Reconfigurable Architecture and Automated Design Flow for Rapid FPGA-based LDPC Code Emulation

Haoran Li, Youn Sung Park, Zhengya Zhang
Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor
lihaoran@umich.edu, parkyoun@umich.edu, zhengya@eecs.umich.edu

ABSTRACT
Multitude of design freedoms of LDPC codes and practical decoders require fast simulations. FPGA emulation is attractive but inaccessible due to its design complexity. We propose a library and script based approach to automate the construction of FPGA emulations. Code parameters and design parameters are programmed either during run time or by script in design time. We demonstrate the architecture and design flow using the LDPC codes for the latest wireless communication standards: each emulation model was auto-constructed within one minute and the peak emulation throughput reached 3.8 Gb/s on a BEE3 platform.

Categories and Subject Descriptors
C.3 [**Special-purpose and application-based systems**]: Signal processing systems

General Terms
Design, Experimentation, Performance

Keywords
LDPC, decoder emulation, decoder architecture

1. INTRODUCTION
Low-density parity-check (LDPC) codes are capacity-approaching codes that can perform very close to the Shannon limit when decoded using the iterative belief propagation algorithm [1], [2]. Over the last few years, we have seen LDPC codes entering a range of important applications, from wireline [3], wireless [4]-[7], satellite [8], optical communications [9] to magnetic storage [10], to improve reliability and spectral efficiency. However, practical performance of LDPC codes can be far from their theoretical limit for two reasons: (1) a practical code's block length is limited to hundreds to thousands of bits to meet latency and complexity constraints; (2) practical decoder implementations introduce non-idealities, such as finite word length and fixed-point quantization effects [11]. It is therefore critically important to evaluate code constructions and practical decoder implementations for each new application that is brought in consideration.

Software-based code and decoder simulation are common in practice. A typical simulation setup shown in Fig. 1 consists of an encoder to produce codewords (or memory that stores known codewords), a modulator that translates bits to real values for transmission, a channel model that generates noise to corrupt the transmitted values, and a decoder that runs the belief propagation algorithm to recover the binary codeword from the real values

FPGA'12, February 22–24, 2012, Monterey, California, USA.

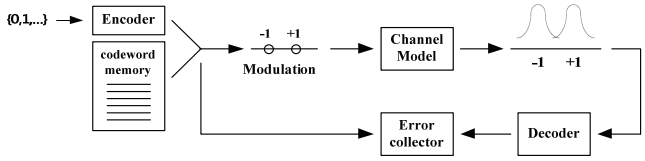
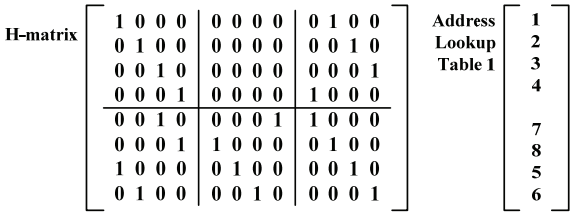


Fig. 1. Code and decoder simulation setup.

received. The decoded word is compared with the transmitted codeword to determine if an error (frame error) has occurred and, if it has, the number of bits that are wrong (bit error). Decoding errors are measured in frame error rate (FER) and bit error rate (BER). When the channel condition is poor, i.e., at low signal-to-noise ratio (SNR), decoding errors occur frequently (high FER and BER), shortening the simulation time. At high SNR, decoding errors occur infrequently (low FER and BER) and the simulation time is longer. Low FER and BER simulation is the bottleneck in code and decoder simulations.

As new generations of applications push for a higher throughput and reliability, the required FER and BER are also extended lower. For example, 10-gigabit Ethernet requires a BER of 10^{-12} or better [3]. Simulating complex decoders for these systems to extremely low BER is a challenge, as it often takes weeks or months to run a belief propagation decoder to reach a BER of 10^{-12} on a high-performance microprocessor. Recently, field-programmable gate arrays (FPGA) have been proposed to accelerate the simulations, showing three orders of magnitude speedup or more [11]-[14]. Despite the impressive speedup, FPGA emulation has not gained wide-spread use. Designing FPGA emulation is not as easy as writing C code. It requires extensive effort in creating hardware architecture and running FPGA synthesis. The barrier renders emulation inaccessible to the vast coding theory and applications community who would otherwise benefit the most in code construction and system evaluation.

In this paper, we address the challenges in creating LDPC code and decoder emulation by creating an automated design flow based on a reconfigurable hardware decoder architecture. The design flow is built upon a decoder library that consists of modules parameterized by code parameters and design parameters. Given a new LDPC code, the design flow instantiates processing elements and constructs a highly parallelized LDPC decoder. We experimented with the LDPC codes for IEEE 802.16e (WiMAX) [4], IEEE 802.11n (Wi-Fi) [5], IEEE 802.15c (wireless personal area network) [6], and IEEE 802.11ad (high-throughput wireless) [7], with block lengths ranging from 576 bits to 2,304 bits and code rates from 1/2 to 5/6. In all cases, the design flow completed decoder construction under one minute, followed by FPGA synthesis that took two hours or less. The resulting decoders operated at real-time or nearly real-time, delivering a throughput up to 3.8 Gb/s on a BEE3 platform, allowing us to reach a BER of 10^{-11} in one hour and below 10^{-12} in one day. We demonstrated the capability of the emulation platform in evaluating the functional performance of various codes. With added network interface and publically available design flow and library, the proposed FPGA



(a) (b)

Fig. 2. An LDPC code example: (a) **H**-matrix, and (b) address lookup table for the first block column.

emulation design flow will contribute to both the theoretical and practical coding research.

2. BACKGROUND

LDPC codes are linear block codes. Each LDPC code is defined by a parity-check matrix **H** of size $m \times n$, where n is the block length and m is the number of parity checks. Almost all the latest applications have adopted LDPC codes whose **H** matrix is constructed using m_b rows and n_b columns of $z \times z$ identity matrix, its cyclic shifts, or zero matrix [3]-[7]. A simple example is given in Fig. 2, where an 8×12 **H** matrix is constructed using 2 rows and 3 columns of 4×4 submatrices and each submatrix is an identity matrix, its cyclic shift, or zero matrix. We surveyed the LDPC codes in the latest communication standards and summarized their **H** matrix structures in Table I. Each **H** matrix has a fixed n_b and often a variable m_b to control the number of parity checks (or code rate) for different channel environments. In poor channel conditions, a high m_b (low code rate) is used to introduce more redundancy for a stronger protection. The submatrix size z controls the code block length: a longer block length offers better protection at the cost of a longer latency and a higher decoding complexity.

The belief propagation decoding of LDPC codes is briefly described as follows. To begin, the received real value for each bit is used to initialize the bit's *prior* likelihood [2]. The subsequent steps are carried out iteratively in a procedure following the **H** matrix [2]. In the horizontal half iteration, we go through each row of the **H** matrix and read the prior likelihoods of the bits that participate in the parity check described by the row, followed by computing an update, known as the *extrinsic*, indicating the likelihood of each bit given the likelihoods from all other bits participating in this parity check. The horizontal half iteration is completed in m horizontal steps. In the vertical half iteration, we go through each column of the **H** matrix and read all the extrinsics corresponding to the parity checks that the bit is part of to compute an updated likelihood, known as the *posterior*. A hard decision is made based on the posterior. The vertical half iteration is completed in n vertical steps. For the second and following iterations, the horizontal half iteration is carried out in the same way as the first iteration, but instead of prior likelihoods, we use modified posterior likelihoods for the computation. More iterations improve the reliability of each bit. If hard decisions of all bits satisfy all the parity-check equations, decoding converges. For a full mathematical description of the belief propagation algorithm, we refer readers to [1], [2]. The algorithm works remarkably well in practice, and usually converges in a small number of iterations.

3. RECONFIGURABLE EMULATION

The belief propagation decoder is the most complex block of a decoder emulation platform. Many high-performance architectures have been introduced for individual codes, but they have to be customized to be useful for other codes. We design an entirely reconfigurable architecture that is applicable to all the codes defined in Section 2.

TABLE I
STRUCTURE OF LDPC CODES USED IN COMMUNICATION SYSTEMS

Standard	z	n_b	m_b	n	m
IEEE 802.11ad	42	16	3,4,6,8	672	126 to 336
IEEE 802.15c	21	32	4,8,16	672	84 to 336
IEEE 802.11n	27,54,81	24	4,6,8,12	648 to 1944	108 to 972
IEEE 802.16e	24,28,…,96	24	4,6,8,12	576 to 2304	96 to 1152
IEEE 802.11an	64	32	6	2048	384

TABLE II
BASIC ARCHITECTURES OF LDPC DECODERS

	Submatrix-parallel	Column-parallel	Row-parallel
Processing elements	z	n_b	m_b
Decoding time per iteration	$n_b \times m_b$	$z \times m_b$	$z \times n_b$

Referring to Table I, the codes are parameterized by z, n_b, m_b, implying three natural ways of parallelizing the decoder: submatrix-parallel, column-parallel, or row-parallel. Consider a submatrix-parallel architecture, where z processing elements (PE) completes z horizontal (or vertical) steps concurrently, thus the decoding time per iteration is proportional to $n_b \times m_b$. A row-parallel architecture requires m_b processing elements for a decoding time of $z \times n_b$. Since the parameters z and m_b are variable for some applications, the submatrix-parallel and row-parallel architectures require pre-allocation of the maximum number of PEs and run-time reconfiguration. Note that the parameter n_b is fixed for each application, so the column-parallel decoder architecture supports all codes for a given application without requiring any over allocation of PEs. The decoding time however varies with m_b. The three basic architectures are listed in Table II for comparison. Additional architectures can be created by parallelizing or serializing the three basic architectures or mixing them. More parallel architectures demand more hardware resources. To achieve the maximum throughput on a given FPGA platform, we can create multiple decoders to run parallel emulations.

3.1 Emulation System Design

We choose the column-parallel architecture for the code and decoder emulation platform shown in Fig. 3, where n_b PEs are allocated and connected to a parity-check node for the horizontal step computation, the output of which is sent to each PE. The vertical step is completed within the PE. The PE datapath is entirely data-driven. The controller only needs to generate the address counter for each PE to access the correct data and write to the correct location. The address sequence is determined by the **H** matrix and stored in an address lookup table. An example of the address lookup table is shown in Fig. 2.

An input generator consists of a Gaussian noise generator and a set of valid codewords stored in a memory. Gaussian noise is added to codewords to emulate the effect of an additive white Gaussian noise channel (AWGN). The channel SNR is adjusted by noise variance.

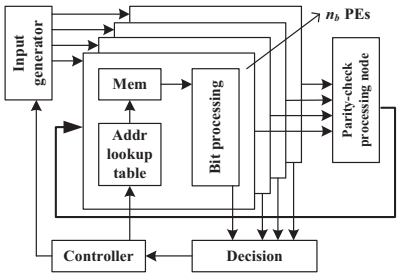


Fig. 3. Column-parallel decoder architecture.

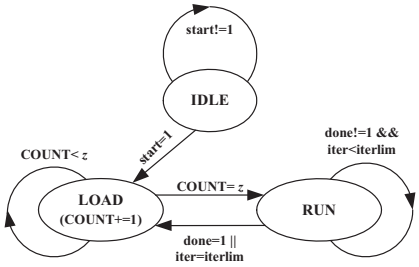


Fig. 4. Emulation control state machine.

A controller orchestrates the emulation. Its simplified state machine is shown in Fig. 4. System starts up in IDLE state. Upon receiving a "start" signal, it transitions to LOAD state, where n_b PEs load inputs from the input generator concurrently. As each PE is assigned to a block column of z bits, the loading will take z clock cycles. A persistent state variable COUNT keeps track of the loading state. COUNT increments by 1 each clock cycle until it reaches z, when the system transitions to RUN state.

The belief propagation algorithm operates by the **H** matrix row-by row for the horizontal half iteration, and then column-by-column for the vertical half iteration. Here we merge the vertical step with the horizontal step by performing vertical steps following each horizontal step. The interleaved processing lengthens the pipeline of each horizontal step, but completes one decoding iteration in approximately m clock cycles.

Towards the end of RUN state when the posterior likelihood of each bit is finalized, the controller enables the decision block to make hard decisions. If decoding converges or the iteration limit is reached, a "done" signal is generated to trigger the transition to LOAD state for loading another input vector; if decoding fails to converge when the iteration limit is reached, a frame error and number of bit errors are recorded before moving to LOAD state; if decoding fails while the iteration limit is not yet reached, it will remain in RUN state by starting another decoding iteration.

3.2 Design- and Run-Time Reconfiguration

The emulation system is entirely parameterized. Given an LDPC code, the number of block columns, n_b, determines the number PEs in the system. The submatrix size z and the number of block rows m_b determine the control schedule, including number of loading cycles, address counter, memory write enable, and decision enable. The **H** matrix structure, i.e., the locations of '1' entries in the **H** matrix of each block column, decides the address lookup table entries. The PE and parity-check node complexities are also determined by these parameters: the depth of the extrinsic memory is $z \times m_b$, the depth of the posterior memory is z, the parity-check node implements a n_b:1 adder tree (for the sum-product algorithm [2]) or a n_b:1 compare-select tree (for the min-sum algorithm [15]).

The parameterized emulation system enables convenient reconfiguration. The number of PE blocks, extrinsic and posterior memory depth, parity-check node topology are design-time reconfigurable or limited run-time reconfigurable by over allocation and selective enabling. Control constants and address lookup table are run-time reconfigurable. Hence it is possible to design one IEEE 802.16e-compatible LDPC decoder to be reconfigured for all 19 LDPC codes specified by the standard [4] by setting control constants and address lookup tables.

We consider additional parameters that are important in code and decoder designs: word length and quantization of prior, extrinsic and posterior likelihoods, limit on the number of decoding iterations, channel SNR, and algorithm control knobs (such as the offset in the min-sum algorithm [15]). Word length and quantization are design-time reconfigurable, but it is expensive to change in run time. Decoding iteration limit, channel SNR and some algorithm controls can be easily reconfigured in run time.

To sum up, many parameters are in consideration for code and decoder designs and the interplays among the parameters are of great interest. To speed up these evaluations, we need a reconfigurable architecture to minimize the number of redesigns, such that one design can be reused for many different evaluations. However, many parameters cannot be made run-time reconfigurable easily, which necessitates redesigns. A fast and automated design flow will greatly facilitate this effort.

4. DESIGN FLOW

We propose a design flow based on the BEEcube Platform Studio (BPS) targeting BEEcube BEE3 multi-FPGA platform [16]. The steps of the design flow are illustrated in Fig. 5. The first step is to establish a Simulink design library that consists of the modules that make up an emulation system: PE, parity-check node, noise generator, decision block, and controller. The modules are designed using Xilinx blockset to be readily synthesized. Design parameters including memory size, word length, and quantization are coded as parameters in the modules. The small number of modules are quick to design and easily reusable.

In the second step, a Matlab script is used to perform four tasks: (1) initialize code structure parameters, z, n_b, and m_b, and design parameters, word length and quantization, that are being referenced in the design library; (2) parse the given LDPC code to build address lookup table for each PE; (3) instantiate n_b PEs, a parity-check node, an input vector generator, a decision block, a controller and connect these modules into a complete decoder in Simulink; and (4) create an interface wrapper using BPS blockset to provide configuration registers for run-time reconfigurable parameters: control schedule, decoding iteration limit, channel SNR, and algorithm knobs, as well as output registers that capture BER and FER. This step is fully automated and can be completed in well under 1 minute.

The third step involves BPS compilation, which takes less than 2 hours based on all the experiments we carried out. The resulting bit file is programmed on the BEE3 FPGA platform for emulation experiments.

The proposed design flow integrates with the BPS flow and simplifies the design process. The library and script will be made available online. We take advantage of Virtex 5 FPGAs' Ethernet functionality by connecting them to the network, each with its own IP address. Remote users can control emulations through function calls in Matlab or C code. We expect this work to contribute to the coding research community and encourage collaborations among researchers.

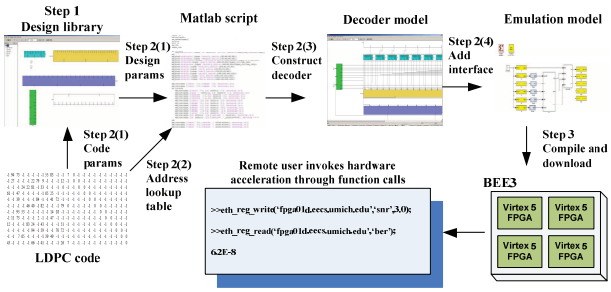


Fig. 5 BEE3-based automated design flow.

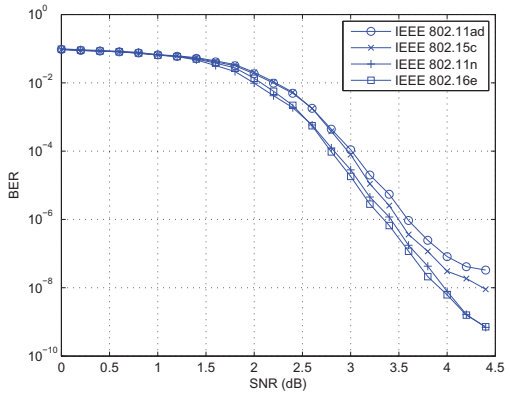


Fig. 6. BER plots of 1/2 LDPC codes used in four communication standards: IEEE 802.11ad, IEEE 802.15c, IEEE 802.11n, IEEE 802.16e. The results are obtained using 10 decoding iterations and 5-bit Q5.0 fixed-point quantization using offset min-sum algorithm.

5. RESULTS

We designed a common design library and applied the automated design flow to LDPC decoders for four different applications, IEEE 802.11ad, IEEE 802.15c, IEEE 802.11n, and IEEE 802.16e. Excluding the initial effort in making the common design library, the flows including compilation completed within two hours. For a better utilization of the target Xilinx Virtex-5 XC5VLX155T device, we created multiple decoder copies, each with its own input generator, decision block and controller, on a single FPGA to run parallel emulations. The device utilization details are listed in Table III based on 5-bit fixed-point quantization and offset min-sum algorithm [15]. The utilization includes fixed overhead created by BPS to handle interfaces and controls. Note that the reported level of parallelism was not limited by the resources available on FPGA, but by the runtime memory of the 32-bit operation system on which the compilation was done.

The designs in Table III have all been successfully compiled and they meet a minimum clock frequency of 100 MHz and deliver throughputs from 380 to 950 Mb/s (in decoding the 1/2 rate code of the longest block length in each standard). With four such FPGAs available on the BEE3 platform [16], we can achieve an emulation throughput up to 3.8 Gb/s, allowing us to reach a BER of 10^{-11} in one hour and below 10^{-12} in one day (with at least 100 bit errors observed for statistical significance).

Fig. 6 shows the performance of four 1/2-rate LDPC codes used in the four communication standards. The 10^{-9} BER point was captured within one minute. Evaluation of code construction and decoder design can be made quickly, e.g., selection of submatrices, code block length, code rate, word length and quantization, iteration limit, algorithm tuning and error floor studies. The rapid FPGA-based code emulation will contribute to future coding theory and applications research.

TABLE III
DEVICE UTILIZATION AND THROUGHPUT OF LDPC EMULATION PLATFORMS (BASED ON XILINX VIRTEX-5 XC5VLX155T)

	802.11ad	802.15c	802.11n	802.16e
# of parallel emulators	6	3	4	4
Slice registers	22,323 (23%)	19,971 (21%)	22,221 (23%)	22,689 (23%)
Slice LUTs	35,723 (37%)	31,497 (32%)	38,883 (40%)	39,214 (40%)
Occupied slices	12,568 (52%)	11,819 (49%)	13,565 (56%)	13,219 (54%)
BRAMs	131 (62%)	130 (61%)	130 (61%)	178 (84%)
Throughput (at 100MHz)	950 Mb/s	380 Mb/s	695 Mb/s	710 Mb/s

6. ACKNOWLEDGMENTS

This work is supported by National Science Foundation under grant CCF-1054270. We acknowledge the donations made by BEEcube, Xilinx, Intel, and the technical advice by Dr. C. Chang and Dr. K. Camera from BEEcube.

7. REFERENCES

[1] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.
[2] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inf. Theory*, vol. 45, pp. 399-431, Mar. 1999.
[3] *IEEE Standard for Local and Metropolitan Area Networks – Specific Requirements Part 3*, Sep. 2006, IEEE Std. 802.3an.
[4] *IEEE Standard for Local and Metropolitan Area Networks Part 16*, Feb. 2006, IEEE Std. 802.16e.
[5] *IEEE Standard for Local and Metropolitan Area Networks – Specific Requirements Part 11*, Feb. 2007, IEEE Std. 802.11n.
[6] *IEEE Standard for Local and Metropolitan Area Networks – Specific Requirements Part 15.3*, Oct. 2009, IEEE Std. 802.15c.
[7] IEEE P802.11 – Task Group AD. (2010). *PHY/MAC Complete Proposal Specification* [Online]. Available: http://www.ieee802.org/11/Reports/tgad_update.htm.
[8] *ETSI Standard TR 102 376 V1.1.1: Digital Video Broadcasting (DVB)*, ETSI Std. TR 102 376, Feb. 2005.
[9] F. Chang, K. Onohara, and T. Mizuochi, "Forward error correction for 100 G Transport Networks," *IEEE Communications Mag.*, vol. 48, no. 3, pp. S48-S55, Mar. 2010.
[10] A. Kavcic and A. Patapoutian, "The read channel," *Proc. IEEE*, vol. 96, no. 11, pp. 1761-1774, Nov. 2008.
[11] Z. Zhang, L. Dolecek, B. Nikolic, V. Anantharam, and M. J. Wainwright, "Design of LDPC decoders for improved low error rate performance: quantization and algorithm choices," *IEEE Trans. Communications*, vol. 57, no. 11, pp. 3258-3268, Nov. 2009.
[12] ———, "Investigation of error floors of structured low-density parity-check codes by hardware emulation," in *IEEE Global Communications Conf.*, Nov. 2006.
[13] Y. Cai, S. Jeon, K. Mai, and B.V.K.V. Kumar, "Highly parallel FPGA emulation for LDPC error floor characterization in perpendicular magnetic recording channel," *IEEE Trans. Magnetics*, vol. 45, no. 10, pp. 3761-3764, Oct. 2009.
[14] X. Chen, J. Kang, S. Lin, and V. Akella, "Accelerating FPGA-based emulation of quasi-cyclic LDPC codes with vector processing," in *Conf. Design, Automation and Test in Europe*, Mar. 2009, pp. 1530-1535.
[15] J. Chen, A. Dholakia, El Eleftheriou, M. P. C. Fossorier, and X. Hu, "Reduced-complexity decoding of LDPC codes," *IEEE Trans. Communications*, vol. 53, no. 8, pp. 1288-1299, Aug. 2005.
[16] BEEcube. (2011). *BEEcube Products* [Online]. Available: http://www.beecube.com/product.