Hindawi
International Journal of Reconfigurable Computing
Volume 2017, Article ID 3689308, 23 pages
https://doi.org/10.1155/2017/3689308

Research Article

FPGA-Based Channel Coding Architectures for 5G Wireless Using High-Level Synthesis

Swapnil Mhaske,[1] Hojin Kee,[2] Tai Ly,[2] Ahsan Aziz,[2] and Predrag Spasojevic[1]

[1] *Wireless Information Networking Laboratory, Rutgers University, New Brunswick, NJ 08902, USA*
[2] *National Instruments Corporation, Austin, TX 78759, USA*

Correspondence should be addressed to Swapnil Mhaske; swapnil.mhaske@rutgers.edu

Received 10 December 2016; Revised 3 April 2017; Accepted 24 April 2017; Published 7 June 2017

Academic Editor: João Cardoso

We propose strategies to achieve a high-throughput FPGA architecture for quasi-cyclic low-density parity-check codes based on circulant-1 identity matrix construction. By splitting the node processing operation in the min-sum approximation algorithm, we achieve pipelining in the layered decoding schedule without utilizing additional hardware resources. High-level synthesis compilation is used to design and develop the architecture on the FPGA hardware platform. To validate this architecture, an *IEEE 802.11n* compliant 608 Mb/s decoder is implemented on the *Xilinx Kintex-7* FPGA using the *LabVIEW FPGA Compiler* in the *LabVIEW Communication System Design Suite*. Architecture scalability was leveraged to accomplish a 2.48 Gb/s decoder on a single *Xilinx Kintex-7* FPGA. Further, we present rapidly prototyped experimentation of an *IEEE 802.16* compliant hybrid automatic repeat request system based on the efficient decoder architecture developed. In spite of the *mixed* nature of data processing—digital signal processing and finite-state machines—*LabVIEW FPGA Compiler* significantly reduced time to explore the system parameter space and to optimize in terms of error performance and resource utilization. A 4x improvement in the system throughput, relative to a CPU-based implementation, was achieved to measure the error-rate performance of the system over large, realistic data sets using accelerated, in-hardware simulation.

1. Introduction

The year 2020 is slated to witness the first commercial deployment of the 5th generation of wireless technology. 5G is expected to deliver a uniform Quality of Service (QoS) of 100 Mb/s and peak data rates of up to 20 Gb/s, with over-the-air latency of less than 1 ms [1]. All of this is with the energy consumption of contemporary cellular systems. Channel coding is crucial to achieve good performance in a communication system. Near-capacity performing codes such as Turbo codes [2] and Low-Density Parity-Check (LDPC) codes [3] typically require high-complexity encoding and decoding methods. Today, the standardization efforts towards realizing 5G cellular systems have already begun [4]. The suitability of a particular channel coding scheme is being discussed; and for a system realization of the size of 5G, the evolution of requirements pertaining to channel coding is naturally expected. In our effort to study and design channel codes based on areas ranging from theoretical performance evaluation up to implementation complexity analysis, we have identified two main requirements in the development process. The first one is *flexibility for future modifications*. To facilitate this, we choose the reconfigurable FPGA platform. Moreover, for this evolving architecture, we aim to observe not only the theoretical complexity versus performance trade-off, but also the implementation complexity versus performance trade-off. This brings us to the second major requirement, which is *real-world rapid prototyping* of our methods. Figure 1 summarizes our research methodology. Even though theoretical simulations validate a novel idea, they fail to comprehensively assess its real-world impact. In an effort towards designing and developing a hardware architecture for channel coding, it is crucial to monitor the performance of the system in real-time, on actual state-of-the-art hardware. This helps us keep track of parameters such as throughput, latency, and resource utilization of the system, each time a modification is done. We would also like to emphasize that *rapid prototyping* can be used not only

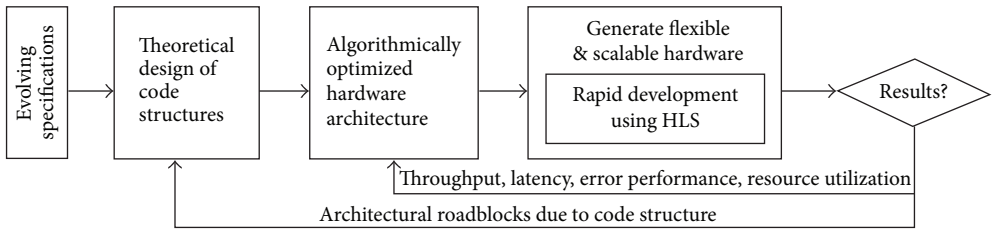


Figure 1: Illustration of our research methodology for the design and development of the channel coding architecture.

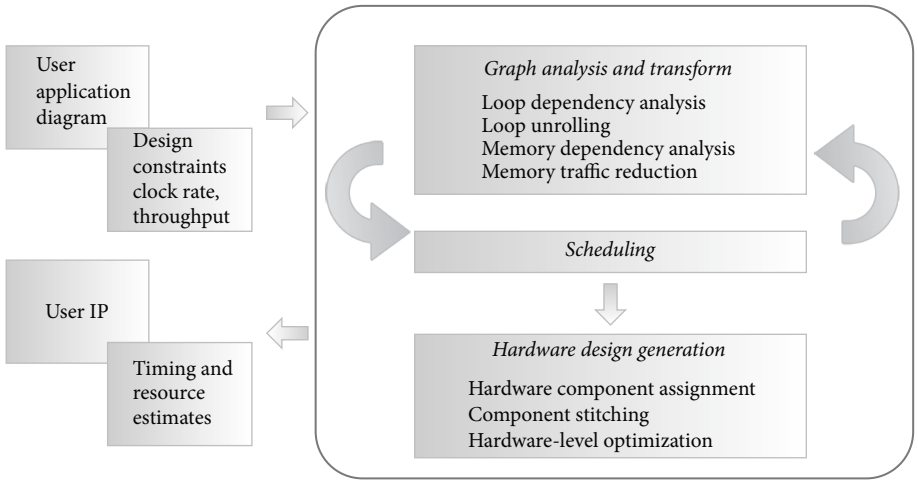


Figure 2: Illustration of the HLS compile flow.

for validating the design on real-world hardware platforms (Sections 5.1 and 5.2), but also for speedup of theoretical simulations (Section 5.3).

To accomplish this, in addition to the use of FPGA-based implementation, we use a High-Level Synthesis (HLS) compiler built in *LabVIEW*, namely, the *LabVIEW FPGA Compiler* [5–9] the details of which (relevant to this work) are given in Section 3. One of the main contributions of this work is the state-of-the-art HLS technology that offers an automated and systematic compilation flow, which generates an optimized hardware implementation from a user's algorithm and design requirements. This methodology empowers domain experts with minimum hardware knowledge to leverage FPGA technology in exploring, prototyping, and verifying their complex domain-specific applications. As shown in Figure 2, our compilation flow takes an application diagram as well as high-level design requirements, such as clock rate and throughput, and produces an optimized implementation with resource and timing estimates. By simply modifying application parameters and design requirements, designers can quickly get new hardware implementations with updated estimates. High-level design (user) requests and estimates enable designers to easily evaluate the current model and requirements and plan further algorithmic exploration. This rapid design process paves the way for domain experts to successfully accomplish the optimized design solution with significant time and cost savings.

QC-LDPC codes or their variants (such as accumulator-based codes [10]) that can be decoded (suboptimally) using Belief Propagation (BP) are highly likely candidates for 5G systems [4]. Insightful work on high-throughput (order of Gb/s) BP-based QC-LDPC decoders is available; however, most of such works focus on an application-specific integrated circuit (ASIC) design [11, 12] which usually requires intricate customizations at the register-transfer level (RTL) and expert knowledge of very-large-scale integration (VLSI) design. A sizeable subset of the above-mentioned work caters to fully-parallel [13] or code-specific [14] architectures. From the point of view of an evolving research solution, this is not an attractive option for rapid prototyping. In the relatively less explored area of FPGA-based implementation, impressive results have recently been presented in works such as [15–17]. 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 [18]. Since our case study is based on fully automated generation of the hardware description language (HDL), we compare our results with some recent HLS-based state-of-the-art implementations [19–22] in Section 6. The main contributions of this work are as follows. In this work, we present a high-throughput FPGA-based *IEEE 802.11n* standard compliant QC-LDPC channel decoder. With the architectural technique of splitting of the node processing,

we achieve the said degree of pipelining without utilizing additional hardware resources. To demonstrate the scalability of the architecture, we present its application to a massively-parallel 2.48 Gb/s USRP-based decoder implementation (also demonstrated on the exhibit floor in the *2014 IEEE GLOBECOM* conference [23]). The final contribution is a method to rapidly prototype the experimentation of a HARQ system based on the efficient decoder architecture developed, using the *IEEE 802.16* standard compliant QC-LDPC code. The system not only comprises digital signal processing (DSP), but also finite-state machines (FSM). In spite of such *mixed* nature of data processing, *LabVIEW FPGA Compiler* was able to significantly reduce the time to explore the overall system parameter space and to optimize resource utilization for the error-rate performance achieved.

The remainder of this article is organized as follows. Section 2 provides a succinct introduction to the QC-LDPC code structure and the corresponding decoding algorithm considered for the architecture. The strategies for achieving high-throughput for the stand-alone QC-LDPC decoder are explained in Section 4. The case studies for the high-throughput decoder, its application demonstrating scalability, and the rapidly prototyped HARQ experiment are detailed in Section 5. A survey of recent state-of-the-art solutions is provided in Section 6. Section 7 concludes the article.

2. Quasi-Cyclic LDPC Codes

LDPC codes are a class of linear block codes that have been shown to achieve near-capacity performance on a broad range of channels. Invented by Gallager [3] in 1962, they are characterized by a Low-Density (sparse) Parity-Check Matrix (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 [24]. 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 $n = k + m$. An example of a Tanner graph representation (due to Tanner [25] who introduced a graphical representation) is shown in Figure 3. Here, the PCM $\mathbf{H}$ is the incidence matrix of a bipartite Tanner graph comprising two sets: the check node (CN) set of m parity-check equations and the variable node (VN) set of n variable or bit nodes; the ith CN is connected to the jth VN if $\mathbf{H}(i, j) = 1$. The column weight $d_c \ll m$ and the row weight $d_r \ll n$, where row weight and column weight are defined as the number of 1s along a row and a column, respectively. An LDPC code is called a regular code if each CN has a degree d_r and each VN has a degree d_c and is called an irregular LDPC code otherwise.

2.1. Parity-Check Matrix. The first LDPC codes by Gallager [3] are random, which 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 [26] belong to the class of structured codes that do not significantly compromise performance relative to randomly constructed LDPC codes.

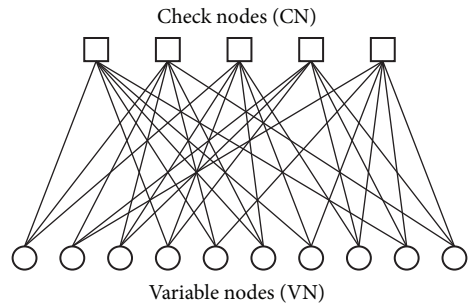


Figure 3: A Tanner graph where the variable nodes (VN), representing the code bits, are shown as circles and the check nodes (CN), representing the parity-check equations, are shown as squares. Each edge in the graph corresponds to a nonzero entry (1 for binary LDPC codes) in the PCM $\mathbf{H}$.

The construction of QC-LDPC codes relies on an $m_b \times n_b$ matrix $\mathbf{H}_b$ sometimes called the *base matrix* which comprises cyclically right-shifted identity and zero submatrices both of size $z \times z$, where, $z \in \mathbb{Z}^+$, $0 \le i_b \le (m_b - 1)$ and $0 \le j_b \le (n_b - 1)$, the shift value,

$$s = \mathbf{H}_b\left(i_b, j_b\right) \in \mathcal{S} = \{-1\} \cup \{0, \ldots, z-1\}. \tag{1}$$

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

$$s \longrightarrow \begin{cases} \mathbf{I}_s, & s \in \mathcal{S} \setminus \{-1\}, \\ \mathbf{0}, & s \in \{-1\}, \end{cases} \tag{2}$$

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 $\mathbf{H}$ comprises the submatrices $\mathbf{I}_s$ and $\mathbf{0}$, it has $m = m_b \cdot z$ rows and $n = n_b \cdot z$ columns. The *base matrix* for the *IEEE 802.11n (2012)* standard [27] with $z = 81$ is shown in Table 1.

2.2. Scaled Min-Sum Approximation Decoding. LDPC codes can be suboptimally decoded using the BP method [3, 28] on the sparse bipartite Tanner graph where the CNs and VNs communicate with each other, successively passing revised estimates of the log-likelihood ratio (LLR) associated in every decoding iteration. In this work, we have employed the efficient decoding algorithm presented in [29], with a pipelining schedule based on the row-layered decoding technique [30], detailed in Section 4.3.

Definition 1. For $1 \le i \le m$ and $1 \le j \le n$, let v_j denote the jth 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 ratio for variable node j be denoted as p_j.

Table 1: Base matrix $\mathbf{H}_b$ for $z = 81$ specified in the *IEEE 802.11n (2012)* standard used in the case study (see Section 5.1). L_1–L_{12} are the layers and B_1–B_{24} are the block columns (see Section 4.3). Valid blocks (see Section 4.3) are bold.

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**

The steps of the scaled-MSA are given below.

(1) Initialization. The a posteriori probability p_j for the VN j and the CTV messages are initialized as

$$p_j^{(0)} = \ln\left\{\frac{P\left(v_j = 0 \mid y_j\right)}{P\left(v_j = 1 \mid y_j\right)}\right\}, \quad 1 \le j \le n,$$
$$r_{ij}^{(0)} = 0, \quad 1 \le i \le m,\ 1 \le j \le n. \tag{3}$$

(2) Iterative Process. During the tth decoding iteration,

$$q_{ij}^{(t)} = p_j^{(t-1)} - r_{ij}^{(t-1)}, \tag{4}$$

$$r_{ij}^{(t)} = a \cdot \prod_{k \in \mathcal{N}(i)\setminus\{j\}} \operatorname{sign}\left(q_{ik}^{(t)}\right) \cdot \min_{k \in \mathcal{N}(i)\setminus\{j\}} \left\{\left|q_{ik}^{(t)}\right|\right\}, \tag{5}$$

$$p_j^{(t)} = q_{ij}^{(t)} + r_{ij}^{(t)}, \tag{6}$$

where $1 \le i \le m$ and $k \in \mathcal{N}(i) \setminus \{j\}$ represents the set of the VN neighbors of CN i excluding VN j and a is the scaling factor used, the rationale behind which is explained below.

(3) Decision Rule. $1 \le i \le m$,

$$\hat{v}_j = \begin{cases} 0, & p_j < 0, \\ 1, & p_j \ge 0. \end{cases} \tag{7}$$

(4) Stopping Criteria. If $\hat{\mathbf{v}}\mathbf{H}^T = 0$ or $t = t_{\max}$ (maximum number of decoding iterations), declare $\hat{\mathbf{v}}$ as the decoded codeword.

It is well known that since the MSA is an approximation of the sum-product algorithm (SPA) [3], the performance of the MSA is relatively worse than the SPA [24]. However, work such as [31] has 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 (set to 0.75) to compensate for the performance loss due to the MSA approximation.

The standard BP algorithm is based on the so-called *flooding* or *two-phase* schedule where each decoding iteration comprises 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 given in [29] however, message updates can propagate across the graph in the same decoding iteration. This provides advantages such that a single processing unit is required for both CN and VN message updates, memory storage is reduced on account of the on-the-fly computation of the VTC messages q_{ij}, and the algorithm converges faster than the standard BP flooding schedule requiring fewer decoding iterations.

3. HLS with *LabVIEW FPGA Compiler*

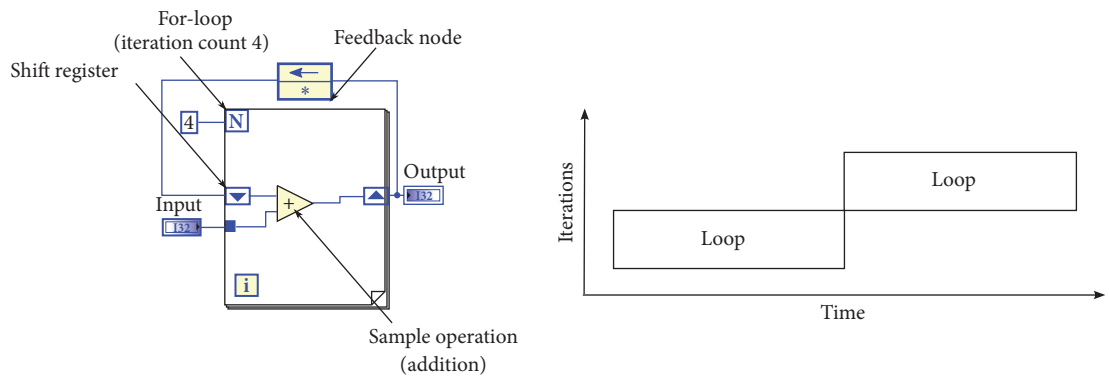
The HLS compiler in *LabVIEW CSDS* [32], namely, *LabVIEW FPGA Compiler*, aims at identifying opportunities to efficiently parallelize in the application's algorithmic description, subject to requirements set by the user. Here, we briefly describe the main techniques [5] embedded into the *LabVIEW FPGA Compiler* toolset that enable efficient high-throughput translation of the algorithm into a VHDL description.

3.1. Memory Dependency Analysis. Loop unrolling on FPGA platforms is a well-known compiler optimization used to exploit parallelism [33]. However, in the presence of execution dependencies between loop iterations, loop unrolling may not contribute to throughput improvement. An example is shown in Figure 4(a) where an execution dependency restricts parallelization of unrolled loops. Although loops have been unrolled by a factor of two as shown in Figure 4(b), the first loop copy waits until the second loop copy execution is finished. Due to the serialized loop execution, the overall performance is the same as the original loop, however at the cost of more FPGA resources used by the new loop copies.

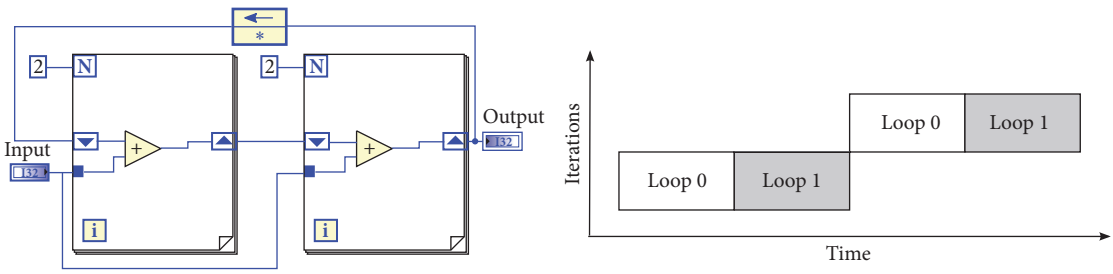
However, if unrolling is performed only when it improves throughput, a trade-off between throughput and resource consumption can be achieved in the implementation. An illustrative example is provided in Figure 5, where a feedback node defines a data dependency across consecutive diagram executions. A Read-After-Write (RAW) dependency between the current memory read operation R_i and a previous memory write operation W_{i-1} is shown in Figure 5(a). This dependency prevents the compiler from pipelining the diagram executions and becomes a bottleneck, restricting the overall throughput as shown in Figure 5(b). However, if the compiler can determine that R_i never reads a memory location that is updated by W_{i-1}, then the ith diagram execution can overlap with the $(i-1)$th execution and achieve better throughput as shown in Figure 5(c). Such an analysis is also applicable to relax WAR and WAW dependencies.

The memory access pattern analysis in *LabVIEW FPGA Compiler* mainly comprises two steps. In the first step, a periodic access pattern is determined by monitoring all the stateful nodes that contribute to each memory access pattern. In the second step, access patterns of memory accessor pairs are compared, and the pairwise worst iteration distance k is computed. This dependent iteration distance is used to create a relaxed interiteration dependency, thus allowing pipelined executions without any memory corruption.

3.2. Memory Access Traffic Relaxation. Loop unrolling may not be effective if the memory access speed cannot keep up with the data throughput request set by the user. This is particularly true for processing intensive applications like the ones studied and implemented in this work. *LabVIEW FPGA Compiler* uses the following techniques to reduce memory traffic such that the performance targets set by the user are met.

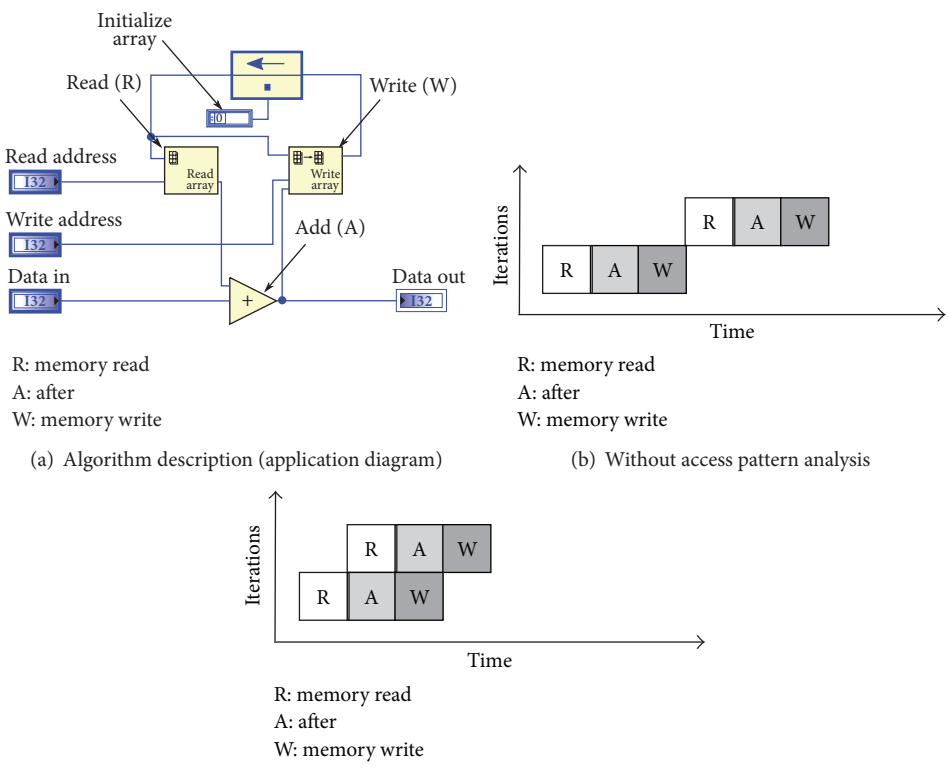


(a) Timing diagram before unrolling



(b) Timing diagram after unrolling

Figure 4: Ineffective loop unrolling. Shown on the left are representative schematics of the *LabVIEW* graphical programming virtual instruments (VI), and on the right are the corresponding timing diagrams.



(a) Algorithm description (application diagram)

(b) Without access pattern analysis

(c) With access pattern analysis

Figure 5: Throughput improvement using access pattern analysis. Shown on (a) is the representative schematic of the *LabVIEW* graphical programming virtual instrument (VI), and on (b and c) are the corresponding timing diagrams.

3.2.1. Memory Partitioning. Memory blocks on modern FPGAs typically have only two ports, one of which is generally read-only. Implementing memories with more ports can become very resource intensive and can drastically reduce the clock rate of the design. The limited amount of memory ports often causes accesses to get serialized. These serialized memory access requests often make computational cores idle, thus resulting in a reduction of the system throughput [34]. Memory partitioning is the division of the original memory block into multiple smaller memory blocks. This partitioning effectively increases FPGA physical memory access ports to allow simultaneous memory read and write operations, thus minimizing the idle time of the computational cores. Memory accessors are grouped into sets, such that accessors within one set are guaranteed to have a nonoverlapping address space with members of another set, allowing the compiler to safely partition the single memory into a memory for each set of accessors. The size of each partition is the size of the address space for that set. The original memory is divided into small partitions based on min-max address ranges of memory accessor groups, and each group is mapped to a separate partition having the matched address range.

LabVIEW FPGA Compiler statically analyzes memory access patterns in a given application diagram and automatically relaxes the memory access bottleneck without impacting the execution of the high-level algorithmic description input to it. Memory traffic is thereby reduced linearly by the partitioning number at no additional memory space cost.

3.2.2. Memory Accessor Jamming. In many applications, memory access is sequential and predictive. When multiple accesses to a memory can be computed in parallel, the values can be accessed together in one clump rather than as many separate smaller accesses. We refer to this as *memory accessor jamming*. This method creates a memory accessor group such that accessor patterns are of the form,

$$i \cdot o_p, i \cdot o_p + 1, \ldots, i \cdot o_p + o_c, \tag{8}$$

where o_p is a periodic offset, i is a loop indexer, and o_c is a constant offset that is smaller than o_p. The multiple accessors in a group are jammed into a single accessor with a wide word length. This word length is the product of the original word length and the jamming factor value. Consequently, memory access traffic is decreased by the value of the jamming factor. Jamming modifies the memory layout by increasing the word length and reducing the address range by the jamming factor, but it does not need any additional memory space. Jamming is well suited for use with loop unrolling because any in-order memory access pattern inside the loop becomes a jammable access pattern after unrolling.

All of the above techniques have been successfully employed by *LabVIEW FPGA Compiler* without any manual intervention from the user. For instance, loop unrolling is primarily employed to process algorithmic metrics described in Section 2.2 for the technique of z-fold parallelization of node metric processing as described in Section 4.2. Here, memory access analysis captures relaxed memory dependencies and achieves the reported throughput without any application-specific compiler directives. Moreover, due to the graph-based iterative decoding nature of the application considered for this work, read-write patterns that lend themselves to memory accessor jamming have been identified by the tool and successfully exploited.

The authors would like to emphasize that the algorithmic compiler *(LabVIEW FPGA Compiler)* translates the application's high-level description to VHDL. The subsequent compilation of VHDL is performed by the *Xilinx Vivado* compiler, the details of which are beyond the scope of this work.

4. Techniques for High-Throughput

To understand the high-throughput requirements for LDPC decoding, let us first define the decoding throughput T of an iterative LDPC decoder.

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 given by $T = (F_c \cdot n)/(N_i \cdot N_c)$ 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.

4.1. 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 (4)–(6) are collectively referred to as the Node Processing Unit (NPU).

Careful observation reveals that, among (4)–(6), processing the CTV messages r_{ij}, $1 \le i \le m$ and $1 \le j \le n$, is the most computationally intensive due to the calculation of the sign and the minimum value of the set of magnitudes of VTC messages q_{ik} received from VN j to CN i, where $k \in \mathcal{N}(i) \backslash \{j\}$. As the degree of CN i is d_{c_i}, the complexity of processing the minimum value (in terms of the comparisons required) is $\mathcal{O}(d_{c_i}^2)$. In a straightforward algorithmic description, 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.

To achieve linear complexity $\mathcal{O}(d_{c_i})$ for the CN message update process in our implementation, the minimum value is computed in two phases or passes. In the first (global) pass, the two smallest values for the CN are computed. These are the first and the second minimum (the smallest value in the set excluding the minimum value of the set). Subsequently, for every incident edge on the said CN, the smallest VN message that does not correspond to the considered edge is selected. In other words, if the said incident edge (for which the CN to VN message is to be sent) has the smallest value (first min), then the second smallest value (second min) obtained in the global pass is sent over this edge, else, the second smallest value (second min) is sent. This pass is called the second (local) pass. A similar approach is found in [11, 35].

Table 2: Arbitrary submatrix $\mathbf{I}_s$ in $\mathbf{H}$, $0 \le J \le n_b - 1$, illustrating the opportunity to parallelize z NPUs.

	VN_{zJ}	$\cdots$	$\mathrm{VN}_{zJ+\ell-1}$	VN_{zJ+l}	$\mathrm{VN}_{zJ+\ell+1}$	$\cdots$	$\mathrm{VN}_{z(J+1)-1}$
NPU_0	0	$\cdots$	0	1	0	$\cdots$	0
NPU_1	0	$\cdots$	0	0	1	$\cdots$	0
$\vdots$	$\vdots$						$\vdots$
NPU_{z-2}	0	$\cdots$	0	0	0	$\cdots$	0
NPU_{z-1}	0	$\cdots$	1	0	0	$\cdots$	0

In a straightforward algorithmic description, this translates to two separate for-loops in tandem: first loop executes $(d_{c_i} - 1)$ times computing the first and the second minimum for the set of VTC message values q_{ik} and the second loop executes $(d_{c_i} - 1)$ times assigning the overall minimum to each branch connecting CN i and VN k, where $1 \le i \le m$, $1 \le j \le n$, and $k \in \mathcal{N}(i) \setminus \{j\}$. Consequently, this reduces the complexity from $\mathcal{O}(d_{c_i}^2)$ to $\mathcal{O}(d_{c_i})$. Based on the functionality of the two passes, the NPU is divided into the Global NPU (GNPU) and the Local NPU (LNPU). The algorithm to accomplish this is as follows.

(1) Global Pass. The Global NPU (GNPU) processes this pass.

(i) Initialization: let ℓ denote the discrete time-steps such that $\ell \in \{0, 1, 2, \ldots, |\mathcal{N}(i) \setminus \{j\}|\}$ 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

$$f^{(0)} = s^{(0)} = \infty. \tag{9}$$

(ii) Comparison: for $1 \le i \le m$, $1 \le j \le n$, and $k(\ell) \in \mathcal{N}(i) \setminus \{j\}$, note that the ordering of the set that ℓ belongs to is induced on the set that $k(\ell)$ belongs to.

$$f^{(\ell)} = \begin{cases} |q_{ik(\ell)}|, & |q_{ik(\ell)}| \le f^{(\ell-1)}, \\ f^{(\ell-1)}, & \text{otherwise}, \end{cases} \tag{10}$$

$$s^{(\ell)} = \begin{cases} |q_{ik(\ell)}|, & f^{(\ell-1)} < |q_{ik(\ell)}| < s^{(\ell-1)}, \\ f^{(\ell-1)}, & |q_{ik(\ell)}| \le f^{(\ell-1)}, \\ s^{(\ell-1)}, & \text{otherwise}. \end{cases} \tag{11}$$

(2) Local Pass. The Local NPU (LNPU) at time $\ell \in \{1, 2, \ldots, |\mathcal{N}(i) \setminus \{j\}|\}$ determines the actual minimum value for each VN $k(\ell)$, as per the equivalence relation:

$$\min_{k(\ell) \in \mathcal{N}(i) \setminus \{j\}} \{|q_{ik(\ell)}|\} \equiv \begin{cases} f^{(\ell_{\max})}, & |q_{ik(\ell)}| \ne f^{(\ell_{\max})}, \\ s^{(\ell_{\max})}, & \text{otherwise}, \end{cases} \tag{12}$$

where $\ell_{\max} = |\mathcal{N}(i) \setminus \{j\}|$. Thus, the computation of the minimum value is accomplished in linear complexity $\mathcal{O}(d_{c_i})$. It was rightly noted by one of the reviewers that initializing the variable $f^{(0)} = \infty$ is unnecessary, resulting in a redundant iteration in (10). However, we would like to note that the implementation was done based on (10) as given in the algorithm.

4.2. z-Fold Parallelization of NPUs. The CN message computation given by (5) is repeated m times in a decoding iteration, that is, 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 2 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.

4.3. Layered Decoding. In the flooding schedule discussed in Section 2.2, *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 its own drawbacks. Firstly, it becomes quickly intractable in hardware due to the complex interconnect pattern between the nodes of the bipartite graph. Secondly, such an implementation usually restricts itself to a specific code structure. Although the efficient scaled-MSA algorithm discussed in Section 2.2 is inherently serial in nature (as the messages are propagated across the bipartite graph more than once every decoding iteration), one can process multiple nodes at the same time if the following condition is satisfied.

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.

The row-layering technique used in this work essentially relies on the condition in Fact 2 being satisfied. In terms of the PCM $\mathbf{H}$, 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 satisfy the condition specified by Fact 2, implying that $|L_u| \ne |L_{u'}|$, $\forall u, u' \in \{1, 2, \ldots, I\}$ is possible. Owing to the structure of QC-LDPC

Table 3: Illustration of message passing in row-layered decoding in a section of the PCM $\mathbf{H}_b$.

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

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 (rows corresponding to $\mathbf{I}_s$), condition in Fact 2 is always satisfied. Hence, in our work, we choose $|L_u| = |L_{u'}| = z$.

From the VN or column perspective, $|L_u| = z$, $\forall u = \{1, 2, \ldots, I\}$ implies that the columns of the PCM $\mathbf{H}$ are also divided into subsets of size z (called *block columns* from now on) given by the set $\mathscr{B} = \{B_1, B_2, \ldots, B_J\}$, $J = n/z = n_b$. The VNs belonging to a block column may participate in CN equations across several layers. We call the intersection of a layer and a block column as a *block*. 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 $\mathbf{H}_b(u', w) \neq -1$. This is observed in Table 3, where 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. For the block column B_2, for instance, 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.

The idea of parallelizing z NPUs seen in Section 4.2 can be extended to layers, where z-sized arrays of NPUs can process message updates for multiple layers, provided they are independent with respect to the block column being processed. In Section 4.4, we discuss pipelining methods that allow us to overcome layer-to-layer dependency and maximize the throughput. Before we discuss the pipelined processing of layers implemented in our decoder, in this section, we present a novel compact (thus efficient) matrix representation leading to a significant improvement in throughput. We call $\mathbf{0}$ submatrices in $\mathbf{H}$ (corresponding to a -1 in $\mathbf{H_b}$) as *invalid* blocks, since there are no edges between the corresponding CNs and VNs. The other submatrices $\mathbf{I}_s$ are called *valid* blocks. In a conventional approach to scheduling, for example, in [12], message computation is done over 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: $\boldsymbol{\beta}_I$, the block index matrix, and $\boldsymbol{\beta}_S$, the block shift matrix. $\boldsymbol{\beta}_I$ 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 $w = 0$
  for $j_b = \{1, 2, \ldots, n_b\}$
    if $\mathbf{H}_b(u, j_b) \neq -1$
      $w = w + 1$; $\boldsymbol{\beta}_I(u, w) = j_b$; $\boldsymbol{\beta}_S(u, w) = \mathbf{H}_b(u, j_b)$.

Let $\mathscr{V}_u$ denote the set of valid blocks for layer L_u, $\forall u = 1, 2, \ldots, I$.

$$\mathscr{V}_u = \{j_b : \mathbf{H}_b(u, j_b) \neq -1\}. \tag{13}$$

Let $J = \max_u |\mathscr{V}_u|$; then, $\forall w = \{1, 2, \ldots, J\}$, we define the block index matrix as

$$\boldsymbol{\beta}_I(u, w) = \begin{cases} j_b, & \mathbf{H}_b(u, j_b) \neq -1, \\ -1, & \text{otherwise.} \end{cases} \tag{14}$$

Similarly, we define $\boldsymbol{\beta}_S$ as

$$\boldsymbol{\beta}_S(u, w) = \begin{cases} \mathbf{H}_b(u, j_b), & \mathbf{H}_b(u, j_b) \neq -1, \\ -1, & \text{otherwise.} \end{cases} \tag{15}$$

The block index (shift) matrix $\boldsymbol{\beta}_I$ ($\boldsymbol{\beta}_S$) is shown in Table 4 (Table 5) for the case of the *IEEE 802.11n* rate-1/2 LDPC code. 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, $\lambda = J/n_b$.

Table 4: Block index matrix $\boldsymbol{\beta}_{\mathbf{I}}$ showing the valid blocks (bold) to be processed.

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

Table 5: Block shift matrix $\boldsymbol{\beta}_{\mathbf{S}}$ showing the right-shift values for the valid blocks to be processed.

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

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

Remark 5. In the QC-LDPC code in our case study, $|\mathscr{V}_u| = 7$ for all layers except layers L_7 and L_{12} where it is 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.

4.4. Area Efficient Pipelining Architecture. In Section 4.3, we saw how dependent layers for a block column cannot be processed in parallel. For instance, in the base matrix $\mathbf{H}_b$ in Table 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 $\boldsymbol{\beta}_{\mathbf{I}}$ shown in Table 4.

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

Proof. It 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 $\boldsymbol{\beta}_{\mathbf{I}}(u, w) = \boldsymbol{\beta}_{\mathbf{I}}(u', w)$, 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 Table 1), the condition

$$\boldsymbol{\beta}_{\mathbf{I}}(u, w) \neq \boldsymbol{\beta}_{\mathbf{I}}\left(u', w\right) \tag{16}$$

must hold $\forall u,\ u' \in \{1, 2, \ldots, I\}$. For the structure of $\boldsymbol{\beta}_{\mathbf{I}}$ shown in Table 4 (by definition of the code), it is not possible to parallelize *all* the layers. However, a degree of parallelization can be achieved by making the layers independent with respect to a block column.

To accomplish this, we rearrange the $\boldsymbol{\beta}_{\mathbf{I}}$ matrix elements from their original order with the following idea. If $\boldsymbol{\beta}_{\mathbf{I}}(u, w) = \boldsymbol{\beta}_{\mathbf{I}}(u', w)$, $u < u'$, then *stagger* the execution of $\boldsymbol{\beta}_{\mathbf{I}}(u', w)$ with respect to $\boldsymbol{\beta}_{\mathbf{I}}(u, w)$ by moving $\boldsymbol{\beta}_{\mathbf{I}}(u', w)$ to $\boldsymbol{\beta}'_{\mathbf{I}}(u', w')$,

Table 6: Rearranged block index matrix $\boldsymbol{\beta}'_I$ used for our work, showing the valid blocks (bold) to be processed.

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

$w < w'$. Table 6 shows one such rearrangement of $\boldsymbol{\beta}_{\mathbf{I}}$ (Table 4) for the QC-LDPC code for our case study. However, some dependencies still remain (shown in bold and italic in Table 6). Note that if we partition $\boldsymbol{\beta}'_{\mathbf{I}}$ into two halves, L_1 to L_6 and L_7 to L_{12}, each half satisfies Fact 2 separately. In other words, $\forall u_f,\ u'_f \in \mathscr{L}_f = \{1, 2, \ldots, 6\}$, $\forall w \in \{1, 2, \ldots, 8\}$, $\boldsymbol{\beta}_{\mathbf{I}}(u_f, w) \neq \boldsymbol{\beta}_{\mathbf{I}}(u'_f, w)$, and $\forall u_s,\ u'_s \in \mathscr{L}_s = \{7, 8, \ldots, 12\}$, $\forall w \in \{1, 2, \ldots, 8\}$, $\boldsymbol{\beta}_{\mathbf{I}}(u_s, w) \neq \boldsymbol{\beta}_{\mathbf{I}}(u'_s, w)$.

We call the set of layers $\mathscr{L}$ satisfying Fact 2 a *superlayer*. Figure 6(a) shows the block-level view of the NPU timing diagram without the pipelining of layers. As seen in Section 4.1, the GNPU and LNPU operate in tandem and in that order, implying that the LNPU has to wait for the GNPU updates to finish. The layer-level picture is depicted in Figure 7(a). This idling of the GNPU and LNPU can be avoided by introducing pipelined processing of blocks given by the following lemma.

Lemma 6. *Within a superlayer, while the LNPU processes messages for the blocks $\boldsymbol{\beta}'(u, w)$, the GNPU can process messages for the blocks $\boldsymbol{\beta}'(u + 1, w)$, $u = \{1, 2, \ldots, |\mathscr{L}| - 1\}$, and $w = \{1, 2, \ldots, J\}$.*

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

Figure 6(c) illustrates the block-level view of this 2-layer pipelining scheme. It is important to note that the splitting of the NPU process into two parts, namely, the GNPU and the LNPU (that work in tandem), is a necessary condition for Lemma 6 to hold. However, at the boundary of the superlayer, Lemma 6 does not hold and pipelining has to be restarted for the next layer as seen in the layer-level view shown in Figure 7(c). This is the classical pipelining overhead.

Definition 7. 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 are shown in Figure 7(c):

$$\eta_p^{(2)} = \frac{|\mathscr{L}|}{|\mathscr{L}| + 1}. \tag{17}$$

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

(1) Since two layers are processed in the pipeline at any given time,

$$|\mathscr{L}| \in \mathscr{F} = \{x : x \text{ is an even factor of } I\}. \tag{18}$$

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

(3) Choice of $|\mathscr{L}|$ should maximize pipelining efficiency η_p,

$$l^* = \arg\max_{|\mathscr{L}| \in \mathscr{F}} \eta_p. \tag{19}$$

In our work, $I = m_b = 12$, $\mathscr{F} = \{2, 4, 6\}$, and $l^* = \arg\max_{|\mathscr{L}| \in \mathscr{F}} \eta_p = 6$. The rearranged block index matrix $\boldsymbol{\beta}'_{\mathbf{I}}$ is shown in Table 6 and the layer-level view of the pipeline timing diagram for the same is shown in Figure 7(d).

Remark 8.

Four-Layer Pipelining. For the case of the *IEEE 802.11n (2012)* QC-LDPC code chosen for this work, the pipelining of four layers might suggest an increase in the throughput; however, this is not the case as depicted in Figure 8. Due to the need for two NPU arrays, the pipelining efficiency of this scheme is

$$\eta_p^{(4)} = \frac{\eta_p^{(2)}}{2}. \tag{20}$$

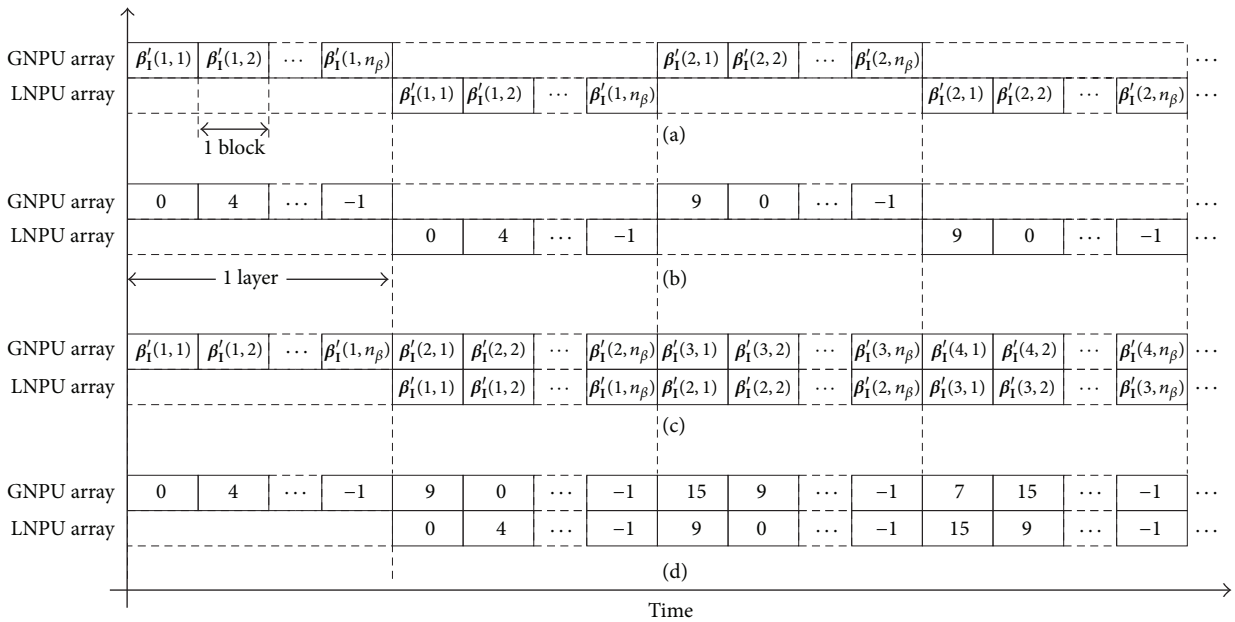


Figure 6: Block-level view of the pipeline timing diagram. (a) General case for a circulant-1 identity submatrix construction based QC-LDPC code (see Section 2) without pipelining. (b) Special case of the *IEEE 802.11n* QC-LDPC code used in this work without pipelining. (c) Pipelined processing of two layers for the general QC-LDPC code case in (a). (d) Pipelined processing of two layers for the *IEEE 802.11n* QC-LDPC code case in (b). This schedule is illustrated by the *block processing* loop in the high-level decoder architecture shown in Figure 9. Here, n_β represents the number of columns of $\boldsymbol{\beta}'_{\mathbf{I}}$.

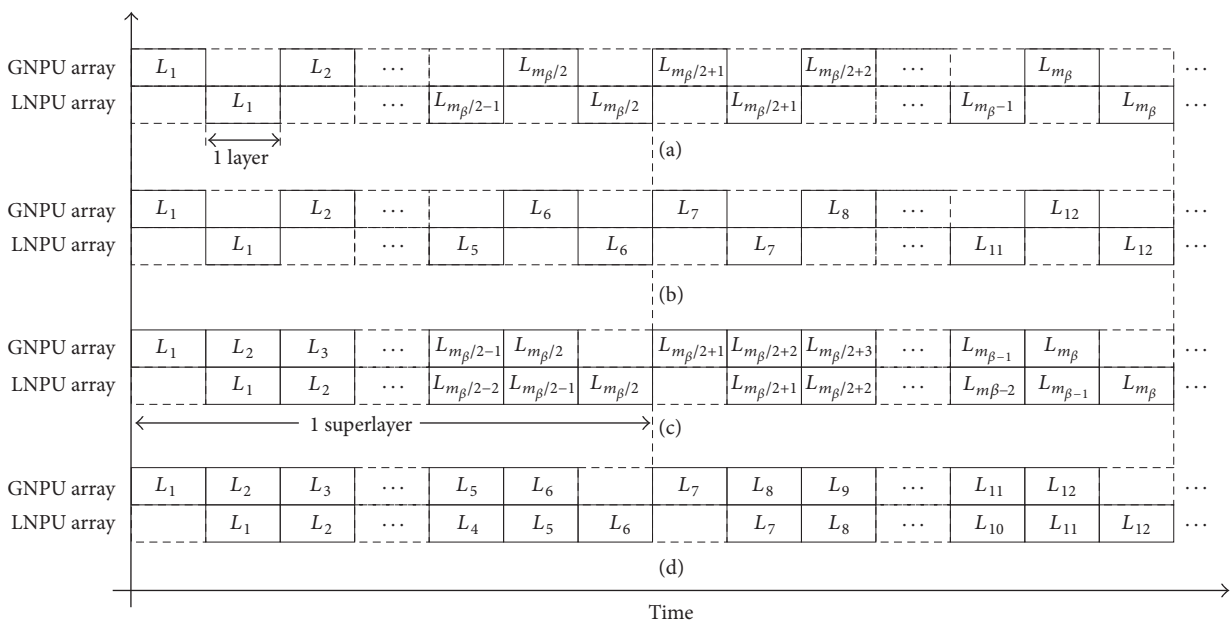


Figure 7: Layer-level view of the pipeline timing diagram. (a) General case for a circulant-1 identity submatrix construction based QC-LDPC code (see Section 2) without pipelining. (b) Special case of the *IEEE 802.11n* QC-LDPC code used in this work without pipelining. (c) Pipelined processing of two layers for the general QC-LDPC code case in (a). (d) Pipelined processing of two layers for the *IEEE 802.11n* QC-LDPC code case in (b). This schedule is illustrated by the *layer processing* loop in the high-level decoder architecture shown in Figure 9. Here, m_β represents the number of rows of $\boldsymbol{\beta}'_{\mathbf{I}}$.

Hence, we limit ourselves to pipelined processing of two layers. To achieve further gains in throughput, without loss of generality, parallel processing of multiple blocks can be performed. For details on this approach of improving throughput, the reader is referred to Appendix B. From the perspective of memory access relaxation (Section 3.2) in *LabVIEW FPGA Compiler*, the proposed 2-layer pipelining is a suitable methodology for the FPGA internal memory with

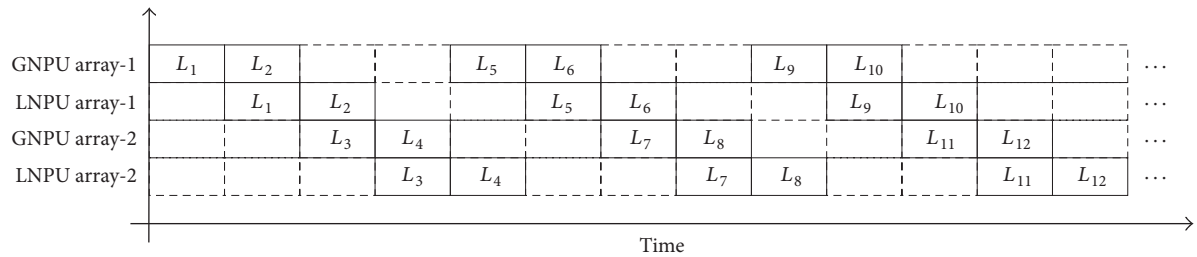


Figure 8: Layer-level view of the pipeline timing diagram for the GNPU and LNPU arrays when two NPU arrays are employed to process four layers. Due to the requirement of two NPU arrays, this method is inefficient compared to the two-layer pipelining method. Moreover, this method is not adopted for the implementation as the number of layers in a parallel run is limited by the number of ports in the shared memory.

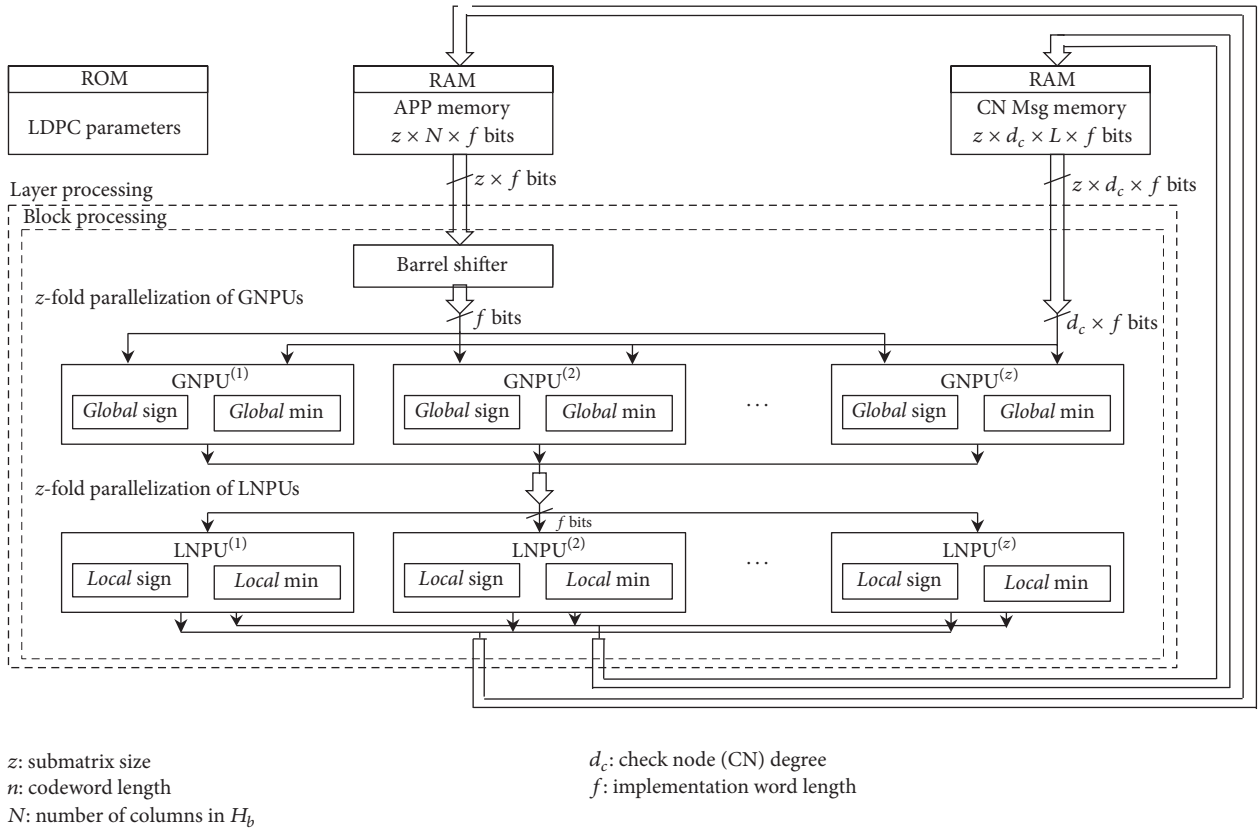


Figure 9: High-level decoder architecture showing the z-fold parallelization of the NPUs with an emphasis on the splitting of the sign and the minimum computation given in (5). Note that, other computations in (3)–(6) are not shown for simplicity here. For both the pipelined and the nonpipelined versions, processing schedule for the inner block processing loop is as per Figure 6 and that for the outer layer processing loop is as per Figure 7.

a single pair of read/write port. This is because two layers running in parallel are timely assigned to a memory read and write port. Since this approach does not have any layer execution postponed due to a resource limitation, we can achieve the theoretical maximum throughput performance. Even if pipelining more than two layers was efficient, for such a method multiple layers need to be processed in parallel. However, the number of layers in a parallel run is limited by the number of ports in the shared memory. Any layers that need processing beyond the shared memory port number would be postponed, and this would prevent us from achieving the theoretical maximum throughput. Deploying multiple decoding cores (as described in Section 5.2) is another way of improving throughput. The downside of this approach is that the memory requirement grows linearly with the number of parallel layers.

High-Level FPGA-Based Decoder Architecture. The high-level decoder architecture is shown in Figure 9. The read-only memory (ROM) holds the LDPC code parameters specified by $\boldsymbol{\beta}_l'$ and $\boldsymbol{\beta}_s'$ along with other code parameters such as the block length and the maximum number of decoding

iterations. Initially, the a posteriori probability (APP) memory is set to the channel LLR values corresponding to all the VNs as per (3). The barrel shifter operates on blocks of VNs APP values 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 in the APP block 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 specified by the Tanner graph of the code. The cyclically shifted APP memory values and the corresponding CN message values for the block in question are fed to the array of z NPUs. Here, the GNPUs compute VN messages as per (4) and the LNPUs compute CN messages as per (5). These messages are then stored back at their respective locations in the random-access memory (RAM) for processing the next block. At the time of writing this paper, we have successfully implemented two versions of the decoder.

(1) 1x. As the name suggests, only one layer is processed at a time by the NPU array; in other words, there is no pipelining of layers. The block-level and the layer-level view of the pipelining are illustrated in Figures 6(b) and 7(b), respectively.

(2) 2x. This version is based on the 2-layer pipeline processing. Pipelining is done in software at the algorithmic description level. The block-level and layer-level views of the pipelined processing are shown in Figures 6(d) and 7(d), respectively. Due to the pipelining overhead, $\eta_p^{(2)} = 6/7 = 0.86$. Comparing this to the 1x version with $\eta = 6/12 = 0.5$, the 2x version is $\eta_p^{(2)}/\eta = 1.7$ times faster than the 1x version.

5. Case Studies

The techniques for improving throughput in an efficient manner, described in Section 4, are realized on hardware using an HLS compiler. The realization is divided into three case studies, namely, an efficiently pipelined *IEEE 802.11n* standard [27] compliant QC-LDPC decoder, an extension of this decoder that provides a throughput of 2.48 Gb/s, and an HARQ experimentation system based on the *IEEE 802.16* standard [36] QC-LDPC code. Each case study is detailed in the following Sections.

5.1. IEEE 802.11n Compliant LDPC Decoder. To evaluate the proposed strategies for achieving high-throughput, we have implemented the scaled-MSA based decoder for the QC-LDPC code in the *IEEE 802.11n (2012)*. 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 is thus capable of supporting all the block lengths for the rate $R = 1/2$ code.

We represent the input LLRs from the channel and the CTV and VTC messages with 6 signed bits and 4 fractional bits. Figure 10 shows the bit-error-rate (BER) performance for the floating-point (FP) and the fixed-point (FxP) data representation with 8 decoding iterations. As expected, the fixed-point implementation suffers by about 0.5 dB compared to the floating-point version at a BER of 10^{-4}, and the gap widens for lower BER values. The decoder algorithm was described using the *LabVIEW CSDS* software. *LabVIEW FPGA Compiler* was then used to generate the very high speed integrated circuit (VHSIC) hardware description language (VHDL) code from the graphical dataflow description. 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 decoder achieves an overall throughput of 608 Mb/s at an operating frequency of 200 MHz and a latency of 5.7 μs at 4 decoding iterations with BER performance shown in Figure 10 (blue curve). Table 7 shows that the resource usage for the 2x version (almost twice as fast due to pipelining) is close to that of the 1x version. The *LabVIEW FPGA Compiler* chooses to use more flip-flops (FF) for data storage in the 1x version, while it uses more block RAM (BRAM) in the 2x version.

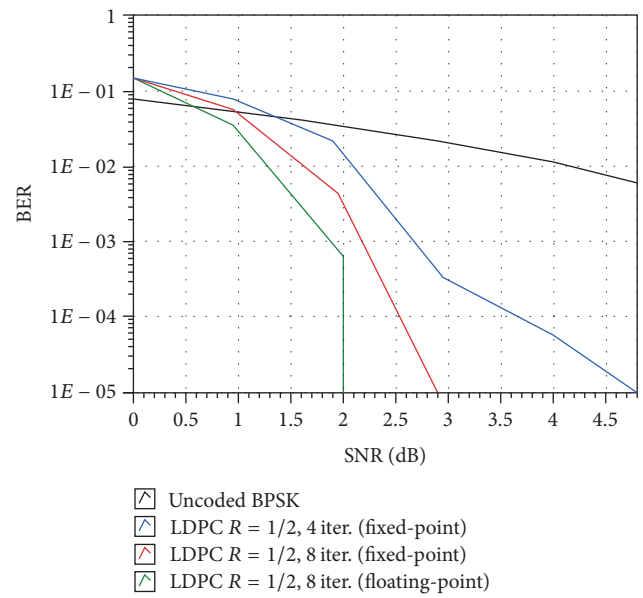


Figure 10: BER performance comparison between uncoded BPSK (rightmost), rate = 1/2 LDPC with 4 iterations using fixed-point data representation (second from right), rate = 1/2 LDPC with 8 iterations using fixed-point data representation (third from right), and rate = 1/2 LDPC with 8 iterations using floating-point data representation (leftmost).

Table 7: LDPC decoder IP FPGA resource utilization and throughput after mapping onto the Xilinx *Kintex-7 FPGA*.

	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 8: Performance and resource utilization comparison, after mapping onto the FPGA, for versions with varying number of cores of the QC-LDPC decoder implemented on the *NI USRP-2953R* containing the Xilinx *Kintex-7 (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

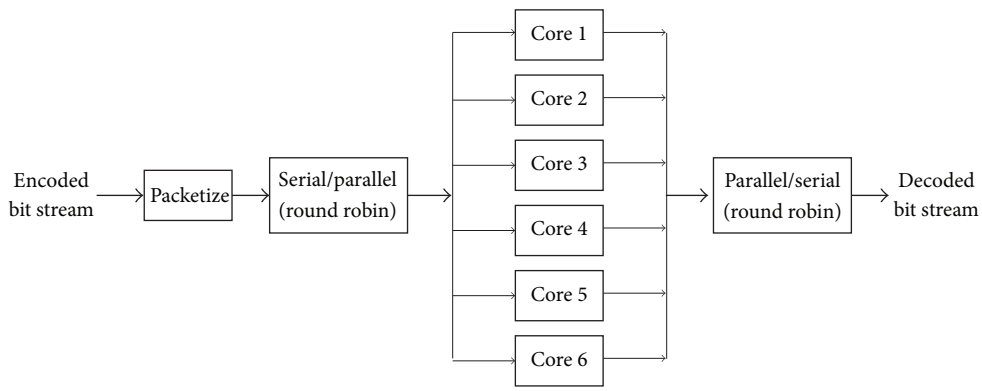


Figure 11: High-level system schematic illustrating the fixed latency, parallel processing of the decoder cores.

Remark 9. The clock rate selection in the HLS compiler generally determines pipeline stage depth of each primitive operation. For example, a higher target clock rate would result in a deeper pipeline stage. This requires more FPGA resources and a relatively longer compile time. Various *target* clock rates were tested, and the one offering the highest throughput in time with the most optimal resource utilization was chosen for the subsequent VHDL compile (e.g., 200 MHz for the compiles shown in Table 8). It is important to note that the HLS compiler provides an accurate throughput and resource estimation after it generates VHDL. This throughput and resource estimation time is short as recorded in the results tables (e.g., Table 8) as *Time to VHDL*. The user can easily find the optimal clock rate in terms of maximal throughput and optimized resource utilization.

5.2. Case Study: A 2.48 Gb/s QC-LDPC Decoder on the Xilinx Kintex-7 FPGA. On account of the scalability and reconfigurability of the decoder architecture in [37], it is possible to achieve high-throughput by employing multiple decoder cores in parallel as detailed in [38]. As shown in Figure 11, the encoded bit stream is packetized into frames of equal size and distributed for decoding in a round-robin manner to the cores operating in parallel. The main contribution of this approach is the elimination of a complicated buffering and handshake mechanism which increases the development time and adds hardware overhead. This is mainly due to

(1) fixed latency of decoding the frames across all cores,

(2) time-staggered operation of cores,

(3) tightly controlled execution of the round-robin serial-parallel-serial conversion process.

To validate the multicore decoder architecture, in this case study, we chose the *IEEE 802.11n (2012)* QC-LDPC code for which $m_b \times n_b = 12 \times 24$, $z = 27, 54$, and 81 resulting in code lengths of $n = 24 \times z = 648, 1296$, and 1944 bits, respectively, and a code rate $R = 1/2$. The decoder *core* (described in Section 5.1) was compiled for a clock rate of 200 MHz and achieves a throughput of 420 Mb/s (first column in Table 8) with pipelining as described in Section 4.4.

The multicore decoder was developed in stages. The first stage is the aforementioned pipelined decoder core to which additional cores were added incrementally as per the scheme depicted in Figure 11. We have listed the resource utilization and the throughput performance for each stage in Table 8 for a qualitative comparison.

5.3. Rapid Prototyping of Hybrid-ARQ System. Hybrid-ARQ (HARQ) is a transmission technique that combines Forward Error Correction (FEC) with ARQ. In HARQ, a suitable FEC

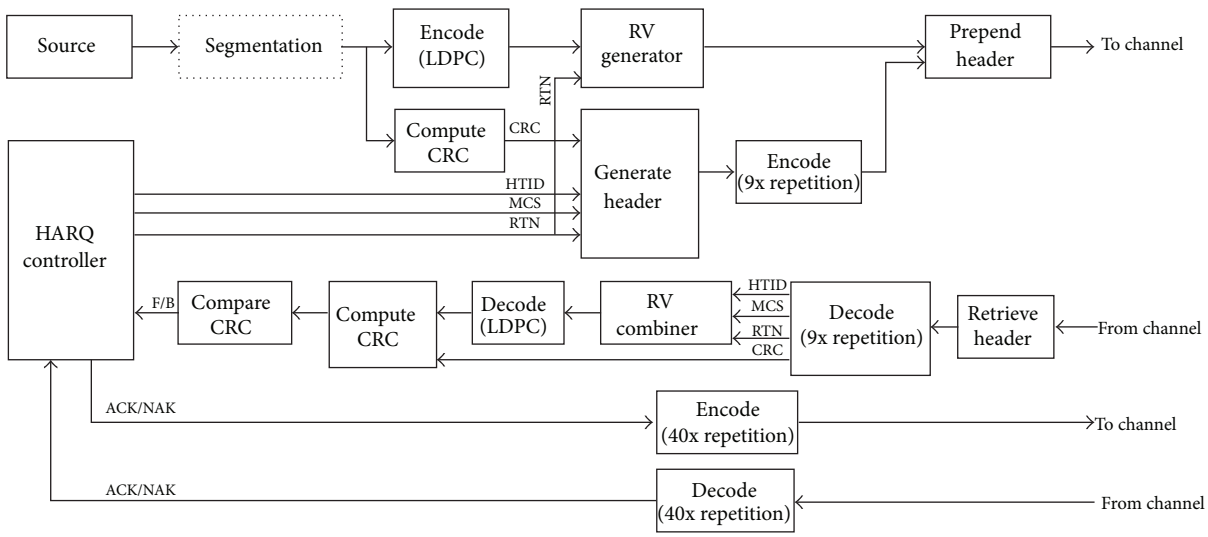


Figure 12: HARQ system schematic for one node. Overall, the system simulation uses two nodes (BS and UE).

code protects the data and error-detection code bits. In its simplest form, the FEC encoded packet—referred to as a Redundancy Version (RV) in this context—is transmitted as per the ARQ mechanism protocol. If the receiver is able to decode the data, it sends an acknowledgement (ACK) back to the transmitter. However, if it fails to recover the data, the receiver sends a negative acknowledgement (NAK) or retransmission request to the transmitter. In this scenario, the FEC simply increases the probability of successful transmission, thus reducing the average number of transmissions required in an ARQ scheme. HARQ has two modes of operation: Type-I and Type-II. In Type-I, a current retransmission is chase-combined [39] with a previously buffered (and failed) retransmission and then decoded. In Type-II HARQ, in the event of a decoding failure, additional code bits are transmitted in every subsequent retransmission. Since, in this mode, all code bits are not transmitted every retransmission, the efficiency of this scheme is higher. However, the complexity is also higher compared to Type-I.

To study the performance of the two HARQ schemes (Type-I and Type-II), we have implemented a baseband bidirectional link with two transceiver *nodes*. This can be compared to a downlink connection between a base station (BS) and user equipment (UE) with a data channel and a feedback channel. Each node is capable of running the HARQ protocol in its two modes. In our work, the BS (initiator of the transmission) operates in the *master* mode and the UE operates in the *slave mode*. A high-level description of the overall system with several subsystems is shown in Figure 12 and the media access control- (MAC-) level operation is described in Appendix A. At the initiator node, each data packet of length $k = 1152$ is encoded with an LDPC *mother* code of rate $R = 1/2$ and the Cyclic Redundancy Check (CRC) value for it is simultaneously computed. The RV generator selects bits from the encoded data to form RVs as per the code rate adaptation algorithm [40]. The header is encoded with a rate 1/9 repetition code. Finally, the RV is appended to the header and sent over the channel.

At the receiver node, header bits are decoded and the RV combiner uses the information in the header to combine the received signal values for Type-I mode or Type-II mode. CRC values from the header and the decoded data are compared to generate a feedback for the initiator node. The feedback (1-bit ACK/NAK) is coded with a rate 1/40 repetition code before sending over the channel. We assume an error-free feedback for this experiment, which is guaranteed by the rate 1/40 repetition code over the SNR range in consideration.

5.3.1. LabVIEW FPGA Compiler for Ease of Experimentation. The HARQ system comprises subsystems that can be classified into two main categories based on the nature of the processing they perform. The *bit-manipulation* subsystems—akin to digital signal processing (DSP)—follow a pattern of processing that does not change significantly on a per transmission basis. In other words, they are more or less stateless. The channel encoders and decoders are examples of this category. The *protocol-sensitive* subsystems on the other hand have to perform functions that are highly sensitive to the state of the system in a given transmission. For instance, the HARQ controller, the RV generator, and the RV combiner maintain a state [40]. With a few examples, this section highlights the ease of modification in a short time that *LabVIEW FPGA Compiler* provides across subsystems which is otherwise not possible for a purely HDL-based description.

Protocol-Sensitive Subsystem Modification. The HARQ controller is essentially a finite-state machine (FSM). For a reliable and an efficient implementation of an FSM on an FPGA, the designer needs to take care of issues such as clock and input signal timing, state encoding scheme, and the choice of the coding style [41]. Modification to the MAC-level protocol directly affects the FSM in our work. For details on the MAC-level operation of the HARQ protocol, the reader is referred to Appendix A. For instance, during experimentation, the frame structure is likely to undergo modifications. Any modification to the frame structure

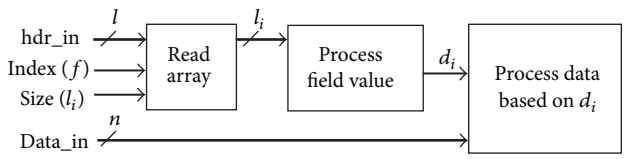


Figure 13: Realization of an example of *protocol-sensitive* subsystem using HLS.

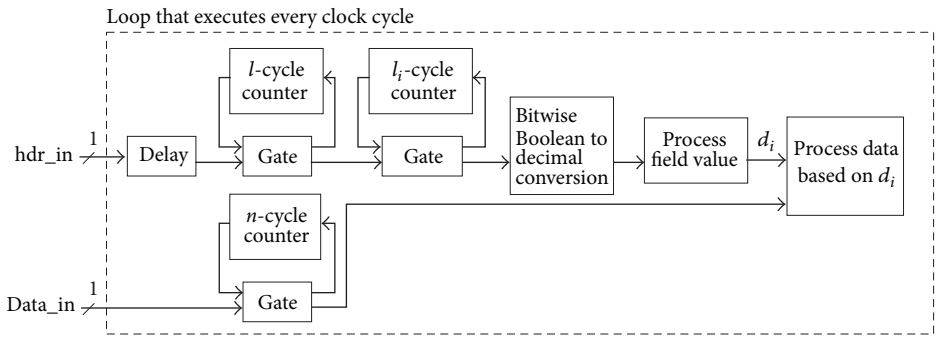


Figure 14: RTL block diagram: realization of an example of *protocol-sensitive* subsystem using HDL.

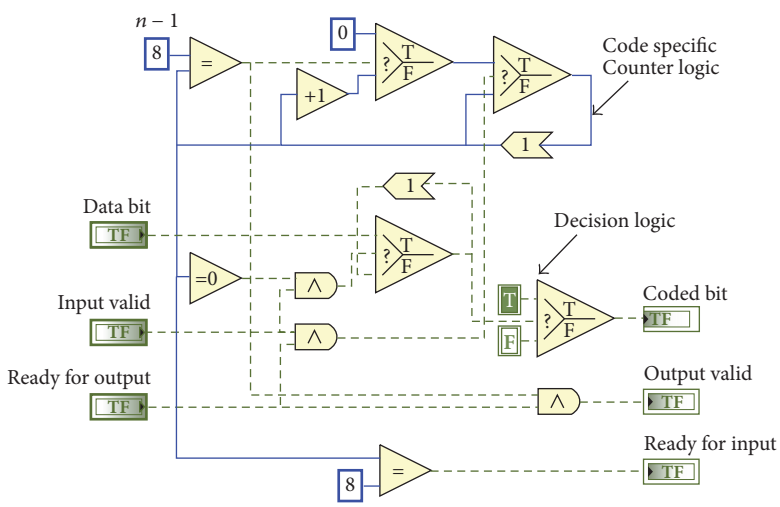


Figure 15: Schematic depiction of the description of a 9x repetition encoder in *LabVIEW FPGA* (subsequently compiled to *VHDL*). Note that the logic required is specific to the codeword size n.

affects nearly all subsystems. One such example is illustrated in Figures 13 and 14 where the ith field (of length l_i bits) of the header is read to process output d_i which is further used to process the data. The description in Figure 13 is agnostic to the modification at the HDL level and the designer can implement a change without HDL domain expertise, whereas in Figure 14 one can see that the description is specific to the subsystem in which the header is being used. Modifying the length of a field, for instance, requires modification of counter logic and adjustment of the delay value. This needs to be repeated for all subsystems that are affected by this change. In contrast, *LabVIEW FPGA Compiler* automatically generates the counter logic and delay values in Figure 14 by propagating the field-length values into the algorithm in Figure 13, allowing the same algorithmic description to be reused for different values of the field-length.

Bit-Manipulation Subsystem Modification. The channel coding subsystems, namely, the LDPC and repetition encoders and decoders, are at the core of the HARQ system. *LabVIEW FPGA Compiler* also eases the implementation of *bit-manipulation* subsystems like these. For example, the 9x repetition encoder description in *LabVIEW FPGA* is shown in Figure 15. Comparing this to the algorithmic description shown in Figure 16, it is evident that modification without much time and effort is facilitated by *LabVIEW FPGA Compiler*. It is important to emphasize here that *LabVIEW FPGA* provides a high-level abstraction to VHDL. However, it is not the same as the *algorithmic description* that we refer to, throughout this brief. This is because *LabVIEW FPGA* is a lower-level description language relative to the algorithmic description that is input to *LabVIEW FPGA Compiler*.

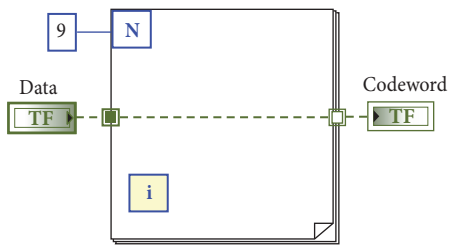


Figure 16: Schematic depiction of the description of a 9x repetition encoder using HLS using just a *for-loop* without any logic specific to the code.

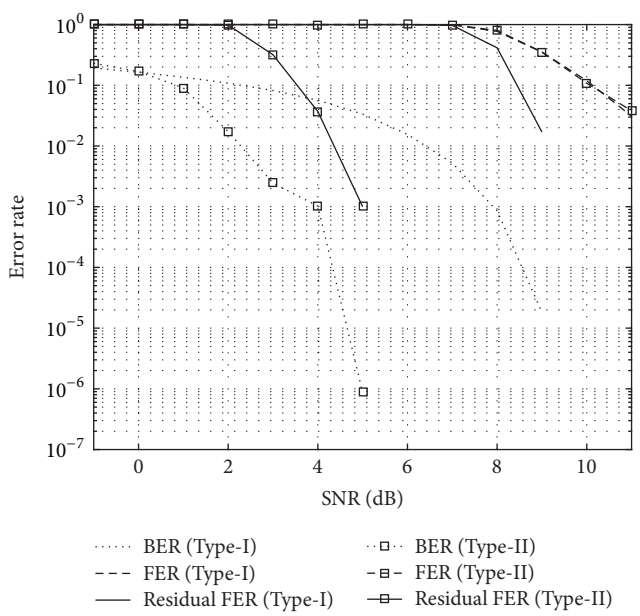


Figure 17: FER performance of Type-I and Type-II schemes. Note that the FER of Type-I and Type-II overlap as expected.

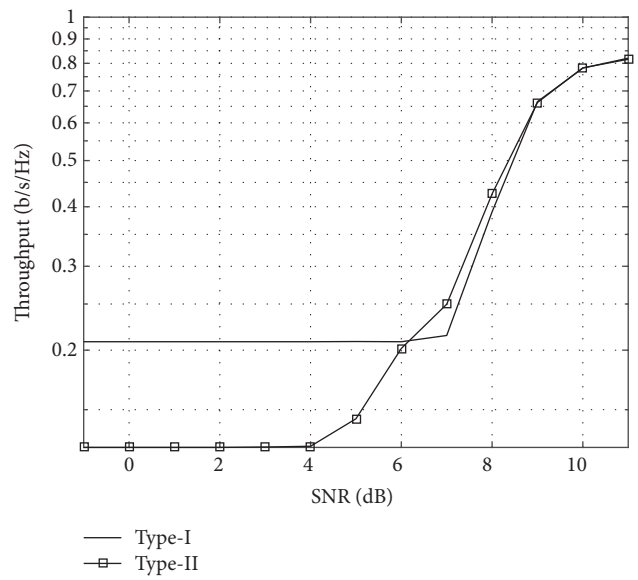


Figure 18: Throughput performance of Type-I and Type-II schemes.

Table 9: Performance and resource utilization, after mapping onto the FPGA, for the HARQ system (that supports both Type-I and Type-II mode of operation) on the *NI USRP-2953R* containing the *Xilinx Kintex-7 (410t)* FPGA.

	Utilization
Clock rate (MHz)	80
Time to generate VHDL (min)	5
Total slice (%)	54
LUT (%)	32
FF (%)	19
DSP (%)	12
BRAM (%)	30

5.3.2. Results. The HARQ system has been implemented on the *Xilinx Kintex-7* series of FPGAs and the algorithmic description was input using *LabVIEW CSDS*. We chose these sets of tools as the FPGA is available in the *NI USRP 2943R* series used for real-world prototyping of our research. At the time of writing this paper, the system performance has been evaluated for the *IEEE 802.16 (2012)* [36] set of QC-LDPC codes. We would like to emphasize here that owing to the ease of modification, we can, in short development cycles, replace the channel codes with other code structures being researched such as the one described in [42]. The error-rate performance for 1k frames of codeword size of $n = 2304$ is shown in Figure 17. The residual Frame Error-Rate (FER) accounts for the errors that the HARQ protocol failed to correct, whereas the FER accounts for errors that happen without the use of the HARQ protocol. The data throughput of the system, defined as R/RTN, and the throughput averaged over the frames per SNR point are plotted in Figure 18. As expected, the performance of the system is improved with HARQ at the cost of a decrease in the throughput. The FPGA resource utilization for the same is given in Table 9.

Scalable Simulation Speedup. Each time any change in the system is made, there is a need to evaluate the performance of the system. This is especially true for testing code structures under research. Error-rate performance in excess of 10^8 bits is required to observe phenomena such as the error-floor of a code [24]. This makes time-efficient simulations not only a luxury but a necessity. In our implementation, while developing a real-world prototype we also get the benefit of a 4x speedup in simulation time using a decoder without pipelining. We measured the execution time for 10k frames over 40 SNR values. We used the *IEEE 802.16 (2012)* specified (2304, 1152) QC-LDPC code, with a 1/9 and a 1/40 repetition code for the header and the feedback, respectively. The decoder was set to perform 4 decoding iterations.

On a host machine, a *Dell Precision T3600* 3.6 GHz *Quad Core Xeon (i7)* with a 16 GB RAM, it took about 4.28 min, whereas on our FPGA testbed it took about 1.02 min resulting in a 4x speedup with a one time *time-to-compile* of approximately 45 min. While the time-to-compile seems significant,

Table 10: Comparative survey of the state-of-the-art. Note that, while there are multiple implementation case studies in [19, 21], we only list here those cases which are the closest in terms of the QC-LDPC code used in our case study, namely the *IEEE 802.11n (WiFi)* (1944, 972) QC-LDPC code with $z = 81$. *Development time (wherever reported) quantifies the programming effort required. This measure of the programming effort has been defined in [20] and is adopted here to facilitate an unambiguous comparison. In our work, the development effort is of the order of a few days of programming effort, once the algorithm to be implemented is finalized; including the total compile time which is of the order of tens of minutes.

Work →	Andrade et al. [19]	Pratas et al. [20]	Andrade et al. [21]	Scheiber et al. [22]	This work
HLS Technology	*Altera OpenCL*	*Maxeler MaxCompiler*	*Altera OpenCL*	*Xilinx Vivado HLS*	*National Instruments LabVIEW FPGA Compiler*
Standard	*IEEE WiMAX*	*ETSI DVB-S2*	*IEEE WiFi*	*IEEE WiFi*	*IEEE WiFi*
LDPC Parameters (n, k, z)	(1152, 576, 48)	(64800, 32400, —)	(1944, 972, 81)	(648, 324, 27)	(1944, 972, 81)
BP Decoding Schedule	flooding	flooding	flooding	layered	serial and layered
Throughput (Mb/s)	103.9	540	21	13.4	608
Decoding Iterations	10	10	10	3	4
Developement Time*	n.a.	~weeks	n.a.	n.a.	~days
FPGA Device	*Altera Stratix 5 D5*	*Xilinx Virtex-5 LX330T*	*Altera Stratix 5 D5*	*Xilinx Spartan-6 LX150T*	*Xilinx Kintex-7 K410T*
Fixed-point Precision (total bits)	8	n.a.	n.a.	n.a.	10
Clock Rate (MHz)	222.6	150	157	122	200
LUT (%)	42.9	n.a.	41	3	8.2
FF (%)	42.3	n.a.	36	2	5.3
BRAM (%)	75.3	n.a.	67	20.9	6.4
DSP (%)	3.8	n.a.	0	0	5.2

n.a.: not available (i.e. not reported in the cited work).

once compiled, for several trials with larger datasets (orders of magnitude larger than experimental value specified above), this time becomes insignificant.

6. A Comparative Survey of State of the Art

A survey of the state of the art for channel code architectures and their implementation using HLS technology reveals that insightful work on the topic has been done. In this section, we list some of the notable contemporary works. While there are a myriad of LDPC architecture designs implemented on the FPGA platform, here we restrict ourselves to a subset of those works that utilize HLS technology. In this section, we list some of the notable contemporary works that fall into this category.

The performance of an implementation depends on a host of factors such as the vendor specific device(s) with its associated HLS technology and the type of channel code in consideration. Thus, the intent of the authors is not to claim an all-encompassing performance comparison demonstrating gains or losses with respect to each other, but to provide the reader with a qualitative survey of the state of the art. Table 10 lists works [19–22] based on the settings from each that are chosen according to the proximity of their relevance to our work.

7. Conclusion

We use an HLS compiler that without expert-level hardware domain knowledge enables us to reliably prototype our research in a short amount of time. With techniques such as timing estimation, pipelining, loop unrolling, and memory inference from arrays, *LabVIEW FPGA Compiler* compiles untimed dataflow algorithms written with loops, arrays, and feedback into VHDL descriptions that achieve a high clock rate and high-throughput. The employed HLS technology significantly reduced the time to explore the system parameter space and optimized it in terms of the error-rate performance and the resource utilization. We propose techniques to achieve a high-throughput FPGA architecture for a QC-LDPC code. The strategies are validated by implementing a standard compliant QC-LDPC decoder on an FPGA. The decoder architecture is scaled up to achieve another highly-parallel realization that has a throughput of 2.48 Gb/s. The HLS compilation process is used to rapidly prototype a HARQ experimentation system using LDPC codes that not only comprises bit-manipulation subsystems but also protocol-sensitive subsystems. This facilitated the error-rate performance measurement of the system over large, realistic data sets at a 4x greater speed than the conventional CPU-based experimentation. Finally, the use of HLS and reconfigurable hardware platforms holds the promise of realizing the architecture suited for the evolving research requirements of 5G wireless technology.

Appendix

A. MAC-Level HARQ Operation

Here, we briefly discuss the operation of the protocol-sensitive subsystems (Section 5.3.1) in the HARQ system for the interested reader. Without loss of generality, for N RVs

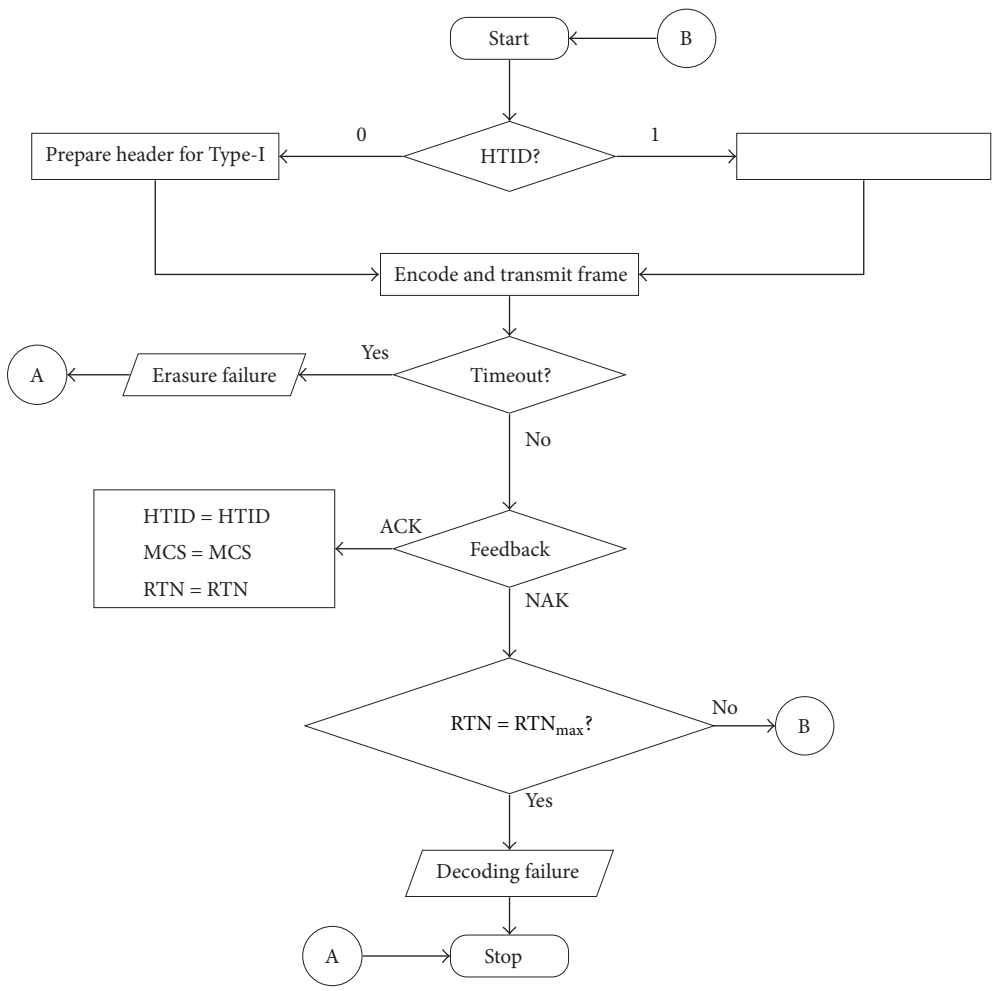


Figure 19: Flowchart showing the *master* mode of operation for the HARQ control algorithm. HTID: HARQ type identifier, MCS: Modulation and Coding Scheme, RTN: ReTransmission Number, and CRC: Cyclic Redundancy Check.

and the maximum number of retransmissions set to $\mathrm{RTN}_{\max}$, the MAC-level operation of the HARQ protocol is shown in Figure 19 for the master mode and Figure 20 for slave mode. For Type-I scheme of HARQ, the RV generator does not puncture any bits and sends the whole *mother* codeword every transmit instance, whereas, in the Type-II scheme, it generates RVs as detailed in [40].

At the receiver, for the Type-I scheme of HARQ, the ith transmit instance performs $\mathbf{B}_i = \mathbf{B}_{i-1} + \mathbf{RV}_i$, where $\mathbf{B}$ denotes the buffer contents and $|\mathbf{B}| = n$ with $\mathbf{B}_0 = \mathbf{0}$. For the Type-II scheme, the RV combiner performs $\mathbf{B}[\sigma(j)] = \mathbf{RV}_i(j)$, where $0 \le i \le (N-1)$, $0 \le j \le (|\mathbf{RV}|-1)$, and $\sigma(j)$ is the position of the jth code bit in the mother codeword determined by the puncturing method.

B. Parallelizing Block Columns

In Section 4, it was concluded that increasing the number of layers to more than two layers in the pipeline provides diminishing returns in efficiency of the pipelining scheme. Here, we present a technique for a multifold increase in throughput by processing multiple blocks in a particular layer. We would like to note that this technique has not been implemented in any of the case studies provided in this article. To gain further throughput improvement, in this approach, we take advantage of the following fact. There is no message exchange across the blocks of a particular layer. In other words, message exchange (and hence dependency) happens only in the vertical direction in $\boldsymbol{\beta}'_{\mathbf{I}}$, where, $\forall u \in \{1, 2, \ldots, I\}$ and $\forall w, w' \in \{1, 2, \ldots, J\}$,

$$\boldsymbol{\beta}'_{\mathbf{I}}(u, w) \neq \boldsymbol{\beta}'_{\mathbf{I}}\left(u, w'\right). \tag{B.1}$$

The matrix $\boldsymbol{\beta}'_{\mathbf{I}}$ is defined in Section 4.4. In the pipelined version, the NPU array processes each block (within a layer) sequentially as shown in Figure 21. However, if we split the blocks into two sets and process each set independent of the other (requiring 2 NPU arrays), we can double the throughput. Owing to this fact, we call this version as the 4x version. Similarly, by employing 4 NPU arrays, we have the 8x version

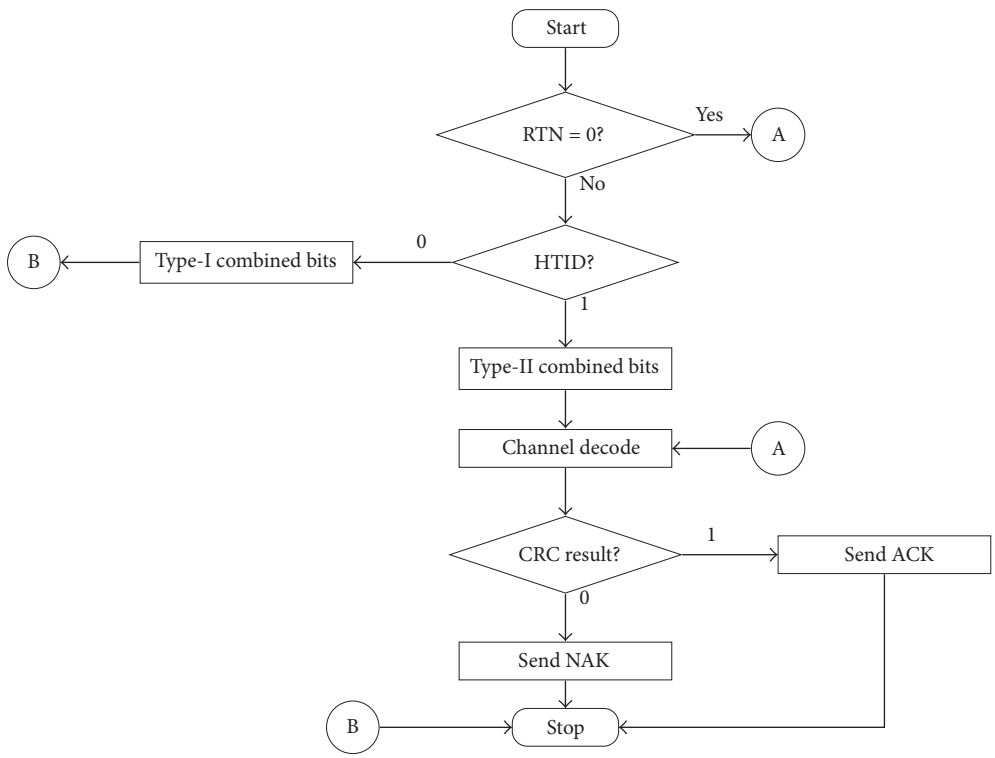


Figure 20: Flowchart showing the *slave* mode of operation for the HARQ control algorithm. HTID: HARQ type identifier, MCS: Modulation and Coding Scheme, RTN: ReTransmission Number, and CRC: Cyclic Redundancy Check.

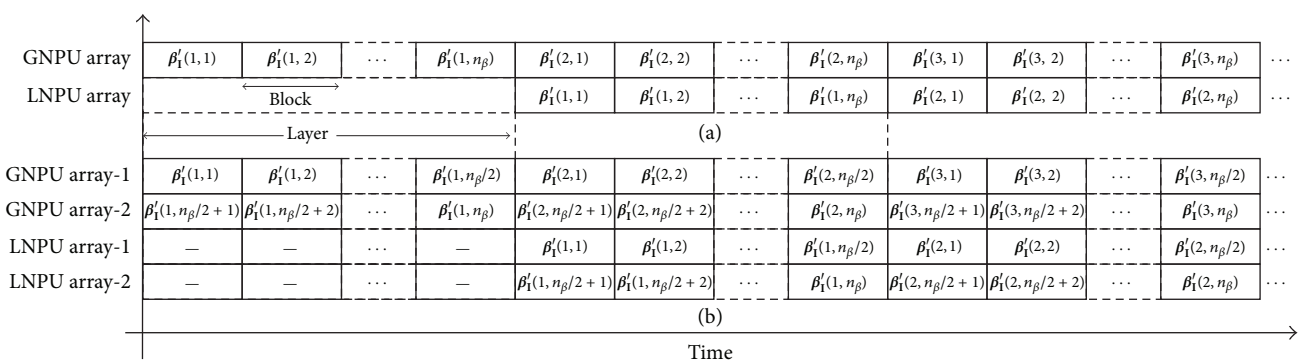


Figure 21: Pipeline timing diagram from the block processing perspective for (a) the 2x version and (b) the 4x version of the massively-parallel decoder architecture. Note that the ordering of $\boldsymbol{\beta}_{\mathbf{I}}'(u, w)$ blocks, $\forall u \in \{1, 2, \ldots, I\}$ and $\forall w \in \{1, 2, \ldots, J\}$ shown here, is not unique, owing to the independence of the blocks as shown earlier. Here, n_β represents the number of columns of $\boldsymbol{\beta}_{\mathbf{I}}'$.

and finally, if we employ 8 NPU arrays, we have the 16x version, thus increasing throughput gradually at each stage.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The authors would like to thank the Department of Electrical and Computer Engineering, Rutgers University, NJ, USA, and the National Instruments Corporation, Austin, TX, USA, for their continual support for this research work.

References

[1] B. Raaf, W. Zirwas, K.-J. Friederichs et al., "Vision for Beyond 4G broadband radio systems," in *Proceedings of the IEEE 22nd International Symposium on Personal, Indoor and Mobile Radio Communications, (PIMRC '11)*, pp. 2369–2373, IEEE, Toronto, Canada, September 2011.

[2] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and encoding: turbo-codes," in

Proceedings of the IEEE International Conference on Communications, pp. 1064–1070, Geneve, Switzerland, May 1993.

[3] R. G. Gallager, "Low-Density Parity-Check Codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[4] "3GPP RAN WG1," in *3rd Generation Partnership Project (3GPP)*, 2016, http://www.3gpp.org/specifications-groups/ran-plenary/ran1-radio-layer-1/home.

[5] H. Kee, S. Mhaske, D. Uliana et al., "Rapid and high-level constraint-driven prototyping using lab VIEW FPGA," in *Proceedings of 2014 IEEE Global Conference on Signal and Information Processing, GlobalSIP 2014*, pp. 45–49, USA, December 2014.

[6] H. Kee, T. Ly, N. Petersen, J. Washington, H. Yi, and D. Blasig, "Compile time execut ion," *U.S. Patent 9 081 583*, 2015.

[7] T. Riche, N. Petersen, H. Kee et al., "Convergence analysis of program variables," *U.S. Patent 9 189 215*, 2015.

[8] H. Kee, H. Yi, T. Ly et al., "Correlation analysis of program structures," U.S. Patent 9 489 181, 2016.

[9] T. Ly, S. Mhaske, H. Kee, A. Arnesen, D. Uliana, and N. Petersen, "Self-addressing memory," U.S. Patent 9 569 119, 2017.

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

[11] Y. Sun and J. R. Cavallaro, "VLSI architecture for layered decoding of QC-LDPC codes with high circulant weight," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 10, pp. 1960–1964, 2013.

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

[13] N. Onizawa, T. Hanyu, and V. C. Gaudet, "Design of high-throughput fully parallel LDPC decoders based on wire partitioning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 3, pp. 482–489, 2010.

[14] T. Mohsenin, D. N. Truong, and B. M. 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, 2010.

[15] A. Balatsoukas-Stimming and A. Dollas, "FPGA-based design and implementation of a multi-GBPS LDPC decoder," in *Proceedings of 22nd International Conference on Field Programmable Logic and Applications, FPL 2012*, pp. 262–269, nor, August 2012.

[16] V. A. Chandrasetty and S. M. Aziz, "FPGA implementation of high performance ldpc decoder using modified 2-bit Min-Sum algorithm," in *Proceedings of 2nd International Conference on Computer Research and Development, ICCRD 2010*, pp. 881–885, mys, May 2010.

[17] R. Zarubica, S. G. Wilson, and E. Hall, "Multi-Gbps FPGA-based Low Density Parity Check (LDPC) decoder design," in *Proceedings of 50th Annual IEEE Global Telecommunications Conference, GLOBECOM 2007*, pp. 548–552, usa, November 2007.

[18] P. Schläfer, C. Weis, N. Wehn, and M. Alles, "Design space of flexible multigigabit LDPC decoders," *VLSI Design*, vol. 2012, Article ID 942893, 2012.

[19] J. Andrade, G. Falcao, and V. Silva, "Flexible design of wide-pipeline-based WiMAX QC-LDPC decoder architectures on FPGAs using high-level synthesis," *Electronics Letters*, vol. 50, no. 11, pp. 839-840, 2014.

[20] F. Pratas, J. Andrade, G. Falcao, V. Silva, and L. Sousa, "Open the Gates: Using High-level Synthesis towards programmable LDPC decoders on FPGAs," in *Proceedings of 2013 1st IEEE Global Conference on Signal and Information Processing, GlobalSIP 2013*, pp. 1274–1277, usa, December 2013.

[21] J. Andrade, F. Pratas, G. Falcao, V. Silva, and L. Sousa, "Combining flexibility with low power: Dataflow and wide-pipeline LDPC decoding engines in the Gbit/s era," in *Proceedings of 25th IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2014*, pp. 264–269, che, June 2014.

[22] E. Scheiber, G. H. Bruck, and P. Jung, "mplementation of an LDPC decoder for IEEE 802.11n using Vivado TM high-level synthesis," in *Proceedings of int. Conf. Electron., Signal Process. and Commun. Syst.*, pp. 45–48, 2013.

[23] H. Kee, D. Uliana, A. Arnesen et al., "A 2.06Gb/s LDPC decoder (exhibit floor demonstration)," in *Proceedings of IEEE Global Commun. Conf.*, 2014, https://www.youtube.com/watch?v=o58keq-eP1A.

[24] D. Costello and S. Lin, *Error Control Coding*, Pearson, 2004.

[25] R. M. Tanner, "A recursive approach to low complexity codes," *Institute of Electrical and Electronics Engineers. Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, 1981.

[26] L. Chen, J. Xu, I. Djurdjevic, and S. Lin, "Near-Shannon-limit quasi-cyclic low-density parity-check codes," *IEEE Transactions on Communications*, vol. 52, no. 7, pp. 1038–1042, 2004.

[27] "EEE 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," in *IEEE P802.11-REVmb/D12*, pp. 1–2910, 2011.

[28] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *Institute of Electrical and Electronics Engineers. Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[29] E. Sharon, S. Litsyn, and J. Goldberger, "Efficient serial message-passing schedules for LDPC decoding," *Institute of Electrical and Electronics Engineers. Transactions on Information Theory*, vol. 53, no. 11, pp. 4076–4091, 2007.

[30] M. M. Mansour and N. R. Shanbhag, "High-throughput LDPC decoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 976–996, 2003.

[31] J. Chen and M. Fossorier, "Near optimum universal belief propagation based decoding of LDPC codes and extension to turbo decoding," in *IEEE Int. Symp. Inf. Theory*, p. 189, June 2001.

[32] National Instruments Corp., *LabVIEW Communications System Design Suite Overview*, 2014, http://www.ni.com/white-paper/52502/en/.

[33] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1995.

[34] Q. Liu, T. Todman, and W. Luk, "Combining optimizations in automated low power design," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pp. 1791–1796, March 2010.

[35] K. K. Gunnam, G. S. Choi, M. B. Yeary, and M. Atiquzzaman, "VLSI architectures for layered decoding for irregular LDPC codes of WiMax," in *Proceedings of 2007 IEEE International Conference on Communications, ICC'07*, pp. 4542–4547, gbr, June 2007.

[36] "IEEE standard for wireless MAN-advanced air interface for broadband wireless access systems," IEEE Std 802.16.1-2012, 2012.

[37] S. Mhaske, H. Kee, T. Ly, A. Aziz, and P. Spasojevic, "High-throughput FPGA-based QC-LDPC decoder architecture," in *Proceedings of 82nd IEEE Vehicular Technology Conference, VTC Fall 2015*, usa, September 2015.

[38] S. Mhaske, D. Uliana, H. Kee, T. Ly, A. Aziz, and P. Spasojevic, "A 2.48Gb/s FPGA-based QC-LDPC decoder: An algorithmic compiler implementation," in *Proceedings of 36th IEEE Sarnoff Symposium, Sarnoff 2015*, pp. 88–93, usa, September 2015.

[39] D. Chase, "Code combining—a maximum-likelihood decoding approach for combining and arbitrary number of noisy packets," *IEEE Transactions on Communications*, vol. 33, no. 5, pp. 385–393, 1985.

[40] S. Mhaske, H. Kee, T. Ly, and P. Spasojevic, "FPGA-accelerated simulation of a hybrid-ARQ system using high level synthesis," in *Proceedings of 2016 IEEE 37th Sarnoff Symposium*, pp. 19–21, Newark, NJ, USA, September 2016.

[41] N. I. Rafla and B. L. Davis, "A study of finite state machine coding styles for implementation in FPGAs," in *Proceedings of 2006 49th Midwest Symposium on Circuits and Systems, MWSCAS'06*, pp. 337–341, pri, August 2007.

[42] B. Young, S. Mhaske, and P. Spasojevic, "Rate compatible IRA codes using row splitting for 5G wireless," in *Proceedings of 2015 49th Annual Conference on Information Sciences and Systems, CISS 2015*, usa, March 2015.