FPGA implementation of low density parity-check (LDPC) coded recording channels

Seyed Mohammad Ehsan Hosseini

2010

Seyed, M. E. H. (2010). FPGA implementation of low density parity-check (LDPC) coded recording channels. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/20856>

<https://doi.org/10.32657/10356/20856>

FPGA IMPLEMENTATION OF LOW DENSITY PARITY-CHECK (LDPC) CODED RECORDING CHANNELS

SEYED MOHAMMAD EHSAN HOSSEINI

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University

in partial fulfillment of the requirement for the degree of

Master of Engineering

2010

Acknowledgements

First and Foremost, I would like to thank my project supervisor, Associate Professor Goh Wang Ling. It is through her patience and invaluable guidance that this work was accomplished. She not only assisted in my research, but helped and advised me on different aspects of my academic and personal life. On top of all, she always makes time for her students.

While a deep understanding of a subject is perhaps the essence of a Master's thesis, it is equally important to obtain a wider perspective of a problem. I am deeply thankful to my co-supervisor, Dr. Chan Kheong Sann, for discussions that stimulated me to look at my work from different angles. He also helped me a lot by reading my manuscripts and providing constructive criticism on all aspects of my writings.

All my friends and colleagues at A*STAR (Agency for Science, Technology and Research in Singapore) Data Storage Institute have been most cooperative by providing an inspiring atmosphere. A special thanks goes to my friend Elidrissy Moulay Rachid for his immediate assistance whenever I faced a problem throughout my work. I would also like to thank Ms Wong Wai Ee for introducing me to softwares and equipments which I have been using during the past couple of years.

Needless to say, my parents have been very important to me. I would like to express my deepest appreciation to my mother, who has undoubtedly been my most important teacher and friend since my childhood. Finally and especially, I thank my Father for his support and encouragement.

Contents

Acknowledgments	iii
Contents	v
Abstract	ix
List of Figures	xi
List of Tables	xiv
List of Symbols and Abbreviations	xv
1 Introduction	1
1.1 Iterative error correction codes	2
1.2 Very low BER performance estimation of LDPC codes	3
1.3 Objectives	5
1.4 Major contributions of the Thesis	6
1.5 Organization of the Thesis	7
2 LDPC Codes and Their Decoding Algorithms	9
2.1 Introduction	9
2.2 Linear block codes	10
2.3 Low density parity-check codes	12
2.3.1 Decoding LDPC codes	14

2.3.2	Overview of the reduced complexity decoding algorithms . . .	19
2.3.3	Quantization effects on the decoding algorithm	22
2.3.4	Error floor of LDPC codes	24
2.4	Summary	26
3	Existing LDPC Decoder Architectures	27
3.1	Introduction	27
3.2	Parallel LDPC decoders	28
3.3	Partially-parallel LDPC decoders	30
3.3.1	A 54 Mbps (3,6)-regular FPGA LDPC decoder	30
3.3.2	High speed LDPC decoder for Quasi-Cyclic codes	32
3.3.3	Memory efficient decoder architecture for QC-LDPC codes . .	35
3.3.4	TDMP decoding for LDPC codes	36
3.3.5	An LDPC decoder for IEEE802.16e based on self-routing network	41
3.3.6	Min-Sum Decoder for high rate Quasi-Cyclic LDPC codes . .	42
3.3.7	FPGA simulator for the evaluation of Quasi-Cyclic codes over magnetic recording channel	44
3.4	Serial LDPC decoders	46
3.5	Other Alternatives	47
3.5.1	Stream oriented LDPC decoding	48
3.5.2	Decoding LDPC codes on the GPUs	48
3.6	Summary	49
4	The Implementation of a Simplified Sum-Product Decoding Algorithm	51
4.1	Introduction	51
4.2	The simplified LDPC decoding algorithm	53
4.3	The Hardware/Software based system architecture	55
4.3.1	The serial decoder architecture	56
4.3.2	The AWGN channel	61

CONTENTS

vii

4.4	Implementation and results	65
4.5	Summary	69
5	Message Packing LDPC Decoder Architecture	71
5.1	Introduction	71
5.2	Overview of the existing flexible partially-parallel LDPC decoders . . .	73
5.3	Message Packing decoding architecture	77
5.3.1	Message packing for variable-to-check delivery	79
5.3.2	Message packing for check-to-variable delivery	83
5.3.3	Discussion and Results on Message Packing Method	85
5.4	FPGA Implementation	89
5.4.1	Variable node processor architecture	90
5.4.2	Check node processor architecture	95
5.4.3	FPGA implementation results	101
5.5	Summary	104
6	FPGA for LDPC Code Evaluation	107
6.1	FPGA-based simulator architecture	107
6.2	FPGA-based simulator design flow	112
6.3	LDPC code evaluation	114
6.4	Summary	117
7	Conclusion	119
7.1	Conclusion	119
7.2	Future work	121
	Publications	123
	Bibliography	125

Abstract

Low Density Parity-Check (LDPC) codes have received lots of attention during the past decade due to their near Shannon-limit performance and decoding at very high rates. However, several issues have been raised in the research work with an aim to achieve the practical implementation of the LDPC codes. The first major issue is the performance evaluation of the LDPC codes and their decoding algorithms in the very low bit error rate (BER) regions. Unfortunately, there have so far been no analytical tools for the performance investigation of LDPC codes and one has to study such codes through simulations. The main problem with the simulations is their prohibitive execution time, especially for applications that require extremely low BERs such as data storage devices. Another concern is the efficiency of the decoding algorithm known as the Sum-Product algorithm. The Sum-Product algorithm is not optimized for short length codes and codes with cycles in their bipartite graph representation. On the other hand, the iterative decoding algorithm is computationally intensive, which imposes a high cost on area and power consumption of the decoder. The third issue is the design of the LDPC decoders to support different code lengths, rates and modes of operation.

This work addresses the above issues by developing a field-programmable gate-array (FPGA) LDPC decoder in order to resolve the lengthy simulations of LDPC codes for next generation magnetic recording systems. At the algorithm level, a simplified version of the Sum-Product algorithm is proposed, which reduces the decoding complexity in the absence of any significant loss in the performance of decoding. Furthermore, a solution is provided for the decoding of arbitrarily constructed LDPC codes, which is to

implement on an FPGA platform. And finally, the proposed flexible LDPC decoder is employed for the simulation of some high rate LDPC codes, intended for storage systems, over an additive white Gaussian noise (AWGN) channel. To perform the necessary simulations for different codes and channel noise levels, a MATLAB graphical user interface (GUI) is developed to assist in the programming of the FPGA-based simulator.

List of Figures

1.1	Magnetic recording system as a communication system.	2
1.2	Structure of iterative decoders.	4
2.1	The 9×12 parity check matrix (a) corresponds to 9 parity-check equations in (b).	13
2.2	The Tanner graph of a (7,3) irregular LDPC code and its parity-check matrix.	14
2.3	Initialization of the variable-nodes and check-nodes	16
2.4	Check-Node update	17
2.5	Variable Node update	17
2.6	The block error rate vs SNR plot of a rate-0.89 LDPC code compared to the other coding schemes.	25
3.1	The architecture of the parallel decoder in [26].	29
3.2	The architecture of (3,6) regular LDPC decoder [30].	32
3.3	The enhanced partially parallel decoder architecture for QC-LDPC codes [31].	34
3.4	Memory-efficient partially-parallel decoder architecture [33].	36
3.5	The architecture-aware parity-check matrix for TDMP decoding.	37
3.6	The TDMP algorithm architecture showing the message passing and SISO decoding.	38
3.7	The chip architecture of TDMP decoder [35].	40
3.8	The architecture of LDPC decoder [29].	42

3.9	The partially parallel LDPC decoder architecture in [28].	43
3.10	FPGA-based simulator presented in [42].	45
3.11	Quasi-Cyclic LDPC decoder module in the FPGA-based simulator [42]. . .	46
3.12	Block diagram of a serial LDPC decoder presented in [44].	47
4.1	The hardware-based simulator architecture.	56
4.2	Block diagram of embedded LDPC decoder and its bus interface to the PLB. .	57
4.3	Block diagram of the check-node processor.	59
4.4	The AWGN generator based on Box-Muller method [43]	62
4.5	BER and FER performance of some unstructured LDPC codes.	66
4.6	Performance comparison of different decoding algorithms.	67
5.1	Prototype of a partially-parallel LDPC decoder.	74
5.2	LDPC decoder architecture by Kang[58].	76
5.3	A general prototype for partially-parallel LDPC decoders.	78
5.4	VTC network	81
5.5	Word structure representing the check node value.	84
5.6	Check-to-variable delivery network.	85
5.7	The resources	88
5.8	The pipelined VNP architecture	92
5.9	The VNP Micro-Program instruction word.	93
5.10	The pipelined CNP architecture.	96
5.11	The Min-Finder block.	99
5.12	The data forwarding unit forwards the updated rows from the Write stage to the Pre-Process and Update stages.	99
5.13	The modifications to the pipelined CNP to support the row correction. . . .	100
6.1	FPGA-based simulator system.	108
6.2	52-bit LFSR implemented using three slices of Virtex-4 FPGA.	109
6.3	The LDPC encoder architecture.	110

LIST OF FIGURES

xiii

6.4	The implementation and verification model.	112
6.5	(2048,384) LDPC code, column weight 6 and 25 decoding iterations. . . .	115
6.6	(4161,3430) LDPC code, column weight 4 and 15 decoding iterations. . . .	115
6.7	(4095,3358) LDPC code, column weight 3 and 15 decoding iterations. . . .	116
6.8	(4092,538) QC-LDPC code, column weight 4 and 15 decoding iterations. .	116

List of Tables

3.1	The performance and specifications of some existing LDPC decoders. . . .	50
4.1	System Performance for different high-rate codes.	68
5.1	The effect of FIFO depth on the number of cycles required.	86
5.2	Cycles required for VTC and CTV delivery for different high rate LDPC codes.	87
5.3	Delivery network synthesized results for Xilinx Virtex-4 FPGA.	89
5.4	Physical implementation results of message passing LDPC decoder	102
5.5	The LDPC decoder performance for several codes.	102
5.6	Comparison of proposed decoder with existing LDPC decoders.	104

List of Symbols and Abbreviations

Abbreviation	Description
ASIC	Application-Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BRAM	Block Random-Access Memory
CNP	Check Node Processor
CTV	Check-to-Variable
FER	Frame Error Rate
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
HDD	Hard Disk Drive
LDPC	Low Density Parity-Check
LFSR	Linear Feedback Shift Register
LLR	Log-Likelihood Ratio
LUT	Look-Up Table
SNR	Signal-to-Noise Ratio
VLSI	Very Large Scale Integration
VNP	Variable Node Processor

Abbreviation		Description
VTC		Variable-to-Check

Chapter 1

Introduction

The advent of digital computers and the explosion of information have triggered the vast growth of data storage systems. During the past decade, the capacity of hard disk drives (HDDs) has been growing exponentially while the cost per stored bit has decreased significantly [1]. This is a result of advancement of different areas such as very large scale integration (VLSI) technology, magnetic materials and write/read heads as well as in the signal processing and coding theory.

In HDDs, the information is stored on the magnetic medium by means of a writing head, that is magnetized due to the writing current waveform. This magnetic field polarizes the medium surface in one of the two possible directions and the information bits can be retrieved later by the read head. The write head, magnetic medium and the read head resemble a transmitter, a physical channel and a receiver, respectively. It is therefore more convenient to consider the magnetic recording system as a communication channel (Figure 1.1), and this permits one to conveniently apply the well-established communications theories and techniques in the study. A typical communication system consists of equalization, detection and coding techniques. In order to analyze a magnetic recording system from the communication perspective, it is necessary to have a deep understanding of the write/read head and the medium so that it can be represented by a mathematical model that can be used in the analysis and simulation of the magnetic

recording system.

Figure 1.1: Magnetic recording system as a communication system.

In the past 10 years, several signal processing and communication advances have boosted the storage capacity, and among them, the error correction codes have played a key role in achieving high capacity HDDs. In today's magnetic recording systems, the Reed-Solomon (RS) coding has been applied as the standard [2]. However, in future systems, a shift toward more powerful iterative schemes like the Turbo codes [3] and Low-density parity-check (LDPC) codes [4], whereby their ultimate capacity approaching have been proven, is inevitable. It is by now well understood that the LDPC codes, despite current constraints of the HDDs, can improve the performance of the recording system almost twice in the case of high storage densities approaching within 1.5 dB of the theoretical limit [5]. The superior performance of LDPC codes has gained the attraction of the researchers working in different areas of communications and VLSI in the past few years.

1.1 Iterative error correction codes

Since the introduction of Shannon's theory on the channel capacity [6], numerous researches have been conducted to narrow the gap between the practical and the theoretical capacity of channels for reliable transmission of data. The Maximum likelihood (ML) decoding theory [7] defines randomly selected codes of sufficient block length as a means of achieving theoretical limit. However, the complexity of the ML decoding

algorithm grows exponentially with the block length, and thus avoiding its realization in many practical systems. Therefore, the goal of the coding research has been to achieve near ML decoding performance while minimizing the decoding complexity. However, it was not fulfilled until the introduction of the iterative codes, Turbo codes and LDPC codes, in 1990s where their performance came to within 0.01 dB of the Shannon's limit.

Undoubtedly, the invention of Turbo codes is a breakthrough in the improvement of coding theory. The gain achieved by Turbo codes over other coding schemes is so high that for many applications the gap between practical system and theoretical limit has been closed. In spite of a near Shannon limit performance, turbo decoding can only be realized using computationally intensive iterative soft-input soft-output (SISO) decoders [3]. In order to minimize the complexity of SISO decoders, other iterative codes with good performance have been investigated. LDPC codes, originally developed by Gallager [4] in 1960s, were neglected for decades due to their decoding complexity. Rediscovered by Mackay and Neal [8], they demonstrated that LDPC codes can compete closely with Turbo codes [9]. As compared to Turbo codes, the Sum-Product Algorithm (SPA) used in decoding of LDPC codes is rather simple. Moreover, the SPA can be carried out in a parallel fashion, making it suitable for high throughput applications [10]. It should be noted that an LDPC decoder consists of SISO decodes, similar to Turbo decoders. However, each SISO unit (engine) comprises of many independent processors, which can work in parallel. This results in a higher throughput in addition to a simpler SISO decoding algorithm. Figure 1.2 demonstrates the principal structure of Turbo decoders and LDPC decoders.

1.2 Very low BER performance estimation of LDPC codes

In spite of the admirable performance of LDPC codes, their application in magnetic storage systems has not yet been realized. For the current state-of-the-art HDDs, the requirements define an overall block error rate of 10^{-12} , which may be further reduced in the future. This very low error rate and the necessity of high coding rates, may limit

Figure 1.2: Structure of iterative decoders.

the deployment of new coding schemes like LDPC codes. The main concern is in the performance of an LDPC coded system at very low error rates, whereby it is still open to question. It is known that LDPC codes exhibit a phenomenon known as *error floor* [11] at low error rates, which may not satisfy the requirements in many cases. At high signal-to-noise ratio (SNR), the error floor saturates the system performance and the error rate does not reduce significantly when the signal power is increased. Since there is no analytical tool, the prediction of error floor is challenging, where it can only be observed via simulations. However, computer simulation of an LDPC coded system for this purpose is quite slow and can only merely attain block error rates of up to 10^{-6} using

C or MATLAB software run on conventional PCs.

Recently, the hardware-based simulators have shown to be a powerful tool for the simulation and performance evaluation of LDPC codes [12]. The best platform for such simulators is the field-programmable gate array (FPGA) because of its flexibility and reconfigurability. FPGAs allow one to implement almost any digital circuit. Moreover, they can be reconfigured or programmed according to the simulation parameters and requirements. The FPGA speeds up the simulation since it removes the system bottleneck, that is the LDPC decoder block. While the FPGA's clock frequency is much lower than that of a processor, it is able to perform concurrent tasks, which evidently accelerates the overall computations. Fortunately, the LDPC decoding algorithm is inherently parallel, and hence, it is ideal for the hardware implementation. However, the decoding framework, which the decoding algorithm provides, is costly in terms of resources and area. The design of an LDPC decoder with high-throughput, flexibility, minimum resources, is therefore perhaps the key concern of an FPGA-based simulator.

1.3 Objectives

The objective of this project is to design and develop an FPGA based LDPC decoder for the simulation of magnetic recording channels on hardware. The decoder should not only have high throughput, but also has to be as flexible as possible in terms of LDPC code support. The flexibility is crucial since the simulator will then not have to be restricted to a particular class of codes. Moreover, the design should be modular and compatible with other blocks like the equalizer and the detector to be implemented in a complex model of the magnetic recording system. It is also desirable to design an automated software in C or MATLAB so that the parity check matrix of the code can be accepted as the input of the software for the generation of a synthesisable VHDL code.

Additionally, other blocks such as channel model, which can be assumed to be a simple additive white Gaussian noise (AWGN) model at the first instance, and the encoder are to be designed in order to test the decoder. These blocks ought to be flexible

and self-automated. Finally, the generated blocks are connected together and the system will be tested for some given LDPC codes. The desired output is the performance curves of LDPC codes, i.e. the bit error rate versus SNR.

1.4 Major contributions of the Thesis

In this work, different algorithms and architectures for decoding of LDPC codes have been studied. In the algorithm level, a new simplification to the decoding algorithm, known as Sum-Product Algorithm, is made. The proposed algorithm can be categorized into Min-Sum decoding algorithms family, in which a correction factor is introduced. Unlike other Min-Sum variations, the proposed algorithm updates the correction factor dynamically throughout decoding according to probabilities of variables. The proposed algorithm is implemented on an FPGA platform and its performance is compared to other simplified algorithms, which shows an improvement. In the architecture level, a novel solution for parallel decoding of unstructured LDPC codes is presented. The proposed solution finds patterns for simultaneous delivery of several messages in the message passing algorithm based on a computer algorithm. A flexible partially-parallel LDPC decoder is designed and implemented, which is capable of decoding random LDPC codes. The proposed decoder takes advantage of a novel architecture for delivery of messages. The implementation results proves that the proposed delivery network significantly occupies less resources compared to conventional delivery networks such as Crossbar networks and Benes networks. Furthermore, the flexible decoder is employed in an FPGA-based simulator, which evaluates the performance of LDPC codes in the presence of additive white Gaussian noise. The simulator is described in VHDL, which is configured using a MATLAB graphical user interface. The hardware-based simulator is able to investigate the performance of LDPC codes in bit error rate regions as low as 10^{-10} in a couple of hours.

1.5 Organization of the Thesis

In Chapter 2, LDPC codes are reviewed in terms of practical issues rather than the information theory details. After the introduction of LDPC codes, the Sum-Product decoding algorithm and its simplifications for LDPC decoding are reviewed more intensively in this chapter. Additionally, a section is devoted for the understanding of the effects of fixed-point realization of the decoding algorithm on its performance since the final goal is to implement the algorithm on hardware. Chapter 3 reviews the published works on the VLSI architectures of LDPC decoders as the main block in the hardware-based simulator. Several works, having novel ideas behind them, are studied to provide a solid knowledge on the the concepts of implementation. Multiple aspects of the LDPC decoders including throughput, area, maximum clock frequency and supported LDPC code are presented and compared to evaluate the effects of different architectures on these metrics. Chapter 4 introduces a new simplification to the SPA, which reduces the number of computations while maintaining a good performance. Furthermore, a Hardware/Software co-design methodology is employed to implement the algorithm on the FPGA. Using the co-design methodology, a simple FPGA-based simulator for LDPC codes is provided. Chapter 4 presents a novel solution for the parallel decoding of arbitrarily constructed LDPC codes on the hardware. A flexible LDPC decoder is implemented on Xilinx Virtex4 FPGA, which delivers the maximum flexibility in terms of code support. This achievement is important since the hardware-based simulator will not be confined to a particular class of LDPC codes. In chapter 5, the flexible LDPC decoder of Chapter 4 is utilized in a hardware-based simulator, that is based on an AWGN channel. A few high-rate LDPC codes, suitable for magnetic recording systems, are investigated throughout the simulations and their performance curves are derived for that channel type at very low BERs. Finally, in Chapter 6, a brief summary of this work is presented along with discussion on the contributions and future work.

Chapter 2

LDPC Codes and Their Decoding Algorithms

2.1 Introduction

The area of digital communications and information theory traces back its origin to 1948 when E. Claude Shannon published his revolutionary paper on the limits of the transmission of reliable data over the communication channels [6]. In this paper, he formalized the concept of information and the maximum bound on the transmission of it. Shannon proved that for a given communication channel, there exists a limit called *capacity* which indicates the maximum amount of information that can be reliably transmitted. A reliable transmission of information that exceeds the capacity is impossible. In order to achieve reliable communication, it is mandatory to send correlated elements through the channel rather than single element. This eventually leads to the concept of *codes*, which is a set of vectors over the input alphabet. These vectors are called *codeword*, which are unique for every input.

The message to be communicated is first *encoded* and turned into a codeword by adding *redundancy* and then sent through the noisy channel. The received message is *decoded* at the receiver into a resembling message that is as close as possible to the original

message [7]. The degree of recovery depends on how good the code is designed according to the channel. Hence, different realizations of channel coding have been proposed. Traditional codes such as the Bose-Chaudhuri-Hocquenghem (BCH) and Convolutional codes have been shown to perform within 4 dB of the Shannon's limit [13]. The iterative codes, which are of interest these days, have come very close to the Shannon's limit. One important class of iterative codes is the Low-Density Parity-Check (LDPC) codes [4] that are able to work as close as 0.0045 dB to the Shannon's limit [9].

In contrast to other iterative codes, the LDPC codes provide some attractive benefits such as their rather simple decoding algorithm known as the *Sum-Product* algorithm. Additionally, the decoding scheme is inherently parallel making it a suitable choice for high rate applications. Due to their remarkable performance and their high throughput decoding, the LDPC codes have found applications in Wireless Local Area Network (WLAN) and Digital Video Broadcasting (DVB) standards. With the use of LDPC codes in various applications, different research areas have been uncovered, to mainly address two key issues: 1) LDPC decoding algorithms as well as its modifications and simplifications, according to the application; and 2) the design and implementation of LDPC decoders with consideration for goals such as flexibility, throughput and power consumption.

In the following section of this chapter, a brief review and definition of the block codes will be provided before introducing the LDPC. Finally, the LDPC decoding algorithms and the modified versions, presented in the literature, are reviewed.

2.2 Linear block codes

In this section a review of the block codes, whereby LDPC codes are a subset of them, are given (for further discussion, refer to [7]). A Linear Block Code $\mathbf{C}$ consists of a set of fixed-length vectors called codeword. The codeword has n elements from a set of q alphabets. In the binary codes, the alphabet set consists of two symbols of 0 and 1 and hence 2^n codewords are possible in a binary block of length n . From these 2^n

codewords, one may select a subset of 2^k where $k < n$ and they all have the same length of n . Consequently, a block of k information bits can be mapped to a codeword of length n . The resulting block code is denoted as (n, k) code and the ratio k/n is defined as the *code rate*, R_c .

Since $\mathbf{C}$ is a subspace of dimension k , $\exists k$ linearly independent vectors $\bar{g}_1, \bar{g}_2, \dots, \bar{g}_k$ which span $\mathbf{C}$. The corresponding mapping ($\bar{u} \rightarrow \bar{c}$) can be viewed as:

$$\bar{c} = u_1 \bar{g}_1 + \dots + u_k \bar{g}_k \quad (2.1)$$

Based on the above discussion, we denote $u_1, u_2, u_3, \dots, u_k$ as the k information bits and $c_1, c_2, \dots, c_n$ the generated codeword of n bits. The encoded codeword can be realized by a set of n equations, and can be expressed in matrix form:

$$\mathbf{C}_m = \mathbf{U}_m \mathbf{G} \quad (2.2)$$

where $\mathbf{G}$ is called the *generator matrix*. It can be also written in the expanded form:

$$\mathbf{C}_m = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} u_1 & u_2 & \dots & u_k \end{bmatrix} \begin{bmatrix} g_{11} & g_{12} & \dots & g_{1n} \\ g_{21} & g_{22} & \dots & g_{2n} \\ \vdots & \vdots & & \vdots \\ g_{k1} & g_{k2} & \dots & g_{kn} \end{bmatrix} \quad (2.3)$$

The elements of $\mathbf{G}$ in binary block codes are 0 and 1 and the matrix multiplication is performed in modulo-2. Consequently, the encoder may consist of n modulo-2 adders (XORs) with different number of inputs tied to different information bits according to the generator matrix.

Since $\mathbf{G}$ has k linearly independent rows, it has a rank of k . Hence, it can be row reduced and put in form of:

$$\mathbf{G} = [\mathbf{I}_k : \mathbf{P}] \quad (2.4)$$

where $\mathbf{I}_k$ is the $k \times k$ identity matrix and $\mathbf{P}$ is a $k \times (n - k)$ matrix that determines the $n - k$ redundant bits. Any linear (n, k) block code has a dual code of $(n, n - k)$ with 2^{n-k} code vectors. The generator matrix for such a code is denoted by $\mathbf{H}$ where each of which is called the *parity-check* matrix. Any code from $\mathbf{C}$ codeword space is orthogonal to any codeword of its space. Hence:

$$\mathbf{C}_m \mathbf{H}' = \mathbf{0} \quad (2.5)$$

where $\mathbf{H}'$ is the transposed matrix of $\mathbf{H}$ and $\mathbf{0}$ is the null matrix. The $\mathbf{H}$ matrix has $n - k$ row vectors each of which multiplied by the codeword in (2.2) and hence $n - k$ vector multiplication is done in (2.5). Each of these multiplications is called a *parity-check equation* and if the multiplication results in the scalar 0, the corresponding parity check equation is satisfied. For a valid codeword like $\mathbf{C}_m$, $n - k$ equations should be satisfied and the function is called *syndrome-check*.

2.3 Low density parity-check codes

The low-density parity-check (LDPC) codes [4], are codes specified by a parity-check matrix which is sparse and has a small number of ones in each row and column in comparison with the total number of elements in rows and columns. They were forgotten for many years until Mackay and Neal re-discovered them in mid 1990s [8].

Based on the structure of the parity-check matrix, the LDPC codes can be grouped into different categories [14]. A *regular* (j, k) LDPC code has a parity-matrix with exactly j ones in each column and k ones in each row when the total number of columns is N and the total number of rows is M . According to this definition, every parity-check equation of a regular LDPC code consists of k bits and every bit is involved in j parity-check equations. In regular LDPC parity-check matrices, the total number of ones is $Mk = Nj$, because there are M rows containing k ones or N columns containing j ones. The code rate is $R = 1 - M/N = 1 - j/k$. The parity-check matrix for a regular $(3, 4)$ LDPC code is shown in Figure 2.1. This parity-check matrix $\mathbf{H}$ yields nine parity-check equations

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

(a) Parity-check matrix

$$\begin{aligned} p_1 &= y_1 + y_4 + y_7 + y_{10} \\ p_2 &= y_2 + y_5 + y_8 + y_{11} \\ p_3 &= y_3 + y_6 + y_9 + y_{12} \\ p_4 &= y_1 + y_5 + y_9 + y_{10} \\ p_5 &= y_2 + y_6 + y_8 + y_{11} \\ p_6 &= y_3 + y_4 + y_7 + y_{12} \\ p_7 &= y_1 + y_6 + y_7 + y_{11} \\ p_8 &= y_2 + y_5 + y_8 + y_{12} \\ p_9 &= y_3 + y_4 + y_9 + y_{10} \end{aligned}$$

(b) Parity-check equations

Figure 2.1: The 9×12 parity check matrix (a) corresponds to 9 parity-check equations in (b).

when it is applied to the received codeword as shown in Figure 2.1-a.

The above parity check matrix has 9 parity (syndrome) check equations performed on the received codeword $\mathbf{Y}$ and are shown in Figure 2.1-b.

Any parity-check code (including an LDPC code) may be represented by a *Tanner graph* [15], which is a visual representation of the parity check matrix $\mathbf{H}$. $\mathbf{H}$ is an $M \times N$ matrix that defines a code in which the N bits of each codeword satisfy M parity check equations. The Tanner graph contains N variable-nodes (bit-nodes), one for each bit, and M check-nodes, one for each parity check equation. The check-nodes are connected to the variable-nodes that they check. Specifically, a branch connects check-node m to variable-node n if and only if $H_{m,n} = 1$. The set of variable-nodes connected to check-node m is denoted as $N(m)$ and the set of check-nodes connected to bit-node n is denoted as $M(n)$. The graph is said to be *bipartite* because there are two types of nodes, variable-nodes and check-nodes, and there is no direct connection between any two nodes of the same type. The Tanner graph representation and the parity-check matrix $\mathbf{H}$ of a (7,3) irregular LDPC code is depicted in Figure 2.2. The check nodes are shown in squares while the circles represent the variable nodes.

Another important concept that can be seen from the graphical representation is the cycles which exist in the Tanner graph. A cycle is a set of edges that starts from a node and ends to itself. The shortest length of cycles is called the *girth* of the graph. As

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

(b) Matrix

Figure 2.2: The Tanner graph of a (7,3) irregular LDPC code and its parity-check matrix.

an example, the girth of the Tanner graph in Figure 2.2 is 4 and is illustrated by the thick bold lines. For the special case of a (j,k) regular LDPC code, each bit is involved in j parity checks. Hence, the number of branches emanating from a variable-node is always j. Similarly, because each parity check involves k bits, the number of branches emanating from each check-node is always k.

2.3.1 Decoding LDPC codes

The decoding of LDPC codes is an iterative algorithm whose complexity grows linearly with the block length of code. The sparseness of the parity matrix of the LDPC codes determines the efficient decoding of LDPC codes known as Sum-Product algorithm or message passing algorithm [9]. The message passing is a decoding technique in which messages are passed from node to node through the Tanner graph. The nodes act as independent processors, collecting incoming messages and producing outgoing messages. The computations carried out at the nodes can be done in parallel and the Tanner graph serves as the framework of the message-passing algorithm. Each message represent an estimate of the *a posteriori log-likelihood ratio* (LLR) of the variable-node associated

with it. The a posteriori LLR, defined as the logarithm of the ratio of a posteriori probabilities $Pr[c = 1|r]$ and $Pr[c = 0|r]$, can be expressed as:

$$\lambda = \log \frac{Pr[c = 1|r]}{Pr[c = 0|r]} \quad (2.6)$$

where c is a binary random variable in $\{0, 1\}$ and r is the received data. λ is positive when c is more likely to be 1 and is negative otherwise. Additionally, the magnitude of the LLR is a measure of the certainty.

When the graph is cycle free, the algorithm converges to a true LLR ratio after finite number of iterations. However, most of the codes have cycles in their Tanner graph and this causes the decoding algorithm to be an approximation. Fortunately, for good codes, the algorithm performance is quite acceptable, and the complexity is low compared to other optimum algorithms.

The decoding algorithm runs up to a maximum number of iterations and after each iteration it checks whether a valid codeword is detected; if so, it stops the iterations otherwise it continues. Hence, the algorithm stops in two cases: first if a valid codeword is detected, and secondly if the maximum number of iterations is reached. There are two types of failure: detected failures and undetected failures. Detected failures are failures where the decoder does not converge to a valid codeword after the maximum number of iterations. Undetected failures are failures where the algorithm converges to a valid codeword other than the transmitted one.

The Sum-Product decoding algorithm consists of four steps: initialization, check-node update, variable-node (bit-node) update and syndrome check. These steps are detailed as follows:

- **Initialization**

Let $u_{m,n}^{(l)}$ be the upward message from check-node m to variable-node n during the l -th iteration, and let $\lambda_n^{(l)}$ denote an estimate of the n -th LLR after l iterations. r_n also shows the channel observation for the n -th bit of the codeword whereby the

white Gaussian noise having a variance of σ^2 is added. The initializing values are:

- $u_{m,n}^{(0)} = 0$, for all $m \in 1, \dots, M$ and $n \in N(m)$;
- $\lambda_n^{(0)} = (2/\sigma^2)r_n$, for all $n \in 1, \dots, N$

The initialization process is graphically shown in Figure 2.3 where each variable node is initialized with received data.

Figure 2.3: Initialization of the variable-nodes and check-nodes

• Check-node Update

Each check node is updated according to the values which receives from the connected variable nodes and its own value stored from the previous iteration (Equation 2.7). The check node update is graphically shown in Figure 2.4 and demonstrates that the value sent from a check node to a connected variable node is computed based on all connected variable node values excluding that variable node.

$$u_{m,n}^{(l)} = -2 \tanh^{-1} \left(\prod_{i \in N(m) \setminus n} \tanh \left(\frac{-\lambda_i^{(l-1)} + u_{m,i}^{(l-1)}}{2} \right) \right) \quad (2.7)$$

• Variable-node (Bit-node) Update

The variable nodes are updated by estimating a posteriori LLR values, which can be expressed for an AWGN channel based on the equation 2.8. The updated values (λ_i) consist of two parts. The first term of the equation 2.8 is called the *intrinsic*

Figure 2.4: Check-Node update

information and represents the contribution from the n -th channel observation. The second term represents the contribution from the other bits and is called the *extrinsic information*. The intrinsic information for the n -th bit is estimated by $2/\sigma^2 r_n$ for an AWGN channel. The variable node update is depicted in Figure 2.5.

$$\lambda_n^{(l)} = \frac{2}{\sigma^2} r_n + \sum_{m \in M(n)} u_{m,n}^{(l)} \quad (2.8)$$

Figure 2.5: Variable Node update

- **Syndrome Check**

In this step, based on the updated LLR values a hard decision is made to estimate the value of b_i using:

$$\hat{c}_i = \begin{cases} 1 & \lambda_i < 0 \\ 0 & \lambda_i > 0 \end{cases} \quad (2.9)$$

After the estimation of each bit, an estimated codeword ,i.e. $\hat{\mathbf{C}}$, is formed. If this codeword satisfies all the parity check equations the algorithm is said to be converged and the syndrome is checked. The syndrome check can be represented in matrix form as:

$$\hat{\mathbf{C}}\mathbf{H}' = \mathbf{0} \quad (2.10)$$

In order to reduce the complexity of the Sum-Product algorithm, the log domain is preferred which was proposed first by Gallager [4]. The only difference is in the check-node update, given by

$$u_{m,n}^{(l)} = \prod_{i \in N(m) \setminus n} \text{sgn}(\lambda_i^{(l-1)} - u_{m,i}^{(l-1)}) \varphi \left(\sum_{i \in N(m) \setminus n} \varphi(|\lambda_i^{(l-1)} - u_{m,i}^{(l-1)}|) \right) \quad (2.11)$$

where:

$$\varphi(x) = \log \left(\frac{e^x + 1}{e^x - 1} \right) \quad (2.12)$$

In log domain, the multiplications are converted to additions, which are more numerically stable and are more robust to quantizations effects and errors. In the log domain Sum-Product algorithm, check-node update is the most computationally complex task. Two factors influences the complexity: first, the complexity of mathematical functions used in the check node update step, which are non-linear functions and second the topology of the algorithm that each check-node sends different messages to the connected variable nodes. These complexities have led to modifications to the original algorithm, to make it simpler and are named reduced-complexity sum product algorithms.

The Sum-Product algorithm is also known as the message passing algorithm and can be interpreted in terms of passing messages through the Tanner graph. Initially, the variable nodes are updated with received information from the channel and then pass them to the check nodes. Check nodes collects these information and are updated in parallel. Then the updated information are passed to the variable nodes to be updated. This type of interpretation provides a benchmark for realizing a parallel decoder.

2.3.2 Overview of the reduced complexity decoding algorithms

As discussed in section 2.3.1 the original Sum-Product algorithm is computationally intensive. This complexity can be expressed as the mathematical functions to be realized, the number of multiplication and the number of additions. In order to address this problem, several approximations have been made to it, which some more referred ones are reviewed.

I. The Min-Sum approximation

The Min-Sum approximation first proposed by Wiberg in his Phd. dissertation [15] is one of the most widely used approximations in the decoders' architectures. It can be proved using several methods [15, 16, 14]. The most straightforward is perhaps, approximation to the Gallager's check-node update in (2.11). The function $\varphi(x)$ is a symmetric function for $x > 0$ and monotonically decreasing, with $\varphi(0) = \infty$ and $\varphi(\infty) = 0$. Let denote $z_{m,i}$ as $z_{m,i} = \lambda_i - u_{m,i}$ in (2.11). The summation in (2.13) is dominated by $\varphi(|z_{m,min}|)$, where $z_{m,min} = \min\{z_{m,i}\}$. Consequently, the check node update in (2.11) can be simplified to:

$$|u_{m,n}| = \varphi\left(\sum_i \varphi(|z_{m,i}|)\right) \approx \varphi(\varphi(|z_{m,min}|)) = |z_{m,min}| \quad (2.13)$$

The min-sum approximation reduces the computational complexity of check-node update greatly by removing the computation of $\varphi(x)$ function and reducing the number of additions. However, this approximation degrades the performance of decoding as the updated values are higher than the actual ones. It should be noted that the sign of the values are computed using the same as the original algorithm.

II. The normalized Min-Sum approximation

The updated values in the Min-Sum check node update (2.13) are greater than the ones computed using the original Sum-Product algorithm, and hence needed to be suppressed.

The performance of the Min-Sum algorithm can be increased by employing a normalization factor α to (2.13) [16]:

$$u_{m,n}^{(l)} = \prod_{i \in N(m) \setminus n} \text{sgn}(\lambda_i^{(l-1)} - u_{m,i}^{(l-1)}) \frac{z_{m,min}}{\alpha} \quad (2.14)$$

Although α depends on the signal-to-noise ratio (SNR) and the number of iterations, it is kept constant during the decoding for the simplicity. The α is computed using density evaluation methods. Interestingly, the normalized Min-Sum algorithm outperforms the original Sum-Product algorithm for short and medium length codes. However, this is not surprising since the Sum-Product algorithm is not optimum for the codes with cycles which most of the LDPC codes have cycles.

III. The Offset Min-Sum approximation

Another approach to improve the Min-Sum algorithm is by introducing an offset to the value computed in the check node update as follows:

$$u_{m,n}^{(l)} = \prod_{i \in N(m) \setminus n} \text{sgn}(\lambda_i^{(l-1)} - u_{m,i}^{(l-1)}) \times \max\{z_{m,min} - \beta, 0\} \quad (2.15)$$

The offset value β can also be seen as a correction factor. In contrast to the normalized scheme, the updated check node value will be set to zero, if it is smaller than the offset and accordingly removes its effect on the next variable node update. Its complexity is similar to the Min-Sum with an extra addition for every check node update and hence it is preferred to the normalized Min-Sum which requires multiplication.

In [17], the normalized and Offset Min-Sum algorithms are evaluated for irregular LDPC codes. It is reported that the performance of the normalized Min-Sum degrades significantly at low block error rates while the Offset Min-Sum still performs well. The normalized algorithm is more subject to the message underestimation for large values of z_{min} , while the offset algorithm underestimates small values of z_{min} . However, they both underestimates significantly when all input messages are large, i.e. when SNR is high,

making the Min-Sum a better approximation for the Sum-Product in high SNR regions. According to this discussion, another method is proposed based on the Offset Min-Sum [17]. The improved offset Min-Sum tries to break the steady state when the decoding does not converge after several iterations. It is suggested to set the offset to zero, if it is not converged after 200 iteration, and perform another additional 50 iterations. It is reported that the improved Offset Min-Sum performs better than the original one for high SNR values.

IV. The degree matched approximation

Although the normalized Min-Sum and Offset Min-Sum algorithms approximate the Sum-Product algorithm well, their correction factors should be computed off-line using density evaluation methods. On the other hand, the factor should be optimized according to the SNR and the check node degrees which means a constant correction factor would degrade the performance.

The degree matched decoding algorithm [18] is an improved Min-Sum algorithm that introduces a correction factor to the Min-Sum check update, similar to the offset Min-Sum. However, the correction factor is computed on the runtime according to the LLR values and check node degrees. The maximum correction factor is shown to have a maximum magnitude of $\ln(d_c - 1)$ in order to approximate the Sum-Product algorithm, where d_c is the check node degree. Based on this fact and the evaluation of histograms for different iterations and SNRs, two versions of the degree matched decoding algorithms is presented. The correction factors are applied based on the thresholds on the LLR values. The two-step degree-matched check node approximation has two threshold determinations, based on the minimum LLR value and the next minimum LLR value. The one-step degree-matched is a simplified version of the two-step with only one threshold determination. Moreover, the threshold is applied only based on the minimum LLR

value as follows:

$$u_{m,n} = \begin{cases} \left(|\lambda_{min}| - \frac{\ln(d_c-1)}{4} \right) \prod_{i \in N_m \setminus n} \text{sgn}(\lambda_i - u_{m,i}) & |\lambda_{min}| \geq 3\ln(d_c - 1)/8; \\ |\lambda_{min}| \prod_{i \in N_m \setminus n} \text{sgn}(\lambda_i - u_{m,i}) & \text{Otherwise} \end{cases} \quad (2.16)$$

The performance results of both versions are also presented showing near Sum-Product algorithm performance for both one-step and two-step degree matched algorithms. Again, the algorithm outperforms the Sum-Product algorithm for some SNR values because the LDPC codes have cycles.

2.3.3 Quantization effects on the decoding algorithm

An issue which is not considered so far is the quantization effects on the decoding performance. It is of great importance since the final target for the LDPC decoding algorithms is hardware and logic circuits with fixed point precision. In this section, a review of the published research work in this area is presented.

In the LDPC decoder design with finite precision, the effect of quantization and data representation may drastically vary according to the decoding algorithm and even the code structure. Chen and *et. al* have studied these effects for two types of regular LDPC codes for some of the well-known algorithms in their work [16]. They have considered q bits for data representation including one sign bit and the step size of Δ which is not necessarily a power of 2. The simulated results shows that for Min-Sum algorithm a quantization of 5 bits with step size of 0.1 can achieve good results even better than the floating point and almost the same as quantization with 6 or 7 bits. Interestingly for all of these quantizations the best results are obtained when the step size is small and so the full scale range. For example, a 5 bit quantization with a step size of 0.1 which covers the range of $[-1.5, 1.5]$ has a gain of 0.13 dB to the quantization with 8 bits and step size of 0.05. The small quantization ranges are beneficial since it applies an upper-bound to the overestimated messages. It is regarded as the *clipping effect*.

The numerical results are also listed for offset Min-Sum for (3,6) and (4,8) regular

LDPC codes. The results for small and large quantization bits seem to be opposing. For small values of q , i.e. four or five bits, the best performance is gained for larger step sizes. The quantization of 5 bits with step size of 0.15 has a gain of 0.36dB over step size of 0.075. On the other hand, large q values perform best for small step sizes or equivalently fine precision, with $q=7$ the best result is obtained for $\Delta = 0.075$ which is equal to 4 fractional bits in the binary system. On average, higher number of bits shows better results than small number of bits.

The original Sum-Product algorithm expressed by Gallager's approach in (2.11) is more computational complex than the other simplified methods due to the mathematical functions used and hence requires special care when implemented in fixed-point. The reported results for regular LDPC codes in [16] demonstrates that the quantized Sum-Product suffers from error floors. The density evaluation results suggest to use two different approximations for $\varphi(x)$ in (2.11), one for the inner function and one for outer function named $\varphi_1(x)$ and $\varphi_2(x)$ respectively. The best performance is achieved when $(q_1, \Delta_1, q_2, \Delta_2) = (7, 1/8, 8, 1/64)$. The performance degrades dramatically when q_2 is reduced. The performance is also compared to the offset Min-Sum and it demonstrates that the quantized Sum-Product suffers from error floor in 10^{-6} BER region while no error floor appears at BER down to 10^{-7} and hence the Sum-Product is much sensitive to quantization errors than simplified algorithms.

In [19] a comprehensive discussion on the quantized Min-Sum algorithm is presented. In the study, the values are clipped symmetrically by a threshold of c_{th} and quantized uniformly in the range $[-c_{th}, c_{th}]$ using q bits. As discussed earlier, the clipping effect improves the performance of Min-Sum. However, the threshold should be optimized. The article has no suggestion on the derivation although it can be derived from simulations for different values. Three different LDPC codes are presented and the optimum c_{th} is claimed to be 2, 2.25 and 2.5. After selecting the optimum threshold values, the simulations are done for different quantization values. The performance curves indicates that by a quantization of $q = 5$ bits, one can obtain good BER performance.

Moreover, it is observed that the quantized Min-Sum suffers from error floors earlier than the un-quantized Min-Sum. The simulations are also performed for offset Min-Sum and compared to the Min-Sum. Again different threshold values are selected for different codes and the simulations are done for different quantization bits. It is observed that the offset Min-Sum improves the error floor by one order of magnitude and performs close to ideal Min-Sum for $q = 5$ and above.

In [20], the quantization effects on the decoding of irregular LDPC codes are studied. The article suggests a quantization of 6 bits comprising of 1 sign bit, 3 integer bits and 2 fractional bits. This quantization scheme is tested for irregular LDPC codes exploited in WLAN applications. The simulations show that both normalized Min-Sum and offset Min-Sum suffer from an early error floor, while the degradation for normalized Min-Sum is much sever. To analyze this effect, the distribution of the extrinsic messages in the decoder for different iterations are plotted. The plots show that these values are saturated after some iterations in the finite precision, i.e. they are either 8 or -8. To address this problem, a modified decoding algorithm is proposed in which uses down-scaled intrinsic information (received information). This method also improves the convergence speed.

2.3.4 Error floor of LDPC codes

According to the Shannon's theorem, an error correction code can be designed for a noisy channel based on the following: if the channel's properties, the desired rate of data transfer R and the probability of error P_e are known, one can, after working out the relevant equations, find out that there exists a solution for the problem using a particular block length N . However, this solution is non-constructive and cannot advice one how to implement the encoder and decoder and may result in a large N which makes the encoder and decoder much more complex. Additionally, the accurate characteristics of the channel may not be known. Berlekamp[21] suggests to design encoding-decoding systems and plot their performance on a variety of idealized channels for different noise levels. Later, you one choose the proper system based on these plots without having

Figure 2.6: The block error rate vs SNR plot of a rate-0.89 LDPC code compared to the other coding schemes.

to specify the accurate behavior of the studied channel. These plots are referred as *performance curves* and usually have bit error probability or block error probability on one axis and the signal-to-noise ratio (SNR) on the other axis. The performance curve of a (1998,1776) rate 0.89, derived by Mackay [22], is depicted in Figure 2.6. The performance curves of other coding schemes with the same rate also plotted.

Performance curves of iterative codes such as turbo codes and LDPC codes are well-known as *water-fall* curves in which the error probability decreases significantly with a slight increase in the signal level. Sometimes, the bottom of the waterfall is observed and is referred as the *Error-Floor*. Error-floor is usually not observed since it is beyond the reach of simulations. The source of the error-floor is identified as the near-codewords [23]. A (w, v) near-codeword of a code with parity-check matrix $\mathbf{H}$ can be defined as a vector $\mathbf{x}$ with weight w whose syndrome $S(\mathbf{x}) = \mathbf{x}\mathbf{H}'$ has a weight of v . Near-codewords with both small w and v tend to cause the error-floor.

Although the error-floor is unlikely to be observed via the conventional simulations, other methods have been introduced to achieve this goal. The hardware platforms

like FPGA or ASIC can be adopted to simulate the performance of LDPC codes [24]. Richardson [11] has presented a computational technique for the error-floor prediction. The author has identified near-codewords as *trapping sets* in the Tanner graph. These sets have a relatively small number of variable nodes such that the induced sub-graph has only a small number of odd-degree check nodes. The method first searches the Tanner graph for the trapping sets and then evaluates their contribution to the error-floor. A similar prediction technique has been introduced in [25] known as importance sampling (IS). IS is a simulation technique where the noise distribution is modified such that more samples are taken from the important regions, which are the ones that cause error-floor in this case. This technique also searches for the trapping sets and then modifies the mean of the noise, i.e. an AGWN channel, for the importance sampling.

While the prediction techniques such as IS enable one to look for the error-floor at very low block error rate regions, their efficiencies rely on the channel type. The published work mainly deal with simple channels such as AWGN or binary symmetric channels (BSC). On the other hand, the hardware-based simulations are more promising in the case of complex channels, i.e. magnetic recording channel, if the channel is modeled correctly.

2.4 Summary

In this chapter, the concept of the LDPC codes and their decoding algorithms have been introduced. The LDPC codes are a group of block codes which show superior performance in the presence of white Gaussian noise. However, their decoding algorithm is rather complex and is based on a framework known as the Tanner graph. Consequently, many simplified versions of the decoding algorithm, known as the Sum-Product algorithm, have been proposed in the recent years. Here, the most famous and widely used algorithms have been covered. The effect of the quantization in fixed point data representation have also been discussed since the main objective of this project is hardware implementation in the fixed point system.

Chapter 3

Existing LDPC Decoder Architectures

3.1 Introduction

Due to the complexity of the decoding of LDPC codes from a hardware architecture point of view, the LDPC decoders have been a challenge for hardware designers. It is probably one of the most complex modules of a communication system, which will be a magnetic recording device in the case of this project.

The LDPC decoders can be categorized into three types of architecture:

- i **Fully parallel architectures**
- ii **Partially parallel architectures**
- iii **Serial architecture**

The architecture of the decoder has a direct impact on three important issues and they are:

- i **Throughput** The inherent parallelism in the decoding of LDPC codes have made them a good option for high data rates and as a result, the throughput of the de-

coder in terms of the number of bits decoded in a second has been the top among the specifications of a decoder.

- ii **Resources** The high throughput of an LDPC decoder comes at the expense of consuming a large number of resources such as: area, memory, routing resources, etc. Hence, the goal is to minimize the resources for a given specification, such as throughput.
- iii **Flexibility and reconfigurability** The architecture of decoders highly relies on the LDPC code while many applications require different type of codes for different situations. So the decoder should support a wide range of codes. This task is probably the most challenging.

In this chapter, focuses will be placed on how these opposing factors are addressed in some published works. Although the target of this thesis is on FPGA implementation, the discussions are not confined to this and ASIC implementations will also be discussed.

3.2 Parallel LDPC decoders

The message passing algorithm is, inherently, a parallel algorithm that sends messages through the Tanner graph. Therefore, the most straightforward implementation is to design it based on the Tanner graph such that every node is a processor which sends or receive messages from or to connected nodes. However, this type of implementation is not easy due to the massive connections. Among the published works such as [26, 27, 28, 29], few have adopted such an architecture, with the most referred work being [26].

The LDPC decoder described in [26] is based on a 1024-bit, length 1/2 LDPC code with 3328 edges in the Tanner graph. The number of variable nodes is 1024 and is the same as the number of columns in the parity check matrix and the number of check nodes is 512. In this parallel architecture, 1024 processing elements (PEs) are implemented for 1024 variable nodes and 512 for check nodes. The interconnections between these two types of PEs are determined by the code (Tanner graph) and the algorithm to be engaged

is the Sum-Product algorithm. The architecture of the reported decoder is shown in Figure 3.1. Only one check node and one variable node are shown in the Figure for the sake of simplicity.

Figure 3.1: The architecture of the parallel decoder in [26].

The messages from check nodes to variable nodes and the messages from variable nodes to check nodes are sent on specific wires. The messages are shown in 4 bits and accordingly $3328 \times 4 \times 2 = 26624$ wires are required in the datapath. Note that registers are only placed at the output of the variable nodes for synchronization.

The decoder is designed using 0.16- μm 1.5V-CMOS process with five levels of metal. The vast amount of wire interconnections and large number of PEs had led them to develop special software to assist the CAD tools in the floor planing and routing. The variable nodes that contain registers are placed around the chip whereas the combinatorial check nodes are located in the middle of the chip. The placement of the check nodes are optimized using a custom software to reduce the distance of each check node to the connected variable nodes. Another software is also developed to insert buffers in the long wires and to conduct the router to reduce congestion.

The implemented chip is run at a clock frequency of 64 MHz. The decoder performs one decoding iteration per clock cycle on a block of 1024 bits. Therefore, the throughput is $64\text{MHz} \times 1024\text{bit} = 64\text{ Gbit/sec/iteration}$. Since 64 decoding iterations are done per

block, the overall throughput is 1 Gbit/sec. The chip area of 52.5 mm^2 and a power dissipation of 690 mW are both dominated by the routing. Although the throughput of the decoder is relatively high, the large area and power consumption are the main drawbacks of the decoder. The fully parallel decoder is also code dependent. This means that it supports only one code and if another code is considered, all the entire architecture (processing nodes and routing) should be redesigned. Finally, the implementation of such a decoder is realized only with the aid of special softwares. The discussed disadvantages limit the application of the fully-parallel LDPC decoders significantly and make them less popular in the research world.

3.3 Partially-parallel LDPC decoders

Direct implementation of an LDPC decoder such as that of [26] has presented a lot of challenges. The complexity and behavior of the mathematical function $\varphi(x)$ (2.14) requires a sufficient number of bits for data representation. This increases the size of a check node and the interconnections linearly. Moreover such a parallel decoder cannot be realized for large codes due to the routing congestion. For example, the average length of wires of [26] is 3 mm and the chip area is 52.5 mm^2 while the code length is only 1024.

To overcome these difficulties, partially-parallel LDPC decoders have been introduced. In partially-parallel decoders the function units are shared, i.e. a check node processor is assigned to a number of check nodes and so is a variable node processor. The communication between these function units is also done via shared resources such as shared memory or interconnection network. In this section, we shall have an insight into the contemporary partially-parallel architectures.

3.3.1 A 54 Mbps (3,6)-regular FPGA LDPC decoder

A common methodology for the design of LDPC decoders is the joint design of code and decoder and hence this method is also named the *architecture aware* code design [10].

This method is in contrast to the random code construction [8] and is used in the design of one of the first LDPC decoder implementations in [30]. The decoder is intended for (3,6) regular LDPC codes and can achieve a throughput of 54 Mbps.

The parity check matrix of the (3,6) LDPC code is constructed based on three sub-matrices $\mathbf{H}_1$, $\mathbf{H}_2$ and $\mathbf{H}_3$, where each of which corresponds to a (1,6) regular code. Accordingly, the code has a parity check matrix, $\mathbf{H} = [\mathbf{H}_1^T, \mathbf{H}_2^T, \mathbf{H}_3^T]$. $\mathbf{H}_1$ is constructed of identity matrices and $\mathbf{H}_2$ is constructed of cyclic shift of identity matrices and $\mathbf{H}_3$ is constructed randomly in such way that the code has a girth of greater than 4. The proposed code has a length of 9216 and rate of 1/2.

The decoder architecture shown in Figure 3.2 consists of thirty six *variable node processor units* (VNUs), eighteen *check node processor units* (CNU) and three shuffle networks, π_1 , π_2 and π_3 . Each VNU is also connected to a RAM that stores the information of $9216 \div 36 = 256$ variable nodes. The bidirectional shuffle networks, π_1, π_2 and π_3 exchange messages between the variable nodes and check nodes. CNU perform the check node computations. As seen in Figure 3.2, each RAM has three inputs/outputs to the shuffle network and this corresponds to a variable node degree of 3 with a depth of 256. The decoder performs a half iteration in 256 clock cycles since every VNU is assigned with 256 variable nodes. Totally, 512 cycles are required for a single iteration which is comprised of five phases: *read*, *shuffle*, *modify*, *unshuffle* and *write*. All the decoding information are quantized to 5 bits.

The decoder is implemented on Xilinx XCV2600E FPGA and can run up to 56 MHz achieving a throughput of 54 Mbps for 18 iterations. The architecture used can be seen as a framework for designing partially parallel LDPC decoders. However, it introduces some limitations. The decoder design is combined with the code design, hence it should be changed for different codes. The decoder only supports regular (3,6) codes. The article also does not cover the implementation of the mathematical functions in check node updating.

Figure 3.2: The architecture of (3,6) regular LDPC decoder [30].

3.3.2 High speed LDPC decoder for Quasi-Cyclic codes

The high speed LDPC decoder reported by [31] involves a decoder that is designed for the quasi-cyclic (QC) LDPC codes. Quasi-cyclic LDPC codes can perform close to the random generated codes, with comparable performance. The parity-check matrix of the QC-LDPC codes contains the shifted versions of the identity matrix and hence it is structured. This structure allows us to design the decoder easily while keeping the throughput as high as possible. Another benefit of the quasi-cyclic codes is that their encoder can also be designed easily with shift registers. A group of Quasi-Cyclic codes presented in [32] have a parity-check matrix structure of:

$$\mathbf{H} = \begin{bmatrix} I(0) & I(0) & \dots & I(0) \\ I(0) & I(p_1, 1) & \dots & I(p_1, L-1) \\ \vdots & & & \vdots \\ I(0) & I(p_{J-1}, 1) & \dots & I(p_{J-1}, L-1) \end{bmatrix} \quad (3.1)$$

where for $1 \leq j \leq J-1$, $1 \leq l \leq L-1$, $I(p_j, l)$ represents the circulant permutation matrix with a one at column- $(r+p_{j,l}) \bmod p$ for row- r . $I(0)$ also stands for an identity matrix of $p \times p$. Consequently, $\mathbf{H}$ is the parity check matrix of a (J, L) regular LDPC

code.

A modified sum-product algorithm is also presented which balances the computational load between the check-nodes and the variable-nodes. Recall that $u_{m,n}$ is the message from check node m to variable-node n ; and $r_{n,m}$ is the message sent from variable node n to check node m . The proposed check node update can thus be expressed as:

$$u_{m,n} = \prod_{i \in N(m) \setminus n} \text{sgn}(r_{i,m}) \sum_{i \in N(m) \setminus n} \varphi(|r_{i,m}|) \quad (3.2)$$

And the modified variable node update:

$$r_{n,m} = \frac{2}{\sigma^2} r_n + \sum_{j \in M(n) \setminus m} \text{sgn}(u_{n,j}) \varphi(u_{n,j}) \quad (3.3)$$

This modification makes the computation of the check-nodes and bit-nodes identical and hence only one type of processing element is needed. In other designs, the processing elements of variable-nodes and check-nodes are separated, and only one set is working at a time. On the contrary, the processing elements are working all the time for the design of [31]. The mathematical function $\varphi(x)$ is realized using lookup tables.

The architecture of the partially-parallel decoder at [31] is depicted in Figure 3.3. The decoder is designed for a Quasi-Cyclic code with a parity check matrix constructed from 32 sub-matrices in a 2×16 manner. Accordingly, two *check node processing units* (CPU) and sixteen *variable-node processing units* (VPU) are considered. CPU1 is responsible for the first 1/2 of the row updates and VPU1 is responsible for updating of the first 1/16 of columns. Other CPUs and VPUs are assigned in the same method. The CPUs and VPUs are connected via shared memories. For each of the sub-matrices there is a memory that stores the messages of the edges defined by it. In a straightforward decoding approach, each CPU (VPU) updates a single row (column) in a cycle but in this design the CPUs (VPUs) perform multiple updates within a cycle to increase the throughput.

This scheme mainly deals with memory partitioning since it tries to process two

Figure 3.3: The enhanced partially parallel decoder architecture for QC-LDPC codes [31].

adjacent rows in a sub-matrix at the same time to increase the throughput. The first solution to this problem is by assigning one memory entry to two adjacent messages. For example, data of row 1 and 2 are placed at the memory position of 1. However, for the processing of columns this may cause a conflict since the adjacent elements in rows may not be adjacent elements in columns. In order to overcome this problem, two methods are proposed. In the first approach, each memory bank is partitioned into $p = 2$ sub-banks, where all the messages of p adjacent rows are stored in different banks. The second solution suggests to store p adjacent rows' data in one memory location and to resolve the conflict using switches. For the second case, different switching networks are presented. In summary, using these two approaches have lead to multiple speedup in comparison with designs where one row of a sub-matrix is processed each time.

Based on the [31] architecture, a decoder for a (8176,7154) QC-LDPC code, which can operate up to 192MHz, was implemented on FPGA. The number of slices occupied is 23052, 26926 flip-flops, 28299 4-input lookup tables and finally 128 block RAMs. The achieved throughput is 172 Mbits/sec for 15 iterations. This high throughput is a result of parallel processing of rows and columns in a sub matrix. The drawback however is

that the decoder supports only regular Quasi-Cyclic codes while other types, which have sub-matrices with different sizes or zero sub-matrices, are introduced for communication standards.

3.3.3 Memory efficient decoder architecture for QC-LDPC codes

Another partially-parallel LDPC decoder architecture for Quasi-Cyclic codes is presented in [33]. It is able to reduce the memory usage by 30 % for high rate codes. This memory saving is gained by re-arranging the normalized Min-Sum algorithm as follows:

$$u_{m,n}^{(k)} = \alpha \times \prod_{i \in N(m) \setminus n} \text{sgn}(\lambda_i^{(k-1)} - u_{m,i}^{(k-1)}) \times \min_{i \in N(m) \setminus n} |\lambda_i^{(k-1)} - u_{m,i}^{(k-1)}| \quad (3.4)$$

$$\lambda_n^{(k)} = \frac{2}{\sigma^2} + \sum_{i \in M(n)} u_{i,n}^{(k)} \quad (3.5)$$

where $u_{m,n}$ is the check node m to variable node n message and λ_n is the updated variable node. In this formulation every check node is updated with a single value, instead of multiple-messages to connected check nodes, leading to memory saving.

The decoder's architecture is shown in Figure 3.4. The R memory stores the updated check node values in compressed form. Every entry of the memory, assigned to a check node, contains four elements: 1) minimum value, 2) the second minimum value 3) The *index* of the second smallest value and 4) the sign of messages. In comparison with storing the individual data, one can derive the second method of memory saving. The *S-memory* stores the λ_i values and *I-memory* contains channel read data.

The check node update is done row-wise, i.e. a check node is updated in a cycle, and that is why only one CNU is illustrated. After a row is updated, the values are sent to the VNUs and they are updated partially. In this manner, the variable node update is also row-wise resulting in $2 \times \text{rows}$ cycles for a single iteration.

The main task of the CNU is to find the minimum and the second minimum values

Figure 3.4: Memory-efficient partially-parallel decoder architecture [33].

in a set of data in parallel. For this task, an optimized sorter is presented which also finds the index of the minimum value. The *data scheduling unit* determines the permutation of data in message delivery, which is based on the cyclic structure of the sub-matrices, and has a similar structure to the barrel shifters [34].

The proposed architecture reduces the memory usage by means of compressing the check node values. Consequently more memory saving can be attained when the check node degree is high. This happens for high rate codes. As an example the architecture is reported to save 37 % of the memory for a 4608 length and 8/9 rate code. The latency is also linearly proportional to the rows number. It therefore performs best for high rate codes that have fewer number of rows. Although, the architecture's concept and idea is novel, it is not synthesized, and the actual throughput and performance are unknown.

3.3.4 TDMP decoding for LDPC codes

Mansour and Shanbhag in their extensive work [10] have proposed a decoding algorithm for LDPC codes and its architecture. The new method known as *Turbo-Decoding Message-Passing (TDMP)* or *layered* algorithm is established based on the analogy be-

tween LDPC codes and the turbo codes that improves the throughput of the decoding. The new concept is the foundation for many other published works, that will be discussed later.

The TDMP decoding algorithm is proposed for structured and regular codes for simplicity. These codes are called *Architecture-Aware*, and such a code is illustrated in Figure 3.5. The parity check matrix corresponds to a regular (4,2) LDPC code that consists of three *row blocks* and six *column blocks*. The row blocks can be seen as a parity check matrix of having four parity check equations in which four variable nodes take part. Accordingly, each of the row blocks can be seen as a *super code* or a *layer* and named as C^1 , C^2 , and C^3 .

$$H = \left[\begin{array}{cccc|cccc|cccc|cccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right] \begin{array}{l} \left. \begin{array}{c} \\ \\ \\ \end{array} \right\} \mathbf{c}^1 \\ \left. \begin{array}{c} \\ \\ \\ \end{array} \right\} \mathbf{c}^2 \\ \left. \begin{array}{c} \\ \\ \\ \end{array} \right\} \mathbf{c}^3 \end{array}$$

Figure 3.5: The architecture-aware parity-check matrix for TDMP decoding.

The simplified decoder architecture is depicted in Figure 3.6 and the decoding procedure is described with reference to this figure. In the layered decoding algorithm, each iteration consists of c sub-iterations and each of which is performed on a layer, or equivalently a super-code is decoded in each sub-iteration. Every super-code can be seen as a parity-check matrix with variable nodes of degree one and the conventional decoding algorithms like Sum-Product can be applied to it. However, in the super-codes, the variable node degree is one that eliminates the variable node update phase. Consequently, the decoding is single-phase and established by the soft-input soft-output (SISO) de-

Figure 3.6: The TDMP algorithm architecture showing the message passing and SISO decoding.

coders D_1 , D_2 and D_3 of Figure 3.6. The updated values are stored in different memory banks for each sub-iteration denoted by $\underline{P}_i$ and then added to the channel observation values (λ) to compute the variable node update value. The input to the SISO decoder in the k -th sub-iteration is the updated LLR values minus the values stored in $\underline{P}_k$ from the previous iteration to minimize the correlation. Because of this operation, different memory banks are needed for the layers. For more details on the algorithm, refer to [10].

The throughput of the algorithm is compared to the conventional message passing algorithm for some regular LDPC codes. The simulations reveal that for the same SNR, TDMP requires less iterations to achieve the same BER. This enables one to reduce the number of iterations for the decoding task, leading to an increase in the throughput. From the memory point of view, it is claimed that the decoder saves more than 50% over other architectures since it only stores the variable-to-check values while other architectures also store check-to-variable values. This is not always true and depends in architecture. As an example, the decoder in subsection 3.3.2 uses shared memory for message passing which stores check-to-variable values in one phase and variable-to-check values in the other phase.

Based on the algorithm and architecture discussed, an LDPC decoder chip is de-

signed and implemented by the same authors in [35]. The decoder is intended for an LDPC code with a parity check matrix of 2048-by-1024 (3,6) regular. The code has 32 column-blocks and 16 row-blocks (layers) of 64×64 sub-matrices. Figure 3.7 shows the decoder architecture. It consists of a Soft-Input Soft-Output (SISO) engine which has 16 quad-SISO modules that processes 64 rows of a super-code. The SISO engine locally accesses to the intermediate (intrinsic) values stored in λ -SRAM. Since the intrinsic values for every sub-iteration is written in the same fashion as it is read, no permutation is required. The Γ values (the reliability of the variable nodes) are stored in the Γ -Memory, and are updated during every sub-iteration that is needed for the following iteration. Therefore, they should be permuted according to the structure of the super-code. A key structure in the decoder is the bi-directional network which consists of six layers of shuffle-exchange network where each of which is controlled by the π_i -memory (six of them totally). The SISO module computations are different from the Gallager's approach in (1.13) and is based on the BCJR algorithm. It is mentioned that this modification is more immune to the quantization errors, enabling one to use less quantization bits, which is four in the design.

The chip was implemented on 0.18- μm CMOS and attains a throughput of 640-Mbps for 10 iterations running at 125 MHz. The chip area is 14.3 mm^2 and is dominated by the bi-directional network (3.28 mm^2) and the SISO Engine (5.16 mm^2). Detailed description of the bi-directional network and the SISO Engine at transistor-level is provided and they are optimized [35]. However, one may further reduce their area by exploring different architectures. The only flexibility that the decoder has is its ability in supporting different code rates from 8/16 to 14/16. However, the column degree of the code should be 3 at most, and this forces a limit on the code design.

Based on the layered decoding algorithm, an architecture is proposed for the decoding of the LDPC codes defined by the IEEE standard IEEE 802.11n which has three code lengths of 648, 1296 and 1944, with code rates of 1/2, 2/3, 3/4 and 5/6 [36]. The code is also irregular and hence it has to be flexible to some extent. The parity check matrix of

Figure 3.7: The chip architecture of TDMP decoder [35].

the code can have three different sizes of the sub-matrices: 27, 54 and 81 which also define the size of the layers. In order to support all of them, the decoder is implemented for the worst case, i.e. maximum code size and layer size. The code is irregular so the check update phase is not done on a fixed number of values. Consequently, the check node processors have a serial architecture that accepts a single input at every cycle and can operate for check nodes of different degrees. The algorithm used is the offset Min-Sum which reduces the complexity significantly in contrast to that of [35].

The decoder is implemented on Xilinx Virtex4 FPGA, occupying 15235 slices and 123 Block RAMs. It also exploits a three stage pipeline in the CPUs. This introduces a little hardware overhead since the data should be forwarded in the pipeline stages. The maximum throughput achieved is 292 Mbps for code 1944-length and 5/6-rate without the pipeline. It is claimed that the pipeline increases the throughput by 3 times. However, no implementation results have been provided. The decoder is also synthesized

on 0.13- μm CMOS process, which consumes 3.68 mm^2 silicon area. This is 25% of the architecture reported in [35]. This saving is mainly due to the simplified algorithm (Min-Sum) and the network architecture. However, the proposed decoder is not fabricated.

A versatile LDPC decoder employing the TDMP approach is proposed by Fewer *et al.* [37] that exploits a joint code, encoder and decoder approach. Another architecture based on this approach is presented by Zhiyong *et al.* [38] in which a single row is considered as a layer offering more flexibility. Hocevar [39] has discussed a decoder for quasi-cyclic codes based on the layered decoding method.

3.3.5 An LDPC decoder for IEEE802.16e based on self-routing network

The IEEE 802.16e standard [40] has considered employing LDPC codes to improve its performance. According to the standard specifications, the LDPC code has different lengths ranging from 576 to 2304 and rates of 1/2, 2/3, 3/4 and 5/6. The code is structured and is composed of sub-matrices of the same size which are either a zero matrix or a right-shifted identity matrix. The size of the sub-matrices also can be one of the nineteen different values ranging from 24 to 96. Moreover, the code is irregular, hence the design of a decoder capable of supporting all of these modes is challenging.

Liu *et al.* [29] have recently published an LDPC decoder chip that aims for the IEEE 802.16e standard and supports all the different modes of the code. The decoder's architecture (Figure 3.8) is based on the maximum sub-matrix size (ninety six), which influences the complexity of both the check node and the variable node processors as well as the permutation networks. However, such a large size is inevitable as far as the performance and flexibility is concerned.

The work introduces a decoding schedule named *phase-overlapping message passing*, which combines the two phases of the message passing algorithm in such way that check node update and variable node update can be done simultaneously. The parity check matrix is divided into layers and two of them are processed at every instance. The two updated layers are used for the partial update of the bit nodes in the bit node pro-

Figure 3.8: The architecture of LDPC decoder [29].

cessor (BNP) and hence check node update and variable node update are overlap each other. The decoder also employs a four stage *self routing* permutation network which is claimed to be more area efficient than the barrel shifters. However, it occupies around 10% of the chip area.

The decoder is fabricated on the 90 nm CMOS process and occupies 6.25 mm^2 of silicon area, and is dominated by the memories. The decoder runs at 150 MHz and achieves a throughput of 105 Mbps for 20 iterations. The messages in the decoder are also represented by 6 bits.

3.3.6 Min-Sum Decoder for high rate Quasi-Cyclic LDPC codes

A recent work by Hao and *et al.* [28] is reported to tackle the relatively high area cost of conventional LDPC decoders with an aim to increase the throughput. The proposed LDPC decoder is intended for high rate regular Quasi-Cyclic codes that are often used in

magnetic recording devices. The final design is synthesized on 65-nm CMOS for regular 8/9-rate QC codes with lengths of 512, 1024, 2048 and 4096.

The decoder's partially parallel architecture, which is shown in Figure 3.9, deals with the area cost by addressing two factors: decoding algorithm and the storage memory for messages. The algorithm employed is the Min-Sum algorithm in which the minimum values are found in a serial manner. Moreover, the check node update and variable node update overlap each other eliminating the memory required for storing variable-to-check values. Thanks to the Min-Sum algorithm, the check-to-variable node values are stored in a compact form for the current and previous iteration, in Register Array A and Register Array B, respectively. The decoding task is done column-wise where s variable nodes are updated in one cycle by means of s parallel-input VNUs and the updated values are sent to the CNUs for the partial update of check nodes. Recalling this fact, a single iteration is eventually done in $Columns/s$ cycles.

Figure 3.9: The partially parallel LDPC decoder architecture in [28].

The decoder is synthesized in 65-nm CMOS process and can run up to 300 MHz achieving a throughput of 2.1 Gbit/s for 16 iterations, which is relatively higher than other reported designs [41, 35, 36, 37, 29]. The decoding messages are quantized in 4 bits, and according to [16] this may not be sufficient. The main contribution of this work is the overlapped message scheduling that can reduce the storage memory, to yield a 100% hardware utilization ratio. Nevertheless, the implementation details especially

for the interconnection networks, which seems to be complex, are not provided in the article.

3.3.7 FPGA simulator for the evaluation of Quasi-Cyclic codes over magnetic recording channel

Although the superior performance of LDPC codes have been demonstrated, their performance in very low bit error rate regions is still open to question. In order to investigate their performance, intense simulations are required. But, the traditional C based simulations are very time consuming due to the iterative decoding of LDPC codes. The hardware based simulations have gained attraction recently, in which the LDPC decoder improves the simulation speed significantly.

A recent work that addresses the investigation of Quasi-Cyclic codes over the magnetic recording channel is published by Hao *et al.* [42]. The work is valuable since it observes the effect of some basic properties of the LDPC codes such as the column weight on the code's performance. The system has an architecture shown in Figure 3.10 and consists of two distinct FPGAs. The first FPGA models the magnetic recording channel as an extended partial response class 4 (EPR4) channel in the presence of white Gaussian noise. This type of simple channel model is ideal but will cause an increase in the simulation speed. The EPR4 simply introduces the inter-symbol interference (ISI) by weighted addition of successive symbols (like a finite impulse response filter). The AWGN generator is based on the Box-Muller method [43] and is realized using lookup tables. The Box-Muller method transforms a uniform distribution to the Gaussian distribution using mathematical equations. The uniformly distributed numbers can be simply generated by the Linear Feedback Shift Registers (LFSRs).

The detector and the LDPC decoder are implemented on the second FPGA. The detector is a parallel max-log-MAP detector to gain a high throughput and contains multiple sub-detectors working in parallel. The output of the detector, which is soft information, is fed to the LDPC decoder and is iteratively decoded. In order to increase

Figure 3.10: FPGA-based simulator presented in [42].

the performance, the LLR values at the output of the detector, may be fed back to the detector. This type of operation is called a *global iteration*. The input to the simulator are random codewords generated by the host PC.

One drawback of the hardware based simulators is the quantization effects which does not exist in the floating point C simulations. In the presented work, the output of the AWGN is quantized into 6 bits. The detector's datapath is of 9 bits whereas the output of the detector and the decoder's datapath are both of 6-bit values.

In order to achieve a fast simulation, the simulator requires a high throughput Quasi-Cyclic LDPC decoder. The conventional Quasi-Cyclic LDPC decoders have m CNUs and n VNUs provided that the parity check matrix is partitioned into $m \times n$ $p \times p$ sub-matrices. Each CNU (VNU) is responsible for the update of p check nodes (variable nodes) and hence a single iteration takes $2p$ cycles. This may limit the throughput if p is large. To overcome this problem, the proposed decoder consists of h CNUs (VNUs) for updating of p check nodes (variable nodes) as illustrated in Figure 3.11. This type of architecture delivers a speedup of h times, where h is a factor of p . The decoder uses shared memory for the message passing. Each memory bank stores the messages of a sub-matrix in which a memory entry is comprised of the messages corresponding to h edges of the Tanner graph. This configuration also requires special memory addressing that are discussed in the article [42].

The simulator is employed to investigate the performance of some high rate Quasi-Cyclic LDPC codes having lengths of greater than 4000. The simulations reveal that

Figure 3.11: Quasi-Cyclic LDPC decoder module in the FPGA-based simulator [42].

LDPC codes with column weights of 3 badly suffers from error floor, whereas codes with column weight of 4 are more robust against the error floor. The effect of sub-matrix size, p , and the weight of the sub-matrices, γ , have also been studied. The results suggests that higher values of p performs better and γ should be at least one, i.e. there should exist no zero sub-matrices. Obviously, the codes with lower rates obtain better performance in comparison to very high-rate ones.

The simulations can be carried out by the FPGA-based simulator thousands of times faster than the software-based simulations. However, the results may not be sufficiently accurate due to two factors: 1) The encoder is implemented in the host PC and every generated codeword is used ten times to prevent it from becoming a bottleneck; and 2) The channel model is very simple and ideal. These properties can be further improved to obtain more accurate results.

3.4 Serial LDPC decoders

The serial LDPC decoders are the other end of the LDPC decoders, in contrast to fully parallel ones. In such decoders [44, 45], there exist only one check node processor (CNP) and one variable node processor (VNP) to perform the operation of check node update and variable node update one at a time, respectively.

The architecture of a serial decoder presented in [44] is shown in Figure 3.13. The

Check Node reads the values of q Memory sequentially. The updated values are then written to the r Memory and are accessed by the variable node in the following phase. The Bit Node reads the updated values along with the received data and performs the variable node update phase. The memories are single port and thus a node update takes the number of cycles equal to its degree.

Figure 3.12: Block diagram of a serial LDPC decoder presented in [44].

Although the throughput of serial decoders is relatively low (few Mbits/sec), they provide the most flexibility in terms of supported LDPC code and are capable of supporting codes with any number of nodes, which may have arbitrary degrees and connections. This fact make them a good candidate for rapid investigation of LDPC codes and decoding algorithms performances. Such a decoder is discussed in chapter 3.

3.5 Other Alternatives

The LDPC decoding task is not limited to highly-customized targets such as ASICs and FPGAs. It can be done on different software platforms ranging from general purpose processors to high-end super computers. Although the throughput is sacrificed on the processor-based systems, the flexibility and ease of implementation is a great advantage over hardware designs. This is of special interest in code studies and simulations. In this section, two alternatives to hardware-only decoders are discussed briefly.

3.5.1 Stream oriented LDPC decoding

The stream-based processing has become popular in personal computers due to the great demand for gaming and multimedia applications. In this type of computation, the same operation can be performed on a number of data simultaneously. For example, vector addition, which is frequently employed in graphical applications, consists of a number of scalar additions on the vector elements. This type of parallelism is known as Single Instruction Multiple Data (SIMD) type. Modern processors, e.g., Intel's Pentium, are capable of supporting SIMD operations via the Streaming SIMD Extensions (SSE), which are basically extensions to their instruction set.

In a recent work [46], the use of SSE for the LDPC decoding has been examined. The basic idea is to decode N codewords in parallel using the SIMD instructions. For this task, the $\mathbf{H}$ matrix is also transformed into a structure that is efficient for streaming. The decoder is implemented on the x86 CPU, with multimedia extensions (MMX), allowing it to decode 4 codewords in parallel. The speedup over the serial decoding depends on the code and the number of edges of the Tanner graph. Based on the results presented, the speedup for large codes is far less than the expected 4x, mainly because of the CPU's cache performance. The number of cache misses increases abruptly for codes with a number of edges larger than a certain amount. It is stated by [46] that decoding of LDPC codes, with less than 14688 edges, on a CPU with 2 MB of L2 cache, can yield a gain of nearly four times.

3.5.2 Decoding LDPC codes on the GPUs

The increasing pressure from the gaming market has led to the vast development of graphics processing units (GPUs) over the past decade. The GPUs have been developed with an aim for massive parallel applications like the vector computations in the graphics. In a GPU silicon die, more space is dedicated to the processing cores rather than the memory and flow control. The recent sub-micron technologies have allowed the GPUs to implement several hundreds of processors on the same chip. This has made the GPU

an ideal platform for high performance computing and therefore, efficient development tools like CUDA [47] have been introduced.

Falacão and *et al.* [48] have deployed the GPUs in the decoding of LDPC codes. In their work, the $\mathbf{H}$ matrix is transformed into a compact representation that also enables streaming of data into several cores by allowing different threads to access the $\mathbf{H}$ information simultaneously. The implementation results show that speedups of 19 to 45 are achievable for LDPC codes having 3072 to 14688 edges decoded in 10 iterations. The highest speedups reported is when 100 decoding iterations have been carried out, which ranges from 292 to 730 times.

Although the GPU approach looks promising, careful examination of the results reveals that the maximum throughput per iteration is around a couple of tens of Mega-bits per second while this parameter is at least several hundred Mega-bits per second for dedicated VLSI hardware. Nonetheless, the low cost and ease of implementation (software programming techniques) makes it an alternative for some applications.

3.6 Summary

In this chapter a brief review of the existing works related to the LDPC decoding architectures have been discussed. The LDPC decoders fall into three categories: fully parallel, partially parallel and serial based on how the Tanner graph is implemented. The parallel decoders deliver the ultimate throughput but the routing complexity is a barrier for implementing such decoders. The partially parallel decoders compromise between the throughput and the ease of implementation as well as the flexibility. The serial decoders, which are the most flexible with the minimum area, suffer from low throughput.

The most elegant works in the field of LDPC decoding are joint designs in which a class of LDPC codes are selected, then decoding algorithm is modified or re-scheduled in such way that it becomes more suitable for hardware realization and finally the architecture is proposed. Most of the works address architecture-aware LDPC codes such as Quasi-Cyclic codes. However, the performance gap between structured codes and ran-

	Platform	Architecture	Code	Throughput/Iter.	Clock Frequency	Area\ Resources
Blanksby [26]	ASIC 0.16 μ m	Parallel	(1024,512) Unstructured	64 Gbit/s	64 MHz	52.5 mm ²
Zhang [30]	FPGA XCV2600E	Partially- Parallel	(9216,4608) (3,6)-regular	972 Mbit/sec	56 MHz	11792 slices 90 BRAMs
Wang [31]	FPGA XC2V6000	Partially- Parallel	(8176,7154) QC	2.58 Gbit/s	192 MHz	23052 slices 128 BRAMs
Mansour [35]	ASIC 0.18 μ m	Partially- Parallel	(2048,1024) QC-irregular	6.4 Gbit/s	125 MHz	14.3 mm ²
Karkooti [36]	FPGA XC4VFX60	Partially- Parallel	(1296,864) QC-irregular	500 Mbit/s	160 MHz	15235 slices 123 BRAMs
Fewer [37]	FPGA XC4VLX40	Partially- Parallel	(6912,5184) Block-structured	4.16Gbit/s	181 MHz	10970 slices 46 BRAMs
Liu [29]	ASIC 90 nm	Partially- Parallel	(2304,1920) QC-irregular	2.1 Gbit/s	150 MHz	6.25 mm ²
Hao [28]	ASIC 65 nm	Partially- Parallel	(4608,4096) QC-regular	33.6 Gbit/s	300 MHz	1.92 mm ²
Levine [44]	FPGA Virtex	Serial	(1200,300) Unstructured	10 Mbit/sec	50 MHz	3039 LUTs 20 BRAMs

Table 3.1: The performance and specifications of some existing LDPC decoders.

dom codes in very low bit error rate regions is not investigated yet and can be considered as a research topic. A summary of reported architectures are tabulated in table 3.1 .

Chapter 4

The Implementation of a Simplified Sum-Product Decoding Algorithm

4.1 Introduction

The superior performance of LDPC codes and their high throughput decoding algorithm due to the inherent parallelism have made them an excellent option for the communication systems. Their adoption in the communication standards such as DVB-S2 [49] and IEEE 802.16e [29] have been reported. However, the application of LDPC codes greatly relies on designing good codes. Unfortunately, there is no general procedure to design good LDPC codes. The successful design of an LDPC code based on designated rules (such as avoiding low girth) requires rigorous testing and simulation to ensure meeting of bit error rate (BER) requirement. This task is computationally intensive and time consuming when carried out using C or MATLAB programs. The bottle-neck of such simulations is in fact the LDPC decoder since it is iterative. The primary focus in this project is hence on the decoder design.

For the purpose of code performance evaluation, the key factor of the decoder is flex-

CHAPTER 4. THE IMPLEMENTATION OF A SIMPLIFIED SUM-PRODUCT DECODING
52 ALGORITHM

ibility. The ease of reconfiguration for any given code is also desirable. Since the real-time functionality is not a concern in simulations, the reduction of throughput, which is a consequence of having greater flexibility, is acceptable. In [50], a field-programmable gate array (FPGA) based simulator is presented. This architecture is capable of investigating the bit error rate (BER) performance of LDPC codes at 10^{-10} . The decoder in the system is partially parallel and can support only structured LDPC codes. Another hardware-based simulator is presented in [42] where the decoder is designed for high rate quasi-cyclic codes.

This project aims to build an FPGA-based LDPC decoder that can support high rate unstructured LDPC codes for the purpose of BER performance evaluation. For the evaluation of LDPC codes, it is not appropriate to design a different decoder for different codes. A flexible decoder that can be readily reconfigured and programmed is, therefore, highly desired. In order to support unstructured parity check matrices, a serial architecture is chosen with an aim to achieve maximum throughput. The decoder is proposed as a peripheral that is instantiated in an embedded processor system. The decoder is connected to the local bus of the embedded processor of the FPGA which is the PowerPC of the Virtex-II Pro FPGAs. The decoder is configured and controlled by software. The advantage of allowing the design peripheral-based is that it can be reused in a system where other peripherals are connected to the processor bus by means of available EDA tools. Such a Hardware/Software system also allows users to simply add other functionalities that are not time-critical by writing C code for the embedded processor. Furthermore, the user does not need to modify the Bitstream for different codes. The reconfiguration is done through re-programming FPGA internal and external memories on-fly, which means the synthesis and implementation phases of the design are done only once. Other than the Hardware/Software implementation, a simplification to the Sum-Product decoding algorithm (2.11) is proposed to make it more appropriate for the hardware implementation.

4.2 The simplified LDPC decoding algorithm

The Sum-Product algorithm is an efficient decoding algorithm of LDPC codes. The parity-check matrix can be viewed as a bipartite graph having two kinds of nodes: check-nodes that correspond to the rows and variable-nodes, resembling the columns of the parity-check matrix. At each iteration, the check-nodes receive messages from variable-nodes and then pass updated values back to the variable-nodes. Decoding ends when all parity check equations are satisfied or a predetermined number of iterations has been reached. Denoting λ_i as the log-likelihood ratio (LLR) of variable node i , $u_{m,n}$ as the message passed from check-node m to variable node n and r_n the receiver's observation, the check-node update can be written as:

$$u_{m,n}^{(l)} = \prod_{i \in N(m) \setminus n} \text{sgn}(\lambda_i^{(l-1)} - u_{m,i}^{(l-1)}) \times \varphi \left(\sum_{j \in N(m) \setminus n} \varphi(|\lambda_j^{(l-1)} - u_{m,j}^{(l-1)}|) \right) \quad (4.1)$$

where:

$$\varphi(x) = \log \left(\frac{e^x + 1}{e^x - 1} \right) \quad (4.2)$$

And the variable-node update is:

$$\lambda_n^{(l)} = \frac{2}{\sigma^2} r_n + \sum_{m \in M(n)} u_{m,n}^{(l)} \quad (4.3)$$

From the above equations, it is obvious that the check-node updates are more computationally complex when compared to the variable-node updates. This complexity arises from two factors: first, the computation of $\varphi(x)$ function and second, the exclusion of the LLR value of each variable-node in the computation of messages passed to it from the connected check-nodes. In other words, the check-node should pass different values to each of the connected variable-nodes.

In order to reduce the computation of the check-node processing, different simplifi-

CHAPTER 4. THE IMPLEMENTATION OF A SIMPLIFIED SUM-PRODUCT DECODING
54 ALGORITHM

cations have been made to it [16]. In this section, a new simplification is proposed that reduces the complexity of the Sum-Product algorithm without any significant degradation of its performance. Let $z_{m,i}$ denotes the absolute value passed from the variable-node i to check-node m :

$$z_{m,i} = |\lambda_i - u_{m,i}| \quad (4.4)$$

Equation (4.1) can therefore be written as:

$$u_{m,n}^{(l)} = \text{sgn}(u_{m,n}^{(l)}) \varphi \left(\sum_{i \in N(m) \setminus n} \varphi(z_{m,i}^{(l-1)}) \right) \quad (4.5)$$

Because of the behavior of $\varphi(x)$, the minimum value of $z_{m,i}$ has more significant effect on the summation of equation (4.5), leading to an approximation known as the Min-Sum. However, a better approximation can be made by denoting $z_{m,min}$ and $z_{m,min2}$ as the minimum and the next minimum values in $\{z_{m,i} | i \in N(m)\}$, respectively. The value of $z_{m,min}$ exists in the computation of all the check-node values except for $u_{m,min}$. It has the most contribution in the summation step of (4.5), for the computation of check-node value. Hence, the following approximation can be made:

$$|u_{m,others}| \approx |u_{m,min2}| \quad (4.6)$$

Where $u_{m,others}$ corresponds to $\{u_{m,i} | i \neq min, min2\}$. (4.5) can be further simplified as follows:

$$u_{m,n}^{(l)} = \begin{cases} \text{sgn}(u_{m,n}^{(l)}) \varphi(\sum_{i \in N(m) \setminus min} \varphi(z_{m,i}^{(l-1)})) & n = min \\ \text{sgn}(u_{m,n}^{(l)}) \varphi(\sum_{i \in N(m) \setminus min2} \varphi(z_{m,i}^{(l-1)})) & n \neq min \end{cases} \quad (4.7)$$

In (4.7), the exclusion is performed only twice, and this helps to reduce the number of computations, especially for check-nodes with higher degrees. In both cases of (4.7), the summation only differs in one term and the sum of the other terms appears in both.

Define Z_m as:

$$Z_m = \varphi \left(\sum_{i \in N(m) \setminus min, min2} \varphi(z_{m,i}) \right) \quad (4.8)$$

Then:

$$|u_{m,n}^{(l)}| = \begin{cases} \varphi(\varphi(z_{m,min2}^{(l-1)}) + \varphi(Z_m^{(l-1)})) & n = min \\ \varphi(\varphi(z_{m,min}^{(l-1)}) + \varphi(Z_m^{(l-1)})) & n \neq min \end{cases} \quad (4.9)$$

Note that the symmetry property of $\varphi(x)$ is used for the approximation. In (4.9), the check-node computation is reduced to one of degree-three and Z_m represents the effect of other variable-nodes connected to check-node m than minimum and next minimum variable-nodes in computation of the magnitude. Using the Jacobian logarithm approach [51], (4.9) can be expressed as:

$$|u_{m,n}^{(l)}| = \begin{cases} z_{m,min2}^{(l-1)} + \log\left(\frac{1+e^{-|z_{m,min2}^{(l-1)}+Z_m^{(l-1)}|}}{1+e^{-|z_{m,min2}^{(l-1)}-Z_m^{(l-1)}|}}\right) & n = min \\ z_{m,min}^{(l-1)} + \log\left(\frac{1+e^{-|z_{m,min}^{(l-1)}+Z_m^{(l-1)}|}}{1+e^{-|z_{m,min}^{(l-1)}-Z_m^{(l-1)}|}}\right) & n \neq min \end{cases} \quad (4.10)$$

In here, it has been assumed that $z_{m,min} < z_{m,min2} < Z_m$. The above equation is similar to the Min-Sum approximation with a correction factor. Unlike other methods such as the Offset-minsum [16], the correction factor used in the proposed algorithm is not a constant but depends on the value of the variable-nodes connected to the check-node and is updated dynamically. The computation of the correction factors can be done using the look-up tables. The computation of (4.9) using (4.10) is less subjected to the quantization effects. This is because the quantization error only appears in the correction factor.

4.3 The Hardware/Software based system architecture

Based on the decoding algorithm of the previous section, the implementation of the decoder is proposed. The design is a Hardware/Software design that takes advantage of the PowerPC processors embedded in the Virtex-II Pro FPGAs. This methodology brings more flexibility since the additional functionality can be added to the system by simply writing C code for the PowerPC. The overall system architecture is shown in Figure 4.1.

Figure 4.1: The hardware-based simulator architecture.

The PowerPC is connected to the Processor Local Bus (PLB) with three other major components: DDR external memory, decoder and additive white Gaussian noise (AWGN) generator. The processor's executable program as well as the parity check matrix structure is located on the DDR memory since the parity check matrix structure contains a large amount of data for long and unstructured LDPC codes.

The parity check matrix data is segmented equally (i.e. data of a known number of rows) and each segment is read from DDR memory and stored in the FPGA's Block-RAMs (BRAMs). Communication between the FPGA and the host PC is performed through a UART interface. The control commands and the parity check matrix data are sent to the system via UART, which also monitors status of the simulator such as number of error blocks detected and the number of error bits in the blocks. The UART interface that is controlled by the processor enables us to reconfigure the system without having to download the bitstream on the FPGA.

4.3.1 The serial decoder architecture

The decoder architecture illustrated in Figure 4.2 is serial, in which the check-node and variable-node operations are done sequentially. The throughput of the parallel decoders is significantly higher than serial ones since they use the inherent parallelism of the Sum-

Product decoding algorithm. However, in the parallel decoders the complexity increases with the code length as well as the check-node and variable-node degrees, which lead to vast amount of wire interconnections. Consequently, the parallel architectures are generally used for structured codes such as quasi-cyclic ones. Since the decoder is intended for the simulation of any kind of LDPC codes as well as the unstructured ones, the serial architecture is chosen. The serial architecture also consumes fewer resources which enables one to implement other peripherals in the embedded processor system for further functionality.

Figure 4.2: Block diagram of embedded LDPC decoder and its bus interface to the PLB.

The decoder consists of two memory spaces for storing the parity check matrix. One contains the location of ones in each row and the other the location of ones in each column named as *Row-to-Col map* memory and *Col-to-Row map* memory respectively. The indices of the ones in the first row are stored at the start of the Row-to-Col map memory followed by a special value indicating the end of the row. The other rows' data are stored in the same manner. At the start of each iteration, the indices of columns in a row are read and the corresponding LLR values are read from the *LLR memory*. These values are sent to the *Check-node processor (CNP)* for the check-node updating.

CHAPTER 4. THE IMPLEMENTATION OF A SIMPLIFIED SUM-PRODUCT DECODING
58 ALGORITHM

As will be shown later, the check-node updating is not done immediately after reading the LLR values and requires the same processing time. In order to save on the memory consumed by the Row-to-Col map on the FPGA, and to take advantage of a free time slot, the Row-to-Col memory consists of two identical banks. While one bank is being used for mapping data, the other is filled with data read from external DDR memory. The maximum number of ones in each row of the parity matrix is limited to be 64 and each memory bank stores the information of 16 rows, given that each index is represented by 16 bits, each bank is of size of $64 \times 16 = 1024$ 16-bit words. This space can be realized using a single BRAM. Consequently, the total space consumed for Row-to-Col memory is two BRAMs. Unlike the check-node update, the variable-node update can be done instantaneously. If the Col-to-Row memory data is written in a segmented fashion, it will slow down the variable-node process. Based on this, the Col-to-Row is updated only once and before the system starts decoding. Given that in the example discussed, the maximum variable-degree is 4 and the maximum code length is 4608, the memory size needed is therefore $4 \times 4608 = 18432$ 16-bit words. This will require 18 BRAMs.

The messages in all stages are represented in (9:5) signed format. The data read from channel (AWGN) are stored in *Rx Data* memory and LLR memory as the initial LLR values. Data from LLR memory is sent to CNP with the architecture provided in Figure 4.3. The CNP handles the check-update for a single row. There is a buffer at the input of the CNP which is filled with the LLR values. These values are then read from the buffer and converted to sign and magnitude and processed individually.

In the sign processing, the stream of sign-bits are stored in a shift register and at the same time XOR-ed together. Based on this value, the shift register is inverted or remains unchanged.

Since the number of LLR values for a row is unknown to the CNP, the sign values are first stored in a last-in-first-out (LIFO) memory and then popped and fed to the shift register. This enables the shift register to have the computed sign values in correct order i.e. the LSB corresponds to the sign of first column of the processed row. This register

Figure 4.3: Block diagram of the check-node processor.

is used for storing updated values as will be shown later.

The main function of CNP is to calculate equation (9), which consists of the computation of four values of minimum, next minimum, correction factor for minimum and correction factor for next minimum. The search for the minimum and the next minimum from the incoming data stream is straight-forward and is done by comparators.

In the CNP, two mathematical functions are evaluated, $\varphi(x)$ and $\psi(x) = \log(1 + e^{-|x|})$. These functions can be realized using the look-up tables (LUTs). However, since $d\varphi(x)/dx$ is large when x is small, if a uniform quantization is used, it results in a large LUT. In order to reduce the size of the LUT, a non-uniform quantization is preferred. The input to these functions, which is 8-bit length, is passed to the leading zero detector (LZD) module. (see [52] for a discussion on LZD). The output of the LZD is used as the address to the LUTs. Using this approach reduces the resources needed in the non-linear function approximation used in the computation of the correction factor.

According to (4.8), the computation of $\varphi(x)$ is done on all of the LLR values except for the minimum and the next minimum. Thus, the computation cannot be done before these values are known. To overcome this delay, all the LLR values (including min and next min) are fed to the LZD and LUT of $\varphi(x)$ and added together, while the *min finder*

CHAPTER 4. THE IMPLEMENTATION OF A SIMPLIFIED SUM-PRODUCT DECODING
60 ALGORITHM

unit simultaneously looks for the min and the next min. After the values are added together, the min and the next min are fed again to the LZD and are subtracted from the previously latched value. Consequently, the summation is available just a few clock cycles after the min value is found. This saves time especially for check-nodes with larger degrees. As per equation (4.8), the evaluation of $\varphi(x)$ is performed once more on the accumulated value after it has been truncated back to 8-bits. The adjusted value is fed back for final computation and the result is passed on to the *correction factor (CF)* unit together with the min and next min values.

In the CF unit, the $\psi(x)$ is evaluated using the same method discussed above, to approximate the second term in (4.10). Each of these terms needs to compute $\psi(x)$ twice (the results are subtracted.). Two LZDs and two LUTs are used for each of the correction factors in order to reduce the latency in the CF unit. The correction factors, which are negative values, are then added to the min and next min values to generate the check-node updates.

Using the Min-Sum decoding algorithm with correction factor reduces the computational load of the check-node update process and also significantly reduces the memory requirement. According to (4.10), the magnitudes of the messages that are passed to the variable-nodes have one of the two values. The first is sent to the variable node with the minimum value and the second is sent to all the other variable nodes. This allows one to store the compressed check-node values. In this method the updated value can be represented using a single slightly longer word, which consists of four parts and is depicted in Figure 4.3. The first part is called the *sign pattern* and carries the sign of the messages passed to the connected variable-nodes. It is 64 bits, for the maximum number of sixty four ones in a row. The second part contains position of the minimum value of the received variable-nodes messages. This part is six bits and indicates the variable-node that is sent a different message (magnitude) from the others. The next two parts are the messages (magnitude) sent to the min variable-node and the other variable-nodes and are each represented by eight bits, giving a total of $64 + 6 + 8 + 8 = 86$ bits for

the register. Assuming the maximum number of rows is 1024, the total memory size required for the check-nodes is 86Kb which is implemented on FPGA using 6 BRAMs. This is significantly lower than $64 \times 9 \times 1024 = 576\text{Kb}$, if the updated values were not compressed.

The next step after the check-node updates is the updating of the variable-nodes or LLR values. The addresses of the check-nodes connected to a variable-node are read from the Col-to-Row memory and the corresponding values are retrieved from the compressed format stored in check-node memory. These addresses are also sent to the *syndrome check* module as they indicate the parity check equations in which the processed variable-node takes part. The hard decision is also done at this stage. Once all the LLR values are updated the done signal is sent to the PowerPC and the status registers are updated. These registers indicate whether the parity check equations have been satisfied and the number of error bits in the block after making a hard decision. If the parity check is satisfied or the maximum number of iterations is reached, the processing for the current block will be deemed as having been completed.

4.3.2 The AWGN channel

In order to test the performance of the LDPC decoder and the codes, an AWGN channel with all-zero input codeword is transmitted. The code performance evaluation highly relies on the statistical properties of the random noise generator especially for the higher signal-to-noise ratio (SNR) regions. In high SNR regions, a vast amount of samples should be generated to detect the errors. Consequently, the noise generator should not have a short period and repeat its cycle during the simulation as it may defeat the results. Another important consideration is the fixed-point representation of the samples, which limits the range and accuracy of the samples.

Among the reported designs, such as that reported in [43], which uses the Box-Muller method, is chosen and implemented which has a very high throughput (200 million samples running at 100MHz) and is able to produce samples of up to 8.2σ from the

CHAPTER 4. THE IMPLEMENTATION OF A SIMPLIFIED SUM-PRODUCT DECODING
62 ALGORITHM

mean. This is critical, especially for higher SNR's, since these extreme values are the ones causing the errors at low BER's.

The Box-Muller method transforms the uniformly distributed random numbers to Gaussian random numbers using the mathematical functions as bellow:

$$x_1 = \sqrt{-2\ln(u_1)} \times \sin(u_2) \quad (4.11)$$

$$x_2 = \sqrt{-2\ln(u_1)} \times \cos(u_2) \quad (4.12)$$

Where u_1 and u_2 are two independent uniform random number in $[0, 1]$. x_1 and x_2 are the generated random numbers. The bit-widths of u_1 and u_2 have direct impact on the maximum value of the generated samples. The implemented architecture and the its data representation according to [43] are illustrated in Figure 4.4.

Figure 4.4: The AWGN generator based on Box-Muller method [43]

As it can be seen from equations (4.11) and (4.12), the implementation of Box-Muller method mainly involves realization of mathematical functions. In this formula, three different functions ought to be realized including sine, logarithm and square-root, where each of which has a different range and behavior. The elementary mathematical functions can be evaluated using different well-known methods such as the polynomial approximation, look-up tables and CORDIC method.

In this work, the functions are evaluated using the piecewise polynomial approximation with uniform segmentation. The functions are approximated by a polynomial for every segment such that it minimizes the error in that segment:

$$y = ((C_d x + C_{d-1})x + \dots)x + C_0 \quad (4.13)$$

where d is the polynomial degree and x is the input. For the sake of simplicity, it is also assumed that all the segments are evaluated by the same degree polynomials. This method significantly reduces the size of look-up tables since one only needs to store the C_i values instead of the function value for every input value. In general, the number of segments is much less than the input points to the function. The input interval is then split into 2^B equally-sized segments, and hence, the B most-significant bits of x will serve as the address to the C_i s table. Therefore, the size of table will be:

$$2^B \times \sum_{i=0}^d B_{C_i} \quad (4.14)$$

where B_{C_i} is the width of C_i coefficient.

The main challenge is on how to minimize the polynomial degree and the coefficient widths so that it can result in the smallest possible coefficients memory. The MiniBit bit-width optimization approach [53] is used to minimize the width of signals while it guarantees the minimum error requirements. As an example, suppose that the minimum

CHAPTER 4. THE IMPLEMENTATION OF A SIMPLIFIED SUM-PRODUCT DECODING
64 ALGORITHM

coefficient widths and input width for the following polynomial have to be found:

$$y = C_1x + C_0 \quad (4.15)$$

The above expression can be written as the:

$$y = a + C_0 \quad (4.16)$$

For the addition/subtraction error can be derived by:

$$\tilde{y} = \tilde{a} + \tilde{C}_0 = a + C_0 + E_{\tilde{a}} + E_{\tilde{C}_0} + 2^{-FB_{\tilde{y}}-1} \quad (4.17)$$

where $2^{-FB_{\tilde{y}}-1}$ is the rounding error for $\tilde{y}$. Thus:

$$E_{\tilde{y}} = E_{\tilde{a}} + E_{\tilde{C}_0} + 2^{-FB_{\tilde{y}}-1} \quad (4.18)$$

Similarly, the multiplication error is:

$$\begin{aligned} \tilde{a} &= C_1x + C_1E_{\tilde{x}} + xE_{\tilde{C}_1} + E_{\tilde{x}}E_{\tilde{C}_1} + 2^{-FB_{\tilde{a}}-1} \\ \Rightarrow E_{\tilde{a}} &= C_1E_{\tilde{x}} + xE_{\tilde{C}_1} + E_{\tilde{x}}E_{\tilde{C}_1} + 2^{-FB_{\tilde{a}}-1} \end{aligned} \quad (4.19)$$

Assuming that y is represented using FB_y fractional bits and is required to be precise up to FB_y fractional bits, (4.18) can be rewritten as:

$$E_{\tilde{y}} < 2^{-FB_y} \Rightarrow E_{\tilde{a}} + E_{\tilde{C}_0} < 2^{-FB_y-1} \quad (4.20)$$

The minimum number of fractional bits for a and c_0 are found using (4.20) by trial and error (substituting the most likely values), which can be done using a simple MATLAB or C code. After finding the minimum fractional bits for a and $E_{\tilde{a}}$, the minimum values for $FB_{\tilde{x}}$ and $FB_{\tilde{C}_1}$ can be found by the same method using equation (4.19).

When approximating a function with piecewise polynomials, one is interested in finding the minimum number of segments for a given degree and precision. A MATLAB code that evaluates the mathematical function using degree one and degree two polynomials, returning the number of segments in each case is therefore developed. The MATLAB code splits the input range to equally sized segments and finds the polynomial coefficients for each segment that best fits the function. This task is easily done using MATLAB's `polyfit` function. The number of segments should be a power of two so that the left most significant bits of the input can be used for indexing. Finally, a trade-off should be made between the number of segments and the polynomial degree. Based on this method, Sine and Square Root functions are approximated with degree one polynomials while Logarithm function is best approximated with degree two polynomials.

4.4 Implementation and results

The described architecture for the embedded serial LDPC decoder is implemented on a Virtex-II Pro (XC2VP30) FPGA. The decoder occupies 729 slices which is less than 6% of the available slices on the FPGA. This low area implementation is a result of the serial architecture. By comparison, the AWGN generator is implemented using 1568 slices. The decoder and the AWGN generator are controlled by the PowerPC processor, and are both programmed in C. The maximum throughput of the simulator is about 2Mbits/sec. Although, it is less than the throughput of the parallel decoders presented in the literature (see table 3.1) it has distinct characteristics, including low area, ability of decoding arbitrary LDPC codes and Hardware/Software co-design that makes it as an affordable and convenient choice for LDPC decoding simulations.

Figure 4.5 shows the BER and frame error rate (FER) performances of some unstructured LDPC codes available from Mackay's website [22]. The codes chosen are of high rate, with block lengths of around 4000, and check-degrees ranging from 17 to 63, demonstrating the flexibility of the decoder. For random LDPC codes, the proposed design can achieve BERs as low as 10^{-9} in a few hours .

Figure 4.5: BER and FER performance of some unstructured LDPC codes.

In order to compare the performance of the simplified algorithm with other Min-Sum family algorithms, the decoder can be readily modified to support Min-Sum and Offset Min-Sum algorithms. The (4095,737) LDPC code, which shows error floor, is simulated and its performance curves for different algorithms are plotted in Figure 4.6. The offset value of the Offset-minsum algorithm is chosen to be 0.5 according to software based simulations. The curves reveal that the proposed algorithm performs slightly better in the waterfall region. However, all the algorithms, including Min-Sum, performs quite close to each other in the error floor region. The proposed algorithm is also advantageous to the Offset-minsum since it does not require off-line computations such as density evaluation.

Table 4.1 shows the throughput of the various codes mentioned earlier for a maximum number of 10 iterations. Since the decoder checks the syndrome after each itera-

Figure 4.6: Performance comparison of different decoding algorithms.

tion, decoding is done after a couple of iterations for the higher SNR regions and leads to higher throughput. It should be noted that the throughput of the decoder is less than the parallel architectures presented in the literature. However, the proposed decoder is able to evaluate any kind of LDPC code regardless of its structure.

CHAPTER 4. THE IMPLEMENTATION OF A SIMPLIFIED SUM-PRODUCT DECODING
68 ALGORITHM

Table 4.1: System Performance for different high-rate codes.

Code	Max Check-degree	Max Variable-degree	Max Throughput (Mbits/sec)	Simulation Time (hrs)
(4095,737)	17	3	2	5.5
(4376,272)	63	4	2.43	3
(4161,731)	23	4	1.45	14

4.5 Summary

In this chapter, the architecture and implementation of an embedded LDPC decoder for the simulation of unstructured LDPC codes is presented. The decoder has a serial architecture and is working along with the embedded processor of the Virtex-II Pro FPGA. The processor manages the simulation by providing handshaking signals, memory data transfer, and communication with PC while the peripherals implemented on the FPGA slices provide fast computations on the flow of data. Such a Hardware/Software co-design provides flexibility of software and offers high throughput of the hardware.

The decoding algorithm used is a simplified version of the Sum-Product algorithm which belongs to the class of Min-Sum algorithms with correction factor. However, in the proposed algorithm, the correction factor is computed based on the LLR values and is updated dynamically.

Chapter 5

Message Packing LDPC Decoder Architecture

5.1 Introduction

One of the key factors in designing digital systems, especially those targeted for communication devices, is the flexibility and adaptability. This is due to different requirements of several new standards proposed every year. Flexibility is even more important in this project since the LDPC decoder is aimed for hardware-based simulations where it has to operate for different codes and modes as the variables of the studies in simulations.

Unfortunately, the majority of reported LDPC decoders support a single or a particular class of LDPC codes. The most flexible decoder architecture is the serial architecture, similar to one described in chapter 4. However, the throughput of serial decoders are quite low compared to a parallel architecture in which the decoder has the same structure as the Tanner graph. In parallel architectures, every node of the graph is implemented as a processing element resulting in a very high decoding throughput. Such a design is proposed in [26], which is designed specifically for a particular code. This fully-parallel decoder is implemented on ASIC technology and some software tools were developed to help the floor-planning of the chip. In spite of this, the fully-parallel architecture can-

not offer flexibility as the interconnections and processing elements are designed for a particular code, and moreover, they are not likely to be feasible using FPGA technology where the routing resources are limited. The trade-off between these two architectures, parallel and serial, is a partially-parallel decoder in which a processing element (PE) is responsible for a subset of nodes on the Tanner graph. The PEs are connected via an interconnection network or shared memory to pass messages among each other. Although the partially parallel architectures are more flexible, they are not explored well for decoding of arbitrarily constructed codes. This is due to the random structure of the Tanner graph for random codes which cause access conflicts to shared resources.

In [54], a partially parallel decoder implementation on both FPGA and ASIC is described. The decoder is designed for 8088 length and rate of 1/2 codes. It supports a class of codes with these specifications. Another FPGA targeted implementation is reported in [31] that supports Quasi-Cyclic LDPC codes. Similarly, designs reported in [55, 56] are optimized for Quasi-Cyclic codes. There are some other designs that support codes defined by IEEE standards. These designs may also be viewed as being flexible in the sense that they support different modes of that particular standard, i.e. different code lengths and code rates, while all these codes have the same structure. An LDPC codec for second generation Digital Video Broadcast (DVB-S2) is presented in [49] that supports codes with eleven different rates and two different lengths. Liu *et al.* [29] implemented an LDPC decoder chip for IEEE 802.16e standard. Their design exploits a self-routing network for message passing and supports different modes of the standard.

Among the reported LDPC decoders, a couple of them are capable of supporting random codes. Two different designs based on the same re-formulation of Sum-Product algorithm are reported individually by Masera *et.al* [57] and Kang *et. al* [58]. Both architectures are implemented on ASIC and have their own advantages and shortcomings. These architectures will be detailed in section 5.2 as they have the same goal as our project.

After a brief overview of flexible architectures, a new solution for the decoding of

randomly constructed LDPC codes is presented in this chapter. The proposed solution is based on two distinct methods for the delivery of messages to the variable nodes and check nodes, which is detailed in section 5.3. Based on the proposed method, an FPGA implementation is presented in section 5.4. The FPGA implementation involves designing PEs for the variable node and check node updates, which are described in subsections 5.4.1 and 5.4.2, respectively. The proposed flexible decoder is examined and compared to other designs in section 5.4.3.

5.2 Overview of the existing flexible partially-parallel LDPC decoders

Although the fully parallel implementation of LDPC decoding algorithm is straightforward and leads to very high throughputs, it suffers from the complexity of interconnections and routing, especially for long codes, and is not a suitable candidate for flexible decoders. In partially-parallel designs, two types of processing elements, known as the check node processor (CNP) and variable node processor (VNP), are used where a subset of check nodes and variable nodes are associated with each other, respectively. At any time instance during the check (variable) node update phase, a CNP (VNP) performs the computations of a single check (variable) node variable on the Tanner graph. The key point in exploiting such an architecture for random codes is on how to deal with the collisions when the messages are passed between nodes of different types. As an example, consider the LDPC code depicted in Figure 5.1 and its decoder architecture that utilizes three VNPs and two CNPs. The CNPs and VNPs are connected via a crossbar network and suppose each of them can send or receive one message at a time. For a given scheduling, CNP1 is performing the CN2 update and CNP2 is updating CN5. CN2 is connected to VN4 and CN5 to VN5 with both associated with VNP2. This which means both CNP1 and CNP2 have requested for access to VNP2 for different data, and hence, a collision will occur, assuming that every VNP/CNP have only one input/output port.

Figure 5.1: Prototype of a partially-parallel LDPC decoder.

In this simple and prototype architecture the collision may be reduced by clever allocation of processing elements to the nodes and proper scheduling. For cyclic codes this is straightforward and leads to collision free architectures. For random codes, however, this may require other methods such as the scheduling computer algorithms.

In the both reported architectures of [57] and [58], the authors have addressed the above discussed problem by re-arranging the Sum-Product algorithm. In the original algorithm, each check node and variable node send different messages to connected nodes while in their approach each node send the same value to all connected nodes. To observe the benefit of this method, consider the prototype decoder when CNP1 is processing CN1 and CNP2 processing CN4. Both these nodes are connected to VN1. In the modified algorithm, VN1 sends equal values to CN1 and CN4, and hence, CNP1 and CNP2 do not conflict with each other when accessing VNP1. In this manner, the number of collisions is reduced. Below is the re-arranged check node update:

$$z_{m,i}^{(l-1)} = \lambda_i^{(l-1)} - u_{m,i}^{(l-1)} \quad (5.1)$$

$$U_m^{(l)} = \prod_{j \in N(m)} \text{sgn}(z_{m,j}^{(l-1)}) \sum_{j \in N(m)} \varphi(|z_{m,j}^{(l-1)}|) \quad (5.2)$$

$$u_{m,i}^{(l)} = \text{sgn}(z_{m,i}^{(l-1)}) \times \text{sgn}(U_m^{(l)}) \varphi(U_m^{(l)} - \varphi(z_{m,i}^{(l-1)})) \quad (5.3)$$

Similarly, the variable node update can be re-written:

$$u_{m,n}^{(l)} = \text{sgn}(z_{m,n}^{(l-1)}) \times \text{sgn}(U_m^{(l)}) \varphi(U_m^{(l)} - \varphi(z_{m,n}^{(l-1)})) \quad (5.4)$$

$$\lambda_n^{(l)} = \frac{2}{\sigma^2} r_n + \sum_{i \in M(n)} u_{i,n}^{(l)} \quad (5.5)$$

$$z_{m,n}^{(l)} = \lambda_n^{(l)} - u_{m,n}^{(l)} \quad (5.6)$$

The equations for check node update and variable node update are very similar and each involves three types of variables: individual messages, global messages and local messages. In (5.1) and (5.4), the individual messages are recovered in the destination side, i.e. each check node recovers the individual messages of variable nodes. The next step is the computation of global messages U_m and λ_n in (5.2) and (5.5). A global message is sent to all connected nodes instead of the individual messages. And finally, the local messages are computed in (5.3) and (5.6), which are kept for the recovery of individual messages in the next iteration. Obviously, this kind of expression has some redundancies. Equation (5.1) is similar to (5.6) and (5.3) is repeated in variable nodes (5.4). This means the computations are increased as a result of broadcasting global messages instead of individual ones especially for variable node update that in the original formulation is nothing than a summation.

The aforementioned reformulation of decoding algorithm had been implemented by Kang [58] and [57], and in different ways. Kang *et al.* [58] used an architecture illustrated in Figure 5.2. This architecture corresponds to an LDPC code with 12 variable

nodes and 6 check nodes. Each VNP computes 4 values corresponding 4 assigned variable nodes and each CNP computes 3 values. These values are stored in registers and are accessed through multiplexers. The multiplexers shown in figure 5.2 are in fact crossbar switches with unbalanced number of inputs and outputs.

Figure 5.2: LDPC decoder architecture by Kang[58].

Every CNP and VNP have a local memory (shown as δ_{ij} and λ_{ij}) to store the local values. The VNP and CNPs are designed so that they are capable of updating of a node regardless of its degree. This means that all the input messages to a single node are processed at the same time leading to higher throughput. Hence, the processing units have multiple inputs. This architecture is implemented for a 1024-bit rate 1/2 LDPC code in ASIC, with having a throughput of 582 Mbps and occupying 6.29 mm². The drawback of this architecture is the complexity of its interconnections and message distribution network, consisting of de-multiplexers and multiplexers, especially for long codes. In order to clarify this complexity, it requires $32 \times 6 = 192$ 1024:1 multiplexers for the delivery of messages from VNPs to CNPs, knowing that there are 32 CNPs and check node degree is 6. This consumes a lot of routing resources on the chip. Additionally, the local memories and output registers occupy a large area of the chip. Consequently, the decoder in [58] is not a good candidate for large codes having nodes with high

degrees, especially for FPGA target. Despite the generality of presented approach, the implemented hardware is not capable of handling different codes since the number of inputs of the multiplexers is based on the code length. The CNP and VNP architectures also rely on the according node degrees.

The modified algorithm is also known as *low traffic belief propagation algorithm* [59]. The proposed architecture in [57] is similar to the prototype decoder shown in Figure 5.1, employing 32 processing elements of each type. Because of the similarity of the check node process and the variable node's, a common processing unit is designed that performs both check node update and variable node update. Therefore, processing units are located at one side of the interconnection network and the other side of the network is tied back to the processing elements. This architecture is more flexible while it delivers a lower throughput. This is due to the fact that the processing elements process one edge of the Tanner graph at any time instance. The interconnection network consists of 32 32:1 multiplexers and a 32×32 Benes network [60]. This architecture also suffers from interconnection complexity making it unsuitable for implementing on FPGA. Another drawback of these two architectures is redundant computations since check nodes and variable nodes perform the same computations in different locations and time as discussed before. Additionally, the Min-Sum algorithm and its modified versions are not able to be utilized, which leads in more complex architecture. As discussed in chapter 2, the modified Min-Sum algorithms can compete the original Sum-Product algorithm via fixed-point realization.

5.3 Message Packing decoding architecture

The Tanner graph representation of LDPC decoding algorithm allows one to implement a fully parallel decoder, in which every node is implemented as a processor. However, this type of implementation requires a vast amount of resources, almost infeasible for large codes and current technologies. On the other hand, *partially-parallel* decoders reduce area by resource sharing, and make the implementation of flexible decoders possible.

Figure 5.3: A general prototype for partially-parallel LDPC decoders.

In the partially-parallel LDPC decoders, there are a number of variable node processors (VNPs) (check node processors (CNPs)), each of which is responsible for the update of a subset of variable nodes (check nodes). An interconnection network may be utilized for message passing according to the Tanner graph. Several kind of architectures can be designed for partially-parallel LDPC decoders. A general prototype of partially-parallel decoders is illustrated in Figure 5.3, which employs a bidirectional crossbar network for message delivery.

In the proposed prototype, it is considered that a processor is capable of processing only one message at a time. If the given LDPC code has E edges on its Tanner graph and p processors of each type are available, the optimal processing time for a half-iteration will clearly be:

$$T_{\frac{1}{2}iter} = \lceil \frac{E}{p} \rceil \quad [\text{cycles}] \quad (5.7)$$

This can be used as a measure for the efficiency of parallelism when compared to the actual processing time. It should be mentioned that these two measures can be equal for the structured codes.

According to the message passing algorithm, there are two types of deliveries namely, the check-to-variable (CTV) and variable-to-check (VTC) deliveries. In each type of de-

livery, it is desired that all processors at the output of the network receive a message at any time instance. Note that the delivery and updating of nodes can be overlapped, i.e. the check nodes are updated during the variable-to-check delivery.

As mentioned earlier, the decoding of high rate LDPC codes are desired in this work where the number of variable nodes is much higher than the check nodes. As a result, the variable node degree is far less than the check node degree. This research, therefore, will take advantage of the properties to simplify the delivery network by proposing two distinct methods of delivery, both named as message packing. These methods will be discussed in the following subsections.

5.3.1 Message packing for variable-to-check delivery

Suppose that the paritally-parallel decoder is employed for decoding an LDPC code with codeword length of N , N variable nodes should be distributed among p VNPs. It is assumed that every VNP is associated with $\lfloor N/p \rfloor$ variable nodes, except for the last one which is associate with $N - (p - 1)\lfloor N/p \rfloor$ variable nodes. Furthermore, there is a simple node assignment, i.e. the first $\lfloor N/p \rfloor$ nodes are assigned to VNP1 and so on. The same discussion applies to M check nodes and p CNPs.

According to equation (2), a variable node sends the same value to all connected check nodes. This allows one to deliver variable node messages in N cycles rather than E cycles, if only one message is delivered via the delivery network at a time. Assume that variable nodes have a maximum degree of j , i.e. a variable node is connected to at most j check nodes. Therefore, a variable node message is delivered to at most j CNPs. If $p > j$, it might be possible to deliver more than one variable node at each cycle. Strictly speaking, one can guaranty to send $q = \lfloor p/j \rfloor$ variable node values over the network, if the values do not conflict each other at the output ports of delivery network. We call these q variable nodes a *pack* and the delivery method, message packing. This would further reduce the variable-to-check delivery time from N to N/q , which is equal to maximum number of packs. Fortunately, j is relatively small for high rate codes, as is

in our case, resulting in a few times reduction. For instance, if $p = 16$ and $j = 4$, there will be ideally four times reduction in the delivery time.

However, two issues should be considered before proposing a practical solution:

1. It is not straightforward or even possible to find a maximum number of N/q packs due the collisions at the CNPs. Recall $CNP(VN_k)$ as the set of CNPs to which variable node VN_k value should be delivered. VN_i and VN_j can be packed together, if $CNP(VN_i) \cap CNP(VN_j) = \phi$.
2. Another important issue is the interconnection network. In the previous discussion, the network is assumed to be capable of non-blocking *multicast* routing, where every output can be connected to any input. A crossbar network is a strictly non-blocking network. Crossbar networks may require large area, especially for large number of inputs/outputs. Therefore, it is desired to employ less expensive multicast networks that are not strictly non-blocking.

This problem is addressed by dividing p VNPs into j groups, with each having an equal number of VNPs. The value of j is preferably a factor of p , which also defines the maximum variable node degree supported by the decoder. Without loosing the generality, by letting $p = 16$ and $j = 4$, each packing group will have four VNPs. Additionally, by forming four VNP sets (packing groups) in the solution, the followings are derived: $G_1 = \{VNP_1, \dots, VNP_4\}$, $G_2 = \{VNP_5, \dots, VNP_8\}$, $G_3 = \{VNP_9, \dots, VNP_{12}\}$ and $G_4 = \{VNP_{13}, \dots, VNP_{16}\}$. With these assumptions, a two-stage delivery network as that shown in Figure 5.4 is proposed, whereby the first stage switches between packing sets and the second one generates the fanout. In general case, the network is built using p/j $j:1$ multiplexers plus p $p/j:1$ multiplexers.

The next step is to individually determine the packs in each of the four groups (G_1 , G_2 , G_3 and G_4), so that they do not conflict at their destination CNPs. For this purpose, a new computer algorithm based on the *Backtracking* computer algorithm is proposed. The Backtracking algorithm performs a depth-first search of the state space tree, where it

Figure 5.4: VTC network

searches a level at a time to find a node that satisfies the constraints before advancing to the next level. If a node cannot be expanded, the node becomes *dead* and the algorithm returns to the previous level. There are four levels in the proposed algorithm, which correspond to four VNPs that belong to a packing group. The algorithm searches each level to find a variable node that does not conflict with the previously selected variable nodes. Since there are four levels, an *answer* will consist of four variable nodes where each of which is selected from a distinct VNP. The pseudo code of the proposed algorithm is described in Algorithm 1.

The proposed algorithm applies a *heuristic* to the original Backtracking algorithm in order to quickly obtain a solution that is close to the optimum solution. The heuris-

Algorithm 1 Message packing for variable nodes

```

 $K \leftarrow 0$ 
 $TempPack \leftarrow \phi$ 
while ( $K > 0$ ) do
   $TempPack[K] \leftarrow Choose(VNP[k])$ 
  while  $\forall i, j \leq K$   $TempPack[i]$  conflict  $TempPack[j]$  and  $TempPack[K] \neq \phi$  do
     $TempPack[K] \leftarrow Choose(VNP[k])$ 
  end while
  if  $TempPack[K] = \phi$  then
     $K \leftarrow K - 1$ 
  else if  $K = p/j$  then
     $Solution.Add(TempPack)$ 
     $SearchSpace.Remove(TempPack)$ 
     $TempPack \leftarrow \phi$ 
     $K \leftarrow 1$ 
  else
     $K \leftarrow K + 1$ 
  end if
end while

```

tic is applied to the function '*Choose()*' of Algorithm 1. The proposed heuristic first selects nodes that are of higher constrain or nodes with fewer options to be expanded. This heuristic performs a forward checking on every candidate node and determines the number of children it can have in the next level, that is the *degrees of freedom*. Hence, *Choose()* selects the node with minimum degrees of freedom. After a pack is found, all of its nodes are removed from the search tree. In order to measure the efficiency of the heuristic, the packing efficiency η_p is defined and it is the ratio of the number of packs derived from the algorithm to the maximum number of possible packs.

The Variable node values are delivered through the interconnection network according to the results of the packing algorithm. Whenever a pack is to be delivered, three steps should be performed: 1) The first stage of multiplexers switches to one of the four possible positions based on the selected packing group; 2) Each of the four VNPs in the packing group present the variable node value that is in the pack; 3) Each multiplexer at the second stage routes one of the four variable node values to the connected CNP. There will be some variable nodes that do not belong to any packs since the algorithm is not optimized, and they need to be delivered individually. Therefore, the packing efficiency

has a direct impact on the delivery time. Let us denote the number of variable nodes assigned to a VNP as S_{vnp} and the number of packing groups as G . Therefore, the delivery time can be evaluated by

$$T = G \cdot S_{vnp} \cdot \eta_p + (1 - \eta_p) \cdot S_{vnp} \cdot p \quad [\text{cycles}] \quad (5.8)$$

In general case, $S_{vnp} \approx N/p$ and the number of packing groups will be $G = \lceil \frac{p}{p/j} \rceil \approx j$. Substituting these values into (5.8) results in the following approximation:

$$T \approx [1 - (1 - \frac{j}{p})\eta_p]N \quad [\text{cycles}] \quad (5.9)$$

It can be observed that the delivery time depends on three factors: the amount of parallelism or p , the maximum variable node degree, j , and the algorithm efficiency η_p .

One may claim that the proposed decoder architecture has a limit on the maximum supported variable node degree, which is four in the proposed prototype. Two possible solutions can be provided. First, one can select the number of packing groups by $G = \lceil \frac{p}{p/j} \rceil$ instead of four. However, the last group may have a smaller size. For example, six packing groups can be formed when $j = 5$, which are: $G_1 = \{VNP_1, VNP_2, VNP_3\}, \dots, G_5 = \{VNP_{13}, VNP_{14}, VNP_{15}\}$ and $G_6 = \{VNP_{16}\}$. The first stage of the multiplexers is still capable of routing these groups to the second stage. A better solution is to employ more CNPs and expand the second stage of multiplexers. For instance, the decoder can support a maximum variable node degree of eight if 32 CNPs are implemented.

5.3.2 Message packing for check-to-variable delivery

The check node update process for high rate LDPC codes is different from variable node update in two ways and this limits the effectiveness of VTC delivery method for CTV delivery. Firstly, a check node sends different values to each of the connected variable nodes, and secondly, for high rate codes, the check node degree is usually much higher than that of the variable node. Hence, different check node values cannot be sent together

Figure 5.5: Word structure representing the check node value.

using the same architecture. In this section, a new method of CTV delivery, in which all messages of a single check node are packed together and are delivered in one cycle, is proposed. This method is more efficient for high rate LDPC codes such as those employed in magnetic storage devices.

In the Min-Sum decoding algorithm (and its variants), the message sent from a check node can only have two magnitudes regardless of its degree. This allows one to store them in a packed form depicted in Figure 5.5. This method was introduced in [10]. It stores four values for each check node in a single binary word, that is in turn stored in a memory entry. Each of these words is called a *row pack*. These four values include: the minimum value, the second minimum, the index (position) of minimum and the *sign pattern*, which contains the sign of all messages being sent to the variable nodes. It is also assumed that magnitudes are represented in eight bits and a maximum check node degree of thirty two is supported. The proposed method is not only engaged for storage of values but for the delivery of check node messages. We call each of these words a *row pack*.

The row packs, stored in the CNPs, are fetched one by one and are sent to all VNPs using a simple architecture illustrated in Figure 5.6. Since the check node degree is usually higher than the number of VNPs (sixteen here) and the variable nodes are distributed evenly among the VNPs and with a good approximation, it is likely that all of the VNPs require the row pack for the update of at least one of the assigned variable nodes. The even distribution of variable nodes is nonetheless not usually perfect and some VNPs may require more than one cycle to update more than one variable node. Hence, a row pack has to be held at the input of VNPs for up to a few cycles, leading in a large delay

Figure 5.6: Check-to-variable delivery network.

in the CTV delivery and variable node update. In order to avoid this, a FIFO is located at the input of every VNP. The unbalanced distribution of different rows compensate each other, i.e. a VNP may process a row in one cycle while it processes the next one in three cycles. The delivery is halted as soon as a FIFO becomes full. The experiments conducted on different LDPC codes have shown that a FIFO size of sixteen is adequate (see next subsection).

5.3.3 Discussion and Results on Message Packing Method

The message packing algorithm, described in Algorithm 1, was implemented using C programming language in order to investigate its performance on some high rate LDPC codes. Once the desired packs are found, the VTC delivery time can be calculated using equation (5.8). Another C code was also developed that simulates the behavior of CTV network including the FIFOs. This code can derive the number of cycles required for CTV delivery. It should be noted that the delivery and node processing are overlapping, i.e. a processor handles a message as soon as it receives it. Therefore, the sum of

Table 5.1: The effect of FIFO depth on the number of cycles required.

p	Code		Wr	FIFO depth							
				2	4	5	6	8	12	16	32
16	I	(4095,3358)	17	1409	1165	1127	1102	1080	1059	1046	1030
	II	(4095,3359)	23	1693	1386	1338	1312	1273	1232	1223	1206
	III	(4376,4094)	63	1459	1230	1204	1187	1159	1130	1114	1104
	IV	(1057,813)	13	406	340	332	323	316	312	312	312
	V	(2048,1536)	16	512	512	512	512	512	512	512	512
	VI	(2048,1664)	32	1196	1014	988	975	968	965	965	965
32	I	(4095,3358)	17	1071	893	870	857	847	836	831	831
	II	(4095,359)	23	1246	997	960	947	928	907	896	889
	III	(4376,4094)	63	869	715	677	660	638	618	606	595
	IV	(1057,813)	13	320	278	273	271	269	269	269	269
	V	(2048,1536)	16	512	512	512	512	512	512	512	512
	VI	(2048,1664)	32	805	689	673	665	656	652	652	652

VTC and CTV delivery cycles represent the total number of cycles required for a single decoding iteration.

A set of high rate LDPC codes are chosen to examine the performance of message packing algorithm. Codes I to IV are unstructured LDPC codes taken from Mackay's website [22]. Code VI is a structured code constructed by using the method reported in [32]. Code V is the LDPC code defined in IEEE802.3an standard.

Another problem to be solved is the size of the FIFOs located at CNPs. Since the width of the delivered row word is relatively high, the depth of FIFO has a great impact on the resources occupied by the CNPs especially for the FPGA target since it is going to be implemented using distributed RAM. On the other hand, choosing a low depth may result in a lot of idle cycles in the message delivery. The delivery and the check node update for some high rate codes are simulated and the number of cycles for this task is estimated in table 5.1. Based on the results, a FIFO size of sixteen is satisfactory which in fact is the same size as of the look-up tables of Xilinx FPGAs. Any smaller sizes will result in the under-utilization of resources.

The CTV and VTC cycles are derived via simulations and the overhead of each phase is provided in table 5.2. The overhead is the number of additional cycles in comparison

Table 5.2: Cycles required for VTC and CTV delivery for different high rate LDPC codes.

Code		VN	CN	Edges	p	IPT [cycles]	VTC		CTV	
		Degree	Degree				cycles	overhead	cycles	overhead
I	(4095,3358) Unstructured	3	17	12285	16	768	1080	40.6%	1046	36.2%
II	(4095,3359) Unstructured	4	23	16380	16	1024	1170	14.2%	1223	19.4%
III	(4376,4094) Unstructured	4	63	17504	16	1094	1286	17.5%	1114	1.8%
IV	(1057,244) Unstructured	3	13	3171	16	199	340	70.8%	312	56.8%
V	(2048,1536) Quasi-cyclic	4	16	8192	16	512	646	26.2%	512	0%
VI	(2048,1664) 10GBASE-T	6	32	12288	32	384	644	67.7%	652	69.8%

with the ideal processing time (IPT). This is the required cycles in a half-iteration when every processor updates an edge of graph every cycle. The VTC overhead ranges from 14.2 % to 70.8% for the sample codes. The minimum overhead is for codes with variable node degree of four since the architecture is optimized for this value. Code V demonstrates an example of structured codes, for which the rows of the parity-check matrix are randomly permuted. Random row permutation, which does not change the code or encoding process, helps the search algorithm to find more packs as far as structured codes are concerned. The CTV overhead is also shown in the last column of table 5.2. The quasi-cyclic code (code V) has no overhead since the graph edges, originating from a check node, are equally assigned to all VNPs and the fact that the check node degree is equal to the number of VNPs. The results presented in table 5.2 confirms that the proposed method provides a convenient framework for general LDPC decoders capable of decoding high rate random LDPC codes as well as structured ones.

The proposed message delivery network provides multicast routing which is constrained so that a maximum number of inputs, *fan-in*, are connected. Therefore, it is desirable to compare it with other multicast networks in terms of resources. A Crossbar network is an example of strictly *non-blocking* network for multicast routing. In strictly non-blocking networks, any unconnected output can be connected to any input. However, the cost of a crossbar increases quadratically. A square $n \times n$ crossbar network consists of n n :1 multiplexers, and is equivalent to $n^2/2$ 2-input multiplexers. As n grows, the cost of crossbar quickly becomes un-affordable. On the other hand, the complexity

Figure 5.7: The resources

of Beneš networks grows in a logarithmic fashion. An $n \times n$ Beneš can be constructed from $2 \log_2 n - 1$ stages of $n/2 \ 2 \times 2$ switches. This requires a much lower amount of resources compared to a crossbar network. However, Beneš networks are *rearrangeable* networks in which establishing a new connection affects previously established connections. The proposed network consists of two stages. The first stage is constructed from $f \ n/f:1$ multiplexers where f is the network fan-in (equal to p/j) and n is the number of ports (equal to p). The second stage is constructed from $n \ f:1$ multiplexers. Thus, the network grows linearly with a constant fan-in. These three networks are compared and the results are shown in Figure 5.7 for two typical values of n . It can be seen that the proposed network is implemented with fewer resources as far as mid-range value of n is required. Additionally, the routing on the proposed network can be performed without needing any kind of special algorithms for routing on Beneš networks. Finally, it should be noted that if f is set to n , the proposed network will result in an $n \times n$ crossbar.

Table 5.3: Delivery network synthesized results for Xilinx Virtex-4 FPGA.

p	Slices			Control bits		
	[58]	[57]	This work	[58]	[57]	This work
16	5.24M	448	192	7680	56	44
32	10.48M	1152	640	15360	144	117

Two similar flexible LDPC decoders are presented in [58] and [57], and both are based on the same reformulation of the Sum-Product algorithm. However, this method does not benefit from simplified decoding algorithms such as the Offset Min-Sum engaged in this work. Additionally, the reformulation causes some redundancies, i.e. the same computations have to be done in both CNPs and VNPs. Therefore, the method proposed in this work will result in much smaller VNP and CNP architectures. The decoder in [57] employs a Beneš network for message passing. The message delivery in [58] is done through a complex network of multiplexers and registers that allows all VNPs and CNPs to access every single node on the Tanner graph. This necessitates a vast amount of resources that is not implementable for the case of large codes with high node degrees, and especially on the FPGA. The proposed delivery network is synthesized on Xilinx Virtex4 FPGA and is compared to architectures of [58] and [57] in table 5.3. A rate 0.9 LDPC code with length of 4000 and check node degree of 40 is considered for the comparisons since the architecture of [58] relies on the code. The enormous amount of slices used by [58] is a result of large code length and high check node degree. Comparison results show that the proposed delivery network requires fewer resources than other designs.

5.4 FPGA Implementation

To verify the proposed method for decoding random LDPC codes, it is desirable to implement the architecture on an FPGA platform. The implementation results such as throughput, resources and clock frequency can be used to examine the effectiveness of the proposed method. Although the required cycles for a single decoding iteration

have been previously derived, the actual throughput will be slightly lower due to the idle cycles inserted between different phases of operation. These idle cycles are usually needed after every state machine transition as a result of pipelining, i.e. a pipelined VNP requires a few more cycles to update the variable nodes, especially after receiving the last data at its input. However, a good design of the hardware will result in numbers that are close those shown in table 5.2.

The architecture of VNPs and CNPs are detailed in this section following the message delivery network that was discussed in section 5.3. The main challenge in the architecture design of VNP and CNP is the assumption of randomness in the LDPC codes. In decoding random LDPC codes, the sequence of incoming messages to the variable nodes and check nodes is unknown to them. Therefore, this sequence should be stored in a memory inside every CNP and VNP. The proposed memory data can be viewed as a micro-program that controls VNPs and CNPs. Since it is also desired to employ pipelining to increase the maximum clock frequency, the processing elements are implemented using an architecture similar to the pipelined processors. The proposed architectures for CNP and VNP are covered separately, despite their major similarities.

5.4.1 Variable node processor architecture

The variable node processor is responsible for updating variable nodes according to:

$$\lambda_n^{(l)} = \frac{2}{\sigma^2} r_n + \sum_{i \in M(n)} u_{i,n}^{(l)} \quad (5.10)$$

Based on the above equation, a VNP simply works as an accumulator in which the incoming messages to each variable node are added together and the result is stored in a register within that VNP. The register assigned to a variable node is updated when an incoming message to the VNP has to be delivered to that variable node. Since the number of variable nodes within a VNP is relatively high, the cost of employing individual registers built on flip-flops is high. A better solution is to implement a memory (RAM) in which memory entries are assigned to variable nodes. In this case, three steps should

be performed for every incoming message to the VNP: 1) The partially updated variable node value should be read from its memory location (Read); 2) The received value must be added to the value read from the memory (Update); 3) The updated value has to be written back to the memory (Write). Thus, three clock cycles are required for updating a variable node while a VNP is to handle an incoming message in one clock cycle.

A pipelined architecture for the VNP can be employed so that each of the three tasks are done in one stage of the pipeline. Therefore, no stage will be held idle at any time instance. This results in a throughput that is the same as when a variable node is updated in one clock cycle. However, the pipelining imposes a couple of issues caused by processing the same variable node at different stages. For instance, a memory collision will occur when the Read and Write stages are both accessing the same memory entry (same variable node).

The proposed pipelined architecture for the VNP is depicted in Figure 5.8. The architecture consists of four stages named as Fetch, Read, Update and Write. Note that the Write stage is illustrated implicitly.

The controller, which is located at the Fetch stage, processes every incoming message to the VNP. The controller constantly monitors the Empty signal of the input FIFO. If the Empty signal is cleared, the controller reads the FIFO by asserting its RD_EN signal. At the same time, it reads the next address of the Micro-Program and fetches one instruction. Therefore, there exists at least one instruction for every data written to the FIFO.

An instruction word, stored in the Micro-Program memory, contains information about the origin of the incoming message as well some control flags. The structure of the instruction words stored in the VNP's Micro-Program is shown in Figure 5.9. The first part of the instruction is REL_POS, which is the check node index of the message to be delivered to a variable node. In other words, it determines which of the variable nodes that are connected to the check node is being updated. The next part is ABS_POS, which points to the variable node that is to be updated. There are also four control flags

Figure 5.8: The pipelined VNP architecture

named as Skip (S), Hold (H), First Update (FU) and Last Update (LU).

It is possible that a CTV message written to FIFO be used for updating more than one variable node. In this case, the controller should not read the FIFO, but instead fetch the next instruction for the update of another variable node that is connected to the same check node. The Hold flag tells the controller about this situation. When the controller reads an instruction e.g. when Hold is set, it avoids reading data from FIFO until it reads the next instruction with a cleared Hold flag. Therefore, it is possible for the number of instructions to be larger than the number of data written to the FIFO.

Figure 5.9: The VNP Micro-Program instruction word.

The Read stage is responsible for reading back the partially updated values and also decoding of CTV data. Based on the variable node update equation, a variable node can be seen as a multiple input adder. However, each variable node is updated serially in the proposed architecture. Thus, an accumulator is needed instead of multiple input adder. The accumulated value, which represent a partially-updated variable node, is stored in a memory called LLR_MEM. There is another memory in the Read stage known as RX_MEM which stores the received channel data, and represents the first term in equation (5.10). These two values are passed on to the Update stage at the next rising edge of the clock. The check node value is also retrieved in the Read stage since it has been received in a packed format (Figure 5.5). The decoded value is then converted to the sign-magnitude format and be sent to the next stage.

The variable node equation is evaluated at the Update stage by a two-input adder. The received check node value is added to either the partially-updated variable node value or the channel received data (RX). When a variable node is updated for the first time, the partially-updated value is essentially zero, and hence, the check node value is added to the RX data. The FU flag indicates this situation, and is connected to the select line of the multiplexer which selects one of the adder inputs.

Finally, the updated values have to be written back to the memory. Due to the pipelined structure, LLR_MEM is being read and written at the same time. A dual port memory in which one port is dedicated to the read action and the second to the write

requests must hence be engaged. Dual port memories can be realized on the Xilinx FPGAs by means of dedicated Block RAMs (BRAMs). A BRAM is a true dual port RAM on the FPGA chip and it does not occupy any of look-up table resources.

The main problem with the pipelined architecture is the time during which different pipeline stages are processing the same variable node. For example, the Read stage is reading back the variable node from the memory while the Update stage is updating the same variable node. The value being read from the memory is invalid since the value of the update stage has not been written back yet. A simple solution to this problem is to stall the pipeline when consequent instructions point to the same variable node. To facilitate this, the Skip flag is employed which makes an instruction ineffective. When Skip is set to one, the controller ignores the current instruction and reads the next instruction. Consequently, the pipeline is stalled for one cycle and this results in a throughput reduction. In any case, the variable node degree is usually low for high-rate codes which makes the stall cycles rare.

Based on the instruction word structure of Figure 5.9, a maximum of 512 variable nodes can be assigned to a single VNP. This is because ABS_POS is represented in 9 bits. Additionally, every check node may have a maximum degree of 32 since REL_POS has 5 bits. Nonetheless, one may increase the number of variable nodes by assigning more bits to the instruction word. However, a width of 18 is optimized for Xilinx FPGAs since the BRAMs can be formatted to have such a width. RX_MEM is physically implemented on the same BRAM as the Micro-Program by taking advantage of dual port RAMs. A 2K×18-bit RAM is used for this purpose in which the first 1536 locations are dedicated to the instructions and the last 512 entries to the RX values. Two BRAMs are consumed and each is of 1K ×18-bit fashion. Another BRAM is used for the LLR_MEM, which yields a total of three BRAMs for a single VNP.

5.4.2 Check node processor architecture

A check node processor updates a group of check nodes on the Tanner graph based on the message passing algorithm. The check node update equation in the Offset Min-Sum algorithm is expressed as:

$$u_{m,n}^{(l)} = \prod_{i \in N(m) \setminus n} \text{sgn}(\lambda_i^{(l-1)} - u_{m,i}^{(l-1)}) \times \max\left\{\min_{i \in N(m) \setminus n} |\lambda_i^{(l-1)} - u_{m,i}^{(l-1)}| - \beta, 0\right\} \quad (5.11)$$

For the sake of simplicity, the CNP architecture is first described based on the Min-Sum algorithm. The required modification in order to adopt Offset Min-Sum algorithm is then explained. The Min-Sum check node update is similar to equation (5.11) in which the offset value, β , is set to zero.

Similar to the variable nodes, check nodes are updated serially in the proposed architecture. But unlike the variable nodes, a check node resembles an array of values where each of which has to be delivered to one of the variable nodes. When a check node receives a message from one of its connected variable nodes, it not only has to update the value to be sent back to that variable node, but also the values to the rest of the affected variable nodes. Therefore, the CNP should update a number of values that equals to the check node degrees upon receiving a single message. The Min-Sum algorithm (and its variants) greatly simplifies this task by allowing one to store all values corresponding to a single check node in a format similar to Figure 5.5. This format which was proposed for message delivery reduces the memory requirements for CNPs as well [10]. The binary word is called as a *row word* or simply a row. Once a row is updated, it implies that all of the edges of the Tanner graph connected to that check node are updated. As a result, the CNP task is to update a row whenever it receives any messages from the VNPs.

The pipelined architecture for CNP is illustrated in Figure 5.10, which consists of six stages. These stages are Fetch, Read Back, Decode, Pre-Process, Update and Write. The last stage is shown implicitly as it resides within the Pre-Process stage.

Figure 5.10: The pipelined CNP architecture.

The Micro-Program is located at the first stage of the pipeline. The controller monitors the input FIFO and fetches instructions whenever FIFO is not empty. The Micro-Program's instructions contain information of the VNP received messages, which con-

sists of two addresses and two flags. The first part of the instruction is ABS_POS. It is the absolute address of the check node that the received message should be delivered to. This address is basically the address of the memory location that stores the assigned check node value. The next part, REL_POS, indicates the variable node or the origin of the incoming message. Unlike the ABS_POS, this address is relative such that a variable node is addressed relative to the rest of the variable nodes connected to the check node. Therefore, REL_POS can have a maximum value equal to the check node degree. There exists two flags, Hold (H) and First Update (FU), which represent special cases. The Hold flag is set to one whenever the next instruction deals with the same variable node message. This causes the controller to hold the FIFO output while it fetches another instruction at the next clock cycle. The FU flag is sent through the pipeline showing that the processed check node is being updated for the first time in that iteration.

According to the check node update (5.11), a check node requires the variable node values (λ_i) as well as its own value from the previous iteration ($u_{m,i}^{(l-1)}$). The check node values from the previous iteration are stored in the CN_MEM_OLD memory that is located at the Read Back stage. This memory is addressed by the ABS_POS and the output data is then passed on to the following stage. The controller also generates a flag representing the first iteration of decoding. The First Iteration (FI) flag is connected to a two-input multiplexer at the output of CN_MEM_OLD. When the FI bit is one, the input of the multiplexer which is tied to zero is selected since the check node values have to be initialized with zero at the first iteration.

The row value read from the CN_MEM_OLD is decoded at the Decode stage. In this stage, the check node value is retrieved from the packed format (row format) by the REL_POS. It basically determines which of the connected variable nodes is currently involved in the update of that check node. Based on the row format, the value of Min2 is selected for the magnitude of the message if REL_POS is equal to MIN_POS. Otherwise, the Min is chosen as the magnitude. The sign of the message is also extracted from the Sign Pattern simply via a multiplexer where REL_POS is connected to its select lines.

Finally, both sign and magnitude are converted to 2's complement format.

The subtraction operation required in equation (5.11) is carried out at Pre-Process stage. Additionally, the partially updated check nodes are stored in the CN_MEM_NEW memory. The CN_MEM_NEW is a dual port memory such that one port is devoted to reading the partially updated rows while the second port is for writing back the updated rows as the final stage of the pipeline (Write). Furthermore, CN_MEM_NEW has a width that is equal to the width of the row word (53 here) but with an extra bit. This bit is called Sign Flag (SF) and is the product of the sign of the incoming messages which has to be updated with every received message. Thus, the CN_MEM_NEW requires a relatively large width but it has a small depth as far as high rate LDPC codes are concerned. For instance, 16 CNPs with 64 × 54-bit CN_MEM_NEW memories yield an LDPC decoder that can support parity-check matrices having a maximum of 1024 rows. Therefore, memories within CNPs are implemented using the distributed memory resources of the FPGA. A distributed memory is constructed using the FPGA's look-up tables as the memory cells. Alternatively, one can build such a memory by using two 512×32-bit BRAMs. However, the BRAMs would be heavily under-utilized.

The main function of check node update phase is carried out at the Update stage. Two modules are involved to update both of the sign and magnitude of the check nodes. The Min-Finder block compares the magnitude of the received value ($|\lambda_i^{(l-1)} - u_{m,i}^{(l-1)}|$) with Min and Min2 of the partially-updated row, and substitutes them if it is necessary. Accordingly, MIN_POS may be replaced with REL_POS. The Min-Finder architecture is shown in Figure 5.11. The Sign Updater module updates the sign pattern with the sign of the incoming message. Finally, the product of all signs is updated via a two input XOR gate which will they update the SF bit.

Lastly, the updated row has to be written back to CN_MEM_NEW. The output registers of the Update stage are tied back to the inputs of the memory at Pre-Process stage as illustrated in Figure 5.10. The Write Back (WB) initially produced by the controller serves as an indicator of valid data at every stage of the pipeline. The WB flag is con-

Figure 5.11: The Min-Finder block.

Figure 5.12: The data forwarding unit forwards the updated rows from the Write stage to the Pre-Process and Update stages.

nected to the write-enable (WE) input of memory at the Write stage, which ensures that only a valid data is written back to the memory.

The data read from the CN_MEM_NEW is invalid when the same check node is being processed at the Update or Write stages. Unlike the VNP architecture, inserting stall cycles may result in a large delay as far as high rate LDPC codes are concerned. A better solution, similar to a simple pipelined reduced instruction set computer (RISC) processor, is to forward the data from the Write stage to the Pre-Process and Update stages. The data forwarding is depicted in Figure 5.12. It is achieved by simply comparing the

Figure 5.13: The modifications to the pipelined CNP to support the row correction.

ABS_POS of different stages.

Once all the check nodes are updated, a *row correction* operation has to be done in order to complete the check node equation. Three tasks have to be done for this purpose:

- 1) The sign pattern of rows stored in CN_MEM_NEW may be inverted according to the SF bit of the same row;
- 2) The Min and Min2 should be adjusted if the Offset Min-Sum algorithm is employed;
- 3) The corrected rows need to be written to the CN_MEM_OLD to be used at the next iteration.

Therefore, the pipelined CNP is modified according to Figure 5.13. The row correction introduces only a few extra cycles to each iteration since high-rate codes have rather small number of check nodes.

5.4.3 FPGA implementation results

The proposed message packing decoder was implemented using 16 CNPs and 16 VNPs and the processors were connected to each other using the described VTC and CTV networks. A Xilinx Virtex4 xc4fx100-10c FPGA, that has 42176 slices and 376 Block RAMs, was employed in this study. The synthesis and implementation are carried out using Xilinx ISE 10.1 software. Additionally, the Modelsim XE is used for the purpose of behavioral simulation. After the design was functionally verified and implemented, the Xilinx ChipScope tool was used in order to test and debug the real hardware. The modern in-circuit debug tools, such as Xilinx ChipScope tool, allows one to avoid lengthy timing simulations in many cases. Nonetheless, timing simulations are still considered as a powerful tool, especially when the in-circuit debugging results does not match the behavioral simulations.

Although the proposed LDPC decoder is universal, one may assign a set of conditions on the supported LDPC code to minimize the amount of resources with respect to the properties of high rate codes. The proposed VTC network of Figure 5.4 is optimized for variable nodes with degree of 4. Moreover, choosing the maximum variable node degree as four simplifies the VTC network control as the select signals of the first stage multiplexers can be connected together. The maximum check node degree was determined by the width of CN_MEM_OLD and CN_MEM_NEW memories at CNPs. These memories are implemented on the distributed memory resources of the FPGA, which are expensive in terms of FPGA resources. Therefore, the maximum check node degree is limited to thirty-two. The depth of these two memories, which has to be a multiple of 16, can determine the maximum number of check nodes. Choosing a depth of 48 will lead to a decoder capable of supporting parity check matrices with maximum $48 \times 16 = 768$ rows. The depth of LLR_MEM of VNP has a direct impact on the maximum variable nodes. The memory has a depth of 512, and hence, the parity-check matrix can have up to 8192 columns. Based on these assumptions, the decoder is implemented and the results are presented in table 5.4.

Table 5.4: Physical implementation results of message passing LDPC decoder

Technology	Xilinx xc4fx100-10ff1517
Clock Frequency	125 MHz
Slices	20,978
LUTs	38,078
Flip-Flops	12,799
BRAMs	75
Parallelism	16
Quantization Format	(9:5)
Max. VNs	8192
Max. CNs	768
Max. VN Degree	4
Max. CN Degree	32

Table 5.5: The LDPC decoder performance for several codes.

Code	VN Degree	CN Degree	Edges	Decoding Cycles per Iteration	Throughput [Mbps/Iter.]
I (4095,3358) Unstructured	3	17	12285	2310	221.16
II (4095,3359) Unstructured	4	23	16380	2593	197.4
III (4161,3430) Unstructured	4	23	16644	2672	194.66
IV (1057,813) Unstructured	3	13	3171	819	161.32
V (2048,1536) Quasi-cyclic	4	16	8192	1340	191.04
VI (2048,1664) 10GBASE-T	6	32	12288	1800	142.22

The throughput of the proposed decoder vary from code to code based on the parity-check matrix specifications. Therefore a fixed throughput cannot be expressed. However, the decoder has been tested for a set of six LDPC codes and the results are provided in table 5.5. The accurate number of cycles for each iteration of decoding has been derived from the Modelsim simulations. The maximum clock frequency, 125 MHz for this technology, is obtained by the static timing report of Xilinx tools. By knowing these two values, one can derive the throughput per iteration. The maximum number of iteration has been set to 25. But, the actual iterations depends on the SNR level of the received signal. For high SNR values, an average of three iterations are carried out before the decoder converges to a valid codeword.

After presenting the results for different codes, it will be interesting to compare the

decoder with other published LDPC decoders. The comparison will however be difficult since the majority of works support only a particular class of codes. Thus, it is desirable to compare the results with respect to the supported type of LDPC codes.

In table 5.6, the proposed architecture is compared with other promising LDPC decoders published in the past few years [30, 31, 35, 36, 37, 29, 28, 57]. Since the proposed decoder's throughput in this work relies on the code, one of the sample codes, code II, is chosen for the comparison. When different FPGA designs are compared, one may observe that the proposed design consumes slightly more resources of the FPGA than other designs. This is due to the implementation of CNP memories and FIFOs on the distributed memories of the FPGA, which are relatively expensive compared to the Block RAMs.

In order to have a fair throughput comparison, a new parameter is defined Edge Processing Time (EPT). EPT is basically the required time for the processing of a single edge of the Tanner graph in one decoding iteration. It can be simply calculated by dividing a single decoding iteration time by the total number of edges of the Tanner graph. EPT allows one to compare the processing power of different decoders regardless of the code specifications, such as block length and code rate. Nonetheless, the parity-check matrix structure has a significant impact on the EPT besides the decoder's architecture. As far as the EPT is concerned, the results reveal that the architectures intended for structured codes [30, 31, 35, 36, 37, 29, 28] outperform the proposed message packing decoder and on average, by an order of magnitude. This data represents a remarkable achievement since it has provided such a flexible platform that is not confined to any specific class of code. The most important feature of this work is thus the possible support of different codes without having to change the hardware. Among the presented LDPC decoders in the literature, a flexible solution by Masera *et al.* [57] has been proposed. They present an ASIC implementation that achieves an EPT of 1075 ps for a (1024,506) LDPC code. This EPT value is closed to the EPT of this work (1270 ps). However, a fair comparison is hard to obtain since they are implemented on different targets and their

Table 5.6: Comparison of proposed decoder with existing LDPC decoders.

	Platform	Architecture	Code	Throughput per Iter.	Clock Frequency	Area\ Resources	EPT (ps)
Zhang [30]	FPGA XCV2600E	Partially-Parallel	(9216,4608) (3,6)-regular	972 Mbit/sec	56 MHz	11792 slices 90 BRAMs	343
Wang [31]	FPGA XC2V6000	Partially-Parallel	(8176,7154) QC	2.58 Gbit/s	192 MHz	23052 slices 128 BRAMs	97
Mansour [35]	ASIC 0.18 μ m	Partially-Parallel	(2048,1024) QC-irregular	6.4 Gbit/s	125 MHz	14.3 mm ²	52
Karkooti [36]	FPGA XC4VFX60	Partially-Parallel	(1296,864) QC-irregular	500 Mbit/s	160 MHz	15235 slices 123 BRAMs	657
Fewer [37]	FPGA XC4VLX40	Partially-Parallel	(6912,5184) Block-structured	4.16Gbit/s	181 MHz	10970 slices 46 BRAMs	NA
Liu [29]	ASIC 90 nm	Partially-Parallel	(2304,1920) QC-irregular	2.1 Gbit/s	150 MHz	6.25 mm ²	187
Hao [28]	ASIC 65 nm	Partially-Parallel	(4608,4096) QC-regular	33.6 Gbit/s	300 MHz	1.92 mm ²	7.44
Masera [57]	ASIC 0.13 μ m	partially-parallel	(1024,506) Unstructured	249.6 Mbit/sec	245 MHz	2.94 ²	1075
This work	FPGA XC4FX100	partially-parallel	(4095,3359) Unstructured	197.4 Mbit/sec	125 MHz	20978 Slices 75 BRAMs	1270

tested codes are different. For instance, the ASIC technology is permitted [57] to run their decoder at a frequency twice higher than this work and this has a direct impact on the EPT. Nevertheless, this work represents a unique FPGA implementation of flexible LDPC decoders, serving as an essential building block for FPGA-based simulation of LDPC codes.

5.5 Summary

In this chapter, a solution for the decoding of high-rate unstructured LDPC codes is proposed. The proposed solution introduces a new method that simplifies the delivery network significantly by applying a schedule with an aim to minimize the collisions. The delivery methods for variable nodes and check nodes are different due to the specifications of high-rate LDPC codes. The delivery to the check nodes is based on a well-known computer algorithm known as Backtracking where an efficient heuristic is applied. The delivery to the variable nodes is a row-wise operation in which the messages of a single check node (equal to a row of the parity check matrix) are packed and sent to all the vari-

able nodes. The algorithm and the delivery network are also simulated and the delivery time is evaluated for different cases.

Based on the proposed solution, a partially parallel LDPC decoder with 16 processing elements of each type is implemented on Xilinx Virtex4 FPGA. The FPGA implementation, which runs at a clock frequency of 125 MHz, is able to achieve a good decoding throughput while ensuring the ability of decoding randomly designed codes. Furthermore, the decoder can be employed for the performance investigation of LDPC codes via the hardware simulations.

Chapter 6

FPGA for LDPC Code Evaluation

This chapter describes the design of a multiple-FPGA system for the simulation of LDPC codes performance at very low BERs. The system to be described incorporates two FPGA, one models the transmitter and the other models the channel and LDPC decoding. Architecture of the LDPC encoder and the random data source will be described briefly. Details on the design of channel model (AWGN) and LDPC decoder can be found in chapters 4 and 5, respectively. After implementing the system, a series of simulations are carried out to evaluate the performances of some high-rate LDPC codes for which the performance curves are presented.

6.1 FPGA-based simulator architecture

The block diagram of the proposed LDPC code simulator is given in Figure 6.1. This simulator is implemented on an Annapolis Wildstar-4 board that has four Xilinx Virtex-4 FPGAs. Three of the FPGAs on this board are chosen from LX series (LX100 here), which are called Computational Processing Elements (CPEs). The fourth FPGA is a Virtex-4 FX100 FPGA, and is called the I/O Processin Element (IOPE). Virtex-4 FX series are optimized for embedded computations, in which two PowerPC processors are embedded on the FPGA silicon. Additionally, the FX series contains greater number of embedded Block RAMs (BRAMs) than the other series. However, they have relatively

Figure 6.1: FPGA-based simulator system.

fewer logic gates.

The first FPGA device is devoted to a random bit generator, which feeds the LDPC encoder with a block of random bits with a size that equates to an information bits frame. Unlike the parity-check matrix, the generator matrix is not sparse, i.e. the number of ones and zeros are comparable. Therefore, one has to dedicate a greater number of memory to store its structure on the hardware. It was hence decided that the encoder be implemented on the FX100 FPGA (IOPE), which is rich in BRAMs.

The random bit generator was implemented using a 52-bit Linear Feedback Shift Register (LFSR). An n -bit LFSR sequences through $(2^n - 1)$ states, where n is the number of registers. At each clock cycle, the contents of registers are shifted one position to the right. Feedback from the predefined registers to the input of first register is put in place through an XOR or XNOR gate. In this work, the 52-bit LFSR derives its feedback from the 52th and 49th bits as suggested by [61], and this was implemented on the FPGA fabric as depicted in Figure 6.2. Finally, the XNOR type was selected to bring the LFSR into the initial state simply by clearing all of the registers.

Any generator matrix of an (n, k) code can be reduced by row operations and column

Figure 6.2: 52-bit LFSR implemented using three slices of Virtex-4 FPGA.

permutations to the *systematic form*,

$$\mathbf{G} = [\mathbf{I}_k | \mathbf{P}] = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & | & p_{11} & p_{12} & \cdots & p_{n-k} \\ 0 & 1 & 0 & \cdots & 0 & | & p_{21} & p_{22} & \cdots & p_{2n-k} \\ \vdots & \vdots & \vdots & & \vdots & | & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 & | & p_{k1} & p_{k2} & \cdots & p_{kn-k} \end{bmatrix} \quad (6.1)$$

where $\mathbf{I}_k$ is the $k \times k$ identity matrix and $\mathbf{P}$ is a $k \times (n - k)$ matrix that determines the $n - k$ parity bits. Taking advantage of the systematic form, one should only multiply the information bits vector by the $\mathbf{P}$ matrix and the resulted vector is put together with the information bits to form a codeword. Due to the assumption of random parity-check matrix, $\mathbf{P}$ matrix does not have a known structure too and its structure should be stored on the FPGA. In this work, it is assumed that each frame (block) of information bits has

Figure 6.3: The LDPC encoder architecture.

a maximum size of 4096. Therefore, a single BRAM can store four columns of $\mathbf{P}$ matrix, when it is set to a $4k \times 4$ -bit configuration. As far as high-rate codes are concerned, the FPGA has enough BRAMs to support different LDPC codes. Consequently, the parity-check bits or redundant bits are generated using an architecture shown in Figure 6.3. The encoder is fed with random information bits and the parity bits is ready as soon as all information bits have been received. Moreover, flip-flops containing parity bits are tied to each other such that they form a large shift-register. This register is shifted to convert the parity bits into a serial format, which are sent to the second FPGA along with the information bits as a valid codeword.

The communication between two FPGAs are handled by means of two FIFO buffers, where each of them is implemented on one of the FPGAs. In order to transmit data, one has to write data to the output FIFO of FPGA 1. The output FIFO is connected to a controller, which monitors its status signals, such as Empty, and reads the FIFO if data has been written to it. The data is sent to the second FPGA through differential pair wires. However, the transmission mechanism is hidden to the user and is done through some cores provided by the board manufacturer. The user only needs to write data to the FIFO and make sure that no overflow occurs. The input FIFO of the second FPGA is filled with data received from the first device. The user has to monitor the Empty signal

of this FIFO and reads it if Empty signal is cleared. The width of two FIFOs is 88 bits, and this implies that data should be transmitted in blocks of 88 bits.

The channel model and LDPC decoder are implemented on the second FPGA. Although the realistic model of magnetic channels is much more complex, the AWGN model can be considered as a starting point for realizing a more accurate model. The AWGN generator generates random samples with Gaussian distribution (zero mean and standard deviation of one) in a quantized (16:11) format. The AWGN generator is designed based on the Box-Muller method [43], which was detailed in Chapter 4. The output of the AWGN generator is multiplied by the standard deviation σ of the noise level, which can be derived by:

$$\frac{E_b}{N_o} = \frac{1}{2R\sigma^2} \quad (6.2)$$

where E_b/N_o is the SNR per bit and R is the code rate. After adjusting the noise level, it is rounded off to a (9:5) format, which is the data representation format in the decoder.

The decoder is modified so that it can help to compute some simulation statistics besides performing the decoding task. A 48-bit counter keeps the track of simulated blocks, and is implemented using DSP48 resources of Virtex-4 FPGA. A DSP48 block consists of an 18-bit multiplier plus a 48-bit adder, which form a Multiply-Accumulate unit. The dedicated 48-bit adder eliminates the long delay of carry chain when the adder is implemented on FPGA's look-up tables. Furthermore, it is able to count the number of simulated blocks up to 10^{14} , and this is large enough for most of the simulations. The noiseless codeword is forwarded to the decoder along the noisy block to compute the number of error bits by comparing it to the actual codeword. Once a codeword is received at the input of the decoder, it is split into equal size blocks, where each of them is sent to one of the VNPs. If the maximum number of iterations has been reached and the decoder is not converged, a hard decision will be made in each of the VNPs and the estimated bit is compared to the actual codeword bit. Hence, every VNP will find the number of errors in the assigned codeword bits. Finally, the number of error bits of different VNPs are added together to form the total number of errors in a block. There is

Figure 6.4: The implementation and verification model.

also a 10-bit counter that counts up whenever the decoder does not converge. This value is used as the number of error blocks.

6.2 FPGA-based simulator design flow

The design flow is based on a hierarchical model that accommodates the verification and implementation as shown in Figure 6.4. This hierarchy consists of several programming codes in C, MATLAB and VHDL, all managed and accessed by a MATLAB graphical user interface (GUI). After the design is implemented on the FPGA devices another interface written in visual C, controls the FPGA system through the PCI-X interface of the workstation. This simple interface sends commands such as start, stop and SNR value. Additionally, it collects the simulation results including number of error bits, error blocks and total simulated blocks in real-time.

The building blocks of the system are described in VHDL and are highly parameterized in order to adapt the LDPC encoder and decoder to each given LDPC code. Different constants, including memory initialization files, are defined in a single VHDL

package, which is called by other VHDL files. Once an LDPC code is selected by the user, the MATLAB GUI software runs an executable file with the proper input parameters. The executable file, which performs the message packing algorithm, saves the packing results on a file. The packing results file is later read by the MATLAB code, which generates the memory initialization files according to it. Additionally, other parameters such as maximum number of decoding iterations and offset value of Offset-Minsum algorithm are adjusted through the GUI. Finally, the MATLAB code generates the VHDL package and copies it along with the memory files to the main VHDL design files location.

The complete VHDL design was deployed for both verification and implementation. Two levels of verification were performed: algorithmic verification and in-circuit test. A bit-precise model of the LDPC decoder and AWGN channel was developed. This model generates a number of files that contain variable node values and check node values during different iterations. A VHDL testbench, which is run by ModelSim-XE software, performed behavioral simulations to compare and verify MATLAB and VHDL results. Hence, the simulator system can be functionally tested for any given codewords. The VHDL design was synthesized and implemented using Xilinx ISE tools called from the Windows Command Prompt. The generated Bitstreams are downloaded on FPGAs through a PCI interface. The Xilinx Chipscope tool is employed for real-time in-circuit verification and debugging. The debug cores are inserted into the design Netlist before the implementation phase and are accessed via JTAG connection. For debugging purpose, the system has been modified such that it performs only one decoding iteration each time it receives a certain command from the host PC. Some test points, such as the number of bit errors at the end of the iterations, have been chosen and their respective signal values were compared to behavioral simulation results. The design has been verified in both phases of mentioned tests.

6.3 LDPC code evaluation

Using the simulator developed, several high-rate LDPC codes with different structures are evaluated.

Figure 6.5 shows the FPGA performance results for (2048,1664) LDPC code defined by IEEE 10GBASE-T standard [62], which has a column weight of 6 and block length of 2048. The decoder runs up to 25 iterations, but the decoding of a block may stop at lower iterations as soon as it converges. As seen in Figure 6.5, there is no trace of error floor at BERs of 10^{-10} and below. This very good performance is a result of high column weight (variable node degree) as suggested by [42]. In Figures 6.6 and 6.7, two unstructured codes, taken from Mackay's website [22], are investigated. These codes have a rate of 0.82 and block lengths of 4161 and 4095, respectively. Despite similar code rates, the (4161,3430) code outperforms (4095,3358) code mainly due to its higher column weight. However, both unstructured codes suffer from error-floor at high SNR values. Finally, a (4092,3564) and rate 0.87 quasi-cyclic code is simulated using the FPGA platform and its performance is illustrated in Figure 6.8. The aforementioned LDPC code has a column weight of 4 and consists of 132×132 size sub-matrices. The performance curve reveals a slight decline in performance at SNRs around 5, which is considered as error-floor.

Figure 6.5: (2048,384) LDPC code, column weight 6 and 25 decoding iterations.

Figure 6.6: (4161,3430) LDPC code, column weight 4 and 15 decoding iterations.

Figure 6.7: (4095,3358) LDPC code, column weight 3 and 15 decoding iterations.

Figure 6.8: (4092,538) QC-LDPC code, column weight 4 and 15 decoding iterations.

6.4 Summary

In this chapter, the design of an FPGA-based simulator for performance evaluation of LDPC codes over an AWGN channel is discussed. Both LDPC encoder and decoder are designed such that they are capable of supporting arbitrarily-constructed LDPC codes. A 52-bit LFSR is employed as the random information source, which has a large period of 10^{15} samples. The random bit generator and LDPC encoder are implemented on a single FPGA while random codewords are passed to the second FPGA device, which implements channel model and LDPC decoder. A MATLAB GUI is used to set the simulation parameters, which are translated to design files by means of a VHDL package and memory initialization files. The FPGA-based simulator is employed to evaluate the performance of some high-rate LDPC codes in the presence of AWGN. The system is able to examine the performance of such codes in BER regions as low as 10^{-10} in a couple of hours.

Chapter 7

Conclusion

7.1 Conclusion

In this work, LDPC decoding techniques were examined from a system and architecture point of view. Efficient methods and architectures were introduced to address the decoder design for hardware-based simulator.

In chapter 4, a new simplified LDPC decoding algorithm was proposed. The algorithm attempts to minimize the computational complexity of the conventional Sum-Product algorithm by reducing the number of mathematical computations to two for each check node. From a different angle, the proposed algorithm is similar to the Min-Sum family algorithms but with the addition of a correction factor. Based on the proposed algorithm, a low cost serial decoder is implemented on the Xilinx Virtex-II FPGA. A Hardware/Software design methodology is employed to facilitate the development of the system by means of adding extra functions into the C code being run by the processor. The Hardware/Software system was used to investigate the performance of some high rate LDPC codes, suitable for the magnetic recording devices over an AWGN channel. The simulator provides a throughput of 2Mbits/sec and is able to observe the code performance at bit error rates as low as 10^{-9} in a few hours. The systems throughput is remarkable when it is compared to the software simulations where they hardly achieve 100 kbits/sec. Moreover, the system is low area (utilization ratio of 6%) and

can be synthesized on small size and low cost FPGAs. The development board used for this purpose costs only few hundred dollars but it can attain speed that rival expensive parallel computers. The implemented simulator is then employed to compare the performance of the proposed algorithm with similar LDPC decoding algorithms, such as the Min-Sum and Offset Min-Sum. The bit error rate curves demonstrated that the proposed algorithm performs slightly better in the waterfall region than the two other algorithms. However, all the algorithms perform quite close to each other in the error floor region. Hence, one can decide whether to choose the algorithm based on the working region or the tradeoff between the hardware overhead and the performance.

For the code performance investigation, although the decoder discussed in chapter 4 offers a much higher throughput compared to software simulations, it might not be sufficient for many applications. In chapter 5, a solution for the decoding of unstructured codes is proposed. The proposed method is interconnection aware since it significantly reduces the complexity of the interconnection network for the message passing significantly. The proposed method, which is called message packing, looks for messages that can be delivered simultaneously to the target nodes in the message passing algorithm and creates packs of messages. Based on this method, a flexible LDPC decoder is implemented on the Xilinx Virtex-4 FPGA, which is run at clock frequencies as high as 125 MHz while delivering a decoding throughput of at least 100 Mb/s per iteration depending on the LDPC code. While providing a comparable flexibility, this throughput is about two orders of magnitude higher than the serial decoder of chapter 4. The proposed LDPC decoder was tested for several high-rate LDPC codes. The experiments revealed the effectiveness of message packing method on various arbitrarily constructed LDPC codes. Additionally, the implemented decoder consumes the same amount of resources as the code-specific partially-parallel decoders presented in the literature.

Chapter 6 describes a simplified magnetic recording system emulator in which the channel is assumed to be AWGN. The proposed FPGA-based simulator consists of two FPGAs connected to each other. The first FPGA implements a random data source along

with an LDPC encoder that is capable of encoding random LDPC codes. The AWGN channel and the LDPC decoder are implemented on the second FPGA. The platform can run the simulations at rates up to tens of megabits per second, which enables the investigation of LDPC codes at BER regions as low as 10^{-10} in a couple of hours. The achieved throughput is three orders of magnitude higher than the software simulations. This high performance is a direct result of parallelism at both the LDPC decoder and the system level. In the decoder level, several processing elements work in parallel to carry out the LDPC decoding algorithm while the whole FPGA-based system consists of different blocks such as encoder, channel model and the decoder which are working in parallel. Finally, the performance curves of some interested high-rate LDPC codes have been derived using the simulator in order to verify its functionality.

7.2 Future work

In the current platform, a random codeword is transferred over an AWGN channel and fed into the decoder. Since the main objective of the proposed idea is emulating a magnetic recording channel, it is interesting to develop a more accurate channel model. Besides the Gaussian noise, other sources of disturbances exist in the magnetic media such as the transition noise and Inter-Symbol Interference (ISI). Once an accurate channel model is considered, the channel noise is no longer white and channel detectors such as soft output Viterbi algorithm (SOVA) or BCJR are required so as to remove ISI. In this case, the detector should be placed between the channel model and the LDPC decoder. Another interesting thing is to increase the level of parallelism by using multiple FPGAs, with each working with independent source of data. The current FPGA board (Annapolis Wildstar 4) has four Virtex 4 FPGAs coupled to each other. Fortunately, the LDPC encoder is designed so that it can generate multiple independent codewords in parallel with minimum extra gates. Therefore, the encoder FPGA can simultaneously provide three other FPGAs with independent codewords to yield a three times simulation speed enhancement.

Publications

1. S. M. E. Hosseini, K. S. Chan and W. L. Goh, "A reconfigurable FPGA implementation of an LDPC decoder for unstructured codes," *Signals, Circuits and Systems, 2008. SCS 2008. 2nd International Conference on*, pp.1-6, Tunisia, 7-9 Nov. 2008.
2. S. M. E. Hosseini, W. L. Goh and K. S. Chan, "A general decoding framework for high-rate LDPC codes," *12th International Symposium on Integrated Systems. ISIC 2009*, Singapore, 14-16 Dec. 2009. Accepted.

Bibliography

- [1] Paul Frank and Roger Wood. A perspective on the future of hard disk drive (hdd) technology. *Asia-Pacific Magnetic Recording Conference, 2006*, pages 1–2, Nov. 2006. [cited at p. 1]
- [2] Ajay Dholakia, Evangelos Eleftheriou, Thomas Mittelholzer, and Marc P. C. Fossorier. Capacity-approaching codes: Can they be applied to the magnetic recording channel? *IEEE Communications Magazine*, 4:122–130, 2004. [cited at p. 2]
- [3] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. 1. In *Communications, 1993. ICC 93. Geneva. Technical Program, Conference Record, IEEE International Conference on*, volume 2, pages 1064–1070 vol.2, May 1993. [cited at p. 2, 3]
- [4] R. G. Gallager. Low-density parity-check codes. *Institute of Radio Engineers Transactions on Information Theory*, IT-8(1):21–28, 1962. [cited at p. 2, 3, 10, 12, 18]
- [5] T. Mittelholzer, A. Dholakia, and E. Eleftheriou. Reduced-complexity decoding of low density parity check codes for generalized partial response channels. *Magnetics, IEEE Transactions on*, 37(2):721–728, Mar 2001. [cited at p. 2]
- [6] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 1948. [cited at p. 2, 9]
- [7] John G. Proakis. *Digital Communications*. McGraw-Hill, 2001. [cited at p. 2, 10]
- [8] D. J. C. MacKay and R. M. Neal. Good codes based on very sparse matrices. In *Cryptography and Coding. 5th IMA Conference. Proceedings*, Cryptography and Coding, 5th IMA Conference. Proceedings, pages 100–11. Springer-Verlag, 1995. [cited at p. 3, 12, 31]

- [9] D. J. C. MacKay and R. M. Neal. Near shannon limit performance of low density parity check codes. *Electronics Letters*, 33(6):457–8, 1997. [cited at p. 3, 10, 14]
- [10] M. M. Mansour and N. R. Shanbhag. High-throughput ldpc decoders. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(6):976–996, 2003. [cited at p. 3, 30, 36, 38, 84, 95]
- [11] Tom Richardson. Error floors of ldpc codes. In *41st Annual Conference on Communications, Control and Computing*, pages 1426–1435, 2003. [cited at p. 4, 26]
- [12] Lingyan Sun, Hongwei Song, Z. Keirn, and B.V.K.V. Kumar. Field programmable gate array (fpga) for iterative code evaluation. *Magnetics, IEEE Transactions on*, 42(2):226–231, Feb. 2006. [cited at p. 5]
- [13] R. C. Bose and D. K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Inform. Control*, 3:68–79, March 1960. [cited at p. 10]
- [14] John R. Barry. Low-density parity-check codes. Online. Available: <http://users.ece.gatech.edu/barry/>. [cited at p. 12, 19]
- [15] Niclas Wiberg. *Codes and decoding on general graphs*. PhD thesis, Linkoping University, 1996. [cited at p. 13, 19]
- [16] J. Chen, A. Dholakia, E. Eleftheriou, M. P. C. A. Fossorier M. P. C. Fossorier, and X. Y. Hu X. Y. Hu. Reduced-complexity decoding of ldpc codes. *Communications, IEEE Transactions on*, 53(8):1288–1299, 2005. [cited at p. 19, 20, 22, 23, 43, 54, 55]
- [17] Chen Jinghu, R. M. Tanner, C. Jones, and A. Yan Li Yan Li. Improved min-sum decoding algorithms for irregular ldpc codes. In R. M. Tanner, editor, *Information Theory, 2005. ISIT 2005. Proceedings. International Symposium on*, pages 449–453, 2005. [cited at p. 20, 21]
- [18] S. L. Howard, C. Schlegel, and V. C. Gaudet. A degree-matched check node approximation for ldpc decoding. In C. Schlegel, editor, *Information Theory, 2005. ISIT 2005. Proceedings. International Symposium on*, pages 1131–1135, 2005. [cited at p. 21]
- [19] Zhao Jianguang, F. Zarkeshvari, and A. H. Banihashemi. On implementation of min-sum algorithm and its modifications for decoding low-density parity-check (ldpc) codes. *Communications, IEEE Transactions on*, 53(4):549–554, 2005. [cited at p. 23]

- [20] Daesun Oh and K.K. Parhi. Performance of quantized min-sum decoding algorithms for irregular ldpc codes. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 2758–2761, May 2007. [cited at p. 24]
- [21] E. R. Berlekamp. The methodology of erro correcting codes. *Information Theory, IEEE Transactions on*, 24:384–386, 1980. [cited at p. 24]
- [22] D.J.C. MacKay. Encyclopedia of sparse graph codes. [Online]. Available: <http://www.inference.phy.cam.ac.uk/mackay/codes/data.html>. [cited at p. 25, 65, 86, 114]
- [23] D. MacKay and M. Postol. Weaknesses of margulis and ramanujan-margulis low-density parity-check codes. *Electronic Notes in Theoretical Computer Science*, 74:1–8, 2003. [cited at p. 25]
- [24] Sun Lingyan, Song Hongwei, Z. Keirn, and B. V. K. V. A. Kumar B. V. K. V. Kumar. Field programmable gate array (fpga) for iterative code evaluation. *Magnetics, IEEE Transactions on*, 42(2):226–231, 2006. [cited at p. 26]
- [25] E. Cavus, C.L. Haymes, and B. Daneshrad. An is simulation technique for very low ber performance evaluation of ldpc codes. In *Communications, 2006. ICC '06. IEEE International Conference on*, volume 3, pages 1095–1100, June 2006. [cited at p. 26]
- [26] A. J. Blanksby and C. J. Howland. A 690-mw 1-gb/s 1024-b, rate-1/2 low-density parity-check code decoder. *Solid-State Circuits, IEEE Journal of*, 37(3):404–412, 2002. [cited at p. xi, 28, 29, 30, 50, 71]
- [27] Liu Chih-Hao, Yen Shau-Wei, Chen Chih-Lung, A. Hsie-Chia Chang Hsie-Chia Chang, A. Chen-Yi Lee Chen-Yi Lee, A. Yar-Sun Hsu Yar-Sun Hsu, and A. Shyh-Jye Jou Shyh-Jye Jou. An ldpc decoder chip based on self-routing network for ieee 802.16e applications. *Solid-State Circuits, IEEE Journal of*, 43(3):684–694, 2008. [cited at p. 28]
- [28] Zhong Hao, Xu Wei, Xie Ningde, and A. Tong Zhang Tong Zhang. Area-efficient min-sum decoder design for high-rate quasi-cyclic low-density parity-check codes in magnetic recording. *Magnetics, IEEE Transactions on*, 43(12):4117–4122, 2007. [cited at p. xii, 28, 42, 43, 50, 103, 104]

- [29] C. H. Liu, S. W. Yen, C. L. Chen, H. C. Chang, C. Y. Lee, Y. S. Hsu, and S. J. Jou. An ldpc decoder chip based on self-routing network for ieee 802.16e applications. *IEEE Journal of Solid-State Circuits*, 43(3):684–694, 2008. [cited at p. xi, 28, 41, 42, 43, 50, 51, 72, 103, 104]
- [30] Zhang Tong and K. K. Parhi. A 54 mbps (3,6)-regular fpga ldpc decoder. In *Signal Processing Systems, 2002. (SIPS '02). IEEE Workshop on*, pages 127–132, 2002. [cited at p. xi, 31, 32, 50, 103, 104]
- [31] Z. Wang and Z. Cui. Low-complexity high-speed decoder design for quasi-cyclic ldpc codes. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(1):104–114, 2007. [cited at p. xi, 32, 33, 34, 50, 72, 103, 104]
- [32] M. P. C. Fossorier. Quasicyclic low-density parity-check codes from circulant permutation matrices. *Information Theory, IEEE Transactions on*, 50(8):1788–1793, 2004. [cited at p. 32, 86]
- [33] Z. Wang and Z. Cui. A memory efficient partially parallel decoder architecture for quasi-cyclic ldpc codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(4):483–488, 2007. [cited at p. xi, 35, 36]
- [34] Jan M. Rabaey. *Digital Integrated Circuits*. Prentice Hall, 2003. [cited at p. 36]
- [35] M. M. Mansour and N. R. Shanbhag. A 640-mb/s 2048-bit programmable ldpc decoder chip. *Solid-State Circuits, IEEE Journal of*, 41(3):684–698, 2006. [cited at p. xi, 39, 40, 41, 43, 50, 103, 104]
- [36] Joseph R. Cavallaro Marjan Karkooti, Predrag Radosavljevic. Configurable, high throughput, irregular ldpc decoder architecture: Tradeoff analysis and implementation. In *Application-specific Systems, Architectures and Processors, 2006. ASAP 2006. IEEE conference on*, 2006. [cited at p. 39, 43, 50, 103, 104]
- [37] C. P. Fewer, M. F. Flanagan, and A. D. Fagan. A versatile variable rate ldpc codec architecture. *Circuits and Systems I: Regular Papers, IEEE Transactions on [Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on]*, 54(10):2240–2251, 2007. [cited at p. 41, 43, 50, 103, 104]
- [38] He Zhiyong, Roy Sebastien, and Fortier Paul. Fpga implementation of ldpc decoders based on joint row-column decoding algorithm. In Roy Sebastien, editor, *Circuits and*

- Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1653–1656, 2007. [cited at p. 41]
- [39] Dale E. Hocevar. A reduced complexity decoder architecture via layered decoding of ldpc codes. In *Signal Processing Systems, 2004. (SIPS '04). IEEE Workshop on*, pages 107–112, 2004. [cited at p. 41]
- [40] *IEEE Std 802.16e-2005 and IEEE Std 802.16-2004/Cor 1-2005 (Amendment and Corrigendum to IEEE Std 802.16-2004)*, 2006. [cited at p. 41]
- [41] L. Zhang, L. Gui, Y. Xu, and W. Zhang. Configurable multi-rate decoder architecture for qc-ldpc codes based broadband broadcasting system. *Broadcasting, IEEE Transactions on*, 54(2):226–235, 2008. [cited at p. 43]
- [42] Zhong Hao, Zhong Tong, and F. Haratsch Erich. Quasi-cyclic ldpc codes for the magnetic recording channel: Code design and vlsi implementation. *Magnetics, IEEE Transactions on*, 43(3):1118–1123, 2007. [cited at p. xii, 44, 45, 46, 52, 114]
- [43] D. U. Lee, J. D. Villasenor, W. Luk, and P. H. W. A. Leong P. H. W. Leong. A hardware gaussian noise generator using the box-muller method and its error analysis. *Computers, IEEE Transactions on*, 55(6):659–671, 2006. [cited at p. xii, 44, 61, 62, 111]
- [44] B. Levine, R. Reed Taylor, and H. Schmit. Implementation of near shannon limit error-correcting codes using reconfigurable hardware. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 217–226, 2000. [cited at p. xii, 46, 47, 50]
- [45] Lei Xiong, Zhenhui Tan, and Dongping Yao. The moderate-throughput and memory-efficient ldpc decoder. In Zhenhui Tan, editor, *Signal Processing, The 8th International Conference on*, volume 3, 2006. [cited at p. 46]
- [46] G. Falcao, V. Silva, M. Gomes, and L. Sousa. Edge stream oriented ldpc decoding. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 237–244, Feb. 2008. [cited at p. 48]
- [47] Nvidia cuda architecture. Online, March 2009. [cited at p. 49]
- [48] Gabriel Falcão, Leonel Sousa, and Vitor Silva. Massive parallel ldpc decoding on gpu. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 83–90, New York, NY, USA, 2008. ACM. [cited at p. 49]

- [49] P. Urard, E. Yeo, L. Paumier, P. Georgelin, T. Michel, V. Lebars, E. Lantreibecq, and B. Gupta. A 135mb/s dvb-s2 compliant codec based on 64800b ldpc and bch codes. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 446–609 Vol. 1, 2005. [cited at p. 51, 72]
- [50] Hu Xinde, B. V. K. Vijaya Kumar, Li Zongwang, and R. Barndt. Error floor estimation of long ldpc codes on partial response channels. In *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pages 259–264, 2007. [cited at p. 52]
- [51] Hu Xiao-Yu, E. Eleftheriou, D. M. Arnold, and A. A. Dholakia A. Dholakia. Efficient implementations of the sum-product algorithm for decoding ldpc codes. In E. Eleftheriou, editor, *Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE*, volume 2, pages 1036–1036E vol.2, 2001. [cited at p. 55]
- [52] V. G. Oklobdzija. An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(1):124–128, 1994. [cited at p. 59]
- [53] D. Lee, A. Abdul Gaffar, O. Mencer, and W. Luk. Minibit: Bit-width optimization via affine arithmetic. In *Proc. ACM/IEEE Design Automation Conference*, pages 837–840, 2005. [cited at p. 63]
- [54] Chen Yanni and D. Hocevar. A fpga and asic implementation of rate 1/2, 8088-b irregular low density parity check decoder. In *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, volume 1, pages 113–117 Vol.1, 2003. [cited at p. 72]
- [55] Sha Jin, Gao Minglun, Zhang Zhongjin, A. Li li Li li, and A. Zhongfeng Wang Zhongfeng Wang. Efficient decoder implementation for qc-ldpc codes. In Gao Minglun, editor, *Communications, Circuits and Systems Proceedings, 2006 International Conference on*, volume 4, pages 2498–2502, 2006. [cited at p. 72]
- [56] Dai Yongmei, Yan Zhiyuan, and Chen Ning. Optimal overlapped message passing decoding of quasi-cyclic ldpc codes. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(5):565–578, 2008. [cited at p. 72]
- [57] G. Masera, F. Quaglio, and F. Vacca. Implementation of a flexible ldpc decoder. *Circuits and Systems II: Express Briefs, IEEE Transactions on [see also Circuits and Systems*

- II: Analog and Digital Signal Processing, IEEE Transactions on*], 54(6):542–546, 2007.
[cited at p. 72, 74, 75, 77, 89, 103, 104]
- [58] Kang Se-Hyeon and Park In-Cheol. Loosely coupled memory-based decoding architecture for low density parity check codes. *Circuits and Systems I: Regular Papers, IEEE Transactions on [Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on]*, 53(5):1045–1056, 2006. [cited at p. xii, 72, 74, 75, 76, 89]
- [59] F. Quaglio, F. Vacca, C. Castellano, A. Tarable, and G. Masera. Interconnection framework for high-throughput, flexible ldpc decoders. In *Proceedings -Design, Automation and Test in Europe, DATE*, volume 2, 2006. [cited at p. 77]
- [60] V. E. Benes. Optimal rearrangeable multistage connection networks. *Bell System Technical Journal*, 43:1641–1656, 1964. [cited at p. 77]
- [61] Maria George and Peter Alfke. Linear feedback shift registers in virtex devices. Technical report, Xilinx Inc., 2007. [cited at p. 108]
- [62] IEEE standard for information technology-telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access method and physical layer specifications. IEEE Std 802.3an-2006 (Amendment to IEEE Std 802.3-2005), 2006. [cited at p. 114]