# FPGA-BASED DESIGN AND IMPLEMENTATION OF A MULTI-GBPS LDPC DECODER

*Alexios Balatsoukas-Stimming and Apostolos Dollas*

Electronic and Computer Engineering Department
Technical University of Crete
73100 Chania, Greece
E-mail: alex@telecom.tuc.gr, dollas@mhl.tuc.gr

## ABSTRACT

We design a very high speed LDPC code decoder architecture for (3,6)-regular codes by employing hybrid quantization, pipelining, and FPGA-specific optimizations. Our pipelined architecture fully addresses the decoder's significant I/O requirements, even when an early termination circuit is employed. The proposed decoder can achieve a throughput of up to 16.9 Gbps at an Eb/N0 of 3.5 dB using a code of length 1152, running at a clock speed of 153 MHz and performing a maximum of 10 decoding iterations, thus outperforming the state of the art by a significant margin. This design was fully implemented and tested on a Xilinx Virtex 5 XC5VLX110 FPGA. We also present an alternative, low-complexity design, which is able to achieve a throughput of up to 21.6 Gbps by sacrifing $0.75$ dB in terms of Eb/N0.

## 1. INTRODUCTION

Low Density Parity-Check (LDPC) codes are a class of capacity approaching channel codes that was invented in 1962 [1]. They have received considerable attention after their recent rediscovery [2]. LDPC codes have been adopted in many present and future wired and wireless standards, such as IEEE 802.3an (10 Gbps Ethernet), IEEE 802.3ba (40/100 Gbps Ethernet), IEEE 802.11n (Wi-Fi), IEEE 802.16e (Wi-Max), DVB-S2 (digital video), and others. Even though irregular LDPC codes have been shown to be capacity approaching over various channels [3], [4], most of these standards still use regular LDPC codes, which also exhibit excellent performance, because they are simpler to implement. LDPC codes are very attractive from a hardware perspective due to the high level of parallelism inherent in their decoding algorithms.

Various LDPC decoder architectures have been proposed. The simplest is the fully parallel approach [5]–[7], where each variable and check node of the graph is transferred to hardware. This approach can provide very high decoding throughput, but it also requires a large amount of hardware resources. Partially parallel decoders [8]–[10] use a fixed and relatively small number of variable and check node processors in order to do some of the processing in parallel. They achieve lower throughputs than fully parallel architectures, but consume significantly less hardware resources. The slowest, but cheapest in terms of hardware resources, is the bit serial [11] approach. A very useful overview is presented in [12]. We are interested in implementing a very high throughput LDPC decoder, so we follow the fully parallel approach.

The main contributions of this work are the use of a hybrid quantization scheme and the introduction of pipelining, which reduces path delays and completely masks the I/O delay. Our architecture is tailored for FPGAs by appropriate problem sizing and FPGA-specific optimizations. We have implemented a decoder design which has a $16\%$ higher throughput and a $0.5$ dB gain in terms of Eb/N0 when compared with the state of the art [7], while using the same amount of logic. When using a slightly larger amount of logic, our decoder can achieve a respectable $34\%$ improvement in throughput. Our alternative, low-complexity design sacrifices $0.75$ dB in terms of Eb/N0 in order to achieve a $71\%$ higher throughput when compared with [7], while using significantly lower amounts of logic.

This paper is organized as follows. In Section 2, we provide some theoretical background on LDPC codes and two iterative decoding algorithms. In Section 3, we explore the effect of message quantization on decoding performance and resource utilization. In Section 4, we present the proposed decoder architecture. In Section 5, we discuss its performance. Section 6 concludes the paper.

Alexios Balatsoukas-Stimming is supported by the Alexander S. Onassis Public Benefit Foundation under scholarship G ZG 028/2010-2011.

This work is partly funded by Robust & Safe Mobile Co-operative Autonomous Systems research project (R3-COP, project number: 101022), funded within the ARTEMIS Joint Technology Initiative as a Joint Undertaking project between the European Commission, the member states and ARTEMIS Industrial Association (ARTEMISIA) and partly funded by Increasing EU citizen security by utilising innovative intelligent signal processing systems for euro-coin validation and metal quality testing research project (SAFEMETAL, project id : 262558), implemented within the Seventh Framework Programme and financed by Community Funds.

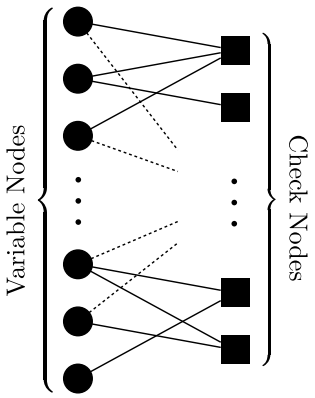



**Fig. 1**. Example of a Tanner graph.

## 2. BACKGROUND

### 2.1. LDPC Codes

An LPDC code $\mathcal{C}$ is defined as the nullspace of an $m \times n$ binary sparse parity-check matrix $\mathbf{H}$, i.e.:

$$\mathcal{C} = \{\mathbf{c} \in \{0,1\}^n : \mathbf{Hc} = \mathbf{0}\}, \tag{1}$$

where additions are performed over GF(2). In other words, the code imposes $m$ even parity constraints on the $n$ codeword bits. These parity constraints are used to recover a codeword that has been corrupted by noise.

LDPC codes can be represented by Tanner graphs [13]. These graphs contain two types of nodes, namely, variable nodes and check nodes. Variable nodes represent codeword bits and check nodes represent even parity constraints on these codeword bits. An edge between variable node $i$ and check node $j$ exists if and only if variable node $i$ participates in parity-check equation $j$. An example of a Tanner graph is presented in Fig. 1. An LDPC code is called $(d_v, d_c)$-regular if all variable nodes have degree $d_v$ and all check nodes have degree $d_c$. In our design, we employed a $(3, 6)$-regular code, partly because it is the best regular LDPC code of rate $0.5$ [3], and also for fair comparison with previous work which has used the same code.

### 2.2. Channel Model

Transmission takes place over a binary memoryless Additive White Gaussian Noise (AWGN) channel:

$$y_i = x_i + n_i, \quad n_i \sim \mathcal{N}(0, \sigma^2), \tag{2}$$

where $x_i$ denotes the $i$-th position of the modulated codeword $\mathbf{x}$, $y_i$ is the corresponding noisy observation, and $\sigma^2$ is the noise variance. We employ Binary Phase Shift Keying (BPSK) modulation, so that:

$$\mathbf{x} = 1 - 2\mathbf{c}. \tag{3}$$

With slight modifications, the decoding algorithm discussed in the sequel can be used with any memoryless channel model.

### 2.3. Decoding of LDPC Codes

The bit-wise Maximum A Posteriori (MAP) decoding rule is usually approximated by the Belief Propagation (BP) [14] algorithm, which proceeds by rounds and exchanges messages between the variable and check nodes. These messages are used at variable nodes to make hard decisions, denoted $\hat{c}_i$, for the codeword bits. Decoding halts when a valid codeword has been decoded (i.e. all constraints are satisfied, $\mathbf{H}\hat{\mathbf{c}} = \mathbf{0}$) or when a preset maximum number of iterations $k$ is reached. The exchanged messages are log-likelihood ratios. In a fully parallel architecture, each variable and check node corresponds to a miniature processor.

Let $\mathcal{C}(i)$ denote the set of check nodes connected to variable node $i$. The message processing rule for the message from variable node $i$ to check node $j \in \mathcal{C}(i)$ is:

$$L_{ij} = \text{LLR}(y_i) + \sum_{k \in \mathcal{C}(i)/j} R_{ki}, \tag{4}$$

where $\text{LLR}(y_i)$ is the channel log-likelihood ratio and $R_{ki}$ is the message from check node $k$ towards variable node $i$. $\text{LLR}(y_i)$ is defined as:

$$\text{LLR}(y_i) \triangleq \log \frac{p_{Y|X}(y_i|+1)}{p_{Y|X}(y_i|-1)} = \frac{2y_i}{\sigma^2}. \tag{5}$$

A hard decision for modulated codeword symbol $i$ can be made at each iteration as:

$$\hat{x}_i = \text{sign}\left(\text{LLR}(y_i) + \sum_{k \in \mathcal{C}(i)} R_{ki}\right). \tag{6}$$

The corresponding hard decision for codeword bit $i$ is calculated as:

$$\hat{c}_i = \begin{cases} 0, & \hat{x}_i = +1, \\ 1, & \hat{x}_i = -1. \end{cases} \tag{7}$$

Let $\mathcal{V}(i)$ denote the set of variable nodes connected to check node $i$. The message processing rule for the message from check node $i$ to variable node $j \in \mathcal{V}(i)$ is:

$$R_{ij} = 2\tanh^{-1}\left(\prod_{k \in \mathcal{V}(i)/j} \tanh\left(L_{ki}/2\right)\right). \tag{8}$$

One complete BP iteration consists of a variable-to-check message update followed by an update of the check-to-variable messages. BP is hard to implement in hardware mainly due to the $\tanh(\cdot)$ function in the check node update rule.

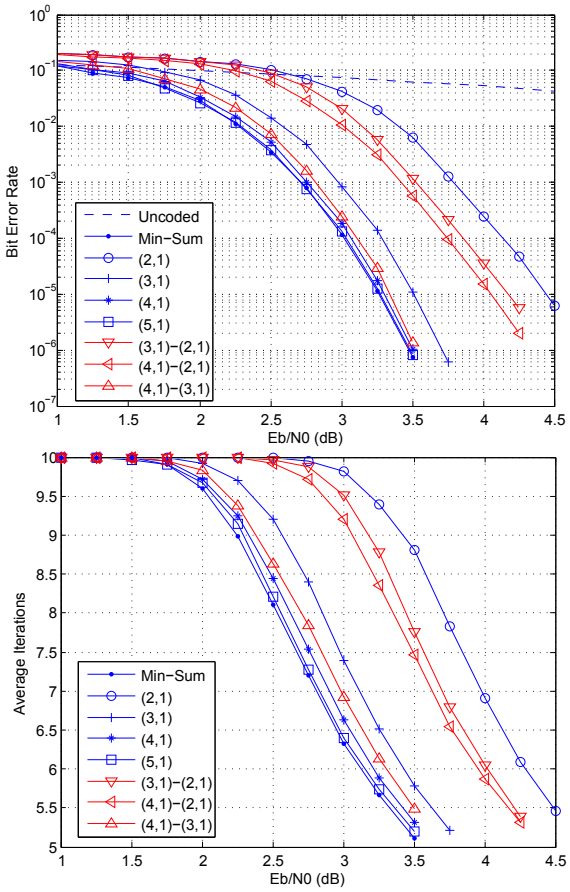



**Fig. 2**. BER and average number of iterations vs. Eb/N0.

A very hardware-friendly approximation of BP can be achieved with the Min-Sum (MS) algorithm [15]. This algorithm uses a much simpler update rule for the check nodes:

$$R_{ij} = \left( \prod_{k \in \mathcal{V}(i)/j} \text{sign}(L_{ki}) \right) \min_{k \in \mathcal{V}(i)/j} |L_{ki}| \,. \qquad (9)$$

The product of signs is a simple XOR operation with $(|\mathcal{V}(i)|-1)$ inputs, while the minimum can be calculated efficiently by a binary comparator tree. The variable node update rule is the same as that of BP.

## 3. MESSAGE QUANTIZATION

While being significantly simpler than BP, the MS algorithm still requires the use of floating point arithmetic. However, for a high speed hardware implementation, we need to quantize the values of the messages. The Tanner graph's randomness can make routing very hard, so it is important to use as few bits as possible for the representation of the messages.

By using fewer bits for quantization, we also simplify the internal structure of the variable node and check node processing units. In a fully parallel decoder architecture, a very large number of instances of variable nodes and check nodes is used. So, even small reductions in resource utilization for each node are magnified by some orders of magnitude and can thus result in significant savings.

**Table 1**. Scaling of initial LLR$(y_i)$ messages.

| Quantization | LLR Scaling |
|---|---|
| (2,1) | $y_i/4\sigma^2$ |
| (3,1) | $y_i/2\sigma^2$ |
| (4,1) | $y_i/\sigma^2$ |
| (5,1) | $2y_i/\sigma^2$ |
| (3,1)–(2,1) | $y_i/4\sigma^2$ |
| (4,1)–(2,1) | $y_i/3.5\sigma^2$ |
| (4,1)–(3,1) | $y_i/1.5\sigma^2$ |

**Table 2**. Resource utilization for $n = 1000$ on a Virtex 5.

| Quantization | Registers | LUTs | Slices |
|---|---|---|---|
| (2,1) | 16,012 | 17,912 | 6,997 |
| (3,1) | 23,012 | 47,413 | 15,398 |
| (4,1) | 30,012 | 112,913 | 38,378 |
| (5,1) | 37,012 | 158,842 | 48,943 |
| (3,1)-(2,1) | 17,012 | 32,914 | 11,001 |
| (4,1)-(3,1) | 24,012 | 61,761 | 16,966 |

### 3.1. Fixed-Point Quantization

An $(n_1, m_1)$ signed fixed point quantization scheme uses a total of $n_1$ bits, of which $m_1$ are used for the fractional part. Results for the Bit Error Rate (BER) and average number of iterations for a fixed-point implementation of MS using various quantization schemes for a (3,6)-regular code of length 1000 when performing a maximum of $k = 10$ decoding iterations are presented in Fig. 2. The initial LLR values have to be scaled appropriately in order to fit in the dynamic range of the quantizer. The scaling we applied is presented in Table 1. We observe that, if we use (4,1) or (5,1) quantization, then we have virtually no loss in performance with respect to the floating point implementation. When using (3,1) quantization we observe a loss of 0.2 dB at a BER in the order of $10^{-6}$. When moving to (2,1) quantization, the gap becomes approximately 1.2 dB at a BER in the order of $10^{-6}$.

Since check nodes do not perform operations that can result in overflows, we expect that reducing the number of quantization bits for check nodes should not have a significant impact on the decoder's performance, while significantly reducing routing complexity. An $(n_1, m_1)$–$(n_2, m_2)$ hybrid quantization scheme uses $(n_1, m_1)$ quantization for the initial LLR values and $(n_2, m_2)$ quantization for the messages to and from check nodes. We observe that the (4,1)–(3,1) scheme performs almost identically to the $(4, 1)$ scheme. In addition, the (3,1)–(2,1) scheme provides acceptable performance at very low complexity. Most other hybrid schemes with less than 4 bits, which are not presented in Fig. 2, re-

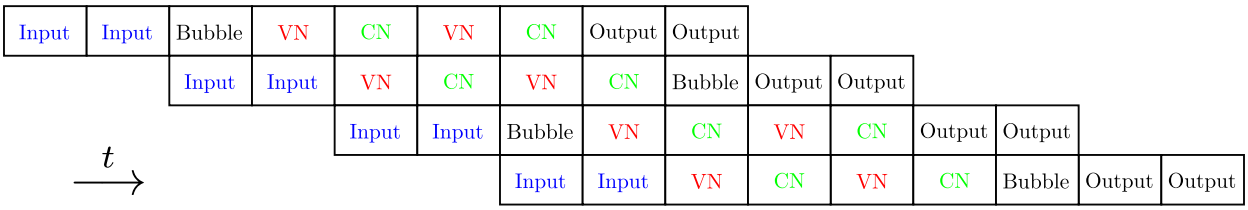



**Fig. 3**. Decoder pipeline scheduling for $k = 2$. Each box represents one clock cycle.

sulted in high error floors.

### 3.2. Effect of Quantization on Resource Utilization

Resource utilization for a code of length $n = 1000$ when using different quantization schemes is presented in Table 2. We observe that, when using (4,1)–(3,1) quantization, we need 25% fewer wires for the routing between variable and check nodes and approximately 45% fewer LUTs than when using (4,1) quantization, with negligible loss in performance. Furthermore, when using (3,1)–(2,1) quantization, we need 50% fewer wires and approximately 71% fewer LUTs than when using (4,1) quantization, but at a non-negligible 0.75 dB loss in performance.

It has to be noted that quantization schemes which use very few bits are prone to error floors. We did not observe any error floor for the quantization schemes presented in Fig. 2 up the BERs we simulated. However, it is possible that some of these schemes are not suitable for applications where very low BERs (e.g. $10^{-10}$) are required.

## 4. DECODER ARCHITECTURE

The amount of LUTs required for the implementation of a logic function on an FPGA depends solely on the number of the function's inputs and outputs. This means that it is possible to significantly reduce the number of LUTs required for a cascade of functions by directly implementing the composition of said functions. For example, if we implement the 4-bit adder and the 4-bit to 3-bit converter in Fig. 4 independently and then connect them, we will have one function with 8 inputs and 4 outputs and one function with 4 inputs and 3 outputs. If we directly implement the composition of the two functions, we will only have one function with 8 inputs and 3 outputs. This way, we have not only eliminated one of the two functions completely, but we have also reduced the complexity of the remaining function. In this design, we took full advantage of this property, which is unique to FPGAs.

### 4.1. Pipelining

In existing FPGA-based implementations of fully parallel LDPC decoders [5]–[7], one decoding iteration is completed in one clock cycle. However, due to routing complexity, this leads to high path delays and, consequently, low clock frequencies. In order to reduce path delays, we split one decoding iteration into two clock cycles by adding registers at the outputs of the variable and check nodes. We effectively create a pipeline with four stages, namely, the Input stage, which is responsible for loading the initial LLRs, the Variable Node (VN) stage, the Check Node (CN) stage, and the Output stage, which is responsible for the output of the hard decisions. Each stage will be discussed in more detail in the sequel.

In order to perform $k$ decoding iterations, we need $2k$ clock cycles, since one decoding iteration consists of activation of the variable nodes, which is followed by activation of the check nodes. So, the input stage has $2k$ clock cycles to load the initial LLRs for the next codeword and the output stage has $2k$ clock cycles to output the hard decisions for the previous codeword. However, for a given clock frequency, throughput is reduced by a factor of 2 with respect to the non-pipelined architecture, since each decoding iteration now requires 2 clock cycles. In order to overcome this problem, we note that, at each clock cycle, either the VN or the CN stage is idle. So, we can overlap the decoding of two codewords, thus completely eliminating the throughput loss. If we impose a phase difference of $k/2$ iterations on the decoding of the two codewords, the input stage has $k$ cycles to load the initial LLRs, while the output stage has $k$ cycles to output the hard decisions for the previous codeword.

A pipeline schedule example when $k = 2$ decoding iterations are performed is presented in Fig. 3. The bubbles can not be avoided, since both codewords would require use of the same stage (VN or CN) at the same time if they were not inserted. Fortunately, they do not have a negative impact on throughput; we can still output one decoded codeword every $k$ cycles. However, they slightly increase decoding latency. More precisely, without the presence of bubbles, in order to input, process, and output a codeword, we would need a total of $4k$ cycles. With the presence of bubbles, we need a total of $4k + 1$ cycles, which, for $k = 10$, is an increase of only 2.5%.

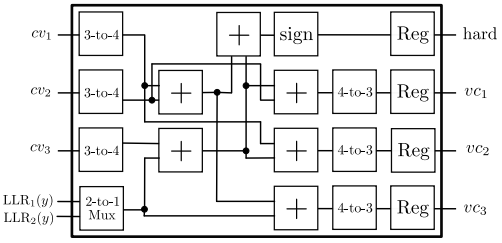



**Fig. 4**. Variable node architecture for the (4,1)–(3,1) design.

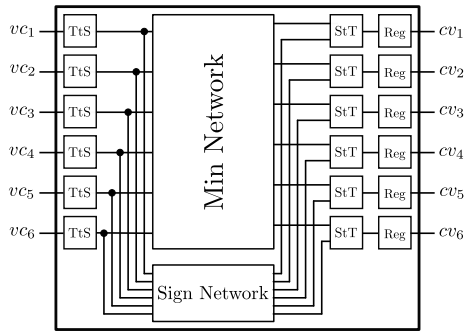



**Fig. 5**. Check node architecture.

### 4.2. Variable Node and Check Node Processing Units

The variable node processing units (VNs) take three $n_2$-bit two's complement messages from the check nodes and one $n_1$-bit two's complement initial LLR message as input and produce three $n_2$-bit two's complement output messages using Eq. (4), as well as a hard decision for the bit in question using Eq. (6) and Eq. (7). Additions are performed using saturation arithmetic. The initial LLR message is chosen by a 2-to-1 multiplexer based on the codeword which is decoded at each clock cycle. The internal architecture of the VNs is presented in Fig. 4.

The check node processing units (CNs) take six $n_2$-bit two's complement messages from the variable nodes and produce six $n_2$-bit two's complement output messages using Eq. (9). It was crucial to reduce the number of quantization bits for the check nodes to less than 3, since deciding which of two 3-bit (or less) numbers is smaller requires a single 6-input Virtex 5 LUT. We mentioned that additions are performed in two's complement; however, comparisons in the check nodes are performed using sign-magnitude representation. So, the CNs need to perform two conversions, one from two's complement to sign-magnitude (TtS) and one from sign-magnitude to two's complement (StT). The internal architecture of the CNs can be seen in Fig. 5.

### 4.3. Input and Output Circuits

With the proposed pipeline and quantization scheme, for a codeword of $n$ bits, the input stage needs to process $n_1 \cdot n$ bits of input in $k$ cycles. So, at every cycle we need to input $m_1 = \frac{n_1 \cdot n}{k}$ bits. The output stage needs to process $n$ bits in $k$ cycles simultaneously with the input stage. This means that at every cycle we need to output $m_2 = \frac{n}{k}$ bits. In total, we need to process $m = m_1 + m_2 = \frac{(n_1+1)n}{k}$ bits per cycle. This gives rise to the following upper bound on the length of the code which we can implement on any given FPGA:

$$n \leq \frac{pk}{(n_1+1)}, \tag{10}$$

where $p$ is the number of available I/O pins.

For example, the Virtex 5 XC5VLX110 FPGA has 800 available I/O pins. So, when using (4,1)-(3,1) quantization and performing 10 decoding iterations, it is impossible to implement a code with $n > 1600$ bits, even if we have sufficient logic available. In practice, some additional control signals are always needed, so Eq. (10) should in fact be interpreted as a strict inequality.

The input and output circuits which we used are depicted in Fig. 6. They are identical to the circuits presented in [5]. The input circuit consists of $\frac{n \cdot n_1}{k}$ serial-in and parallel-out (SIPO) registers of $k$ bits each, which are arranged in parallel in order to input all $n \cdot n_1$ LLR bits in $k$ cycles. The output circuit consists of $\frac{n}{k}$ parallel-in and serial-out (PISO) registers of $k$ bits each, which are also arranged in parallel in order to output all $n$ hard decision bits in $k$ cycles.

### 4.4. Overall LDPC Decoder Datapath

In the overall datapath of our LDPC decoder, we have one instance of the input circuit and one instance of the output circuit, which form the Input and Output Stages, respectively. Furthermore, we have one instance of the variable node processing unit for each variable node in the code's Tanner graph, and one instance of the check node processing unit for each check node in the code's Tanner graph. These processing units are connected in the same way as the corresponding variable and check nodes are connected in the Tanner graph. The set of all VNs forms the VN Stage and the set of all CNs forms the CN Stage. Finally, we have a small FSM, which is responsible for generating all control signals. A block diagram of the overall datapath is depicted in Fig. 6.

## 5. RESULTS

In Tables 3 and 5, we present implementation results for the (4,1)–(3,1) decoder for $n = 1000$. This design was downloaded and tested on a Virtex 5 XC5VLX110 FPGA (Speed Grade -3). In addition, we present Post PAR results for a decoder with $n = 1152$, for comparison with [7]. In Tables 4 and 5, we present Post PAR results for our low-complexity (3,1)–(2,1) decoder. The LDPC codes were

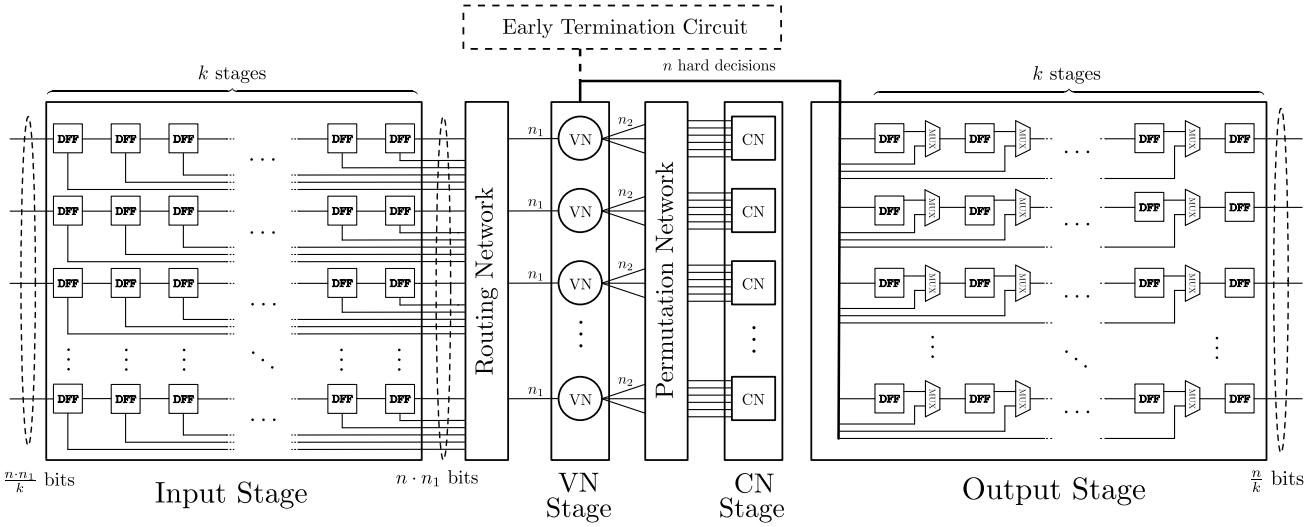



**Fig. 6**. Overall LDPC decoder datapath.

**Table 3**. Resource utilization for the (4,1)-(3,1) design.

| Size | n=1000 | n=1152 |
|---|---|---|
| Device | Virtex 5 XC5VLX110 | Virtex 5 XC5VLX155 |
| Registers | 24,012/69,120 (34%) | 27,372/97,280 (28%) |
| LUTs | 61,761/69,120 (89%) | 72,290/97,280 (74%) |
| Slices | 16,966/17,280 (98%) | 21,488/24,320 (88%) |
| IOBs | 502/800 (62%) | 572/800 (71%) |
| Clock | 154.30 MHz | 154.30 MHz |
| Throughput | 7.72 Gbps | 8.89 Gbps |
| Eb/N0 at BER $10^{-6}$ | 3.5 dB | 3.5 dB |

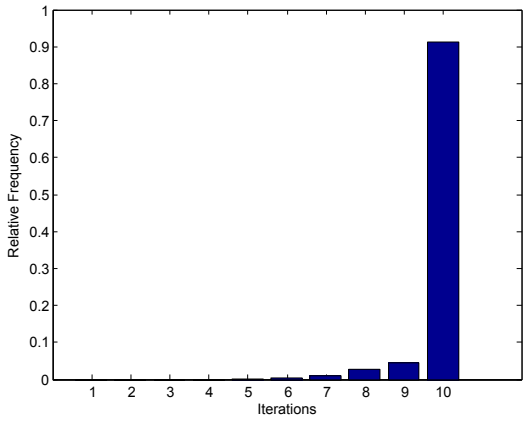
**Table 4**. Resource utilization for the (3,1)-(2,1) design.

| Size | n=1000 | n=1152 |
|---|---|---|
| Device | Virtex 5 XC5VLX110 | Virtex 5 XC5VLX110 |
| Registers | 17,012/69,120 (24%) | 19,392/69,120 (28%) |
| LUTs | 32,914/69,120 (46%) | 34,960/69,120 (51%) |
| Slices | 11,001/17,280 (63%) | 11,771/17,280 (68%) |
| IOBs | 402/800 (50%) | 458/800 (57%) |
| Clock | 211.40 MHz | 211.40 MHz |
| Throughput | 10.57 Gbps | 12.18 Gbps |
| Eb/N0 at BER $10^{-6}$ | 4.25 dB | 4.25 dB |

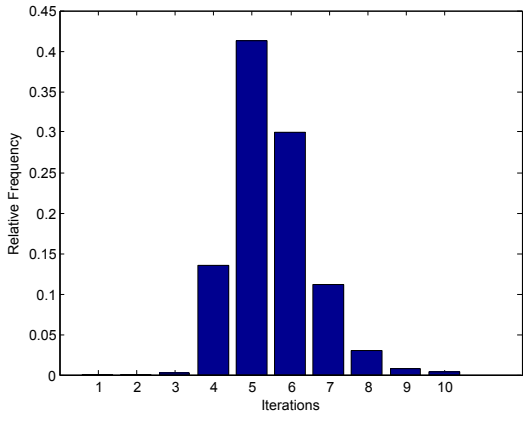
constructed using the Progressive Edge Growth (PEG) algorithm [16]. VHDL code generation is automated by a script. Information throughput, measured in Gbps, is calculated as:

$$T_{\text{max}} = \frac{r \times n \times f}{k}, \tag{11}$$

where $r$ is the code's rate, $n$ is the code's length, $f$ is the clock frequency in GHz, and $k$ is the number of decoding iterations. For our decoder, $k = 10$ and $r = 0.5$. Most papers in the literature (e.g. [7]–[11]) ignore the I/O delay in the calculation of the throughput. We also ignore it, but for good reason; once the pipeline is full, the I/O overhead is completely masked.




(a) Eb/N0 = 2 dB.




(b) Eb/N0 = 3.5 dB.

**Fig. 7**. Number of iterations for $n = 1000$.

**Table 5**. Comparison with previous work.

| | This work | | | [7] | [5] | [17] |
|---|---|---|---|---|---|---|
| Code Length | 1000 | 1152 | 1152 | 1152 | 1200 | 1024 |
| Code Type | $(3,6)$-regular | $(3,6)$-regular | $(3,6)$-regular | $(3,6)$-regular | $(3,6)$-regular | $(3,6)$-regular |
| Decoding Algorithm | Min-Sum | Min-Sum | Min-Sum | MMS | Min-Sum | Min-Sum |
| LLR Quantization | 4 bits | 4 bits | 3 bits | 4 bits | 3 bits | 4 bits |
| Message Quantization | 3 bits | 3 bits | 2 bits | 2 bits | 3 bits | 4 bits |
| Clock Frequency | 154.30 MHz | 154.30 MHz | 211.40 MHz | 149.00 MHz | 100.00 MHz | 400.00 MHz |
| Max. Iterations | 10 | 10 | 10 | 10 | 10 | $\sim$15 (31 cycles) |
| Max. Delay | 220 ns | 220 ns | 161 ns | 121 ns | 180 ns | 78 ns |
| Eb/N0 at $10^{-6}$ BER | 3.5 dB | 3.5 dB | 4.25 dB | 4 dB | $>4$ dB | 3.2 dB |
| Av. Iter. at $10^{-6}$ BER | 5.8 | 5.8 | 5.6 | 6.8 | n/a | n/a |
| Av. Throughput | 14.6 Gbps | 16.9 Gbps | 21.6 Gbps | 12.6 Gbps | n/a | n/a |
| Min. Throughput | 7.7 Gbps | 8.9 Gbps | 12.2 Gbps | 8.6 Gbps | 6.0 Gbps | 13.2 Gbps |
| Device | Virtex5 XC5VLX110 | Virtex 5 XC5VLX155 | Virtex 5 XC5VLX110 | Virtex 5 XC5VLX110 | Virtex 4 XC4VLX200 | 90 nm CMOS ASIC |

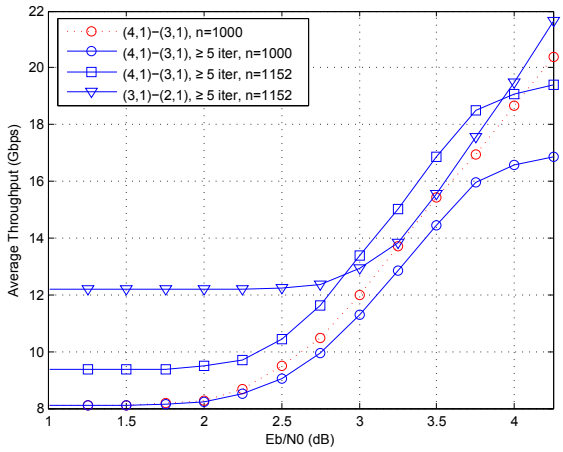



**Fig. 8**. Average throughput vs. Eb/N0.

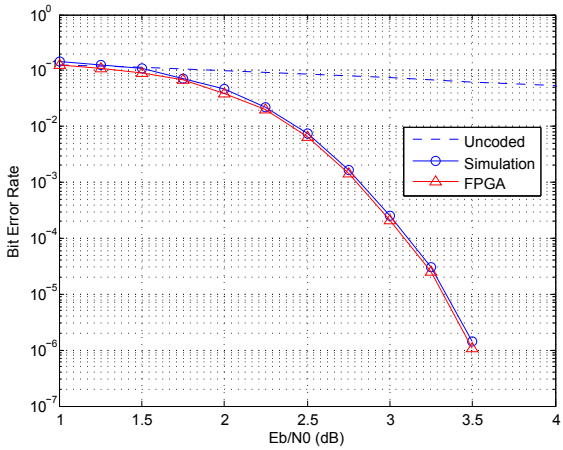



**Fig. 9**. BER vs Eb/N0 for the (4,1)–(3,1) decoder.

### 5.1. Early Termination Circuit

An early termination circuit halts decoding when $\mathbf{H}\hat{\mathbf{c}} = \mathbf{0}$. In this case, average throughput is calculated based on the throughput at maximum iterations, $T_{\max}$, and the average number of iterations at each Eb/N0, $k_{\text{avg}}$, as:

$$T_{\text{avg}} = \frac{kT_{\max}}{k_{\text{avg}} - 1/2} \tag{12}$$

The denominator is derived as follows. If, for a given input, $k'$ decoding iterations have to be performed, we have $(k'-1)$ full iterations of two cycles and the last iteration will stop at the VN stage, since this is where the early termination circuit is implemented. So, in total we will have $2(k'-1)+1 = 2k'-1$ cycles, i.e. $k'-1/2$ iterations.

In this case, significant I/O problems arise due to the non-uniform distribution of the number of iterations. Many papers in the literature do not address this problem when considering early termination circuits (e.g. [7]). In Fig. 7, we present a histogram of the required number of iterations for successful decoding for a total of $10^4$ codewords. If we force the decoder to perform at least, say, $k/2$ iterations before termination, this should not have a significant impact on the average throughput at high Eb/N0, while guaranteeing that we have $k$ cycles to load the input data for the next codeword. At low Eb/N0, the degradation of the average throughput will be negligible. The output is more problematic, since we do not know exactly when decoding of the previous codeword will terminate. If it terminates after fewer iterations than the previous codeword, then the output stage will still be busy. However, if we can guarantee that we can output the data in $k/2-1$ cycles, the output stage will always be free for the next codeword.

In Fig. 8, we present results that can be achieved by performing at least 5 decoding iterations. We observe that the loss in average throughput is indeed small and more prominent at high Eb/N0. We also observe that, as Eb/N0 is increased, the average throughput does not increase indefi-

nitely, since the average number of iterations can not drop below 5. In addition, we compare BER results of a software simulation and our FPGA implementation in Fig. 9.

### 5.2. Comparison With Previous Work

By comparing our results with those presented in [7], we see that our (4,1)–(3,1) decoder can achieve a 16% higher average throughput (14.6 Gbps vs. 12.6 Gbps), using the same amount of logic, even though we implemented a shorter code. At the same time, our decoder requires a $0.5$ dB lower Eb/N0 to achieve a BER of $10^{-6}$. The minimum throughput is 1 Gbps lower, due to the smaller code size. We also observe that, even though we implemented a more complex decoder, we can achieve higher clock frequencies, due to the pipeline registers. The decoding delay is increased because decoding requires more cycles, but it is still very low. For $n = 1152$, we see that the minimum throughput is slightly higher than that of [7] while the average throughput is 34% higher than that of [7] (16.9 Gbps vs. 12.6 Gbps). Our low-complexity decoder can achieve a clock frequency of 211.40 MHz, resulting in a throughput of $21.6$ Gbps at an Eb/N0 of $4.25$ dB, where the BER is in the order of $10^{-6}$.

In a comparison with a recent ASIC implementation of a fully parallel LDPC decoder, we observe that FPGA-based implementations still have a long way to go, since none of our decoders can match the throughput, delay, or Eb/N0 performance of the decoder presented in [17]. Nevertheless, our results are encouraging.

## 6. CONCLUSION

We presented a fully parallel LDPC decoder architecture which is able to achieve a throughput of $16.9$ Gbps at an Eb/N0 of $3.5$ dB using a code of length $1152$. We employed (4,1)–(3,1) hybrid quantization in order to reduce routing complexity and we used pipelining in order to reduce path delays. We also applied FPGA-specific optimizations and problem sizing to minimize LUT utilization. Furthermore, we have fully addressed the I/O problem by providing I/O circuits and pipeline scheduling that are able to mask the I/O delay. We also presented a low-complexity (3,1)–(2,1) decoder which can achieve a throughput of $21.6$ Gbps at a BER of $10^{-6}$ by sacrificing $0.75$ dB in terms of Eb/N0. Our decoders are, to the best of our knowledge, the fastest fully parallel FPGA-based LDPC decoders in the literature.

## 7. REFERENCES


[1] R. Gallager, "Low-density parity-check codes," *IRE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962.

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

[3] T. Richardson and R. Urbanke, "Design of capacity approaching irregular low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 619–637, Feb. 2001.

[4] T. Richardson and R. Urbanke, *Modern Coding Theory*, Cambridge University Press, 2008.

[5] S. G. Wilson, R. Zarubica and E. Hall, "Multi-Gbps FPGA-based low density parity check (LDPC) decoder design," in *Proc. Global Telecommunications Conf., GLOBECOM '07*, Nov. 2007, pp. 548–552.

[6] S. Mannor, S. S. Tehrani and W.J. Gross, "Fully parallel stochastic LDPC decoders," *IEEE Trans. Sig. Proc.*, vol. 56, no. 11, pp. 5692–5703, 2008.

[7] V. A. Chandrasetty and S. M. Aziz, "An area efficient LDPC decoder using a reduced complexity min-sum algorithm," *Integration, the VLSI Journal*, vol. 45, no. 2, pp. 141–148, Aug. 2011.

[8] Y. Chen and D. Hocevar, "A FPGA and ASIC implementation of rate 1/2, 8088-b irregular low density parity check decoder," in *Proc. Global Telecommunications Conf., GLOBECOM '03*, Dec. 2003, vol. 1, pp. 113–117.

[9] Z. Wang and Z. Cui, "Low-complexity high-speed decoder design for quasi-cyclic LDPC codes," *IEEE Trans. VLSI Syst.*, vol. 15, no. 1, pp. 104–114, Jan. 2007.

[10] X. Chen, J. Kang, S. Lin and V. Akella, "Memory system optimization for FPGA-based implementation of quasi-cyclic LDPC codes decoders," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 1, pp. 98–111, Jan. 2011.

[11] A. C. Carusone, A. Darabiha and F. R. Kschischang, "A bit-serial approximate min-sum LDPC decoder and FPGA implementation," in *Proc. IEEE Int. Symp. Circuits and Systems, ISCAS 2006*, May 2006, 4 pp.

[12] P. Schläfer, C. Weis, N. Wehn and M. Alles, "Design space of flexible multigigabit LDPC decoders," in *VLSI Design*, 2012.

[13] M. R. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inf. Theory*, vol. 27, pp. 533–547, Sep. 1981.

[14] J. Pearl, *Probabilistic Reasoning in Intelligent Systems*, Morgan Kaufmann, 1988.

[15] N. Wiberg, *Codes and Decoding on General Graphs*, Ph.D. thesis, Linköping University, Linköping, Sweden, 1996.

[16] E. Eleftheriou, X.-Y. Hu and Dieter M. Arnold, "Regular and irregular progressive edge growth Tanner graphs," *IEEE Trans. Inf. Theory*, vol. 51, no. 1, pp. 386–398, Jan. 2005.

[17] N. Ozinawa, T. Hanyu and V. C. Gaudet, "Design of high-throughput fully parallel LDPC decoders based on wire partitioning," *IEEE Trans. VLSI Syst.*, vol. 18, no. 3, pp. 482–489, Mar. 2010.