



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ & ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

**Μελέτη, προσομοίωση και υλοποίηση
πομποδέκτη κωδίκων LDPC σε Xilinx FPGA
για εφαρμογές ασύρματων επικοινωνιών**

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΝΕΚΤΑΡΙΟΥ-ΓΕΩΡΓΙΟΥ Α. ΦΥΤΡΑΚΗ

Επιβλέπων: Δημήτριος Σούντρης
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2010



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ & ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

**Μελέτη, προσομοίωση και υλοποίηση
πομποδέκτη κωδίκων LDPC σε Xilinx FPGA
για εφαρμογές ασύρματων επικοινωνιών**

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΝΕΚΤΑΡΙΟΥ-ΓΕΩΡΓΙΟΥ Α. ΦΥΤΡΑΚΗ

Επιβλέπων: Δημήτριος Σούντρης
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή τη 13η Ιουλίου 2010.

.....
Δημήτριος Σούντρης
Επίκουρος Καθηγητής Ε.Μ.Π.

.....
Κιαμάλ Πεκμεστζί
Καθηγητής Ε.Μ.Π.

.....
Γεώργιος Οικονομάκος
Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2010

.....

Νεκτάριος-Γεώργιος Α. Φυτράκης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Νεκτάριος-Γεώργιος Α. Φυτράκης, 2010.

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Σκοπός της παρούσας διπλωματικής εργασίας είναι η μελέτη των τεχνικών κωδικοποίησης διαύλου που χρησιμοποιούνται στις ψηφιακές επικοινωνίες και των υλοποιήσεών τους. Εξαιτίας του θορύβου στον οποίο είναι εκτεθειμένες οι επικοινωνίες, και ο οποίος μπορεί να εισάγει σφάλματα στα μηνύματα που μεταδίδονται από τον πομπό στο δέκτη, οι τεχνικές αυτές περιλαμβάνουν τρόπους ανίχνευσης και διόρθωσης σφαλμάτων που επιτρέπουν την ανακατασκευή των αρχικών δεδομένων.

Οι κώδικες LDPC (Low-Density Parity-Check) συμπεριλαμβάνονται στους πιο δυνατούς γραμμικούς κώδικες διόρθωσης σφαλμάτων, καθώς επιτρέπουν επιδόσεις κοντά στο όριο της χωρητικότητας του καναλιού. Για το λόγο αυτό είναι πολύ διαδεδομένοι και αρκετά καινούρια πρότυπα ψηφιακών επικοινωνιών βασίζονται σε αυτούς, όπως το πρόσφατο πρωτόκολλο DVB-S2 για τις δορυφορικές μεταδόσεις της ψηφιακής τηλεόρασης.

Η αποκωδικοποίηση των κωδίκων LDPC είναι μια επαναληπτική διαδικασία που χρησιμοποιεί τον αλγόριθμο belief-propagation, ο οποίος είναι μια τεχνική μετάδοσης μηνυμάτων πραγματικών τιμών ανάμεσα στις ακμές ενός γράφου του κώδικα. Επειδή υλοποιήσεις υψηλής πολυπλοκότητας και λειτουργία υψηλής κατανάλωσης αντίκεινται στους αποκωδικοποιητές υψηλής ταχύτητας, προτείνονται επιπλέον μερικές απλοποιήσεις και βελτιστοποιήσεις.

Τέσσερις διαφορετικές τεχνικές δοκιμάστηκαν για την επίδοσή τους σε κανάλια θορύβου έπειτα από την προσομοίωσή τους με μηνύματα διαμορφωμένα κατά Binary Phase-shift keying (BPSK) και μεταδιδόμενα μέσω καναλιών λευκού αθροιστικού θορύβου (AWGN), Rician και Rayleigh fading.

Στο τελευταίο στάδιο, η τεχνική με την καλύτερη αναλογία επίδοσης προς πολυπλοκότητα επιλέχθηκε για την υλοποίηση ενός LDPC πομποδέκτη σε ένα FPGA της οικογένειας Spartan-3E της Xilinx. Το τελικό σχέδιο αναλύεται λεπτομερώς, περιγράφοντας τα συστατικά του και τη λειτουργία τους, ενώ στο τέλος σχολιάζεται το ποσοστό χρησιμοποίησης του υλικού της πλατφόρμας που απαιτείται.

Λέξεις κλειδιά: ψηφιακές επικοινωνίες, κωδικοποίηση διαύλου, πομποδέκτης, κωδικοποιητής, αποκωδικοποιητής, block codes, ανίχνευση σφαλμάτων, διόρθωση σφαλμάτων, fading channels, low-density parity-check codes, belief propagation, επαναληπτική αποκωδικοποίηση, πρώιμος τερματισμός, VHDL, FPGA

Abstract

The purpose of this diploma thesis is the study of the digital communications channel code decoding schemes and their hardware implementations. Since many communication channels are subject to channel noise, and thus errors may be introduced during transmission from the source to the receiver, these schemes include error detection and correction techniques which enable reconstruction of the original data.

The Low-Density Parity-Check (LDPC) codes are among the most powerful linear error correcting codes, since they enable performance near the limits of the channel capacity. As a result, they have received a lot of attention and several new digital communication standards have adopted them, such as the recent DVB-S2 standard for the satellite transmission of digital television.

The decoding of LDPC codes is an iterative process which uses the belief-propagation algorithm, a message passing technique which defines real-valued messages passing along edges in a code graph. Since high complexity implementations and high power operation are inappropriate for high-speed LDPC decoders, several proposals for optimization and simplifications are also described.

Four different decoding techniques have been tried and tested for their performance under noisy channel by computer-based simulations of messages modulated under the Binary Phase-shift keying (BPSK) modulation scheme and transmitted through Additive White Gaussian Noise (AWGN), Rician and Rayleigh fading channel models.

Finally, the algorithm with the best ratio of performance versus complexity is chosen as the decoding scheme of an LDPC transceiver implemented on a FPGA platform of the Xilinx Spartan-3E family. The design is analyzed in detail, describing its components and their operation and, ultimately, the device utilization required.

Keywords: digital communication, channel coding, transceiver, encoder, decoder, block codes, error detection, error correction, fading channels, low-density parity-check codes, belief propagation, iterative decoding, early termination, VHDL, FPGA

Ευχαριστίες

Με την ολοκλήρωση της διπλωματικής μου εργασίας, θα ήθελα να απευθύνω τις ευχαριστίες μου στους ανθρώπους που συνέβαλαν ουσιαστικά στην εκπόνησή της.

Καταρχάς, στον επιβλέποντα καθηγητή μου κ. Δ. Σούντρη, που με ώθησε στο αντικείμενο των ενσωματωμένων συστημάτων και με καθοδήγησε από το πρώτο μέχρι το τελευταίο στάδιο της εργασίας. Επίσης, στον καθηγητή κ. Κ. Πεκμεστζή και τον λέκτορα κ. Γ. Οικονομάκο, για το ενδιαφέρον τους στη διπλωματική μου εργασία και για τη συνεισφορά τους ως μέλη της εξεταστικής επιτροπής.

Στα πλαίσια της εργασίας μου, είχα την ευκαιρία να συνεργαστώ με τους διδάκτορες Γ. Κιοκέ και Δ. Μπεκιάρη, η βοήθεια των οποίων υπήρξε καθοριστική στη διαμόρφωση του τελικού αποτελέσματος και της μορφής της εργασίας.

Τέλος, η πιο σημαντική βοήθεια και υποστήριξη προήλθε από την οικογένειά μου που ήταν πάντα δίπλα μου κάθε στιγμή. Για το λόγο αυτό, οι κόποι, οι προσπάθειες, οι επιτυχίες και οι αποτυχίες, τα ξενύχτια με τα διαβάσματα καθώς και όλες οι αναμνήσεις από τα φοιτητικά μου χρόνια, με επιστέγασμα την εκπόνηση αυτής της διπλωματικής εργασίας, αφιερώνονται σε αυτούς που νοιάστηκαν περισσότερο για να τα βιώσω· στους γονείς μου και στον αδερφό μου...

Αθήνα, Ιούλιος 2010

Contents

1	Introduction	1
1.1	Telecommunication systems	1
1.2	Signals	2
1.3	Transceivers	2
1.4	Low-Density Parity-Check (LDPC) codes	3
1.5	Field-Programmable Gate Arrays (FPGAs)	3
1.6	Motivation and objectives	3
1.7	Document structure	4
2	The communication model	7
2.1	The classical communication scheme	7
2.2	Optimal code decoding	8
2.2.1	The Shannon–Hartley theorem	8
2.2.2	The Nyquist-Shannon sampling theorem	10
2.2.3	Optimal word decoding	10
2.2.4	Optimal symbol decoding	11
2.3	Linear block codes decoding	11
2.3.1	Binary block codes	11
2.3.2	Decoding of binary block codes	12
2.3.3	Belief propagation	15
3	Low-Density Parity-Check (LDPC) codes	19
3.1	History	19
3.2	Description of LDPC codes	20
3.2.1	Definition	20
3.2.2	Classes of LDPC codes	20
3.2.3	Code rate	21
3.3	Construction of LDPC codes	22
3.3.1	Random based construction	22
3.3.2	Deterministic based construction	22
3.4	Encoding	22
3.4.1	Lower-triangle shape based encoding	22
3.4.2	Low-density generator matrices	23
3.4.3	Cyclic parity-check matrices	24

3.4.4	Iterative encoding	24
3.5	Decoding	24
3.5.1	Scheduling	24
3.5.2	Performance	25
4	Low-power and memory-efficient LDPC decoding	27
4.1	Min-sum decoding	27
4.2	Low power parallel decoders	28
4.2.1	Partially-parallel decoders	28
4.2.2	Fully-parallel decoders	31
4.3	Early termination	32
4.3.1	Description	32
4.3.2	Implementation	33
4.4	Quantization	33
4.4.1	Description	33
4.4.2	Function	34
5	Performance of BPSK-modulated LDPC codes	37
5.1	Decoding techniques for LDPC codes	37
5.1.1	Hard-decision (bit-flip) decoders	37
5.1.2	Soft-decision probability-domain SPA decoders	38
5.1.3	Log-domain SPA decoders	40
5.2	Bit error rate analysis	41
5.2.1	Performance analysis of hard-decision (bit-flip) decoders . .	43
5.2.2	Performance analysis of soft-decision probability-domain decoders	45
5.2.3	Performance analysis of log-domain decoders	45
5.2.4	Performance analysis of simplified log-domain decoders . .	48
5.3	Fading channels	51
5.3.1	Rayleigh fading	51
5.3.2	Rician fading	55
6	Implementation of an LDPC transceiver	63
6.1	Design Summary	63
6.2	Encoder implementation	65
6.2.1	Description	65
6.2.2	Implementation	66
6.2.3	Simulation	69
6.3	Decoder implementation	74
6.3.1	Description	74
6.3.2	Implementation	76
6.3.3	Simulation	82
7	Conclusion and future perspectives	91

A	MATLAB simulations source codes	95
A.1	Hard-decision (bit-flip) decoder	95
A.2	Probability-domain SPA decoder	96
A.3	Log-domain SPA decoder	98
A.4	Simplified log-domain SPA decoder	100
B	Implementation source codes	103
B.1	Block RAM VHDL source code	103
B.2	Encoder VHDL source codes	104
B.2.1	Encoder implementation	104
B.2.2	Encoder simulation testbench	108
B.3	Decoder VHDL source codes	111
B.3.1	Decoder implementation	111
B.3.2	Decoder simulation testbench	122
	Bibliography	127

List of Figures

2.1	Basic communication model	7
2.2	Example of a $(6, 3)$ linear code in \mathbb{F}_2	13
2.3	An example of a cycle free bipartite graph of a code \mathcal{C}	15
2.4	The operations of the graph of Figure 2.3 for calculating T_1	16
2.5	The operations of the graph of Figure 2.3 for calculating T_3	17
3.1	Shape of parity-check matrix for efficient encoding as proposed by (MacKay, Wilson and Davey 1999)	23
3.2	Proposal for efficient encoding of a parity-check matrix by (Richardson and Urbanke 2001)	24
3.3	Typical error probability curve of iterative decoding algorithms	26
4.1	A partially-parallel LDPC decoder	28
4.2	Supply voltage reduction obtainable by increased parallelism	29
4.3	The fully-parallel iterative LDPC decoder architecture	31
5.1	MATLAB simulation flowchart of an LDPC transceiver	42
5.2	Message decoding with belief propagation	44
5.3	BER analysis of a hard-decision (bit-flip) decoder under AWGN channel	46
5.4	BER analysis of a probability-domain SPA decoder under AWGN channel	47
5.5	BER analysis of a log-domain SPA decoder under AWGN channel	49
5.6	BER analysis of a simplified log-domain SPA decoder under AWGN channel	50
5.7	Density of Rayleigh flat fading envelope with $N = 10^6$ and $f_D = 100\text{Hz}$	52
5.8	Performance of Rayleigh fading channels compared to AWGN channels	52
5.9	BER analysis of a hard-decision (bit-flip) decoder under Rayleigh channel	53
5.10	BER analysis of a probability-domain SPA decoder under Rayleigh channel	53
5.11	BER analysis of a log-domain SPA decoder under Rayleigh channel	54
5.12	BER analysis of a simplified log-domain SPA decoder under Rayleigh channel	54
5.13	Performance of Rician fading channels compared to AWGN and Rayleigh channels for different values of K	55

5.14	BER analysis of a hard-decision (bit-flip) decoder under Rician channel	58
5.15	BER analysis of a probability-domain SPA decoder under Rician channel	59
5.16	BER analysis of a log-domain SPA decoder under Rician channel . .	60
5.17	BER analysis of a simplified log-domain SPA decoder under Rician channel	61
5.18	BER performance of a simplified log-domain SPA decoder operating for 7 iterations under AWGN, Rician and Rayleigh fading channels .	62
6.1	Transceiver's FPGA implementation block diagram	64
6.2	Encoder technology schematic top view	65
6.3	Encoder memory utilization	69
6.4	Block RAM component technology schematic top view	69
6.5	Full view of the encoder simulation	71
6.6	Encoder simulation during initialization	71
6.7	The first execution of the encoder simulation	72
6.8	Stages between different input messages sent to the encoder	72
6.9	Time required to compute parity-check bits	73
6.10	Error check on the bits exported by the encoder	73
6.11	Decoder technology schematic top view	74
6.12	Quantizer in the decoder block	75
6.13	Decoder memory utilization	77
6.14	Division of the block RAMs utilized by the decoder into smaller banks	79
6.15	Full view of the decoder simulation	84
6.16	Initialization and first execution of the encoder simulation	85
6.17	Stages of the decoding process	86
6.18	Time required to calculate the decoded bits	87
6.19	The messages which the decoder receives in its input	88
6.20	Bit error rate extraction on the bits exported by the decoder	89

Chapter 1

Introduction

Overview:

The first chapter outlines the field of interest of this thesis by introducing the main keywords and notations which are used in the text. First, the concept of telecommunication systems is presented, followed by the piece of information they transfer and exchange, signals. Afterwards, the definition of the transceivers is presented, succeeded by one of the available error correcting codes they use, the Low-Density Parity-Check coding scheme. Finally, the integrated circuits which will be used to implement this application, the Field-Programmable Gate Arrays, are also discussed.

1.1 Telecommunication systems

In today's modern society, the term *communication* enters people's lifestyle in many different and various ways, making it difficult to observe the diversity of its forms. Whether at job or at leisure, people come across various modern means of *telecommunication* and *digital communication* systems, such as the radio, telephone, television and the Internet. Aided by these means and systems of communication, people are able to instantaneously contact others, however far they maybe, stay informed about whatever happens everywhere, commit everyday transactions and, of course, entertain. It has nowadays become impossible to imagine a world without these means, even though most of them were only invented during the previous century.

The piece of information exchanged by the communication systems also varies, as its form can be visual, audio, text, or a mixture of them. However, regardless the kind of information, it can generally be considered as, simply, a *signal*. The telecommunication systems viewpoint adopted in this text will focus on the attributes and various possible manipulations of the electrical signals that characterize these systems.

1.2 Signals

In the fields of communications, in signal processing and generally in electrical engineering, a *signal* is any time-varying or spatial-varying quantity that carries a certain amount of information. In the physical world, any quantity measurable through time or over space is actually a signal, supposed that it contains some piece of useful or usable information (otherwise it is considered as *noise*). Signals used in telecommunication systems are usually time-variable, therefore their *independent variable* is usually time.

Communication signals (taken to be a function of time) are usually easier to handle when represented over frequency. Information represented in the time domain describes when something occurs and what the amplitude of the occurrence is. In contrast, information represented in the frequency domain is more indirect; by measuring the frequency, phase, and amplitude of a signal showing periodic motion, information can often be obtained about the system producing the motion. The representation over time or frequency can be achieved by means of *signal processing*, which is the area of applied mathematics that deals with analyzing signals, in either discrete or continuous time in order to perform useful operations on them.

The complete range of frequencies of a system or signal, from the lowest to the highest, is its frequency spectrum. Specifically, signals and systems whose range of frequencies is measured from zero to a maximum *bandwidth* (or highest signal frequency) are described as *baseband*.

1.3 Transceivers

Mainly used in wireless communications, a *transceiver* is a combined transmitter/receiver in a single package. It is, therefore, a device capable of both transmitting and receiving analog or digital signals, and this ability is extensively used in cellular telephones or two-way radios for instance. If the transmit and receive functions do not share common circuitry or a single housing, the device is referred to as a transmitter-receiver instead. *Physical layer (PHY)* transceivers are commonly used for sending and receiving network signals over various types of channels, like telephone lines, optic fiber, or the air. When the receiver is silenced while transmitting, the transceiver is in *half-duplex* mode; on the contrary, when allowing reception of signals during transmission periods, the transceiver is in *full-duplex* mode.

Transceivers are able to perform consistency checking in the background while operating. Use of an optimal error correcting code allows data transmission at rates near the channel capacity with arbitrarily low probability of error.

1.4 Low-Density Parity-Check (LDPC) codes

In information theory, *Low-Density Parity-Check* codes are a sub-class of linear error correcting coding schemes, which are methods of transmitting messages over noisy transmission channels. LDPC codes can be described as the null space of a sparse $\{0, 1\}$ -valued parity-check matrix as well as by a bipartite graph, *Tanner graph*, which represents the rows and columns of the parity-check matrix.

LDPC codes, originally developed in 1963 by Robert Gallager but long time ignored due to impractical implementation, have now become more popular in modern communication systems with advanced VLSI technology. Several new digital communication standards have adopted LDPC codes due to the excellent error correction performance they feature, their inherently-parallel decoding algorithm and freedom from patent protection. Thus, LDPC decoders appear to be the best solution for future communication applications that demand performance near the limits of the channel capacity by transmitting close to the theoretical limit (*Shannon limit*).

1.5 Field-Programmable Gate Arrays (FPGAs)

It has been a recent trend to implement modern communication applications, such as digital signal processing, radio, aerospace and defense systems, on integrated circuits designed to be configured by the customer or designer after manufacturing (hence *field-programmable* gate arrays – *FPGAs*). Though FPGAs can be used to implement any logical function, they especially find applications in any area or algorithm that can make use of the massive parallelism offered by their architecture.

FPGAs consist of an array of programmable logic components, called *logic blocks*, and a hierarchy of reconfigurable interconnects (*routing channels*) that allow the blocks to be wired together. Most FPGAs include memory elements, which may be simple flip-flops or more complete blocks of memory.

The behavior of the FPGA is defined by a schematic design or a *hardware description language* (*HDL*), the most common being *VHDL* and *Verilog*. Then, using an electronic design automation tool, a technology-mapped netlist is generated that can be fitted to the actual FPGA architecture, finally (re)configuring the FPGA to a specific application.

The *Spartan-3E* family, which will be used to implement a transceiver as part of this thesis, is a high-performance FPGA family fabricated by *Xilinx*. Each platform offers a variety of features to address the needs of a wide variety of advanced logic designs.

1.6 Motivation and objectives

Traditionally, the study of the digital communication systems has been increasingly attractive, as a result of the ever-growing demand of data communication and

the digital processing options and flexibilities which are not available with analog transmission. Since digital circuits are less subject to distortion and interference than their analog counterparts, they have been used in almost all recent communication applications and this trend does not seem to fade in the future.

Communication involves sending data over noisy channels, which results in introducing erroneous samples during transmission from the source to the receiver. In order to overcome this and achieve reliable transmission, error detection and correction techniques are utilized, which enable reconstruction of the original data. Such techniques are typically adopted in every communication scheme nowadays; therefore, research is constantly focused on inventing increasingly better performing schemes, with an ambition to ultimately approach the channel capacity (Shannon limit).

The invention of LDPC codes has had a major impact on telecommunication systems due to their ability to perform close to the Shannon limit by using an iterative algorithm. For this reason, several new digital communication standards have adopted them as the coding scheme of their choice. In addition, the fact that several research groups have recently developed LDPC decoders running on FPGAs, has made LDPC codes more popular in advanced communication systems, such as *Mobile WiMAX*, with advanced VLSI technology.

The aim of this thesis is to review the concepts of digital communication, study the effect of the decoding schemes on the performance of the LDPC codes, compare and contrast their strong and weak points under several channels, and finally implement a generic LDPC transceiver which utilizes the best performing decoding technique.

1.7 Document structure

In order to present the work provided in this thesis, the document is organized in the following way:

Chapter 2 initially presents the classical communication scheme and discusses the optimal code decoding of transmitted information. Then, the decoding of linear block codes is also discussed and it is finally shown that optimal code decoding is possible using an iterative algorithm.

Chapter 3 introduces the concept of LDPC codes, first by briefly reviewing their history and then by defining their operation. Afterwards, the construction of LDPC codes is described. The chapter also discusses the encoding and decoding of LDPC codes, focusing on their performance.

Chapter 4 investigates hardware architectures for LDPC decoders amenable to low-complexity implementation as well as low-voltage and low-power operation. It is shown that increased parallelism coupled with reduced supply voltage is an effective technique to reduce the power consumption of LDPC decoders, which have inherent parallelism. In addition, a scheme to efficiently terminate the

iterative decoding earlier, when convergence has been detected, is proposed to further reduce the power consumption. Finally, the chapter describes a quantization scheme, which can be used to increase the memory efficiency while decreasing the hardware complexity of the implementation, at the cost of storing slightly inaccurate information.

Chapter 5 presents a performance analysis of four different LDPC codes decoding techniques. It describes the operation of hard-decision (bit-flip), probability-domain and log-domain decoders, along with a simplified version of the log-domain decoding algorithm. In addition, it presents the results of a computer-based simulation in order to describe the performance, in terms of bit error rate, versus implementation cost trade-off imposed by each technique. The decoders are simulated under Additive White Gaussian Noise (AWGN) and fading channels and the results of the simulations offer graphic comparison between them.

Chapter 6 presents the implementation of an LDPC transceiver on a Xilinx Spartan-3E FPGA. The chapter initially presents an overview of the whole design and afterwards analyzes each entity of it in detail. This is done firstly by describing the operation of each entity and the components used by them, then, by presenting the operation stages of the encoding and decoding procedures and finally, by verifying the correct operation of each entity by run-time simulations of the implemented design.

Chapter 7 offers an overall discussion over the work covered in this thesis, presenting the final remarks and future perspectives of the designed LDPC transceiver.

In addition, two appendices are included to present the source codes which have been used to simulate the different decoding techniques and implement the transceiver on a FPGA.

Chapter 2

The communication model

Overview:

In this chapter the classical communication scheme is initially reviewed. Then the optimal code decoding is discussed, preceded by the theorems which affect its operation. Finally, the decoding of linear block codes is discussed and it is shown that, under the cycle-free hypothesis, the optimal code decoding is possible using a simple iterative algorithm.

2.1 The classical communication scheme

The basic scheme for channel code encoding and decoding is depicted in Figure 2.1. The typical communication model consists of the *source* block which delivers information, the *encoder* block which delivers a coded version of the originally sent message, the *channel* over which the message is transmitted and, finally, the *decoder* block, which in Figure 2.1 may be one of two possible types.

Information derived from the source block is delivered by the mean of sequences of row vectors x of length K . These sequences pass through the encoder block, thus producing a *codeword*, which is the coded version of x . The encoding process delivers a codeword, a row vector c of length N , based on the encoding scheme used in the communication system. The *code rate* of the selected encoder is defined by the ratio:

$$R = \frac{K}{N} \quad (2.1)$$

The codeword c is then sent over a, usually noisy, channel, which is the medium the message is transmitted through. Therefore, a new vector y is created, which is

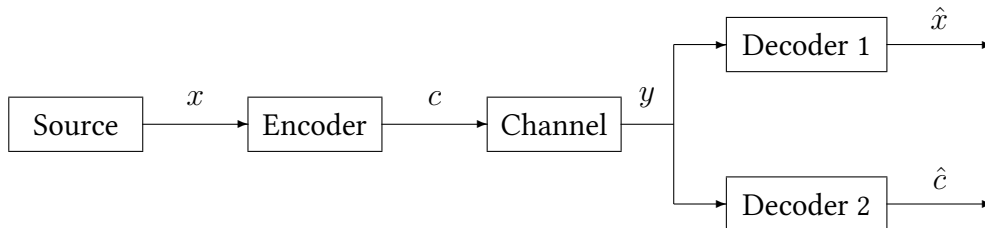


Figure 2.1: Basic communication model

then received by the decoder. Vector y is a distorted version of c , provided that the channel is not ideal, so its length is also N elements. The channel is a non-deterministic mapper between its input c and its output y . We make two assumptions: first, that y depends on c via a conditional probability density function (PDF), $p(y|c)$; and second, that the channel is memoryless, therefore $p(y|c) = \prod_{n=1}^N p(y_n|c_n)$.

Figure 2.1 depicts two possible types of decoder: type 1 decoders try to compute the best estimation \hat{x} of the source word x , while type 2 decoders aim at computing the best estimation \hat{c} of the sent codeword c derived from the encoder. The latter type of decoders then post process \hat{c} , extracting \hat{x} by a reverse process of encoding, when the code is non-systematic¹.

Both types of decoder can perform two methods of decoding: *soft decoding*, during which the output samples y_n of the channel are not decided, and *hard decoding*, during which the output samples y_n are decided. Soft decoders use the channel specifications to compute the probability for each y_n to be each one of the code-alphabet element denoted $y_{d_i} \in \mathcal{A}_C : \{\Pr(y_n = y_{d_i}), 0 \leq i \leq |\mathcal{C}|\}$, and then decide upon the value of this probability. The output of soft decoders includes both the decided word $y_d = (y_{d_1}, \dots, y_{d_N})$, $y_{d_i} \in \mathcal{A}_C$, where \mathcal{A}_C denotes the alphabet of the code symbols, and the probability of each decided symbol $\Pr(y_n = y_{d_n})$. On the other hand, hard decoders decide without using the knowledge of the probability set, since each of the output samples y_n of the channel is associated with the most probable code-alphabet element, followed by a processing performed on $y_d = (y_{d_1}, \dots, y_{d_N})$ trying to detect and correct the transmission errors.

2.2 Optimal code decoding

2.2.1 The Shannon–Hartley theorem

Optimal channel coding refers to establishing a communication link, transmitting the maximum amount of error-free digital data (*information*) that can be transmitted with a specified bandwidth in the presence of the noise interference, under the assumption that the signal power is bounded and the Gaussian noise process is characterized by a known power or power spectral density. The famous channel coding theorem, as demonstrated by Claude E. Shannon and Ralph V. L. Hartley [1], states that below a maximum rate R equal to the capacity C of the channel, it is possible to find error correction codes to achieve any given probability of error:

Shannon theorem for channel coding. *Let a discrete channel have the capacity C and a discrete source the entropy per second H . If $H \leq C$ there exists a coding system such that the output of the source can be transmitted over the channel with an*

¹In coding theory, a *systematic code* is one in which the input data are embedded in the encoded output, hence adding redundant information to data (e.g. transmitting data with a checksum). On the contrary, a *non-systematic code* is one in which the output does not contain the input bits.

arbitrarily small frequency of errors (or an arbitrarily small equivocation). If $H \geq C$ it is possible to encode the source so that the equivocation is less than H .

When comparing different coding schemes, their performance is measured by their gap to the capacity. The *Shannon capacity* of the band-limited AWGN channel is given by

$$C = B \log_2 (1 + \text{SNR}) \text{ [bit/s]} \quad (2.2)$$

where B refers to the channel bandwidth and SNR is the sound-to-noise ratio:

$$\text{SNR} = \frac{P_S}{P_N} \quad (2.3)$$

where P_S is the transmitted signal power and P_N the channel noise power. In addition we have:

$$P_N = N_0 B \quad (2.4)$$

$$P_S = R_I E_b \quad (2.5)$$

where N_0 is the one sided noise power spectrum density, E_b is the energy per information bit and R_I is the information rate, defined by:

$$R_I = \frac{R \log_2 \mathbb{M}}{T_s} \text{ [bit/s]} \quad (2.6)$$

where R is the code rate, \mathbb{M} is the size of the constellation of the modulation, while T_s is the symbol time duration.

Combining equations 2.4, 2.5 with 2.3 we have that:

$$\text{SNR} = \frac{R_I E_b}{B N_0} = \eta \left(\frac{E_b}{N_0} \right) \quad (2.7)$$

where E_b/N_0 is the normalized signal-to-noise ratio (SNR) measure (sometimes called the “signal-to-noise ratio per information bit”), and η is the spectral efficiency, defined as the ratio of information rate to channel bandwidth:

$$\eta = \frac{R_I}{B} \quad (2.8)$$

By definition, the maximal information rate is equal to the capacity, therefore we get the following equations:

$$\eta_{\max} = \frac{R_{I_{\max}}}{B} = \frac{C}{B} \quad (2.9)$$

using the definition of η as shown in equation 2.8, which, using equation 2.2, yields to:

$$R_{I_{\max}} = B \log_2 \left(1 + \eta_{\max} \left(\frac{E_b}{N_0} \right) \right) \quad (2.10)$$

thus giving:

$$\eta_{\max} = \log_2 \left(1 + \eta_{\max} \left(\frac{E_b}{N_0} \right) \right) \quad (2.11)$$

finally giving the *Shannon bound* as:

$$\left(\frac{E_b}{N_0}\right)_{\min} = \frac{2^{\eta_{\max}} - 1}{\eta_{\max}} \quad (2.12)$$

or

$$\left(\frac{E_b}{N_0}\right) \geq \frac{2^{\eta} - 1}{\eta} \quad (2.13)$$

Notice that the Shannon limit on E_b/N_0 is a monotonic function of η . For $\eta = 2$, it is equal to $3/2$ (1.76 dB), for $\eta = 1$, it is equal to 1 (0 dB) and as $\eta \rightarrow 0$, it approaches $\ln 2 \approx 0.69$ (−1.59 dB), which is called the *ultimate Shannon limit* [2] on E_b/N_0 .

Summing up, the Shannon theorem can be summarized as a conditional check of the information rate's relation to the channel capacity: reliable communications require that $R_I \leq C$ whereas $R_I \geq C$ leads to unreliability. The limit of reliable data rate of a channel depends on bandwidth and signal-to-noise ratio (SNR) according to equation 2.2, which can be solved to get the Shannon-limit bound on E_b/N_0 as in equation 2.12.

2.2.2 The Nyquist-Shannon sampling theorem

According to the the Nyquist-Shannon sampling theorem [1], also known as the *Cardinal Theorem of Interpolation Theory*, a real bandlimited analog signal which has a bandwidth B and has been sampled can be perfectly reconstructed from an infinite sequence of samples if the sampling rate exceeds $2B$ samples per second without any inter-symbol interference:

Nyquist sampling theorem. *If a function $x(t)$ contains no frequencies higher than B hertz, it is completely determined by giving its ordinates at a series of points spaced $1/(2B)$ seconds apart.*

The $2B$ samples are then independent and they are carried on $2B$ signal dimensions [dim] per second.

2.2.3 Optimal word decoding

As mentioned previously, the decoder tries to find the codeword \hat{c} which is the most probable to have been sent over the channel, based on the channel output y and on the knowledge of the code:

$$\hat{c} = \arg \max_{c' \in \mathcal{C}} \Pr(c = c' | y) \quad (2.14)$$

This is the *word maximum a posteriori*² (W-MAP) decoder. If the a priori probabilities $\Pr(c)$ are identical, therefore the source is equally probable, this can be expressed as:

$$\hat{c} = \arg \max_{c' \in \mathcal{C}} \Pr(y|c = c') \quad (2.15)$$

This is named *word maximum likelihood* (W-ML) decoding. The W-MAP and W-ML decoders are two equivalent and optimal decoders if the source is equally probable.

The only way to achieve an optimal W-MAP decoder is by testing each codeword (2^k for a binary source).

2.2.4 Optimal symbol decoding

Similarly to above, but if the symbol (or bit) error rate is concerned, the *bit maximum a posteriori* (B-MAP) decoder and the *bit maximum likelihood* (B-ML) decoders give an estimation of the codeword symbols c_n :

$$\hat{c}_n = \arg \max_{c' \in \mathcal{A}_C} \Pr(c_n = c' | y) \quad (2.16)$$

$$\hat{c}_n = \arg \max_{c' \in \mathcal{A}_C} \Pr(y | c_n = c') \quad (2.17)$$

2.3 Linear block codes decoding

2.3.1 Binary block codes

Let u be a k -bit information sequence and v be the corresponding n -bit codeword. Then, a total of 2^k n -bit codewords constitute a (n, k) code. A *linear* code of length n and rank k is a linear subspace \mathcal{C} with dimension k of the vector space \mathbb{F}_q^n , where \mathbb{F}_q is the finite field (or *Galois field*)³ with q elements. Such a code with parameter q is called a *q-ary code*, e.g., when $q = 5$, the code is a 5-ary code. If $q = 2$ the code is described as a *binary code*, while if $q = 3$ it is called a *ternary code*.

The code \mathcal{C} can be defined by the list of all the codewords as:

$$\mathcal{C} = \{c^{(i)}, i \in \{0, \dots, 2^k - 1\}\} \quad (2.18)$$

This representation is, of course, unique. For example, a $(6, 3)$ code is $\mathcal{C} = \{000000, 100110, 010101, 001011, 110011, 101101, 011110, 111000\}$.

Alternatively, the code can also be defined by a vector base \mathcal{B}_C of k independent codewords as $\{c^{(i)}, i \in \{0, \dots, k - 1\}\}$, which is not unique. The vector base \mathcal{B}_C , however, has some useful equivalent representations:

²According to Bayes' rule, the posterior probabilities $\Pr(c = c' | y)$ are expressed by:

$$\Pr(c | y) = \frac{p(y|c) \Pr(c)}{p(y)} = \frac{p(y|c) \Pr(c)}{\sum_{c \in \mathcal{C}} p(y|c) \Pr(c)}$$

³In abstract algebra, a finite field is a field that contains only finitely many elements. It is also named Galois field in honor of Évariste Galois [3].

Generator matrix: If we arrange the k codewords into a $k \times n$ matrix G , whose rows are the vectors of the base \mathcal{B}_C , G is called a *generator matrix* for \mathcal{C} : let $u = (u_0, u_1, \dots, u_{k-1})$, where $u_i \in \mathbb{F}_q$. Then $\mathcal{C} = (c_0, c_1, \dots, c_{n-1}) = uG$. The rows of G are linearly independent, since G is assumed to have rank k .

Parity-check matrix: A (n, k) linear code can also be specified by a $(n - k) \times n$ matrix H with elements in \mathbb{F}_q as $\mathcal{C} = \{c^{(i)} / c^{(i)} \cdot H^T = 0\}$. H is called a *parity-check matrix*: each row of H is a parity-check equation on some bits of the codeword. Note that for any given matrix G , many solutions for H are possible.

Tanner graph: Code \mathcal{C} with parity matrix H can also be described by a bipartite graph⁴, or *Tanner graph* [4], with vertex set $V = V_1 \cup V_2$, which has one vertex in V_1 for each row of H and one vertex in V_2 for each column of H , and there is an edge between two vertices i and j exactly when $h_{ij} \neq 0$. Therefore, the elements of V_1 are the variable nodes, denoted by vn_m , and the elements of V_2 are the check nodes, denoted by cn_m . Each variable node vn_m is associated with one code symbol c_n and each check node cn_m is associated with the m -th parity-check constraint (row) of H . The Tanner graph representation of error correcting codes is useful in explaining their decoding algorithms by the exchange of information along the edges of these graphs.

Figure 2.2 displays an example of the different ways of describing a $(6, 3)$ linear code in \mathbb{F}_2 .

Short cycles of Tanner graphs have negative impact on decoding. Cycles necessarily have even length, however length 2 is not possible. The requirement that a Tanner graph should not have short cycles is an intricate part in the construction of efficient LDPC codes. Note that, however, the degrading effect of short-length cycles diminishes as the code length increases and is strongly reduced with length greater than 1.000 bits.

2.3.2 Decoding of binary block codes

Using equation 2.16 for binary block codes, so that the symbols will equivalently be named bits, we get that:

$$\hat{c}_n = \begin{cases} 0, & \text{if } \Pr(c_n = 0|y) > \Pr(c_n = 1|y) \\ 1, & \text{if } \Pr(c_n = 0|y) < \Pr(c_n = 1|y) \end{cases} \quad (2.19)$$

The decoder receives the word $y = (y_1, \dots, y_N)$ which can be split into two sets, y_n and $y_{n' \neq n}$. Then, as c_n, y_n are independent of $y_{n' \neq n}$, the above probabilities can be

⁴In the mathematical field of graph theory, a *bipartite graph*, or *bigraph*, is a graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in the first set to one in the second. Such graph, equivalently, does not contain any odd-length cycles.

(a)

$$\mathcal{C} = \{000000, 011001, 110010, 101011, \\ 111100, 100101, 001110, 010111\}$$

(b)

$$G = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{array} \right]$$

(c)

$$H = \left[\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{array} \right]$$

(d)

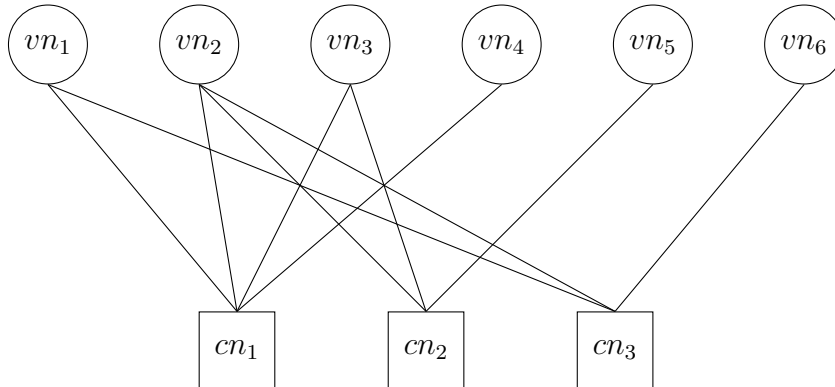


Figure 2.2: Example of a $(6, 3)$ linear code in \mathbb{F}_2 : (a) the code \mathcal{C} can be defined by the list of all the codewords; (b) generator matrix G can be obtained as $G = [I_k | P]$, where I_k is the $k \times k$ identity matrix and P is a $k \times (n - k)$ matrix; (c) the parity-check matrix H can be put into the form $[-P^T | I_{n-k}]$ (noting that in this special case of being a binary code $P = -P$); (d) the Tanner graph can be designed based on the parity check H .

expressed as:

$$\begin{aligned}
 \Pr(c_n|y) &= \Pr(c_n|y_n, y_{n' \neq n}) \\
 &= \frac{p(c_n, y_n, y_{n' \neq n})}{p(y_n, y_{n' \neq n})} \\
 &= \frac{p(y_n|c_n, y_{n' \neq n}) p(c_n|y_{n' \neq n})}{p(y_n|y_{n' \neq n}) p(y_{n' \neq n})} \\
 &= \frac{p(y_n|c_n) \Pr(c_n|y_{n' \neq n})}{p(y_n|y_{n' \neq n})} \tag{2.20}
 \end{aligned}$$

In addition, from equation 2.19 derives that:

$$\hat{c}_n = 0 \Rightarrow \frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)} > 1 \Rightarrow \log \frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)} > 0 \tag{2.21}$$

$$\hat{c}_n = 1 \Rightarrow \frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)} < 1 \Rightarrow \log \frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)} < 0 \tag{2.22}$$

Using the above formulas it is obvious that an equivalent way to decide for optimal bit error rate (BER) for binary block codes is to calculate the sign of:

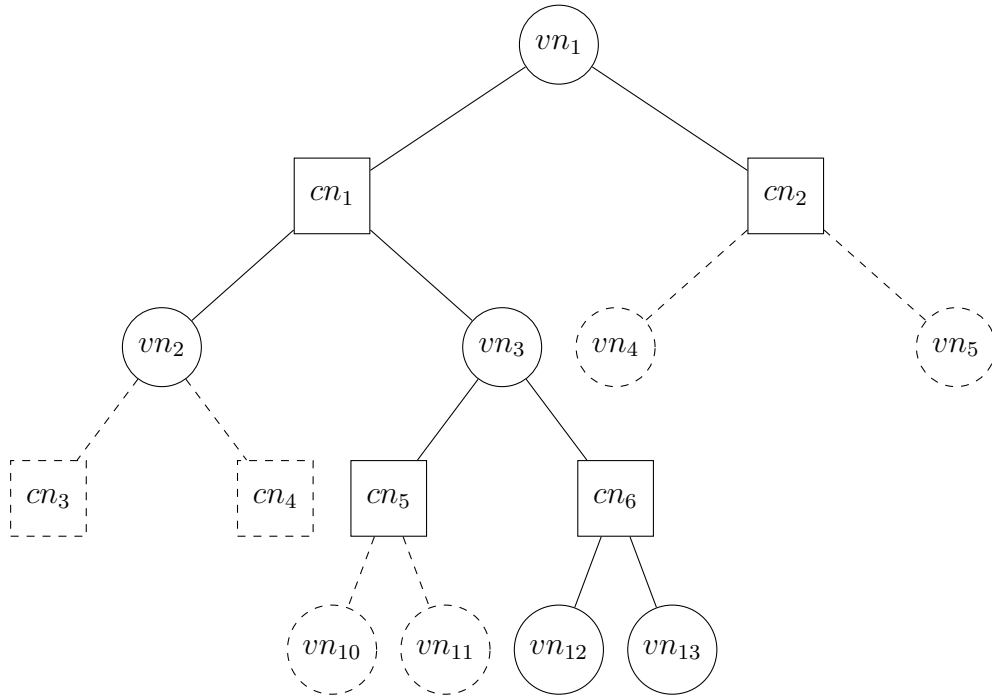
$$\log \frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)} \tag{2.23}$$

which, using equation 2.20 gives:

$$\underbrace{\log \frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)}}_{T_n} = \underbrace{\log \frac{p(y_n|c_n = 0)}{p(y_n|c_n = 1)}}_{I_n} + \underbrace{\log \frac{\Pr(c_n = 0|y_{n' \neq n})}{\Pr(c_n = 1|y_{n' \neq n})}}_{E_n} \tag{2.24}$$

where:

- T_n is the *overall information* of the bit n , and it is related to the two a posteriori probabilities on the bit n . The sign of T_n enables estimation of c_n and the magnitude of T_n , $|T_n|$, is the reliability of the decision.
- I_n is the *intrinsic information* of the bit n , and it is related to the received symbol y_n and to the channel parameters c_n .
- E_n is the *extrinsic information* of the bit n . It shows the improvement of information gained when the coded symbols respect the parity-check constraints. However, this improvement does not necessarily mean an increase of the reliability expressed by $|T_n|$.

Figure 2.3: An example of a cycle free bipartite graph of a code \mathcal{C}

2.3.3 Belief propagation

Optimal decoding of error correcting codes is feasible using a simple iterative message passing algorithm called *belief propagation*⁵. An important hypothesis related to the Tanner graph representation is required, however, which is known as the cycle-free graph hypothesis:

Cycle-free graph hypothesis. *The bipartite graph (or Tanner graph) of the code \mathcal{C} is cycle-free. A graph is cycle-free if it contains no path which begins and ends at the same bit node without going backwards. When the graph is not cycle-free, the minimum cycle length is called the girth of the graph.*

This hypothesis ensures that the bipartite graph of the code has a tree representation, so that each variable node v_n and each check code c_n appear exactly once in the tree. Figure 2.3 shows an example of a cycle free bipartite graph of a code \mathcal{C} .

Let $T_{n,m}$ be the information which is sent by a variable node vn_n to its connected check node cn_m . Then:

$$T_{n,m} = T_n - E_{n,m} \quad (2.25)$$

where $E_{n,m}$ is the information given by each of the parity-check constraints on the bit c_n . The partial results $T_{n,m}$ and $E_{n,m}$ are called *messages*, since they are transmitted from nodes to nodes.

In the example of Figure 2.3 the total information T_1 of the bit (variable) node vn_1 is calculated in 4 steps, as many as the depth between the leaves of the tree and the

⁵The belief propagation algorithm was first proposed by Judea Pearl in 1982 [5].

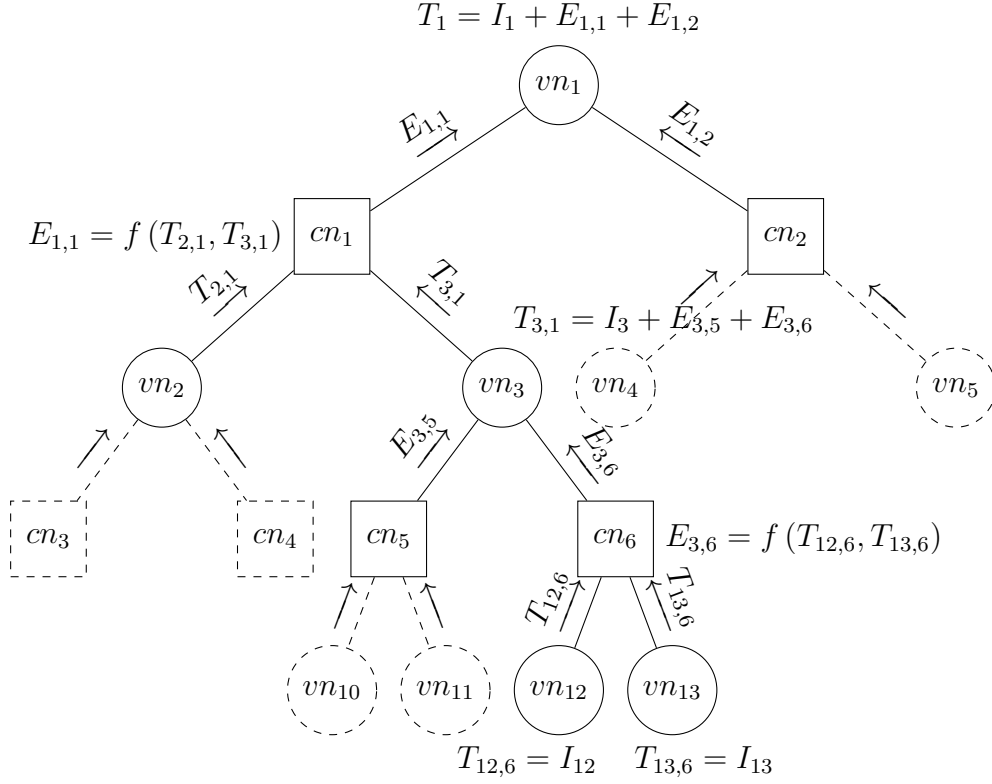


Figure 2.4: The operations of the graph of Figure 2.3 for calculating T_1

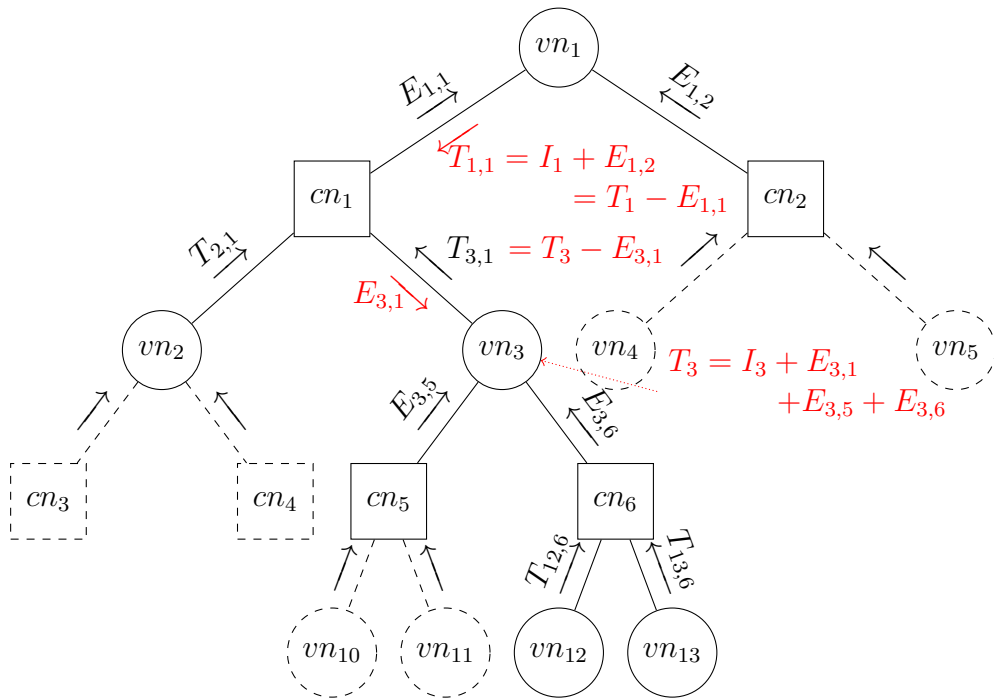
variable node in consideration. During the calculation process, some partial results from previous calculations are reused, while the others are not replaced by different results, but by messages in the opposite way. Therefore, the calculation of all T_n can be processed in parallel as the cycle free hypothesis lets them be all independent.

Figure 2.4 shows the operations of the graph of Figure 2.3 for calculating T_1 and Figure 2.5 for T_3 .

During calculation, the general behavior of each node is to process all the messages all the time: when one or more incoming messages on the variable node vn_n (respectively check node cn_m) has changed, the variable node (resp. check node) processes all the possible outgoing messages, in a process called *check* (resp. *variable*) *node update*. However, the nodes do not have to process conditionally to the processing of other nodes, letting them behave like independent processors.

The process is repeated until the total information T_n of each bit is computed. Notice that each node performs a repetition of an iteration, while each processor performs an iteration. The scheduling of the different processors does not affect the convergence of the algorithm.

Low-density parity-check (LDPC) codes are a class of block codes which can be decoded with the belief propagation algorithm, as described in the next chapter.

Figure 2.5: The operations of the graph of Figure 2.3 for calculating T_3

Chapter 3

Low-Density Parity-Check (LDPC) codes

Overview:

The third chapter introduces the LDPC codes, first by briefly reviewing their history and followed by defining their function. Then, the two possible ways of constructing LDPC codes are described and afterwards different ways of encoding of LDPC codes are proposed. The chapter is concluded by describing different ways of iterative decoding of LDPC codes and illustrating their performance.

3.1 History

In 1962, Robert G. Gallager [6] developed an iterative decoding algorithm which he applied to a new class of codes. Gallager named these codes *Low-Density Parity-Check* (LDPC) codes¹, because the parity-check matrices they used had to be sparse in order to perform well. Since then, however, LDPC codes had been ignored because they were impractical to implement at the time, requiring unavailable high complexity computation.

Another class of capacity-approaching codes were discovered by C. Berrou et al. in 1993. These *turbo codes* had remarkable performance, making them the coding scheme of choice of the time, used for applications such as deep space satellite communications, which also raised the interest toward iterative techniques.

LDPC codes were discovered again after a long time in 1995 when D. J. MacKay and R. M. Neal set up a link between the iterative algorithm used in LDPC codes to Pearl's algorithm (Pearl 1988) from the artificial intelligence community (bayesian networks). The articles of MacKay and Neal have been the kick off of great work in the field of LDPC codes.

In 1996, M. Sipser and D. A. Spielman used the first decoding algorithm (algorithm A) of R. G. Gallager to decode expander codes. In 1998, McEliece et al. showed that

¹LDPC codes are also known as *Gallager codes* in honor of R. G. Gallager who developed this coding scheme in his doctoral dissertation at MIT.

turbo decoding of turbo codes is an instance of Pearl's belief propagation algorithm.

Meanwhile, graph representation has gradually become a standard representation of error correcting codes. Using the work of Tanner (Tanner 1981) and N. Wiberg et al. (Wiberg 1996), F. R. Kschischang in 1998 denoted by factor graphs a class of graphs associated with the sum-product algorithm, which aim at describing many different algorithms by the same formalism.

Mainly inspired by two major revolutions in the channel coding community, the graph-based code-description and the iterative decoding techniques, LDPC codes have recently been developed past turbo codes, and have now been adopted for several new digital communication standards due to their excellent error correction performance, freedom from patent protection and inherently parallel decoding algorithm. Such examples include the recent DVB-S2 standard for the satellite transmission of digital television and 10GBase-T Ethernet, which sends data at 10 gigabits per second over twisted-pair cable [7–9].

3.2 Description of LDPC codes

3.2.1 Definition

Gallager defined a (N, j, k) LDPC code as a block code of length N having a small fixed number j of ones in each column of the parity-check matrix H and a small fixed number k of ones in each row of H .

This class of codes is to be decoded by the iterative algorithm described in Chapter 2. As shown, the algorithm computes exact a posteriori probabilities, under the hypothesis that the Tanner graph of the code is cycle-free. However, LDPC codes generally do have cycles, and for this reason the sparseness of the parity-check matrix aims at reducing the number of cycles and at increasing the size of the cycles. In addition, as the length N of the code increases, the cycle-free hypothesis becomes more realistic. The iterative algorithm performs quite well on these graphs, although not optimally.

3.2.2 Classes of LDPC codes

A code is called *regular* if every column and every row of its parity-check matrix has a fixed number of ones, j and k respectively. This means that each bit is implied in j parity-check constraints and each parity-check constraint is the exclusive-or (XOR) of k bits. Gallager's original LDPC code design was a regular LDPC code.

On the contrary, codes whose parity-check matrix does not have a constant number of non-zero entries in their rows or in their columns are called *irregular*. Codes of this type are specified by a the distribution degree of the bit $\lambda(x)$ and of the parity constraints $\rho(x)$:

$$\lambda(x) = \sum_{i=2}^{d_v} \lambda_i x^{i-1} \quad (3.1)$$

Notation	Description
\mathcal{L}_q	LDPC code in \mathbb{F}_q (e.g. \mathcal{L}_2 are binary LDPC codes)
$\mathcal{L}_q(M, N)$	LDPC code in \mathbb{F}_q of length N and rate $1 - M/N$
$\mathcal{L}_q(N, \lambda, \rho)$	LDPC code in \mathbb{F}_q of length N and degree distribution defined by $\lambda(x)$ and $\rho(x)$
$\mathcal{L}_q(N, j, k)$	<i>regular</i> LDPC code in \mathbb{F}_q of length N with $\lambda(x) = x^{j-1}$ and $\rho(x) = x^{k-1}$

Table 3.1: Different classes of LDPC codes

$$\rho(x) = \sum_{i=2}^{d_c} \rho_i x^{i-1} \quad (3.2)$$

where λ_i denotes the proportion of non-zero entries of the parity-check matrix H which belongs to the columns of H of weight i . Similarly, ρ_i denotes the proportion of non-zero entries of H which belongs to the rows of H of weight i . By definition it is $\lambda(1) = \rho(1) = 1$.

Table 3.1 lists some classes of the LDPC codes. Gallager's original LDPC code used in [6] is a regular $(N = 20, j = 3, k = 4)$ LDPC code, which is in the class $\mathcal{L}_2(20, 3, 4) = \mathcal{L}_2(20, x^2, x^3)$.

3.2.3 Code rate

The code rate R of LDPC codes is defined by $R \geq R_d \triangleq 1 - \frac{M}{N}$ where R_d is the *design code rate*. It is $R = R_d$ if the parity-check matrix has full rank. It is also shown that as N increases the parity-check matrix is almost sure to be full rank (Miller and Cohen 2003), therefore in this text it will be assumed that $R = R_d$ unless otherwise mentioned.

The rate R is linked to the other parameters of the class by:

$$R = 1 - \frac{\sum_i \rho_i / i}{\sum_i \lambda_i / i} = 1 - \frac{j}{k} \quad (3.3)$$

and in general, depending on j it is:

$$R = 1 - \frac{M}{N} \quad (3.4)$$

when j is odd and

$$R = 1 - \frac{M-1}{N} \quad (3.5)$$

when j is even.

3.3 Construction of LDPC codes

3.3.1 Random based construction

Construction of LDPC codes implies construction of a particular parity-check matrix H , whose design must meet the asymptotical constraints (e.g. the parameters of the code's class, the degree distribution or the rate) with the practical constraints (e.g. finite dimension, size of girths). During the design process there is a compromise that must be taken into consideration though: increasing the girth, the sparseness of H has to be decreased, effectively reducing the code performance due to low minimum distance. On the other hand, maximizing minimum distance requires that the sparseness be increased, consequently creating low-length girth, as the dimensions of H are finite, and thus reducing the convergence of the belief propagation algorithm.

One of the two possible techniques for constructing LDPC codes, and in fact the first chronologically, is the random based one. The constructions of Gallager in 1962 as well as MacKay and Neal in 1995 were random.

For the random based construction, the parity-check matrix is the concatenation and/or superposition of sub-matrices, which are created by processing some permutations on a particular sub-matrix. This sub-matrix may be random or not and usually has a column weight of 1.

The advantage of random constructions is that they do not have many constraints apart from the girth's value and they can fit quite well to the parameters of any given class. However, they do not guarantee that the girth will be small enough, so either post-processing or additional constraints are added, increasing the complexity of the design.

3.3.2 Deterministic based construction

The second construction technique, deterministic constructions, have been developed to deal with the girth problem, but their explicit design can lead to easier encoding and easier hardware implementation as well. There are two branches in combinatorial mathematics that are involved in such designs: finite geometry and Balanced Incomplete Block Designs (BIBDs), which seem to be more efficient than previous algebraic constructions based on expander graphs.

However, deterministic constructions lack the variety of combinations of parameters the random ones offer. Therefore, it may be hard to find a combination that fits the specifications of a given system.

3.4 Encoding

3.4.1 Lower-triangle shape based encoding

The encoding process of LDPC codes is actually their weak point, due to the fact that a sparse parity-check matrix does not necessarily have a sparse generator matrix,

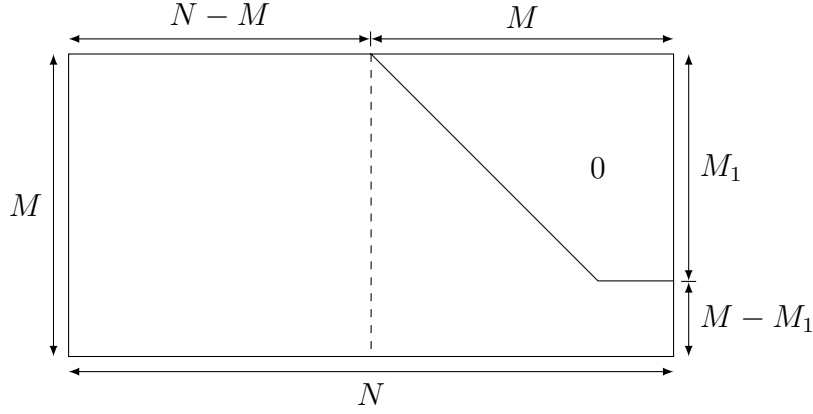


Figure 3.1: Shape of parity-check matrix for efficient encoding as proposed by (MacKay, Wilson and Davey 1999)

and in fact, it appears to be particularly dense.

One possible encoding scheme is to create a parity-check matrix with an almost lower-triangular shape as depicted in Figure 3.1. This was the approach of (MacKay, Wilson and Davey 1999). The lower-triangular constraint affects the performance of the encoding: instead of computing the product $c = uG^t$, the equation $H \cdot c^t = 0$ is solved, where c is the unknown variable.

The encoding is systematic:

$$\{c_1, \dots, c_{N-M}\} = \{u_1, \dots, u_{N-M}\} \quad (3.6)$$

The next M_1 c_i are recursively computed by using the lower-triangular shape of the parity-check matrix as:

$$c_i = -pc_i \times (c_1, \dots, c_{i-1})^t, \text{ for } i \in \{N-M+1, \dots, N-M+M_1\} \quad (3.7)$$

The last $M - M_1$ c_i , $i \in \{N-M+M_1, \dots, N\}$ have to be solved without reduced complexity, thus the higher M_1 is, the less complex the encoding will be.

Another approach was proposed by T. Richardson and R. Urbanke (Richardson and Urbanke 2001), which is depicted in Figure 3.2. The authors also propose some greedy algorithms which transform any given parity-check matrix H into an equivalent H' using columns and permutations, minimizing g depicted on the picture, so that H' will still be sparse. Then, complexity of the encoding will be $\mathcal{O}(N + g^2)$, where g is a small fraction of N .

3.4.2 Low-density generator matrices

As mentioned before, one of the problems of LDPC codes is that their generator matrices are usually not sparse, because of the inversion. An approach by (Oenning and Moon 2001) is to construct H both sparse and systematic, and then:

$$H = (P, I_M) \text{ and } G = (I_{N-M}, P^t) \quad (3.8)$$

where G is a sparse generator matrix (LDGM).

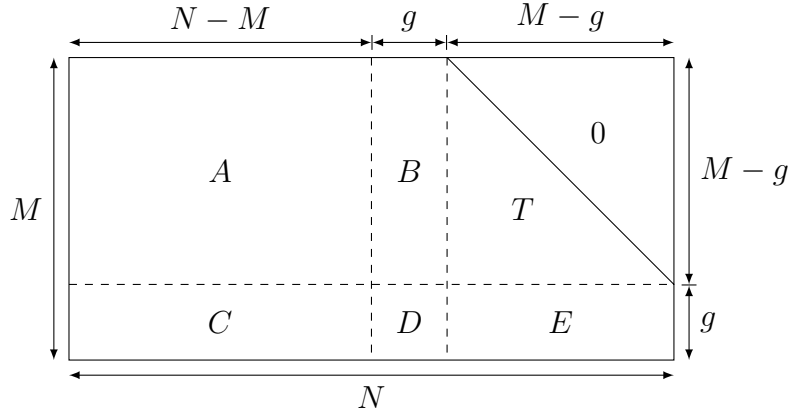


Figure 3.2: Proposal for efficient encoding of a parity-check matrix by (Richardson and Urbanke 2001)

3.4.3 Cyclic parity-check matrices

Cyclic or pseudo-cyclic codes are the most popular codes that can be easily encoded. A Gallager-like construction using cyclic shifts is proposed in (Okamura 2003), which enables having a cyclic-based encoder. LDPC codes constructed by finite geometry or BIBDs are also cyclic or pseudo-cyclic (Kou, Lin and Fossorier 2001; Ammar et al. 2002; Vasic 2002).

3.4.4 Iterative encoding

A class of parity-check codes which can be iteratively encoded using the same graph-based algorithm as the decoder was proposed in (Haley, Grant and Buetefuer 2002). However, the codes do not seem to perform as well for irregular cases as the random ones.

3.5 Decoding

3.5.1 Scheduling

Decoding of LDPC codes is processed by applying the optimal iterative decoding algorithm described in Chapter 2. If the graph of the code has cycles (the cycle-free hypothesis does not apply) then the optimality is lost. In such cases, the good performance achieved yields to use it as a good approximation. This algorithm is called the *belief propagation* (BP) algorithm.

The BP algorithm is scheduled, which means that the messages of the graph of the code are propagated in certain order. If the graph is cycle-free then the scheduling does not affect the convergence of the algorithm. Two schedules have been proposed for implementation purpose (Kschischang and Frey 1998):

Two way scheduling: A serial oriented schedule, in which only the relevant messages are processed and passed.

Flooding schedule: A parallel oriented schedule, in which all the nodes of the same type are processed and then the nodes of the other type are also all updated. The update for the nodes of a type can be either serial (one node at a time) or parallel without affecting the output messages.

For codes without cycle-free graph representations, the flooding schedule is used, since the behavior of each node processor is more simplified.

Mao and Banihashemi described a probabilistic schedule, with the idea of avoiding the auto-confirmation messages induced by the cycles of the graph. Their propagation would be avoided by sometimes not activating the node processors that should be in the flooding schedule (Mao and Banihashemi 2001):

Probabilistic schedule: Let g_n be the girth of the variable node vn_n . Also, let g_{\max} be the maximum size of girths g_n , $n \in \{1, \dots, N\}$. Then, the smallest number of iterations avoiding the auto-confirmation of information of the variable node vn_n on itself is $\frac{g_n}{2}$, since one iteration is a data path of 2 edges long. Therefore, each variable node vn_n should be updated only as long as iteration $i < \frac{g_n}{2}$, and afterwards it is idled. When more than $\frac{g_{\max}}{2}$ iterations have to be processed, the variable nodes are all updated at iterations $k \cdot \frac{g_{\max}}{2}$, where k is an integer. Then the same activation rule applies on vn_n by comparing $i \bmod \left(\frac{g_{\max}}{2}\right)$ to $\frac{g_n}{2}$.

Other scheduling techniques have been proposed as well. The authors of (Zhang and Fosserier 2002) proposed a shuffle BP algorithm which converges faster than the standard BP algorithm, by updating the information as soon as it has been computed, so that the next node processor to be updated could use a more up-to-date information. The authors of (Yeo, Nikolić and Anantharam 2001) proposed in a serialized architecture a staggered architecture which consist of processing serially the parity-check processors: information sent to the check node under process takes into account information of the previous iteration as well as information of the current iteration which has been updated by all the previous check nodes.

3.5.2 Performance

The error rate of iterative decoding algorithms has a typical curve, which is shown in Figure 3.3. There are three regions that can be distinguished on the solid line curve:

- the first region, below the convergence threshold, where the code is not very efficient. While in this region, even if the number of iterations is increased, the performance is not improved.
- the waterfall region, where the error rate has a huge negative slope which increases as the number of iterations increases.

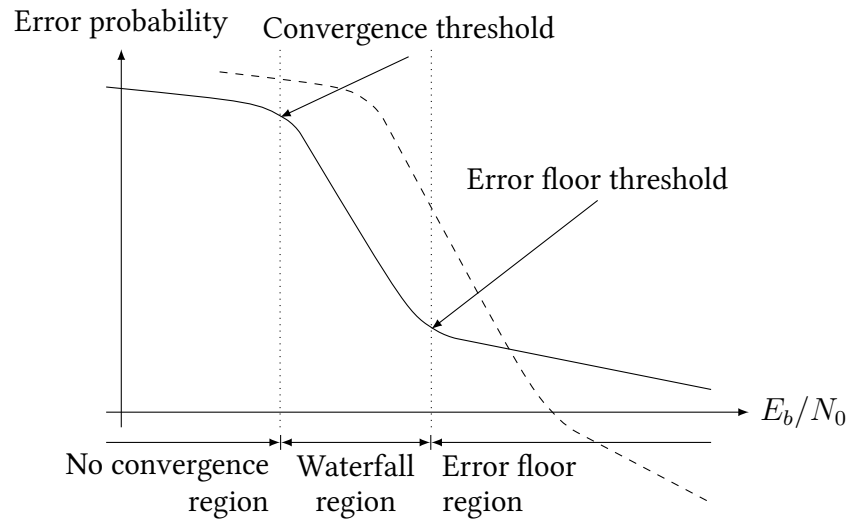


Figure 3.3: Typical error probability curve of iterative decoding algorithms. The dashed curve illustrates the trade-off between the waterfall and the error floor region, since the lower error floor comes at the expense of a higher convergence threshold.

- the error floor region, where the error rate slope is lower than the one of the waterfall region. The error floor is due to the minimum hamming distance of the code, and for LDPC codes it is also caused by near codewords (also called pseudo-codewords).

Chapter 4

Low-power and memory-efficient LDPC decoding

Overview:

This chapter investigates hardware architectures for LDPC decoders amenable to low-complexity implementation as well as low-voltage and low-power operation. First, a type of iterative message-passing decoding is described. Afterwards, the increased parallelism coupled with reduced supply voltage is proposed as an effective technique to reduce the power consumption of LDPC decoders, which have inherent parallelism. Therefore, a partially-parallel and fully-parallel decoder architecture is described. Then, a scheme to efficiently early terminate the iterative decoding, under certain conditions, is proposed to further reduce the power consumption. Finally, a quantization scheme is described, which can be used to increase the memory efficiency while decreasing the hardware complexity of the implementation, at the cost of storing slightly inaccurate information.

4.1 Min-sum decoding

A generic LDPC decoder architecture is shown in Figure 4.1. It comprises K_v shared variable nodes units (VNUs), K_c shared check nodes units (CNUs) and a shared memory fabric used to communicate messages between the VNUs and CNUs. The outputs of VNUs fetched from the memory are the inputs to each CNU, whose outputs, after some computations, are written back into the extrinsic memory. Similarly, inputs to each VNU arrive from the channel and several CNUs via the memory, and after performing the message update, the outputs of the VNUs are also written back into the extrinsic memory for use by the CNUs in the next decoding iteration. The process continues with all CNUs and VNUs alternately performing their computations for a fixed number of iterations before the decoded bits are obtained from one final computation performed by the VNUs.

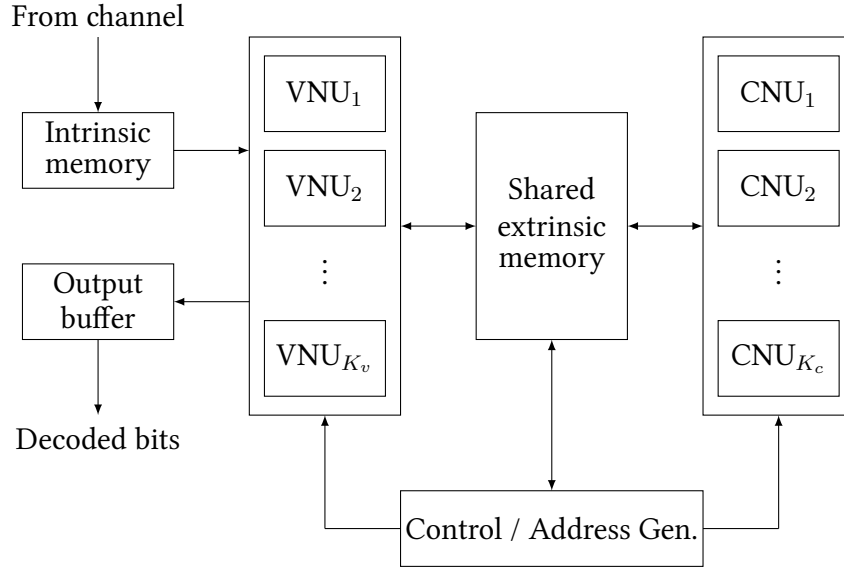


Figure 4.1: A partially-parallel LDPC decoder

The computing operations taking place in each iteration are part of the *min-sum decoding* [10] algorithm, which is a type of iterative message-passing decoding, also proposed as an approximation to the belief propagation (BP) algorithm. It is also referred to as the BP-based algorithm. The min-sum algorithm is a soft-decision, iterative algorithm for decoding binary-LDPC codes and is commonly used due to its simplicity and good BER performance.

During the process, each decoding iteration consists of updating and transferring extrinsic messages between neighboring variable nodes and check nodes. The messages state a belief about the value of the corresponding received bit expressed in the form of log likelihood ratio (LLR). At the beginning of the decoding process, the variable nodes pass the LLR value of the received symbols, i.e. the intrinsic message, to all the neighboring check nodes. Then, in each iteration, a check node update is followed by a variable node update. In the check node update phase the outgoing message on each edge of a check node has the sign of the parity of the signs of the incoming messages from all other edges and its magnitude calculated as the minimum of the magnitudes of the incoming messages. In the variable node update phase the outgoing message on each edge of a variable node is calculated as the sum of all the incoming messages from all other edges plus the intrinsic message from the channel.

4.2 Low power parallel decoders

4.2.1 Partially-parallel decoders

The decoder presented in Figure 4.1 will be able to perform more computations in parallel if the number of VNUs and CNUs is increased. With such increased

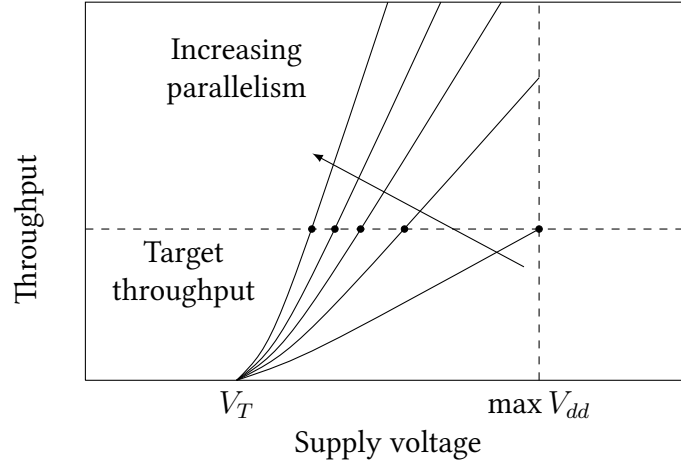


Figure 4.2: Supply voltage reduction obtainable by increased parallelism

parallelism, when operated from a fixed supply voltage, the decoder may achieve higher throughput, with attendant increases in power and area.

However, increased parallelism can also permit a system to operate from a lower supply voltage with constant throughput, therefore greatly decreasing power consumption [11]. In addition, the advantages offered by parallelism in power consumption are mitigated by the overhead associated with multiplexing and demultiplexing the inputs and outputs amongst several parallel computing units.

The iterative decoding of LDPC codes has inherent parallelism: all of the signals required for each iteration are already available in parallel in the extrinsic memory. Therefore, the iterative decoding of LDPC codes is well suited to implementation with a low supply voltage. The reduced supply voltage obtained by increasing parallelism is described qualitatively in Figure 4.2.

As shown in Figure 4.2, there is a practical limit to the power savings obtained by the decoder's parallelism when the number of VNUs and CNUs equal the total number of variable and check node computations required in each iteration. Further increases of K_v or K_c are not straightforwardly possible, because the required input messages are not available in the memory. The figure also shows that, unless the targeted throughput is low, the supply voltage will remain significantly higher than the MOS threshold voltage. However, sub-threshold circuits, although seemingly energy-efficient, they are mostly suitable for low-to-mid performance systems with relaxed constraints on throughput [12].

The benefits that high parallelism induces can be proved mathematically by comparing a reference decoder with K_v VNUs and K_c CNUs (decoder A) versus one with increased parallelism having $k \cdot K_v$ VNUs and $k \cdot K_c$ CNUs, where $k > 1$ (decoder B) [13].

Power consumption

The dynamic power consumption of both decoders, operated at a clock frequency f from a supply voltage V_{dd} is:

$$P = f C_{\text{eff}} V_{dd}^2 \quad (4.1)$$

where C_{eff} is the effective capacitance of each decoder including an activity factor. However, since the total number of messages stored in each iteration is constant, the memory size is the same for both decoders. Therefore, only the effective capacitance of the VNUs and CNUs are scaled by increasing parallelism.

Effective capacitance

Let β be the fraction of the reference decoder's (A) total effectiveness that scales with increasing parallelism, i.e. k . Also let C_M be the effective capacitance associated with the memory and C_A be the total effective capacitance of the reference design. Then:

$$\beta = \frac{C_A - C_M}{C_A} \quad (4.2)$$

Since C_M does not scale with k , the effective capacitance of the decoder with increased parallelism (B) is:

$$C_B = (1 + \beta (k - 1)) C_A \quad (4.3)$$

Supply voltage

Let f_A be the clock frequency the reference design operates and f_B the one of decoder B, that is k times lower than the reference's while maintaining the same throughput:

$$f_B = \frac{f_A}{k} \quad (4.4)$$

Since the aim is low-power operation, it can be supposed that each decoder operates from the lowest supply voltage possible that will support its targeted clock frequency. Hence, if V_{dd_A} is the supply voltage of the reference design, then the decoder with increased parallelism can be operated from a lower supply voltage V_{dd_B} , which is analyzed as [14]:

$$V_{dd_B} = u_{sc} V_{dd_A} \quad (4.5)$$

where:

$$u_{sc} = m + \frac{(1 - m)^2}{2k} + \sqrt{\left(m + \frac{(1 - m)^2}{2k}\right)^2 - m^2} \quad (4.6)$$

is the normalized voltage and

$$m = \frac{V_t}{V_{dd_A}} \quad (4.7)$$

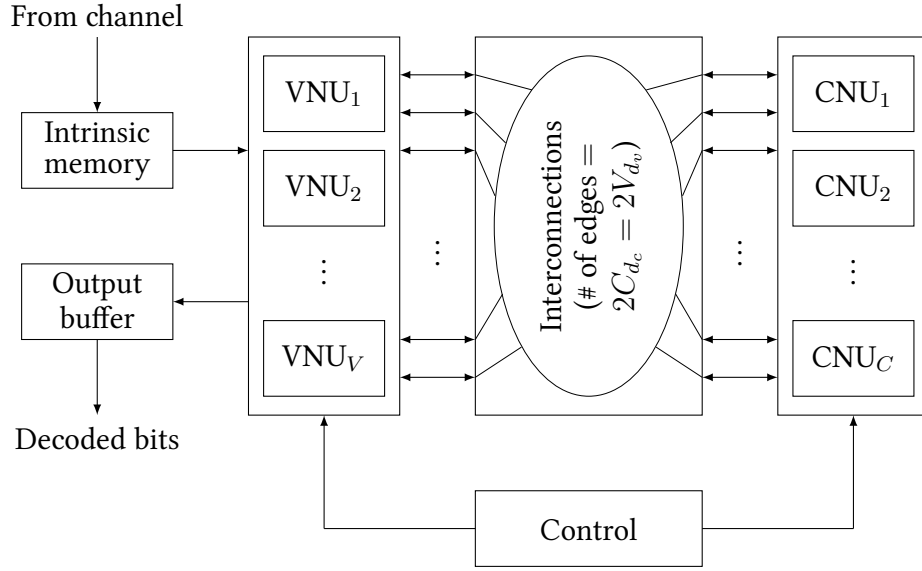


Figure 4.3: The fully-parallel iterative LDPC decoder architecture

Power savings

With the preceding analysis, the power savings offered by the decoder with increasing parallelism is:

$$P_B = \frac{u_{sc}^2}{k} (1 + \beta (k - 1)) P_A \equiv \eta P_A \quad (4.8)$$

4.2.2 Fully-parallel decoders

A fully-parallel architecture can be implemented by having a separate VNU or CNU designated for each variable node or check node in the Tanner graph of the code. A high-level fully-parallel iterative decoder architecture, based on a $(4, 15)$ -regular LDPC code with 660 variable nodes and 176 check nodes (V and C respectively) is shown in Figure 4.3. In this architecture, each extrinsic message is only written by one VNU or CNU, therefore the extrinsic memory can be distributed among VNUs and CNUs and no address generation is needed. Hence, the extrinsic memory block of Figure 4.1 has been replaced with the interconnections and Control/Address Generation block has been replaced with a simple Control block.

The advantage of the fully-parallel architecture is that it can be applied to irregular codes with no constraint on the code structure, unlike most partially-parallel decoders that are based on a particular code construction, such as the $(3, k)$ -regular construction in [15], or the Architecture-Aware code construction in [16]. The implementation of irregular codes is possible simply by instantiating the VNUs and CNUs of the desired degree and connecting them based on the graph of the code. However, the timing performance of the decoder for irregular codes will be typically limited by a critical path through the nodes with the highest degree.

In addition, the major challenge the implementation of high-parallel decoders has, is the large area and the overhead effects, such as the routing complexity [17]. In order to reduce the effect of routing complexity, a proposal to use a bit-serial message-passing scheme, where multi-bit messages are communicated between the nodes over multiple clock cycles has been made in [18]. The bit-serial message-passing also requires less logic to perform the min-sum decoding algorithm, since both the “min” and the “sum” operations are inherently bit-serial, and therefore bit-serial VNUs and CNUs can be efficiently implemented to generate only partial 1-bit extrinsic messages every clock cycle.

The use of bit-serial message-passing pushes the practical code length limit to higher values, and for this reason it is able to implement fully-parallel decoders for cutting-edge high-speed standards, such as 10GBase-T or Mobile WiMAX, which specify code lengths of 2048 and 2304 respectively. However, the maximum length of the LDPC codes that can be implemented in a bit-serial fully-parallel decoder will be eventually limited by the routing complexity.

4.3 Early termination

4.3.1 Description

The concept of *early termination* is to eliminate iterations of the decoding process which provide diminishing incremental improvements in decoder performance. The general decoder design implements a fixed, a-priori determined number of iterations, which are usually based on worst-case simulations. On the contrary, since most bit errors will have generally been corrected within the first few decoding iterations, the decoder may continue performing iterations even though it will usually converge to its final output much sooner.

The idea is to create a decoder which can automatically detect when it has converged to its final output and when it does, it will shut down all VNUs and CNUs for the remainder of each frame, saving power.

An approach in this area is to focus on identifying particular bits within each frame that appear likely to have converged, and then stop updating extrinsic messages for those reliable bits, while other unreliable bits are still being decoded [19, 20]. However, these bits are sometimes incorrectly identified, so the decoder’s performance suffers, unless an additional post-processing decoder is used in order to mitigate this performance degradation [21]. There is, however, an overhead associated with identifying the reliable bits and with the post-processing decoder that reduces the power savings of this approach.

Another approach, instead of trying to identify individual bits that appear to have converged early, is to monitor the entire frame to determine when the decoder has converged to a valid codeword, and then deactivate the entire decoder for the rest of the iterations in order to save power [13].

4.3.2 Implementation

A general approach to implement the detection the decoder's convergence to a correct codeword is the *syndrome checking*: it refers to making final decisions in each VNU at the end of each iteration and then checking if all parity constraints are satisfied. This approach, although straightforward, introduces a considerable hardware cost to the design, since it requires additional hardware to enable the VNUs to make the hard decisions at the end of each iteration, and it must also distribute those hard decisions to the destination check nodes in every iteration where syndrome checking can be performed.

The distribution of the VNUs' hard decisions to the check nodes can be done either by using additional hard wires from VNUs to the neighboring CNU's or by using the same wires that are used for transferring extrinsic messages in a bit-serial time multiplexed way. Both approaches, however, are inefficient because they either increase the routing complexity by adding extra global wires or decrease the decoding throughput by increasing the number of clock cycles per iteration, respectively.

An alternative approach is to check the parity of the normal variable-to-check messages that are already required by the decoding iterations: if the parity of all these messages are satisfied, the decoding for that frame can be stopped and the final hard decision can be computed at the beginning of the next iteration. This method, however, on average requires one extra iteration to terminate, compared to the conventional syndrome checking approach, but on the other hand, it does not increase the number of VNU-to-CNU wires, and it also does not require extra clock cycles per iteration to distribute the results of hard decisions to the CNU's. In addition, this approach induces less hardware overhead, since most of the calculations are already part of the normal VNU and CNU operations.

4.4 Quantization

4.4.1 Description

The standard belief propagation scheme uses real-valued messages which pass along edges in the code's graph and which are typically stored and updated in a very accurate representation, such as floating numbers. Computing and storing such an accurate representation, however, imposes high complexity, which, for high-speed LDPC decoders, should be avoided to reduce degradation of performance. For this reason, a low-complexity LDPC quantization scheme has been proposed to make efficient hardware implementation possible [22].

With quantization, the memory needed to store messages passing along edges in a code graph scales with the n -bit quantization as $\mathcal{O}(n)$. In addition, the number of interconnect wires to connect variable nodes and check nodes is also proportional to the n -bit quantization and the complexity of interconnect routing scales at least linearly with n . Therefore, a smaller n -bit quantization is generally a good idea as it makes the message update process easier for the variable and check nodes,

whose logic complexity are often more than linear with the n -bit quantization. In fact, the worst case imposes an n -bit-input n -bit-output look-up table to have logic complexity $\mathcal{O}(2^n)$.

Memory efficiency is highly considered when implementing LDPC decoders on FPGAs¹, even though newer generations of FPGA chips, such as Xilinx Virtex-II and Virtex-4, are equipped with high-capacity on-board memory for storage-demanding applications of signal processing (DSP). The memory block division of these devices imposes a practical constraint that also needs to be taken into consideration: the block memory is only divisible into fixed bit length, such as 4-bit wide, or high-resolution, such as 9-bit, 18-bit and 36-bit [25, 26], hence, to efficiently utilize the available memory, the n -bit quantization scheme applied should be compatible to the block memory division. The trade-off in this case is that higher bit length provides good resolution at the cost of limiting the size of the code the device can implement, as well as significantly increasing the amounts of power required to consume. On the contrary, an efficient low bit length quantization scheme can allow decoding of larger codes, and it is also very attractive if it can achieve small quantization loss.

4.4.2 Function

The decoding process that takes place is a quantized belief propagation algorithm, a message passing rule similar to the default algorithm described in previous chapters, but with the difference that the messages representing the likelihood ratios are compressed by each variable or check node before being transmitted to the adjacent nodes. The operation of each variable node occurs in the log-likelihood ratio (LLR) domain, or “*reliability*” domain. On the contrary, for check nodes, the domain is called “*unreliability*” domain. By domain, we mean the environment where updates can be performed through simple additions and subtractions, and where the values are typically represented by more bits than are required to transmit and store internode messages.

Two functions, Q_v and Q_c , are utilized to quantize the messages in the reliability and unreliability domains respectively into n -bit compressed messages. Complementary to these functions are also ϕ_v and ϕ_c , whose purpose is to restore the n -bit compressed messages into the reliability and unreliability domains for variable and check nodes respectively. A message that is compressed from one domain (e.g. the reliability domain) can always be restored into the other domain (the unreliability domain in this example) and vice-versa, since variable nodes always send messages to check nodes and vice-versa.

The process begins with a channel quantizer, Q_{ch} , that captures and quantizes the channel’s information, takes real-valued log-likelihood ratios, and finally produces a quantized representation. A reconstruction function, ϕ_{ch} , is implemented to take a message produced by the channel quantizer and output a value to be used by the variable node. Then, the process goes as follows:

¹Proposals for implementation of LDPC code decoders on Xilinx FPGA devices can be found in [23, 24].

- At each iteration, the variable node produces the variable-to-check messages $v_{i \rightarrow j}$, which for the first iteration (0) are given by:

$$v_{i \rightarrow j}(0) = Q_{ch}(\text{channel}_i), \quad i \in \{1, \dots, n\} \quad (4.9)$$

- At each next iteration, let it be the k^{th} iteration, the parity check phase occurs first. All CNUs read the variable-to-check messages $v_{i \rightarrow j}$ from some edge memory connecting the i^{th} variable node to the j^{th} check node in the code graph, and update the message by:

$$u_{j \rightarrow i}(k) = Q_c \left(\sum_{i' \neq i} \phi_c(v_{i' \rightarrow j}(k-1)) \right), \quad j \in \{1, \dots, r\} \quad (4.10)$$

where i' ranges over all edges connected to the j^{th} check node excluding i , Q_c is the quantization rule for the check-to-variable message $u_{j \rightarrow i}$ and ϕ_c is the reconstruction function for the variable-to-check message $v_{i \rightarrow j}$. Finally, they write the resulting check-to-variable messages, $u_{j \rightarrow i}$, back to the edge memory according to the code graph connections.

- Afterwards, the variable phase occurs, when n VNUs read the check-to-variable messages $u_{j \rightarrow i}$ from edge memory and update the message by:

$$v_{i \rightarrow j}(k) = Q_v \left(\phi_{ch}(Q_{ch}(\text{channel}_i)) + \sum_{j' \neq j} \phi_v(u_{j' \rightarrow i}(k)) \right), \quad i \in \{1, \dots, n\} \quad (4.11)$$

where j' ranges over all edges connected to the j^{th} node excluding j , Q_v is the quantization rule for the variable-to-check message $v_{i \rightarrow j}$, ϕ_v is the reconstruction function for the check-to-variable message $u_{j \rightarrow i}$ and ϕ_{ch} is the reconstruction function for the channel message $Q_{ch}(\text{channel}_i)$. Finally, they write the resulting variable-to-check message $v_{i \rightarrow j}$ back to edge memory according to the code graph connections.

- At the final iteration, let it be the K^{th} iteration, the variable nodes make hard decisions X_i as:

$$X_i = \begin{cases} 0, & \text{if } \sum_j u_{j \rightarrow i}(K) \geq 0 \\ 1, & \text{if } \sum_j u_{j \rightarrow i}(K) < 0 \end{cases} \quad (4.12)$$

Chapter 5

Performance of BPSK-modulated LDPC codes

Overview:

This chapter presents a performance analysis of several standard LDPC codes decoding techniques. At first, the algorithm behind each technique is presented, and afterwards a computer-based simulation is suggested in order to describe the performance versus implementation cost trade-off imposed. The simulation extracts information about the bit error rate (BER) of each technique, offering a graphic comparison between them. In the end, the best performing decoding technique will be chosen to implement on a FPGA.

5.1 Decoding techniques for LDPC codes

LDPC codes can be decoded using iterative decoding algorithms, in order to improve the code's performance. These algorithms generally perform local calculations and pass the local results via messages. This step is typically repeated several times during the decoding process. Different proposals have been made for decoding messages, which include both hard-decision and soft-decision belief propagation or sum-product algorithm (SPA) techniques. For messages which are BPSK modulated under Additive White Gaussian Noise (AWGN) and fading channels, some such proposals will be discussed.

5.1.1 Hard-decision (bit-flip) decoders

With this technique, the bits of a binary message will be decoded as 1 in a variable node if the majority of the incoming bits (from the source and the check nodes connected to the variable node) is 1, otherwise it will be decoded as 0.

More specifically, the procedure begins with all variable nodes vn_i sending a message to their connected check nodes cn_j containing the bit they believe to be the

correct one for them. At this (first) stage, the only information a variable node has, is the corresponding received i^{th} bit of the source codeword c , let it be y_i .

Then, every check node cn_j calculates a response to every connected variable node vn_i . The response message contains the bit that the check node believes to be the correct one for this variable node, assuming that the other variable nodes connected to the check node are correct. Therefore, at this step, a check node looks at the message received from its connected variable nodes and calculates the bit that each other connected variable node should have in order to fulfill the parity check equation. Note that this might also be the point at which the decoding algorithm terminates. This will be the case if all check equations are fulfilled. Techniques, such as early termination, can be used in cases that the decoding algorithm contains a loop, in order to stop the process when a threshold for the amount of loops has been reached.

The final step requires the variable nodes to receive the messages from the check nodes and use this additional information to decide if their originally received bit is correct. A simple way to do this is to decide upon the majority of their incoming information. Specifically, each variable node in this step has the source (original) information concerning its bit, as well as the suggestions from the check nodes. Upon deciding, the variable nodes can send another message with their (hard) decision for the correct value to the check nodes.

This type of decoders are usually simple to implement, since they do not employ complicated probability or log-likelihood function. This, however, may result in inferior performance compared to other decoders for very low values of E_b/N_0 .

5.1.2 Soft-decision probability-domain SPA decoders

Contrary to hard-decision decoding, which is made without using the knowledge of the probability set, soft-decision decoding of LDPC codes is based on the concept of belief propagation with the decoding process basing its decisions on the value of the probability for each bit to be 0 or 1 (for binary messages). The underlying idea for updating node information, however, is the same as in hard-decision decoding.

In the first step of the decoding process all variable nodes vn_i send their variable-to-check messages $v_{i \rightarrow j}$ to their connected check nodes cn_j . Every variable-to-check message contains always the pair $v_{i \rightarrow j}(0)$ and $v_{i \rightarrow j}(1)$ which stands for the amount of belief that y_i is 0 or 1 respectively. Since no other information is available at this step, $v_{i \rightarrow j}(1) = P_i$ and $v_{i \rightarrow j}(0) = 1 - P_i$, where $P_i = \Pr(c_i = 1|y_i)$.

In the following step, the check nodes calculate their response check-to-variable

messages $u_{j \rightarrow i}$ as¹:

$$u_{j \rightarrow i}(0) = \frac{1}{2} + \frac{1}{2} \prod_{i' \neq i} (1 - 2v_{i' \rightarrow j}(1)) \quad (5.2)$$

and

$$u_{j \rightarrow i}(1) = 1 - u_{j \rightarrow i}(0) \quad (5.3)$$

In this way, the check nodes calculate the probability that there is an even number of 1s among the variable nodes except vn_i (as i' spans all nodes except i). This probability is equal to the probability $u_{j \rightarrow i}(0)$ that vn_i is a 0.

Afterwards, the variable nodes update their response variable-to-check messages $v_{i \rightarrow j}$ as:

$$v_{i \rightarrow j}(0) = K_{ij}(1 - P_i) \prod_{j' \neq j} u_{j' \rightarrow i}(0) \quad (5.4)$$

and

$$v_{i \rightarrow j}(1) = K_{ij}P_i \prod_{j' \neq j} u_{j' \rightarrow i}(1) \quad (5.5)$$

where the constants K_{ij} are chosen to ensure that $v_{i \rightarrow j}(0) + v_{i \rightarrow j}(1) = 1$.

In the final step, the variable nodes update their current estimation \hat{c}_i of their variable c_i by calculating the probabilities for 0 and 1 as:

$$Q_i(0) = K_i(1 - P_i) \prod_j u_{j \rightarrow i}(0) \quad (5.6)$$

and

$$Q_i(1) = K_iP_i \prod_j u_{j \rightarrow i}(1) \quad (5.7)$$

and then voting for the bigger one:

$$\hat{c}_i = \begin{cases} 0, & \text{if } Q_i(0) > Q_i(1) \\ 1, & \text{if } Q_i(0) < Q_i(1) \end{cases} \quad (5.8)$$

If the current estimated codeword fulfills the parity check equations the algorithm may terminate. Otherwise, the process must continue and termination may be ensured through a maximum number of iterations.

¹Equation 5.2 uses the result from Gallager that for a sequence of M independent binary digits a_i with a probability of p_i for $a_i = 1$, the probability that the whole sequence contains an even number of 1s is then:

$$\frac{1}{2} + \frac{1}{2} \prod_{i=1}^M (1 - 2p_i) \quad (5.1)$$

5.1.3 Log-domain SPA decoders

The probabilistic type of soft-decision decoders has better performance than hard-decision decoders, since they use probability function for deciding, even though there are still some performance issues, such as numerical stability problems, due to the many multiplications of probabilities. In addition, an important issue is that for large block lengths the results will come very close to zero. In order to prevent this, it is possible to change into the log-domain and doing additions instead of multiplications. The result is a more stable algorithm that even has performance advantages since additions are computationally less expensive.

Before presenting the algorithm, some notations are introduced. First, for a Binary Symmetric Channel (BSC) with error probability p_i , the *log-likelihood ratio* (LLR) in favor of a 1 bit is defined as:

$$L(c_i) \triangleq \log \frac{1 - p_i}{p_i} \quad (5.9)$$

for each variable c_i and:

$$L(v_{i \rightarrow j}) \triangleq \log \frac{v_{i \rightarrow j}(0)}{v_{i \rightarrow j}(1)} \quad (5.10)$$

for each variable-to-check message $v_{i \rightarrow j}$.

The most frequently involved computation in the process can be defined as:

$$\phi(x) = -\log \tanh\left(\frac{1}{2}x\right) = \log \frac{e^x + 1}{e^x - 1} \quad (5.11)$$

which has the property of:

$$\phi^{-1}(x) = \phi(x), \text{ for } x > 0 \quad (5.12)$$

Finally, the log-likelihood ratio for variable-to-check messages can be separated as:

$$L(v_{i \rightarrow j}) = \alpha_{ij} \beta_{ij} \quad (5.13)$$

where:

$$\alpha_{ij} = \text{sign}(L(v_{i \rightarrow j})) \quad (5.14)$$

and

$$\beta_{ij} = \text{abs}(L(v_{i \rightarrow j})) \quad (5.15)$$

In the first step of the log-domain decoding algorithm, all variable nodes send their variable-to-checks message to their connected check nodes as:

$$L(v_{i \rightarrow j}) = 2y_i / \sigma^2 \quad (5.16)$$

for an Additive White Gaussian Noise (AWGN) channel with noise standard deviation σ .

In the following step, the check nodes calculate their response check-to-variable as:

$$L(u_{j \rightarrow i}) = \prod_{i' \neq i} \alpha_{i'j} \cdot \phi \left[\sum_{i' \neq i} \phi(\beta_{i'j}) \right] \quad (5.17)$$

Then, the variable nodes update their response variable-to-check messages as:

$$L(v_{i \rightarrow j}) = L(c_i) + \sum_{j' \neq j} L(u_{j' \rightarrow i}) \quad (5.18)$$

In the final step of the iteration, the variable nodes update their current estimation \hat{c}_i of their variable c_i by calculating its log-likelihood ratio as:

$$L(Q_i) = L(c_i) + \sum_j L(u_{j \rightarrow i}) \quad (5.19)$$

and then deciding upon the sign of the LLR:

$$\hat{c}_i = \begin{cases} 0, & \text{if } L(Q_i) > 0 \\ 1, & \text{if } L(Q_i) < 0 \end{cases} \quad (5.20)$$

A modified version of log-domain SPA decoder can also be proposed: this type of decoding process replaces probabilities P_i with \min_i . For further simplification, log-likelihood function can be replaced with incoming signal waveform directly, hence simplified log-domain decoder does not need noise variance information.

5.2 Bit error rate analysis

The main challenge when implementing an LDPC decoder is to choose the most preferred decoding technique regarding the possibly specific type of messages the decoder will handle. For this reason, the techniques discussed in this section have been simulated on PC by a MATLAB script². The script includes all steps of LDPC transmissions and its operation flowchart can be seen in Figure 5.1. According to this, the script operates as follows:

1. First, it creates an LDPC matrix for a code, with specific parameters for the number of rows and columns, as well as an option for eliminating cycles of length four in the factor graph of the parity-check matrix and the distribution of the user-defined number of 1s in the columns and rows of the matrix.

²MATLAB (<http://www.mathworks.com/products/matlab>) is a numerical computing environment which allows matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages. The script used for simulation is based on Radford M. Neal's programs collection written in C programming language, which are available at <http://www.cs.toronto.edu/~radford/ftp/LDPC-2006-02-08>.

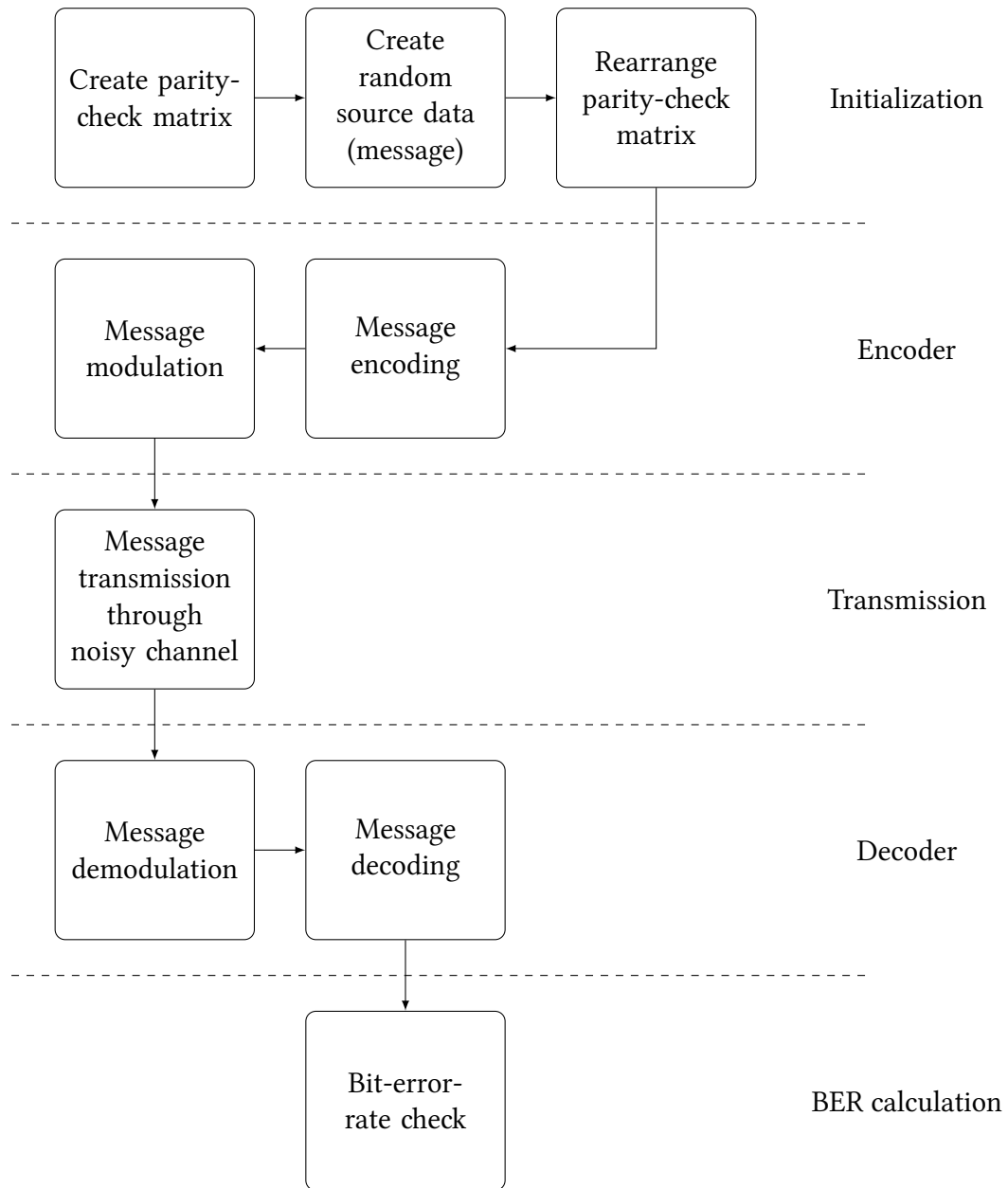


Figure 5.1: MATLAB simulation flowchart of an LDPC transceiver

2. Then, it creates messages of random source data, which will be the information messages to be transmitted.
3. The script will try to rearrange the parity-check matrix to a concatenation of two square matrices, A and B , with A being decomposed to LU , where L is a lower triangular and U is upper triangular matrix.
4. Afterwards, the encoding stage takes place, during which, the script generates parity check bits for the binary source and encodes the message blocks as codewords.
5. The message is subsequently modulated before transmission by the Binary phase-shift keying (BPSK) digital modulation scheme using a carrier wave of defined frequency.
6. Then, the bits of the message are transmitted through a noisy channel, which results to certain data at the output of the channel being related to the codeword sent with random noise. First, a channel model in which a linear addition of white noise with a constant spectral density and a Gaussian distribution of amplitude is simulated, which is known as Additive White Gaussian Noise (AWGN). After this set of simulations, another set of channels experiencing Rician and Rayleigh fading is demonstrated.
7. On the other side of the channel, the reverse process starts with demodulating the incoming message.
8. The received blocks are then decoded using one of the iterative belief propagation decoding techniques presented in the previous section. The decoding process follows the steps shown in Figure 5.2.
9. Finally, the script extracts information about the performance of each decoding technique, producing graphical presentations of the bit error rates (BER) depending on the number of iterations and E_b/N_0 using MATLAB's bit error rate analysis tool, `bertool`.

The results which are yielded by the simulation procedures of each decoding technique are discussed in their corresponding following subsections.

5.2.1 Performance analysis of hard-decision (bit-flip) decoders

Hard-decision (bit-flipping) sum-product algorithm LDPC decoders are simple to implement, since they do not employ complicated probability or log-likelihood function. In MATLAB, decoders of this type can be simulated using the code presented in Section A.1 of Appendix A. This code shows the operations which are done during the iterations of the decoding process.

The number of iterations of the decoding process, as well as the value of E_b/N_0 of the transmitted signal have strong impact on the performance of the decoder. The

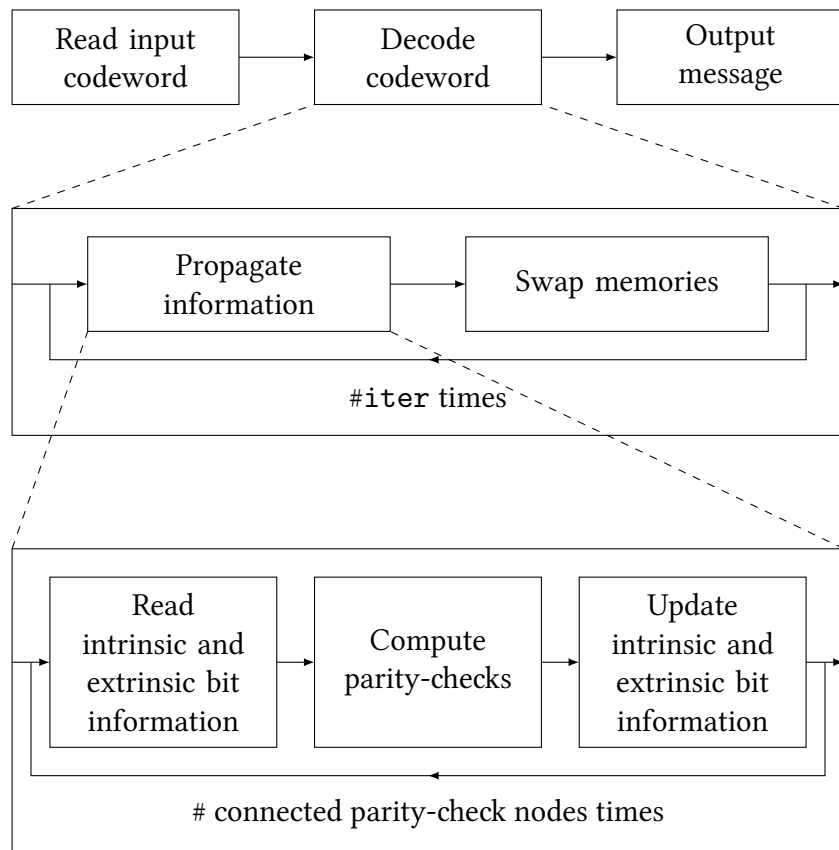


Figure 5.2: Message decoding with belief propagation

process is simulated in MATLAB using `bertool`, which runs the specified code for different values of E_b/N_0 (0 to 8dB) and number of iterations (1 to 7) using random messages as sources up to a total of 10^6 bits. For each number of iterations, the results of the simulation are presented in Figure 5.3.

The results clearly expose the decoder's weakness to decode correctly for very low values of E_b/N_0 . Regardless the number of iterations, for the lowest value of E_b/N_0 simulated, 0dB, the bit error rate rises over $3 \cdot 10^{-1}$, which means that almost one in three bits is erroneous. However, as E_b/N_0 values increase, the decoder's performance tends to improve gradually, with the bit error rate slightly bordering 10^{-2} for $E_b/N_0 = 8\text{dB}$. Therefore, hard-decision (bit-flipping) decoders cannot be trusted for reliable transmissions, especially when operating under noisy channels, maybe with the exception of situations where their very simple hardware implementation makes up for their weak performance.

5.2.2 Performance analysis of soft-decision probability-domain decoders

As opposed to the bit-flipping decoders, probability-domain sum-product algorithm decoders base their decisions on the value of the probability for each bit to be 0 or 1 in the case of binary messages. To accomplish this, they employ more complicated functions which make use of the probability set of the code, effectively increasing the implementation complexity. In return, probability-domain decoders are expected to perform better even for low values of E_b/N_0 . The simulation is operated by the code presented in Section A.2 of Appendix A and yields the results presented in Figure 5.4.

According to the graphical representations, the decoder's performance is by far superior to a bit-flipping one's for all values of E_b/N_0 and number of iterations. All executions start with bit error rate just around 10^{-1} for $E_b/N_0 = 0\text{dB}$, and this gradually falls as E_b/N_0 increases further, ultimately reaching values close to 10^{-6} when $E_b/N_0 = 8\text{dB}$.

As a result, in situations where performance is matters the most, probability-domain decoders will most likely satisfy the high needs of reliable communication. However, this comes at the cost of raised implementation complexity. To overcome this dilemma, different techniques have been proposed which offer a better trade-off between operation performance and implementation complexity.

5.2.3 Performance analysis of log-domain decoders

Log-domain decoders share the same main steps with the probability-domain decoders in the way they decide upon the bits of a received message, but differ in the way they calculate that decision. Instead of probability functions, this type of decoders use log-likelihood ratios (LLRs) as response messages between the check and variable nodes, a method which is computationally less intensive than the one of their counterparts, but on the other hand, may be less accurate thus producing more

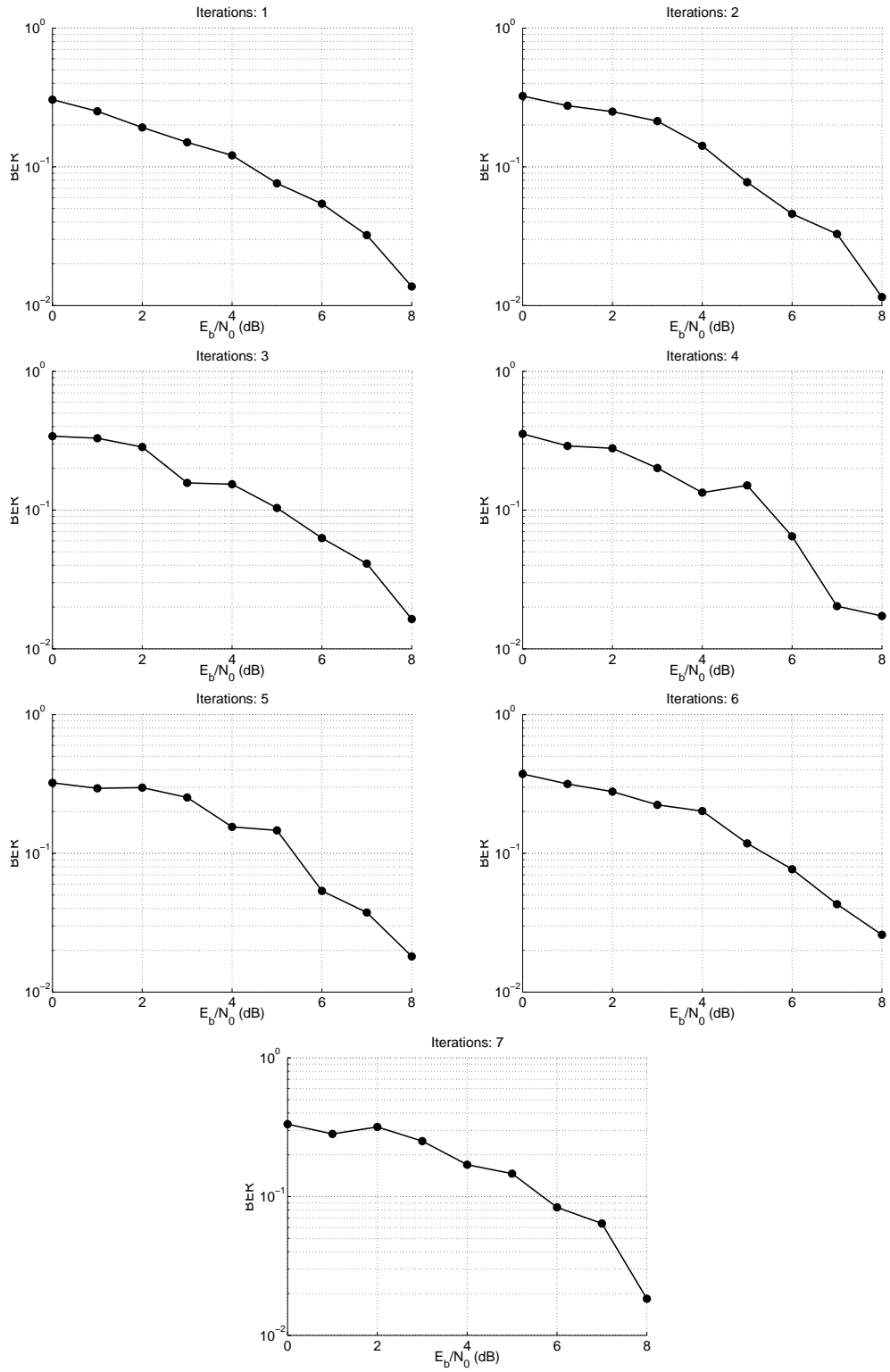


Figure 5.3: BER analysis of a hard-decision (bit-flip) decoder under AWGN channel

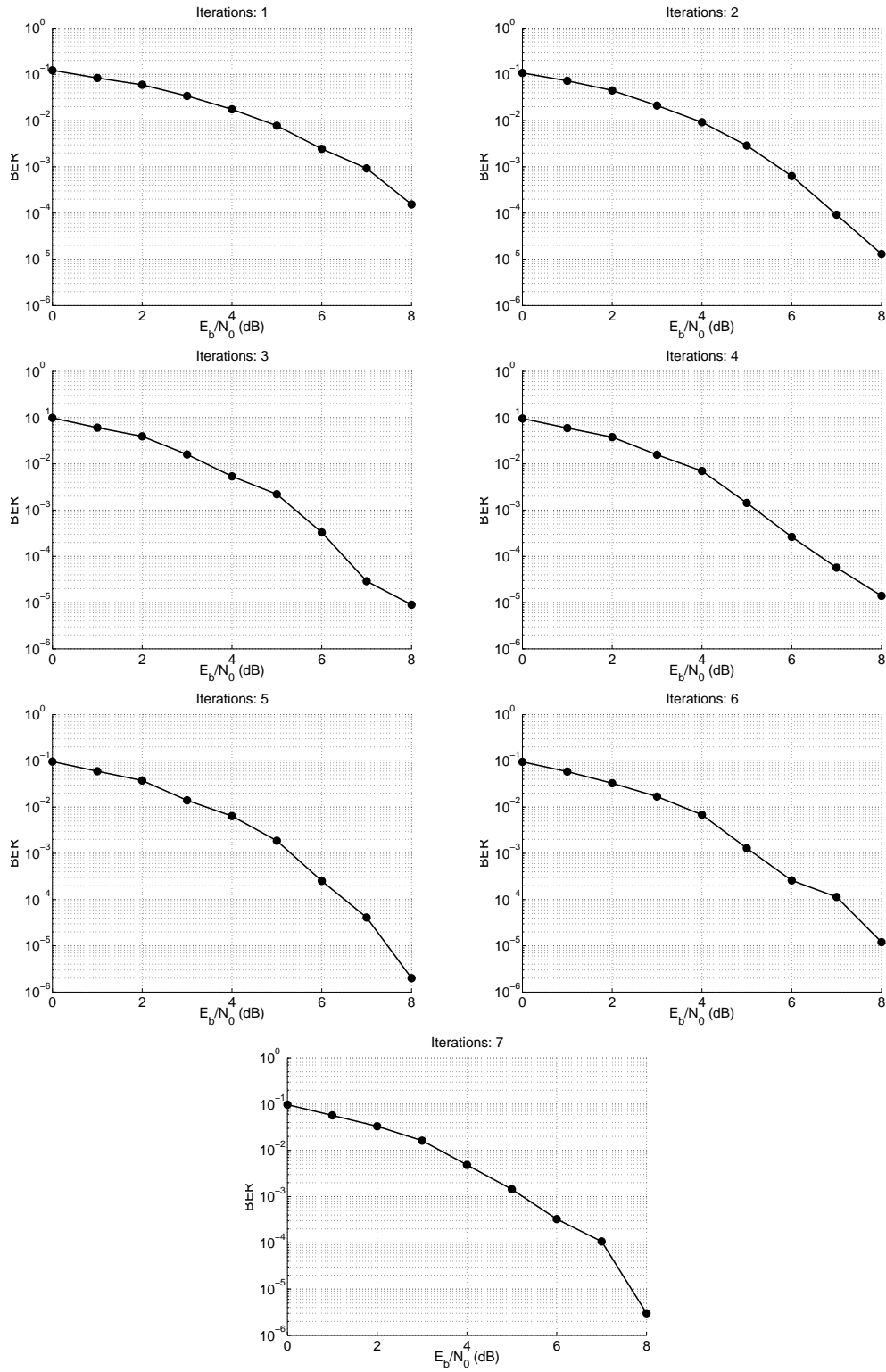


Figure 5.4: BER analysis of a probability-domain SPA decoder under AWGN channel

erroneous bits. A decoder of this type can be simulated using the code presented in Section A.3 of Appendix A and its performance is shown in Figure 5.5.

Like the probability-domain decoders, log-domain ones have superior performance compared to bit-flipping decoders for all values of E_b/N_0 and number of iterations. For lower values of E_b/N_0 , log-domain decoders also have similar performance curve to the probability-domain ones, as seen by the values of BER for $E_b/N_0 < 3$ dB. For higher values, performance increases steadily, reaching values in the range between 10^{-4} (for 1 and 2 iterations) and 10^{-6} (for 4 iterations), with most of them tending to end up around 10^{-5} for $E_b/N_0 = 8$ dB, which is a performance close to the performance of probability-domain decoders.

The number of iterations heavily affect the performance of this type of decoders, since with only 1 iteration the decoder cannot surpass the limit of 10^{-4} BER, which only 2 or more iterations can. On the other hand, a big number of iterations seem to also impair performance, instead of improve it. Best performance in terms of bit error rate is observed for 4 iterations, while more or less iterations gradually decrease it.

5.2.4 Performance analysis of simplified log-domain decoders

A modified version of the log-domain sum-product algorithm decoders can be created by replacing some of their heavy computations with simpler ones. Specifically, this type of decoders avoid calculating the logarithmic and exponential function $\phi(x)$, and instead they use the minimum value of the log-likelihood ratio of the exchanged messages. Therefore, this type of decoders are simpler to implement, but this comes at the cost of possibly lower performance than the original log-domain decoders. The code which simulates a simplified log-domain decoder is presented in Section A.4 of Appendix A and its performance analysis can be seen in Figure 5.6.

According to the plots, the performance curves of simplified log-domain decoders are similar to the ones of the original log-domain decoders. Both types are better than hard-decision decoders, and start with bit error rate around 10^{-1} , with values gradually decreasing as E_b/N_0 increases. For the highest value of E_b/N_0 which was simulated, 8 dB, bit error rate usually lies between 10^{-5} and 10^{-6} as the original log-domain decoders do.

The lowest value of BER is observed for 6 iterations, even though it is clear that the more E_b/N_0 increases, the less impact the number of iterations seem to have on the performance of the decoder. The worst performance is observed for only 1 iteration, but this case is common to all decoding techniques.

All things concerned, the simplified log-domain decoders perform relatively well, similar to the original log-domain decoders and close to the probability-domain ones, even though the performance is not exactly the same as their counterparts. However, their simpler implementation and computationally less expensive operation make up for this possible weakness, especially when they are needed to be implemented in small embedded systems. Therefore, choosing a simplified log-domain decoder to be implemented on a FPGA is a reasonable decision.

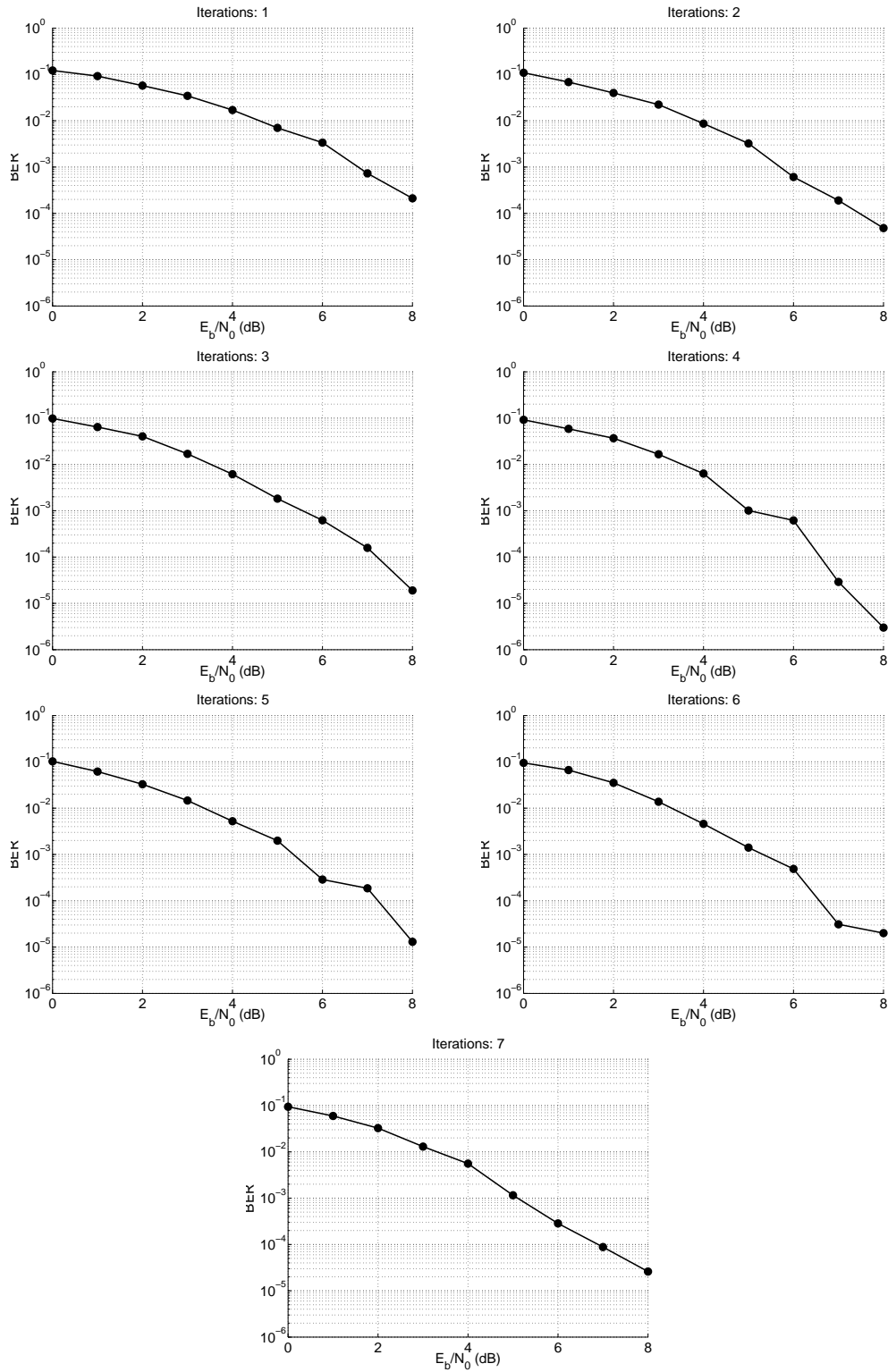


Figure 5.5: BER analysis of a log-domain SPA decoder under AWGN channel

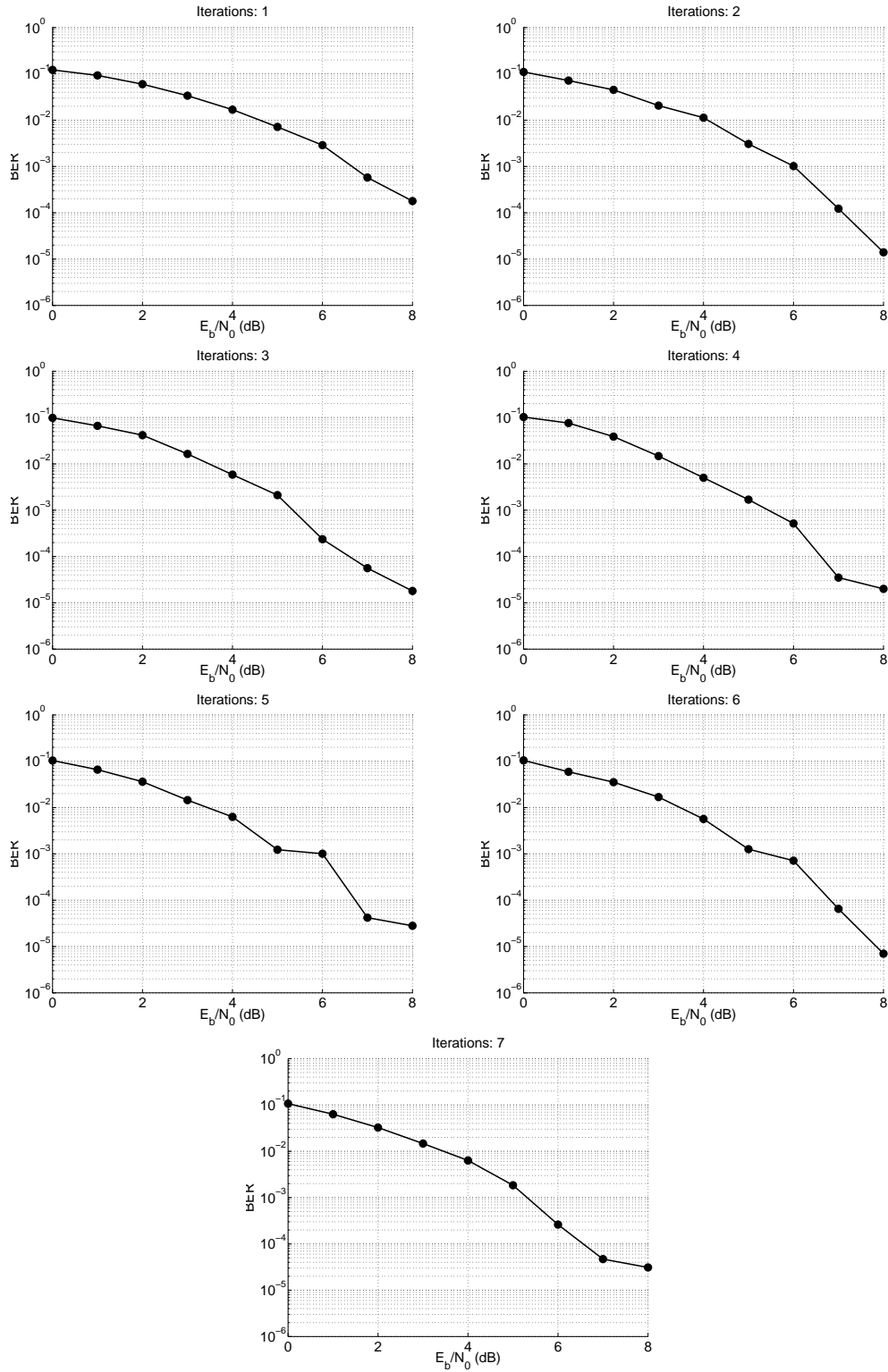


Figure 5.6: BER analysis of a simplified log-domain SPA decoder under AWGN channel

5.3 Fading channels

5.3.1 Rayleigh fading

Signal impairments in a wireless channel arise not only from noise and interference, which are common to most electromagnetic communication systems, but also from propagating effects, which are unique to wireless communications. The term “*fading*” is used to describe this cases; it denotes variation of received signal amplitude and phase, with respect to both time and distance. The fading results from interaction between the propagating wavefront, the mobile receiver, and nearby objects, and for this reason it is often modeled as a random process.

If the maximum spread of time delay (τ_{\max}) between multipath signal components is much less than the symbol period (T_s), then relative multipath delay (fading which is due to multipath propagation) can be ignored. In this case all multipath components may be regarded as one with respect to time and the condition is then called “flat” or non-frequency-selective fading. This happens because the channel coherence bandwidth (over which the channel is strongly auto-correlated) is wide (the curve is flat) compared with the signal bandwidth.

In environments where no line-of-sight (LOS) signal component is likely to reach the receiver, scattering of the wavefront by many nearby objects is expected. Therefore, the received signal is regarded as equally probable from any direction. This condition is termed *isotropic scattering*, and may be modeled using a Rayleigh distribution, wherein the random variable $R = \sqrt{X^2 + Y^2}$ is a function of two independent, zero-mean, normally distributed random (Gaussian) variables X and Y . The fading in such a system is termed “*Rayleigh fading*”. If there is a dominant line-of-sight, *Rician fading* may be more applicable. The model behind Rician fading is similar to that for Rayleigh fading, except that in Rician fading a strong dominant component is present, which is usually the line-of-sight wave.

In a flat Rayleigh fading model X and Y represent the in-phase and quadrature components of the channel impulse response. Both are random processes, and consequently the resultant envelope R is also a random process.

The Jakes sum-of-sinusoid method [27] was used with MATLAB to simulate flat Rayleigh fading, and statistical properties of the simulated fading channel were investigated. Figure 5.7 shows the envelope’s probability density function (pdf) compared to the theoretical Rayleigh distribution using variance $\sigma^2 = 1$.

Decoding performance under Rayleigh fading channel

Fading channels can cause poor performance in a communication system and can result in a loss of signal power which can be over some or all of the signal bandwidth. Fading can also be a problem because it changes over time and also varies with geographical position and radio frequency. Even though communication systems are often designed to adapt to such impairments, the fading can still change faster than the adaptations and, in these cases, the probability of experiencing a fade on the channel becomes the limiting factor in the link’s performance.

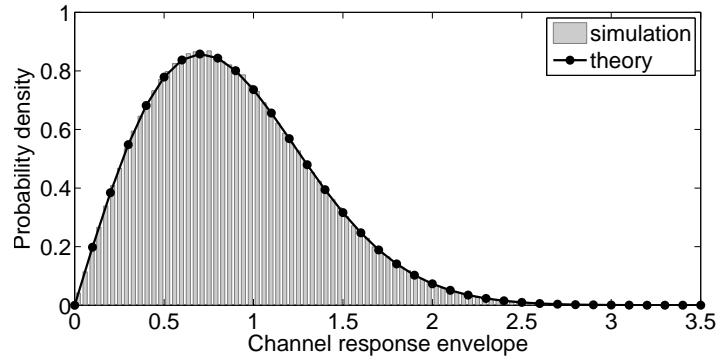


Figure 5.7: Density of Rayleigh flat fading envelope with $N = 10^6$ and $f_D = 100\text{Hz}$

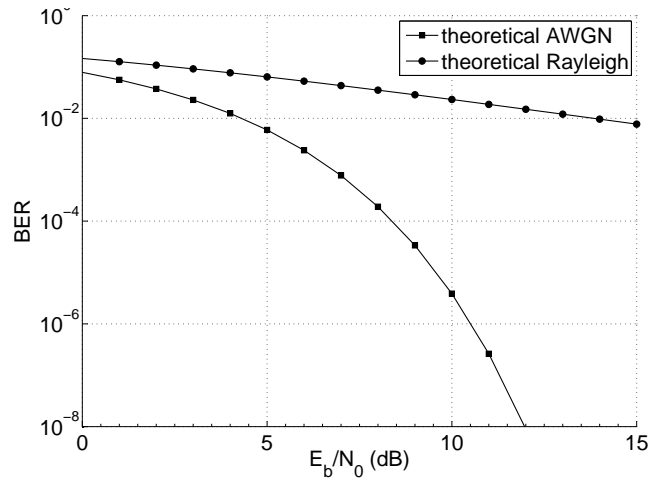


Figure 5.8: Performance of Rayleigh fading channels compared to AWGN channels

The impairments imposed by Rayleigh fading channels compared to those introduced by AWGN channels is presented in Figure 5.8. According to this representation, bit error rate in cases of Rayleigh channels barely falls under 10^{-2} when $E_b/N_0 = 15\text{dB}$. On the contrary, decoding performance in AWGN channels can perfectly reach 10^{-7} bit error rate at a value of $E_b/N_0 = 12\text{dB}$.

Hard-decision (bit-flip) decoders

Like in AWGN simulations, hard-decision decoders have shown to be the worst performing algorithm, which is the cost of their simple implementation. With bit error rates starting over $3 \cdot 10^{-1}$, which means that out of a total 10^6 bits, 30% are erroneous, this type of decoders most likely cannot be trusted for reliable communication in fading channels. This value gradually falls to almost $2 \cdot 10^{-2}$ when $E_b/N_0 = 15\text{dB}$. The performance curve produced in MATLAB is shown in Figure 5.9. According to the simulation results, increasing the number of iterations of the belief propagation algorithm does not improve performance, which may be due to the already big number of erroneous bits.

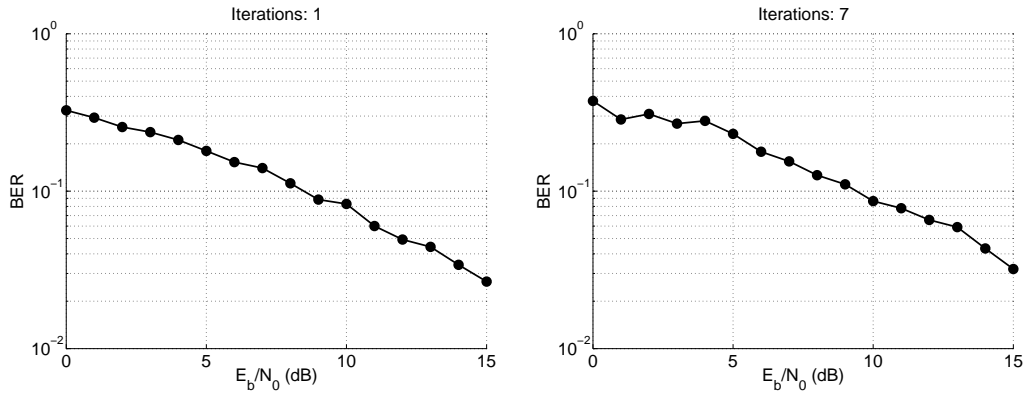


Figure 5.9: BER analysis of a hard-decision (bit-flip) decoder under Rayleigh channel

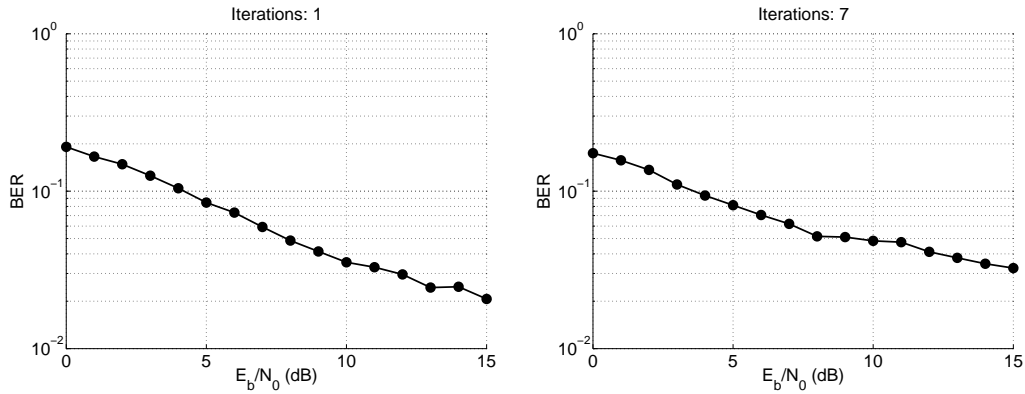


Figure 5.10: BER analysis of a probability-domain SPA decoder under Rayleigh channel

Soft-decision probability-domain decoders

Probability-domain decoders start with better bit error rates than bit-flip ones, with starting values just under $2 \cdot 10^{-1}$. These values continue to fall as E_b/N_0 increases, up to a point when further increases in E_b/N_0 do not seem to significantly improve performance. This is clearly seen in Figure 5.10, in which, for 7 iterations, when E_b/N_0 increases further than 8dB, bit error rate decrease is gradually lesser. Consequently, there is a threshold in the value of E_b/N_0 , after which probability-domain decoding performance is almost steady. This threshold is also found for 1 iteration of the decoding algorithm, but in this case it corresponds to a higher value of E_b/N_0 , around 13dB. In addition, as the value of E_b/N_0 increases, it seems that the impact the number of iterations have upon the decoding performance is gradually decreasing.

Log-domain decoders

Performance of log-domain decoders shares a similar start with probability-domain ones, since their bit error rate starts at a little less than $2 \cdot 10^{-1}$ at $E_b/N_0 =$

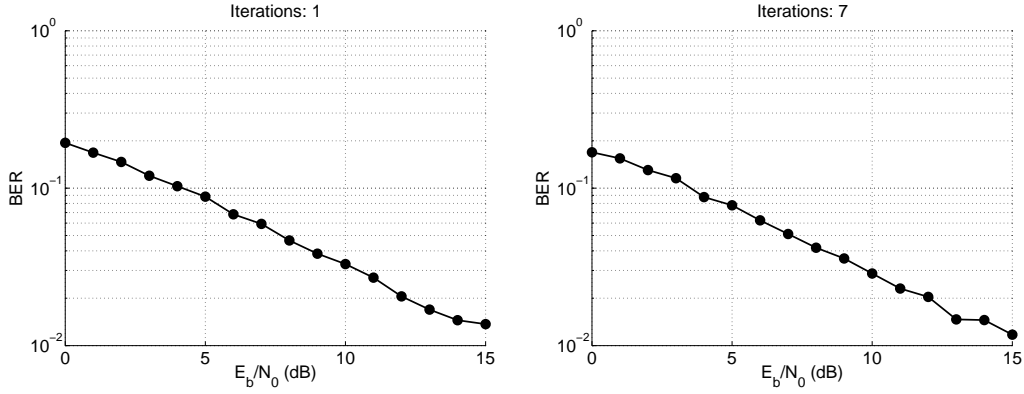


Figure 5.11: BER analysis of a log-domain SPA decoder under Rayleigh channel

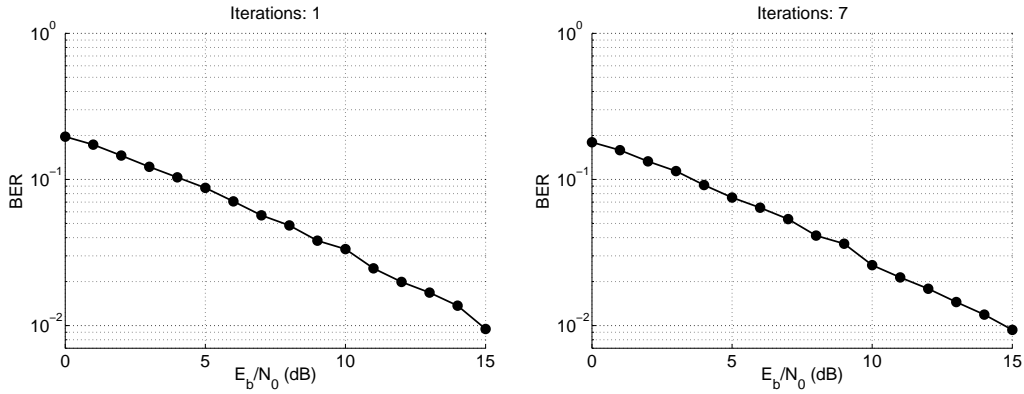


Figure 5.12: BER analysis of a simplified log-domain SPA decoder under Rayleigh channel

0dB and gradually falls as E_b/N_0 increases, showing a possible threshold point at $E_b/N_0 = 14$ dB, after which performance improve rate seems to decrease. The performance curve of these decoders is presented in Figure 5.11, and according to it, log-domain decoders can achieve bit error rate values which are very close to $2 \cdot 10^{-2}$ at higher values of E_b/N_0 .

Simplified log-domain decoders

The modified version of log-domain decoders presented in subsection 5.2.4 has a similar performance curve to its original counterparts as seen in Figure 5.12. With a starting value of BER close to $2 \cdot 10^{-1}$ for 1 iteration of the decoding algorithm (which falls to $1,8 \cdot 10^{-1}$ for 7 iterations), performance of this type of decoders constantly falls throughout the simulation runtime, without showing any thresholds of decreasing performance rate as probability-domain decoders do. At higher values of E_b/N_0 , simplified log-domain decoders decrease their bit error rate to values close or even less than 10^{-2} (when $E_b/N_0 = 15$ dB), which is very close to the values of the theoretical performance curve presented in Figure 5.8, especially as the number of iterations increases.

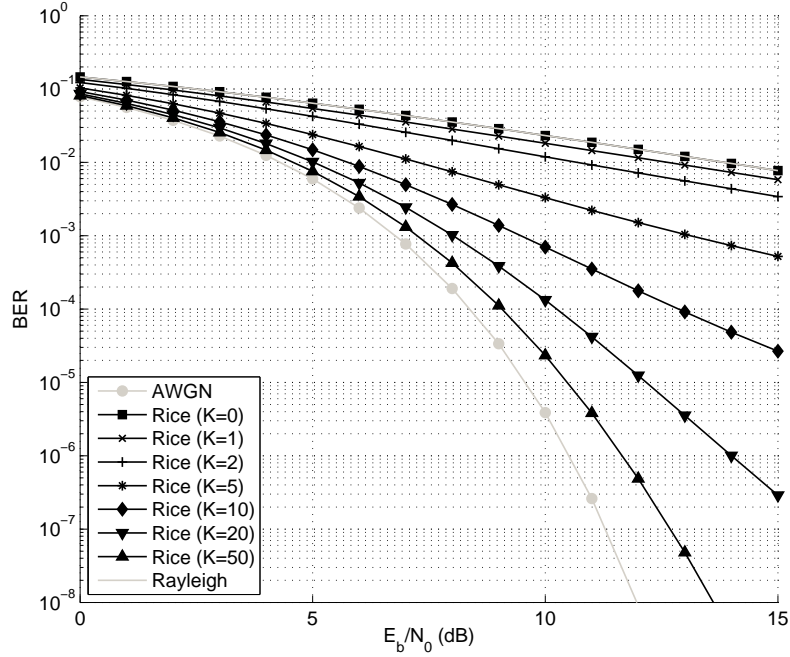


Figure 5.13: Performance of Rician fading channels compared to AWGN and Rayleigh channels for different values of K

5.3.2 Rician fading

The model behind Rician fading is similar to that for Rayleigh fading, except that in Rician fading a strong dominant component is present. This dominant component is typically the line-of-sight wave. On the contrary, in Rayleigh fading there is no line-of-sight signal, and for this reason it is sometimes considered as a special case of the more generalized concept of Rician fading.

The Rician K -factor is defined as the ratio of signal power in dominant component over the (local-mean) scattered power. In the expression for the received signal, the power in the line-of-sight equals $\frac{C^2}{2}$, where C is the amplitude of the line-of-sight component. For example, in indoor channels with an unobstructed line-of-sight between transmit and receive antenna the K -factor is usually between 4 and 12dB. Rayleigh fading is recovered for $K = 0$, which corresponds to $-\infty$ dB.

Decoding performance under Rician fading channel

Like all fading channels, Rician fading imposes degradation to the decoding performance of messages transmitted through such channel. The theoretical performance of Rician fading channels compared to AWGN and Rayleigh fading is depicted in Figure 5.13. According to this, performance of Rayleigh fading equals Rician when $K = 0$, since in this case the two channels are equivalent. As K increases, so does performance, since stronger dominant components favor more

error-free transmissions. In fact, when K gets a quite big value, like $K = 50$, the impairments imposed by fading can almost be absorbed by the Gaussian channel, and the performance curve in this case tends to follow the theoretical AWGN curve very closely. Therefore, performance of Rician fading channels is upper bounded by AWGN and lower bounded by Rayleigh channels.

Hard-decision (bit-flip) decoders

Mediocre performance has been a typical characteristic of hard-decision decoders, which is also proved in Rician channels. Bit error rates of bit-flip decoders are at higher levels than the rest of the decoding algorithms, mainly due to their simple implementation. According to their performance curves, which are presented in Figure 5.14, bit error rate lies between 10^{-1} and 10^{-2} for $K = 2$ and for the values of $E_b/N_0 \geq 6$ dB. As K increases, so does performance, which lays between 10^{-2} and 10^{-3} for $K = 5$ and between 10^{-3} and 10^{-4} for $K = 10$, when E_b/N_0 is greater than 10 dB. In addition, once again, increasing the number of iterations of the belief propagation algorithm does not seem to improve performance, which has also been observed in Rayleigh fading channel simulations.

Soft-decision probability-domain decoders

Probability-domain decoders have better performance compared to bit-flip decoders, due to the fact that they utilize more complex calculations during the decoding process. Bit error rates start with values around 10^{-1} in all cases which have been simulated. However, these values gradually fall as E_b/N_0 and K increase, reaching values close to 10^{-2} for $K = 2$, 10^{-3} for $K = 5$ and 10^{-4} for $K = 10$. In addition, there is a threshold in the values of E_b/N_0 , after which performance seem to improve with gradually lower rates, as shown in Figure 5.15. This behavior has been observed in Rayleigh fading channels as well, which is expected, since Rayleigh fading channels can be considered a special case of Rician channels. Once again, increasing the number of iterations of the decoding process does not seem to improve performance, which has also been observed in Rayleigh channels.

Log-domain decoders

Simulation of log-domain decoders under Rician fading channel shows the best results in performance compared to all other techniques. As shown in Figure 5.16, bit error rate for log-domain decoders starts around 10^{-1} at $E_b/N_0 = 0$ dB for all values of K , and it gradually falls as E_b/N_0 and K increase. For higher values of E_b/N_0 , bit error rate falls under 10^{-2} for $K = 2$, 10^{-3} for $K = 5$ and 10^{-4} for $K = 10$ respectively, which is very close to the theoretical performance curve depicted in Figure 5.13. Increasing the number of iterations seems to slightly improve performance.

Simplified log-domain decoders

The last decoding technique which has been simulated, the simplified log-domain decoders, perform quite similarly to the original log-domain ones. According to Figure 5.17, their performance curves are very close to their original counterparts, which are also very close to the theoretical performance curves. For higher values of E_b/N_0 , bit error rate has values between 10^{-2} and 10^{-3} for $K = 2$, between 10^{-3} and 10^{-4} for $K = 5$ and it can even approach 10^{-5} when $K = 10$. Similarly to log-domain decoders, their simplified version seems to improve in performance as the number of iterations increases.

Consequently, after observing the decoding performance of all types of decoders under both AWGN and fading channels, it is reasonable to choose the simplified form of log-domain decoders to implement on FPGA, since they match the good performance of log-domain decoders with the simpler implementation and computationally less expensive operation. The values of bit error rates for a sensible number of iterations, such as 7 iterations, under all simulated channels of transmission is presented in Figure 5.18. According to this, in order to achieve bit error rates lower than 10^{-2} , or 1%, a Gaussian channel requires messages with E_b/N_0 values greater than 3dB. On the other hand, fading channels require much more energy per bit to achieve these rates, since a Rician channel with K -factor of 7dB, i.e. 5 in linear scale, requires that E_b/N_0 is over 7dB, while under Rayleigh channel this value must be even greater, requiring normalized signal-to-noise ratios of over 15dB. However, in applications where reliability is not very critical, all channels can perform similarly well for very low values of E_b/N_0 , bounded between 0 – 3dB, even without the need of an increased number of iterations.

The implementation of the simplified log-domain decoder, along with the encoder of the LDPC transceiver, is described in the next chapter. In the end, the implemented system will be checked to verify that its performance meets the bit error rate values expressed in this chapter.

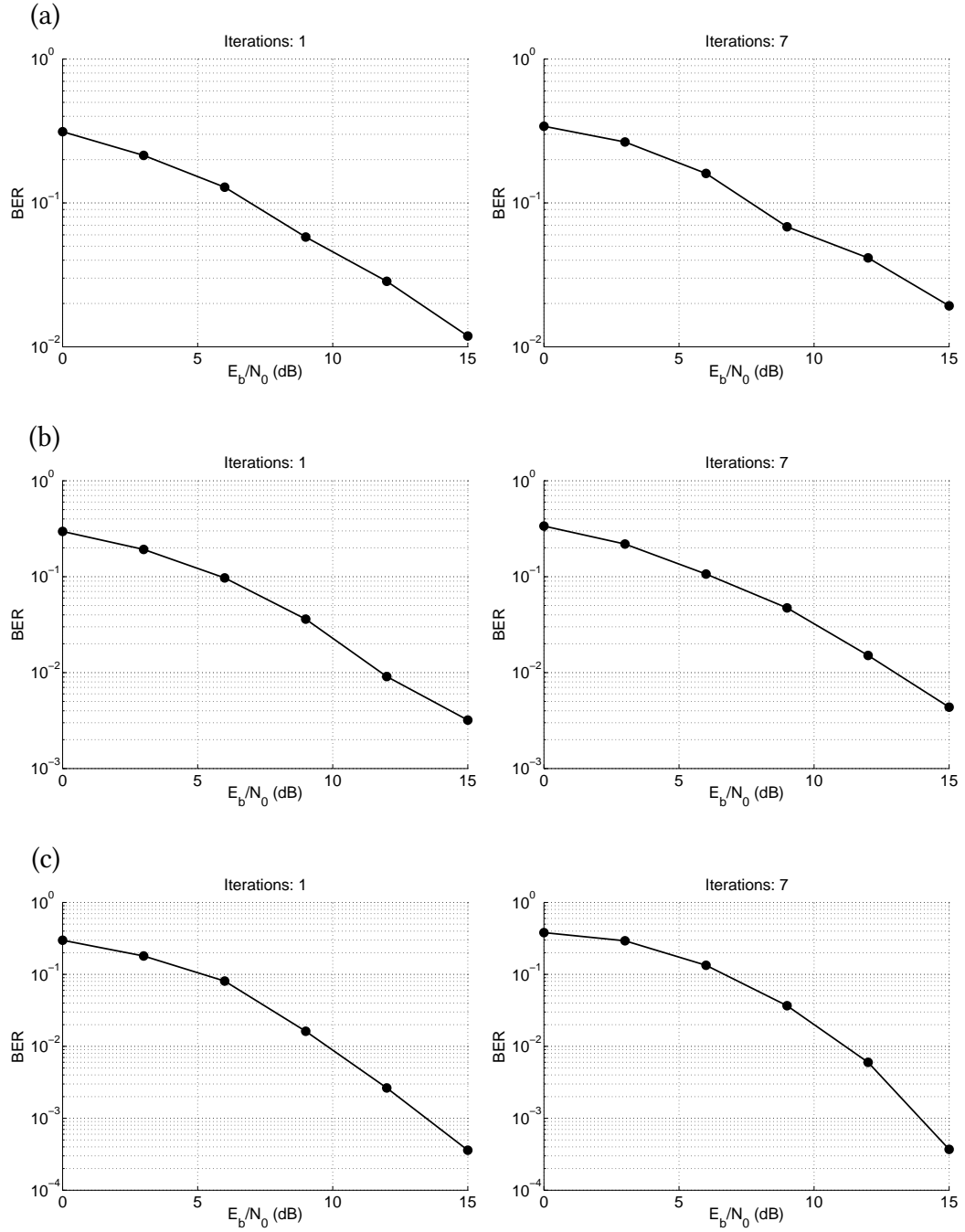


Figure 5.14: BER analysis of a hard-decision (bit-flip) decoder under Rician channel for: (a) $K = 2$, (b) $K = 5$ and (c) $K = 10$.

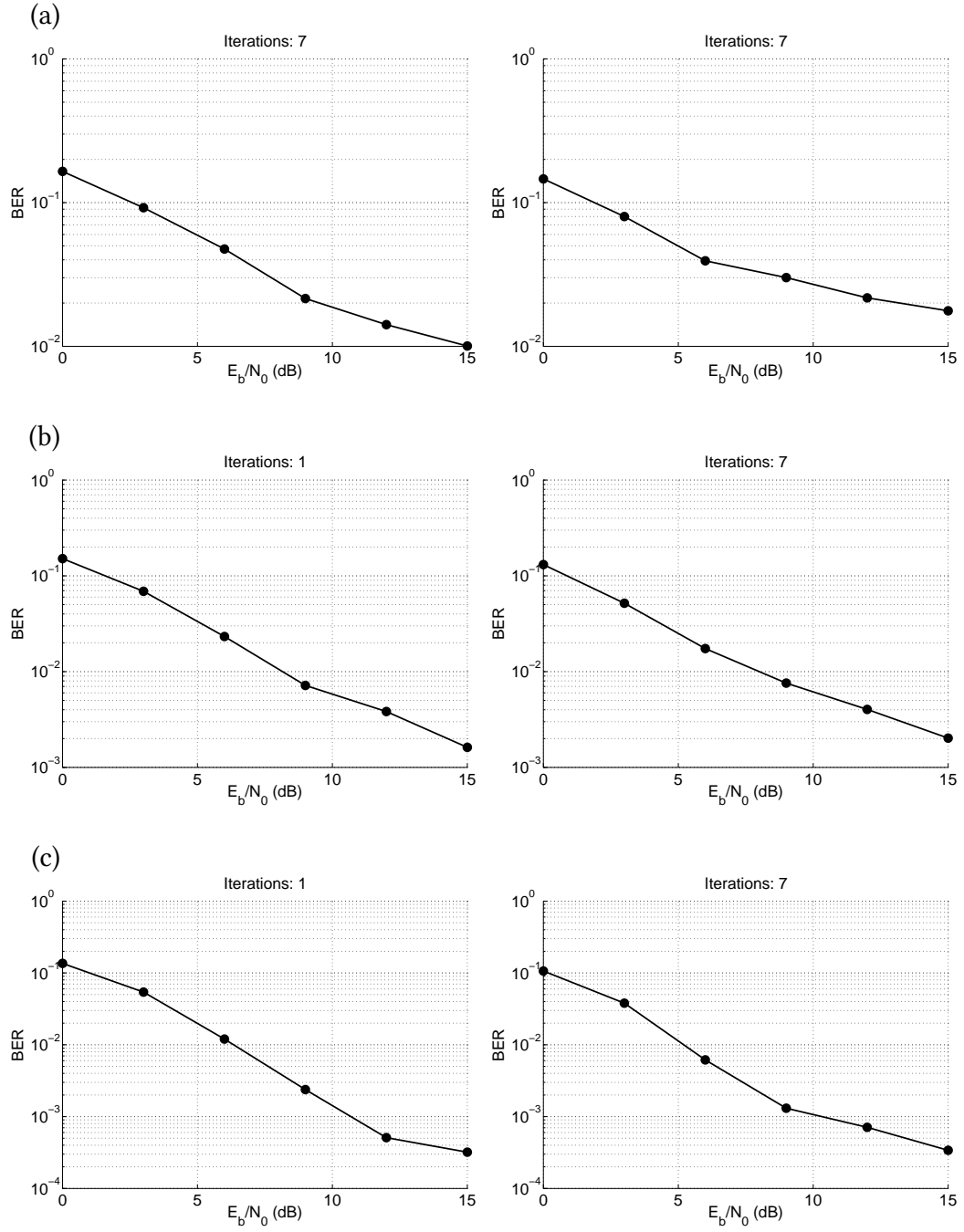


Figure 5.15: BER analysis of a probability-domain SPA decoder under Rician channel for: (a) $K = 2$, (b) $K = 5$ and (c) $K = 10$.

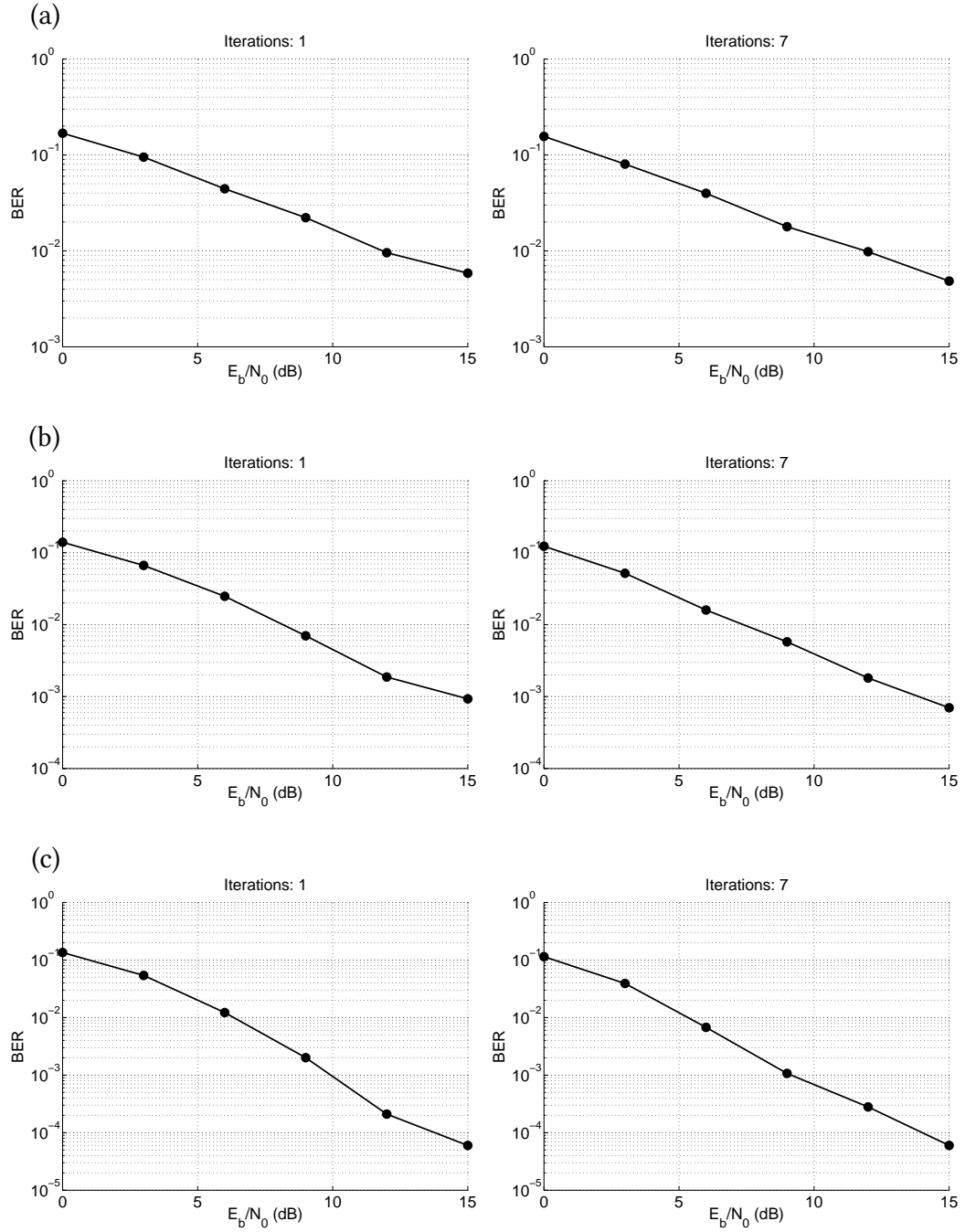


Figure 5.16: BER analysis of a log-domain SPA decoder under Rician channel for: (a) $K = 2$, (b) $K = 5$ and (c) $K = 10$.

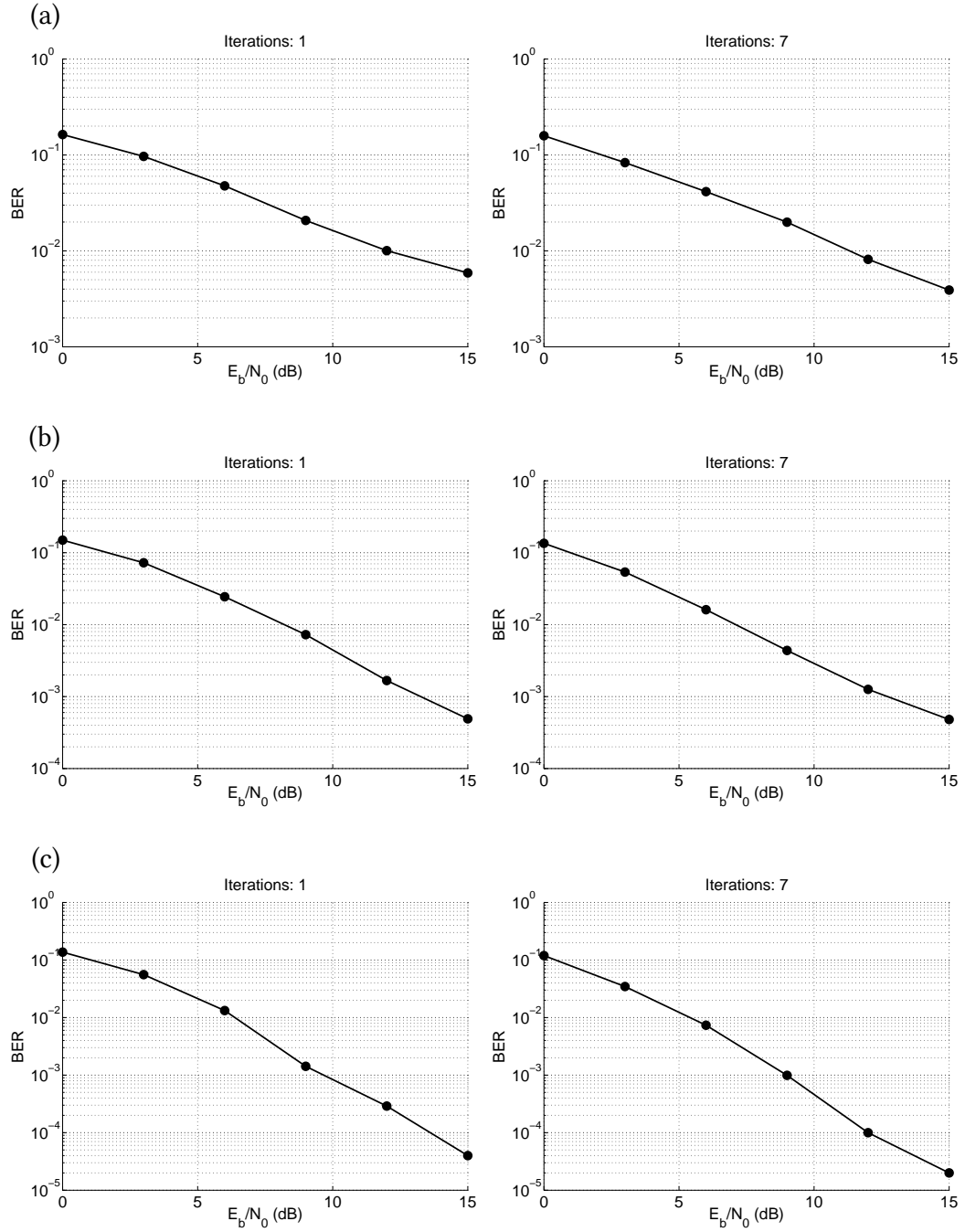


Figure 5.17: BER analysis of a simplified log-domain SPA decoder under Rician channel for: (a) $K = 2$, (b) $K = 5$ and (c) $K = 10$.

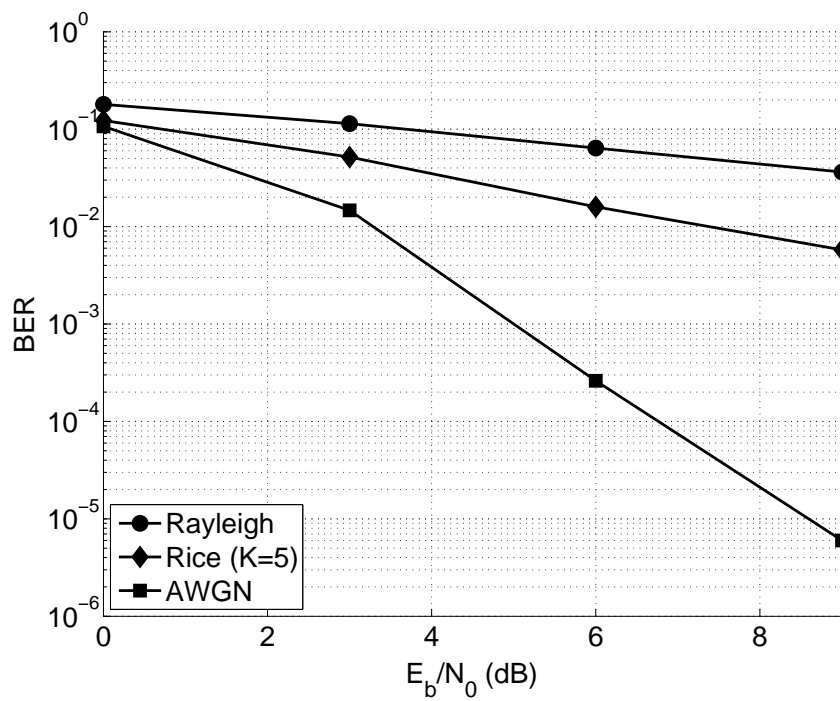


Figure 5.18: BER performance of a simplified log-domain SPA decoder operating for 7 iterations under AWGN, Rician and Rayleigh fading channels

Chapter 6

Implementation of an LDPC transceiver

Overview:

This chapter presents the implementation of an LDPC transceiver on a Xilinx Spartan-3E FPGA. After an overview of the whole design, each entity of it is analyzed in detail. This is done firstly by describing each entity's function and the components used by them. Then, the operation stages are presented, which show a more in-depth step-by-step view of the encoding and decoding procedures. Finally, the correct operation of each entity has been verified by run-time simulations and some snapshots are included depicting this.

6.1 Design Summary

The final part of this work deals with the implementation of a transceiver on an embedded system. The transceiver has defined specifications and is designed to be implemented on a FPGA of the Xilinx Spartan-3E family [28]. Its operation includes all steps of LDPC codes transmissions and the designed transceiver has the capabilities of both transmitters and receivers. Therefore, the transceiver is able to read source data from its input and encode them, and after being modulated and transmitted over a noisy channel, the transceiver can follow the reverse procedure, i.e. receive the encoded messages, demodulate and decode them, ultimately exporting them through its output port. The decoding technique which will be used by the decoder is the simplified form of the log-domain decoding algorithm described in subsection 5.1.3. In addition, for testing purposes, the input and output data of the transceiver will be monitored in order to extract information about errors in data decoding (bit error rate), concluding in observations about its efficiency.

The design includes two entities, which are the encoder and the decoder of the transceiver. The operation of both entities is described in VHSIC (very-high-speed integrated circuit) hardware description language (VHDL) in a total of more than 1.400 source lines of code (SLOC). The language statements of the entities are then

Language	VHDL
Platform	Xilinx Spartan-3E
Model	XC3S250E FT256-5
Simulator	Xilinx ISim 11.5
Synthesizer	Xilinx XST 11.5 & Leonardo Spectrum 2009
Testing	Testcases with random input sources for various values of E_b/N_0
Verification	MATLAB simulations results

Table 6.1: Tools used for the implementation of the transceiver

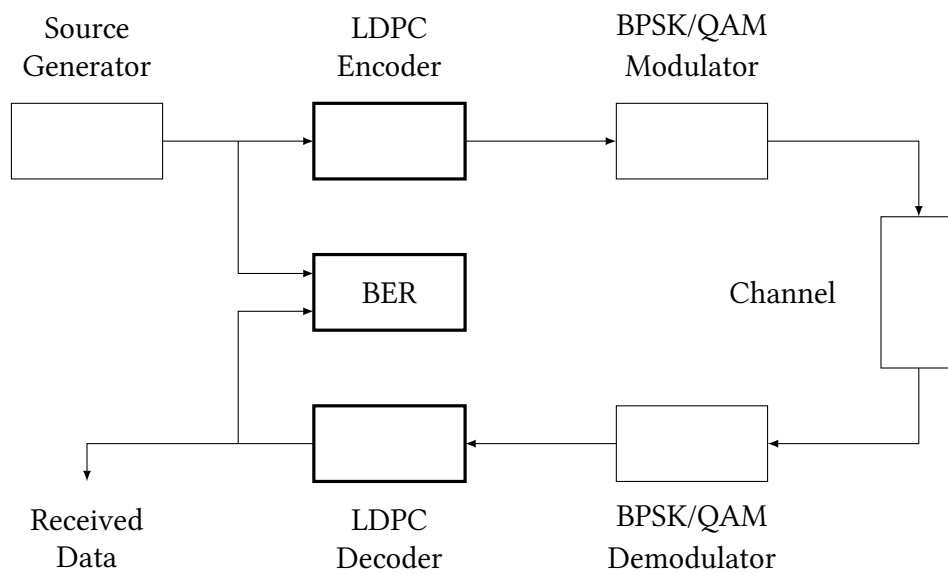


Figure 6.1: Transceiver's FPGA implementation block diagram

transformed into hardware logic operations, which in turn produce an equivalent netlist of generic hardware primitives to implement their specified behavior [29]. In addition, their operation is verified by means of simulation, both in the field of implementation and in the algorithmic field. Table 6.1 lists all the tools which are used in the design to carry out the above tasks.

The encoder and the decoder are parts of a greater design which describes the complete implementation of the transceiver on the FPGA, and which is presented in the block diagram of Figure 6.1. In this model, the implemented parts are shown in thicker outline than the rest of the system. This complete model can, ultimately, be used to verify its operation under a standardized protocol carrying out wireless local area network (WLAN) computer communication in the 5 GHz frequency band, which has been created and maintained by the IEEE LAN/MAN Standards Committee, under the family name IEEE 802.11 [30].

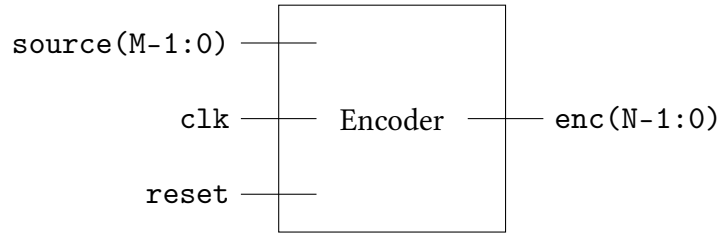


Figure 6.2: Encoder technology schematic top view; the input ports are at the left side, the output at the right.

6.2 Encoder implementation

6.2.1 Description

The purpose of the encoder is to read source data from its input and then encode them, exporting valid codewords for its parity-check matrix. This procedure includes the following steps:

1. Read source data: First, the encoder reads its input information in frames of k bits (messages).
2. Encode: Then, it maps the k bits of the source messages to n bits codewords using its parity-check matrix H , which is an $M \times N$ matrix, where $N = n$ and $M = n - k$. For rate- $\frac{1}{2}$ applications, it is $N = n$ and $M = k = \frac{n}{2}$. The parity-check matrix can be separated into two square matrices A and B of order M as: $H = [A|B]$. The codewords which are produced by the encoder include the source message bits preceded by the parity-check bits for this frame, as: $x = [c|s]$, where x is the output codewords, c the computed M parity-check bits and s the original input message of k bits. The codewords are valid, which means that they satisfy all parity checks on the source message, if:

$$H \cdot x = 0 \Rightarrow A \cdot c + B \cdot s = 0 \Rightarrow c = A^{-1} \cdot B \cdot s \quad (6.1)$$

where A^{-1} is the inverse of the square matrix A .

3. Transmit: Finally, the encoded message becomes BPSK modulated and is transmitted by the encoder through a noisy channel.

The encoder uses signals to read, manipulate, exchange and, finally, output information. The input and output ports of the system can be seen in the top view of the technology schematic shown in Figure 6.2: signals `clk`, `reset` and `source` are inputs, whereas `enc` refers to the output. The encoder also utilizes internal signals in order to proceed to the different states of the encoding process. A summary of the main signals which control the operation of the encoder is the following:

clk (input signal): Signal indicating the clock ticks of the system. The clock period is defined by the implementation.

reset (input signal): Signal indicating state of system initialization. During initialization any source message to the input port of the encoder is not processed and the system does not produce any output.

source (input signal): Source message to be encoded, which is a logic vector of M bits. The design is described using generics, and in this implementation source messages are set to be logic vectors of 8 bits, but can be anything set by the designer at design time.

enc (output signal): Encoded version of the source message; the output of the system. The output messages are logic vectors of N bits. For any rate- $\frac{1}{2}$ encoder, the output messages are vectors of $N = 2 \cdot M$ bits. In this implementation the produced codewords are 16 bits long.

clk_period (internal signal): Internal signal of the simulation testbench indicating the clock period of the system. In the following simulations this signal is set to represent 1ns clock period.

state (internal signal): Internal signal of the encoder (Unit-Under-Test (*UUT*) during the simulation testbench) indicating its state of operation. The encoding process completes in a total of 6 states. During state 0 the encoder is practically inactive since its only action on every clock tick is to check whether the source of the system has changed (indicating a new source message) and if it has, then it continues to the encoding process of the new source message. During the next states the calculation of the parity-check bits takes place, and, finally, during state 5 the encoded message is exported to the system's output in the form of:

$$\text{encoded message} = [\text{parity-check bits}|\text{source}] \quad (6.2)$$

6.2.2 Implementation

The encoder is implemented using VHDL to describe its operation, which can be found in subsection B.2.1 of Appendix B. The encoder's operation is divided in a total of 6 states operated by a finite state machine (FSM)¹. The entity is controlled by a process, which is triggered by clock events and reset button events. When the reset button is pressed (the `reset` signal gets a "1" value) the system is initialized, i.e. all actions are stopped and the operation state is reset to 0. Otherwise, on each positive clock tick, the system checks its state and acts accordingly. When one state has finished its function, the system moves to the next state up until state 5, during which the encoder outputs the encoded message through its output port and afterwards restarts to state 0.

¹A finite-state machine (FSM) is a behavior model composed of a finite number of states, transitions and actions between those states. Its operation begins from one of the states (*start state*), goes through transitions depending on input to different states and ends in any of those available final states.

During its operation, the encoder moves through 4 different main stages which either focus on arithmetic calculations or detecting changes in the encoder's environment. These stages are described next.

Stage 1: System reset

While the reset button is pressed and the `reset` signal gets a "1" value the system stops any calculations currently doing and resets its internal components. The encoder utilizes two block RAMs to store the rows of the parity-check matrix $H = [A|B]$ and the inverse of matrix A , A^{-1} , which are being rewritten while the system is resetting. Therefore, the encoder re-reads the two matrices, H and A^{-1} during this stage, either by an external source or by its own hardcoded values set a-priori, and stores their values (per row) to the corresponding block memories. Afterwards, it sets its operation state to 0.

Stage 2: System idle

When the encoder is at state 0 and during all the time it remains in this state, the encoder checks the input port, `source`, for changes, which indicate a new input message to be encoded. If a new input is detected, the system moves on to the next state, otherwise it remains on state 0. If the encoder changes its operation state to encode the new message, it first clears all variables used by the encoding process during previous runs.

Stage 3: Message encoding

The encoding process of a new message takes place during states 1 and 4. These states use the parity-check matrix and the input message to compute the parity-check bits which correspond to the combination of the two. The encoder repeats these steps as many times as needed, the number of which is determined by the dimensions of the parity-check matrix, M and N . The parity-check bits are normally produced by the multiplication of A^{-1} with B and the source message. However, since all these matrices use zeroes and ones as values, the multiplications can be transformed into simple signal value checks to simplify the design, as:

```
-- tmpbits = B * source:
-- calculate [H(i)(N-M+j) * source(M-1-j)] by
-- replacing multiplication with signal checks
if h_dout(N-M+temp_j) = '1' and cur_source(M-1-temp_j) = '1'
then
    temp := temp + 1;
end if;
```

and then continuing to:

```
-- chkbits = inv_A * tmpbits
-- calculate [inv_A(i)(j) * tmpbits(M-1-j)] by
-- replacing multiplication with signal checks
if inv_a_dout(temp_j) = '1' and tmpbits(M-1-temp_j) = '1'
```

```

then
    temp := temp + 1;
end if;

```

The temporary and final parity-check bits are then calculated by the mod of their `temp` variable, each independent to the other since they change values at different states, to the number 2. For binary numbers, division by 2 equals to one right shift of the binary number, and the modulo operation of the division is then the last bit (farthest to the right) of the number.

Stage 4: Codeword exporting

Finally, during state 5, the system sends to its output port the signal of the encoded message, `enc`, which consists of the parity-check bits followed by the original input message. The encoder then saves the source message currently encoded in order to compare it to the next messages received to its input port, and then moves to state 0 to repeat its operation cycle.

For a $(M = 8, N = 16)$ application, the memory components used by the encoder are one 8×16 -bit block RAM for parity-check matrix H and one 8×8 -bit block RAM for the inverse matrix A^{-1} as shown in Figure 6.3. Each memory component consists of 4 input ports and 1 output port. Input port `clk` is synchronized to the system clock, while port `we` is the “write-enable” signal of the memory, which enables or disables writing to memory addresses: when it gets a “1” value, input messages to the memory component will be written, otherwise data are only read and input data are not stored. Through port `data_in` input data are received by the memory component and when write is enabled, `address` indicates the address to which these data will be saved. On the other hand, when in read-only mode, `address` will dictate where the component should recover data from. Finally, the output port `data_out` exports the recovered data from the specified address, and when in write mode, this will export the new data in a write-after-read sequence. The top view schematic of the memory components is presented in Figure 6.4. Signals `address`, `data_in` and `data_out` are logic vectors of variable width in general, and the encoder and decoder define explicitly their width according to the data they will store in each memory block. The VHDL code of the block RAM is presented in Section B.1 of Appendix B.

The encoder described with this VHDL code has been implemented on a FPGA of the Xilinx Spartan-3 family, the model XC3S250E FT256-5 of the Spartan-3E FPGAs sub-family, which includes 2.448 slices used for high-performance general logic applications and hierarchical memory architecture of 216Kbits of block RAM, which provides data storage in the form of 18Kbit dual-port blocks. The complete specifications list of the Spartan-3E family can be found in [28].

The implementation occupies a total of 147 slices out of the 2.448 slices, which means that the device utilization is 6%. This is highly affected by the simple operation

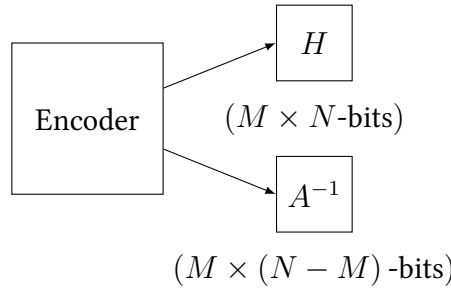


Figure 6.3: Encoder memory utilization

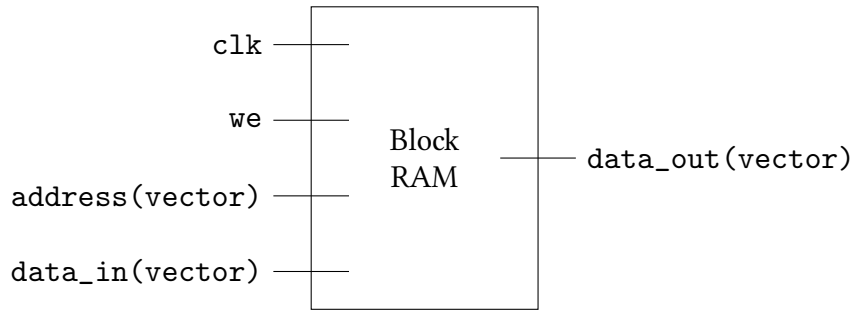


Figure 6.4: Block RAM component technology schematic top view

of the encoder, which has been further simplified by replacing multiplications with signal checks. The device utilization summary of the design is presented in Table 6.2.

The synthesis process of the encoder produces an *RTL schematic* describing its operation. In the RTL design, the encoder's behavior is defined in terms of the flow of signals (or transfer of data) between hardware registers, and the logical operations performed on those signals. A *technology schematic* is also produced, which shows a representation of the design in terms of logic elements optimized to the target device or technology, for example, in terms of LUTs, carry logic, I/O buffers, and other technology-specific components. This schematic offers a technology-level representation of the VHDL code optimized for a specific FPGA architecture.

6.2.3 Simulation

After having been described in VHDL, the encoder is simulated to verify its correct operation. The system is simulated in the Xilinx ISim environment using a testbench specifically created for the encoder and which is presented in subsection B.2.2 of Appendix B. During the simulation process, input data are read from a specified file on the computer on which the simulation runs and are passed to the input port of the encoder. Afterwards, the encoding process takes place, and when it has finished, the codewords produced in the output port of the system are checked with the expected codewords for each specific input message. The expected codewords are also read from the same file the input messages have been read.

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	100	4.896	2%
Number of 4 input LUTs	169	4.896	3%
Number of occupied Slices	147	2.448	6%
Number of Slices containing only related logic	147	147	100%
Number of Slices containing unrelated logic	0	0	0%
Total Number of 4 input LUTs	233	4.896	4%
Number used as logic	169		
Number used as a route-thru	64		
Number of bonded IOBs	26	172	15%
IOB Flip Flops	24		
Number of RAMB16s	3	12	25%
Number of BUFGMUXs	1	24	4%
Average Fanout of Non-Clock Nets	2,88		

Table 6.2: Device utilization summary of a ($M = 8, N = 16$) encoder

When all source messages of the input file have been encoded, the testbench extracts information about possible errors in the parity-check bits created by the encoder. No erroneous bit is allowed, and such case is unacceptable since it would cause non-deterministic behavior of the encoder. Therefore, the outcome of the testbench must always be zero erroneous bits out of the total of both parity-check bits calculated and the original bits of the source which are also part of the codewords.

The following images present several snapshots of the simulation of the encoder's operation as suggested by ISim. This simulation has provided the encoder with 125 vectors of 8-bit inputs and has received another 125 vectors of 16-bits codewords, i.e. a total of 1.000 input bits and 2.000 output bits. The input bits are random $\{0, 1\}$ -valued vectors created by MATLAB and the expected output bits are also produced by a MATLAB function created for simulation purposes.

A full view of the simulation execution is shown in Figure 6.5. Two more specific views of the initialization and the first execution of the simulation are shown in Figures 6.6 and 6.7. The next figures focus on the states of the encoder during the encoding process of a source message and during consecutive source messages (Figures 6.8 and 6.9). Finally, the bit error check, which is done after the encoder has finished encoding all input messages, is shown in Figure 6.10.

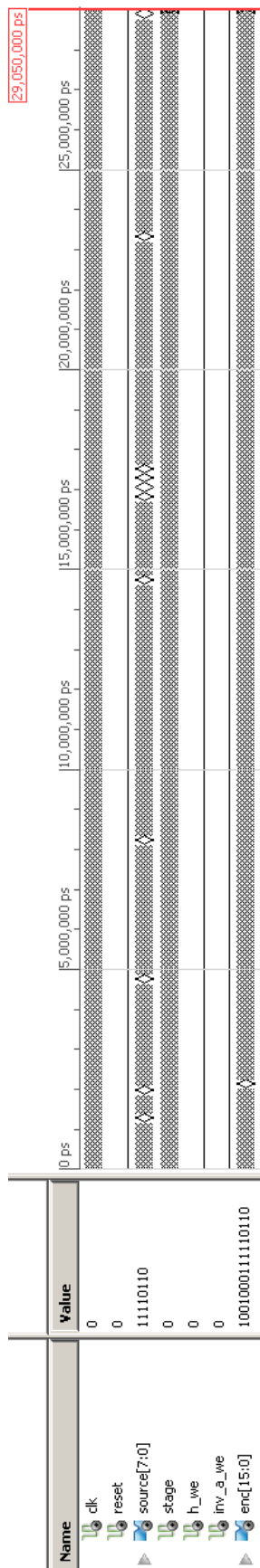


Figure 6.5: Full view of the encoder simulation, which shows the states of the system’s signals – along with the source messages to be encoded and the codewords produced in the output. Notice that during normal operation of the encoder no writings to memory are done, hence the “write-enable” signals **h_we** and **inv_a_we** constantly have a “0” value.

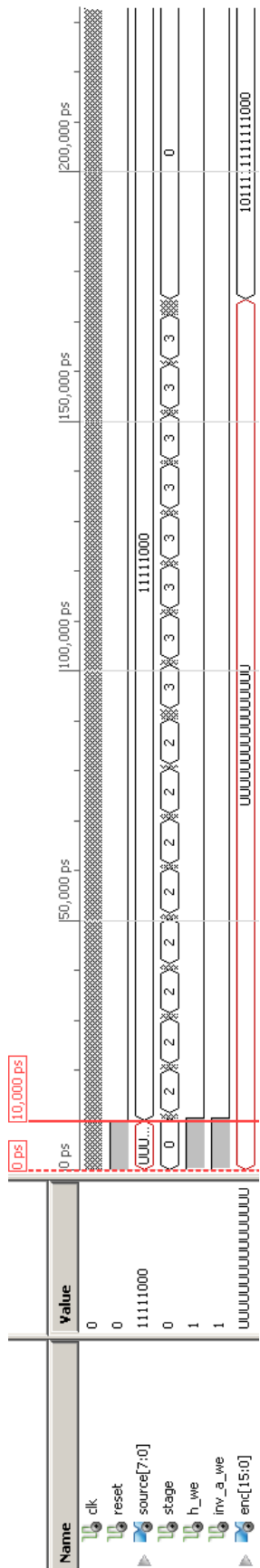
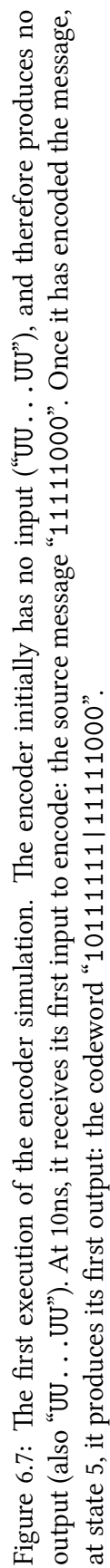
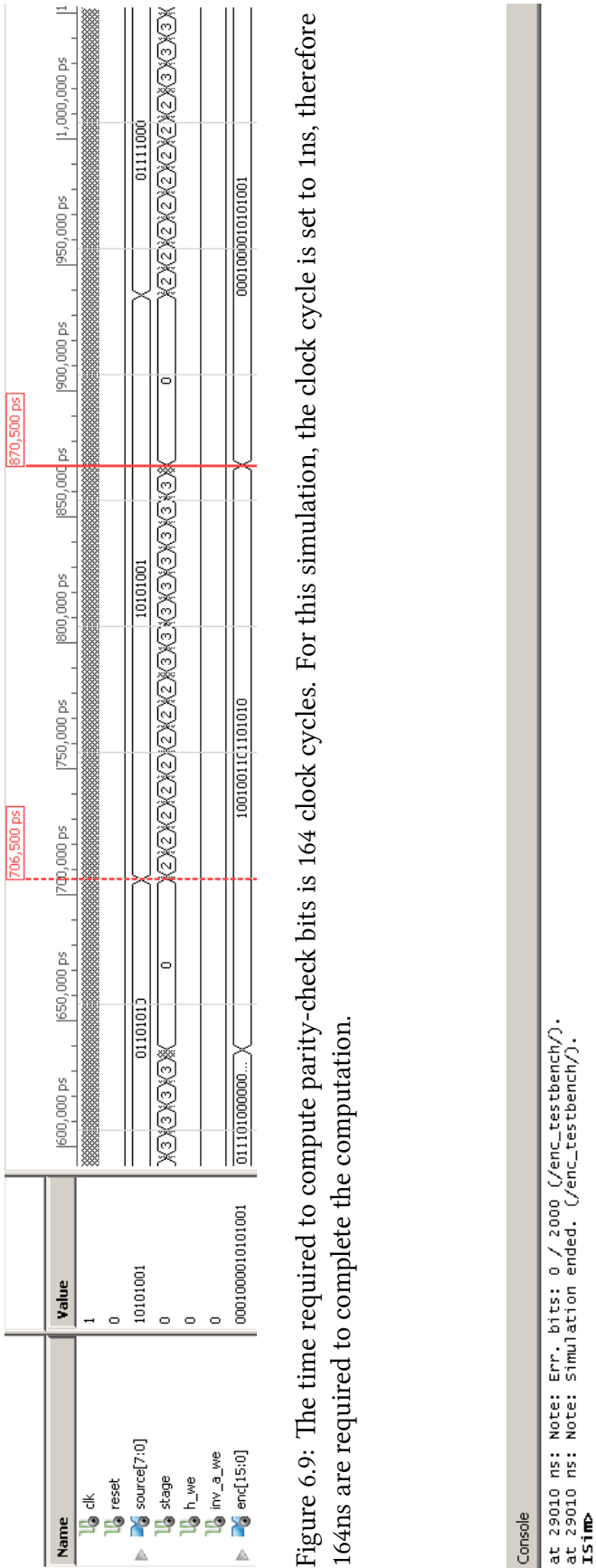


Figure 6.6: Encoder simulation during initialization, when the **reset** signal forces the system to set its operation state to 0 (it was undefined at the very beginning of the simulation) and produces no (or undefined) output (“UU . . . UU”). While the system is resetting, the encoder rewrites the values of the parity-check matrix and the inverse matrix of *A* to the block RAMs, therefore the “write-enable” signals of both memories are set to “1” for this period.





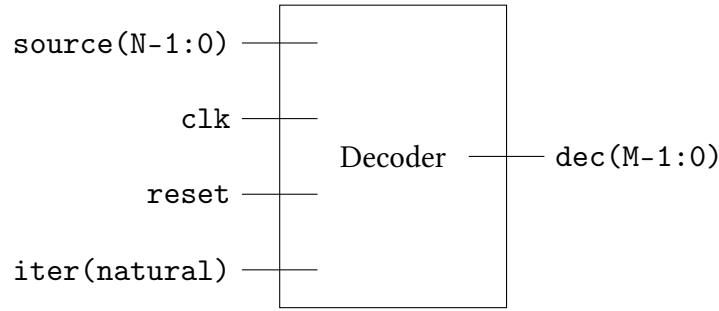


Figure 6.11: Decoder technology schematic top view

6.3 Decoder implementation

6.3.1 Description

LDPC codes are decoded at the receiver's side using one decoding algorithm, such as the ones described in the previous chapter, aiming at retrieving the original information sent by the transmitter. Having tried and compared different approaches of decoding techniques, this implementation has adopted the simplified log-domain message passing algorithm described in subsection 5.1.3 (page 40) and which was simulated with `bertool` using the code presented in Section A.4 of Appendix A (page 100) to decode the codewords received at the decoder's input port. This process exchanges soft-information iteratively between the variable and check nodes created by the parity-check matrix, in the form of messages describing log-likelihood ratios (LLRs), and, in the end, it produces the decoded version of its input message, which, ideally, should be exactly the same as the originally sent message. The procedure includes the following steps:

1. Read encoded source data: The decoder receives encoded data from its input port in frames of n bits. These messages are naturally distorted by channel noise.
2. Decode: Each frame is, then, decoded trying to produce the originally sent message by the source. The decoder initially produces messages of n bits, the first $n - k$ of which are parity-check bits and the last k bits compose the original message. These messages are ideally the same as the output of the encoder which encoded the source message, i.e. the produced codeword. The decoding process is repeated several times, the maximum number of which is defined by the user at run time using an extra input port to the decoder.
3. Extract original message: After decoding the input codeword, the decoder finally extracts the bits of the originally sent message by keeping the last k bits of the decoded message.

The top view of the decoder's technology schematic is presented in Figure 6.11. As seen in it, the decoder equips 4 input ports (which define signals `clk`, `reset`,

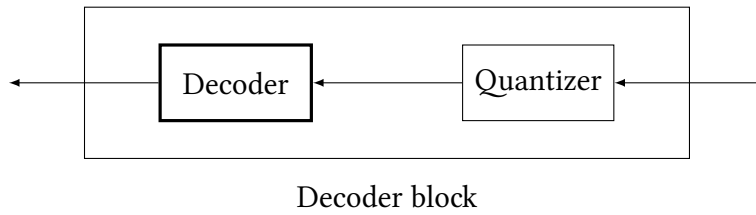


Figure 6.12: Quantizer in the decoder block

`enc` and `iter`) and 1 output port (defining signal `dec`). The decoder also utilizes internal signals used during the states of the decoding process. A summary of the main signals is presented below:

`clk` (input signal): Signal indicating the clock ticks of the system. The clock period is defined by the implementation.

`reset` (input signal): Signal indicating state of system initialization. During initialization the source message is not processed and the decoder does not produce any output.

`source` (input signal): Encoded version of the source message initially sent by the source generator. Originally sent source messages are logic vectors of M bits and the encoded version of them are logic vectors of N bits. These N bits however, after being transmitted through a noisy channel have their values altered, resulting in being mapped to vectors of real numbers instead of bits. Real numbers are hard to handle by an embedded system; for this reason a quantizer is equipped to represent the real data to signed fixed point notation, which can in turn be easily translated into logic vectors of fixed length. Therefore, this implementation equips an additional block preceding the decoder, which maps the vectors of real data received by the channel to an array of logic vectors of fixed length and which is shown in Figure 6.12. In this way, the input port of the decoder accepts streams of N logic vectors which represent N fixed point numbers. Using generics, the value of N is set to 16, but this can be set to any value decided at design time.

`iter` (input signal): Signal indicating the maximum number of iterations of the belief propagation decoding algorithm. This design implements a decoder which utilizes the early termination scheme described in Section 4.3; thus the number of iterations actually implemented by the decoder at run time may be less than the maximum number set at design time if the decoder detects that it has converged to its final output before exhausting that maximum number allowed by design. This signal is described in integer format, and in the implementation this is translated into a logic vector of length proportionate to the range of the numbers allowed.

`dec` (output signal): Decoded version of the encoded input message; the output of the decoder. The output messages are logic vectors of M bits, same length as

the input messages originally sent to the encoder. The decoded output should be as identical as possible to the message originally transmitted, and ideally is bit-by-bit the same. For this implementation, generics dictate that the output messages are of 8 bits length.

clk_period (internal signal): Internal signal of the simulation testbench indicating the clock period for the simulation; in the following simulations this signal is set to represent 1ns clock period.

state (internal signal): Internal signal of the decoder (Unit-Under-Test (*UUT*) during the simulation testbench) indicating its state of operation. The decoding process completes in a total of 26 states. Similarly to the encoding process, during state 0 the decoder checks on every clock tick whether the source of the system has changed (indicating a new input message) and if it has, then it moves through the next states of the decoding process of the new input message. Finally, the output of the decoded message is exported during the final state.

6.3.2 Implementation

The operation of the decoder is described in VHDL using the code presented in subsection B.3.1 of Appendix B and it is divided into 26 states operated by a finite state machine (FSM). In a way similar to the encoder, the decoder's operation is controlled by a process, which is triggered by clock events and reset button events. If the **reset** signal gets the value "1" (indicating the reset button is pressed), then the system is initialized, stopping all actions and resetting the operation state to 0. Else, on each positive clock tick, the system checks the state of its state and functions according to it. In this way, the system moves gradually to the next states up until the final state, during which the decoder produces the decoded version of the input codeword through its output port and afterwards restarts to state 0.

The decoder utilizes 7 block RAMs to save data used and calculated during the decoding process. The implementation of the RAMs is the same as those used by the encoder. For this (8, 16) decoder the storage requirements are the following:

- one 8×16 -bit memory to store the parity-check matrix H of the same dimensions. Each address of the memory points to a line of H .
- two 128×16 -bit memories to store the check nodes and variable nodes locations which exchange messages during the decoding process. In each address of these memories the decoder saves the location of the nodes in binary format. The memories are divided in banks of 8 and 16 cells respectively, indicating the connected variable nodes each check node has and vice versa.
- one 128×1 -bit memory to store the values of the a_{ij} matrix produced during the decoding process. This matrix saves the signs of the log-likelihood ratios of the variable-to-check messages in the form of "1" in case of negative number and "0" when positive or zero.

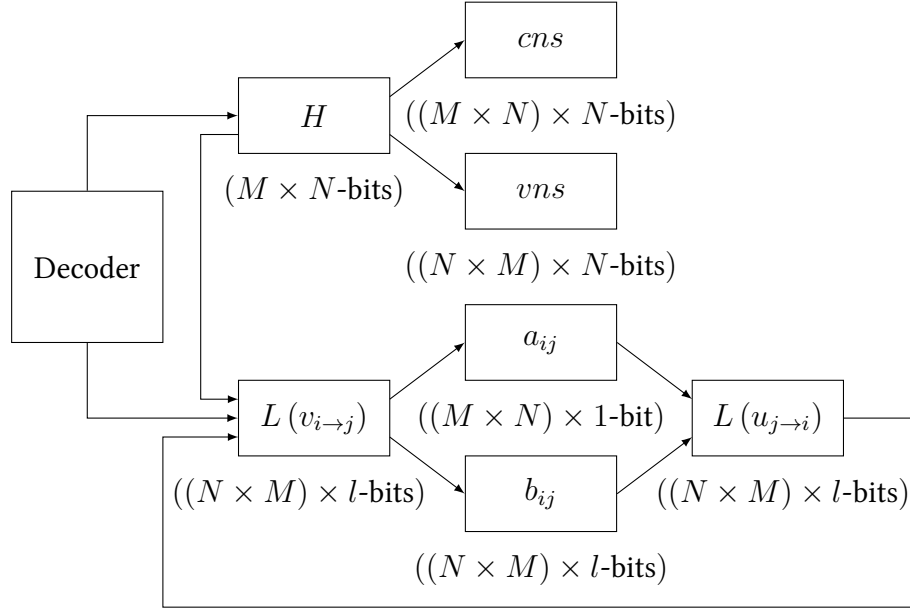


Figure 6.13: Decoder memory utilization; l is the length of the bit vectors of the input numbers' signed fixed point representation.

- three 128×8 -bit memories to store the value of the b_{ij} matrix produced at the same time as the a_{ij} matrix, as well as the values of the LLRs of the variable-to-check messages $L(v_{i \rightarrow j})$ and check-to-variable messages $L(u_{j \rightarrow i})$. These memories save a number in signed fixed point notation in each address in the form of logic vectors of 8 bits.

The way these block RAMs are implemented on the FPGA is presented in Figure 6.13 and the way these RAMs are divided into smaller memory banks is shown in Figure 6.14.

During its operation, the decoder moves through 6 different main stages, each of which serves a specific role in the decoding process. These stages are described next.

Stage 1: System reset

A value of “1” to the signal `reset` indicates that the reset button is pressed, and when this case is detected, the system stops any calculations initiated by the decoding process of an input codeword and resets its connected component which is needed a-priori for decoding and is not calculated during the process. This is the parity-check matrix, H , which is stored as an $(M \times N\text{-bits})$ array to a block RAM. In order to (re)write the values of the parity-check matrix to the memory, the “write-enable” signal is set to “1” during this stage, and all memory addresses are filled with the values of each row of H in succession. When the writing is complete, the system sets its operation state to 0.

Stage 2: System idle

During the time after decoding an input codeword and before receiving a new one, the decoder remains at state 0. When in this state, the decoder checks its input port, source, to detect changes, indicating a new input codeword to be decoded. If a new codeword is detected, the system moves on to state 1 and successively to the following states of the decoding procedure, otherwise it remains on state 0 and repeats the check at every positive clock tick. Before moving to the first decoding state, the system sets the “write-enable” signal of the block RAMs cn s and vn s to “1”, because these matrices will be calculated and stored in the following state. The system also stops any writing to the parity-check matrix by setting the “write-enable” signal of its corresponding memory block to “0”.

Stage 3: Message decoding — (i) Check and variable nodes detection

The first stage of the decoding process includes the initial steps after a new input has been received for decoding. During states 1 to 3 the decoder initially finds the connected check nodes and variable nodes of the parity-check matrix H and then associates the input codeword with the non-zero elements of H , in order to produce the first variable-to-check LLR messages, $L(v_{i \rightarrow j})$. The check and variable nodes are stored in two separate block RAMs in the way which is presented in Figure 6.14. In each memory address the location of the connected check node or variable node is stored, starting from zero, and the last row of each bank stores the total number of connected nodes to the corresponding node of each bank, also starting from zero.

For example, for a specific application, the connected variable nodes to check node 0, cn_0 , are the variable nodes 2 and 8, notated as vn_2 and vn_8 respectively. For this application, the check nodes memory bank will have the value 0010 (number 2 in the decimal numeral system) in address 0000 and value 1000 (decimal number 8) in address 0001, for an implementation which uses 4 bits to store data to the memory blocks and 4 bits for the address index. In addition, the value 0001 will be stored in address 1111 (or 15), indicating that the first check node has 2 (0001 + 1) connected variable nodes, which can be found in the cells $cn_0(0)$ and $cn_0(1)$. This procedure continues for the rest of the check nodes and is likewise repeated for the variable nodes block memory.

After this stage the decoding process is ready to begin the iterations of the belief propagation algorithm at the following states.

Stage 4: Message decoding — (ii) Horizontal step, check nodes update

At state 4 the iterations of the decoding process begin and the following states up to the final one are repeated for as many times as either the maximum number of iterations indicate or until the decoder has detected that its output has converged.

At first, during state 4, the LLRs of the variable-to-check messages are separated to the sign (a_{ij}) and absolute value (b_{ij}) of them, which are both stored to two separate memory blocks. The block RAM of a_{ij} has dimensions $((M \times N) \times 1\text{-bit})$, since

Address	Value	Address	Value
0	$cn_0(0)$	0	$vn_0(0)$

7	$cn_0(7)$...
8	$cn_1(0)$	15	$vn_0(15)$

15	$cn_1(7)$...
16	$cn_2(0)$	112	$vn_7(0)$

23	$cn_2(7)$...

120	$cn_{15}(0)$	127	$vn_7(15)$
	...		
127	$cn_{15}(7)$		

Figure 6.14: Division of the block RAMs utilized by the decoder into smaller banks; on the left side is the check nodes memory block, on the right side is the variable nodes memory block of a $(M = 8, N = 16)$ LDPC decoder. Similarly, all block RAMs are divided into memory banks corresponding to the rows of their respective matrices.

$(M \times N)$ items of 1 bit each are stored there. The bit gets a zero value for positive sign or zero number, and on the other hand, a value of one for negative numbers. The block RAM of b_{ij} has dimensions $((M \times N) \times l\text{-bits})$, because it will store $(M \times N)$ items of l -bit length signed fixed point numbers.

The horizontal step of the decoding process takes place during states 5 to 16, and during these states the check nodes update their estimations and send their response check-to-variable LLRs to their connected variable nodes.

These steps are repeated per row for each check node of the application. Initially, each check node gathers information from all connected variable nodes in the form of LLR messages during state 6. Then, it continues to find the minimum value of b_{ij} from those received by their connected variable nodes, and afterwards, the multiplication of the a_{ij} values of the connected nodes takes place. In a way similar to the encoder, the multiplication is replaced by signal checks in order to reduce the implementation's complexity:

```
-- calculate product of a_i'j which can be either 1
-- or -1, therefore only counting the number of -1s
-- is necessary to calculate the value of the product
if a_ij_dout = '1' then
    neg_a_ij := neg_a_ij + 1;
end if;
```

and then the sign of the multiplication can be found by checking the parity of the total number of -1 s; an even count leads to positive product whereas an odd one leads to negative product, which is represented using the two's complement of the currently minimum binary number:

```
-- response check-to-variable LLRs
if neg_a_ij mod 2 = 0 then
    -- positive product
    L_u_ji_din <= cur_min;
else
    -- negative product
    L_u_ji_din <= (not cur_min) + '1';
end if;
```

After the last step, the response check-to-variable messages have been calculated and they are stored to a separate memory block with dimensions $((M \times N) \times l\text{-bits})$, since $(M \times N)$ l -bit length signed fixed point numbers will be stored there. When this is complete, the decoder moves on to the following stage.

Stage 5: Message decoding — (iii) Vertical step, variable nodes update

In the next stage, the variable nodes update phase takes place (vertical step) and the variable nodes calculate their response variable-to-check LLRs to send to their connected check nodes. This process starts at state 17 and completes at state 23. During this period the variable nodes gather information from all their connected check nodes in the form of LLR messages and update their response variable-to-check messages by the summation of the received $L(v_{i \rightarrow j})$ messages successively per column.

The calculated variable-to-check messages are then stored to their own memory block, also sized $((M \times N) \times l\text{-bits})$ as it will store similar items to the check-to-variable messages memory block, in order to be used by their connected check nodes in the following iteration of the decoding algorithm.

During the following states, 24 and 25, the variable nodes also decide upon the value of the decoded bits; however, this decision will be used only if its taken on the last iteration of the process. After state 25, the system returns to state 17 and checks if there are more variable nodes yet to calculate their response messages, and if there are, it repeats the variable nodes update for these nodes, otherwise both horizontal and vertical step have finished and the decoding process of the current iteration is complete. The decoder utilizes an implementation of the early-termination scheme: at the end of each iteration it checks whether the decoded bits of the current iteration equal the decoded bits of the previous iteration, and if they do then the decoder detects that it has most likely converged to its final output and proceeds to the exporting of these bits in state 26. On the other hand, if the decoder has not converged yet, it will return to state 4 of the horizontal step to repeat the procedure for the next iteration, up to the maximum number of iterations defined by the corresponding input port:

```
-- early termination check:
-- if the decoded output of this iteration equals
-- the output of the previous iteration then stop
if cur_iter < iter and dec_pre /= dec_temp then
    cur_iter <= cur_iter + 1;
    temp_i := 0;
    temp_j := 0;
    L_v_ij_add <= L_v_ij_add + '1';
    dec_pre <= dec_temp;
    stage <= 4;
else
    stage <= 26;
end if;
```

Stage 6: Decoded bits exporting

Finally, when the decoder reaches state 26, the input codeword has been decoded and the bits have been decided. The decoder sends the decoded bits to the output of the system and resets its operation state to 0. This step also includes the removing of the preceding parity-check bits, as only the last M bits of the decoded message will be exported, since they are the ones which correspond to the bits of the originally sent message.

Using the VHDL code which describes the operation of a generic (M, N) LDPC decoder, a specific implementation for $(M = 8, N = 16)$ codes has been made. The design is also intended for a Xilinx Spartan-3E FPGA, the same model 3S250E FT256-5 as the one used for the encoder.

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	938	4.896	19%
Number of 4 input LUTs	1.799	4.896	36%
Number of occupied Slices	1.169	2.448	47%
Number of Slices containing only related logic	1.169	1.169	100%
Number of Slices containing unrelated logic	0	0	0%
Total Number of 4 input LUTs	2.032	4.896	41%
Number used as logic	1.799		
Number used as a route-thru	233		
Number of bonded IOBs	142	172	82%
IOB Flip Flops	136		
Number of RAMB16s	5	12	41%
Number of BUFGMUXs	1	24	4%
Average Fanout of Non-Clock Nets	3,41		

Table 6.3: Device utilization summary of a ($M = 8, N = 16$) decoder

The implementation of the decoder is a lot more complex than that of the encoder since it requires more computations, use of larger arrays of data and the storing of these data to more memory blocks. However, using some simplifications like those described previously, the device utilization is kept at reasonable rates, which is a crucial part of area-restricted applications, such as the implementation on an embedded system.

The decoder occupies a total of 1.169 slices out of the 2.448 slices available on the FPGA, meaning that the device utilization is at 47% in terms of occupied slices. The device utilization summary of the design is presented in Table 6.3.

The synthesis process of the decoder also produces an RTL schematic which describes its operation in terms of the flow of signals (or transfer of data) between hardware registers, and the logical operations performed on those signals. In addition, a technology schematic is produced, which describes the design in terms of logic elements optimized to the target device or technology, such as LUTs, carry logic, I/O buffers, and other technology-specific components.

6.3.3 Simulation

The decoder has also been simulated to verify that it functions correctly using the Xilinx ISim environment and a testbench specifically created for it, which can be found in subsection B.3.2 of Appendix B. Like the simulation process for the encoder, during the decoder's simulation, input data are read from a specified file and are passed to the input port of the decoder. Input data represent distorted versions of the codewords sent by the encoder, and the level of distortion depends on the normalized signal-to-noise ratio (E_b/N_0). The representation of the input data is in signed fixed point number format, with 1 bit used for the sign, 3 bits for the integer part and 4 bits for the decimal part, therefore the input data is a stream of 16 8-bit logic vectors,

representing 16 real numbers (codewords). Afterwards, these input codewords are decoded, and when the process has finished, the decoded bits exported to the output port of the system are checked with the initially sent codewords, which are also read from the same file the input messages have been read.

Once all source messages of the input file have been decoded, the testbench extracts information about the number of the erroneous bits and the bit error rate (BER) of the decoded messages. The bit error rate is expected to have values similar to those extracted by the MATLAB simulations using `bertool` for all different values of E_b/N_0 and maximum number of iterations (which have been presented, for example, in Figure 5.6 for an AWGN channel). In some cases, a minor increase in BER is allowed, due to the penalty imposed to accuracy by the representation of the input real numbers to fixed point notation, which can result in loss of information.

The simulation has provided the decoder with 250 vectors of 16-bit inputs, which are distorted versions of codewords produced by the encoder in MATLAB, therefore receiving a total of 4.000 bits. The decoder decodes all these bits, as part of the decoding process, however, it extracts only half of them, which are the bits which correspond to the originally sent messages without the parity-check bits. Therefore, the bits checked for correctness are the 2.000 output bits the decoder extracts.

In the following images several snapshots of the simulation of the decoder's operation as produced by ISim are presented. The simulation runs for $E_b/N_0 = 5\text{dB}$ and a maximum of 7 iterations. First, a full view of the simulation execution is shown in Figure 6.15. Then, the initialization and the first execution of the simulation are shown in Figure 6.16. Afterwards, a full view of the states of the decoding process is presented in Figure 6.17 and the time required to complete the decoding process is shown in Figure 6.18. Finally, Figure 6.19 shows the representation of the input values the decoder expects and Figure 6.20 shows the bit error rate extraction on the exported bits which is used in order to verify the decoder's correct operation.

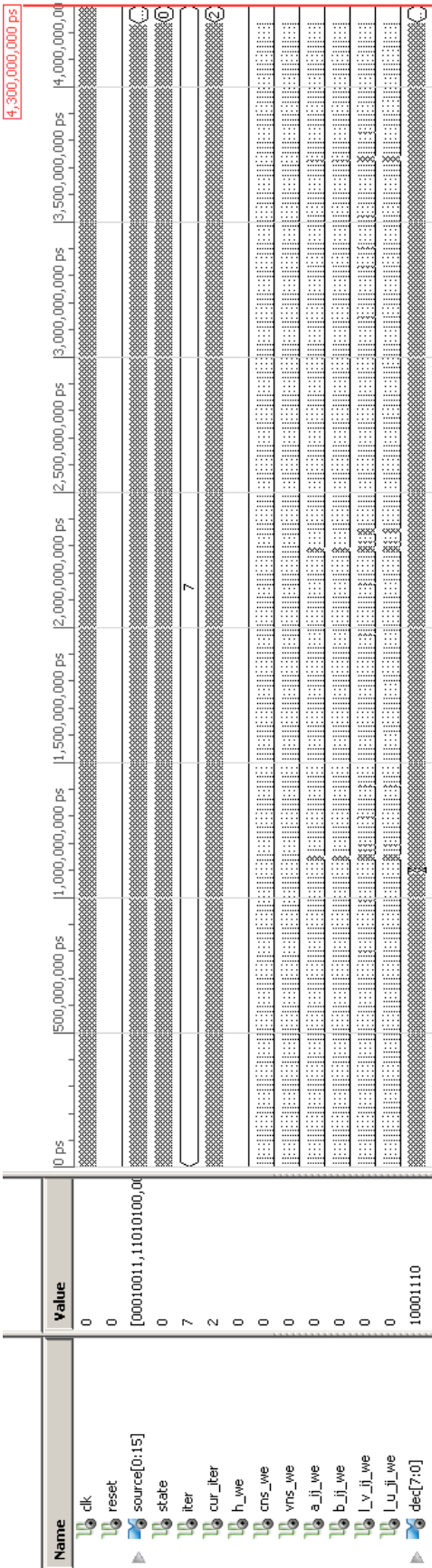
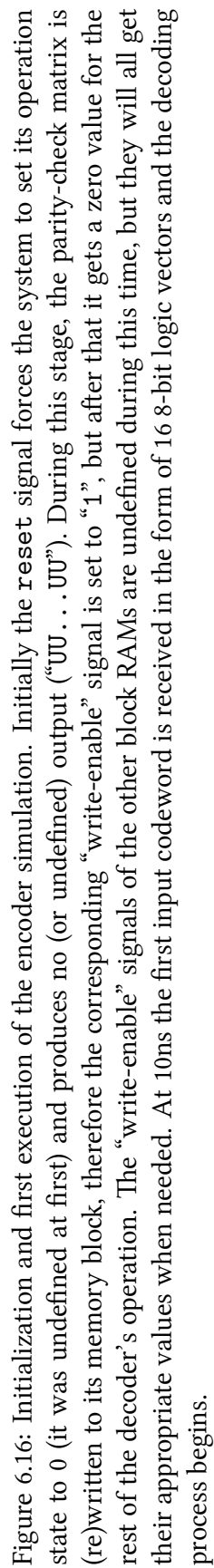


Figure 6.15: Full view of the decoder simulation, which shows the states of the system’s signals; among them are the input messages to be decoded and the decoded messages produced in the output as well as the “write-enable” signals of the memory blocks.



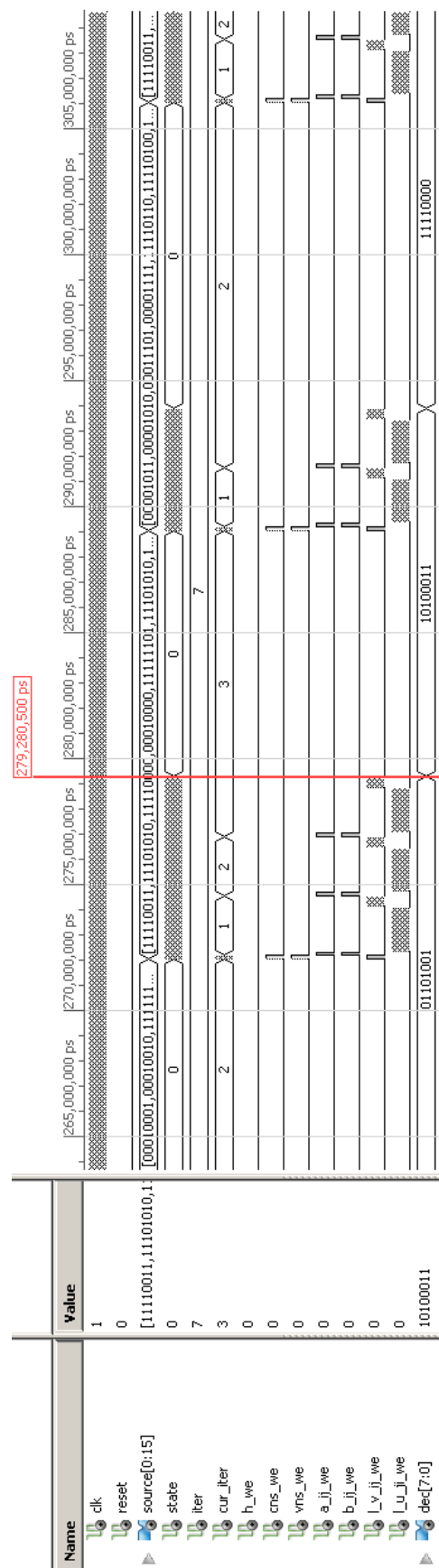


Figure 6.17: Stages of the decoding process. After exporting the decoded bits of the previous message and before receiving a new one, the decoder remains at state 0, during which it does not decode again the same source. On the other hand, once a new input message has been detected, the decoded gradually passes between all states of the decoding process and repeats the decoding states for as many times as indicated by the number of iterations set by its input port, except for if the decoder detects that it has converged to its final output earlier. In this simulation, the good energy per bit to noise density ratio ($E_b/N_0 = 5\text{dB}$) lets the decoder finish in usually 2 or 3 iterations, as seen by the value of the cur_iter signal.

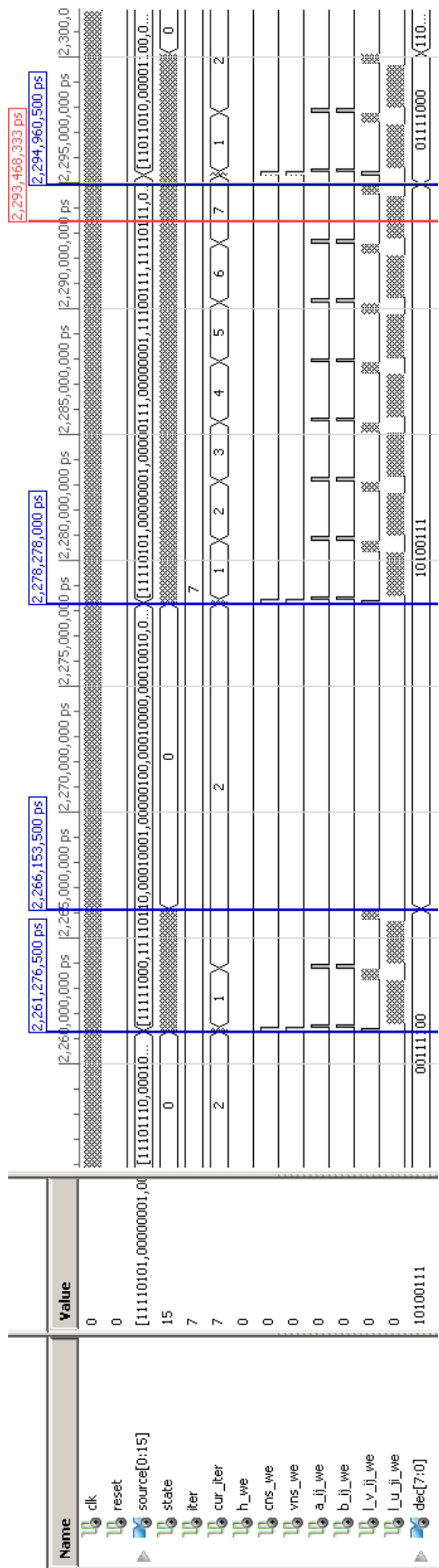


Figure 6.18: The time required to calculate the decoded bits is 16.682 clock cycles, when all 7 iterations are done. However, this period is reduced dramatically when less iterations are required, for example, for 2 iterations the needed clock cycles are 4.877. For this simulation, the clock cycle is set to 1ns, therefore up to 16.682ns are required to complete the calculation of the decoded bits.

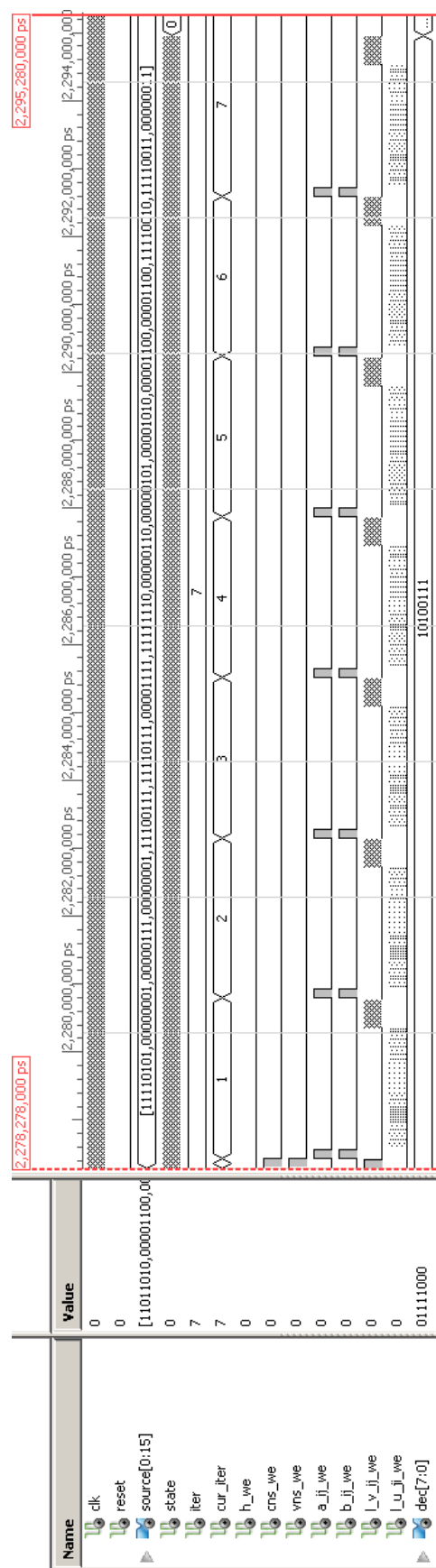



Figure 6.19: The messages which the decoder receives in its input port. The input codewords are expected to be in signed fixed point format with 1 bit assigned to represent the sign, 3 bits for the integer part and 4 bits for the decimal part. In this snapshot, the decoder reads the stream of these 16 8-bit vectors: “11110101” which represents $-0,6875$, “00000001” which represents $0,0625$, “00000111” which represents $0,4375$ and so on.

A screenshot of a MATLAB console window with a light gray background. The text is black and shows the results of a simulation. It includes timestamps, error counts, bit error rate (BER), and a final status message.

```
at 4250510 ns: Note: Err. bits: 1 / 2000 (/dec_testbench/).  
at 4250510 ns: Note: BER: 5.000000e-04 (/dec_testbench/).  
at 4250510 ns: Note: Simulation ended (/dec_testbench/).  
ISIM>
```

Figure 6.20: Bit error rate extraction on the bits exported by the decoder. The testbench monitors the output decoded bits of the decoder and checks them with the expected bits produced by MATLAB simulations in order to extract information about the bit error rate (BER) in the decoding procedure. The bit error rate should be similar to the values produced by `bertool`. In this simulation, the bit error rate is $5 \cdot 10^{-4}$ under $E_b/N_0 = 5\text{dB}$ for an AWGN channel.

Chapter 7

Conclusion and future perspectives

Overview:

The final chapter offers an overall review of the work covered in this thesis, presenting the final remarks and future perspectives of the designed LDPC transceiver.

The ever-growing communication needs of today's modern society has led to the rediscovery of the largely forgotten Low-Density Parity-Check (LDPC) codes, which were invented in 1962 by Robert G. Gallager. Along with turbo codes, LDPC codes are currently considered as the best performing channel codes, since they are capable of approaching very close to the Shannon limit.

LDPC codes are members of a large family of error correcting codes, the linear error correcting codes, which are methods of transmitting messages over noisy transmission channels. LDPC codes are defined by a sparse parity-check matrix and are controlled by a wide variety of parameters. As a result, LDPC codes offer a wide versatility which enables them to optimize their performance to a variety of applications and fit various different channel specifications.

Recent examples of LDPC codes utilization include optical communications, magnetic storage and satellite transmissions. Several new digital communication standards have adopted LDPC codes, such as the new DVB-S2 standard for the satellite transmission of digital television and 10GBase-T Ethernet, which sends data at 10 gigabits per second over twisted-pair cable. Nowadays, the implementation of LDPC codes is a hot topic in the field of digital communication, with researchers constantly conceiving new decoding techniques in an attempt to gradually improve performance and reach the ultimate limit of message transmission, the channel capacity.

This thesis focused on reviewing the concepts of digital communication which include encoding and decoding of binary messages transmitted through noisy channels and dealt specifically with the LDPC coding scheme. Three pillars support the work of the thesis:

Description of the communication scheme and LDPC codes

The first goal was to describe the communication scheme, which is used in modern telecommunications, and explore the options, capabilities but also the limitations offered by it. It has been shown that optimal code decoding is possible using a simple iterative algorithm, which is employed in LDPC codes. It has also been shown that performance of LDPC codes, as members of the iterative decoding algorithms, has a typical waterfall-like curve, which allows big improvements on the error probability rates between specific values of signal to noise ratios. In addition, several available techniques to further improve performance and implementation of LDPC codes have been presented; these techniques aim at reducing the required supply voltage and power consumption of an LDPC decoder, as well as increasing memory efficiency.

Comparison of different decoding algorithms

The second goal was to compare and contrast the effect of the decoding schemes on the performance of the LDPC codes. Four different algorithms have been observed for their performance, in terms of bit error rates, under Additive White Gaussian Noise (AWGN) channels, as well as Rician and Rayleigh fading channels. The results produced in this work suggest that the bit error rate is heavily affected by the chosen decoding technique and that it is inversely proportional to the implementation complexity. As a result, the challenge when designing an LDPC decoder is to choose the technique which offers the best trade-off between performance and complexity. In addition, the simulations clearly show the limitations imposed in performance by the transmission channels, which must also be taken into consideration when designing a decoder for a specific application; as a matter of fact, the figures show that performance under fading channels is degraded by many decades on the logarithmic scale compared to AWGN channels.

Implementation of an LDPC transceiver

The third goal was to choose the decoding technique with the best ratio of performance versus implementation complexity and utilize it on an LDPC transceiver designed for a FPGA of the Xilinx Spartan-3E family. The fact that the Spartan family belongs to the lower level platforms produced by Xilinx limits the hardware components available to the transceiver; therefore an efficient and less hardware demanding design had to be made. The decoder of the transceiver utilizes several optimization techniques, from the ones which have been described in previous sections, in order to achieve this.

The contribution of this thesis will be to offer a unified framework to the various decoding techniques described in it, in terms of operation performance and implementation complexity. In addition, it offers a generic architecture for LDPC

transceivers which can be used to implement on FPGA platforms. The design parameters can be set independently each time the transceiver is implemented (hence the term *generic*), enabling the transceiver to fit a wide variety of application and implementation specifications.

In the future, additional components can be added into the proposed transceiver in order to further improve its performance, reduce its hardware requirements, or enhance its operation by enabling different functions as well. The design is available to anyone interested to work in the field of LDPC codes and can also be used as an efficient tool to study the operation of LDPC codes and the way they are implemented on hardware chips. The simulation codes are provided as well, enabling experimentation with the different decoding techniques of LDPC codes.

Appendix A

MATLAB simulations source codes

Overview:

The first appendix includes the source codes which simulate the operation of the four different decoders discussed in this thesis. The codes are simulated in MATLAB and are implemented as functions, which are called by a main program according to the desired decoding technique. The following listings include the steps of the iterations of the decoding process for each decoding algorithm.

A.1 Hard-decision (bit-flip) decoder

```
1 % Iteration steps during the decoding process
2 % of a hard-decision (bit-flip) decoder
3 %
4 % in_msg:    incoming message
5 % H:        parity-check matrix
6 % iter:     number of iterations of the decoding process
7
8 for n = 1:iter
9
10    % horizontal step (check nodes update)
11    for i = 1:M
12
13        % connected variable nodes
14        vns = find(H(i, :));
15
16        % collect information from all connected
17        % variable nodes
18        for k = 1:length(vns)
19            u_ji(i, vns(k)) = mod(sum(v_ij(i, vns))...
20                                   + v_ij(i, vns(k)), 2);
21        end
22
23    end % horizontal step
24
25    % vertical step (variable nodes update)
```

```

26     for j = 1:N
27
28         % connected check nodes
29         cns = find(H(:, j));
30
31         % ones from connected check nodes
32         cns_ones = length(find(u_ji(cns, j)));
33
34         for k = 1:length(cns)
35             % update v_ij based on the majority of incoming
36             % information from connected check nodes
37             % and input source
38             if cns_ones + in_msg(j) >= length(cns)...
39                 - cns_ones + u_ji(cns(k), j)
40                 v_ij(cns(k), j) = 1;
41             else
42                 v_ij(cns(k), j) = 0;
43             end
44         end
45
46         % hard decision on bit decoding
47         if cns_ones + in_msg(j) >= length(cns) - cns_ones
48             dec(j) = 1;
49         else
50             dec(j) = 0;
51         end
52
53     end % vertical step
54
55 end % iterations

```

Listing A.1: Hard-decision (bit-flip) decoder MATLAB code

A.2 Probability-domain SPA decoder

```

1  % Iteration steps during the decoding process
2  % of a probability-domain sum-product algorithm decoder
3  %
4  % in_msg:    incoming message
5  % H:        parity-check matrix
6  % iter:     number of iterations of the decoding process
7
8  for n = 1:iter
9
10     % horizontal step (check nodes update)
11     for i = 1:M
12
13         % connected variable nodes
14         vns = find(H(i, :));
15
16         % collect information from all connected
17         % variable nodes
18         for k = 1:length(vns)

```



```

19
20     % column products
21     u_ji_temp = 1;
22     for l = 1:length(vns)
23         if l ~= k
24             u_ji_temp = u_ji_temp * (v_ij_0(i, vns(l))...
25                 - v_ij_1(i, vns(l)));
26         end
27     end
28
29     % response check-to-variable messages
30     u_ji_0(i, vns(k)) = (1 + u_ji_temp) / 2;
31     u_ji_1(i, vns(k)) = (1 - u_ji_temp) / 2;
32
33 end
34
35 end % horizontal step
36
37 % vertical step (variable nodes update)
38 for j = 1:N
39
40     % connected check nodes
41     cns = find(H(:, j));
42
43     % update v_ij
44     for k = 1:length(cns)
45
46         % row products
47         prod_u_ji_0 = 1;
48         prod_u_ji_1 = 1;
49         for l = 1:length(cns)
50             if l ~= k
51                 prod_u_ji_0 = prod_u_ji_0 * u_ji_0(cns(l), j);
52                 prod_u_ji_1 = prod_u_ji_1 * u_ji_1(cns(l), j);
53             end
54         end
55
56         % update constants K_ij
57         K_ij_0(cns(k), j) = P_0(j) * prod_u_ji_0;
58         K_ij_1(cns(k), j) = P_1(j) * prod_u_ji_1;
59
60         % response variable-to-check messages
61         v_ij_0(cns(k), j) = K_ij_0(cns(k), j)...
62             ./ (K_ij_0(cns(k), j) + K_ij_1(cns(k), j));
63         v_ij_1(cns(k), j) = K_ij_1(cns(k), j)...
64             ./ (K_ij_0(cns(k), j) + K_ij_1(cns(k), j));
65     end
66
67     % update constants K_i
68     K_i_0 = P_0(j) * prod(u_ji_0(cns, j));
69     K_i_1 = P_1(j) * prod(u_ji_1(cns, j));
70
71     % calculate Q_i

```

```

72     Q_i_0 = K_i_0/(K_i_0 + K_i_1);
73     Q_i_1 = K_i_1/(K_i_0 + K_i_1);
74
75     % soft-decision on bit decoding depending
76     % on the bigger one of the Q_i's
77     if Q_i_1 > Q_i_0
78         dec(j) = 1;
79     else
80         dec(j) = 0;
81     end
82
83     end % vertical step
84
85 end % iterations

```

Listing A.2: Probability-domain SPA decoder MATLAB code

A.3 Log-domain SPA decoder

```

1  % Iteration steps during the decoding process
2  % of a log-domain sum-product algorithm decoder
3  %
4  % in_msg:    incoming message
5  % H:         parity-check matrix
6  % iter:      number of iterations of the decoding process
7
8  % find variable and check nodes
9  [H_rows, H_cols] = find(H);
10
11 for n = 1:iter
12
13     % the LLR of variable-to-check messages is separated as:
14     a_ij = sign(L_v_ij);
15     b_ij = abs(L_v_ij);
16
17     % calculate phi function
18     for l = 1:length(H_rows)
19         phi_b_ij(H_rows(l), H_cols(l)) = ...
20             log((exp(b_ij(H_rows(l), H_cols(l))) + 1) / ...
21                 (exp(b_ij(H_rows(l), H_cols(l))) - 1));
22     end
23
24     % horizontal step (check nodes update)
25     for i = 1:M
26
27         % connected variable nodes
28         vns = find(H(i, :));
29
30         % collect information from all connected
31         % variable nodes
32         for k = 1:length(vns)
33
34             % sum of phi(b_i'j)

```

```

35         sum_phi_b_ij = sum(phi_b_ij(i, vns)) ...
36             - phi_b_ij(i, vns(k));
37
38         % in order to avoid division by zero (or extremely
39         % small number) set a minimum limit to the value
40         % of the sums
41         if sum_phi_b_ij < 1e-10
42             sum_phi_b_ij = 1e-10;
43         end
44
45         % calculate phi(sum(phi(b_ij)))
46         phi_sum_phi_b_ij = log((exp(sum_phi_b_ij) + 1) / ...
47             (exp(sum_phi_b_ij) - 1));
48
49         % calculate product of a_i'j
50         prod_a_ij = prod(a_ij(i, vns)) * a_ij(i, vns(k));
51
52         % response check-to-variable LLRs
53         L_u_ji(i, vns(k)) = prod_a_ij * phi_sum_phi_b_ij;
54
55     end
56
57 end % horizontal step
58
59 % vertical step (variable nodes update)
60 for j = 1:N
61
62     % connected check nodes
63     cns = find(H(:, j));
64
65     % response variable-to-check LLRs
66     for k = 1:length(cns)
67         L_v_ij(cns(k), j) = L_c_i(j) + sum(L_u_ji(cns, j)) ...
68             - L_u_ji(cns(k), j);
69     end
70
71     % calculate L_Q_i
72     L_Q_i = L_c_i(j) + sum(L_u_ji(cns, j));
73
74     % decide upon the sign of the LLR
75     if L_Q_i < 0
76         dec(j) = 1;
77     else
78         dec(j) = 0;
79     end
80
81 end % vertical step
82
83 end % iterations

```

Listing A.3: Log-domain SPA decoder MATLAB code

A.4 Simplified log-domain SPA decoder

```

1  % Iteration steps during the decoding process
2  % of a simplified log-domain sum-product algorithm decoder
3  %
4  % in_msg:    incoming message
5  % H:         parity-check matrix
6  % iter:      number of iterations of the decoding process
7
8  for n = 1:iter
9
10     % the LLR of variable-to-check messages is separated as:
11     a_ij = sign(L_v_ij);
12     b_ij = abs(L_v_ij);
13
14     % horizontal step (check nodes update)
15     for i = 1:M
16
17         % connected variable nodes
18         vns = find(H(i, :));
19
20         % collect information from all connected
21         % variable nodes
22         for k = 1:length(vns)
23
24             % get the minimum value of b_ij
25             min_b_ij = realmax;
26             for l = 1:length(vns)
27                 if l ~= k
28                     if b_ij(i, vns(l)) < min_b_ij
29                         min_b_ij = b_ij(i, vns(l));
30                     end
31                 end
32             end
33
34             % calculate product of a_i'j
35             prod_a_ij = prod(a_ij(i, vns)) * a_ij(i, vns(k));
36
37             % response check-to-variable LLRs
38             L_u_ji(i, vns(k)) = prod_a_ij * min_b_ij;
39
40         end
41
42     end % horizontal step
43
44     % vertical step (variable nodes update)
45     for j = 1:N
46
47         % connected check nodes
48         cns = find(H(:, j));
49
50         % response variable-to-check LLRs
51         for k = 1:length(cns)

```

```
52         L_v_ij(cns(k), j) = L_c_i(j) + sum(L_u_ji(cns, j)) ...
53         - L_u_ji(cns(k), j);
54     end
55
56     % calculate L_Q_i
57     L_Q_i = L_c_i(j) + sum(L_u_ji(cns, j));
58
59     % decide upon the sign of the LLR
60     if L_Q_i < 0
61         dec(j) = 1;
62     else
63         dec(j) = 0;
64     end
65
66     end % vertical step
67
68 end % iterations
```

Listing A.4: Simplified log-domain SPA decoder MATLAB code

Appendix B

Implementation source codes

Overview:

The second appendix includes the source codes which describe the transceiver's implementation on a Xilinx FPGA. The codes are written in VHDL and are separated into the two entities of the transceiver, the encoder and the decoder, as well as the block RAM module they both utilize. Each implementation is succeeded by its simulation testbench which has been used in order to verify correct operation.

B.1 Block RAM VHDL source code

```
1  -- Write-after-read block RAM implementation
2  -- synchronized to system clock and operated
3  -- by write enable signal.
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  use ieee.std_logic_arith.all;
8  use ieee.std_logic_unsigned.all;
9
10 entity sync_ram is
11     port (
12         clk          : in  std_logic;
13         we           : in  std_logic;
14         address      : in  std_logic_vector;
15         data_in      : in  std_logic_vector;
16         data_out     : out std_logic_vector);
17 end sync_ram;
18
19
20 architecture rtl of sync_ram is
21
22     type ram_type is array (0 to (2**address'length)-1) of
23         std_logic_vector(data_in'range);
24     signal ram : ram_type;
25     signal read_address : std_logic_vector(address'range);
26
27 begin
28     process(clk)
29     begin
```

```

31
32     if rising_edge(clk) then
33
34         if we = '1' then
35             ram(conv_integer(unsigned(address))) <= data_in;
36         end if;
37
38         read_address <= address;
39
40     end if;
41
42 end process;
43
44 data_out <= ram(conv_integer(unsigned(read_address)));
45
46 end architecture rtl;

```

Listing B.1: Synchronous write-after-read block RAM

B.2 Encoder VHDL source codes

B.2.1 Encoder implementation

```

1  -- Low-density parity-check codes encoder
2  -- for rate-1/2 applications.
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.std_logic_arith.all;
7  use ieee.std_logic_unsigned.all;
8
9
10 entity encoder is
11
12     generic (
13         M          : natural := 8;
14         N          : natural := 16);
15
16     port (
17         clk        : in  std_logic;
18         reset      : in  std_logic;
19         source     : in  std_logic_vector(M-1 downto 0);
20         enc        : out std_logic_vector(N-1 downto 0));
21
22 end encoder;
23
24
25 architecture behavioral of encoder is
26
27     -- block ram component description
28     component sync_ram
29     port (
30         clk        : in  std_logic;
31         we         : in  std_logic;
32         address    : in  std_logic_vector;
33         data_in    : in  std_logic_vector;
34         data_out   : out std_logic_vector);
35     end component;
36
37     -- signals declaration
38     signal state    : natural range 0 to 5;
39     signal cur_source : std_logic_vector(M-1 downto 0);
40     signal pre_source : std_logic_vector(M-1 downto 0);
41     signal tmpbits  : std_logic_vector(M-1 downto 0);
42     signal chkbits  : std_logic_vector(M-1 downto 0);

```



```

43
44  -- parity-check matrix ram signals
45  signal h_we      : std_logic;
46  signal h_add     : std_logic_vector(2 downto 0);
47  signal h_din     : std_logic_vector(0 to N-1);
48  signal h_dout    : std_logic_vector(0 to N-1);
49
50  -- inverse parity-check matrix ram signals
51  signal inv_a_we   : std_logic;
52  signal inv_a_add  : std_logic_vector(2 downto 0);
53  signal inv_a_din  : std_logic_vector(0 to M-1);
54  signal inv_a_dout : std_logic_vector(0 to M-1);
55
56  begin
57
58  -- component associations
59  H_ram      : sync_ram port map (clk, h_we, h_add, h_din, h_dout);
60  inv_A_ram  : sync_ram port map (clk, inv_a_we, inv_a_add, inv_a_din,
    inv_a_dout);
61
62  process(clk, reset)
63
64  -- variable declarations
65  variable temp      : natural;
66  variable temp_i    : natural;
67  variable temp_j    : natural;
68
69  begin
70
71  if clk'event and clk = '1' then
72
73  if reset = '1' then
74
75  -- predefined [8x16] parity-check matrix
76  -- H = [A B] and the inverse matrix of A
77  -- (inv_A) stored to memory
78  h_we <= '1';
79  h_add <= "000";
80  h_din <= "1000000010110000";
81
82  inv_a_we <= '1';
83  inv_a_add <= "000";
84  inv_a_din <= "10000000";
85
86  case h_add is
87
88  when "000" =>
89  h_add <= h_add + '1';
90  h_din <= "0100001001000100";
91  inv_a_add <= inv_a_add + '1';
92  inv_a_din <= "11010011";
93
94  when "001" =>
95  h_add <= h_add + '1';
96  h_din <= "0010000000010100";
97  inv_a_add <= inv_a_add + '1';
98  inv_a_din <= "00100000";
99
100  when "010" =>
101  h_add <= h_add + '1';
102  h_din <= "1001000000000001";
103  inv_a_add <= inv_a_add + '1';
104  inv_a_din <= "10010000";
105
106  when "011" =>
107  h_add <= h_add + '1';
108  h_din <= "0000100011001000";

```

```

109         inv_a_add <= inv_a_add + '1';
110         inv_a_din <= "00001000";
111
112     when "100" =>
113         h_add <= h_add + '1';
114         h_din <= "0010110000100011";
115         inv_a_add <= inv_a_add + '1';
116         inv_a_din <= "00101100";
117
118     when "101" =>
119         h_add <= h_add + '1';
120         h_din <= "0001001110000010";
121         inv_a_add <= inv_a_add + '1';
122         inv_a_din <= "10010011";
123
124     when "110" =>
125         h_add <= h_add + '1';
126         h_din <= "00000000100001101";
127         inv_a_add <= inv_a_add + '1';
128         inv_a_din <= "00000001";
129
130     when "111" =>
131         -- initialization
132         state <= 0;
133
134     when others =>
135         null;
136
137 end case;
138
139
140 else
141
142     case state is
143
144     when 0 =>
145
146         -- main process has started
147         -- stopping any writing to parity-check matrix
148         -- and inverse A initialized by reset button
149         h_we <= '0';
150         inv_a_we <= '0';
151
152         if source /= pre_source then
153
154             cur_source <= source;
155
156             -- read first rows of H and inv_A
157             h_add <= "000";
158             inv_a_add <= "000";
159
160             -- continue to next state
161             temp := 0;
162             temp_i := 0;
163             temp_j := 0;
164             state <= 1;
165
166         else
167
168             state <= 0;
169
170         end if;
171
172
173     when 1 =>
174
175         state <= 2;

```

```

176
177
178      -- start of encoding process:
179      -- chkbits = inv_A * B * source
180      when 2 =>
181
182          -- tmpbits = B * source
183          if temp_i < M then
184
185              if temp_j < M then
186
187                  -- calculate [H(i)(N-M+j) * source(M-1-j)] by
188                  -- replacing multiplication with signal checks
189                  if h_dout(N-M+temp_j) = '1' and cur_source(M-1-temp_j) =
190                      '1' then
191                      temp := temp + 1;
192                  end if;
193
194                  temp_j := temp_j + 1;
195
196              else
197
198                  -- calculate temporary bits
199                  if temp mod 2 = 1 then
200                      tmpbits(M-1-temp_i) <= '1';
201                  else
202                      tmpbits(M-1-temp_i) <= '0';
203                  end if;
204
205                  -- repeat for next row of H
206                  temp := 0;
207                  temp_i := temp_i + 1;
208                  temp_j := 0;
209                  h_add <= h_add + '1';
210                  state <= 1;
211
212              end if;
213
214          else
215
216              -- continue to next state
217              temp := 0;
218              temp_i := 0;
219              temp_j := 0;
220              state <= 3;
221
222          end if;
223
224      when 3 =>
225
226          -- chkbits = inv_A * tmpbits
227          if temp_i < M then
228
229              if temp_j < M then
230
231                  -- calculate [inv_A(i)(j) * tmpbits(M-1-j)] by
232                  -- replacing multiplication with signal checks
233                  if inv_a_dout(temp_j) = '1' and tmpbits(M-1-temp_j) = '1'
234                      then
235                      temp := temp + 1;
236                  end if;
237
238                  temp_j := temp_j + 1;
239
240              else

```

```

241         -- calculate check bits
242         if temp mod 2 = 1 then
243             chkbits(M-1-temp_i) <= '1';
244         else
245             chkbits(M-1-temp_i) <= '0';
246         end if;
247
248         -- repeat for next row of H
249         temp := 0;
250         temp_i := temp_i + 1;
251         temp_j := 0;
252         inv_a_add <= inv_a_add + '1';
253         state <= 4;
254
255     end if;
256
257     else
258
259         -- check bits have been calculated;
260         -- continue to final state
261         state <= 5;
262
263     end if;
264
265
266     when 4 =>
267
268         state <= 3;
269
270
271     when 5 =>
272
273         -- output encoded data: enc = [chkbits source]
274         enc(N-1 downto M) <= chkbits;
275         enc(M-1 downto 0) <= cur_source;
276
277         -- restart
278         state <= 0;
279         pre_source <= cur_source;
280
281
282     when others =>
283
284         null;
285
286     end case;
287
288     end if; -- reset = '1'
289
290     end if; -- clk'event & clk = '1'
291
292 end process;
293
294 end behavioral;

```

Listing B.2: LDPC codes encoder for rate- $\frac{1}{2}$ applications

B.2.2 Encoder simulation testbench

```

1  -- Simulation testbench for the
2  -- low-density parity-check codes encoder.
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.std_logic_arith.all;
7  use ieee.std_logic_unsigned.all;
8  use std.textio.all;

```

```

9
10 entity enc_testbench is
11     generic (
12         M          : integer := 8;
13         N          : integer := 16);
14 end enc_testbench;
15
16
17 architecture behavior of enc_testbench is
18
19     file source_vectors : text open read_mode is "source_vectors.txt";
20     type array_of_bit_N is array(N-1 downto 0) of bit;
21     type array_of_bit_M is array(M-1 downto 0) of bit;
22
23     -- component declaration for the Unit Under Test (UUT)
24     component encoder
25     port (
26         clk      : in  std_logic;
27         reset    : in  std_logic;
28         source   : in  std_logic_vector(M-1 downto 0);
29         enc      : out std_logic_vector(N-1 downto 0);
30     end component;
31
32     -- inputs
33     signal clk      : std_logic := '0';
34     signal reset    : std_logic := '1';
35     signal source   : std_logic_vector(M-1 downto 0);
36
37     -- outputs
38     signal enc      : std_logic_vector(N-1 downto 0);
39
40     -- clock period definitions
41     constant clk_period : time := 1 ns;
42
43     -- verification signal
44     signal enc_check    : std_logic_vector(N-1 downto 0);
45
46 begin
47
48     -- instantiate the Unit Under Test (UUT)
49     uut: encoder port map (
50         clk      => clk,
51         reset    => reset,
52         source   => source,
53         enc      => enc);
54
55
56     -- clock process
57     clk_process: process
58
59     begin
60         clk <= '0';
61         wait for clk_period/2;
62         clk <= '1';
63         wait for clk_period/2;
64     end process;
65
66
67     -- stimulus process
68     stim_proc: process
69
70         -- I/O variable declarations
71         variable source_buf : line;
72         variable enc_buf    : line;
73         variable source_var : array_of_bit_M;
74         variable enc_var    : array_of_bit_N;
75

```

```

76      -- error check variable declarations
77      variable tot_errors : integer;
78      variable tot_bits   : integer;
79
80      -- wait time between consecutive inputs
81      -- variable declaration
82      variable wait_time   : integer;
83
84  begin
85
86      -- reset the system
87      wait for clk_period * 10;
88      reset <= '0';
89
90      -- start decoding simulation
91      report "Starting simulation";
92      tot_errors := 0;
93      tot_bits := 0;
94
95      -- set wait time
96      wait_time := 230;
97
98      -- read data from file
99      while not endfile(source_vectors) loop
100
101          -- read source vectors to be encoded
102          readline(source_vectors, source_buf);
103
104          for i in M-1 downto 0 loop
105              read(source_buf, source_var(i));
106          end loop;
107
108          -- convert input message
109          -- to std_logic format
110          for i in M-1 downto 0 loop
111              if source_var(i) = '1' then
112                  source(i) <= '1';
113              else
114                  source(i) <= '0';
115              end if;
116          end loop;
117
118          -- read expected encoded output
119          readline(source_vectors, enc_buf);
120
121          for i in N-1 downto 0 loop
122              read(enc_buf, enc_var(i));
123          end loop;
124
125          -- convert expected encoded output
126          -- to std_logic format
127          for i in N-1 downto 0 loop
128              if enc_var(i) = '1' then
129                  enc_check(i) <= '1';
130              else
131                  enc_check(i) <= '0';
132              end if;
133          end loop;
134
135          -- wait until decoding has finished
136          wait for wait_time * 1ns;
137
138          -- check output with the expected one
139          -- and calculate erroneous bits
140          for i in N-1 downto 0 loop
141              if enc(i) /= enc_check(i) then
142                  tot_errors := tot_errors + 1;

```

```

143         end if;
144     end loop;
145
146     -- increase total number of decoded bits
147     tot_bits := tot_bits + N;
148
149     -- wait before sending next input message
150     wait for clk_period * 2;
151
152 end loop;
153
154 -- when all messages have been encoded
155 -- the simulation prints the total number
156 -- of erroneous encoded bits to the console
157 report "Err. bits: " & integer'image(tot_errors) & " / " &
    integer'image(tot_bits);
158 report "Simulation ended.";
159
160 wait;
161
162 end process;
163
164 end;
```

Listing B.3: Simulation testbench for the LDPC codes encoder

B.3 Decoder VHDL source codes

B.3.1 Decoder implementation

```

1  -- Low-density parity-check codes simplified
2  -- log-domain sum-product algorithm decoder
3  -- for rate-1/2 applications including early
4  -- termination scheme and input quantization
5  -- to signed fixed point format.
6
7  library ieee;
8  use ieee.std_logic_1164.all;
9  use ieee.std_logic_arith.all;
10 use ieee.std_logic_unsigned.all;
11
12 -- input message type description
13 package dec_pkg is
14     type input_array is array(0 to 15) of std_logic_vector(7 downto 0);
15 end dec_pkg;
16
17 library ieee;
18 use ieee.std_logic_1164.all;
19 use ieee.std_logic_arith.all;
20 use ieee.std_logic_unsigned.all;
21 use work.dec_pkg.all;
22
23 entity decoder is
24
25     generic (
26         M           : natural := 8;
27         N           : natural := 16;
28         add_length  : natural := 7;
29         sfixed_length : natural := 8);
30
31     port (
32         clk       : in std_logic;
33         reset     : in std_logic;
34         source    : in input_array;
35         iter      : in natural range 1 to 10;
```

```

36         dec          : out std_logic_vector(M-1 downto 0));
37
38 end decoder;
39
40
41 architecture behavioral of decoder is
42
43     -- block ram component description
44     component sync_ram
45     port (
46         clk          : in  std_logic;
47         we            : in  std_logic;
48         address       : in  std_logic_vector;
49         data_in       : in  std_logic_vector;
50         data_out      : out std_logic_vector);
51     end component;
52
53     type temp_cns_array is array(0 to N-1) of natural;
54
55     -- signals declaration
56     signal state       : natural range 0 to 26;
57     signal cur_iter    : natural; -- range 0 to 11;
58     signal cur_source  : input_array;
59     signal pre_source  : input_array;
60     signal L_c_i       : input_array;
61     signal dec_temp    : std_logic_vector(0 to N-1);
62     signal dec_pre     : std_logic_vector(0 to N-1);
63
64     -- parity-check matrix ram signals
65     signal h_we        : std_logic;
66     signal h_add       : std_logic_vector(2 downto 0);
67     signal h_din       : std_logic_vector(0 to N-1);
68     signal h_dout      : std_logic_vector(0 to N-1);
69
70     -- a_ij matrix ram signals
71     signal a_ij_we     : std_logic;
72     signal a_ij_add    : std_logic_vector(add_length-1 downto 0);
73     signal a_ij_din    : std_logic_vector(0 to 0); -- bit
74     signal a_ij_dout   : std_logic_vector(0 to 0); -- bit
75
76     -- b_ij matrix ram signals
77     signal b_ij_we     : std_logic;
78     signal b_ij_add    : std_logic_vector(add_length-1 downto 0);
79     signal b_ij_din    : std_logic_vector(sfixed_length-1 downto 0);
80     signal b_ij_dout   : std_logic_vector(sfixed_length-1 downto 0);
81
82     -- vns matrix ram signals
83     signal vns_we      : std_logic;
84     signal vns_add     : std_logic_vector(add_length-1 downto 0);
85     signal vns_din     : std_logic_vector(0 to N-1);
86     signal vns_dout    : std_logic_vector(0 to N-1);
87
88     -- cns matrix ram signals
89     signal cns_we      : std_logic;
90     signal cns_add     : std_logic_vector(add_length-1 downto 0);
91     signal cns_din     : std_logic_vector(0 to N-1);
92     signal cns_dout    : std_logic_vector(0 to N-1);
93
94     -- L_v_ij matrix ram signals
95     signal L_v_ij_we   : std_logic;
96     signal L_v_ij_add  : std_logic_vector(add_length-1 downto 0);
97     signal L_v_ij_din  : std_logic_vector(sfixed_length-1 downto 0);
98     signal L_v_ij_dout : std_logic_vector(sfixed_length-1 downto 0);
99
100    -- L_u_ji matrix ram signals
101    signal L_u_ji_we    : std_logic;
102    signal L_u_ji_add   : std_logic_vector(add_length-1 downto 0);

```



```

103     signal    L_u_ji_din      : std_logic_vector(sfixed_length-1 downto 0);
104     signal    L_u_ji_dout     : std_logic_vector(sfixed_length-1 downto 0);
105
106 begin
107
108     -- component associations
109     H_ram      : sync_ram port map (clk, h_we, h_add, h_din, h_dout);
110     vns_ram    : sync_ram port map (clk, vns_we, vns_add, vns_din, vns_dout);
111     cns_ram    : sync_ram port map (clk, cns_we, cns_add, cns_din, cns_dout);
112     a_ij_ram   : sync_ram port map (clk, a_ij_we, a_ij_add, a_ij_din, a_ij_dout);
113     b_ij_ram   : sync_ram port map (clk, b_ij_we, b_ij_add, b_ij_din, b_ij_dout);
114     L_v_ij_ram : sync_ram port map (clk, L_v_ij_we, L_v_ij_add, L_v_ij_din,
115                                     L_v_ij_dout);
116     L_u_ji_ram : sync_ram port map (clk, L_u_ji_we, L_u_ji_add, L_u_ji_din,
117                                     L_u_ji_dout);
118
119     process(clk, reset)
120
121         -- variable declarations
122         variable temp_i      : natural;
123         variable temp_j      : natural;
124         variable temp_k      : natural;
125         variable temp_l      : natural;
126
127         variable temp_cns    : temp_cns_array;
128         variable temp_vns    : natural;
129         variable temp_cns_i  : natural;
130
131         variable vns_temp    : natural;
132         variable cns_temp    : natural;
133
134         variable cur_min     : std_logic_vector(M-1 downto 0);
135         variable neg_a_ij    : natural;
136         variable sum_L_u_ji  : std_logic_vector(M-1 downto 0);
137         variable L_Q_i       : std_logic_vector(M-1 downto 0);
138
139     begin
140
141         if clk'event and clk = '1' then
142
143             if reset = '1' then
144
145                 -- predefined [8x16] parity-check matrix H to memory
146                 h_we <= '1';
147                 h_add <= "000";
148                 h_din <= "1000000010110000";
149
150                 case h_add is
151
152                     when "000" =>
153                         h_add <= h_add + '1';
154                         h_din <= "0100001001000100";
155
156                     when "001" =>
157                         h_add <= h_add + '1';
158                         h_din <= "0010000000010100";
159
160                     when "010" =>
161                         h_add <= h_add + '1';
162                         h_din <= "1001000000000001";
163
164                     when "011" =>
165                         h_add <= h_add + '1';
166                         h_din <= "0000100011001000";
167
168                     when "100" =>
169                         h_add <= h_add + '1';

```

```

168         h_din <= "00101110000100011";
169
170     when "101" =>
171         h_add <= h_add + '1';
172         h_din <= "0001001110000010";
173
174     when "110" =>
175         h_add <= h_add + '1';
176         h_din <= "0000000100001101";
177
178     when "111" =>
179         -- initialization
180         state <= 0;
181
182     when others =>
183         null;
184
185 end case;
186
187
188 else
189
190     case state is
191
192     when 0 =>
193
194         -- main process has started
195         -- stopping any writing to parity-check matrix
196         -- initialized by reset button
197         h_we <= '0';
198
199         if source /= pre_source then
200
201             cur_source <= source;
202             cur_iter <= 0;
203
204             vns_we <= '1';
205             vns_add <= conv_std_logic_vector(0, add_length);
206             vns_din <= conv_std_logic_vector(0, N);
207
208             cns_we <= '1';
209             cns_add <= conv_std_logic_vector(0, add_length);
210             cns_din <= conv_std_logic_vector(0, N);
211
212             L_v_ij_we <= '1';
213             L_v_ij_add <= conv_std_logic_vector(0, add_length);
214             L_v_ij_din <= conv_std_logic_vector(0, sfixed_length);
215
216             -- read first row of H
217             h_add <= "000";
218             h_we <= '0';
219
220             state <= 1;
221
222         else
223
224             state <= 0;
225
226         end if;
227
228
229     -- start of decoding process
230     when 1 =>
231
232         -- initializations of the decoding process
233         for i in 0 to N-1 loop
234             L_c_i(i) <= not(cur_source(i)) + '1';

```

```

235         temp_cns(i) := 0;
236     end loop;
237
238     vns_we <= '0';
239     cns_we <= '0';
240
241     -- continue to next state
242     temp_i := 0;
243     temp_j := 0;
244     temp_cns_i := 0;
245     temp_vns := 0;
246     state <= 2;
247
248
249     when 2 =>
250
251         -- initially associate the L_c_i matrix
252         -- with non-zero elements of H and find
253         -- connected check and variable nodes
254         if temp_j < N then
255
256             if H_dout(temp_j) /= '0' then
257
258                 L_v_ij_we <= '1';
259                 L_v_ij_add <= conv_std_logic_vector(temp_i*N + temp_j,
260                     add_length);
261                 L_v_ij_din <= L_c_i(temp_j);
262
263                 vns_we <= '1';
264                 vns_add <= conv_std_logic_vector(temp_i*N + temp_vns,
265                     add_length);
266                 vns_din <= conv_std_logic_vector(temp_j, N);
267                 temp_vns := temp_vns + 1;
268
269                 cns_we <= '1';
270                 cns_add <= conv_std_logic_vector(temp_j*M +
271                     temp_cns(temp_j), add_length);
272                 cns_din <= conv_std_logic_vector(temp_i, N);
273                 temp_cns(temp_j) := temp_cns(temp_j) + 1;
274
275             else
276
277                 L_v_ij_we <= '1';
278                 L_v_ij_add <= conv_std_logic_vector(temp_i*N + temp_j,
279                     add_length);
280                 L_v_ij_din <= conv_std_logic_vector(0, sfixed_length);
281
282             end if;
283
284             temp_j := temp_j + 1;
285
286             if temp_j = N-1 then
287                 h_add <= h_add + '1';
288             end if;
289
290             state <= 2;
291
292         else
293
294             -- connected variable nodes
295             vns_add <= conv_std_logic_vector(temp_i*N + N-1, add_length);
296             vns_din <= conv_std_logic_vector(temp_vns - 1, N);
297
298             -- continue to next row
299             temp_i := temp_i + 1;
300
301             -- repeat if necessary

```

```

298         if temp_i < M then
299
300             -- repeat for the next row of H
301             temp_j := 0;
302             temp_vns := 0;
303             state <= 2;
304
305         else
306
307             -- proceed to next state
308             L_v_ij_we <= '0';
309             L_v_ij_add <= conv_std_logic_vector(0, add_length);
310             state <= 3;
311
312         end if;
313
314     end if;
315
316 when 3 =>
317
318     if temp_cns_i < N then
319
320         -- connected check nodes
321         cns_add <= conv_std_logic_vector(temp_cns_i*M + M-1,
322             add_length);
323         cns_din <= conv_std_logic_vector(temp_cns(temp_cns_i) - 1,
324             N);
325
326         -- repeat for the next column of cns
327         temp_cns_i := temp_cns_i + 1;
328         state <= 3;
329
330     else
331
332         -- proceed to next state
333         temp_i := 0;
334         temp_j := 0;
335         vns_we <= '0';
336         cns_we <= '0';
337         L_v_ij_add <= conv_std_logic_vector(1, add_length);
338         -- prefetch number of connected check nodes for the
339         -- first variable node [vn matrix (0)(N-1)]
340         vns_add <= conv_std_logic_vector(N-1, add_length);
341         cur_iter <= 1;
342         state <= 4;
343
344     end if;
345
346 when 4 =>
347
348     -- repeat the process for the number of iterations
349     if cur_iter < iter+1 then
350
351         L_v_ij_add <= L_v_ij_add + '1';
352
353         -- the LLR of variable-to-check messages is separated:
354         -- a_ij = 1 if sign(L_v_ij) = -1
355         -- b_ij = abs(L_v_ij)
356         if temp_i < M then
357
358             a_ij_we <= '1';
359             a_ij_add <= conv_std_logic_vector(temp_i*N + temp_j,
360                 add_length);
361             b_ij_we <= '1';
362             b_ij_add <= conv_std_logic_vector(temp_i*N + temp_j,

```

```

362         add_length);
363     if L_v_ij_dout(M-1) = '0' then
364
365         -- positive number or zero
366         a_ij_din <= "0";
367         b_ij_din <= L_v_ij_dout;
368
369     else
370
371         -- negative number
372         a_ij_din <= "1";
373         b_ij_din <= (not L_v_ij_dout) + '1';
374
375     end if;
376
377     temp_j := temp_j + 1;
378
379     if temp_j = N then
380
381         -- repeat for the next row
382         temp_j := 0;
383         temp_i := temp_i + 1;
384         state <= 4;
385
386     end if;
387
388 else
389
390     -- proceed to next state
391     temp_i := 0;
392     temp_j := 0;
393     temp_k := 0;
394     temp_l := 0;
395     a_ij_we <= '0';
396     b_ij_we <= '0';
397     L_v_ij_add <= conv_std_logic_vector(0, add_length);
398
399     -- initially maximum possible number
400     cur_min := "01111111";
401
402     -- sign of a_ij's multiplication
403     neg_a_ij := 0;
404
405     state <= 5;
406
407 end if;
408
409 else
410
411     -- done with decoding;
412     -- proceed to next state
413     a_ij_we <= '0';
414     b_ij_we <= '0';
415     L_v_ij_add <= conv_std_logic_vector(0, add_length);
416     state <= 26;
417
418 end if;
419
420
421 when 5 =>
422
423     -- horizontal step (check nodes update)
424     if temp_i < M then
425
426         vns_add <= conv_std_logic_vector(temp_i*N + temp_k,
427             add_length);

```

```

427         state <= 6;
428
429     else
430
431         temp_j := 0;
432         temp_k := 0;
433         temp_l := 0;
434         sum_L_u_ji := conv_std_logic_vector(0, sfixed_length);
435         -- prefetch number of connected variable nodes for the
436         -- first check node [cn matrix (0)(M-1)]
437         cns_add <= conv_std_logic_vector(M-1, add_length);
438         state <= 17;
439
440     end if;
441
442     L_u_ji_we <= '0';
443
444
445     when 6 =>
446
447         -- number of connected variable nodes
448         -- to check node i
449         vns_temp := conv_integer(vns_dout);
450
451         -- collect information from all connected
452         -- variable nodes
453         if temp_j <= vns_temp then
454
455             if temp_k <= vns_temp then
456                 state <= 7;
457             else
458                 if temp_l <= vns_temp then
459                     vns_add <= conv_std_logic_vector(temp_i*N + temp_l,
460                                                         add_length);
461                 else
462                     vns_add <= conv_std_logic_vector(temp_i*N + temp_j,
463                                                         add_length);
464                 end if;
465
466                 state <= 10;
467             end if;
468
469             else
470
471                 temp_i := temp_i + 1;
472                 temp_j := 0;
473
474                 if temp_i < M then
475                     vns_add <= conv_std_logic_vector(temp_i*N + N-1,
476                                                         add_length);
477                 else
478                     -- ready for next iteration
479                     vns_add <= conv_std_logic_vector(N-1, add_length);
480                 end if;
481
482                 state <= 5;
483
484             end if;
485
486         when 7 =>
487
488             b_ij_add <= conv_std_logic_vector(temp_i*N +
489                                                 conv_integer(vns_dout), add_length);
490             state <= 8;
491
492

```

```

490         when 8 =>
491             state <= 9;
492
493
494
495         when 9 =>
496
497             -- get the minimum value of b_ij
498             if temp_k /= temp_j then
499                 if b_ij_dout < cur_min then
500                     cur_min := b_ij_dout;
501                 end if;
502             end if;
503
504             temp_k := temp_k + 1;
505
506             vns_add <= conv_std_logic_vector(temp_i*N + N-1, add_length);
507             state <= 5;
508
509
510         when 10 =>
511
512             -- calculate product of a_i'j which can be either 1
513             -- or -1, therefore only counting the number of -1s
514             -- is necessary to calculate the value of the
515             -- product
516
517             if temp_l <= vns_temp then
518                 state <= 11;
519             else
520                 state <= 14;
521             end if;
522
523
524         when 11 =>
525
526             a_ij_add <= conv_std_logic_vector(temp_i*N +
527                 conv_integer(vns_dout), add_length);
528             state <= 12;
529
530
531         when 12 =>
532
533             state <= 13;
534
535
536         when 13 =>
537
538             if a_ij_dout = "1" then
539                 neg_a_ij := neg_a_ij + 1;
540             end if;
541
542             temp_l := temp_l + 1;
543             vns_add <= conv_std_logic_vector(temp_i*N + N-1, add_length);
544             state <= 5;
545
546
547         when 14 =>
548
549             a_ij_add <= conv_std_logic_vector(temp_i*N +
550                 conv_integer(vns_dout), add_length);
551             state <= 15;
552
553
554         when 15 =>
555
556             L_u_ji_add <= conv_std_logic_vector(temp_i*N +

```

```

555         conv_integer(vns_dout), add_length);
556     state <= 16;
557
558     when 16 =>
559
560         if a_ij_dout = '1' then
561             neg_a_ij := neg_a_ij + 1;
562         end if;
563
564         -- response check-to-variable LLRs
565         if neg_a_ij mod 2 = 0 then
566
567             -- positive product
568             L_u_ji_din <= cur_min;
569
570         else
571
572             -- negative product
573             L_u_ji_din <= (not cur_min) + '1';
574
575         end if;
576
577         L_u_ji_we <= '1';
578         temp_k := 0;
579         temp_l := 0;
580         temp_j := temp_j + 1;
581
582         -- cur_min = maximum possible number = "011...11";
583         cur_min := "01111111";
584
585         neg_a_ij := 0;
586         vns_add <= conv_std_logic_vector(temp_i*N + N-1, add_length);
587         state <= 5;
588
589
590     when 17 =>
591
592         L_v_ij_we <= '0';
593
594         -- vertical step (variable nodes update)
595         if temp_j < N then
596
597             cns_add <= conv_std_logic_vector(temp_j*M + temp_k,
598                 add_length);
599             state <= 18;
600
601         else
602
603             -- includes early termination check:
604             -- if the decoded output of this iteration equals
605             -- the output of the previous iteration then stop
606             if cur_iter < iter and dec_pre /= dec_temp then
607                 cur_iter <= cur_iter + 1;
608                 temp_i := 0;
609                 temp_j := 0;
610                 L_v_ij_add <= L_v_ij_add + '1';
611                 dec_pre <= dec_temp;
612                 state <= 4;
613             else
614                 state <= 26;
615             end if;
616
617         end if;
618
619     when 18 =>

```



```

620
621      -- number of connected check nodes
622      -- to variable node i
623      cns_temp := conv_integer(cns_dout);
624
625      -- response variable-to-check LLRs
626      if temp_k <= cns_temp then
627          state <= 19;
628      else
629          cns_add <= conv_std_logic_vector(temp_j*M + temp_l,
630              add_length);
631          state <= 22;
632      end if;
633
634      when 19 =>
635
636          L_u_ji_add <= conv_std_logic_vector((conv_integer(cns_dout))*N
637              + temp_j, add_length);
638          state <= 20;
639
640      when 20 =>
641
642          state <= 21;
643
644      when 21 =>
645
646          sum_L_u_ji := sum_L_u_ji + L_u_ji_dout;
647          temp_k := temp_k + 1;
648          cns_add <= conv_std_logic_vector(temp_j*M + M-1, add_length);
649          state <= 17;
650
651      when 22 =>
652
653          state <= 23;
654
655      when 23 =>
656
657          L_Q_i := L_c_i(temp_j) + sum_L_u_ji;
658          L_v_ij_add <= conv_std_logic_vector((conv_integer(cns_dout)*N +
659              temp_j), add_length);
660          L_u_ji_add <= conv_std_logic_vector((conv_integer(cns_dout)*N +
661              temp_j), add_length);
662          state <= 24;
663
664      when 24 =>
665
666          if temp_l <= cns_temp then
667
668              state <= 25;
669
670          else
671
672              -- decide upon the sign of the LLR
673              if L_Q_i(M-1) = '1' then
674                  dec_temp(temp_j) <= '1'; -- negative
675              else
676                  dec_temp(temp_j) <= '0'; -- positive
677              end if;
678
679          temp_j := temp_j + 1;
680          temp_k := 0;

```

```

683         temp_l := 0;
684         sum_L_u_ji := conv_std_logic_vector(0, M);
685         cns_add <= conv_std_logic_vector(temp_j*M + M-1, add_length);
686         L_v_ij_add <= conv_std_logic_vector(0, add_length);
687         state <= 17;
688
689     end if;
690
691
692     when 25 =>
693
694         if temp_j >= N then
695             L_v_ij_add <= conv_std_logic_vector(0, add_length);
696         end if;
697
698         L_v_ij_we <= '1';
699         L_v_ij_din <= L_Q_i + ((not L_u_ji_dout) + '1');
700         temp_l := temp_l + 1;
701         cns_add <= conv_std_logic_vector(temp_j*M + M-1, add_length);
702         state <= 17;
703
704
705     when 26 =>
706
707         -- output decoded bits (in descending order)
708         dec(M-1 downto 0) <= dec_temp(M to N-1);
709
710         -- restart
711         pre_source <= cur_source;
712         state <= 0;
713
714
715     when others =>
716
717         null;
718
719     end case;
720
721     end if; -- reset = '1'
722
723     end if; -- clk'event & clk = '1'
724
725 end process;
726
727 end behavioral;

```

Listing B.4: LDPC codes simplified log-domain decoder for rate- $\frac{1}{2}$ applications

B.3.2 Decoder simulation testbench

```

1  -- Simulation testbench for the
2  -- low-density parity-check codes simplified
3  -- log-domain sum-product algorithm decoder
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  use ieee.std_logic_arith.all;
8  use ieee.std_logic_unsigned.all;
9
10 -- signed fixed point format conversion
11 -- library
12 library ieee_proposed;
13 use ieee_proposed.fixed_pkg.all;
14
15 -- noisy version of input source description
16 package dec_tb_pkg is
17     type array_of_real_in is array(0 to 15) of real;

```

```

18 end dec_tb_pkg;
19
20
21 library ieee;
22 use ieee.std_logic_1164.all;
23 use ieee.std_logic_arith.all;
24 use ieee.std_logic_unsigned.all;
25 use work.dec_pkg.all;
26 use work.dec_tb_pkg.all;
27 use std.textio.all;
28
29 library ieee_proposed;
30 use ieee_proposed.fixed_pkg.all;
31
32 entity dec_testbench is
33     generic (
34         M          : integer := 8;
35         N          : integer := 16);
36 end dec_testbench;
37
38
39 architecture behavior of dec_testbench is
40
41     file source_vectors : text open read_mode is "source_vectors.txt";
42     type array_of_bit is array(M-1 downto 0) of bit;
43     type input_sfixed is array(0 to N-1) of sfixed(3 downto -4);
44
45     -- component declaration for the Unit Under Test (UUT)
46     component decoder
47     port (
48         clk      : in  std_logic;
49         reset    : in  std_logic;
50         source   : in  input_array;
51         iter     : in  natural range 0 to 10;
52         dec      : out std_logic_vector(M-1 downto 0));
53     end component;
54
55     -- inputs
56     signal clk      : std_logic := '0';
57     signal reset    : std_logic := '1';
58     signal source   : input_array;
59     signal iter     : natural range 1 to 10 := 7;
60
61     -- outputs
62     signal dec      : std_logic_vector(M-1 downto 0);
63
64     -- clock period definitions
65     constant clk_period : time := 1 ns;
66
67     -- verification signal
68     signal dec_check    : std_logic_vector(M-1 downto 0);
69
70 begin
71
72     -- instantiate the Unit Under Test (UUT)
73     uut: decoder port map (
74         clk      => clk,
75         reset    => reset,
76         source   => source,
77         iter     => iter,
78         dec      => dec);
79
80
81     -- clock process
82     clk_process: process
83
84     begin

```

```

85     clk <= '0';
86     wait for clk_period/2;
87     clk <= '1';
88     wait for clk_period/2;
89 end process;
90
91
92 -- stimulus process
93 stim_proc: process
94
95     -- I/O variable declarations
96     variable source_buf      : line;
97     variable dec_buf         : line;
98     variable source_sfixed   : input_sfixed;
99     variable source_var      : array_of_real_in;
100    variable dec_var          : array_of_bit;
101
102    -- BER variable declarations
103    variable tot_errors       : integer;
104    variable tot_bits         : integer;
105    variable ber              : real;
106
107    -- wait time between consecutive inputs
108    -- variable declaration
109    variable wait_time        : integer;
110
111 begin
112
113     -- reset the system
114     wait for clk_period * 10;
115     reset <= '0';
116
117     -- start decoding simulation
118     report "Starting simulation";
119     tot_errors := 0;
120     tot_bits := 0;
121
122     -- set wait time
123     wait_time := 17000;
124
125     -- read data from file
126     while not endfile(source_vectors) loop
127
128         -- read source vectors to be decoded
129         readline(source_vectors, source_buf);
130
131         for i in 0 to N-1 loop
132             read(source_buf, source_var(i));
133         end loop;
134
135         -- convert real to signed fixed point format
136         for i in 0 to N-1 loop
137             source_sfixed(i) := to_sfixed(source_var(i), source_sfixed(i));
138         end loop;
139
140         -- convert sfixed to std_logic_vector
141         for i in 0 to N-1 loop
142             source(i) <= to_slv(source_sfixed(i));
143         end loop;
144
145         -- read expected decoded output
146         readline(source_vectors, dec_buf);
147
148         for i in M-1 downto 0 loop
149             read(dec_buf, dec_var(i));
150         end loop;
151

```

```

152      -- convert expected decoded output
153      -- to std_logic format
154      for i in M-1 downto 0 loop
155          if dec_var(i) = '1' then
156              dec_check(i) <= '1';
157          else
158              dec_check(i) <= '0';
159          end if;
160      end loop;
161
162      -- wait until decoding has finished
163      wait for wait_time * 1ns;
164
165      -- check output with the expected one
166      -- and calculate erroneous bits
167      for i in M-1 downto 0 loop
168          if dec(i) /= dec_check(i) then
169              tot_errors := tot_errors + 1;
170          end if;
171      end loop;
172
173      -- increase total number of decoded bits
174      tot_bits := tot_bits + M;
175
176      -- wait before sending next input message
177      wait for clk_period * 2;
178
179  end loop;
180
181      -- when all messages have been decoded
182      -- the simulation calculates the stream's
183      -- bit error rate (BER)
184      ber := real(tot_errors)/real(tot_bits);
185
186      -- and prints the result to the console
187      report "Err. bits: " & integer'image(tot_errors) & " / " &
          integer'image(tot_bits);
188      report "BER: " & real'image(ber);
189      report "Simulation ended";
190
191      wait;
192  end process;
193
194  end;

```

Listing B.5: Simulation testbench for the LDPC codes simplified log-domain decoder

Bibliography

- [1] C. E. Shannon, “Communication in the presence of noise,” *Proc. Institute of Radio Engineers*, vol. 37, pp. 10–21, Jan. 1949, reprinted Feb. 1998. (available at <http://www.stanford.edu/class/ee104/shannonpaper.pdf>).
- [2] N. Benvenuto and G. Cherubini, *Algorithms for Communications Systems and Their Applications*, p. 508. John Wiley and Sons, 2002.
- [3] E. Galois, “Sur la théorie des nombres,” *Bulletin des Sciences mathématiques XIII*, vol. 428, 1830.
- [4] R. Tanner, “A recursive approach to low complexity codes,” *IEEE Transactions on Information Theory*, vol. 27, pp. 533–547, Sept. 1981.
- [5] J. Pearl, “Reverend Bayes on inference engines: A distributed hierarchical approach,” vol. AAAI-82 of *Proceedings of the Second National Conference on Artificial Intelligence*, (Menlo Park, California), pp. 133–136, AAAI Press, 1982. (available at <https://www.aaai.org/Papers/AAAI/1982/AAAI82-032.pdf>).
- [6] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, Massachusetts: M.I.T. Press, 1963. (available at <http://www.inference.phy.cam.ac.uk/mackay/gallager/papers/ldpc.pdf>).
- [7] “IEEE 802.3 10GBase-T Task Force,” Nov. 2004. (available at <http://www.ieee802.org/3/10GBT/public/nov104/ungerboeck-1-1104.pdf>).
- [8] “IEEE 802.16e Standard,” Feb. 2006. (available at <http://standards.ieee.org/getieee802/download/802.16e-2005.pdf>).
- [9] “Draft European Telecommunication Standards Institute EN 302 307 V1.1.1,” 2004-06.
- [10] N. Wiberg, *Codes and decoding on general graphs*. PhD thesis, Linköping University, Linköping, 1996.
- [11] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, “Low-power CMOS digital design,” *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 437–484, Apr. 1992.

- [12] B. H. Calhoun, A. Wang, and A. Chandrakasan, "Modeling and sizing for minimum energy operation in subthreshold circuits," *IEEE Journal of Solid-State Circuits*, vol. 40, pp. 1778–1786, Sept. 2005.
- [13] A. Darabiha, A. C. Carusone, and F. R. Kschischang, "Power reduction techniques for LDPC decoders," *IEEE Journal of Solid-State Circuits*, vol. 43, pp. 1835–1845, Aug. 2008.
- [14] S. Lee, N. R. Shanbhag, and A. C. Singer, "A low-power VLSI architecture for turbo decoding," *International Symposium on Low Power Electronics and Design*, pp. 366–371, Aug. 2003.
- [15] T. Zhang and K. K. Parhi, "Joint $(3, k)$ -regular LDPC code and decoder/encoder design," *IEEE Transactions on Signal Processing*, vol. 52, pp. 1065–1079, Apr. 2004.
- [16] M. M. Mansour and N. R. Shanbhag, "High-throughput LDPC decoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, pp. 976–996, Dec. 2003.
- [17] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 Low-Density Parity-Check decoder," *IEEE Journal of Solid-State Circuits*, vol. 37, pp. 404–412, Mar. 2002.
- [18] A. Darabiha, A. C. Carusone, and F. R. Kschischang, "A bit-serial approximate Min-Sum LDPC decoder and FPGA implementation," *International Symposium on Circuits and Systems*, (Kos, Greece), May 2006.
- [19] E. Zimmermann, G. Fettweis, P. Pattisapu, and P. K. Bo-ra, "Reduced complexity LDPC decoding using forced convergence," *International Symposium on Wireless Personal Multimedia Communications*, 2004.
- [20] L. W. A. Blad and O. Gustafsson, "An early decision decoding algorithm for LDPC codes using dynamic thresholds," *European Conference on Circuit Theory and Design*, pp. III/285–III/288, Aug. 2005.
- [21] E. Zimmermann, P. Pattisapu, and G. Fettweis, "Bit-flipping post-processing for forced convergence decoding of LDPC codes," *European Signal Processing Conference*, (Antalya, Turkey), 2005.
- [22] J. K.-S. Lee and J. Thorpe, "Memory-efficient decoding of LDPC codes," *International Symposium on Information Theory (ISIT)*, (Adelaide, South Australia), pp. 459–463, Sept. 2005. (available at www.systems.caltech.edu/~jeremy/research/papers/memory_efficient.pdf).
- [23] J. Lee, B. Lee, J. Thorpe, K. Andrews, S. Dolinar, and J. Hamkins, "A scalable architecture of a structured LDPC decoder," *International Symposium on Information Theory (ISIT)*, (Chicago, Illinois), p. 292, June 2004.

- [24] T. Zhang and K. K. Parhi, "A 54 Mbps (3,6)-regular FPGA LDPC decoder," IEEE Workshop on Signal Processing Systems (SIPS), (San Diego, California), pp. 127–132, Oct. 2002.
- [25] "18 Kbit Block SelectRAM Resources, Xilinx Virtex-II Platform FPGAs Complete Data Sheet," p. 21. (available at <http://direct.xilinx.com/bvdocs/publications/ds031.pdf>).
- [26] "Block RAM summary, Xilinx Virtex-4 User Guide," p. 109. (available at <http://direct.xilinx.com/bvdocs/userguides/ug070.pdf>).
- [27] W. C. Jakes, *Microwave Mobile Communications*. New York: John Wiley & Sons Inc, Feb. 1975.
- [28] "Spartan-3E FPGA Family Data Sheet," Aug. 2009. (available at http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf).
- [29] "IEEE 1076.6: IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis," 2004. (available at <http://ieeexplore.ieee.org/servlet/opac?punumber=9308>).
- [30] "IEEE 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," June 2007. (available at <http://standards.ieee.org/getieee802/download/802.11-2007.pdf>).