3.1. Linear Complexity Node Processing 9

3.2. z-fold Parallelization of NPUs 12

3.3. Layered Decoding 12

3.4. Compact Representation of $\mathbf{H}_b$ 15

4. Layer-Pipelined Decoder Architecture 18

4.1. Pipelining GNPU and LNPU Arrays 19

5. Case Study 26

6. Application

2.48Gb/s QC-LDPC Decoder on the NI USRP-2953R 29

6.1. Multi-core Decoder 29

6.2. Results 33

7. Related Work

3GPP UMTS Turbo Decoder 35

8. Conclusion 39

References 40

List of Tables

2.1. Base matrix $\mathbf{H}_b$ for $z = 81$ specified in IEEE 802.11n (2012) standard used in the case study. $L_1 - L_{12}$ are the layers and $B_1 - B_{24}$ are the block columns (see Section 3.3). Valid blocks (see section 3.4) are highlighted. 5

3.1. Arbitrary submatrix $\mathbf{I}_s$ in $\mathbf{H}$, $0 \leq J \leq n_b - 1$, illustrating the opportunity to parallelize z NPUs. 11

3.2. Illustration of Message Passing in row-layered decoding in a Section of the PCM $\mathbf{H}_b$. 14

3.3. Block index matrix $\boldsymbol{\beta_I}$ showing the valid blocks (highlighted) to be processed. 15

3.4. Block shift matrix $\boldsymbol{\beta_S}$ showing the right-shift values for the valid blocks to be processed. 17

4.1. Rearranged Block Index Matrix $\boldsymbol{\beta}'_I$ used for our work, showing the valid blocks (highlighted) to be processed. 19

5.1. LDPC Decoder IP FPGA Resource Utilization & Throughput on the Xilinx *Kintex-7 FPGA*. 27

6.1. Performance and resource utilization comparison for the Baseline architecture with the Pipelined architecture of the QC-LDPC decoder on the *NI USRP-2953R* containing the Xilinx *Kintex7 (410t)* FPGA. 33

6.2. Performance and resource utilization comparison for versions with varying number of cores of the QC-LDPC decoder implemented on the *NI USRP-2953R* containing the Xilinx *Kintex7 (410t)* FPGA. 34

7.1. Turbo Decoder IP FPGA Resource Utilization & Throughput on the Xilinx Kintex-7 XC7K325T-2L-FFG900. 37

List of Figures

2.1. A Tanner graph where VNs (representing the code bits) are shown as circles and CNs (representing the parity-check equations) are shown as squares. Each edge in the graph corresponds to a non-zero entry (1 for binary LDPC codes) in the PCM **H**. 4

2.2. An instance of extrinsic message computation at CN i and its transfer to VN j on the Tanner graph. It is important to note that the VTC message from VN j is not included in the computation (indicated by a dashed line) as CN i intends to send it to VN j itself. 7

3.1. For-loop view of processing complexity (a) without two pass computation (b) with two pass computation. 10

4.1. Block-level view of the pipeline timing diagram. (a) General case for a circulant-1 identity submatrix construction based QC-LDPC code (see Section 2.1) without pipelining. (b) Special case of the IEEE 802.11n QC-LDPC code used in this work without pipelining (c) Pipelined processing of two layers for the general QC-LDPC code case in (a). (d) Pipelined processing of two layers for the IEEE802.11n QC-LDPC code case in (b). 22

4.2. Layer-level view of the pipeline timing diagram. (a) General case for a circulant-1 identity submatrix construction based QC-LDPC code (see Section 2.1) without pipelining. (b) Special case of the IEEE 802.11n QC-LDPC code used in this work without pipelining (c) Pipelined processing of two layers for the general QC-LDPC code case in (a). (d) Pipelined processing of two layers for the IEEE802.11n QC-LDPC code case in (b). 23

4.3. High-level decoder architecture. showing the z-fold parallelization of the NPUs with an emphasis on the splitting of the sign and the minimum computation given in equation (2.3). Note that, other computations in equations (2.1)-(2.4) are not shown for simplicity here. For both the pipelined and the non-pipelined versions, processing schedule for the inner Block Processing loop is as per Fig. 4.1 and that for the outer Layer Processing loop is as per Fig. 4.2. 25

5.1. Bit Error Rate (BER) performance comparison between uncoded BPSK (rightmost), rate=1/2 LDPC with 4 iterations using fixed-point data representation (second from right), rate=1/2 LDPC with 8 iterations using fixed-point data representation (third from right), rate=1/2 LDPC with 8 iterations using floating-point data representation (leftmost). . . 28

6.1. Top-level VI describing the parallelization of the QC-LDPC decoder [1] on the *NI USRP-2953R* containing the Xilinx *Kintex7 (410t)* FPGA. . . 32

6.2. Bit Error Rate (BER) performance comparison between uncoded BPSK (green) and the 2.48Gb/s, rate=1/2, QC-LDPC decoder (red) on the *NI USRP-2953R* containing the Xilinx *Kintex7 (410t)* FPGA. 34

7.1. Turbo Decoder Iterative Log-MAP Decoder 36

7.2. BER performance of the aforementioned turbo decoder (curve on the left in black) versus uncoded BPSK (curve on the right in red) 37

Chapter 1

Introduction

For the next generation of wireless technology collectively termed as Beyond-4G and 5G (hereafter referred to as 5G), peak data rates of upto ten Gb/s with overall latency less than 1ms [2] are envisioned. However, due to the proposed operation in the 30-300GHz range with challenges such as short range of communication, increasing shadowing and rapid fading in time, the processing complexity of the system is expected to be high. In an effort to design and develop a channel coding solution suitable to such systems, in this report, we present a high-throughput, scalable and reconfigurable FPGA decoder architecture for circulant-1 identity matrix construction based QC-LDPC codes.

It is well known that the structure offered by QC-LDPC codes [3] makes them amenable to time and space efficient decoder implementations relative to random LDPC codes. We believe that, given the primary requirements of high decoding throughput, QC-LDPC codes or their variants (such as accumulator-based codes [4]) that can be decoded using belief propagation (BP) methods are highly likely candidates for 5G systems. Thus, for the sole purpose of validating the proposed architecture, we chose a standard compliant code, with a throughput performance that well surpasses the requirement of the chosen standard. The proposed decoder architecture can be used for a wide range of circulant-1 identity construction based QC-LDPC codes many of which have been accepted in several standards such as IEEE 802.11n/ac [5], IEEE 802.16e/m [6] and DVB [7].

Insightful work on high-throughput (order of Gb/s) BP-based QC-LDPC decoders is available, however, most of such works focus on an ASIC design [8], [9] which usually requires intricate customizations at the Register Transfer Level (RTL) level and expert knowledge of VLSI design. A sizeable subset of which caters to fully-parallel [10] or

code-specific [11] architectures. From the point of view of an evolving research solution this is not an attractive option, especially for rapid-prototyping. In the relatively less explored area of FPGA-based implementation, impressive results have recently been presented in works such as [12],[13] and [14]. However, these are based on fully-parallel architectures which lack flexibility (code specific) and are limited to small block sizes (primarily due to the inhibiting routing congestion) as discussed in the informative overview in [15]. Since our case study is based on fully-automated generation of the Hardware Description Language (HDL), a fair comparison is done with another state-of-the-art implementation [16] in Chapter 5. Moreover, in this report, we provide without loss of generality, strategies to achieve a high-throughput FPGA-based architecture for a QC-LDPC code based on a circulant-1 identity matrix construction.

The main contribution of this work is a compact representation (matrix form) of the PCM of the QC-LDPC code which provides a multi-fold increase in throughput. In spite of the resulting reduction in the *degrees of freedom* for pipelined processing, we achieve efficient pipelining of two-layers and also provide without loss of generality an upper bound on the pipelining depth that can be achieved in this manner. The splitting of the node processing allows us to achieve the said degree of pipelining without utilizing additional hardware resources. The algorithmic strategies were realized in hardware for our case study by the *FPGA IP* [17] compiler in *LabVIEW*^{TM} *CSDS*^{TM} which translated the entire software-pipelined high-level language description into VHSIC Hardware Description Language (VHDL) in approximately 3 minutes, enabling state-of-the-art rapid-prototyping. The scalability of the proposed architecture has been demonstrated in an application that achieves a throughput of 2.48Gb/s [18] on the NI USRP-2953R.

The remainder of this report is organized as follows. Chapter 2 describes the QC-LDPC codes and the decoding algorithm chosen for this implementation. The strategies for achieving high throughput are explained in Chapter 3. The details of the layered decoding technique applied in this work are given in Chapter 4. The case study for the IEEE 802.11n (2012) standard is discussed in Chapter 5, developments that contributed towards this work are presented in Chapter 7 and we conclude with Chapter 8.

Chapter 2

Quasi-Cyclic LDPC Codes and Decoding

LDPC codes (due to R. Gallager [19]) are a class of linear block codes that have been shown to achieve near-capacity performance on a broad range of channels and are characterized by a low-density (sparse) PCM representation.

Mathematically, an LDPC code is a null-space of its $m \times n$ PCM $\mathbf{H}$, where m denotes the number of parity-check equations or parity-bits and n denotes the number of variable nodes or code bits [3]. In other words, for a rank m PCM $\mathbf{H}$, m is the number of redundant bits added to the k information bits, which together form the codeword of length $n = k + m$. In the Tanner graph representation (due to Tanner [20]), $\mathbf{H}$ is the incidence matrix of a bipartite graph comprising of the check node (CN) set of m parity-check equations and the variable node (VN) set of n variable or bit nodes; the i^{th} CN is connected to the j^{th} VN if $\mathbf{H}(i,j) = 1$, $1 \leq i \leq m$ and $1 \leq j \leq n$. A toy example of a Tanner graph is shown in Fig. 2.1. The set of VNs that are connected with an edge to CN i (hereafter referred to as $\mathcal{N}(i)$) is called as the neighbor set of CN i. Similarly, $\mathcal{N}(j)$ is the neighbor set of VN j. The degree d_{c_i} (d_{v_j}) of a CN i (VN j) is equal to the number of 1s along the i^{th} row (j^{th} column) of $\mathbf{H}$. For constants $c_c, c_v \in \mathbb{Z}_{>0}$ and $c_c << m, c_v << n$, if $\forall i, j$, $d_{c_i} = c_c$ and $d_{v_j} = c_v$, then the LDPC code is called as a regular code and is called an irregular code otherwise.

2.1 Quasi-Cyclic LDPC Codes

The first LDPC codes by Gallager are random i.e. the neighbors of an arbitrary CN or VN are randomly chosen, subject to constraints specified in the code construction algorithm. One such constraint is the *girth* of the Tanner graph underlying the LDPC code structure. Here girth refers to the length of the smallest cycle in the Tanner graph.

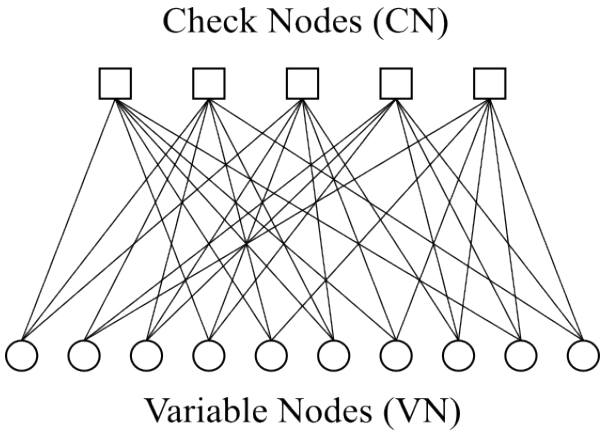

Figure 2.1: A Tanner graph where VNs (representing the code bits) are shown as circles and CNs (representing the parity-check equations) are shown as squares. Each edge in the graph corresponds to a non-zero entry (1 for binary LDPC codes) in the PCM $\mathbf{H}$.

Random code structures complicate the decoder implementation, mainly because a random interconnect pattern between the VNs and CNs directly translates to a complex wire routing circuit on hardware. QC-LDPC codes belong to the class of structured codes that are relatively easier to implement without significantly compromising performance.

The construction of identity matrix based QC-LDPC codes relies on an $m_b \times n_b$ matrix $\mathbf{H}_b$ sometimes called as the *base matrix* which comprises of cyclically right-shifted identity and zero submatrices both of size $z \times z$ where, $z \in \mathbb{Z}^+, 1 \leq i_b \leq m_b$ and $1 \leq j_b \leq n_b$, the shift value,

$$s = \mathbf{H}_b(i_b, j_b) \in \mathcal{S} = \{-1\} \cup \{0, \dots z-1\}$$

The PCM matrix $\mathbf{H}$ is obtained by *expanding* $\mathbf{H}_b$ using the mapping,

$$s \longrightarrow \begin{cases} \mathbf{I}_s, & s \in \mathcal{S} \backslash \{-1\} \\ \mathbf{0}, & s \in \{-1\} \end{cases}$$

where, $\mathbf{I}_s$ is an identity matrix of size z which is cyclically right-shifted by $s = \mathbf{H}_b(i_b, j_b)$ and $\mathbf{0}$ is the all-zero matrix of size $z \times z$. As $\mathbf{H}$ is composed of the submatrices $\mathbf{I}_s$ and $\mathbf{0}$, it has $m = m_b.z$ rows and $n = n_b.z$ columns. $\mathbf{H}$ for the *IEEE 802.11n (2012)* standard [5] (used for our case study) with $z = 81$ is shown in Table 2.1.

Layers ↓	**Blocks ⟶**																							
	B_1	**B_2**	**B_3**	**B_4**	**B_5**	**B_6**	**B_7**	**B_8**	**B_9**	**B_{10}**	**B_{11}**	**B_{12}**	**B_{13}**	**B_{14}**	**B_{15}**	**B_{16}**	**B_{17}**	**B_{18}**	**B_{19}**	**B_{20}**	**B_{21}**	**B_{22}**	**B_{23}**	**B_{24}**
L_1	57	-1	-1	-1	50	-1	11	-1	50	-1	79	-1	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
L_2	3	-1	28	-1	0	-1	-1	-1	55	7	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
L_3	30	-1	-1	-1	24	37	-1	-1	56	14	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1
L_4	62	53	-1	-1	53	-1	-1	3	35	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1
L_5	40	-1	-1	20	66	-1	-1	22	28	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1
L_6	0	-1	-1	-1	8	-1	42	-1	50	-1	-1	8	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1
L_7	69	79	79	-1	-1	-1	56	-1	52	-1	-1	-1	0	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1
L_8	65	-1	-1	-1	38	57	-1	-1	72	-1	27	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1
L_9	64	-1	-1	-1	14	52	-1	-1	30	-1	-1	32	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1
L_{10}	-1	45	-1	70	0	-1	-1	-1	77	9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1
L_{11}	2	56	-1	57	35	-1	-1	-1	-1	-1	12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0
L_{12}	24	-1	61	-1	60	-1	-1	27	51	-1	-1	16	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0

Table 2.1: Base matrix $\mathbf{H}_b$ for $z = 81$ specified in IEEE 802.11n (2012) standard used in the case study. $L_1 - L_{12}$ are the layers and $B_1 - B_{24}$ are the block columns (see Section 3.3). Valid blocks (see section 3.4) are highlighted.

2.2 Scaled Min-Sum Algorithm (MSA) for Decoding QC-LDPC Codes

Owing to the sparsity of the PCM of an LDPC code and the computational power available today, it is practicable to decode LDPC codes using iterative message passing (MP) or belief propagation (BP) [19, 21] (name picked up from Bayesian-inference literature) on the bipartite Tanner graph. Gallager's method of BP decoding - called as the Sum-Product Algorithm (SPA) [19] is a general algorithm that provides near-optimal performance for a wide range of channels. The MSA is a reduced complexity version of the SPA which has a small loss in performance.

As the name suggests, in MP decoding, the CNs and VNs communicate with each other, successively passing revised estimates or *messages* of the a posteriori probability (APP) log-likelihood ratios (LLR) of the associated VNs or code bits, in every decoding iteration. In this work we have employed the efficient decoding algorithm presented in [22], with pipelined processing of layers based on the row-layered decoding technique [23], detailed in Section 3.3.

Definition 1. *For $1 \leq i \leq m$ and $1 \leq j \leq n$, let v_j denote the j^{th} bit in the length n codeword and $y_j = v_j + n_j$ denote the corresponding received value from the channel corrupted by the noise sample n_j. Let the variable-to-check (VTC) message from VN j to CN i be q_{ij} and, let the check-to-variable (CTV) message from CN i to VN j be r_{ij}. Let the a posteriori probability (APP) ratio for VN j be denoted as p_j.*

The steps of the scaled-MSA [24] are given below.

1. Initialize the APP ratio and the CTV messages as,

$$p_j^{(0)} = ln\left\{\frac{P(v_j = 0|y_j)}{P(v_j = 1|y_j)}\right\}, \quad 1 \leq j \leq n \tag{2.1}$$

$$r_{ij}^{(0)} = 0, \quad 1 \leq i \leq m, 1 \leq j \leq n$$

2. Iteratively compute at the t^{th} decoding iteration,

$$q_{ij}^{(t)} = p_j^{(t-1)} - r_{ij}^{(t-1)} \tag{2.2}$$

$$r_{ij}^{(t)} = a \cdot \prod_{k \in \mathcal{N}(i)\backslash\{j\}} sign\left(q_{ik}^{(t)}\right) \cdot \min_{k \in \mathcal{N}(i)\backslash\{j\}} \left\{|q_{ik}^{(t)}|\right\} \tag{2.3}$$

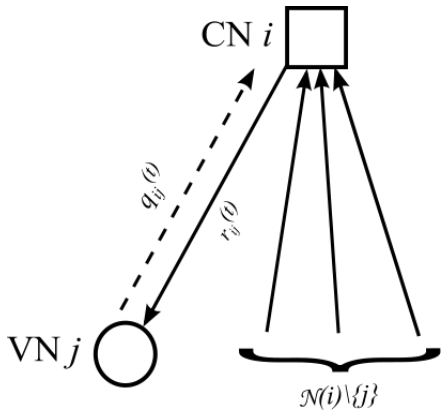

Figure 2.2: An instance of extrinsic message computation at CN i and its transfer to VN j on the Tanner graph. It is important to note that the VTC message from VN j is not included in the computation (indicated by a dashed line) as CN i intends to send it to VN j itself.

$$p_j^{(t)} = q_{ij}^{(t)} + r_{ij}^{(t)} \tag{2.4}$$

where, $1 \leq i \leq m$, and $k \in \mathcal{N}(i)\backslash\{j\}$ represents the set of the VN neighbors of CN i excluding VN j. An instance of this message exchange is shown in Fig. 2.2. Let t_{max} be the maximum number of decoding iterations.

3. Decision on the code bit v_j, $1 \leq j \leq n$ as,

$$\hat{v}_j = \begin{cases} 1, & p_j \leq 0 \\ 0, & otherwise \end{cases} \tag{2.5}$$

4. If $\hat{\mathbf{v}}\mathbf{H}^T = 0$, where $\hat{\mathbf{v}} = (\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n)$, or $t = t_{max}$, declare $\hat{\mathbf{v}}$ as the decoded codeword.

It is well known that since the MSA is an approximation of the SPA, the performance of the MSA is relatively worse than the SPA [3]. However, in [24] it has been shown that scaling the CTV messages r_{ij} can improve the performance of the MSA. Hence, we scale the CTV messages by a factor a (=0.75).

Remark 1. The standard MP algorithm is based on the so-called *flooding* or *two-phase* schedule where, each decoding iteration comprises of two phases. In the first phase, VTC messages for all the VNs are computed and in the second phase the CTV messages for all the CNs are computed, strictly in that order. Thus, message updates from one

side of the graph propagate to the other side only in the next decoding iteration. In the algorithm that we have used [22] however, message updates can propagate across the graph in the same decoding iteration resulting in the following advantages:

1. *Single processing unit*: Since the VTC messages q_{ij} given by equation (2.2) can be computed on the fly (i.e. q_{ij} updates are immediately consumed by the CTV r_{ij} computation) in the same decoding iteration, there is no need to have a separate VN processing unit, unlike that in the standard MP algorithm. The single node processing unit (NPU) computes both the CN and the VN messages.

2. *Reduced memory storage*: The on the fly computation of the VTC messages q_{ij} also implies that there is no need to store the VN messages separately.

3. *Fast convergence*: It has been shown in [22] that the algorithm we have employed in our work converges faster than the standard MP flooding schedule. This means that comparable performance can be achieved by the algorithm described in this Section with fewer decoding iterations, thus helping the high-throughput implementation of the decoder.

Chapter 3

Strategies to Achieve High-throughput

To understand the high-throughput requirements for LDPC decoding, let us first define the decoding throughput T of the hardware realization of an LDPC decoder based on an iterative MP decoding algorithm.

Definition 2. *Let F_c be the clock frequency, n be the code length, N_i be the number of decoding iterations and N_c be the number of clock cycles per decoding iteration, then the throughput of the decoder is broadly given by, $T = \frac{F_c \cdot n}{N_i \cdot N_c}$ b/s.*

Even though, n and N_i are functions of the code and the decoding algorithm used, F_c and N_c are determined by the hardware architecture. Architectural optimization such as the ability to operate the decoder at higher clock rates with minimal latency between decoding iterations can help achieve higher throughput. We have employed the following techniques to increase the throughput given by Definition 2.

3.1 Linear Complexity Node Processing

As noted in Section 2.2, separate processing units for CNs and VNs are not required unlike that for the flooding schedule. The hardware elements that process equations (2.2)-(2.4) are collectively referred to as the Node Processing Unit (NPU).

Careful observation reveals that, among equations (2.2)-(2.4), processing the CTV messages r_{ij}, $1 \leq i \leq m$ and $1 \leq j \leq n$ is the most computationally intensive due to the calculation of the sign, and the minimum value operations. Note that, computing the CTV message r_{ij} from CN i to VN j (in equation 2.3) involves processing the metrics for VNs in the set $\mathcal{N}(i)\backslash\{j\}$, not $\mathcal{N}(i)$. Thus, the complexity of processing these values is $\mathcal{O}(d_{c_i}^2)$. A naive implementation of the minimum process in (2.3) could be an

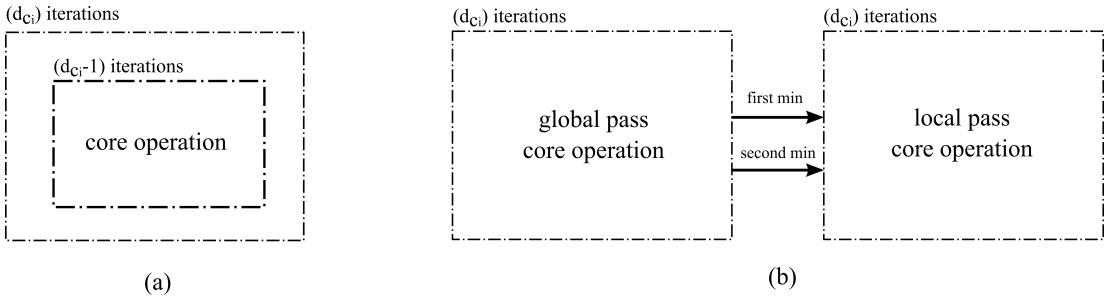

Figure 3.1: For-loop view of processing complexity (a) without two pass computation (b) with two pass computation.

algorithm that traverses over all d_{c_i} branches of CN i and compares two values at a time translating into d_{c_i} comparator-based *min-trees* with $(d_{c_i} - 1)$ leaf nodes each. In software, this translates to two nested for-loops, an outer loop that executes d_{c_i} times and an inner loop that executes $(d_{c_i} - 1)$ times. This is shown in Fig. 3.1 (a).

To achieve linear complexity $\mathcal{O}(d_{c_i})$ for the minimum value computation, we split the process into two phases or passes: the *global* pass where the first and the second minimum (the smallest value in the set excluding the minimum value of the set) for all the neighboring VNs of a CN are computed and the *local* pass where the first and second minimum from the global pass are used to compute the minimum value for each neighboring VN. Based on the functionality of the two passes, the NPU is divided into the Global NPU (GNPU) and the Local NPU (LNPU). The detailed steps of the algorithm are given below.

1. *Global Pass*:

   i. *Initialization*: Let ℓ denote the discrete time-steps such that, $\ell \in \{0\} \cup \{1, 2, \ldots, |\mathcal{N}(i)|\}$ and let $f^{(\ell)}$ and $s^{(\ell)}$ denote the value of the first and the second minimum at time ℓ respectively. The initial value at time $\ell = 0$ is,

   $$f^{(0)} = s^{(0)} = \infty. \tag{3.1}$$

   ii. *Comparison*: Let $k_i(\ell) \in \mathcal{N}(i)$, $\ell = \{1, 2, \ldots, |\mathcal{N}(i)|\}$, denote the index of the ℓ^{th} neighboring VN of CN i. Note that, $k_i(\ell)$ depends on i and ℓ, specifically, for a given CN i it is a bijective function of ℓ. An increment from $(\ell - 1)$ to

	$\mathbf{VN_{zJ}}$	...		$\mathbf{VN_{zJ+l}}$		...	$\mathbf{VN_{z(J+1)-1}}$
$\mathbf{NPU_0}$	0	...	0	1	0	...	0
$\mathbf{NPU_1}$	0	...	0	0	1	...	0
$\vdots$	$\vdots$						$\vdots$
$\mathbf{NPU_{z-2}}$	0	...	0	0	0	...	0
$\mathbf{NPU_{z-1}}$	0	...	1	0	0	...	0

Table 3.1: Arbitrary submatrix $\mathbf{I}_s$ in $\mathbf{H}$, $0 \leq J \leq n_b - 1$, illustrating the opportunity to parallelize z NPUs.

ℓ corresponds to moving from the edge CN $i \leftrightarrow$ VN $k_i(\ell-1)$ to the edge CN $i \leftrightarrow$ VN $k_i(\ell)$.

$$f^{(\ell)} = \begin{cases} |q_{ik_i(\ell)}|, & |q_{ik_i(\ell)}| \leq f^{(\ell-1)} \\ f^{(\ell-1)}, & otherwise. \end{cases} \tag{3.2}$$

$$s^{(\ell)} = \begin{cases} |q_{ik_i(\ell)}|, & f^{(\ell-1)} < |q_{ik_i(\ell)}| < s^{(\ell-1)} \\ f^{(\ell-1)}, & |q_{ik_i(\ell)}| \leq f^{(\ell-1)} \\ s^{(\ell-1)}, & otherwise. \end{cases} \tag{3.3}$$

Thus, $f^{(\ell_{max})}$ and $s^{(\ell_{max})}$ are the first and second minimum values for the set of VN neighbors of CN i, where, $\ell_{max} = |\mathcal{N}(i)|$.

2. *Local Pass*: Let the minimum value as per equation (2.3) for VN $k_i(\ell)$ be denoted as $q^{min}_{ik_i(\ell)}$, $\ell \in \{1, 2, \ldots, |\mathcal{N}(i)|\}$ then,

$$q^{min}_{ik_i(\ell)} = \begin{cases} f^{(\ell_{max})}, & |q_{ik_i(\ell)}| \neq f^{(\ell_{max})} \\ s^{(\ell_{max})}, & otherwise. \end{cases} \tag{3.4}$$

In software, this translates to two consecutive for-loops, each executing $(d_{c_i} - 1)$ times. This is shown in Fig. 3.1 (b). Consequently, this reduces the complexity from $\mathcal{O}(d^2_{c_i})$ to $\mathcal{O}(d_{c_i})$. A similar approach is also found in [25], [8]. The sign computation is processed in a similar manner.

3.2 z-fold Parallelization of NPUs

The CN message computation given by equation (2.3) is repeated m times in a decoding iteration i.e. once for each CN. A straightforward serial implementation of this kind is slow and undesirable. Instead, we apply a strategy based on the following understanding.

Fact 1. *An arbitrary submatrix* $\mathbf{I}_s$ *in the PCM* $\mathbf{H}$ *corresponds to* z *CNs connected to* z *VNs on the bipartite graph, with strictly* 1 *edge between each CN and VN.*

This implies that no CN in this set of z CNs given by $\mathbf{I}_s$ shares a VN with another CN in the same set. Table 3.1 illustrates such an arbitrary submatrix in $\mathbf{H}$. This presents us with an opportunity to operate z NPUs in parallel (hereafter referred to as an *NPU array*), resulting in a z-fold increase in throughput.

3.3 Layered Decoding

In Section 3.2 we saw that z CNs (and hence VNs) computations can be done in parallel by virtue of the circulant-1 identity matrix based structure of the CN-VN connections. A straightforward way of improving throughput by multiple folds is to process more than z CTV (and hence VTC) message updates i.e. to extend the parallelism beyond the boundary of a submatrix of size z. A greedy approach to maximize throughput in this manner is to process all the m CTV updates at once. This is the classical *fully parallel* implementation which in principle is a manifestation of the flooding schedule (see Remark 1) where *all* nodes on one side of the bipartite graph can be processed in parallel. Although, such a *fully parallel* implementation may seem as an attractive option for achieving high-throughput performance, it has the following drawbacks.

1. Firstly, it becomes quickly intractable in hardware due to the complex interconnect pattern.

2. Secondly, such an implementation caters to the graph of a particular code in a code ensemble. In other words, the routing interconnect determined by the code is frozen in hardware.

Nevertheless, there is still scope to achieve a degree of parallelism greater than z without sacrificing the flexibility of the implementation. This can be accomplished if the following condition is satisfied by the processing schedule.

Fact 2. *From the perspective of CN processing, two or more CNs can be processed at the same time (i.e. they are independent of each other) if they do not have one or more VNs (code bits) in common.*

This is central to the row-layering [23] technique used in this work. To understand this in detail let us define the following.

In terms of $\mathbf{H}$, an arbitrary subset of rows can be processed at the same time provided that, no two or more rows have a 1 in the same column of $\mathbf{H}$. This subset of rows is termed as a *row-layer* (hereafter referred to as a *layer*). In other words, given a set $\mathcal{L} = \{L_1, L_2, \ldots, L_I\}$ of I layers in $\mathbf{H}$, $\forall u \in \{1, 2, \ldots, I\}$ and $\forall i, i' \in L_u$, then, $\mathcal{N}(i) \cap \mathcal{N}(i') = \phi$.

Observing that, $\sum_{u=1}^{I} |L_u| = m$, in general, L_u can be any subset of rows as long as the rows within each subset satisfy the condition in Fact 2; implying that, $|L_u| \neq |L_{u'}|$, $\forall u, u' \in \{1, 2, \ldots, I\}$ is possible. It is not hard to see that, this is more so true in the case of unstructured or random codes. In fact, if $|L_u| \neq |L_{u'}|$ the hardware complexity increases since the processing effort changes from one layer to another. For instance, the memory allocation for each layer could be different, resulting in dynamic memory-address generation patterns.

Owing to the structure of QC-LDPC codes, the choice of $|L_u|$ (and hence I) becomes much obvious. Submatrices $\mathbf{I}_s$ in $\mathbf{H}_b$ (with row and column weight of 1) guarantee that, for the z CNs corresponding to the rows of $\mathbf{I}_s$), always satisfy the condition in Fact 2. Thus, keeping in mind the ease of implementation, $|L_u| = |L_{u'}| = z$ is the most obvious choice.

From the VN or column perspective, $|L_u| = z$, $\forall u = \{1, 2, \ldots, I\}$ implies that, the columns of $\mathbf{H}$ are also divided into subsets of size z (hereafter referred to as *block columns*) given by the set $\mathcal{B} = \{B_1, B_2, \ldots, B_J\}$, $J = \frac{n}{z} = n_b$. Observing that VNs belonging to a block column may participate in CN equations across several layers,

Layers ↓		**Blocks ⟶**			
	...	$\mathbf{B_2}$	$\mathbf{B_3}$	$\mathbf{B_4}$	...
$\mathbf{L_1}$	...	↓	↓	↓	...
$\mathbf{L_2}$	...	↓	28	↓	...
$\mathbf{L_3}$	...	↓	↓	↓	...
$\mathbf{L_4}$	...	53	↓	↓	...
$\mathbf{L_5}$	...	↓	↓	20	...
$\mathbf{L_6}$	...	↓	↓	↓	...
$\mathbf{L_7}$	...	79	79	↓	...
$\mathbf{L_8}$	...	↓	↓	↓	...
$\mathbf{L_9}$	...	↓	↓	↓	...
$\mathbf{L_{10}}$	...	45	↓	70	...
$\mathbf{L_{11}}$	...	56	↓	57	...
$\mathbf{L_{12}}$	...	↓	61	↓	...
		to L_4	to L_2	to L_5	

Table 3.2: Illustration of Message Passing in row-layered decoding in a Section of the PCM $\mathbf{H}_b$.

we further divide the block columns into *blocks*, where a block is the intersection of a layer and a block column. One may visualize a block in the decoder architecture as a submatrix in $\mathbf{H}_b$, owing to the equivalence between the two. Two or more layers $L_u, L_{u'}$ are said to be *dependent* with respect to the block column B_w if, $\mathbf{H}_b(u, w) \neq -1$ and, $\mathbf{H}_b(u', w) \neq -1$ and are said to be *independent* otherwise.

For example, in Table 3.2 we can see that layers L_4, L_7, L_{10} and L_{11} are dependent with respect to block column B_2. Assuming that the message update begins with layer L_1 and proceeds downward, the arrows represent the directional flow of message updates from one layer to another. Thus, layer L_7 cannot begin updating the VNs associated with block column B_2 before layer L_4 has finished updating messages for the same set of VNs and so on. It is worthwhile to reiterate that the idea of parallelizing z NPUs seen in Section 3.2 can be extended to layers, NPU arrays can process message updates for multiple independent layers. It is clear that, dependent layers limit the degree of parallelization available to achieve high-throughput. In Chapter 4, we discuss

Layers ↓	**Blocks ⟶**							
	$\mathbf{b_1}$	$\mathbf{b_2}$	$\mathbf{b_3}$	$\mathbf{b_4}$	$\mathbf{b_5}$	$\mathbf{b_6}$	$\mathbf{b_7}$	$\mathbf{b_8}$
$\mathbf{L_1}$	0	4	6	8	10	12	13	-1
$\mathbf{L_2}$	0	2	4	8	9	13	14	-1
$\mathbf{L_3}$	0	4	5	8	9	14	15	-1
$\mathbf{L_4}$	0	1	4	7	8	15	16	-1
$\mathbf{L_5}$	0	3	4	7	8	16	17	-1
$\mathbf{L_6}$	0	4	6	8	11	17	18	-1
$\mathbf{L_7}$	0	1	2	6	8	12	18	19
$\mathbf{L_8}$	0	4	5	8	10	19	20	-1
$\mathbf{L_9}$	0	4	5	8	11	20	21	-1
$\mathbf{L_{10}}$	1	3	4	8	9	21	22	-1
$\mathbf{L_{11}}$	0	1	3	4	10	22	23	-1
$\mathbf{L_{12}}$	0	2	4	7	8	11	12	23

Table 3.3: Block index matrix $\boldsymbol{\beta_I}$ showing the valid blocks (highlighted) to be processed.

pipelining methods that allow us to overcome layer-to-layer dependency and improve throughput.

3.4 Compact Representation of $\mathbf{H}_b$

Before we discuss the pipelined processing of layers, we present a novel compact (thus efficient) matrix representation leading to a significant improvement in throughput. To understand this, let us call **0** submatrices in **H** as *invalid* blocks, where there are no edges between the corresponding CNs and VNs, and the submatrices $\mathbf{I}_s$ as *valid* blocks. In a conventional approach to scheduling (for example in [9]), message computation is done for all the valid and invalid blocks. To avoid processing invalid blocks, we propose an alternate representation of $\mathbf{H}_b$ in the form of two matrices: $\boldsymbol{\beta}_I$ (Table 3.3), the block index matrix and $\boldsymbol{\beta}_S$ (Table 3.4), the block shift matrix. Matrices $\boldsymbol{\beta}_I$ and $\boldsymbol{\beta}_S$ hold the index locations and the shift values (and hence the connections between the CNs and VNs) corresponding to *only* the valid blocks in $\mathbf{H}_b$, respectively. Construction of $\boldsymbol{\beta}_I$ is based on the following definition,

Definition 3. *Construction of $\boldsymbol{\beta}_I$ is as follows.*

for $u = \{1, 2, \ldots, I\}$

  set $w = 0$, $j_b = 0$

  for $j_b = \{1, 2, \ldots, n_b\}$

    $j_b = j_b + 1$

    if $\mathbf{H}_b(u, j_b) \neq -1$

      $w = w + 1; \boldsymbol{\beta}_I(u, w) = j_b; \boldsymbol{\beta}_S(u, w) = \mathbf{H}_b(u, j_b)$.

To observe the benefit of this alternate representation, let us define the following ratio.

Definition 4. *Let λ denote the compaction ratio, which is the ratio of the number of columns of $\boldsymbol{\beta}_I$ (which is the same for $\boldsymbol{\beta}_S$) to the number of columns of $\mathbf{H}_b$. Hence, $\lambda = \frac{J}{n_b}$.*

The compaction ratio λ is a measure of the compaction achieved by the alternate representation of H_b. Compared to the conventional approach, scheduling as per the $\boldsymbol{\beta}_I$ and $\boldsymbol{\beta}_S$ matrices improves throughput by $\frac{1}{\lambda}$ times. In our case study, $\lambda = \frac{8}{24} = \frac{1}{3}$, thus providing a throughput gain of $\frac{1}{\lambda} = 3$.

Remark 2. In the irregular QC-LDPC code in our case study, all layers comprise of 7 blocks each, except layer L_7 and L_{12} which have 8. With the aim of minimizing hardware complexity by maintaining a static memory-address generation pattern (does not change from layer-to-layer), our implementation assumes regularity in the code. The decoder processes 8 blocks for each layer of the $\boldsymbol{\beta}_I$ matrix resulting in some throughput penalty. The results from processing the invalid blocks in L_7 and L_{12} are not stored in the memory.

Layers ↓	**Blocks ⟶**							
	$\mathbf{b_1}$	$\mathbf{b_2}$	$\mathbf{b_3}$	$\mathbf{b_4}$	$\mathbf{b_5}$	$\mathbf{b_6}$	$\mathbf{b_7}$	$\mathbf{b_8}$
$\mathbf{L_1}$	57	50	11	50	79	1	0	-1
$\mathbf{L_2}$	3	28	0	55	7	0	0	-1
$\mathbf{L_3}$	30	24	37	56	14	0	0	-1
$\mathbf{L_4}$	62	53	53	3	35	0	0	-1
$\mathbf{L_5}$	40	20	66	22	28	0	0	-1
$\mathbf{L_6}$	0	8	42	50	8	0	0	-1
$\mathbf{L_7}$	69	79	79	56	52	0	0	0
$\mathbf{L_8}$	65	38	57	72	27	0	0	-1
$\mathbf{L_9}$	64	14	52	30	32	0	0	-1
$\mathbf{L_{10}}$	45	70	0	77	9	0	0	-1
$\mathbf{L_{11}}$	2	56	57	35	12	0	0	-1
$\mathbf{L_{12}}$	24	61	60	27	51	16	1	0

Table 3.4: Block shift matrix $\boldsymbol{\beta_S}$ showing the right-shift values for the valid blocks to be processed.

Chapter 4

Layer-Pipelined Decoder Architecture

The value of partitioning the PCM $\mathbf{H}$ into layers lies in the fact that, independent layers can be determined and processed in a pipelined manner resulting in an almost-parallel performance. We call this almost-parallel due to the inherent pipelining overhead involved. In Section 3.3 we saw how dependent layers for a block column cannot be processed in parallel. For instance, in $\mathbf{H}_b$ in Table 2.1, VNs associated with the block column B_1 participate in CN equations associated with all the layers except layer L_{10}, suggesting that there is no scope of parallelization of layer processing at all. This situation is better observed in $\boldsymbol{\beta_I}$ shown in Table 3.3. Layer independence can be defined in terms of $\boldsymbol{\beta_I}$ as follows.

Fact 3. If a block column of $\boldsymbol{\beta_I}$ has a particular index value appearing in more than one layer, then the layers corresponding to that value are dependent with respect to that block column.

Proof. Follows directly by applying Fact 2 to Definition 3. □

In other words, $\forall u, u' \in \{1, 2, \ldots, I\}$, $\forall w \in \{1, 2, \ldots, J\}$, if, $\boldsymbol{\beta_I}(u, w) = \boldsymbol{\beta_I}(u', w)$ then, the layers L_u and $L_{u'}$ are dependent. It is obvious that, to process all layers in parallel (L_1 to L_{12} in 2.1), the condition,

$$\boldsymbol{\beta_I}(u, w) \neq \boldsymbol{\beta_I}(u', w) \tag{4.1}$$

must hold for $\forall u, u' \in \{1, 2, \ldots, I\}$. We call the set of layers $\mathcal{L}_s$ satisfying Fact 3 as a *superlayer*. As will be seen later, the formation of superlayers of suitable size is crucial to achieve parallelism in the architecture.

Layers ↓	**Blocks ⟶**							
	$\mathbf{b_1}$	$\mathbf{b_2}$	$\mathbf{b_3}$	$\mathbf{b_4}$	$\mathbf{b_5}$	$\mathbf{b_6}$	$\mathbf{b_7}$	$\mathbf{b_8}$
$\mathbf{L_1}$	**0**	**4**	8	13	6	10	12	-1
$\mathbf{L_2}$	9	**0**	**4**	**8**	13	14	2	-1
$\mathbf{L_3}$	15	9	**0**	**4**	**8**	5	14	-1
$\mathbf{L_4}$	7	15	16	**0**	**4**	**8**	1	-1
$\mathbf{L_5}$	17	7	3	16	**0**	4	**8**	-1
$\mathbf{L_6}$	6	17	18	11	-1	0	4	8
$\mathbf{L_7}$	19	6	**0**	**8**	1	2	18	12
$\mathbf{L_8}$	4	19	5	**0**	**8**	20	**10**	-1
$\mathbf{L_9}$	21	**4**	11	5	**0**	**8**	20	-1
$\mathbf{L_{10}}$	1	21	**4**	3	22	9	**8**	-1
$\mathbf{L_{11}}$	**0**	1	23	**4**	3	22	**10**	-1
$\mathbf{L_{12}}$	8	**0**	2	23	**4**	12	7	11

Table 4.1: Rearranged Block Index Matrix $\boldsymbol{\beta}'_I$ used for our work, showing the valid blocks (highlighted) to be processed.

4.1 Pipelining GNPU and LNPU Arrays

In Section 3.1 we saw that the GNPU feeds the LNPU necessarily in that order. While processing the updates for a particular block within a layer, the LNPU must wait for the GNPU to finish generating the global update values. This is shown in Fig. 4.1 (a), where the GNPU and the LNPU idle alternately. We call this version of our implementation as the $1x$ version pointing to the fact that there isn't any multi-fold improvement in throughput. A natural improvement is to pipeline the processing of the GNPU and the LNPU arrays. However, this cannot be scheduled in the original order of blocks specified by $\boldsymbol{\beta_I}$. The remedy is to rearrange the $\boldsymbol{\beta_I}$ matrix elements from their original order. If $\boldsymbol{\beta_I}(u, w) = \boldsymbol{\beta_I}(u', w)$, $u < u'$ then *stagger* the execution of $\boldsymbol{\beta_I}(u', w)$ with respect to $\boldsymbol{\beta_I}(u, w)$ by placing $\boldsymbol{\beta_I}(u', w)$ in $\boldsymbol{\beta'_I}(u', w')$ such that, $w < w'$. The pipelining schedule is based on the following Lemma.

Lemma 1. *Within a superlayer, while the LNPU processes messages for the blocks* $\boldsymbol{\beta'}(u, w)$*, the GNPU can process messages for the blocks* $\boldsymbol{\beta'}(u+1, w)$*,* $u = \{1, 2, \ldots, |\mathcal{L}|-$

$1\}$ *and* $w = \{1, 2, \ldots, J\}$.

Proof. Follows directly from the layer independence condition in Fact 2. □

Fig. 4.1(c) illustrates the block-level view of this 2-layer pipelining scheme. We call this the $2x$ version of our implementation due to the almost doubling of throughput with respect to the non-pipelined $1x$ version. At the boundary of the superlayer Lemma 1 does not hold and pipelining has to be restarted for the next layer as seen in the layer-level view shown in Fig. 4.2(c). This is simply due to the fact that the condition in Fact 3 only holds within a superlayer. In the following, we impose certain constraints on the size of the superlayers in $\mathbf{H}$.

Definition 5. *Without loss of generality, the pipelining efficiency* η_p *is the number of layers processed per unit time per NPU array.*

For the case of pipelining two layers shown in Fig. 4.2(c),

$$\eta_p^{(2)} = \frac{|\mathcal{L}_s|}{|\mathcal{L}_s| + 1} \tag{4.2}$$

Thus, we impose the following conditions on $|\mathcal{L}_s|$:

1. Since, two layers are processed in the pipeline at any given time, provided that I is even,

$$|\mathcal{L}_s| \in \mathcal{F} = \{x : x \text{ is an even factor of } I\}.$$

   It is important to note that, for any value of $|\mathcal{L}_s| \in \mathcal{F}$, $\mathcal{L}_s$ must be a superlayer.

2. Given a QC-LDPC code, $|\mathcal{L}_s|$ is a constant. This is to facilitate a symmetric pipelining architecture which is a scalable solution.

3. Choice of $|\mathcal{L}_s|$ should maximize pipelining efficiency $\eta_p^{(2)}$,

$$l^* = \underset{|\mathcal{L}_s| \in \mathcal{F}}{\arg\max}\, \eta_p^{(2)} \tag{4.3}$$

Case Study: Table 4.1 shows one such rearrangement of $\boldsymbol{\beta_I}$ for the QC-LDPC code for our case study in Table 3.3. Note that, some dependencies still remain, these unresolved dependencies are shown in a boldfaced font in Table 4.1. For example, in the case study, the chosen IEEE 802.11n (2012) code has $I = m_b = 12$, $\mathcal{F} = \{2, 4, 6\}$ and, $l^* = \arg\max_{|\mathcal{L}_s| \in \mathcal{F}} \eta_p^{(2)} = 6$. The rearranged block index matrix $\boldsymbol{\beta'_I}$ is shown in Table 4.1 and the layer-level view of the pipeline timing diagram for the same is shown in Fig. 4.2(d).

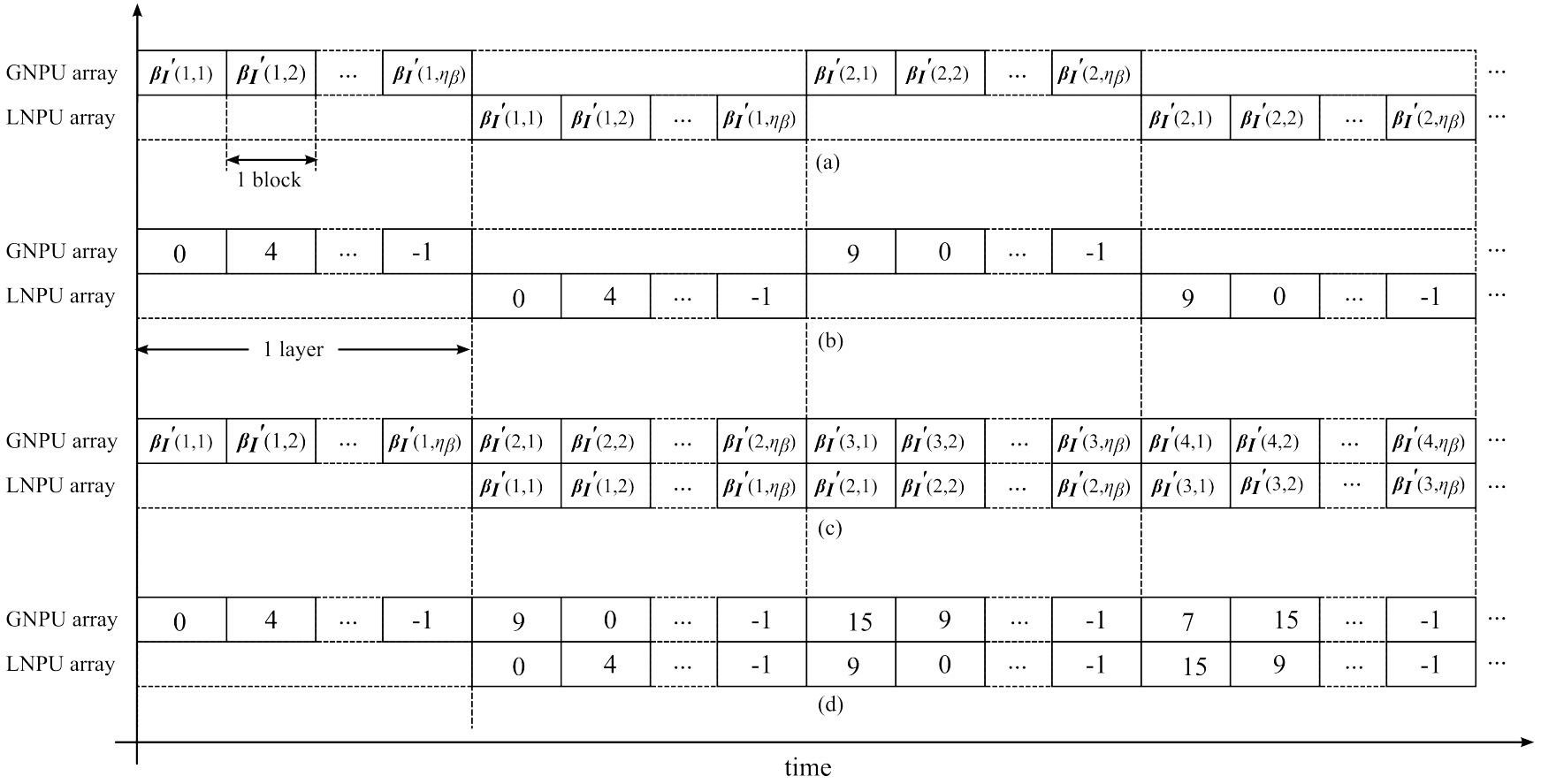

Figure 4.1: Block-level view of the pipeline timing diagram. (a) General case for a circulant-1 identity submatrix construction based QC-LDPC code (see Section 2.1) without pipelining. (b) Special case of the IEEE 802.11n QC-LDPC code used in this work without pipelining (c) Pipelined processing of two layers for the general QC-LDPC code case in (a). (d) Pipelined processing of two layers for the IEEE802.11n QC-LDPC code case in (b).

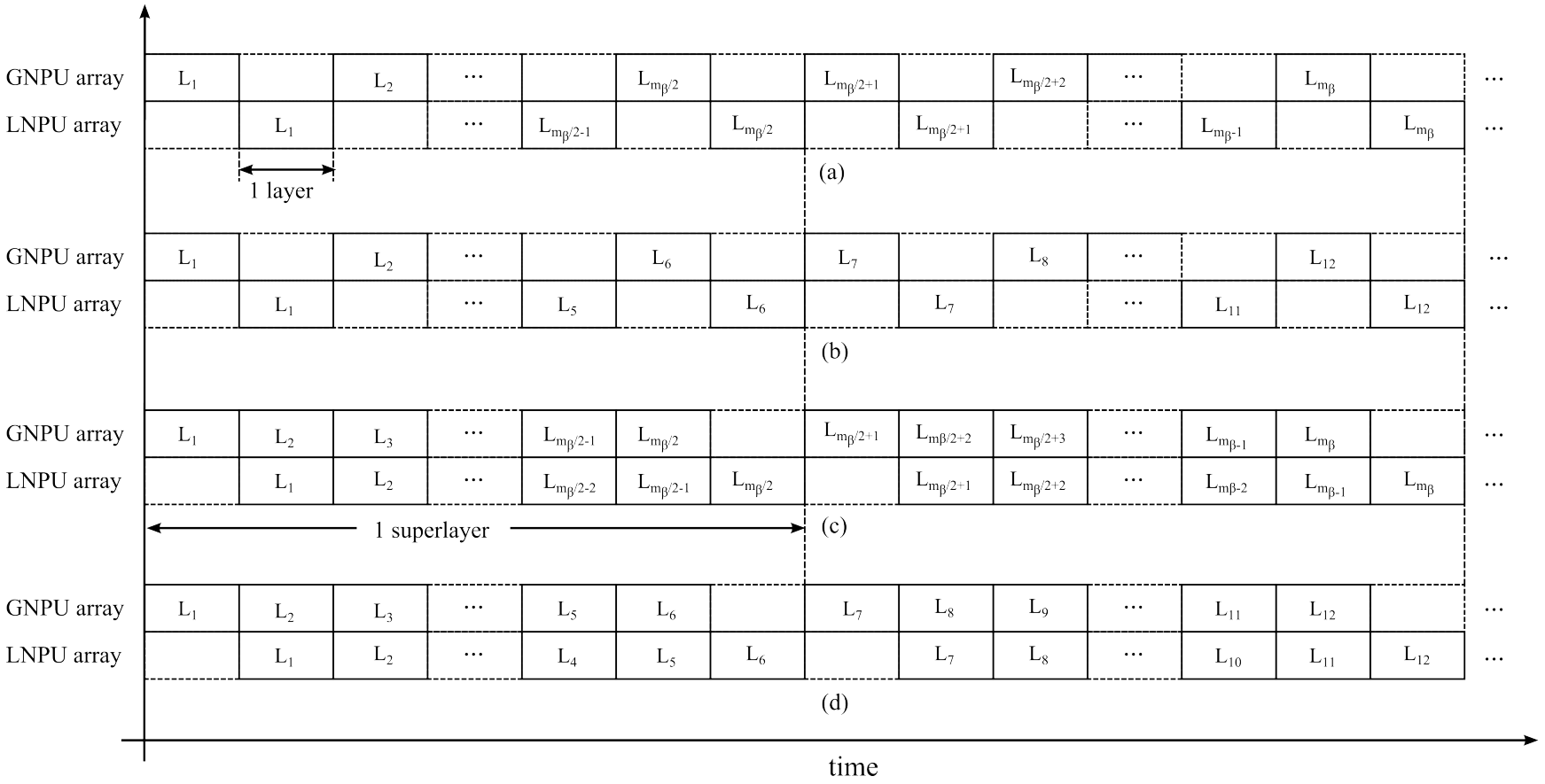

Figure 4.2: Layer-level view of the pipeline timing diagram. (a) General case for a circulant-1 identity submatrix construction based QC-LDPC code (see Section 2.1) without pipelining. (b) Special case of the IEEE 802.11n QC-LDPC code used in this work without pipelining (c) Pipelined processing of two layers for the general QC-LDPC code case in (a). (d) Pipelined processing of two layers for the IEEE802.11n QC-LDPC code case in (b).

High-level FPGA-based Decoder Architecture: The high-level decoder architecture is shown in Fig. 4.3. The ROM holds the LDPC code parameters specified by the $\boldsymbol{\beta}'_I$ and the $\boldsymbol{\beta}'_s$ along with other code parameters such as the block length and the maximum number of decoding iterations. The APP memory is initialized with the channel LLR values corresponding to all the VNs as per equation (2.1). The barrel shifter operates on blocks of VNs (APP values in equation (2.4)) of size $z \times f$, where f is the fixed-point word length used in the implementation for APP values. It circularly rotates the values to the right by using the shift values from the $\boldsymbol{\beta}'_s$ matrix in the ROM, effectively implementing the connections between the CNs and VNs. The cyclically shifted APP memory values and the corresponding CN message values for the block in question are fed to the NPU arrays. Here, the GNPUs compute VN messages as per equation (2.2) and the LNPUs compute CN messages as per equation (2.3). These messages are then stored back at their respective locations in the RAMs for processing the next block. Once the processing of all blocks within a layer is done, blocks in the next layer are processed as depicted by the *Block Processing* loop within the *Layer Processing* loop in Fig. 4.3.

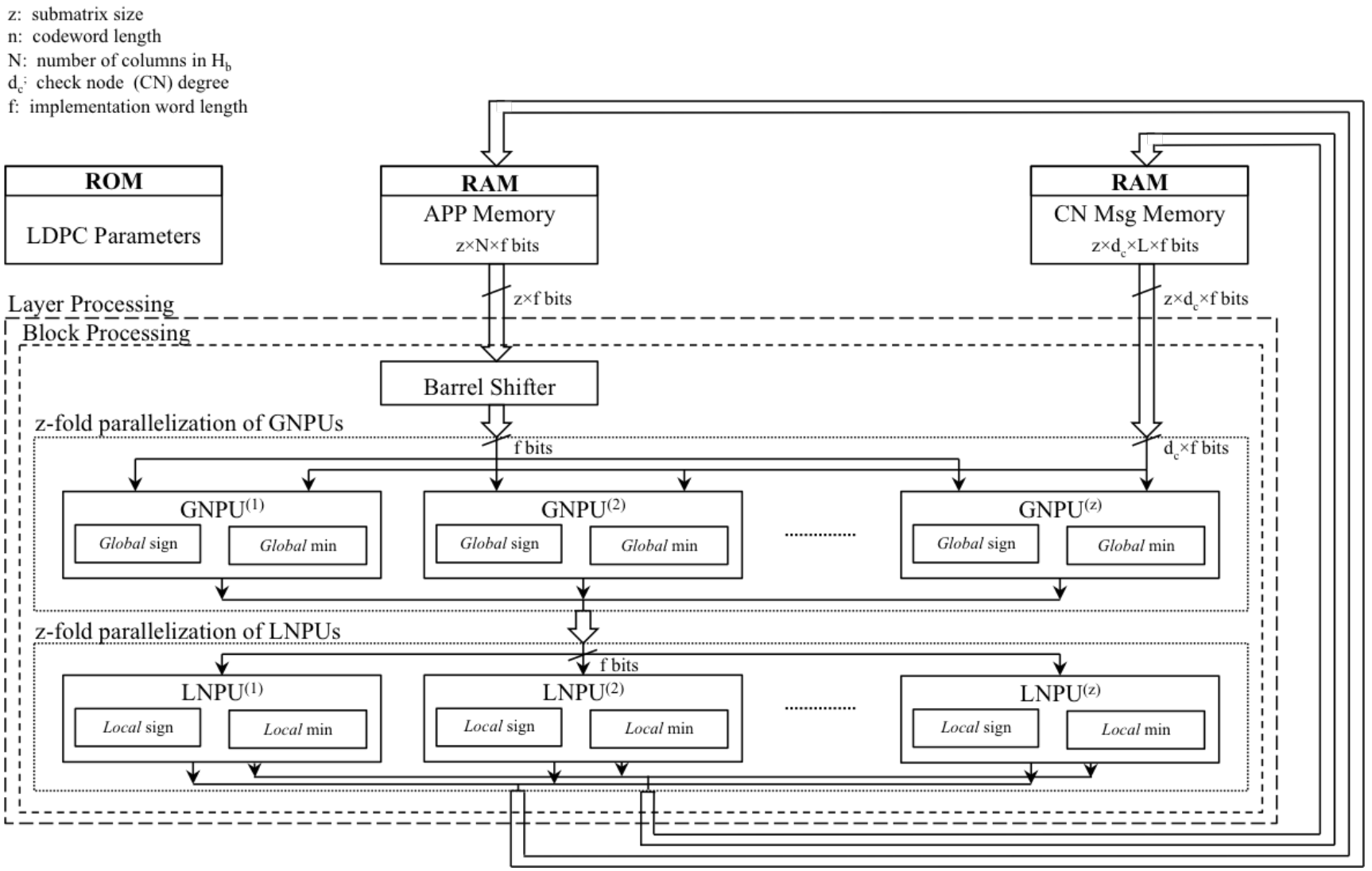

Figure 4.3: High-level decoder architecture. showing the z-fold parallelization of the NPUs with an emphasis on the splitting of the sign and the minimum computation given in equation (2.3). Note that, other computations in equations (2.1)-(2.4) are not shown for simplicity here. For both the pipelined and the non-pipelined versions, processing schedule for the inner Block Processing loop is as per Fig. 4.1 and that for the outer Layer Processing loop is as per Fig. 4.2.

Chapter 5

Case Study

We believe that the family of structured LDPC codes are highly likely candidates for 5G systems. Thus, to demonstrate the initial phase of our FPGA decoder architecture [1], we provide a case study based on the QC-LDPC code specified in the *IEEE 802.11n (2012)* standard [5]. For this code, $m_b \times n_b = 12 \times 24$, $z = 27$, 54 and 81 resulting in code lengths of $n = 24 \times z = 648$, 1296 and 1944 bits respectively. Our implementation supports the submatrix size of $z = 81$ and hence is capable of supporting all the block lengths for the rate $R = 1/2$ code.

At the time of writing this paper, we have implemented two versions, the $1x$ (block level view in Fig. 4.1 (a) and (b) and layer level view in Fig. 4.2 (a) and (b)) and the $2x$ (block level view in Fig. 4.1 (c) and (d) and layer level view in Fig. 4.2 (c) and (d)). For both versions, input LLRs from the channel and the CTV and VTC messages are represented with 6 signed bits and 4 fractional bits. Fig. 6.2 shows the bit-error rate (BER) performance for the floating-point and the fixed-point data representation (0.5dB worse as expected, Fig. 6.2) with 8 decoding iterations.

The decoder algorithm for both versions was described using the *FPGA IP* compiler [17] in *LabVIEW*^{TM} *CSDS*^{TM}. We would like to emphasize here that, both the versions were described in software at the algorithmic description level and not the HDL level. The algorithmic compiler translated the high-level description to an HDL description for the case study decoder implementation in approximately 3 minutes. The VHDL code was synthesized, placed and routed using the *Xilinx Vivado* compiler on the *Xilinx Kintex-7* FPGA available on the *NI PXIe-7975R* FPGA board.

The $2x$ version achieves an overall throughput of 608Mb/s at an operating frequency of 260MHz and a latency of 5.7μs with 4 decoding iterations. As seen in Table 5.1,

	1x	**2x**
Device	*Kintex-7k410t*	*Kintex-7k410t*
Throughput(Mb/s)	337	608
FF(%)	9.1	5.3
BRAM(%)	4.7	6.4
DSP48(%)	5.2	5.2
LUT(%)	8.7	8.2

Table 5.1: LDPC Decoder IP FPGA Resource Utilization & Throughput on the Xilinx *Kintex-7 FPGA*.

resource usage for the *2x* version is close to that of the *1x* version in spite of the $1.8x$ gain in throughput. The *FPGA IP* compiler chooses to use more FF for data storage in the *1x* version, while it uses more BRAM in *2x* version. A contemporary implementation of the *IEEE 802.11n* LDPC decoder on an FPGA (using high-level algorithmic description compiled to an HDL) shown in [16]. The decoder in [16] utilizes 2% of slice registers, 3% of slice LUTs and 20.9% of Block RAMs on the *Spartan-6 LX150T* FPGA with a comparable BER performance.

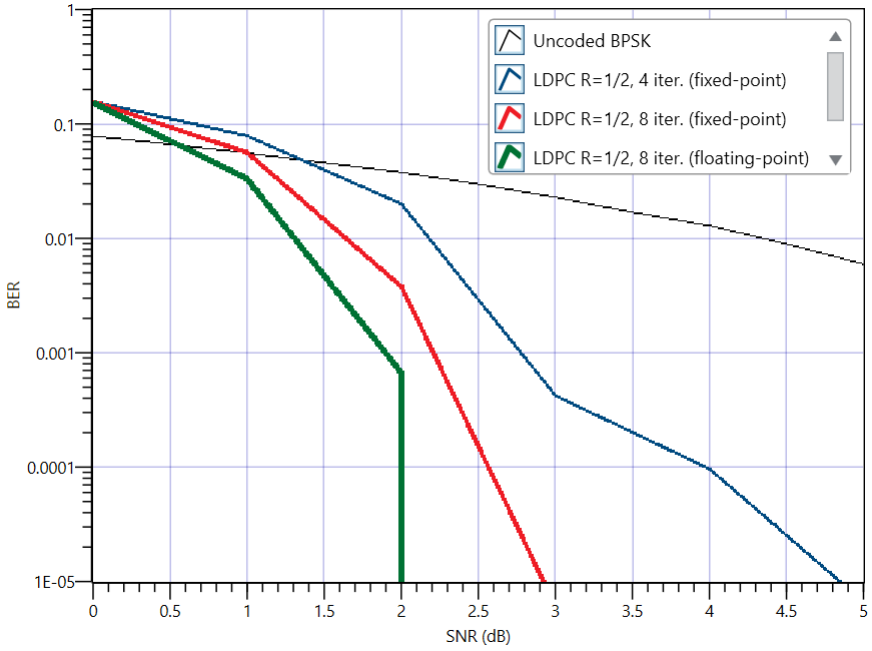

Figure 5.1: Bit Error Rate (BER) performance comparison between uncoded BPSK (rightmost), rate=1/2 LDPC with 4 iterations using fixed-point data representation (second from right), rate=1/2 LDPC with 8 iterations using fixed-point data representation (third from right), rate=1/2 LDPC with 8 iterations using floating-point data representation (leftmost).

Chapter 6

Application
2.48Gb/s QC-LDPC Decoder on the NI USRP-2953R

In this chapter we present an application of the decoder architecture, a 2.48Gb/s FPGA-based QC-LDPC decoder implemented on the *NI USRP-2953R* (which has the *Xilinx Kintex7 (410t)* FPGA) using the *FPGA IP* compiler in *LabVIEW*^{TM} *CSDS*^{TM}. Massive-parallelization was accomplished by employing 6 decoder cores in parallel without any modification at the HDL level. This compiler translated the entire high-level description of the parallelization (done in a graphical algorithmic dataflow language) to VHDL and further generated an optimized hardware implementation from the algorithmic description. This application demonstrates:

1. The scalability of our decoder architecture [1] described in this report.
2. The ability of the *LabVIEW*^{TM} *CSDS*^{TM} tools to rapidly prototype high-level algorithmic description onto FPGA hardware.

This application has been demonstrated in *IEEE GLOBECOM'14* where the QC-LDPC code for our case study was decoded with a throughput of 2.06 Gb/s. This throughput was achieved by using five decoder cores in parallel on the *Xilinx K7 (410t)* FPGA in the NI USRP-2953R.

6.1 Multi-core Decoder

The implementation of the massively-parallel decoder described in this Chapter is based on the $2x$ decoder version.

Remark 3. Note that, at the time of developing this application the algorithmic description for the $1x$ and $2x$ was recompiled and it achieved a throughput of 290Mb/s

and 420Mb/s (at 200MHz) respectively as given in Table 6.1; the compilation results in Table 6.1 and Table 5.1 are different. For the sake of clarity, in the context of this application, the recompiled versions of the $1x$ and the $2x$ version are hereafter referred to as the *Baseline* and the *Pipelined* version respectively. Also, a *core* in the context of this application refers to the Pipelined version.

As discussed in Chapter 5, the core operates for $m_b \times n_b = 12 \times 24$, $z = 27$, 54 and 81 resulting in code lengths of $n = 24 \times z = 648$, 1296 and 1944 bits respectively and a code rate $R = \frac{1}{2}$. It is worthwhile to note that, for the Pipelined version of the decoder, pipelining was fully described in software. Moreover, the algorithm was described in a high-level language - graphical code in LabVIEW$^{\text{TM}}$ (i.e. not in a hardware description language). The algorithmic compiler in *LabVIEW^{TM}* CSDS$^{\text{TM}}$ translated the high-level description into a VHDL description.

On account of the scalability and reconfigurability of the decoder architecture in [1], it is possible to achieve high throughput by employing multiple decoder cores in parallel. Fig. 6.1 shows the top-level multi-core decoder virtual instrument (VI), where 6 cores are deployed on a single *Xilinx Kintex7 FPGA (410t)*. The high-level operation of the decoder is described in the steps below (corresponding to the highlighted sections in Fig. 6.1):

1. Serial stream of the encoded data is read as frames from the host-to-target Direct Memory Access (DMA) mechanism. Here, host may be an arbitrary processing platform such as a PC or a real-time controller and target is the *Xilinx Kintex7 FPGA (410t)* on the *NI USRP-2953R*. This data is subsequently stored in the Dynamic Random Access Memory (DRAM).

2. Request frames from the DRAM.

3. Read and buffer frames from the DRAM.

4. Distribute incoming frames to the cores in a round-robin manner.

5. Perform decoding with fixed-latency, parallel processing of frames staggered with respect to time. Buffer the decoded frames.

6. Collect the decoded frames and serialize them with respect to the round-robin manner used in step (3).

7. Write frames to the target-to-host DMA mechanism.

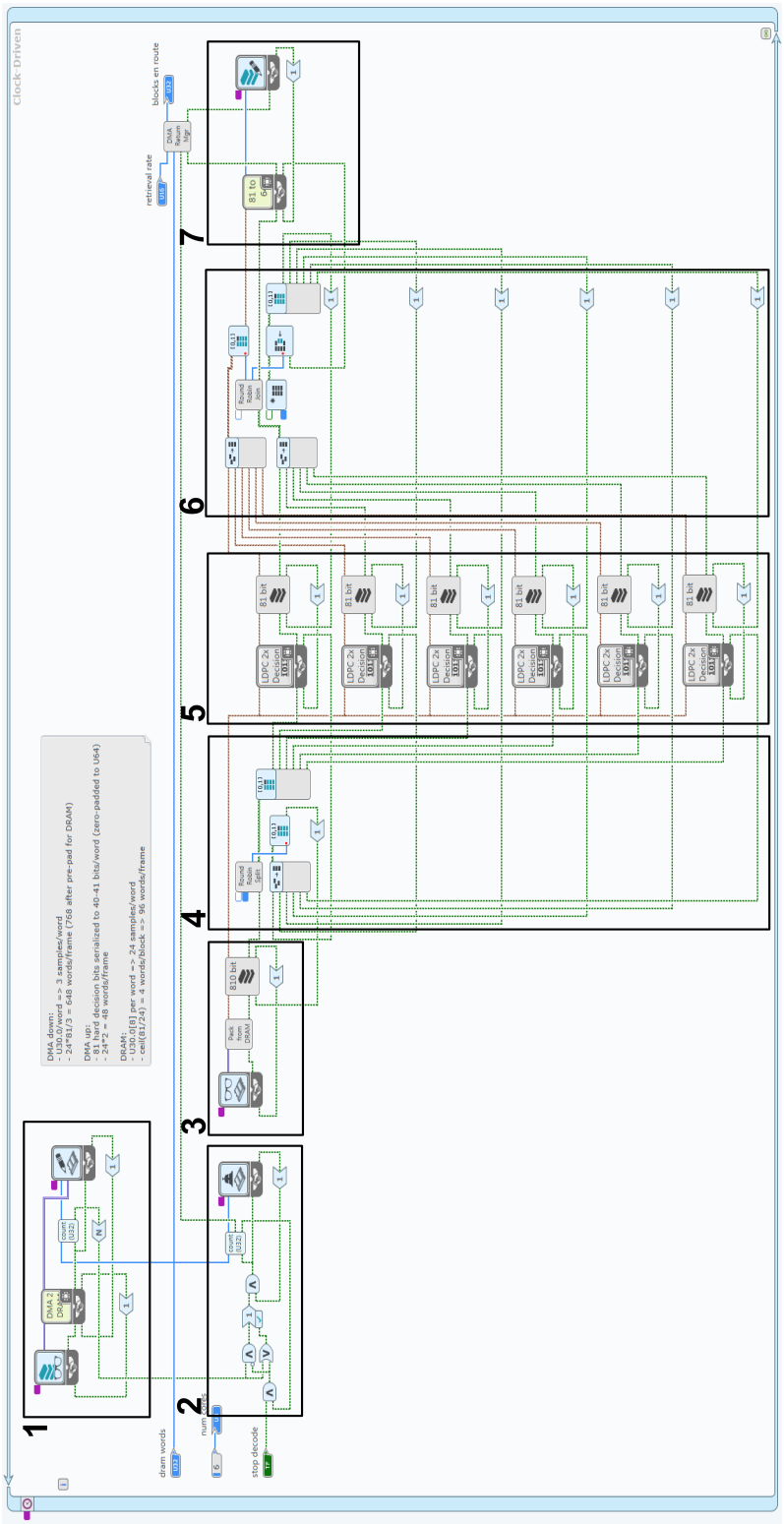

Figure 6.1: Top-level VI describing the parallelization of the QC-LDPC decoder [1] on the *NI USRP-2953R* containing the Xilinx *Kintex7 (410t)* FPGA.

	Baseline	Pipelined
Throughput (Mb/s)	290	420
Clock Rate (MHz)	200	200
Time to generate VHDL (min)	2.02	2.08
Total Compile Time (min)	≈ 36	≈ 36
Total Slice (%)	26	28
LUT (%)	16	18
FF (%)	9	10
DSP (%)	5	5
BRAM (%)	11	11

Table 6.1: Performance and resource utilization comparison for the Baseline architecture with the Pipelined architecture of the QC-LDPC decoder on the *NI USRP-2953R* containing the Xilinx *Kintex7 (410t)* FPGA.

6.2 Results

The performance and resource utilization of the Baseline and the Pipelined version is compared in Table 6.1. The resources consumed by the Pipelined decoder are almost the same as that of the Baseline decoder, in spite of the 1.5 times increase in throughput performance. The 2.48Gb/s decoder was developed in stages, where at each stage a core was added (except for stage 3) and the performance and resource figures were recorded. The results of each stage are compared in Table 6.2. The Bit Error Rate (BER) performance of the 2.48Gb/s version (with 6 cores) is shown in Fig. 6.2.

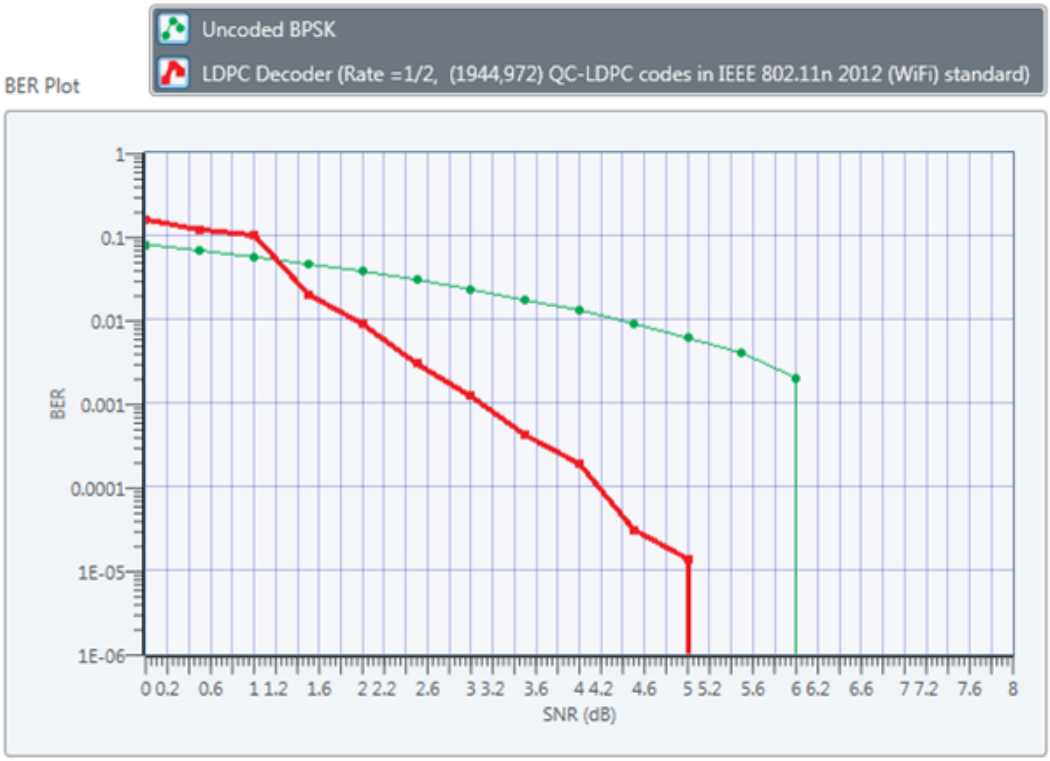

Figure 6.2: Bit Error Rate (BER) performance comparison between uncoded BPSK (green) and the 2.48Gb/s, rate=1/2, QC-LDPC decoder (red) on the *NI USRP-2953R* containing the Xilinx *Kintex7 (410t)* FPGA.

Cores	**1**	**2**	**4**	**5**	**6**
Throughput (Mb/s)	420	830	1650	2060	2476
Clock Rate (MHz)	200	200	200	200	200
Time to VHDL (min)	2.08	2.08	2.08	2.02	2.04
Total Compile (min)	≈ 36	≈ 60	≈ 104	≈ 132	≈ 145
Total Slice (%)	28	44	77	85	97
LUT (%)	18	28	51	62	73
FF (%)	10	16	28	33	39
DSP (%)	5	11	21	26	32
BRAM (%)	11	18	31	38	44

Table 6.2: Performance and resource utilization comparison for versions with varying number of cores of the QC-LDPC decoder implemented on the *NI USRP-2953R* containing the Xilinx *Kintex7 (410t)* FPGA.

Chapter 7

Related Work
3GPP UMTS Turbo Decoder

Turbo codes are a class of concatenated error-correcting codes known for their near-capacity performance. We have implemented the Turbo code specified in the 3GPP UMTS standard [26]. The UMTS turbo code is a rate 1/3 Parallel Concatenated Convolutional Code (PCCC) with two 8-state recursive systematic convolutional (RSC) constituent codes: a feedforward polynomial of 15_o and a feedback polynomial of 13_o. The two constituent codes are separated by an interleaver fundamental to the performance of a turbo code [27, 3].

Turbo codes perform at near-capacity owing to their near-random code design. However, they also exhibit enough structure allowing iterative decoding (although suboptimal). Fig. 7.1 shows LabVIEW source code for the iterative decoder. The Log-MAP BCJR [28] based soft-in soft-out (SISO) decoders for each constituent code work in an iterative manner until the final estimate for each bit has been achieved. One full iteration with the indicated feedback path *Lapp2_xk* consists of two half iterations, one for each of the RSC decoders. Although MAP decoding for each SISO decoder achieves the best performance [3], it does so using a forward (α and γ)-backward (β) recursion (unlike the popular Viterbi decoder [29] which primarily only uses a forward recursion) and an information bit decoding stage. Unlike the relatively simpler add-compare-select (ACS) recursive computation in the Viterbi, the Log-MAP BCJR uses the Jacobi logarithmic approximation ($max^*(x, y)$). These two factors play a major role in increasing the complexity of the Log BCJR relative to the Viterbi algorithm.

In this implementation, we have employed the sliding window version of the BCJR (SW-BCJR), first introduced in [30]. Here the input data block is divided into smaller

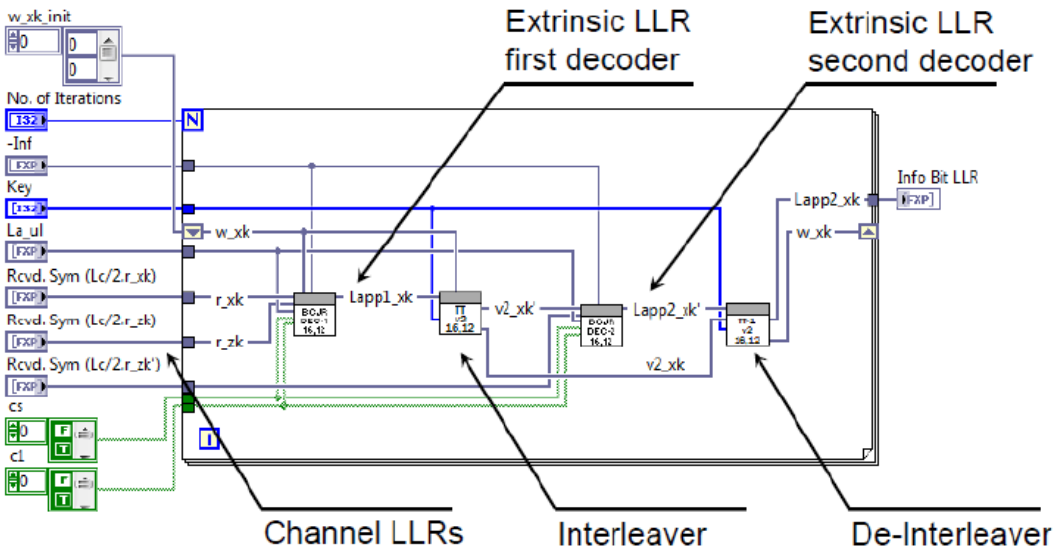

Figure 7.1: Turbo Decoder Iterative Log-MAP Decoder

segments or *windows*. Since the decoder operates on smaller segments, decisions are forced with a smaller delay consequently reducing the decoding delay and the memory requirement.

Given the bi-directional nature of processing and the presence of an interleaver and deinterleaver pair, the challenge lies in implementing high-throughput iterative decoders with a small delay. Since the bit decoding has an intra-iteration dependency to the α and γ recursion, the naive implementation is to run all the computation blocks serially. However, there is no inter-iteration dependency between these two computations, and the *next* α and γ recursion and the *current* bit decoding block can run in parallel. By analyzing the *memory access pattern* this inter-iteration dependency can be detected and hence eliminated. The LabVIEW FPGA compiler captures both the dataflow dependency and memory access pattern dependency between computational blocks and exploits data parallelism across iterations.

Fig. 7.2 shows the bit-error-rate(BER) performance of the implemented turbo decoder and the uncoded binary phase shift key (BPSK) modulated data stream. The simulation is performed with a coded block length of 1024 bits, two decoding iterations and fixed-point word length of (16,4). The BER (Y-axis) is plotted using ten frames

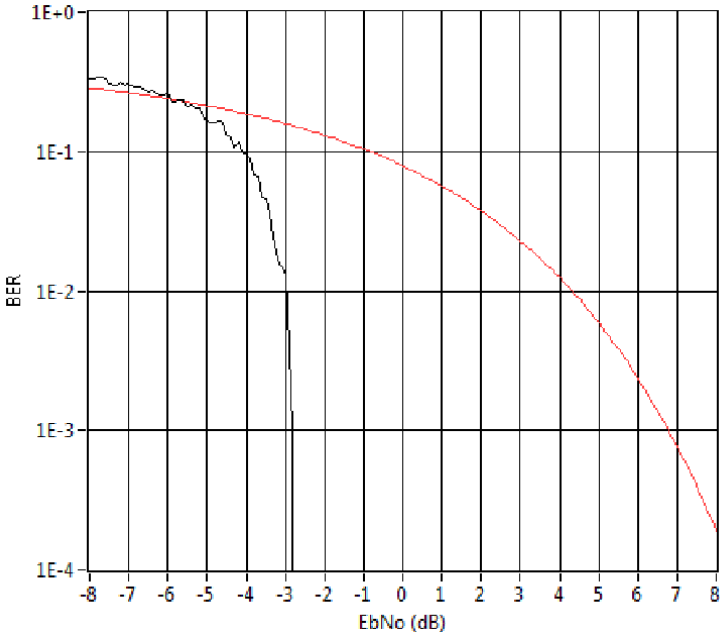

Figure 7.2: BER performance of the aforementioned turbo decoder (curve on the left in black) versus uncoded BPSK (curve on the right in red)

.

	Data Width	**Throughput**	**LUT/FF Pairs**	**RAM**
LabVIEW	16 bits	3.388 Mb/s	13,774	44
Xilinx IP [31]	5 bits	35.58 Mb/s	4,717	10

Table 7.1: Turbo Decoder IP FPGA Resource Utilization & Throughput on the Xilinx Kintex-7 XC7K325T-2L-FFG900.

(1024 bits each) per signal-to-noise ratio (SNR) point (X-axis).

Table 7.1 shows the quality-of-result (QoR) of the turbo decoder design automatically generated by LabVIEW compiler. Xilinx hand-designed IP [31] shows better hardware QoR mainly because of two reasons. Firstly, because our core processing has wider input data. From the perspective of FPGA resource utilization, it is well known that wider data processing results in a relatively expensive computational circuit. Secondly, the Xilinx turbo decoder IP employs the *Max-Log MAP* decoding algorithm for the constituent decoders. In our implementation, we use the *Log MAP* algorithm, which

has a higher processing complexity [3]. However, since *Max-Log MAP* is an approximation to the the *Log MAP* decoding algorithm, our IP shows a better BER performance. The improvement in the BER performance in our implementation is an indicator of the trade off between the computational complexity of the decoding algorithm and the hardware resource utilization.

Chapter 8

Conclusion

In this work we have proposed techniques to achieve high-throughput performance for a MSA-based decoder for QC-LDPC codes. The proposed compact representation of the PCM provides significant improvement in throughput. An IEEE 802.11n (2012) decoder is implemented which attains a throughput of 608Mb/s (at 260MHz) and a latency of 5.7μs on the *Xilinx Kintex-7* FPGA. The *FPGA IP* compiler greatly reduces prototyping time and is capable of implementing complex signal processing algorithms. With little or no modification this decoder can be applied to a large family of standard compliant QC-LDPC codes such as those specified in IEEE 802.16e [6] and Digital Video Broadcast (DVB) [7]. The *NI USRP-2953R* application validates the scalability of our decoder architecture by deploying multiple decoder cores in parallel.

References

[1] S. Mhaske, H. Kee, T. Ly, A. Aziz, and P. Spasojevic, "High-Throughput FPGA-based QC-LDPC Decoder Architecture," in *Vehicular Technology Conference (VTC Fall), 2015 IEEE 82nd*, Sep 2015, pp. 1–5.

[2] M. Cudak, A. Ghosh, T. Kovarik, R. Ratasuk, T. Thomas, F. Vook, and P. Moorut, "Moving Towards Mmwave-Based Beyond-4G (B-4G) Technology," in *IEEE 77th VTC Spring '13*, June 2013, pp. 1–5.

[3] D. Costello and S. Lin, *Error control coding*. Pearson, 2004.

[4] W. Ryan and S. Lin, *Channel Codes: Classical and Modern*. Cambridge University Press, 2009.

[5] "IEEE Std. for Information Technology–Telecommunications and information exchange between LAN and MAN–Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE P802.11-REVmb/D12, Nov 2011*, pp. 1–2910.

[6] "IEEE Std. for Air Interface for Broadband Wireless Access Systems," *IEEE P802.16, Aug 2012*, pp. 1–2544.

[7] "EN 302 307-1 V1.4.1, Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications; Part 1: DVB-S2 ," *ETSI 2014*, pp. 1–80.

[8] Y. Sun and J. Cavallaro, "VLSI Architecture for Layered Decoding of QC-LDPC Codes With High Circulant Weight," *IEEE Transactions on VLSI Systems*, vol. 21, no. 10, pp. 1960–1964, Oct 2013.

[9] K. Zhang, X. Huang, and Z. Wang, "High-throughput layered decoder implementation for QC-LDPC codes," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 6, pp. 985–994, Aug 2009.

[10] N. Onizawa, T. Hanyu, and V. Gaudet, "Design of high-throughput fully parallel ldpc decoders based on wire partitioning," *IEEE Transactions on VLSI Systems*, vol. 18, no. 3, pp. 482–489, Mar 2010.

[11] T. Mohsenin, D. Truong, and B. Baas, "A low-complexity message-passing algorithm for reduced routing congestion in ldpc decoders," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 5, pp. 1048–1061, May 2010.

[12] A. Balatsoukas-Stimming and A. Dollas, "FPGA-based design and implementation of a multi-Gbps LDPC decoder," in *International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 262–269.

[13] V. Chandrasetty and S. Aziz, "FPGA Implementation of High Performance LDPC Decoder Using Modified 2-Bit Min-Sum Algorithm," in *International Conference on Computer Research and Development*, May 2010, pp. 881–885.

[14] R. Zarubica, S. Wilson, and E. Hall, "Multi-gbps fpga-based low density parity check (ldpc) decoder design," in *IEEE GLOBECOM '07*, Nov 2007, pp. 548–552.

[15] P. Schläfer, C. Weis, N. Wehn, and M. Alles, "Design space of flexible multigigabit ldpc decoders," *VLSI Design*, vol. 2012, p. 4, 2012.

[16] E. Scheiber, G. H. Bruck, and P. Jung, "Implementation of an LDPC decoder for IEEE 802.11n using Vivado TM High-Level Synthesis," in *International Conference on Electronics, Signal Processing and Communication Systems*, 2013.

[17] H. Kee, S. Mhaske, D. Uliana, A. Arnesen, N. Petersen, T. L. Riche, D. Blasig, and T. Ly, "Rapid and high-level constraint-driven prototyping using LabVIEW FPGA," in *2014 IEEE , GlobalSIP 2014*, 2014.

[18] S. Mhaske, D. Uliana, H. Kee, T. Ly, A. Aziz, and P. Spasojevic, "A 2.48Gb/s QC-LDPC Decoder Implementation on the NI USRP-2953R," in *arxiv.org*, pp. 1–5, arxiv.org.

[19] R. G. Gallager, "Low-density parity-check codes," *Information Theory, IRE Transactions on*, vol. 8, no. 1, pp. 21–28, 1962.

[20] R. Tanner, "A recursive approach to low complexity codes," *Information Theory, IEEE Transactions on*, vol. 27, no. 5, pp. 533–547, Sep 1981.

[21] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *Information Theory, IEEE Transactions on*, vol. 47, no. 2, pp. 498–519, Feb 2001.

[22] E. Sharon, S. Litsyn, and J. Goldberger, "Efficient Serial Message-Passing Schedules for LDPC Decoding," *IEEE Transactions on Information Theory*, vol. 53, no. 11, pp. 4076–4091, Nov 2007.

[23] M. Mansour and N. Shanbhag, "High-throughput LDPC decoders," *IEEE Transactions on VLSI Systems*, vol. 11, no. 6, pp. 976–996, Dec 2003.

[24] J. Chen and M. Fossorier, "Near optimum universal belief propagation based decoding of ldpc codes and extension to turbo decoding," in *IEEE ISIT '01*, 2001, p. 189.

[25] K. Gunnam, G. Choi, M. Yeary, and M. Atiquzzaman, "VLSI Architectures for Layered Decoding for Irregular LDPC Codes of WiMax," in *IEEE ICC '07*, June 2007, pp. 4542–4547.

[26] "LTE, Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and Channel Coding," *3GPP TS 36.212 version 10.5.0, Oct 2012*, Oct 2012.

[27] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding," in *IEEE ICC '93*, 1993, pp. 1064–1070.

[28] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," in *IEEE Transactions on Information Theory, Vol. 20, Mar 1974*, Mar 1974, p. 284287.

[29] J. G. D. Forney, "The viterbi algorithm," in *Proceedings of the IEEE, Mar 1973*, 1973.

[30] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Soft-output decoding algorithms in iterative decoding of turbo codes," in *TDA Progress Report, NASA JPL, Feb 1996*, 1996.

[31] "Xilinx DS318 3GPP Turbo Decoder v4.0," *xilinx.com*.