Hardware Implementation of Fixed-Point Decoder for Low-Density Lattice Codes

by

Rachna Srivastava

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Dr. Matthieu Arzel Professor, Mathematical & Electrical Engineering, IMT-Atlantique Bretagne
Supervisor(s):	Dr. Vincent C. Gaudet Professor, Dept. of Electrical & Computer Engineering, University of Waterloo
	Dr. Patrick Mitran Professor, Dept. of Electrical & Computer Engineering, University of Waterloo
Internal Members:	Dr. Amir Keyvan Khandani Professor, Dept. of Electrical & Computer Engineering, University of Waterloo
	Dr. Mark Aagaard Associate Professor, Dept. of Electrical & Computer Engineering, University of Waterloo
Internal-External Member:	Dr. William Melek Professor, Dept. of Mechanical & Mechatronics Engineering, University of Waterloo

Author's Declaration

Abstract

Low-density lattice codes (LDLCs) are a special class of lattice codes that can be decoded efficiently using iterative decoding and approach the capacity of the additive white Gaussian noise (AWGN) channel. The construction and intended applications are substantially different from that of more familiar error-correcting codes such as low-density parity check (LDPC) codes, Polar, and Turbo codes. Lattice codes in general have shown great theoretical promise to mitigate interference, possibly leading to significantly higher rates between users in multi-user networks. Research on LDLCs has concentrated on demonstrating the theoretically achievable performance limits of LDLCs, and until now there has been no reported hardware implementation, mainly due to the complexity of message-passing for LDLC decoding.

This thesis contributes to the hardware implementation of the LDLC decoding. We present several fixed-point decoder implementations covering different parts of the architectural design space, on a field-programmable gate array (FPGA) device.

We first present the FPGA implementation of a fixed-point arithmetic LDLC decoder where the Gaussian mixture messages that are exchanged during the iterative decoding process are approximated to a single Gaussian. A detailed quantization study is performed to find the minimum number of bits required for the fixed-point decoder implementation to attain a frame-error-rate (FER) performance similar to floating-point. Efficient numerical methods are used to approximate the non-linear functions required in the decoder. A two-node serial LDLC decoder is implemented on an Intel Arria 10 FPGA as a hardware proof-of-concept attaining a throughput of 440 Ksymbols/sec at high signal-to-noise ratio (SNR). This throughput is obtained at clock frequency of 125 MHz and for a block length of 1000. By exploiting the inherent parallelism of iterative decoding, several parallel message processing blocks are then used to improve the throughput by a factor of 13×. Finally, we propose a pipelined architecture where the decoder achieves a throughput of 10.5 Msymbols/sec, that is, $\sim 24\times$ improvement over the serial decoder.

Then, we implement a multi-Gaussian decoder where the Gaussian mixture messages exchanged during the decoding process have two components. We develop efficient techniques to reduce the decoder complexity for hardware implementation, e.g., selecting the strongest component from the Gaussian mixture as the final decision in iterative decoding, and a simplified method for coefficient computation during the product operation at the variable nodes. With a thorough quantization analysis and applying methods devised to approximate the non-linear functions, we design the multi-Gaussian decoders in fixed-point arithmetic. We first implemented a serial architecture with a single check node and a single variable node. Then, a partially parallel architecture with a single check node and a variable node message processing block with two-stage pipelining is implemented to achieve an effective parallelism of 5 variable nodes. The pipelined architecture achieves an improvement of ~ 0.75 dB in decoding performance over the single Gaussian decoder of degree 3 with an overall design throughput of 550 Ksymbols/sec.

In the final part of the thesis, we further explore the design space and develop complex LDLC decoder designs for higher degrees. We characterize the decoding performance of these decoders and present the design throughputs for different architectures on the target FPGA. Based on these results, we provide insights that will help users to select the most suitable LDLC decoder for a particular application. However this is attained with additional hardware cost and reduced design throughput. A single-Gaussian decoder of degree 7 achieved an FER improvement of 0.75 dB over a single-Gaussian decoder of degree 3 with a throughput of 3.03 Msymbols/sec. The multi-Gaussian Gaussian decoder of degree 7 (with two components in the Gaussian mixture) attains 1.75 dB improvement in FER over the multi-Gaussian decoder of degree 3, and its overall design throughput is ~ 84 Ksymbols/sec. From a broader perspective, the LDLC decoders with higher degrees and larger mixture messages provide a significant improvement in decoding performance. For ultra-reliable applications, a multi-Gaussian decoder of degree 7 is most suitable while for a very high throughput requirement single-Gaussian decoder of degree 3 is the best choice.

We also characterize the performance of multi-Gaussian decoders where the Gaussian mixture messages contain more than two components. Based on the results, the multi-Gaussian decoder with mixture messages that contain 5 components gain approximately $\sim 0.1 - 0.2$ dB (for degree 3 and 7) and ~ 0.3 dB (for degree 5) over multi-Gaussian decoder where mixture messages have only two components.

Acknowledgements

I would like to take this opportunity to thank the people who are very important in my journey to complete this research work and thesis.

Foremost, I must thank my supervisors Professor Gaudet and Professor Mitran for their continuous support and guidance in my Ph.D. study and research. Their smart inputs, excellent guidance and constant encouragement helped me immensely during my research and writing of the thesis. I have learned so much from working with them and I truly believe these learnings will help me to succeed in my future endeavours. Their expert technical supervision is truly motivating for any student. I highly recommend Prof. Gaudet and Prof. Mitran as supervisors.

I would like to thank my all the committee members, Professor Khandani, Professor Melek, Professor Aagaard for their valuable inputs during my comprehensive exam and thereafter, for taking the time to read my thesis and attend my defense. A special thanks to Prof. Aagaard, for finding time to discuss the data flow and architectural optimizations for the decoder hardware. These insightful discussions were very important to improve the decoder architecture in order to achieve higher design throughput. Furthermore, I would like to express my gratitude to Prof. Matthieu Arzel for accepting to serve as the external examiner for my thesis defense. I must thank all the fellow students who contributed to my learning and a joyful graduate experience, including Meysam Shahrbaf, Subhajit, Ahmed Wagdy, Hai and Kazem. I am also thankful to the ECE computer support staff, and administrative staff at the University of Waterloo. In particular, I am grateful to Phil Regier for all of his help with CAD tool support. I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for the funding provided for this research and CMC Microsystems for providing the essential CAD tools.

I would like to thank my family members; my parents, parents-in-law, brothers and sisters for their unceasing support and encouragement. Finally, I would like to thank you my husband, Prateek and my son, Daksh who have experienced several ups and downs with me for last several years. Without their tremendous understanding and constant motivation, it would be impossible for me to complete my PhD journey.

Dedication

To

my PAPA,

my husband, Prateek

& my son, Daksh.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past decade, there has been significant growth in the number of connected devices. Industry analysis suggests that the number of wireless devices is expected to grow to 29.3 billion by 2023, i.e., an increase of more than a 40% compared to 2020 [1, 2].

Widespread network connectivity, with the expectation of reliability and security, poses new challenges for the wireless industry. The increasing demand to support an even larger number of mobile wireless devices with limited bandwidth resources pushes the need for multi-user networks. In these networks, when multiple users transmit and receive data over a shared frequency band simultaneously, *multi-user signal interference* is observed. Due to this interference as well as channel noise, the transmitted data is often not received correctly. Therefore, to transmit data reliably researchers strive to find new error correcting codes (ECCs) with efficient decoding techniques [3–5].

Low-density parity check (LDPC) codes have attained tremendous popularity as the most powerful and practical class of codes implemented in modern wireless networks. The success of LDPC codes can be largely attributed to their capacity approaching performance and relatively low implementation complexity.

While LDPC codes are a class of linear codes over binary (and other finite) alphabets, there is another class of linear codes over the real numbers analogous to LDPC codes, termed *low-density lattice codes (LDLCs)*. LDLCs are continuous-alphabet codes introduced by Sommer *et al.* [6]. These codes have gained attention as many recent coding techniques using lattices have shown to be effective in mitigating multi-user channel interference.

The sparse (low-density) H-matrix that is used for the code construction also renders iterative decoding (also called message-passing) an efficient decoding method [7, 8] similar to LDPC codes.

However, the primary focus of LDLC research so far has been to demonstrate the theoretically achievable performance limits of LDLCs [7, 9–25]. Not much work has been done towards the hardware implementation of LDLC decoding. This is mainly because in iterative decoding of LDLCs, the messages that are passed are *continuous functions* (e.g., Gaussian mixtures for the AWGN channel). In the literature, to demonstrate the theoretical performance of LDLCs, the continuous functions are either sampled and quantized or represented as Gaussian mixtures using parameter lists. However, even with all these reduction strategies, LDLC decoding is challenging to implement in hardware.

1.1 Motivation and Contributions

As practical hardware is key to leverage the proven capabilities of LDLCs, the high-level aim of this thesis is to contribute towards the hardware implementation of LDLC decoding.

More specifically, in this thesis, we present several fixed-point decoder implementations covering different parts of the architectural design space on a field-programmable gate array (FPGA) device. First we describe the fundamental problems encountered on the way to achieving a practical decoder hardware implementation along with a detailed study of the approaches to address those complexities. Then, we demonstrate a proof-of-concept single-Gaussian LDLC decoder on an Intel FPGA. Equipped with the knowledge obtained from single-Gaussian LDLC decoder implementations, we extend the work and achieve a hardware implementation for a multi-Gaussian LDLC decoder. Multi-Gaussian LDLC decoding poses significantly higher design challenges; nonetheless, improved decoding performance is achieved although at a cost of reduced throughput. The multi-Gaussian decoder achieves a performance close to that of [7], where the continuous messages are sampled in 1024 discrete data-points.

The contributions of this thesis are as follows:

In Chapter 3, we study the design and hardware implementation details of a fixed-point single-Gaussian LDLC decoder of degree 3. In this iterative decoder, the exchanged messages are single Gaussians; the Gaussian-mixture messages generated during the intermediate steps in the iterative decoding are reduced to a single Gaussian using a moment-matching method in each decoding iteration [12].

We also perform a comprehensive quantization analysis to find the minimum word length for fixed-point arithmetic representation of the values in the iterative decoding.

Efficient numerical methods are devised to approximate the required non-linear functions, i.e., division and exponentiation, and subsequently characterize their effect on decoder performance.

We then evaluate different hardware architectures and design trade-offs for the single-Gaussian LDLC decoder. A serial architecture with a single check node and a single variable node is implemented as a baseline architecture to provide a proof-of-concept implementation for LDLC decoding in FPGA. To exploit the parallelism of iterative decoding, a decoder architecture with a single check node and 20 variable nodes is included. In order to maximize the re-use of the FPGA resources and enhance the throughput, an LDLC decoder with a single check node and with pipelining to achieve an effective parallelism equivalent to 50 variable nodes is also implemented. Altogether, we can achieve peak decoding at the rate of 10.5 Msymbols/sec on a single Arria 10 FPGA.

In Chapter 4, we study the hardware implementation details of a multi-Gaussian LDLC decoder of degree 3 on the target FPGA. Here the messages exchanged in the decoder are the parametric representation of the Gaussian mixtures instead of a single Gaussian [9,11].

Potential Gaussian mixture reduction methods are compared in terms of computational complexity versus frame error rate (FER) performance, and a suitable method is chosen to reduce a Gaussian mixture to a smaller mixture with a fixed number of components in each decoding iteration. Possible design optimizations to reduce the decoder design complexity in fixed-point arithmetic, and the resulting effect on the decoding performance, are investigated.

A study of possible decoder architectures on target FPGA device is presented along with resource requirements and design throughput details. We further compare single-Gaussian and multi-Gaussian decoders of degree 3 for decoding performance and design throughput on a target FPGA.

In Chapter 5, we push the design limits for the LDLC decoders even further. We implement single-Gaussian and multi-Gaussian LDLC decoders on FPGA devices for a broader range of design parameters, e.g., degrees 5 and 7, and with larger Gaussian mixture messages that contain more than two components. The results obtained from these implementations play a vital role to determine an appropriate LDLC decoder for a certain application.

1.2 Thesis Outline

The rest of this thesis is organized as follows.

Chapter 2 provides the basic definitions of lattice codes along with the constraints and properties of low-density lattice codes. The basic iterative decoding algorithm for LDLCs, where the messages exchanged in decoding are continuous functions, is included. It also presents a summary of the LDLC decoders published in the literature and compares their empirical performance.

Chapter 3 presents the implementation details for a single-Gaussian LDLC decoder of degree 3 on a target FPGA device. A study of the optimization techniques to reduce decoder complexity, a detailed quantization analysis of the fixed-point arithmetic, and efficient techniques to implement the required non-linear functions are described.

Chapter 4 presents the hardware implementation details for a multi-Gaussian LDLC decoder with degree 3. Chapter 4 also provides a detailed comparison of the single-Gaussian and the multi-Gaussian decoder with degree 3 and block length of 1000.

In Chapter 5, we push the design space boundaries on the target FPGA and present the single-Gaussian and multi-Gaussian decoder implementations for degrees 5 and 7 and also present multi-Gaussian decoders where the Gaussian mixture messages comprise more than two components.

Finally, Chapter 6 summarizes the contributions of this thesis and outlines the design aspects that were not explored in the thesis but that would constitute a valuable extension as future work.

Chapter 2

Literature Survey

2.1 Channel Capacity and Channel Codes

Shannon's channel capacity theorem [26] shows that provided the rate of transmission is less than a particular rate called channel capacity it is possible, in principle to transmit information with arbitrarily small probability of error.

Codes that allow data transmission at the rates near channel capacity and with low probability of error are often referred to as 'near Shannon-limit codes' or (sometimes inaccurately as) 'capacity achieving codes'.

2.1.1 Codes for Binary Channels

While Shannon's capacity theorem showed that long random codebooks can achieve the capacity of binary (e.g., binary symmetric) and finite alphabet channels, thereafter, it was discovered that capacity could also be achieved using structured linear codes, where each codeword is a linear combination of a set of basis vectors.

Among the class of linear binary codes, *turbo codes* [27] and *low-density parity-check codes* are codes which allow efficient iterative decoding [28–31]. Specifically, turbo codes and LDPC codes are regarded as near Shannon-limit error-correcting-codes with practical decoding algorithms. Turbo codes use concatenated convolutional encoders and interleavers, whereas LDPC codes are block codes that use a sparse parity-check matrix for encoding. The sparse nature of the parity-check matrix in LDPC codes provides the flexibility to perform decoding operations in parallel; this is very advantageous for a high-speed and efficient hardware implementation of a decoder.

2.1.2 Codes for Additive White Gaussian Noise Channels

Shannon showed that for the additive white Gaussian noise (AWGN) channel, continuous alphabet codes with random codebooks where each code letter is drawn i.i.d. according to a Gaussian-distribution can approach the capacity of the channel [32]. In the class of codes for the AWGN channel, lattice codes are structured codes that are the Euclidean space analogue of linear block codes and have gained much attention lately as they can also, in principle, achieve AWGN channel capacity [33, 34] and many coding techniques using lattices are shown to be effective in mitigating *multi-user channel interference* [4, 35]. For example, among lattice based decoding strategies, *compute-and-forward* first recovers enough linear combinations of the received message and then extracts the intended message [3, 5, 36–48]. Lattice codes are well suited for this purpose because their linear structure matches the additivity of the channels, where the sum of two codewords superimposed additively lies in the lattice.

Low-density lattice codes (LDLCs) studied in this thesis are a class of the lattice codes.

2.2 LDLC and Related Codes

2.2.1 Block Codes

Low-density lattice codes are closely related to *linear block codes* over finite fields. Hence it is worth revisiting the structure and properties of block codes.

In a block code, $\mathcal{C}$, over a finite field $\mathbb{F}$ of size $q = |\mathbb{F}|$, an information sequence is segmented into message blocks of fixed length; each message block, denoted by $\underline{m}$, consists of k information symbols over $\mathbb{F}$ (typically $|\mathbb{F}| = 2$). The encoder generates a block of n coded symbols, denoted by $\underline{x}$, based on the k information symbols. This code with n-tuple codeword and k information symbols is called an (n, k) *block code* [49–51] over $\mathbb{F}$. The *rate R of a block code* is given as

$$R = \frac{k}{n}. \tag{2.1}$$

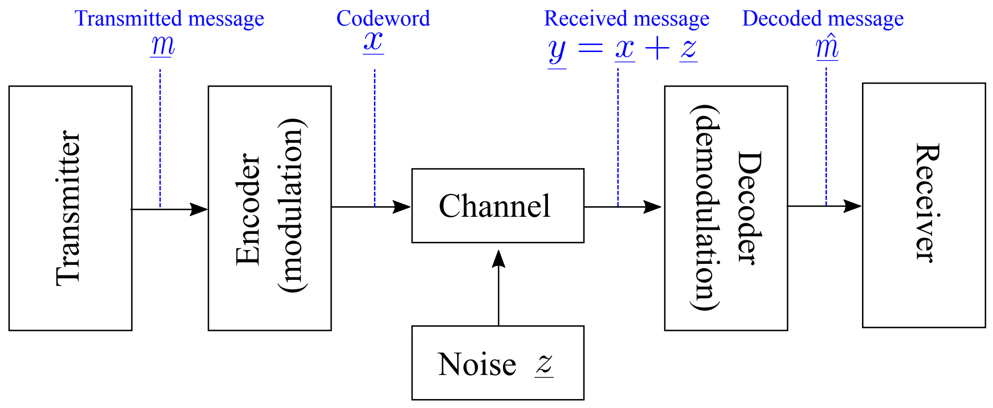


Figure 2.1: Schematic of a general communication channel

There are q^n distinct sequences of length n over $\mathbb{F}$ and from this, q^k codewords are selected to form the code. For a block code to be useful, the q^k codewords must be distinct. Therefore, there should be a one-to-one correspondence between a message $\underline{m}$ and its codeword $\underline{x}$. A *block code* is *linear* over a finite-field $\mathbb{F}$ if the set of codewords is a linear subspace of dimension k of the space $\mathbb{F}^n$.

Encoding and Decoding (Generator Matrix and Parity-check Matrix)

For every linear block code k-linearly independent vectors, $\underline{g}_1, \underline{g}_2, \ldots, \underline{g}_k$ can be identified such that all the codewords can be obtained by a linear combination of these vectors. These vectors, $\underline{g}_1, \underline{g}_2, \ldots, \underline{g}_k$ are called basis vectors. For every linear code, there exists a k-by-n generator matrix, G, such that,

$$\underline{x} = \underline{m} \cdot G, \tag{2.2}$$

where $\underline{x} = \{x_1, x_2, \ldots, x_n\}$, $\underline{m} = \{m_1, m_2, \ldots, m_k\}$ and G is represented as,

$$G = \begin{bmatrix} \underline{g}_1 \\ \underline{g}_2 \\ \vdots \\ \underline{g}_k \end{bmatrix} = \begin{bmatrix} g_{11} & g_{12} \cdots g_{1n} \\ g_{21} & g_{22} \cdots g_{2n} \\ \vdots & \\ g_{k1} & g_{k2} \cdots g_{kn} \end{bmatrix}.$$

A parity-check matrix, H, of a (n, k) linear block code is a full-rank $(n-k)$-by-n matrix satisfying,

$$H \cdot G^T = 0. \tag{2.3}$$

As a consequence of (2.3),

$$\underline{x} \cdot H^T = 0. \tag{2.4}$$

This condition is also known as parity-check equation.

Using elementary row operations and column permutations, any generator matrix can be converted to a generator matrix for an equivalent code that is in systematic form, in which the left side of the matrix is the identity matrix, i.e.,

$$G = [I_k|P], \tag{2.5}$$

where I_k is the $k \times k$ identity matrix and P is a $(k) \times (n-k)$ parity matrix. Similarly, a systematic form for a parity-check matrix can be obtained, which has an identity matrix at the right side, i.e.,

$$H = [-P^T|I_{n-k}]. \tag{2.6}$$

Fig. 2.1 illustrates the basic communication channel model, where the encoder generates a codeword $\underline{x}$ corresponding to message $\underline{m}$. While codeword $\underline{x}$ is transmitted over the channel, noise $\underline{z}$ is added to it and the noisy message $\underline{y} = \underline{x} + \underline{z}$ is received. The received message $\underline{y}$ is decoded at the receiver's end and an estimate $\underline{\hat{m}}$ of the message $\underline{m}$ is obtained.

Minimum Distance

For block codes, the Hamming distance (or simply distance), $d(\underline{x}_1, \underline{x}_2)$ between two codewords $\underline{x}_1$ and $\underline{x}_2$ is the number of positions where $\underline{x}_1$ and $\underline{x}_2$ have different symbols, e.g., for $\underline{x}_1$=011001101110 and $\underline{x}_2$=011001010001, $d(\underline{x}_1, \underline{x}_2) = 6$.

The minimum distance, d_{min}, for block code $\mathcal{C}$, is defined as the smallest distance between any pair of codewords in the code. For a given block code $\mathcal{C}$, d_{min} is then

$$d_{min} = min\{d(\underline{x}_1, \underline{x}_2) : \underline{x}_1, \underline{x}_2 \in \mathcal{C}, \underline{x}_1 \neq \underline{x}_2\}. \tag{2.7}$$

Modulation

Codewords in finite fields cannot be directly transmitted over continuous-time channels. Instead, the codewords are transmitted over the channel using digital modulation, where one or more properties of a periodic waveform, called the carrier signal, is modulated depending upon the symbols to be transmitted. Traditionally, in binary-phase-shift-keying (BPSK), only one sinusoid is taken as a basis function for modulation. Modulation is achieved by varying the phase of the basis function depending on the codeword bits ('0' or '1') [52]. Eqns. (2.8) and (2.9) outline the BPSK modulation technique, where $S_0(t)$ is transmitted when the bit is '0' and $S_1(t)$ is transmitted when bit is '1'.

$$S_0(t) = Acos(\omega t), \tag{2.8}$$

$$S_1(t) = Acos(\omega t + \pi). \tag{2.9}$$

This modulation is very robust but unsuitable for high data-rate applications as it transmits only one symbol at a time. To overcome this disadvantage, higher-level modulation schemes are used such as 8-PSK, 16-QAM, etc. These modulation schemes can transmit several binary symbols per modulated symbol, each represented by different states of magnitude and phase of the carrier.

Encoding and modulation need not be treated as separate processes; these two can be integrated together by matching the encoding technique to the modulation scheme. The two step process whereby the information bits are converted first into a coded bit stream and then into a modulated signal is then replaced by a single process which converts the data stream directly into a suitable signal for transmission over the channel [53].

2.2.2 Low-Density Parity-Check Codes

Low-density lattice codes are motivated by the design simplicity of LDPC block codes. Therefore, the basic concepts of the LDPC code design and decoding methodology are first reviewed.

The term *low-density* in low-density parity check codes refers to the characteristic that the parity-check matrix (H) used for the LDPC code design contains only a few '1's in comparison to '0's (i.e., "the density of '1's is low"). LDPC codes are arguably one of the best error correction codes in existence at present. Their main advantage is that they provide a performance which is very close to capacity and use decoding algorithms with complexity linear to the code size. Moreover, they are well suited for implementations that can make extensive use of parallelism.

LDPCs were first introduced by Gallager in his PhD thesis in the 1960s [28]. Due to the high computational effort (for the time) required for encoder and decoder implementation, LDPC codes were mostly ignored until rediscovered in the mid 1990s by Neal and Mackay [29, 30].

To understand LDPC code design and structure, let's assume that an LDPC code is constructed using a parity-check matrix, H, of size (m, n) where m is number of rows and n is the number of columns in the matrix. The number of '1's in each row and column are referred as the row and column weights, wt_r and, wt_c respectively. An example of an (4,8) LDPC parity-check matrix, with $wt_r = 4$ and $wt_c = 2$ is given below:

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

In [54] Tanner introduced an effective graphical representation for LDPC codes. These graphs, known as *Tanner graphs*, are effective as they not only provide a complete representation of the code, but they also help to understand the decoding algorithm. Tanner graphs are bipartite graphs, which means that the nodes of the graph are separated into two distinct sets (or types) and edges can only connect nodes of two different types. The two types of nodes in a Tanner graph are called variable nodes and check nodes. Each row of the H matrix represents a check node and each column represents a variable node. An edge in a Tanner graph is a connection between a variable node and a check node so the number of edges in the Tanner graph and the number of ones in the H matrix are equal. The i^{th} check node is connected to the j^{th} variable node if the element h_{ij} of H is a 1.

A series of interconnected nodes with same origin and termination, given no other nodes repeat, is called a cycle. The length of the cycle is given by the number of edges in it and the girth of a graph is defined to be the size of the smallest cycle. Figure 2.2 shows a graphical representation of the example (4,8) LDPC H-matrix described above. This H matrix contains a 4-cycle loop highlighted in the bipartite graph using dotted lines.

LDPC Encoding

LDPC codes are linear block codes. If the message is $\underline{m}$ and the generator matrix is G, then the codeword is given as:

$$\underline{x} = \underline{m} \cdot G. \tag{2.10}$$

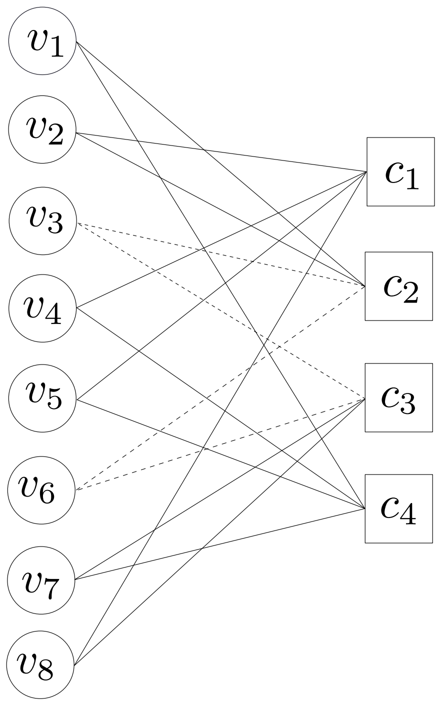


Figure 2.2: Bipartite graph representation of the example (4,8) LDPC parity-check matrix.

In order to obtain G from the H matrix, the H matrix is first converted into systematic form by applying proper elementary operations as shown in (2.6). Based on the systematic form of the H matrix, the generator matrix is obtained as

$$G = [I_k | P^T]. \tag{2.11}$$

LDPC Decoding

LDPC decoding is performed through an iterative decoding algorithm based on the Tanner graph. LDPC decoding algorithms are termed as message-passing iterative decoding algorithms which perform local calculations at check (variable) nodes and pass those local results via messages to variable (check) nodes [29, 30, 55–63]. Based on the information exchanged between the check node and variable nodes, LDPC decoding can be classified into two types:

- Hard-decision decoding
- Soft-decision decoding

Hard-Decision Decoding: In hard-decision decoding the messages passed in decoding iterations are binary bits and the decoding decision is based on this binary information. The bit flipping algorithm of [29] is an example of hard decision decoding.

In the bit flipping algorithm, a check node finds the bit in error by checking the parity of the data stream received from all the variable nodes connected to it; the parity may be even or odd. If the number of 1's received at check nodes satisfies the required parity, then it sends the same data back to variable nodes connected to it; otherwise it flips the bit (from variable nodes) which is involved in the largest number of unsatisfied parity checks and re-sends it to the variable node. A variable node utilizes the incoming check node messages to decide if the bit at its position is a '0' or a '1' by majority rule. The variable node then sends this hard-decision to its connected check nodes.

The decoding process is terminated whenever a valid code word has been found, i.e., all the parity check equations are satisfied.

Soft-Decision Decoding: Soft-decision decoding is based on the idea of belief propagation. In this scheme, the decoder makes an inference for each bit of the transmitted vector by calculating the marginal distribution of each bit, conditional on the received vector. The sum-product algorithm (SPA) is a soft-decision message passing algorithm which is similar to the bit-flipping algorithm [30]. The main difference between the SPA and bit-flipping algorithm is that in the SPA, check node and variable node messages are conditional probabilities instead of binary bits. These conditional probabilities are expressed as log-likelihood ratios of bits where the log-likelihood ratio is defined as,

$$log\,likelihood\,ratio\;(LLR) = \log\left(\frac{x_i = \text{`0'}\,|\,y_i}{x_i = \text{`1'}\,|\,y_i}\right). \tag{2.12}$$

2.2.3 Low-Density Lattice Codes

Low-density lattice codes (LDLCs) were first introduced by Sommer et al. [6]. As opposed to LDPCs or algebraic codes which are defined over a finite field, lattice codes are defined over real numbers. LDLCs belong to the class of lattice codes.

Lattice Codes

Lattice codes are regarded as the Euclidean space analogue of linear block codes. In a lattice code, an integer-valued information sequence is converted to a point in Euclidean space.

Definition 1. *(Lattice)*

An n-dimensional lattice, $\Lambda \subset \mathbb{R}^n$, is defined as all the integer linear combinations of n given linearly independent basis vectors, $\underline{g}_1, \ldots, \underline{g}_n \in \mathbb{R}^n$. Taking the basis vectors as the columns of the generator matrix G, (i.e., $G = (\underline{g}_1, \ldots, \underline{g}_n)$), the lattice Λ is given by

$$\Lambda = \{\underline{x} \in \mathbb{R}^n : \underline{x} = G\underline{b},\ \underline{b} \in \mathbb{Z}^n\}. \tag{2.13}$$

Figure 2.3 shows the graphical illustration of an example 2-dimensional lattice with basis vectors $\underline{g}_1$ and $\underline{g}_2$.

Definition 2. *(Voronoi Region)*

The Voronoi region of a lattice point $\underline{x} \in \mathbb{R}^n$ is the subset of $\mathbb{R}^n$ which is closer to $\underline{x}$ than to any other lattice point. It is generally represented as ν. For any lattice $\Lambda(G)$, all the Voronoi regions have the same volume, denoted as $V(\Lambda)$, which is equal to the determinant of the generator matrix, G, i.e.,

$$V(\Lambda) = |\det(G)|. \tag{2.14}$$

Low-density lattice codes (LDLCs) are lattice codes which have some specific constraints on the matrix $H = G^{-1}$.

Though most of the published papers on LDLCs have termed the inverse of the generator matrix as 'parity-check matrix', in this thesis it is called 'constraint matrix'. This naming convention is more reasonable as the elements of the inverse of the LDLC generator matrix are real numbers.

Low-Density Lattice Codes (LDLCs)

Definition 3. *(LDLC) A low-density lattice code is an n-dimensional lattice code defined by a non-singular generator matrix that satisfies the condition that the constraint matrix, $H = G^{-1}$, is sparse.*

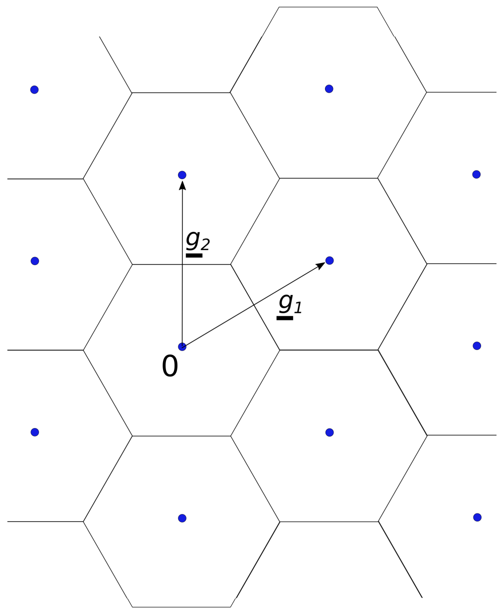


Figure 2.3: Graphical illustration of an example 2-dimensional lattice with basis vectors $\underline{g}_1$ and $\underline{g}_2$.

Similar to the LDPC codes, the constraint matrix H is the core of an LDLC design as it fully describes the code. For a given n-by-n constraint matrix H, the row degree r_i of the i^{th} row is the number of non-zero elements in that row. Similarly the column degree c_i is the number of non-zero elements in the i^{th} column of H.

Definition 4. *(Regular LDLC)*

An n-dimensional LDLC is said to be regular if all the row degrees and column degrees of the constraint matrix are equal to a common degree d.

Definition 5. *(Latin-Square or Magic Square LDLC)*

An n-dimensional regular LDLC with degree d is called a Latin-Square or Magic-Square LDLC if every row and every column of H has the same d non-zero values $\bar{h}_1 \geq \bar{h}_2 \geq \ldots \geq \bar{h}_d > 0$ except for possible sign flips. The sequence of non-zero values $\bar{h}_1, \bar{h}_2, \ldots, \bar{h}_d$ in descending order is referred as the generating sequence of a particular LDLC.

An example of a constraint matrix of degree 3 is shown below [7]:

$$H = G^{-1} = \begin{bmatrix} 0 & -0.8 & 0 & -0.5 & 1 & 0 \\ 0.8 & 0 & 0 & 1 & 0 & -0.5 \\ 0 & 0.5 & 1 & 0 & 0.8 & 0 \\ 0 & 0 & -0.5 & -0.8 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0.5 & 0.8 \\ 0.5 & -1 & -0.8 & 0 & 0 & 0 \end{bmatrix}. \tag{2.15}$$

The generating sequence for this example is 1, 0.8, 0.5.

In [7], Sommer et al. used a regular Latin-square H matrix for LDLC code design and published the decoder performance results for various block lengths. For ease of comparison, it is preferred to keep the volume of the Voronoi region normalized to unity. So for the designed LDLCs, the determinant of the generator matrix is 1, i.e.,

$$|\det(G)| = 1. \tag{2.16}$$

The constraint matrix H determines the bipartite graph for the LDLCs and thus has a significant influence on decoding procedure. The next section describes the H matrix generation algorithm and other important parameters related to LDLC decoding.

2.3 Parameters for LDLC Code Design

2.3.1 Constraint Matrix

Similar to LDPC codes, a bipartite graph with few if any short cycles is preferred in order to achieve faster decoding convergence and better error correction capability. Sommer [7] proposed an efficient algorithm to create the H matrix for LDLCs which ensures that this matrix does not contain any 2-cycles (i.e., two parallel edges which originate from same variable and check nodes) or 4-cycles (i.e., two variable nodes that are both connected to the same pair of check nodes). Therefore, an LDLC matrix generated using this algorithm [7] contains no 2-cycles or 4-cycles and the girth of the bipartite graph is at least 6.

Algorithm 1 provides the pseudo code for the H matrix generation [7]. Here the (i, j) element of matrix P is denoted by $P_{i,j}$ and the k^{th} column of a matrix P is denoted by $P_{:,k}$.

Algorithm 1: Pseudo Code for H-matrix Generation [7]

```
##Initialization
Choose $d$ random permutations on $\{1, 2, \cdots n\}$.
Arrange the permutations in an $d \times n$ matrix, $P$ such that each row holds a permutation.
$c = 1$; # column index
$cyclefree\_columns = 0$; # number of consecutive columns without cycles
# cycle removal:
while cyclefree_columns < $n$ do
    changed_permutation = 0;
    if exists $i \neq j$ such that $P_{i,c} = P_{j,c}$ then
        # a 2-cycle was found in column $c$
        changed_permutation = i;
    else
        if exists $c_0 \neq c$ such that $P_{:,c}$ and $P_{:,c_0}$ have two or more common elements then
            # a 4-cycle was found at column c
            changed_permutation = line of P for which the first common element appears in
              column c;
    if changed_permutation $\neq 0$ then
        # a permutation should be modified to remove cycle
        choose a random integer $1 \leq i \leq n$;
        swap locations $c$ and $i$ in
        permutation changed_permutation;
        $cyclefree\_columns = 0$;
    else
        # no cycle was found in column $c$
        $cyclefree\_columns = cyclefree\_columns + 1$
    # increase column index
    $c = c + 1$;
    if $c > n$ then
        $c = 1$;
# Finally, build $H$ from the permutations
Initialize H as an $n \times n$ zero matrix;
for $i = 1 : n$ do
    for $j = 1 : d$ do
        $H_{P_{j,i},i} = \bar{h}_j \cdot random\_sign$;
```

The algorithm inputs are block length n, degree d and generating sequence, $\bar{h}_1 \geq \bar{h}_2 \geq \ldots \geq \bar{h}_d > 0$ and the output is a constraint matrix, H.

The constraint matrix H generated in Algorithm 1 is further normalized by $\sqrt[n]{|\det(H)|}$ to yield $|\det G| = |\det H| = 1$.

2.3.2 Distance from Channel Capacity

Sommer et al. applied Poltyrev's [64] definition of capacity for the lattice codes in context of low-density lattice codes.

Poltyrev [64] showed that for the AWGN channel without restrictions (with no power limit), code rate is a meaningless measure since it can be increased without limit. He suggested instead a generic definition of capacity for lattice codes with no power restriction. According to this, capacity for lattice codes is defined as the maximal possible codeword density that can be recovered reliably at the receiver. This generalized capacity implied that there exists a lattice G of high enough dimension n that enables transmission with arbitrarily small error probability, if and only if the channel noise variance satisfies

$$\sigma^2 < \sqrt[n]{|\det(G)|^2}/2\pi e,$$

where $e = 2.71828...$ is Euler's number (also known as the natural constant).

Since for the designed LDLCs [7], $|\det(G)| = 1$ and the AWGN channel is without power restrictions, it is possible to quantify distance from capacity or the maximal performance limit as the distance of the noise variance σ^2 from $1/2\pi e$, i.e., by $-10\log_{10}(2\pi e\sigma^2)$ in dB. All decoder performance curves in [7] have been drawn between symbol error rate (SER) and the distance from capacity, $-10\log_{10}(2\pi e\sigma^2)$, in dB.

2.3.3 Generating Sequence

Section 2.3.1 described the algorithm to construct an LDLC constraint matrix for a generating sequence. Sommer et al. proposed two approaches for choosing the non-zero elements of the generating sequence.

In the first approach, the generating sequence is given by the reciprocals of the smallest d prime numbers $\frac{1}{2}, \frac{1}{3}, \frac{1}{5}, \frac{1}{7}, \frac{1}{11}, \frac{1}{13} \cdots$. In order to achieve fast decoding convergence it is important that when the Gaussian mixture messages are multiplied at the variable node (see Section 2.4), the correct peaks should align and all other peaks should be attenuated in amplitude. Choosing the generating sequence to be reciprocals of the smallest d prime numbers is helpful to attain faster convergence in LDLC decoding [7].

Simulation results show that increasing d beyond 7 gives negligible improvement in the decoder performance. However, performance does improve by adding some 'dither' to the sequence, so the generating sequence in [7] is given as, $\frac{1}{2.31}, \frac{1}{3.17}, \frac{1}{5.11}, \frac{1}{7.33}, \frac{1}{11.71}, \frac{1}{13.11} \frac{1}{17.55}$.

In the second approach, the generating sequence is, $1, \epsilon, \epsilon \cdots \epsilon$ where $\epsilon = \frac{1}{\sqrt{d}}$ and < 1 . For example, for $d = 7$ the generating sequence is $1, \frac{1}{\sqrt{7}}, \frac{1}{\sqrt{7}}, \frac{1}{\sqrt{7}}, \frac{1}{\sqrt{7}}, \frac{1}{\sqrt{7}}, \frac{1}{\sqrt{7}}$ as used in [10], [12].

2.4 LDLC Decoding for the Additive White Gaussian Noise Channel

For LDLCs, the lattice codeword, $\underline{x} = G \cdot \underline{b}$ is transmitted over the additive white Gaussian noise (AWGN) channel where the generator matrix is obtained as $G \triangleq H^{-1}$.

The receiver observes a noisy codeword, $\underline{y}$, according to

$$\underline{y} = \underline{x} + \underline{z}, \tag{2.17}$$

where $\underline{z}$ represents a vector of independent and identically distributed (i.i.d.) Gaussian noise samples with common variance, σ^2.

The sparse nature of the constraint matrix used for code design makes LDLC decoding similar to the LDPC decoding described in Subsection 2.2.2. In the literature, one of the preferred methods for LDLC decoding is iterative message passing over the bipartite graph. During the iterative decoding process, in each decoding iteration, the variable nodes send messages to the check nodes along the edges of the bipartite graph and vice-versa.

Iterative Decoding Algorithm The iterative decoding algorithm proposed by Sommer et al. in [7] estimates the probability density function (PDF) of the codeword $\underline{x}$ based on the received vector $\underline{y}$. Moreover, for simplicity, it is advantageous to estimate the marginal PDF of each of the symbols of the codeword (lattice point), conditioned on the whole received vector $\underline{y}$, i.e., $f_{x_k \mid y}(x_k \mid \underline{y})$ which is estimated using iterative decoding.

Since every variable node decodes the marginal PDF of one of the symbols in the transmitted lattice point, there is a one-to-one correspondence between variable nodes and symbols of the lattice point. Further, in this thesis, we choose to represent the variable nodes with the same notation as symbols of the lattice point, i.e., x_k.

Initialization

At the start of the decoding process, each variable node x_k sends a message $f_k^{(0)}(t)$, for a continuous-time variable t, received from the AWGN channel along all the edges connected to it. The continuous function $f_k^{(0)}(t)$ is given as

$$f_k^{(0)}(t) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_k - t)^2}{2\sigma^2}}. \tag{2.18}$$

As shown in Figure 2.4, for the example matrix in (2.15), variable node x_2 sends message $f_2^{(0)}(t)$ to all the connected check nodes, where $f_2^{(0)}(t)$ is given by

$$f_2^{(0)}(t) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_2 - t)^2}{2\sigma^2}}. \tag{2.19}$$

Basic Iteration: Check node message

Similar to LDPC codes, each check node in an LDLC represents an appropriate constraint that corresponds to

$$\sum_{l=1}^{d} h_l x_{kl} = i, \tag{2.20}$$

where d is the degree of the constraint matrix H and x_{kl}, $l = 1, 2, \ldots, d$ are the variable nodes connected to a check node with edge weights of h_l, where h_l is one of the $\bar{h}$'s with a possible sign flip. In general, i can take any value in $\mathbb{Z}$, but the value of i is restricted within a certain limit, i.e., $i \in \mathcal{L} \subset \mathbb{Z}$ where $\mathcal{L}$ is a subset of integers. This is helpful in decoding as the components with means far from the channel message have almost zero mixing coefficients after the computations at the variable node, and therefore can be ignored.

In (2.20) one can solve for x_{kj} to obtain:

$$x_{kj} = \frac{i - \sum\limits_{\substack{l=1 \\ l \neq j}}^{l=d} h_l x_{kl}}{h_j}. \tag{2.21}$$

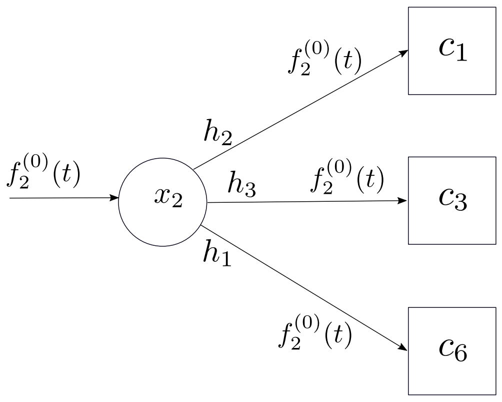


Figure 2.4: Initialization - a variable node sends the received channel PDF to all the connected check nodes.

To compute the outgoing message from check node c_k along edge h_j, we first need to compute the PDF of x_{kj} in (2.21) when $i = 0$. To compute this PDF, two important properties of the PDFs of random variables are used.

Lemma 1. *(PDF of Sum of Independent Random Variables) Let A and B be two independent random variables and $f_A(t)$ and $f_B(t)$ their respective PDFs. The PDF of the sum $(A + B)$ is*

$$f_{(A+B)}(t) = f_A(t) * f_B(t). \tag{2.22}$$

Lemma 2. *(PDF of a Scaled Random Variable) If A and B are two random variables, such that $B = rA + s$ where $r \neq 0$, the PDF of B is*

$$f_B(t) = \frac{1}{|r|} f_A\Big(\frac{x - s}{r}\Big). \tag{2.23}$$

In the special case that $s = 0$ in Lemma 2,

$$f_{rA}(t) = \frac{1}{|r|} f_A\Big(\frac{x}{r}\Big). \tag{2.24}$$

Now assume $f_l(t)$, $l = 1, 2, \ldots, d$ are the PDFs received at the check node along the edges with weights, h_l, $l = 1, 2, \ldots, d$. Then using Definitions 1 and 2, the PDF of $\sum\limits_{\substack{l=1 \\ l \neq j}}^{l=d} h_l x_{kl}$, denoted by $\tilde{p}_j(t)$, can be computed as,

$$\tilde{p}_j(t) = \frac{1}{|h_1|} f_1\Big(\frac{t}{h_1}\Big) * \cdots * \frac{1}{|h_{j-1}|} f_{j-1}\Big(\frac{t}{h_{j-1}}\Big) * \frac{1}{|h_{j+1}|} f_{j+1}\Big(\frac{t}{h_{j+1}}\Big) * \cdots * \frac{1}{|h_d|} f_d\Big(\frac{t}{h_d}\Big). \tag{2.25}$$

The constant in (2.25) can be ignored as it would be normalized later and thus $\tilde{p}_j(t)$ can be rewritten as

$$\tilde{p}_j(t) = f_1\Big(\frac{t}{h_1}\Big) * \cdots * f_{j-1}\Big(\frac{t}{h_{j-1}}\Big) * f_{j+1}\Big(\frac{t}{h_{j+1}}\Big) * \cdots * f_d\Big(\frac{t}{h_d}\Big). \tag{2.26}$$

Using Definition 2, the PDF of $-\sum_{l=1}^{d} \frac{h_l x_{kl}}{h_j}$ denoted by $p_j(t)$, is computed as

$$p_j(t) = \tilde{p}_j(-h_j t). \tag{2.27}$$

The term $\frac{i}{h_j}$ in (2.21) creates the periodic shift/extension of the check node message x_{kj} for different values of i. Message $p_j(t)$ in (2.27) is periodically extended with a period of $\frac{1}{|h_j|}$ and after the periodic extension the final outgoing check node message, $Q_j(t)$, can be written as,

$$Q_j(t) = \sum_{i=-\infty}^{\infty} p_j\Big(t - \frac{i}{|h_j|}\Big). \tag{2.28}$$

Figure 2.5 shows the outgoing check node message, $Q_3(t)$, along the edge with weight h_3 computed as,

$$\tilde{p}_3(t) = f_1\Big(\frac{t}{h_1}\Big) * f_2\Big(\frac{t}{h_2}\Big), \tag{2.29}$$

$$p_3(t) = \tilde{p}(-h_3 t), \tag{2.30}$$

$$Q_3(t) = \sum_{i=-\infty}^{\infty} p_3\Big(t - \frac{i}{|h_3|}\Big). \tag{2.31}$$

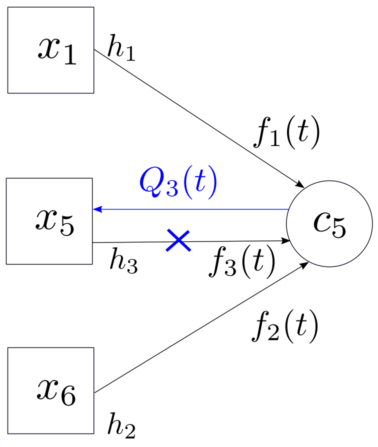


Figure 2.5: Check node operation - the outgoing message on the edge with weight h_3 is computed with all the incoming messages, except the one on edge with weight h_3.

Basic Iteration: Variable node message

Each variable node computes a local estimate of codeword symbol x_k based on all the incoming check node messages to that node. Since the code symbols are assumed i.i.d., the outgoing variable node message along an edge is the product of the channel PDF and all incoming check node messages except the one coming along that edge.

Assume that a given variable node x_k is connected to the check nodes $c_{k1}, c_{k2}, \ldots, c_{kd}$, where d is the degree of H. If $Q_l(t)$, $l = 1, 2, \ldots, d$ is the incoming message from check node c_{kl} connected with edge with weight h_l (one of the $\bar{h}$'s with a possible sign flip) to this variable node in the previous half-iteration, the outgoing variable node message to the check node c_{kj} is calculated in following steps:

- Product step: the channel PDF and all the incoming messages to the variable node, except $Q_j(t)$ (the incoming message along the edge with weight h_j) are first multiplied, i.e.,

$$\tilde{f}_j(t) = e^{-\frac{(y_k-t)^2}{2\sigma^2}} \prod_{\substack{l=1 \\ l\neq j}}^{d} Q_l(t). \tag{2.32}$$

- Normalization step: as the product computed in (2.32) is not a PDF, it is further

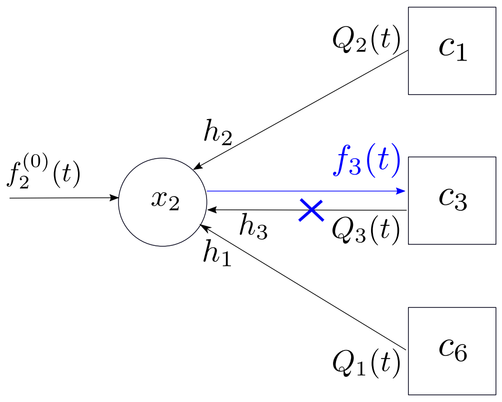


Figure 2.6: Variable node operation - the outgoing message on the edge with weight h_3 is computed with channel message and all the incoming check node messages, except the one on edge with weight h_3.

normalized according to

$$f_j(t) = \frac{\tilde{f}_j(t)}{\int_{-\infty}^{\infty} \tilde{f}_j(t)dt}. \tag{2.33}$$

Fig. 2.6 shows the outgoing variable node message from x_2 along the edge with weight h_3 computed according to (2.34) and (2.35) below for the example matrix (2.15):

$$\tilde{f}_3(t) = e^{-\frac{(y_2-t)^2}{2\sigma^2}} Q_1(t)Q_2(t), \tag{2.34}$$

$$f_3(t) = \frac{\tilde{f}_3(t)}{\int_{-\infty}^{\infty} \tilde{f}_3(t)dt}. \tag{2.35}$$

Here $Q_1(t)$ and $Q_2(t)$ are the incoming check node messages along the edge with weights h_1 and h_2 respectively.

Final decision

After reaching the desired number of decoding iterations, the final PDF of an estimated codeword symbol, $\hat{w}_k$, is calculated by multiplying the channel PDF and all the incoming

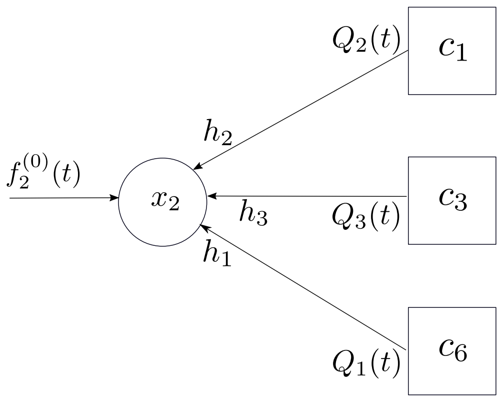


Figure 2.7: Final decision in Iterative decoding.

check node messages to the variable node without omitting any, i.e.,

$$\tilde{f}_k^{final}(t) = e^{-\frac{(y_k - t)^2}{2\sigma^2}} \prod_{l=1}^{d} Q_l(t). \tag{2.36}$$

The decoded codeword symbol, $\hat{w}_k$, is obtained at the variable node as,

$$\hat{w}_k = \underset{t}{\text{argmax}}\ \tilde{f}_k^{final}(t). \tag{2.37}$$

Fig. 2.7 shows all the incoming messages at the variable node x_2 for the example matrix (2.15). The final decoded codeword symbol $\hat{w}_2$ in this case is computed according to:

$$\tilde{f}_2^{final}(t) = e^{-\frac{(y_2 - t)^2}{2\sigma^2}} Q_1(t)Q_2(t)Q_3(t), \tag{2.38}$$

$$\hat{w}_2 = \underset{t}{\text{argmax}}\ \tilde{f}_2^{final}(t). \tag{2.39}$$

2.5 LDLC Decoders in the Literature

From a theoretical point of view, the iterative decoding Algorithm 2.4, in principle, shows good decoding performance. Nevertheless, its implementation is not practical either in software or hardware. This is primarily due to the fact that when the channel is AWGN, although the initial message received from the channel is a single-Gaussian PDF, the operations at the check and variable nodes generate Gaussian mixture messages. Further, as the decoding iterations progress, the number of components in the Gaussian mixtures grows exponentially and the implementation eventually has extremely large storage requirements and computational cost.

In prior works, two approaches are primarily applied to reduce the decoder complexity. In the first approach, the continuous functions are approximated by sampling and quantization [7], and these samples are exchanged. In the second approach, the Gaussian mixture messages are represented by Gaussian parametric lists [9,10,12], and reduced as necessary to keep the number of mixture components small.

2.5.1 Sampled PDF LDLC decoder

In the sampled approach [7], the PDFs of the continuous codeword (continuous functions) are sampled with a resolution of $\frac{1}{64}$ and quantized in discrete-levels. The sampling range is a finite symmetric region of length 4 around the noisy channel message.

It is the sampled and quantized values that are passed iteratively between the check and variable nodes. Due to the sampling and quantization resolution this approach has a large storage requirement and significant computational complexity. Fig. 2.8 shows the symbol error rate versus distance from capacity, $-10\log 10(2\pi e^2)$, for the sampled PDF LDLC decoder [7].

2.5.2 Parametric LDLC decoders

In [9,11], an alternate approach is used where Gaussian mixtures, represented as a list of parameters, (i.e., means, variances and weights/coefficients) are passed between the nodes during message passing.

A Gaussian mixture propagated during the iterative decoding is denoted by

$$GM(t) = \sum_{k=1}^{N} \frac{c_k}{\sqrt{2\pi V_k}} e^{-\frac{(t-m_k)^2}{2V_k}}. \tag{2.40}$$

Here N is the number of Gaussian components in the mixture, m_k, V_k and $c_k \geq 0$ are the mean, variance and mixing coefficient/weight of the k^{th} component. A Gaussian mixture can then be efficiently represented by a set of triples $\{(m_1, V_1, c_1), \ldots, (m_N, V_N, c_N)\}$. If the coefficients sum to 1, i.e., $\sum_{k=1}^{N} c_k = 1$, then the Gaussian mixture is normalized.

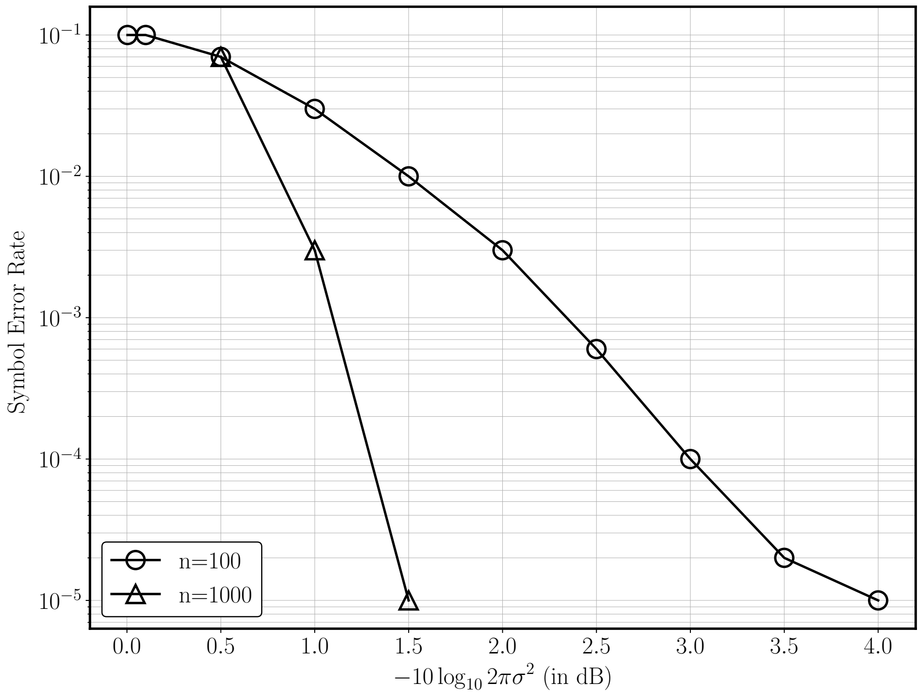


Figure 2.8: Symbol error rate versus distance from capacity, $-10 \log 102\pi e^2$ (in dB) of the sampled PDF LDLC decoder for block length, $n = 100$, $d = 5$ and $n = 1000$, $d = 7$ with 200 decoding iterations simulated by Sommer et al.

In parametric decoders, the continuous functions generated in the iterative decoding are Gaussian mixtures represented by a set of triples. Along with the efficient message representation, Gaussian mixture reduction algorithms are used to reduce the size of the messages after each decoding iteration. These techniques reduce the message size and computational complexity of the decoder significantly.

The following section describes the iterative decoding algorithm and the Gaussian mixture reductions techniques applied in the parametric decoders.

Iterative Decoding Algorithm for Parametric Decoders

Initialization At the start of the decoding process, variable node x_k sends the received channel message, represented by a triple $(m_0, V_0, 1)$, to all the connected check nodes where m_0 is y_k and V_0 is channel variance, σ^2.

Basic Iteration: Check Node Message In the parametric iterative decoding, the incoming variable node messages (Gaussian mixtures) are triples of means, variances and weights of Gaussian components and the outgoing message is computed by convolution of the Gaussian mixtures.

Convolution of Gaussian Mixtures: Suppose that two Gaussian mixtures $f(t)$ and $g(t)$ are:

$$f(t) = \sum_{i=1}^{F} \frac{c_{1i}}{\sqrt{2\pi V_{1i}}} e^{-\frac{(t-m_{1i})^2}{2V_{1i}}}. \tag{2.41}$$

$$g(t) = \sum_{i=1}^{G} \frac{c_{2i}}{\sqrt{2\pi V_{2i}}} e^{-\frac{(t-m_{2i})^2}{2V_{2i}}}. \tag{2.42}$$

The result of $f(t) * g(t)$ is then also a mixture, $h(t)$ which consists of $F \times G$ components:

$$h(t) = f(t) * g(t) = \sum_{i=1}^{F} \frac{c_{1i}}{\sqrt{2\pi V_{1i}}} e^{-\frac{(t-m_{1i})^2}{2V_{1i}}} * \sum_{j=1}^{G} \frac{c_{2j}}{\sqrt{2\pi V_{2j}}} e^{-\frac{(t-m_{2j})^2}{2V_{2j}}}, \tag{2.43}$$

$$= \sum_{i=1}^{F} \sum_{j=1}^{G} \frac{c_{1i}}{\sqrt{2\pi V_{1i}}} \frac{c_{2j}}{\sqrt{2\pi V_{2j}}} e^{-\frac{(t-m_{1i})^2}{2V_{1i}}} * e^{-\frac{(t-m_{2j})^2}{2V_{2j}}}. \tag{2.44}$$

As shown in (2.44), the convolution of two Gaussian mixtures is calculated by convolving each possible pair of Gaussians between the two mixtures. This simplifies the convolution of Gaussian mixtures to convolutions of pairs of Gaussians.

The convolution of two Gaussian mixture components is again a Gaussian. If two Gaussian components with triples (m_1, V_1, c_1) and (m_2, V_2, c_2) are convolved, the resultant Gaussian is represented by the triple (m_r, V_r, c_r) where,

$$m_r = m_1 + m_2, \tag{2.45}$$

$$V_r = V_1 + V_2, \tag{2.46}$$

$$c_r = c_1 c_2. \tag{2.47}$$

The periodic extension step (2.28) is performed on the Gaussian mixture $h(t)$ computed in (2.44).

Basic Iteration: Variable Node Message As mentioned in Section 2.4, in the parametric iterative decoding at each variable node, x_k, the channel message $(y_k, \sigma^2, 1)$ and the incoming check node messages (represented by set of triples of means, variances and weights) are multiplied. In order to understand the operations at the variable node let's look into the multiplication of two Gaussian mixtures in detail.

Product of Gaussian Mixtures: Given two Gaussian mixtures $f(t)$ and $g(t)$ given by:

$$f(t) = \sum_{i=1}^{F} \frac{c_{1i}}{\sqrt{2\pi V_{1i}}} e^{-\frac{(t-m_{1i})^2}{2V_{1i}}} \tag{2.48}$$

$$g(t) = \sum_{i=1}^{G} \frac{c_{2i}}{\sqrt{2\pi V_{2i}}} e^{-\frac{(t-m_{2i})^2}{2V_{2i}}}, \tag{2.49}$$

the result of the product $f(t)g(t)$ is then

$$f(t)g(t) = \sum_{i=1}^{F} \frac{c_{1i}}{\sqrt{2\pi V_{1i}}} e^{-\frac{(t-m_{1i})^2}{2V_{1i}}} \sum_{j=1}^{G} \frac{c_{2j}}{\sqrt{2\pi V_{2j}}} e^{-\frac{(t-m_{2j})^2}{2V_{2j}}}, \tag{2.50}$$

$$= \sum_{i=1}^{F} \sum_{j=1}^{G} \frac{c_{1i}}{\sqrt{2\pi V_{1i}}} \frac{c_{2j}}{\sqrt{2\pi V_{2j}}} e^{-\frac{(t-m_{1i})^2}{2V_{1i}}} e^{-\frac{(t-m_{2j})^2}{2V_{2j}}}. \tag{2.51}$$

As shown in (2.51), the product of two Gaussian mixtures is calculated by multiplying each possible pair of components between the two mixtures. This simplifies the product of Gaussian mixtures to the product of pairs of Gaussians.

The product of two Gaussians is a scaled Gaussian. If two Gaussians with triples (m_1, V_1, c_1) and (m_2, V_2, c_2) are multiplied and the resultant Gaussian is denoted by the triple (m_m, V_m, c_m), then:

$$V_m = \frac{V_1 V_2}{V_1 + V_2}, \tag{2.52}$$

$$m_m = V_m\left(\frac{m_1}{V1} + \frac{m_2}{V_2}\right), \tag{2.53}$$

$$c_m = \frac{c_1 c_2}{\sqrt{2\pi(V_1 + V_2)}} e^{\frac{-(m_1-m_2)^2}{2(V_1+V_2)}}. \tag{2.54}$$

Final Decision Similar to Section 2.4, in this step of the parametric iterative decoding, at every variable node the channel message and all the incoming check node messages (without omitting any) are multiplied together. Using the resultant triples, a quantized function for $f_{x_k}(x_k \mid \underline{y})$ is calculated. The peak value of this quantized function is the estimated codeword symbol $\hat{w}_k$.

Gaussian Mixture Reduction (GMR)

As discussed in Section 2.5.2, after the operations at the check and variable nodes during the iterative decoding process, Gaussian mixture messages are generated. The number of components in the Gaussian mixtures grows exponentially as a function of the number of decoding iterations. In order to achieve a feasible decoder with a limited storage and computational load, it is important to limit the number of components in the Gaussian mixtures by reducing the mixtures.

Although there are several Gaussian mixture reduction techniques [65], in this thesis we describe two Gaussian mixture reduction methods which have been used in parametric LDLC decoders [9–12]:

- Gaussian mixture reduction using greedy algorithm,
- Gaussian mixture reduction by merging Gaussians in a range.

Gaussian Mixture Reduction Using Greedy Algorithm:

This Gaussian mixture reduction method uses a greedy algorithm approach to reduce a Gaussian mixture to a mixture with fewer components.

It is based on the idea of minimizing the squared distance (SD) between two distributions $p(t)$ and $q(t)$; here $p(t)$ is original Gaussian mixture and $q(t)$ is a reduced mixture, which is an approximation of $p(t)$ with a fewer components. The squared distance between two distributions $p(t)$ and $q(t)$ is given as

$$SD(p(t), q(t)) = \int_{-\infty}^{\infty} (p(t) - q(t))^2 dt. \tag{2.55}$$

To obtain the reduced mixture, first the penalty (squared distance) for merging each possible pair of components in the mixture and replacing the pair with a single-Gaussian is computed. Whichever pair has the lowest penalty is substituted by a single Gaussian in the mixture. The components of the Gaussian mixture are merged in this greedy fashion until the desired number of components in the reduced Gaussian mixture are reached [10, 12].

Assume $p(t)$ is a normalized Gaussian mixture with two components denoted by triples $\{(m_1, V_1, c_1), (m_2, V_2, c_2)\}$, i.e., $c_1 + c_2 = 1$. The single Gaussian approximation, $q(t)$, for these two components is represented by a triple $(m_{\mathcal{MM}}, V_{\mathcal{MM}}, c_{\mathcal{MM}})$ that minimizes the squared distance between $p(t)$ and $q(t)$. In the special case of Gaussian mixtures, the squared distance between $p(t)$ and $q(t)$ is called *Gaussian quadratic loss* (*GQL*).

As shown in [66] the optimal parameters of the triple can be obtained using the *moment matching* method, i.e., moment matching minimizes the squared distance between the mixture with 2 components and the reduced single-Gaussian. After moment matching, the parameters of the reduced Gaussian are:

$$m_{\mathcal{MM}} = \sum_{k=1}^{2} c_k m_k, \tag{2.56}$$

$$V_{\mathcal{MM}} = \sum_{k=1}^{2} (c_k V_k + c_k (m_k - m_{\mathcal{MM}})^2), \tag{2.57}$$

$$c_{\mathcal{MM}} = c_1 + c_2. \tag{2.58}$$

The penalty for merging the two Gaussian components to a single Gaussian using moment matching is given by

$$\begin{aligned} SD(p(t), q(t)) = &\frac{1}{2\sqrt{\pi V}} + \frac{c_1^2}{2\sqrt{\pi V_1}} + \frac{c_2^2}{2\sqrt{\pi V_2}} - \frac{2c_1}{\sqrt{2\pi(V+V_1)}} e^{-\frac{(m-m_1)^2}{2(V+V_1)}} \\ &- \frac{2c_2}{\sqrt{2\pi(V+V_2)}} e^{-\frac{(m-m_2)^2}{2(V+V_2)}} + \frac{2c_1 c_2}{\sqrt{2\pi(V_1+V_2)}} e^{-\frac{(m_1-m_2)^2}{2(V_1+V_2)}}. \end{aligned} \tag{2.59}$$

The derivation of (2.59) is provided in the Appendix A.

The steps for the greedy GMR algorithm are as follows.

The input to the Gaussian mixture reduction algorithm is a Gaussian mixture of N components, `InputMix`, defined as

$$\texttt{InputMix} = (t_1, t_2, \ldots, t_N), \tag{2.60}$$

where $t_l = (m_l, V_l, c_l)$ is the l^{th} component of the Gaussian mixture, `InputMix`.

Algorithm 2: Pseudo Code for GMR using greedy algorithm

```
while $|\texttt{InputMix}| > M$ do
    • Calculate $GQL$ for each pair of components in InputMix and determine the pair
      $(t_i, t_j)$ with the smallest $GQL$
          $(t_i, t_j) = \operatorname*{argmin}_{t_i, t_j \in \texttt{InputMix}, i \neq j} GQL(t_i, t_j)$
    • Group these two components $(t_i, t_j)$ into a single component $t'$ according to second
      moment matching.
    • Add this triple $t'$ to InputMix and erase $t_i, t_j$ from InputMix.
```

The algorithm output is a reduced Gaussian mixture with M components, `ReducedMix`, given as

$$\texttt{ReducedMix} = (t_1, t_2, \ldots, t_M). \tag{2.61}$$

Algorithm 2 provides the pseudo code for this Gaussian mixture reduction method. At the end of the while loop, input mixture `InputMix` is reduced to mixture `ReducedMix` that contains M Gaussian components.

Parametric decoders in [10–12] have reduced the Gaussian mixture message to a smaller mixture with fewer components using this Gaussian mixture reduction method. The performance of the decoder, where the reduced mixture contains two components, is close to the sampled PDF decoder [7] as shown in Fig. 2.9.

Gaussian Mixture Reduction by Merging Gaussians in a Range:

This Gaussian mixture reduction method uses LDLC propagation properties in order to group Gaussian components in the mixture efficiently. This method has two steps.

First, the strongest component in the mixture is identified, where the strength of a component is defined by the *weight to standard deviation ratio*. For any component in a Gaussian mixture, denoted by a triple (m_i, V_i, c_i), the *weight to standard deviation ratio* computed as

$$\textit{weight to standard deviation ratio} = \frac{c_i}{\sqrt{V_i}}. \tag{2.62}$$

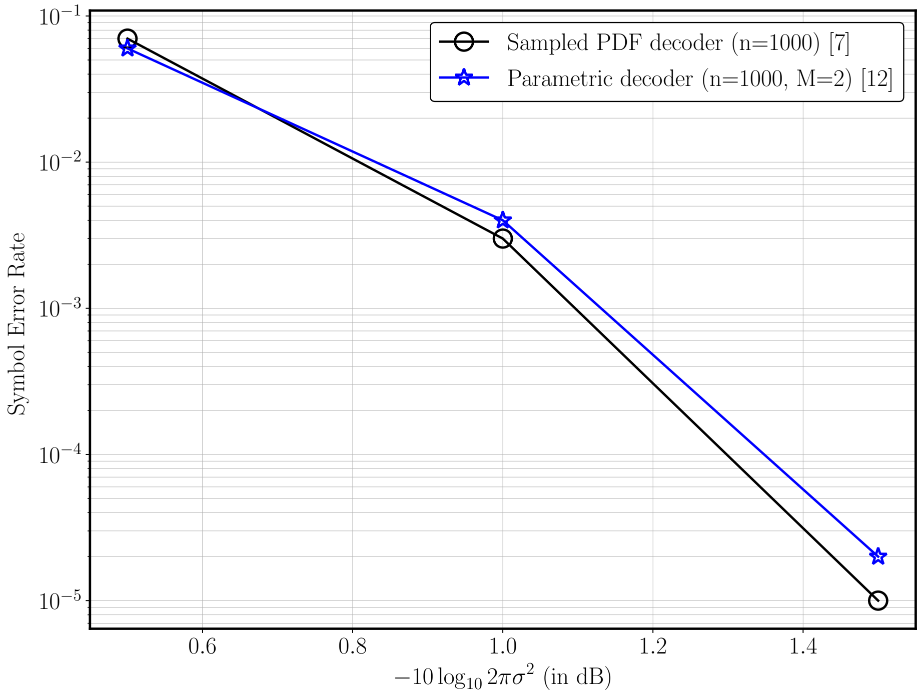


Figure 2.9: Symbol error rate vs. distance from capacity, $-10\log_{10} 2\pi e\sigma^2$ (in dB) for the parametric LDLC decoder using Gaussian mixture reduction method based on greedy algorithm with $n = 1000$, $d = 7$ and number of decoding iterations = 200.

Further, the mixture components that are within a certain range of the strongest Gaussian in the mixture are found. These components are reduced according to some rule which provides a good approximation of the original mixture. Usually a single Gaussian is obtained through *moment matching*.

Approximation using Moment Matching: If the strongest Gaussian and other components which are within a range from the strongest Gaussian, are denoted by the mixture:

$$GM_R(t) = \sum_{k=1}^{L} \frac{c_k}{\sqrt{2\pi V_k}} e^{-\frac{(t-m_k)^2}{2V_k}}, \tag{2.63}$$

the single Gaussian which provides a good approximation of $GM_R(t)$ is the second moment matched Gaussian, i.e., the Gaussian with same mean and variance [67]. The mean, variance and weight of this single Gaussian is denoted as $m_{\mathcal{MM}}$, $V_{\mathcal{MM}}$ and $c_{\mathcal{MM}}$ respectively.

To get this single Gaussian, mixture $GM_R(t)$ is first normalized as

$$\widetilde{GM_R}(t) = \sum_{k=1}^{L} \frac{b_k}{\sqrt{2\pi V_k}} e^{-\frac{(t-m_k)^2}{2V_k}}, \tag{2.64}$$

where $b_k = \frac{c_k}{\sum_{l=1}^{L} c_k}$. The parameters $m_{\mathcal{MM}}, V_{\mathcal{MM}}$ and $c_{\mathcal{MM}}$ are then calculated as,

$$m_{\mathcal{MM}} = \sum_{k=1}^{L} b_k m_k \tag{2.65}$$

$$V_{\mathcal{MM}} = \sum_{k=1}^{L} (b_k V_k + b_k (m_k - m_{\mathcal{MM}})^2) \tag{2.66}$$

$$c_{\mathcal{MM}} = \sum_{k=1}^{L} c_k. \tag{2.67}$$

The steps of GMR using moment matching are as follows.

The input Gaussian mixture consists of N components and is denoted by `InputMix`. Mixture `InputMix` can be entirely described by a set of triples as,

$$\texttt{InputMix} = \{(m_1, V_1, c_1), (m_2, V_2, c_2) \cdots (m_N, V_N, c_N)\}. \tag{2.68}$$

The output is a reduced Gaussian mixture denoted by `ReducedMix`, that would contain M components and is represented by a list of triples as

$$\texttt{ReducedMix} = \{(m_1, V_1, c_1), (m_2, V_2, c_2), \cdots, (m_M, V_M, c_M)\}. \tag{2.69}$$

Algorithm 3 then provides the pseudo code for this Gaussian mixture reduction method.

In [9] the Gaussian mixture messages in iterative decoding are reduced to a smaller mixture using the range based algorithm. The decoder performance is simulated for different block lengths of up to $n = 10^6$ and code degree of $d = 7$, for a different number of components in the reduced mixture. Experimental results have shown that the decoder attains essentially the same performance as the sampled PDF decoder [7]. This algorithm shows better storage requirements and relatively lower computational complexity compared to the sampled PDF decoder. Fig. 2.10 compares this decoder performance for $M = 6$ and $M = 2$ to the sampled PDF LDLC decoder for block length of 1000 and degree, 7.

There are other efficient Gaussian mixture reduction techniques available in literature, e.g., [68]. The technique in [68] applies to the Gaussian mixtures with two mixture components only while the work in this thesis explores a larger design space.

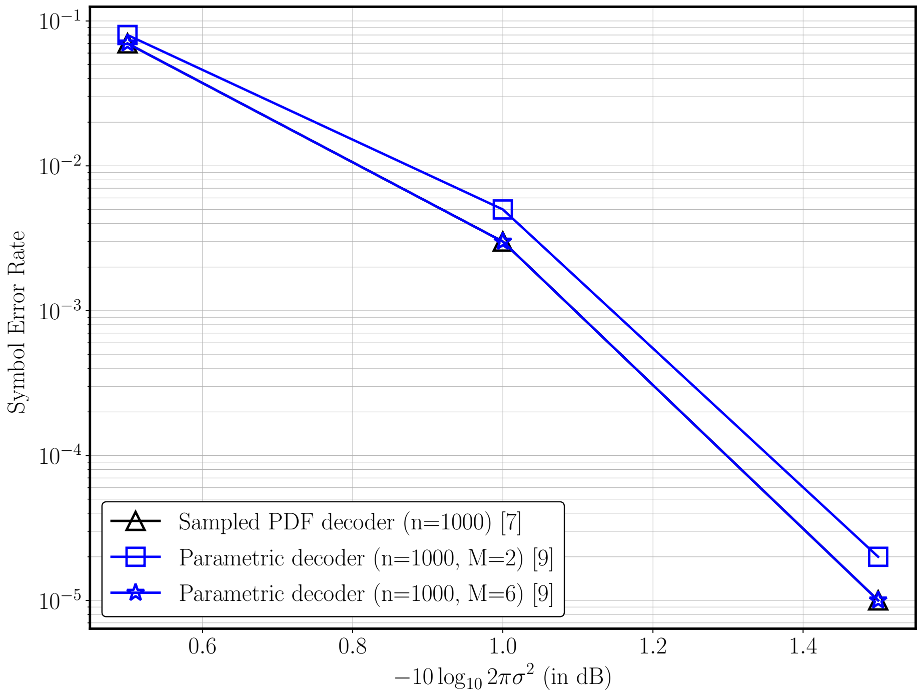


Figure 2.10: Symbol error rate vs. the distance from capacity, $-10\log_{10} 2\pi e\sigma^2$, (in dB) using Gaussian mixture reduction method based on merging Gaussians in a range for $n = 1000$, $d = 7$ with number of decoding iterations = 200.

2.6 Summary

This chapter we first described the basic definitions and properties of LDLCs and then an iterative decoding algorithm where the exchanged messages were continuous functions was presented. We also provided an overview of the LDLC decoders available in the literature, primarily categorized as sampled PDF LDLC decoder and parametric LDLC decoders. We detailed the two GMR techniques used in the parametric LDLC decoders, i.e. GMR using greedy algorithm and GMR by merging Gaussians in a range.

The background knowledge and the survey of available literature enables us to pick an appropriate LDLC decoder for the hardware implementation. Further, in the thesis we present a fixed-point implementation of the LDLC decoding on an Intel FPGA.

Algorithm 3: Pseudo Code for GMR by Merging Gaussians in a Range

```
while (|InputMix| > 0 or |ReducedMix| < M) do
    • Find the component $k$ in InputMix that has maximum *weight to standard deviation ratio*,
          $$k_{max} = \underset{k}{\text{argmax}} \frac{c_k}{\sqrt{V_k}}.$$
    • Define a range of length 2A around $m_{k_{max}}$, the mean of the component that has the maximum *weight to standard deviation ratio*,
          $$range = [m_{k_{max}} - A, m_{k_{max}} + A].$$
    • Find the components in mixture that fall in this range,
          $$\mathcal{S} = \{(c_k, m_k, V_k) \mid |(m_k - m_{k_{max}})| < A\}.$$
    • Calculate the single Gaussian $(m_{\mathcal{MM}}, V_{\mathcal{MM}}, c_{\mathcal{MM}})$ for $\mathcal{S}$ according to the (2.65), (2.66) and (2.67).
    • Add the single Gaussian calculated in step 5 to the reduced mixture, ReducedMix,
          $$\texttt{ReducedMix} = \texttt{ReducedMix} \cup (m_{\mathcal{MM}}, V_{\mathcal{MM}}, c_{\mathcal{MM}})$$
    • Erase the components $\mathcal{S}$ from InputMix, i.e.,
          $$\texttt{InputMix} = \texttt{InputMix} - \mathcal{S}.$$
```

Chapter 3

Single-Gaussian LDLC Decoder Implementation

In Chapter 2, we discussed an iterative decoding algorithm for low-density lattice codes, where the messages exchanged between check nodes and variable nodes are continuous functions (probability density functions). We also discussed a parametric decoding approach, where messages are represented as lists of Gaussian parameters, i.e., means, variances and coefficients. Then, Gaussian mixture reduction algorithms used in parametric decoders were introduced.

In order to achieve successful proof-of-concept hardware for LDLC decoding, it is important to select an appropriate decoder architecture, whose impact on resource usage must be well understood.

3.1 Selection of an LDLC Decoder for Hardware Implementation

We begin our search for a suitable hardware architecture by comparing several decoding algorithms in terms of computational complexity and performance.

In the sampled PDF LDLC decoder presented in [7], PDFs are sampled and discretely quantized; the decoder shows the best reported performance for an LDLC decoder, but its cost is high, especially in terms of data storage. We consider this decoder the baseline for the comparison of LDLC decoding performance, as it performs sampling of the continuous message functions, i.e., a close approximation of the real message. Parametric LDLC decoders described in Subsection 2.5.2 reduce the complexity of the sampled decoder and offer a less costly implementation with reduced storage requirements.

A parametric LDLC decoder proposed by Kurkoski et al. [11] approximates the PDF of *each* message by a Gaussian mixture with $M = 100$ and $M = 10$ and for a block length of up to 10,000. The decoder uses a Gaussian mixture reduction method based on a greedy algorithm, described in Subsection 2.5.2. Simulation results show that the performance loss for this decoder is no greater than 0.2 dB compared to a sampled PDF LDLC decoder. Figure 3.1 shows a performance comparison for the sampled PDF and the parametric LDLC decoder with number of components in the reduced mixture set to $M = 10$.

Though the cost of the parametric LDLC decoder proposed in [11] is still high, it can be argued that a parametric implementation of an LDLC decoder is more feasible, as its performance is comparable to the sampled PDF decoder but with reduced storage requirements.

We now present a review of other parametric approaches in the literature.

Performance: The decoder in [9] performs Gaussian mixture reduction by merging Gaussian components within a specified range, and achieves an SER comparable to the sampled PDF decoder of [7] for $M = 6$, $d = 7$ and block length up to 100,000. For $M = 2$, it shows no degradation for block length of 100 and $d = 5$; however it suffers a slight degradation that varies between 0.1 and 0.2 dB for block lengths of 1000 and greater with $d = 7$.

In [12], a decoder based on Gaussian mixture reduction using a greedy algorithm is presented. For this decoder, when $M = 2$, the performance losses are no more than 0.1 dB for block lengths of $n = 1000$ and $n = 10,000$ with degree $d = 7$, while for $n = 100$ (and $d = 3$), this may increase to 0.3 dB at low SNR. Further, $M = 1$, i.e, when the messages exchanged between the check nodes and variable nodes are reduced to a single Gaussian, led to a reduced memory requirement for the decoder but at a significant performance loss. For $n = 100$ and $d = 3$, the performance loss is approximately 1 dB, as compared to the sampled PDF decoder.

Complexity: The complexity of the Gaussian mixture reduction (GMR) method that merges Gaussian components within a specified range depends on sorting mixture components in the order of their strengths, i.e., *the amplitude to standard deviation ratio*, finding

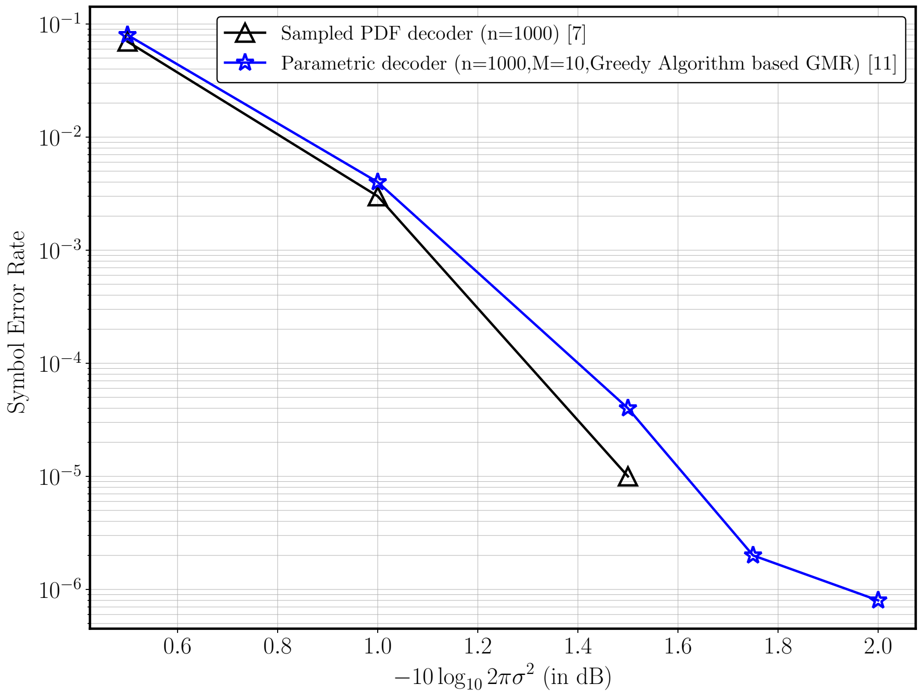


Figure 3.1: Comparison of symbol error rate vs. distance from capacity, $-10\log_{10} 2\pi e\sigma^2$ (in dB) between the sampled PDF decoder for block length, $n = 1000$, $d = 7$ and a parametric decoder for $n = 1000$, number of components $(M) = 10$ and $d = 6$.

the Gaussian components that are within a certain range of the strongest component, and grouping them together into a single Gaussian, as discussed in Section 2.5.2. The computational complexity of this operation is proportional to the number of Gaussian mixture mergers performed during the approximation, i.e., $\leq M^2$.

As described in Subsection 2.5.2, the complexity of the GMR method based on a greedy algorithm lies in going over the complete Gaussian mixture and finding the pair of components with least GQL. The computational complexity of this method is $O(M^4)$, where M is the number of Gaussian components in the reduced mixture. This can be explained as follows: the operations performed at the check and variable nodes during iterative decoding (convolution and multiplication, respectively) create an output of M^2 components for a message size of M components. If there is a Gaussian mixture with k components, we calculate the GQL between each pair of k components. Thus, the complexity of operation is $O(k^2)$, so when $k = M^2$, the overall complexity is proportional to M^4.

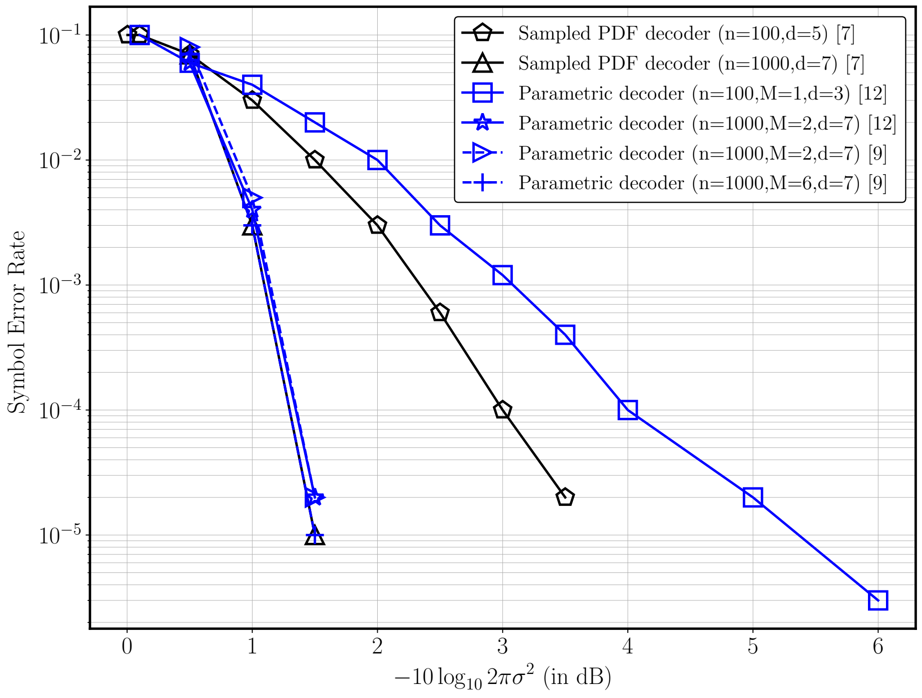


Figure 3.2: Comparison of performance between the sampled PDF and parametric LDLC decoders for block length 100 and 1000.

Fig. 3.2 depicts the performance of the sampled PDF decoder [7] and parametric decoders [9], [12] for different block lengths. Table 3.1 compares the complexity and performance of various parametric LDLC decoders with respect to the sampled PDF decoder [7].

Since the focus of this thesis is to achieve a successful hardware implementation of an LDLC decoder within the limited resources of a target FPGA, we choose the parametric LDLC decoder with $M = 1$ for the first baseline hardware implementation. For clarity, this implementation is referred to as *single-Gaussian LDLC decoder* in this thesis.

The next section describes the iterative decoding algorithm for the single-Gaussian LDLC decoder.

Table 3.1: Performance and computational complexity of various parametric LDLC decoders with respect to the sampled PDF decoder for different block lengths.

Decoder	**Reference**	**n**	**d**	M	**# of GM Merges**	**Performance**
I)	[9]	1000	7	6	$\leq M^2$	Identical
II)	[9]	1000	7	2	$\leq M^2$	$\tilde{0}$.1 - 0.2 dB (loss)
III)	[12]	1000	7	2	$O(M^4)$	$\tilde{0}$.1 dB (loss)
IV)	[12]	100	3	1	$O(M^4)$	$\tilde{1}$.0 dB (loss)

3.2 Iterative Decoding for the single-Gaussian LDLC decoder

For the single-Gaussian LDLC decoder implemented in this thesis, the messages are reduced to a single-Gaussian and only the mean and variance are exchanged. A single Gaussian is a special case of a Gaussian mixture defined by (2.40) when $N = 1$, and can therefore be represented by the triple (m, V, c). If the single Gaussian is normalized, i.e., $c = 1$, then this can be reduced to the mean-variance tuple (m, V).

During the periodic extension step described in Subsection 2.4, the single-Gaussian LDLC decoder generates Gaussian mixtures; however, these mixtures are reduced to a single normalized Gaussian before message passing. Thus, mean-variance pairs are exchanged as messages between check nodes and variable nodes. The decoding algorithm follows a flooding schedule where all the variable nodes, and subsequently all the check nodes, pass new messages to the connected nodes in each iteration [69].

The basic steps of the iterative decoding algorithm for a single-Gaussian decoder are presented below.

3.2.1 Initialization

At the start of the decoding process, each variable node x_k, receives a single-Gaussian message from the AWGN channel given by (m_0, V_0), i.e., (y_k, σ^2) where y_k is the mean and σ^2 is the variance of the received single Gaussian channel message. This initial message is sent along all the edges connected to this variable node as shown in Fig. 3.3 (for degree 3).

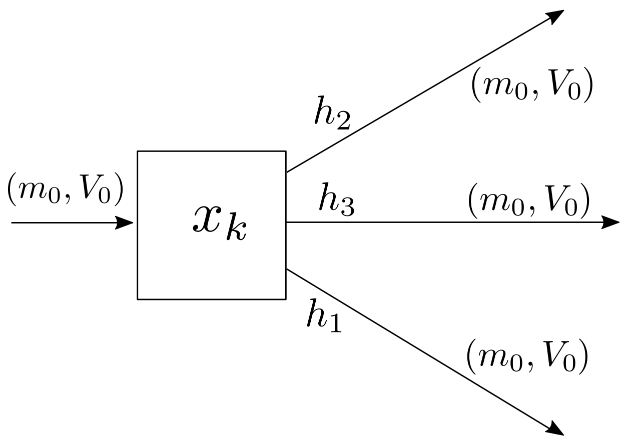


Figure 3.3: Initialization in iterative decoding - a variable node sends the message received from the channel to all connected check nodes.

3.2.2 Basic Iteration: Check Node Message

Each check node has d input messages coming along the edges connected to it with weights h_p, $p = 1, \ldots, d$ where h_p is one of the $\bar{h}$'s with a possible sign flip as shown in Fig. 3.4 for degree 3. The incoming messages are single Gaussians given by (m_ℓ, V_ℓ), where $\ell = 1, 2 \ldots, d$. The mean of the outgoing check node message along the edge with weight h_p is obtained by first multiplying for $\ell \neq p$, the mean of the ℓ^{th} message with $\frac{h_\ell}{h_p}$, then summing the results over $\ell \neq p$ and a sign flip. The variance of the outgoing check node message along the edge with weight h_p is obtained by first multiplying for $\ell \neq p$, the variance of the ℓ^{th} message with $\frac{h_\ell^2}{h_p^2}$, then summing the results over $\ell \neq p$. This outgoing message is the single-Gaussian $(\overline{m_p}, \overline{V_p})$ given by

$$\overline{m_p} = -\sum_{\ell \neq p} \frac{h_\ell m_\ell}{h_p}, \tag{3.1}$$

$$\overline{V_p} = \sum_{\ell \neq p} \frac{h_\ell^2 V_\ell}{h_p^2}. \tag{3.2}$$

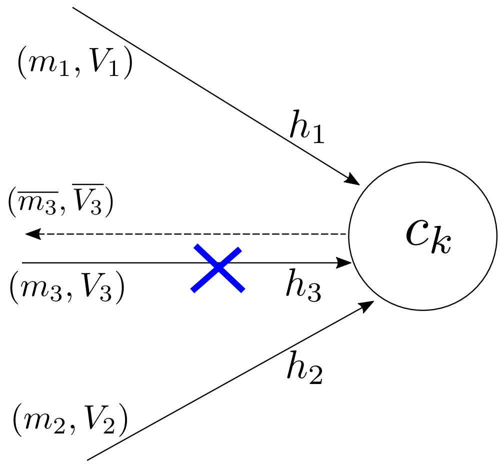


Figure 3.4: Illustration of all the incoming messages, and the outgoing message along the edge with weight h_3 at a check node. The outgoing message on the edge with weight h_3 is obtained by convolving all the incoming messages except the one on this edge.

3.2.3 Basic Iteration: Variable Node Message

Each variable node receives d single-Gaussian messages along the edges that can be denoted by $(\overline{m_\ell}, \overline{V_\ell})$ for $\ell = 1, 2 \ldots, d$. Fig. 3.5 shows the incoming messages at a variable node x_k, in a single-Gaussian decoder of degree 3.

There are two primary steps performed at the variable nodes: a 1) periodic extension step and a 2) product step.

1) The periodic extension step generates periodic Gaussian mixtures from the incoming messages. In [7], this step is performed as a part of check node operations and the variable node receives the periodically extended Gaussian mixtures. However, in the single Gaussian decoder, the periodic extension step occurs at the variable nodes and check nodes always send single-Gaussian messages to the variable nodes [12]. This significantly reduces the storage requirements for the check node messages.

In the periodic extension step, the mean of the incoming check node message along an edge with weight h_l is first periodically extended as below,

$$\overline{m_\ell}(i) = \overline{m_\ell} + \frac{i}{h_\ell}, \tag{3.3}$$

where i denotes the i^{th} extension. In principle, the variable i in (3.3) can take any integer value, but in practice the range is restricted within a certain limit, i.e. $i \in \mathcal{L} \subset \mathbb{Z}$ to reduce

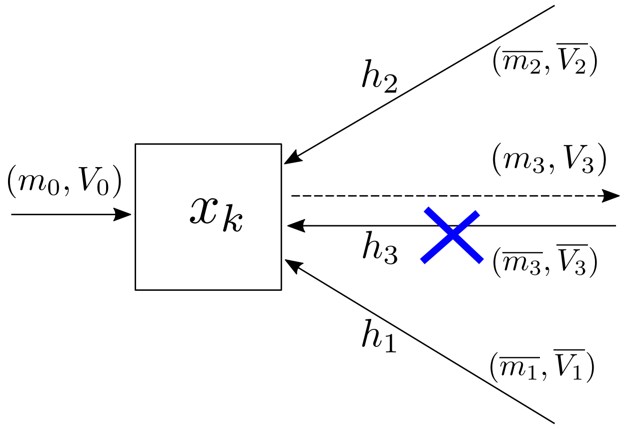

h_2
$(\overline{m_2}, \overline{V_2})$
(m_3, V_3)
(m_0, V_0)
x_k
h_3
$(\overline{m_3}, \overline{V_3})$
h_1
$(\overline{m_1}, \overline{V_1})$

Figure 3.5: Illustration of all the incoming messages and the outgoing message along the edge with weight h_3 at a variable node, x_k. Here the outgoing message on the edge with weight h_3 is obtained by multiplying all the incoming messages except the one on this edge.

the complexity of the decoder.

2) The outgoing variable node message along the edge with weight h_p is computed by taking the product of the channel message, denoted by (m_0, V_0), and all of the Gaussian mixtures obtained after the periodic extension step, except the mixture associated on that edge. The product result is then further reduced to a single-Gaussian using the second moment-matching-method [65].

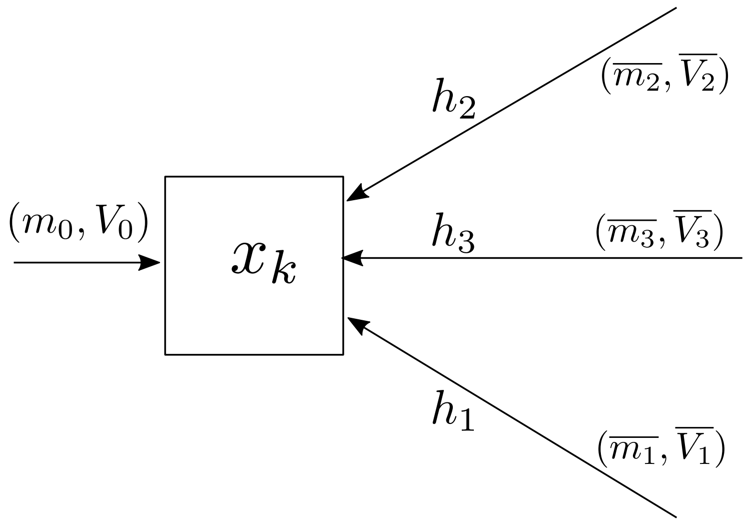
The product of two Gaussian mixtures is calculated by the pair-wise multiplication of each possible pair of components between the two mixtures, as discussed in Subsection 2.5.2. If two Gaussian components with triples $(\tilde{m}_1, \tilde{V}_1, \tilde{c}_1)$ and $(\tilde{m}_2, \tilde{V}_2, \tilde{c}_2)$ are multiplied, the resultant Gaussian is given by the triple (m_F, V_F, c_F) calculated as,

$$V_F = \frac{\tilde{V}_1 \tilde{V}_2}{\tilde{V}_1 + \tilde{V}_2}, \tag{3.4}$$

$$m_F = V_F \Big(\frac{\tilde{m}_1}{\tilde{V} 1} + \frac{\tilde{m}_2}{\tilde{V}_2} \Big), \tag{3.5}$$

$$c_F = \frac{\tilde{c}_1 \tilde{c}_2}{\sqrt{2\pi(\tilde{V}_1 + \tilde{V}_2)}} e^{\frac{-(\tilde{m}_1 - \tilde{m}_2)^2}{2(\tilde{V}_1 + \tilde{V}_2)}}. \tag{3.6}$$

The product step at the variable node generates a Gaussian mixture that must be reduced to a single Gaussian before it can be sent along an outgoing edge of the node. The



(m_0, V_0)
x_k
h_2
$(\overline{m_2}, \overline{V_2})$
h_3
$(\overline{m_3}, \overline{V_3})$
h_1
$(\overline{m_1}, \overline{V_1})$

Figure 3.6: Final decision at variable node x_k - channel message and all the incoming check node messages are multiplied (without omitting any).

single Gaussian approximation for the Gaussian mixture is computed using the second-moment-matching method described in Subsection 2.5.2.

For a Gaussian mixture message denoted by triples of mean, variance and mixing coefficients, i.e., by $\{(m_1,V_1,\ c_1),\ \ldots,\ (m_N, V_N, c_N)\}$, the second-moment-matched single Gaussian, $(m_{\mathsf{MM}}, V_{\mathsf{MM}})$ is obtained by first normalizing the mixture according to $r_k = c_k/(\sum_{k=1}^{N} c_k)$, and then parameters m_{MM} and V_{MM} are calculated as

$$m_{\mathsf{MM}} = \sum_{k=1}^{N} r_k m_k, \tag{3.7}$$

$$V_{\mathsf{MM}} = \sum_{k=1}^{N} (r_k V_k + r_k (m_k - m_{\mathsf{MM}})^2). \tag{3.8}$$

In order to compute the variable node message efficiently, a forward-backward recursive algorithm is used [11].

Let's denote the periodically extended messages with $\texttt{MPeriodic}_\ell$ where $\ell = 1, 2 \ldots, d$ and the Gaussian mixture reduction to a single Gaussian (including the normalization step) by $\texttt{GMR}$. The pseudo code for this forward-backward recursion algorithm is given in

Algorithm 4. The algorithm is initialized with the channel message. Here "$\cdot$" denotes product of Gaussian mixtures.

Once the forward-backward messages, FW_ℓ and BW_ℓ for $\ell = 1, 2 \ldots, d$ are computed, the outgoing variable node messages, i.e., (m_ℓ, V_ℓ) for $\ell = 1, 2 \ldots, d$ are obtained as

$$(m_\ell, V_\ell) = FW_\ell \cdot BW_\ell. \tag{3.9}$$

3.2.4 Final Decision

After every iteration, to get $\hat{\underline{b}}$, i.e., an estimate the integer vector $\underline{b}$, the channel message and all incoming check node messages, shown in Fig. 3.6 are multiplied at each variable node.

However, $\hat{\underline{b}}$ is not estimated directly. To get $\hat{\underline{b}}$, first an estimate $\hat{w}_k$ of the transmitted codeword element x_k for $k = 1, 2 \ldots, n$ is computed as $FW_2 \cdot BW_1$. The variable, $\hat{w}_k$ is the mean of the single Gaussian obtained after the multiplication and moment matching step. Then $\hat{\underline{b}}$, is estimated as

$$\hat{\underline{b}} = \lfloor H \cdot \hat{\underline{w}} \rceil, \tag{3.10}$$

where $\lfloor \rceil$ denotes coordinate-wise integer rounding [70]. The decoded integer vector, $\hat{\underline{b}}$, is computed after every decoding iteration and the iterative decoding process is terminated as soon as an output is equal to the encoder input. This is to speed up the simulations with the assumption that further iterations would not diverge from the correct decision. In practice, a cyclic redundancy check could be used to validate that the decoder output is correct and terminate decoding early.

Early stopping reduces the average number of iterations required for decoding and is commonly used in iterative decoding [19, 29–31, 54, 71–74].

3.3 Frame Error Rate to Measure the Decoder Performance

In the literature, the LDLC decoder performance is measured by the symbol error rate. However, frame error rate (FER) provides a more effective way to measure the decoder performance, as even a single erroneous symbol in an entire frame counts as a frame error.

Algorithm 4: Forward-backward recursive algorithm

```
## initialization
$FW_1 = (m_0, 2V_0)$
$BW_d = (m_0, 2V_0)$
## main loop
for $j = 1$ to $d-1$ do
    $FW_{(j+1)} = \texttt{GMR}(FW_j \cdot \texttt{MPeriodic}_j)$
    $BW_{(d-j)} = \texttt{GMR}(BW_{(d-j+1)} \cdot \texttt{MPeriodic}_{(d-j+1)})$
end
```

For example, a frame of 10,000 bits is in error even if only one of the symbols is wrong, and if every such frame had exactly one incorrect symbol, then the FER would be 100% while the SER would be 10^{-4}. Usually, erroneous frames are discarded and a re-transmission is requested.

Therefore, in this thesis we measure the decoder performance by the frame error rate, and the included performance graphs are for frame error rate versus distance from capacity. As the simulation results in [12] only reports SER, to validate the performance of our *reference single-Gaussian LDLC decoder*, in Fig. 3.7 we show the SER of [12] as well as the SER and FER of our single-Gaussian decoder for block length of 100 and degree 3.

Fig. 3.8 shows the frame error rate of a single-Gaussian decoder for block length of 1000 and degree 3. The simulated results presented are for random lattice codewords in the integer range $\underline{b} \in \mathcal{L}^n$, where $\mathcal{L}$ = {-2, -1, 0, 1, 2} and generating sequence $\{1, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\}$.

3.4 Optimizations to Reduce the Decoder Complexity

3.4.1 Fixed-Point Arithmetic for Hardware Implementation

The LDLC decoder performance presented so far is for floating-point arithmetic. However, in the design space of this thesis we do not need floating-point precision and the decoder can be implemented on a fixed-point device (shown further in thesis). However, a key aspect of fixed-point arithmetic is to determine the range and precision requirements of the design.

In a fixed-point representation, every number has a fixed word length of W bits that consists of W_i integer bits, W_f fractional bits and a sign bit. In this thesis, our fixed-point

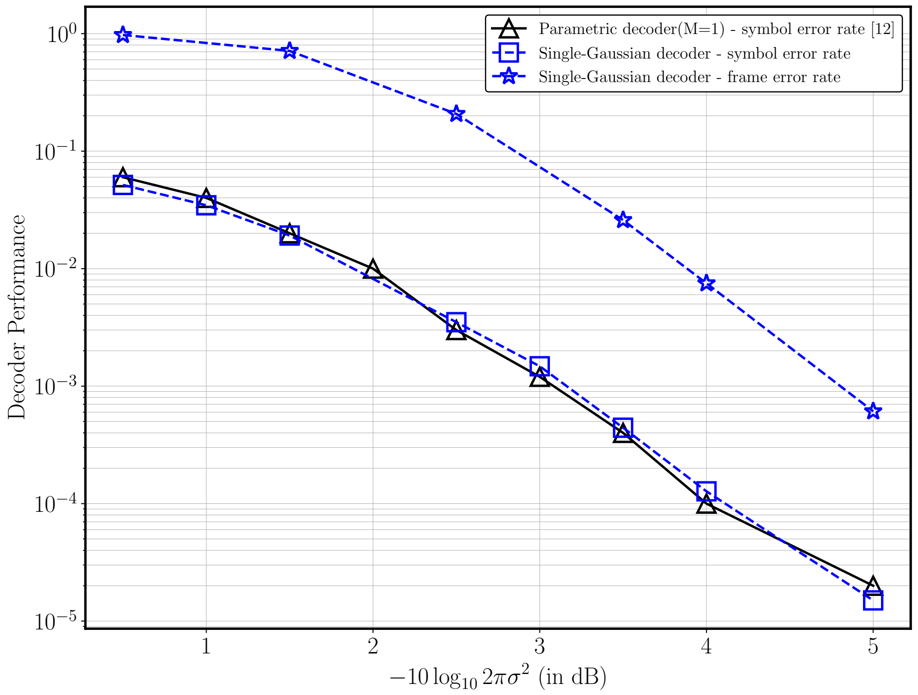


Figure 3.7: Comparison of symbol error rate between single-Gaussian LDLC decoder and parametric decoder, $M = 1$ for $n = 100$ and d=3. The corresponding frame error rate for the single-Gaussian LDLC decoder is also shown.

representation is denoted by $\text{Q}W_i.W_f$, e.g., Q10.8, where 10 bits represent the integer range and 8 bits are for the fractional precision, and one sign bit (19 bits total).

3.4.2 Minimum Variance

For improved numerical stability at the variable nodes, the smallest allowable variance is limited to a certain minimum value denoted by minvar. In the literature minvar is $0.03\sigma^2$ [9]; however, based on our simulations, minvar can be increased to $0.1\sigma^2$ without any loss in decoder performance as shown in Fig. 3.9. Here σ is the standard deviation of the Gaussian channel noise. In this thesis, any variance less than $0.1\sigma^2$ is increased back to $0.1\sigma^2$.

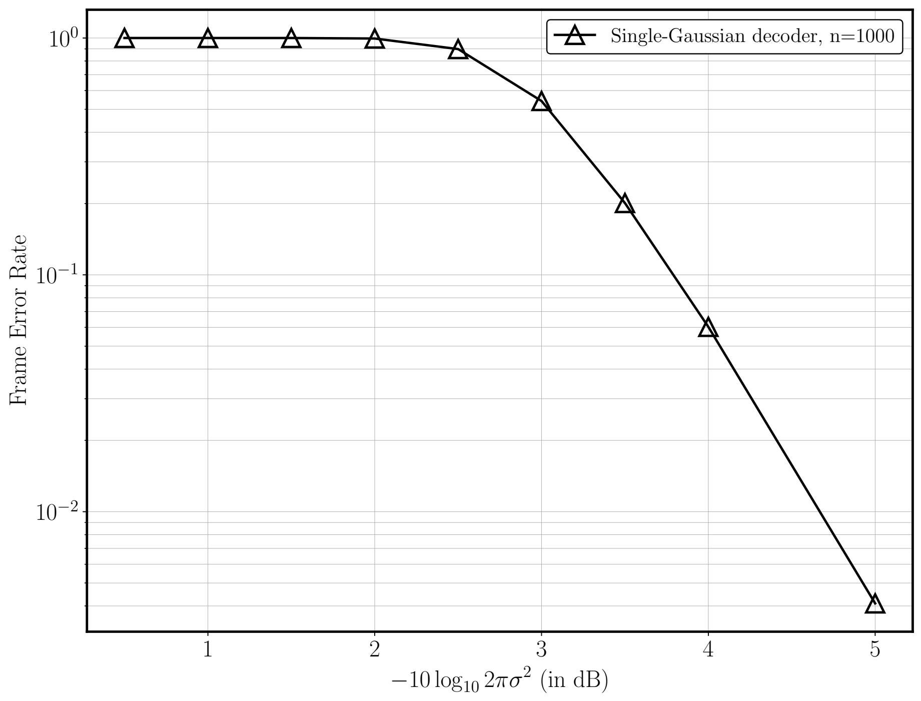


Figure 3.8: Frame error rate of the single-Gaussian LDLC decoder for $n = 1000$ and $d = 3$.

3.4.3 Variances Measured with Respect to the Channel Variance

All variances in this implementation are measured relative to the channel variance, e.g., for $V = 2$ in the implementation the actual variance is $2\sigma^2$. This is an important optimization to limit the fractional bits required for the fixed-point decoder and to achieve a lower power consumption and overall smaller cost of hardware implementation.

3.4.4 Optimizations for the Coefficient Computation at the Variable nodes

The computation of FW_j·`MPeriodic`$_j$ in Algorithm 4 computes the product of a single Gaussian (FW_j) with a Gaussian mixture (`MPeriodic`$_j$). The single Gaussian is normalized. Thus, it has a single component of weight '1'. The Gaussian mixture is obtained by periodically extending a normalized single Gaussian. Thus all the weights of the mixture are equal and are also '1'. In addition, all the variances of the mixture are equal to that

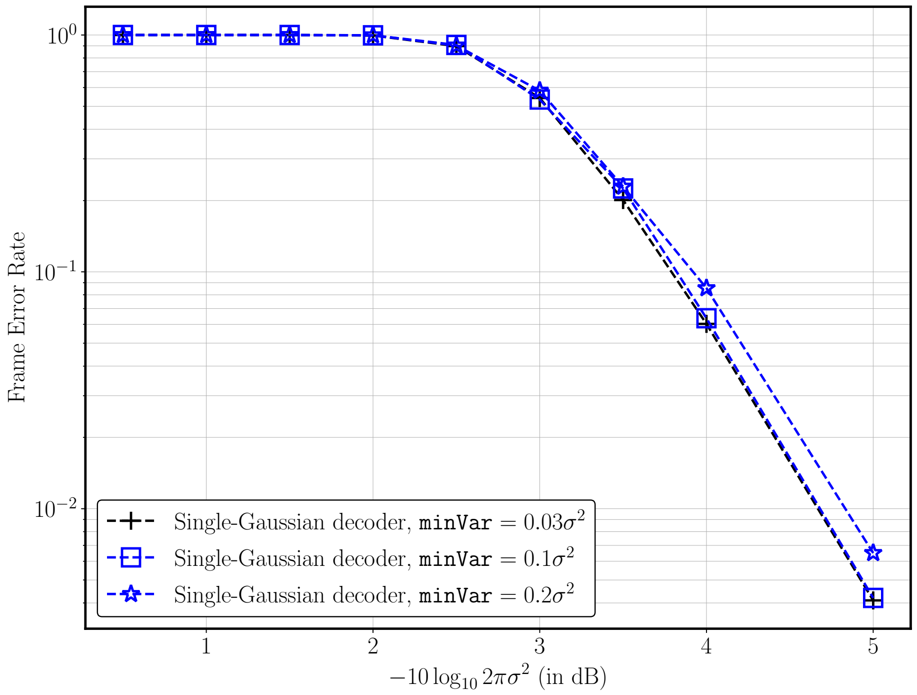


Figure 3.9: Frame error rate of the single-Gaussian decoder with different `minVar` values for $n = 1000$ and $d = 3$.

of the single Gaussian that was periodically extended and hence, are all equal. Therefore, the term $\frac{\tilde{c}_1\tilde{c}_2}{\sqrt{2\pi(\tilde{V}_1+\tilde{V}_2)}}$ in (3.6), which must be computed for each component in the product $FW_j\cdot$`MPeriodic`$_j$, is the same.

Since the components in the product are explicitly normalized in the Gaussian mixture reduction step that follows the computation of the product, to reduce complexity, for the computation of the product $FW_j\cdot$`MPeriodic`$_j$, the weights in (3.6) are instead replaced with

$$c_F = e^{\frac{-(\tilde{m}_1-\tilde{m}_2)^2}{2(\tilde{V}_1+\tilde{V}_2)}}. \tag{3.11}$$

Similarly, the weights in (3.6) are also replaced with (3.11) for the computation of the product $BW_{(d-j+1)}\cdot$ `MPeriodic`$_{(d-j+1)}$. To further improve the hardware efficiency, the exponent calculation in (3.11), is performed as,

$$c_F = e^{\frac{-(\tilde{m}_1-\tilde{m}_2)^2}{(\tilde{V}_1+\tilde{V}_2)}}, \tag{3.12}$$

and the factor of $\frac{1}{2}$ in the exponent of (3.11) is accommodated in the exponentiation function approximation described further in the chapter in Subsection 3.5.1.

For numerical stability, the coefficients computed during the product step at the variable nodes are scaled such that the largest exponential term in the mixture is '1'.

3.4.5 Number of Decoding Iterations

In [7, 9, 10, 12], the reported performance results for the LDLC decoders are with 200 decoding iterations. However, in order to obtain reasonable decoding latency and to limit the power consumption, fewer decoding iterations are preferred and determining a suitable number of decoding iterations is important for a feasible hardware implementation.

Fig. 3.10 shows the decoder performance versus the number of decoding iterations at a distance from capacity of 3.5 dB as well as 5 dB. As the graph suggests, with 20 decoding iterations, the decoder can achieve comparable performance to 200 decoding iterations, but in significantly less run time. Also, as described in Section 3.2, the decoding process is terminated early (in less than 20 iterations) as soon as an output is equal to the encoder input.

3.5 Fixed-Point Quantization Study

3.5.1 Approximation of Non-Linear Functions

The fixed-point implementation has two non-trivial non-linear functions: division and exponentiation.

Approximation of division function using Newton-Raphson method

A straightforward method to approximate division in fixed-point is integer long division. However, integer long division computation, i.e., $\texttt{Qdiv}(u, a) = (u \ll W_f)/a$ can be expensive in terms of time and hardware.

The division function has been approximated in the literature using Newton-Raphson methods as well as other techniques, e.g. polynomial approximation, that are simpler than integer long division [75, 76].

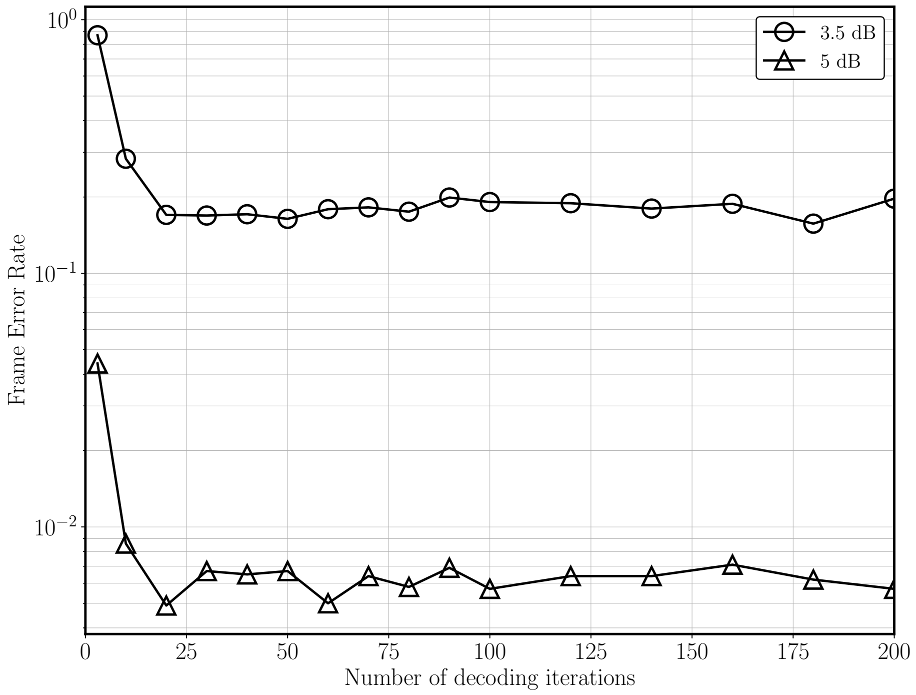


Figure 3.10: Performance of the single-Gaussian decoder for different number of decoding iterations at distance from capacity of 3.5 dB and 5 dB for $n = 1000$ and $d = 3$.

Using the Newton-Raphson (NR) method, the division function `Qdiv` can be implemented as

$$\texttt{Qdiv}(u, a) = \texttt{Qmul}(u, \texttt{NR\_reciprocal}(a)), \tag{3.13}$$

where `NR_reciprocal`(a) is the reciprocal of a calculated using the NR approach (shown in Appendix B), which is then multiplied with u using the fixed-point multiplication function, `Qmul`.

For the Newton-Raphson method, convergence to the correct solution depends critically on a reasonable initial guess. In a fixed-point decoder, this initial guess is obtained using a look-up table (LUT). To reduce the look-up table size and minimize approximation errors, we do not approximate the reciprocal of a, but instead, the fixed-point number a is written as $q \times (s \cdot 2^P)$ where P is an integer, q is ± 1 and s is a non-negative fixed-point number with $1 \leq s < 2$. The reciprocal of s is then calculated using `NR_reciprocal`. This reciprocal is multiplied with u, scaled back by 2^{-P} and further multiplied with q to get the value of

u/a.

In this method, the reciprocal of s is always in the range $0.5 < 1/s \leq 1$, which can be represented precisely enough with a small number of fractional bits.

The division function is thus implemented in the fixed-point LDLC decoder as,

$$\texttt{Qdiv}(u,a) = q \times (\texttt{Qmul}(u, \texttt{NR\_reciprocal}(s)) \gg P). \tag{3.14}$$

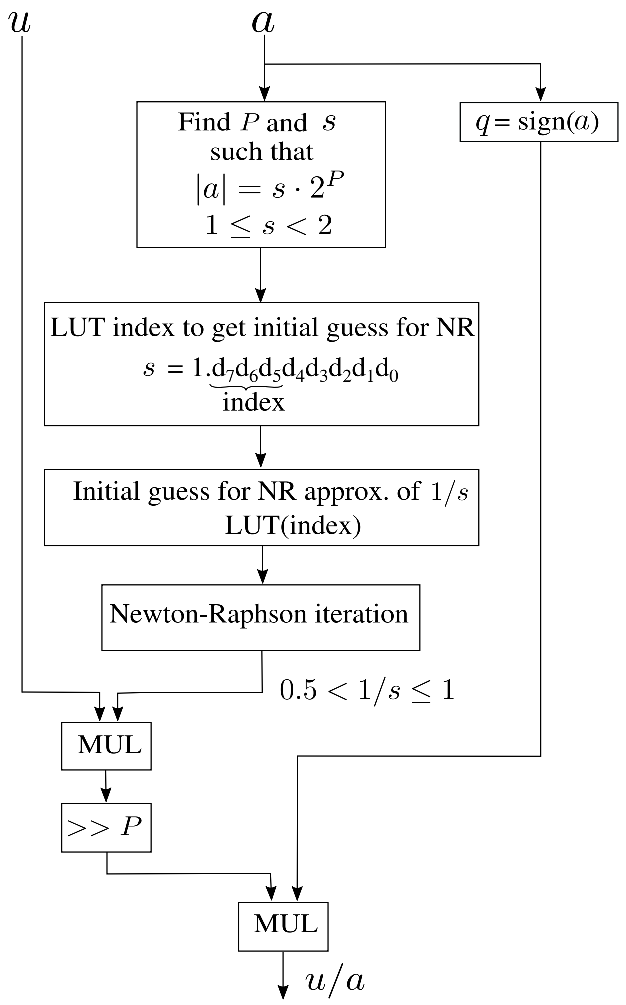

u
a
Find P and s such that $|a| = s \cdot 2^P$ $1 \leq s < 2$
q= sign(a)
LUT index to get initial guess for NR
s = $1.\underbrace{d_7d_6d_5}_{\text{index}}d_4d_3d_2d_1d_0$
Initial guess for NR approx. of $1/s$ LUT(index)
Newton-Raphson iteration
$0.5 < 1/s \leq 1$
MUL
>> P
MUL
u/a

Figure 3.11: Flow-chart to demonstrate the division function approximation in fixed-point arithmetic using Newton-Raphson (NR) method, used at the variable nodes.

Simulations were performed to find an optimal LUT size to get a reasonable initial guess for the NR approximation and ensure high accuracy of the division result with a minimum number of iterations. Specifically the performance with LUT sizes of 4, 8 and 16 entries were computed by numerical simulation using the procedure described below. For LUT sizes of 8 and 16 entries, similar FER performance is obtained after two NR iterations while one NR iteration results in performance loss compared to 2 iterations. For a LUT size of 4 entries, FER performance is 0.2 dB worse than that of the 8 entry LUT even after 2 or more iterations. Based on these results, we create a LUT with only 8 numerical values.

In order to obtain these 8 initial values, the range of s, i.e., 1 to 2, is divided into 8 equal sub-intervals: 1, $1\frac{1}{8}$, $1\frac{2}{8}$, $1\frac{3}{8}$, $1\frac{4}{8}$, $1\frac{5}{8}$, $1\frac{6}{8}$, $1\frac{7}{8}$, 2. Then, the mid-points of these sub-parts are computed. As we want to calculate the reciprocal of s, the mid-points of these sub-intervals is obtained by their geometric means. For example, the geometric mean of $1\frac{1}{8}$ and $1\frac{2}{8}$ is $(1\frac{1}{8} \times 1\frac{2}{8})^{1/2}$. Then we compute the reciprocal of this geometric mean, i.e., $(1\frac{1}{8} \times 1\frac{2}{8})^{-1/2}$.

In a similar fashion, the other entries of the look up table are computed, i.e., $\{(1 \times 1\frac{1}{8})^{-1/2}, (1\frac{1}{8} \times 1\frac{2}{8})^{-1/2}, \ldots, (1\frac{7}{8} \times 2)^{-1/2}\}$, and converted to a fixed-point representation that is used for the rest of the decoder.

For a fixed-point number, s, we use the 3 bits after the leading 1 (since $1 \leq s < 2$) as the index for the LUT to obtain the initial guess. The complexity of this method is constant time, i.e., $\mathcal{O}(1)$ [77]. From an FPGA perspective, a larger LUT (5-bit or 6-bit) can provide a better initial guess for the NR approximation with no additional circuitry. However, it is important to know the smallest required LUT size for possible future ASIC implementation. Fig. 3.11 summarizes the steps to approximate division for the fixed-point decoder.

Approximation of exponential function using LUTs

A direct implementation of the exponential function in FPGA has large resource requirements and design complexity. In the literature, the exponential function has been approximated in fixed-point arithmetic using various techniques, e.g., approximating the exponential function by a Taylor series and exponential approximation using parabolic synthesis [78–81]. Decomposing the exponential function into the product of a few components where each component is computed using lookup-tables is a simple and feasible implementation with limited FPGA resources.

In an LDLC decoder implementation, the exponent is always non-positive. Specifically, we approximate $\exp(-a/2)$ for $a \geq 0$, where the division by two accounts for the factor of $\frac{1}{2}$ in the exponent of (3.6).

For ease of computation, the exponential function $\exp(-a/2)$ is written as the product of three easily computable terms.

In particular, a is decomposed into 3 parts as

$$a = I_2 2^{P_2} + I_1 2^{P_1} + I_0 2^{P_0}, \tag{3.15}$$

where $P_0 < P_1 < P_2$ are the positions of the least significant bit of each part and I_0, I_1, I_2 are integers that depend on a such that $0 \leq I_0 < 2^{P_1 - P_0}$, $0 \leq I_1 < 2^{P_2 - P_1}$ and $0 \leq I_2 < 2^{W_i - P_2}$. Fig. 3.12 illustrates the relationship between a and I_0, I_1 and I_2. Since I_0 is comprised of $P_1 - P_0$ bits, its range is from 0 to $2^{(P_1 - P_0)} - 1$. Likewise I_1 is comprised of $P_2 - P_1$ bits and its range is from 0 to $2^{(P_2 - P_1)} - 1$ and I_2 comprises of $W_i - P_2$ bits with its range from 0 to $2^{(W_i - P_2)} - 1$.

Then the exponential is given as,

$$\begin{aligned}\exp(-a/2) = \exp&(-I_2 2^{P_2}/2) \\ &\times \exp(-I_1 2^{P_1}/2)\exp(-I_0 2^{P_0}/2).\end{aligned} \tag{3.16}$$

Decomposing a into three smaller parts thus allows for three smaller look-up tables instead of a single large lookup table to approximate the exponential.

We choose P_0, P_1 and P_2 carefully, e.g., $P_0 = -W_f$, P_2 is the smallest positive integer such that $\exp(-2^{P_2}/2)$ underflows the fixed-point representation of the LDLC decoder and $P_1 = \lfloor (P_0 + P_2)/2 \rfloor$. Due to the choice of P_2, if $I_2 > 0$ then $\exp(-a/2)$ is approximated as 0. Otherwise $I_2 = 0$ and thus $\exp(-I_2 2^{P_2}/2) = 1$ and only two small look-up tables are sufficient to compute $\exp(-I_1 2^{P_1}/2)$ and $\exp(-I_0 2^{P_0}/2)$.

The first lookup table contains $2^{(P_2 - P_1)}$ entries to approximate $\exp(-I_1 2^{P_1}/2)$ for possible I_1 values, i.e., 0 to $2^{(P_2 - P_1)} - 1$. Likewise, the second lookup table approximates $\exp(-I_0 2^{P_0}/2)$ for all possible $2^{(P_1 - P_0)}$ values of I_0. The exponential approximation in fixed-point representation (Q12.8) is explained with an example in Appendix C.

3.5.2 Optimal Word Length and Newton-Raphson (NR) Iterations for Fixed-point decoder

In order to find the optimal word length for the fixed-point representation, simulations are performed for different values of W_i and W_f. Fig. 3.13 compares the frame error rates for

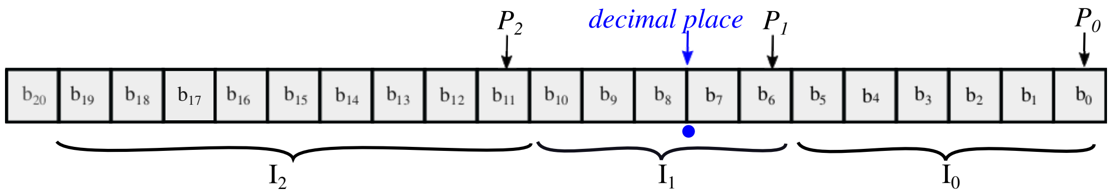

P_2
decimal place
P_1
P_0
b_{20} b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
I_2
I_1
I_0

Figure 3.12: Diagram to show the relationship between a and I_0, I_1 and I_2 as used in the approximation of the exponential function in fixed-point arithmetic at the variable nodes.

different values of W_f while keeping W_i large and varying the number of NR iterations for block length $n = 1000$. Fig. 3.14 compares decoder performance for different values of W_i while W_f is fixed.

A key observation in Fig. 3.13 is that at 4.5 dB the FER for Q14.8 with 2 NR iterations is 0.13 dB better compared to Q14.18. The LDLC decoder is sub-optimal because it is both iterative and parametric in nature. Therefore, it is anticipated that some approximations could potentially improve the decoder performance.

To understand this behaviour, simulations were performed with a floating-point decoder where the components of the Gaussian mixture message at the variable node that have coefficients less than a certain threshold, denoted $\mathsf{coeff}_{\rm th}$, are removed from the Gaussian mixture. As illustrated in Fig. 3.15, the FER does not monotonically increase with $\mathsf{coeff}_{\rm th}$, but instead achieves a minimum at approximately $\mathsf{coeff}_{\rm th} \approx \mathsf{0.03}$. Based on these simulation results, an appropriate choice of W_f helps the decoder by naturally underflowing the fixed point representation of small coefficients. However if W_f is further reduced, then performance deteriorates.

A similar trend has previously been seen in published fixed-point turbo decoders, where the quantization methodology leads to fixed-point implementations where the bit error rate (BER) can be slightly better than the BER of floating-point implementation [82]. The results reported in Fig. 3.14 demonstrate that the decoder performance degrades with smaller W_i due to the computation errors that occur from the saturation in arithmetic operations, primarily multiplication.

Based on the results in Fig. 3.13 and Fig. 3.14, a word length of 21 with 12 integer bits, 8 fractional bits and a sign bit is an appropriate choice for a fixed-point representation. As demonstrated in Fig. 3.14 the single-Gaussian decoder achieves an FER of $3 \cdot 10^{-3}$ at $-10\log_{10} 2\pi e\sigma^2 = 5$ dB which is slightly better than the floating-point.

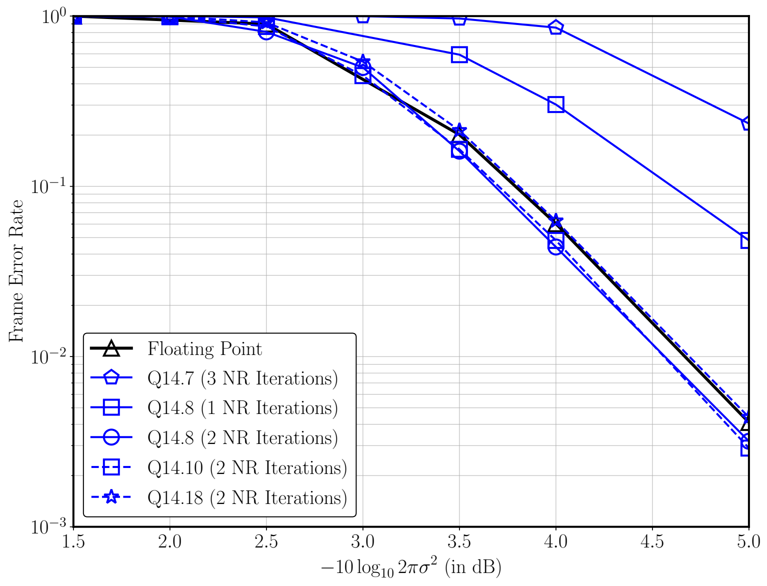


Figure 3.13: FER for different numbers of fractional bits and Newton-Raphson iterations for $n = 1000$, $d = 3$ where $-10\log_{10} 2\pi e\sigma^2$ is distance from the theoretical noise limit.

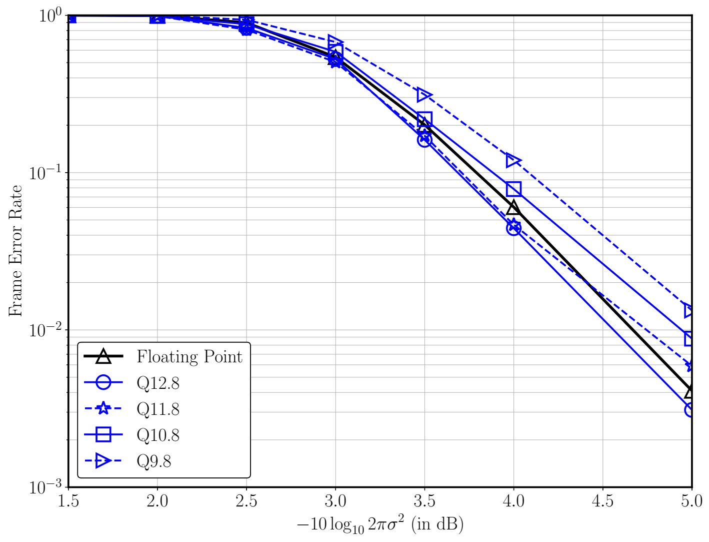


Figure 3.14: Frame error rate for different numbers of integer bits and two Newton-Raphson iterations with $n = 1000$, $d = 3$.

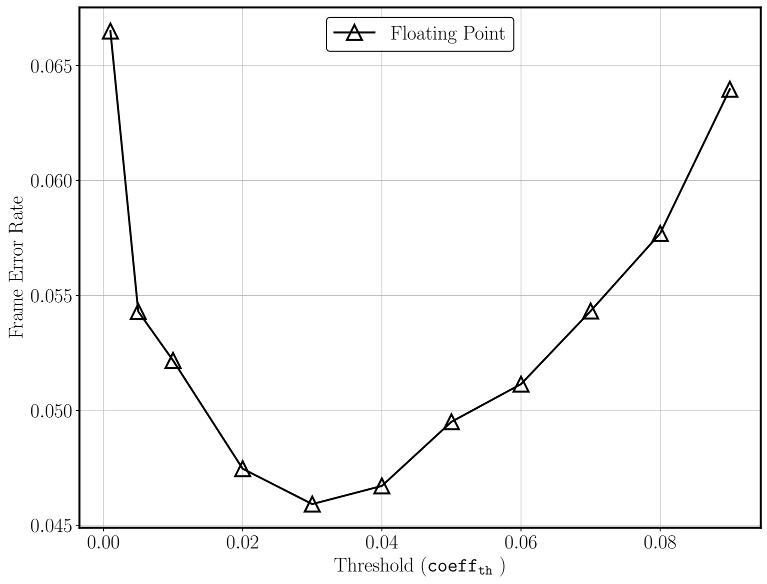


Figure 3.15: Effect of removing small coefficients from Gaussian mixture in floating point LDLC decoder at $-10\log_{10} 2\pi e\sigma^2 = 4$ dB, $n = 1000$ and $d = 3$.

3.6 LDLC Decoder FPGA Implementation

We now present our FPGA implementation results including 3 architectures: A) an architecture with a single check node and a single variable node, B) an architecture where parallelism and hardware resources are exploited to implement 20 variable nodes and a single check node and C) an architecture with a single check node and with pipelining to achieve an effective parallelism equivalent to 50 variable nodes. Decoder architectures A, B and C are implemented on an Intel FPGA (Arria 10, 10AX115N3F45I2SG).

As the decoder design is complex and multiplication intensive, it is not feasible to fit it on a smaller FPGA, e.g., Intel Cyclone. If given a larger FPGA, e.g., Intel Stratix 10, it would likely be possible to further exploit parallelism by balancing the check and variable node computation times.

All the implemented architectures have been verified for correct behaviour with gate level and post-fit timing netlist simulations. Fig. 3.16 summarizes the design and verification flow followed for the LDLC decoder architecture implementations.

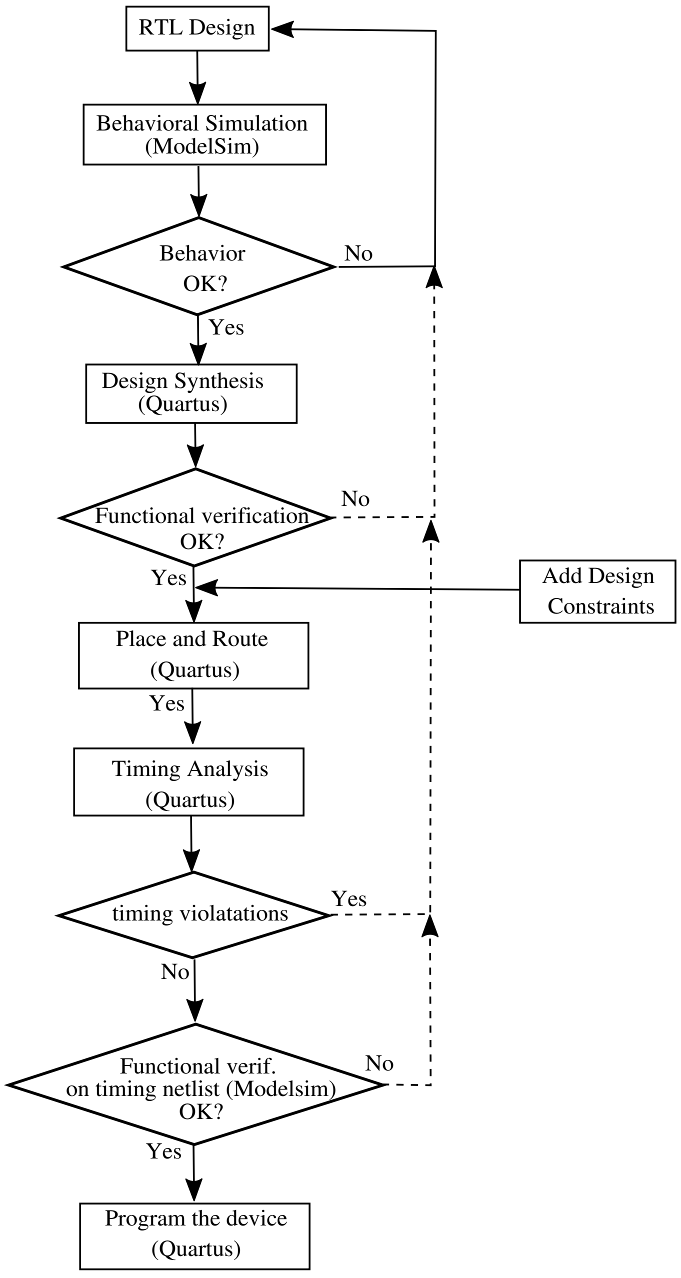

RTL Design
Behavioral Simulation (ModelSim)
Behavior OK?
No
Yes
Design Synthesis (Quartus)
Functional verification OK?
No
Yes
Add Design Constraints
Place and Route (Quartus)
Yes
Timing Analysis (Quartus)
timing violatations
Yes
No
Functional verif. on timing netlist (Modelsim) OK?
No
Yes
Program the device (Quartus)

Figure 3.16: The design and verification flow.

Architecture A) A single check node and a single variable node: A fully parallel LDLC decoder implementation is large and does not fit on the target reconfigurable device. However, there are possible approaches to build the complete decoder on a target FPGA device that can fit a few check and variable nodes.

To better understand the issues involved in an LDLC decoder implementation and make key estimates, e.g., resource requirements and performance, as a baseline design Fig. 3.17 presents a serial architecture for the decoder. This implementation contains one check node and one variable node. The check node and variable node messages generated during decoding iterations are stored in two separate single-ported memory banks. Read-only-memories (ROMs) are used to store check node connections to variable nodes and vice-versa, according to the H matrix. The edge weights of the connections are stored in a separate ROM.

In order to compute the outgoing messages from a check node, c_k, the *message routing network* looks up the variable nodes connected to c_k and the edge weights associated with these connections from the respective ROMs. Then, it fetches the corresponding means and variances from the *variable node message memory* and the *check node message processing block* computes the outgoing messages. The *variable node message processing block* receives the check node messages and computes the outgoing variable node messages in a similar fashion.

Check node message processing block The check node message processing block consists of a check node unit that performs convolution of the incoming messages according to (3.1) and (3.2). Fig. 3.18 and Fig. 3.19 show the mean and variance computations of the outgoing check node message that can be implemented with only a few adaptive logic modules (ALMs), digital signal processing (DSP) blocks and registers. Fig. 3.20 depicts the timing diagram for the check node message processing block in architectures A, B, and C.

Variable node message processing block At the variable nodes, the outgoing message along an edge is computed by taking the product of the channel message and all the Gaussian mixtures (obtained after the periodic extension), except the one associated with that edge. As discussed in Subsection 3.2.3 this message computation is performed using a forward backward recursive algorithm.

Fig. 3.21 shows the top-level architecture of the variable node unit (`VNU`) used in variable node message processing block of architecture A. The variable node unit consists of two sub-blocks, i.e, `FWBW` computation block and `VOut` computation block. The `FWBW` computation

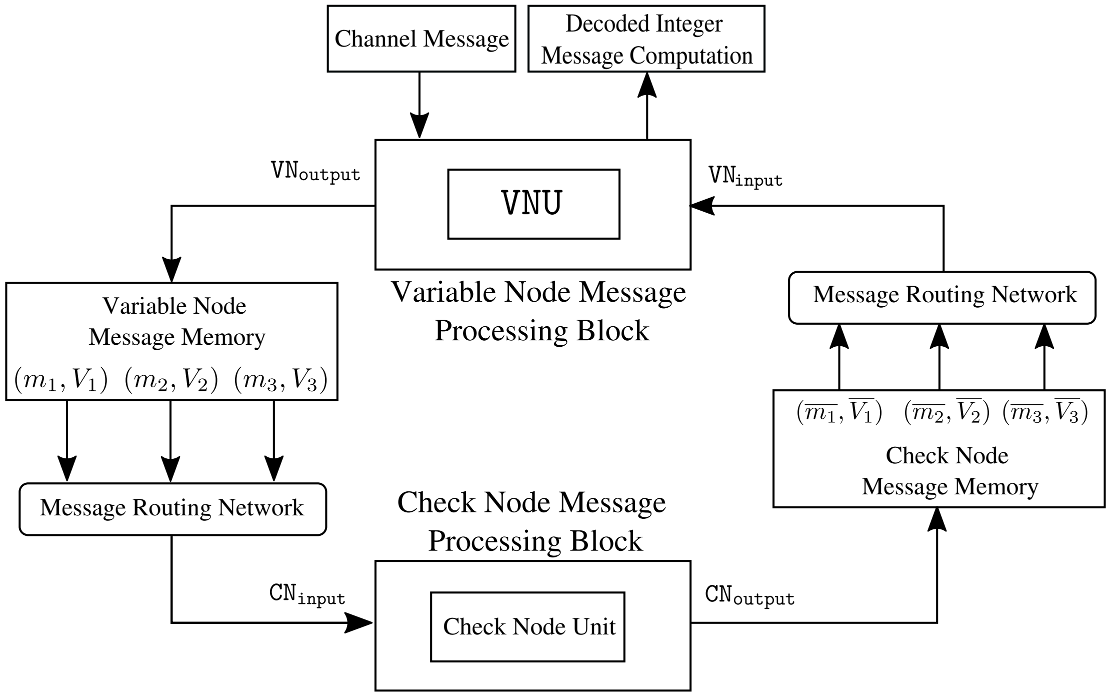

Channel Message
Decoded Integer Message Computation
$\mathtt{VN}_{\mathtt{output}}$
VNU
$\mathtt{VN}_{\mathtt{input}}$
Variable Node Message Processing Block
Message Routing Network
Variable Node Message Memory
(m_1, V_1) (m_2, V_2) (m_3, V_3)
$(\overline{m_1}, \overline{V_1})$ $(\overline{m_2}, \overline{V_2})$ $(\overline{m_3}, \overline{V_3})$
Check Node Message Memory
Message Routing Network
Check Node Message Processing Block
$\mathtt{CN}_{\mathtt{input}}$
Check Node Unit
$\mathtt{CN}_{\mathtt{output}}$

Figure 3.17: Block diagram of a two-node serial single-Gaussian LDLC decoder with a single check node and a single variable node (architecture A).

block computes the FW_ℓ and BW_ℓ for $\ell = 1, 2 \ldots, d$ and the `VOut` computation block reads-in the forward-backward messages and generates the outgoing variable node messages (m_ℓ, V_ℓ) for $\ell = 1, 2 \ldots, d$ as well as the estimate for the transmitted codeword, $\hat{w}_k$. The timing diagram for the variable node message processing block in architecture A is shown in Fig. 3.22.

This serial implementation was designed as a proof-of-concept for LDLC decoding in hardware. However, more than one check node and/or variable node with design optimizations can provide considerable improvement in decoding speed.

Architecture B) A single check node and 20 variable nodes: The variable node described above requires 140 clock cycles for message computation while the check node takes a single cycle, and thus the variable node limits the throughput. Several parallel variable nodes can render variable-node message computation faster and boost decoder throughput significantly. To exploit the inherent parallelism of iterative decoding we implement 20 parallel variable nodes with the available resources on the target FPGA (of

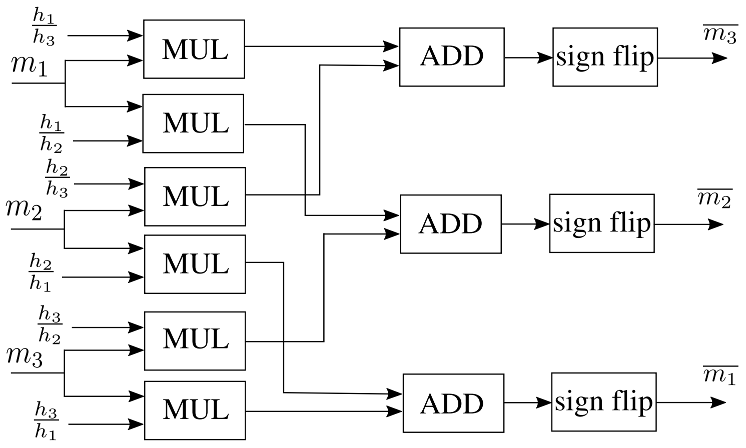


Figure 3.18: Block diagram for the mean computation of the outgoing messages at the check node. The mean is computed by first multiplying each incoming message with its respective edge weight (except the one on the outgoing edge), summing the results and further dividing the result of the summation by the outgoing edge weight and a sign flip.

course, a larger FPGA could potentially fit even more variable nodes).

Fig. 3.23 shows the decoder architecture where the check node message processing block has a single check node and the variable node message processing block contains 20 parallel variable node units denoted by $\texttt{VNU}_\texttt{p}$, with inputs $\texttt{VN}_{\texttt{input\{p\}}}$ and outputs, $\texttt{VN}_{\texttt{output\{p\}}}$ for $p = 0, 1, 2 \ldots, 19$. Fig. 3.24 shows the timing diagram for the variable node message processing block in architecture B. The message routing network fetches check node messages for one variable node every clock cycle and the incoming messages are driven to $\texttt{VN}_{\texttt{input\{p\}}}$ for $p = 0, 1, 2 \ldots, 19$ in 20 clock cycles sequentially.

Architecture C) A single check node and with pipelining to achieve an effective parallelism equivalent to 50 variable nodes: After additional data flow and design optimizations, in the variable node unit shown in Fig. 3.21, the `FWBW` computation block requires 109 clock cycles while the calculations in the `VOut` computation block take 10 clock cycles. This implies that one `VOut` computation block can be sufficient to process the output from 10 `FWBW` computation blocks (when pipelined), which could provide significant hardware savings.

For efficient variable node message computation, we implement a two-stage pipeline in the variable node message processing block. The first stage of the pipeline consists of 10 `FWBW` computation blocks that compute the FW_ℓ and BW_ℓ messages corresponding to 10

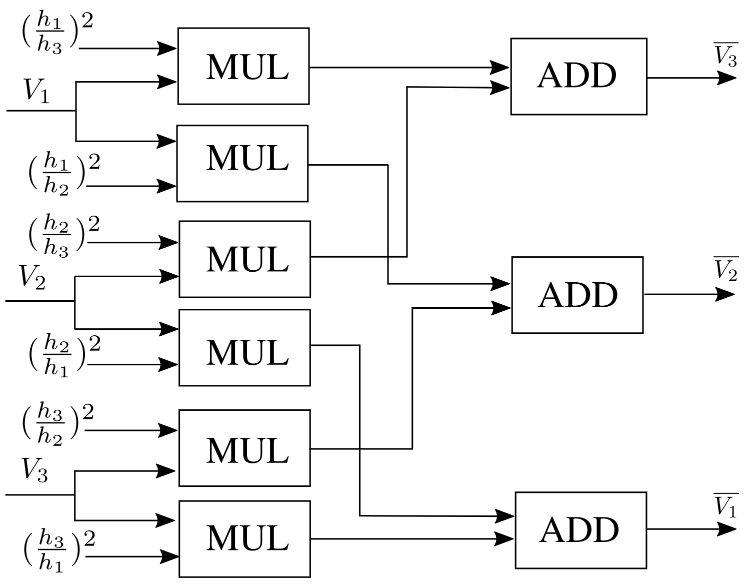


Figure 3.19: Block diagram for the variance computation of the outgoing check node messages.

variable nodes, x_k for $k = 0, 1, 2 \ldots, 9$, according to Algorithm 4. Further, the second stage block reads-in stage 1 output and computes outgoing variable node messages according to (3.9) corresponding to a variable node. The design components are reused in different clock cycles within the two pipeline stages. For convenience, this two pipelined stage is termed as `VNUCluster`.

The resources on the target FPGA are sufficient to implement 5 parallel `VNUCluster` blocks ($\texttt{VNUCluster}_\texttt{p}$ for $p = 0, 1, 2 \ldots, 4$), achieving a parallelism equivalent to 50 variable node units (`VNUs`). Thus rendering significantly reduced computation time for each variable node message generation overall. Fig. 3.25 shows the top-level block diagram of the variable node message processing block used in architecture C that consists of 5 `VNUCluster` blocks. The sub-blocks of the pipelining inside the `VNUCluster` blocks are shown specifically for $\texttt{VNUCluster}_0$. Here, 10 forward-backward message computation blocks, i.e., $\texttt{FWBW\{p\}}_\texttt{VNUCluster0}$ with inputs, $\texttt{inp\{p\}}_\texttt{VNUCluster0}$ and outputs, $\texttt{op\{p\}}_\texttt{VNUCluster0}$ for $p = 0, 1, 2 \ldots, 9$ comprise the first stage of the pipeline. The second stage consists of the $\texttt{VOut}_0$ computation block with input $\texttt{In}_0$ and output $\texttt{Out}_0$. Fig. 3.26 shows the timing diagram for the various signals used in the two pipelining stages of the $\texttt{VNUCluster}_0$ block.

The resource requirement and throughput of the variable node message processing block used in architectures A, B, and C, are provided in Table 3.2 and Table 3.3 respectively.

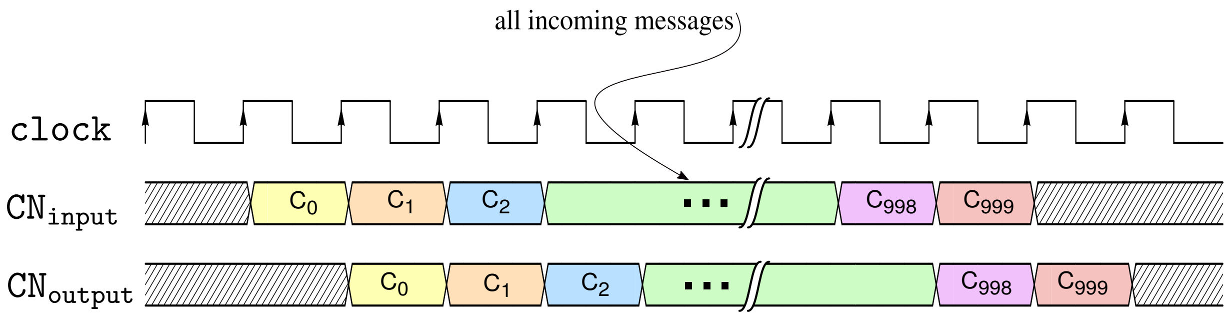


Figure 3.20: Timing diagram of the check node message processing block in architecture A , B and C.

Table 3.2: Resource requirements of the variable node message processing block in architecture A, B and C.

Resource	Arch. A	Arch. B	Arch. C
ALM	8151	321128	406281
Dedicated Regs.	6464	146260	229380
DSPs	160	1509	1507

Based on Table 3.2 and Table 3.3, it is evident that parallelism and pipelining boost throughput of the variable node message processing block significantly. However, it is achieved with an extra hardware cost.

Fig. 3.27 shows a high-level block diagram for decoder architecture C, that consists of a single check node and 5 `VNUCluster` blocks.

Performance and Resource Usage All three architectures achieve the frame error rate shown in Fig. 3.14 at a clock frequency of 125 MHz. If the decoder is operated at a higher frequency, some critical paths in the design may have timing issues. Therefore, 125 MHz is the recommended fastest clock for our architectures in the target technology.

With further critical path optimizations architecture A can run at a higher frequency, while architectures B and C are difficult to improve because of the very large use of resources, impeding the place-and-route process and then penalizing the frequency.

The resource usage for the decoder architectures A, B and C is provided in Table 3.4.

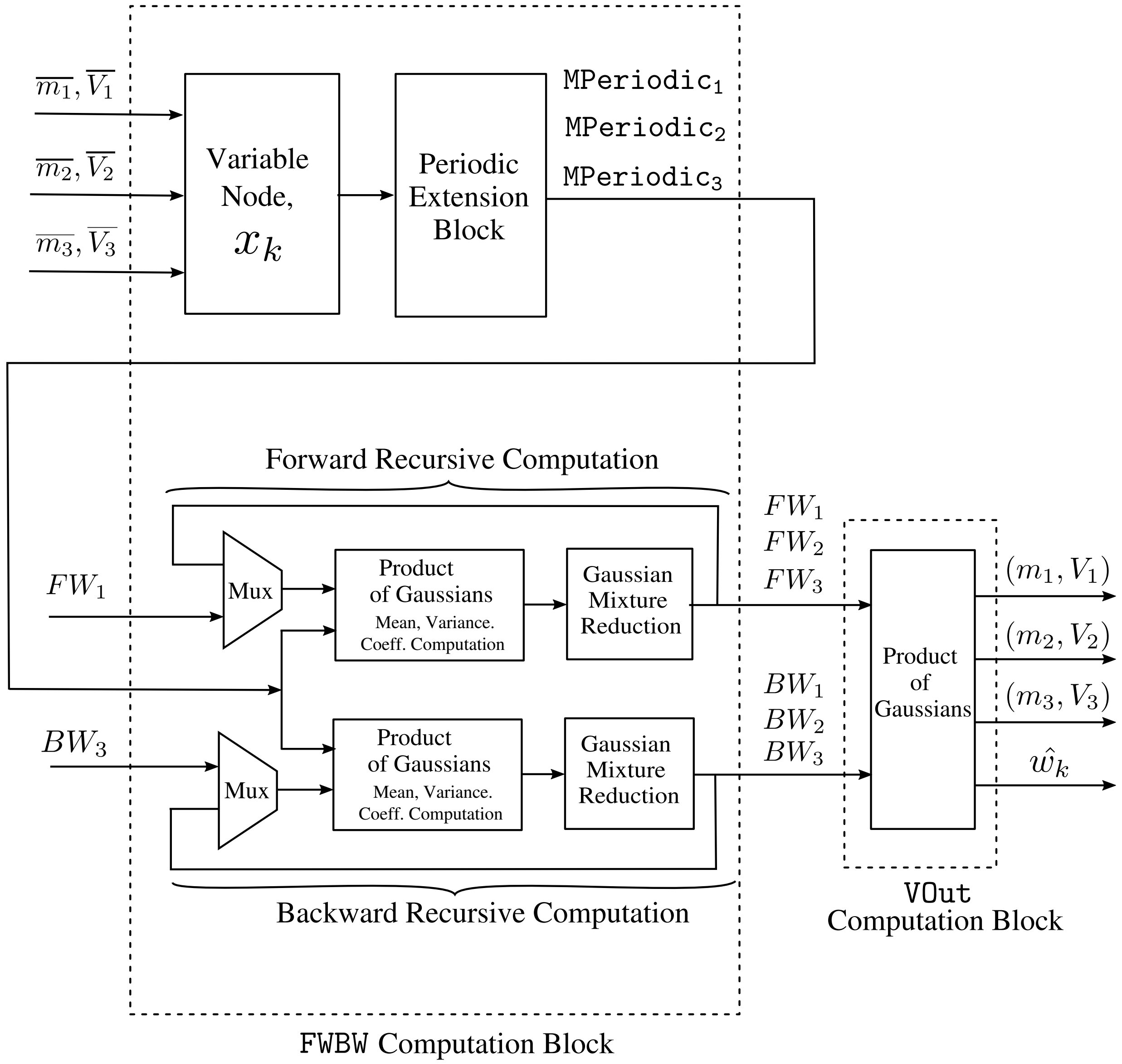


Figure 3.21: High-level architecture of a variable node unit (VNU) in single-Gaussian decoder with $d = 3$. At a variable node, x_k, the incoming check node messages are periodically extended, FW_ℓ and BW_ℓ for $\ell = 1, 2 \ldots, d$ are computed in FWBW computation block and finally the outgoing variable node messages, (m_l, V_l) for $\ell = 1, 2 \ldots, d$ and estimate for transmitted codeword, $\hat{w_k}$ is obtained in VOut computation block.

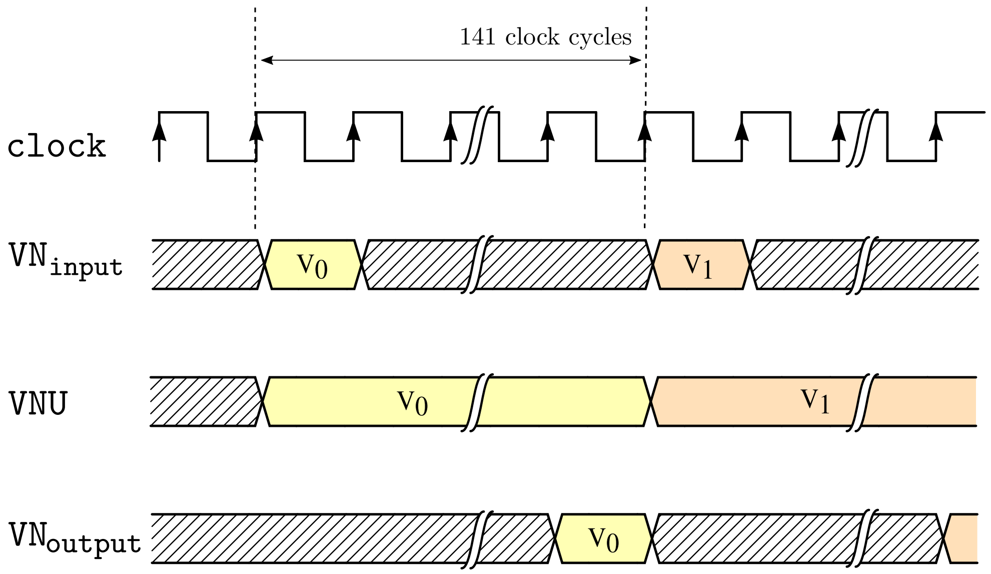


Figure 3.22: Timing diagram of the variable node message processing block in architecture A.

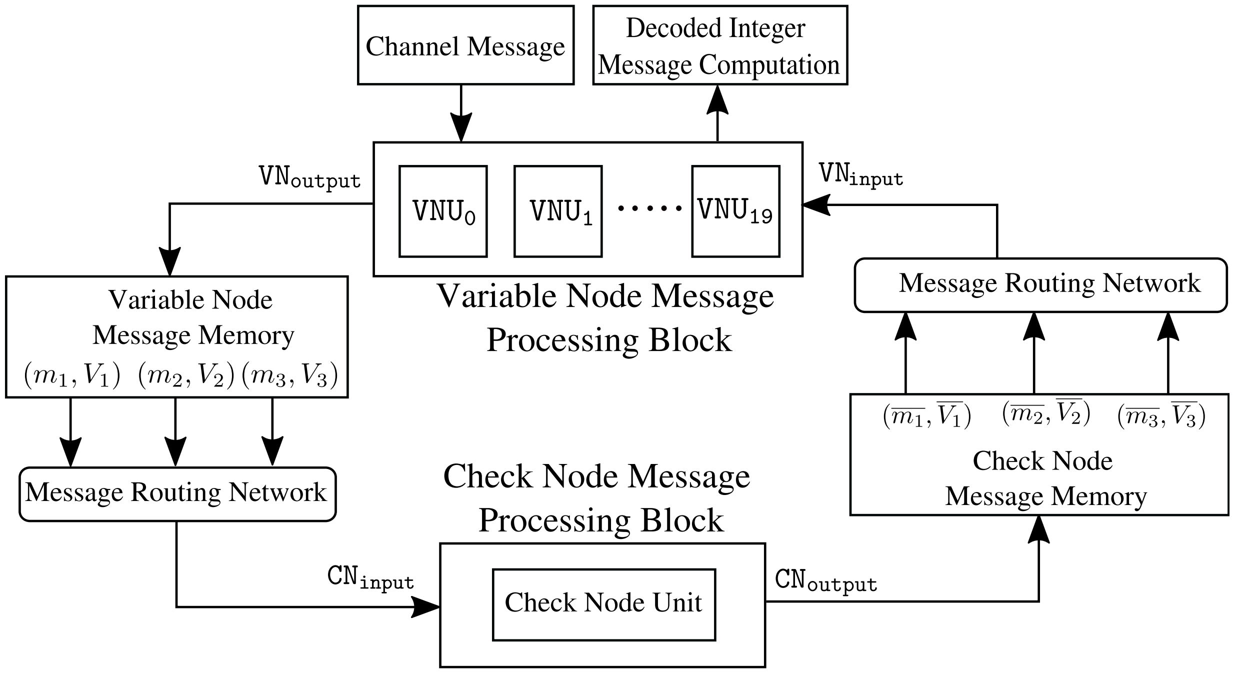
Fig. 3.28 shows the throughput comparison for these architectures. Architecture C attains a throughput of 10.5 Msymbols/sec at a distance of 5 dB from capacity which is a 24× improvement over the baseline implementation A and a 1.8× improvement over architecture B. Note that the decoder throughput varies over SNR values due to early termination in the iterative decoding process.

The storage requirement for this implementation is $O(n \cdot d)$ and the computational complexity is $O(n \cdot d \cdot R)$ where n is block length, d is degree for the LDLC design and R is the number of periodic extensions.

3.7 Summary

This chapter described the performance results and design strategies used for a fixed-point single-Gaussian LDLC decoder implementation in hardware. After developing approaches to address the complexities of the hardware implementation, e.g., efficient approximations of the non-linear functions and a comprehensive quantization study, we have achieved a successful FPGA implementation of a decoder for low-density-lattice codes.

With the detailed knowledge gained from the serial and partially parallel single-Gaussian



Channel Message
Decoded Integer Message Computation
VN$_{output}$
VNU$_0$
VNU$_1$
VNU$_{19}$
VN$_{input}$
Variable Node Message Processing Block
Variable Node Message Memory
(m_1, V_1) (m_2, V_2) (m_3, V_3)
Message Routing Network
Message Routing Network
$(\overline{m_1}, \overline{V_1})$ $(\overline{m_2}, \overline{V_2})$ $(\overline{m_3}, \overline{V_3})$
Check Node Message Memory
Check Node Message Processing Block
CN$_{input}$
Check Node Unit
CN$_{output}$

Figure 3.23: Top-level block diagram of the LDLC decoder with one check node and 20 parallel variable node units (architecture B).

LDLC decoder implementations, we now move on to the implementation of LDLC decoders where messages exchanged are Gaussian mixtures, where we expect to improve frame error rate performance.

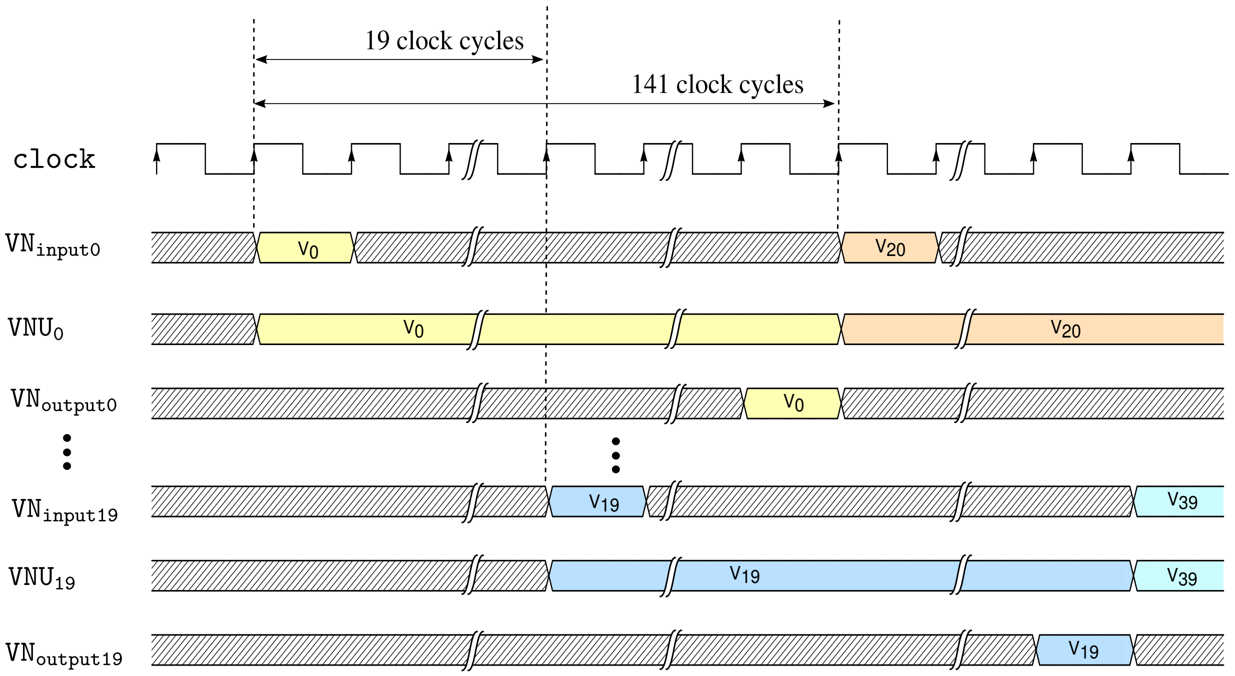


Figure 3.24: Timing diagram of the variable node message processing block in architecture B.

Table 3.3: Throughput (clock cycles/message) of the variable node message processing block in the architectures A, B and C.

Architecture	Throughput cycles/message
A	140
B	9.2
C	3.9

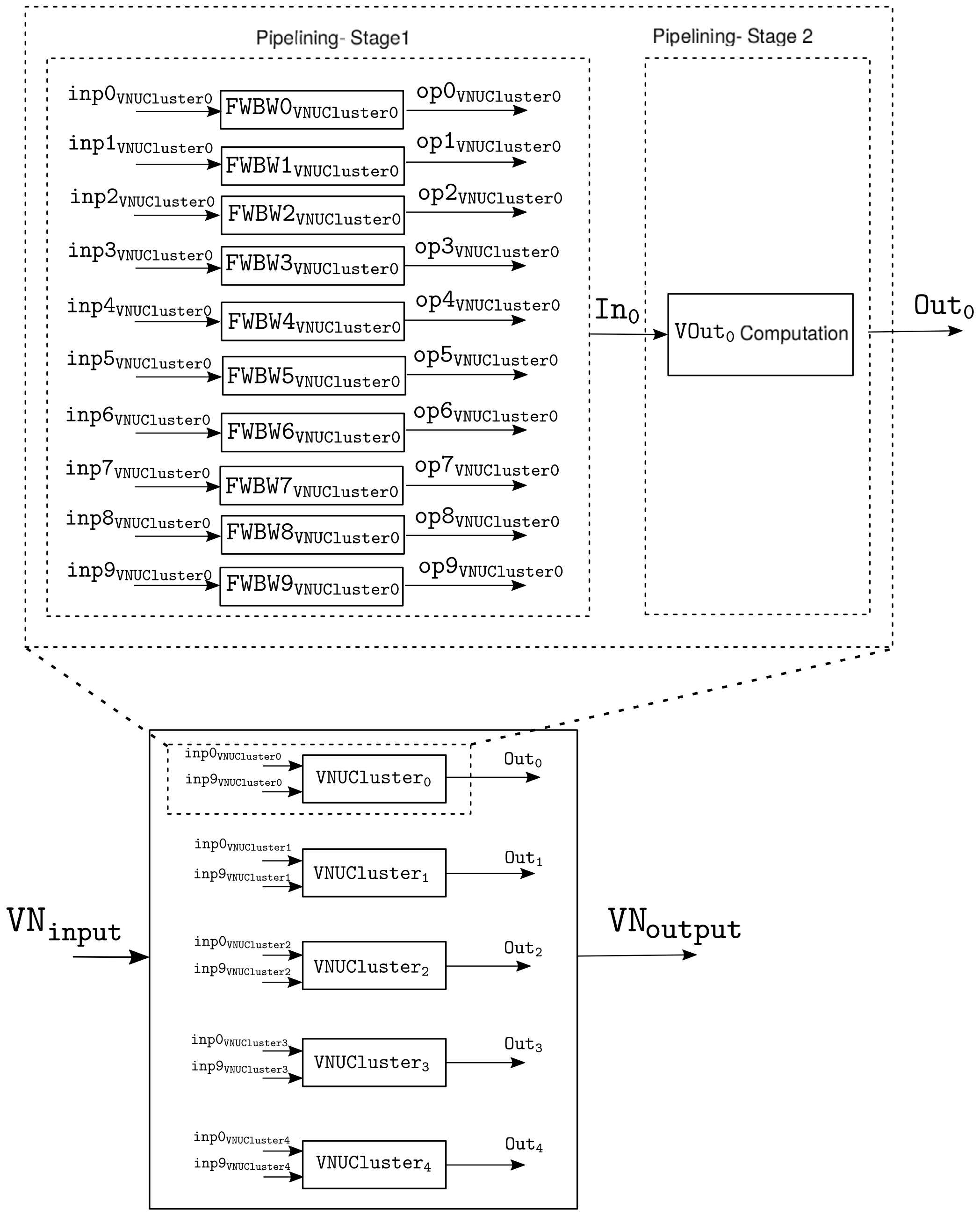
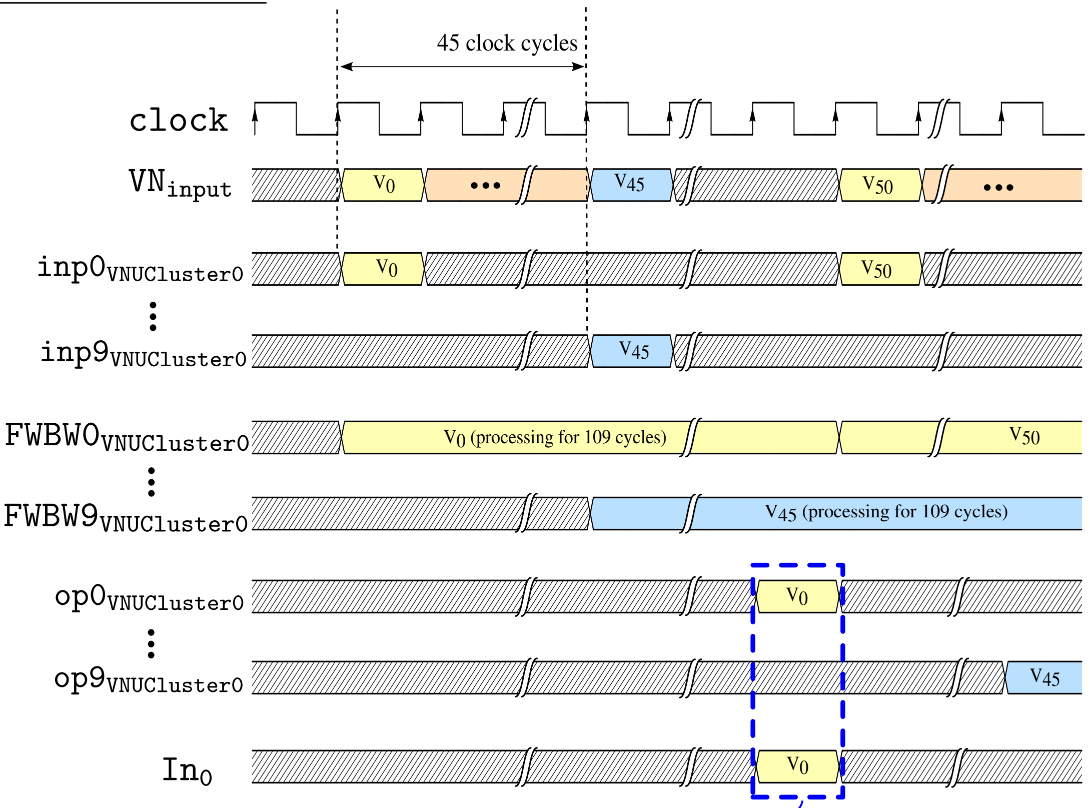


Figure 3.25: High-level diagram of the variable node message processing block used in architecture C, that consists of 5 `VNUCluster` blocks. The two stage pipelining used in `VNUCluster` blocks is shown specifically for $\texttt{VNUCluster}_0$.

Pipelining Stage 1:



Pipelining Stage 2:

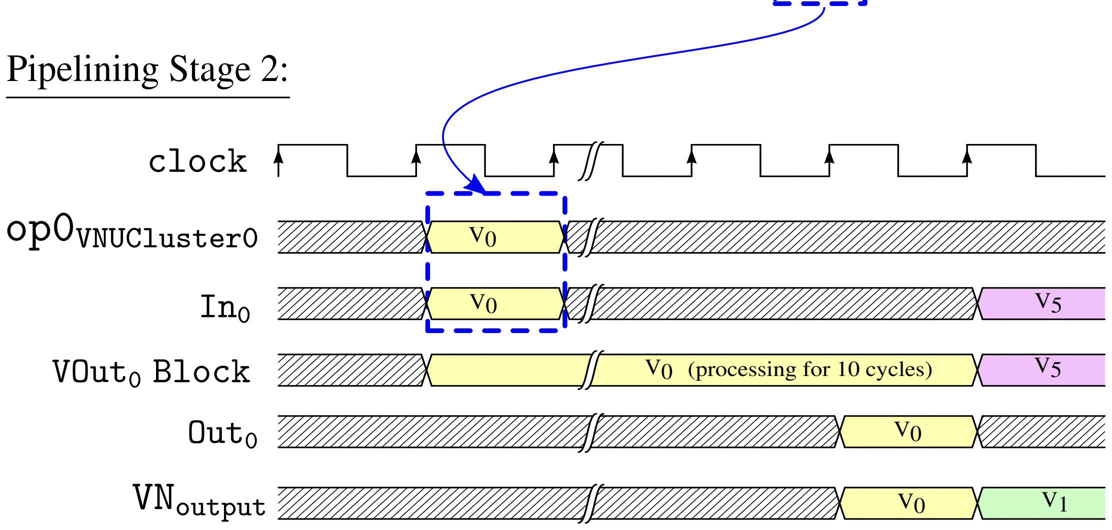


Figure 3.26: Timing diagram of `VNUCluster` block used in variable node message processing block of architecture C. The waveforms are shown particularly for `VNUCluster`$_0$ block.

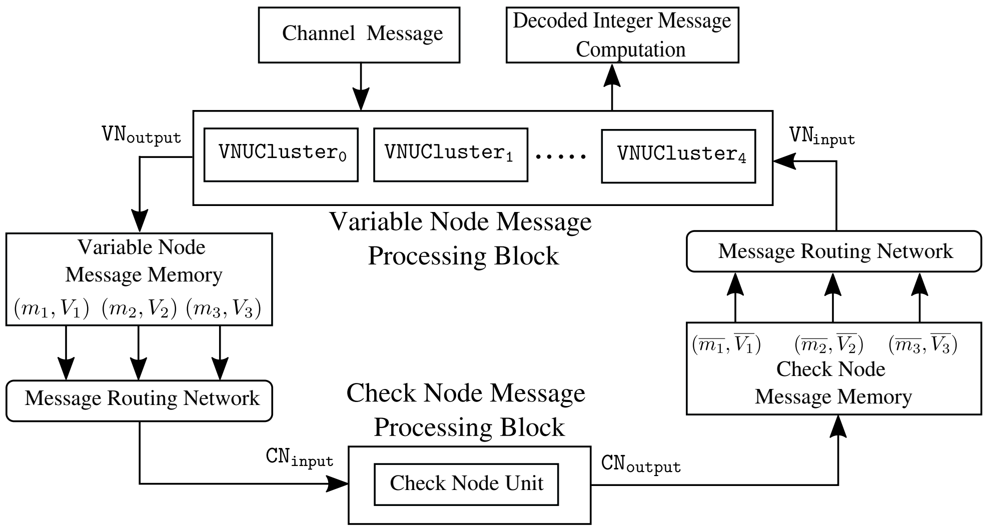

Channel Message
Decoded Integer Message Computation
$\mathtt{VN}_{\mathtt{output}}$
$\mathtt{VNUCluster}_0$
$\mathtt{VNUCluster}_1$
.....
$\mathtt{VNUCluster}_4$
$\mathtt{VN}_{\mathtt{input}}$
Variable Node Message Processing Block
Variable Node Message Memory
(m_1, V_1) (m_2, V_2) (m_3, V_3)
Message Routing Network
Message Routing Network
$(\overline{m_1}, \overline{V_1})$ $(\overline{m_2}, \overline{V_2})$ $(\overline{m_3}, \overline{V_3})$
Check Node Message Memory
Check Node Message Processing Block
$\mathtt{CN}_{\mathtt{input}}$
Check Node Unit
$\mathtt{CN}_{\mathtt{output}}$

Figure 3.27: Top-level architecture for the LDLC decoder with a single check node and with two-stage pipelining to achieve an effective parallelism equivalent to 50 variable nodes (architecture C).

Table 3.4: Resource usage of different architectures for single-Gaussian decoder

Resource	A 1 check node, 1 var.node	B 1 check node, 20 var.nodes	C 1 check node, parallelism equivalent to 50 var. nodes
ALMs (lut and reg)	12,560	328,490	411,436
Dedicated Registers	11,038	169,843	300,280
DSPs (27x27 mult.)	171	1,518	1,518
BRAMs	30	12	47

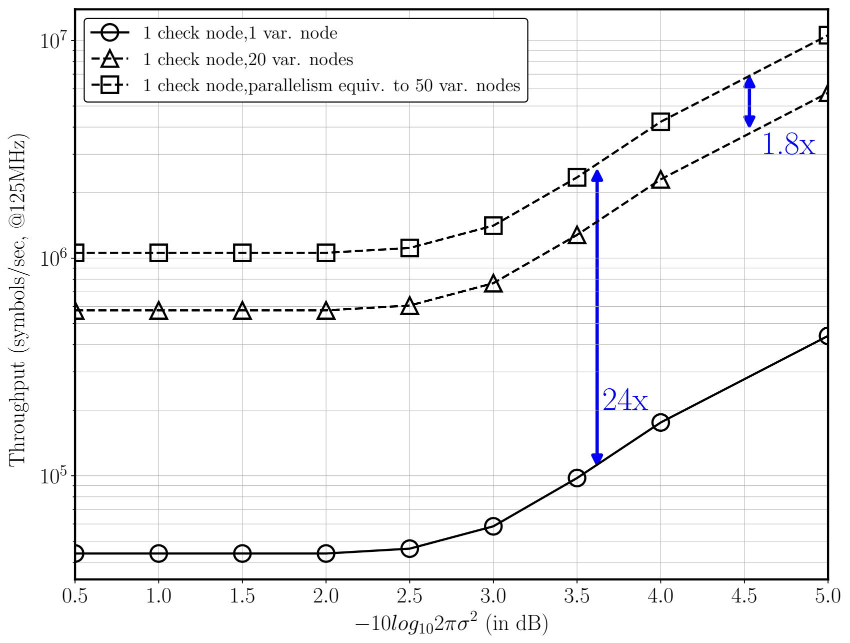


Figure 3.28: Throughput comparison of different decoder architectures for $n = 1000$ and clock frequency of 125 MHz.

Chapter 4

Multi-Gaussian LDLC Decoder

In Chapter 3, the hardware implementation of a single-Gaussian LDLC decoder on an FPGA device was presented. It described a comprehensive study to choose a suitable architecture for the first known hardware implementation of an LDLC decoder. This implementation used an iterative decoding algorithm where the Gaussian mixture messages are reduced to a single-Gaussian. Then, a detailed quantization analysis of the fixed-point arithmetic used for FPGA implementation, along with hardware throughput and resource requirements was described.

Following this, a natural extension of the work is to look at the hardware implementation of a parametric LDLC decoder where Gaussian mixture messages are represented by multiple triples of Gaussian parameters, i.e., means, variances and coefficients [9, 11, 12]. This approach is significantly more challenging compared to the single-Gaussian case; however, it provides better frame error rate performance. In this thesis, for convenience, a decoder where the Gaussian mixture messages exchanged during the iterative decoding process contain multiple components is termed, *multi-Gaussian decoder*.

This chapter describes the criteria used to select an appropriate multi-Gaussian decoder for a hardware implementation. Then, we include the design optimizations applied to reduce the decoder complexity and different hardware architectures implemented on the target FPGA. Subsequently, the resource requirements and throughput of different architectures as well as design trade-offs are discussed.

4.1 Selection of an Appropriate Decoder for the Hardware Implementation

Table 3.1 in Chapter 3 presents a comparison of various parametric decoders published in the literature, where multi-Gaussian decoders II and III achieve similar decoding performance when the Gaussian mixture messages exchanged during the iterative decoding process contain only two components. Based on Table 3.1, decoder II attains performance similar to that of III with significantly lower design complexity, and is thus preferred choice for hardware implementation.

In decoder II, the messages are two triples of the Gaussian parameters, means, variances and coefficients, i.e., $M = 2$. The Gaussian mixture messages generated during the variable node and check node operations are reduced to a mixture with two components according to Algorithm 3 (described in Subsection 2.5.2). For a reasonable comparison of the multi-Gaussian decoder with the single-Gaussian decoder in terms of frame error rate and design throughput, we choose a multi-Gaussian decoder with degree 3 for the initial hardware implementation.

4.2 Iterative Decoding for Multi-Gaussian LDLC Decoder ($M = 2$)

Subsection 2.4 presented an iterative decoding algorithm where the messages exchanged during the decoding iterations are continuous functions; then in Section 3.2, the iterative decoding process for the single-Gaussian LDLC decoder was described. Here, we outline the operations that occur in the *multi-Gaussian decoder* ($M = 2$) at the check nodes and variable nodes during the iterative decoding process.

4.2.1 Initialization

At the start of the decoding process, each variable node x_k sends the received channel message given by $(m_0, V_0, 1)$, along all the edges connected to this variable node where m_0 is y_k and V_0 is σ^2, as shown in Fig. 4.1.

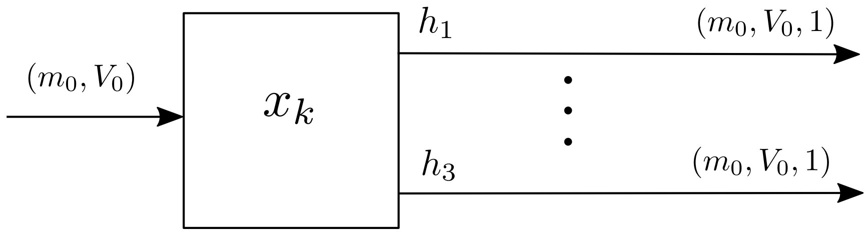


Figure 4.1: A variable node, x_k sends the message received from the channel to all connected check nodes.

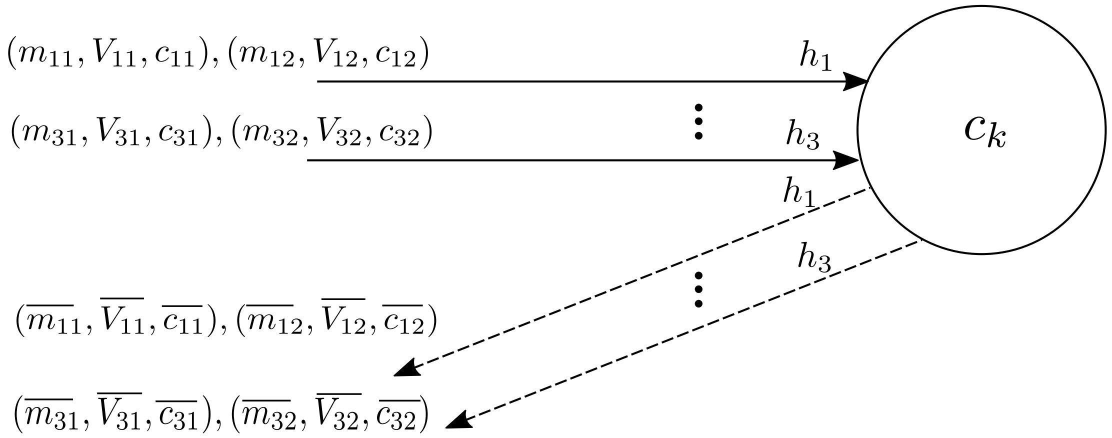

$(m_{11}, V_{11}, c_{11}), (m_{12}, V_{12}, c_{12})$
$(m_{31}, V_{31}, c_{31}), (m_{32}, V_{32}, c_{32})$
h_1
h_3
c_k
h_1
h_3
$(\overline{m_{11}}, \overline{V_{11}}, \overline{c_{11}}), (\overline{m_{12}}, \overline{V_{12}}, \overline{c_{12}})$
$(\overline{m_{31}}, \overline{V_{31}}, \overline{c_{31}}), (\overline{m_{32}}, \overline{V_{32}}, \overline{c_{32}})$

Figure 4.2: Illustration of all the incoming and the outgoing messages at a check node in multi-Gaussian decoder with $M = 2$ and $d = 3$.

4.2.2 Basic Iteration: Check Node Message

Each check node has d input messages coming along the edges connected to it with weights h_p, $p = 1, \ldots, d$, where h_p is one of the $\bar{h}$'s with a possible sign flip as shown in Fig. 4.2. The incoming messages contain two components given by $\{(m_{\ell 1}, V_{\ell 1}, c_{\ell 1}), (m_{\ell 2}, V_{\ell 2}, c_{\ell 2})\}$, where $\ell = 1, 2 \ldots, d$.

The outgoing message along an edge is computed by the convolution of all the incoming messages except the one incoming on that edge. To achieve this, each possible pair of components between the mixtures is convolved. If two Gaussians with triples (m_1, V_1, c_1) and (m_2, V_2, c_2) are convolved, the resultant Gaussian represented by triple (m_r, V_r, c_r) is obtained using (2.45), (2.46) and (2.47).

Convolution of two Gaussian mixture messages that contain two components, i.e., $M = 2$, generates a Gaussian mixture with four components. It is then reduced into a mixture

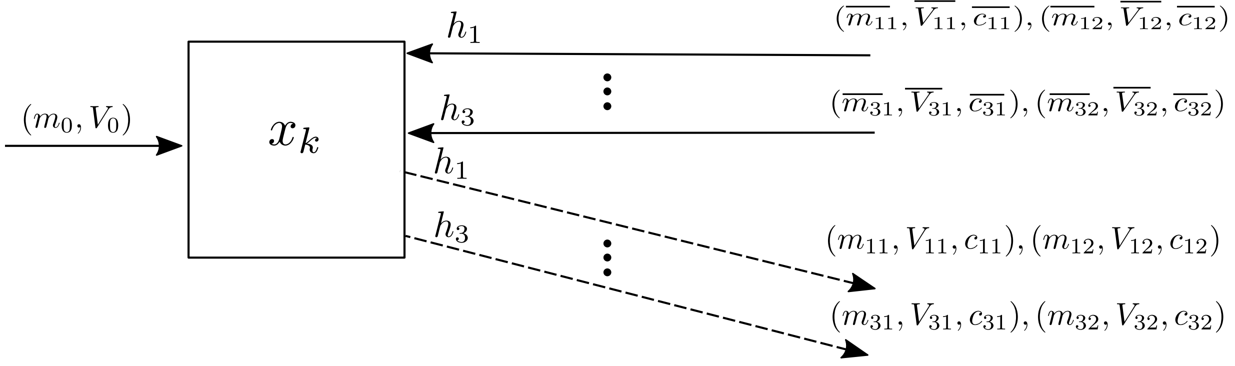

(m_0, V_0)
x_k
h_1
h_3
h_1
h_3
$(\overline{m_{11}}, \overline{V_{11}}, \overline{c_{11}}), (\overline{m_{12}}, \overline{V_{12}}, \overline{c_{12}})$
$(\overline{m_{31}}, \overline{V_{31}}, \overline{c_{31}}), (\overline{m_{32}}, \overline{V_{32}}, \overline{c_{32}})$
$(m_{11}, V_{11}, c_{11}), (m_{12}, V_{12}, c_{12})$
$(m_{31}, V_{31}, c_{31}), (m_{32}, V_{32}, c_{32})$

Figure 4.3: All the incoming and outgoing messages at a variable node in multi-Gaussian decoder with $M = 2$ and $d = 3$.

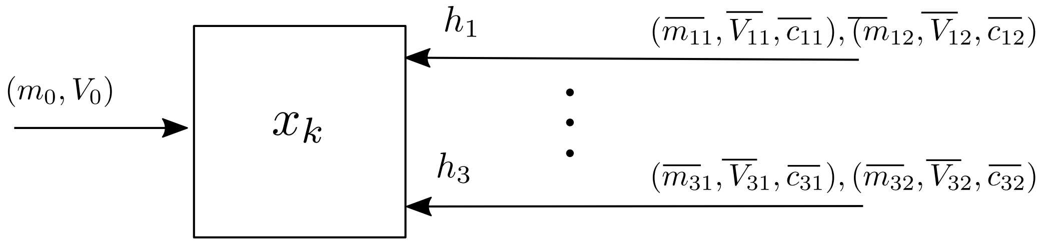

(m_0, V_0)
x_k
h_1
h_3
$(\overline{m_{11}}, \overline{V_{11}}, \overline{c_{11}}), (\overline{m_{12}}, \overline{V_{12}}, \overline{c_{12}})$
$(\overline{m_{31}}, \overline{V_{31}}, \overline{c_{31}}), (\overline{m_{32}}, \overline{V_{32}}, \overline{c_{32}})$

Figure 4.4: For final decision, the channel message and all the incoming messages to a variable node, x_k, are multiplied.

with two components using an algorithm based on merging the Gaussian components in a range (Algorithm 3).

In order to compute the outgoing messages efficiently, a recursive forward-backward algorithm is applied at the check nodes [11]. The pseudo-code for the algorithm is provided in Algorithm 5. In the algorithm, '*' denotes convolution of Gaussian mixtures and the incoming check node messages, i.e, $((m_{\ell 1}, V_{\ell 1}, c_{\ell 1}), (m_{\ell 2}, V_{\ell 2}, c_{\ell 2}))$ for $\ell = 1, 2 \ldots, d$ are denoted as, $\texttt{CheckNodeMessage}_\ell$. Here, the Gaussian mixture reduction (including the normalization step) is represented by `GMR_CNRangebased`.

We first compute the forward-backward messages, denoted by $CNFW_\ell$ and $CNBW_\ell$ for $\ell = 1, 2, \ldots, d$. Then the outgoing check node messages, i.e., $((\overline{m_{\ell 1}}, \overline{V_{\ell 1}}, \overline{c_{\ell 1}}), (\overline{m_{\ell 2}}, \overline{V_{\ell 2}}, \overline{c_{\ell 2}}))$ for $\ell = 1, 2 \ldots, d$ are calculated as below:

$$((\overline{m_{\ell 1}}, \overline{V_{\ell 1}}, \overline{c_{\ell 1}}), (\overline{m_{\ell 2}}, \overline{V_{\ell 2}}, \overline{c_{\ell 2}})) = \texttt{GMR\_CNRangebased}(CNFW_\ell * CNBW_\ell). \tag{4.1}$$

Algorithm 5: Forward-backward recursive algorithm at check node

```
## initialization
$CNFW_1 = ((0, 0.03, 1), (0, 0.03, 1))$
$CNBW_d = ((0, 0.03, 1), (0, 0.03, 1))$
$CNFW_2 =$ CheckNodeMessage$_1$
$CNBW_{d-1} =$ CheckNodeMessage$_d$
## main loop
for $i = 2$ to $d - 1$ do
    $CNFW_{(i+1)} =$ GMR_CNRangebased(CheckNodeMessage$_i * CNFW_i$)
    $CNBW_{(d-i)} =$ GMR_CNRangebased($CNBW_{(d-i+1)} *$ CheckNodeMessage$_{(d-i+1)}$)
end
```

Algorithm 6: Forward-backward recursive algorithm at variable node

```
## initialization
$VNFW_1 = ((m_0, 2, 1), (0, 0.03, 1)$
$VNBW_d = ((m_0, 2, 1), (0, 0.03, 1)$
## main loop
for $i = 1$ to $d - 1$ do
    $VNFW_{(i+1)} =$ GMR_VNRangebased($VNFW_i \cdot$ MPeriodic$_i$)
    $VNBW_{(d-i)} =$ GMR_VNRangebased($VNBW_{(d-i+1)} \cdot$ MPeriodic$_{(d-i+1)}$)
end
```

4.2.3 Basic Iteration: Variable Node Message

Each variable node receives d Gaussian mixture messages with two components denoted by $\{(\overline{m_{\ell 1}}, \overline{V_{\ell 1}}, \overline{c_{\ell 1}}), (\overline{m_{\ell 2}}, \overline{V_{\ell 2}}, \overline{c_{\ell 2}})\}$, where $\ell = 1, 2 \ldots, d$ as shown in Fig. 4.3. These received messages are periodically extended at the variable node. Similar to the single-Gaussian implementation in Chapter 3, in a multi-Gaussian decoder, the periodic extension step is performed at the variable nodes instead of the check nodes.

In this step, the mean of the incoming check node message along an edge with weight h_l is first periodically extended as below:

$$\overline{m_{\ell 1}}(i) = \overline{m_{\ell 1}} + \frac{i}{h_\ell}, \tag{4.2}$$

$$\overline{m_{\ell 2}}(i) = \overline{m_{\ell 2}} + \frac{i}{h_\ell}, \tag{4.3}$$

where i denotes the i^{th} extension. In principle, similar to the single-Gaussian decoder the variable i can be any integer value; however, in practice the range is restricted within a certain limit to a subset of integers.

The periodic extension step converts each incoming message with two components ($M = 2$) into a larger Gaussian mixture, e.g., taking R periodic extensions generates a Gaussian mixture with $M \cdot R$ components. The variable node message along the edge with weight h_p is computed by taking the product of the channel message, denoted by $(m_0, V_0, 1)$, and all the Gaussian mixtures obtained after the periodic extension, except the mixture associated with that edge.

The product of two Gaussian mixtures is calculated by the pair-wise multiplication of each possible pair of components between the two mixtures, as described in Subsection 2.5.2. The Gaussian mixture generated in the product-step must be reduced to a Gaussian mixture with only two components before it can be sent along an outgoing edge of the node. This Gaussian mixture reduction is obtained using the Gaussian mixture reduction Algorithm 3 in Subsection 2.5.2.

Similar to the single-Gaussian decoder, a forward-backward recursion algorithm is used to generate variable node messages in the multi-Gaussian decoder. The algorithm is initialized with the channel message. Let's assume the periodically extended messages are denoted by $\texttt{MPeriodic}_\ell$ for $\ell = 1, 2 \ldots, d$ and the Gaussian mixture reduction (including the normalization step) by `GMR_VNRangebased`. The pseudo-code for the recursive algorithm is provided in Algorithm 6. Here, "$\cdot$" denotes the product of Gaussian mixtures. First, the forward-backward messages, $VNFW_\ell$ and $VNBW_\ell$ for $\ell = 1, 2 \ldots, d$ are computed using Algorithm 6 and then, the outgoing variable node messages, i.e., $((m_{\ell 1}, V_{\ell 1}, c_{\ell 1)}, (m_{\ell 2}, V_{\ell 2}, c_{\ell 2}))$ for $\ell = 1, 2 \ldots, d$ are obtained as,

$$((m_{\ell 1}, V_{\ell 1}, c_{\ell 1}), (m_{\ell 2}, V_{\ell 2}, c_{\ell 2})) = \texttt{GMR\_VNRangebased}(VNFW_\ell \cdot VNBW_\ell). \tag{4.4}$$

4.2.4 Final Decision

To estimate $\hat{\underline{b}}$ in the multi-Gaussian decoder, first an estimate $\hat{w}_k$ of the transmitted codeword element x_k for $k = 1, 2 \ldots, n$ is computed. In order to estimate the value of $\hat{w}_k$, first a Gaussian mixture is created around the channel message, y_k, using the set of triples obtained after the multiplication and Gaussian mixture reduction step at the variable node [9] and the peak value of this Gaussian mixture is the estimated codeword symbol $\hat{w}_k$.

Then, $\hat{\underline{b}}$ is estimated as

$$\hat{\underline{b}} = \lfloor H \cdot \underline{\hat{w}} \rceil, \tag{4.5}$$

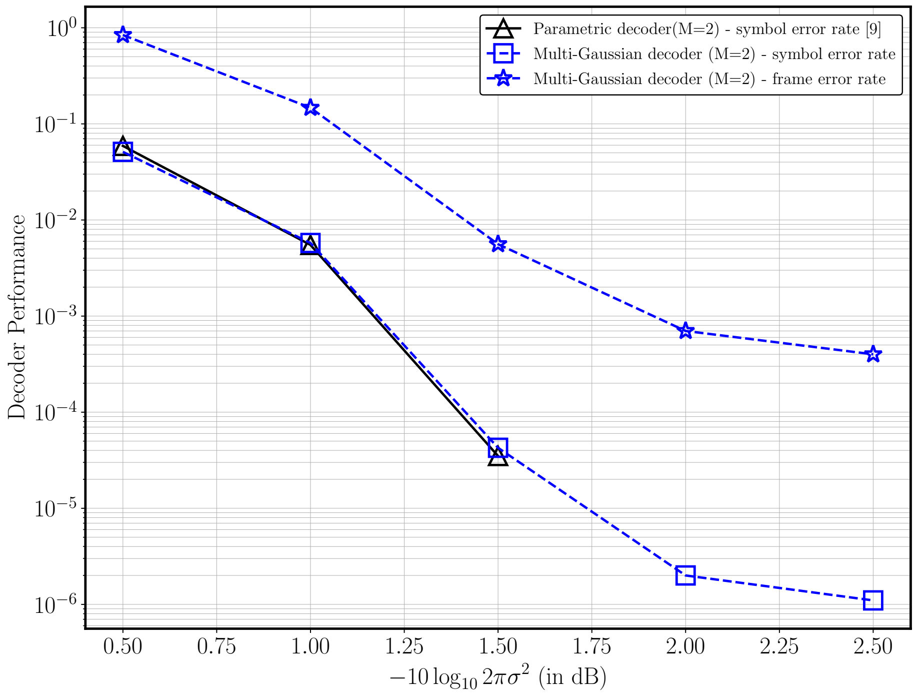


Figure 4.5: Frame error rate and symbol error rate performance of a multi-Gaussian LDLC decoder ($M = 2$) for block length $n = 1000$ and $d = 7$.

where $\lfloor \rceil$ denotes coordinate-wise integer rounding. Consistent with the decoding methodology of the single-Gaussian decoder, we do an early termination where the iterative decoding is terminated as soon as the decoder output is equal to the encoder input.

4.3 Frame Error Rate to Measure the Decoder Performance

The published results in the literature report the multi-Gaussian decoder SER performance for a degree of 7; therefore, to benchmark the multi-Gaussian decoder simulation, first the SER for block length $n = 1000$ and $d = 7$ is computed and compared with results in [9]. As seen in Fig. 4.5, the simulated SER shows good agreement with the results in [9]. In Fig. 4.5 the slope of the blue curves flatten in the higher SNR regime. This decoder performance is with floating point arithmetic so the flattening is not due to quantization error usually

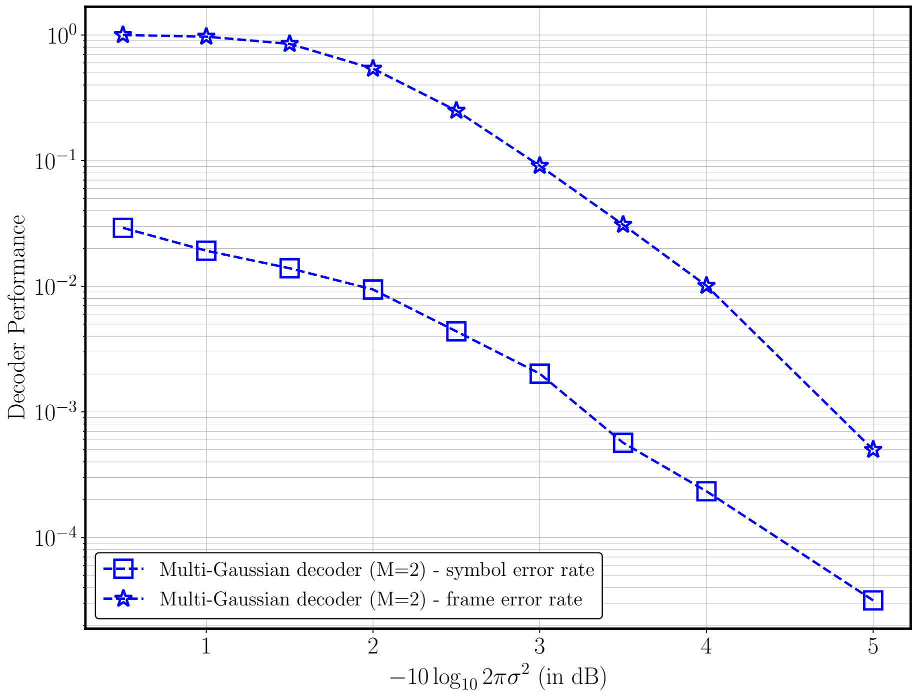


Figure 4.6: Frame error rate and symbol error rate performance of a multi-Gaussian LDLC decoder ($M = 2$) for block length $n = 1000$, $d = 3$, and 200 decoding iterations.

observed with fixed-point arithmetic. It may be a limitation of the code or decoding algorithm (in this thesis the LDLC decoding algorithm is taken from the literature [7] and the decoder performance is extended for high SNR values not reported in the literature).

Using this verified simulation setup, the SER and FER for a multi-Gaussian decoder for block length $n = 1000$ and $d = 3$ is simulated, as shown in Fig. 4.6.

Simulation results provided are for a multi-Gaussian decoder with $M = 2$, $n = 1000$ and degree, $d = 3$. The generating sequence is $\{1, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\}$ and random lattice codewords are in the integer range $\underline{b} \in \mathcal{L}^n$, where $\mathcal{L}$ = {-2, -1, 0, 1, 2}.

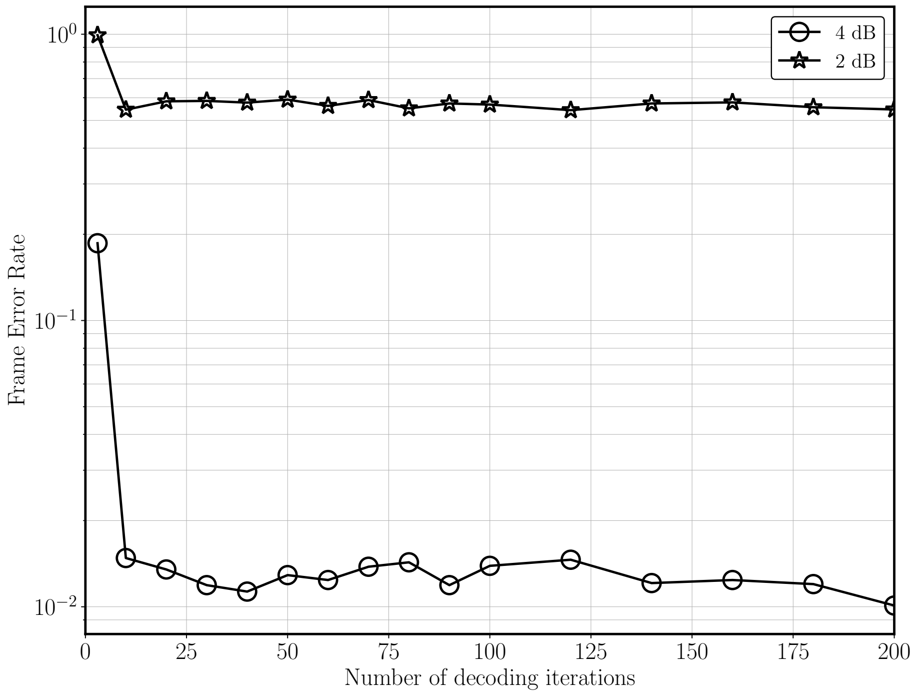


Figure 4.7: Frame error rate of the multi-Gaussian decoder (floating-point with no approximations) versus number of decoding iterations for $n = 1000$ and $d = 3$ at distance from capacity of 2 dB ad 4 dB.

4.4 Optimizations to Reduce the Decoder Complexity

4.4.1 Fixed-point Arithmetic for Hardware Implementation

To implement the multi-Gaussian decoder on a target FPGA, a fixed-point arithmetic implementation is favoured, as was the case for the single-Gaussian implementation. Consistent with the convention established for the single-Gaussian decoder in Chapter 3, in the multi-Gaussian decoder also each fixed point number is represented by $QW_i.W_f$, where W_i integer bits, W_f fractional bits (along with a sign bit).

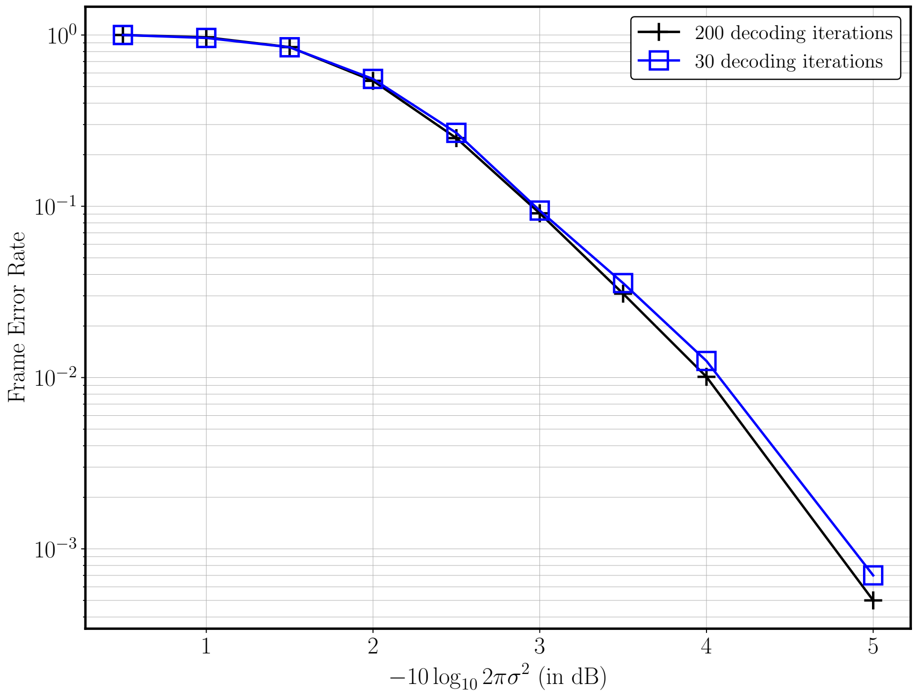


Figure 4.8: Frame error rate of a multi-Gaussian decoder (floating-point with no approximations) for $n = 1000$ and $d = 3$ with 200 and 30 decoding iterations.

4.4.2 Number of Decoding Iterations

In [6, 7, 9, 10, 12], the SERs for LDLC decoders are simulated for 200 decoding iterations. However, similar to the single-Gaussian decoder presented in Chapter 3, for reduced power consumption and a reasonable decoding latency fewer decoding iterations are preferred in the multi-Gaussian decoder.

Fig. 4.7 shows the decoder FER performance versus number of decoding iterations at a distance from capacity of 2 dB and 4 dB. As seen in the graph, 30 decoding iterations achieve comparable performance to 200 decoding iterations, albeit in significantly less run time. Fig. 4.8 shows a comparison of the decoding performance for 200 and 30 decoding iterations.

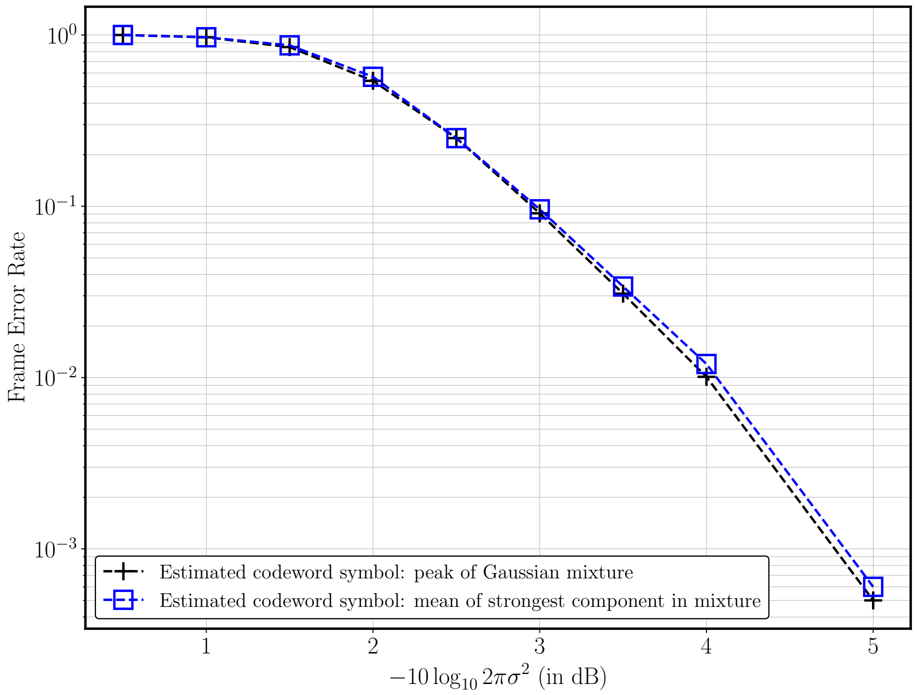


Figure 4.9: Frame error rate of a multi-Gaussian decoder using a method for final decision (in iterative decoding) where the mean of the strongest component in the Gaussian mixture is considered as the estimated codeword symbol, $\hat{w}_k$, for $n = 1000$, $d = 3$ and 30 decoding iterations.

4.4.3 Final Decision in Iterative Decoding Algorithm

As described in Section 4.2, for the final decision in the parametric decoder, at a variable node x_k, the channel message and all the incoming check node messages are multiplied, and then using the multiplication result, a Gaussian mixture is generated. The peak value of this Gaussian mixture is the estimated codeword symbol $\hat{w}_k$, further used for estimating the decoded integer vector $\underline{\hat{b}}$.

An exact implementation of the Gaussian mixture generation for the final decision at a variable node is resource and time intensive. For a practical decoder implementation it is important to devise a simpler method to estimate the codeword symbol $\hat{w}_k$. An intuitive approach is to take the mean of the strongest component in the Gaussian mixture as the estimate of the codeword symbol.

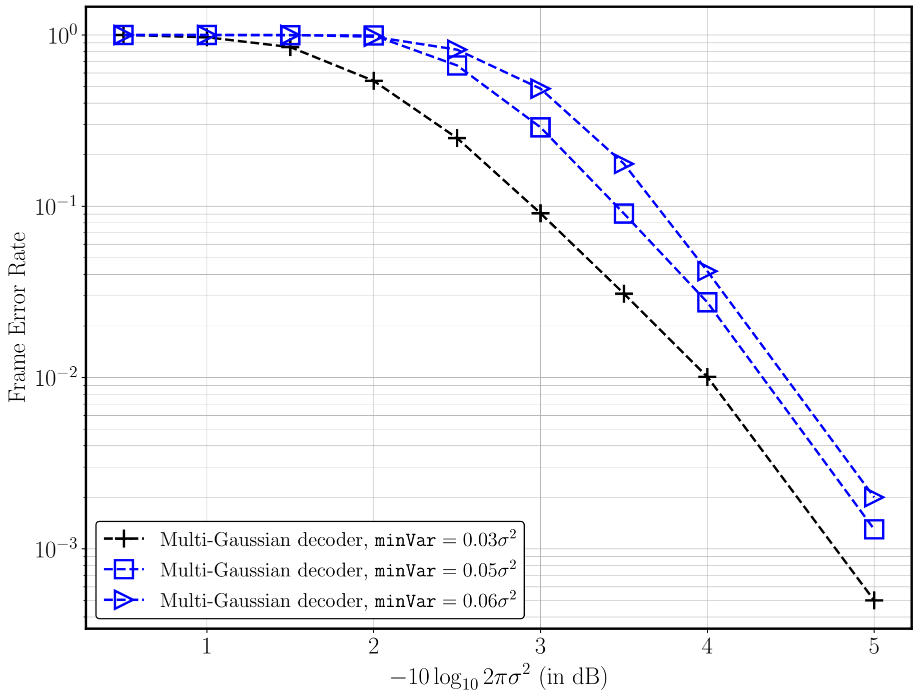


Figure 4.10: Frame error rate of a multi-Gaussian decoder (floating-point) with different minimum variance, `minVar` values for $n = 1000$ and $d = 3$.

The multi-Gaussian decoder FER is simulated using this approach. A decoder using this final decision method attains a frame error rate similar to the one where the peak of the Gaussian mixture is $\hat{w_k}$, as shown in Fig. 4.9, albeit with significantly less design complexity.

4.4.4 Minimum Variance

Following the convention established in the single-Gaussian decoder, for better numerical stability, at the variable nodes in the multi-Gaussian decoder, the smallest allowable variance is limited to a certain minimum value denoted by minvar.

Simulations are performed to investigate if a larger value for minvar can be used for the multi-Gaussian decoder without any significant degradation on the decoder performance. As seen in Fig. 4.10, the FER of the decoder deteriorates with minvar values larger than $0.03\sigma^2$, where σ^2 is the channel noise variance.

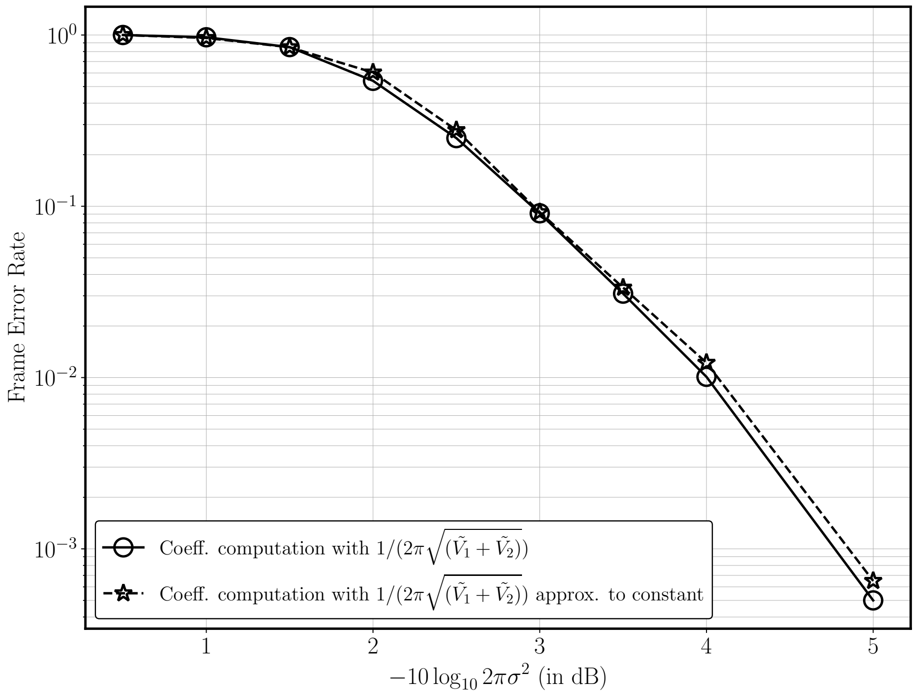


Figure 4.11: Frame error rate of a multi-Gaussian decoder (floating-point) for $n = 1000$ and $d = 3$ when $\frac{1}{2\pi\sqrt{(\tilde{V_1}+\tilde{V_2})}}$ component in the coefficient computation, i.e., (3.6) is approximated to a constant.

4.4.5 Variance Measured with Respect to the Channel Variance

Similar to the single-Gaussian decoder in Chapter 3, all variances in the multi-Gaussian decoder are measured relative to the channel variance.

4.4.6 Optimizations in the Coefficient Computation at the Variable Nodes

During the iterative decoding process of Section 4.2, at each variable node, a pairwise multiplication is computed for each possible pair of the components between the two Gaussian mixtures. The computation of $VNFW_i{\cdot}\texttt{MPeriodic}_i$ in Algorithm 6 described in Subsection 4.2.2 computes the product of a Gaussian mixture with two components ($VNFW_i$) with a larger Gaussian mixture, $\texttt{MPeriodic}_i$ with $M \cdot R$ components.

In Equation (3.6), i.e., $c_F = \frac{\tilde{c}_1\tilde{c}_2}{\sqrt{2\pi(\tilde{V}_1+\tilde{V}_2)}} e^{\frac{-(\tilde{m}_1-\tilde{m}_2)^2}{2(\tilde{V}_1+\tilde{V}_2)}}$, the term $\tilde{c}_1\tilde{c}_2 e^{\frac{-(\tilde{m}_1-\tilde{m}_2)^2}{2(\tilde{V}_1+\tilde{V}_2)}}$ dominates the overall value of c_F. Based on the simulation results shown in Fig. 4.11, replacing $\frac{1}{\sqrt{2\pi(\tilde{V}_1+\tilde{V}_2)}}$ with a constant (i.e., '1') does not significantly impact the decoder's performance. Therefore, we compute c_F as,

$$c_F = \tilde{c}_1\tilde{c}_2 e^{\frac{-(\tilde{m}_1-\tilde{m}_2)^2}{2(\tilde{V}_1+\tilde{V}_2)}}. \tag{4.6}$$

The weights in the computation of the product $VNBW_{(d-i+1)} \cdot \texttt{MPeriodic}_{(d-i+1)}$ are calculated in similar fashion.

Similar to the single-Gaussian decoder, the factor of $\frac{1}{2}$ in the exponent of (4.6) is accommodated in the exponentiation function approximation. Thus, the exponent is computed as

$$c_F = \tilde{c}_1\tilde{c}_2 e^{\frac{-(\tilde{m}_1-\tilde{m}_2)^2}{(\tilde{V}_1+\tilde{V}_2)}}. \tag{4.7}$$

For improved numerical stability, the coefficients generated during the product step are scaled such that the strongest exponential term, i.e, $e^{\frac{-(\tilde{m}_1-\tilde{m}_2)^2}{(\tilde{V}_1+\tilde{V}_2)}}$ is '1'.

4.5 Fixed Point Quantization Study

4.5.1 Approximation of Non-Linear Functions

The multi-Gaussian decoding algorithm has three non-trivial non-linear functions: division, exponentiation and square-root.

The square root function is used in the coefficient computation at the variable nodes (2.54) and to identify the strongest component in the mixture during GMR at the check nodes and variable nodes (Algorithm 3 in Subsection 2.5.2). With the explanation provided in Subsection 4.4.6, both the square root computations can be considered constant. Thus only two non-linear functions are required for decoder implementation: 1) division and 2) exponentiation.

Approximation of Division Function using Newton-Raphson

The multi-Gaussian fixed-point decoder uses the division function approximation devised in Subsection 3.5.1 where the division $\texttt{Qdiv}(u, a)$ is computed using (3.14).

The look-up table previously used in the single-Gaussian decoder to obtain the initial values for Newton-Raphson iteration is reused for the multi-Gaussian decoder.

Approximation of Exponential Function using LUTs

Similar to the single-Gaussian decoder, the exponent in the coefficient computation (4.7) at the variable nodes of the multi-Gaussian LDLC decoder (parametric decoder) is always non-positive.

Following the methodology developed to approximate $\exp(-a/2)$ for the single-Gaussian decoder in Chapter 3, for the multi-Gaussian decoder we calculate $\exp(-a/2)$ for $a \geq 0$ and first decompose the exponential function $\exp(-a/2)$ as the product of three easily computable terms as,

$$\exp(-a/2) = \exp(-I_2 2^{P_2}/2) \exp(-I_1 2^{P_1}/2) \exp(-I_0 2^{P_0}/2). \tag{4.8}$$

Initially, for the multi-Gaussian decoder we select $P_0 = -W_f$, P_2 as the smallest positive integer such that $\exp(-2^{P_2}/2)$ underflows the fixed-point representation of the LDLC decoder, and $P_1 = \lfloor (P_0 + P_2)/2 \rfloor$. Due to the choice of P_2, two small look-up tables are sufficient to compute $\exp(-I_1 2^{P_1}/2)$ and $\exp(-I_0 2^{P_0}/2)$.

Based on simulation results, P_1 is increased further (by 3 bits) without much change in the value of $\exp(-I_0 2^{P_0}/2)$. The value of component, $\exp(-I_0 2^{P_0}/2)$ does not vary much and is replaced by '1' in exponential approximation with no significant effect on the decoder performance. Hence, only one lookup table to compute $\exp(-I_1 2^{P_1}/2)$ is sufficient for exponential approximation, and (4.8) can be re-written as

$$\exp(-a/2) = \exp(-I_2 2^{P_2}/2) \exp(-I_1 2^{P_1}/2). \tag{4.9}$$

Fig. 4.12 illustrates the relationship between a and I_0, I_1 and I_2 for the multi-Gaussian decoder.

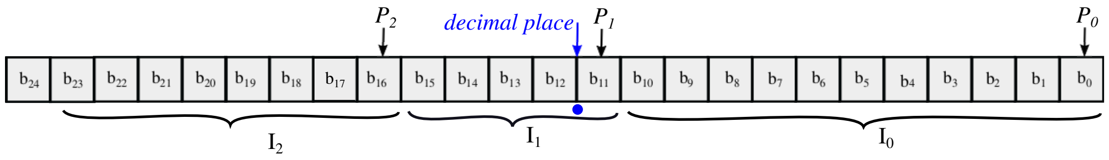

P_2
decimal place
P_1
P_0
b_{24} b_{23} b_{22} b_{21} b_{20} b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
I_2
I_1
I_0

Figure 4.12: Diagram to show the relationship between a and I_0, I_1 and I_2, as used in the approximation of the exponential function in fixed-point arithmetic at the variable nodes in the multi-Gaussian decoder.

4.5.2 Optimal word length and Newton-Raphson (NR) iterations for fixed-point decoder

To keep the hardware cost low, we aim to obtain the minimum possible word length for the fixed-point representation that can attain the floating point decoding performance. Simulations are performed to find decoder performance for different values of W_i and W_f. Fig. 4.13 compares the FERs for different values of W_f while keeping W_i large and varying the number of NR iterations for block length of 1000 and degree, 3. Fig. 4.14 compares decoder performance for different values of W_i while W_f is fixed.

As seen in Fig. 4.13, in the multi-Gaussian decoder, fixed-point performance improves with smaller W_f (up to a certain limit) compared to the floating-point decoder. If W_f is reduced any further, then the FER performance deteriorates. This is similar to the trend observed for the single-Gaussian LDLC decoder in Subsection 3.5.2, where the decoder performance slightly improves with smaller number of fractional bits.

Fig. 4.14 plots the effect of different numbers of integer bits on the decoder performance. Similar to the single-Gaussian decoder, the degradation seen in the the decoder performance with smaller W_i is due to the computation errors that occur from the saturation in multiplication operations.

Based on the results in Fig. 4.13 and Fig. 4.14, a word length of 25 with 12 integer bits, 12 fractional bits and one sign bit is an appropriate choice for a fixed-point representation.

4.6 LDLC Decoder FPGA Implementation

In the previous section, we investigated the quantization requirements for a multi-Gaussian decoder implementation in fixed-point arithmetic and characterized the FER performance. Here, we present the decoder architectures implemented on a target FPGA, along with the

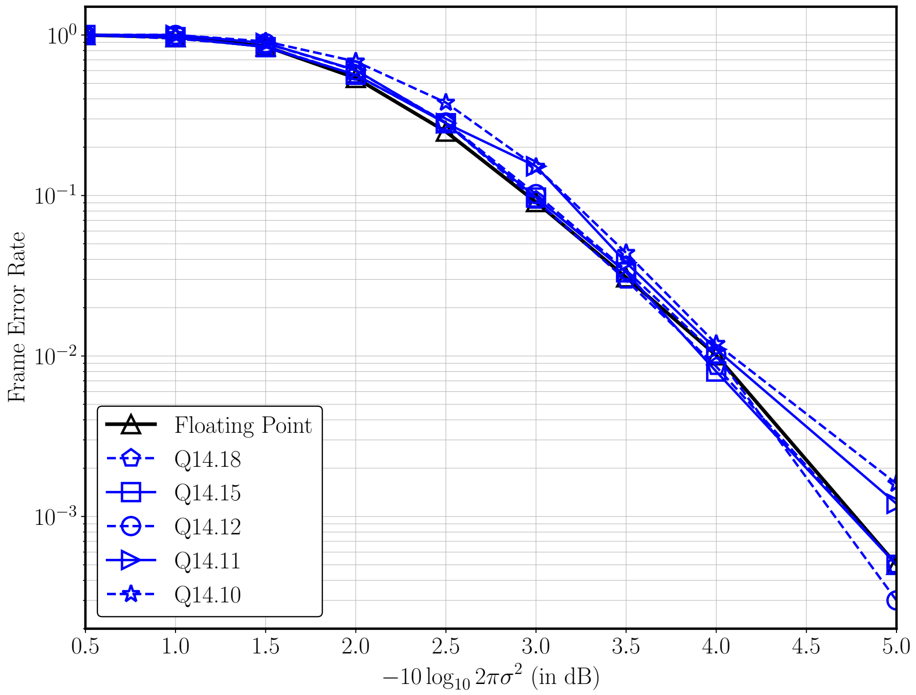


Figure 4.13: FER for different numbers of fractional bits and two NR iterations for $n = 1000$ and $d = 3$ where $-10\log_{10} 2\pi e\sigma^2$ is distance from the theoretical noise limit.

resource requirements and throughput results: D) an architecture with a single check node and a single variable node, E) an architecture with a single check node and with pipelining to achieve an effective parallelism of 5 variable nodes.

Architecture D) A single check node and a single variable node: A serial decoder with a single check node and a single variable node is chosen for the preliminary multi-Gaussian decoder hardware implementation. The estimate of the resource requirements of a serial decoder is important to benchmark the benefits of adding parallelism in the decoder implementation.

Two separate dual-ported memory banks are used to save the check node and variable node messages generated during iterative decoding. Read-only-memories (ROMs) are used to store check node connections to variable nodes and vice-versa, according to the H matrix. The edge weights of the connections are stored in a separate ROM. A block diagram of the two-node serial multi-Gaussian decoder in shown in Fig. 4.15.

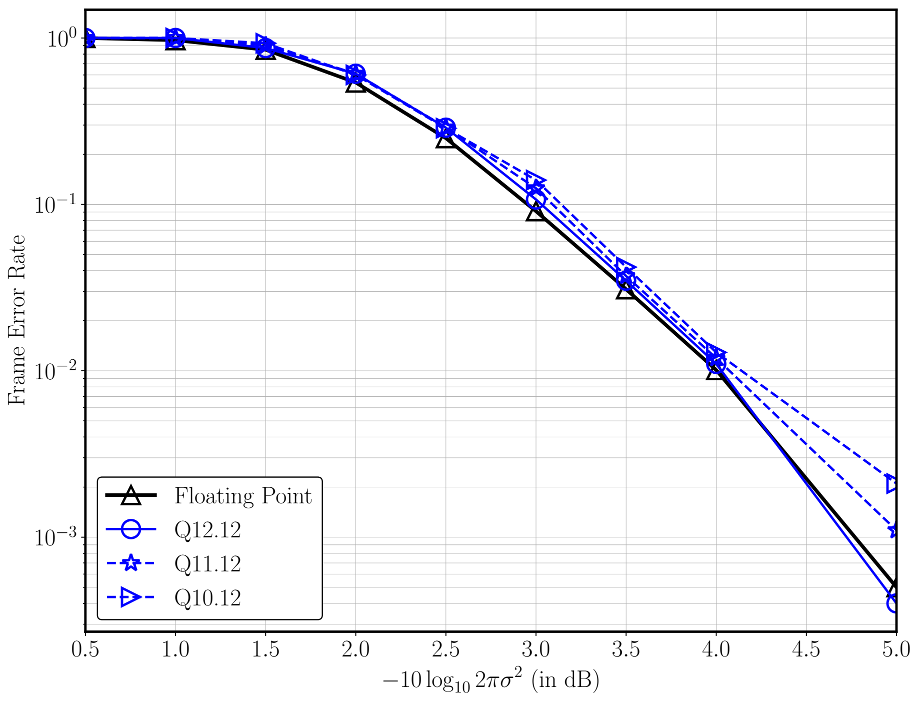


Figure 4.14: FER performance of the multi-Gaussian decoder with different numbers of integer bits and two NR iterations for $n = 1000$ and $d = 3$.

To compute the outgoing messages from a check node, c_k, the *message routing network* finds out the three variable node connections to c_k and the associated edge weights from the respective ROMs. Then, it reads corresponding means, variances and coefficients from the variable node message memory and the *check node message processing block* computes the outgoing messages. Likewise, the *variable node message processing block* receives the check node messages and computes the outgoing variable node messages.

Check node message processing block In multi-Gaussian LDLC decoder, unlike the single-Gaussian decoder the check node message processing block is complex.

The top-level architecture of the check node unit (`CNU`) is presented in Fig. 4.16. It contains two blocks, `CNFWBW` computation block which computes the forward-backward messages and `CNVOut` computation block which reads in the forward-backward messages and calculates the outgoing check node messages. Fig. 4.17 depicts the timing diagram for the check node unit in architectures D and E.

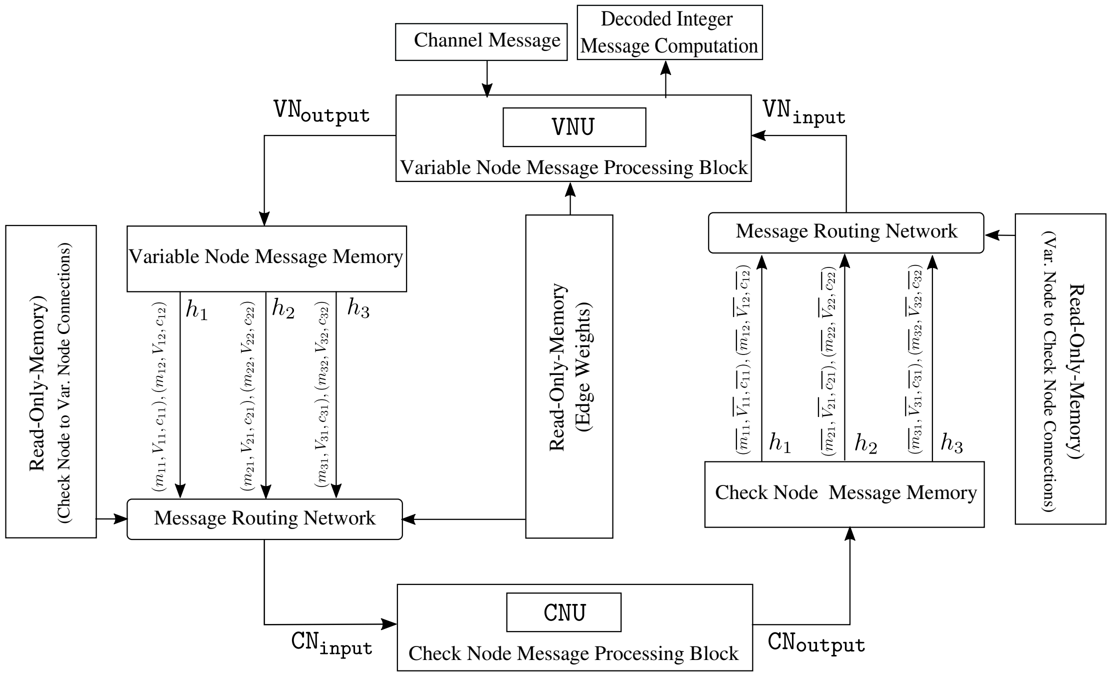
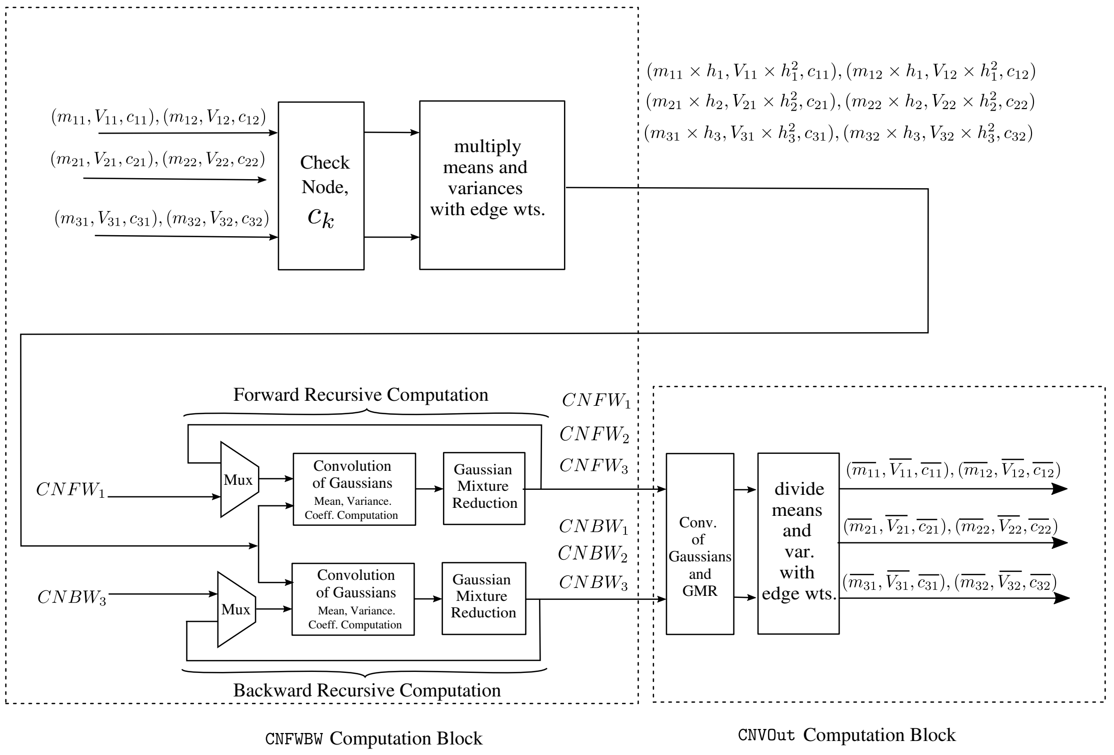

Channel Message
Decoded Integer Message Computation
VN$_{\text{output}}$
VNU
Variable Node Message Processing Block
VN$_{\text{input}}$
Message Routing Network
Read-Only-Memory (Check Node to Var. Node Connections)
Variable Node Message Memory
Read-Only-Memory (Edge Weights)
Read-Only-Memory (Var. Node to Check Node Connections)
h_1
h_2
h_3
Check Node Message Memory
Message Routing Network
CN$_{\text{input}}$
CNU
Check Node Message Processing Block
CN$_{\text{output}}$

Figure 4.15: Block diagram of a two-node serial multi-Gaussian LDLC decoder with one single check node unit, CNU and one single variable node unit, VNU (architecture D).

Variable node message processing block As described in Subsection 4.2.3, to compute the outgoing message along an edge at each variable node, first the $d-1$ incoming check node messages are periodically extended and then we iteratively calculate the product of the channel message and the periodically extended messages, except for the one associated with that edge. After every product computation, the result is reduced to a Gaussian mixture with only two components using Gaussian mixture reduction method (Algorithm 3 described in Subsection 2.5.2). The top-level architecture of the variable node is presented in Fig. 4.18. The timing diagram for the variable node message processing block is shown in Fig. 4.19.

Table. 4.1 shows the hardware resources, i.e., adaptive logic modules (ALMs), registers and DSPs used for the check node and variable node implementation in the serial implementation. At a clock frequency of 105 MHz, architecture D attains a throughput of 245 Ksymbols/sec at a distance of 5 dB from capacity, as shown in Fig. 4.20.



$(m_{11}, V_{11}, c_{11}), (m_{12}, V_{12}, c_{12})$
$(m_{21}, V_{21}, c_{21}), (m_{22}, V_{22}, c_{22})$
$(m_{31}, V_{31}, c_{31}), (m_{32}, V_{32}, c_{32})$
Check Node, c_k
multiply means and variances with edge wts.
$(m_{11} \times h_1, V_{11} \times h_1^2, c_{11}), (m_{12} \times h_1, V_{12} \times h_1^2, c_{12})$
$(m_{21} \times h_2, V_{21} \times h_2^2, c_{21}), (m_{22} \times h_2, V_{22} \times h_2^2, c_{22})$
$(m_{31} \times h_3, V_{31} \times h_3^2, c_{31}), (m_{32} \times h_3, V_{32} \times h_3^2, c_{32})$
Forward Recursive Computation
$CNFW_1$
Mux
Convolution of Gaussians Mean, Variance. Coeff. Computation
Gaussian Mixture Reduction
$CNFW_1$
$CNFW_2$
$CNFW_3$
$CNBW_1$
$CNBW_2$
$CNBW_3$
$CNBW_3$
Mux
Convolution of Gaussians Mean, Variance. Coeff. Computation
Gaussian Mixture Reduction
Backward Recursive Computation
Conv. of Gaussians and GMR
divide means and var. with edge wts.
$(\overline{m_{11}}, \overline{V_{11}}, \overline{c_{11}}), (\overline{m_{12}}, \overline{V_{12}}, \overline{c_{12}})$
$(\overline{m_{21}}, \overline{V_{21}}, \overline{c_{21}}), (\overline{m_{22}}, \overline{V_{22}}, \overline{c_{22}})$
$(\overline{m_{31}}, \overline{V_{31}}, \overline{c_{31}}), (\overline{m_{32}}, \overline{V_{32}}, \overline{c_{32}})$
CNFWBW Computation Block
CNVOut Computation Block

Figure 4.16: High-level architecture of a check node unit (CNU) in the multi-Gaussian decoder of degree 3.

Architecture E) A single check node and pipelining to achieve an effective parallelism equivalent to 5 variable nodes: Based on the hardware resource requirement for architecture D shown in Table 4.1, we conclude that there are resources available on the target FPGA that can be utilized to add parallelism.

In architecture D, a check node takes 59 clock cycles to calculate a check node message while a variable node requires 329 clock cycles to compute a variable node message. In order to achieve balanced check node and variable node message computation time overall, it is reasonable choice to add more variable nodes to architecture D.

In variable node unit (VNU), shown in Fig. 4.18, the VNFWBW computation block requires 268 clock cycles to compute forward-backward messages while the calculations in the VNVOut computation block take 53 clock cycles. This implies that one VNVOut computation block is adequate to process the output for 5 VNFWBW computation blocks.

As shown in Fig. 4.21, in architecture E, the check node message processing block

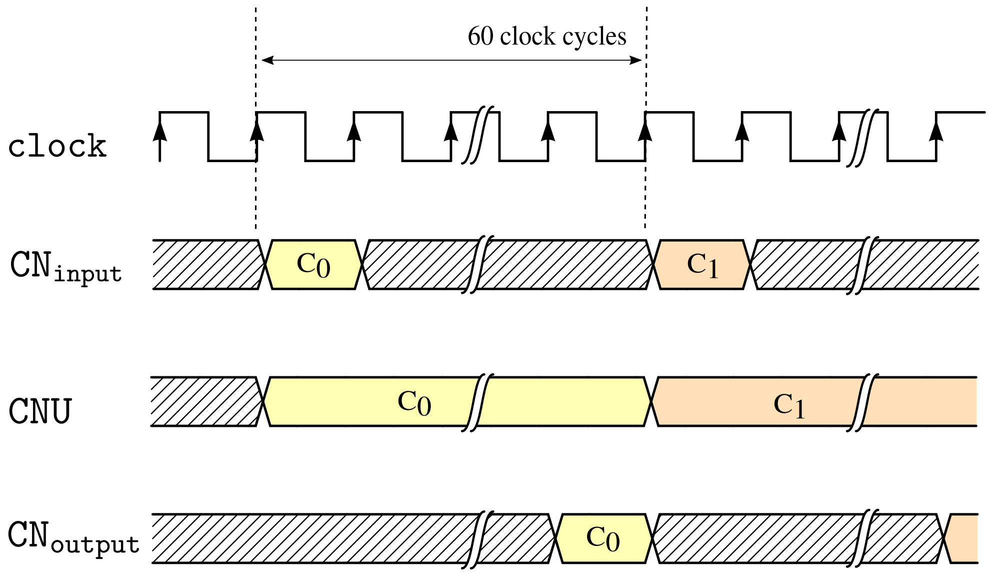
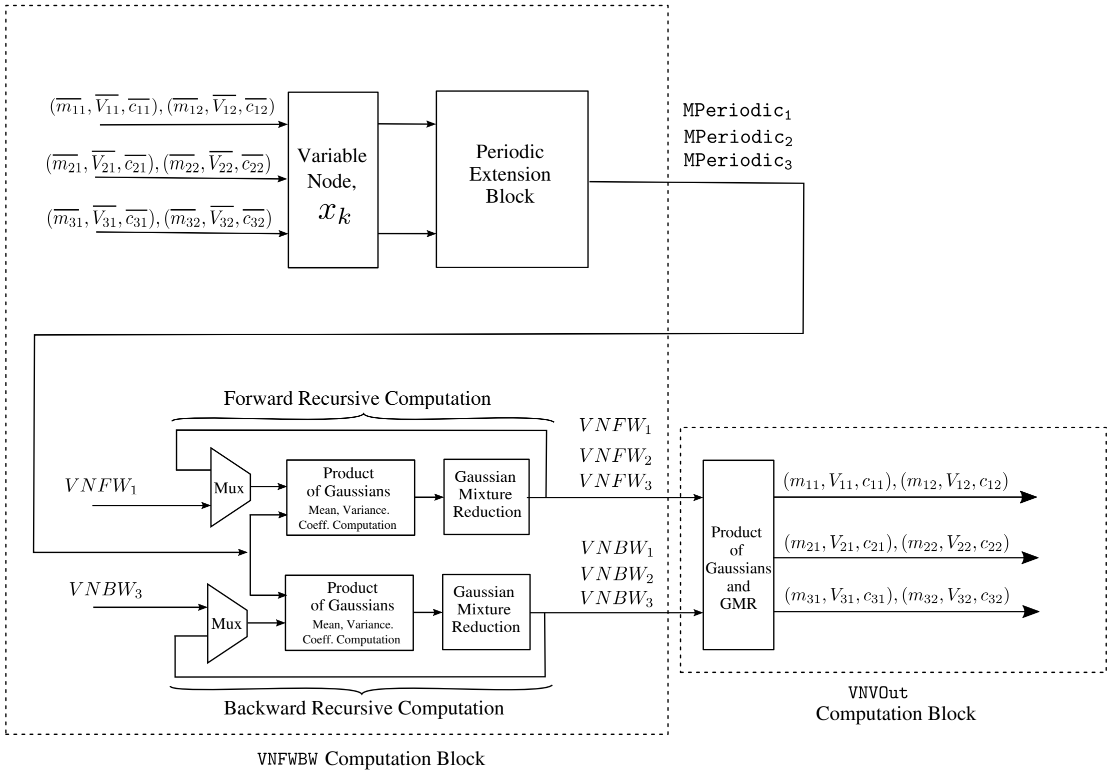


Figure 4.17: Timing diagram of the check node message processing block in both architectures D and E of the multi-Gaussian decoder.

contains a single check node unit and the variable node message processing block consists of a `MNUCluster` block. The `MNUCluster` block (shown in Fig. 4.22) is a two-stage pipeline, the first stage consists of five parallel `VNFWBW` computation blocks and the second stage, i.e., `VNVOut` computation block reads in the output of first pipeline stage and generates the outgoing variable node message. Fig. 4.23 shows timing diagram of the variable node message processing block in architecture E.

The resource requirements for architecture E are provided in Table 4.2. The two-stage pipelined architecture used in the variable node message processing block helps to attain an overall 2.25× improvement over a two node serial implementation, i.e., architecture D, as depicted in Fig. 4.24. However, this throughput improvement is obtained with additional hardware resources, as shown in Table 4.2. Further optimizations can improve the critical path in architecture D and it can run at a higher frequency. However architecture E has large usage of resources and complex routing, that limits the operating frequency.

An implementation with three parallel variable nodes does better than architecture E but does not fit on the board; the implementation is limited by the available DSPs on the FPGA. To obtain higher throughput compared to the two variable node implementation and fit the design on the FPGA, the partially parallel architecture (E) is implemented.



$(\overline{m_{11}}, \overline{V_{11}}, \overline{c_{11}}), (\overline{m_{12}}, \overline{V_{12}}, \overline{c_{12}})$
$(\overline{m_{21}}, \overline{V_{21}}, \overline{c_{21}}), (\overline{m_{22}}, \overline{V_{22}}, \overline{c_{22}})$
$(\overline{m_{31}}, \overline{V_{31}}, \overline{c_{31}}), (\overline{m_{32}}, \overline{V_{32}}, \overline{c_{32}})$
Variable Node, x_k
Periodic Extension Block
MPeriodic$_1$
MPeriodic$_2$
MPeriodic$_3$
Forward Recursive Computation
$VNFW_1$
Mux
Product of Gaussians Mean, Variance. Coeff. Computation
Gaussian Mixture Reduction
$VNFW_1$
$VNFW_2$
$VNFW_3$
$VNBW_3$
$VNBW_1$
$VNBW_2$
$VNBW_3$
Backward Recursive Computation
VNFWBW Computation Block
Product of Gaussians and GMR
$(m_{11}, V_{11}, c_{11}), (m_{12}, V_{12}, c_{12})$
$(m_{21}, V_{21}, c_{21}), (m_{22}, V_{22}, c_{22})$
$(m_{31}, V_{31}, c_{31}), (m_{32}, V_{32}, c_{32})$
VNVOut Computation Block

Figure 4.18: High-level architecture of a variable node unit (VNU) in multi-Gaussian decoder of degree 3.

4.7 Comparison of Single-Gaussian and Multi-Gaussian LDLC Decoder ($M = 2$)

Fig. 4.25 compares decoding performance of the single-Gaussian decoder ($M = 1$) and multi-Gaussian decoder ($M = 2$) for block length of 1000 and degree 3, where the multi-Gaussian decoder achieves an improvement of ~ 0.75 dB over the single-Gaussian decoder (a much higher gain in decoding performance is obtained with higher degrees, presented in Chapter 5 of the thesis). This is primarily due to the fact that the Gaussian mixture messages that are exchanged in the multi-Gaussian decoder during the iterative decoding process contain more components and also a more accurate Gaussian mixture reduction method is applied.

Table. 3.4 and Table 4.2 show the resource requirements for the single-Gaussian decoder

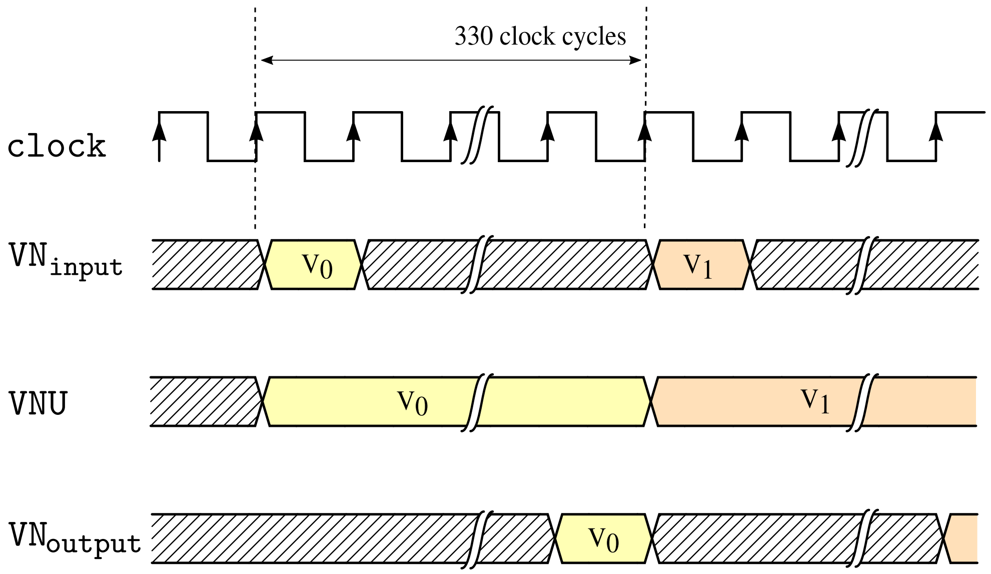


Figure 4.19: Timing diagram of the variable node message processing block in architecture D of the multi-Gaussian decoder.

(architecture C) and the multi-Gaussian decoder (architecture E). Due to huge hardware requirements of the check node and variable nodes in the multi-Gaussian decoder, only a parallelism equivalent of 5 variable nodes can be obtained with the available resources on the target FPGA attaining an throughput of 550 Ksymbols/sec.

Based on the results in Fig. 4.25, we can conclude that the multi-Gaussian decoder achieves ~ 0.75 dB improvement in decoding performance compared to the single-Gaussian decoder. However, due a simpler design and smaller resource requirements, it was possible to add more parallelism in the variable node message processing block of the single-Gaussian decoder and thus the single-Gaussian decoder throughput is significantly higher.

4.8 Summary

In this chapter, we studied different aspects of the hardware implementation of a multi-Gaussian LDLC decoder on a field-programmable device.

First the selection criteria to choose a multi-Gaussian decoder for an FPGA implementation was presented. Then, we described an iterative decoding algorithm for the multi-Gaussian decoder (parametric) where the messages exchanged between a check node

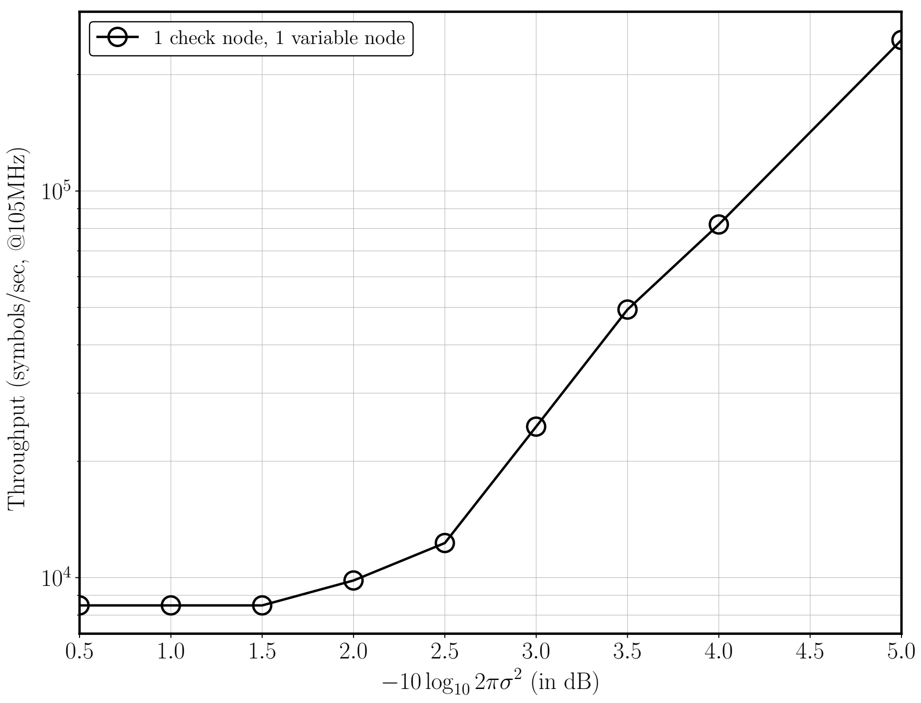


Figure 4.20: Throughput of the multi-Gaussian decoder (degree 3) with a single check node and a single variable node.

and a variable node are denoted by two triples of Gaussian parameters, i.e., means, variance and coefficients.

We simulated the multi-Gaussian decoder in floating-point and achieved decoding performance identical to that published in the literature for block length of 1000 and degree of 7 (this validated of our simulation setup). We then characterized the SER and FER of the multi-Gaussian decoder for degree of 3. Then, we discussed various optimizations techniques that can be applied to the multi-Gaussian decoder design in order to achieve feasible hardware using limited resources. Further, a quantization analysis was presented to approximate required non-linear functions, i.e, division and exponentiation in fixed-point arithmetic and we characterized decoder performance for different number of fractional and integer bits. These simulations helped to determine an optimal word length for the fixed-point implementation of the decoder.

We also described different architectural implementations of the multi-Gaussian LDLC decoder. A serial two-node (a single check node and a single variable node) architecture,

Table 4.1: Resource usage of the LDLC decoder, check node and variable node processing blocks in architecture D.

Resource	Decoder	Check Node	Variable node
ALM	120342	9877	68640
Dedicated Regs.	87436	10083	46048
DSPs	672	105	564
BRAM	48	0	0

Table 4.2: Resource usage of the decoder, check node and variable node processing blocks in architecture E.

Resource	LDLC decoder	Check Node	Variable node
ALM	371645	11009	317433
Dedicated Regs.	203044	10828	160118
DSPs	1422	65	1354
BRAM	48	0	0

and then a partially parallel pipelined architecture (with a single check node and a two-stage pipelining in the variable node message processing block) was implemented.

The multi-Gaussian decoder implementation achieved an FER improvement of 0.75 dB compared to the single-Gaussian decoder for block length, 1000 and degree of 3. The design throughput of partially parallel implementation of multi-Gaussian decoder was 550 Ksymbols/sec at a clock frequency of 105 MHz.

In the next chapter, we present single-Gaussian and multi-Gaussian LDLC decoders for degrees 5 and 7 and study the trade-offs between the frame error rate and design throughput of these decoders on a target FPGA. We also study the decoding performance of the multi-Gaussian decoders with larger number of components in the Gaussian mixture messages exchanged during the iterative decoding ($M = 5$).

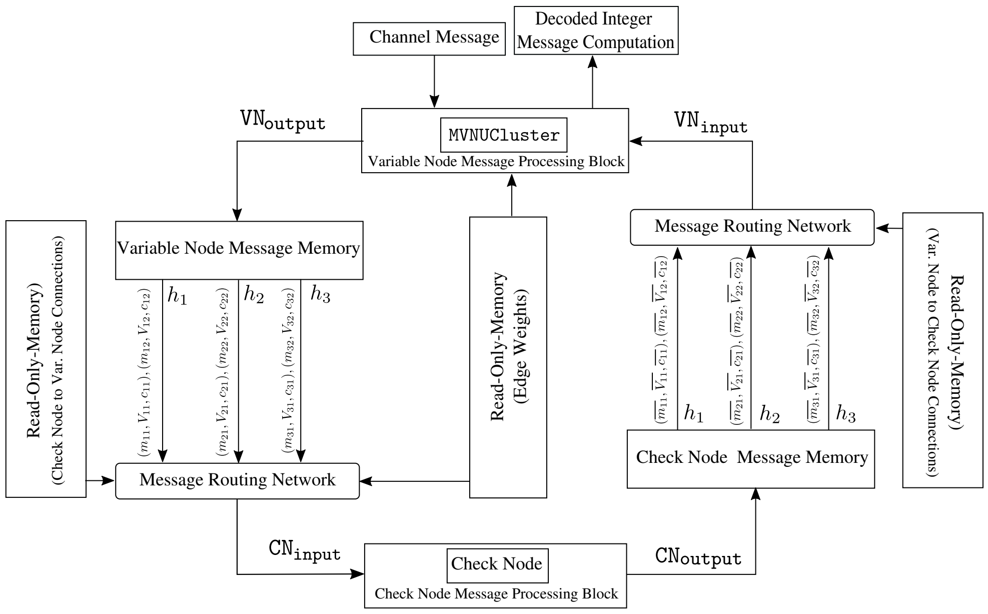

Channel Message
Decoded Integer Message Computation
VN$_{\mathtt{output}}$
MVNUCluster
Variable Node Message Processing Block
VN$_{\mathtt{input}}$
Variable Node Message Memory
Read-Only-Memory (Edge Weights)
Message Routing Network
Read-Only-Memory) (Var. Node to Check Node Connections)
Read-Only-Memory) (Check Node to Var. Node Connections)
h_1
h_2
h_3
$(m_{11}, V_{11}, c_{11}), (m_{12}, V_{12}, c_{12})$
$(m_{21}, V_{21}, c_{21}), (m_{22}, V_{22}, c_{22})$
$(m_{31}, V_{31}, c_{31}), (m_{32}, V_{32}, c_{32})$
$(\overline{m_{11}}, \overline{V_{11}}, \overline{c_{11}}), (\overline{m_{12}}, \overline{V_{12}}, \overline{c_{12}})$
$(\overline{m_{21}}, \overline{V_{21}}, \overline{c_{21}}), (\overline{m_{22}}, \overline{V_{22}}, \overline{c_{22}})$
$(\overline{m_{31}}, \overline{V_{31}}, \overline{c_{31}}), (\overline{m_{32}}, \overline{V_{32}}, \overline{c_{32}})$
Check Node Message Memory
Message Routing Network
CN$_{\mathtt{input}}$
Check Node
Check Node Message Processing Block
CN$_{\mathtt{output}}$

Figure 4.21: Block diagram of a partially parallel multi-Gaussian LDLC decoder with one single check node and variable node message processing block with pipelining to achieve an effective parallelism equivalent to 5 variable nodes (architecture E).

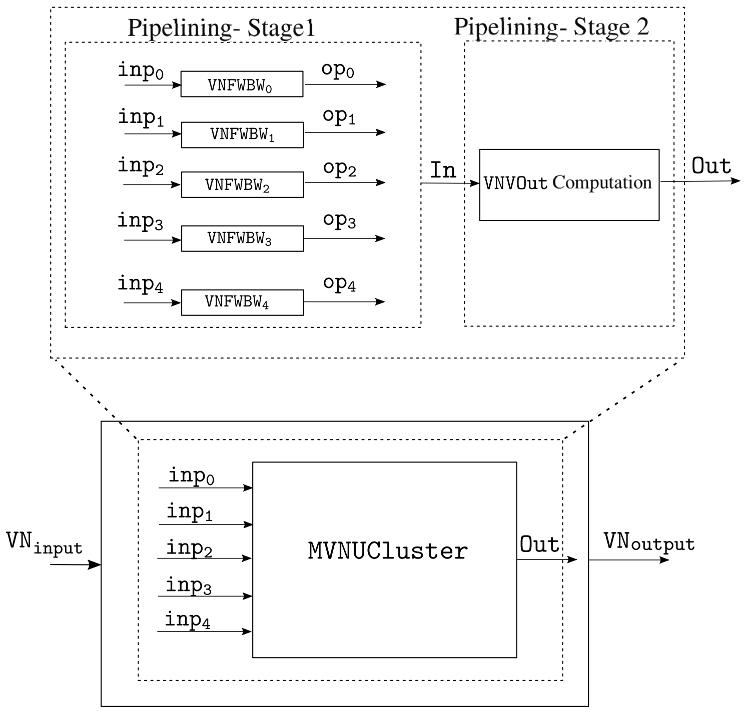


Figure 4.22: Block diagram of two-stage pipelining used in the variable node message processing block (architecture E) of the multi-Gaussian decoder .

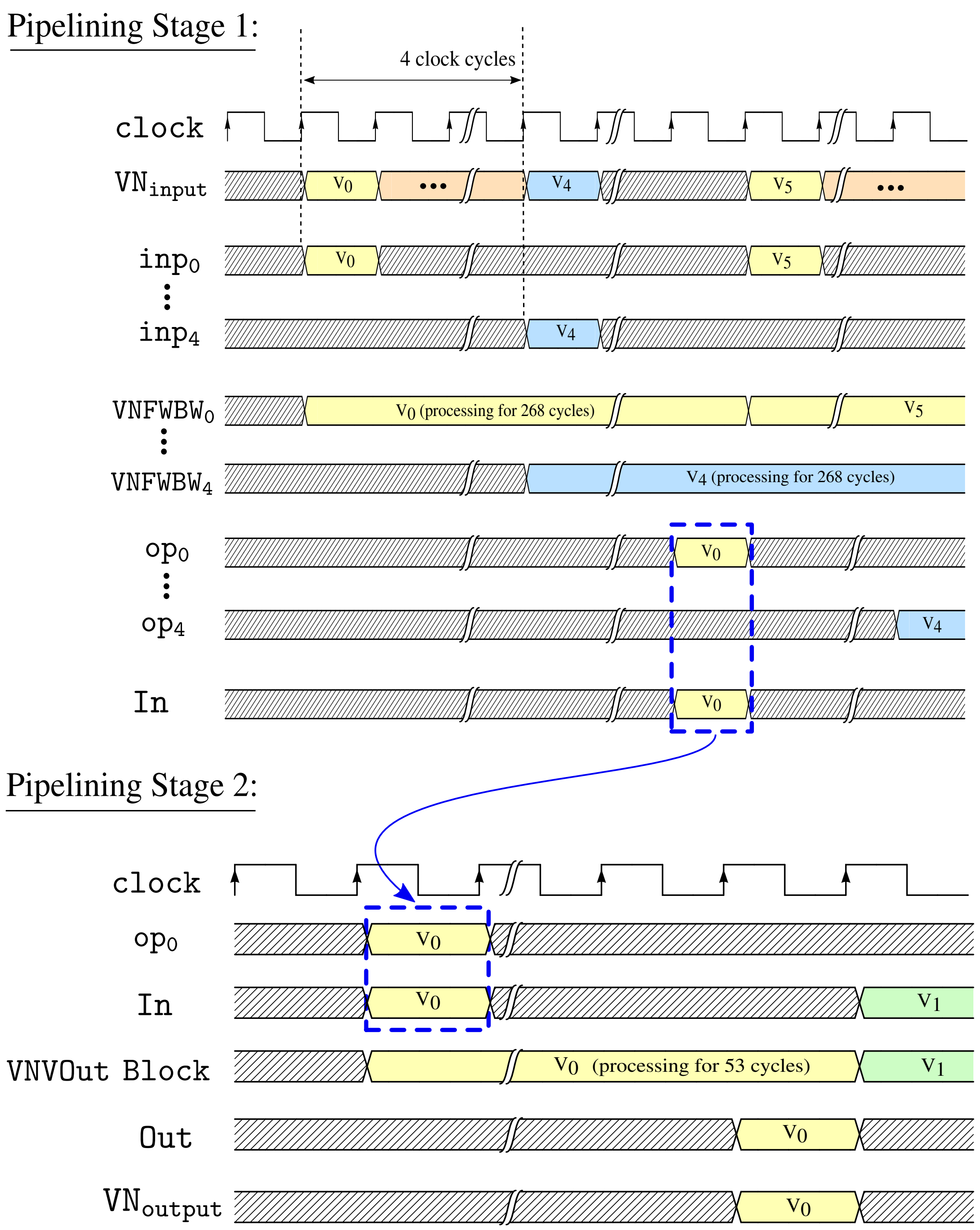


Figure 4.23: Timing diagram of the variable node message processing block in architecture E of the multi-Gaussian decoder.

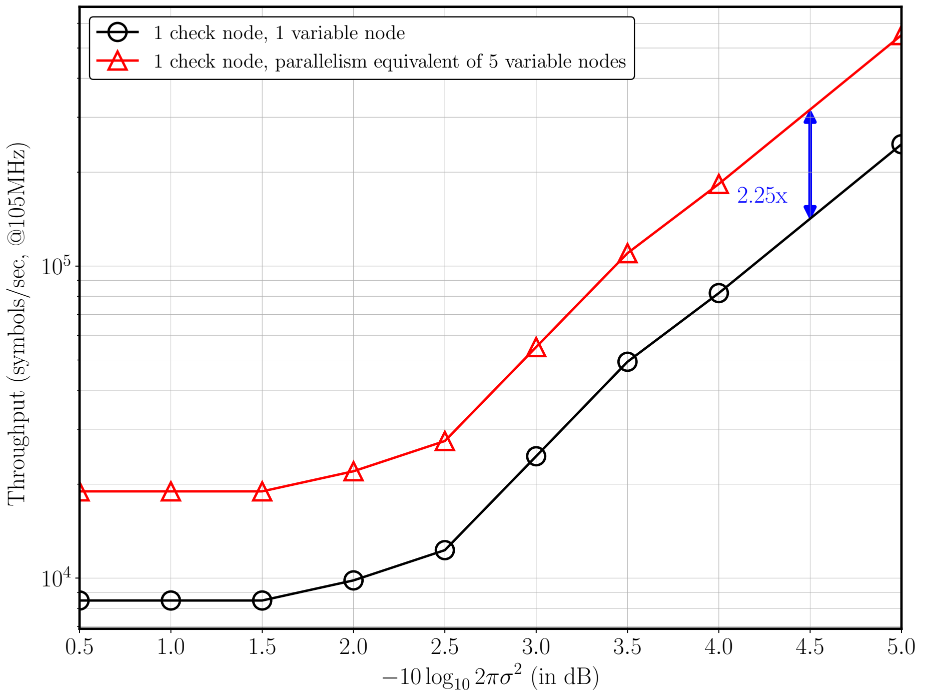


Figure 4.24: Throughput comparison of architectures D and E (of the multi-Gaussian LDLC decoder) for block length 1000 and degree 3.

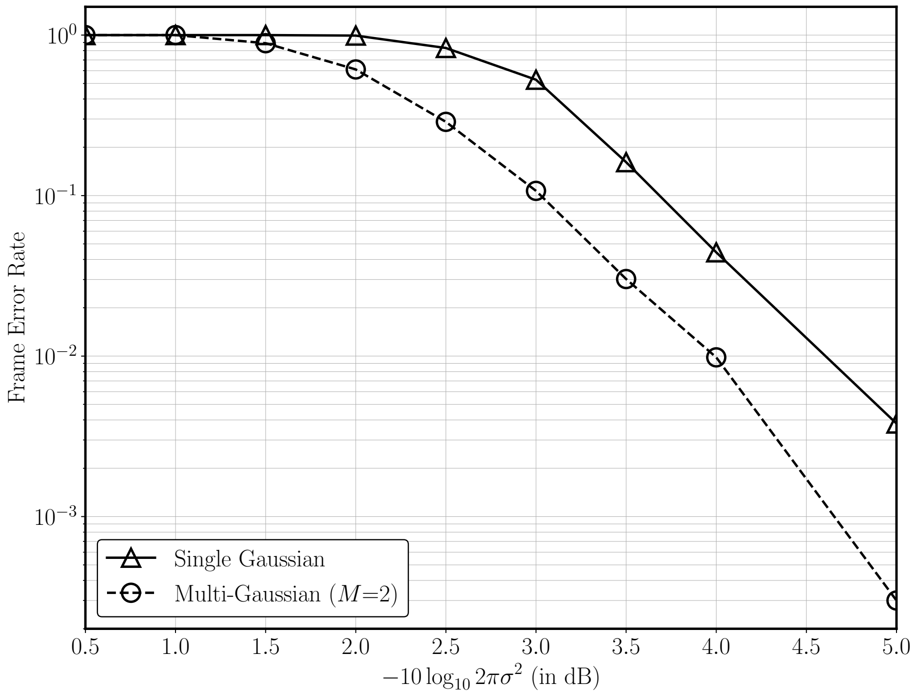


Figure 4.25: FER comparison of the single-Gaussian and multi-Gaussian fixed-point LDLC decoder ($M = 2$) implementations for block length of 1000 and degree 3.

Chapter 5

Pushing the Design Limits

In Chapter 3, we implemented a *single-Gaussian LDLC decoder*, where the continuous functions (Gaussian mixtures) exchanged in the sampled PDF decoder [7] are approximated with a single Gaussian using a set of Gaussian parameters, namely mean and variance. After a detailed quantization analysis to determine the minimum word length for the fixed-point representation, serial and partially parallel architectures of the single-Gaussian decoder were implemented on the target FPGA for a block length of 1000 and degree 3.

Based on the knowledge gained from the single-Gaussian decoder, in Chapter 4 we presented *multi-Gaussian decoder* implementations for degree 3, where the messages exchanged in the iterative decoder are reduced to Gaussian mixtures with two components represented by two sets of triples of means, variances and coefficients. A comprehensive study of the quantization requirements, along with efficient methods to approximate the required non-linear functions, helped to achieve fixed-point implementations of a multi-Gaussian decoder. A *i)* two-node serial (with a single check node and a single variable node) and *ii)* a partially parallel decoder architecture (with a single check node and with pipelining to achieve an effective parallelism equivalent of 5 variable nodes) were implemented on the FPGA device for block length of 1000. The pipelined architecture attained an overall 2.25$\times$ improvement in throughput over the two node serial implementation.

Chapter 4 also included a comparison of decoding performance and throughput of the single-Gaussian and multi-Gaussian decoders. The multi-Gaussian decoder attained an improvement of $\sim$ 0.75 dB in frame error rate, compared to the single-Gaussian decoder for degree of 3. However, the improvement in the decoding performance is obtained at additional hardware cost and design throughput was reduced by a factor of 19.

In this chapter, we implement single-Gaussian and multi-Gaussian LDLC decoders (with M =2) for degrees 5 and 7 on the target FPGA and study the trade-offs between frame error rate performance and design throughput. Higher degree decoders are expected to gain in decoding performance at the cost of additional hardware for implementation. We also characterize the decoding performance of the multi-Gaussian decoders where the Gaussian mixture messages contain larger number of components, i.e., $M = 5$.

Simulation results provided are with generating sequence $\{1, \frac{1}{\sqrt{5}}, \frac{1}{\sqrt{5}}, \frac{1}{\sqrt{5}}, \frac{1}{\sqrt{5}}\}$ for degree 5, and $\{1, \frac{1}{\sqrt{7}}, \frac{1}{\sqrt{7}}, \frac{1}{\sqrt{7}}, \frac{1}{\sqrt{7}}, \frac{1}{\sqrt{7}}, \frac{1}{\sqrt{7}}\}$ for degree 7. The random lattice codewords are obtained in the integer range $\underline{b} \in \mathcal{L}^n$, where $\mathcal{L}$ = {-2, -1, 0, 1, 2}.

5.1 Single-Gaussian Decoders

5.1.1 Single-Gaussian Decoder, $d = 5$

Optimal Number of Decoding Iterations Simulations are performed to obtain the ideal number of decoding iterations for a floating-point single-Gaussian decoder of degree 5. Similar to the single-Gaussian decoder of degree 3, only 20 decoding iterations are needed to attain a frame error rate comparable to 200 decoding iterations, as shown in Fig. 5.1.

Fixed-Point Quantization Analysis We use the methods devised in Chapter 3 to approximate division and exponentiation functions.

In order to obtain the word length for the fixed-point decoder, we performed simulations for different values of W_i and W_f. Fig. 5.2 compares the FER for different values of W_f while keeping W_i large for block length of 1000 and degree 5 while Fig. 5.3 presents decoder performance for different values of W_i when W_f is fixed.

Based on the results in Fig. 5.2 and Fig. 5.3, a word length of 28 with 12 integer bits, 15 fractional bits and a sign bit is a suitable choice for a fixed-point representation.

FPGA Implementation We implement a partially parallel architecture for the single-Gaussian decoder with degree 5 as shown in Fig. 5.4.

Similar to Chapter 3, the check and variable node message processing blocks receive the means and variances of the connected nodes and the associated edge weights on the connections through the message routing network and compute the outgoing messages, as described for a single-Gaussian decoder with degree 3.

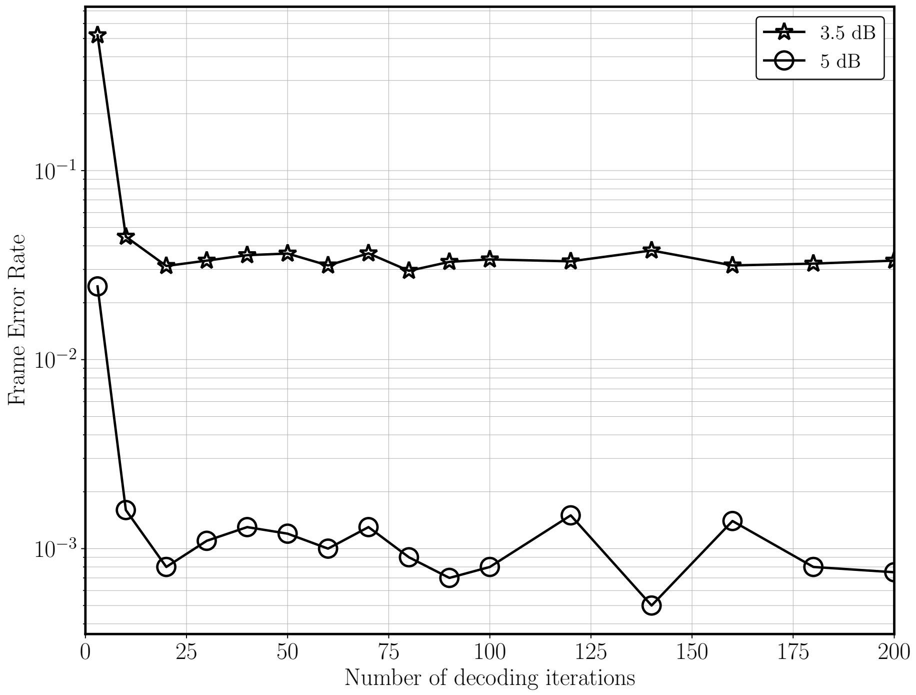


Figure 5.1: Performance of the floating-point single-Gaussian decoder for different number of decoding iterations at distance from capacity of 3.5 dB and 5 dB, $n = 1000$ and $d = 5$.

The check node processing block consists of a check node unit (`CNU`) similar to one used in Chapter 3. However, it receives 5 variable node messages as input and generates 5 corresponding outgoing messages. The timing diagram for the check node message processing block is shown in Fig. 5.5.

Likewise, the primary design component of the variable node message processing block is a variable node unit, similar to the `VNU` block in Chapter 3. However, in this decoder, `VNU` is comprised of 5 incoming and outgoing messages. The outgoing messages at a variable node are computed using a forward-backward recursive algorithm described in Chapter 3.

The variable node unit requires 248 clock cycles to compute the outgoing variable node messages at a variable node while the check node takes 6 clock cycles. In order to attain a decoder architecture with more balanced check node and variable node message computation times, we implement a pipelined architecture for variable node message computation. The `FWBW` computation block requires 236 clock cycles whereas the calculations in the `VOut` computation block take 12 clock cycles. To speed up the variable node message computa-

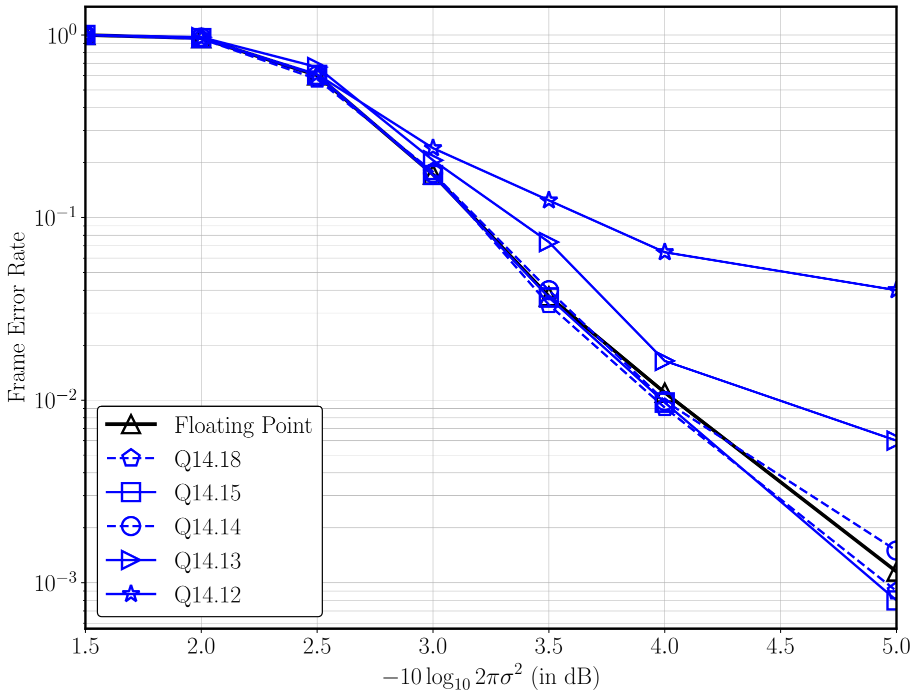


Figure 5.2: Frame error rate comparison between the floating-point decoder (without approximations) and the fixed-point single-Gaussian decoder for different numbers of fractional bits and two NR iterations with $n = 1000$ and $d = 5$.

tion, we use `VNUCluster` blocks with a two-stage pipeline designed in Chapter 3. With the available resources on the available FPGA, we can implement two `VNUCluster` blocks, and thus achieve an effective parallelism equivalent to 20 variable node units (`VNUs`). Fig. 5.6 shows the timing diagram for the variable node message processing block.

The resource requirements of the check node message processing block, variable node message processing block and LDLC decoder are provided in Table 5.1. The decoder architecture attains a throughput of 3.9 Msymbols/sec at a clock frequency of 100 MHz, as observed in Fig. 5.7.

5.1.2 Single-Gaussian Decoder, $d = 7$

It is evident from the simulation results in Fig. 5.8 that similar to the earlier implemented single-Gaussian decoders, 20 decoding iterations is again a suitable choice for the single-

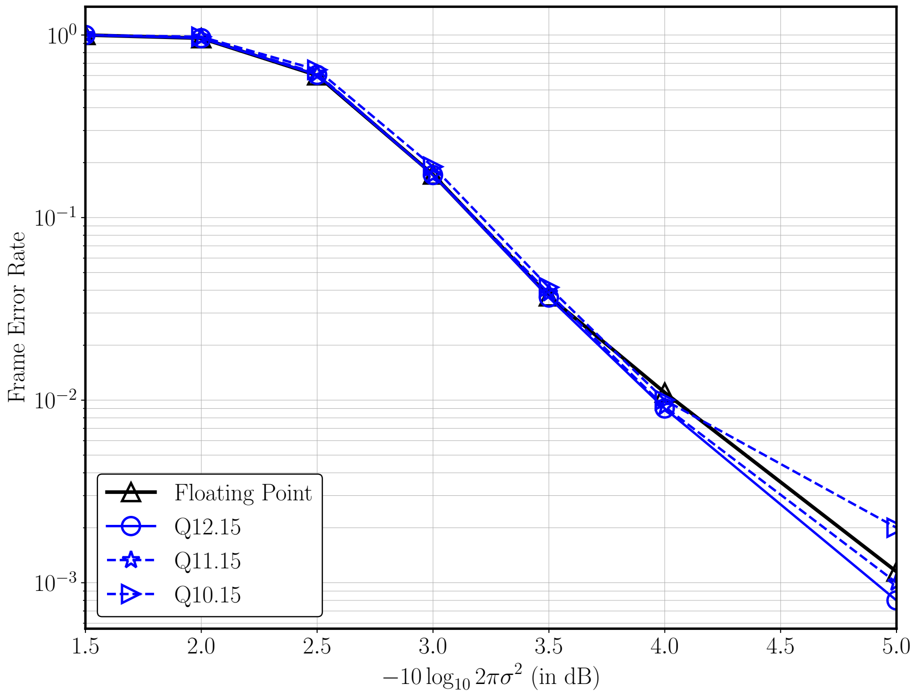
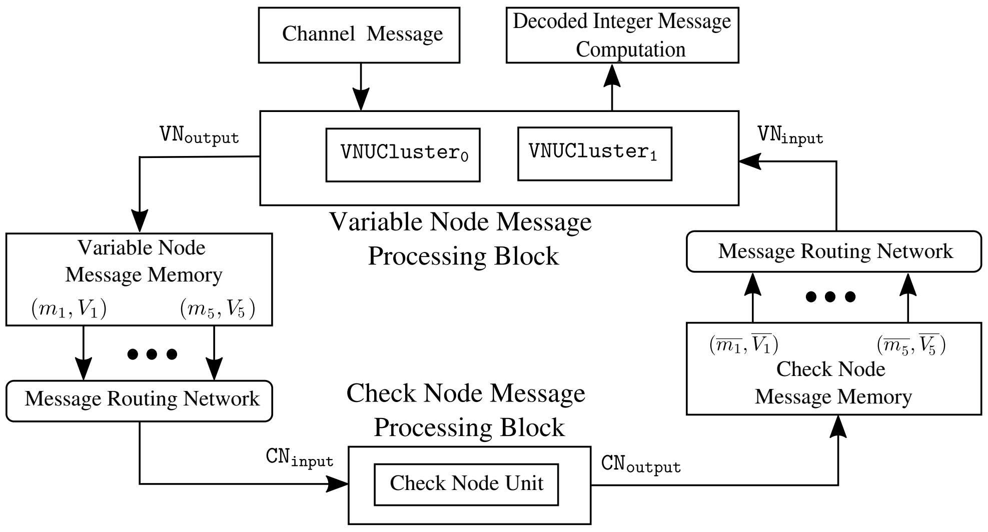


Figure 5.3: Frame error rate of single-Gaussian decoder for different numbers of integer bits and two NR iterations with $n = 1000$, $d = 5$ (floating-point decoder performance is without any approximation).

Gaussian decoder of degree 7.

Fixed-Point Quantization Analysis Fig. 5.9 and Fig. 5.10 show the decoder performance of various values of fractional bits, W_f, and integer bits, W_i, while keeping W_i and W_f fixed respectively. Based on characterization results, a word length of 28, which comprises of 15 fractional bits, 12 integer bits and 1 sign bit is an appropriate choice for the fixed-point representation of this decoder.

FPGA Implementation We implemented a partially parallel architecture presented in Fig. 5.11 for the single-Gaussian LDLC decoder of degree 7. The check node and the variable node units are similar to the ones used in Chapter 3; however, both units use 7 incoming and 7 outgoing messages.



Channel Message
Decoded Integer Message Computation
VNoutput
VNUCluster0
VNUCluster1
VNinput
Variable Node Message Processing Block
Variable Node Message Memory
(m_1, V_1)
(m_5, V_5)
Message Routing Network
Message Routing Network
$(\overline{m_1}, \overline{V_1})$
$(\overline{m_5}, \overline{V_5})$
Check Node Message Memory
Check Node Message Processing Block
CNinput
Check Node Unit
CNoutput

Figure 5.4: Top-level architecture for single-Gaussian LDLC decoder of degree 5, with a single check node and two-stage pipelining to achieve an effective parallelism equivalent to 20 variable nodes.

The check node needs only 9 clock cycles to compute the outgoing messages whereas the variable node unit requires 368 clock cycles. Fig. 5.12 shows the timing diagram for the check node message processing block. In a variable node unit, the `FWBW` computation block requires 352 clock cycles while the calculations in the `VOut` computation block take 13 clock cycles. We implement two `VNUCluster` blocks, and achieve an effective parallelism equivalent to 20 variable node units. Fig. 5.13 shows the timing diagram for the variable node message processing block in this implementation.

The resource usage of the check node message processing block, variable node message processing block and LDLC decoder are provided in Table 5.2. The decoder architecture attains a throughput of 3.1 Msymbols/sec at a clock frequency of 100 MHz as shown in Fig. 5.14.

5.2 Performance and Throughput Comparison of Single-Gaussian Decoders

Fig. 5.15 compares the decoding performance and throughput of the single Gaussian decoder of degree 3, 5 and 7 on the target FPGA device. As seen in the graph the decoding

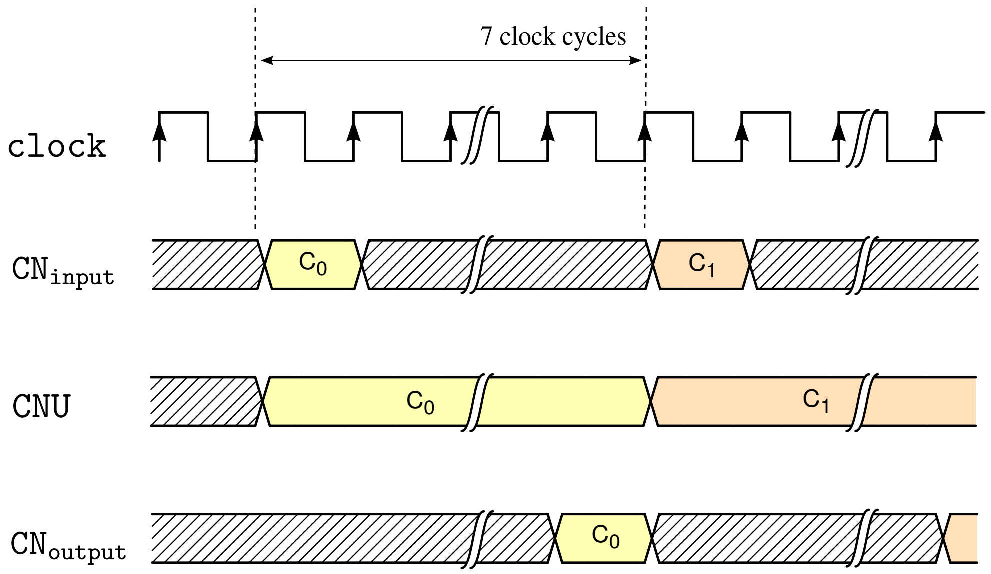


Figure 5.5: Timing diagram for the check node message processing block in single-Gaussian decoder of degree 5.

performance of the single-Gaussian decoder with degree 7 is ~ 0.75 dB better compared to the decoder of degree 3; however, the throughput deteriorates by a factor of $\frac{1}{3.5}$.

After implementation of single-Gaussian decoders and analysis of their decoding performance and design throughput trade-offs, in next section we implement multi-Gaussian decoders for degree 5 and 7 and study the architectural and design trade-offs for the decoders.

5.3 Multi-Gaussian Decoders

5.3.1 Multi-Gaussian Decoder, $d = 5$ and $M = 2$

Similar to the multi-Gaussian decoder of degree 3 presented in Chapter 4, 30 decoding iterations are suitable for this decoder as shown in Fig. 5.16. We characterize the decoder performance for different number of W_f (W_i) while keeping $W_i(W_f)$ constant. Based on the results shown in Figs. 5.17 and 5.18, a word length of 28, with 12 integer bits, 15 fractional bits and a sign bit is appropriate for the fixed-point representation of the decoder.

Table 5.1: Resource usage of single-Gaussian decoder ($M = 1$), $d = 5$.

Resource	Check Node	Variable Node	Decoder
ALMs	5638	322093	400044
Registers	2131	197335	241751
BRAM(M20K)	0	28	54
Variable Precision DSPs	0	920	920

Table 5.2: Resource usage of single-Gaussian decoder ($M = 1$), $d = 7$.

Resource	Check Node	Variable Node	Decoder
ALMs	2975	301407	390181
Registers	3691	214684	261635
BRAM(M20K)	0	40	78
Variable Precision DSPs	31	1487	1518

FPGA Implementation For the multi-Gaussian LDLC decoder of degree 5, we implemented a partially parallel architecture presented in Fig. 5.19. The check node and the variable node units in this architecture are similar to the design blocks used in Chapter 4; however, they each receive 5 messages and compute 5 outgoing messages.

In this implementation, the check node message processing block contains a single check node unit and variable node message processing block consists of a two-stage pipeline similar to the `MNUCluster` block of Chapter 4. However, for the `MNUCluster` block used in this decoder, the first stage consists of two parallel `VNFWBW` computation blocks and the second stage is a `VNVOut` computation block, which reads in the output of first pipeline stage and generates the outgoing message.

Fig. 5.20 and Fig. 5.21 show the timing diagram for the check node and variable node message processing block for this architecture. The resources required for the check node, variable node message processing blocks and LDLC decoder are provided in Table 5.3. The decoder architecture attains a throughput of 145 Ksymbols/sec at a clock frequency of 105 MHz, as shown in Fig. 5.22.

Table 5.3: Resource usage of multi-Gaussian decoder, $d = 5$ and $M = 2$.

Resource	Check Node	Variable Node	Decoder
ALMs	45262	222580	335870
Registers	21836	110982	173253
BRAM(M20K)	0	0	865
Variable Precision DSPs	150	1358	1518

5.3.2 Multi-Gaussian Decoder, $d = 7$ and $M = 2$

Fig. 5.23 plots the decoder performance for different number of decoding iterations at a distance from capacity of 2.5 dB and 4 dB. As seen in the graph, with 30 decoding iterations the decoder achieves a performance comparable to 200 decoding iterations.

We characterized the multi-Gaussian decoder performance to obtain the optimal word length for the fixed-point implementation of the decoder. Based on the results, shown in Figs. 5.24 and 5.25, a word length of 28 with 12 integer bits 15 fractional bits and a sign bit is suitable for the fixed-point representation of the decoder.

FPGA Implementation In this section, we present a serial LDLC decoder architecture with a single check node and a single variable node designed on a target FPGA (shown in Fig. 5.26). The check node and variable node units are similar to the blocks used in Chapter 4, with 7 incoming messages and 7 out going messages corresponding to degree 7.

The variable node unit requires 1015 clock cycles for outgoing message computation while the check node takes 197 clock cycles. The available hardware resources on the target FPGA are sufficient to implement a serial architecture with a single check node and a single variable node.

The timing diagrams for the check node and variable node message processing block for this architecture are shown in Fig. 5.27 and Fig. 5.28. The hardware resources required for the check node, variable node message processing blocks and LDLC decoder implementation are provided in Table 5.4. This serial decoder architecture achieves a throughput of 84 Ksymbols/sec at a distance of 5 dB from capacity for a clock frequency of 100 MHz as shown in Fig. 5.29.

So far in thesis, for the implemented multi-Gaussian decoders, the Gaussian mixture messages exchanged during the iterative decoding comprised of only two components. How-

Table 5.4: Resource usage of multi-Gaussian decoder, $d = 7$ and $M = 2$.

Resource	Check Node	Variable Node	Decoder
ALMs	63505	279008	352443
Registers	22401	81236	109580
BRAM(M20K)	10	80	140
Variable Precision DSPs	139	1379	1518

ever, it is compelling to characterize the decoding performance of the multi-Gaussian decoders where the Gaussian mixture messages contain more than two components in the mixture. To understand the effect of larger Gaussian mixtures on the decoding performance, in Subsection 5.3.3 to Subsection 5.3.5 we present multi-Gaussian fixed-point LDLC decoders for degree 3, 5, and 7 with M=5.

5.3.3 Multi-Gaussian Decoder, $d = 3$ and $M = 5$

Simulation results demonstrated (Fig. 5.30) that for this multi-Gaussian decoder 30 decoding iterations are sufficient to attain a frame error rate comparable to 200 decoding iterations.

Based on the quantization results, shown in Fig. 5.31 and Fig. 5.32, a word length of 28 with 12 integer bits 15 fractional bits and a sign bit is appropriate for the fixed-point representation of the decoder.

We designed a serial implementation of the multi-Gaussian decoder with a single check node and a single variable node for degree d=3 and M=5. However, due to large resource requirements the design does not fit on the target device. As the DSP requirement of the implementation is large, it does not fit on a Stratix GX FPGA either.

5.3.4 Multi-Gaussian Decoder, $d = 5$ and $M = 5$

Based on the results shown in Fig. 5.33 we choose 30 decoding iterations for the multi-Gaussian decoder implementation of d=5 and M=5. Further, the quantization simulation results presented in Fig. 5.34 and Fig. 5.35 suggest that a word length of 28 with 12 integer bits 15 fractional bits and a sign bit is suitable for the fixed-point decoder implementation.

Similar to the multi-Gaussian decoder with degree 3 and M=5, we designed a serial implementation of this multi-Gaussian decoder, but it does not fit on the target Intel FPGA.

5.3.5 Multi-Gaussian Decoder, $d = 7$ and $M = 5$

As shown in Fig. 5.36 30 decoding iterations are suitable choice for the multi-Gaussian decoder implementation of degree d=7 and M=5. According to the quantization study (shown in Fig. 5.37 and Fig. 5.38) a word length of 28 with 12 integer bits 15 fractional bits and a sign bit is appropriate for the fixed-point decoder implementation.

From hardware implementation perspective, this multi-Gaussian decoder does not fit on the target FPGA. However, a larger FPGA device that contains more hardware resources can fit the decoder implementation.

5.4 Performance and Throughput Comparisons

A comparison of decoding performance and throughput of the multi-Gaussian decoders of degree 3, 5 and 7 (for M =2) is shown in Fig. 5.39. It is evident from the results that the multi-Gaussian decoder of degree 7 attains a performance improvement of $\sim$ 1.75 dB compared to the decoder of degree 3. Nonetheless, throughput of the multi-Gaussian decoder with degree 7 is $\sim$ 15% compared to that of the decoder with degree 3.

Fig. 5.40 compares the performance of the multi-Gaussian decoder of degrees 3, 5 and 7 where Gaussian mixture messages consist of 5 components, i.e., M=5. As seen in the graphs, there is a minimal improvement of $\sim$ 0.1 dB for degrees 3 and 7 while we get achieve a boost of $\sim$ 0.3 dB in frame error rate for degree 5.

5.4.1 Single-Gaussian vs. Multi-Gaussian Decoders

In the previous sections, we implemented single-Gaussian and multi-Gaussian decoders of degrees 3, 5 and 7 on an Intel Arria 10 FPGA. We characterized the frame error rate performance and obtained design throughputs for various architectures of the decoders. Fig. 5.41 presents a comparison of the frame error rates and throughputs of the single-Gaussian and multi-Gaussian decoders of degree 3, 5 and 7. As seen in the graphs, for high throughput requirements, e.g., 10^7 symbols/sec, a single-Gaussian decoder of degree

Table 5.5: Performance and Throughput Comparison of LDLC Decoders.

Degree	Number of Components (M)	gain(dB)	Throughput (Ksymbols/sec.)
3	1	0 (reference)	$10,500$
	2	~ 0.75	550
	5	~ 0.85 (simulation)	doesn't fit on FPGA
5	1	~ 0.5	$3,800$
	2	~ 2.25	145
	5	~ 2.55 (simulation)	doesn't fit on FPGA
7	1	~ 0.7	$3,080$
	2	~ 2.5	84
	5	~ 2.6 (simulation)	doesn't fit on FPGA

3 is a suitable choice while for ultra-reliable communication applications a multi-Gaussian decoder of degree 7 is appropriate.

Table 5.5 provides a summary of the fixed-point LDLC decoder implementations developed in this thesis in terms of decoding performance and design throughput. For this comparison the first known hardware implementation of an LDLC decoder (so far to us), i.e., the *single Gaussian decoder* presented in Chapter 3, is considered as a reference.

5.5 Summary

In this chapter, we first studied the decoding performance of the single-Gaussian and multi-Gaussian decoders ($M = 2$) for degrees 5 and 7 and implemented different decoder architectures on the target hardware.

Then, we characterized the frame error rate performance of the multi-Gaussian decoders where Gaussian mixture messages consist of more than two components. This chapter presented a detailed study of different LDLC decoder implementations, decoding performance and design throughput trade-offs. This study is particularly important to choose an appropriate decoder for an application based on decoding accuracy and design throughput requirements.

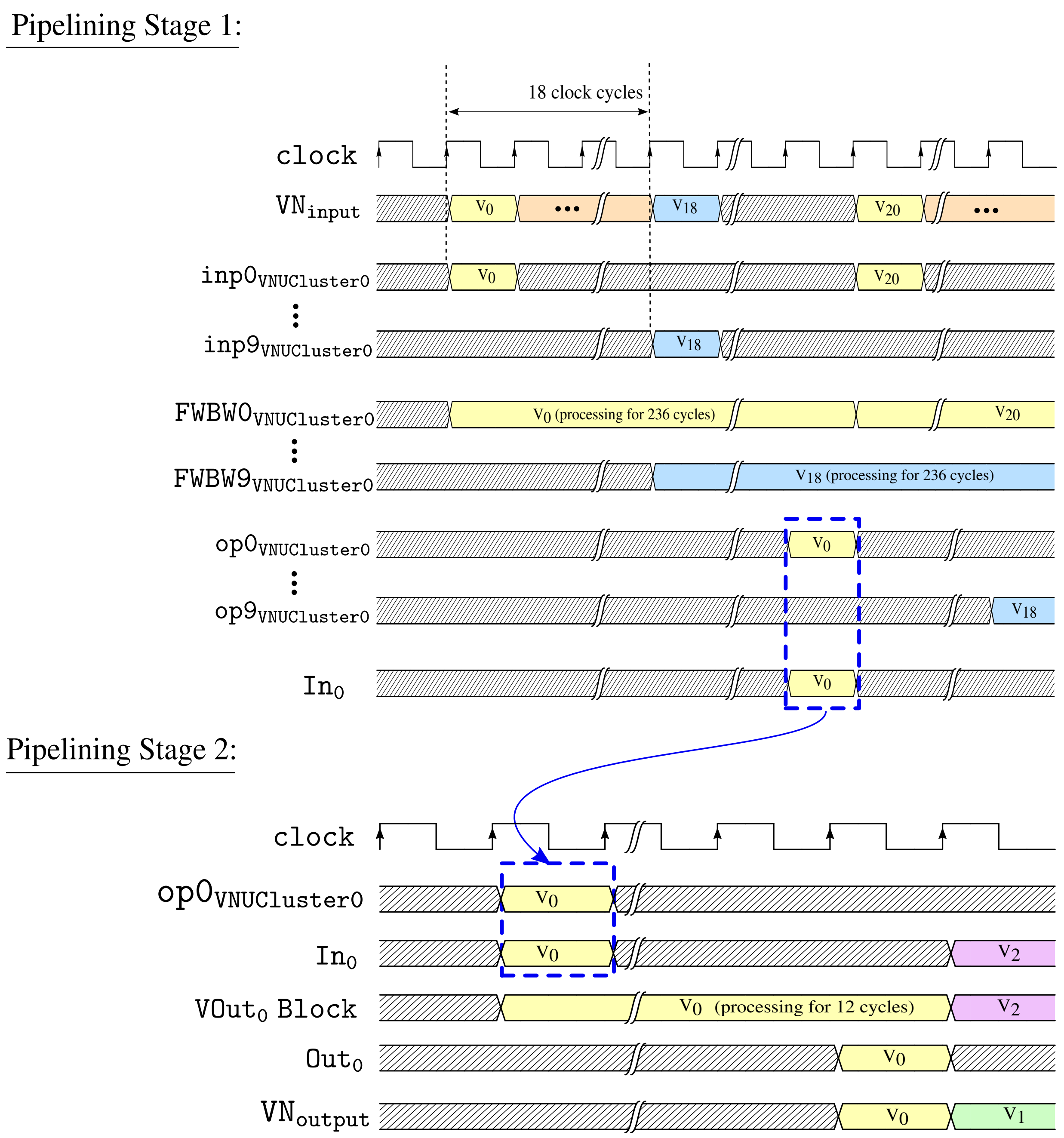


Figure 5.6: Timing diagram for the variable node message processing block in single-Gaussian decoder of degree 5. The waveforms are shown specifically for $\texttt{VNUCluster}_0$ block.

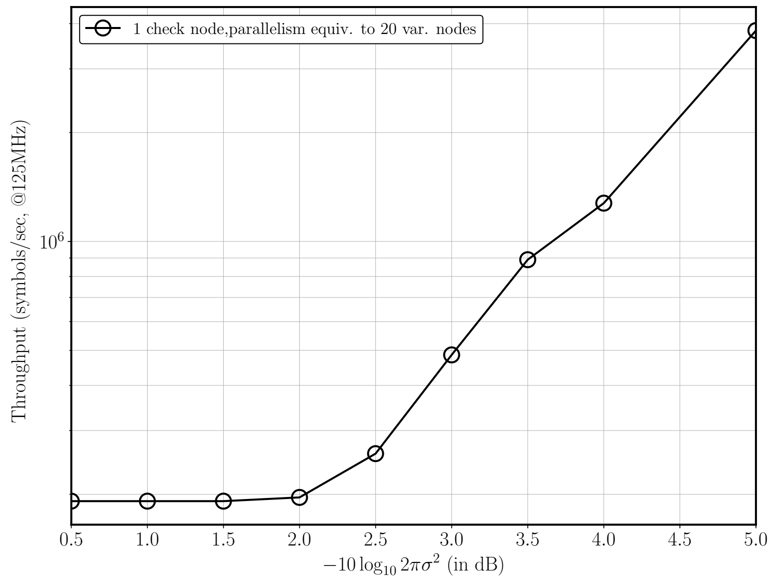


Figure 5.7: Throughput of the single-Gaussian decoder of degree 5 for block length 1000.

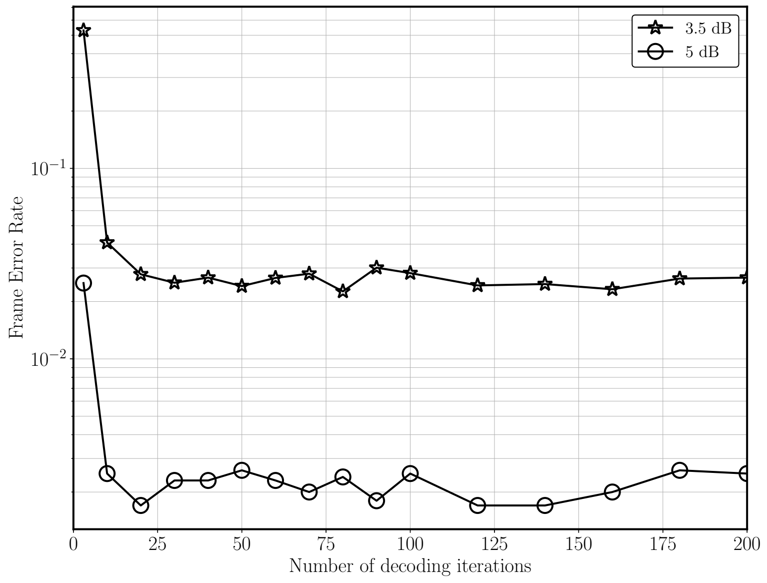


Figure 5.8: Performance of the single-Gaussian decoder (floating-point) versus number of decoding iterations at distance from capacity of 3.5 dB and 5 dB for $n = 1000$ and $d = 7$.

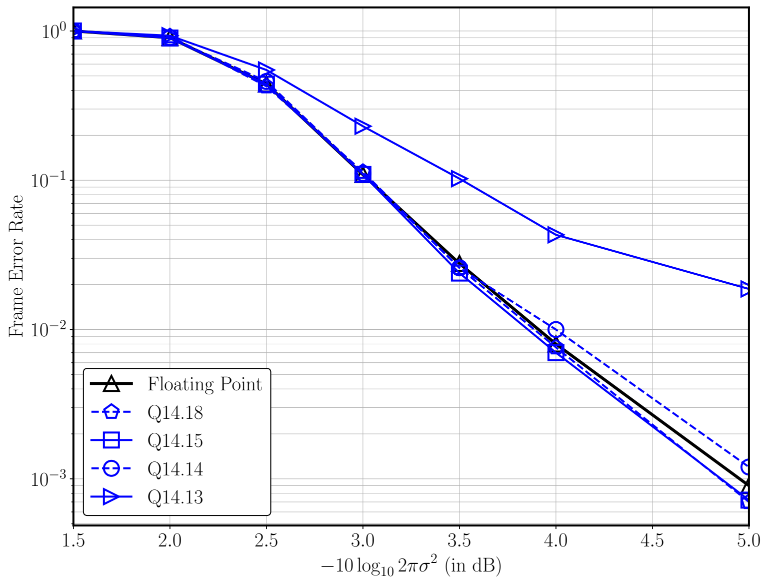


Figure 5.9: FER comparison of floating-point (without approximations) and fixed-point single-Gaussian decoder for different W_f and two NR iterations with $n = 1000$ and $d = 7$.

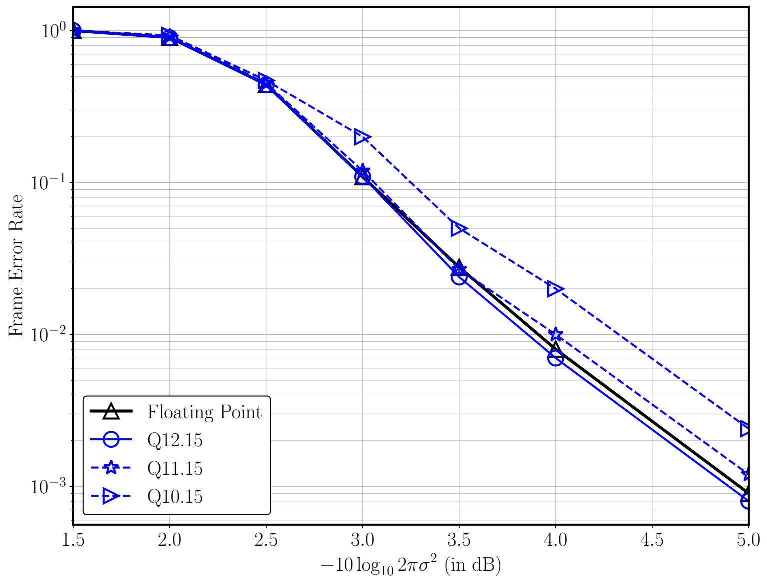
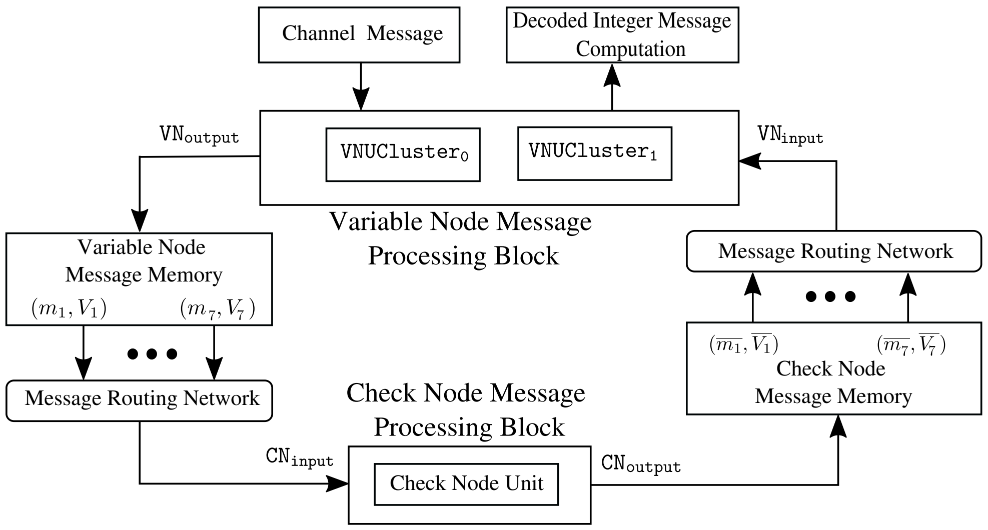


Figure 5.10: FER of single-Gaussian decoder for different W_i and two NR iterations with $n = 1000$, $d = 7$ (floating-point decoder is without any approximation).



Channel Message
Decoded Integer Message Computation
VN$_{\texttt{output}}$
VNUCluster$_0$
VNUCluster$_1$
VN$_{\texttt{input}}$
Variable Node Message Processing Block
Variable Node Message Memory
(m_1, V_1)
(m_7, V_7)
Message Routing Network
Message Routing Network
$(\overline{m_1}, \overline{V_1})$
$(\overline{m_7}, \overline{V_7})$
Check Node Message Memory
Check Node Message Processing Block
CN$_{\texttt{input}}$
Check Node Unit
CN$_{\texttt{output}}$

Figure 5.11: Top-level architecture for the fixed-point single-Gaussian LDLC decoder of degree 7, with a single check node and two-stage pipelining to achieve an effective parallelism equivalent to 20 variable nodes.

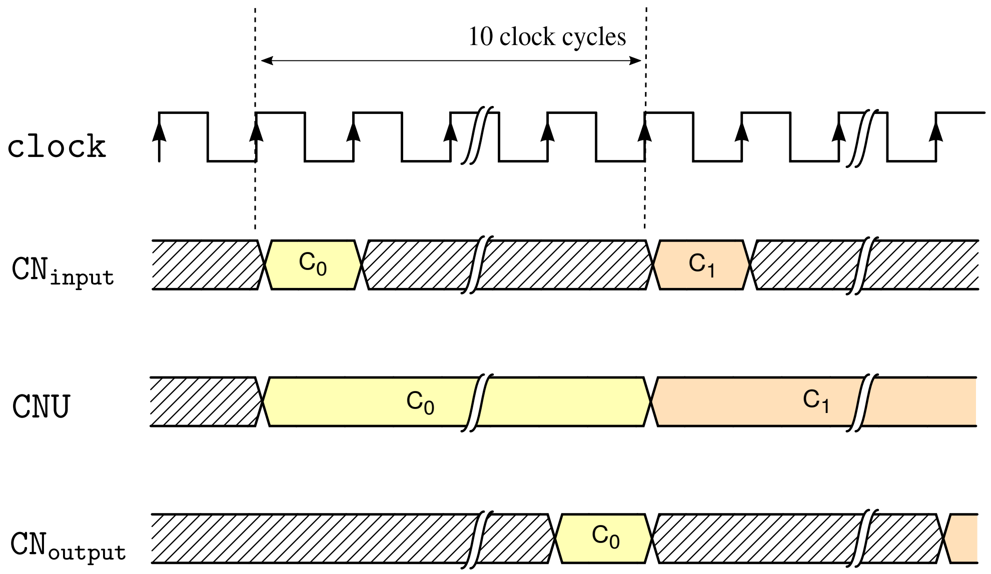


Figure 5.12: Timing diagram for the check node message processing block in single-Gaussian decoder of degree 7.

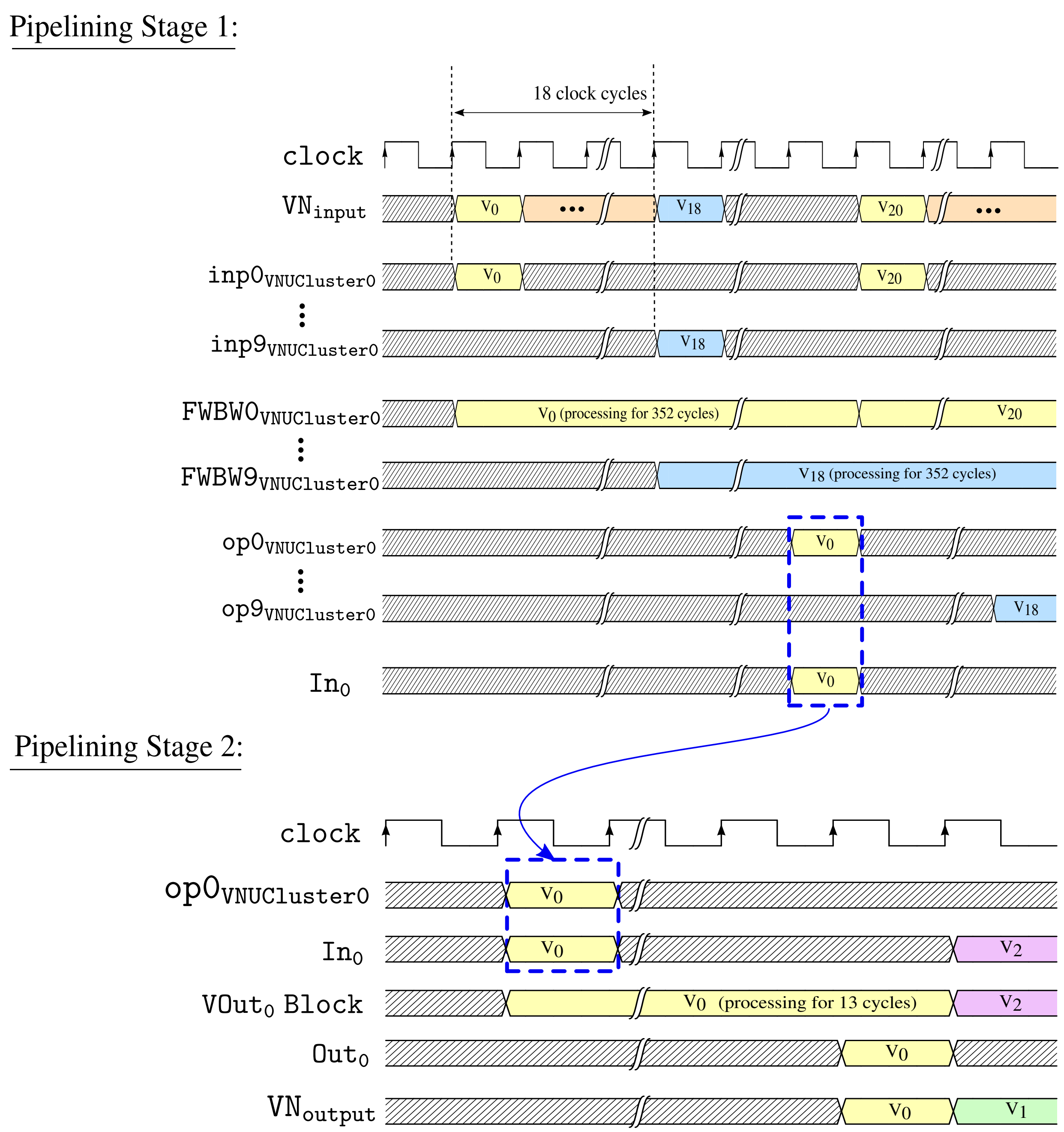


Figure 5.13: Timing diagram for the variable node message processing block in single-Gaussian decoder of degree 7. The waveforms are shown specifically for $\texttt{VNUCluster}_0$ block.

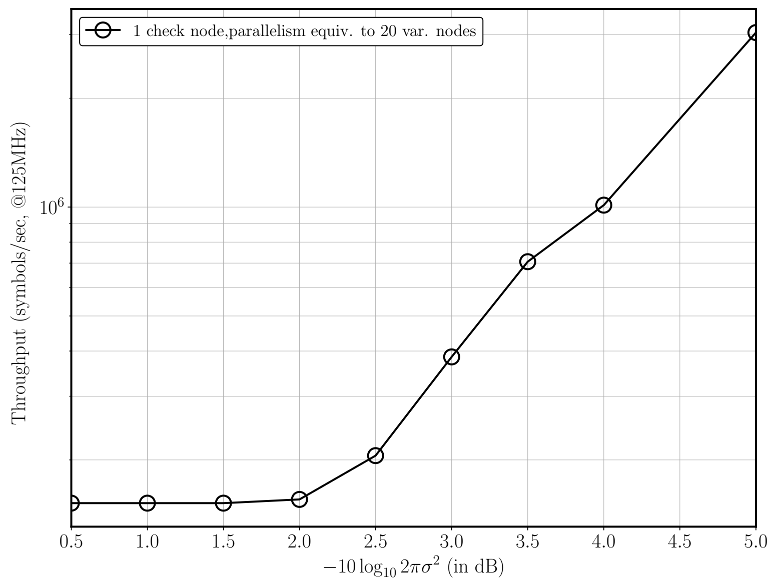


Figure 5.14: Throughput of fixed-point single-Gaussian decoder with $n = 1000$ and $d = 7$.

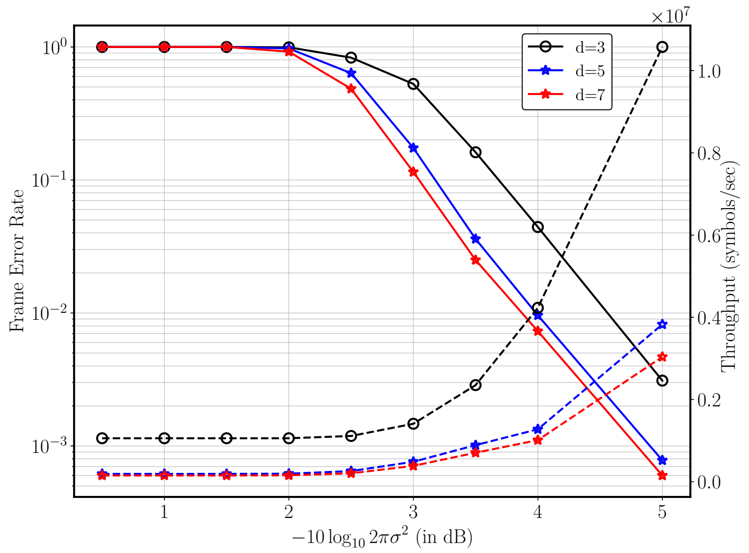


Figure 5.15: FER and throughput comparison for the fixed-point single-Gaussian decoders of degree 3, 5 and 7 with block length 1000.

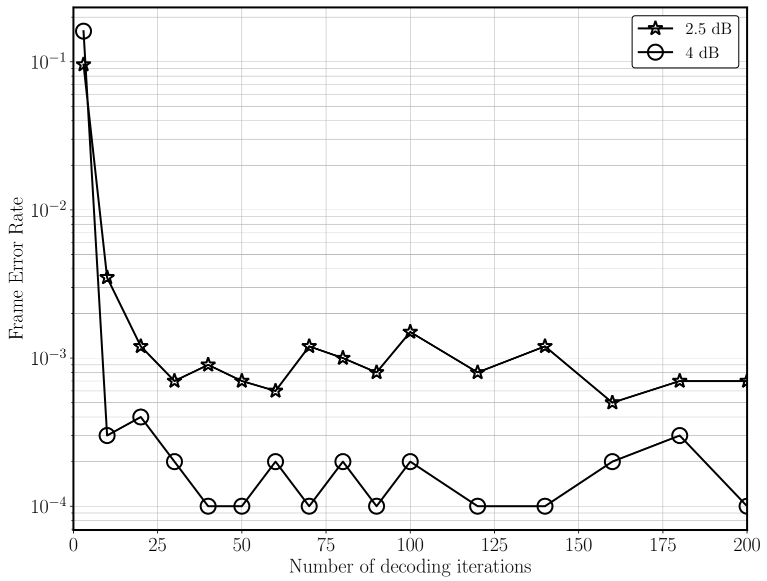


Figure 5.16: FER of the floating-point multi-Gaussian decoder at distance from capacity of 2.5 dB and 4 dB for different numbers of decoding iterations with $n = 1000$ and $d = 5$.

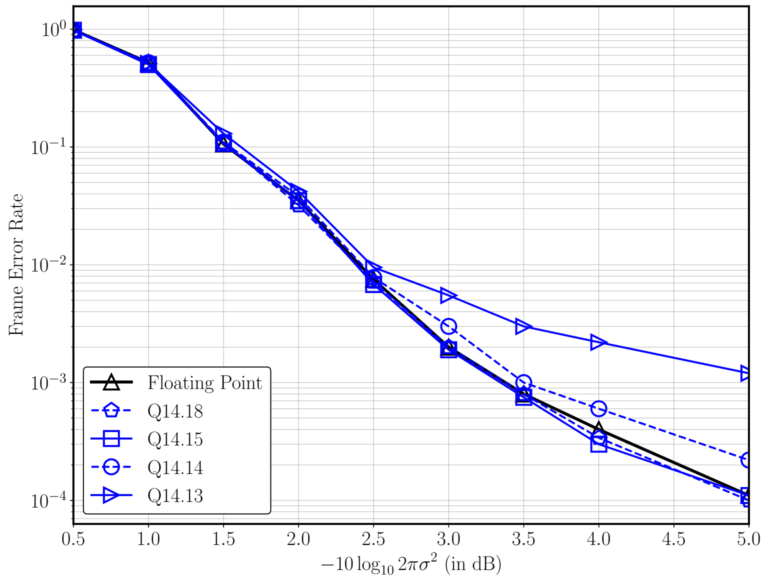


Figure 5.17: FER of the floating-point (without approximations) and fixed-point multi-Gaussian decoder for different W_f and two NR iterations with $n = 1000$ and $d = 5$.

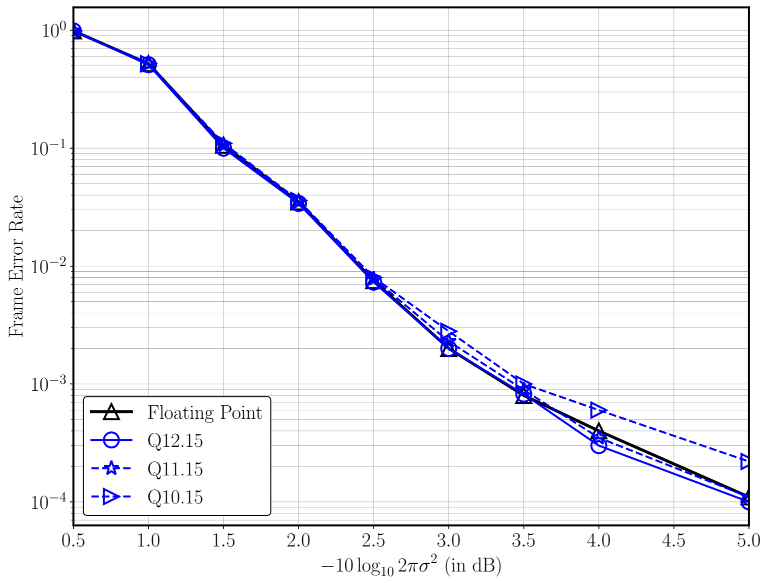
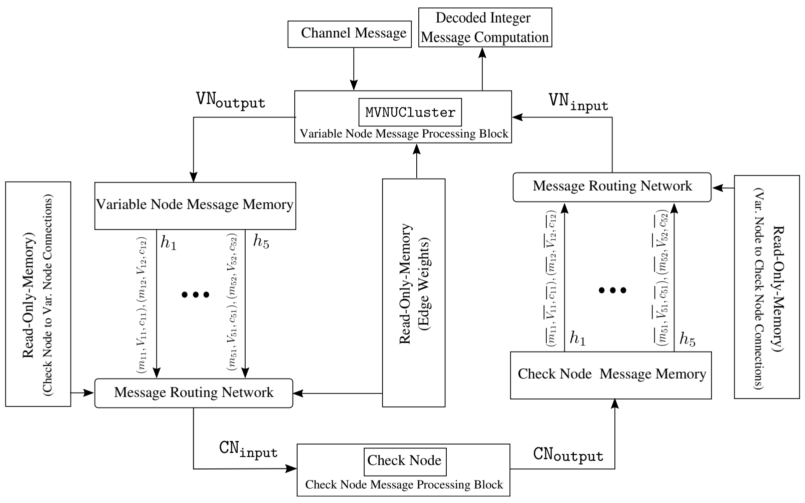


Figure 5.18: FER performance of the multi-Gaussian decoder with different W_i and two NR iterations for $n = 1000$ and $d = 5$ (floating-point decoder performance is without any approximation).



Channel Message
Decoded Integer Message Computation
$\texttt{VN}_{\texttt{output}}$
MVNUCluster
Variable Node Message Processing Block
$\texttt{VN}_{\texttt{input}}$
Read-Only-Memory (Check Node to Var. Node Connections)
Variable Node Message Memory
h_1
h_5
$(m_{11}, V_{11}, c_{11}), (m_{12}, V_{12}, c_{12})$
$(m_{51}, V_{51}, c_{51}), (m_{52}, V_{52}, c_{52})$
Read-Only-Memory (Edge Weights)
Message Routing Network
$(\overline{m_{11}}, \overline{V_{11}}, \overline{c_{11}}), (\overline{m_{12}}, \overline{V_{12}}, \overline{c_{12}})$
$(\overline{m_{51}}, \overline{V_{51}}, \overline{c_{51}}), (\overline{m_{52}}, \overline{V_{52}}, \overline{c_{52}})$
Check Node Message Memory
Read-Only-Memory (Var. Node to Check Node Connections)
Message Routing Network
$\texttt{CN}_{\texttt{input}}$
Check Node
Check Node Message Processing Block
$\texttt{CN}_{\texttt{output}}$

Figure 5.19: Top-level architecture of the fixed-point multi-Gaussian LDLC decoder (degree, 5) with a single check node and with two-stage pipelining to achieve an effective parallelism equivalent to 2 variable nodes.

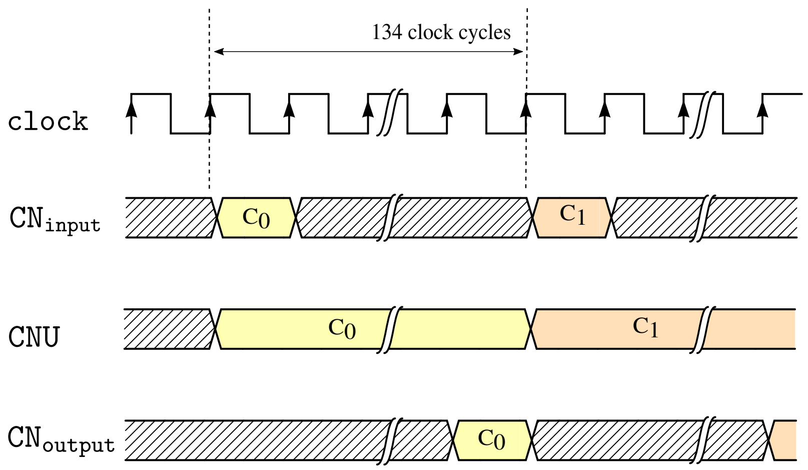


Figure 5.20: Timing diagram of the check node message processing block in multi-Gaussian decoder (degree 5).

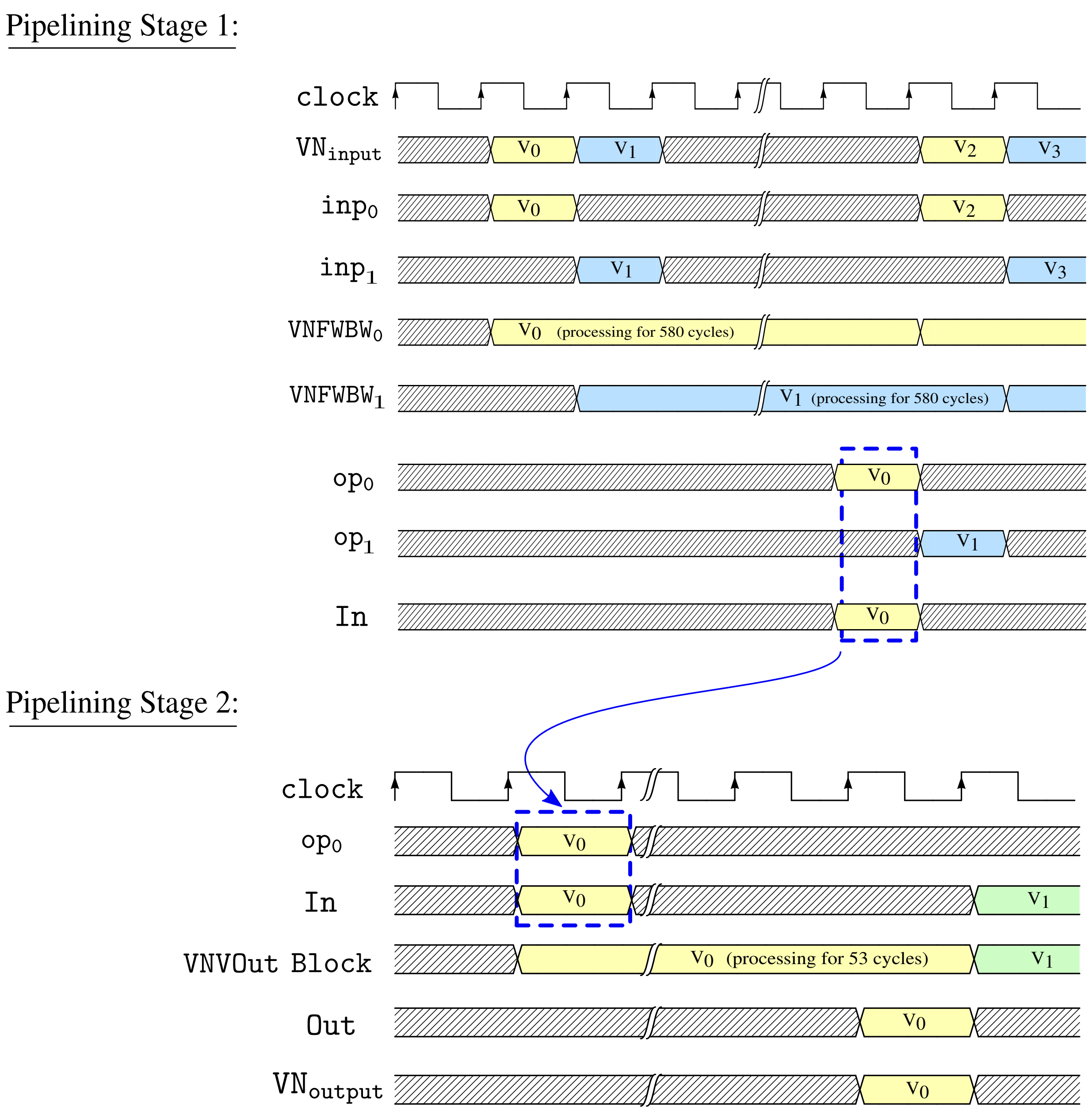


Figure 5.21: Timing diagram of the variable node message processing block for multi-Gaussian decoder (degree 5).

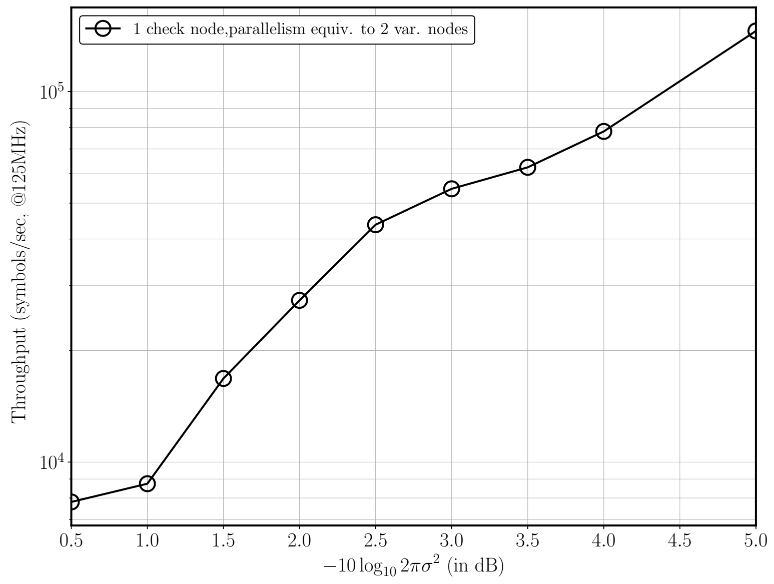


Figure 5.22: Throughput of the fixed-point multi-Gaussian decoder (M=2) with degree 5 and block length 1000.

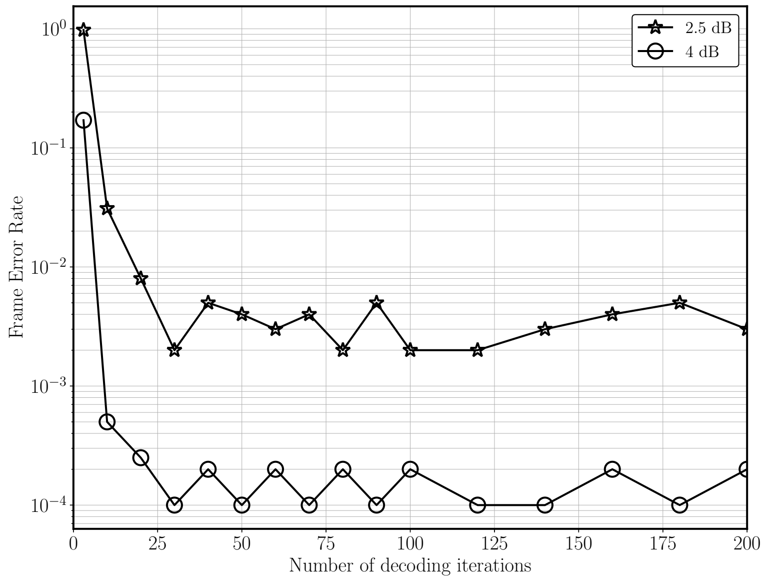


Figure 5.23: Performance of floating-point multi-Gaussian decoder (d=7, n=1000, M=2) for different number of decoding iterations at distance from capacity of 2.5 dB and 4 dB.

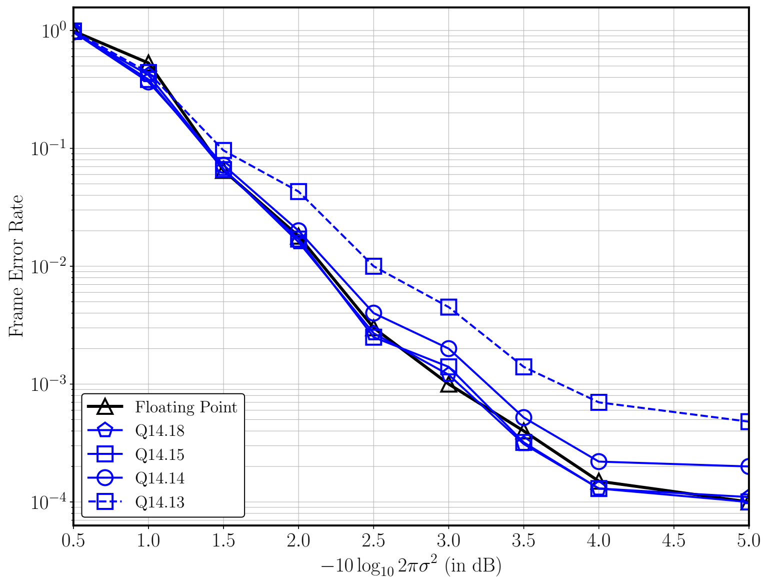


Figure 5.24: FER of the floating-point (without approximations) and the fixed-point multi-Gaussian decoder for different W_f and two NR iterations for $d = 7$ and $n = 1000$.

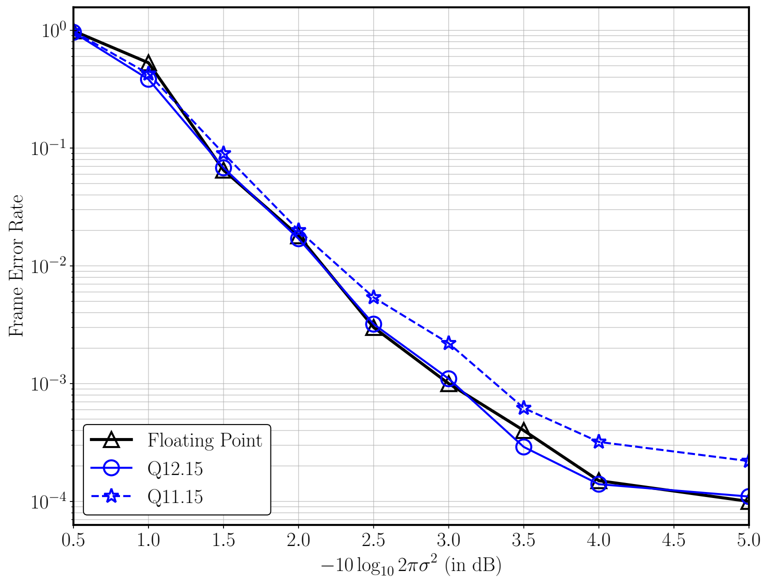


Figure 5.25: FER of the multi-Gaussian decoder for different W_i and two NR iterations with degree 7 and $n = 1000$ (floating-point decoder is without approximation).

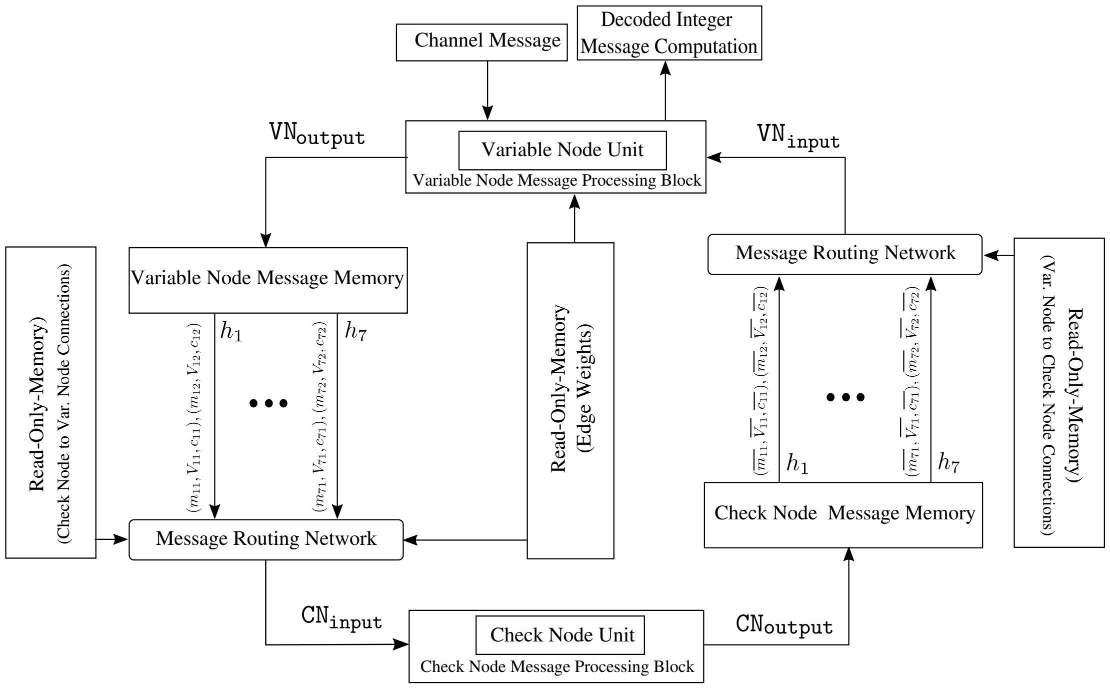

Channel Message
Decoded Integer Message Computation
$\mathtt{VN_{output}}$
Variable Node Unit
Variable Node Message Processing Block
$\mathtt{VN_{input}}$
Read-Only-Memory) (Check Node to Var. Node Connections)
Variable Node Message Memory
h_1
h_7
$(m_{11}, V_{11}, c_{11}), (m_{12}, V_{12}, c_{12})$
$(m_{71}, V_{71}, c_{71}), (m_{72}, V_{72}, c_{72})$
Message Routing Network
Read-Only-Memory (Edge Weights)
Message Routing Network
$(\overline{m_{11}}, \overline{V_{11}}, \overline{c_{11}}), (\overline{m_{12}}, \overline{V_{12}}, \overline{c_{12}})$
$(\overline{m_{71}}, \overline{V_{71}}, \overline{c_{71}}), (\overline{m_{72}}, \overline{V_{72}}, \overline{c_{72}})$
Check Node Message Memory
Read-Only-Memory) (Var. Node to Check Node Connections)
$\mathtt{CN_{input}}$
Check Node Unit
Check Node Message Processing Block
$\mathtt{CN_{output}}$

Figure 5.26: Block diagram of a two-node serial fixed-point multi-Gaussian LDLC decoder of degree 7 (one single check node and one single variable node).

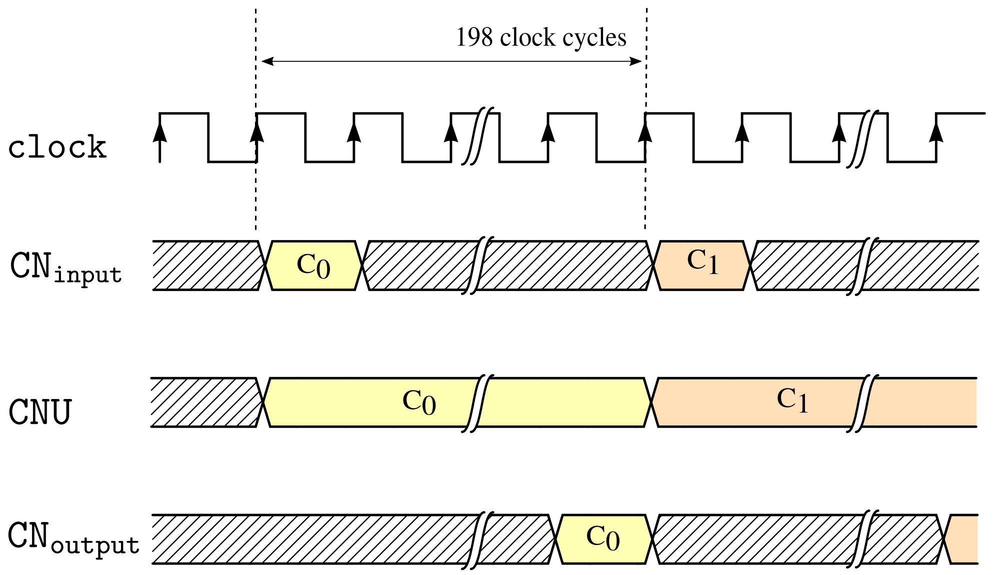


Figure 5.27: Timing diagram of the check node message processing block in multi-Gaussian decoder of degree 7.

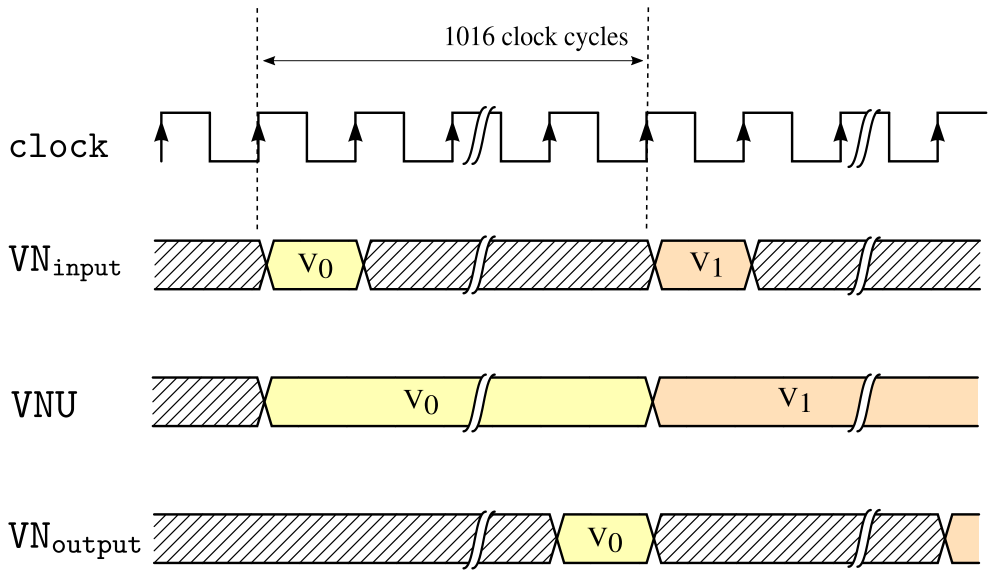


Figure 5.28: Timing diagram of the variable node message processing block for multi-Gaussian decoder of degree 7.

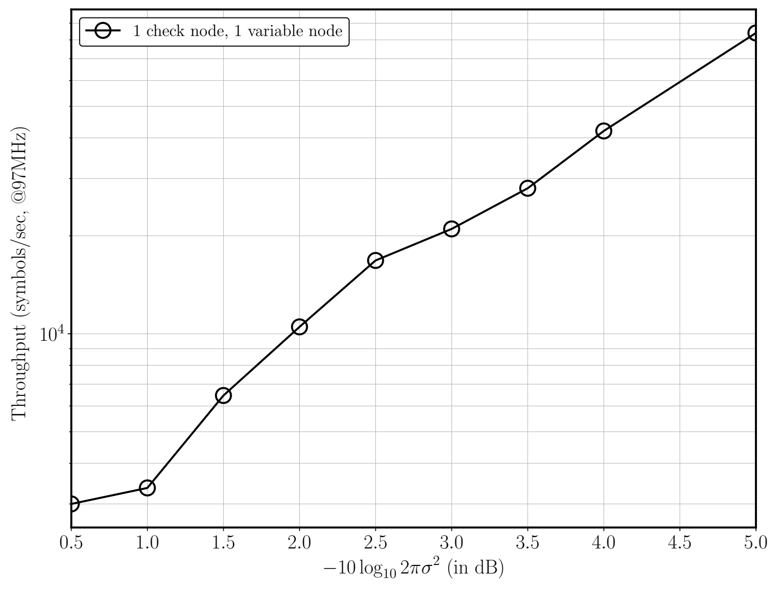


Figure 5.29: Throughput of the fixed-point multi-Gaussian decoder of degree 7 for block length 1000.

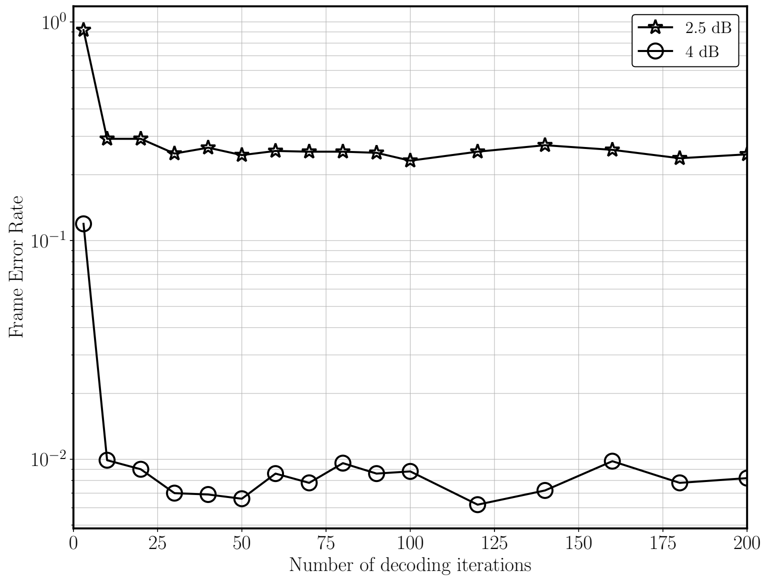


Figure 5.30: Performance of the multi-Gaussian decoder for different number of decoding iterations at distance from capacity of 2.5 dB and 4 dB with $M = 5$, $d = 3$ and $n = 1000$.

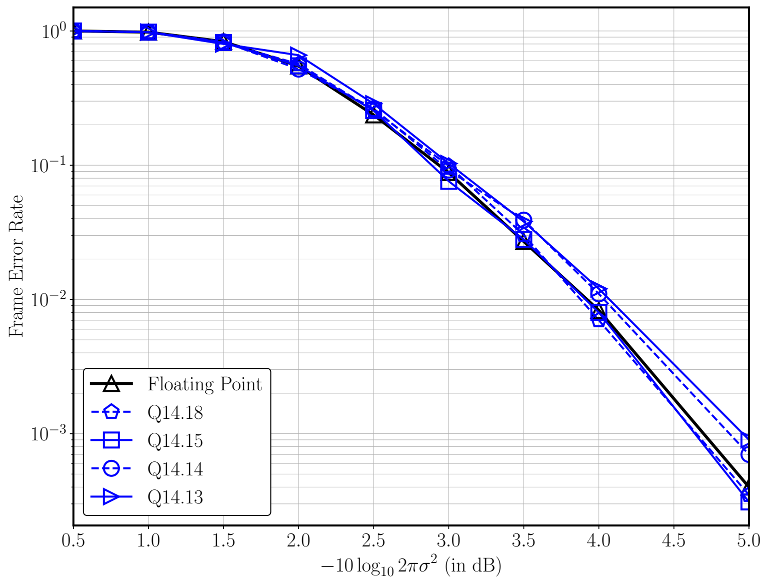


Figure 5.31: FER of the floating-point (without approximations) and the fixed-point multi-Gaussian decoder (M=5) for different W_f and two NR iterations with $d = 3$ and $n = 1000$.

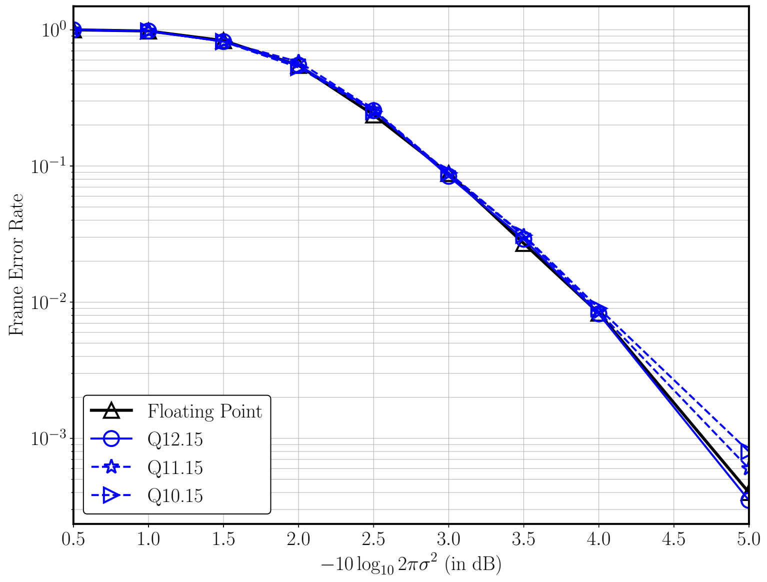


Figure 5.32: FER of multi-Gaussian decoder (M=5) for different W_i and two NR iterations with $d = 3$, $n = 1000$ (floating-point performance is without approximations).

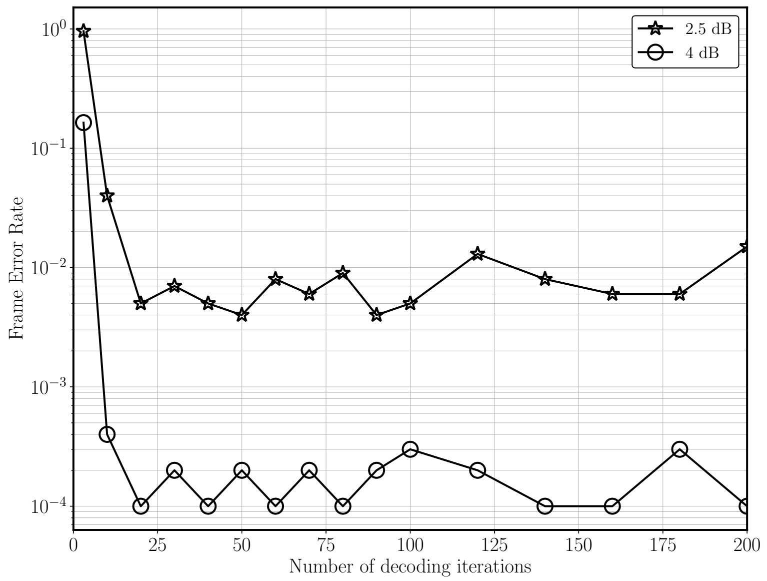


Figure 5.33: Performance of the multi-Gaussian decoder for different number of decoding iterations at distance from capacity of 2.5 dB and 4 dB with $M = 5$, $d = 5$ and $n = 1000$.

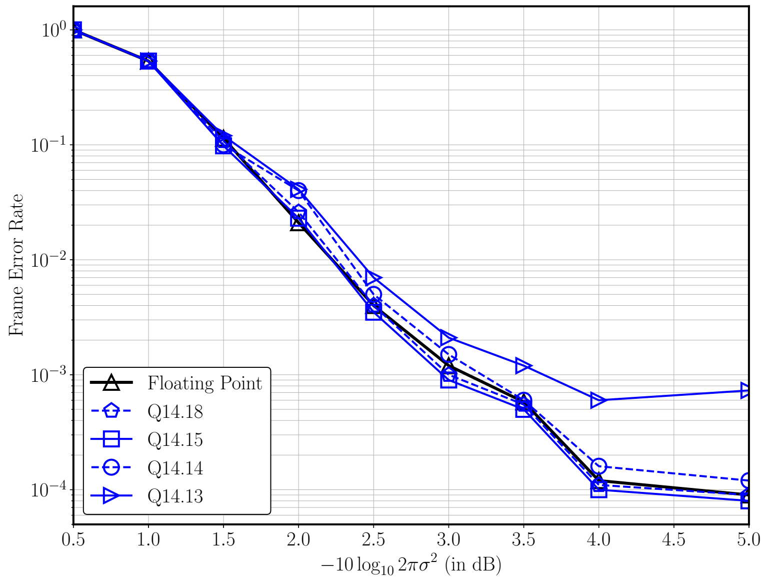


Figure 5.34: FER of the floating-point (without approximations) and fixed-point multi-Gaussian decoder (M=5) for different W_f and two NR iterations for $n = 1000$, $d = 5$.

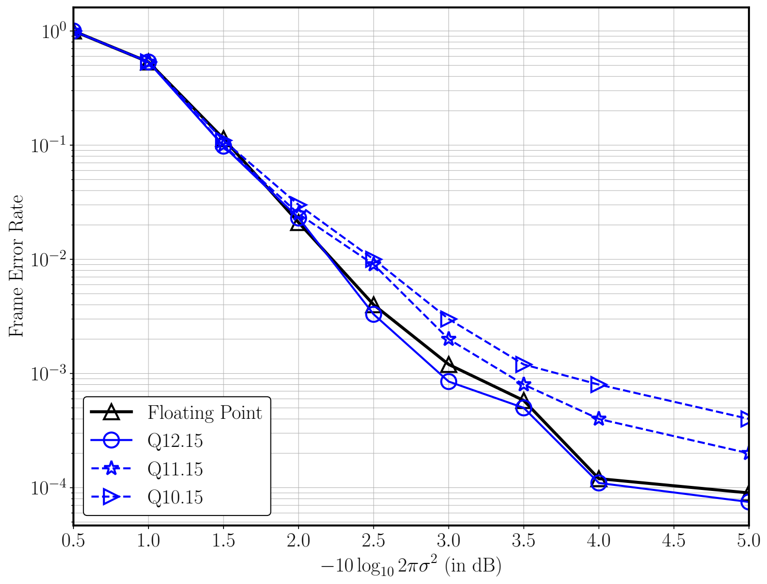


Figure 5.35: FER of the multi-Gaussian decoder (M=5) for different W_i and two NR iterations with $n = 1000$ and $d = 5$ (floating-point performance is without approximation).

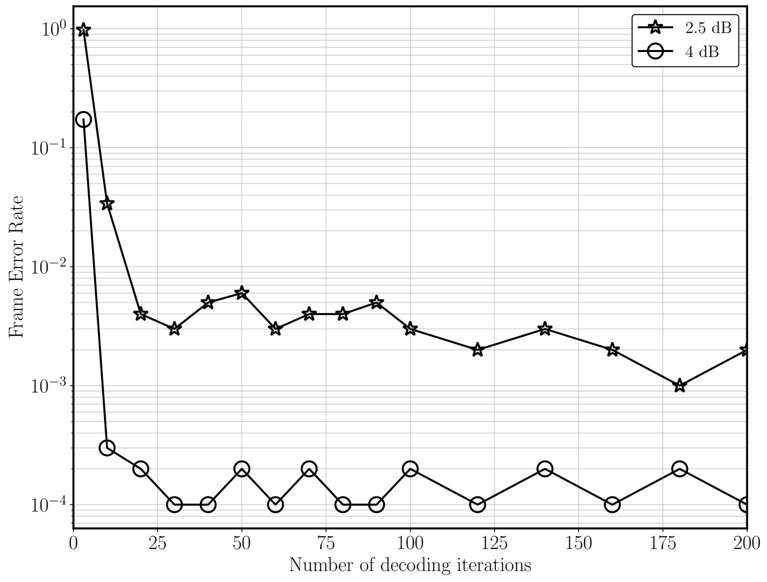


Figure 5.36: Performance of the multi-Gaussian decoder for different number of decoding iterations at distance from capacity of 2.5 dB and 4 dB with $M = 5$, $d = 7$ and $n = 1000$.

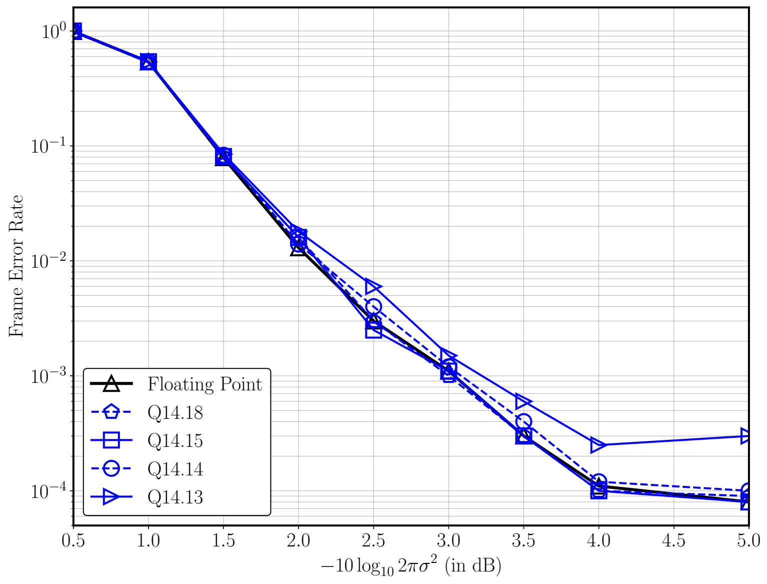


Figure 5.37: FER of the floating-point (without approximations) and the fixed-point multi-Gaussian decoder (M=5) for different W_f and two NR iterations for $d = 7$, $n = 1000$.

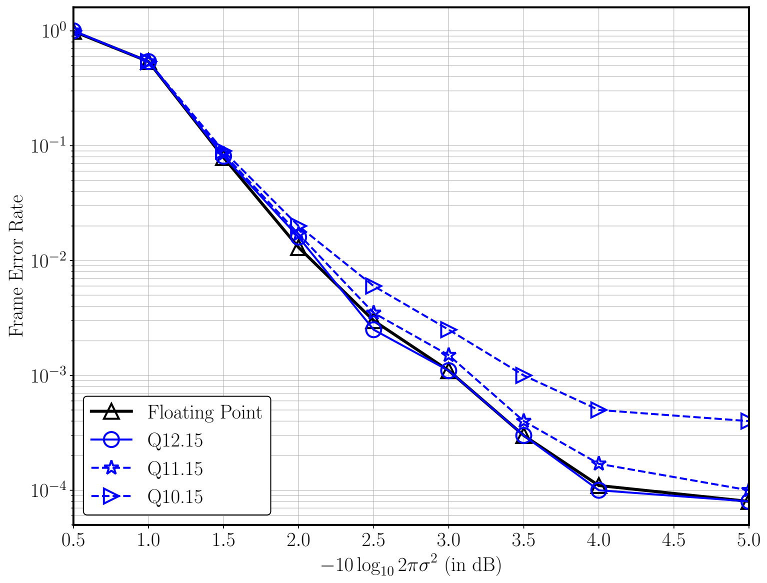


Figure 5.38: FER of the multi-Gaussian decoder (M=5) for different W_i and two NR iterations with $d = 7$, $n = 1000$ (floating-point performance is without approximation).

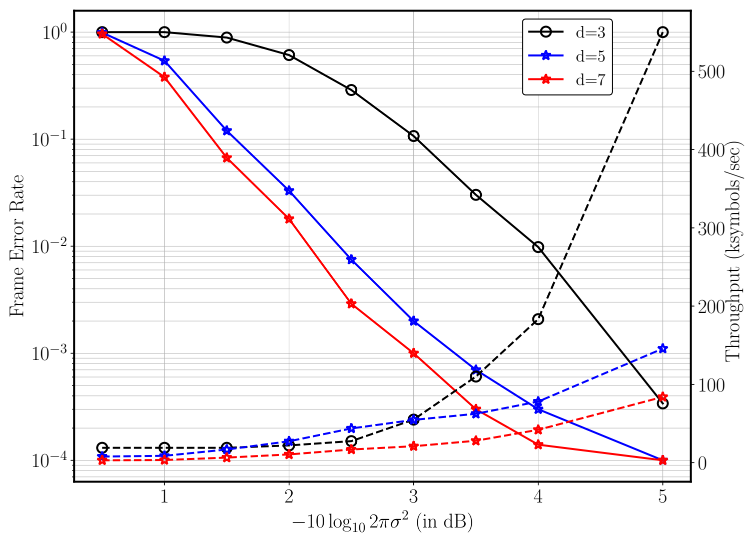


Figure 5.39: FER and throughput comparison of the fixed-point multi-Gaussian decoders with degree 3 , 5 and 7 for block length 1000 and M=2.

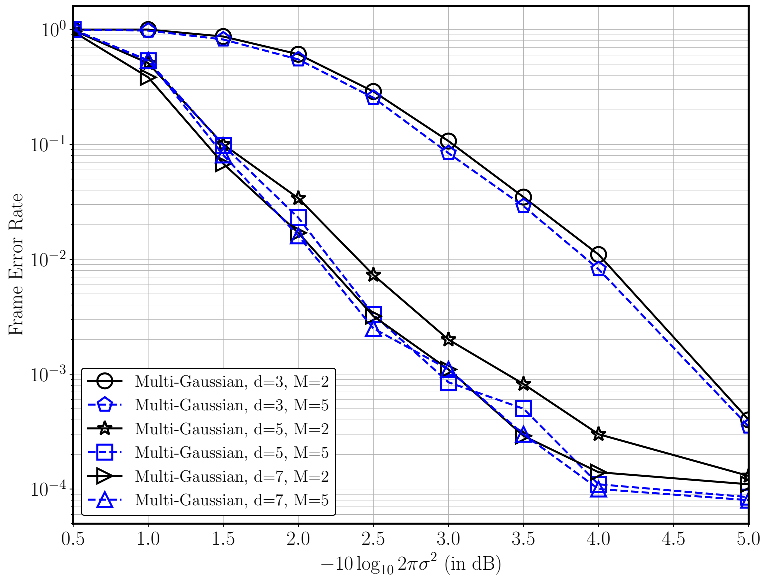


Figure 5.40: FER of the multi-Gaussian decoder with larger Gaussian mixtures exchanged in iterative decoding (i.e., $M = 5$) for degree 3, 5 , 7 and $n = 1000$.

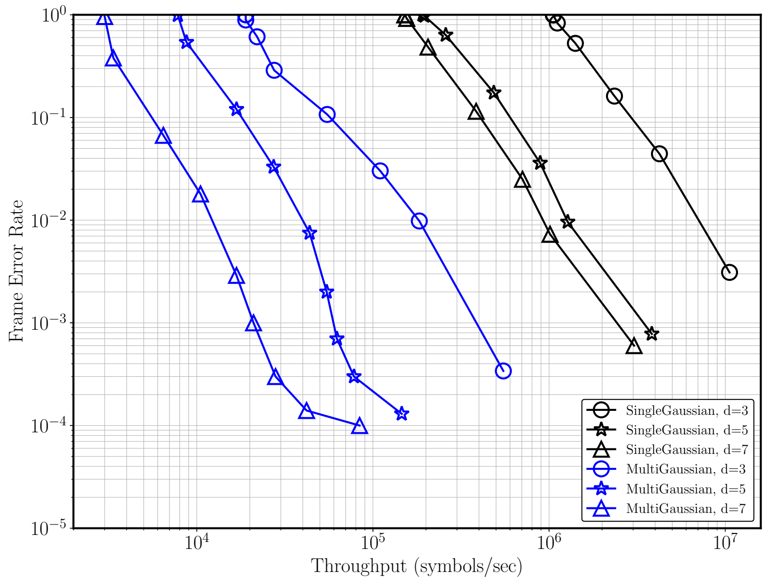


Figure 5.41: Comparison of the frame error rate and throughput of the fixed-point single-Gaussian and multi-Gaussian decoders for block length 1000.

Chapter 6

Concluding Remarks

6.1 Summary of Contributions and Conclusions

While the literature has demonstrated the ideal decoding capabilities of LDLC decoders assuming floating point arithmetic, practical hardware implementations are missing. A hardware implementation of an LDLC decoder is important to realize the potential of LDLC codes in real applications. Therefore, in this thesis we presented the first hardware implementation of decoders for low-density lattice codes. Specifically, fixed-point single-Gaussian and multi-Gaussian LDLC decoders were designed on hardware with a detailed exploration of various parameters of the design space. The decoding performance and design throughput were studied for different degrees of the H-matrix (used for LDLC code construction) and sizes of the Gaussian mixture message.

The following are the major contributions of the thesis:

In Chapter 3, we presented a fixed-point single-Gaussian LDLC decoder implementation on an FPGA. To the best of our knowledge, this is the first proof-of-concept of LDLC decoding in hardware. We implemented three different architectures for the decoder and studied FER and design throughput trade-offs between (a) a serial architecture with a single check node and a single variable node, (b) an architecture with a single check node and 20 parallel variable nodes, and (c) an architecture with a single check node and with pipelining to achieve an effective parallelism equivalent to 50 variable nodes.

The primary accomplishments of the single-Gaussian LDLC decoder implementation are as follows:

- We completed a detailed quantization analysis (to get suitable word length) and devised methods to approximate the required non-linear functions of (a) division and (b) exponentiation in fixed-point arithmetic.

- We designed a serial architecture with a single check node and a single variable node that attained a design throughput of 440 KSymbols/sec. Then a partially parallel architecture with a single check node and 20 variable nodes was designed that gained an $\sim 13\times$ improvement in throughput over the serial architecture. To further improve the decoder throughput, we implemented an architecture with pipelining (in the variable node message processing block) that comprises a single check node and a parallelism equivalent to 50 variable nodes. This implementation achieved a design throughput of $\sim 24\times$ compared to the serial architecture.

In Chapter 4 we implemented a multi-Gaussian LDLC decoder of degree 3 where the Gaussian mixture messages that are exchanged during iterative decoding consist of two components. This decoder was specifically designed with the aim of improving the performance of the LDLC decoder presented in Chapter 3.

We first designed a multi-Gaussian decoder implementation in floating-point arithmetic and matched the performance results against the literature results. Then, using efficient methods to approximate non-linear functions and a word length of 28 (12 integer bits, 15 fractional bits and a sign bit) we obtained a decoder implementation in fixed-point arithmetic.

We implemented 2 different architectures for the multi-Gaussian decoder: (a) a serial architecture with a single check node and a single variable node, and (b) a partially parallel architecture with a single check node and pipelining to achieve an effective parallelism equivalent to 5 variable nodes. The serial architecture achieved a throughput of 245 Ksymbols/sec and the partially parallel decoder with an effective parallelism equivalent to 5 variable nodes (at the variable node message processing block) attained a $2.25\times$ improvement in throughput over the serial implementation.

The multi-Gaussian decoder of degree 3 achieved an overall improvement of ~ 0.75 dB in FER compared to the single-Gaussian decoder of degree 3 for block length 1000. However, due to the more complex check node and variable node message processing blocks this FER improvement was obtained at a cost of reduced decoder throughput. The single-Gaussian decoder (Chapter 3) throughput was $19\times$ compared to the multi-Gaussian decoder (Chapter 4) for degree 3.

In Chapter 5 of the thesis, we explored the design space for single-Gaussian and multi-Gaussian decoders. The implementation of LDLC decoders with a wide range of architectural and design parameters presents different throughputs, hardware costs and decoding performance trade-offs. In this chapter, we developed complex LDLC decoders of higher degrees, i.e., 5 and 7, with larger Gaussian mixture messages, i.e. $M = 5$.

The decoding performance and throughput results for the single-Gaussian and multi-Gaussian decoders suggest that for applications which require very high data reliability, multi-Gaussian decoders are the preferred choice. However, in high throughput regimes, single-Gaussian LDLC decoders are more desirable.

6.2 Directions for Future Work

In the following section, we present possible design enhancements for the single-Gaussian and multi-Gaussian decoders.

Simultaneous Processing of Check Node and Variable Node Messages The current implementations of the check node and variable node message processing blocks do not operate simultaneously as they may overwrite data which has not yet been processed by the other node. Decoder architectures where both nodes operate simultaneously can be explored in the future. This design enhancement would prevent half of the hardware from remaining idle during execution and would improve the decoder throughput. However, this could involve unknown challenges and can be the focus for future research.

Pipelining Unrolling of the recursive forward-backward computation block used in the variable node message computation and potential pipelining can be an area of study in the future work.

Variable Precision Bits for Fixed-Point LDLC Decoders The LDLC decoders have used a fixed number of fractional bits throughout the design. However the implementations can be extended to a different number of fractional bits in different blocks. This can help to reduce the decoder cost further in hardware.

ASIC Implementation In this thesis, we have implemented LDLC decoders on an FPGA device to achieve a proof-of-concept of LDLC decoding in hardware. FPGAs are a suitable choice for an early prototype of LDLC decoders in hardware and constitute a viable alternative to an ASIC implementation. However, for practical applications that require high performance and low-power consumption, ASIC implementations are preferred. Thus, an ASIC implementation of LDLC decoders is important future work. The results obtained in this thesis can be used as a baseline for ASIC implementation.

Greater Block Lengths In this thesis, we explored the design space of FPGA implementations of the single-Gaussian and multi-Gaussian decoders for block length of 1000. However, it would be quite compelling in future research to study the architectural design space for LDLC decoder implementations with greater block lengths, e.g., 10000 or more, on a larger FPGA or in an ASIC implementation in order to achieve higher decoding performance for LDLCs in hardware.

Applications Now that the hardware decoding of LDLCs was shown to be feasible, an important direction for the future research is to explore the application of the LDLC decoders in real world communication networks.

References

[1] Cisco, "Cisco annual internet report (2018–2023)," *Report*, Mar. 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[2] Satista, "Number of wireless local area network (wlan) connected devices worldwide from 2016 to 2021," *Report*, Jul. 2020. [Online]. Available: https://www.statista.com/statistics/802706/world-wlan-connected-device/

[3] O. Ordentlich, J. Zhan, U. Erez, M. Gastpar, and B. Nazer, "Practical code design for compute-and-forward," *IEEE International Symposium on Information Theory*, pp. 1876–1880, July 2011.

[4] M. N. Hasan and B. M. Kurkoski, "Practical compute-and-forward approaches for the multiple access relay channel," *IEEE International Conference on Communications*, pp. 1–6, May 2017.

[5] S. H. Lee, A. Ghiya, S. Vishwanath, S. S. Hwang, and S. Kim, "Structured dirty-paper coding using low-density lattices," *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 3350–3353, Mar. 2010.

[6] N. Sommer, M. Feder, and O. Shalvi, "Low-density lattice codes," *IEEE International Symposium on Information Theory*, pp. 88–92, July 2006.

[7] ——, "Low-density lattice codes," *IEEE Transactions on Information Theory*, vol. 54, no. 4, pp. 1561–1585, April 2008.

[8] M. R. Sadeghi, A. H. Banihashemi, and D. Panario, "Iterative decoding algorithm of lattices," *Canadian Conference on Electrical and Computer Engineering*, vol. 3, pp. 1417–1420, May 2004.

[9] Y. Yona and M. Feder, "Efficient parametric decoder of low density lattice codes," *IEEE International Symposium on Information Theory*, pp. 744–748, June 2009.

[10] R. A. P. Hernandez and B. M. Kurkoski, "The three/two Gaussian parametric LDLC decoder," *IEEE Information Theory Workshop*, pp. 172–176, Oct. 2015.

[11] B. M. Kurkoski and J. Dauwels, "Message-passing decoding of lattices using Gaussian mixtures," *IEEE International Symposium on Information Theory*, pp. 2489–2493, July 2008.

[12] B. Kurkoski and J. Dauwels, "Reduced-memory decoding of low-density lattice codes," *IEEE Communications Letters*, vol. 14, no. 7, pp. 659–661, July 2010.

[13] Y. Li and Z. Bie, "Message-passing decoding algorithm of low-density lattice codes with Gaussian approximation," *IEEE International Conference on Wireless Information Technology and Systems*, pp. 1–4, Nov. 2012.

[14] C. Ling and J. Belfiore, "Achieving the AWGN channel capacity with lattice Gaussian coding," *IEEE International Symposium on Information Theory*, pp. 1416–1420, July 2013.

[15] R. A. P. Hernandez and B. M. Kurkoski, "Low complexity construction of low density lattice codes based on array codes," *International Symposium on Information Theory and its Applications*, pp. 264–268, Oct. 2014.

[16] B. Chen, D. N. K. Jayakody, and M. F. Flanagan, "Distributed low-density lattice codes," *IEEE Communications Letters*, vol. 20, no. 1, pp. 77–80, Jan. 2016.

[17] P. Mitran and H. Ochiai, "Parallel concatenated convolutional lattice codes with constrained states," *IEEE Transactions on Communications*, vol. 63, no. 4, pp. 1081–1090, April 2015.

[18] B. M. Kurkoski, J. Dauwels, and H. A. Loeliger, "Power-constrained communications using LDLC lattices," *IEEE International Symposium on Information Theory*, pp. 739–743, June 2009.

[19] J. Xu, C. Duan, D. Zhao, Y. Wang, and F. Xie, "Early stopping criterion for message-passing decoding of LDLC," *International Conference on Advanced Communication Technology*, pp. 315–318, July 2015.

[20] Y. Li and Z. Bie, "Message-passing decoding algorithm of low-density lattice codes with Gaussian approximation," *IEEE International Conference on Wireless Information Technology and Systems*, pp. 1–4, Nov. 2012.

[21] P. Elias, "Coding for noisy channels," *IRE Convention Record*, vol. 3, no. 4, pp. 37–46, Mar. 1955.

[22] R. de Buda, "The upper error bound of a new near-optimal code," *IEEE Transactions on Information Theory*, vol. 21, no. 4, pp. 441–445, July 1975.

[23] ——, "Some optimal codes have structure," *IEEE Journal on Selected Areas in Communications*, vol. 7, no. 6, pp. 893–899, Aug. 1989.

[24] T. Linder, C. Schlegal, and K. Zeger, "Corrected proof of de Buda's theorem (lattice channel codes)," *IEEE Transactions on Information Theory*, vol. 39, no. 5, pp. 1735–1737, 1993.

[25] H. Loeliger, "Averaging bounds for lattices and linear codes," *IEEE Transactions on Information Theory*, vol. 43, no. 6, pp. 1767–1773, 1997.

[26] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, Jul.-Oct. 1948.

[27] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1," *IEEE International Conference on Communications*, vol. 2, pp. 1064–1070, May 1993.

[28] R. G. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[29] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, Mar. 1999.

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

[31] C. Howland and A. Blanksby, "A 220 mW 1 GB/s 1024-bit rate-1/2 low density parity check code decoder," *IEEE Custom Integrated Circuits Conference*, pp. 293–296, 2001.

[32] C. E. Shannon, "Probability of error for optimal codes in a Gaussian channel," *The Bell System Technical Journal*, vol. 38, no. 3, pp. 611–656, 1959.

[33] R. Urbanke and B. Rimoldi, "Lattice codes can achieve capacity on the AWGN channel," *IEEE Transactions on Information Theory*, vol. 44, no. 1, pp. 273–278, 1998.

[34] U. Erez and R. Zamir, "Achieving 1/2 log (1+SNR) on the AWGN channel with lattice encoding and decoding," *IEEE Transactions on Information Theory*, vol. 50, no. 10, pp. 2293–2314, 2004.

[35] B. Nazer and M. Gastpar, "Compute-and-forward: Harnessing interference through structured codes," *IEEE Transactions on Information Theory*, vol. 57, no. 10, pp. 6463–6486, Oct. 2011.

[36] J. Zhu and M. Gastpar, "Lattice codes for many-to-one interference channels with and without cognitive messages," *IEEE Transactions on Information Theory*, vol. 61, no. 3, pp. 1309–1324, Mar. 2015.

[37] A. Mejri and G. Rekaya-Ben Othman, "Efficient decoding algorithms for the compute-and-forward strategy," *IEEE Transactions on Communications*, vol. 63, no. 7, pp. 2475–2485, 2015.

[38] Y. Tan and X. Yuan, "Compute-compress-and-forward: Exploiting asymmetry of wireless relay networks," *IEEE Transactions on Signal Processing*, vol. 64, no. 2, pp. 511–524, 2016.

[39] A. Mejri, G. R. Othman, and J. C. Belfiore, "Lattice decoding for the compute-and-forward protocol," *International Conference on Communications and Networking*, pp. 1–8, 2012.

[40] C. Feng, D. Silva, and F. R. Kschischang, "Blind compute-and-forward," *IEEE Transactions on Communications*, vol. 64, no. 4, pp. 1451–1463, 2016.

[41] M. Nokleby and B. Aazhang, "Cooperative compute-and-forward," *IEEE Transactions on Wireless Communications*, vol. 15, no. 1, pp. 14–27, 2016.

[42] Y. Wang and A. Burr, "Physical-layer network coding via low density lattice codes," *European Conference on Networks and Communications*, pp. 1–5, 2014.

[43] R. Zamir, S. Shamai, and U. Erez, "Nested linear/lattice codes for structured multiterminal binning," *IEEE Transactions on Information Theory*, vol. 48, no. 6, pp. 1250–1276, June 2002.

[44] N. E. Tunali, K. R. Narayanan, and H. D. Pfister, “Spatially-coupled low density lattices based on construction A with applications to compute-and-forward,” *IEEE Information Theory Workshop*, pp. 1–5, Sept. 2013.

[45] Y. Huang, K. R. Narayanan, and P. Wang, “Adaptive compute-and-forward with lattice codes over algebraic integers,” *IEEE International Symposium on Information Theory*, pp. 566–570, 2015.

[46] J. Belfiore, “Lattice codes for the compute-and-forward protocol: The flatness factor,” *IEEE Information Theory Workshop*, 2011.

[47] S. Gelincik and G. R. Othman, “Lattice codes for C-RAN based sectored cellular networks,” *IEEE International Conference on Communications*, pp. 1–7, 2020.

[48] Y. Huang and K. R. Narayanan, “Multistage compute-and-forward with multilevel lattice codes based on product constructions,” *IEEE International Symposium on Information Theory*, pp. 2112–2116, 2014.

[49] S. Lin and D. J. Costello, *Error control coding: fundamentals and applications.* NJ: Pearson/Prentice Hall, 2004.

[50] R. Roth, *Introduction to Coding Theory.* USA: Cambridge University Press, 2006.

[51] B. Sklar and F. Harris, “The ABCs of linear block codes,” *IEEE Signal Processing Magazine*, vol. 21, pp. 14 – 35, Aug. 2004.

[52] J. G. Proakis, *Digital Communications 5th Edition.* McGraw Hill, 2007.

[53] A. G. Burr, “Block versus trellis: an introduction to coded modulation,” *Electronics and Communication Engineering Journal*, vol. 5, no. 4, pp. 240–248, Aug. 1993.

[54] R. Tanner, “A recursive approach to low complexity codes,” *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, Sept. 1981.

[55] K. Chung and J. Heo, “Improved belief propagation (BP) decoding for LDPC codes with a large number of short cycles,” *IEEE Vehicular Technology Conference*, vol. 3, pp. 1464–1466, 2006.

[56] G. Han and X. Liu, “An efficient dynamic schedule for layered belief-propagation decoding of LDPC codes,” *IEEE Communications Letters*, vol. 13, no. 12, pp. 950–952, 2009.

[57] J. Chen, Y. Zhang, and R. Sun, "An improved normalized min-sum algorithm for LDPC codes," *IEEE/ACIS International Conference on Computer and Information Science*, pp. 509–512, 2013.

[58] Y. Cao, "An improved LDPC decoding algorithm based on min-sum algorithm," *International Symposium on Communications Information Technologies*, pp. 26–29, 2011.

[59] W. Han, J. Huang, and Fangfei Wu, "A modified min-sum algorithm for low-density parity-check codes," *IEEE International Conference on Wireless Communications, Networking and Information Security*, pp. 449–451, 2010.

[60] V. V. Vityazev, E. A. Likhobabin, and E. A. Ustinova, "Min-sum algorithm-structure based decoding algorithms for LDPC codes," *Mediterranean Conference on Embedded Computing*, pp. 256–259, 2014.

[61] P. Dhanorkar and M. Kalbande, "Design of LDPC decoder using message passing algorithm," *International Conference on Communication and Signal Processing*, pp. 1923–1926, 2017.

[62] R. Lehmann and G. M. Maggio, "An approximate analytical model of the message passing decoder of LDPC codes," *IEEE International Symposium on Information Theory,*, p. 31, 2002.

[63] P. Radosavljevic, A. de Baynast, and J. R. Cavallaro, "Optimized message passing schedules for LDPC decoding," *Asilomar Conference on Signals, Systems and Computers*, pp. 591–595, 2005.

[64] G. Poltyrev, "On coding without restrictions for the AWGN channel," *IEEE Transactions on Information Theory*, vol. 40, no. 2, pp. 409–417, Mar. 1994.

[65] D. F. Crouse, P. Willett, K. Pattipati, and L. Svensson, "A look at Gaussian mixture reduction algorithms," *International Conference on Information Fusion*, July 2011.

[66] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye, *Probability & statistics for engineers and scientists*, 8th ed. Pearson Education, 2007.

[67] H. Cramer, *Mathematical Methods of Statistics (PMS-9).* Princeton University Press, 1999.

[68] S. Liu, Y. Hong, E. Viterbo, A. Marelli, and R. Micheloni, "Efficient decoding of low density lattice codes," *IEEE Wireless Communications Letters*, vol. 8, no. 4, pp. 1195–1199, 2019.

[69] F. Kschischang, B. Frey, and H. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[70] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. Oxford University Press, 2010.

[71] M. Ferrari, S. Bellini, and A. Tomasoni, "Safe early stopping for layered LDPC decoding," *IEEE Communications Letters*, vol. 19, no. 3, pp. 315–318, 2015.

[72] T. Chen, "An early stopping criterion for LDPC decoding based on average weighted reliability measure," *Cross Strait Quad-Regional Radio Science and Wireless Technology Conference*, pp. 123–126, 2012.

[73] J. Sodha, "Early stopping criterion for LDPC," *International Conference on Circuits, System and Simulation*, pp. 134–137, 2017.

[74] J. Wang, J. He, and X. Xu, "An early stopping criterion for LDPC decoder based on CMMB standard," *International Conference on BioMedical Engineering and Informatics*, pp. 1432–1434, 2012.

[75] E. Libessart, M. Arzel, C. Lahuec, and F. Andriulli, "A scaling-less Newton–Raphson pipelined implementation for a fixed-point reciprocal operator," *IEEE Signal Processing Letters*, vol. 24, no. 6, pp. 789–793, 2017.

[76] A. Rodriguez-Garcia, L. Pizano-Escalante, R. Parra-Michel, O. Longoria-Gandara, and J. Cortez, "Fast fixed-point divider based on Newton-Raphson method and piecewise polynomial approximation," *International Conference on Reconfigurable Computing and FPGAs*, pp. 1–6, 2013.

[77] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[78] S. S. Jadhav, C. Gloster, J. Naher, C. Doss, and Y. Kim, "An FPGA-based application-specific processor for implementing the exponential function," *South east Conference*, pp. 1–8, 2020.

[79] P. Nilsson, A. U. R. Shaik, R. Gangarajaiah, and E. Hertz, "Hardware implementation of the exponential function using Taylor series," *NORCHIP*, pp. 1–4, 2014.

[80] C. Chang, S. Chen, B. Chen, J. Wang, and J. Wang, “A division-free algorithm for fixed-point power exponential function in embedded system,” *International Conference on Orange Technologies*, pp. 223–226, 2013.

[81] P. Pouyan, E. Hertz, and P. Nilsson, “A VLSI implementation of logarithmic and exponential functions using a novel parabolic synthesis methodology compared to the CORDIC algorithm,” *European Conference on Circuit Theory and Design*, pp. 709–712, 2011.

[82] H. Michel, A. Worm, and N. Wehn, “Influence of quantization on the bit-error performance of turbo-decoders,” *IEEE Vehicular Technology Conference*, vol. 1, pp. 581–585, 2000.

[83] T. M. Cover and J. A. Thomas, *Elements of Information Theory.* Wiley, 1991.

APPENDICES

Appendix A

Derivation of the Squared Distance Equation

This derivation is adapted from [11,83]. The squared distance, SD (i.e., Gaussian quadratic loss (GQL)) between a Gaussian mixture $p(t)$ and its approximation $q(t)$ is defined as ,

$$SD(p(t), q(t)) = \int_{-\infty}^{\infty} (p(t) - q(t))^2 dx. \tag{A.1}$$

Let's assume a normalized Gaussian mixture $p(t)$ with two components, represented by $\{(c_1, m_1, V_1), (c_2, m_2, V_2)\}$ where $c_1 + c_2 = 1$. The distribution of $p(t)$ can be written as

$$p(t) = \frac{c_1}{\sqrt{2\pi V_1}} e^{-\frac{(t-m_1)^2}{2V_1}} + \frac{c_2}{\sqrt{2\pi V_2}} e^{-\frac{(t-m_2)^2}{2V_2}}. \tag{A.2}$$

If the single-Gaussian approximation $q(t)$ is represented by the triple $(m_{\mathcal{MM}}, V_{\mathcal{MM}}, c_{\mathcal{MM}})$, then the distribution for $q(t)$ is given by

$$q(t) = c_{\mathcal{MM}} \frac{1}{\sqrt{2\pi V_{\mathcal{MM}}}} e^{-\frac{(t-m_{\mathcal{MM}})^2}{2V_{\mathcal{MM}}}}. \tag{A.3}$$

Further combining (A.2) and (A.3), (A.1) can be written as,

$$SD(p(t), q(t)) = \int_{-\infty}^{\infty} \Big(\frac{c_1}{\sqrt{2\pi V_1}} e^{-\frac{(t-m_1)^2}{2V_1}} + \frac{c_2}{\sqrt{2\pi V_2}} e^{-\frac{(t-m_2)^2}{2V_2}} - \frac{1}{\sqrt{2\pi V_{\mathcal{MM}}}} e^{-\frac{(t-m_{\mathcal{MM}})^2}{2V_{\mathcal{MM}}}} \Big)^2 dt. \tag{A.4}$$

Integration in (A.4) will give two type of terms, *squared* terms and $cross - product$ terms. Integrating the first *squared* term yields,

$$\int_{-\infty}^{\infty}\Big(\frac{c_1}{\sqrt{2\pi V_1}}e^{-\frac{(t-m_1)^2}{2V_1}}\Big)^2 dt = \int_{-\infty}^{\infty}\Big(\frac{c_1}{\sqrt{2\pi V_1}}\frac{c_1}{\sqrt{2\pi V_1}}e^{-\frac{2(t-m_1)^2}{2V_1}}\Big)dt \tag{A.5}$$

$$= \frac{{c_1}^2}{2\sqrt{\pi V_1}}\int_{-\infty}^{\infty}\Big(\frac{1}{\sqrt{2\pi\frac{(V_1)}{2}}}e^{-\frac{(t-m_1)^2}{2(\frac{V_1}{2})}}\Big)dt \tag{A.6}$$

$$= \frac{{c_1}^2}{2\sqrt{\pi V_1}} \tag{A.7}$$

Similarly the other *squared* terms integrate to $\frac{{c_2}^2}{2\sqrt{\pi V_2}}$ and $\frac{1}{2\sqrt{\pi V_{\mathcal{MM}}}}$.

For the first $cross - product$ term we have,

$$\int_{-\infty}^{\infty}\Big(\frac{2c_1}{\sqrt{2\pi V_1}}e^{-\frac{(t-m_1)^2}{2V_1}}\frac{c_2}{\sqrt{2\pi V_2}}e^{-\frac{(t-m_2)^2}{2V_2}}\Big)dt \tag{A.8}$$

$$= \int_{-\infty}^{\infty}\frac{2c_1c_2}{\sqrt{2\pi V_1}\sqrt{2\pi V_2}}e^{-\left(\frac{(t-m_1)^2}{2V_1}+\frac{(t-m_2)^2}{2V_2}\right)}dt \tag{A.9}$$

$$= \int_{-\infty}^{\infty}\frac{2c_1c_2}{\sqrt{2\pi}\sqrt{2\pi V_1V_2}}e^{-\left(\frac{(t-m_1)^2}{2V_1}+\frac{(t-m_2)^2}{2V_2}\right)}dt \tag{A.10}$$

$$= \int_{-\infty}^{\infty}\frac{2c_1c_2}{\sqrt{2\pi}\sqrt{2\pi V_1V_2}}e^{-\left(\frac{V_2(t-m_1)^2+V_1(t-m_2)^2}{2V_1V_2}\right)}dt \tag{A.11}$$

$$= \int_{-\infty}^{\infty}\frac{2c_1c_2}{\sqrt{2\pi}\sqrt{2\pi(V_1V_2)\frac{(V_1+V_2)}{V_1+V_2}}}e^{-\left(\frac{(V_1+V_2)t^2-2(m_1V_2+m_2V_1)t+V_1m_2^2+V_2m_1^2}{2V_1V_2}\right)}dt \tag{A.12}$$

$$= \int_{-\infty}^{\infty}\frac{2c_1c_2}{\sqrt{2\pi(V_1+V_2)}\sqrt{2\pi\frac{V_1V_2}{V_1+V_2}}}e^{-\left(\frac{(V_1+V_2)t^2-2(m_1V_2+m_2V_1)t+V_1m_2^2+V_2m_1^2}{2V_1V_2}\right)}dt \tag{A.13}$$

$$= \int_{-\infty}^{\infty}\frac{2c_1c_2}{\sqrt{2\pi(V_1+V_2)}\sqrt{2\pi\frac{V_1V_2}{V_1+V_2}}}e^{-\left(\frac{t^2-2\frac{(m_1V_2+m_2V_1)}{(V_1+V_2)}t+\frac{(V_1m_2^2+V_2m_1^2)}{(V_1+V_2)}}{2\frac{V_1V_2}{V_1+V_2}}\right)}dt \tag{A.14}$$

Solving the integration and re-arranging the terms, we obtain

$$\frac{2c_1c_2}{\sqrt{2\pi(V_1+V_2)}}e^{-\left(\frac{(m_1-m_2)^2}{2(V_1+V_2)}\right)}. \tag{A.15}$$

Similarly the other two *cross−product* terms integrate to $\frac{2c_1}{\sqrt{2\pi(V_{\mathcal{MM}}+V_1)}}e^{-\left(\frac{(m_{\mathcal{MM}}-m_1)^2}{2(V_{\mathcal{MM}}+V_1)}\right)}$ and $\frac{2c_2}{\sqrt{2\pi(V_{\mathcal{MM}}+V_2)}}e^{-\left(\frac{(m_{\mathcal{MM}}-m_2)^2}{2(V_{\mathcal{MM}}+V_2)}\right)}$. Therefore, the GQL or the penalty for merging the two Gaussian components to a single Gaussian is given by

$$\begin{aligned} SD(p(t),q(t)) = {} & \frac{1}{2\sqrt{\pi V_{\mathcal{MM}}}} + \frac{c_1^2}{2\sqrt{\pi V_1}} + \frac{c_2^2}{2\sqrt{\pi V_2}} - \frac{2c_1}{\sqrt{2\pi(V_{\mathcal{MM}}+V_1)}}e^{-\frac{(m_{\mathcal{MM}}-m_1)^2}{2(V_{\mathcal{MM}}+V_1)}} \\ & - \frac{2c_2}{\sqrt{2\pi(V_{\mathcal{MM}}+V_2)}}e^{-\frac{(m_{\mathcal{MM}}-m_2)^2}{2(V_{\mathcal{MM}}+V_2)}} + \frac{2c_1c_2}{\sqrt{2\pi(V_1+V_2)}}e^{-\frac{(m_1-m_2)^2}{2(V_1+V_2)}}. \end{aligned} \tag{A.16}$$

Appendix B

Newton Raphson Approximation for a Reciprocal Function

The Newton-Raphson approximation can be used to compute the reciprocal of a given number, a. To compute the reciprocal let's assume,

$$f(x) = \frac{1}{a},\ or \tag{B.1}$$

$$f(x) = a^{-1}. \tag{B.2}$$

The Newton-Raphson equation is,

$$x_{i+1} = x_i - \frac{f(x_i)}{f_1(x_i)}, \tag{B.3}$$

where i denotes the iteration number and $f_1(x_i)$ is the first derivative of $f(x_i)$. The function which is used to compute the reciprocal of a is $f(x) = (x^{-1}) - a$. For function $f(x)$, the Newton-Raphson iteration (B.3) is given as,

$$x_{i+1} = x_i(2 - ax_i). \tag{B.4}$$

Appendix C

Example Exponential Approximation in Fixed Point Representation

Here we will explain the exponential approximation applied in the LDLC decoder implementation with a specific example of a fixed-point representation. Specifically, consider Q12.8 that is used in the single-Gaussian decoder (in Chapter 3), with 12 integer bits, 8 fractional bits and a sign bit. Recall that we wish to compute $\exp(-a)$ where $a \geq 0$.

Let $b_0, \ldots, b_{20}$ be the bits that represent the fixed point number a in two's complement and b_0 is the least-significant bit. Then

$$a = b_0 \times 2^{-8} + b_1 \times 2^{-7} + b_2 \times 2^{-6} + \cdots + b_{17} \times 2^{9} + b_{18} \times 2^{10} + b_{19} \times 2^{11} - b_{20} \times 2^{12}. \tag{C.1}$$

Since $a \geq 0$, the sign bit $b_{20} = 0$, and therefore

$$a = b_0 \times 2^{-8} + b_1 \times 2^{-7} + b_2 \times 2^{-6} + \cdots + b_{17} \times 2^{9} + b_{18} \times 2^{10} + b_{19} \times 2^{11}. \tag{C.2}$$

After factoring and algebraic manipulations, we have

$$\begin{aligned} a = (b_0 + b_1 2^1 + \cdots + b_5 2^5)2^{-8} &+ (b_6 + b_7 2^1 + b_8 2^2 + b_9 2^3 + b_{10} 2^4)2^{-2} \\ &+ (b_{11} + b_{12} 2^1 + \cdots + b_{19} 2^8)2^3, \end{aligned} \tag{C.3}$$

and after rearranging terms in (C.3), a is represented as

$$\begin{aligned} a = \underbrace{(b_{11} + b_{12} 2^1 + \cdots + b_{19} 2^8)}_{\triangleq I_2} \underbrace{2^3}_{\triangleq 2^{P_2}} &+ \underbrace{(b_6 + b_7 2^1 + b_8 2^2 + b_9 2^3 + b_{10} 2^4)}_{\triangleq I_1} \underbrace{2^{-2}}_{\triangleq 2^{P_1}} \\ &+ \underbrace{(b_0 + b_1 2^1 + \cdots + b_5 2^5)}_{\triangleq I_0} \underbrace{2^{-8}}_{\triangleq 2^{P_0}}. \end{aligned} \tag{C.4}$$

where $\triangleq$ denotes equality by definition. From this, we have that

$$a = I_2 2^{P_2} + I_1 2^{P_1} + I_0 2^{P_0}. \tag{C.5}$$

where

$$I_0 = b_0 + b_1 2^1 + b_2 2^2 + b_3 2^3 + b_4 2^4 + b_5 2^5 \tag{C.6}$$
$$I_1 = b_6 + b_7 2^1 + b_8 2^2 + b_9 2^3 + b_{10} 2^4 \tag{C.7}$$
$$I_2 = b_{11} + b_{12} 2^1 + b_{13} 2^2 + b_{14} 2^3 + b_{15} 2^4 + b_{16} 2^5 + b_{17} 2^6 + b_{18} 2^7 + b_{19} 2^8 \tag{C.8}$$
$$P_0 = -8 \tag{C.9}$$
$$P_1 = -2 \tag{C.10}$$
$$P_2 = 3 \tag{C.11}$$

As is evident, $P_0 < P_1 < P_2$ by construction. One could swap the definitions of P_0 and P_1 in (C.9) – (C.10) as well as I_0 and I_1 in (C.6) – (C.7), and (C.5) would still hold. But by convention we pick to keep P_0, P_1 and P_2 ordered, i.e., $P_0 < P_1 < P_2$.

The range of I_0 is the range of $b_0 + b_1 2^1 + b_2 2^2 + b_3 2^3 + b_4 2^4 + b_5 2^5$, which is from 0 to $2^6 - 1$. Similarly the range for I_1 is from 0 to $2^5 - 1$ and that of I_2 is from 0 to $2^9 - 1$.

Finally, I_0 has b_0 as its least-significant bit and is weighted by 2^{-8} when computing a. Likewise I_1 has b_6 as its least-significant bit and is weighted by 2^{-2} when computing a. Since the most-significant bit of I_0 (i.e., b_5) is the bit before the least-significant bit of I_1 (i.e., b_6), the number of bits that comprise I_0 is given by $6 = -2 - (-8) = P_1 - P_0$. Hence, the range of I_0 is from 0 to $2^{P_1 - P_0} - 1 = 63$, as confirmed by (C.6).

The two lookup tables used in the exponential approximation are look-up tables to approximate $\exp(-I_1 2^{P_1})$ from I_1 and to approximate $\exp(-I_0 2^{P_0})$ from I_0.

LUT for $\exp(-I_1 2^{P_1})$**:** This lookup table contains 32 entries corresponding to 32 possible values for I_1, i.e., 0 to $(2^5 - 1)$. Ideally the entries should be $\exp(-\frac{I_1}{4})$ since $P_1 = -2$. Due to a numerical optimization (i.e., to accommodate the division by two in the exponent part of the coefficient calculation at the variable nodes (3.6)) we actually compute $\exp(-I_1 2^{P_1}/2) = \exp(-\frac{I_1/2}{4})$. Table C.1 shows all 32 entries for the $\exp(-I_1 2^{P_1}/2)$ look-up table. Please note that the first 12 zeros of the Q12.8 representation are not stored in the LUT.

Table C.1: Look up table for $\exp(-I_1 2^{P_1}/2)$

LUT Index (I_1)	Value ($\exp(-I_1 2^{P_1}/2)$)	Q12.8 representation
0	$\exp\left(-\frac{0/2}{4}\right)$	00000000000100000000
1	$\exp\left(-\frac{1/2}{4}\right)$	00000000000011100001
2	$\exp\left(-\frac{2/2}{4}\right)$	00000000000011000111
3	$\exp\left(-\frac{3/2}{4}\right)$	00000000000010110000
4	$\exp\left(-\frac{4/2}{4}\right)$	00000000000010011011
5	$\exp\left(-\frac{5/2}{4}\right)$	00000000000010001001
6	$\exp\left(-\frac{6/2}{4}\right)$	00000000000001111001
7	$\exp\left(-\frac{7/2}{4}\right)$	00000000000001101011
8	$\exp\left(-\frac{8/2}{4}\right)$	00000000000001011110
9	$\exp\left(-\frac{9/2}{4}\right)$	00000000000001010011
10	$\exp\left(-\frac{10/2}{4}\right)$	00000000000001001001
11	$\exp\left(-\frac{11/2}{4}\right)$	00000000000001000001
12	$\exp\left(-\frac{12/2}{4}\right)$	00000000000000111001
13	$\exp\left(-\frac{13/2}{4}\right)$	00000000000000110010
14	$\exp\left(-\frac{14/2}{4}\right)$	00000000000000101100
15	$\exp\left(-\frac{15/2}{4}\right)$	00000000000000100111
16	$\exp\left(-\frac{16/2}{4}\right)$	00000000000000100011
17	$\exp\left(-\frac{17/2}{4}\right)$	00000000000000011111
18	$\exp\left(-\frac{18/2}{4}\right)$	00000000000000011011
19	$\exp\left(-\frac{19/2}{4}\right)$	00000000000000011000
20	$\exp\left(-\frac{20/2}{4}\right)$	00000000000000010101
21	$\exp\left(-\frac{21/2}{4}\right)$	00000000000000010011
22	$\exp\left(-\frac{22/2}{4}\right)$	00000000000000010000
23	$\exp\left(-\frac{23/2}{4}\right)$	00000000000000001110
24	$\exp\left(-\frac{24/2}{4}\right)$	00000000000000001101
25	$\exp\left(-\frac{25/2}{4}\right)$	00000000000000001011
26	$\exp\left(-\frac{26/2}{4}\right)$	00000000000000001010

27	$\exp\left(-\frac{27/2}{4}\right)$	000000000000000001001
28	$\exp\left(-\frac{28/2}{4}\right)$	000000000000000001000
29	$\exp\left(-\frac{29/2}{4}\right)$	000000000000000000111
30	$\exp\left(-\frac{30/2}{4}\right)$	000000000000000000110
31	$\exp\left(-\frac{31/2}{4}\right)$	000000000000000000101

LUT for $\exp(-I_0 2^{P_0})$**:** This lookup table contains 64 entries corresponding to 64 possible values for I_0, i.e., 0 to (2^6-1). Considering the same numerical optimization used in the first LUT, the LUT entries are: $\exp(-\frac{0/2}{256}), \exp(-\frac{1/2}{256}), \cdots, \exp(-\frac{62/2}{256}), \exp(-\frac{63/2}{256})$, converted into Q12.8 fixed-point representation.