LOW-COMPLEXITY DECODING ALGORITHMS AND ARCHITECTURES
FOR NON-BINARY LDPC CODES

by

FANG CAI

Submitted in partial fulfillment of the requirements
For the degree of Doctor of Philosophy

Thesis Adviser: Dr. Xinmiao Zhang

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

August, 2013
We hereby approve the thesis/dissertation of

Fang Cai

candidate for the Doctor of Philosophy degree *

(signed) Xinmiao Zhang

(chair of the committee)

Christos Papachristou

Daniel Saab

Francis Merat

06/13/2013

*We also certify that written approval has been obtained for any proprietary material contained therein.
## Table of Contents

**List of Tables**

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
</tbody>
</table>

**List of Figures**

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi</td>
</tr>
</tbody>
</table>

**Acknowledgments**

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>viii</td>
</tr>
</tbody>
</table>

**Abstract**

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
</tbody>
</table>

**Chapter 1. Introduction**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Summary of Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Relaxed Min-max Decoding Algorithm and Architecture</td>
<td>5</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Enhanced Iterative Hard Reliability-based Majority-logic Decoding Algorithm and Architectures</td>
<td>5</td>
</tr>
<tr>
<td>1.1.3</td>
<td>Power Representation Based Decoding Algorithm and Architecture</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Outline of the Thesis</td>
<td>8</td>
</tr>
</tbody>
</table>

**Chapter 2. NB-LDPC Decoding Algorithms**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Existing Works for Belief Propagation Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Belief-propagation Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2</td>
<td>FFT-based Decoder</td>
<td>10</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Mixed Domain Decoder</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Extended Min-sum (EMS) Algorithm</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Existing works based on EMS algorithm</td>
<td>11</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Dataflow of EMS algorithm</td>
<td>12</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Simplified Min-sum algorithm</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Min-max Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Existing works based on Min-max algorithm</td>
<td>15</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Dataflow of Min-max algorithm</td>
<td>16</td>
</tr>
</tbody>
</table>
4.3.2 IHRB-MLGD Architecture for Cyclic NB-LDPC Codes 90
4.4 Partial-parallel Architectures for E-IHRB-MLGD ........ 97
  4.4.1 Extra storage ........................................ 99
  4.4.2 Selector ............................................... 100
  4.4.3 VNU modification ...................................... 101
4.5 Complexity Analyses and Comparisons ..................... 102
4.6 Summary .................................................... 110

Chapter 5. Power Representation Based Decoding Algorithm and Architecture 114
  5.1 Introduction .............................................. 114
  5.2 Check Node Processing for the Min-max Algorithm ....... 115
    5.2.1 Min-max Check Node Processing Based on Power Representa-
    5.2.2 Min-max CNU Architectures Based on Power Representa-
    5.2.3 Reducing Shifting Latency Overhead .................. 124
  5.3 Check Node Processing for the SMSA .................... 125
    5.3.1 SMSA Check Node Processing Based on Power Representa-
    5.3.2 SMSA CNU Architecture Based on Power Representation 128
  5.4 Complexity Comparisons ............................... 132
  5.5 Summary ................................................... 136

Chapter 6. Conclusions and Future Research 137
  6.1 Conclusions .............................................. 137
  6.2 Future Research ......................................... 138

Bibliography 140
List of Tables

3.1 Example of Algorithm F .................................. 42
3.2 Example for Function $\Phi$ ................................ 49
3.3 Example for Function $\Theta$ ............................... 51
3.4 Configurations and sizes of memories ...................... 76
3.5 CNU complexities for (837, 726) QCNB-LDPC code ....... 77
3.6 Decoder complexities for (837, 726) QCNB-LDPC code .... 77

4.1 Word lengths of reliability measures used in simulations ...... 84
4.2 Extra variables need to be stored to implement E-IHRB-MLGD 98
4.3 Extra value updating in the E-IHRB decoding ................ 101
4.4 NB-LDPC decoder complexities ................................ 111
4.5 Generalized complexity analyses for IHRB and E-IHRB decoders 112
4.6 FPGA Implementation results of (403, 226) QCNB-LDPC decoders on Xilinx Virtex-5 XC5VLX200t Device ......... 113
4.7 Synthesis results using SMIC 0.18$\mu$m CMOS technology .... 113

5.1 Latency overhead reduction examples for codes over $GF(32)$ 125
5.2 Finite field element additions over $GF(8)$ for Step I3a ...... 129
5.3 Finite field element additions over $GF(8)$ for Step I3b ...... 130
5.4 Synthesis results of the Min-max CNUs in Fig. 5.1 and [1] using IBM 90nm CMOS Technology ...................... 132
5.5 Synthesis results for Min-max CNUs over $GF(8)$ using IBM 90nm CMOS Technology .......................... 134
5.6 Synthesis results for $K = 4$ SMSA CNUs over $GF(32)$ with 90nm CMOS Technology .............................. 135
## List of Figures

2.1 Trellis of variable-to-check messages ........................................ 21
2.2 FERs of a (744, 653) NB-LDPC code over $GF(2^5)$ and a (3683, 3175) binary LDPC code under AWGN channel .................. 25
2.3 BERs of a (744, 653) NB-LDPC code over $GF(2^5)$ and a (3683, 3175) binary LDPC code under AWGN channel ............. 26
2.4 FERs of a (248, 124) NB-LDPC code over $GF(2^5)$ and a (1270, 635) binary LDPC code under AWGN channel ............. 27
2.5 BERs of a (248, 124) NB-LDPC code over $GF(2^5)$ and a (1270, 635) binary LDPC code under AWGN channel ............. 28
2.6 FER of a (837, 726) QCNB-LDPC code over $GF(2^5)$ with 15 iterations and $n_m = 16$ ........................................ 29
2.7 WERs of NB-LDPC decoding for a (255, 175) cyclic EG code over $GF(2^8)$ ......................................................... 32
2.8 WERs of NB-LDPC decoding for a (403, 226) QC code over $GF(2^5)$ ................................................................. 33
3.1 Trellises of variable-to-check messages: (a) original trellis; (b) reordered trellis .................................................. 37
3.2 FERs of a (837, 726) QCNB-LDPC code over $GF(2^5)$ and a (4191, 3602) binary LDPC with 15 decoding iterations under the AWGN channel .................................................. 43
3.3 FERs of a (248, 137) QCNB-LDPC code over $GF(2^5)$ and a (1270, 652) binary LDPC with 15 decoding iterations under the AWGN channel .................................................. 44
3.4 Architecture of trellis basis constructor ............................. 55
3.5 Architecture of minimum basis constructor ......................... 58
3.6 Architecture of basis recovery module ................................. 62
3.7 Top level architecture of QCNB-LDPC decoder ...................... 66
3.8 Scheduling of the computations in the decoding process ....... 71
4.1 WERs of the E-IHRB-MLGD for a (255, 175) cyclic EG code over $GF(2^8)$ ................................................................. 84
4.2 WERs of the E-IHRB-MLGD for a (403, 226) QC code over $GF(2^5)$ .................................................. 85
4.3 (a) IHRB-MLGD architecture for QCNB-LDPC codes (b) VNU architecture when all $q$ messages are kept .............. 88
4.4 IHRB-MLGD architecture for cyclic NB-LDPC codes .......... 90
4.5 Computation scheduling in IHRB-MLGD for cyclic codes .... 91
4.6 VNU architecture when $n_m < q$ messages are kept ........... 94
4.7 E-IHRB-MLGD architecture for cyclic NB-LDPC codes .... 99
4.8 Architecture for the selector ................................... 100
4.9 VNU architecture modification ............................... 101
5.1 Proposed CNU modified from [1] for $GF(8)$ .................... 119
5.2 Proposed CNU modified from [2] for $GF(8)$ .................... 122
5.3 Proposed architecture for the SMSA over $GF(8)$ ............ 128
Acknowledgments

First and foremost, I wish to thank my advisor, Professor Xinmiao Zhang, for her guidance and support throughout my studies and research at Case Western Reserve University. I am very grateful for her recognition, inspiration, and the exposure and opportunities that I have received during the course of my study.

I would also like to thank Professor Christos Papachristou, Professor Daniel Saab and Professor Francis Merat for their support as members of my defense committee. I am also grateful to Prof. Shu Lin from UC Davis, for his inspiration and encouraging discussion.

My thanks go to the members of our group, particularly, Jiangli Zhu for many insightful discussions during the course work and mental support. I would also express my gratitude to my roommate Wenyuan Yu for his care during these years. I would like to thank Xiang Zou for many inspirational discussions and good time we share. I also would like to thank my friends at Case Western Reserve University, particularly, Hangwei Qian, Xinmu Wang, Yang Chen, Wei Su, David Tian and Ting He, friends at University of Washington, particularly, Tong Zhang and Helen Zhang for their supports and the good time we had. A special thank goes to many others who supported me with their friendship, especially, Dr. Min Li at Pennsylvania State University.
and Dr. Xiaohua Xu at University of Toledo, with whom I had many academic and spiritual discussions. My thanks also go to all the friends in Melton Foundation.

This dissertation is dedicated to my parents. They gave me life and their love has been with me every single moment.
Low-complexity Decoding Algorithms and Architectures for Non-binary LDPC Codes

Abstract

by

FANG CAI

Non-binary low-density parity-check (NB-LDPC) codes can achieve better error-correcting performance than their binary counterparts when the code length is moderate at the cost of higher decoding complexity. The high complexity is mainly caused by the complicated computations in the check node processing and the large memory requirement. In this thesis, three decoding algorithms and corresponding VLSI architectures are proposed for NB-LDPC codes to lower the computational complexity and memory requirement.

The first design is based on the proposed relaxed Min-max decoding algorithm. A novel relaxed check node processing scheme is proposed for the Min-max NB-LDPC decoding algorithm. Each finite field element of $GF(2^p)$ can be uniquely represented by a linear combination of $p$ independent field elements. Making use of this property, an innovative method is developed in this paper to first find a set of the $p$ most reliable variable-to-check messages with independent field elements, called the minimum basis. Then the check-to-
variable messages are efficiently computed from the minimum basis. With very small performance loss, the complexity of the check node processing can be substantially reduced using the proposed scheme. In addition, efficient VLSI architectures are developed to implement the proposed check node processing and overall NB-LDPC decoder. Compared to the most efficient prior design, the proposed decoder for a (837, 726) NB-LDPC code over $GF(2^5)$ can achieve 52% higher efficiency in terms of throughput-over-area ratio.

The second design is based on a proposed enhanced iterative hard reliability-based majority-logic decoding. The recently developed iterative hard reliability-based majority-logic NB-LDPC decoding has better performance-complexity tradeoffs than previous algorithms. Novel schemes are proposed for the iterative hard reliability-based majority-logic decoding (IHRB-MLGD). Compared to the IHRB algorithm, our enhanced (E-)IHRB algorithm can achieve significant coding gain with small hardware overhead. Then low-complexity partial-parallel NB-LDPC decoder architectures are developed based on these two algorithms. Many existing NB-LDPC code construction methods lead to quasi-cyclic or cyclic codes. Both types of codes are considered in our design. Moreover, novel schemes are developed to keep a small proportion of messages in order to reduce the memory requirement without causing noticeable performance loss. In addition, a shift-message structure is proposed by using memories concatenated with variable node units to enable efficient partial-parallel decoding for cyclic NB-LDPC codes. Compared to previous designs based on the Min-max decoding algorithm, our proposed decoders
have at least tens of times lower complexity with moderate coding gain loss.

The third design is based on a proposed check node decoding scheme using power representation of finite field elements. Novel schemes are proposed for both the Min-max and the simplified Min-sum check node processing by making use of the cyclical-shift property of the power representation of finite field elements. Compared to previous designs based on the Min-max algorithm with forward-backward scheme, the proposed check node units (CNUs) do not need the complex switching network. Moreover, the multiplications of the parity check matrix entries are efficiently incorporated. For a Min-max NB-LDPC decoder over $GF(32)$, the proposed scheme reduces the CNU area by at least 32%, and leads to higher clock frequency. Compared to the prior simplified Min-sum based design, the proposed CNU is more regular, and can achieve good throughput-area tradeoff.
Chapter 1

Introduction

Error-correcting coding has become one integral part in nearly all the modern data transmission and storage systems. Low-density parity-check (LDPC) codes are a class of linear codes which provide near-capacity performance on a large collection of data transmission and storage channel while still maintaining implementable decoders. Due to the powerful error-correcting capability, LDPC codes have been used as error-correcting codes with applications in wireless communications, magnetic and optical recording, and digital television.

Since the first proposition of LDPC codes by Gallager in his 1960 doctoral dissertation, LDPC codes were mostly ignored for the following three decades. One notable exception is the work of Tanner [3] in which a graphical representation of LDPC codes was introduced, now widely called Tanner graph. In mid-nineties, LDPC codes were re-discovered by MacKay, Luby, Wiberg and others [4], [5], [6]. In their works, the sparsity of the parity check matrix of LDPC codes was utilized and belief propagation method proposed by [7] was adopted to significantly reduce the decoding complexity. After that, numerous research efforts have been done and various decoding algorithms are
The knowledge of binary LDPC codes are the basics for understanding the non-binary counterparts. Developing good code construction method, decent decoding algorithms and efficient decoder architectures are three steps that need to be well coordinated.

In the code designing aspect, the coding theorists are seeking for better codes either for improving the error correcting performance or facilitating the decoder design. Particularly, the class of structured codes constructed through algebraic and combinatorial methods [8], [9] and [10] draw much of the attention. The structured codes such as QC-LDPC codes [11], [12] consist of square sub-matrices which can be a permutation or zero matrix. This type of codes are can lead to efficient memory address generation in the decoder. Moreover, the design of partial parallel decoder can largely benefit from the structure of QC-LDPC codes. Meanwhile, the method for evaluating the code performance called density evolution is proposed which has been adopted in the Turbo code analysis before [13]. The capacity of the LDPC code under message passing algorithm can be found in [14]. Usually, the density evolution are for analyzing the performance of the constructed random codes derived through computer search.

Several algorithms have been proposed to reduce the computational complexity of the belief propagation algorithm. The most notable work is the Min-sum algorithm [15]. In the Min-sum algorithm, the complicated hyperbolic function in the check node processing for the belief propagation algorithm
is reduced to simple comparisons with small performance loss. The normalized and offset min-sum algorithms are proposed shortly afterwards to improve the performance of the proposed Min-sum algorithm [16]. The improvement are mainly due to the alleviation of the overestimation in the original Min-sum algorithm. The performance of the two improved Min-sum algorithms for LDPC codes are also analyzed by means of density evolution [17].

There are a versatile of decoders in literatures proposed for binary LDPC targeting on small silicon area, high throughput and efficient power consumption. In general, the LDPC decoder architecture can be categorized into three classes: fully parallel, partially parallel and serial decoder. The serial decoders are seldom adopted due to the low throughput.

For fully parallel decoder, each node in the tanner graph maps to a processing unit. Hence, all the processing units operate in parallel [18] [19]. Although fully parallel decoder can achieve high throughput than the partial parallel ones, they suffer from large silicon area and routing congestion. The latter is the primary obstacle in the design since the large difference in the length of wirings can cause formidable timing issues [20]. Thus, only short-length code can be practically implemented in fully parallel decoder. To alleviate the routing complexity, bit-serial check node processing scheme [21] is proposed. Moreover, several other methods including split row [22] and multi-split [23] are adopted to reduce the wiring complexity between the processing units.

Partial parallel decoder can achieve better area and throughput trade-
off. To improve the throughput, different scheduling schemes are adopted to reduce the waiting time [24], [25] or to break the sequential tie [26]. A memory efficient decoder can be found in [27] in which the extrinsic messages are stored in a compressed version. To adapt to different code length and rate, scalable decoder design are proposed [28], [29]. The designs that take care of error floor can be found in [30], [31]. Moreover, several low-power design can be found in [32], [33].

When the code length is moderate, non-binary low-density parity-check (NB-LDPC) codes can achieve better error-correcting performance than binary LDPC codes at the cost of higher decoding complexity. However, the complicated computations in the check node processing and the large memory requirement put obstacles to efficient hardware implementations. VLSI architecture design connects advanced error-correcting theories and their efficient hardware implementations through algorithmic and architectural level optimizations. Such optimizations are crucial to meet all kinds of implementation requirements set up by hardware applications. In this thesis, efficient VLSI architectures design targeting at non-binary (NB) LDPC decoder are presented. The designs include three decoders based on three proposed decoding algorithms, which are relaxed Min-max algorithm, enhanced iterative hard reliability-based majority-logic decoding algorithm (IHRB-MLGD) and power representation based decoding algorithm.
1.1 Summary of Contributions

1.1.1 Relaxed Min-max Decoding Algorithm and Architecture

In [34], a relaxed Min-max check node processing scheme for NB-LDPC decoding is proposed. By utilizing the property that each element in $GF(2^p)$ can be uniquely represented by a linear combination of $p$ independent field elements, all the entries in a c-to-v message vector can be derived efficiently from a set called minimum basis. The minimum basis contains only $p$ most reliable v-to-c messages with linearly independent field elements. Moreover, relaxations are adopted on which messages can be included in the minimum basis, and novel schemes are developed to derive the minimum basis for each c-to-v message vector from the same check node by slight adjustment. This further reduces the computation complexity and memory requirement, and simulations show that the proposed schemes only lead to negligible performance loss. Efficient VLSI architectures are also developed in this paper for the proposed check node processing and overall NB-LDPC decoder. For a (837, 726) code over $GF(2^5)$, synthesis results show that our decoder can achieve 52% higher efficiency in terms of throughput-over-area ratio than the decoder in [2], which is the most efficient existing design.

1.1.2 Enhanced Iterative Hard Reliability-based Majority-logic Decoding Algorithm and Architectures

In [35] an enhanced (E-)IHRB algorithm is proposed. Through incorporating the probability information from the channel into the message initialization of the IHRB algorithm and excluding the contribution of the
same check node from the variable-to-check (v-to-c) message, the E-IHRB algorithm can bridge the performance gap between the IHRB and ISRB algorithms with small complexity overhead. Novel partial-parallel architectures are also developed in this paper to efficiently implement the IHRB and E-IHRB algorithms through algorithmic and architectural optimizations. Many construction methods of NB-LDPC codes lead to quasi-cyclic (QC) or cyclic codes [36]. For QC codes, our IHRB decoder processes one row of sub-matrices at a time, and the variable node units (VNUs) are implemented by simple logic when all $q$ messages in a vector are kept. Cyclic NB-LDPC codes have the advantage that their encoders can be implemented easily by linear feedback shift registers. However, these cyclic codes usually involve finite fields of high order, in which case keeping all $q$ messages leads to large memory requirement. Novel schemes are developed in this paper to store only a small proportion of the messages without incurring noticeable performance loss. In addition, a shift-message decoder architecture is proposed for cyclic NB-LDPC codes to enable efficient partial-parallel processing. The message shifting is accomplished through concatenating memories with VNUs to reduce the area requirement. It is non-trivial to extend the IHRB decoder architecture to implement the E-IHRB algorithm since recording the messages from check nodes may lead to large memory overhead, especially when the column weight of the code is not small. By making use of the properties of the IHRB algorithm, an innovative approach is proposed in this paper to reverse the contributions from check nodes through remembering only a few extra values for each variable node.
Compared to the Min-max decoder architectures in [1] [37] [38], which are the most efficient existing QCMB-LDPC decoder designs, the proposed E-IHRB decoder architecture has at least tens of times lower complexity with moderate coding gain loss. Parts of this paper on IHRB decoder design have appeared in [39].

1.1.3 Power Representation Based Decoding Algorithm and Architecture

In [40], novel check node processing schemes are proposed for both the forward-backward based Min-max algorithm and the simplified Min-sum algorithm by making use of the power representation of finite field elements. If a vector consists of all nonzero finite field elements ordered according to the exponents of their power representations, then multiplying the vector by a nonzero field element is equivalent to a cyclic shift of the elements in the vector. By making use of this property, the switching network in the forward-backward Min-max CNU can be eliminated through shifting the input and output vectors in the elementary steps of the check node processing. Moreover, an innovative scheme is developed to incorporate the multiplications by the parity check matrix entries into the proposed check node processing. These results for the Min-max algorithm with forward-backward scheme have been presented in [41]. In this extended paper, a novel technique is proposed to reduce the latency overhead caused by shifting the vectors through scaling the rows of the parity check matrix, which does not affect the error correcting performance of the decoding algorithm. Power representation of field elements
is also exploited in this paper to implement the CNU for the SMSA. Compared to the previous SMSA CNU in [42], the proposed design consists of identical hardware units and is more regular. It is also partial-parallel and more suitable to applications with limitations on chip area.

1.2 Outline of the Thesis

This thesis is organized as follows.

Chapter 2 first gives a brief overview of different decoding algorithms for NB-LDPC codes. These decoding algorithms includes belief propagation (BP), FFT-based BP algorithms, mixed domain decoding, extended Min-sum algorithm and the simplified Min-sum algorithm. Next, details of Min-max decoding algorithms are introduced, which includes both forward-backward and trellis-based based check node processing. Moreover, the IHRB-MLGD and ISRB-MLGD algorithms are introduced. This chapter also provides some background knowledge of the parity matrix and Tanner graph. Such knowledge is important for the understanding of the decoding algorithms.

Chapter 3 focused on the the proposed relaxed Min-max algorithm and corresponding decoder architecture. A novel basis-construction method is first proposed for the check node processing. Then the check node unite (CNU) architecture is provided. The proposed overall decoder architecture is presented in the following section. Besides, an interleaving scheduling is proposed. Lastly complexity analysis and comparison results are provided.
Chapter 4 proposes the efficient enhanced IHRB algorithm for NB-LDPC decoding. A novel enhanced (E-)IHRB algorithm is first presented. Both of the IHRB and E-IHRB decoder architectures are presented next. The architectures for quasi-cyclic (QC) and cyclic NB-LDOPC codes are developed also. In addition, the complexity analysis and comparison results are given.

Chapter 5 proposes the decoder designs relying on the power representation of finite field elements. The proposed check node processing schemes and architectures for the forward-backward based Min-max algorithm are first presented. Then a novel technique for reducing the overhead latency is introduced. Next, the CNU architectures for simplified Min-sum algorithm (SMSA) is developed. The complexity comparisons and comparison results are provided in the end.

Chapter 6 provides the conclusion and some ideas for the future research.
2.1 Existing Works for Belief Propagation Algorithm

2.1.1 Belief-propagation Algorithm

LDPC codes can be decoded by the belief propagation (BP) algorithm. In the belief propagation for non-binary codes, the complexity of check node processing dominates. A forward-backward scheme was proposed for BP in [43] to decompose the check node processing into iterative steps. At the cost of larger memory, the number of computations can be significantly reduced. A log-domain variation of the forward-backward scheme was presented in [44].

2.1.2 FFT-based Decoder

The computations for check node processing can be considered as convolutions of the incoming messages, which can be computed as term by term products in the frequency domain. Using this idea, a fast decoding algorithm for LDPC codes over $GF(2^p)$ ($p > 1$) has been proposed in [45]. This algorithm also isolates the effects of the non-binary entries of parity check matrix from check node processing through introducing multiplication/division nodes in the Tanner graph. As a result, the complexity of this algorithm scales with $p2^p$. A variation of this algorithm that uses the logarithm of the channel prob-
ability is presented in [46]. Compared to decoders in the probability domain, log-domain decoders are less sensitive to quantization noise. In addition, multiplications in the probability domain can be converted to additions in the log domain.

2.1.3 Mixed Domain Decoder

However, computing Fourier transform in the log domain is very difficult. To take advantage of both log-domain decoding and simplified convolution computation in the frequency domain, a mixed-domain decoder is proposed in [47]. In this decoder, the Fourier transform is done in the probability domain and the check and variable node processing is carried out in the log domain.

In [47], three serial NB-LDPC decoders, in which the processing of one check or variable node is carried out at a time, have been implemented on FPGA devices. Among these decoders, the one in the mixed domain is the most efficient. However, it can only achieve around 1 Mbps throughput with large hardware resource. In addition, the memory requirement will increase significantly when the order of the finite field becomes higher.

2.2 Extended Min-sum (EMS) Algorithm

2.2.1 Existing works based on EMS algorithm

Despite the simplified check node processing, frequency-domain decoders demand a large number of multiplications and extra complexity for
the Fourier and inverse transforms. In addition, mixed-domain decoders require large memory to implement the look-up tables (LUTs) that are needed for the conversions between the domains. On the other hand, the extended Min-sum algorithm (EMS) [48, 49], which is an approximation of the belief propagation in the log domain and an extension of the Min-sum algorithm for binary LDPC codes [50], requires only additions in the check node processing. With proper scaling or offsetting, the EMS algorithm only leads to less than 0.1dB coding gain loss compared to the belief propagation. To reduce the memory requirement, it was also proposed in [48, 49] to keep \( n_m < q \) messages on each edge of the Tanner graph at the cost of small performance degradation.

Implementation architectures for the check and variable node processing with \( n_m < q \) messages based on the EMS algorithm have been proposed in [51]. This paper considers either serial or fully-parallel design. In order to reduce the latency of computing the minimum of sums, the messages need to be kept sorted. Parallel sorters that are capable of inserting a value into a sorted vector of length \( n_m \) in one clock cycle may lead to large area requirement, especially when multiple check or variable node units need to be implemented.

### 2.2.2 Dataflow of EMS algorithm

For each received symbol of NB-LDPC codes, the corresponding transmitted symbol can be any of the \( GF(q) \) elements. Hence, the messages passed through the edges of the Tanner graph are vectors of size \( q \) instead of single values as in the binary case. A generalization of the belief propagation
for NB-LDPC codes can be found in [43]. Compared to probability-domain decoders, log-domain decoders are less sensitive to quantization noise. In addition, the log-domain belief propagation can be approximated by the EMS [49] and Min-max [52] algorithms with small performance degradation. In the EMS, the reliability vector for a variable \( z \) consists of \( q \) log-likelihood ratios (LLRs): 

\[
L(\alpha) = \log\left(\frac{P(z = \beta)}{P(z = \alpha)}\right),
\]

where \( \alpha \) is an element of \( GF(q) \) and \( \beta \) is the most likely symbol for \( z \). Using this definition, all LLRs are non-negative and they can be used as metrics to measure how far away the symbols are from the most likely symbol. The smaller the \( L(\alpha) \), the more likely that the variable \( z = \alpha \). Denote the reliability vector from check node \( m \) to variable node \( n \) by \( v_{m,n} \), and that from variable node \( n \) to check node \( m \) by \( u_{m,n} \). Let \( S_c(n) \) be the set of check nodes connected to variable node \( n \), and \( S_v(m) \) represent the set of variable nodes connected to check node \( m \). Express by \( \mathcal{L}(m|a_n = \alpha) \) the set of sequences of finite field elements \( (a_j) (j \in S_v(m) \setminus n) \) such that

\[
\sum_{j \in S_v(m) \setminus n} h_{m,j} a_j = h_{m,n} \alpha.
\]

Here \( h_{i,j} \) is the entry of the \( H \) matrix in the \( ith \) row and \( jth \) column. Assume that the LLR vector from the channel for variable node \( n \) is \( \delta_n \). The EMS decoding can be carried out according to Algorithm A.
Algorithm A: The EMS Algorithm

Initialization: \( u_{m,n}(\alpha) = \gamma_n(\alpha) \)

Iterations:

- **Check node processing**

  \[
  v_{m,n}(\alpha) = \min_{(a_j) \in \mathcal{L}(m|a_n = \alpha)} \left( \sum_{j \in \mathcal{S}_v(m) \setminus n} u_{m,j}(a_j) \right)
  \]

- **Variable node processing**

  \[
  u'_{m,n}(\alpha) = \tau_n(\alpha) + \sum_{i \in \mathcal{S}_c(n) \setminus m} v_{i,n}(\alpha)
  \]

  \[
  u_{m,n}(\alpha) = u'_{m,n}(\alpha) - u'_{m,n}(\omega) \tag{2.1}
  \]

- **A posteriori information computation**

  \[
  \tilde{\tau}_n(\alpha) = \tau_n(\alpha) + \sum_{i \in \mathcal{S}_c(n)} v_{i,n}(\alpha)
  \]

2.2.3 Simplified Min-sum algorithm

The Simplified Min-sum algorithm proposed in [42] relaxes the constraint of the check node processing. From the definition of the set \( \mathcal{L}(m|a_n = \beta) \), the LLRs from distinct variable-to-check (v-to-c) message vectors are added up to compute \( v_{m,n}(\beta) \) in the EMS algorithm. In the SMSA [42, 53], a relaxation is made such that multiple LLRs from the same v-to-c vector can be included in the addition. Also each finite field element in a vector is added up
with the most likely element in the vector, so that the zero field element in
the transformed vector is always the most reliable and has zero LLR [42]. As
a result, the check node processing can be simplified as

\[ v_{m,n}(\beta) = \min_{\beta = \oplus \sum_{l=1}^{K} a_l} \left( \sum_{i \in S_v(m) \setminus n} \min_{l=1}^{k} u_{m,i}(a_l) \right), 1 \leq k \leq K \] (2.2)

The computations in (2.2) are repeated for \( k = 1, 2, \ldots, K \), and the minimum
sum from all possible \( k \) is taken as \( v_{m,n}(\beta) \). \( K \) decides the maximum number
of nonzero LLRs that are allowed to participate in the summation, and is
referred to as the order of the SMSA. Apparently, larger \( K \) leads to better
error correcting performance at the expense of higher computational cost.

### 2.3 Min-max Algorithm

#### 2.3.1 Existing works based on Min-max algorithm

The complexity of the check node processing is further reduced in the
Min-max algorithm [52] with slightly lower coding gain. In this algorithm,
'max' instead of 'sum' is carried out in the check node processing. In [54]
the Min-max check node processing architecture is proposed by keeping all
q messages, and hence also suffers from large memory requirement when the
order of the finite field is not small. In addition, this architecture has very
long latency since it computes all possible combinations of field elements from
each message vector. A parallel Min-max check node processing architecture
was developed in [1]. Although it can achieve high speed, it requires large area
due to the high level of parallel processing. This architecture also becomes less
efficient when $q$ is larger or the code rate is lower. Two decoding algorithms and architectures have been proposed for the Min-max algorithm [37] and [38]. These two approaches significantly reduces the decoding complexity.

### 2.3.2 Dataflow of Min-max algorithm

The Min-max decoding can be carried out according to Algorithm B.

---

**Algorithm B: the Min-max Decoding Algorithm**

**Initialization:** $u_{m,n}(\alpha) = \delta_n(\alpha)$

**Iterations:**

- **Check node processing**

\[
v_{m,n}(\alpha) = \min_{(a_j)\in L(m|a_n=\alpha)} \left( \max_{j\in S_v(m)\setminus n} u_{m,j}(a_j) \right)
\]  

(2.3)

- **Variable node processing**

\[
u'_{m,n}(\alpha) = \delta_n(\alpha) + \sum_{i\in S_c(n)\setminus m} v_{i,n}(\alpha)
\]  

(2.4)

\[
u_{m,n}(\alpha) = u'_{m,n}(\alpha) - \min_{\omega \in GF(q)} (u'_{m,n}(\omega))
\]

- **A posteriori information computation**

\[
\tilde{\delta}_n(\alpha) = \delta_n(\alpha) + \sum_{i\in S_v(n)} v_{i,n}(\alpha)
\]  

(2.5)
In the Min-max algorithm, the normalization in the variable node processing is necessary to ensure numerical stability and maintain that the smallest LLR in a vector is always zero. After each iteration, hard decision for the $ith$ symbol can be made as $r_i = \arg\min_{\alpha} \tilde{\delta}_i(\alpha)$. The iterations can be carried out until $H[r_0, r_1, r_2, \cdots]^T = 0$ or the maximum iteration number has been reached. The Min-max algorithm is suboptimal since the check node processing in (2.3) underestimates the LLRs. To compensate for the underestimation, scaling or offsetting can be applied to the sums of the check-to-variable (c-to-v) LLRs in (2.4) and (2.5) before they are added up with the channel LLRs.

2.3.3 Boundary tracking based check node processing for Min-max algorithm

In [37], a boundary tracking based check node processing is proposed for Min-max algorithm. Employing the forward-backward scheme, the elementary step of the CNU is to compute an output message vector from two input message vectors. As mentioned previously, a message vector consists of two parts: LLRs and corresponding finite field elements. Denote the input vectors by row and column vectors. The corresponding LLR vectors are $V_r = [V_r(0), V_r(1), \cdots, V_r(n_m - 1)]$ and $V_c = [V_c(0), V_c(1), \cdots, V_c(n_m - 1)]$, and the corresponding finite field element vectors are $a_{V_r} = [a_{V_r}(0), a_{V_r}(1), \cdots, a_{V_r}(n_m - 1)]$ and $a_{V_c} = [a_{V_c}(0), a_{V_c}(1), \cdots, a_{V_c}(n_m - 1)]$. When the length of the vectors is limited to $n_m$, the entries in the output LLR vector for the Min-max decoding are the $n_m$ minimum values of $\max(V_r(i), V_c(j))$ with different $a_{V_r}(i) + a_{V_c}(j)$ for any combination of $i$ and $j$ less than $n_m$. Denote the output LLR and cor-
responding finite field element vectors by \( V_o \) and \( a_{V_o} \), respectively. Represent storing a LLR \( L \) and its associated field element \( a_L \) into the output vector by \( \{ V_o, a_{V_o} \} \leftarrow \{ L, a_L \} \). The output vector of the elementary step in the Min-max-based CNU can be derived according to Algorithm C, since the messages in a vector are stored in the order of increasing LLR.

Since the minimum of 'max' LLRs need to be kept, the comparisons of the LLRs start from the first entries in the row and column vectors. In the case that \( V_r(i) < V_c(j) \) happens, \( V_r(i) \) was the 'max' value when compared with any previous \( V_r(k) \) with \( 0 \leq k < j \), since the LLRs in each vector are sorted. Hence, \( V_r(i) \) should be inserted into the output vector together with field element \( a_{V_r}(i) + a_{V_c}(k) \) for each \( 0 \leq k < j \). In addition, if \( a_{V_r}(i) + a_{V_c}(k) \) coincides with the field element of a previously inserted LLR, it should not be inserted again since the previous LLR is smaller. After this process is completed, the next entry for the row vector is read out in the next clock cycle to be compared with \( V_c(j) \). Similarly, in the case that \( V_r(i) \geq V_c(j) \), \( V_c(j) \) will be inserted into the output vector with field element \( a_{V_r}(k) + a_{V_c}(j) \) for each \( 0 \leq k < i \).

2.3.4 Path construction based check node processing for Min-max algorithm

[38], a novel check node processing method and corresponding efficient architectures for the Min-max NB-LDPC decoding are proposed. Employing
Algorithm C: the Elementary Step of CNU

Initialization: $i = 0$, $j = 1$, $m = 0$;
$\text{num}=$maximum allowable iteration number

loop:
if $V_r(i) < V_c(j)$
for $k = 0$ to $j - 1$
{  
  $m = m + 1$;
  if $m = \text{num}$, goto stop
  if $(a_{V_r}(i) + a_{V_c}(k)) \notin a_{V_o}$
    \{$V_o, a_{V_o}$} ← \{$V_r(i), a_{V_r}(i) + a_{V_c}(k)$\}
}
i = i + 1; goto loop
else
for $k = 0$ to $i - 1$
{  
  $m = m + 1$;
  if $m = \text{num}$, goto stop
  if $(a_{V_r}(k) + a_{V_c}(j)) \notin a_{V_o}$
    \{$V_o, a_{V_o}$} ← \{$V_c(j), a_{V_r}(k) + a_{V_c}(j)$\}
}
j = j + 1; goto loop

stop:
the Min-max algorithm, each c-to-v probability from a check node equals one of only a few most reliable v-to-c probabilities to this node. Based on this observation, a limited number of the most reliable v-to-c messages are first sorted out. These messages can be considered as nodes in a trellis. Then the c-to-v messages to all connected variable nodes are generated independently using a path construction scheme without storing other intermediate values. As a result, both the memory requirement and logic gate count can be reduced significantly without sacrificing the speed. Compared to the forward-backward check node unit (CNU) architecture in [37], the proposed architecture can reduce the memory and area requirements to 19.5% and 22.5%, respectively, when applied to a (837, 726) NB-LDPC code over $GF(2^5)$.

It is a challenging task to compute $v_{m,n}$ from the corresponding most reliable v-to-c messages, especially when only the $n_m$ most reliable entries of $v_{m,n}$ are needed, and it is desired not to waste computations on the rest unreliable entries. In this subsection, a novel path construction scheme is presented to efficiently derive the smallest LLRs in $v_{m,n}$ from the corresponding sorted smallest v-to-c LLRs.

For a check node, the messages from the $d_c$ connected variable nodes can be represented by a trellis of $d_c$ stages. Each stage has $n_m$ nodes, representing the $n_m$ entries in the corresponding v-to-c message vector. Fig. 2.1 shows such a trellis. Note that the nodes in a stage are ordered according to increasing LLR, and the normalization in (2.1) guarantees that the nodes in the first row have zero LLR. In the computation of $v_{m,n}$, a field element sequence $(a_j)$
corresponds to a path in the trellis that is formed by connecting the nodes with $a_j$ in the $j$th stage. Such a path passes exactly one node in each stage, except the stage for variable node $n$. Define the LLR and finite field element of a path the maximum of the LLRs and sum of finite field elements, respectively, of all nodes in the path. Accordingly, computing the $n_m$ entries of $v_{m,n}$ with the smallest LLRs is equivalent to finding the $n_m$ paths with the smallest LLRs and different finite field elements that pass exactly one node in each stage, except the stage for variable node $n$.

Apparently, the path that passes only the zero-LLR nodes in the first row of the trellis is the path with the smallest LLR. This path is denoted by path 0 in Fig 2.1. Assume that all $d_c \times (n_m - 1)$ nonzero LLRs of the trellis can be sorted in non-decreasing order as $x(1), x(2), \cdots$. Their associated field elements are $\alpha(1), \alpha(2), \cdots$, and they belong to variable nodes with indices $e(1), e(2), \cdots$. The other paths for $v_{m,n}$ computation can be constructed iteratively by changing nodes in path 0 to nonzero-LLR nodes with $e(i) \neq n$.  

Figure 2.1: Trellis of variable-to-check messages
To ensure that the paths with the smallest LLRs are constructed first, so that no computation is wasted on calculating the unreliable c-to-v messages that will not be stored, the nonzero-LLR nodes are included in the path construction gradually according to increasing LLR. In another word, the node corresponding to $x(i)$ will be considered for path construction before the node corresponding to $x(i + 1)$. When a node with $x(i)$ and $e(i) \neq n$ is considered, the node in the same stage in each previously constructed path will be replaced by this node. In this way, all possible paths can be constructed. However, only one node from each stage can be included in a path. Hence, replacing a nonzero-LLR node in the same stage of a previous path would only lead to an identical path that can be constructed by replacing the zero-LLR node in that stage of another previous path. Therefore, if a previous path has a nonzero-LLR node in stage $e(i)$, there is no need to construct another path by replacing that nonzero-LLR node with the node corresponding to $x(i)$. If a newly constructed path has a field element that is different from those of all previous paths, $x(i)$ and $\alpha(i)$ will be stored as a c-to-v message. Such a process can be repeated until $n_m$ entries are derived for the c-to-v message vector.

The following example can help explain the path construction process. Denote the nodes in the $i$th row and $j$th column of the trellis by $o_{i,j}$ ($0 \leq i < n_m, 0 \leq j < d_c$). Assume $e(1), e(2), e(3), \cdots = 1, 0, 0, \cdots$. Accordingly, the nodes corresponding to $x(1), x(2), x(3), \cdots$ are $o_{1,1}, o_{1,0}, o_{2,0}, \cdots$, respectively. Without loss of generality, consider the path construction for
v_{m,n_{d_c-1}} computation. Since \( e(1) \neq d_c - 1 \), \( o_{1,1} \) can replace the node of the same stage in path 0 to form path 1 as shown in Fig. 2.1. Two paths have been constructed so far. Next, the node \( o_{1,0} \) can be included in the path construction since \( e(2) \neq d_c - 1 \). Replacing the node of stage 0 in each of path 0 and 1 by \( o_{1,0} \), another two paths, path 2 and 3, can be constructed. Now, there are four valid paths, and \( o_{2,0} \) can be considered next since \( e(3) \neq d_c - 1 \). Similarly, another two new paths can be constructed by replacing the node of stage 0 in path 0 and 1 by \( o_{2,0} \). These two new paths are denoted by path 4 and 5 in Fig. 2.1. However, both path 2 and 3 have a nonzero-LLR node, \( o_{1,0} \), in stage 0. Replacing \( o_{1,0} \) in path 2 and 3 by \( o_{2,0} \) would lead to the same path 4 and 5, respectively. Hence, the path construction of replacing the node of the same stage as \( o_{2,0} \) in path 2 and 3 can be skipped.

In order to keep track of the paths, path \( i \) is associated with a binary vector \( P_i \) of \( d_c \) bits. If path \( i \) passes a nonzero-LLR node in the \( j \)th stage, then the \( j \)th bit of \( P_i \), denoted by \( P_i(j) \), is '1'. Otherwise, \( P_i(j) = 0 \). Using these vectors, whether a path already has a nonzero-LLR node in a certain stage can be easily tested. Moreover, if path \( j \) is constructed by replacing a zero-LLR node in path \( i \), then the field element of path \( j \) can be computed from that of path \( i \) by taking care of the difference. Represent the field element of the zero-LLR node in stage \( j \) by \( z(j) \), and define \( \alpha_{sum} = \sum_{0 \leq t < d_c} z(t) \). Denote the c-to-v LLR and field element vectors for variable node \( n \) by \( L_n \) and \( \alpha_{L_n} \), respectively. \( L_n \) and \( \alpha_{L_n} \) can be computed by Algorithm D.
Algorithm D: The Path Construction Algorithm

Initialization: \( L_n(0) = 0, \alpha_{L_n}(0) = \alpha_{sum} + z(n) \)
\[ i = 1, \ \text{cnt} = 1, \ P_0 = [0, 0, \cdots, 0] \]

loop: if \( \text{cnt} < n_m \)
\[ \text{if } e(i) \neq n \]
\[ j = \text{cnt} \]
\[ \text{for } k = 0 \ \text{to } j - 1 \]
\[ \alpha = \alpha_{L_n}(k) \oplus z(e(i)) \oplus \alpha(i) \]
\[ \text{if } (P_k(e(i)) \neq 1) \ \& \ (\alpha \notin \alpha_{L_n}) \]
\[ L_n(\text{cnt}) = x(i); \ \alpha_{L_n}(\text{cnt}) = \alpha \]
\[ P_{\text{cnt}}(e(i)) = 1 \]
\[ P_{\text{cnt}}(t) = P_k(t) \ \text{for } t \neq e(i) \]
\[ \text{cnt} = \text{cnt} + 1 \]
\[ i = i + 1; \ \text{goto loop} \]

2.4 Error Correcting Performance of Extended Min-sum and Min-max algorithm

Compared to the EMS algorithm, \( \sum_{j \in S_v(m) \setminus n} u_{m,j}(a_j) \) is replaced by \( \max_{j \in S_v(m) \setminus n} u_{m,j}(a_j) \) in (2.3). As a result, the Min-max algorithm can be implemented by a more efficient architecture as will be shown in later chapters of the thesis. On the other hand, the performance of the Min-max algorithm is only slightly worse. Fig. 2.2 and 2.3 show respectively the frame error rates (FERs) and bit error rates (BERs) of the EMS and Min-max algorithms for a \((744, 653)\) QCNB-LDPC code over \(GF(2^5)\) with 15 decoding iterations under the additive white Gaussian noise (AWGN) channel. This code is constructed using the method based on the multiplicative group of a finite field proposed in [55]. In this construction, a \( r \times t \) sub-matrix is first picked from a cyclic matrix whose first row is \((0, \alpha - 1, \cdots, \alpha^{q-2} - 1)\). Here \( \alpha \) is a primitive element.
of $GF(q)$, and $r \leq q$ and $t < q$ depend on the length and rate of the code needs to be constructed. Then the $H$ matrix is formed by dispersing each element in the picked sub-matrix to a matrix of dimension $(q-1) \times (q-1)$. The first row of the matrix resulted from dispersing an element $\alpha^i$ is all zero, except that the $i$th entry is $\alpha^i$. Each of the other rows is a right cyclic-shift of the previous row multiplied by $\alpha$. For the $(744, 653)$ code construction, $r = 24$ and $t = 3$ are used. As it can be observed from Fig. 2.2, the Min-max algorithm has only around 0.08dB performance degradation at FER=$10^{-3}$ compared to the EMS algorithm for the $(744, 653)$ code. In the decoding of NB-LDPC codes, a vector of $q$ messages needs to be recorded for each edge of the Tanner graph. This leads to vast memory requirement, especially when the order of the finite field is large. In order to reduce the memory requirement, it was proposed in [49] to keep only the $n_m < q$ most reliable messages for each vector in the
Figure 2.3: BERs of a (744, 653) NB-LDPC code over $GF(2^5)$ and a (3683, 3175) binary LDPC code under AWGN channel

EMS algorithm. A similar idea can be applied to the Min-max algorithm: keeping only the $n_m < q$ messages with the smallest LLRs in each vector. As it can be observed from Fig. 2.2, keeping only $n_m = 16$ messages for the (744, 653) code constructed over $GF(2^5)$ only leads to around 0.045dB performance loss in the Min-max algorithm. For reference, Fig. 2.2 and 2.3 also show the performance of a binary (3683, 3175) code constructed with the method in [56] using the Min-sum decoding. This binary code has similar code rate and code word length in terms of bits as the (744, 653) NB-LDPC code over $GF(2^5)$.

To show the effect of the Min-max decoding and keeping $n_m < q$ messages on a different code, Fig. 2.4 and 2.5 illustrate the performance of a (248, 124) code over $GF(2^5)$. This code has much shorter length and lower rate than the (744, 653) code. As it can be observed, using the Min-max decoding and keeping $n_m$ messages also only lead to very small coding gain loss for this
Figure 2.4: FERs of a (248, 124) NB-LDPC code over $GF(2^5)$ and a (1270, 635) binary LDPC code under AWGN channel code. For reference, the performance of the Min-sum decoding for a (1270, 635) binary LDPC code is shown in Fig. 2.4 and 2.5. These two codes are also constructed using the methods in [55] and [56].

Fig. 2.6 shows some simulation results of the frame error rates (FERs) of the Min-max algorithm for a (837,726) QC NB-LDPC codes over $GF(2^5)$ using layered decoding with 15 iterations and $n_m = 16$ under the additive white Gaussian noise (AWGN) channel. This code is constructed based on Reed-Solomon codes with two information symbols [57]. The curves labeled ‘FB’ use the forward-backward check node processing, and the ones labeled ‘trellis’ use the proposed check node processing based on path construction with the maximum loop number set to $2n_m$ in Algorithm D. It can be observed from Fig. 2.6 that the proposed scheme has the same performance as the forward-backward check node processing. In addition, using $w = 5$ only leads
2.5 The IHRB-MLGD and ISRB-MLGD Algorithm

In [58] and [59], novel partial-parallel architectures are developed to efficiently implement the IHRB-MLGD and ISRB-MLGD algorithm through algorithmic and architectural optimizations.

In the IHRB algorithm, the reliability measures of the received symbols are updated in each iteration based on the hard-decision symbols. Throughout this paper, the superscript \((k)\) is added to denote the values in the \(kth\) decoding iteration whenever necessary. In addition, a vector of variables with subscripts is also represented by the same variable with deleted subscript. For a NB-
Figure 2.6: FER of a (837, 726) QCNB-LDPC code over $GF(2^5)$ with 15 iterations and $n_m = 16$

LDPC code of length $n$, let $\mathbf{z}^{(k)} = [z_0^{(k)}, z_1^{(k)}, \ldots, z_{n-1}^{(k)}]$ be the hard decision vector of the received symbols in the $k$th decoding iteration, and $\mathbf{z}^{(0)}$ consists of the hard decisions made from the channel output. $R_j^{(k)} = [R_{j,0}^{(k)}, R_{j,1}^{(k)}, \ldots, R_{j,q-1}^{(k)}]$ is the reliability measure vector of the $j$th received symbol, representing the probabilities that the $j$th received symbol equals each field element. Assume that the $H$ matrix has $m$ rows. Define $N_i = \{j : 0 \leq j < n, h_{i,j} \neq 0\}$, $M_j = \{i : 0 \leq i < m, h_{i,j} \neq 0\}$, and the set $\{\alpha_0, \alpha_1, \ldots, \alpha_{q-1}\}$ consists of all elements of $GF(q)$. The IHRB algorithm can be described by Algorithm E [60].

In Algorithm E, the reliability measures, $R_{j,l}^{(k)}$, are hard-initialized as either $\gamma$ or zero. $\gamma$ is a positive integer, and can be set to the maximum value allowed for the reliability measures. When $R_{j,l}^{(k+1)}$ becomes larger than $\gamma$, the
**Algorithm E: IHRB-MLGD algorithm**

*Initialization:* $R^{(0)}_{j,l} = \gamma$ if $z^{(0)}_j = \alpha_l$; $R^{(0)}_{j,l} = 0$ if $z^{(0)}_j \neq \alpha_l$

for $k = 0 : I_{max}$

E1: Stop if $z^{(k)}H^T = 0$

for $i=0$ to $m-1$

for $j \in N_i$

E2: $\sigma_{i,j} = h_{i,j}^{-1} \sum_{u \in N_i \setminus j} z^{(k)}_u h_{i,u}$

E3: if $\sigma_{i,j} = \alpha_l$

\[ R^{(k)}_{j,l} = R^{(k)}_{j,l} + 1 \]

\[ R^{(k+1)}_{j,l} = R^{(k)}_{j,l} \]

for $j=0$ to $n-1$

E4: $z^{(k+1)}_j = \text{field element of max}_l R^{(k+1)}_{j,l}$

"clipping" method in [60] is applied to subtract the same value from each reliability measure in a vector, so that the largest measure remains $\gamma$. For those reliability measures become negative after the subtraction, zero is used instead. Accordingly, the reliability measures will remain as integers in the range of $[0, \gamma]$. Moreover, $\sigma_{i,j}$ is the extrinsic check sum from check node $i$ to variable node $j$. The decoding can be stopped when a codeword is found, or the maximum decoding iteration number, $I_{max}$, is reached.

In the ISRB algorithm, the reliability measures are initialized according to the probability information from the channel as $R^{(0)}_{j,l} = \varphi_{j,l} \mu$ ($0 \leq j < n$, $0 \leq l < q$), where $\varphi_{j,l}$ is the probability that the $jth$ received symbol equals $\alpha_l$ and $\mu$ is a scaling factor that can be derived from simulations to optimize the error-correcting performance. Assume that $\alpha_l$ is an element of $GF(2^p)$, and hence can be represented by a $p$-bit binary tuple $(b_{l,0}, b_{l,1}, \cdots, b_{l,p-1})$. In addition, BPSK modulation is adopted and the log-likelihood ratio of a bit $b$ at the
receiver is \( LLR(b) = \ln(P(b = 1)/P(b = 0)) \). Then \( \varphi_{j,t} \) can be computed as \( \sum_{0 \leq i < p}(b_{i,t}LLR(b_{i,t})) \), which is actually normalized to the probability of the zero field element for the \( jth \) received symbol. Compared to the IHRB algorithm, another difference in the ISRB algorithm is that the updating of \( R_{j,t} \) by adding one in the A3 step of Algorithm E is replaced by adding soft information \( \phi_{i,j} = \min_{u \in N_i \setminus j} \max_{l} \varphi_{u,l} \).

### 2.6 Error Correcting Performance of IHRB-MLGD and ISRB-MLGD Algorithm

Fig. 2.7 shows the word error rates (WERs) of NB-LDPC decoding algorithms for a (255, 175) cyclic code over \( GF(2^8) \) with \( d_c = d_v = 16 \) constructed based on Euclidean geometry (EG)[36]. The last numbers in the legends are the maximum iteration numbers. In our simulations for the IHRB algorithm, the constant \( \gamma \) takes the format of \( 2^w - 1 \) (\( w \in \mathbb{Z}^+ \)) in order to make full use of the information that can be represented by \( w \) bits. Table 4.1 lists the word lengths, \( w \), used for the reliability measures to derive the simulation curves in Fig. 2.7. Simulations using other word lengths have also been carried out. For the purpose of conciseness, the results are not included in this paper. However, it was found that more noticeable performance loss will be resulted if word lengths shorter than those listed in Table 4.1 are used. As it can be observed from Fig. 2.7, the IHRB and ISRB algorithms can achieve at least 2.5dB and 3.5dB, respectively, coding gain over the one-step (OS) MLGD. In addition, both of these two algorithms converge very fast for this cyclic EG-


LDPC code. In the legends of Fig. 2.7, \( n_m \) denotes the number of messages kept in each vector. For the NB-LDPC decoding curves without specified \( n_m \), all \( q \) messages are kept. It can be seen from Fig. 2.7 that keeping the \( n_m = 16 \) most reliability messages instead of all \( q = 256 \) messages for each vector does not lead to noticeable performance loss. Moreover, the ISRB algorithm can achieve almost the same performance as the Min-max algorithm for this code with high column weight. Fig. 2.8 shows the WERs of NB-LDPC decoding algorithms for a (403, 226) QCNB-LDPC code over \( GF(2^5) \) with \( d_v = 8 \) and \( d_c = 13 \) constructed based on the multiplicative group of finite fields [36]. The same word lengths as listed in Table 4.1 are used, and similar coding gain can be observed for the IHRB and ISRB algorithms over the OSMLGD. However, the ISRB algorithm can not achieve similar performance as the Min-max algorithm since this code has smaller \( d_v \).

![Figure 2.7: WERs of NB-LDPC decoding for a (255, 175) cyclic EG code over \( GF(2^8) \)](image-url)
Fig. 2.7 shows the word error rates (WERs) of NB-LDPC decoding algorithms for a (255, 175) cyclic code over $GF(2^8)$ with $d_c = d_v = 16$ constructed based on Euclidean geometry (EG)\cite{36}. The last numbers in the legends are the maximum iteration numbers. In our simulations for the IHRB algorithm, the constant $\gamma$ takes the format of $2^w - 1$ ($w \in \mathbb{Z}^+$) in order to make full use of the information that can be represented by $w$ bits.

Although the ISRB algorithm can achieve better performance than the IHRB algorithm, it has much higher hardware complexity due to two main reasons: i) soft information is accumulated to the reliability measures $R_{j,l}^{(k)}$ in the ISRB algorithm. Hence $R_{j,l}^{(k)}$ need significantly longer word length than that in the IHRB algorithm, as evident from Table 4.1. The longer word length leads to not only longer critical path in the computation, but also larger memory requirement, which accounts for the majority of NB-LDPC decoder area; ii) the clipping in the ISRB algorithm has much longer critical path
and more complicated logic since each $\phi_{i,j}$ can increase the reliability measure by multiple possible values as compared to only one in the IHRB algorithm. Because of these drawbacks, the architecture design for the ISRB algorithm is not further pursued.

2.7 Layered Decoding

Layered decoding [61], [62], [63] has been widely adopted to reduce the memory requirement and increase the convergence speed of binary LDPC decoding. In this scheme, the parity check matrix is divided into block rows, also called layers, and the decoding is applied to one layer at a time. The c-to-v messages derived from the decoding of one layer is used right away to update the v-to-c messages in the decoding of the next layer. This technique can be extended to NB-LDPC decoding. Both Algorithm A and B take advantage of the layered decoding. We studied the effects of layered decoding on QCNR-LDPC codes using the Min-max algorithm. It can be observed from Fig. 2.2 that the layered decoding can lead to around 0.03 dB coding gain compared to the non-layered case for the (744, 653) code when $n_m = 16$ messages are kept and the maximum iteration number is set to 15. In the case that all $q$ messages are kept, the resulted coding gain is slightly higher.
Chapter 3

Relaxed Min-max Decoding Algorithm and Architecture

3.1 Introduction

In this chapter, a relaxed Min-max check node processing scheme for NB-LDPC decoding is proposed. By utilizing the property that each element in $GF(2^p)$ can be uniquely represented by a linear combination of $p$ independent field elements, all the entries in a c-to-v message vector can be derived efficiently from a set called minimum basis. The minimum basis contains only $p$ most reliable v-to-c messages with linearly independent field elements. Moreover, relaxations are adopted on which messages can be included in the minimum basis, and novel schemes are developed to derive the minimum basis for each c-to-v message vector from the same check node by slight adjustment. This further reduces the computation complexity and memory requirement, and simulations show that the proposed schemes only lead to negligible performance loss. Efficient VLSI architectures are also developed in this chapter for the proposed check node processing and overall NB-LDPC decoder. For a (837, 726) code over $GF(2^5)$, synthesis results show that our decoder can achieve 52% higher efficiency in terms of throughput-over-area ratio than the decoder in [42], which is the most efficient existing design.
3.2 Relaxed Min-max Check Node Processing

For a check node, the message vectors from the $d_c$ connected variable nodes can be represented by a trellis of $d_c$ stages. Each stage has $q$ nodes, representing the $q$ entries in the corresponding v-to-c message vector. Fig. 3.1(a) shows such a trellis for $d_c = 6$ and $q = 4$. To simplify the notation, the v-to-c (c-to-v) message vector $u_{m,n} (v_{m,n})$ for one check node is denoted by $u_j (v_j)$ ($0 \leq j < d_c$). Moreover, the nodes in a stage are placed according to their corresponding finite field elements in polynomial representation in the order of $0, 1, 2, \ldots, q-1$. Using this trellis representation of the v-to-c messages, the computations of the c-to-v messages in $v_j$ are equivalent to finding the paths that pass exactly one node in each stage, except the $j$th stage. Define the LLR and field element of the path as the maximum of the LLRs and sum of the field elements, respectively, of the nodes in the path. Several paths corresponding to the computation of $v_{d_c-1}$ are shown in Fig. 3.1(a) as examples. Moreover, the LLRs and field elements of the paths are also labeled, and '⊕' means exclusive-OR logic operation. Among the paths whose field element equals $\alpha$, the one with the minimum LLR is called the optimal path for $v_j(\alpha)$ in this chapter. Apparently, $v_j(\alpha)$ equals the LLR of the corresponding optimal path [38].

To simplify the check node processing, the v-to-c message vector reordering technique proposed in [42] can be adopted. Assume that $\hat{\alpha}_j$ is the field element corresponding to the most reliable message in $u_j$. Then $u_j$ is re-ordered as $\tilde{u}_j$ such that $\tilde{u}_j(\beta) = u_j(\alpha)$ if $\beta = \hat{\alpha}_j \oplus \alpha$. After re-ordering, the
messages in $\tilde{u}_j$ are still arranged according to the field elements in polynomial representation in the order of $0, 1, 2, \ldots, q-1$. Fig. 3.1(b) shows the reordered v-to-c message trellis assuming that the $\hat{\alpha}_j$ for stage $0, 1, 2, 3, 4$ are $2, 1, 3, 0, 1$, respectively. After the c-to-v messages are computed, reverse reordering will be applied. Due to the normalization in (2.1), $\tilde{u}_j(0) = 0$, and hence all the nodes in the first row of the re-ordered v-to-c message trellis have zero field element and zero LLR. As a result, path 0 in Fig. 3.1(b) is the optimal path for $\tilde{v}_j(0)$, and $\tilde{v}_j(0)$ is always zero. Moreover, the computation of $\tilde{v}_j(\beta)$ ($\beta \neq 0$) can be simplified. Note that the optimal path for $\tilde{v}_j(\beta)$ ($\beta \neq 0$) contains at least one node that has nonzero LLR and nonzero field element. Since zero field element and zero LLR do not contribute to the field element and LLR of

Figure 3.1: Trellises of variable-to-check messages: (a) original trellis; (b) reordered trellis
the path, the nodes in the first row of the reordered trellis can be excluded from constructing the paths for $\tilde{v}_j(\beta)$ ($\beta \neq 0$) computation. The reordered trellis without the first row as shown in the dashed rectangle in Fig. 3.1(b) is called the modified trellis in this chapter. Since the paths with the minimum LLRs are the optimal paths corresponding to the c-to-v messages, zero-LLR nodes constitute a major part of all the optimal paths when $d_c$ is not small. Accordingly, the c-to-v messages can be derived from only a few nodes in the modified trellis with the smallest nonzero LLR.

3.2.1 Relaxed Min-max Check Node Processing

The original Min-max algorithm implicitly requires that the nodes in a path should come from different stages of the trellis. Inspired by the work in [42], this requirement is relaxed in our design, and two or more nodes in one path can come from the same stage. In another word, any set of the nodes in the modified trellis, except those in the $j$th stage, with the sum of their field elements equal to $\beta$ can form a candidate path for $\tilde{v}_j(\beta)$ computation. In the case that two or more nodes from the same stage are included in a path, the second and other nodes can be considered as nodes with the same field elements from other stages. The nodes with the same field elements in other stages may have different LLRs, either larger or smaller. Hence, including more than one nodes from the same stage in a path may lead to over or under estimation of the path LLR. The original Min-max algorithm is already an approximation of BP, and this over or under estimation will either compensate
or deviate the error value caused by the Min-max approximation. As a result, including multiple nodes from the same stage in a path only leads to negligible performance loss as will be shown later in this chapter.

Making use of the relaxation, a novel scheme is proposed next to greatly reduce the complexity of the Min-max check node processing. Without loss of generality, consider the cases that \( q = 2^p \). Any field element in \( GF(2^p) \) can be uniquely represented by a binary linear combination of \( p \) independent field elements. In our relaxed Min-max check node processing, all messages in the c-to-v message vector, \( \tilde{v}_j \), are derived from a set of exactly \( p \) nodes with the smallest LLRs and independent field elements that do not lie in the \( j \)th stage of the modified trellis. Such a set is called the minimum basis of \( \tilde{v}_j \) in this chapter, and is denoted by \( B_j \). It should be noted that multiple nodes in \( B_j \) can come from the same stage of the trellis. Since any \( \beta \in GF(2^p) \) can be written as a sum of the field elements in a subset of \( B_j \), and the nodes in \( B_j \) have the smallest LLRs, \( \tilde{v}_j(\beta) \) equals the maximum LLR of the nodes in \( B_j \) whose field elements add up to be \( \beta \) according to (2.3).

\( B_j \) can be constructed by a four-step procedure as follows. In the first step, the nodes with the minimum and second minimum LLR of each row in the modified trellis are found. Since the nodes in the same row have the same field element, these nodes can also be called the Min1 and Min2 nodes of the corresponding field element. Two vectors \( M_{\text{min}1} = \{(m1(\beta), I(\beta))\} \) and \( M_{\text{min}2} = \{m2(\beta)\} \) \((1 \leq \beta < q)\) are derived, where \( m1(\beta) \) and \( m2(\beta) \) are the LLRs of the Min1 and Min2 nodes, respectively, and \( I(\beta) \) is the stage index.
of the Min1 node. The Min1 and Min2 nodes for all the $q - 1$ nonzero field elements are the nodes with the smallest LLRs in the modified trellis. Also the Min1 and Min2 nodes in the same row are from different stages of the trellis. Therefore, all the entries in the minimum basis must come from the Min1 and Min2 nodes. The first step serves as preprocessing to reduce the number of nodes involved in later computations. In the second step, a vector $M_j = \{m_j(\beta)\}$ associated with each $\tilde{v}_j$ is obtained such that $m_j(\beta) = m_1(\beta)$ if $j \neq I(\beta)$ and $m_j(\beta) = m_2(\beta)$ if $j = I(\beta)$. This prevents the nodes in the $j$th stage from being included in the $\tilde{v}_j$ computation. In the third step, the entries in $M_j$ are sorted according to increasing order, and the result is denoted by the vector $\bar{M}_j = \{(m_j^{(i)}, \alpha_j^{(i)})\} (1 \leq i < q)$, where $\alpha_j^{(i)}$ is the field element associated with the sorted LLR $m_j^{(i)}$, and $m_j^{(a)} \leq m_j^{(b)}$ if $a < b$. Finally, the minimum basis $B_j = \Psi(\bar{M}_j) = \{(m_j^{(i)}, \alpha_j^{(i)})\} (1 \leq i \leq p)$ is derived from $\bar{M}_j$ and the nodes in $B_j$ are also arranged in the order of increasing LLR. Here the function $\Psi$ is to extract the minimum basis from a sorted set. Starting from the first node in $\bar{M}_j$, one $(m_j^{(i)}, \alpha_j^{(i)})$ is tested at time. If the field element of the node is independent of those of the nodes already put in $B_j$, it will be added to $B_j$. This process is continued until there are $p$ entries in $B_j$.

The proposed relaxed Min-max check node processing scheme is summarized in Algorithm F. The F1 step of this algorithm reorders the messages. The minimum basis for each c-to-v message vector computation is derived in F2~a5 steps. In Step F6, $\tilde{v}_j$ is recovered from $B_j$, where $w_1, w_2, \ldots, w_s$ is a subset of $s$ indices from $\{1, 2, \ldots, p\}$ and $1 \leq s \leq p$. All possible combina-
Algorithm F: Relaxed Min-max Check Node Processing

Initialization: $\alpha_{sum} = \sum_{0 \leq j < d_c} \hat{\alpha}_j$

F1: $\tilde{u}_j(\beta) = u_j(\alpha)$ if $\beta = \hat{\alpha}_j \oplus \alpha$ ($0 \leq j < d_c$) 

F2: compute $M_{min1} = \{(m1(\beta), I(\beta))\}$ ($1 \leq \beta < q$) 
    $M_{min2} = \{m2(\beta)\}$ ($1 \leq \beta < q$) 

for $j = 0$ to $d_c - 1$

F3: $M_j = \{m_j(\beta)\}$ ($1 \leq \beta < q$) 
    $m_j(\beta) = \begin{cases} 
    m1(\beta), & j \neq I(\beta) \\
    m2(\beta), & j = I(\beta) 
    \end{cases}$

F4: Sort $M_j$ to obtain $\bar{M}_j = \{(m_j^{(i)}(\alpha_j^{(i)}))\}$ ($1 \leq i < q$)

F5: $B_j = \Psi(\bar{M}_j) = \{(m_j^{(i)}(\alpha_j^{(i)}))\}$ ($1 \leq i \leq p$)

F6: $\tilde{v}_j(0) = 0$
    $\tilde{v}_j(\beta) = \max\{m_j^{(w_1)}, m_j^{(w_2)}, \ldots, m_j^{(w_s)}\}$
    if $\beta = \alpha_j^{(w_1)} \oplus \alpha_j^{(w_2)} \oplus \cdots \oplus \alpha_j^{(w_s)}$ ($1 \leq s \leq p$)

F7: $v_j(\alpha) = \tilde{v}_j(\beta)$ if $\alpha = \beta \oplus \alpha_{sum} \oplus \hat{\alpha}_j$

In Algorithm F, $M_{min1}$ and $M_{min2}$ can be considered as a 'compressed' version of the v-to-c message vectors. Storing these compressed messages leads to similar memory saving as in the path construction and simplified check node processing in [38] and [42]. On the other hand, the simple and parallelizable
computations in Step F6 enable more efficient c-to-v message vector recovering.

Table 3.1: Example of Algorithm F

<table>
<thead>
<tr>
<th>Step</th>
<th>( \beta )</th>
<th>( M_{min1} )</th>
<th>( M_{min2} )</th>
<th>( M_0 )</th>
<th>( \bar{M}_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td></td>
<td>((0.7,2))</td>
<td>((0.1,3))</td>
<td>((0.5,0))</td>
<td>((0.4,2))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((0.4,2))</td>
<td>((0.6,0))</td>
<td>((0.2,5))</td>
<td>((1.2,4))</td>
</tr>
<tr>
<td>F3</td>
<td>( \beta )</td>
<td>((0.7))</td>
<td>((0.1))</td>
<td>((0.6))</td>
<td>((0.4))</td>
</tr>
<tr>
<td>F4</td>
<td></td>
<td>((0.1,2))</td>
<td>((0.2,6))</td>
<td>((0.4,4))</td>
<td>((0.6,3))</td>
</tr>
<tr>
<td>F5</td>
<td></td>
<td>((0.1,2))</td>
<td>((0.2,6))</td>
<td>((0.6,3))</td>
<td>(0)</td>
</tr>
</tbody>
</table>

To better explain the procedure in Algorithm F, an example of some key values involved in Steps F2-F6 are listed in Table 3.1. In this table, \( GF(2^3) \) and \( d_c = 6 \) are considered. For Steps F3-F6, the values corresponding to the computation of \( \tilde{v}_0 \) are shown. Since the nodes in stage 0 need to be excluded from the computation of \( \tilde{v}_0 \), \( M_0 \) takes the LLR value from \( M_{min2} \) instead of \( M_{min1} \) for the third and fifth entries because \( I(3) = I(5) = 0 \). In Step F4, \( \bar{M}_0 \) is obtained by sorting the entries in \( M_0 \) in increasing order. The Function \( \Psi \) extracts the \( p \) entries of \( \bar{M}_0 \) with the smallest LLR and independent field elements to form \( B_0 \). The first and second entries, \((0.1,2)\) and \((0.2,6)\), in \( \bar{M}_0 \) are put into \( B_0 \) first since their field elements are independent. However, \( 4 = 2 \oplus 6 \) over \( GF(2^3) \). Hence the third entry of \( \bar{M}_0 \) can not be included in \( B_0 \). Next, the fourth entry of \( \bar{M}_0 \), \((0.6,3)\), is tested. The field element ’3’ is
Figure 3.2: FERs of a (837, 726) QCNB-LDPC code over $GF(2^5)$ and a (4191, 3602) binary LDPC with 15 decoding iterations under the AWGN channel.

...
coding iterations under the additive white Gaussian noise (AWGN) channel. In this code, $d_c = 27$ and $d_v = 4$. The $H$ matrix of a QCNB-LDPC code can be divided into sub-matrices that are zero or shifted identity matrices with nonzero entries replaced by elements of $GF(q)$. This type of codes enable more efficient partial-parallel processing. The code used in the simulation is constructed based on Reed-Solomon codes with two information symbols [57]. It can be observed from Fig. 3.2 that the proposed relaxed Min-max check node processing in Algorithm F leads to negligible performance loss compared to the original Min-max algorithm. Fig. 3.2 also shows the performance of a $(4191, 3602)$ binary LDPC code constructed with the method in [56] using the Min-sum decoding. This binary code has similar code rate and code word length in terms of bits as the $(837, 726)$ NB-LDPC code over $GF(2^5)$. 

Figure 3.3: FERs of a $(248, 137)$ QCNB-LDPC code over $GF(2^5)$ and a $(1270, 652)$ binary LDPC with 15 decoding iterations under the AWGN channel.
To show the effects of the relaxed Min-max check node processing on a code with smaller check node degree, Fig. 3.3 illustrates the FERs of a (248, 137) code over $GF(2^5)$ with $d_c = 8$ and $d_v = 4$. As it can be observed, using the relaxed Min-max check node processing in Algorithm F also only leads to very small coding gain loss compared to the original Min-max algorithm. However, the gap is slightly larger than that for the (837, 726) code. The reason is that the probability that multiple nodes from the same stage are used in the same path becomes larger when $d_c$ is smaller. This could further deviate the Min-max approximation. For reference, the performance of the Min-sum decoding for a (1270, 652) binary LDPC code is included in Fig. 3.3.

3.2.2 Simplified Relaxed Min-max Check Node Processing

The $M_j$ vector varies for different $j$, since it includes the entries from $M_{min2}$ if the corresponding Min1 nodes happen to be from the $j$th stage of the trellis. Hence, sorting is re-applied on $M_j$ to derive $\bar{M}_j$, and $B_j$ is recomputed from $\bar{M}_j$ for computing each $v_j$ in Algorithm F. However, it was found from simulations that having precise values of $M_{min2}$ is not necessary, and $M_{min2}$ can be replaced by compensation values when needed. As a result, a set of a few nodes with the smallest LLRs from the entire modified trellis can be first derived, and then each $B_j$ can be computed by slightly modifying the set. Inspired by these, a simplified relaxed Min-max check node processing method is proposed in this chapter to further reduce the complexity.

In the simplified relaxed method, three steps are taken to compute
$B_j$. First, only the $q - 1$ Min1 nodes are found and their LLRs and stage indices are kept in the vector $M_{\text{min}1}$. Next, a set $B = B_I \cup B_{II}$ is derived directly from $M_{\text{min}1}$ through a modified basis construction function denoted by $\Phi$. $B_I$ consists of $p$ nodes from $M_{\text{min}1}$ with the smallest LLRs and linearly independent field elements, and $B_{II}$ is the supplementary set used for making up for the $M_{\text{min}2}$ values. $B_{II}$ contains $n_c$ nodes with the smallest LLRs that are not included in $B_I$, and larger $n_c$ leads to better error-correcting performance. For clarity, $B$ is called the basis of the trellis since there is only one such basis for the entire trellis. In the third step, a function $\Theta$ is employed to derive the minimum basis $B_j$ for different $j$ by modifying $B_I$ using the extra information kept in $B_{II}$. Next, the details of the two functions $\Phi$ and $\Theta$ are presented.

The function $\Phi$ described below computes the trellis basis $B$ from $M_{\text{min}1}$. The nodes in $B_I$ and $B_{II}$ are denoted by $(m_i(\beta), \alpha_i(\beta), I_i(\beta))$ ($1 \leq i \leq p$) and $(m_n(i), \alpha_n(i), I_n(i))$ ($1 \leq i \leq n_c$), respectively. Moreover, $w_1, w_2, \ldots, w_s$ is a subset of $s$ indices in $\{1, 2, \ldots, p\}$.

The parts above and under the dashed line in Function $\Phi$ take care of the construction of $B_I$ and $B_{II}$, respectively. To construct $B_I$, $\beta$ is first checked to see whether it can be expressed as any linear combination of the field elements already in $B_I$. If not, the node $(m_1(\beta), \beta, I(\beta))$ from $M_{\text{min}1}$ will be included in $B_I$. Otherwise, there must exist a subset of nodes already in $B_I$ with the sum of their field elements equal to $\beta$. Assume this subset of nodes is $\{(m_i(\beta), \alpha_i(\beta), I_i(\beta))\}$ with $i = w_1, w_2, \ldots, w_s$ ($1 < s \leq |B_I|$), and the node with
Function Φ: Construct the trellis basis $B = B_I \cup B_{II}$ from unsorted set $M_{\text{min}}$

**Input:** $M_{\text{min}} = \{(m_1(\beta), I(\beta))\} \ (1 \leq \beta < q)$

**Initialization:** $B_I = \emptyset$, $B_{II} = \emptyset$

for $\beta = 1$ to $q - 1$

if $(\beta \notin \text{span}\{\alpha'(1), \alpha'(2), \ldots, \alpha'(|B_I|)\})$

$B_I = B_I \cup \{(m_1(\beta), \beta, I(\beta))\}$

else assume $\beta = \sum_{k=1}^{s} \alpha'(w_k) \ (1 < s \leq |B_I|)$

if $(m_1(\beta) < \max\{m'(w_k)\} = m'(w_l))$

$B_I = B_I \setminus \{(m'(w_l), \alpha'(w_l), I'(w_l))\}$

$B_I = B_I \cup \{(m_1(\beta), \beta, I(\beta))\}$

$(m, \alpha, I) = (m'(w_l), \alpha'(w_l), I'(w_l))$

else

$(m, \alpha, I) = (m_1(\beta), \beta, I(\beta))$

----------------------

if $|B_{II}| < n_c$

$B_{II} = B_{II} \cup \{(m, \alpha, I)\}$

else if $(m < \max\{m''(k)\} = m''(l)) \ (1 \leq k \leq n_c)$

$B_{II} = B_{II} \setminus \{(m''(l), \alpha''(l), I''(l))\}$

$B_{II} = B_{II} \cup \{(m, \alpha, I)\}$
index \( w_l \) has the largest LLR among the nodes in the subset. In the case that \( m_1(\beta) < m'(w_l) \), \( \{m_1(\beta), \beta, I(\beta)\} \) is inserted into \( B_I \) and \( (m'(w_l), \alpha'(w_l), I'(w_l)) \) is excluded from \( B_I \). The nodes in \( B_{II} \) can come from either the nodes that were not inserted in \( B_I \) or nodes that were excluded from \( B_I \), which are denoted by the temporary node \( (m_t, \alpha_t, I_t) \) in Function \( \Phi \). If the number of nodes in \( B_{II} \) is less than \( n_c \), \( (m_t, \alpha_t, I_t) \) is inserted into \( B_{II} \) directly. Otherwise, \( m_t \) is compared with the node with the maximum LLR in \( B_{II} \), \( m''(l) \). If \( m_t < m''(l) \), then \( (m''(l), \alpha''(l), I''(l)) \) is replaced by \( (m_t, \alpha_t, I_t) \) in \( B_{II} \).

The process described in function \( \Phi \) can guarantee that the constructed \( B_I \) consists of \( p \) nodes from \( M_{min} \) with the smallest LLRs and linearly independent field elements. The reason is that when the node \( (m'(w_l), \alpha'(w_l), I'(w_l)) \) is excluded from \( B_I \), it is replaced by the node \( (m_1(\beta), \beta, I(\beta)) \) with smaller LLR. Moreover, the space spanned by the field elements in \( B_I \) remains invariant after the replacement. To facilitate later computations, when an entry is inserted into \( B_I \), it is inserted into appropriate location so that the entries in \( B_I \) are ordered according to increasing LLR. In addition, it can be derived from Function \( \Phi \) that the nodes in \( B_{II} \) have the smallest LLRs among the nodes not in \( B_I \). The nodes in \( B_{II} \) are also arranged in the order of increasing LLR.

As an example, when the input set \( M_{min1} \) equals that in Table 3.1, the contents of \( B_I \) and \( B_{II} \) with \( n_c = 2 \) in each iteration of Function \( \Phi \) are listed in Table 3.2. In the first two iterations, \( (0.7, 1, 2) \) and \( (0.1, 2, 3) \) are put into \( B_I \) since their field elements, ’1’ and ’2’, are linearly independent. In the third
Table 3.2: Example for Function Φ

<table>
<thead>
<tr>
<th>β</th>
<th>(m_1(\beta), I(\beta))</th>
<th>(B_I)</th>
<th>(B_{II})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0.7,2)</td>
<td>(0.7,2)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(0.1,3)</td>
<td>(0.1,2,3)</td>
<td>(0.7,1,2)</td>
</tr>
<tr>
<td>3</td>
<td>(0.5,0)</td>
<td>(0.1,2,3)</td>
<td>(0.5,3,0)</td>
</tr>
<tr>
<td>4</td>
<td>(0.4,2)</td>
<td>(0.1,2,3)</td>
<td>(0.4,4,2)</td>
</tr>
<tr>
<td>5</td>
<td>(0.6,0)</td>
<td>(0.1,2,3)</td>
<td>(0.4,4,2)</td>
</tr>
<tr>
<td>6</td>
<td>(0.2,5)</td>
<td>(0.1,2,3)</td>
<td>(0.2,6,5)</td>
</tr>
<tr>
<td>7</td>
<td>(1.2,4)</td>
<td>(0.1,2,3)</td>
<td>(0.2,6,5)</td>
</tr>
</tbody>
</table>

iteration, \(\beta = 3 = 2 \oplus 1\) and \(m_1(\beta) = 0.5 < \max\{0.7, 0.1\} = 0.7\). Thus, \((m_1(\beta), \beta, I(\beta)) = (0.5, 3, 0)\) replaces \((0.7, 1, 2)\) in \(B_I\). Also \((mt, at, It) = (0.7, 1, 2)\) is included in \(B_{II}\) since \(|B_{II}| = 0\) at this time. When \(\beta = 5\), although \(\beta = 5 = 2 \oplus 3 \oplus 4\), \((0.6, 5, 0)\) could not replace any node in \(B_I\) because \(m_1(\beta) = 0.6 > \max\{0.1, 0.4, 0.5\} = 0.5\). Instead, it is inserted into \(B_{II}\) since \(B_{II}\) is not full yet. When \(\beta = 6 = 2 \oplus 4\), \(m_1(\beta) = 0.2\) is less than 0.4, thus \((0.2, 6, 5)\) replaces \((0.4, 4, 2)\) in \(B_I\), and \((mt, at, It) = (0.4, 4, 2)\) replaces \((0.7, 1, 2)\) in \(B_{II}\) because \(mt = 0.4\) is less than the maximum LLR in \(B_{II}\).

The Function Θ described below computes the minimum basis \(B_j\) from the trellis basis \(B\). To facilitate hardware implementation, the stage index of each node is added in \(B_j\).

At the beginning, \(B_j\) is initialized as \(B_I\). Then the LLRs of the nodes from stage \(j\) are replaced by a compensation value equaling the maximum of the LLRs in \(B_I\) and \(B_{II}\) multiplied by a predefined constant factor \(\gamma\). Adopting this compensation value for each node from stage \(j\) results in over-estimation.
Function Θ: Construct minimum basis $B_j$ from $B$

Input: $B_I = \{(m^{(i)}, \alpha^{(i)}, I^{(i)})\}$ ($1 \leq i \leq p$)
$B_{II} = \{(m^{(i)}, \alpha^{(i)}, I^{(i)})\}$ ($1 \leq i \leq n_c$)

Initialization: $B_j = \{(m_{j}^{(i)}, \alpha_{j}^{(i)}, I_{j}^{(i)})\} = B_I$

for $i = 1$ to $p$
  if $(I_j^{(i)} = j)$
    $m_j^{(i)} = \gamma \cdot \max\{m^{(p)}, m^{(n_c)}\}$
  for $i = 1$ to $n_c$
    if $(I_j^{(i)} \neq j)$
      $(\alpha_{j}^{(i)} = \sum_{k=1}^{s} \alpha_{j}^{(w_k)})$
      $l = \max\{w_k | I_j^{(w_k)} = j\}$
      $B_j = B_j \setminus \{(m_{j}^{(l)}, \alpha_{j}^{(l)}, I_{j}^{(l)})\}$
      $B_j = B_j \cup \{(m_{j}^{(i)}, \alpha_{j}^{(i)}, I_{j}^{(i)})\}$

of some of the c-to-v messages and could lead to performance loss as have been observed from our simulations. To solve this problem, the compensation is modified by making use of the nodes in $B_{II}$, which are additional Min1 nodes not included in $B_I$. For each node $(m^{(i)}, \alpha^{(i)}, I^{(i)})$ in $B_{II}$, there exists a subset of $B_I$, and accordingly $B_j$, whose sum of the field elements equals $\alpha^{(i)}$. Also it is possible that multiple nodes in the subset are from stage $j$. If $I^{(i)} \neq j$, then the last node in the subset that is from stage $j$ will be replaced by $(m^{(i)}, \alpha^{(i)}, I^{(i)})$.

Given the trellis basis $B = B_I \cup B_{II}$, Table 3.3 shows the contents of several example minimum bases after the two 'for-loops' in Function Θ. For $B_0$ construction, only the third entry in $B_I$ is from stage 0. It is replaced by the compensation value $\gamma \max\{m^{(3)}, m^{(2)}\} = 0.6 \gamma$ in the first loop. In the second loop, $I^{(1)} \neq 0$ and $\alpha^{(1)} = 4 = \alpha_0^{(1)} \oplus \alpha_0^{(2)} = 2 \oplus 6$. However, neither $I_0^{(1)}$ nor $I_0^{(2)}$ is '0', and thus $(m^{(1)}, \alpha^{(1)}, I^{(1)})$ does not replace any
Algorithm G: Simplified Relaxed Min-max Check Node Processing

\textbf{Initialization}: \( \alpha_{\text{sum}} = \sum_{0 \leq j < d_c} \tilde{\alpha}_j \)

G1: \( \tilde{u}_j(\beta) = u_j(\alpha) \) if \( \beta = \tilde{\alpha}_j \oplus \alpha, (0 \leq j < d_c) \)

G2: compute \( M_{\text{min}1} = \{ (m(1(\beta), I(\beta)) \} (1 \leq \beta < q) \)

G3: \( B = B_I \cup B_{II} = \Phi(M_{\text{min}1}) \)

\quad for \( j=0 \) to \( d_c-1 \)

G4: \( B_j = \Theta(B, j) \)

G5\sim G6: same as F6\sim F7

The proposed simplified relaxed Min-max check node processing is summarized in Algorithm G. In this algorithm, the basis for the entire trellis, \( B \),

\begin{table}[h]
\centering
\caption{Example for Function \( \Theta \)}
\begin{tabular}{|c|c|c|c|}
\hline
\( B_I \) & \( B_{II} \) & \( \Theta(B, 0) \) & \( \Theta(B, 1) \) \\
\hline
\( B_0 \) & 1st loop & (0.1,2,3) & (0.1,2,3) & (0.6, \gamma, 3, 0) \\
2nd loop & (0.1,2,3) & (0.1,2,3) & (0.6, \gamma, 3, 0) \\
\hline
\( B_1 \) & 1st loop & (0.1,2,3) & (0.1,2,3) & (0.5,3,0) \\
2nd loop & (0.1,2,3) & (0.1,2,3) & (0.5,3,0) \\
\hline
\( B_3 \) & 1st loop & (0.6, \gamma, 2, 3) & (0.1,2,3) & (0.6, \gamma, 3, 0) \\
2nd loop & (0.4,4,2) & (0.1,2,3) & (0.6, \gamma, 3, 0) \\
\hline
\end{tabular}
\end{table}
is derived based on the Min1 nodes using Function $\Phi$. Then Function $\Theta$ is adopted to directly compute the minimum bases from the trellis basis. Instead of recomputing $B_j$ each time as in Algorithm F, Function $\Theta$ replaces the nodes in $B_I$ from stage $j$ with compensation values derived by making use of the nodes in $B_{II}$. As a result, the computations are greatly simplified.

Compared to the check node processing in [38] and [42], our relaxed simplified check node processing can achieve much higher efficiency. The check node processing in [42] needs a large parallel network of computation units to derive the $q$ messages in a vector independently. On the other hand, our design exploits the data dependencies among the computations of the messages inside not only each c-to-v vector, but also all the c-to-v vectors from the same check node. As a result, the required number of computations is substantially reduced. Although the design in [38] also exploits data dependencies, the messages in a c-to-v vector have to be computed in an iterative manner that leads to long latency. On the contrary, our design can generate all $q$ messages in a vector in parallel without duplicating the hardware units $q$ times.

Some simulation results for Algorithm G are shown in Fig. 3.2 and Fig. 3.3. Many values have been tried for $\gamma$, and it was found that $\gamma = 1.125$ can lead to the best performance. This value is used to derive the results in Fig. 3.2 and Fig. 3.3. Compared to Algorithm F, Algorithm G does not lead to noticeable performance loss when $n_c$ is at least 2 for the (837, 726) code over $GF(2^5)$. Moreover, there is only about 0.01dB performance loss compared with the original Min-max algorithm. The reason is that the majority of the
most reliable v-to-c messages, which are used for check node processing, have been kept in $B$. When $p$ is larger, larger $n_c$ may be required to guarantee the performance. For the $(248, 137)$ code, Algorithm G with $n_c = 2$ leads to around 0.03dB performance loss compared to Algorithm F. This is also because the probability that multiple nodes from the same stage are used in the same path becomes larger when $d_c$ is smaller. This degradation can be reduced by using larger $n_c$ as shown in Fig. 3.3.

### 3.3 Relaxed Min-max CNU Architecture

Since Algorithm G has lower complexity than Algorithm F and can achieve similar performance with small $n_c$, it is chosen for the architecture design for the CNU. Our CNU architecture consists of five parts: i) a reordering network to take care of the initialization and reordering of the LLRs in each v-to-c message vector according to the G1 step of Algorithm G; ii) a Min1 selector that finds the Min1 node of each row in the modified trellis (Step G2); iii) a trellis basis constructor that constructs the trellis of the basis from the Min1 nodes (Step b3); iv) a minimum basis constructor that computes the minimum basis for each c-to-v message vector from the trellis basis (Step b4); v) a basis recovery module that generates each c-to-v message vector from the corresponding minimum basis (Step b5~b6). In the following, the architectures for these parts are presented.
3.3.1 Reordering Network and Min1 Selector

In the reordering network, \(q \log_2 q\) multiplexors are required and the switching of the multiplexors is controlled by the bits in \(\hat{\alpha}_j\). The detailed architecture can be found in [1]. Moreover, an adder-register loop is adopted to compute \(\alpha_{sum}\). The Min1 selector consists of \(q - 1\) compare and select structures. All the \(q - 1\) nonzero LLRs in a v-to-c message vector are input to the Min1 selector at a time, and each LLR is compared to the temporary minimum LLR with the same field element stored in the memory. If an input LLR is smaller, it replaces the corresponding temporary minimum LLR, and the corresponding stage index is updated as the stage index of the current input vector. This process is repeated for each v-to-c message vector and takes \(d_c\) clock cycles. At the end, \(M_{min1}\) can be found in the memory.

3.3.2 Trellis Basis Constructor

The architecture of the trellis basis constructor is divided into two parts, Part I and II, for constructing \(B_I\) and \(B_{II}\), respectively. These two parts work simultaneously and a total of \(q - 1\) clock cycles are required to derive \(B_I\) and \(B_{II}\) from \(M_{min1}\). Fig. 3.4 shows the architecture of Part I for deriving \(B_I\). To find the \(p\) nodes with the smallest LLRs, register arrays of length \(p\) and accompanying comparators and multiplexors are required. The LLR and field element registers in Fig. 3.4 are initialized as the maximum possible value and zero field element, respectively. One node from \(M_{min1}\) is input to this architecture at a time. The comparators compare the LLR and the field
element of the input node, which is derived using a counter, with the LLRs and all the linear combinations of the field elements in the register arrays. The comparison results control the select signals of the multiplexors and enable signals of the registers. Each register can either load the input, keep its value unchanged or take the value of the previous register. Moreover, there is an extra register array sharing the same select and enable signals for inserting the indices of the nodes. For simplicity, this register array is not shown in Fig. 3.4 since the indices do not affect the control logic.

At any time, the nodes in the register array are in the order of increasing LLR and the associated field elements are independent of each other. Suppose that a node \((m_1(\beta), \beta, I(\beta))\) is at the input and the nodes \((m^{(1)}, \alpha^{(1)}, I^{(1)}), (m^{(2)}, \alpha^{(2)}, I^{(2)}), \ldots, (m^{(s)}, \alpha^{(s)}, I^{(s)})\) are currently in the register array. \(m_1(\beta)\) is compared with the values in the LLR registers. The output, \(c_i\), of the \(i\)th LLR comparator is ‘1’ if \(m_1(\beta)\) is smaller than the LLR stored in the \(i\)th register and is ‘0’ otherwise. At the same time, \(\beta\) is compared with all

Figure 3.4: Architecture of trellis basis constructor
combinations of field elements in the registers to check the dependency. The field element comparator after the \(i\)th register compares \(\beta\) with the sums of \(\alpha'^{(i)}\) and all possible combinations of \(\alpha'^{(1)}, \alpha'^{(2)}, \ldots, \alpha'^{(i-1)}\). The output, \(x_i\), of the field element comparator is '1' if \(\beta\) is equal to one of the combinations. Also since the field elements in the register array are independent of each other, \(\beta\) can be uniquely represented as a sum of the field elements in the register array. Hence, at most one of the \(x_i\) can be '1'. Accordingly, \(x_p\) can be simplified as 
\[(x_1 + x_2 + \ldots + x_{p-1})'\] when \(|B_I| = p\).

When all the element comparators output '0', \(\beta\) is independent of the elements in the register array. Accordingly, the position into which \((m_1(\beta), \beta, I(\beta))\) should be inserted is solely decided by the LLR comparison results \(c_i\). If \(x_l = '1'\), then \(\beta\) can be represented by a linear combination of the elements in the register array with the largest index being \(l\). In the case that \(m_1(\beta) \geq m'^{(l)}\), \((m_1(\beta), \beta, I(\beta))\) will not be inserted. Otherwise, \((m_1(\beta), \beta, I(\beta))\) will be inserted into an appropriate location \(k\) \((k \leq l)\), such that the LLRs in the register array are kept in increasing order. At the same time, the registers for \((m'^{(l)}, \alpha'^{(l)}, I'^{(l)})\) will be overwritten to keep the elements in the register array independent. Note that the space spanned by the field elements in the register array after replacing \(\alpha'^{(l)}\) with \(\beta\) remains invariant, and \(m'^{(l)}\) is replaced by a smaller LLR \(m_1(\beta)\). In this process, \((m_1(\beta), \beta, I(\beta))\) can only be inserted before the \(l\)th register when \((m'^{(l)}, \alpha'^{(l)}, I'^{(l)})\) should be overwritten. Therefore, the nodes after the \(l\)th position should remain unchanged, and the registers after the \(l\)th position are disabled.
Signals $g_i$ are introduced to facilitate the derivation of the control logic. Define $g_1='1'$ and

$$g_i = (x_1 + x_2 + \cdots + x_{i-1})' (1 < i \leq p). \quad (3.1)$$

$g_i='0'$ (1 < $i$ ≤ $p$) if $\beta$ can be expressed as any linear combination of the elements stored before the $i$th register. Because the field elements in $M_{min}$ are all distinct, $x_1='0'$ and hence $g_2$ is always '1'. Since the field element registers are initialized as zero, when $|B_1| < p$, $x_j='1'$ for some $j$ ($i < j \leq p$) if $x_i='1'$. Therefore, initializing the field elements to zero does not affect the signals $g_i$. Making use of $g_i$, the enable signals of the registers, $e_i$, and the select signals of the multiplexors, $s_i$, in Fig. 3.4 can be generated as

$$\begin{cases} 
    e_i = (c_1 + c_2 + \cdots + c_i)g_i (1 \leq i \leq p) \\
    s_i = (c_1 + c_2 + \cdots + c_{i-1})' (2 \leq i \leq p).
\end{cases} \quad (3.2)$$

The architecture of Part II for deriving $B_{II}$ is similar to that for $B_I$. However, the field element comparators and logic for checking the independency of the input element are not required. As a result, the architecture for deriving $B_{II}$ is a sorter of length $n_c$, and the nodes in the registers are also kept according to increasing order of their LLRs. Moreover, a multiplexor is required at the input of Part II to choose between the $(m^{I}(l), \alpha^{I}(l), I^{I}(l))$ deleted from $B_I$ and $(m_1(\beta), \beta, I(\beta))$ from $M_{min}$. According to Function $\Phi$, the input to Part II is $(m^{I_I}(l), \alpha^{I_I}(l), I^{I_I}(l))$ if the node at the input of Part I has been inserted. As a result, $e_1 + s_2e_2 + \cdots + s_ge_p$ is used as the selection signal for this multiplexor. In addition, another multiplexor is adopted to route $\{m^{I_I}(l), \alpha^{I_I}(l), I^{I_I}(l)\}$.
from the register array of $B_I$, and the $x_i$ signals that indicate the location of $\{m'^{(l)}, \alpha'^{(l)}, I'^{(l)}\}$ serve as the selection signals for the routing.

3.3.3 Minimum Basis Constructor

![Diagram](image)

Figure 3.5: Architecture of minimum basis constructor

Fig. 3.5 illustrates the architecture for the minimum basis constructor that computes $B_j$ from the trellis basis $B$ for an example case of $p = 5$ and $n_c = 2$. The overall process is to first load all the $p$ nodes from $B_I$ into the register arrays for $B_j$ and locate the positions of the nodes from stage $j$. Then the nodes in $B_{II}$ are read out one at a time in the order of increasing LLR, and each node is tested to see whether it should replace a node in $B_j$ according to Function $\Theta$. If yes, it is inserted into the appropriate position of $B_j$ and the
node that should be replaced is overwritten, so that the nodes in $B_j$ are still in the order of increasing LLR after the replacement. The entire process is completed in $n_c + 1$ clock cycles. The register arrays for $B_j$, which are labeled as $m_j'^{(i)}$, $\alpha_j'^{(i)}$ and $f_j'^{(i)}$ in Fig. 3.5, have similar architectures as those in the dashed blocks in Fig. 3.4. The registers have enable signals $e_i$, and there are multiplexors with select signal $s_i$ to decide the update values for the registers. However, opposite to that in Fig. 3.4, each register in the $B_j$ array takes the value from the register on its right.

In the first clock cycle, the LLRs and field elements of the nodes in $B_I$ are loaded into the register arrays $m_j'^{(i)}$ and $\alpha_j'^{(i)}$ for $B_j$. In addition, those of the nodes in $B_{II}$ are loaded into shift registers labeled as $m_j''^{(i)}$ and $\alpha_j''^{(i)}$ in Fig. 3.4. At the same time, the indices of the nodes in both $B_I$ and $B_{II}$ are compared with $j$. The comparison result is ‘1’ if the index equals $j$, and the results are stored into the register array $f_j'^{(i)}$ for $B_I$ and the shift register $f_j''^{(i)}$ for $B_{II}$ in Fig. 3.4. Moreover, the compensation value $\gamma \cdot \max\{m_j''^{(p)}, m_j''^{(n_c)}\}$ is computed and stored in a separate register. To facilitate later computations that require the nodes in $B_j$ to be in the order of increasing LLR, this compensation value is actually not stored into the register array for $B_j$. Instead, only the locations of the nodes whose indices are $j$ are recorded in $f_j'^{(i)}$.

In the second clock cycle, $(m_j''^{(1)}, \alpha_j''^{(1)}, f_j''^{(1)})$, the node with the smallest LLR in $B_{II}$, is read out. If $f_j''^{(1)} = j$, this node is also from the $j$th stage, and no further operation is needed. Otherwise, the $q - 1$ nonzero sums of all possible combinations of the $p$ ($q = 2^p$) field elements in the $B_j$ register
array are computed using combinational logic. The sums are compared with $\alpha'^{(1)}$ to generate a $q - 1$-bit vector. Since the field elements in $B_j$ are linearly independent, only one bit in this vector is asserted. This vector is passed through a binary encoder to generate a $p$-bit vector. The nonzero bits in the $p$-bit vector tell the locations of the nodes whose field elements add up to be $\alpha'^{(1)}$. For example, if $\alpha'^{(1)} = \alpha'^{(1)}_j + \alpha'^{(2)}_j + \alpha'^{(5)}_j$, then the encoder output is '11001'. This $p$-bit vector is masked by $f'^{(i)}_j$ to generate signals $d_i$ ($1 \leq i \leq p$), which are sent to the control unit. This masking is necessary since only the nodes from the $j$th stage in $B_j$ can be possibly replaced. Assume that $\max\{i|d_i = '1'\} = l$. $m'^{(1)}$ must be larger than $m'^{(l)}_j$ since the nodes in $B_I$ are the linearly independent nodes with the smallest LLRs. Also the nodes in $B_j$ are ordered according to increasing LLR. Hence, $(m'^{(1)}, \alpha'^{(1)}, f'^{(1)})$ can be only inserted after the $l$th nodes in $B_j$, and the registers before the $l$th position in the $B_j$ arrays should be disabled. Meanwhile, $m'^{(1)}$ is compared with the LLRs in $B_j$ to decide the position for insertion, so that the nodes in $B_j$ are still in the order of increasing LLR after the insertion. Similarly, the comparison result $c_i$ is asserted if $m'^{(1)} < m'^{(i)}_j$. In each of the later clock cycles, one node is read from $B_{II}$ at a time. Since the nodes in $B_{II}$ are also ordered according to increasing LLR, the same logic can be adopted to decide whether the node from $B_{II}$ should replace a node in $B_j$, and where it should be inserted into the $B_j$ register array.

In summary, the logic for the control unit in Fig. 3.4 when the $k$th node
is read out from $B_{II}$ can be implemented according to the following equations.

$$\begin{align*}
\bar{e}_i &= (c_1 + c_2 + \cdots + c_i)' \\
\hat{e}_i &= (d_{i+1} + d_{i+2} + \cdots)' \\
\tilde{e} &= (d_1 + d_2 + \cdots + d_p) f^{n(k)} \\
e_i &= \bar{e}_i \hat{e}_i \tilde{e} (1 \leq i \leq p) \\
s_i &= c_1 + c_2 + \cdots + c_{i+1} (1 \leq i < p) .
\end{align*}$$ (3.3)

In the above equations, $\bar{e}_i$ come from the comparison results of the LLRs and decide which position in the $B_j$ register arrays to insert the node from $B_{II}$ if it indeed needs to be inserted. $\hat{e}_i$ disable the registers to the left of the $l$th position in the $B_j$ arrays if all $d_i$ for $i > l$ are zero. Note that $\hat{e}_p$ is always '1'.

Moreover, $\tilde{e}$ is the decision of whether the insertion should be made or not. The Boolean products of these three signals are the enable signals for the $B_j$ register arrays.

### 3.3.4 Basis Recovery Module

Fig. 3.6 shows the basis recovery module that implements Step b5 and b6 of Algorithm G. $GF(2^3)$ is chosen as an example for the purpose of simple illustration. The $p$ nodes in the minimum basis $B_j$, $(m_j^{(i)}, \alpha_j^{(i)}, f_j^{(i)})$ (1 $\leq i \leq p$), and the compensation value, $\gamma \cdot \max\{m^{(p)}, m^{m(n_c)}\}$, are read out simultaneously, and the c-to-v message vector, $v_j$, is computed in one clock cycle. According to Step b5, the data path of the basis recovery module is mainly a routing network that selects one of the $p$ LLRs from $B_j$, the compensation value or zero to become each $v_j(\beta)$ (0 $\leq \beta < q$), depending on how $\beta$ is represented by the linear combination of the field elements of the
nodes in $B_j$. Each $v_j(\beta)$ can be computed by a $p + 1$-input AND-OR tree as shown in Fig. 3.6 (a), and $q$ copies of such a tree are employed to compute all the $q$ messages in $v_j$ simultaneously. The control signals in the trees, $s_{i,\beta}$ and $r_{i,\beta}$ ($0 \leq i \leq p$), are generated using the architecture in Fig. 3.6 (b). For computing $v_j(\beta)$, only one of $s_{i,\beta}$ equals '1', and hence only one input is passed through the tree. In addition, if the stage index of any node in $B_j$ is $j$, then the compensation value $\gamma \cdot \max\{m^{(p)}, m^{(nc)}\}$ is used instead for that node. This is achieved by the multiplexors with control signals $r_{i,\beta}$ in the tree. Note that if a node with stage index $j$ in $B_j$ has been replaced by a node in $B_{II}$ in Function $\Theta$, its stage index is also updated and is no longer $j$.

In order to generate the control signals $s_{i,\beta}$ and $r_{i,\beta}$, all the $2^p = q$ combinations of the $p$ field elements in $B_j$ are divided into $p + 1$ groups as

Figure 3.6: Architecture of basis recovery module
shown in Fig. 3.6 (b) and their sums are computed. Group \( i \) contains \( \alpha_j^{(i)} \) plus all possible combinations of \( \alpha_j^{(k)} \) \((0 < k < i)\). Hence, Group \( i \) \((i > 0)\) has \(2^{(i-1)}\) combinations of elements. In addition, to incorporate the reverse reordering in Step b6, \( t_j = \alpha_{sum} + \hat{\alpha}_j \) is added to the sums of the combinations of the field elements. Taking advantage that the \( p \) nodes in \( B_j \) are stored in the order of increasing LLR, the maximum LLR of the nodes included in each combination in Group \( i \) is \( m_j^{(i)} \). Therefore, if \( \beta \) equals the sum of a combination of elements in Group \( i \), \( v_j(\beta) \) equals \( m_j^{(i)} \) or the compensation value \( \gamma \cdot \max\{m_j^{(p)} , m_j^{(nc)}\} \) if any node of the field elements in the combination is from the \( j \)th stage. This is because that \( \gamma > 1 \), and hence the compensation value is larger than any of the LLRs in \( B_I \) and \( B_{II} \). Note that Group 0 contains only \( t_j \). Hence \( v_j(t_j) = 0 \).

To derive all the \( s_{i,\beta} \) signals in one clock cycle, the sums of the combinations of field elements are computed simultaneously. Each sum is passed through a binary decoder to generate a \( q \)-bit binary vector, which is only ‘1’ in the \( \beta \)th bit if the sum is \( \beta \). The vectors derived from the sums in the same group are bit-wise ORed. Since the sums are all distinct, each decoder output vector has a single ‘1’ in different positions. Hence, the outputs of the OR gates for Group \( i \) have exactly \( 2^{(i-1)} \) ‘1’ bits, and a ‘1’ bit in the \( \beta \)th position indicates that the one of the sums in this group equals \( \beta \). Therefore, the \( \beta \)th bits from the \( p \) group vectors tell which group \( \beta \) lies in, and can be used as the control signals \( s_{i,\beta} \) \((0 \leq i < p)\).

A similar approach is adopted to derive all the \( r_{i,\beta} \) signals. Recall that
the compensation value should only be used for $v_j(\beta)$ if any of the nodes whose field elements add up to be $\beta$ is from the $j$th stage of the trellis. The flag of whether a node is from the $j$th stage has been recorded as $f_j^{(i)}$ in the minimum basis constructor architecture. Hence, the decoder outputs should be masked by the corresponding flags to generate $r_{i,\beta}$.

Using the proposed CNU architecture, only $q-1$ v-to-c messages need to be stored at the output of the Min1 selector. Then the trellis basis constructor makes further compression such that only $p + n_c$ of them are needed for each check node. Compared to storing $(d_c - 2)q$ intermediate messages for each of the forward and backward processes, the memory requirement has been reduced to a fraction. The memory requirement of the proposed CNU is also lower than those in [38] and [42], which are $1.5n_m$ ($n_m$ is chosen as 16 for $GF(2^5)$) and $2(q - 1)$ messages, respectively. The CNU design in [42] requires an expensive computing network that carries out $(q - 1)((3q/4 + 1)d_c - 3)$ comparisons on the LLRs. On the other hand, in our proposed CNU, the G2 and b3 steps of Algorithm G need $(q - 1)(d_c - 1)$ and $(q - 1)(p + n_c)$ comparisons, respectively. In addition, $n_cpd_c$ comparisons are required in the $\Theta$ function of Step G4. Hence, the total number of comparisons required in our CNU is $(q - 1)(p + n_c + d_c - 1) + n_cpd_c$. It is much less than that in [42]. Although our CNU design has more complicated control logic, it accounts for a smaller part of the CNU. Hence, our proposed CNU has much lower gate count than that in [42]. Compared to our design, the CNU in [38] requires less logic gates. However, it computes the messages in each c-to-v message vector.
serially and needs $2n_m$ clock cycles to finish. On the contrary, the proposed design is capable of computing all the messages in a vector simultaneously, and the derivations of the LLRs are simplified through representing each field element as a combination of the elements in a basis. Accordingly, the CNU in [38] would need much more logic gates to achieve the same throughput as the proposed design. The complexity reduction that can be achieved by the proposed CNU will be shown quantitatively for an example NB-LDPC code in Section 3.4.

3.4 Simplified Min-max NB-LDPC Decoder Architecture

In this section, the proposed CNU is adopted to develop a partial-parallel decoder architecture for QCNB-LDPC codes. Using the construction methods in [57], the $H$ matrix of a regular QCNB-LDPC code over $GF(q)$ can be divided into sub-matrices of dimension $(q - 1) \times (q - 1)$. Each of them is either zero or a cyclically shifted identity matrix with nonzero entries replaced by elements of $GF(q)$. Since the CNUs constitute a significant part of the decoder, adopting the proposed CNU can lead to significant reduction in the decoder complexity. Moreover, to further reduce the memory requirement of the decoder, only the nodes in the $M_{min}$ vector or the trellis basis, which can be also considered as ”compressed” messages, are stored in the decoding process. The v-to-c and c-to-v message vectors are not stored, and are computed from the basis when needed.
3.4.1 Top Level Decoder Architecture

Figure 3.7: Top level architecture of QCNB-LDPC decoder

The top level architecture of our proposed partial-parallel decoder for a QCNB-LDPC code with column weight $d_v$ is shown in Fig. 3.7. In our design, the messages associated with one column of $H$ is taken care of at a time, and all the messages in the same vector are processed simultaneously. There are five types of memory blocks in Fig. 3.7. Each copy of RAM $M$ is capable of storing the $M_{\min 1}$ vector and $\alpha_{\text{sum}}$ for each of the $q-1$ check nodes associated with one block row of the $H$ matrix. Assume that $w$ bits are used to represent each LLR. The size of each RAM $M$ is $(q-1) \times ((q-1) \times (w + \lceil \log_2 d_c \rceil) + p)$ bits. A total of $d_v$ copies of RAM $M$ are required in our design, and the computations for $d_v$ check nodes are carried out simultaneously. These $d_v$ check nodes are connected to the same variable node and each of them is from
a different block row of the $H$ matrix for QCNB-LDPC codes. In addition, there are $d_v$ copies of RAM A blocks in our decoder. Each of them stores $q - 1$ $\alpha_{sum}$ associated with the check nodes in a block row of $H$. A RAM S block stores the most reliable symbol of each v-to-c message vector sent to the $q - 1$ check nodes in a block row. Hence, its size is $d_c \times (q - 1) \times p$ bits. Each RAM B records the trellis bases for the check nodes in a block row, and its size is $(q - 1) \times (p + n_c) \times (w + p + \lceil \log_2 d_c \rceil)$ bits. $d_v$ copies of RAM B are necessary to carry out the computations for $d_v$ check nodes simultaneously. One single RAM C is included in the decoder. It stores the channel information, and has $(q - 1) \times d_c \times w \times (q - 1)$ bits. Besides these five types of memory blocks, the decoder also has memories storing the coefficients of $H$ and hard-decision symbols for final decoding output, which are not shown in Fig. 3.7. Taking advantage of the structure of the $H$ matrix constructed using the method in [57], the coefficients of $H$ are stored in $d_v$ ROM blocks, each of which has $d_c \times p$ bits. Moreover, the size of the memory for the decoding output is $(q - 1) \times d_c \times p$. The width, depth and number of blocks of each type of RAMs used in our decoder are summarized in Table 3.4.

In order to increase the throughput and hardware utilization efficiency, the decoding processes of two consecutive words are interleaved. This scheduling will be detailed in Section 3.4.2. As a result of the interleaving, the numbers of RAM M, A, S, B, C, and hard-decision decoding output memory blocks need to be doubled. This is denoted by the ‘interleaving factor’ in Table 3.4.

At the beginning of the decoding, the channel information is loaded
into RAM C, and is used as the v-to-c messages in the first decoding iteration. In the following iterations, the channel information is added up with the c-to-v messages derived from the previous iteration in the variable node unit (VNU) to compute the v-to-c messages. $d_v$ v-to-c message vectors associated with one column of $H$ are computed in the VNU in each clock cycle. They are also normalized according to (2.1). The multiplication blocks multiply field elements of the messages by the corresponding nonzero entries of $H$, and their outputs are sent to the reordering networks. The re-ordered vectors are fed to the Min1 selectors, which iteratively compare the incoming vectors with the corresponding temporary $M_{min1}$ vectors stored in RAM M. Meanwhile, $\hat{\alpha}_j$ are also added up iteratively to derive $\alpha_{sum}$, which are stored at the same memory addresses as the corresponding $M_{min1}$. The updated $M_{min1}$ and $\alpha_{sum}$ are written back to RAM M in the next clock cycle. The final $M_{min1}$ vectors and $\alpha_{sum}$ for all check nodes are available after all columns of $H$ are processed. Then the $M_{min1}$ vectors for the $d_v$ check nodes connected to the same variable node are passed to the trellis basis constructors each time. Meanwhile, the corresponding $\alpha_{sum}$ are written to RAM A. The trellis bases are stored into RAM B. After that, the minimum bases for the $d_v$ c-to-v message vectors corresponding to the same column of $H$ are computed in each clock cycle, and the basis recovery follows. At the end, the $d_v$ c-to-v message vectors are sent back to the VNU after their field elements are divided by the corresponding entries in $H$. In this decoding process, neither the v-to-c nor c-to-v messages are stored, and hence the memory requirement is greatly reduced.
The architectures for the units inside the CNU have been detailed in the previous section. Next, the architectures for the other blocks in Fig. 3.7 are described. In the VNU, $d_v$ c-to-v message vectors and the channel information are added up using a tree structure to calculate the \textit{a posteriori} information. Then the \textit{a posteriori} information is subtracted by each of the $d_v$ c-to-v message vectors to derive the v-to-c message vectors. Each normalizer in Fig. 3.7 is capable of normalizing one vector at a time. It consists of a $q$-input comparator tree for finding the smallest LLR in each vector and $q$ subtractors to minus the minimum LLR from each LLR in the vector. The field element of the smallest LLR in each vector is also recorded. After multiplied by the corresponding entry in $H$, it becomes $\hat{\alpha}_j$, whose bits are used as the control signals for the reordering network.

Power representation of finite field elements is adopted in the multiplication blocks in Fig. 3.7 to reduce the gate count using the idea from [2]. However, polynomial representation of field elements is used in other functional blocks of the decoder since finite field additions can not be done directly in power representation. Therefore, polynomial representation is converted to power representation first using a fixed routing network implementing one-to-one mapping in the multiplication blocks. Such a routing network does not cost any logic gate. Then the multiplications in power representation can be done by a barrel shifter as shown in [2]. After the multiplications, reverse routing is applied. The division blocks in the decoder can be implemented based on the same architecture.
3.4.2 Computation Scheduling

Fig. 3.8 shows the scheduling of the computations in the proposed decoder. The latencies $t_1 \sim t_4$ labeled in this figure are caused by pipelining of the modules in the decoder. In our design, one column of $H$ is processed at a time, and the $d_c$ v-to-c message vectors for the same column are computed in each clock cycle. Accordingly, $d_c(q - 1)$ clock cycles are required for the VNU in each decoding iteration. The Min1 selectors can start as soon as v-to-c messages are available. The final $M_{\text{min1}}$ vector for a check node is not computed until all the v-to-c message vectors connected to this check node are processed, and the trellis basis constructors can only start after that. $q - 1$ clock cycles are required for constructing each trellis basis. To match the speed of the VNU and Min1 selectors, $\lceil q/d_c \rceil$ copies of the architecture in Fig. 3.4 are employed for each trellis basis constructor block in Fig. 3.7, and they are used in a ping-pong manner. The minimum basis constructors start after the trellis bases for all check nodes are available. Considering that one clock cycle is needed to load $B_I$ and $B_{II}$ into the register arrays of the minimum basis constructor in Fig. 3.5, constructing a minimum basis takes $n_c + 1$ clock cycles. However, the construction of the minimum basis needs to be done in one clock cycle to match the speed of the other modules in the decoder. Hence, $n_c + 1$ copies of the architecture in Fig. 3.5 are employed for each minimum basis constructor block in Fig. 3.4, and they are also used in a ping-pong manner. The basis recovery starts as soon as the corresponding minimum basis is available, and the computed c-to-v message vectors are sent
to the VNU right after to start the decoding for the next iteration.

Each decoder module requires around the same number of clock cycles. However, the VNU can not start the computations for the next decoding iteration until the c-to-v message vectors are recovered as shown in Fig. 3.8. This leads to only about 50% hardware utilization efficiency. To solve this problem and increase the throughput, two received blocks are interleaved. The darker bars in Fig. 3.8 belong to the decoding of a second word. To facilitate the interleaving, the numbers of all memories, except the coefficient memories, need to be doubled as shown in Table 3.4.

3.5 Complexity Analyses and Comparisons

The hardware complexity of the proposed design is analyzed and compared with prior work in this section by using a (837, 726) QCNB-LDPC code over $GF(2^5)$ constructed based on Reed-Solomon codes with two information symbols [57]. The $H$ matrix of this code can be divided into $4 \times 27$ submatrices of dimension $31 \times 31$, and $d_c = 27$, $d_v = 4$. As it can be observed from Fig. 3.2, using $w = 5$ bits to represent each LLR only leads to negligible performance loss for this code.
The proposed CNU and overall decoder with \( w = 5 \) are synthesized by Synopsys Design Compiler using SMIC 180\( \mu m \) CMOS process. The synthesis results for the CNU are listed in Table 3.5. The gate count is derived through dividing the total area from the synthesis report by the area of a single NAND2 gate. [42] did not provide the synthesis results of the CNU separately. However, the rest parts of the decoder in [42] are almost the same as those in our decoder. For the purpose of comparison, the area of the rest parts is subtracted from the overall decoder area, and the difference is listed as the complexity of the CNU from [42] in Table 3.5. Moreover, [38] did not provide any synthesis result, and the gate count for the CNU in [38] is from architectural analysis. The proposed decoder and that in [42] employ four CNUs, and the one in [38] adopts \( q - 1 \) CNUs. The gate counts and memory sizes in Table 3.5 are for all the CNUs employed in the decoders, and the numbers of clock cycles needed to finish the check node processing for an entire decoding iteration are listed. These clock cycle numbers do not include the initial latencies caused by pipelining at the beginning of the decoding. It can be computed that, compared to the CNU in [42], our proposed CNU requires only 19\% logic gates and less memory. To achieve the same throughput as the proposed CNU, the CNU in [38] would require almost twice logic gates.

The complexities of the overall decoders are listed in Table 3.6. In the equivalent total gate count computation, it is assumed that storing a bit in memory takes the area of 1.5 NAND2 gates [42]. The pipelining latencies in our design are \( t_1 = 6 \), \( t_2 = q - d_c = 5 \), \( t_3 = n_c + 1 = 3 \) and \( t_4 = 2 \). Hence, the
15-iteration decoding of each word needs \(d_c(q-1)+t_1+t_2+d_c(q-1)+t_3+t_4 = 837 \times 2 + 6 + 5 + 3 + 2)/2 \times 15 = 12675\) clock cycles.

Among existing NB-LDPC decoder designs, the one in [42] is the most efficient. It is a simplified EMS decoder for QC codes, and also processes all the messages associated with one column of \(H\) at a time. When interleaving is adopted, the proposed decoder can achieve slightly higher throughput than the decoder in [42]. Our design needs to store less intermediate messages than that in [42]. However, two data blocks are interleaved in our design in order to achieve higher throughput, which leads to almost doubling the memory. As a result, the overall memory requirement of our decoder is higher than that of [42]. On the other hand, the proposed decoder requires less than one third of the area for logic gates. This is mainly because that the proposed CNU is much more efficient. Overall, our decoder only requires 67% the area of the decoder in [42]. In terms of throughput-over-area ratio, our decoder is 52% more efficient in terms of throughput-over-area ratio.

In both our decoder and that in [42], \(q-1\) LLRs are stored for each code position as the channel information. In the case that bit reliabilities instead of symbol reliabilities are available from the channel, the number of LLRs need to be stored for each code position can be reduced to \(p = \log_2 q\). Accordingly, the size of RAM C in our design can be reduced by a factor of around \(q/p\). RAM C accounts for a majority part of the memory requirement in our design. It can be calculated that the memory requirement of our decoder is only 119% of that in [42] if bit reliability is available as the channel information. This
translates to 112% higher efficiency in our design.

The two recent QCNB-LDPC decoders in [38] [2] are also compared in Table 3.6. Both of them are layered decoders. The design in [38] computes multiple vectors at a time, but the messages in a vector are calculated one after another. On the contrary, the decoder in [2] employs parallel processing for each vector, but only computes one vector at a time. As a result, these designs require several times more clock cycles than the proposed decoder. Although our decoder computes all the messages in a vector simultaneously and has higher level of parallelism, the proposed novel check node processing computes all messages efficiently from a basis. As a result, the logic gate count is not necessarily higher and our design is several times more efficient compared to those in [38] and [2].

The binary LDPC decoder architecture in [64] is one of the most efficient. For comparison, the synthesis results of this decoder for a (4191, 3602) binary QC-LDPC code with 5-bit word length and 15 decoding iterations are listed in Table 3.6. Although binary LDPC decoders have much lower complexity than NB-LDPC decoders, their error-correcting performance is much inferior as can be observed from Figs. 3.2 and 3.3.

3.6 Summary

This chapter developed a novel relaxed check node processing scheme for the Min-max NB-LDPC decoding algorithm. By making use of the property that each finite field element can be uniquely represented by a linear com-
bination of the elements in the minimum basis, all the entries in a message vector are computed simultaneously in an efficient way. Moreover, an innovative method is proposed to derive the minimum basis for each message vector from a common trellis basis. This further reduces the memory requirement and computation complexity. Adopting the proposed check node processing, the overall decoder can achieve substantially higher efficiency than all existing designs.
Table 3.4: Configurations and sizes of memories

<table>
<thead>
<tr>
<th>Memory</th>
<th># of blocks</th>
<th>Memory depth</th>
<th>Data width</th>
<th>Interleaving factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM M</td>
<td>$d_v$</td>
<td>$(q - 1)$</td>
<td>$(q - 1)(w + \lceil \log_2 d_c \rceil) + p$</td>
<td>2</td>
</tr>
<tr>
<td>RAM A</td>
<td>$d_v$</td>
<td>$(q - 1)$</td>
<td>$p$</td>
<td>2</td>
</tr>
<tr>
<td>RAM S</td>
<td>$d_v$</td>
<td>$(q - 1)d_c$</td>
<td>$p$</td>
<td>2</td>
</tr>
<tr>
<td>RAM B</td>
<td>$d_v$</td>
<td>$(q - 1)$</td>
<td>$(p + n_c)(w + p + \lceil \log_2 d_c \rceil)$</td>
<td>2</td>
</tr>
<tr>
<td>RAM C</td>
<td>1</td>
<td>$(q - 1)d_c$</td>
<td>$w(q - 1)$</td>
<td>2</td>
</tr>
<tr>
<td>Coefficient memory</td>
<td>$d_v$</td>
<td>$d_c$</td>
<td>$p$</td>
<td>1</td>
</tr>
<tr>
<td>Hard-decision output memory</td>
<td>1</td>
<td>$(q - 1)d_c$</td>
<td>$p$</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 3.5: CNU complexities for (837, 726) QCNB-LDPC code

<table>
<thead>
<tr>
<th></th>
<th>Logic gate (NAND2)</th>
<th>Memory (bits)</th>
<th># of clks /iteration</th>
<th>Clock freq. (Mhz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min1 Sel.</td>
<td>20267</td>
<td>39060</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Trellis BC</td>
<td>46805</td>
<td>13020</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Minimum BC.</td>
<td>56651</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Basis Rec.</td>
<td>28871</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Proposed CNU</td>
<td>152594</td>
<td>52080</td>
<td>837</td>
<td>200</td>
</tr>
<tr>
<td>CNU [42]</td>
<td>784245</td>
<td>58280</td>
<td>837</td>
<td>200</td>
</tr>
<tr>
<td>CNU [38]</td>
<td>72478</td>
<td>55240</td>
<td>3456</td>
<td>200</td>
</tr>
</tbody>
</table>

Table 3.6: Decoder complexities for (837, 726) QCNB-LDPC code

<table>
<thead>
<tr>
<th>Decoder architecture</th>
<th>[64] (binary decoder)</th>
<th>[2]</th>
<th>[38]</th>
<th>[42]</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Logic gate (NAND2)</td>
<td>502k</td>
<td>228k</td>
<td>374k</td>
<td>892k</td>
<td>260k</td>
</tr>
<tr>
<td>Memory (bits)</td>
<td>193k</td>
<td>718k</td>
<td>426k</td>
<td>268k</td>
<td>407k</td>
</tr>
<tr>
<td>Total gate (NAND2)</td>
<td>792k</td>
<td>1310k</td>
<td>1013k</td>
<td>1294k</td>
<td>871k</td>
</tr>
<tr>
<td>Total gate (normalized)</td>
<td>0.61</td>
<td>1.02</td>
<td>0.78</td>
<td>1</td>
<td>0.67</td>
</tr>
<tr>
<td># of clks (15 iter.)</td>
<td>495</td>
<td>53541</td>
<td>62240</td>
<td>12995</td>
<td>12675</td>
</tr>
<tr>
<td>Clock freq. (Mhz)</td>
<td>250</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>Throughput (Mbps)</td>
<td>2116</td>
<td>16</td>
<td>13</td>
<td>64</td>
<td>66</td>
</tr>
<tr>
<td>Efficiency</td>
<td>53.9</td>
<td>0.24</td>
<td>0.26</td>
<td>1</td>
<td>1.52</td>
</tr>
</tbody>
</table>

77
Chapter 4

Enhanced Iterative Hard Reliability-based Majority-logic Decoding Algorithm and Architecture

4.1 Introduction

In this chapter, an enhanced (E-)IHRB algorithm is proposed. Through incorporating the probability information from the channel into the message initialization of the IHRB algorithm and excluding the contribution of the same check node from the variable-to-check (v-to-c) message, the E-IHRB algorithm can bridge the performance gap between the IHRB and ISRB algorithms with small complexity overhead. Novel partial-parallel architectures are also developed in this chapter to efficiently implement the IHRB and E-IHRB algorithms through algorithmic and architectural optimizations. Many construction methods of NB-LDPC codes lead to quasi-cyclic (QC) or cyclic codes [36]. For QC codes, our IHRB decoder processes one row of sub-matrices at a time, and the variable node units (VNUs) are implemented by simple logic when all \( q \) messages in a vector are kept. Cyclic NB-LDPC codes have the advantage that their encoders can be implemented easily by linear feedback shift registers. However, these cyclic codes usually involve finite fields of high order, in which case keeping all \( q \) messages leads to large memory requirement. Novel
schemes are developed in this chapter to store only a small proportion of the messages without incurring noticeable performance loss. In addition, a shift-message decoder architecture is proposed for cyclic NB-LDPC codes to enable efficient partial-parallel processing. The message shifting is accomplished through concatenating memories with VNUs to reduce the area requirement. It is non-trivial to extend the IHRB decoder architecture to implement the E-IHRB algorithm since recording the messages from check nodes may lead to large memory overhead, especially when the column weight of the code is not small. By making use of the properties of the IHRB algorithm, an innovative approach is proposed in this chapter to reverse the contributions from check nodes through remembering only a few extra values for each variable node. Compared to the Min-max decoder architectures in [1] [37] [38], which are the most efficient existing QCNB-LDPC decoder designs, the proposed E-IHRB decoder architecture has at least tens of times lower complexity with moderate coding gain loss.

4.2 Enhanced IHRB-MLGD Algorithm

One reason that the IHRB algorithm has performance loss compared to the ISRB algorithm is that the soft information is lost in the initialization of the reliability measures: the reliability measure for the hard-decision symbol is set to $\gamma$, while all other measures in the same vector are set to zero. Recording different initialization values does not cost extra memory, if the word length is not changed. Hence, the IHRB algorithm can be enhanced by initializing the
reliability measures according to the probability information from the channel in a way similar to that in the ISRB algorithm. Nevertheless, to reduce the decoder hardware complexity, the maximum reliability measure in each vector should be set to a constant $\gamma$ in the format of $2^w - 1$ ($w \in \mathbb{Z}^+$) as before, so that it does not need to be stored, and the clipping can be simplified as will be detailed in later sections. Such initialization can be done first as $R_{j,l}^{(0)} = \lfloor \varphi_{j,l} \rho \rfloor$, $(0 \leq l < q)$, where $\rho$ is a constant that has similar effect as the $\mu$ in the ISRB decoding. The optimum value of $\rho$ can be derived from simulations, and is affected by the data format of reliability measures and dynamic range of the symbol reliabilities from the channel. Then, the reliability measures in each vector are offset by the same value so that the maximum measure becomes $\gamma$. In case some measures become negative after the offset, zero is used instead. Accordingly, the reliability measures will also remain as integers in the range of $[0, \gamma]$ during the decoding if this enhancement is adopted.

Another approach for initializing the reliability measures using soft information has been proposed in [65]. In this approach, each reliability measure is offset by the smallest measure in the vector, and then divided by the difference of the largest and smallest measures in the vector. Compared to this approach, our initialization scheme is more hardware friendly, since it needs constant multiplications instead of integer divisions. Moreover, in the case that the probabilities of different finite field elements for a received symbol are very close, the approach in [65] can increase the gaps of the reliability measures for different finite field elements. As a result, it may become harder for the
decoding to converge to the second most reliable or other field element, even if it is the correct one. On the other hand, our initialization does not change the relative gaps between the reliability measures in a vector, and does not have this problem. From simulations, it was found that our initialization method can achieve small coding gain over that in [65].

In the IHRB algorithm, the v-to-c messages from variable node $j$ to all connected check nodes are $z_{j}^{(k)}$. They are the same, and the message that a check node supplied in the previous decoding iteration is not excluded from the v-to-c message to this check node as in the Min-max or other BP-based decoding. As a result, the v-to-c messages are not extrinsic information, and the decoding process is more likely to be trapped in loops. Therefore, the performance of the IHRB algorithm can be further improved by excluding the contribution from the same check node in the v-to-c messages. Such a task can be very challenging. Fortunately, each $\sigma_{i,j}$ computed from Algorithm E only increases one of the reliability measures by one. By making use of this property, the contribution from a check node can be excluded from the v-to-c message according to the following analysis. Denote the largest and second largest reliability measures in the vector for the $j$th symbol at the beginning of the $k$th decoding iteration by $R_{j}^{(k)\text{max}}$ and $R_{j}^{(k)\text{max}2}$, respectively. Hence the finite field element corresponding to $R_{j}^{(k)\text{max}}$ is $z_{j}^{(k)}$. Represent the field element corresponding to $R_{j}^{(k)\text{max}2}$ by $z_{j}^{*(k)}$, which is the second most-likely symbol, and the extrinsic message from variable node $j$ to check node $i$ by $z_{i,j}^{(k)}$. If $\sigma_{i,j}^{(k-1)} \neq z_{j}^{(k)}$, then $R_{j}^{(k)\text{max}}$ will not be changed after the contribution
from $\sigma_{i,j}^{(k-1)}$ is excluded. Accordingly, $z_{i,j}^{(k)} = z_{j}^{(k)}$. If $\sigma_{i,j}^{(k-1)} = z_{j}^{(k)}$, three cases need to be considered.

**Case 1**: $R_j^{(k)\text{max}} = R_j^{(k)\text{max}^2}$. In this case, without the contribution of $\sigma_{i,j}^{(k-1)}$, $R_j^{(k)\text{max}}$ would be one less and smaller than $R_j^{(k)\text{max}^2}$. Therefore, $z_{i,j}^{(k)}$ should be $z_j^{(k)}$.

**Case 2**: $R_j^{(k)\text{max}} = R_j^{(k)\text{max}^2} + 1$. Without the contribution of $\sigma_{i,j}^{(k-1)}$, $R_j^{(k)\text{max}} = R_j^{(k)\text{max}^2}$, and hence $z_{i,j}^{(k)}$ can be either $z_j^{(k)}$ or $z'_j^{(k)}$. However, it was found from simulations that setting $z_{i,j}^{(k)}$ to $z_j^{(k)}$ can lead to better performance. Intuitively, this is because that $z_j^{(k)}$ can introduce some disturbance so that the decoder can jump out of loops.

**Case 3**: $R_j^{(k)\text{max}} > R_j^{(k)\text{max}^2} + 1$. Removing the contribution of $\sigma_{i,j}^{(k-1)}$ does not affect that $R_j^{(k)\text{max}} > R_j^{(k)\text{max}^2}$ in this case. Therefore, $z_{i,j}^{(k)}$ should be $z_j^{(k)}$.

Based on these analyses, our proposed E-IHRB algorithm can be described by the pseudo codes in Algorithm G.

Compared to the IHRB algorithm in Algorithm E, the initialization in Algorithm G for the E-IHRB algorithm is different. In addition, the H5 and H6 steps are added to derive extrinsic information. Recording each $z_{i,j}^{(k+1)}$ to be used in the next decoding iteration may cancel out the low-memory advantage of the IHRB algorithm. Instead, we propose to store $z_j^{(k+1)}$ and $z'_j^{(k+1)}$, as well as necessary information with regard to which one of these two should be $z_{i,j}^{(k+1)}$. This scheme leads to significantly lower storage requirement compared to recording each $z_{i,j}^{(k+1)}$, especially when the column weight of $H$ is not small, which is the case in order for the IHRB algorithm to achieve good
Algorithm H: E-IHRB-MLGD algorithm

 Initialization: \( R^{(0)}_{j,l} = \max(\lfloor \varphi_{j,l} \rho \rfloor + \gamma - \max_l(\lfloor \varphi_{j,l} \rho \rfloor), 0) \)

\[ z^{(0)}_{i,j} = z_{j}^{(0)} \]

for \( k = 0 : I_{\text{max}} \)

G1: Stop if \( z^{(k)} H^T = 0 \)

for \( i = 0 \) to \( m-1 \)

for \( j \in N_i \)

G2: \[ \sigma_{i,j}^{(k)} = h_{i,j}^{-1} \sum_{u \in N_i \setminus j} z_{i,u}^{(k)} h_{i,u} \]

H3: if \( (\sigma_{i,j} = \alpha_l) \)

\[ R_{j,l}^{(k)} = R_{j,l}^{(k)} + 1 \]

\[ R_{j,l}^{(k+1)} = R_{j,l}^{(k)} \]

for \( j = 0 \) to \( n-1 \)

H4: \[ R_{j,l}^{(k+1)_{\text{max}}} = \max_l R_{j,l}^{(k+1)} \]

\[ z_j^{(k+1)} = \text{field element of } R_{j,l}^{(k+1)_{\text{max}}} \]

H5: \[ R_{j,l}^{(k+1)_{\text{max} 2}} = \text{second largest } R_{j,l}^{(k+1)} (0 \leq l < q) \]

\[ z_j^{(k+1)} = \text{field element of } R_{j,l}^{(k+1)_{\text{max} 2}} \]

for \( i \in M_j \)

H6: \[ (\sigma_{i,j} = z_j^{(k+1)}) & (R_{j,l}^{(k+1)_{\text{max}}} \leq R_{j,l}^{(k+1)_{\text{max} 2}} + 1) \]

\[ z_{i,j}^{(k+1)} = z_{i,j}^{(k+1)} \]

else \[ z_{i,j}^{(k+1)} = z_{j}^{(k+1)} \]

error-correcting performance. The computation of \( z_{i,j}^{(k+1)} \) will be detailed in Section V.

Fig. 4.1 shows the performance of the E-IHRB algorithm for the \((255, 175)\) cyclic EG-LDPC code over GF\((2^8)\). \( n_m = 16 \) is employed to derive all the curves in this figure. In addition, the same word lengths as listed in Table 4.1 are used for the ISRB and Min-max algorithms. For the IHRB and E-IHRB curves labeled as \( \gamma = 7 \) and 15, the word lengths used are 3-bit and 4-bit, respectively. Moreover, \( \rho = 0.25 \) and 0.5 are used in the simulations.
for the E-IHRB decoding with $\gamma = 7$ and 15, respectively. In the legends, 'init' denotes that the initialization of Algorithm H is employed, and 'extr' represents that extrinsic information is computed from Step H5\~H6 and used in the check node processing at Step G2. It can be observed from Fig. 4.1 that initializing the reliability measures according to the channel information alone can lead to around 0.7dB coding gain. However, larger $\gamma$ needs to be used in order to incorporate the soft information in the initialization. Furthermore, the E-IHRB algorithm employing both the extrinsic v-to-c messages and soft initialization can achieve almost the same performance as the ISRB algorithm for this cyclic EG-LDPC code.

Table 4.1: Word lengths of reliability measures used in simulations

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>IHRB</th>
<th>ISRB</th>
<th>Min-max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word length w (bits)</td>
<td>3</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

84
Fig. 4.2 illustrates the WERs of the E-IHRB algorithm for the (403, 226) QC code over \( GF(2^5) \). Since the column weight is smaller for this code, \( \gamma = 7 \) does not need to be increased in the E-IHRB decoding. Moreover, \( \rho = 0.8 \) is employed in our simulations. As shown in this figure, the E-IHRB algorithm can also achieve significant coding gain over the IHRB algorithm for this QC code. For reference, the performance curve for a (2016, 1133) binary QC-LDPC code using the Min-sum decoding is included in Fig. 4.2. This code has similar rate and length in terms of bits as the (403, 226) QCNB-LDPC code over \( GF(2^5) \). Even though the binary Min-sum decoding is a soft-decision decoding algorithm, its performance is much worse than the E-IHRB algorithm that uses hard message updating for the QCNB-LDPC code.

Compared to the IHRB algorithm, there are two major differences in the ISRB algorithm: soft reliability measure initialization and soft reliability
measure updating. It should be noted that if the ISRB algorithm adopts soft initialization, but not soft updating, it reduces to an algorithm very similar to the E-IHRB algorithm with only soft initialization. Nevertheless, our proposed initialization leads to lower decoder hardware complexity since the maximum reliability measure in a vector is always a constant $\gamma$. Extra performance improvement is achieved in the ISRB algorithm by updating the reliability measures using soft information. However, as explained previously, this leads to significantly longer word length and accordingly larger memory and more complicated clipping logic. Alternatively, extra coding gain can be achieved by using extrinsic information as proposed in our second enhancement scheme to the IHRB decoder, which still updates the reliability measures using hard decisions. In summary, there are analogies between the soft initializations in the ISRB and proposed E-IHRB algorithms. However, extra coding gain over soft initialization can be achieved with much lower hardware complexity using the second enhancement scheme in our proposed E-IHRB algorithm.

Although the computations in the IHRB and E-IHRB algorithms are simpler than those in the ISRB and other soft NB-LDPC decoding algorithms, mapping them directly to hardware still leads to very high complexity, especially when the involved finite field has high order. Efficient partial-parallel architectures for these two decoding algorithms are developed next through algorithmic and architectural modifications for both QC and cyclic NB-LDPC codes.
4.3 Partial-parallel Architectures for IHRB-MLGD

Since the E-IHRB algorithm is based on the IHRB algorithm, this section presents architecture design for the IHRB algorithm first. Then the modifications needed to implement the E-IHRB algorithm are introduced in the next section.

Fully-parallel decoders require overwhelming complexity for NB-LDPC codes that are not very short. On the other hand, serial design can not achieve fast decoding speed. Hence, this chapter considers partial-parallel architectures that can achieve speed-area tradeoff. Many construction schemes for NB-LDPC codes in [36] lead to either cyclic or QC codes. Compared to random codes, these codes enable more efficient partial-parallel processing due to the regularity in the $H$ matrix. In this section, partial-parallel IHRB decoder architectures are developed for these two types of codes.

4.3.1 IHRB-MLGD Architecture for QCNB-LDPC Codes

The $H$ matrix of a QCNB-LDPC code consists of sub-matrices that are either zero or shifted identity matrices with nonzero entries replaced by finite field elements. In our design, one row of sub-matrices of $H$ is processed at a time. Assume that $H$ consists of $r \times t$ sub-matrices of dimension $(q-1) \times (q-1)$. If none of the sub-matrices is zero, then $d_c = t$ and $d_v = r$. The top level architecture of our proposed IHRB decoder for QCNB-LDPC codes is shown in Fig. 4.3(a). RAM $R_i$ $(1 \leq i < q)$ stores reliability measure vectors $R_{i(q-1)b+i-1}^{(k)}$ for $b = 0, 1, \cdots, t-1$, and all the $q$ measures in a vector are stored in the same
address location. A similar scheme is used to store the hard-decision symbols in RAM $z$, except that each RAM $z$ consists of two blocks: one for $z^{(k)}$ and one for $z^{(k+1)}$. Using this storage scheme, the messages for one block column ($q - 1$ columns) of $H$ can be accessed simultaneously.

The $z^{(k)}$ for one block column are sent to the $q - 1$ check node units (CNUs) at a time after permutation and multiplication. The permutation block routes $z^{(k)}$ to proper CNUs based on the locations of the corresponding nonzero entries of $H$, and the multiplication block multiplies the nonzero entries of $H$ to $z^{(k)}$. To simplify computations, the $\sigma_{i,j}$ in step A2 can be rewritten as $z_j^{(k)} + h_{i,j}^{-1} \sum_{u \in N_i} z_u^{(k)} h_{i,u}$. Accordingly, the check sum $s_i^{(k)} = \sum_{u \in N_i} z_u^{(k)} h_{i,u}$ only needs to be computed once for each check node, and can be shared in computing $\sigma_{i,j}$ with different $j$. The CNUs compute these check sums using adder-register loops. After $t$ clock cycles, the check sums are ready and loaded into the registers on the bottom of CNUs, and the check sum computation for
The next block row of $H$ starts.

The check sums are multiplied by $h_{i,j}^{-1}$ in the division block of Fig. 4.3(a) and the products are reversely permuted before they are added up with $z_j^{(k)}$ to compute $\sigma_{i,j}$ in the VNUs, whose architecture is shown in Fig. 4.3(b). One row of sub-matrices is processed at a time in our design, and each sub-matrix has at most one nonzero entry in each column. Hence, at most one of the measures in each $R_j^{(k)}$ vector can be added by one each time. The multiplexor at the output of the memory storing $R_j^{(k)}$ in Fig. 4.3(b) selects $R_j^{(k)}$ if $\sigma_{i,j} = \alpha_l$, and passes it to the adder to be increased by one. In our design, the clipping is done at the same time as the reliability measure updating, and can be simplified since at most one measure in each vector is increased by one at a time. If the $R_j^{(k)}$ for which $\sigma_{i,j} = \alpha_l$ is already $\gamma$ before the addition by one, it should remain unchanged and each of the other nonzero reliability measures in the same vector is subtracted by one. The outputs of the second-row multiplexors in the top part of Fig. 4.3(b) are the updated and clipped reliability measures, except $R_j^{(k)}$, which should come from the 2-to-1 multiplexor in the middle. To address this issue, a binary decoder is employed to convert $\sigma_{i,j}$ to a binary tuple, which is only '1' in the $l$th bit if $\sigma_{i,j} = \alpha_l$. This binary tuple is used as the select signals for the top-row multiplexor.

According to the initialization in Algorithm E and the clipping method adopted, the $j$th hard-decision symbol can be only replaced by $\sigma_{i,j}$ when the corresponding $R_j^{(k)}$ is already $\gamma$ before it is increased by one. Hence, instead of being updated at the end of each decoding iteration, which requires finding the
index of the largest reliability measure in each vector, \( z_j^{(k+1)} \) can be updated using a multiplexor as shown in Fig. 4.3(b) during the processing of each block row of \( H \). The \( q - 1 \) VNUs update the reliability measure vectors and hard-decision symbols for one block column at a time. Hence, the variable node processing (E3 and E4 steps of Algorithm E) for a block row of \( H \) can be completed in \( t \) clock cycles. These computations can overlap with the check node processing for the next block row. Hence, the decoding with \( I_{\text{max}} \) iterations takes around \((1 + rI_{\text{max}})t\) clock cycles in this IHRB decoder for QCNB-LDPC codes.

### 4.3.2 IHRB-MLGD Architecture for Cyclic NB-LDPC Codes

This subsection considers the decoder design for cyclic NB-LDPC codes whose \( H \) matrix consists of a single circulant matrix. Using the construction methods in [36], each row in a cyclic \( H \) is cyclically shifted previous row mul-
tiplied by $\omega$, where $\omega$ is a primitive element of $GF(q)$. However, the nonzero entries in a row appear at irregular locations. If the CNU or VNU has multiple messages to process at a time, the hardware complexity will increase significantly. Hence, the decoder architecture developed previously for QC codes can not achieve efficient partial-parallel processing for cyclic codes. Moreover, large finite fields are usually used for cyclic LDPC codes. In this case, the storage of the reliability measures requires very large memory. In this subsection, a novel shift-message structure is developed to facilitate efficient partial-parallel decoding for cyclic NB-LDPC codes. Moreover, a low-complexity VNU architecture is developed so that only the $n_m < q$ most reliable measures are kept for each vector without causing noticeable performance loss.

Figure 4.5: Computation scheduling in IHRB-MLGD for cyclic codes

Fig. 4.4 shows our proposed IHRB decoder architecture for cyclic NB-LDPC codes. Assume that there are $d$ nonzero entries in the first row of the cyclic $H$, and they are located at positions $p_0, p_1, \ldots, p_{d-1}$. Our design employs $d$ CNUs. Each CNU has the same architecture as that shown in Fig. 4.3(a), and computes a check sum in $d$ clock cycles. However, CNU$_i$ starts one clock cycle after CNU$_{i-1}$ in this cyclic decoder. The hard-decision symbols $z_k$
and $z^{(k+1)}$ are stored in two sets of shift registers, and are cyclically shifted by one position to the left in each clock cycle. Accordingly, all CNUs can read from the same $d$ registers of $z^{(k)}$ located at positions $p_0, p_1 - 1, \cdots, p_{d-1} - (d - 1)$. As a result, the permutation network for routing messages to CNUs is simplified. The connections illustrated in Fig. 4.4 is for an example code with $p_0, p_1, p_2, \cdots = 0, 3, 7, \cdots$. Each multiplexor in this figure sends one of the $d$ hard-decision symbols to the connected CNU at a time after it is multiplied by the corresponding entry in $H$. The select signals of these multiplexors can be generated by counters. Similarly, the counter for the $(i + 1)th$ multiplexor lags behind that for the $ith$ multiplexor by one clock cycle. It takes $d$ clock cycles to compute the first check sum in CNU$_0$. After that, one additional check sum will be available from CNU$_1$, CNU$_2$, $\cdots$ in each clock cycle. Moreover, after a CNU finished computing the check sum for row $i$, it can start the check sum computation for row $i + d$ in the next clock cycle. The scheduling of the check node processing is illustrated in Fig. 4.5. The gray areas in this figure indicate the clock cycles in which the check sums for the corresponding rows are ready.

Our design employs $d$ VNUs. Once the check sum for row $i$ is computed by a CNU, it is sent to all VNUs after multiplied by the corresponding $h^{-1}_{i,j}$, which are done by the top-row multipliers in Fig. 4.4. VNU$_j$ takes care of the reliability measure and hard-decision symbol updating for the $jth$ column of $H$ with nonzero entry in row $i$, and the variable node processing for all nonzero entries in a row is carried out simultaneously by the $d$ VNUs. Since exactly one check sum will be available in each clock cycle after the initial latency,
the reliability measures and hard-decision symbols can be updated based on
the same idea as in the partial-parallel QCNB-LDPC decoder presented in the
previous subsection. The scheduling of the variable node processing is also
shown in Fig. 4.5. The digits in the bar for the VNU$s are the row numbers of
$H$ for which the variable node processing is carried out. Considering the check
node processing latency, each decoding iteration takes $n + d$ clock cycles.

To compute $\sigma_{i,j}$, the VNU$s need to read from the $z^{(k)}$ shift register.
Also the updated hard-decision symbols need to be written back to the $z^{(k+1)}$
shift register. The reliability measures can be shifted together with $z^{(k)}$ and
$z^{(k+1)}$ so that the VNU$s can be also connected to fixed locations of the $z^{(k)}$ and
$z^{(k+1)}$ shift registers. Due to the check node processing latency, by the time
that the check sum for a row of $H$ is available, every hard-decision symbol
used in that check sum computation has been shifted by $d$ positions in the
registers. Therefore, VNU$_j$ needs to access location $(p_j - d) \mod n$ in the $z^{(k)}$
and $z^{(k+1)}$ shift registers. Reliability measure shifting can be also done using
registers. However, each register requires about three times the area of a
memory cell. Storing the large amount of reliability measures in registers may
lead to very high area requirement. On the other hand, storing the measures
in a single RAM prohibits multiple access by the VNU$s. To solve this problem,
we propose to employ $d$ pieces of RAMs and use VNU$s as connecting ports
to "shift" messages from one RAM to another, as illustrated in Fig. 4.4. The
RAM between VNU$_j$ and VNU$_{j+1}$ is of depth $p_{j+1} - p_j$, and the address for
the RAM access can be generated by a mod $p_{j+1} - p_j$ counter. Each time, the
VNUs read messages from the RAMs on their right, and store the updated messages to the RAMs on their left.

When finite fields of high order are used for code construction, storing all $q$ reliability measures in each vector can lead to overwhelming complexity. Fortunately, keeping a small number of the most reliability messages in each vector does not lead to noticeable performance loss if proper compensation schemes are adopted. Fig. 4.6 shows our VNU architecture that keeps $n_m < q$ messages for each vector. In this case, both the reliabilities and corresponding field elements need to be stored, and they are denoted by $R_{v_j}$ and $R_{α_j}$, respectively. The memories in Fig. 4.6 that store these messages are actually from the RAM pieces shown in Fig. 4.4.

At the beginning of the decoding, $R_{v_j}$ is initialized as $γ, 0, 0, \cdots$ and
$R\alpha_j^{(0)}$ is initialized as $z_j^{(0)}, 0, 0, \cdots$. In addition, the $l$th entry of $R_j^{(k)}$ is called empty if $Rv_j^{(k)}_{,l} = 0$. To tell whether each entry is empty, an $n_m$-bit flag vector $f_j^{(k)}$ can be generated by passing each entry of $Rv_j^{(k)}$ through a NOR gate. Accordingly, the flag equals '1' if the corresponding entry is empty. Since not all $q$ messages are kept, $\sigma_{i,j}$ is first compared with each field element in $R\alpha_j^{(k)}$. If there is a match, the corresponding entry in $Rv_j^{(k)}$ is added by one. If there is no match but empty spaces are available, then $\sigma_{i,j}$ is inserted into the first empty space of $R\alpha_j^{(k)}$, and the corresponding entry in $Rv_j^{(k)}$ is increased from zero to one. Otherwise, $\sigma_{i,j}$ is discarded. The GF comparators in Fig. 4.6 consist of $n_m$ copies of bit-wise comparators, each of which outputs a '1' when there is a match. In the case that $\sigma_{i,j}$ equals the field elements in those empty entries, false matching signals will be generated. To block these signals, the comparator outputs are masked by the flag bits. Hence, the output of the OR gate labeled as $A$ in Fig. 4.6 will be only asserted when there is a real match. In addition, the priority encoder takes the flags and generates an $n_m$-bit binary string, such that the $l$th bit equals '1' only if $l$ is the location of the first empty entry. Such an encoder can be implemented with $\log_2 n_m + 1$ gates in the critical path [66]. Based on whether there is a real match, either the masked comparator outputs or the priority encoder outputs are used as the select signals for the last row of multiplexors in Fig. 4.6 to decide whether to keep the same field elements, or insert $\sigma_{i,j}$ into the first empty location of $R\alpha_j^{(k+1)}$.

The AND-OR tree in the top left corner of Fig. 4.6 outputs the relia-
bility measure whose field element matches $\sigma_{i,j}$, or zero if there is no match. The GF comparator outputs instead of masked comparator outputs are used as the select signals in this tree to reduce the critical path. Even if there are false matching signals in the GF comparator outputs, which happen when $\sigma_{i,j}$ equals the field elements in those empty entries, the output of this tree is still zero since the reliability measures for the empty entries are zero. Then the output of the AND-OR tree is added by one, which covers the cases of both reliability measure updating by one when there is a match, and reliability measure setting to one for newly inserted entries. Clipping is done using the same method as that for the VNU architecture in Fig. 4.3(b) for the case that all $q$ messages are kept. Hence, there are similar structures, such as subtractors by one and zero testers followed by multiplexors, in Fig. 4.6. Also the updated reliability measures are routed back to the memory in a similar way as that in Fig. 4.3(b). It is possible that the decoding converges to different symbols in later iterations, but the symbols can not be inserted into the vector because there is no empty space. To solve this problem, we propose to clear the entries whose reliability measures equal one at the end of each decoding iteration. These entries are of the lowest reliability, and the curves in Fig. 4.1 are derived by following this scheme. Such a scheme can be implemented by adding a ”$=1?$” tester for each entry of $Rv_j^{(k)}$. The control block in Fig. 4.6 generates multiplexor select signals based on the results of the testing blocks and whether it is one of the last $d$ rows of $H$ in a decoding iteration. This block can be implemented with very simple logic.
4.4 Partial-parallel Architectures for E-IHRB-MLGD

Compared to the IHRB algorithm, there are two enhancements in the E-IHRB decoding listed in Algorithm H: the initialization and the extrinsic message computation in the H5 and H6 steps. Using different initial values does not require architectural modifications in the decoder except when $\gamma$ is changed. Larger $\gamma$ may have more bits, and hence lead to larger memory for storing the reliability measures in the E-IHRB decoder. Moreover, in the case that $n_m < q$ messages are kept for each vector, the least reliable entries need to be cleared at the end of each decoding iteration to make rooms for symbols that the decoding may converge to later. If $\gamma$ is larger, the entries with larger reliability measures need to be cleared. Therefore, different testers may need to be added for each entry of the $R_{v_j}^{(k)}$ vector in the VNU architecture in Fig. 4.6. For example, when $\gamma$ is increased from 7 to 15 in the E-IHRB decoding for the (255, 175) EG-LDPC code over $GF(2^8)$, it was found that the entries with reliability measure less than or equal to two need to be cleared to achieve almost the same performance as keeping all $q$ messages. In this case, "$=2?" testers are required in the VNUs. Nevertheless, significant modifications need to be made on the IHRB decoder to incorporate the H5 and H6 steps.

As it was explained in Section III, $z_{i,j}^{(k+1)}$ are not stored in order to reduce the memory requirement. Instead, $z_j^{(k+1)}$ and $z_j^{(k+1)}$ are recorded, and one of them is picked to be $z_{i,j}^{(k+1)}$ when needed in decoding iteration $k+1$. The condition testing in Step H6 also requires the knowledge of $\sigma_{i,j}^{(k)}$ and $R_j^{(k+1)max2}$. $R_j^{(k+1)max}$ does not need to be stored since it is always $\gamma$ using the initialization
method in Algorithm H and clipping. Similar to that in the IHRB decoding, \( \sigma_{i,j}^{(k)} \) can be computed from the check sum \( s_i^{(k)} \) and \( z_{i,j}^{(k)} \), which in turn needs to be selected from \( z_j^{(k)} \) and \( z_{j}^{(k)} \). To stop this process from going into infinite depth, flags, \( f_{z_{i,j}} \), are stored to denote whether \( z_j^{(k)} \) or \( z_{j}^{(k)} \) has been selected as \( z_{i,j}^{(k)} \). These flags are updated with Step H6 of Algorithm H in each iteration.

To avoid finding the largest and the second largest reliability measures in each vector at the end of the decoding iteration, \( z_j^{(k+2)} \), \( z_{j}^{(k+2)} \) and \( R_j^{(k+2)max2} \) are also updated with each addition to the reliability measure in Step H3 of iteration \( k+1 \), and the updated values need to be stored. Hence, the decoder needs to record the hard-decision symbols and the second most likely symbols for three consecutive iterations. On the other hand, Step H6 only requires the information if \( R_j^{(k+1)max2} \geq R_j^{(k+1)max} - 1 = \gamma - 1 \). Therefore, instead of using a second copy of memory to store \( R_j^{(k+1)max2} \), flags, \( f_{r_j} \), are recorded to indicate whether this inequality is true. Table 4.2 summarizes the extra variables need to be stored in order to implement the enhancement in Step H5 and H6.

For both QC and cyclic codes, E-IHRB decoder architectures can be developed based on the IHRB decoders presented in the previous section. For example, the E-IHRB decoder for cyclic codes can be implemented by the architecture illustrated in Fig. 4.7. For QC codes, similar modifications can

<table>
<thead>
<tr>
<th>Variable</th>
<th>( z^{(k)} ), ( z^{(k+1)} )</th>
<th>( R^{max2} )</th>
<th>( s )</th>
<th>( f_z )</th>
<th>( f_r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (bits)</td>
<td>4n( \log_2 q )</td>
<td>n( \log_2 \gamma )</td>
<td>m( \log_2 q )</td>
<td>nd, v</td>
<td>n</td>
</tr>
</tbody>
</table>
be made on the architecture in Fig. 4.3(a) to implement E-IHRB decoding. The required modifications are detailed below.

4.4.1 Extra storage

The extra variables listed in Table 4.2 need to be stored in the E-IHRB decoder. To record the extra $z$ and $z'$, copies of the RAM $z$ in Fig. 4.3(a) and shift registers in Fig. 4.7 need to be added for QC and cyclic decoders, respectively. In addition, the flags $fr$ are stored in the same manner as $z$ and $z'$, and one entry is added in the memory for each $R_j$ vector to record $R_{j_{\text{max}}}^2$. The check sums $s_i$ ($0 \leq i < m$) are stored in a single RAM. However, the $fz$ flags are stored in separate memories located in the selector blocks.

Figure 4.7: E-IHRB-MLGD architecture for cyclic NB-LDPC codes
4.4.2 Selector

A selector as shown in Fig. 4.8 needs to be added before each CNU to carry out the selection in Step H6. The selectors can be inserted before the permutation block in Fig. 4.3 for QC decoders, or before the multipliers as shown in Fig. 4.7 for cyclic decoders. The output of the GF comparator in Fig. 4.8 is '1' if $\sigma_{i,j}^{(k)} = z_{j}^{(k+1)}$. It is ANDed with the $fr_j$ flag, which is '1' if $R_j^{(k+1)\max} \geq \gamma - 1$. Hence, the output of the AND gate is used to pick the value for $z_{i,j}^{(k+1)}$, and is the updated $fz_{i,j}$ flag. This flag is stored in RAM, and will be used for picking the value for $z_{i,j}^{(k+1)}$ again when it is used in computing $\sigma_{i,j}^{(k+1)}$ in the next decoding iteration. $z_{i,j}^{(k+1)}$ is also needed in the VNU for computing the extrinsic check sum for the same iteration. To void memory access conflicts, $fz_{i,j}$ is also delayed by a shift register, and then sent to the corresponding VNU in QC decoders. For cyclic decoders, all $z_{i,j}^{(k+1)}$ for $j \in N_i$ are needed in the VNU simultaneously. In this case, the shift registers can
serve as a serial-to-parallel converter. All the updated flag bits $f_{z_i,j}$ with $j \in N_i$ are shifted in serially before they are sent to the VNUs in parallel.

### 4.4.3 VNU modification

During the variable node processing in the $k$th iteration of the E-IHRB decoding, $z_{j}^{(k+1)}$, $z'_{j}^{(k+1)}$, and $R_{j}^{(k+1)max^2}$ need to be updated according to each $\sigma_{i,j}^{(k)}$ and corresponding reliability measure $R_{j,l}^{(k+1)}$ that is available at the point labeled as 'a' in Fig. 4.3(b) and Fig. 4.6. The updating can be done according

![Figure 4.9: VNU architecture modification](image)

Table 4.3: Extra value updating in the E-IHRB decoding

<table>
<thead>
<tr>
<th>$R = \gamma$?</th>
<th>$R = R_{max^2}$?</th>
<th>$\sigma = z$?</th>
<th>updated values</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>$\sigma$ (or $z$) $z'$ $R_{max^2} - 1$</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>$\sigma$ (or $z$) $z'$ $R_{max^2} - 1$</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>$\sigma$ $z$ $R_{max^2} - 1$</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>-</td>
<td>$z$ $\sigma$ $R + 1$</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>-</td>
<td>$z$ $z'$ $R_{max^2}$</td>
</tr>
</tbody>
</table>
to Table 4.3. To simplify the notations, superscripts and subscripts are omitted in this table. The updating of $R_j^{(k+1)max^2}$ can be decided according to whether $R_j^{(k+1)l}$ equals current $R_j^{(k+1)max}$ or $R_j^{(k+1)max^2}$. $R_j^{(k+1)max}$ is always $\gamma$ in our algorithm, and $R_j^{(k+1)max^2}$ can equal $\gamma$ also. In addition, $R_j^{(k+1)max^2}$ should be subtracted by one when clipping needs to be done. To make sure that the updated $z_j^{(k+1)}$ and $z_j^{(k+1)'}$ are not the same, the testing of whether $\sigma_i^{(k)}$ equals current $z_j^{(k+1)}$ is also required.

The updating function in Table 4.3 can be described by Boolean equations, and implemented by the architecture shown in Fig. 4.9. The multiplexors for $z_j^{(k+1)}$ updating in the VNU architectures in Fig. 4.3(b) and Fig. 4.6 should be replaced by this architecture to implement E-IHRB decoding. In Fig. 4.9, the inputs labeled as 'a', 'b' and 'c' come from the signals with the same labels in the VNU architectures shown in Fig. 4.3(b) and Fig. 4.6. These signals are $R_j^{(k+1)l}$, whether $R_j^{(k+1)l} = \gamma$, and $R_j^{(k+1)l} + 1$, respectively. In addition, the flag $fr_j$ indicating whether the updated $R_j^{(k+1)max^2}$ is at least $\gamma - 1$ is also generated by the architecture in Fig. 4.9. As mentioned previously, $\gamma$ is set to an integer in the format of $2^w - 1$ ($w \in \mathbb{Z}^+$) in our design. Accordingly, this test can be implemented by ANDing all bits of the updated $R_j^{(k+1)max^2}$, except the least significant bit.

### 4.5 Complexity Analyses and Comparisons

In this section, the complexities of the proposed decoders are analyzed for example codes. Moreover, comparisons with prior designs and synthesis
results are provided. For cyclic decoders, the (255, 175) EG-LDPC code over $GF(2^8)$ with $d = d_c = d_v = 16$ is chosen. The error-correcting performance of this code is shown in Fig. 4.1. QCNB-LDPC codes can not be constructed to have exactly the same length and rate as cyclic EG-LDPC codes. In addition, relatively high column weight is required for IHRB and E-IHRB algorithms to achieve good error-correcting performance. For the purpose of hardware complexity comparison, a (403, 226) QCNB-LDPC code over $GF(2^5)$ with $d_v = 8$ and $d_c = 13$ is considered. This code has similar length as the (255, 175) cyclic EG-LDPC code over $GF(2^8)$ in terms of bits, but lower rate in order to have relatively high column weight. The performance curves of this QC code can be found in Figs. 2.8 and 4.2.

Table 4.4 lists the complexities of the IHRB and E-IHRB decoders employing both enhancement schemes derived from architectural analysis for the example codes. Each XOR gate can be implemented by eight transistors in CMOS technology. Each 2-input AND or OR gate needs six transistors, and hence requires around $3/4$ the area of an XOR. This assumption is used to derive the equivalent XOR gate count for each functional unit, which is listed in the parentheses in Table 4.4. Also each register with preset and clear needs about three times the area of an XOR. Accordingly, the total numbers of logic gates needed in the decoders can be estimated. The integer comparators and adders used in the architecture in [1] have different word lengths due to the adder tree employed in the VNU. Hence, they have different gate counts. For the purpose of conciseness, the integer comparators and adders with different
word lengths are listed in the same categories, and the ranges of the gate
counts are shown in the parentheses. The row for "other logic gates" refer
to the individual gates, such as the AND, OR, and NOR gates in Fig. 4.6,
that can not be put into any other categories listed in the table. Table 4.4
also lists the total memory, total logic gate, and throughput of each decoder
normalized with respect to those of the Min-max decoder in [1], so that the
relative complexities of different decoders can be easily observed.

For the cyclic IHRB and E-IHRB decoders for the (255, 175) code over
$GF(2^8)$, only the $n_m = 16$ most reliable messages are stored for each vector.
Despite the larger finite field involved, this scheme keeps the memory require-
ment manageable. Moreover, using our proposed architecture, no noticeable
performance loss can be observed compared to keeping all $q = 256$ messages.
$\gamma = 7$ and 15 are adopted for the IHRB and E-IHRB decoders, and hence
each reliability measure in these decoders has three and four bits, respectively.
The longer word length of the reliability measures, as well as the storage of
$R_{max}^2$, the check sums, and the flags, lead to the larger RAM requirement in
the cyclic E-IHRB decoder compared to that in the cyclic IHRB decoder. The
number of registers in the cyclic E-IHRB decoder is around three times of that
in the cyclic IHRB decoder, since registers are used to store hard-decision sym-
bols in cyclic decoders. Moreover, the E-IHRB decoder needs more logic gates
to implement the selector blocks, the modifications in the VNUs, and extra
multiplexors. It can be calculated that the cyclic E-IHRB decoder needs 25%
more memory and almost twice logic gates compared to the cyclic IHRB de-
coder. On the other hand, incorporating only the enhanced reliability measure initialization would lead to 9% memory increase and 4% logic gate increase. Both the cyclic IHRB and E-IHRB decoders require \((255 + 16) \times 25 = 6775\) clock cycles to decode a received word. The critical path of the cyclic IHRB decoder lies in the VNUs shown in Fig. 4.6. It starts with the GF comparators and passes the AND-OR tree in the top left corner, the ”+1” block, the multiplexor in shade, the control unit, and the two multiplexors after. In total, the critical path has 16 gates. In the case of the cyclic E-IHRB decoder, the reliability measure has one more bit, and hence the ’+1’ block has one more gate in the path.

In the IHRB and E-IHRB decoders for the \((403, 226)\) QCNB-LDPC code over \(GF(2^5)\), all \(q = 32\) messages are kept for each vector. In addition, \(\gamma = 7\) is adopted for both decoders. Nevertheless, the hard-decision symbols are stored in RAMs in QC decoders. The larger number of hard-decision symbols that need to be stored causes the memory increase in the QC E-IHRB decoder compared to that in the QC IHRB decoder. Similarly, there are more logic gates in the QC E-IHRB decoder due to the selector blocks, and modified VNUs. In total, the E-IHRB decoder requires 28% more memory and 22% more gates than the IHRB decoder for the \((403, 226)\) QCNB-LDPC code. The area overhead for implementing the enhancement schemes for QC codes is less than that for cyclic codes. This is mainly because that, in the case of QC codes, the extra hard-decision symbols are stored in memories, which usually cost less area than registers. To decode each received word, both
the IHRB and E-IHRB decoders for the (403, 226) QC code require around 
\((1 + 8 \times 25) \times 13 = 2613\) clock cycles. The critical paths of both decoders have 
eleven gates.

For reference, Table 4.5 lists the complexities of the major resources 
needed to implement the proposed decoder architectures expressed in terms 
of code parameters and word length. Although the complexity of "other logic 
gates" in Table 4.4 can not be expressed using a meaningful formula, it only 
accounts for a few percent of the overall decoder complexity. For a given QC 
or cyclic NB-LDPC code and word length, the complexities of the proposed 
IHRB and E-IHRB decoder architectures can be easily estimated from Table 
4.5.

The partial-parallel Min-max decoder architectures in [1, 37, 38] are the 
most efficient in exiting NB-LDPC decoder designs. All of them are for QC 
codes. For reference, these architectures are scaled for the (403, 226) QCNB-
LDPC code, and their complexities are also listed in Table 4.4. It can be 
observed from this table that the IHRB and E-IHRB decoders require only a 
fraction of the area of the Min-max decoders, although they can not achieve 
as good performance as the Min-max decoders for the cyclic code as shown 
in Fig. 4.2. Using extrinsic information as proposed in the second enhance-
ment scheme can only achieve small additional coding gain over the first en-
hancement. Nevertheless, the E-IHRB decoders adopting both enhancement 
schemes are still valuable for applications that require better performance, 
but can not afford the complexity of Min-max or other BP-based decoders.
The much smaller area requirement of the IHRB and E-IHRB decoders can be mainly attributed to smaller memory, which dominates the overall decoder area. In addition, the latency of the Min-max decoding is more than ten times longer for the same QC code, although less number of iterations are required to converge. The reason is that the messages in a vector cannot be processed simultaneously without causing large area in the Min-max decoders. The performance of the IHRB and E-IHRB decoding can be improved by increasing the column weight $d_v$. In this case, the Min-max decoders will be even less efficient, since their memory requirements increase fast with $d_v$. On the other hand, the memory requirement of the IHRB decoder is independent of $d_v$ and that of the E-IHRB decoder has only a very small part increasing with $d_v$. It should be noted that the IHRB and E-IHRB decoders have limitations. Their error-correcting performance deteriorates when $d_v$ is small, and error-floor can show up in low signal-to-noise ratio region. However, many practical applications demand high-rate codes, which usually have lower column weight.

FPGA synthesis results of a QC Min-max decoder were reported in [37]. [1] provided synthesis results using CMOS technology, and no synthesis result was included in [38]. To further evaluate our design, the proposed IHRB and E-IHRB decoders for the (403, 226) QC code are modeled using Verilog and implemented on a Xilinx Virtex-5 XC5VLX200t device. In addition, the design in [37] is re-implemented using the same device for the same code, and the results after place and route are listed in Table 4.6. It can be calculated that the Min-max decoder in [37] requires around 7.98 times the slice look-up tables.
(LUTs) in the IHRB decoder, and its throughput is around 34 times lower. These ratios are very close to the relative logic gate numbers and latencies of these two decoders analyzed in Table 4.4. The ratio of the numbers of Block RAMs used in these two decoders does not equal the ratio of the RAM sizes listed in Table 4.4 due to the reason that most of the Block RAMs are not fully occupied. Each Block RAM has 36k bits in Virtex-5 devices. On average, 25% of each Block RAM is occupied in the Min-max decoder. Nevertheless, each RAM R has 1248 bits and each part of RAM z has only 65 bits in the IHRB decoder for the (403, 226) code. RAM R and RAM z are implemented using Block RAMs in this work to show separate complexities for the RAM and logic parts. Small RAMs can be also implemented as distributed RAMs on FPGAs to reduce the Block RAM usage. It can be also calculated from Table 4.6 that the ratio of the slice LUTs in the IHRB and E-IHRB decoders is very close to the ratio of the non-RAM complexities in these decoders estimated from Table 4.4.

To compare with the results in [1], our proposed decoders are also synthesized using SMIC 0.18μm CMOS process under 250Mhz clock frequency. The results are listed in Table 4.7. The gate count is derived through dividing the total area by the area of a individual gate. It also includes the complexity of interconnect. Our memory compiler and library are not optimized. Each memory cell has eight transistors, and only memories with depth in the format of $2^e$ ($e \geq 4$) can be generated. Moreover, there is large area overhead for memories with smaller depth. However, memory dominates the
area of the proposed decoders. If more optimized memory compiler and library are available, our proposed design will occupy significantly smaller area. In [1], synthesis results are reported for a (620, 310) QCNB-LDPC code over $GF(2^5)$ with $d_v = 3$ and $d_c = 6$ under 200Mhz clock frequency using the same process. Synthesis results cannot be scaled according to code parameters directly. When the architecture in [1] is applied to the (403, 226) code instead of the (620, 310) code, the decoder has the following differences: i) The (403, 226) decoder has 62 less CNUs since the corresponding $H$ matrix has 62 less rows; ii) It also needs $q - 1 = 31$ VNUs. However, the complexity of each VNU for the (403, 226) decoder is higher due to the larger column weight; iii) The memory requirement is increased from 694k in the (620, 310) decoder to 1096k in the (403, 226) decoder due to the larger column weight. Considering these issues, the area of the (403, 226) decoder will be similar or higher than that reported in [1] for the (620, 310) decoder. In addition, the throughput will be dropped to 15Mbps in the (403, 226) decoder when 15 iterations are carried out. Hence, from CMOS synthesis results, the proposed IHRB and E-IHRB decoders can still achieve several hundred times higher efficiency in terms of throughput-over-area ratio than the Min-max decoder in [1] for the same QCNB-LDPC code even though our memory compiler and library are not optimized.
4.6 Summary

This chapter proposed enhancement schemes to the IHRB decoding algorithm for NB-LDPC codes. The proposed schemes lead to significant coding gain with small complexity overhead. In addition, efficient architectures were developed for both QC and cyclic NB-LDPC codes based on the IHRB and E-IHRB algorithms. With moderate performance loss, the proposed decoders can achieve at least tens of times higher efficiency compared to previous designs based on the Min-max algorithm.
<table>
<thead>
<tr>
<th></th>
<th>IHRB cyclic (255, 175) code</th>
<th>E-IHRB cyclic (255, 175) code</th>
<th>IHRB QC (403, 226) code</th>
<th>E-IHRB QC (403, 226) code</th>
<th>Min-max[1]</th>
<th>Min-max[37]</th>
<th>Min-max[38]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$GF(2^8)$</td>
<td>$GF(2^8)$</td>
<td>$GF(2^5)$</td>
<td>$GF(2^5)$</td>
<td>$GF(2^5)$</td>
<td>$GF(2^5)$</td>
<td>$GF(2^5)$</td>
</tr>
<tr>
<td></td>
<td>$d = 16$</td>
<td>$d = 16$</td>
<td>$d_v = 8, d_c = 13$</td>
<td>$d_v = 8, d_c = 13$</td>
<td>$d_v = 8, d_c = 13$</td>
<td>$d_v = 8, d_c = 13$</td>
<td>$d_v = 8, d_c = 13$</td>
</tr>
<tr>
<td></td>
<td>25 iterations</td>
<td>25 iterations</td>
<td>25 iterations</td>
<td>25 iterations</td>
<td>15 iterations</td>
<td>15 iterations</td>
<td>15 iterations</td>
</tr>
<tr>
<td></td>
<td>$w = 3$ bits</td>
<td>$w = 4$ bits</td>
<td>$w = 3$ bits</td>
<td>$w = 3$ bits</td>
<td>$w = 5$ bits</td>
<td>$w = 5$ bits</td>
<td>$w = 5$ bits</td>
</tr>
<tr>
<td>RAM</td>
<td>44.8k</td>
<td>56.1k</td>
<td>42.7k</td>
<td>56.8k</td>
<td>1096.0k</td>
<td>855.7k</td>
<td>308.0k</td>
</tr>
<tr>
<td>ROM</td>
<td>0.25k</td>
<td>0.25k</td>
<td>7.2k</td>
<td>7.2k</td>
<td>32.2k</td>
<td>13.9k</td>
<td>13.9k</td>
</tr>
<tr>
<td>total memory</td>
<td>45.05k</td>
<td>56.35k</td>
<td>49.9k</td>
<td>61.0k</td>
<td>1128.2k</td>
<td>869.6k</td>
<td>321.9k</td>
</tr>
<tr>
<td>total memory (normalized)</td>
<td>0.040</td>
<td>0.050</td>
<td>0.044</td>
<td>0.057</td>
<td>1.000</td>
<td>0.771</td>
<td>0.285</td>
</tr>
<tr>
<td>register</td>
<td>4336</td>
<td>13007</td>
<td>465</td>
<td>620</td>
<td>280953</td>
<td>39990</td>
<td>3581.5</td>
</tr>
<tr>
<td>GF multiplier (# of XORs)</td>
<td>32(100)</td>
<td>48(100)</td>
<td>62(43)</td>
<td>93(43)</td>
<td>248(43)</td>
<td>16(43)</td>
<td>16(43)</td>
</tr>
<tr>
<td>GF adder (# of XORs)</td>
<td>32(8)</td>
<td>48(8)</td>
<td>62(5)</td>
<td>93(5)</td>
<td>0</td>
<td>93(5)</td>
<td>155(5)</td>
</tr>
<tr>
<td>2:1 1-bit Multiplexors</td>
<td>9752</td>
<td>15128</td>
<td>10540</td>
<td>11561</td>
<td>317440</td>
<td>18905</td>
<td>23090</td>
</tr>
<tr>
<td>+1,-1 block (# of XORs)</td>
<td>27(3)</td>
<td>288(5)</td>
<td>1023(3)</td>
<td>1054(3)</td>
<td>0</td>
<td>372(5)</td>
<td>0</td>
</tr>
<tr>
<td>=0?, =1?, $\geq \gamma_?$, $\geq \gamma - 1?$ tester (# of XORs)</td>
<td>528(2)</td>
<td>544(3)</td>
<td>1023(2)</td>
<td>1054(2)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>=? tester (# of XORs)</td>
<td>0</td>
<td>16(6)</td>
<td>0</td>
<td>31(5)</td>
<td>0</td>
<td>186(6)</td>
<td>0</td>
</tr>
<tr>
<td>GF comparator (# of XORs)</td>
<td>256(13)</td>
<td>288(13)</td>
<td>0</td>
<td>62(8)</td>
<td>0</td>
<td>2480(8)</td>
<td>1984(8)</td>
</tr>
<tr>
<td>integer comparator (# of XORs)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>62(8)</td>
<td>0</td>
<td>2480(8)</td>
<td>1984(8)</td>
</tr>
<tr>
<td>integer adder (# of XORs)</td>
<td>5648</td>
<td>5972</td>
<td>5030</td>
<td>6028</td>
<td>8432(30~54)</td>
<td>124(30)</td>
<td>124(30)</td>
</tr>
<tr>
<td>other logic gates</td>
<td>37.2k</td>
<td>72.2k</td>
<td>25.1k</td>
<td>30.7k</td>
<td>2018.0k</td>
<td>195.5k</td>
<td>188.7k</td>
</tr>
<tr>
<td>total logic gates (# of XORs)</td>
<td>0.018</td>
<td>0.056</td>
<td>0.012</td>
<td>0.015</td>
<td>1.000</td>
<td>0.097</td>
<td>0.094</td>
</tr>
<tr>
<td>total logic gates (normalized)</td>
<td>0.018</td>
<td>0.056</td>
<td>0.012</td>
<td>0.015</td>
<td>1.000</td>
<td>0.097</td>
<td>0.094</td>
</tr>
<tr>
<td>critical path (# of gates)</td>
<td>17</td>
<td>17</td>
<td>11</td>
<td>11</td>
<td>18</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td># of clock cycles</td>
<td>6775</td>
<td>6775</td>
<td>2613</td>
<td>2613</td>
<td>26820</td>
<td>69472</td>
<td>69472</td>
</tr>
<tr>
<td>throughput (normalized)</td>
<td>4.46</td>
<td>4.19</td>
<td>16.81</td>
<td>16.81</td>
<td>1.00</td>
<td>0.46</td>
<td>0.46</td>
</tr>
</tbody>
</table>
Table 4.5: Generalized complexity analyses for IHRB and E-IHRB decoders

<table>
<thead>
<tr>
<th></th>
<th>IHRB cyclic decoder</th>
<th>E-IHRB cyclic decoder</th>
<th>IHRB QC decoder</th>
<th>E-IHRB QC decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>$(\lceil \log_2 q \rceil + w)n_m$</td>
<td>$(\lceil \log_2 q \rceil + w)n_m$</td>
<td>$nqw + 2n\lceil \log_2 q \rceil$</td>
<td>$n(q+1)w + (d_v + 1)n + (6n + m)\lceil \log_2 q \rceil$</td>
</tr>
<tr>
<td>ROM</td>
<td>$2d\lceil \log_2 q \rceil$</td>
<td>$2d\lceil \log_2 q \rceil$</td>
<td>$2d_v dc\lceil \log_2 q \rceil + 2(q-1)$</td>
<td>$(d_v + d_c - 1)\lceil \log_2 q \rceil$</td>
</tr>
<tr>
<td>register</td>
<td>$2(d + n)\lceil \log_2 q \rceil$</td>
<td>$(4d + 6n)\lceil \log_2 q \rceil + n$</td>
<td>$3(q - 1)\lceil \log_2 q \rceil$</td>
<td>$4(q - 1)\lceil \log_2 q \rceil$</td>
</tr>
<tr>
<td>GF multiplier</td>
<td>$2d$</td>
<td>$3d$</td>
<td>$2(q - 1)$</td>
<td>$3(q - 1)$</td>
</tr>
<tr>
<td>GF adder</td>
<td>$2d$</td>
<td>$3d$</td>
<td>$2(q - 1)$</td>
<td>$3(q - 1)$</td>
</tr>
<tr>
<td>2:1 1-bit Muxes</td>
<td>$\lceil \log_2 q \rceil((d-1)(d+1) + 2n + dn_m) + w(2n_m + 1)d$</td>
<td>$\lceil \log_2 q \rceil((d-1)(d+1) + 4n + dn_m + 5) + w(2n_m + 3)d$</td>
<td>$(q-1)((\lceil \log_2 q \rceil (2)\lceil \log_2 q \rceil + 1) + w(3q - 1))$</td>
<td>$(q-1)((\lceil \log_2 q \rceil (2)\lceil \log_2 q \rceil + 6) + w(3q + 1))$</td>
</tr>
<tr>
<td>+1,-1 block</td>
<td>$(n_m + 1)d$</td>
<td>$(n_m + 2)d$</td>
<td>$(q - 1)(q + 1)$</td>
<td>$(q - 1)(q + 2)$</td>
</tr>
<tr>
<td>=0?, =1?, ≥ γ - 1? tester</td>
<td>$(2n_m + 1)d$</td>
<td>$(2n_m + 2)d$</td>
<td>$(q - 1)(q + 1)$</td>
<td>$(q - 1)(q + 2)$</td>
</tr>
<tr>
<td>=? comparator</td>
<td>0</td>
<td>0</td>
<td>$q - 1$</td>
<td>$q - 1$</td>
</tr>
<tr>
<td>GF comparator</td>
<td>$n_m d$</td>
<td>$(n_m + 2)d$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>critical path(# of gates)</td>
<td>$\lceil \log_2 \lceil \log_2 q \rceil \rceil + \lceil \log_2 n_m \rceil + w + 6$</td>
<td>$\lceil \log_2 \lceil \log_2 q \rceil \rceil + \lceil \log_2 n_m \rceil + w + 6$</td>
<td>$\lceil \log_2 q \rceil + \lceil \log_2 (q - 1) \rceil + 1$</td>
<td>$\lceil \log_2 q \rceil + \lceil \log_2 (q - 1) \rceil + 1$</td>
</tr>
<tr>
<td># of clock cycles</td>
<td>$(d + n)I_{max}$</td>
<td>$(d + n)I_{max}$</td>
<td>$d_v(1 + d_v I_{max})$</td>
<td>$d_v(1 + d_v I_{max})$</td>
</tr>
</tbody>
</table>
Table 4.6: FPGA Implementation results of (403, 226) QCNB-LDPC decoders on Xilinx Virtex-5 XC5VLX200t Device

<table>
<thead>
<tr>
<th></th>
<th>IHRB</th>
<th>E-IHRB</th>
<th>Min-max [37]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice LUTs</td>
<td>7841</td>
<td>9153</td>
<td>62605</td>
</tr>
<tr>
<td>Slice Flip Flops</td>
<td>529</td>
<td>716</td>
<td>38660</td>
</tr>
<tr>
<td>BRAMs</td>
<td>56</td>
<td>71</td>
<td>95</td>
</tr>
<tr>
<td>Max. Freq. (Mhz)</td>
<td>117.6</td>
<td>109.8</td>
<td>90.9</td>
</tr>
<tr>
<td>Throughput (Mbps)</td>
<td>90.68</td>
<td>84.67</td>
<td>2.63</td>
</tr>
</tbody>
</table>

Table 4.7: Synthesis results using SMIC 0.18μm CMOS technology

<table>
<thead>
<tr>
<th></th>
<th>IHRB QC (403, 226) code $d_v = 8, d_c = 13$ 25 iterations</th>
<th>E-IHRB QC (403, 226) code $d_v = 8, d_c = 13$ 25 iterations</th>
<th>Min-max [1] (620, 310) code $d_v = 3, d_c = 6$ 10 iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gate count</td>
<td>0.45M</td>
<td>0.59M</td>
<td>14.9M</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>250Mhz</td>
<td>250Mhz</td>
<td>200Mhz</td>
</tr>
<tr>
<td>Throughput</td>
<td>193Mbps</td>
<td>193Mbps</td>
<td>60Mbps</td>
</tr>
</tbody>
</table>
Chapter 5

Power Representation Based Decoding Algorithm and Architecture

5.1 Introduction

In this chapter, novel check node processing schemes are proposed for both the forward-backward based Min-max algorithm and the SMSA by making use of the power representation of finite field elements. If a vector consists of all nonzero finite field elements ordered according to the exponents of their power representations, then multiplying the vector by a nonzero field element is equivalent to a cyclic shift of the elements in the vector. By making use of this property, the switching network in the forward-backward Min-max CNU can be eliminated through shifting the input and output vectors in the elementary steps of the check node processing. Moreover, an innovative scheme is developed to incorporate the multiplications by the parity check matrix entries into the proposed check node processing [41]. A novel technique is proposed to reduce the latency overhead caused by shifting the vectors through scaling the rows of the parity check matrix, which does not affect the error correcting performance of the decoding algorithm. Power representation of field elements is also exploited in this chapter to implement the CNU for the SMSA. Compared to the previous SMSA CNU in [42], the proposed design consists of
identical hardware units and is more regular. It is also partial-parallel and more suitable to applications with limitations on chip area.

For codes over $GF(32)$, synthesis results show that the proposed CNU for the Min-max algorithm with forward-backward scheme achieves 32% area reduction and 14% higher clock frequency compared to the CNU architecture in [1], despite a very small possible increase in the clock cycles. It can achieve even more significant saving over a folded version of the CNU in [2].

5.2 Check Node Processing for the Min-max Algorithm

5.2.1 Min-max Check Node Processing Based on Power Representation

In our design, the LLRs in a vector are stored according to their corresponding finite field elements in power representation in the order of $1, \alpha, \ldots, \alpha^{q-2}, 0$, where $\alpha$ is a primitive element of $GF(q)$. Multiplying a vector that consists of all nonzero field elements, $[1, \alpha, \ldots, \alpha^{q-2}]$, by a nonzero field element is equivalent to cyclically shifting the vector. Moreover, $\beta = \beta_1 + \beta_2$ if and only if $\alpha \beta = \alpha \beta_1 + \alpha \beta_2$. Hence, the elementary step of the forward-backward Min-max check node processing can be implemented equivalently as $L_o(\alpha \beta) = \min_{\beta = \beta_1 + \beta_2} (\max(L_1(\alpha \beta_1), L_2(\alpha \beta_2)))$. In another word, if both input LLR vectors are cyclically shifted by one position, the output vector under the same constraint will be also cyclically shifted by one position. Making use of this property, the connections of the ‘min’ and ‘max’ units can remain the same, and accordingly the message switching network can be eliminated,
if the input and output vectors are cyclically shifted by the same number of positions. Note that the LLR corresponding to zero field element should not be included in the cyclical vector shifting.

Previously, finite field elements are multiplied/divided by the nonzero entries of $H$ using separate units outside of the CNU. The multiplications and divisions are done by finite field multipliers in [1], and barrel shifters in [2]. When power representation of finite field elements is adopted, an alternative approach is to cyclically shift the input and output vectors using shift registers. However, this would require extra clock cycles for each elementary step in the forward-backward processing when the corresponding shifting can not be overlapped with other computations.

Next, a novel method is proposed to efficiently incorporate the multiplications/divisions into the Min-max check node processing. Since the check node number $m$ does not change during the corresponding check node processing, it is dropped from the notations when no ambiguity occurs. Assume that the $i$th nonzero entry in row $m$ of $H$ is $\alpha^{h_i}$. Moreover, for the field element, $\beta$, used as the index of the message vectors for variable node $n_i$, such as $u_{m,n_i}$ and $f_i$, define $\beta = \alpha^{h_i} \beta'$. Then the forward process should instead compute

$$f_i(\beta) = \min_{\beta = \beta_1 + \alpha^{h_i} \beta_2} (\max(f_{i-1}(\beta_1), u_{m,n_i}(\beta_2')))$$

(5.1)

if the original v-to-c message vector, $u_{m,n_i}(\beta_2')$, is directly sent to the CNU. Thus, the multiplications have been incorporated in the forward process. The constraint in (5.1) can be also expressed as $\alpha^{h_i} \beta' = \alpha^{h_i-1} \beta_1 + \alpha^{h_i} \beta_2'$. As a
result, the forward computation can be done equivalently as

\[ f_i(\beta') = \min_{\beta' = \alpha^{(h_{i-1} - h_i)\beta'_1 + \beta'_2}} (\max(f_{i-1}(\beta'_1), u_{m,n_i}(\beta'_2))). \] (5.2)

Similarly, the elementary step of the backward process becomes

\[ b_i(\beta') = \min_{\beta' = \alpha^{(h_{i+1} - h_i)\beta'_1 + \beta'_2}} (\max(b_{i+1}(\beta'_1), u_{m,n_i}(\beta'_2))). \] (5.3)

Note that, using power representation, multiplying a field element to any \( \beta \) variable in the constraints of the above equations translates to cyclically shifting the corresponding vector. Hence, the shifting of the original v-to-c message vectors as required in (5.1) is switched to the shifting of intermediate message vectors if the modified forward and backward processes as in (5.2) and (5.3) are carried out. Moreover, the constraint for the first elementary step of the forward process used to be \( \beta = \alpha^{h_0} \beta'_1 + \alpha^{h_1} \beta'_2 \) if the v-to-c messages are sent to the CNU directly. Similarly, the constraint can be converted to \( \beta' = \alpha^{-h_0 - h_1} \beta'_1 + \beta'_2 \), in which case the shifting of the \( u_{m,n_i} \) vector is no longer needed. The same saving can be also achieved in the first elementary step of the backward process.

Compared to the forward computation according to (5.1), the modifications proposed so far do not lead to much reduction on vector shifting, which translates to barrel shifters or extra clock cycles if shift registers are used. However, the proposed modifications enable the elimination of the shifting in the merging process. The c-to-v vector \( v_{m,n_i} \) after considering the division can be computed as

\[ v_{m,n_i}(\beta') = \min_{\alpha^{-h_i} \beta' = \beta_1 + \beta_2} (\max(f_{i-1}(\beta_1), b_{i+1}(\beta_2))). \]
In this case, the vector derived from merging the intermediate vectors from the forward and backward processes needs to be cyclically shifted by $-h_i$ positions before it becomes the c-to-v message vector. Alternatively, it can be derived that

$$v_{m,n_i}(\beta') = \min_{\beta' = \alpha^{(h_{i-1}-h_i)}\beta'_1 + \alpha^{(h_{i+1}-h_i)}\beta'_2} (\max(f_{i-1}(\beta'_1), b_{i+1}(\beta'_2))).$$

Therefore, the c-to-v message vectors will be directly available after merging without any shifting, if the intermediate vectors that have been derived during the modified forward and backward processes as shown in (5.2) and (5.3) are cyclically shifted by $(h_{i-1} - h_i)$ and $(h_{i+1} - h_i)$ positions, respectively. Inspired by this, $f_{i-1}(\beta'') = f_{i-1}(\alpha^{(h_{i-1}-h_i)}\beta')$ and $b_{i+1}(\beta'') = b_{i+1}(\alpha^{(h_{i+1}-h_i)}\beta')$ are stored as intermediate results in our design. In this case, the proposed elementary steps of the forward-backward process that incorporate the multiplications and divisions are

$$f_i(\beta') = \min_{\beta' = \beta'_1 + \beta'_2} (\max(f_{i-1}(\beta''_1), u_{m,n_i}(\beta''_2)))$$

$$b_i(\beta') = \min_{\beta' = \beta'_1 + \beta'_2} (\max(b_{i+1}(\beta''_1), u_{m,n_i}(\beta''_2)))$$

$$v_{m,n_i}(\beta') = \min_{\beta' = \beta'_1 + \beta'_2} (\max(f_{i-1}(\beta''_1), b_{i+1}(\beta''_2))).$$

(5.4)

It can be observed that (5.4) bears the same form as the equations for the original forward-backward process, and no further shifting is needed after the merging process. Note that the shifting of the intermediate results does not require extra latency except in the first and the last elementary step. This will be detailed in the next subsection.
5.2.2 Min-max CNU Architectures Based on Power Representation

The proposed scheme can be applied to both the Min-max CNU architectures in [1] and [2] to reduce the complexity. Moreover, the multiplication modules of the corresponding decoders can be eliminated. When applied to the design in [1], the elementary step of the check node processing for $GF(8)$ can be implemented using the architecture shown in Fig. 5.1. In this architecture, one single column of $q$ multiplexors replaces the $q \log_2 q$ multiplexor switching network between the 'min' and 'max' units in the design of [1], although the $q$-to-1 multiplexor shown in grey is added. As a result, the area and critical path are both reduced.

The proposed CNU in Fig. 5.1 finishes one elementary step in $q$ clock cycles. At the beginning of each step, both $L_1$ and $L_2$ input vectors are loaded into registers, and they are stored in the order of $1, \alpha, \ldots, \alpha^{q-2}, 0$ according to
power representation. Then the LLRs in $L_1$ are sent to the 'max' units serially. In each clock cycle, the LLR from $L_1$ is simultaneously compared with each of the LLRs from $L_2$ in the 'max' units. Then the larger values are routed to the 'min' units according to the constraint, and compared with the previous minimum LLRs of the same field elements. The shifting of the LLRs in $L_1$ according to the constraint has already been taken care of before they are sent to the 'max' units. Hence, the constraint for deciding the routing between the 'max' and 'min' units is in the format of $\beta' = \beta''_1 + \beta''_2$. The routing is only implemented for the constraint in the first clock cycle, i.e. $\beta' = 1 + \beta''_2$. For example, assume that the irreducible polynomial used to construct $GF(8)$ is $x^3 + x + 1$. Then $1 + \alpha^2 = \alpha^6$. Hence, the output of the third 'max' is routed to the input of the seventh 'min'. In the following $q - 2$ clock cycles, both $L_2$ and $L_o$ vectors are cyclically shifted by one position at a time, and hence the routing can remain unchanged. In the $q$th clock cycle, the LLR input from $L_1$ has zero field element. Zero plus any field element equals that field element. Hence, the output of each 'max' unit should be sent to the 'min' unit in the same row. This modification on the routing can be done through changing the select signals for the 2-to-1 multiplexor column from '1' in the first $q - 1$ clock cycles to '0' in the last clock cycle. The last registers in the $L_2$ and $L_o$ columns hold the LLRs with zero field elements, and do not participate in the shifting.

At the beginning of each elementary step of the forward and backward processes, the $L_o$ from the previous step is loaded into the shift registers on the
left of Fig. 5.1 to become the $L_1$ for the current step. If $L_1$ needs to be shifted by $x$ positions before it is sent to the 'max' units, the $q$-to-1 multiplexor selects the output of the $x$th register. Moreover, $L_1$ is shifted by one position in the registers in each clock cycle, and hence the select signal for the multiplexor does not change. After $L_1$ is shifted by $x$ positions, the whole vector is stored into memory to be used in the merging process. In this way, the shifting of the intermediate results shown in the constraints of (5.2) does not introduce extra clock cycles. Similar shifting is done on the intermediate vectors from the backward process. As a result, the merging process does not require any vector shifting at the beginning or end of the elementary steps to generate the c-to-v message vectors. However, the output vectors from the last elementary steps of the forward and backward processes need to be shifted by $h_{d_c-2}-h_{d_c-1}$ and $h_1 - h_0$ positions, respectively, to become c-to-v message vectors. These shifting requires $(h_1 - h_0) \mod (q - 1) + (h_{d_c-2} - h_{d_c-1}) \mod (q - 1)$ extra clock cycles using the CNU in Fig. 5.1 if they can not be overlapped with other computations.

The proposed method can be also applied to simplify the CNU architecture in [2]. Our design for $GF(8)$ is shown in Fig. 5.2. It finishes one elementary step in $q$ clock cycles. Alternatively, copies of the max-min trees can be connected to the input vectors in similar ways and compute multiple output messages simultaneously. In the case that $q$ max-min trees are adopted, the proposed architecture finishes one elementary step in one clock cycle, and looks almost the same as that in [2]. However, a fully-parallel CNU has large
area, and [2] did not consider how the CNU should be folded in the case that the area requirement needs to be reduced. Since the routing of the input vectors to the $q$ copies of max-min trees are all different in the design of [2], a folded version of this architecture would need a large switching network in the dashed block area of Fig. 5.2 if the input vectors are not shifted. Moreover, [2] assumed that layered LDPC decoding is adopted, so that the shifting of the output vectors from the merging process can be combined with the shifting of the v-to-c vectors for the next layer. Thus, one barrel shifter is required at the input of the CNU. If non-layered decoding is used, an extra barrel shifter needs to be added to the output of the CNU in [2].

The input and output vectors are stored in registers in the architecture shown in Fig. 5.2. For $GF(q)$, there are exactly $q$ pairs of field elements
whose sums equal a given value. The pairs of LLRs whose corresponding field elements added up to be $\alpha^0 = 1$ are sent to the $q$ 'max' units simultaneously in the first clock cycle. Taking $GF(8)$ constructed using the irreducible polynomial $x^3 + x + 1$ as an example, the pairs of field elements that added up to be 1 are $0 + 1$, $\alpha^3 + \alpha$, $\alpha^6 + \alpha^2$, $\alpha + \alpha^3$, $\alpha^5 + \alpha^4$, $\alpha^4 + \alpha^5$, $\alpha^2 + \alpha^6$, and $1 + 0$. The first terms in these expressions are the field elements from $L_1$ and the second are from $L_2$. Then the tree of 'min' units generates the minimum among the larger LLRs corresponding to these pairs, which is $L_o(1)$. Both input vectors are cyclically shifted in each clock cycle. Then in the second clock cycle, the pairs of field elements corresponding to the LLRs input to the 'max' units become $0 + \alpha$, $\alpha^4 + \alpha^2$, $1 + \alpha^3$, $\alpha^2 + \alpha^4$, $\alpha^6 + \alpha^5$, $\alpha^5 + \alpha^6$, $\alpha^3 + 1$, and $\alpha + 0$. Hence, the connections between the input vectors and the 'max' units remain the same, and $L_o(\alpha)$ is computed. Similarly, $L_o(\alpha^2), \ldots, L_o(\alpha^6)$ are computed serially in the following clock cycles. The field elements and those in the parentheses by the 'max' units in Fig. 5.2 label the LLRs sent to the 'max' units in the first and second clock cycles, respectively. Moreover, $L_o(0)$ is computed iteratively using the units on the bottom right corner of Fig. 5.2. In the first $q - 1$ clock cycles, they take one pair of LLRs with the same nonzero field elements at a time from the shifting input vector registers. Then, in the last clock cycle, the pair of LLRs with zero field elements are processed. Therefore, a total of $q$ clock cycles are required for the process of one elementary step of the proposed architecture.

During the forward and backward processes, the $L_o$ computed in the
previous elementary step is loaded into the $L_1$ registers at the beginning of the current step. Similarly, $L_1$ also needs to be shifted before it is stored into the memory or sent to the ’max’ units. The shifting can be done through controlling the writing locations of the LLRs in the $L_o$ registers in the previous elementary step. If $L_1$ needs to be shifted by $x$ locations, then in the previous step, each LLR in $L_o$ is written into the $x$th register, while the registers are cyclically shifted by one position in each clock cycle. Using this scheme, the vector shifting does not require extra latency, except in the first elementary steps of the forward and backward processes. Therefore, the latency overhead of the proposed CNU in Fig. 5.2 is also $(h_1 - h_0) \mod (q - 1) + (h_{d_{c-2}} - h_{d_{c-1}}) \mod (q - 1)$ if the shifting can not be overlapped with other computations.

5.2.3 Reducing Shifting Latency Overhead

In this subsection, a novel method is proposed to reduce the extra clock cycles required for shifting the message vectors in the proposed CNUs.

Multiplying a finite field element to every entry in a row of the parity check matrix $H$ does not change the constraints that need to be satisfied in the check node processing. Making use of this property, a constant $\alpha^\delta$ can be multiplied to a row of $H$ in order to minimize the extra clock cycles needed for shifting the message vectors without affecting the error correcting performance. For simplicity, $(h_1 - h_0) \mod (q - 1)$ and $(h_{d_{c-2}} - h_{d_{c-1}}) \mod (q - 1)$ are denoted by $\Delta h_0$ and $\Delta h_{d_{c-1}}$, respectively. Both $\Delta h_0$ and $\Delta h_{d_{c-1}}$ are nonnegative, and two cases need to be considered.
Table 5.1: Latency overhead reduction examples for codes over $GF(32)$

<table>
<thead>
<tr>
<th>row of $H$</th>
<th>$h_1 - h_0$, $h_{d-2} - h_{d-1}$</th>
<th>$\Delta h_0$, $\Delta h_{d-1}$</th>
<th>Original latency</th>
<th>case</th>
<th>$\delta$</th>
<th>Reduced latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>example 1</td>
<td>-9, -14</td>
<td>22, 17</td>
<td>22+17</td>
<td>1</td>
<td>-17</td>
<td>5</td>
</tr>
<tr>
<td>example 2</td>
<td>-19, 23</td>
<td>12, 23</td>
<td>12+23</td>
<td>1</td>
<td>-12</td>
<td>11</td>
</tr>
<tr>
<td>example 3</td>
<td>24, -20</td>
<td>24, 11</td>
<td>24+11</td>
<td>1</td>
<td>-11</td>
<td>13</td>
</tr>
<tr>
<td>example 4</td>
<td>1, 30</td>
<td>1, 30</td>
<td>1+30</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**Case 1:** $|\Delta h_0 - \Delta h_{d-1}| < \frac{q-1}{2}$. In this case, $\delta$ can be chosen as $-\min\{\Delta h_0, \Delta h_{d-1}\}$ and the latency overhead is reduced to $|\Delta h_0 - \Delta h_{d-1}|$.

**Case 2:** $|\Delta h_0 - \Delta h_{d-1}| > \frac{q-1}{2}$. In this case, $\delta$ can be chosen as $(q-1) - \max\{\Delta h_0, \Delta h_{d-1}\}$ and the latency overhead is reduced to $(q-1) - |\Delta h_0 - \Delta h_{d-1}|$.

Accordingly, the latency overhead can be reduced to at most $\frac{q-1}{2}$.

Table 5.1 shows some examples of how the shifting latency overhead can be reduced by the proposed method for a code over $GF(32)$. It can be seen from this Table that substantial saving can be achieved.

### 5.3 Check Node Processing for the SMSA

In this section, the SMSA is reexamined and a regular CNU architecture for the SMSA is proposed by utilizing the power representation of finite field elements.
Algorithm I: The SMSA Check Node Processing

I1: compute \( \min_1(\beta), I(\beta), \min_2(\beta) \) (\( \beta = 1, \alpha, \ldots, \alpha^{q-2} \))
for \( i = 0 \) to \( d_c - 1 \)

I2: \( M_i = \{ \min_i(\beta) \} \) (\( \beta = 1, \alpha, \ldots, \alpha^{q-2} \))
\[
\min_i(\beta) = \begin{cases} 
\min_1(\beta), & i \neq I(\beta) \\
\min_2(\beta), & i = I(\beta)
\end{cases}
\]

I3: \( v_{m,n,i}(0) = 0 \)
\[
v_{m,n,i}(\beta) = \min_{\beta = 0} \left( \sum_{l=1}^{k} \min_i(\beta_l) \right), 1 \leq k \leq K
\]

5.3.1 SMSA Check Node Processing Based on Power Representation

In the SMSA, the minimum and second minimum LLRs of each field element in the input v-to-c message vectors can be first found. Then \( \min_{i \in S_{v(m)}} u_{m,i}(a_l) \) in (2.2) can be chosen from these values. Accordingly, the check node processing in the SMSA can be carried out as Algorithm I.

In the I1 step, \( \min_1(\beta) \) and \( \min_2(\beta) \) are the minimum and second minimum LLRs associated with \( \beta \) among all the input v-to-c message vectors, respectively. \( I(\beta) \) is the index of the v-to-c message vector that provides \( \min_1(\beta) \). To compute the \( i \)th c-to-v message vector, \( v_{m,n,i} \), a vector \( M_i \) is formed in the I2 step without including the messages from the \( i \)th v-to-c vector. Then the I3 step computes \( v_{m,n,i} \) from \( M_i \). Since the vectors in the SMSA are transformed by adding the most likely field element to each element in the vector, \( v_{m,n,i}(0) \) is always zero [42]. \( v_{m,n,i}(\beta) \) for \( \beta \neq 0 \) are computed by repeating the summations and taking the minimum sum.

126
Following an idea similar to that in the forward-backward scheme, the computations in Step I3 have been decomposed into sub-steps [42]. For example, when $K = 4$, Step I3 can be decomposed into two sub-steps as follows:

**I3a:**

$$L(\beta) = \min_{\beta = \beta_1 \oplus \beta_2} \{\min_i(\beta_1) + \min_i(\beta_2), \min_i(\beta)\}$$

$$1 \leq \beta_1, \beta_2 < \alpha^{q-1}$$

**I3b:**

$$v_{m,n_i}(\beta) = \min_{\beta = \beta_1 \oplus \beta_2} \{L(\beta_1) + L(\beta_2), L(\beta)\}$$

$$1 \leq \beta_1, \beta_2 < \alpha^{q-1}$$

$I3a$ is equivalent to the check node processing with $K = 2$ and $I3b$ further combines the results of $I3a$ to achieve the SMSA with $K = 4$. For larger $K$, more sub-steps can be added to carry out min-sum over the results derived in previous sub-steps. Each sub-step is identical and the computations can be described as $L_o(\beta) = \min_{\beta = \beta_1 + \beta_2} (L_{in}(\beta_1) + L_{in}(\beta_2), L_{in}(\beta))$, where $L_{in}$ and $L_o$ are the input and output vector, respectively. Different from the forward-backward process for the EMS algorithm, the two operands of the addition in these sub-steps for the SMSA are from the same vector. Hence, the additions of half of the LLR pairs are repeated in Step $I3a$, and the redundancy becomes more in later steps.

To take the redundancy of the additions into account, the CNU in [42] adds up less pairs of LLRs in the hardware units for later sub-steps. This leads to irregularity. In the next sub-section, a regular CNU architecture is developed to better exploit the redundancy using power representation of finite
field elements. Also our design is partial parallel. It requires much smaller area than the fully-parallel CNU in [42], and can achieve good throughput-area tradeoff.

5.3.2 SMSA CNU Architecture Based on Power Representation

![Proposed architecture for the SMSA over GF(8)](image)

Figure 5.3: Proposed architecture for the SMSA over GF(8)

The I1 and I2 steps can be implemented by simple sorters. Fig. 5.3 shows the proposed architecture for implementing the I3 step of the SMSA check node processing for a code over GF(8). In this figure, \(L_{in1}\) and \(L_{o1}\) are register arrays with \(q - 1\) entries for computing \(v_{m,n_i}\). \(L_{in2}\) and \(L_{o2}\) are register arrays for computing a second vector, \(v_{m,n_j}\), in an overlapped scheme that will be detailed next.

Table 5.2 shows all possible additions of field element pairs over GF(8) needed in the I3a step. Since the two operands of each addition are from the...
Table 5.2: Finite field element additions over $GF(8)$ for Step I3a

<table>
<thead>
<tr>
<th>clk 1</th>
<th>clk 2</th>
<th>clk 3</th>
<th>clk 4</th>
<th>clk 5</th>
<th>clk 6</th>
<th>clk 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\alpha$</td>
<td>$\alpha^2$</td>
<td>$\alpha^3$</td>
<td>$\alpha^4$</td>
<td>$\alpha^5$</td>
<td>$\alpha^6$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$1 + \alpha$</td>
<td>$\alpha^2 + \alpha$</td>
<td>$\alpha^3 + \alpha$</td>
<td>$\alpha^4 + \alpha$</td>
<td>$\alpha^5 + \alpha$</td>
<td>$\alpha^6 + \alpha$</td>
</tr>
<tr>
<td>$\alpha^2$</td>
<td>$1 + \alpha^2$</td>
<td>$\alpha + \alpha^2$</td>
<td>$\alpha^3 + \alpha^2$</td>
<td>$\alpha^4 + \alpha^2$</td>
<td>$\alpha^5 + \alpha^2$</td>
<td>$\alpha^6 + \alpha^2$</td>
</tr>
<tr>
<td>$\alpha^3$</td>
<td>$1 + \alpha^3$</td>
<td>$\alpha + \alpha^3$</td>
<td>$\alpha^2 + \alpha^3$</td>
<td>$\alpha^3 + \alpha^3$</td>
<td>$\alpha^4 + \alpha^3$</td>
<td>$\alpha^5 + \alpha^3$</td>
</tr>
<tr>
<td>$\alpha^4$</td>
<td>$1 + \alpha^4$</td>
<td>$\alpha + \alpha^4$</td>
<td>$\alpha^2 + \alpha^4$</td>
<td>$\alpha^3 + \alpha^4$</td>
<td>$\alpha^5 + \alpha^4$</td>
<td>$\alpha^6 + \alpha^4$</td>
</tr>
<tr>
<td>$\alpha^5$</td>
<td>$1 + \alpha^5$</td>
<td>$\alpha + \alpha^5$</td>
<td>$\alpha^2 + \alpha^5$</td>
<td>$\alpha^3 + \alpha^5$</td>
<td>$\alpha^4 + \alpha^5$</td>
<td>$\alpha^6 + \alpha^5$</td>
</tr>
<tr>
<td>$\alpha^6$</td>
<td>$1 + \alpha^6$</td>
<td>$\alpha + \alpha^6$</td>
<td>$\alpha^2 + \alpha^6$</td>
<td>$\alpha^3 + \alpha^6$</td>
<td>$\alpha^4 + \alpha^6$</td>
<td>$\alpha^5 + \alpha^6$</td>
</tr>
</tbody>
</table>

same input vector, this table is symmetric. Only the min-sum operations of the LLRs associated with half of the entries are actually needed. The sum of two identical field elements is zero, and the corresponding LLR is always set to zero in Algorithm I. Hence, the min-sum corresponding to the entries in the diagonal are not needed. Assume that the min-sum associated with the field element additions in the lower triangle of Table 5.2 are carried out for computing $L_{o1}$. As it can be observed, the number of min-sum operations needed decreases in each clock cycle. To fully utilize the computation units, we propose to associate the additions in the upper triangle of the table with the computation of another c-to-v vector, and overlap the computations of two vectors in the same CNU. In this way, all the min-sum units are active all the time.

In the beginning, $M_i$ is loaded into $L_{in1}$ and $L_{o1}$, and $M_j$ is loaded into $L_{in2}$ and $L_{o2}$ in Fig. 5.3. The entries of all these vectors are stored in the order
Table 5.3: Finite field element additions over $GF(8)$ for Step I3b

<table>
<thead>
<tr>
<th>$v_{m,n_1}$</th>
<th>clk 1</th>
<th>clk 2</th>
<th>clk 3</th>
<th>clk 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \oplus \alpha = \alpha^3$</td>
<td>$\alpha \oplus \alpha^2 = \alpha^5$</td>
<td>$\alpha^2 \oplus \alpha^3 = \alpha^6$</td>
<td>$\alpha^3 \oplus \alpha^4 = \alpha^5$</td>
<td></td>
</tr>
<tr>
<td>$1 \oplus \alpha^2 = \alpha^6$</td>
<td>$\alpha \oplus \alpha^4 = \alpha$</td>
<td>$\alpha^2 \oplus \alpha^4 = \alpha$</td>
<td>$\alpha^3 \oplus \alpha^5 = \alpha^2$</td>
<td></td>
</tr>
<tr>
<td>$1 \oplus \alpha^3 = \alpha$</td>
<td>$\alpha \oplus \alpha^4 = \alpha^2$</td>
<td>$\alpha^2 \oplus \alpha^5 = \alpha^3$</td>
<td>$\alpha^3 \oplus \alpha^6 = \alpha^4$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$v_{m,n_2}$</th>
<th>clk 1</th>
<th>clk 2</th>
<th>clk 3</th>
<th>clk 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \oplus \alpha^4 = \alpha^5$</td>
<td>$\alpha \oplus \alpha^3 = \alpha^6$</td>
<td>$\alpha^2 \oplus \alpha^6 = 1$</td>
<td>$\alpha^3 \oplus 1 = \alpha$</td>
<td></td>
</tr>
<tr>
<td>$1 \oplus \alpha^5 = \alpha^6$</td>
<td>$\alpha \oplus \alpha^5 = \alpha^3$</td>
<td>$\alpha^2 \oplus 1 = \alpha^6$</td>
<td>$\alpha^3 \oplus \alpha = 1$</td>
<td></td>
</tr>
<tr>
<td>$1 \oplus \alpha^6 = \alpha^2$</td>
<td>$\alpha \oplus 1 = \alpha^3$</td>
<td>$\alpha^2 \oplus \alpha = \alpha^4$</td>
<td>$\alpha^3 \oplus \alpha^2 = \alpha^5$</td>
<td></td>
</tr>
</tbody>
</table>

of 1, $\alpha, \ldots, \alpha^{q-2}$ according to power representation. In the first clock cycle, the LLR of the first entry in $L_{in1}$ is added up with each of its rest $q - 2$ LLRs according to Table 5.2. In this clock cycle, no computation is done for $L_{in2}$. The wiring between the 'sum' and 'min' units is decided by the sum of the field elements in the second column of Table 5.2. In each of the following clock cycles, all the register arrays are cyclically shifted by one position, and the first entries from the $L_{in1}$ and $L_{in2}$ vectors are added up with the LLRs from the same vectors according to Table 5.2 through setting appropriate select signals for the multiplexors. The computations over the entries in $L_{in1}$ occupy one less 'min' and 'sum' unit, and the calculations over $L_{in2}$ use one more 'min' and 'sum' unit in each clock cycle. Also the 'min' units select the previous minimum values in $L_{o1}$ or $L_{o2}$ to compare based on whether the sum is for the lower or upper triangle of Table 5.2. Then each updated minimum value is written back to the same vector. A total of $q - 1$ clock cycles are required to complete the I3a step for computing two c-to-v vectors.

The I3b step carries out min-sum computations over the results of Step
I3a, and can be implemented by re-using the architecture in Fig. 5.3. After Step I3a is completed, the contents of $L_{o1}$ and $L_{o2}$ are copied to $L_{in1}$ and $L_{in2}$, respectively, and then Step I3b can start. Since the inputs of the I3b step are already the min-sum results of messages, there are more redundancy in the min-sum computations in the I3b step. For example, $\beta_1 = (\alpha + \alpha^2)$ and $\beta_2 = (\alpha^3 + \alpha^4)$ yields the same sum $\beta = \beta_1 + \beta_2$ as $\beta_1 = (\alpha + \alpha^3)$ and $\beta_2 = (\alpha^2 + \alpha^4)$. It has been derived in [42] that for each $\beta$, as long as no less than $((q - 4)/4)$ different pairs of $\beta_1$ and $\beta_2$ are included in the min-sum computation in Step I3b, the error correcting performance loss is negligible. Based on this, we propose to implement the I3b step for computing both $v_{m,n_i}$ and $v_{m,n_j}$ according to Table 5.3. For $GF(8)$, $(q - 4)/4 = 1$, and the sums in Table 5.3 cover each possible value of $\beta$ at least once. Although other operands can be chosen for the additions to generate each possible sum at least once, the operands in Table 5.3 can be easily generated by shifting vectors ordered according to power representation.

Different from the overlapped scheme for the I3a step, the first half of the 'sum' and 'min' units in Fig. 5.3 are dedicated for the $v_{m,n_i}$ computation, and the other half are used for $v_{m,n_j}$ calculation in the I3b step. In each clock cycle, all the vectors are cyclically shifted by one position as well. The first entry of $L_{in1}$ are added up with the second through $q/2$th entries of $L_{in1}$ in the first half of the adders, and the first entry of $L_{in2}$ are added up with the $(q/2 + 1)$th through the $q - 1$th entries of $L_{in2}$ in the second half of the adders. It can be observed from Table 5.3 that repeating the shifting and min-sum
Table 5.4: Synthesis results of the Min-max CNU s in Fig. 5.1 and [1] using IBM 90nm CMOS Technology

<table>
<thead>
<tr>
<th>CNU[1]</th>
<th>Our CNU in Fig. 5.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>GF(8)</td>
<td>GF(16)</td>
</tr>
<tr>
<td>Area (μm²)</td>
<td>11383</td>
</tr>
<tr>
<td>Area reduction</td>
<td>-</td>
</tr>
<tr>
<td>Max. Frequency (Mhz)</td>
<td>240</td>
</tr>
<tr>
<td>Frequency increases</td>
<td>-</td>
</tr>
</tbody>
</table>

computations for \(q/2\) clock cycles would cover each possible sum \(\beta\) at least once. As a result, \(q/2\) clock cycles are required to complete the I3b step for computing two c-to-v vectors.

If \(K > 4\) is needed, more steps can be done by re-using the same architecture. It was also mentioned in [42] that in Step I3a, including \((q-2)/2\) pairs of \(\beta_1\) and \(\beta_2\) in the min-sum computation for each \(\beta\) would not cause much performance loss. Inspired by this, another table for Step I3a can be derived using an approach similar to that for Table 5.3 in order to take advantage of the cyclical shifting property of power representation.

5.4 Complexity Comparisons

The hardware complexity of the proposed designs are analyzed and compared with prior works in this section. The comparisons are derived based on synthesis results using Synopsys Design Compiler.

The proposed CNU architecture in Fig. 5.1 and that in [1] are synthe-
sized using IBM 90nm CMOS process for finite fields of different orders, and the results are listed in Table 5.4. The maximum clock frequency is derived as the highest frequency that can be set without causing timing violation in the synthesis. From Table 5.4, the proposed CNU requires substantially less area compared to that in [1], and can achieve higher clock frequency. These improvements come from the eliminated switching network. Moreover, the area reduction and frequency increase will be more significant when the order of the finite field, $q$, becomes higher in general. The reason is that the area of the switching network in [1] occupies a more significant part of the CNU area when $q$ increases. Also each time when $q$ is doubled, the switching network in [1] has one extra stage of logic gate. Although the multiplexor in grey in Fig. 5.1 of the proposed design also increases when $q$ becomes higher, the proposed design has fixed wirings between the ‘max’ and ‘min’ units. The proportion of the CNU area used for these wirings does not change much with $q$, and the latency of the wirings only changes slightly. The latency overhead for shifting the last output vectors from the forward and backward processes is at most $(q - 1)/2$ using the proposed method. Even if these shifting can not be overlapped with other computations, the extra clock cycles needed is a very small part of the overall CNU latency for practical NB-LDPC codes. Hence, our proposed design has higher throughput than that in [1] considering the faster achievable clock frequency. Another advantage of our design is that the multiplications and divisions of the nonzero entries of $H$ are already incorporated. Nevertheless, extra multipliers are required in the design of [1],

133
whose complexity is not included in Table 5.4. As a result, the savings of the proposed design would be more significant when the multipliers in [1] are included.

Table 5.5: Synthesis results for Min-max CNUs over \( GF(8) \) using IBM 90\( nm \) CMOS Technology

<table>
<thead>
<tr>
<th></th>
<th>( q )-folded CNU [2]</th>
<th>Our CNU in Fig. 5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area (( \mu m^2 ))</td>
<td>14383</td>
<td>5628</td>
</tr>
<tr>
<td>Area reduction</td>
<td>-</td>
<td>61%</td>
</tr>
<tr>
<td>Max. frequency (Mhz)</td>
<td>208</td>
<td>258</td>
</tr>
<tr>
<td>Frequency increase</td>
<td>-</td>
<td>18%</td>
</tr>
</tbody>
</table>

The proposed CNU architecture in Fig. 5.2 and a \( q \)-folded version of the CNU developed in [2] are also synthesized. \( GF(8) \) is considered and the results are compared in Table 5.5. The proposed design requires less than half the area and can achieve higher clock frequency. The improvements also come from the eliminated switching network. Similarly, the area saving and frequency increase achievable by our design will be more significant with increasing \( q \). Moreover, since the design in Fig. 5.2 does not require the \( q \)-to-1 multiplexor as in the CNU in Fig. 5.1, the savings brought by the switching network elimination are more significant. In addition, the CNU in [2] needs a barrel shifter to handle the multiplications of the nonzero entries of \( H \), and it is not included in the synthesis. One barrel shifter requires similar area as that of the routing network in [1]. If the barrel shifter is included, the savings of the proposed design would be more significant. The implementation of our design
in Fig. 5.2 adopts the proposed method for reducing the latency overhead. As a result, the number of extra clock cycles required for the vector shifting in the first elementary steps of the forward and backward processes is also at most \((q - 1)/2\). Accordingly, our design in Fig. 5.2 can also achieve higher throughput than the \(q\)-folded version of the architecture in [2] for NB-LDPC codes with moderate or small \(q\). When folding is not required, the proposed CNU has almost the same complexity as that in [2]. Nevertheless, the output vectors of the CNU in [2] are not c-to-v message vectors. Shifting them would require extra complexity.

Table 5.6: Synthesis results for \(K = 4\) SMSA CNUs over \(GF(32)\) with 90nm CMOS Technology

<table>
<thead>
<tr>
<th></th>
<th>CNU [42]</th>
<th>Our CNU in Fig. 5.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area ((\mu m^2))</td>
<td>372244</td>
<td>35714</td>
</tr>
<tr>
<td>Area (normalized)</td>
<td>1</td>
<td>0.09</td>
</tr>
<tr>
<td>No. of clks</td>
<td>1</td>
<td>23.5</td>
</tr>
<tr>
<td>Frequency(Mhz)</td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>

The proposed design in Fig. 5.3 is compared with the SMSA CNU in [42] in Table 5.6. In this table, the area requirements do not include those of the sorters for computing the minimum and second minimum LLRs of each field element. A serial sorter can be simply implemented by a comparator-register loop. The design in [42] can finish computing a c-to-v message vector in one clock cycle after \(M_i\) is available. For \(K = 4\), the proposed design computes two c-to-v vectors in an overlapped manner in two rounds, which
need \( q - 1 + q/2 = 47 \) clock cycles for codes over \( GF(32) \). Hence, on average, the computation of a vector takes \( 47/2 = 23.5 \) clock cycles. Although the proposed design cannot achieve as fast speed as the fully-parallel design in [42], its area requirement is less than 10%. Also as can be observed from Fig. 5.3, the proposed design is very regular and consists of columns of identical hardware units. When larger \( K \) is needed, more sub-steps of the min-max computations can be implemented by re-using the same architecture. On the contrary, the CNU in [42] is irregular since the hardware units for each later sub-step have less adders. Moreover, to achieve larger \( K \), more hardware units need to be added and the critical path would also increase.

5.5 Summary

This chapter proposed novel check node processing schemes for the forward-backward Min-max and the simplified Min-sum NB-LDPC decoding. By making use of the cyclical-shift property of the power representation of finite field elements, the complex switching network in the forward-backward Min-max CNU is eliminated. Moreover, the multiplications/divisions of the entries of \( H \) can be efficiently incorporated. Applying the proposed schemes to existing Min-max CNU designs lead to significant area saving. For the SMSA, the proposed CNU is more regular and the design-to-market time can be shortened. In addition, the same architecture can be re-used for more rounds to implement SMSA with larger order so that the error-correcting performance is improved.
Chapter 6

Conclusions and Future Research

6.1 Conclusions

This thesis focuses on the design of decoding algorithms and VLSI architectures for NB-LDPC decoders. The covered topics include NB-LDPC decoders based on relaxed Min-max algorithm and IHRB-MLGD algorithms. Moreover, a decoder using power representation of finite field elements has been introduced.

A novel relaxed check node processing scheme for the Min-max NB-LDPC decoding algorithm is included. By making use of the property that each finite field element can be uniquely represented by a linear combination of the elements in the minimum basis, all the entries in a message vector are computed simultaneously in an efficient way. Moreover, an innovative method is proposed to derive the minimum basis for each message vector from a common trellis basis. This further reduces the memory requirement and computation complexity. Adopting the proposed check node processing, the overall decoder can achieve substantially higher efficiency than all existing designs.

Enhancement schemes to the IHRB decoding algorithm for NB-LDPC
codes are presented. The proposed schemes lead to significant coding gain with small complexity overhead. In addition, efficient architectures were developed for both QC and cyclic NB-LDPC codes based on the IHRB and E-IHRB algorithms. With moderate performance loss, the proposed decoders can achieve at least tens of times higher efficiency compared to previous designs based on the Min-max algorithm.

Novel check node processing schemes for the forward-backward Min-max and the simplified Min-sum NB-LDPC decoding are also proposed. By making use of the cyclical-shift property of the power representation of finite field elements, the complex switching network in the forward-backward Min-max CNU is eliminated. Moreover, the multiplications/divisions of the entries of $H$ can be efficiently incorporated. Applying the proposed schemes to existing Min-max CNU designs lead to significant area saving. For the SMSA, the proposed CNU is more regular and the design-to-market time can be shortened. In addition, the same architecture can be re-used for more rounds to implement SMSA with larger order so that the error-correcting performance is improved.

6.2 Future Research

This thesis has contributed to reducing the area of the VLSI design for the NB-LDPC decoders based on several low-complexity decoding algorithms. However, these architectures can be further improved.

In the relaxed presented in Chapter 3, the data dependency in different
components of the CNU impedes the further speedup of the overall decoder. Moreover, the control logic is relatively complex, which elongate the design-to-market time. In addition, the interleaving technique used for improving the HUE reduce the flexibility in dealing with the input data. These shortcomings require some efforts to modify relaxed Min-max algorithm and can be an interesting topic.

In IHRB-MLGD decoder presented in Chapter 4, the error correcting capability improved by E-IHRB is still quite limited. Moreover, separating the memory blocks into small pieces will inevitably reduce the silicon area. In addition, the IHRB-MLGD does not perform well when variable node degree is small. Future research can be conducted to address these problems and devoted to further improving the performance and reducing the hardware complexity of MLGD-based algorithms for NB-LDPC decoding.

In the power representation based check node processing in Chapter 5, only check node unit is discussed. Future research will address reducing the overall NB-LDPC decoder complexity by using the proposed schemes.
Bibliography


