Adaptive multiset stochastic decoding of non-binary LDPC codes

Alexandru Ciobanu

Master of Engineering

Department of Electrical and Computer Engineering

McGill University

Montréal, Québec

December 2011

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Engineering

ACKNOWLEDGMENTS

I would like to thank Professor Warren J. Gross for introducing me to the field of LDPC decoding, for maintaining a team and environment where my ideas could come to fruition, and for all the feedback and guidance. I am thankful to Gabi Sarkis for providing and diligently improving his excellent simulator, for the countless explanations and discussions. I am grateful to Dr. Saied Hemati for his assistance and encouragement. Finally, I would like to thank Professor Emmanuel Boutillon of Université de Bretagne-Sud for the constructive feedback provided as part of his review of this work.

ABSTRACT

In this thesis, we propose a new stochastic decoding algorithm for non-binary LDPC codes with $d_v = 2$, which is based on the concept of a mutliset, a generalization of the set that allows for multiple occurrences of the same element. The algorithm is called Adaptive Multiset Stochastic Algorithm (AMSA) and represents probability mass functions as multisets, which simplifies the structure of the variable node. AMSA reduces the run-time complexity of one decoding cycle to $O(q)$ for regular memory architectures, and to $O(1)$ if a custom SRAM architecture is used. Two fully-parallel AMSA decoders are implemented on FPGA for two versions of a (192,96) (2,4)-regular code, one over GF(64) and the other over GF(256), both achieving a maximum clock frequency of 108 MHz and a throughput of 65 Mbit/s at $E_b/N_0 = 2.4$ dB. We also propose an SRAM architecture for ASIC implementations that reduces the run-time complexity of a decoding cycle to $O(1)$ and achieves a throughput of 698 Mbit/s at the same noise level. The algorithm has a frame error rate (FER) of 3.5×10^{-7} at $E_b/N_0 = 2.4$ dB when using the GF(256) version of the code. To the best of our knowledge, the implemented decoders are the first fully-parallel non-binary LDPC decoders over GF(64) and GF(256) reported in the literature.

ABRÉGÉ

Dans cette thèse, nous proposons un nouvel algorithme de décodage stochastique pour des codes LDPC non-binaires avec $d_v = 2$, qui est basé sur le concept de multi-ensemble, une généralisation de l'ensemble où un élément peut apparaître plusieurs fois. L'algorithme est appelé Algorithme Stochastique à Multiensembles Adaptifs (ASMA) et représente des fonctions de masse comme multiensembles, ce qui simplifie la structure du nœud de variable. ASMA réduit la complexité d'exécution d'une itération de décodage à $O(q)$ pour les architectures de mémoire ordinaire, et $O(1)$ si une architecture SRAM personnalisée est utilisée. Deux décodeurs ASMA tout-parallèles sont mis en œuvre sur FPGA pour deux versions d'un code (192,96) (2,4)-réguliers, l'un sur GF(64) et le l'autre sur GF(256), et tous les deux atteignent une fréquence d'horloge maximale de 108 MHz et un débit de 65 Mbit/s à E_b/N_0 = 2.4 dB. Nous proposons aussi une architecture SRAM pour les implémentations ASIC qui réduit la complexité d'exécution d'un cycle de décodage à $O(1)$ et atteint 698 Mbit/s au même niveau de bruit. L'algorithme a un taux d'erreur de trame de 3.5×10^{-7} à $E_b/N_0 = 2.4$ dB pour la version GF(256) du code. Au meilleur de notre connaissance, les décodeurs présentés ici sont les premiers décodeurs LDPC non-binaires opérant sur GF(64) et GF(256) et tout-parallèles rapportés dans la littérature.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

ABSTRACT iii

ABRÉGÉ iv

LIST OF FIGURES vii

LIST OF TABLES x

1 Introduction 1

1.1 Contributions of this Work 3
1.2 Outline 4

2 Background 5

2.1 Iterative LDPC Decoding over GF(q) 5
2.1.1 The Sum-Product Algorithm 5
2.1.2 The FFT-SPA Algorithm 9
2.1.3 The Log-SPA Algorithm 11
2.1.4 The Extended Min-Sum Algorithm 13
2.2 Stochastic LDPC Decoding over GF(q) 16
2.2.1 Stochastic Representation of Probabilities 16
2.2.2 Messages in Stochastic Decoding Algorithms 17
2.2.3 Stochastic Decoding Over GF(q) 18
2.2.4 Relaxed Half-Stochastic Decoding Over GF(q) 20
2.2.5 Redecoding 21
2.3 Architectures and Implementations of GF(q) LDPC Decoders . . . 22

3 The Adaptive Multiset Stochastic Algorithm 24

3.1 Multiset Representation of a Probability Mass Function 24
3.2 Algorithm Definition and Analysis 27
3.2.1 The *Add* Routine 28
3.2.2 The *Remove* Routine 32

3.2.3 The *Sample* Routine . . . 35
3.3 Non-Binary LDPC Decoding with AMSA . . . 35
3.4 Complexity Analysis . . . 37
3.5 Redecoding . . . 39

4 Circuit Implementation . . . 41

4.1 Structure of fully-parallel decoders . . . 42
4.2 Variable Node . . . 42
4.2.1 Pseudo-Random Number Generator . . . 43
4.2.2 Hardware Representation of the Likelihoods Table . . . 44
4.2.3 Edge Memories - a Hardware Representation of the Mulitset S 45
4.2.4 Hardware Implementation of the *Remove* Routine . . . 46
4.2.5 Hardware Implementation of the *Add* Routine . . . 48
4.2.6 Hardware Implementation of the *Sample* Routine . . . 50
4.2.7 State Machine . . . 52
4.3 Check Node . . . 53
4.4 Permutation Block . . . 55
4.5 Message Passing Scheduling . . . 56
4.6 Synthesis Results . . . 58
4.7 ASIC-specific Considerations . . . 60
4.7.1 Single Clock-Cycle Update Memory Design . . . 60

5 Simulation Results and Analysis . . . 64

5.1 Performance . . . 64
5.2 Throughput and Latency . . . 66
5.3 Efficient Quantization . . . 69
5.4 Accelerated Convergence *Add* Routine . . . 71

6 Conclusion and Future Work . . . 76

6.1 Advances . . . 76
6.2 Future Work . . . 77

REFERENCES . . . 78

LIST OF FIGURES

Figure page

2–1 Tanner graph transformation by adding permutation nodes on the edges. 8

2–2 Tanner graph with the check node computation performed in the frequency domain over GF(q). 10

2–3 Comparison of SPA and RHS algorithm for a GF(64) (2,4)-regular code with $n = 192$ and $k = 96$ on the AWGN channel. 21

3–1 Schematic representation of a variable node of degree $d_v = 2$ with the multisets S_0 and S_1, and the channel likelihoods table L_{CH}. 28

3–2 Comparison of the non-binary RHS algorithm and AMSA by tracking the probability of the correct symbol in an edge memory. 32

4–1 The 32-bit LFSR with the feedback taps corresponding to the polynomial. 44

4–2 The interface of a likelihoods memory corresponding to the probabilities l_j where $j = 1, \ldots, q$ as implemented on FPGA. 45

4–3 The interface of an edge memory corresponding to a multiset containing at most M GF(q) symbols. 46

4–4 Removing a symbol from memory at random index $R_1 < L$ by overwriting it with the symbol at index $L - 1$. 47

4–5 Circuit for the *Remove* routine. 48

4–6 Adding k instances of a symbol to the memory, the equivalent of the $S \leftarrow S \cup \{$ k instances of $\alpha_j\}$ operation in the *Add* routine. 49

4–7 Circuit for the part of the *Add* routine responsible for computing the number of symbols to add. 50

4–8 Circuit for the *Sample* routine as implemented for AMSA-128. 52

4–9 The finite state machine controlling the computation in the variable node. 53

4–10 Circuit representation of the check node computation. 55

4–11 The edge of the Tanner graph connecting variable node i (VN_i) and check-node j (CN_j). 56

4–12 The flood scheduling method where $done_i$ is the signal that all variable nodes that send messages to check node i (CN_i) have completed their computation. 56

4–13 The layered decoding method where $done_i$ is the signal that all variable nodes that send messages to check node i (CN_i) have completed their computation. 57

4–14 The FPGA chip floor plan after the synthesis, and place and route of the GF(256) AMSA-512 fully-parallel decoder. 59

4–15 Schematic representation of the scenarios in Table 4–4 and the modifications they make to the memory. 61

4–16 Architecture of SRAM that can perform any of the scenarios in Table 4–4 in one cycle. 62

5–1 Frame error rate performance of the AMSA algorithm 65

5–2 Throughput of the FPGA implementation and estimated throughput for the ASIC implementation of the AMSA algorithm, at clock frequency $f = 108$ MHz. 67

5–3 Settling curves for the GF(64) and GF(256) versions of the (192,96) code. 68

5–4 The impact of using the efficient quantization method on the performance. 70

5–5 The impact of using the efficient quantization method on the average number of decoding cycles. 71

5–6 The impact of using the accelerated convergence *Add* routine on the average number of decoding cycles. 73

5–7 The impact of using the accelerated convergence *Add* routine on the performance. 74

5–8 Circuit for computing the term $l_\alpha + c_A A(t)$ in the accelerated convergence *Add* routine. 75

LIST OF TABLES

Table		page
2–1	Comparison of implementations of GF(q) LDPC decoders in literature	23
3–1	Equivalence of operations on a pmf and a multiset representation of a pmf	26
3–2	The number of operations required and run-time complexity for fully-parallel implementations of the AMSA algorithm	37
3–3	Summary of space complexity for AMSA	38
4–1	Comparison of GF(64) AMSA-128 and GF(256) AMSA-512 fully-parallel decoders	43
4–2	Computations corresponding to each state	54
4–3	Summary of the hardware resources used by the fully-parallel GF(64) AMSA-128 and GF(256) AMSA-512 fully-parallel decoders on Altera Stratix IV GX EP4SGX230. Note that these results are after place and route.	58
4–4	Possible scenarios based on decisions made in *Add* and *Remove* routines	61

Chapter 1

Introduction

Introduced in 1962 by Gallager [1] and rediscovered by MacKay three decades later [2], low-density parity-check (LDPC) codes are linear error-correcting block codes built using sparse bipartite graphs.

LDPC codes are widely recognized for the channel capacity-approaching performance [2, 3, 4] and the high degree of parallelism in decoding operations. These properties led to the inclusion of LDPC codes in multiple communication standards like DVB-S2 (satellite broadband) [5], ITU-T G.hn (networking over power lines, phone lines, and coaxial cable) [6], IEEE 802.3an (10GBase-T Ethernet) [7], IEEE 802.11n-2009 (Wi-Fi) [8], IEEE 802.16e (WiMAX) [9], and others. Additionally, LDPC codes found their use in storage systems [10, 11].

Non-binary LDPC codes have been shown to have better resilience against burst errors [11] and mixed types of noise and interference [12], are better suited for higher-order modulation, and provide a considerable performance boost to medium and short length codes. Because of these properties non-binary LDPC codes were studied

within DAVINCI project which aims at further reducing the gap between state-of-the-art performance of practical codes and Shannon capacity [13, 14]. The non-binary LDPC codes used in this work were designed under the auspices of this project.

The performance improvements associated with the generalization of LDPC codes to non-binary GF(q) fields came at a cost. The complexity of the Sum-Product Algorithm (SPA) under GF(q) becomes $O(q^2)$, which limits the feasibility of non-binary decoders to lower-order GF(q) fields. There have been multiple attempts at tackling the complexity problem with algorithms like FFT-SPA [15], Log-SPA [16], and Extended Min-Sum [17] (EMS) but the complexity remains high and a fully-parallel implementation of these decoders is not practical. For example, it has been reported in [18] about an LDPC decoder implementation for a GF(64) (192,96) (2,4)-regular code with a complexity of $O(n_m\sqrt{n_m})$ where $n_m < q$ and a throughput of 3.8 Mbit/s.

Stochastic decoding for LDPC codes was introduced in [19] as a way of reducing hardware complexity [20] while matching and even improving on the performance of reference algorithms like SPA in both binary [19, 21, 22] and non-binary [23] cases. In stochastic decoding, instead of the probabilities of symbols actual symbols are sent as messages with the probabilities being encoded in the statistics of the stream. Despite the implementation advantages of non-binary stochastic decoders compared to the previously reported decoders, designing a fully-parallel decoder remains challenging, especially for high order fields GF($q \geq 64$).

In order to address the problems mentioned above, this thesis proposes a new stochastic decoding algorithm, an architecture, and a fully-parallel implementation of a decoder for practical $d_v = 2$ LDPC codes over GF(64) and GF(256). This

algorithm considerably reduces the complexity of the computations performed in the variable nodes (VNs) while inheriting the benefits of very simple check nodes (CNs) and interleaver circuit [20, 24].

Another issue that affects both binary and non-binary LDPC codes is the so-called "error floor" - a degradation of performance in the high signal-to-noise ratio (SNR) region [25]. To tackle this problem, this work successfully extends the redecoding technique [22] to non-binary codes. Redecoding is a technique, introduced originally for relaxed half-stochastic (RHS) decoding of binary LDPC codes, that improves bit-error-rate (BER) performance and lowers error floors by making multiple decoding attempts on codewords that fail to decode initially.

1.1 Contributions of this Work

This work proposes a new stochastic decoding algorithm for non-binary LDPC codes with $d_v = 2$. The algorithm is called Adaptive Multiset Stochastic Algorithm (AMSA) and reduces the run-time complexity of one decoding iteration to $O(q)$ for implementations using regular memory, and to $O(1)$ with a custom SRAM architecture.

We also propose a method of accelerating the convergence of the decoder while improving the BER performance by using a proportional-integral strategy in the variable node update. This allows to reduce the average number of decoding cycles required for the AMSA-256 GF(64) decoder at 2.4 dB by 27%. Note that this method is applicable for all configurations of the AMSA decoder including the GF(64) and GF(256) implementations.

Furthermore, we propose a fully-parallel architecture for AMSA which we apply to two practical codes from the DAVINCI project [13, 14]. The decoders achieve clock frequencies of 108 MHz and a throughput of 65 Mbit/s on FPGA and 698 Mbit/s on ASIC at an SNR of 2.4 dB, which are, to the best of our knowledge, the highest reported throughput for these codes. We show that AMSA decoder architecture scales gracefully with the order of the field q. The length of the memory blocks scales with $O(q)$ while the width of the memory blocks, the number of wires in the decoder, the length of the registers, and the size of control logic scale with $O(\log q)$. To the best of our knowledge, these are the first fully-parallel non-binary LDPC decoders for GF(64) and GF(256) presented in the literature.

Finally, we design a suitable SRAM architecture for the ASIC implementation of the AMSA decoder that can write a single value in multiple locations in one write cycle. This architecture reduces the run-time complexity of a decoding cycle of the fully-parallel AMSA decoder from $O(q)$ to $O(1)$. On FPGA this operation becomes $O(q)$, still a considerable improvement over the current $O(q^2)$ algorithms.

1.2 Outline

A background on LDPC decoding and relevant topics is given in Chapter 2. For the full discussion of the Adaptive Multiset Stochastic Algorithm and the hardware implementation see Chapters 3 and 4, respectively. Chapter 5 is concerned with the analysis of the simulation results. Chapter 6 concludes this thesis and proposes several directions for future work.

Chapter 2

Background

This chapter is organized in three parts. The first part reviews iterative LDPC decoding over GF(q) and the proposed algorithms. The second part focuses on stochastic decoding and how it applies to non-binary LDPC decoding. The last part compares the hardware implementations of non-binary LDPC decoders reported in literature.

2.1 Iterative LDPC Decoding over GF(q)

2.1.1 The Sum-Product Algorithm

The Sum-Product Algorithm (SPA) was originally extended to non-binary LDPC codes over GF(2^p) by Davey and MacKay in [26]. The authors use a memoryless binary symmetric channel (BSC) with additive noise of variance $\sigma^2 = 1$. They define the likelihood of received symbol x_n being equal to a to be

$$f_n^a := \prod_{i=1}^{p} g_{n_i}^{a_i}$$

for each $a \in GF(2^p)$ where $g_{n_i}^{a_i}$ is the likelihood of the ith bit of x_n to be equal to the ith bit of a.

The set of of all symbols that participate in the parity check m are denoted by $\mathcal{N}(m)$, and the set of parity checks that depend on symbol n is denoted by $\mathcal{M}(n)$.

The decoding problem is to find the most likely vector $\mathbf{x}$ such that $\mathbf{Hx} = \mathbf{z}$, where $\mathbf{z}$ is the syndrome vector. The two values q_{mn}^a and r_{mn}^a are associated to each non-binary value h_{mn} found in the parity check matrix. The first one, q_{mn}^a, is the probability that symbol n of $\mathbf{x}$ is equal to a, given the outputs of all checks except the one corresponding to m. Similarly, r_{mn}^a is the probability of parity check m being satisfied by making symbol n of $\mathbf{x}$ equal to a, given the probabilities $q_{mn'}^a$ where $n' \in \mathcal{N}(m)$.

As presented by Davey and MacKay, the Sum-Product Algorithm for GF(q) consists of the following steps:

1. *Initialization*

   Quantities q_{mn}^a are initialized to f_n^a.

2. *Update* r_{mn}^a

   The new r_{mn}^a values are computed:

$$r_{mn}^a = \sum_{\mathbf{x}':x'_n=a} \Pr\left(z_m \mid \mathbf{x}'\right) \prod_{j \in \mathcal{N}(m)\backslash n} q_{mj}^{x'_j} \tag{2.1}$$

   where $\Pr\left(z_m \mid \mathbf{x}'\right)$ is 1 if $\mathbf{x}'$ satisfies the parity check m and 0 otherwise.

3. *Update* q_{mn}^a

   For each m and n and for $\alpha \in GF(q)$ the update is done as follows:

$$q_{mn}^a = \alpha_{mn} f_n^a \prod_{j \in \mathcal{M}(n)\backslash m} r_{jn}^a \tag{2.2}$$

where α_{mn} is a normalization factor chosen such that $\sum_{a=1}^{q} q_{mn}^{a} = 1$.

4. Tentative decoding

The symbols of the tentative codeword $\hat{\mathbf{x}}$ are computed:

$$\hat{x}_n = \underset{a}{\operatorname{argmax}}\, f_n^a \prod_{j \in \mathcal{M}(n)} r_{jn}^a \tag{2.3}$$

If $\mathbf{H}\hat{\mathbf{x}} = \mathbf{z}$ then $\hat{\mathbf{x}}$ is a valid codeword and decoding stops; otherwise, the decoding continues. Failure is declared when a preset maximum number of iterations is reached.

Davey and MacKay give the complexity of one decoding iteration as $O(Ntq^2)$ where N is codeword length and t is the average column weight in matrix $\mathbf{H}$.

After the publication of [26], a graph transformation and an alternative notation were introduced [15] aiming at a simpler presentation of the non-binary SPA equations. The initial observation was that parity checks, corresponding to rows of $\mathbf{H}$, can be written as

$$\sum_{k=1}^{d_c} h_k(x) i_k(x) = 0 \mod p(x)$$

where d_c is the degree of the check, $i_k(x)$ are the codeword symbols, $h_k(x)$ are the associated non-zero values in matrix $\mathbf{H}$, and $p(x)$ is the primitive polynomial of GF(q). The product $h_k(x)i_k(x)$ performed under modulo $p(x)$ is actually a permutation of the message values passed between the variable and the check nodes. In this light, the Tanner graph can be transformed by adding permutation nodes to each edge as shown in Figure 2–1.

On the updated graph, the following notation is used for messages: $\{V_{pv}\}_{v=1,\ldots,d_v}$ is the set of messages entering the a variable node of degree d_v, $\{U_{vp}\}_{v=1,\ldots,d_v}$ are the output messages for a variable node, $\{U_{pc}\}_{c=1,\ldots,d_c}$ are the inputs for a check node,

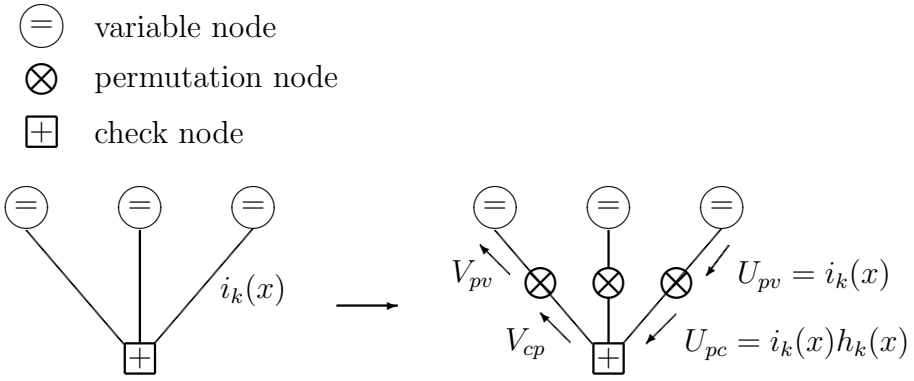

Figure 2–1: Tanner graph transformation by adding permutation nodes on the edges.

and $\{V_{cp}\}_{c=1,\ldots,d_c}$ are the outputs for a check node. The indexes show the direction of the message, for example *cp* means that the message is originating from the check node and aiming at the permutation node, and *pv* stands for the direction from the permutation node to the variable node.

With this new notation the SPA algorithm can be rewritten like this:

1. *Initialization*

   The decoder is initialized using the channel likelihoods $L[i_1, \ldots, i_p] = \prod_{l=1}^{p} l(i_l)$ where $l(i_l) = \Pr(y_l \mid b_l = i_l)$ where b_l is the lth bit of the symbol, and y_l is the corresponding noisy bit received.

2. *Product step*

   The output messages are computed for a variable node of degree d_v:

$$U_{tp} = L \prod_{v=1, v\neq t}^{d_v} V_{pv} \tag{2.4}$$

   where $t = 1, \ldots, d_v$ and all the products are tensor dot products. Note that after normalization all the U_{tp} messages add up to 1. This step is equivalent with Equation (2.2) in the Davey and MacKay notation.

3. Permutation step

In the direction from the variable node to the check node, the permutation operation is:

$$U_{pc}[i_1, \ldots, i_p] = U_{vp}[j_1, \ldots, j_p] \tag{2.5}$$

with $(i_1, \ldots, i_p) \in \{0, 1\}^p$ and $i(x) = h(x)j(x)$. In the reverse direction the same principle applies, but $h^{-1}(x)$ is used.

4. Check step

When using as input permuted messages all the check nodes behave identically, allowing for the sum-product update to be written as a convolution of probability densities:

$$V_{tp} = \circledast_{c=1, c\neq t}^{d_c} U_{pc} \tag{2.6}$$

where $t = 1, \ldots, d_c$. The convolution can also be expressed as a more recognizable sum of products using the tensorial notation:

$$V_{tp}[i_{t_1}, \ldots, i_{t_p}] = \sum_{\{i_c(x)\}_{c\neq t}} \prod_{c=1, c\neq t}^{d_c} U_{pc}[i_{c_1}, \ldots, i_{c_p}] \times 1I_{\sum_{c=1}^{d_c} i_c(x)=0} \tag{2.7}$$

The indicator function $1I_{\sum_{c=1}^{d_c} i_c(x)=0}$ is 1 when the condition $\sum_{c=1}^{d_c} i_c(x) = 0$ is satisfied and 0 otherwise. Note that this indicator function is equivalent to the $\Pr(z_m \mid \mathbf{x}')$ term in Equation (2.1).

2.1.2 The FFT-SPA Algorithm

The complexity of the check node computation in Equations (2.1), (2.6), and (2.7) is $O(d_c q^2)$ which motivated the search for a reduced complexity alternative. The idea of performing this computation in the frequency domain was proposed in [27, 28] and expanded in [29, 11].

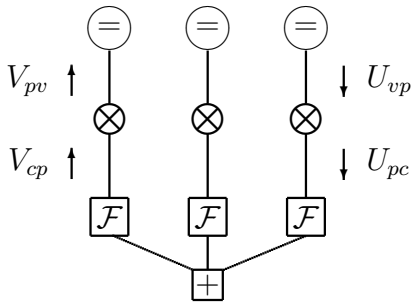

Figure 2–2: Tanner graph with the check node computation performed in the frequency domain over GF(q).

In frequency domain the convolution from Equation (2.6) becomes a product:

$$V_{tp} = \mathcal{F}^{-1}\left(\prod_{c=1,c\neq t}^{d_c} \mathcal{F}(U_{pc})\right) \tag{2.8}$$

where $t = 1, \ldots, d_c$. The complexity of this product is reduced to $O(d_c pq)$ for GF(2^p). This concept is illustrated in Figure 2–2 by showing the Fourier transform nodes in the Tanner graph.

It was shown in [30] that the Fourier transform over GF(q) reduces to a Hadamard transform and that $\mathcal{F}(U) = UH_m$ where U is a message vector and H_m is the appropriately sized Hadamard matrix. Since matrix H_m contains only 1 and -1 values, the UH_m product can be performed using additions only. It was reported in [11] that, in addition to the frequency domain transformation used here, by using a logarithmic representation to transform products into sums, the authors were able to implement a full decoder by using only addition and subtraction operations. The next section describes how SPA computations can be carried out in the logarithm domain.

2.1.3 The Log-SPA Algorithm

Logarithm domain SPA decoding was first introduced for the binary case in [16] and then extended to GF(q) in [31]. There are two main benefits stemming from the application of logarithm domain computations in SPA. Firstly, multiplications are transformed to additions. Secondly, logarithm domain representations are affected less by quantization in fixed-point realizations [32, 33].

In [31] Wymeersch et al. use the notion of a Log Likelihood Ratio Vector (LLRV) defined for a random variable v from GF(q) as

$$\mathbf{L}(v) = [L(v = \alpha_1) \dots L(v = \alpha_{q-1})]^T$$

where

$$L(v = \alpha_i) = \log \frac{\Pr(v = \alpha_i)}{\Pr(v = 0)}.$$

They denote by $\mathbf{L}(m \rightarrow n)$ an LLRV message sent from check node m to variable node n, and by $\mathbf{L}(m \leftarrow n)$ an LLRV message sent from variable n to check node m. Using this notation, the logarithm domain SPA is performed in the following steps:

1. *Demapping and initialization*

   Similarly to previous variations of SPA, the first step is to initialize the decoder using the channel likelihoods as follows

   $$\mathbf{L}(m \leftarrow n) = \mathbf{L}_{ch}(c_n)$$
   $$\mathbf{L}(m \rightarrow n) = 0$$

   where $\mathbf{L}_{ch}(c_n)$ is the LLRV for the nth codeword symbol and is calculated according to the channel model.

2. *Tentative decoding*

The a posteriori LLRVs are computed for all c_n, $1 \leq n \leq N$:

$$\mathbf{L}_{post}(c_n) = \mathbf{L}_{ch}(c_n) + \sum_{j \in \mathcal{M}(n)} \mathbf{L}(j \to n) \tag{2.9}$$

where N is the codeword length and $\mathcal{M}(n)$ has the same meaning as defined in Section 2.1.1. From each LLRV the most likely symbol is chosen and if all the checks are satisfied the decoding is stopped.

3. Horizontal step

The messages from variable node n to check node m are computed by the following equation:

$$\mathbf{L}(m \leftarrow n) = \mathbf{L}_{ch}(c_n) + \sum_{j \in \mathcal{M}(n) \setminus m} \mathbf{L}(j \to n) \tag{2.10}$$

4. Vertical step

For each check node m and adjacent varaible node $n_{m,k}$ two GF(q) random variables are introduced:

$$\sigma_{m,n_{m,l}} = \sum_{j \leq l} h_{m,n_{m,j}} c_{n_{m,j}}$$

$$\rho_{m,n_{m,l}} = \sum_{j \geq l} h_{m,n_{m,j}} c_{n_{m,j}}$$

which are used to compute the messages from the check node m to each of the the variable nodes $n_{m,k}$:

$$\mathbf{L}(m \to n_{m,k}) = \mathbf{L}\left(h^{-1}_{m,n_{m,k}} \sigma_{m,n_{m,k-1}} + h^{-1}_{m,n_{m,k}} \rho_{m,n_{m,k+1}}\right) \tag{2.11}$$

where $1 \leq m \leq M$, $n \in \mathcal{N}(m)$, $h_{m,n}$ are the non-zero entries in the parity check matrix corresponding to variable node n and check node m.

Finally, note that Log-SPA is equivalent to SPA in terms of performance and computation complexity.

2.1.4 The Extended Min-Sum Algorithm

The Extended Min-Sum (EMS) algorithm was introduced in [15] as a way of reducing the complexity of the generalized Min-Sum Algorithm (MSA) over GF(q) and, more specifically, the complexity of the check node computation. It uses a logarithm domain representation for its messages called log-density-ratio (LDR) defined for $z \in GF(q)$ as

$$\mathbf{L}(z) = [L[0] \dots L[q-1]]^T$$

where

$$L[i] = \log \frac{\Pr(z = \alpha_i)}{\Pr(z = \alpha_0)}$$

and α_i are GF(q) symbols.

In EMS, the size of a message is decreased from q to $n_m \leq q$; additionally, the values in the message vectors are kept in sorted order. The reduction is obtained by discarding the $q - n_m$ smaller LDR values and replacing them by a quantity γ. For example given a message A represented by its LDR vector of length q

$$A = [A[0] \dots A[q-1]]^T$$

with likelihoods $A[i]$ in decreasing order, the corresponding reduced message B is

$$B = [A[0] \dots A[n_m - 1]\gamma_A]^T$$

where $\gamma_A \leq A[n_m - 1]$ and compensates for the discarded values. For practical purposes $\gamma_A = A[m] - \mathit{Offset}$ as given in [17] along with the derivations and the heuristics for optimizing *Offset*.

For messages V_{cp} and U_{vp} (see Figure 2–1) the additional vectors $\beta_{V_{cp}}$ and $\beta_{U_{vp}}$ are defined such that the kth largest values in V_{cp} and U_{vp} are associated with the GF(q) symbols found at $\beta_{V_{cp}}[k]$ and $\beta_{U_{vp}}[k]$ respectively.

The EMS algorithm for parameter $n_m \leq q$ can be described in the following steps:

1. *Initialization*

   The $\{U_{vp_i}\}_{i=0,\ldots,d_v-1}$ messages for all variable nodes are initialized with the n_m largest values of the corresponding LLR vectors obtained from the channel.

2. *Variable node update*

   Let V and I be two input messages for a variable node, and let U be the output message to be computed, with β_V, β_I, and β_U being the corresponding index vectors. The variable node computation uses a temporary vector T:

$$T[k] = V[k] + Y \tag{2.12}$$

$$T[n_m + k] = \gamma_V + I[k] \tag{2.13}$$

   where

$$Y = \begin{cases} I[l] & \text{if } \beta_I[l] = \beta_V[k] \\ \gamma_I & \text{if } \beta_I[l] \notin \beta_V \end{cases}$$

   and $k, l = 0, \ldots, n_m - 1$. The output U consists of the largest n_m values in T.

3. *Permutation step*

The messages are permuted according to the non-zero values from the parity check matrix. In the case of EMS, the permutation is done on the index vectors:

$$\beta_{U_{p_i c}}[k] = h_i \beta_{U_{vp_i}}[k] \tag{2.14}$$

where h_i is the corresponding non-zero value from the parity check matrix and $k = 0, \ldots, n_m - 1$. The reverse permutation is computed similarly but using h_i^{-1}.

4. *Check node update*

Let U and I be two input messages and V an output message for the check node, and let β_U, β_I, β_V be the associated index vectors. The elements of the message vector V are obtained as follows:

$$V[i] = \max_{S(\beta_V[i])} (U[j] + I[p]) \tag{2.15}$$

where $S(\beta_V[i])$ is the set of all possible symbol combinations that satisfy the parity check equation $\beta_V[i] \oplus \beta_U[j] \oplus \beta_I[p] = 0$ for $i, j, p = 0, \ldots, n_m - 1$.

5. *Post-processing*

In order to avoid the convergence of the messages to the largest value, a post-processing step is required where the smallest value in a message is subtracted from the rest of the values:

$$U_{vp_i}[k] = U_{vp_i}[k] - U_{vp_i}[n_m - 1] \tag{2.16}$$

$$V_{cp_j}[k] = V_{cp_j}[k] - V_{cp_j}[n_m - 1] \tag{2.17}$$

where $i = 0, \ldots, d_v - 1$, $j = 0, \ldots, d_c - 1$ and $k = 0, \ldots, n_m - 1$.

The complexity of the EMS algorithm is $O(n_m \log n_m)$ for parameter $n_m \leq q$.

2.2 Stochastic LDPC Decoding over GF(q)

Stochastic decoding is inspired by the technique of stochastic computing [34] where quantities are represented as Bernoulli sequences of bits and the information is conveyed through the statistics of the stream. This stream representation allows for complex computations to be implemented with simple hardware, and reduces the number of interconnecting wires required. The result of these reductions in complexity are circuits that can sustain higher clock rates [34].

The following sections show how stochastic decoding applies to the problem of decoding non-binary LDPC codes.

2.2.1 Stochastic Representation of Probabilities

A stochastic stream of bits can be used to represent a probability. For example, the probability $p = 0.461538462$ can be represented by the stream $s_1 = 0101001011010\ldots$ corresponding to $p = \frac{n_1}{n_t} = \frac{6}{13} = 0.461538462$ where n_1 is the number of times 1 was observed and n_t is the total number of received bits. Note that the order of the bits in the stream is not important and that all permutations of the bits in the stream are alternative representations of p.

The concept can be extended to the non-binary case where a stochastic stream of symbols can be used to represent the distribution of probabilities of the symbols. Let $\alpha, \beta, \gamma, \delta$ be the four possible symbols in GF(4). The non-binary stream $s_{nb} = \gamma\beta\alpha\alpha\gamma\beta\delta\alpha\gamma\alpha\alpha\gamma\ldots$ is equivalent to the following distribution of probabilities:

$$p_\alpha = \frac{5}{12} = 0.41(6),\ \ p_\beta = \frac{2}{12} = 0.16(6),\ \ p_\gamma = \frac{4}{12} = 0.33(3),\ \ p_\delta = \frac{1}{12} = 0.08(3)$$

The distributions created this way are always normalized because $\sum_{x \in GF(q)} n_x = n_t$ where n_x is the number or times symbol x has been observed and n_t is the total number of symbols.

2.2.2 Messages in Stochastic Decoding Algorithms

It is important to understand the difference in the nature of the messages in SPA or MSA, and the ones used in stochastic decoding. In stochastic decoding a codeword symbol x can be seen as a random variable defined on GF(q), and a message is symbol x itself [19]. The probability distribution associated with x can be inferred from statistical properties of the stochastic stream as shown in the previous section. In contrast, in SPA and MSA a message is an expression of the distribution of probabilities associated with x, a probability mass function (pmf). It can be represented as a collection of probabilities as done in Section 2.1.1, or as a collection of logarithm-domain likelihoods as in Section 2.1.3. In MSA and SPA messages are vectors of length q, while in EMS the length is reduced to $n_m \leq q$ (see Section 2.1.4).

The amount of information needed to represent an SPA or MSA message is qw bits (or $n_m w$ bits for EMS) where w is the number of bits required to store each probability or likelihood. The number of bits needed to represent a stochastic message is $\log q$. As it can be seen, the stochastic messages are considerably more compact than their SPA, MSA, and even EMS counterparts.

The small size of the stochastic message brings two benefits. Firstly, it reduces the hardware complexity of the non-binary decoders both in the number of interconnection wires and in the processing units themselves [20]. Secondly, shorter messages improve the average throughput when the received vector is close to a codeword and

few cycles are enough for convergence. With alternative approaches, even if one iteration is required, larger messages have to be passed between the nodes resulting in reduced throughput.

It is also important to point out the difference between an SPA, MSA, or EMS decoding iteration and a stochastic decoding iteration. In the former case during one iteration the nodes exchange full pmf messages, while in the stochastic case only one symbol is exchanged. To emphasize this difference, the stochastic decoding iteration is referred to as a decoding cycle (DC) [35].

Finally, SPA, MSA, and EMS decoders follow a deterministic trajectory, and when a local optimum is reached it results in decoding failure. Stochastic decoders, on the other hand, follow stochastic trajectories, meaning that repeating the decoding process can yield an alternative path that avoids the local optimum. This idea was exploited in [23] for binary decoders with a method called redecoding, which will be discussed in detail later.

2.2.3 Stochastic Decoding Over GF(q)

Stochastic decoding was proposed in [19] for the binary case. Early applications of the technique had limited decoding performance [21, 36]. The first successful application of stochastic decoding for LDPC codes was reported in [35], followed by a fully parallel decoder for binary LDPC codes [37].

The technique was extended to non-binary LDPC codes in [24] and consists of the following steps:

1. *Initialization*

   The $\{U_{vp_i}\}_{i=0,\ldots,d_v-1}$ messages are initialized with the initial likelihood values according to the channel model.

2. Variable node update

Given the variable node inputs V_{iv}, the output messages U_{vp} are computed:

$$U_{vp}(t) = \begin{cases} a & \text{if } V_{iv} = a \text{ for all } i \neq p \\ \xi & \text{otherwise} \end{cases} \tag{2.18}$$

where ξ is a random sample generated from the $U_{vp}(t)$ statistics.

3. Permutation step

The permutation operation is the same as in Equation (2.5).

4. Check node computation

The stochastic check node was initially proposed in [19] for the binary case. The non-binary check node output messages for a given edge p is computed by summing the input messages from all edges except p itself:

$$V_{cp}(t) = \sum_{i=1, i\neq p}^{d_c} U_{ic}(t) \tag{2.19}$$

where the sum is under GF(q).

A common problem for the stochastic decoders is latching, the undesired scenario when a group of nodes form a cycle and lock into a state of reduced or no switching resulting in poor bit-error-rate (BER) performance[35, 21]. Two solutions to latching are proposed in [35]. Firstly, edge memories (EM) and tracking forecast memories (TFM) [37] are introduced on the edges between the nodes in order to randomly reorder the symbols in the streams and thus break any correlation. Secondly, the Noise-Dependent Scaling (NDS) technique is used to increase switching activity.

2.2.4 Relaxed Half-Stochastic Decoding Over GF(q)

The Relaxed Half-Stocastic (RHS) decoding algorithm for LDPC codes was proposed in [22] and represents a combination of SPA and stochastic decoding techniques. The algorithm uses successive relaxation to convert stochastic streams into LLR values. In fact, an RHS decoder can be seen as a hybrid decoder operating in both LLR and stochastic domains. Structurally, the difference between an RHS and a stochastic decoder is the variable node, with the interleaver and the check nodes being identical.

In the non-binary version, Tracking Forecast Memories [37] are used to store the probabilities associated with the corresponding stochastic streams. For an incoming symbol $x \in GF(q)$ the memories are updated according to the following rule:

$$\mathrm{PMF}_t[i] = \begin{cases} (1-\beta)\mathrm{PMF}_{t-1}[i] + \beta & \text{if } i = x \\ (1-\beta)\mathrm{PMF}_{t-1}[i] & \text{otherwise} \end{cases} \tag{2.20}$$

for all $i \in GF(q)$ where $\mathrm{PMF}_t[i]$ is the probability of x being equal to symbol i at time t, and $\beta \leq 1$ is the relaxation paramter.

The RHS algorithm was successfully generalized for GF(q) and was shown in [23] to have a performance close to that of SPA as shown in Figure 2–3. Additionally, an optimized version of RHS called RD2 was introduced in [38] for the case when $d_v = 2$. It eliminates the need to perform term-by-term pmf multiplications in Equation 2.4 by updating the product of the pmfs directly as follows:

$$\mathrm{PMF}_t[i] = \begin{cases} (1-\beta)\mathrm{PMF}_{t-1}[i] + \beta L[i] & \text{if } i = x \\ (1-\beta)\mathrm{PMF}_{t-1}[i] & \text{otherwise} \end{cases} \tag{2.21}$$

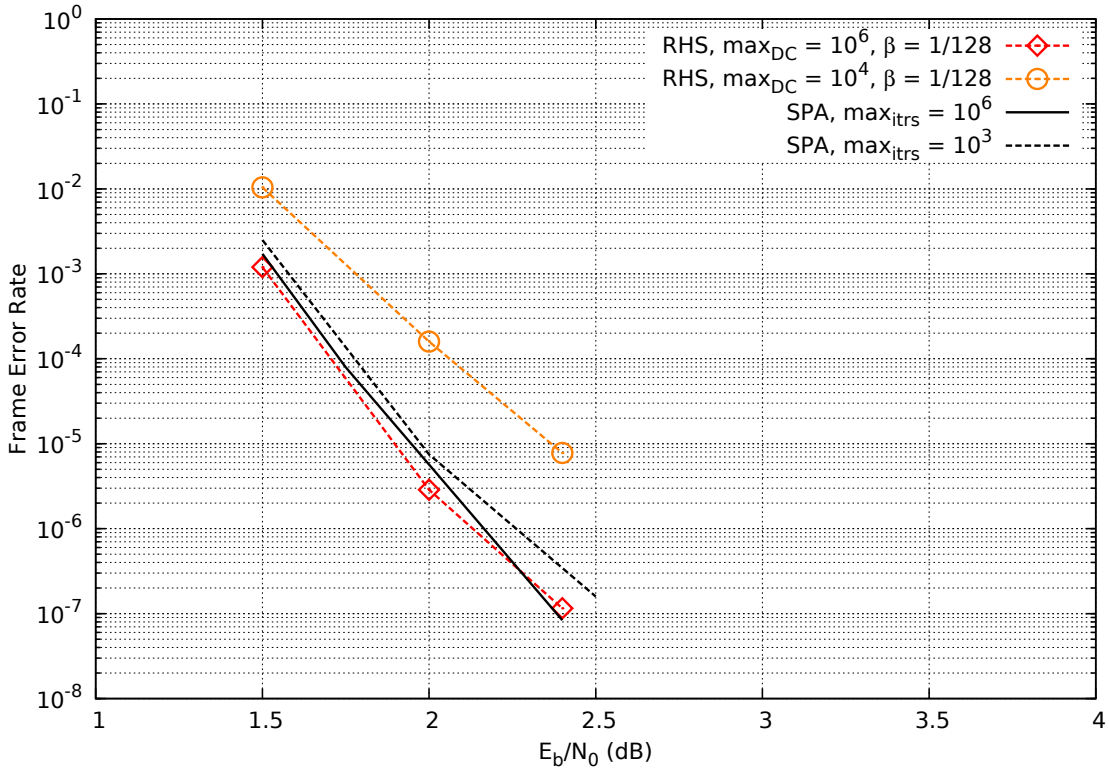

Figure 2–3: Comparison of SPA and RHS algorithm for a GF(64) (2,4)-regular code with $n = 192$ and $k = 96$ on the AWGN channel. The SPA results are shown for 10^6 and 10^3 maximum iterations while the RHS results are shown for 10^6 and 10^4 maximum decoding cycles.

where $L[i]$ is the likelihood of symbol i.

2.2.5 Redecoding

It has been shown in [22] that stochastic decoders are capable of decoding some of the frames that initially failed to decode. The method involves restarting the decoder with the same received soft-values vector but using a different random seed. This technique is called redecoding, and was shown to improve the BER performance

and lower the error floors. In [39] redecoding was generalized to dithered decoding algorithms, which can be applied to non-stochastic decoders as well.

2.3 Architectures and Implementations of GF(q) LDPC Decoders

In the case of GF(q) LDPC decoders there are some architectures proposed in the literature but there are few implementations (see Table 2–1). In [40] Spagnol et al. propose a serial architecture and FPGA implementation for a GF(8) LDPC decoder. In [41] a partially-parallel implementation of the EMS algorithm is given for GF(4). In [42] an architecture for decoding non-binary quasi-cyclic LDPC codes is proposed along with an ASIC implementation for a (620, 310) GF(32) code.

An architecture that works for higher order fields GF($q \geq 64$) is presented without implementation in [17]. An optimized version of this architecture with an FPGA implementation is provided in [18].

Table 2–1: Comparison of implementations of GF(q) LDPC decoders in literature

Implementation	[41]	[40]	[42]	[18]
Galois Field	GF(4)	GF(8)	GF(32)	GF(64)
Code	$N = 486$ $K = 972$	$N = 720$	$N = 620$ $K = 310$	$N = 192$ $K = 96$ (2,4)-regular
Type of architecture	partially-parallel	serial	partially-parallel	serial
Max. decoding iterations	not reported	15	10	8
Platform	FPGA	FPGA	ASIC	FPGA
Frequency	131.4 MHz	99.73 MHz	200 MHz	75 MHz
Throughput	50 Mbit/s	1.09 Mbit/s	60 Mbit/s	3.8 Mbit/s

Chapter 3

The Adaptive Multiset Stochastic Algorithm

In this chapter we introduce the Adaptive Multiset Stochastic (AMS) algorithm. First, we identify a data structure that allows for efficient storage, updating, and sampling of probability mass functions (pmf). Then, we define three operations on this data structure and show how they can be used for stochastic decoding.

Note that AMS algorithm proposes a new variable node computation while doing the permutation and check node updates as shown in Equation (2.5) and Equation (2.19), respectively. Therefore, the following sections will focus on the variable node computation.

3.1 Multiset Representation of a Probability Mass Function

As discussed in Chapter 2, in non-binary decoding, a GF(q) symbol s_i can be seen as a random variable and the associated pmf is a q-tuple of probabilities $(p_1, p_2, \ldots, p_q)$ where $p_j = P(s = \alpha_j)$ and $\alpha_j \in GF(q)$. Note that the pmf is normalized, i.e. $\sum_{j=1}^{q} p_j = 1$. In this section we show that multisets can be used as approximate representations for such pmfs.

Definition Let S be a multiset containing symbols from GF(q).

A multiset is a generalization of the concept of a set that allows for multiple instances of the same element. The cardinality of a multiset S is denoted by $|S|$ and represents the total number of instances of elements.

Definition Let $f_j = \frac{n_j}{|S|}$ be the probability of finding GF(q) symbol α_j in S, where n_j is the number of times α_j appears in S.

Definition Let s be a GF(q) symbol and let the pmf associated with it be defined by the probabilities $p_j = \Pr(s = \alpha_j)$.

Proposition 3.1.1 *There exists a multiset S such that $|p_j - f_j| < \varepsilon$ for all $\varepsilon > 0$ and $1 \leq j \leq q$.*

Proof Let n be a large integer, then we can set $n_j = \lfloor np_j \rfloor$. Then

$$f_j = \frac{\lfloor np_j \rfloor}{n} = \frac{np_j - \mathrm{frac}(np_j)}{n} = p_j - \frac{\mathrm{frac}(np_j)}{n}$$

where frac(x) is the fractional part of x. Observe that $0 \leq \mathrm{frac}(np_j) \leq 1$ and that it is always possible to find a large enough value for n such that

$$\frac{\mathrm{frac}(np_j)}{n} < \varepsilon.$$

So it is always possible to build a multiset S that satisfies the condition. ∎

As it is shown in Table 3–1, a multiset representation of a pmf allows for the same operations as a regular pmf. In order to increase a probability p_α by Δ_1, one or more instances of symbol α are added to S. The exact number of instances to add, k_1, can be calculated from the following equation:

$$\frac{n_\alpha}{|S|} + \Delta_1 = \frac{n_\alpha + k_1}{|S| + k_1}$$

Table 3–1: Equivalence of operations on a pmf and a multiset representation of a pmf

	pmf	multiset representation of pmf
Increase probability	$p_\alpha \leftarrow p_\alpha + \Delta_1$	$S \leftarrow S \cup \{k_1 \text{ instances of } \alpha\}$
Decrease probability	$p_\alpha \leftarrow p_\alpha - \Delta_2$	$S \leftarrow S \setminus \{k_2 \text{ instances of } \alpha\}$
Normalization	$p_i \leftarrow \frac{p_i}{\sum_{j=1}^{q} p_j}$ for all i	Not needed. S is implicitly normalized.
Sampling	$c_i \leftarrow \sum_{j=1}^{i} p_j$ $r \leftarrow U(0,1)$ $s \leftarrow \underset{i}{\text{argmin}}\lvert c_i - r\rvert$	$s \leftarrow$ a random symbol from S.

and solving for k_1 gives:

$$k_1 = \frac{\Delta_1 |S|}{1 - \frac{n_\alpha}{|S|} - \Delta_1}$$

Note that only an integer number of symbols can be added to S, either $\lfloor k_1 \rfloor$ or $\lceil k_1 \rceil$, depending on the desired strategy. In order to decrease p_α by Δ_2 a number k_2 of α symbols need to be removed from S, and is calculated similarly.

The steps required in order to sample a normalized pmf are presented in Table 3–1 and include calculating a cumulative density function (CDF) shown as c_i, and finding i where c_i is closest in value to r, a uniform random variable between 0 and 1. In contrast, sampling a multiset representation of a pmf is trivial, and is equivalent to picking a random symbol from S.

Finally, normalization is not needed when using a multiset representation because the probabilities are represented as ratios that sum up to 1 at all times:

$$\sum_{j}^{q} \frac{n_j}{n} = \frac{\sum_j^q n_j}{n} = \frac{n}{n} = 1$$

As it can be seen from Proposition 3.1.1 and Table 3–1, a multiset can be used as an approximate representation of a probability mass function that has the advantage of being simple to sample and not requiring normalization. The following sections will show how multisets can be used for stochastic decoding of regular LDPC codes with $d_v = 2$.

3.2 Algorithm Definition and Analysis

As it was shown in Section 2.2.4, the RHS algorithm allows to perform stochastic decoding using SPA nodes. This is achieved by using pmfs to keep track of changes in the statistics of the stochastic streams incoming to the variable node, or, as done in [38] for the $d_v = 2$ case, to keep track of changes in the statistics of the stochastic streams outgoing from the variable node. There are two disadvantages to using pmfs for these purposes. Firstly, the update Equations (2.20) and (2.21) require that q probabilities are recalculated at each update, where q is the number of symbols in GF(q). Secondly, sampling such a pmf takes up to q steps and requires computing or updating a cumulative density function.

The Adaptive Multiset Stochastic algorithm proposed in this work, addresses these problems by using multisets instead of pmfs. Similar to the algorithm in [38], in the case of $d_v = 2$ codes it allows to generate the variable node outputs without explicitly performing the product in Equation (2.4), but by associating multisets to edges and updating them. Figure 3–1 gives the structure of a variable node of degree two with its input symbols α_0 and α_1, which, together with the corresponding likelihoods l_0 and l_1, are used to update the multisets S_0 and S_1. Note that the output symbols β_0 is a sample from S_1 and output symbols β_1 is a sample from S_0.

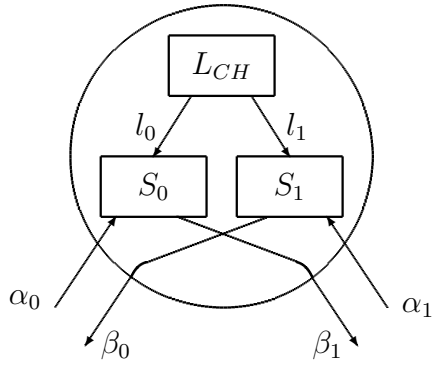

Figure 3–1: Schematic representation of a variable node of degree $d_v = 2$ with the multisets S_0 and S_1, and the channel likelihoods table L_{CH}. The incoming messages are α_0 and α_1, and the outgoing messages are β_0 and β_1.

In the context of this configuration, we algorithmically define three routines that operate on the multisets S_0 and S_1: the *Add* routine (Algorithm 1), the *Remove* routine (Algorithm 2), the the *Sample* routine (Algorithm 3). Together these routines constitute the Adaptive Multiset Stochastic algorithm.

3.2.1 The *Add* Routine

The *Add* routine updates a multiset S by adding zero or more instances of incoming symbol α to it. If no symbols are added, obviously, the multiset S is unchanged, but when one or more symbols are added the probability p_α associated with symbol α is increased while the probabilities of all other symbols are decreased. The routine makes use of the floor operator because only an integer number of symbols can be added to S. The fractional part is compared against the uniform random variable R_1 to decide weather or not an additional symbol should be added. As it is shown in Proposition 3.2.1, the expected number of symbols added by this routine is $l_\alpha(M - |S|)$. The term $M - |S|$ is the difference between the maximum

Algorithm 1: The *Add* routine of AMSA. S is the multiset to be added to, M is the upper bound on $|S|$, α is the incoming symbol, l_α is the channel likelihood of symbol α.

Input: S, symbol α, l_α
Output: S with zero one or more instances of symbol α added

```
1 R_1 ← uniform random real value from interval (0, 1)
2 x ← l_α · (M − |S|)
3 if frac(x) ≥ R_1 then
4 |   k ← ⌊x⌋ + 1
5 else
6 |   k ← ⌊x⌋
7 end
8 S ← S ∪ { k instances of α}
```

capacity of S and its current size, and can be interpreted as the empty part of S, or the spare capacity of S.

Proposition 3.2.1 *For incoming symbol α, the expected number of symbols added by the Add routine to S is $l_\alpha(M - |S|)$.*

Proof The *Add* routine is an experiment that can have two outcomes: $\lfloor x \rfloor$ symbols are added, or $\lfloor x \rfloor + 1$ symbols are added, where $x = l_\alpha(M - |S|)$. Using the notation from Algorithm 1, the probabilities of the outcomes are given by:

$$\Pr(k = \lfloor x \rfloor + 1) = \Pr(\text{frac}(x) \geq R_1) = \text{frac}(x)$$

$$\Pr(k = \lfloor x \rfloor) = \Pr(\text{frac}(x) < R_1) = 1 - \text{frac}(x)$$

The expected value of k is:

$$\begin{aligned} E(k) &= \lfloor x \rfloor (1 - \text{frac}(x)) + (\lfloor x \rfloor + 1)\text{frac}(x) \\ &= \lfloor x \rfloor - \lfloor x \rfloor \text{frac}(x) + \lfloor x \rfloor \text{frac}(x) + \text{frac}(x) \\ &= \lfloor x \rfloor + \text{frac}(x) \\ &= x = l_\alpha(M - |S|) \end{aligned}$$

∎

Now let us examine what is the effect of adding k instances of symbol α to S on p_α, the probability of α according to S, and p_β, where β is a symbol from GF(q) other than α. Before the addition we have $p_\alpha(t) = \frac{n_\alpha}{|S|}$ and $p_\beta(t) = \frac{n_\beta}{|S|}$, and after addition:

$$\begin{aligned} p_\alpha(t+1) &= \frac{n_\alpha + k}{|S| + k} \\ p_\beta(t+1) &= \frac{n_\beta}{|S| + k} \end{aligned}$$

Expressing $p_\alpha(t+1)$ in terms of $p_\alpha(t)$:

$$\begin{aligned} \frac{p_\alpha(t+1)}{p_\alpha(t)} &= \frac{n_\alpha + k}{|S| + k} \cdot \frac{|S|}{n_\alpha} \\ p_\alpha(t+1) &= p_\alpha(t) \cdot \frac{|S|}{|S| + k} \cdot \frac{n_\alpha + k}{n_\alpha} \\ &= p_\alpha(t) \cdot \left(1 - \frac{k}{|S| + k}\right) \cdot \left(1 + \frac{k}{n_\alpha}\right) \\ &= p_\alpha(t) \cdot \left(1 - \frac{k}{|S| + k}\right) + p_\alpha(t) \left(1 - \frac{k}{|S| + k}\right) \frac{k}{n_\alpha} \\ &= p_\alpha(t) \cdot \left(1 - \frac{k}{|S| + k}\right) + \frac{n_\alpha}{|S|} \frac{|S|}{|S| + k} \frac{k}{n_\alpha} \\ &= p_\alpha(t) \cdot \left(1 - \frac{k}{|S| + k}\right) + \frac{k}{|S| + k} \end{aligned}$$

And if we substitute k by its expected value $l_\alpha(M - |S|)$ from Proposition 3.2.1:

$$p_\alpha(t+1) = p_\alpha(t) \cdot \left(1 - \frac{l_\alpha(M-|S|)}{|S| + l_\alpha(M-|S|)}\right) + \frac{l_\alpha(M-|S|)}{|S| + l_\alpha(M-|S|)}$$

Let us denote $\omega_\alpha = \frac{l_\alpha(M-|S|)}{|S|+l_\alpha(M-|S|)}$, then the equation becomes:

$$p_\alpha(t+1) = p_\alpha(t) \cdot (1 - \omega_\alpha) + \omega_\alpha$$

Similarly, expressing $p_\beta(t+1)$ in terms of $p_\beta(t)$ we get:

$$p_\beta(t+1) = p_\alpha(t) \cdot (1 - \omega_\alpha)$$

Now, we can write the update equation of the *Add* routine for the incoming symbol $\alpha \in GF(q)$ as:

$$p_s[t+1] = \begin{cases} (1-\omega_\alpha) \cdot p_s[t] + \omega_\alpha & \text{if } s = \alpha \\ (1-\omega_\alpha) \cdot p_s[t] & \text{otherwise} \end{cases} \tag{3.1}$$

where $p_s[t]$ is the probability of symbol s at time t. Note that this update maintains the probabilities normalized, i.e. $\sum_{s \in GF(q)} p_s[t] = 1$ for all t.

Equation (3.1) is recognizable as the RHS update from Equation (2.20) but instead of using a constant term β it uses ω_α, a function of the likelihood l_α corresponding to incoming symbol α.

This result is confirmed by experimental data. Figure 3–2 shows how the probability of the correct symbol evolves during the decoding process in an edge memory using the non-binary RHS update from Equation (2.20) and AMSA as defined in this section. At any given point in time, the difference between the two probabilities is due to the fact that the multiset used by AMSA is only an approximation of a pmf.

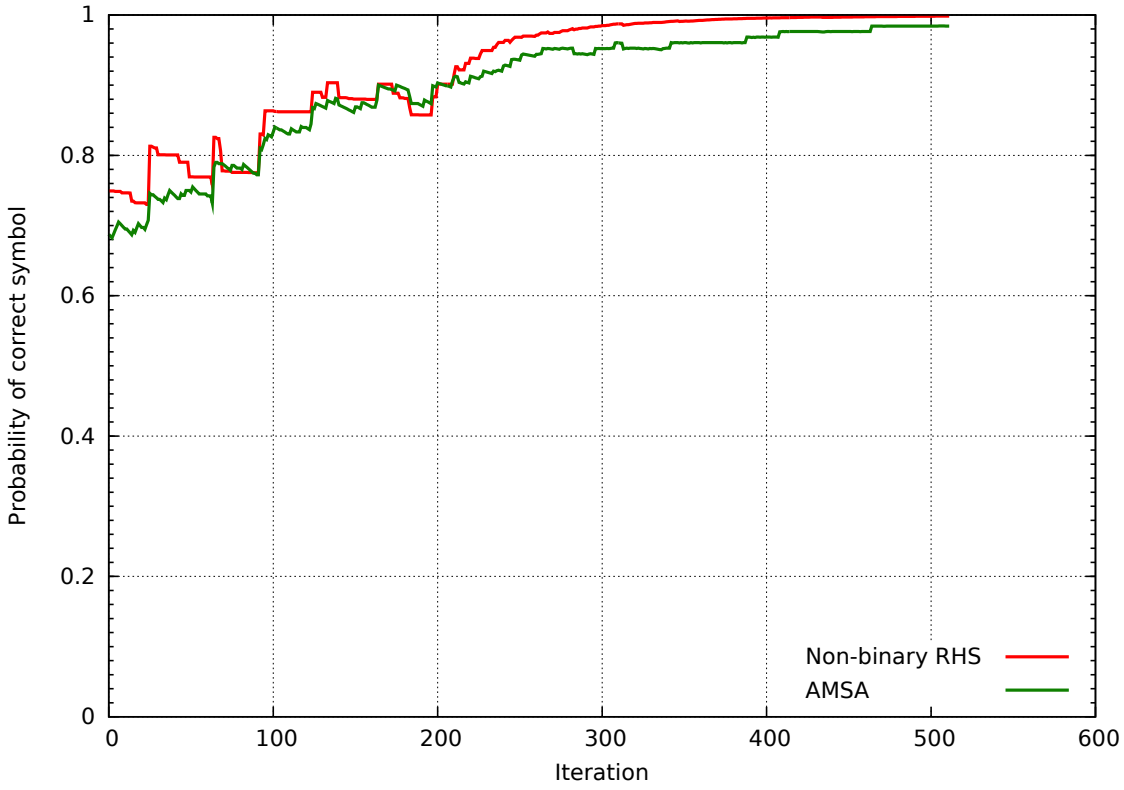

Figure 3–2: Comparison of the non-binary RHS algorithm and AMSA by tracking the probability of the correct symbol in an edge memory. The maximum size of the multiset is $M = 128$.

3.2.2 The *Remove* Routine

The goal of the *Remove* routine is to uniformly and gradually remove symbols from multiset S. The decision whether to remove a symbol or not is based on the result of a probabilistic experiment of comparing the value of a uniform random variable R_2 from the interval $[1, M]$ to $|S|$. As it is shown in Proposition 3.2.2, the expected number of symbols removed by *Remove* is $\frac{|S|}{M}$. Intuitively, that means that when $|S|$ is closer in value to M, i.e. S is close to maximum capacity, it is more likely that a symbol will be removed, and as $|S|$ approaches zero, i.e. S becomes empty, it is less likely that a symbol will be removed. Note that the *Remove* routine enforces

Algorithm 2: The *Remove* routine of AMSA

Input: S
Output: S with possibly one symbol removed
$R_2 \leftarrow$ uniform random intger from interval $[1, M]$
if $R_2 < |S|$ **then**
  $S \leftarrow S \setminus \{\text{random symbol from } S\}$
end

the lower bound $1 \leq |S|$ on the cardinality of S, because the condition $R_2 < 1$ is never true.

Proposition 3.2.2 *The expected number of symbols removed by the Remove routine in one invocation is* $\frac{|S|}{M}$.

Proof Let r be the random variable associated with the number of symbols removed by the *Remove* routine, then $r \in \{0, 1\}$ because either one or zero symbols are removed. The probabilities of the outcomes are given by:

$$\Pr(r = 1) = \Pr(R_2 \leq |S|) = \frac{|S|}{M}$$
$$\Pr(r = 0) = 1 - \frac{|S|}{M}$$

The expected value of r is:

$$E(r) = 1 \cdot \frac{|S|}{M} + 0 \cdot \left(1 - \frac{|S|}{M}\right) = \frac{|S|}{M}$$

∎

Lemma 3.2.3 *Let* n_α *be the number of times symbol* α *appears in* S*, then the probability that the Remove routine removes symbol* α *is* $\frac{n_\alpha}{M}$.

Proof From Proposition 3.2.2, the probability of removing a symbol is $\frac{|S|}{M}$. The probability of finding symbol α is S is $\frac{n_\alpha}{|S|}$. The probability of removing a symbol α is equal to the product of the above probabilities and is equal to $\frac{n_\alpha}{M}$. ∎

As it is shown in Proposition 3.2.4, *Remove* does not change the expected value of the probabilities of the symbols in S. This means that by invoking the *Add* and *Remove* routines, the expected values of the probabilities will be updated as in Equation 3.1.

Proposition 3.2.4 *Let $p_\alpha(t)$ be the probability of symbol α in S, and let $E(p_\alpha(t+1))$ be the expected probability of the same symbol after the invocation of Remove, then $E(p_\alpha(t+1)) = p_\alpha(t)$, i.e. the expected value of the probability is not changed by Remove.*

Proof Looking at a symbol α from S, the *Remove* routine is an experiment with three outcomes: no symbol is removed, symbol α is removed, and another symbol $\beta \neq \alpha$ is removed. Let p_{oi} be the probability of outcome i, then $p_{o1} = 1 - \frac{|S|}{M}$, and, from Lemma 3.2.3, $p_{o2} = \frac{n_\alpha}{M}$, and, finally, $p_{o3} = \frac{|S|-n_\alpha}{M}$.

When no symbol is removed, the probability is unchanged, and equal to $\frac{n_\alpha}{|S|}$. When α is removed, its new probability is $\frac{n_\alpha-1}{|S|-1}$. Finally, when a symbol other than

Algorithm 3: The *Sample* routine of AMSA

Input: S
Output: A random symbol α from S
1 $\alpha \leftarrow$ a random symbol from S

α is removed, the probability of α is $\frac{n_\alpha}{|S|-1}$. We can now compute the expected value:

$$\begin{aligned}
E(p_\alpha(t+1)) &= \left(1 - \frac{|S|}{M}\right)\frac{n_\alpha}{|S|} + \frac{n_\alpha}{M}\frac{n_\alpha - 1}{|S|-1} + \frac{|S| - n_\alpha}{M}\frac{n_\alpha}{|S|-1} \\
&= \left(1 - \frac{|S|}{M}\right)\frac{n_\alpha}{|S|} + \frac{n_\alpha^2 - n_\alpha + |S|n_\alpha - n_\alpha^2}{M(|S|-1)} \\
&= \left(1 - \frac{|S|}{M}\right)\frac{n_\alpha}{|S|} + \frac{n_\alpha}{M} \\
&= \frac{n_\alpha}{M}\left(\frac{M - |S|}{|S|} + 1\right) \\
&= \frac{n_\alpha}{|S|} = p_\alpha(t)
\end{aligned}$$

∎

3.2.3 The *Sample* Routine

The *Sample* routine is used to generate samples according to the probabilities of the symbols in S. Unlike *Add* and *Remove*, this routine does not modify S. The probability that the symbol returned by *Sample* is α is equal to p_α, the probability of finding α in S, and $p_\alpha = \frac{n_\alpha}{|S|}$, which is equivalent to sampling a pmf with the same symbol probabilities.

3.3 Non-Binary LDPC Decoding with AMSA

As shown in Figure 3–1, for a variable node of degree two, AMSA uses two multisets, S_0 and S_1, each associated with an edge. The *Add* and *Remove* routines

update the multisets, while the *Sample* routine generates the output symbols. The steps of a decoding cycle using AMSA is given below:

1. *Initialization*

   The decoding process starts when the soft-decision sequence $\mathbf{y}$ is received from the channel. The decoder front-end uses $\mathbf{y}$ to compute, for each variable node, a set of initial likelihoods l_j for all symbols in GF(q) where $j = 1, \ldots, q$. Additionally, M samples are generated according to the likelihoods l_j and added to the multisets S_0 and S_1.

2. *Variable node update*

   The processing of the variable node can be done in parallel. Keeping the notation from Figure 3–1 it can be expressed with the following invocations of the routines:

   $$\begin{array}{ll} Remove(S_0) & Remove(S_1) \\ Add(S_0, \alpha_0, l_0) & Add(S_1, \alpha_1, l_1) \\ \beta_1 \leftarrow Sample(S_0) & \beta_0 \leftarrow Sample(S_1) \end{array}$$

   Additionally, the variable node computes the belief $\underset{i}{\operatorname{argmax}}\, l_i$ where $i \in \{0, 1\}$, and l_i is the likelihood of the input symbol α_i.

3. *Permutation step*

   The permutation operation is the same as in Equation (2.5).

4. *Check node computation*

   The check node computation is done as in Equation (2.19).

5. *Tentative decoding*

   If the beliefs computed in the variable nodes satisfy all the parity checks, decoding stops. Otherwise decoding continues with the next decoding cycle.

Table 3–2: The number of operations required and run-time complexity for fully-parallel implementations of the AMSA algorithm

	Number of operations	Run-time complexity of a fully-parallel implementation (regular memory)	Run-time complexity of a fully-parallel implementation (custom SRAM)
Add	$O(d_v q)$	$O(q)$	$O(1)$
Remove	$O(d_v)$	$O(1)$	$O(1)$
Sample	$O(d_v)$	$O(1)$	$O(1)$
Variable Node	$O(d_v q)$	$O(q)$	$O(1)$
Check Node	$O(d_c)$	$O(1)$	$O(1)$
Permutation	$O(1)$	$O(1)$	$O(1)$
Belief	$O(d_v)$	$O(1)$	$O(1)$

Decoding continues until all the parity checks are satisfied, or until a maximum preset number of decoding cycles is reached.

3.4 Complexity Analysis

This section summarizes the computational complexities for the variable node, check node, and permutation operation within AMSA. It also discusses the implications of fully-parallel circuit implementations on the run-time complexity of the decoder.

In the first column of Table 3–2 are presented the upper bounds on the number of operations needed for each stage of the decoding process. Note that in the case of the *Add* routine, the complexity is more intuitively $O(d_v M)$, but since M scales linearly with q for all practical purposes, it was presented as $O(d_v q)$ to simplify comparison with other results in the literature.

Table 3–3: Summary of space complexity for AMSA

	Space complexity (bits)	Comments
Memories	$O(d_v M \log q + qw)$	d_v edge memories and q w-bit likelihoods memories
Variable Node logic	$O(\log M)$	determined by size of memory indexes
Check Node	$O(1)$	no memory required
Permutation	$O(q \log q)$	$q \times \log q$ lookup table

The second column of the table gives the upper limit on the run-time of the algorithm computations using regular memory, where by regular memory we understand a memory unit that requires $O(k)$ steps in order to write k values. This approach was used for the FPGA implementation presented in Chapter 4. In the cases where operations could be carried in parallel *and* implemented as such on hardware, the corresponding reduction in complexity was noted. For instance, the *Sample* routine is shown to require $O(d_v)$ operations because we have to execute it on each of the d_v edges of the variable node. However, in the FPGA implementation all d_v edges are instantiated and can execute the routine in parallel in $O(1)$ time. Indeed, in this FPGA implementation, all the edges of a variable node are sampled in parallel in 1 clock-cycle. Similarly, the rest of routines can be parallelized to reduce the run-time complexity.

The third column provides the run-time complexities when using the custom SRAM architecture (see Section 4.7.1), which is possible on ASIC platforms. In this case the *Add* routine becomes $O(1)$ allowing to perform all the variable node computations also in $O(1)$. This reduction in run-time complexity of AMSA results in increased throughput of the decoder. See Section 5.2 for details.

Table 3–3 presents the memory space requirements for the AMSA algorithm. Note that a GF(q) symbol is represented in hardware by $\log q$ bits and that the multisets are implemented as memories, while w is a quantization parameter - the number of bits used to represent probabilities.

This thesis provides the fully-parallel hardware implementations for a GF(64) and a GF(256) decoder in FPGA. In fact, with AMSA, even higher order fields are feasible like GF(512), as shown in Section 4.6. More details about fully-parallel FPGA and ASIC implementations of the AMSA algorithm are given in Chapter 4.

3.5 Redecoding

As it can be seen from the description of the *Add*, *Remove*, and *Sample* routines, they make use of three uniform random variables (R_1 and R_2). The use of random variables during decoding is characteristic to stochastic decoding and makes the process non-deterministic. It has been shown in [22] that if a stochastic decoder fails to decode a codeword, it can succeed by trying to decode it again using different seeds for the pseudo-random number generators.

Redecoding can be seen as a tradeoff mechanism between error-rate performance and latency. A redecoding configuration has two parameters: the number of attempts r_a, and the maximum number of decoding cycles for each attempt $\max_{DC}$. These parameters can be changed at run-time making the AMS decoder suitable for variable latency application.

Latency scales with the $r_a \max_{DC}$ product and with $1/f$ where f is the working frequency of the hardware implementation.

The technique of redecoding has been successfully extended to non-binary stochastic decoding in this work in order to improve performance and remove the error floor. The impact of redecoding on the performance can be observed in Figure 5–1.

Chapter 4

Circuit Implementation

In this chapter we look at two fully-parallel FPGA implementations of the AMSA decoder. Let AMSA-M be a configuration of the AMSA algorithm, one that uses multisets of maximum size M. The first decoder is GF(64) AMSA-128 meaning that it uses a GF(64) code and that parameter M is equal to 128. The second implemented decoder is GF(256) AMSA-512, and as it will be shown in Section 4.6, its size confirms the complexity analysis done in Section 3.4.

Additionally, in Section 4.7.1, an optimized Static Random-Access Memory (SRAM) architecture is proposed for ASIC implementations. It further reduces the run-time complexity of a decoding cycle to $O(1)$ while simplifying the control logic of the decoder.

In what follows, GF(q) symbols are represented as $\log q$ bit words. Symbol likelihood values l_j are represented in fixed-point format. Note that l_j are positive sub-unitary numbers, which in the case of w-bit quantization means that the smallest non-zero value that can be represented is $(0.0000\ldots0001)_2 = 2^{-w}$ and the largest value is $(0.1111\ldots111)_2 = \sum_{k=1}^{w} 2^{-k} = 1 - 2^{-w}$. For details on the efficient quantization scheme used in this implementation see Section 5.3.

Both the GF(64) and GF(256) DAVINCI codes used in this work are (192,96) (2,4)-regular codes, meaning that there are 192 degree-two variable nodes and 96 degree-four check nodes. The GF(64) code has only 9 different non-zero values in its parity-check matrix and the GF(256) code has only 4 distinct non-zero values in the parity-check matrix.

The FPGA platform used for implementation is the EP4SGX230-KF40C2 chip from the Altera Stratix IV GX family. It provides, among other things, 182,400 Adaptive Lookup Tables (ALUTs), 182,400 registers, 1,235 M9K memory blocks, and 1,288 18x18-bit Digital Signal Processing (DSP) blocks. As it is shown in Table 4–3, these resources are more than sufficient for the fully-parallel implementation of both GF(64) AMSA-128 and GF(256) AMSA-512 decoders.

4.1 Structure of fully-parallel decoders

The GF(64) AMSA-128 and the GF(256) AMSA-512 decoders will include 192 variable nodes, 96 check nodes, and 384 edges with an edge memory on each. Table 4–1 gives a more detailed comparison of the decoders. In the next sections, we will present the implementation details of the main components of the AMSA decoders.

4.2 Variable Node

A degree-two variable node uses multisets S_0 and S_1 on its edges, and a likelihoods table, as it was illustrated earlier in Figure 3–1. In a hardware implementation, each of these is represented by a memory. Much of the AMSA variable node computation is, in fact, updating these memories depending on the input symbols and the internal state of the node. From an architectural point of view, the variable node is

Table 4–1: Comparison of GF(64) AMSA-128 and GF(256) AMSA-512 fully-parallel decoders

	GF(64) AMSA-128	GF(256) AMSA-512
Variable Nodes	192	192
Check Nodes	96	96
Edge Memories	384	384
Permutation Blocks	1152	1152
Length of symbols	6 bits	8 bits
Length of memory indices	7 bits	9 bits
Memory size	128×6 bits	512×8 bits
Length of Random Number Generator	32 bits	32 bits
Quantization parameter, w	12 bits	12 bits

a state machine that controls the reading and writing operations to the memories. The state machine is defined in Section 4.2.7.

4.2.1 Pseudo-Random Number Generator

The *Add* and *Remove* routines (Algorithms 1 and 2) make use of the uniform random variables R_1 and R_2, respectively. The *Sample* routine also uses random bits to select a random symbol from S. An efficient and practical way to generate pseudo-random numbers is to use linear feedback shift registers (LFSR) that can achieve a maximum sequence period of $2^n - 1$ where n is the length of the register in bits.

Each variable node uses a 32-bit LFSR with the $x^{31} + x^{21} + x + 1$ feedback polynomial, illustrated in Figure 4–1, that achieves the maximum-length sequence of $2^{32} - 1$ [43]. The uniform random variables R_i can be mapped to bit ranges of the

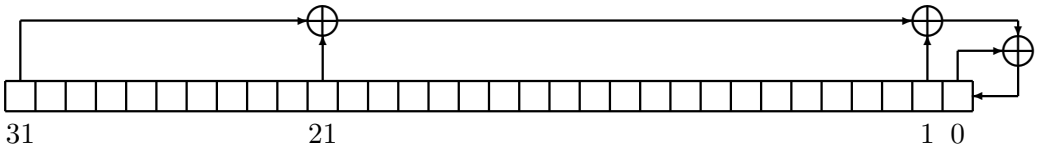

Figure 4–1: The 32-bit LFSR with the feedback taps corresponding to the polynomial. The symbol $\oplus$ represents modulo 2 addition which is implemented with XOR gates.

LFSR or mapped to bits in any other order. Several such mappings were considered in this work, all of them having a negligible effect on the overall performance of the decoder. Variable R_1 is w-bit in length because it is used for comparisons with the likelihood values. Variable R_2 has a length of $\log M$ bits because it is used as an index in memory arrays of length M. Similarly, *Sample* uses a random memory index of $\log M$ bits to select a random element from the memory representing the multiset S.

4.2.2 Hardware Representation of the Likelihoods Table

The likelihood values are computed from the channel soft sequence $\mathbf{y}$ and are denoted throughout this work as l_j where $j = 1, \ldots, q$. Each likelihood, valued between 0 and 1, is represented in fixed-point format using w bits.

As in the case of the edge memories shown below, the q likelihood values are organized in a $q \times w$ bits dual-port memory. Within each variable node the *Add* and the belief computation routines make use of the likelihood values. In both cases the access is read-only. The only time the likelihood memory is written to is during the initialization phase of the algorithm.

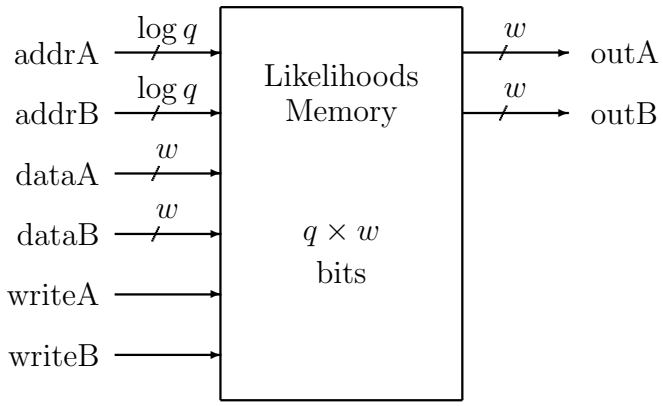

Figure 4–2: The interface of a likelihoods memory corresponding to the probabilities l_j where $j = 1, \ldots, q$. Values are represented using w bits.

4.2.3 Edge Memories - a Hardware Representation of the Mulitset S

As presented in Section 3.2, in AMSA, we associate multisets S_0 and S_1 with the edges of a variable node. We also impose upper and lower bounds on the cardinality of these multisets $1 \leq |S| \leq M$.

The hardware representation for a multiset is a memory array of length M. For practical reasons and efficient utilization of memory resources M is chosen to be a power of 2. On the Altera Stratix IV FPGA platform used in this work, the so-called M9K memory blocks were used for this purpose. Each block has a capacity of 8192 bits with configurable dual read-write ports. Note that the space required to store $M = 128$ GF(64) symbols is $128 \log 64 = 768$ bits, meaning that decoders with larger M or q will fit in the same number of memory blocks.

For a length-n (d_v, d_c)-regular code, nd_v such blocks will be instantiated in the fully-parallel decoder. Both codes used in this work have 384 edges in the Tanner graph, therefore we use 384 M9K memory blocks to represent the associated multisets.

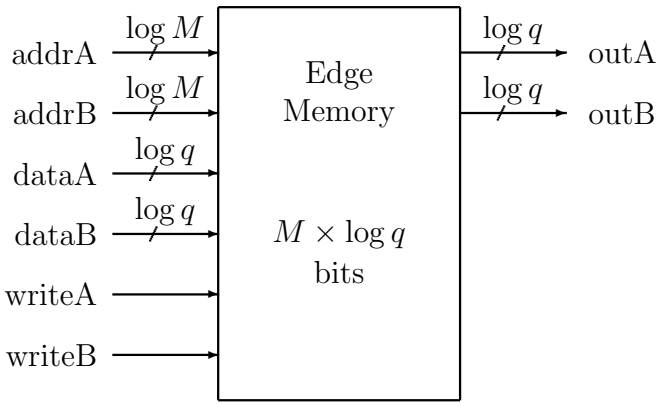

Figure 4–3: The interface of an edge memory corresponding to a multiset containing at most M symbols from GF(q).

4.2.4 Hardware Implementation of the *Remove* Routine

From the mathematical point of view, the *Remove* routine randomly removes a symbol from the multiset S, which is an unordered collection. Additionally, as shown in Proposition 3.2.4, it removes symbols uniformly, such that the expected value of the probabilities of symbols in S does not change.

On the hardware implementation, we represent S with a memory of length M, which is implicitly an ordered collection. Normally, if the order of the elements had to be preserved, removing an arbitrary element from the memory would imply shifting up to $M - 1$ elements by one position to the left, which would take $O(M)$ steps to perform.

Fortunately, in AMSA we are not concerned with the order of the elements in the memory, a property inherited from the unordered multisets. In this case, removing an element can be achieved by overwriting it with the last element in sequence and reducing the length of the sequence by one. This operation takes only $O(1)$ steps.

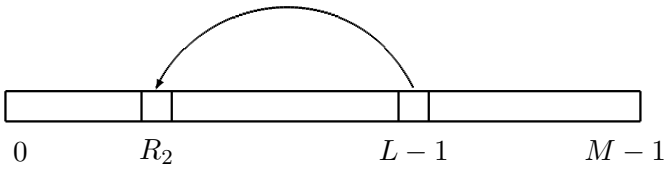

Figure 4–4: Removing a symbol from memory at random index $R_1 < L$ by overwriting it with the symbol at index $L-1$ and, finally, decrementing L. Note that here all indexes are zero-based.

Algorithm 4: The set of commands implementing the *Remove* routine.

```
1 R2 ← LFSR
2 if R2 < L then
3 |   MEM[R2] ← MEM[L − 1]
4 |   L ← L − 1
5 end
```

Note that, in Figure 4–4, $L = |S|$ is the number of symbols in the multiset and $M - L$ is the size of the unused, or empty, segment of the memory. The hardware implementation of the *Remove* routine from Algorithm 2 can be expressed as in Algorithm 4, where $MEM[i]$ represents the ith element in the memory array. Note that it is not necessary to fetch from memory the value $MEM[L-1]$ each time because it can be stored and updated in a register.

In terms of hardware, the routine is very simple to implement. The $R_2 < L$ test is done by a $\log M$-bit comparator which controls the read-write mode of the edge memory. As shown in Figure 4–5, the index L is stored in a register and is decremented if a symbol is removed. The last symbol in the sequence, the one at index $L-1$ and denoted by *LastSymbol* is also registered in order to avoid the need to fetch its value from the memory before writing, and is updated according to the

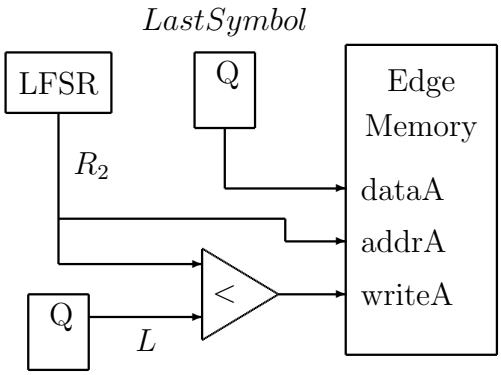

Figure 4–5: Circuit for the *Remove* routine.

following rule:

$$LastSymbol = \begin{cases} \alpha & \text{if } k > r \\ LastSymbol & \text{if } k = r \\ MEM[L-1] & \text{if } k < r \end{cases}$$

where k is the number of instances of symbol α added by the *Add* routine, and r is the number of symbols removed by the *Remove* routine. Note that in the last case, since r is at most one, k is zero, meaning that no symbols need to be added. This allows for the memory to be read to update *LastSymbol*.

4.2.5 Hardware Implementation of the *Add* Routine

Following the same conventions and notation, a multiset S with a maximum cardinality of M but containing $L < M$ elements is equivalent, in our case, to a memory array of length M partitioned into two segments: one with L elements stored at indexes from 0 to $L-1$ and an unused segment from index L to $M-1$. In this context adding k instances of a symbol to the multiset S (Algorithm 1 line 8) is

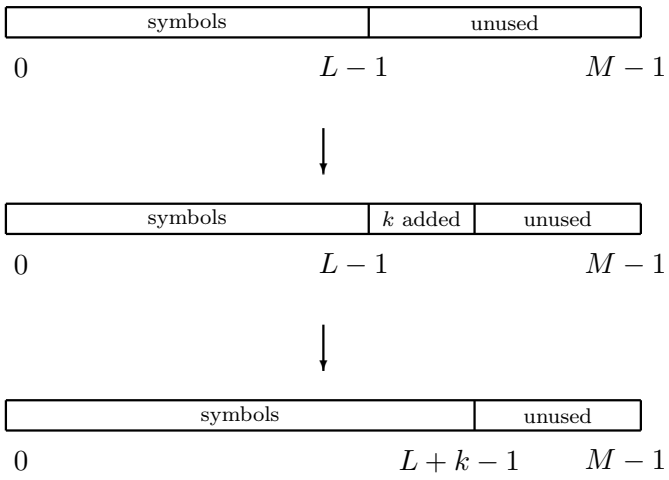

Figure 4–6: Adding k instances of a symbol to the memory, the equivalent of the $S \leftarrow S \cup \{$ k instances of $\alpha_j\}$ operation in the *Add* routine.

equivalent to extending the first segment by k positions and implicitly reducing the unused part by the same amount. This is illustrated in Figure 4–6.

On FPGA, the task of writing k symbols to the edge memory is done by the state machine defined in Section 4.2.7. The run-time complexity of this approach is $O(q)$. The constant factor in $O(q)$ can be improved on FPGAs by using multi-port memory blocks, using multiple parallel memory blocks instead of a single memory block, or configuring the memory blocks to fit multiple symbols in each memory word.

On ASIC, with a special memory architecture, the state machine is not needed for this operation, and the run-time complexity can be reduced to $O(1)$.

The circuit for computing k, the number of symbols to add, is given in Figure 4–7, and it uses a $w \times \log M$-bit multiplier and two w-bit comparators and two multiplexers. On the FPGA system used here, each of the 1,288 DSP blocks provides

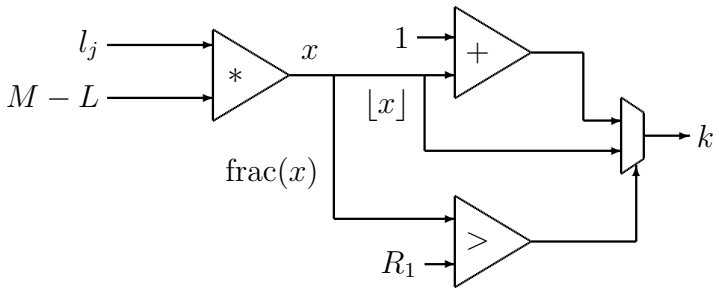

Figure 4–7: Circuit for the part of the *Add* routine responsible for computing the number of symbols to add. The result of the product x is $w + \log M$ bits long, with frac(x) corresponding to the w least significant bits and the rest corresponding to $\lfloor x \rfloor$.

a 18×18-bit multiplier, more than enough to instantiate one for each of the 384 edges. On ASIC it is possible to use truncated multipliers [44, 45] to minimize the area.

4.2.6 Hardware Implementation of the *Sample* Routine

Mathematically the *Sample* routine is trivial, it randomly selects one of the L symbols in S. This translates to the requirement to find a uniformly distributed random integer in the interval $[0, L-1]$. In the case when L is a power of 2, the problem reduces to generating $\log L$ random bits. In our case, however, L is variable and not necessarily a power of 2.

One way to generate random integers in the $[0, L-1]$ range is to first generate a random integer $X \in [0, 2^p - 1]$ where $L \leq 2^p$ and then compute the remainder of X/L. Unfortunately the inclusion of a circuit implementation of an integer divider is not practical.

This implementation uses an adapted version of the acceptance-rejection sampling method [46]. It stipulates that in order to generate a random sample from

an arbitrary probability density function $f(x)$ one can first sample an envelope distribution with the density function $g(x)$ that is easy to sample, and for which the ratio $f(x)/g(x)$ is bounded by $c > 0$ and preferably close to 1. Let x be a sample from $g(x)$ and let u be a sample from $U(0, 1)$ (the uniform distribution), we can now determine if x is acceptable as a sample of $f(x)$ with the following test:

$$u < \frac{f(x)}{cg(x)} \tag{4.1}$$

If the condition holds, the sample x is accepted, otherwise it is rejected and a new attempt has to be made.

Since the *Sample* routine has to generate a sample at every invocation, this implementation uses a series of fallback values in case of rejection as shown in Figure 4–8. The notation is as follows: R is a random variable, L is the current size of the multiset S, $R(3:7)$ represents the sequence of bits $b_3, \ldots, b_7$ from R where b_1 is the most significant bit, and $L(1:2)R(3:7)$ stands for the concatenation of the specified sequences of bits.

Three $\log M$-bit comparators are used in parallel to implement three acceptance-rejection tests. If the sample used in the first test passes the test it is routed to the output x, otherwise we fallback to the sample on the next level, and if necessary next level, and so on. If all the tests fail, the last fallback value is $L-1$ which is guaranteed to be a valid sample. One can create a longer chain of comparators for more uniform sampling, the tradeoff being the longer critical path for the computation of x.

Observe that the final distribution of values of x is not truly uniform in the mathematical sense, but rather biased towards values closer in value to L, this is due to replacing the most significant bits in R with those from L. This bias can be

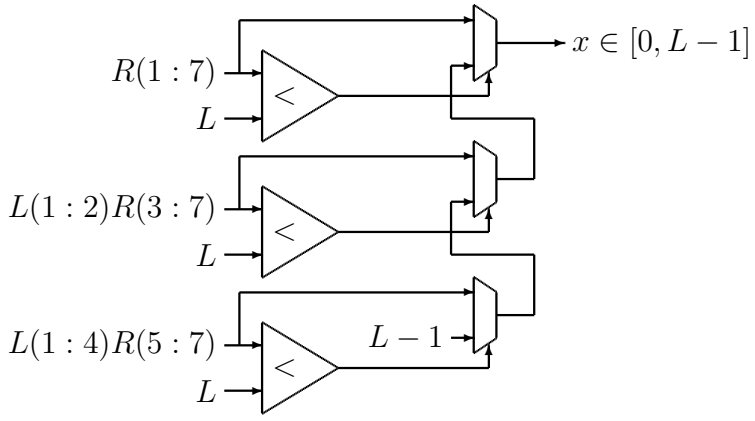

Figure 4–8: Circuit for the *Sample* routine as implemented for AMSA-128, where the output x is used as address on the edge memory in order to read a random symbol.

directed to the other end of the $[0, L-1]$ interval by replacing the most significant bits of R with zeros. Alternatively, the bias can be reduced by using a random bit and multiplexing among the two options.

Experimental results have shown that AMSA decoders converges faster to the correct codeword when the former version is used (as shown in Figure 4–8). This is due to the fact that the *Add* routine adds new incoming symbols at the end of the memory and using more recent results seems more suitable.

4.2.7 State Machine

On FPGA platforms, the limitations of the memory blocks result in a more complex control logic compared to an ASIC implementation. This is mainly due to the fact that the *Add* routine might require up to M cycles to write all the added symbols to memory if single port memory blocks are used, and $M/2$ when dual port memories are used. For this implementation, the control logic of the variable node computation has been implemented as a state machine.

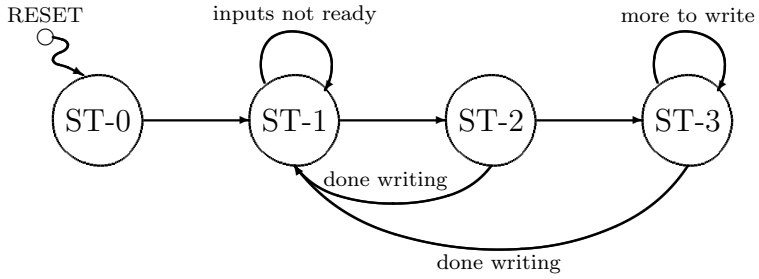

Figure 4–9: The finite state machine controlling the computation in the variable node.

Table 4–2 lists the assignment of the operations to states, which are are explained in detail in the following paragraphs.

The *Load* routine, corresponding to state ST-0, loads the q likelihood values into the likelihood memory and the M initial symbols into the edge memories (denoted as eram0 and eram1).

State ST-1 serves as a synchronization point. In fact, variable nodes will not transition to ST-2 until inputs on all their edges are ready. The readiness of the inputs for each edge is signaled via dedicated 1-bit lines. For a discussion of two different ways of message passing scheduling for AMSA see Section 4.5.

Since the edge memory blocks have two ports, the *Add* routine requires $k/2$ cycles to add k symbols. When $k \leq 2$, ST-2 transitions directly to ST-1, otherwise the ST-3 finishes writing all the symbols, and then moves to ST-1.

4.3 Check Node

The check node in stochastic decoders is considerably simpler than the SPA equivalent. As discussed in Section 2.2.3, it is a sum under GF(q), implementable directly with XOR gates.

Table 4–2: Computations corresponding to each state

State	Routine	Memory port	Action
ST-0	*Load*	cram-A	load likelihoods
		cram-B	load likelihoods
		eram0-A	load symbols for edge 0
		eram0-B	load symbols for edge 0
		eram1-A	load symbols for edge 1
		eram1-B	load symbols for edge 1
ST-1		cram-A	get likelihood for input 0
		cram-B	get likelihood for input 1
	Sample	eram0-B	sample edge memory 0
		eram1-B	sample edge memory 1
	Remove	eram0-A	remove symbol from edge 0
		eram1-A	remove symbol from edge 1
ST-2		cram-A	–
ST-3		cram-B	–
	Add	eram0-A	add symbol to edge memory 0
		eram0-B	add symbol to edge memory 0
		eram1-A	add symbol to edge memory 1
		eram1-B	add symbol to edge memory 1

In addition to the output messages v_i, the hardware implementation contains an additional output bit to signal whether the parity check is satisfied. The satisfaction is determined based on the belief symbols of the variable nodes, rather than the edge outputs.

On FPGA platforms, the circuit shown in Figure 4–10 maps directly to $O(\log q)$ lookup tables. It is thus one of the smallest elements of the Tanner graph in terms of hardware resources used.

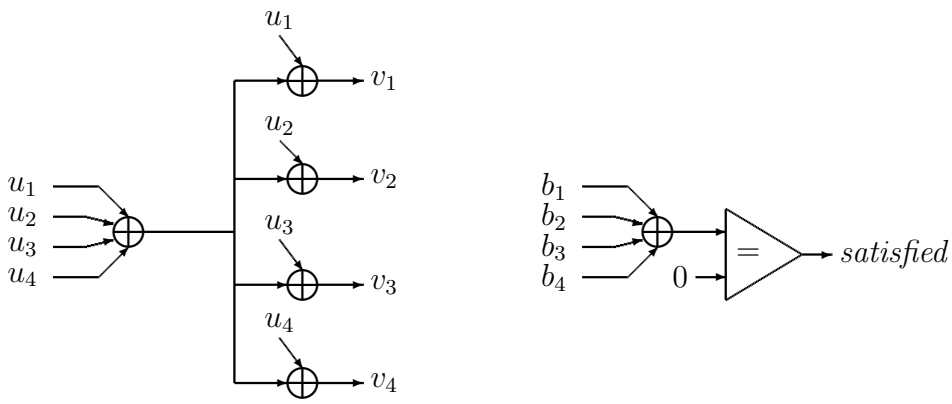

Figure 4–10: Circuit representation of the check node computation, where u_i is the input from edge i and v_i the corresponding output, b_i is the belief symbol incoming from edge i, and *satisfied* is a bit flag indicating that the parity check is satisified.

4.4 Permutation Block

On each edge of the Tanner graph (see Figure 4–11) there are three GF(q) multiplication-by-constant operations performed: one for the message from the variable node to the check node, another one for the reverse direction, and one for the belief message from the variable node to the check node. Even though the number of such blocks is large, as shown in Table 4–1, the hardware complexity of each of them is small.

A GF(q) multiplication by a constant is efficiently implemented by a $q \times \log q$-bit LUT, assuming q is a power of 2. Modern FPGA platforms provide 6-input LUT resources which can be used directly for $q \leq 64$ or combined for $q > 64$. A GF(64) permutation nodes is implemented using six 6-input LUTs while the GF(256) version requires fifteen 6-input LUTs.

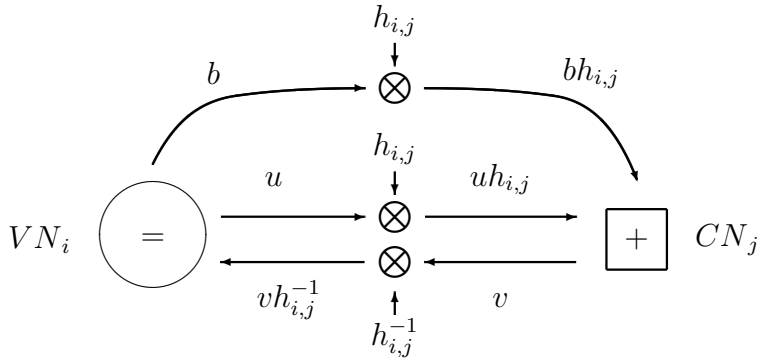

Figure 4–11: The edge of the Tanner graph connecting variable node i (VN_i) and check node j (CN_j). The $h_{i,j}$ multiplication factor is the non-zero value from H matrix associated with this edge. Message b is the variable node belief.

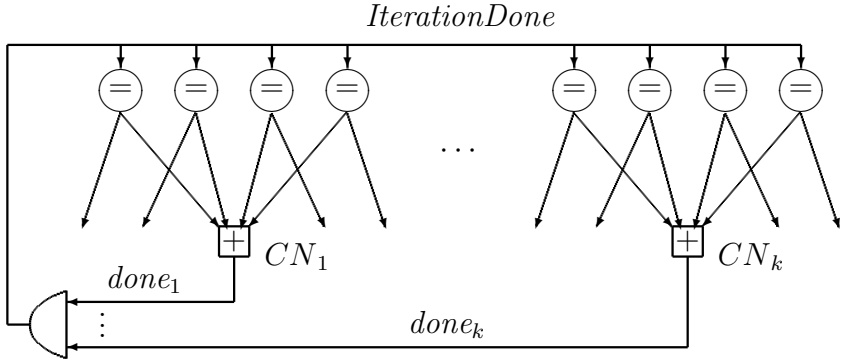

Figure 4–12: The flood scheduling method where $done_i$ is the signal that all variable nodes that send messages to check node i (CN_i) have completed their computation. The global signal *IterationDone* is used to synchronize all variable nodes.

4.5 Message Passing Scheduling

The discussion in this section is specific to FPGA platforms only where, due to interface limitations of the memory blocks, up to $\lceil \frac{M-1}{2} \rceil$ cycles might be needed to complete the *Add* routine. On ASIC, with a custom memory architecture, all the variable nodes are implicitly synchronized since they all take constant time to complete the variable node computation.

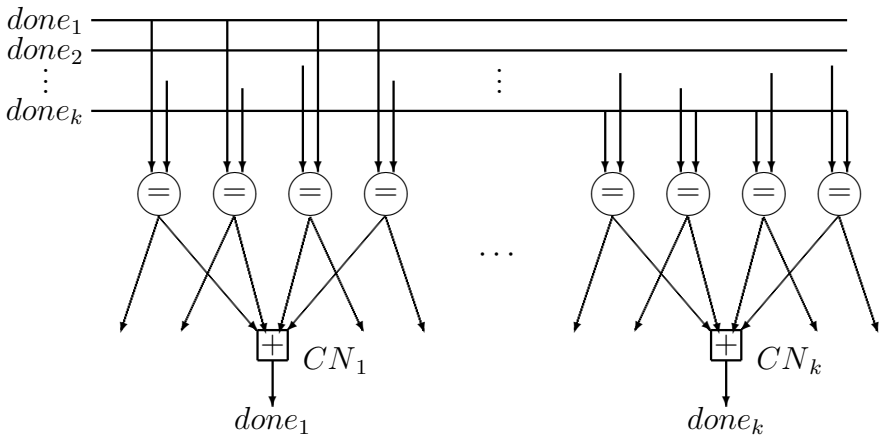

Figure 4–13: The layered decoding method where $done_i$ is the signal that all variable nodes that send messages to check node i (CN_i) have completed their computation. Each variable node uses the appropriate $done_i$ signals to independently decide if it is ready to proceed with computations.

The first, perhaps most natural, way of scheduling is to wait for all the variable nodes to complete their computation, and only then commence the computation for the next iteration. The principle is illustrated in Figure 4–12, and is called flood scheduling.

A disadvantage of using this method is that the number of cycles to complete an iteration is $\max t_i$ where t_i is the number of steps it took to complete the computation for variable node i, and $i = 1, \ldots, n$. In other words, in each iteration, variable nodes will be waiting for the slowest variable node to finish processing.

The second method, called layered decoding, is to synchronize variable nodes locally around check nodes. This way if all d_c incoming edges associated with a check node have completed their computation, the outputs of the check node are flagged ready for use. As soon as a variable node has inputs ready for all its d_v edges it can start processing, i.e. it transitions from state ST-1 to ST-2 in the state machine described in Section 4.2.7. The method is illustrated in Figure 4–13.

Table 4–3: Summary of the hardware resources used by the fully-parallel GF(64) AMSA-128 and GF(256) AMSA-512 fully-parallel decoders on Altera Stratix IV GX EP4SGX230. Note that these results are after place and route.

Resource	Total available on FPGA	GF(64) AMSA-128 (fully-parallel)	GF(256) AMSA-512 (fully-parallel)
Adaptive Look-up Tables	182,400	66,885 (37%)	91,376 (50%)
Adaptive Logic Modules	91,200	0	0
Registers	182,400	23,150 (13%)	30,840 (17%)
Simple multipliers (12x12)	1,288	384 (30%)	384 (30%)
M9K memory blocks	1,235	576 (47%)	576 (47%)
Total block memory bits	14,625,792	453,120 (3%)	2,162,688 (15%)

Layered decoding allows the decoding process to start earlier in the sense that there is no need to wait for all variable nodes to finish loading their data during initialization, some can start processing as soon as their $done_i$ signals are ready.

Note that in both Figures 4–12 and 4–13 signals $done_i$ are outputs of the check nodes in addition to the ones discussed in Section 4.3.

4.6 Synthesis Results

The FPGA chip used in this work is Altera Stratix IV GX EP4SGX230-KF40C2. Two configurations of the fully-parallel AMSA decoder were realized on the FPGA: GF(64) AMSA-128 and GF(256) AMSA-512. The synthesis results for both designs are given in Table 4–3. Figure 4–14 shows the floor plan of the FPGA chip with the GF(256) AMSA-512 decoder on it.

An important thing to notice is that the synthesis results confirm the complexity analysis done in Section 3.4 and summarized in Table 3–3. The total amount of

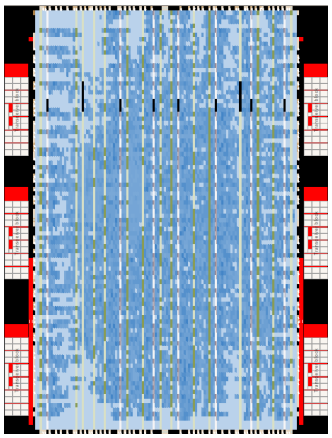

Figure 4–14: The FPGA chip floor plan after the synthesis, and place and route of the GF(256) AMSA-512 fully-parallel decoder. Different colors indicate different type of hardware resources used, while darker shades indicate higher percentage of usage of the logic elements in a particular block.

memory used scaled, as predicted, with $O(d_v M \log q + qw)$:

$$\frac{2 \times 512 \log 256 + 256 \times 12}{2 \times 128 \log 64 + 64 \times 12} \approx \frac{15\% \text{ total memory bits}}{3\% \text{ total memory bits}}$$

Similarly, the size of the variable node control logic, which is built using ALUTs and registers, scales, as predicted, with $O(\log q)$:

$$\frac{\log 256}{\log 64} \approx \frac{50\% \text{ ALUTs}}{37\% \text{ ALUTs}} \approx \frac{17\% \text{ registers}}{13\% \text{ registers}}$$

Note that the number of memory blocks did not increase because in both cases the number of edge memories is the same, and each edge memory fits in a M9K block.

Furthermore, it is possible to reliably estimate the size of any configuration of the AMSA decoder. Assuming that a GF(512) code is available with the same structure, i.e. GF(512) (192,96) and (2,4)-regular code, let us estimate what would be the size of a GF(512) AMSA-1024 decoder based on such a code. The amount of ALUTs, registers, and block memory bits can be calculated based on the synthesis

results of GF(256) AMSA-512:

$$\text{ALUTs usage} = \frac{\log 512}{\log 256} \times 50\% \approx 56\%$$
$$\text{registers usage} = \frac{\log 512}{\log 256} \times 17\% \approx 19\%$$
$$\text{block memory bits usage} = \frac{2 \times 1024 \log 512 + 1024 \times 12}{2 \times 512 \log 256 + 256 \times 12} \times 15\% \approx 48\%$$

Note that the number of multipliers is the same for all decoders because the number of edges in the Tanner graph does not change. In this case the number of memory block used will change because the size of an edge memory exceeds the size of a M9K block. The results indicate that a fully-parallel implementation of a GF(512) AMSA-1024 decoder would probably fit on the FPGA platform used in this work.

4.7 ASIC-specific Considerations

4.7.1 Single Clock-Cycle Update Memory Design

We have seen that the *Remove* routine (Algorithm 2) is a memory write operation. The same can be said about *Add* (Algorithm 1). This section will present details of how the two can be combined and executed in a single write operation.

Given that both routines are non-deterministic, there are four possible scenarios to look at (see Table 4–4 and Figure 4–15).

One important observation to make about these scenarios is that when values are being written to the memory, all the values are the same. This means that by designing an SRAM memory with a custom address decoder that enables multiple SRAM cells at the same time the write operation can be done in one cycle.

Table 4–4: Possible scenarios based on decisions made in *Add* and *Remove* routines

Scenario	*Add*	*Remove*	Changes to edge memory
A	adds α k times	remove symbol	α written in k locations
B	adds α k times	does not remove symbol	α written in k locations
C	nothing added	does not remove symbol	no change
D	nothing added	remove symbol	last symbol written

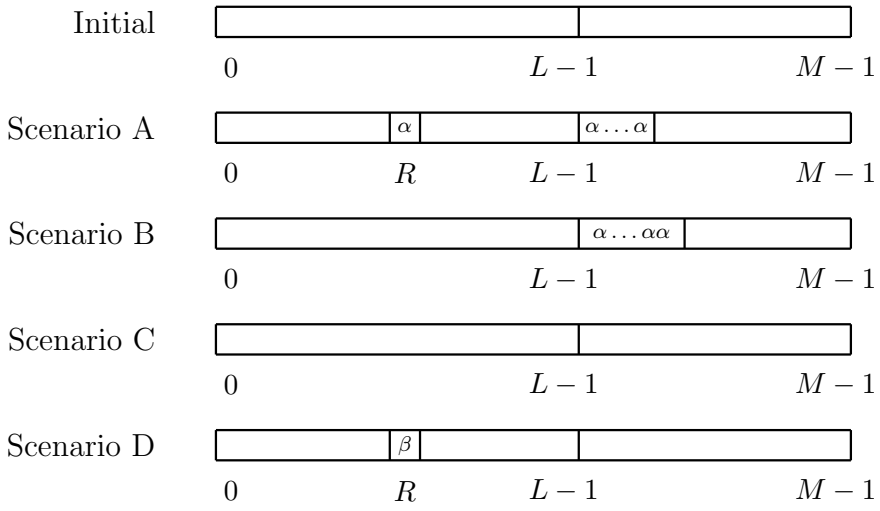

Figure 4–15: Schematic representation of the scenarios in Table 4–4 and the modifications they make to the memory.

Figure 4–16 provides an architecture for such an SRAM memory. The Address Decoder used in the circuit is a standard address decoder. The Mask Overlay unit sets high all the address lines in the segment $[L, M-1]$. Thus, having selected multiple SRAM cells, the data will be written to multiple locations of the memory in one cycle. Even though more than the necessary k symbols are written (see Algorithm 1 line 8) in the $[L, M-1]$ segment of the memory, only k will be in S, the rest being outside and, thus, not having any effect.

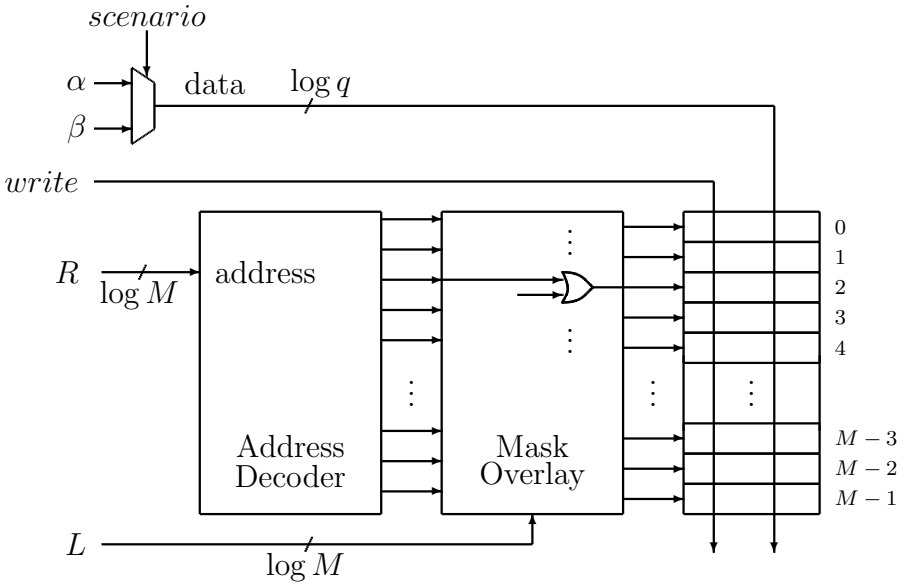

Figure 4–16: Architecture of SRAM that can perform any of the scenarios in Table 4–4 in one cycle.

We can consider the Address Decoder and Mask Overlay pair as a Custom Address Decoder for the SRAM memory block. In order to estimate how much bigger is the Custom Address Decoder compared to the standard Address Decoder, both have been implemented in VHDL and compared in terms of logical resources used. In the case of $M = 128$ the size increased by 43% while for $M = 256$ the size increased by 38%. Considering that the decoder represents only a part of the total area of an SRAM block (the rest being occupied by the SRAM cells, sense amplifiers, etc.), the overall area increase for an SRAM memory block is further reduced. It is possible to create a controller that will enable exactly k lines by adding an additional overlay that sets low all the lines in the segment $[L + k, M]$, the result being that only the lines in the segment $[L, L + k - 1]$ will be selected for writing.

As it was shown in Section 5.2, this SRAM memory design reduces the decoding run-time to $O(1)$ and increases the throughput by two orders of magnitude compared to the results available in literature for the same code.

Chapter 5

Simulation Results and Analysis

This chapter presents and analyzes the simulation results for different configurations of the AMSA decoder. Additionally, in Section 5.3 an optimized quantization scheme is proposed, and in Section 5.4 the method of accelerated convergence is introduced, which improves throughput without sacrificing performance.

5.1 Performance

The performance of several configurations of the AMSA algorithm are given in Figure 5–1 for the Additive white Gaussian noise (AWGN) channel. The codes used here are the GF(64) (192,96) (2,4)-regular and GF(256) (192,96) (2,4)-regular. Both codes are from the DAVINCI project.

For each code, several different configurations of the AMSA algorithm are used. For the GF(64) code the AMSA-256 and AMSA-512 configurations are used. For the GF(256) code the AMSA-256, AMSA-512, and AMSA-1024 configurations are used. As it can be seen from Figure 5–1, the decoder configurations with larger values of M have better performance. This is explained by the fact that when M is larger the

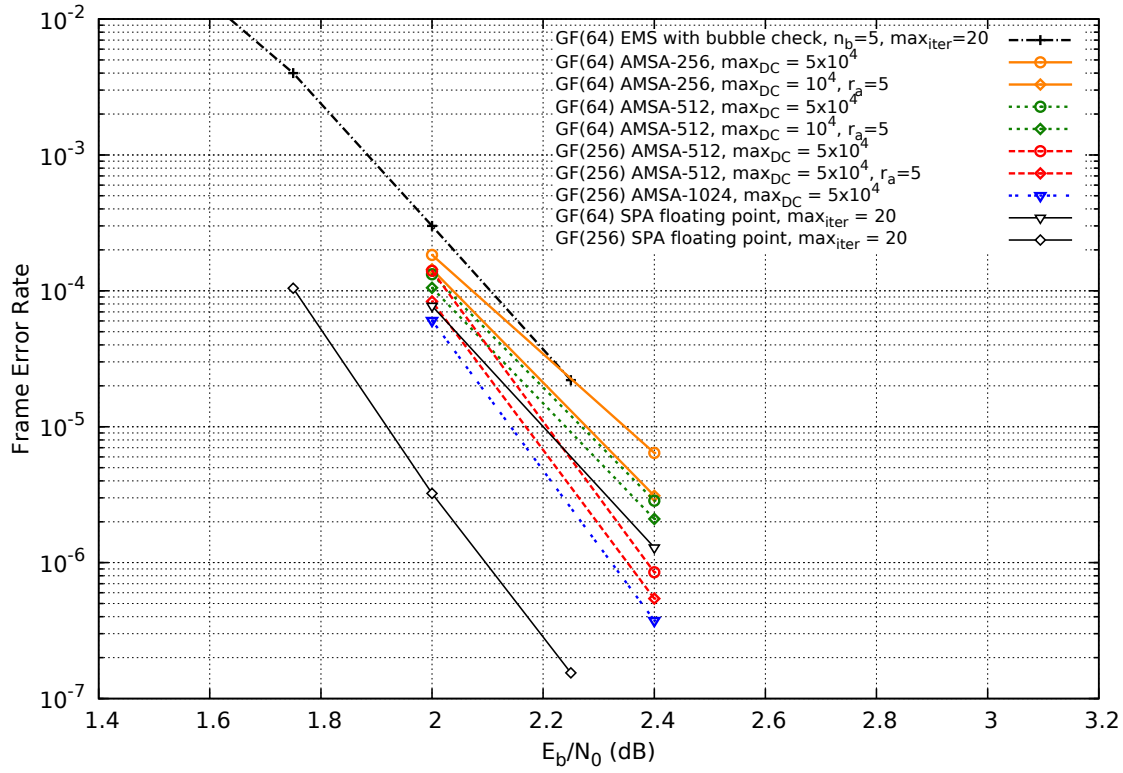

Figure 5–1: Frame error rate performance of the AMSA algorithm compared to the Bubble Check algorithm from [18]. The two codes used here are (192,96) (2,4)-regular over GF(64) and GF(256). The channel is AWGN. Parameter r_a stands for the number of redecoding attempts. For all AMSA configurations, likelihoods are represented in $w = 12$ bits.

multiset representation is more precise, as it was shown in Section 3.1. Parameter M is a tradeoff between memory size and BER performance.

Another parameter that affects performance is redecoding. In Figure 5–1, the performance is compared for the same configuration with and without redecoding but keeping the total number of decoding cycles equal. For example, for the GF(64) AMSA-256 decoder the frame error rate at $E_b/N_0 = 2.4$dB is 6×10^{-6} with 5×10^4

maximum allowed decoding cycles, while if doing 5 redecoding attempts of 10^4 maximum decoding cycles each, the performance is improved to 3×10^{-6}. Additionally, it can be observed that for the same AMSA-256 configuration, redecoding reduces the error floor. As it can be seen, for the same total number of decoding cycles it is possible to improve the performance and reduce error floor by using redecoding.

For the GF(64) code, AMSA-512 is only 0.04 dB away from the floating-point SPA performance at FER of 2×10^{-6}. On the other hand, when using the GF(256) version of code, the difference between the SPA results and AMSA-1024 are of about 0.22 dB at FER of 4×10^{-6}. Note that the SPA algorithm used here uses a floating-point representation, while the AMSA decoders use quantized values represented in $w = 12$ bits.

In comparing the performance of a fully-parallel AMSA decoder to SPA it must be taken into account that a fully-parallel SPA decoder for the codes used in this work is, at the moment, impractical.

Note that AMSA matches and improves on the performance of the EMS algorithm implementation in [18] while also considerably improving the throughput, as shown in the following section.

5.2 Throughput and Latency

As the SNR increases the AMSA decoder takes fewer decoding cycles to complete decoding. This implies that the throughput is a function of the SNR. Indeed, by looking at Figure 5–2 it can be seen that throughput is a linear function of SNR for all configurations of AMSA decoders. The throughput shown here is the coded throughput of the decoder.

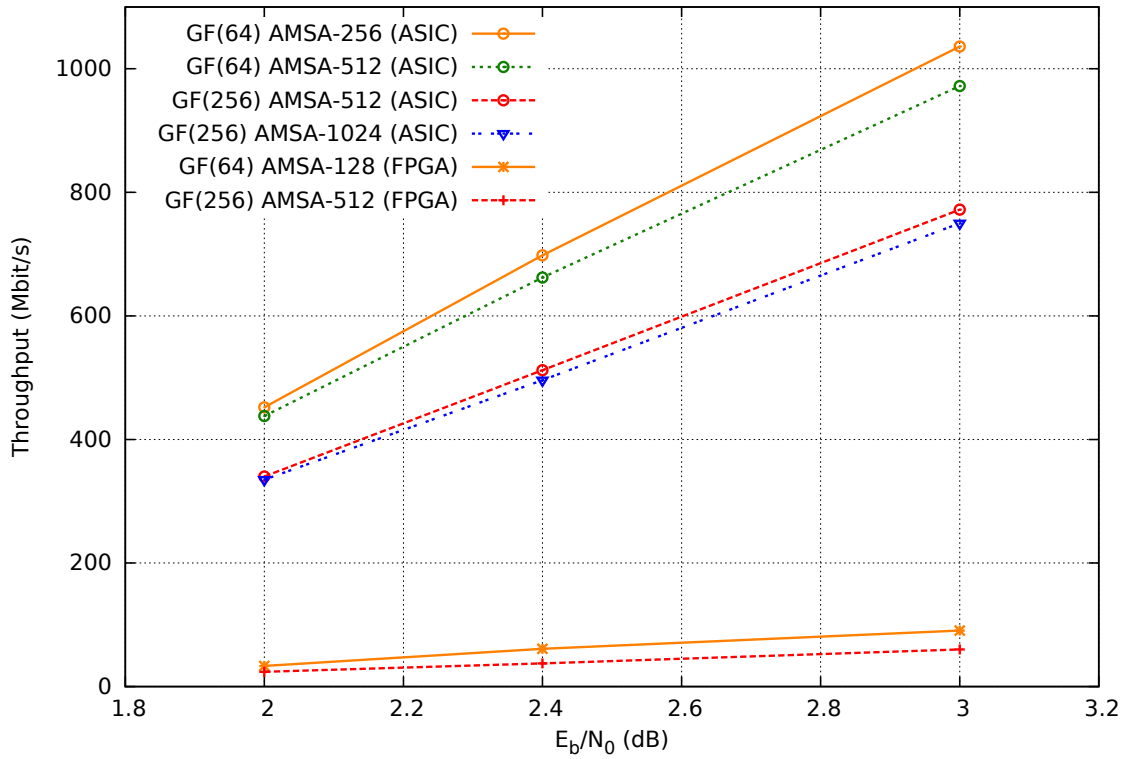

Figure 5–2: Throughput of the FPGA implementation and estimated throughput for the ASIC implementation of the AMSA algorithm based on the GF(64) and GF(256) version of the (192,96) (2,4)-regular DAVINCI code, at clock frequency $f = 108$ MHz.

In literature, the highest throughput for a hardware implementation for the GF(64) (192,96) code used here is of 3.8 Mbit/s [18] using EMS with the Bubble Check algorithm. AMSA presents an improvement in throughput of an order of magnitude for an FPGA implementation and of more then two orders of magnitude for ASIC implementations due to the special SRAM architecture.

As a general rule, stochastic decoders have a higher latency compared to other decoding approaches. In this case the limit on number of decoding cycles max_{DC} is set to 5×10^4 even though the average number of decoding cycles at E_b/N_0 of 2.4 dB

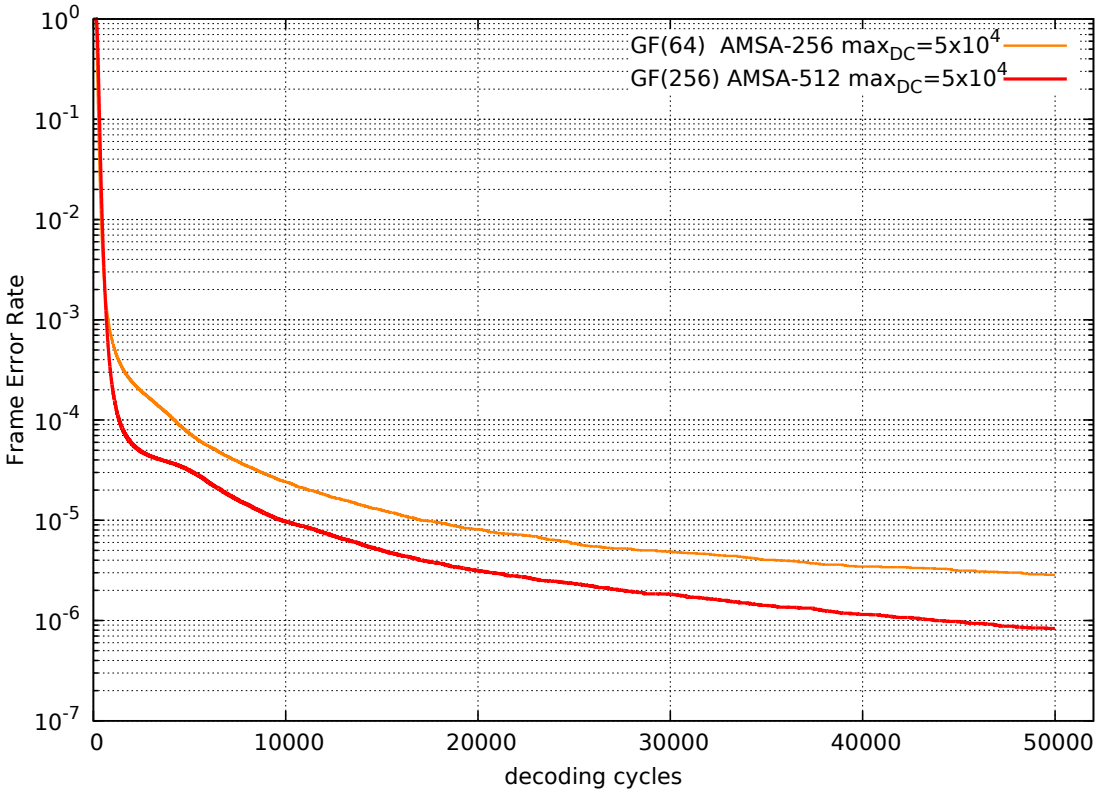

Figure 5–3: Settling curves for the GF(64) and GF(256) versions of the (192,96) code.

is approximately 180. Looking at the settling curves from Figure 5–3 it can be seen why a large max_{DC} is required, because there is a non-negligible number of frames decoded after a large number of iterations. For the $O(1)$ run-time complexity ASIC architecture a decoding cycle corresponds to a clock cycle, meaning that for a clock frequency of $f = 108$MHz, the latency introduced by a decoder with $max_{DC} = 5 \times 10^4$ is 0.5 ms.

In the case of SPA, the average number of iterations at E_b/N_0 of 2.4 dB is approximately 4 for both the GF(64) and GF(256) version of the code. In this case

latency cannot be directly estimated without knowing how many clock cycles an SPA iteration takes for the particular implementation.

5.3 Efficient Quantization

The algorithm presented in this thesis performs its computations using integer quantities with the exception of the *Add* routine where channel likelihoods are used (see Algorithm 1). As discussed in Chapter 4, the likelihoods are represented in fixed-point format using w bits. This parameter is, in fact, a tradeoff mechanism between hardware complexity and the BER performance of the decoder. This section presents a quantization strategy that allows to reduce the value of w without performance loss.

Let p be a probability with a value between 0 and 1 and let $p^{(w)}$ be the value of p represented in w bits in fixed-point format, then $p^{(w)} = \sum_{i=1}^{w} 2^{-i} b_i$ where b_i is the ith most significant bit of the fixed-point representation of p. In the context of LDPC decoding, the goal is to find the smallest w that results in an acceptable BER performance. The deterioration in performance with decreasing w can be explained by the fact that all probabilities less than 2^{-w} are represented as zero (see the results for the cases not using the efficient quantization in Figure 5–4). In AMSA, a symbol α with likelihood $p_\alpha = 0$ is never added to the multiset S because the product $l_j \cdot (M - |S|)$ from Algorithm 1 is zero, which results in the symbol never being used as output of the variable node. This is also true in SPA, because if a message $V_{pv}[\alpha] = 0$ for the corresponding symbol α, then the result of the product in Equation (2.4) will also be zero for the the symbol, i.e. $U_{tp}[\alpha] = 0$.

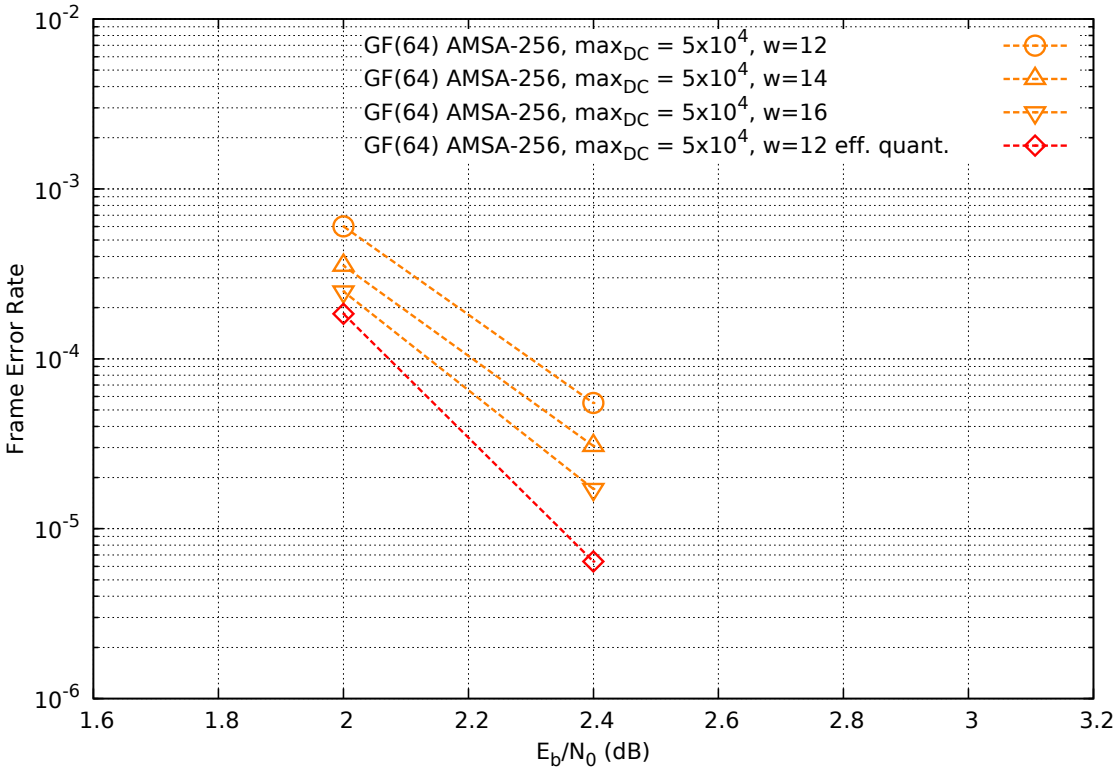

Figure 5–4: The impact of using the efficient quantization method on the performance. The decoder used is AMSA-256, the code is a GF(64) (192,96) (2,4)-regular code, and the channel is AWGN. The maximum number of decoding cycles is 5×10^4.

The alternative approach used in this work is to represent probabilities with w bits, but avoid zero likelihoods. The w-bit representation of p is computed as follows:

$$p^{(w)} = \begin{cases} \sum_{i=1}^{w} 2^{-i} b_i & \text{if } p \geq 2^{-w} \\ 2^{-w} & \text{if } p < 2^{-w} \end{cases}$$

As it can be seen in Figure 5–4, for the same w the performance is considerably improved by using this optimization. Note that throughput is also slightly improved in the lower SNR region (see Figure 5–5).

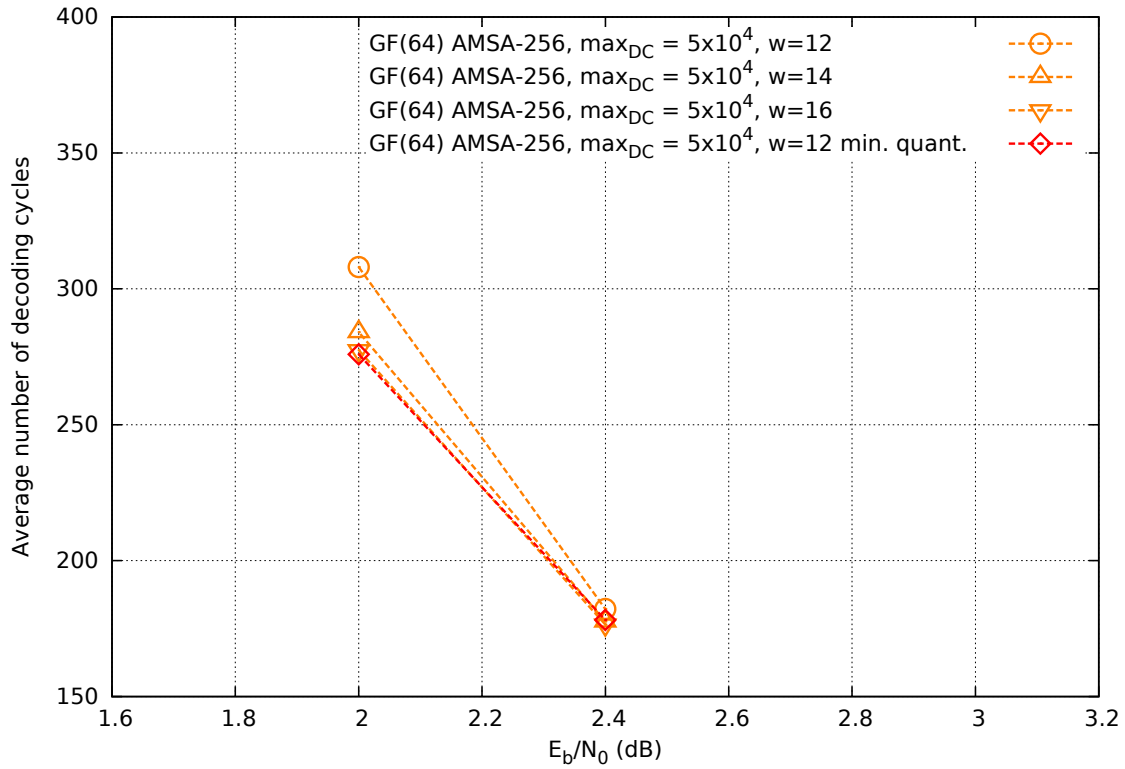

Figure 5–5: The impact of using the efficient quantization method on the average number of decoding cycles. The decoder used is AMSA-256, the code is a GF(64) (192,96) (2,4)-regular code, and the channel is AWGN. The maximum number of decoding cycles is 5×10^4.

5.4 Accelerated Convergence *Add* Routine

As discussed in Section 2.2.2, a constant distribution of probabilities can be represented by a non-binary stream or any of its permutations. In stochastic LDPC decoding edge memories and the associated distributions of probabilities are updated at every decoding cycle, which implies a continuous change in the statistics of the stochastic stream. In this case, not all permutations of the symbols in a stochastic stream represent the same distribution. For example, let us consider two streams $s_1 = \alpha\alpha\beta\alpha\beta\alpha\delta\alpha\beta\alpha\gamma\alpha\ldots$ and $s_2 = \beta\gamma\beta\delta\beta\alpha\alpha\alpha\alpha\alpha\alpha\ldots$ that are permutations of

Algorithm 5: The accelerated *Add* routine of AMSA where M is the upper limit imposed on the cardinality of multiset S, α is the current incoming symbol with likelihood l_α, and β is the previous symbol. The coefficient c_A is used in association with the accumulator A.

Input: S, symbols α and β, l_α, accumulator A
Output: S with possibly added one or more instances of symbol α, A

1 **if** $\alpha = \beta$ **then**
2 $\quad A \leftarrow \min(A + l_\alpha, 1)$
3 **else**
4 $\quad A \leftarrow 0$
5 **end**
6 $x \leftarrow (l_\alpha + c_A A) \cdot (M - |S|)$
7 **if** $\text{frac}(x) \geq R_1$ **then**
8 $\quad k \leftarrow \lfloor x \rfloor + 1$
9 **else**
10 $\quad k \leftarrow \lfloor x \rfloor$
11 **end**
12 $S \leftarrow S \cup \{$ k instances of $\alpha\}$
13 $\beta \leftarrow \alpha$

each other and let $s_i[t] \in GF(q)$ be the symbol in stream s_i at time $t \geq 0$. Even though the number of times symbol α appears in both streams is equal, the fact that in s_2 the most recent symbols $s_i[t-k]$ where $k < t$. are all α, indicates that, possibly, the associated distribution has converged to a configuration that maximizes the probability of symbol α. In contrast, s_1 did not stabilize to a certain value.

In order to take advantage of this observation, an improved version of the *Add* routine, that recognizes such patterns and accelerates the convergence of the distribution associated with the edge, is proposed in Algorithm 5. The value of A increases up to the maximum value of 1 as long as the sequence of identical symbols continues.

The term $l_\alpha + c_A A$ on line 6 in Algorithm 5 is similar to a proportional-integral (PI) term used in PI control where the output of the manipulated variable (MV) is

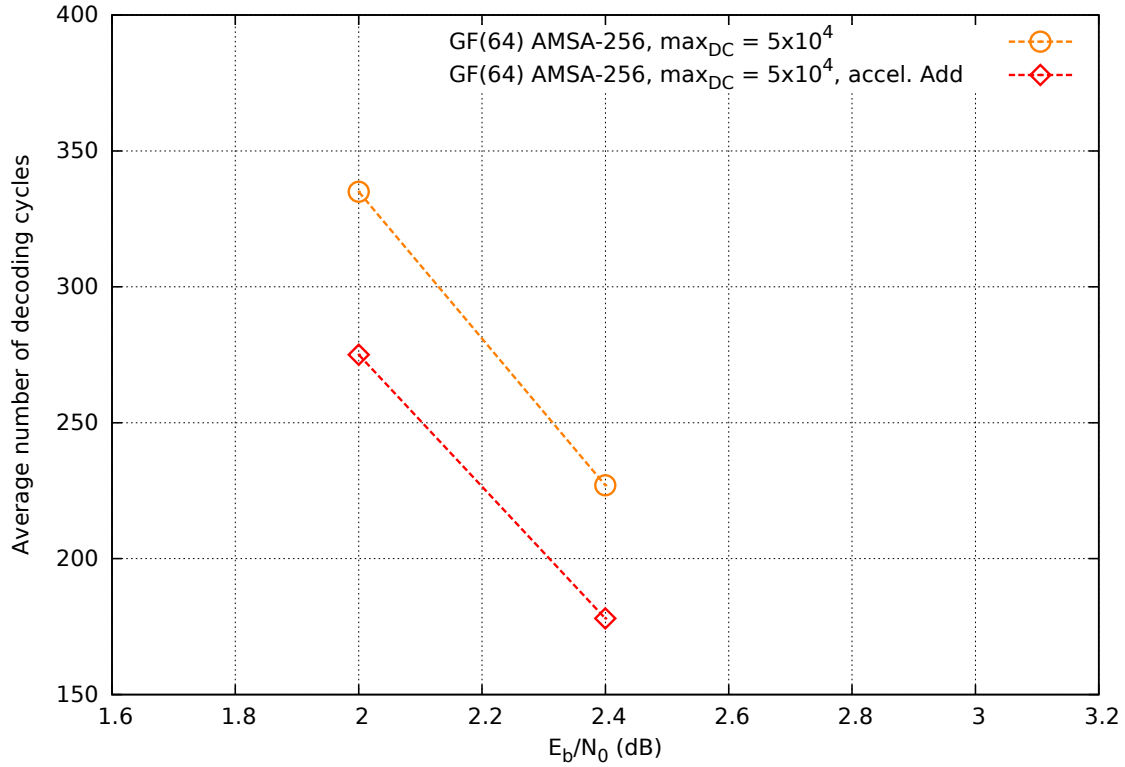

Figure 5–6: The impact of using the accelerated convergence *Add* routine on the average number of decoding cycles. The coefficient c_A is equal to 1/4. The decoder used is AMSA-256, the code is a GF(64) (192,96) (2,4)-regular code, and the channel is AWGN. The maximum number of decoding cycles is 5×10^4. All probabilities are represented with $w = 12$ bits.

given by $\mathrm{MV}(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau$ with $e(t)$ being the error, and K_i and K_p being the proportional and integral coefficients, respectively. The difference is in the way the integral component is calculated $A(t) = \int_{t-k}^{t} l_\alpha(\tau) d\tau$ where k is the length of the current run of identical symbols in the stream. Note that it is possible to implement a proportional-integral strategy by calculating $A_i(t) = \int_0^t l_\alpha(\tau) d\tau$ where A_i is the integral term associated to symbol i in GF(q), but it is less practical because it requires $O(q)$ memory.

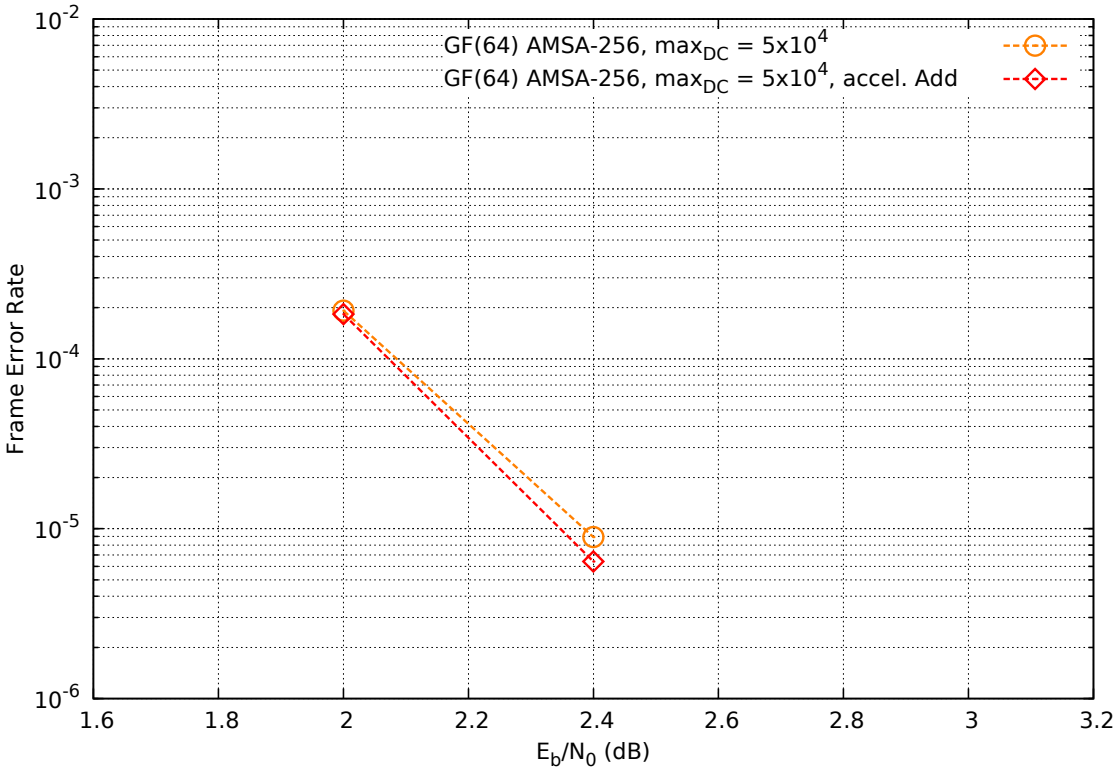

Figure 5–7: The impact of using the accelerated convergence *Add* routine on the performance. The coefficient c_A is equal to 1/4. The decoder used is AMSA-256, the code is a GF(64) (192,96) (2,4)-regular code, and the channel is AWGN. The maximum number of decoding cycles is 5×10^4. All probabilities are represented with $w = 12$ bits.

The accelerated convergence method reduces the average number of decoding cycles and also improves BER performance. As it can be seen in Figure 5–6, the average number of decoding cycles at 2.4 dB is reduced by approximately 27%, which translates into an improvement in throughput by the same factor. Figure 5–7 shows the impact of the method on the performance. Various values have been tried for the coefficient c_A, and the value that maximizes the benefits in terms of performance and throughput is 1/4.

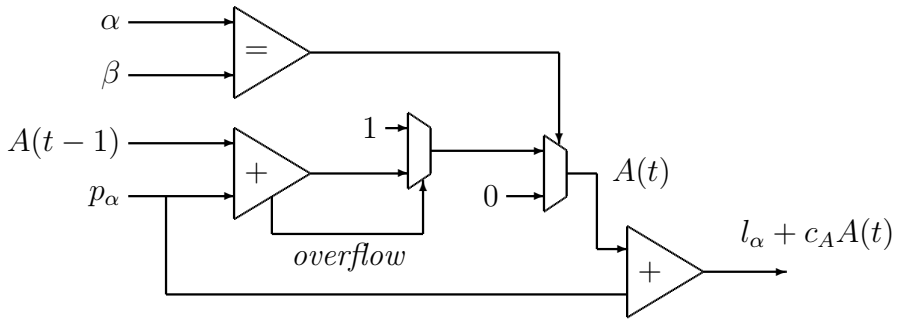

Figure 5–8: Circuit for computing the term $l_\alpha + c_A A(t)$ in the accelerated convergence *Add* routine. Note that the rest of the circuit presented in Figure 4–7 is unchanged.

The additional logic required for the accelerated *Add* routine is given in Figure 5–8 while the rest of the circuit is unchanged (see Figure 4–7). The value of c_A is chosen to be a negative power of two in order to eliminate the need for a multiplier. Note that in order to compute $A(t)$ the previous value $A(t-1)$ has to be stored in a memory.

Chapter 6

Conclusion and Future Work

6.1 Advances

Based on a multiset representation for probability mass functions, the Adaptive Multiset Stochastic Algorithm was introduced and applied for the non-binary stochastic decoding of LDPC codes with $d_v = 2$. Additionally, the concept of redecoding was applied to non-binary LDPC decoding and shown to improve performance and lower the error floors.

The AMS algorithm was used for the FPGA implementation of two fully-parallel LDPC decoders over GF(64) and GF(256). To the best of our knowledge, these are the first fully-parallel LDPC decoders over GF(64) and GF(256) reported in literature.

The FPGA decoders achieve a clock frequency of 108 MHz and a throughput of about 95 Mbit/s at E_b/N_0 of 3.0 dB. For the GF(256) decoder, the frame error rate (FER) performance at E_b/N_0 of 2.4 dB is 3.5×10^{-7} with redecoding, and 8.5×10^{-7} without redecoding. These implementations are also the highest throughput decoders reported for the particular codes used.

An SRAM architecutre for ASIC was proposed that reduces the run-time complexity of an AMSA decoding cycle to $O(1)$. The estimated throughput for a fully-parallel ASIC decoder using this SRAM architecture is of 512 Mbit/s at 2.4 dB for the GF(256) version, and 698 Mbit/sec at 2.4 dB for the GF(64) case.

6.2 Future Work

Stochastic decoders, in general, have a higher decoding latency compared to other decoding methods. The accelerated convergence method described in Chapter 5 helped reduce the average number of required decoding cycles by 27% without sacrificing the BER performance. Further investigation in this direction can lead to more improvements.

The results of this work show than the GF(256) version of the decoder has a better performance that the GF(64) version without being much more complex in terms of hardware resources used. It is, therefore, interesting to investigate how to efficiently apply AMSA to LDPC codes of higher order like GF(512) or GF(1024).

As shown in Section 5.4, stochastic streams can provide, in addition to the statistics based on frequencies of symbols, information in the form of patterns that can be recognized and used to improve the performance of the decoder.

An interesting research direction is to enable the stochastic LDPC decoders to adapt to the changes in the E_b/N_0 ratio of the channel by adjusting parameters like the maximum number of decoding cycles, and the number of redecoding attempts.

This work used the AWGN channel for evaluating the performance of the proposed algorithm. An assessment of the performance of AMSA on other channel models would be valuable.

REFERENCES

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

[2] D. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inf. Theory*, vol. 45, no. 2, pp. 399–431, 1999.

[3] S.-Y. Chung, J. Forney, G. D., T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 db of the Shannon limit," *IEEE Commun. Lett.*, vol. 5, no. 2, pp. 58–60, 2001.

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

[5] A. Morello and V. Mignone, "DVB-S2: The second generation standard for satellite broad-band services," *Proc. IEEE*, vol. 94, no. 1, pp. 210–227, 2006.

[6] V. Oksman and S. Galli, "G.hn: The new ITU-T home networking standard," *IEEE Commun. Mag.*, vol. 47, no. 10, pp. 138–145, 2009.

[7] *IEEE Standard for Information technology-Specific requirements - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Std. IEEE 802.3an.

[8] *IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput*, IEEE Std. IEEE 802.11an-2009.

[9] *IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Broadband Wireless Access Systems*, IEEE Std. IEEE 802.16e.

[10] J. Lu and J. M. F. Moura, "Structured LDPC codes for high-density recording: large girth and low error floor," *IEEE Trans. Magn.*, vol. 42, no. 2, pp. 208–213, 2006.

[11] H. Song and J. R. Cruz, "Reduced-complexity decoding of q-ary LDPC codes for magnetic recording," *IEEE Trans. Magn.*, vol. 39, no. 2, pp. 1081–1087, 2003.

[12] B. Zhou, J. Kang, S. Song, S. Lin, K. Abdel-Ghaffar, and M. Xu, "Construction of non-binary quasi-cyclic LDPC codes by arrays and array dispersions," *IEEE Trans. Commun.*, vol. 57, no. 6, pp. 1652–1662, 2009.

[13] I. Gutierrez, G. Bacci, J. Bas, A. Bourdoux, H. Gierszal, A. Mourad, and S. Pleftschinger, "DAVINCI non-binary LDPC codes: Performance and complexity assessment," in *Proc. Future Network and Mobile Summit*, 2010, pp. 1–8.

[14] A. Mourad and I. Gutierrez, "System level evaluation of DAVINCI non-binary LDPC codes," in *Proc. Future Network and Mobile Summit*, 2010, pp. 1–9.

[15] D. Declercq and M. Fossorier, "Decoding algorithms for nonbinary LDPC codes over GF(q)," *IEEE Trans. Commun.*, vol. 55, no. 4, pp. 633–643, 2007.

[16] X.-Y. Hu, E. Eleftheriou, D.-M. Arnold, and A. Dholakia, "Efficient implementations of the sum-product algorithm for decoding LDPC codes," in *Proc. IEEE Global Telecommunications Conf. GLOBECOM '01*, vol. 2, 2001, p. 1036.

[17] A. Voicila, D. Declercq, F. Verdier, M. Fossorier, and P. Urard, "Low-complexity decoding for non-binary LDPC codes in high order fields," *IEEE Trans. Commun.*, vol. 58, no. 5, pp. 1365–1375, 2010.

[18] E. Boutillon and L. Conde-Canencia, "Simplified check node processing in nonbinary LDPC decoders," in *Proc. 6th Int Turbo Codes and Iterative Information Processing (ISTC) Symp*, 2010, pp. 201–205.

[19] V. C. Gaudet and A. C. Rapley, "Iterative decoding using stochastic computation," *Electronics Letters*, vol. 39, no. 3, pp. 299–301, 2003.

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

[21] C. Winstead, V. C. Gaudet, A. Rapley, and C. Schlegel, "Stochastic iterative decoders," in *Proc. Int. Symp. Information Theory ISIT 2005*, 2005, pp. 1116–1120.

[22] F. Leduc-Primeau, S. Hemati, W. J. Gross, and S. Mannor, "A relaxed half-stochastic iterative decoder for LDPC codes," in *Proc. IEEE Global Telecommunications Conf. GLOBECOM 2009*, 2009, pp. 1–6.

[23] G. Sarkis, S. Hemati, S. Mannor, and W. J. Gross, "Relaxed half-stochastic decoding of LDPC codes over GF(q)," in *Proc. 48th Annual Allerton Conf. Communication, Control, and Computing (Allerton)*, 2010, pp. 36–41.

[24] G. Sarkis, S. Mannor, and W. J. Gross, "Stochastic decoding of LDPC codes over GF(q)," in *Proc. IEEE Int. Conf. Communications ICC '09*, 2009, pp. 1–5.

[25] T. Richardson, "Error floors of LDPC codes," *Proc. 41st Allerton Conf. on Communications, Control, and Computing*, vol. 1, p. 1, 2003.

[26] M. C. Davey and D. MacKay, "Low-density parity check codes over GF(q)," *IEEE Commun. Lett.*, vol. 2, no. 6, pp. 165–167, 1998.

[27] D. MacKay and M. Davey, "Evaluation of Gallager codes for short block length and high rate applications," in *Proc. IMA Workshop Codes, Syst., Graphical Models*, 1999.

[28] T. Richardson and R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 599–618, 2001.

[29] L. Barnault and D. Declercq, "Fast decoding algorithm for LDPC over $GF(2^q)$," in *Proc. IEEE Information Theory Workshop*, 2003, pp. 70–73.

[30] X. Li and M. R. Soleymani, "A proof of the Hadamard transform decoding of the belief propagation algorithm for LDPCC over gf(q)," in *Proc. VTC2004-Fall Vehicular Technology Conference 2004 IEEE 60th*, vol. 4, Sep. 26–29, 2004, pp. 2518–2519.

[31] H. Wymeersch, H. Steendam, and M. Moeneclaey, "Log-domain decoding of LDPC codes over GF(q)," in *Proc. IEEE International Conference on Communications*, H. Steendam, Ed., vol. 2, 2004, pp. 772–776 Vol.2.

[32] J. Chen, A. Dholakia, E. Eleftheriou, M. P. C. Fossorier, and X.-Y. Hu, "Reduced-complexity decoding of LDPC codes," *IEEE Trans. Commun.*, vol. 53, no. 8, pp. 1288–1299, 2005.

[33] H. Wymeersch, H. Steendam, and M. Moeneclaey, "Computational complexity and quantization effects of decoding algorithms for non-binary ldpc codes," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '04)*, H. Steendam, Ed., vol. 4, 2004, pp. iv–669–iv–672 vol.4.

[34] B. Gaines, *Advances in Information Systems Science*. Plenum, New York, 1969, ch. 2, pp. 37–172.

[35] S. Sharifi Tehrani, W. J. Gross, and S. Mannor, "Stochastic decoding of LDPC codes," *IEEE Commun. Lett.*, vol. 10, no. 10, pp. 716–718, 2006.

[36] W. J. Gross, V. C. Gaudet, and A. Milner, "Stochastic implementation of LDPC decoders," in *Proc. Conf Signals, Systems and Computers Record of the Thirty-Ninth Asilomar Conf*, 2005, pp. 713–717.

[37] S. Sharifi Tehrani, A. Naderi, G.-A. Kamendje, S. Hemati, S. Mannor, and W. J. Gross, "Majority-based tracking forecast memories for stochastic LDPC decoding," *IEEE Trans. Signal Process.*, vol. 58, no. 9, pp. 4883–4896, 2010.

[38] G. Sarkis and W. J. Gross, "Efficient stochastic decoding of non-binary LDPC codes with degree-two variable nodes," in *IEEE Communication Letters*, 2011, submitted for publication.

[39] F. Leduc-Primeau, S. Hemati, S. Mannor, and W. J. Gross, "Lowering error floors using dithered belief propagation," in *Proc. IEEE Global Telecommunications Conf. GLOBECOM 2010*, 2010, pp. 1–6.

[40] C. Spagnol, E. M. Popovici, and W. P. Marnane, "Hardware implementation of $GF(2^m)$ LDPC decoders," *IEEE Trans. Circuits Syst. I*, vol. 56, no. 12, pp. 2609–2620, 2009.

[41] Y. Sun, Y. Zhang, J. Hu, and Z. Zhang, "FPGA implementation of nonbinary quasi-cyclic LDPC decoder based on EMS algorithm," in *Proc. Int. Conf. Communications, Circuits and Systems ICCCAS 2009*, 2009, pp. 1061–1065.

[42] J. Lin, J. Sha, Z. Wang, and L. Li, "Efficient decoder design for nonbinary quasicyclic LDPC codes," *IEEE Trans. Circuits Syst. I*, vol. 57, no. 5, pp. 1071–1082, 2010.

[43] P. Alfke. (1996, July) Application note: Efficient shift registers, LFSR counters, and long pseudo-random sequence generators. Xilinx Inc. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf

[44] M. J. Schulte and J. Swartzlander, E. E., "Truncated multiplication with correction constant [for DSP]," in *Proc. [Workshop] VLSI Signal Processing, VI*, 1993, pp. 388–396.

[45] S. S. Kidambi, F. El-Guibaly, and A. Antoniou, "Area-efficient multipliers for digital signal processing applications," *IEEE Trans. Circuits Syst. II*, vol. 43, no. 2, pp. 90–95, 1996.

[46] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.