UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

VITOR KIELING

An Area Efficient FPGA Implementation for the Syndrome based Non Binary LDPC Check Node Algorithm

Trabalho de graduação

Trabalho realizado na
Technische Universität Kaiserslautern.

Orientador:
Dipl.-Ing. Philipp Schläfer

Co-orientador:
Prof. Dr. Valter Roesler

Porto Alegre
2015

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do ECP: Prof. Raul Fernando Weber
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

1 Resumo

Com os novos limites de taxa de transmissão impostos pelas aplicações dos dias de hoje, arquiteturas de hardware dedicadas se fazem necessárias para obtermos uma correção de erros adequada a estes. Para tal, a evolução das soluções baseadas em códigos *Low Density Parity Check* para códigos não-binários é necessária, e esta, implica em um aumento na complexidade de decodificação. O melhor algoritmo desenvolvido até hoje que provê um bom compromisso entre complexidade de hardware e performance é o *Extended Min-Sum* usando o esquema de *Forward Backward*. Infelizmente, este apresenta baixa taxa de transferência e alta latência. Neste trabalho um novo hardware para o *Check-Node* de um decodificador LDPC não-binário baseado no algoritmo *Syndrome-Based* é apresentado. Ele mostra uma performance equivalente ao EMS-FWBW, combinado de área reduzida, baixa latência e alta vazão.

Lista de Abreviaturas e Siglas

CN	Check-Node
EMS	Extended Min-Sum
LDPC	Low Density Parity Check
LLR	Log-likelihood Ratio
NB-LDPC	Non-Binary Low Density Parity Check
SYN	Syndrome-Based
VN	Variable Node

Lista de Figuras

1 Grafo de Tanner 10
2 Comparação Binário x Não-Binário 10
3 Algoritmo de Syndrome 11
4 Exemplo dos sets D_i e Limites de Distância 12
5 Arquitetura inicial 14
6 Mudança de posição do ordenador 15
7 Esquema de sondas 16
8 Redução da saída do ordenador 16
9 Valores comuns aos cálculos de D_1 17
10 Arquitetura de Hardware Final 18
11 Hardware Overview 18
12 Communications Performance 19
13 Area comparison 20
14 Throughput Comparison 21

Lista de Tabelas

1 Parâmetros de código 13
2 Parâmetros da implementação e do algoritmo SYN 13
3 Final results 21

Sumário

1 Resumo **2**

2 Introdução **7**

3 Códigos LDPC Não-Binários **9**
3.1 Algoritmo Syndrome-Based 11

4 Arquitetura de Hardware **12**
4.1 Arquitetura Inicial 13
4.2 Otimizações 14
4.3 Arquitetura Final 17

5 Resultados **19**
5.1 Performance de Comunicação 19
5.2 Área 20
5.3 Vazão 20
5.4 Resultados Finais 21

6 Conclusão **22**
6.1 Trabalhos futuros 22

Referências Bibliográficas **23**

2 Introdução

Neste relatório é apresentado um resumo estendido em português para a Universidade Federal do Rio Grande do Sul do trabalho original em anexo. O trabalho de conclusão original, em inglês, foi realizado e apresentado na Technische Universität Kaiserslautern, em Kaiserslautern, na Alemanha. As referências encontram-se no trabalho completo em anexo.

A chave para o sucesso de sistemas de comunicação é, desde os primórdios, confiabilidade de dados. O meio de aumentarmos essa confiabiliade é a codificação de canais. Em um caso típico, ao transmitirmos dados, estes podem não chegar ao destino intactos, devido a ruídos ou interferências de diferentes causas que podem corromper a integridade dos dados. A codificação de canais reduz os efeitos da corrupção de dados adicionando uma certa quantidade de informação redundante num momento prévio à transmissão, o que chamamos de codificação, para então usando esses dados, recuperar a informação corrompida ou destruída, que chamamos de decodificação.

Códigos *Low Density Parity Check* (LDPC) [1] são códigos de correção de erros que tem sido usados com sucesso nas últimas decadas, em vários padrões de comunicação, como WiMAX, WiFi, DVB-S2X, DVB-T2 e mostram uma performance de correção que beira o limite teórico de Shannon para códigos de palavras longas. Contudo, para códigos de palavras curtas ou médias, estes, sofrem uma degradação na performance. Para solucionar tal problema foram desenvolvidos códigos não-binários.

Códigos *Non-Binary Low Density Parity Check* (NB-LDPC) [2] não apresentam o problema de degradação na performance para palavras curtas ou médias, e conseguem reduzir ainda mais a lacuna entre o limite de performance teórico e o das soluções existentes. Estes códigos, no entanto, trazem porém um grande aumento na complexidade de decodificação. O algoritmo de *Extended Min-Sum* (EMS) [3] é considerado o melhor algoritmo voltado para arquiteturas de hardware publicado até o momento, porém, apresenta baixa taxa de transferência e alta latência. Pensando em solucionar estes problemas o algoritmo de *Syndrome-Based* (SYN) [4] foi desenvolvido.

O objetivo deste trabalho é desenvolver uma arquitetura de hardware eficiente para o *Check-Node* de um decodificador NB-LDPC, baseada no algoritmo SYN. Se faz necessário que esta arquitetura mantenha a performance de comunicação do estado da arte, aumente a eficiência e reduza a complexidade de arquiteturas apresentadas até então baseadas no algoritmo de EMS.

Na seção 3 é apresentada uma breve introdução aos códigos NB-LDPC, além do algoritmo de syndrome. Na seção seguinte, são apresentadas as modificações realizadas no algoritmo e a arquitetura de hardware são introduzidas. Na seção 5 temos os resultados deste trabalho, e na última seção as conclusões obtidas e idéias para trabalhos futuros.

3 Códigos LDPC Não-Binários

A extensão de códigos LDPC para códigos NB-LDPC foi descoberta em 1998 por Mackay e Davey. Tal extensão é dada uma vez que códigos LDPC binários estão definidos sobre um campo finito (também chamado *Galois Field*) de tamanho 2 ($GF(2)$), e códigos não-binários por sua vez definem-se em campos de tamanho 2^p ($GF(2^p)$). A diferença básica é que agora estamos processando um agrupamento de bits, o qual nomeamos elemento, ao invés de somente uns e zeros. A quantidade de bits q usada em cada elemento, varia de acordo com o tamanho do campo finito utilizado em cada código ($GF(2^q)$).

Códigos não-binários apresentam várias vantagens em relação à códigos binários[5]: ótimo desempenho de comunicação mesmo com códigos de palavras de tamanho pequeno, complexidade de modulação reduzida na cadeia de comunicação, uma vez que os bits modulados podem ser mapeados diretamente para símbolos, e a performance superior mesmo se comparada com turbo codes, que são até hoje amplamente utilizados.

Para representar um código LDPC usamos uma matriz de checagem de paridade $H_{m,n}$, e nesta definimos:

- m linhas, onde cada uma representa uma equação de checagem de paridade, que chamamos de *Check-Node* (CN) .

- n colunas, uma para cada bit de código, que chamamos de *Variable Node* (VN) .

- d_c : número de elementos não-zero de uma linha, chamado *check node degree.*

- d_v : número de elementos não-zero de uma coluna, chamado *variable node degree.*

A partir disto, podemos representar o código como um grafo de Tanner. As conexões são dadas de forma simples: um check node i está conectado a um variable node j uma vez que haja um elemento não nulo na posição h_{ij} da matriz.

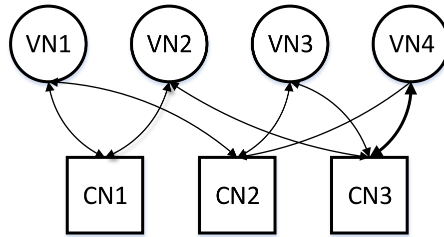

Figura 1: Grafo de Tanner

A decodificação baseada nesta representação em grafo funciona da seguinte forma: os VNs recebem informações do canal, esta informação é passada aos seus CNs conectados, que realizam operações baseado nas suas informações recebidas de diferentes variable nodes, e passam tal informação processada de volta a estes. Os VNs então verificam se a palavra esta decodificada, caso não esteja, uma nova iteração é iniciada, caso esteja decodificada o processo termina.

Analisa-se o crescimento do número de mensagens trocadas na comparação entre um código LDPC binário e um não-binário. Ao invés de para cada conexão enviarmos uma probabilidade associada a um bit ter o valor 1, agora estamos enviando q valores por cada conexão, e junto de cada um deles uma probabilidade associada.

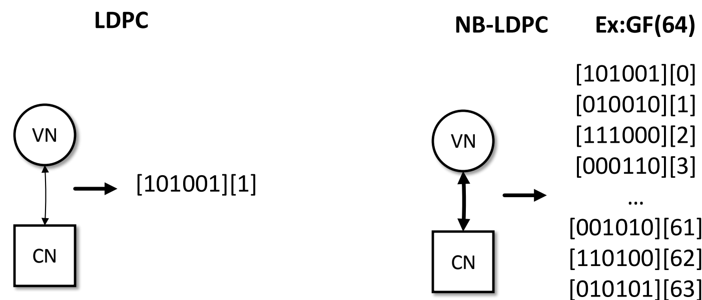

Figura 2: Comparação Binário x Não-Binário

Como a operação realizada no check node (também chamada de *Belief Propagation*) é uma convolução de todas as entradas, em uma implementação

direta a complexidade cresce com $O(q^2)$. Os algoritmos apresentados até o momento só conseguem obter uma implementação de hardware com complexidade aceitável para valores muito baixos de q, com $q \leq 16$.

3.1 Algoritmo Syndrome-Based

O algoritmo Syndrome-Based[5] é uma primeira tentativa de decodificar eficientemente códigos baseados em campos finitos de alta ordem ($q > 16$). A idéia básica do algoritmo é construir conjuntos de *syndromes* baseado nas probabilidades de cada símbolo. Uma *syndrome* é definida como a soma de uma tupla composta por um elemento do campo e sua probabilidade representada em *Log-likelihood Ratio* (LLR) $\{GF(q), LLR\}$ de cada conjunto de entrada do check node. Na figura abaixo podemos ver alguns exemplos. É considerado que as entradas estão ordenadas de acordo com a probabilidade de cada elemento, sendo o elemento mais abaixo o mais confiável:

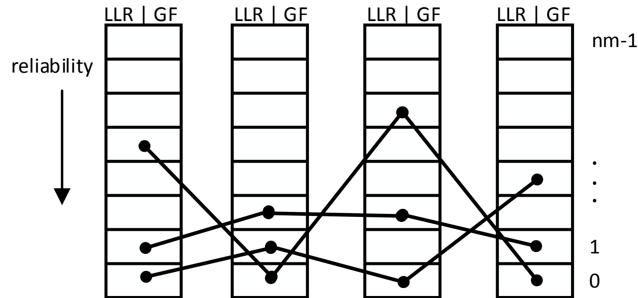

Figura 3: Algoritmo de Syndrome

Criado o conjunto S contendo todas as *syndromes*, estamos interessados nas *syndromes* com as melhores probabilidades. Analisando como as syndromes são construídas, podemos ver que elas quebram o princípio de *Belief Propagation*, que diz que cada valor de saída não pode estar correlacionado com a própria entrada. Assim é necessário um passo adicional para remover a correlação de cada syndrome com o valor de entrada, gerando conjuntos dedicados S_i que não tem correlação com sua própria entrada i. Após

isso é feito um ordenamento de cada set S_i e são selecionados os valores mais confiáveis destes. Duas técnicas ainda são usadas para reduzir o número de cálculos a serem feitos:

Primeiro construímos sets D_i, sendo i o número de entradas do check node, e separamos os sets em sub-sets. Em cada um desses sub-sets D_i, estarão syndromes que contém no máximo i elementos diferentes dos elementos mais confiáveis. Assim, o subset D_0 conterá somente uma syndrome, baseada em todos elementos mais confiáveis. Desta forma, as syndromes mais confiáveis acabarão isoladas nestes sets, e podemos ver que syndromes em sets mais altos como D_3 e D_4 raramente contribuirão para a saída, o que nos permite limitar a quantidade de sub-sets a ser usada.

Junto desta técnica, limitamos a distância máxima dos elementos mais confiáveis de cada syndrome em cada set D_i, de forma a isolar syndromes mais confiáveis. Cada set D_i terá uma distância máxima d_i, porém respeitando a regra de que se maior for o desvio permitido do elemento mais confiável, menor será a distância máxima.

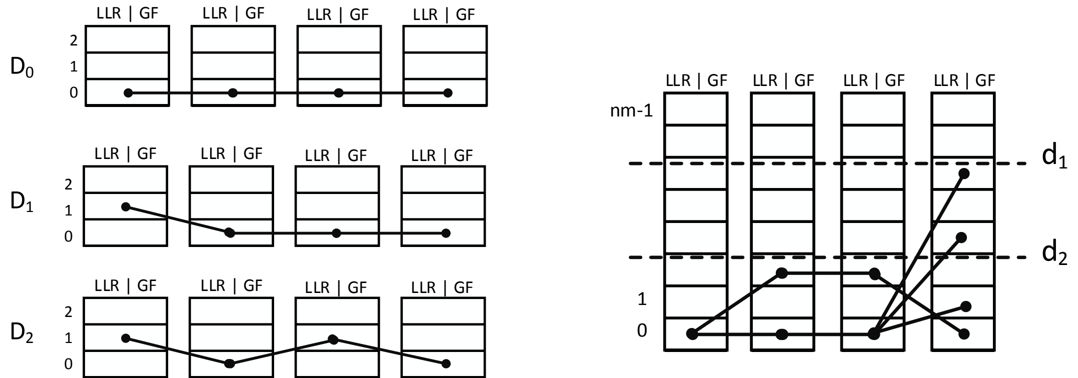

Figura 4: Exemplo dos sets D_i e Limites de Distância

4 Arquitetura de Hardware

A partir do algoritmo descrito na seção anterior é preciso desenvolver uma arquitetura de hardware adequada. É importante citar que tal arquitetura foi desenvolvida e otimizada baseada em parâmetros de código previamente defi-

nidos, de forma a obter a melhor performance possível para estes parâmetros. Uma vez que estes sejam mudados, serão necessárias adaptações equivalentes na arquitetura.

$GF(q)$	64
d_c	4
d_v	2
n_m	13
Iterations	10

Tabela 1: Parâmetros de código

D_1	18
D_2	3
Sondas consideradas	6
I/O por clock cycle (elementos)	3

Tabela 2: Parâmetros da implementação e do algoritmo SYN

4.1 Arquitetura Inicial

Baseado no algoritmo descrito acima podemos imaginar uma arquitetura da seguinte forma:

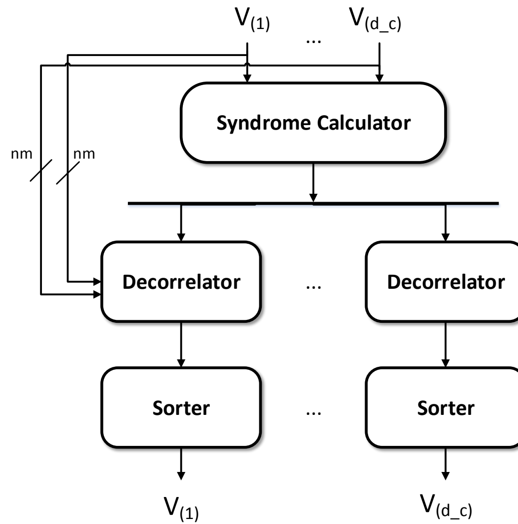

Figura 5: Arquitetura inicial

Alguns problemas dessa arquitetura são o alto uso de registradores para guardar as syndromes calculadas, a grande quantidade de ordenadores antes de cada saída ser atualizada, e estes ainda sendo de tamanho grande, devido ao tamanho dos sets S_i.

4.2 Otimizações

Um meio de reduzir a quantidade de ordenadores e registradores é realizar o ordenamento dos valores de entrada no início da cadeia, e então selecionar uma parcela destes valores para serem geradas syndromes. Assim, usaremos somente um ordenador, e ainda reduziremos o número de registradores a serem usados dependendo do número de valores previamente já selecionados.

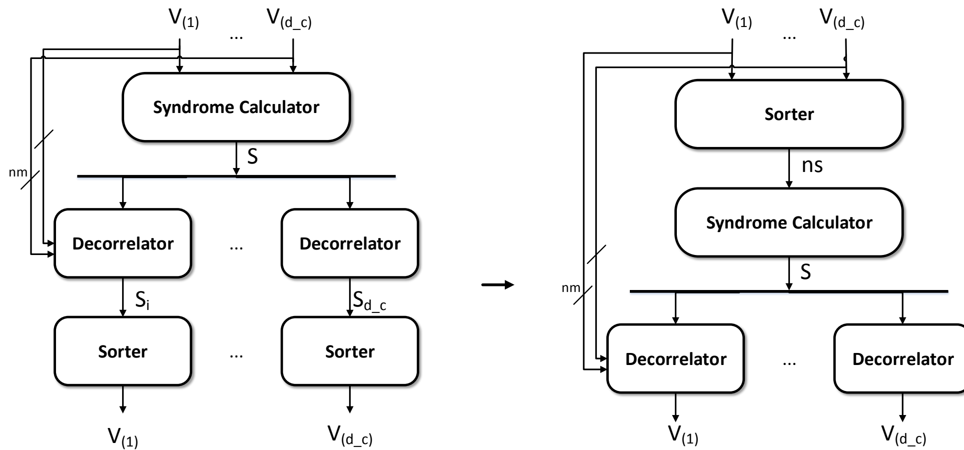

Figura 6: Mudança de posição do ordenador

Analisando o decoder por inteiro, podemos notar que um ordenamento exato não é necessário, pelo fato de que uma vez um VN recebendo mensagens de seus CNs conectados, terá de reordená-las de qualquer jeito. Assim o ordenador usado pode ser adaptado para um ordenamento sub-ótimo.

Baseado nessa análise, para reduzir o tamanho do módulo de ordenamento ainda mais, usa-se uma técnica de sondas. Um número de sondas é distribuído sobre os valores de entrada, e então cada sonda é agrupada a seus valores vizinhos. O ordenamento é então realizado baseado somente no valor de probabilidade da sonda.

É possível reduzir este módulo de ordenamento ainda mais se realizarmos o ordenamento de somente um grupo de cada entrada por ciclo de clock, como mostrado abaixo. Os blocos cinza são as sondas, e foram devidamente agrupadas.

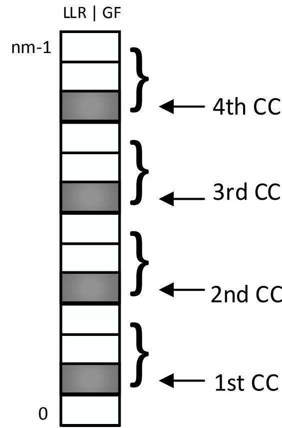

Figura 7: Esquema de sondas

Com essa modificação, teríamos no módulo seguinte um número de syndromes a ser calculado proporcional ao número de VNs conectados. Modificando o ordenador para obtermos somente uma sonda ordenada por ciclo de clock, ao invés de *ns* como mostrado anteriormente, o módulo que cálcula as syndromes é reduzido ainda mais.

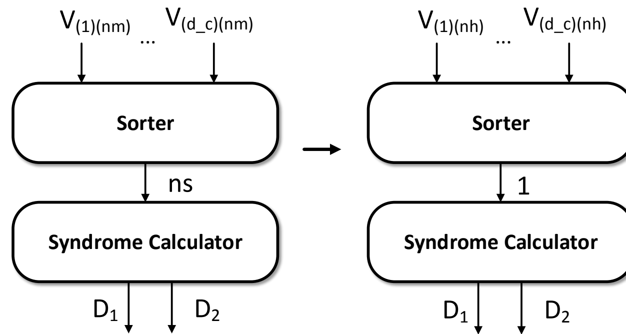

Figura 8: Redução da saída do ordenador

Sequencializando o processamento, teremos agora um valor para D_1 sendo calculado a cada ciclo de clock. Contudo, os valores para D_2, dependem de duas sondas. Estes valores não são necessáriamente sequenciais, pois uma vez

que duas sondas ordenadas são baseadas na mesma entrada, o valor formado por estas não é um valor D_2 válido, assim não sabemos ao certo quando haverá um valor D_2. Para o caso deste ser gerado em paralelo com D_1, precisamos um ordenador adicional na saída.

Analisando as syndromes D_1 de uma mesma entrada, nota-se que elas compartilham os mesmo valores nos cálculos. Para remover estes valores redundantes foi criado um módulo que pré-calcula tais valores, a fim de reduzir o hardware no módulo *syndrome calculator*. Este módulo pré-calcula valores para cada set D_1 e para todas possibilidades de sets D_2 válidos.

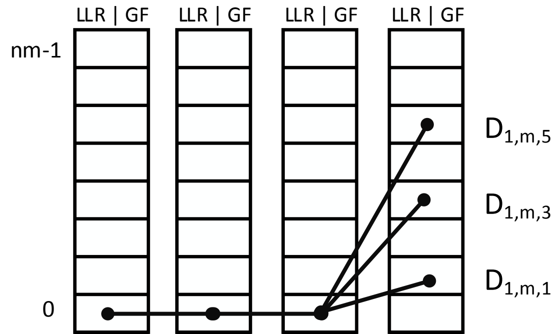

Figura 9: Valores comuns aos cálculos de D_1

4.3 Arquitetura Final

Juntando esta última modificação com o módulo adicional de ordenamento temos uma nova arquitetura. Analisando-a, e comparando com a arquitetura inicial, podemos ver que agora usamos apenas dois ordenadores de tamanho reduzido, o modulo *syndrome calculator* foi também drásticamente reduzido em relação ao número de registradores necessários, e também a fiação necessária é menor, pois levamos em conta valores baseados somente nos elementos mais confiáveis para a decorrelação. Todos os problemas existentes na arquitetura anterior foram solucionados.

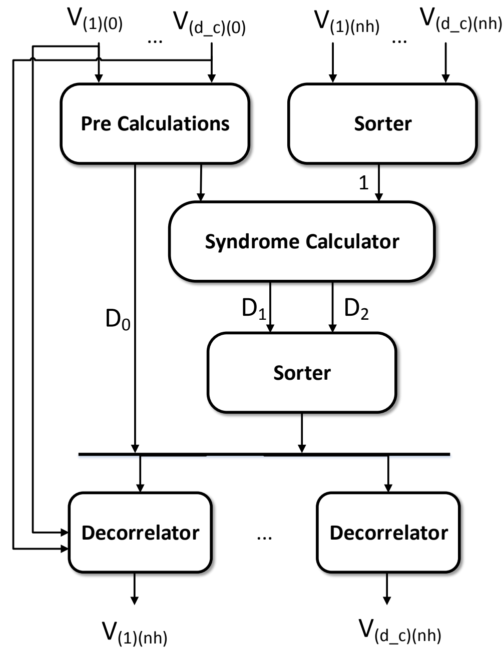

Figura 10: Arquitetura de Hardware Final

Um overview da implementação realizada, de acordo com parâmetros iniciais pré escolhidos pode ser visto abaixo, nota-se os registradores de pipeline e larguras das interconexões usadas.

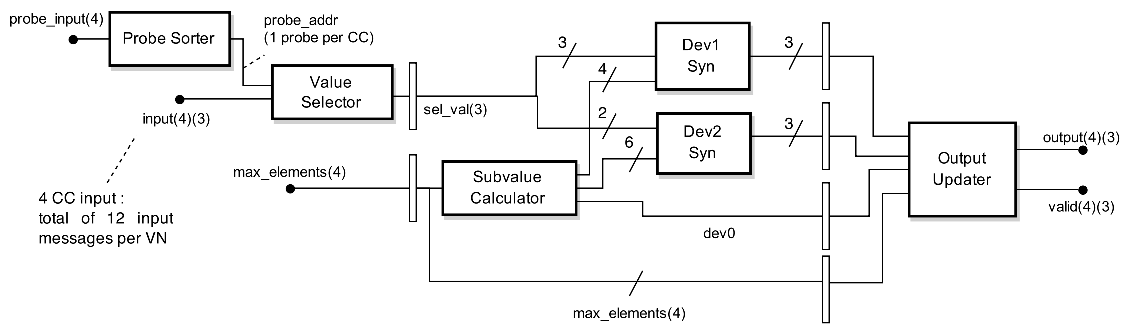

Figura 11: Hardware Overview

5 Resultados

Os resultados de síntese apresentados nesta seção foram feitos usando um FPGA Xilinx Virtex 5 XC5VLX50T com *speed grade* -3. Todos foram obtidos após o *Place and Route*. Foram realizadas comparações com o hardware baseado no algoritmo de *Extended Min-Sum* usando o esquema de *Forward Backward*, que é até hoje o melhor algoritmo desenvolvido voltado para implementações de hardware.

5.1 Performance de Comunicação

Na figura abaixo comparamos a performance de comunicação do hardware desenvolvido ao algoritmo de estado da arte até hoje utilizado. Para uma comparação justa os parâmetros utilizados em ambos os códigos foram os mesmos. Como pode ser observado, para valores de SNR de 2 a 4.5 a performance atingida é equivalente, ainda, para valores de 4.5 a 5.5, o hardware desenvolvido apresenta uma performance ainda melhor.

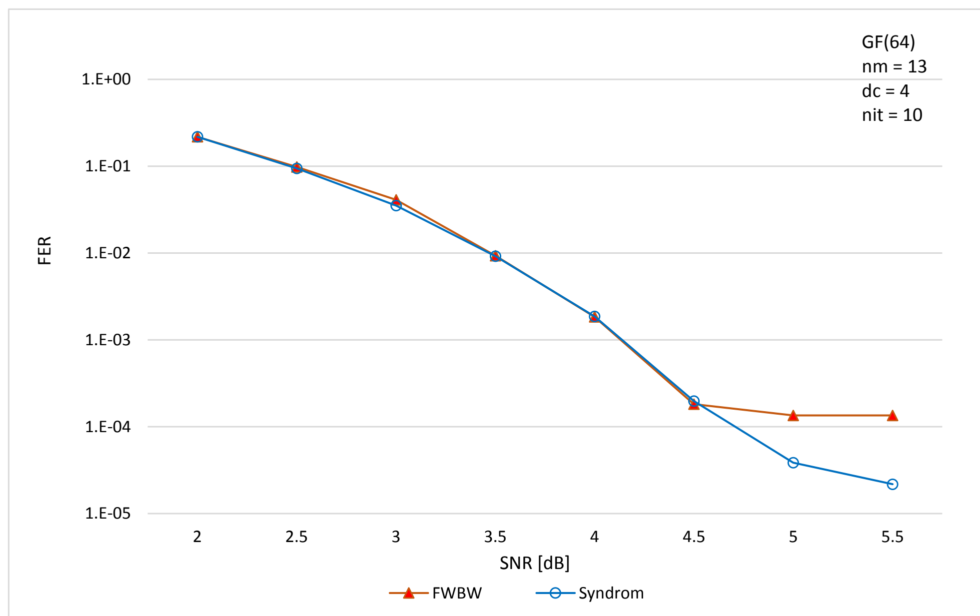

Figura 12: Communications Performance

5.2 Área

Para a comparação de área, também é preciso ser dito que as quantizações para amostragem dos bits em ambas implementações são exatamente as mesmas. A arquitetura apresentada apresenta apenas 41% da área ocupada pelos registradores, e 48% da área ocupada pelas LUTs, se comparado à implementação do algoritmo de FWBW.

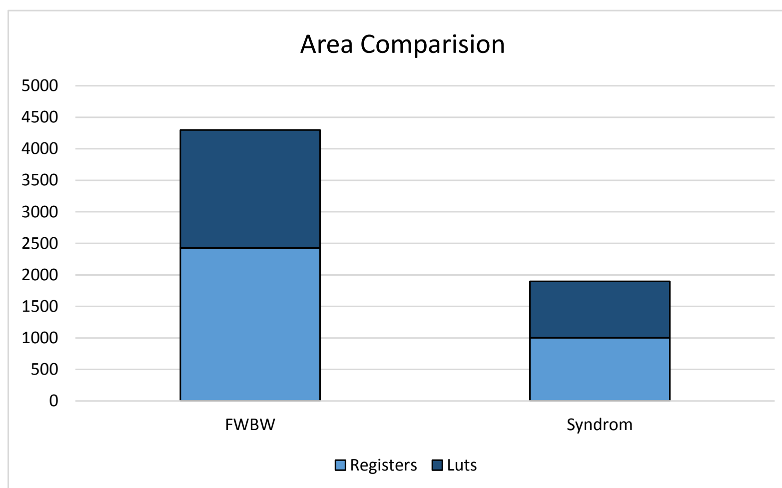

Figura 13: Area comparison

5.3 Vazão

Na análise de *throughput*, primeiro consideramos a quantidade de elementos, e em segundo a relação da quantidade de elementos por unidade de área. Na primeira, nossa arquitetura ultrapassa o estado da arte em mais de 6 vezes, e na segunda o número chega a mais de 14 vezes.

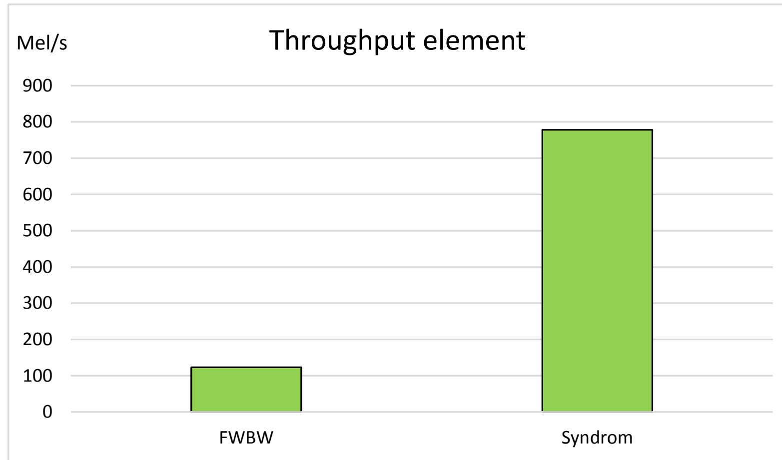

Figura 14: Throughput Comparison

5.4 Resultados Finais

Ainda, uma outra arquitetura foi desenvolvida, para testarmos a escalabilidade da arquitetura apresentada em termos de área e throughput. O resultado obtido mostra um crescimento quase linear. Abaixo, segue a tabela com todos os resultados de síntese obtidos.

	EMS-FWBW	Syndrome-Based	Scaled Syndrome
Registers	2426	1002	2200
Luts	1872	898	2077
Frequency (Mhz)	123.2	259.4	263.4
Throughput (Mel/s)	123.1	778.68	1592
Throughput/area (Kel/s/unit)	28.64	409.8	371.2
Latency (CC)	21	12	6

Tabela 3: Final results

6 Conclusão

Neste trabalho apresenta-se um novo hardware para o CN de um decodificador NB-LDPC. Ele é baseado no algorítmo *Syndrome-Based*, que provê implementações de hardware eficientes, e é uma alternativa ao até então utilizado algoritmo de *EMS-FWBW*. Essa é a primeira tentativa de decodificação eficiente para um código baseado em campos de alta ordem ($q > 64$). A arquitetura apresenta alta vazão, eficiência de área e baixa latência.

Em uma comparação direta com a arquitetura de estado da arte baseada no algoritmo de *EMS-FWBW*, o hardware apresentado ocupa menos da metade da área obtendo cerca de seis vezes a vazão. Em relação à performance de comunicação, o hardware apresentado obtém a mesma performance do estado da arte, podendo ainda para valores particulares de ruído excedê-la.

Na arquitetura apresentada, os parâmetros utilizados foram usados estratégicamente baseado nos parâmetros do código a ser usado, caso modificaçoes neste sejam feitas, serão necessárias modificações no hardware, de forma a atingir a mesma performance.

6.1 Trabalhos futuros

Tendo em mente tecnologias à serem desenvolvidas, modulações de ordem mais alta ($256QAM$) terão de ser utilizadas em sistemas de comunicação. Para isso seria interessante estender essa idéia e investigar o algoritmo *Syndrome-Based* e uma implementação de hardware considerando campos de ordem ainda maior, como $q = 256$.

Até hoje não foi apresentado algoritmo que escale eficientemente com o valor de q para uma complexidade fixa. Como atingir tal, mantendo a performance do estado da arte é chamado de *Holy Grail* da decodificação não binária.

Referências

[1] Robert G. Gallager. Low Density Parity Check Codes. *The Bell System Technical Journal*, 1963.

[2] Matthew C. Davey and David MacKay. Low-Density Parity Check Codes over GF(q). *IEEE COMUNICATIONS LETTERS*, June 1998.

[3] D. Declercq and M. Fossorier. Decoding algorithms for nonbinary LDPC codes over GF. *IEEE Transactions on Communications*, April 2007.

[4] P. Schläfer, N. Wehn, M. Alles, T. Lehnigk-Emden, and E. Boutillon. Syndrome Based Check Node Processing of High Order NB-LDPC Decoders. *International Conference on Telecommunications*, 2015.

[5] S. Pfletschinger, A. Mourad, E. Lopez, D. Declercq, and G. Bacci. Performance evaluation of non-binary LDPC codes on wireless channels. *Proceedings of ICT Mobile Summit*, June 2009.

University of Kaiserslautern

Department of Electrical Engineering and Information Technology

Microelectronic Systems Design Research Group

Bachelor Thesis

An Area Efficient FPGA Implementation for the Syndrome based Non Binary LDPC Check Node Algorithm

---

Presented:	July 13, 2015
Author:	Vitor Kieling
Research Group Chief:	Prof. Dr.-Ing. N. Wehn
Tutor:	Dipl.-Ing. Philipp Schläfer

Statement

I declare that this thesis was written solely by myself and exclusively with help of the cited resources.

Porto Alegre, July 13, 2015

Vitor Kieling

Acknowledgments

I would like to express my special thanks to my advisor Dipl.-Ing. Philipp Schläfer. For conducting my work at the University of Kaiserslautern, for all the knowledge shared, virtuous ideas and guidance provided through the period of this work.

I am thankful to my co-advisor Msc. Vladimir Rybalkin, for his teaching, suggestions and patience, and to Prof. Dr.-Ing. Norbert Wehn for the opportunity and the amazing work structure provided at the Microelectronic Systems Design Research Group.

Last, I want to express all my gratitude to my parents, Gilberto Kieling and Ziara Kieling, without them, nothing would have been possible.

Abstract

With the new throughput limits required for today applications, dedicated hardware architectures are needed to obtain proper error correction. To achieve that, the evolution from binary LDPC codes to non-binary LDPC codes is necessary, and this evolution implies in a greater decoding complexity. The best algorithm that provides a good hardware complexity and performance trade-off is the *Extended Min-Sum* using the *Forward Backward* scheme. Unfortunately, it presents low throughput and high latency. In this work, a new hardware for a check node of a NB-LDPC decoder based on the *Syndrome-Based* algorithm is presented. It provides the state-of-the-art communication performance, combined with reduced area,low latency and high throughput.

Contents

1 Introduction **5**

2 Channel Coding **6**
2.1 Basics . . . 6
2.2 Channel Capacity . . . 7
2.3 Linear Block Codes . . . 7
2.3.1 Generator Matrix . . . 8
2.3.2 Parity Check Matrix . . . 8
2.4 Convolutional Codes . . . 8
2.5 Decoding Algorithms . . . 9
2.5.1 Optimal Decoding Problem - Maximum Likelihood . . . 10

3 Low Density Parity Check Codes **11**
3.1 Representation . . . 11
3.2 Encoding . . . 12
3.3 Iterative Decoding . . . 13
3.3.1 Belief Propagation Decoding . . . 13
3.3.2 Bit Flipping Decoding . . . 15
3.4 Decoding Performance and Limitations . . . 16

4 Non Binary Low Density Parity Check Codes **18**
4.1 Definition . . . 18
4.2 Advantages . . . 19
4.3 Decoding . . . 19
4.3.1 Reducing Complexity - Sub-Optimal Decoding . . . 19
4.3.2 Extended Min-Sum Algorithm . . . 20
4.3.3 Syndrome-Based Algorithm . . . 23
4.3.3.1 Reducing Syndrome Set . . . 24

5 Hardware Architecture **28**
5.1 Existing Solutions Analysis . . . 28
5.2 Optimizing Area . . . 29
5.3 CSE Simulation Environment . . . 33
5.4 Hardware overview . . . 33
5.4.1 Probe Sorter . . . 35
5.4.2 Value Selector . . . 36
5.4.3 Subcalculations Module . . . 37
5.4.4 Dev1 Syndrom and Dev2 Syndrom . . . 38
5.4.5 Output Updater . . . 39

6 Results **41**
6.1 Communications Performance . . . 41
6.2 Area Analysis . . . 41
6.3 Throughput Analysis . . . 42
6.4 Scaling the architecture . . . 43
6.5 Final results . . . 44

7 Conclusion **45**
7.1 Future Work . . . 45

8 Appendix **46**
8.1 Galois Field Arithmetic 46
8.1.1 Definition 46
8.1.2 Construction 46
8.1.3 Representation 47
8.1.4 Addition and Multiplication 48
8.2 Syndrome Algorithm Execution Example 48
8.3 Hardware Top Module Code 53
List of Figures 58
List of Tables 60
List of Listings 61
List of Abbreviations 62
References 63

1 Introduction

Nowadays we can not even picture a world without cell phones, internet, digital television, and another hundreds of systems that comunicates with each other, and sometimes even ubiquity, make our lives easier.

The key for the success of those communication systems is data reliability, and the most well-known tool to achieve this reliability is channel coding. In a typical scenario, when we transmit data, this data will not reach its destiny intact. There will be noise and interference from different sources that will corrupt the integrity of this data, leading us to not receive the data we wanted. Channel coding solve this problem by adding a certain amount of redundant information, to this data we want to transmit. We call it encoding. Using this data, at the receiver, we will be able to recover the original data, even if it has been corrupted. That is what we call decoding.

Low Density Parity Check (LDPC) codes, first time presented by Gallager[1] in 1963, are well-known error correction codes that has been used in the last decades in many commercial communication standards, such as WiMAX, WiFi, DVB-S2X, DVB-S2, DVB-T2. They have been proved to perform very close to the Shannon limit [2] for very long codeword lenghts. However, when it comes to small or moderate code word lengths, or when a high order modulation is used on the transmission (which is the case for upcoming high-speed standards), the LDPC codes suffer an undesirable degradation in communications performance.

This extension of the LDPC codes, called *Non-Binary Low Density Parity Check* (NB-LDPC) codes [3], which are the topic of this thesis, resolve this problems by extending the *Galois Field* (GF) size used in their binary counterpart from 2 to q (with q>2), and allows for a simplified demapper, directly mapping the received information to groups of bits that we call symbols, instead of single bits.

The gain in communication performance obtained with NB-LDPC codes comes with a significant increase in the decoding complexity and there are already developed algorithms with excellent communications performance using the Fourier domain [4]. To match new communication standards, dedicated hardware architectures are needed, and for that, those algorithms are still too complex. The best algorithm published so far which provides a good hardware complexity and performance trade-off is the *Extended Min-Sum* (EMS) Algorithm[13]. Unfortunately, it suffers from low throughput and high latency. In this thesis a new check node hardware architecture, based on the *Syndrome-Based* (SYN) Algorithm[5], is presented. It has state-of-the-art communications performance, combined with low area, low latency and high throughput.

In the first chapter we give a brief overview at the basics of channel coding. Section 3 explains a little about LDPC codes and in section 4, the extension of them to non-binary codes. We give in section 5 all the explanation about the designed hardware, including diagrams and circuits. In section 6 we present the results of our work, including a fair comparison with today state-of-the-art decoders.

2 Channel Coding

2.1 Basics

To allow a better understanding of what Channel Coding is, it is necessary to describe a communication system and its modules. The system below describes a basic communication system.

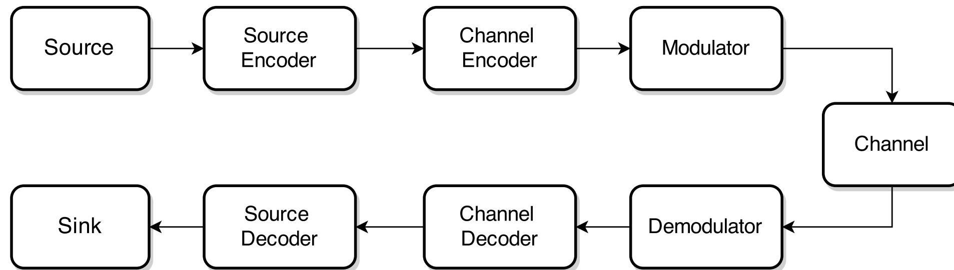

Figure 2.1: Communication Chain

Source and Sink : information source provided from any digital system that has some data to be transmited.

Source encoder and source decoder : the encoder converts the information source bits into a new bit sequency with a more efficient representation. It is also usually called compression. For media related date (audio, video, image) we also define two kinds of compression: *lossless* and *lossy*. The first generating a compressed file with no quality loss, and the latter, reducing the information provided even further in the trade of quality, frequently used where the loss can be acceptable or inconspicious.

Channel encoder and channel decoder : the idea behind the channel encoder is to add some redundant information in order to protect the data, to be transmitted over a channel that can be subjected to distortion, noise and interference. With these added redundant bits, the decoder should be able to recover the original data, despite the noise injected by the channel.

Modulator and demodulator : the modulator converts the bit stream provided by the encoder to a signal in a form that should usually match with the channel to be used. Common modulation methods are: *amplitude* modulation, *frequency* modulation and *phase* modulation, which are sometimes combined. The demodulator makes the counterpart, recover the bit stream based on the used modulation.

Channel : the channel is the physical medium through which the data stream is transmitted. Channels can add noise and interference to the data being transmitted. We model it, using a probabibilistic model. Therefore the channel output will be determined by the sum of the input x with the noise n :

$$y = x + n \tag{2.1}$$

We usually define a communication channel with a triple, consistent of: an input alphabet, an output alphabet, and for each input-output pair, a transition probability $p(i, o)$.
Before defining a code, we describe channel capacity, and cite that there are two different well-known types of codes, block and convolutional codes. Being the block codes, linear or non-linear. In this thesis we focus on linear codes.

2.2 Channel Capacity

Based on the above mentioned model, Shannon defined *channel capacity*[2], which measures how much information can be transmitted through a channel. We must remember that this notion of capacity is only a theorical limit, it does not guarantee the existance of such a scheme that achieves this limit. This can be calculated as follows:

$$C = W \log(1 + SNR) \ [bits/s] \tag{2.2}$$

with SNR the signal-to-noise power ratio detected at the receiver and W the channel bandwidth. Setting a tranmission rate R to a $R < C$ value, an error probability as small as desired can be achieved. In channel coding, the *Sphere Packing Bound* (SP59) limit is also used. It is based on the Shannon limit, but takes also into account the code ratio to calculate the communications limit.

2.3 Linear Block Codes

We define an alphabet over a finite field, also called Galoin field, of size 2, $GF(2)$ (see appendix), that means we have only two possible symbols, beeing them 0 and 1, one bit. Using block coding, we segment the information sequence into blocks of a fixed size k, having then $K = 2^k$ possible distinct messages. At the encoder, each of this input messages will be encoded into a longer binary sequence of size n, being $n > k$, we call that a *codeword*. With the set of all this messages, we have a (n, k) block code, with $n-k$ redundant bits, that will be used to correct the errors caused by the communication noise or interference. The ratio between the number of information bits and the number of redundant bits added by the encoder we call *coderate*.

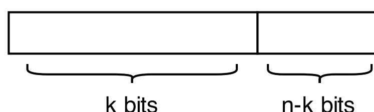

Figure 2.2: Block code codeword

Another interesting parameter used in some linear block codes is the minimum distance, or Hamming distance [6], which is the number of positions in which two different codewords c_1 and c_2 differ. That means, a receiver based on a d distance code, is able to detect up to $d-1$ tranmission errors, since changing only $d-1$ positions of a codeword can never lead to an unwanted, but valid, codeword.

A popular notation used for a linear block code is $(n, k, d)_q$, describing a block cover over an alphabet of size q, with a block lenght n, message lenght k and distance d. For maximum distance separable codes, the d parameter may be missing.
Some examples of famous linear block codes are: Reed-Solomon Codes, Hamming Codes, Hadamard Codes.

2.3.1 Generator Matrix

Since a linear block code $C(n, k)$ is a k-dimensional subspace of all the possible combinations over a $GF(q)$, exists a set G of k independent codewords in C, $G = (g_0, g_1, ..., g_k)$, such that every codeword $v \in C$ is a linear combination of these k linearly independent codewords. Simplifying, for a given information vector $u = (u_0, u_1, ..., u_k)$ to be encoded, the resulting codeword $v = (v_0, v_1, ..., v_k)$ can be expressed as the linear combination of the rows of a $k \times n$ matrix G with the information bits in u:

$$v = u.G \tag{2.3}$$

This matrix G we call Generator Matrix for the given block code. Expanding it to a better visualization:

$$v = \begin{bmatrix} u_0 \\ u_1 \\ ... \\ u_{k-1} \end{bmatrix} . \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ ... & ... & ... & ... \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} \tag{2.4}$$

2.3.2 Parity Check Matrix

Defined the generator matrix, a parity check matrix can be derived from it (or vice-versa). This latter matrix is used in the decoding to check if a codeword c is valid given the linear block code C, respect to the equation:

$$H.c^T = 0 \tag{2.5}$$

or the common-used equivalent form

$$c.H^T = 0 \tag{2.6}$$

As C is a (n, k) k-dimensional block code, the null (or dual) space of this code, denoted C_d, is a $(n, n-k)$, $(n-k)$-dimensional code. Using $H = (h_0, h_1, ..., h_{n-k-1})$ as the set of $n-k$ linearly independent codewords in the basis of C_d. Then, the codewords in C_d form the following $(n-k) \times n$ parity check matrix:

$$H = \begin{bmatrix} h_0 \\ h_1 \\ ... \\ h_{n-k-1} \end{bmatrix} = \begin{bmatrix} h_{0,0} & h_{0,1} & ... & h_{0,n-1} \\ h_{1,0} & h_{1,1} & ... & h_{1,n-1} \\ ... & ... & ... & ... \\ h_{n-k-1,0} & h_{n-k-1,1} & ... & h_{n-k-1,n-1} \end{bmatrix} \tag{2.7}$$

Then H is what we call a parity check matrix for the code C, which is also a generator matrix for the dual code C_d, and C is said to be the *nullspace* of H. Following that, we can also think of the equation

$$G.H^T = O \tag{2.8}$$

where O is the $k \times (n-k)$ zero matrix.

2.4 Convolutional Codes

Convolutional codes were introduced in 1955 by Peter Elias[7]. It was thought that convolutional codes could be decoded with arbitrary quality at the expense of computation and delay. In 1967, Andrew Viterbi determined that convolutional codes could be maximum-likelihood decoded with reasonable complexity using time invariant trellis

based decoders - the Viterbi algorithm[8]. Unlike block codes, where the information bits are followed by the parity bits, convolutional coding spread the information bits along the bit sequence. That means that the convolutional codes map information to code bits not block wise, but sequentially convolve the sequence of information bits according to some rule. We also have the n and k code parameters, but in addition, a new m parameter, that is the number of memory elements (registers) used to store data from past bits. That makes the code defined by a triple $[n, k, m]$.

The encoder basically uses a *sliding window*, to calculate $r > 1$ parity bits by combining various subsets of bits in the window. Then in every time step, this window overlap and slide, calculating new parity bits. This sliding represents the *convolution* of the encoder over the data, which provide the term *convolutional codes*.

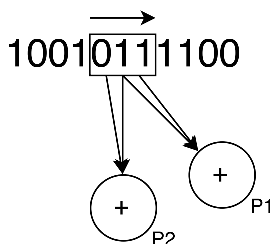

Figure 2.3: Convolutional coding

In the example figure, we are generating 2 parity checks. One P1 based on the second and third bits of the window, and one P2 based on the first and second bits of the window. The size of the window, in bits, is called constraint length. With a bigger window, the number of parity bits influenced by any information bit is larger. Due to this fact, with a bigger window we might have some resilience to errors. The trade-off is that it will also take more time to decode codes using a long constraint length.

The ability to perform economical maximum likelihood decoding is one of the major benefits of convolutional codes. They are also especially useful for iterative processing, for example, the turbo codes. Turbo codes are a new class of iterated short convolutional codes, which can closely approach the Shannon limits with lower decoding complexity, and are beeing sparsely used nowadays.

This is only a brief overview about convolutional codes since they are not the topic of this thesis, but aswell important, beeing worth to be mentioned.

2.5 Decoding Algorithms

We usually deal with two distinct types of decoding: those based on *hard* decoding, and those based on *soft* decoding. When hard decoding is used, we deal only with a sent message, and if this message is either correct or not. When soft decoding is used, what we use is an associated reliability with a sent message, that means, a probability that the sent message is true or not. Of course, the soft decoding scheme performs better in the presence of corrupted information than the hard decoding scheme, althought the decoding of this type of messages is costly.

2.5.1 Optimal Decoding Problem - Maximum Likelihood

Based that we have sent a codeword, how to tell which codeword was sent if we know that errors might have been inserted during the transmission and if we are not sure how can we trust in this received information?

What we can do is analyse the probability that this codeword was sent, list all the possible K codewords and a conditional probability for every one of them. After that, return the one that have the highest probability. That is called *Maximum Likelihood* (ML) Decoding.
In essence, what we want is to find a codeword $c_i \in C$, given a received vector r which maximizes $P(c = c_i|r)$. This is also called *maximum a posteriori* rule. By Bayes rule:

$$P(c|r) = \frac{P(c)P(r|c)}{P(r)} \tag{2.9}$$

Since $P(r)$ is independent of c, and if we assume that each codeword is chosen with an equal probability, to maximize this equation we only have to maximize $P(r|c)$:

$$P(r|c) = \prod_{i=1}^{n} P(r_i|c_i) \tag{2.10}$$

The ML decoding is a NP-problem, that means the calculation complexity grows exponentially with the number of input symbols. With the years, the to-be-achieved goal, is to find an algorithm that provides a great performance as the ML, but with a reduced calculation complexity.

3 Low Density Parity Check Codes

LDPC Codes are a class of the before mentioned linear block codes. They provide communications performance near the Shannon limit and were first proposed by Gallager in his PhD dissertation in 1960[1]. They remained silent for the followed 35 years. In the mid 90s however, the study of this codes has been re-discovered, and up to today they are widely used in many applications.
With the growing popularity of the non-binary LDPC codes, we usually call this codes binary LDPC codes, since they are defined over a GF(2).

3.1 Representation

From their name, LDPC codes are codes defined by a very spare, or low density, parity check matrix, that is, the H matrix has a very low number of non-zero entries. That sparseness guarantee a decoding complexity that increases only linearly with the code length.

$$H = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{3.1}$$

LDPC codes are usually designed by first constructing a parity check matrix, and then from this one derivating a generator matrix.
The difference between LDPC codes and other block codes is in the way they are decoded. Instead of using the ML scheme, that uses short block codes to make the ML task less complex, LDPC codes are decoded iteratively, using a graphical representation of the H matrix.

From the $H_{m,n}$ matrix we observe:
- m rows, each one representing a parity check equation, that corresponds to the function of a *Check Node* (CN) .
- n columns, one for each code bit, that corresponds to the function of a *Variable Node* (VN) .
- d_c : the number of non-zero elements in a row, we name it check node degree.
- d_v : the number of non-zero elements in a column, we name it variable node degree.

Using that notation we can represent the H matrix using a Tanner graph, with the defined nodes. The rule for the connections is simple: a check node j is connected to a variable node i if the position h_{ij} in the H matrix is a 1.

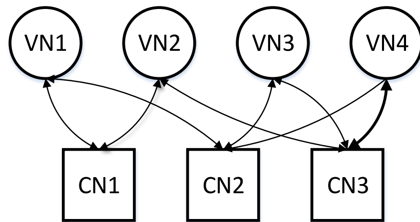

Figure 3.1: Tanner Graph Representation

Regarding the iterative process over this graph we may have two possible scheduling schemes:

- *parallel* : in each iteration, all the CNs are updated, and after that, with the new information computed by the CNs, the VNs are updated.

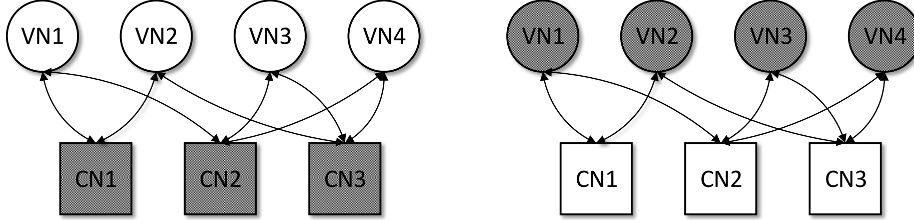

Figure 3.2: Parallel Scheduling

- *serial* : can be layered, or shuffled.

- *layered* : first one CN is updated, then after that, all the VNs connected to this CN are updated, repeating that for the following CNs, until all of them are updated.

- *shuffled* : the opposite of the layered scheme, first one VN is updated, and after that all the CNs connected to this VN are updated, repeating until all the VNs are updated.

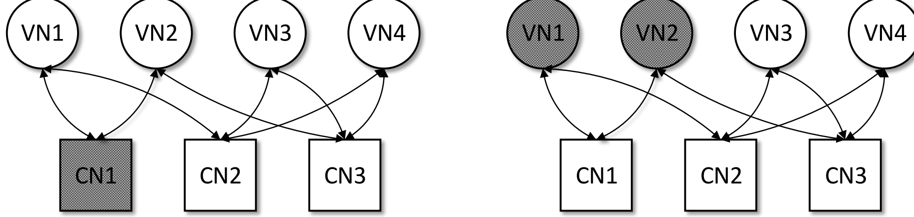

Figure 3.3: Serial Scheduling

The serial schedule reduces the possible parallelization, but also reduces the number of iterations necessary by the used decoding algorithm to perform a full decoding.

3.2 Encoding

Encoding is done with a generator matrix, as shown in chapter 2. An usual approach in LDPC codes is first to built a parity-check matrix H, and from this one, derivate the generator matrix G:

$$c = [\frac{s}{p}]^T \tag{3.2}$$

The code vector c is divided into two systematic parts s and p. From $c.H^T = 0$ we get:

$$\begin{aligned} &[H_1|H_2].[\frac{s}{p}] = 0 \\ &H_1.s + H_2.p = 0 \end{aligned} \tag{3.3}$$

The parity check matrix is divided into two parts. H_1 for the systematic part and H_2 for the quadratic part. After solving to the parity check vector we get:

$$p = H_2^{-1}.H_1.s = G'.s \tag{3.4}$$

To calculate the G matrix, this H_2 part must be inversible. Then, we generate the G matrix by adding the identity matrix:

$$G = [I|G'] \tag{3.5}$$

3.3 Iterative Decoding

As mentioned above, LDPC codes uses an iterative decoding, also called message-passing decoding, because messages are passed from VNs to CNs and vice-versa. The messages are exchanged between VNs and CNs until the message is sucessfully decoded or a pre-defined number of iterations is met. We may name different message-passing algorithms based on the type of messages exchanged, or the type of operation performed on the nodes. Some of them may use a hard decoding scheme, such as bit flipping algorithms, and some a soft decoding scheme, for example, the belief propagation decoding.

3.3.1 Belief Propagation Decoding

The *Belief Propagation* (BF) Algorithm operates with bit probabilities. We usually use $log - likelihood\ ratios$ to represent the probabilities, for the mathematical convenience of turning multiplications into additions, leading to a lower implementation complexity, and for the robustness provided when quantizating messages with a small number of bits. Therefore, this type of decoding is also called sum-product decoding, due to the fact that it allows the CN and VN calculations to be done using only sum and product operations. Since we are using only a binary representation, it is easy to see that for example, once we have the probability of a bit to be 1, $P(x = 1)$, it easy to find a zero probability because $P(x = 0) = 1 - P(x = 1)$, therefore we only need to store one probability for a given x value. To calculate the log-likelihood ratio we use:

$$L(x) = \ln(\frac{P(x = 0)}{P(x = 1)}) \tag{3.6}$$

If $p(x = 0) > p(x = 1)$ then $L(x)$ is positive, and we know that the bit is more probable to be 0, else, if $L(x)$ is negative, we know that the bit is more probable to be 1. In every VN we have the *instrinsic* information, provided by the channel, and the *extrinsic* information, received by every connected CN (parity checks). What we want is to calculate the *Maximum a Posteriory* (MAP) probability for every codeword bit, which is the probability of a bit n to be 1 conditional to the event N that all parity checks are satifstied.

$$P_n = P(c_n = 1|N) \tag{3.7}$$

The extrinsic information received by the CNs must follow a basic principle: every extrinsic information received by a VN m most not be correlated with its own intrinsic information. This exact extrinsic message $R_{m,n}$ from CN m to VN n is the *Log-likelihood Ratio* (LLR) of the probability that the bit n causes the parity check m to be satisfied. $W_{m,n}$ is the message from VN to CN. $N(m)$ is the set of variable nodes connected to the mth check node, and $M(n)$ the set of check nodes connected to the nth variable node.

x_n is the intrinsic channel information for a given VN. This probability if the VN bit n is a 0 is given by:

$$P_{m,n}^{ext} = \frac{1}{2} - \frac{1}{2} \prod_{n' \in M'_n \neq n} (1 - 2P_{n'}^{int}) \tag{3.8}$$

and then we calculate the LLR:

$$R_{m,n} = LLR(P_{m,n}^{ext}) = \ln \frac{P_{m,n}^{ext}}{1 - P_{m,n}^{ext}} \tag{3.9}$$

The total LLR for the nth VN, L_n, will be the sum of every message provide by the connected CNs, plus its own instrinsic x_n:

$$L_n = LLR(P_n^{int}) = x_n + \sum_{n \in M(n)} R_{m,n} \tag{3.10}$$

Below, we also add an algorithmic version of the Belief-Propagation Algorithm.

```
1  procedure Decode(x)
2    It = 0                                //Initialization
3    for n = 1:N do
4      for m = 1:M do
5        W_{m,n} = x_n
6      end for
7    end for
8
9    repeat
10     for m = 1:M do                      //Step 1: Check node messages
11       for n ∈ M_n do
12         R_{m,n} = log(P^{ext}_{m,n} / (1 - P^{ext}_{m,n}))
13       end for
14     end for
15
16     for n = 1:N do                      //Test
17       L_n = ∑_{n∈M(n)} R_{m,n} + x_n
18       c_n = (1, L_n 0|0, L_n > 0)
19     end for
20
21     if It = It_{max} or H.c^T = 0 then
22       Finished
23     else
24       for n = 1:N do                  //Step 2: Variable node messages
25         for m ∈ M_n do
26           W_{m,n} = ∑_{n∈M_n} R_{m,n} + x_n
27         end for
28       end for
29       It = It + 1
30     end if;
31   until Finished
32 end procedure
```

Listing 3.1: Algorithm Belief-Propagation Decoding [9]

3.3.2 Bit Flipping Decoding

In the bit flipping algorithm only hard decision messages are exchanged between the nodes. There is no probability, just a bit message. The check node detects if the parity check is satisfied based on a module-2 sum of all the incoming bit messages, and then send the result of this sum back to the variable nodes. If most of the messages received by a variable node are different from its original value, it flips the current value.

The sparseness of the matrix H let the VN bits be spread over different parity check equations and all of them are unlikely to contain the same set of variable node connections. The basic principle is that if codeword bit is involved in a large number of incorrect parity check equations, this bit is probably incorrect. The existance of cycles or some similarity between the parity check equations may break this principle and

therefore an incorrect bit may be considered correct, or flipped unnecessary.

It is also clear that if we have a very low *Signal-to-noise Ratio* (SNR) bit flippping decoding is unpractical. However, for high SNR values, this type of decoding has been shown to provide simplified hardware implementations, resulting in a really low hardware cost, but mantaining a really good performance.

Below we add an algorithmic version of the Bit Flipping Algorithm.

```
1  procedure Decode(y)
2   It = 0                                  //Initialization
3   for n = 1 : N do
4     W_n = y_n
5   end for
6
7   repeat
8     for m = 1 : M do                      //Step 1: Check messages
9       for n = 1 : N do
10        R_{m,n} = ∑_{n'∈N_m, n'≠n} (W_{n'} mod 2)
11      end for
12    end for
13
14    for n = 1 : N do                      //Step 2: Bit messages
15      if the messages R_{m,n} disagree with y_n then
16        W_n = (r_n + 1 mod 2)
17      end if
18    end for
19
20    for m = 1 : m do                      //Test
21      L_m = ∑_{n'∈N_m} (W_{n'} mod 2)
22    end for
23    if all L_m = 0 or It = It_max then
24      Finished
25    else
26      It = It + 1
27    end if
28  until Finished
29 end procedure
```

Listing 3.2: Algorithm Bit-flipping Decoding [9]

3.4 Decoding Performance and Limitations

LDPC codes can provide performance close to the Shannon limit, a high parallelizable decoder and low error floor [10]. However, this is only obtainable using long codeword lengths. If the codeword length is small or moderate, or high order modulation is used, the decoding performance decreases. To address that problem, non-binary LDPC codes have been designed for high order GF values and shown great potential. The figure below shows a comparison between LDPC codes of different lengths:

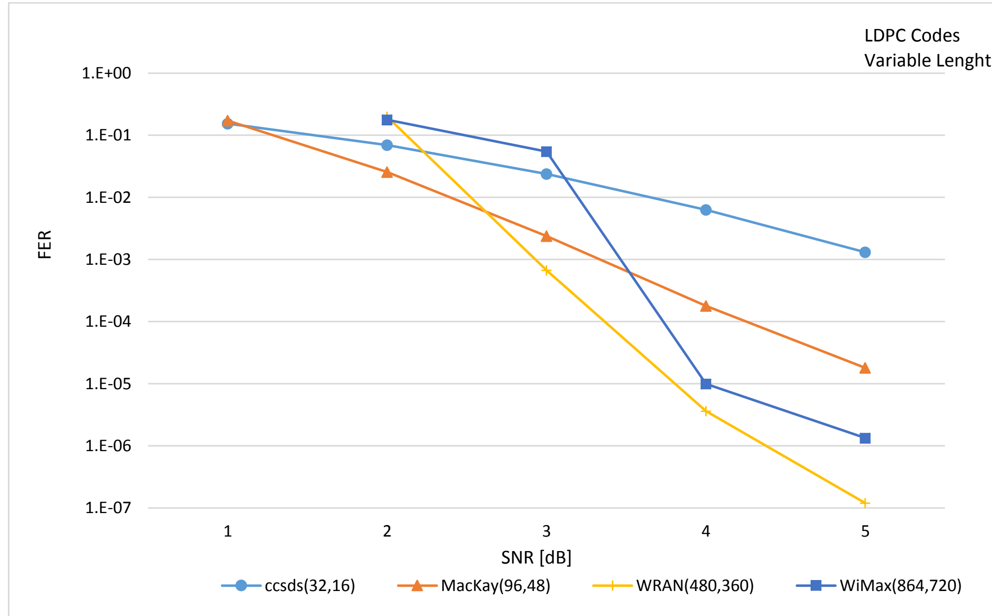

Figure 3.4: Comparison between LDPC codes of variable length

4 Non Binary Low Density Parity Check Codes

4.1 Definition

The extension of Gallagers LDPC codes over $GF(2)$ to codes over $GF(q)$ was first discovered in 1998, by MacKay and Davey [3]. The basic difference from their binary counterpart, is that in non-binary LDPC codes we are now using symbols defined over a q value, with $q = 2^p$, that we also call elements, instead of only having 1s and 0s, that means we are grouping bits to symbols.

GF(64)

011001		25
110001	→	49
000011		3

Figure 4.1: Map bits to symbols over GF(q)

Analysing one edge of our Tanner Graph, with this scheme, we are now sending q probability messages, related to q different elements, instead of only one probability in the binary counterpart. It is clear that the number of necessary calculations, and storage memory size, is way higher than before.

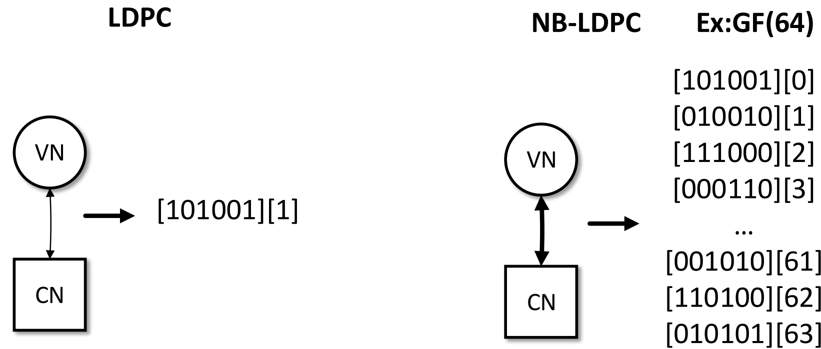

Figure 4.2: Binary x Non Binary Comparison

At every VN n, given that where α_k are the GF elements, i.e. $GF(q) = \alpha_0, \alpha_1, ..., \alpha_{q1}$ and y is the received symbol, we are now calculating for a received symbol a probability for every other symbol:

$$\forall x \in GF(q) : L_n(x) = \ln(\frac{P(c_n = \alpha_k|y)}{P(c_n = \alpha_0|y)}) \tag{4.1}$$

Since the BP check node operation is really a convolution of all his input messages, therefore we have a complexity of $O(q^2)$ in a straight forward implementation. Besides that, all the concepts used in binary LDPC codes are still valid. To define a code we will need: (N, K) length, code rate $R = K/N$, the q field size, and the $H_{m,n}$ parity check matrix.

4.2 Advantages

Besides a growing complexity, NB-LDPC codes have a great performance even for short codes. They are proven to outperform LDPC codes and turbo codes [11]. Using them, we can also have a reduced demapper complexity, because the modulated bits can be directly mapped to symbols (using a matched modulation and field size).

4.3 Decoding

Optimal ML decoding solution is still unpractical with NB-LDPC codes. Their complexity lead us to large area and low throughput hardware implementations. The state-of-the-art techniques used nowadays are based on the BP algorithm, but using sub-optimal techniques, reducing the decoding complexity, in order to achieve practical hardware implementations.
First, an important technique to reduce check node complexity and memory usage will be presented, and then two decoding algorithms: the EMS Algorithm, that is the actual state-of-the-art algorithm used in hardware implementations, and the Syndrom Algorithm, that is a new algorithm focusing on high order fields and possible hardware solutions.

4.3.1 Reducing Complexity - Sub-Optimal Decoding

Considering that we have at a CN, for every connected input, a vector of size q messages. We known that processing all those values is a complex task. A well known method [12] to reduce the used memory and the number of calculations necessary, is to truncate the number of input messages from q to a reduced number n_m, and then also store only n_m values from every input vector.

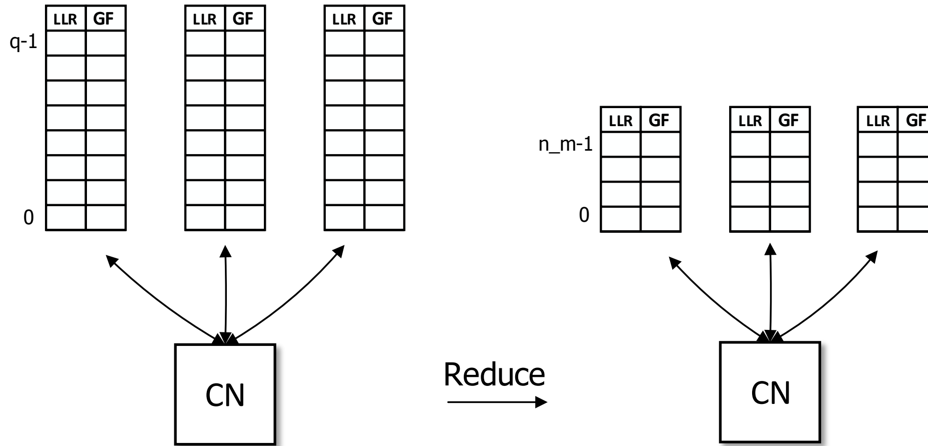

Figure 4.3: Input Reduction

For example if we consider a $GF(64)$ and a d_c equal to 4. In every message output, we will need to find a minima based on q^{d_c-1} possibilities. Selecting an n_m value, for example 13, we reduce the number of possibilities to ${n_m}^{d_c-1}$, example:

$$64^3 = 262144 \tag{4.2}$$

to

$$13^3 = 2197 \tag{4.3}$$

It has been shown that the performance loss regarding the use of this truncation scheme is small or negligible. In the graphic below we may see a comparison of some selected n_m values and their correspondent performance compared with the use of all the q elements:

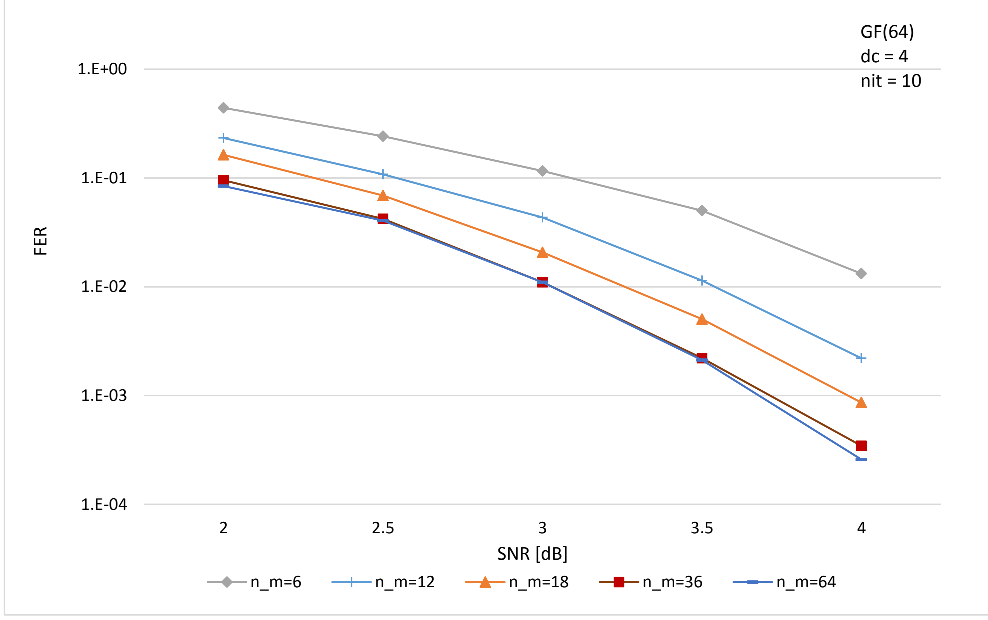

Figure 4.4: Performance comparison using different n_m values

This comparison was done using the EMS algorithm with a forward backward scheme. Analysing the figure we can see that with a n_m equal to 36 we still have almost the same performance of using all the elements. A very used value for the n_m parameter is 12 since the performance loss is less than $0.5dB$ comparing to the full message input. If we reduce it even further, to values lower than that, the performance loss is not negligible and it becomes unworthy. This technique is largely used in NB-LDPC implementations.

4.3.2 Extended Min-Sum Algorithm

The EMS algorithm [13] using the *Forward Backward* (FWBW) scheme is the state-of-the-art algorithm for hardware implementations. It offers a good trade-off between hardware complexity and communications performance. The complexity is still on the check node implementation. We can divide the decoding process in four parts: initialization, VN update, CN update and permutations in *Permutation Nodes* (PN) .

First we have to calculate the LLR symbols for the received codeword. For a received symbol y_n in the VN n, a set of q LLR values must be calculated. We assume that all symbols are equiprobable.

$$\forall x \in GF(q) : L_n(x) = \ln\left(\frac{P(y_n|c_v=x_n)}{P(y_n|c_v=x)}\right) \tag{4.4}$$

with $x_n = \max_{\forall x \in GF(q)} P(y_n|c_n = x)$

This definition use an increasing LLR value to represent a decreasing realiability, being that when the $LLR = 0$ we have the most realiable value. We denote:

- U_{vp} messages from VN to PN
- U_{pc} messages from PN to CN
- V_{cp} messages from CN to PN
- V_{pv} messages from PN to VN.

In the first iteration the VNs forward the instrinsic channel values to the CNs. The VNs outputs are directly the LLR values: $U_{vp}[x] = L_n[x]$. In the next iterations the VN combine the instrinsic values with the d_v incoming sets from the connectes CNs, so the updated extrinsic message U_{vp} are calculated:

$$U_{vp}[x] = L_n(x) + \sum_{t=1,t\neq p} d_v V_{tv}[x], \forall x \in GF(q), p = 1...d_v. \tag{4.5}$$

To achieve the same LLR structure, with the 0 as the most reliable and increasing value for less reliable, a normalization must be applied of the U_{vp} messages, respect to the most reliable symbol, to the output of the VN. The next step is the permutation according to the matrix H, where $h_{c,v}$ is the GF element in the c, v position of the H matrix:

$$U_{pc}[x] = U_{vp}[h_{c,v}^{-1}.x], \forall x \in GF(q), p = 1...d_v. \tag{4.6}$$

Then at the CN update, we process d_c edges from U_{pc}:

$$V_{cp}[x] = \min_{\forall permofx_t} \sum_{t=1,t\neq p} d_c U_{tc}[x_t], \forall x \in GF(q), p = 1...d_c. \tag{4.7}$$

For every GF element x, all possible input permutations that mets the parity check constrain, given by the sum of the according GF elements, must be calculated. After that, the one with the highest reliability is chosen. This CN computation is where the complexity of the decoder relies, given by the high number of possible elements.
The messages sent back to the VNs must be reverse permuted:

$$V_{pv}[x] = V_{cp}[h_{c,v}.x], \forall x \in GF(q), p = 1...d_c. \tag{4.8}$$

Before the next iteration, the decoder checks if the parity check is met based on the most reliable of each VN n:

$$x_n = \min_{x \in GF(q)} (L_n[x] + \sum_{p=1} d_v V_{pv}[x]) \tag{4.9}$$

Below we can see an algorithmic version of the EMS Algorithm:

```
1  procedure Decode(X)
2    It = 0                              //Initialization
3    for n = 1:N do
4      for p = 1:d_v do
5        for x = 1:Q do
6          U_vp[x] = L_n(x)
7        end for
8      end for
9    end for
10
11   repeat
12   for n = 1:N do
13     for p = 1:d_v do
14       for x = 1:Q do                  // Permutation of the VN msgs
15         U_pc[x] = U_vp[h^{-1}_{c,v}.x]
16       end for
17     end for
18   end for
19
20   for m = 1:M do
21     for p = 1:d_c do                  // CN operation
22       for x = 1:Q do
23         V_cp[x] = min_{∀permofx_t} ∑_{t=1,t≠p} d_c U_tc[x_t]
24                                       // Reverse permutation
25         V_pv[x] = V_cp[h_{c,v}.x]
26       end for
27     end for
28   end for
29
30   for n = 1:N do                      // Get most reliable
31     x_n = min_{x∈GF(q)}(L_n[x] + ∑_{p=1} d_v V_pv[x])n
32   end for
33
34   if It = It_max or H.X^T = 0 then
35     Finished
36   else
37      for n = 1:N do
38         for p = 1:d_v do
39           for x = 1:Q do              // Process VN to PN msgs
40             U_vp[x] = L_n(x) + ∑_{t=1,t≠p} d_v V_tv[x]
41           end for
42         end for
43      end for
44      It = It + 1
45   end if;
46  until Finished
47 end procedure
```

Listing 4.1: Extended Min-Sum Algorithm

4.3.3 Syndrome-Based Algorithm

The Syndrome-Based check node algorithm [5] is a first approach for efficient computation of higher-order $GF(q > 16)$. It aims to address the drawbacks of the state-of-the-art algorithms, such as the EMS with the FWBW scheme, which are high latency and low throughput. Up to today is has only been done for lower-order fields, but those dont provide the gain in communications performance that we want to achieve comparing with binary LDPC codes.

In the SYN algorithm, the basic idea is to build syndrome sets based in the inputs LLRs. A syndrome is defined as a sum of one $\{GF(q), LLR\}$ tuple from each input set. In the figure below we can see some syndrome examples:

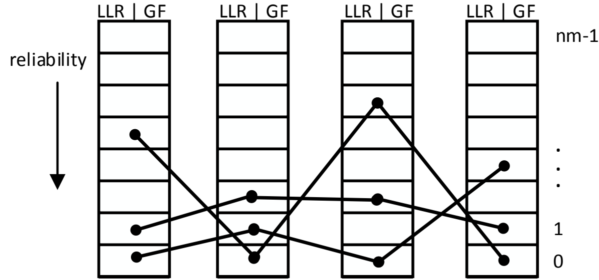

Figure 4.5: Syndrome Algorithm

Every VN input V carries n_m messages. K_{ic} is the set of the most reliable elements from the ith VN connected to the CN c. We calculate the *syndrome* reliability $R(x_1, ..., x_{dc})$ and GF element $E(x_1, ..., x_{dc})$ as:

$$R(x_1, ..., x_{dc}) = \sum_{i=1}^{d_c} V_{ic}[x_i], x_i \in K_{ic} \tag{4.10}$$

$$E(x_1, ..., x_{dc}) = \sum_{i=1}^{d_c} x_i, x_i \in K_{ic} \tag{4.11}$$

One syndrome is then the combination of this two values:

$$SYN(x_1, ..., x_{dc}) = \{R(x_1, ..., x_{dc}), E(x_1, ..., x_{dc})\} \tag{4.12}$$

A syndrome set S contains all syndromes:

$$S = \{SYN(x_1, ..., x_{dc}); \forall x_1, ..., x_{dc} \in K_{tc}\} \tag{4.13}$$

If we analyse the way the syndromes are calculated, we can see that it breaks the basic BP principle, which states that every output must not be correlated with his own input, as we have seen in chapter 3. In that case, an additional step to decorrelate every input and output is needed:

$$R^i(x_1, ..., x_{dc}) = R(x_1, ..., x_{dc}) - V_{ic}[x_i], x_i \in K_{ic} \tag{4.14}$$

$$E^i(x_1, ..., x_{dc}) = E(x_1, ..., x_{dc}) - x_i, x_i \in K_{ic} \tag{4.15}$$

With that we will have a dedicated syndrome set for every output i, that has no correlation with its own input i:

$$SYN^i(x_1, ..., x_{dc}) = \{R^i(x_1, ..., x_{dc}), E^i(x_1, ..., x_{dc})\} \tag{4.16}$$

$$S^i = \{SYN^i(x_1, ..., x_{dc}); \forall x_1, ..., x_{dc} \in K_{tc}\} \tag{4.17}$$

Once that is done, we sort the syndromes by their LLR values and use the n_m most realiable from each set as their output. With the explained algorithm, we may think of an architecture as follows:

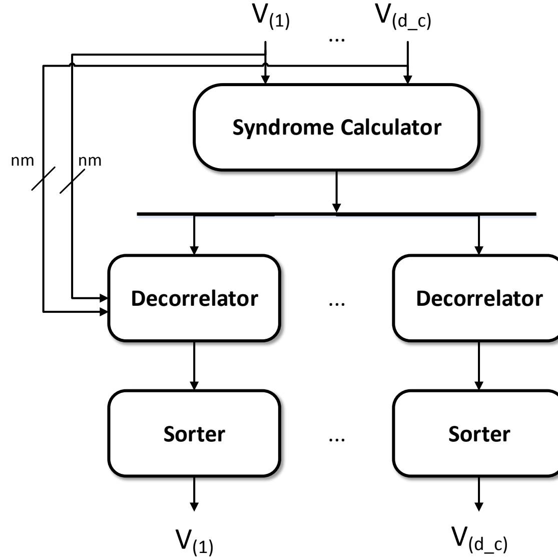

Figure 4.6: SYN CN Architecture

The problems with this presented algorithm without further reductions is that we have too many possible syndroms to calculate, resulting in big sets, and therefore alot of values to sort in order to find the most reliable values. In the next sections, two approaches to reduce the complexity of the algorithm will be presented.

4.3.3.1 Reducing Syndrome Set To compute the output, we use only a part of the calculated syndroms, making all the other calculations redundant. The idea here is to reduce the number of syndromes separating the ones that have a high reliability associated from the others with a low reliability. We build $d_c + 1$ deviation sets D_i and separate the syndrome sets in sub-sets:

$$S = \cup_{i=0}^{d_c} D_i \tag{4.18}$$

In each sub-set, every syndrome may only deviate in exactly i elements from the most reliable element. The subset D_0 will contain only one syndrome, based on all the most reliable elements. In that way, it is easier to access syndroms with higher reliability. After some observations, it has been seen that syndromes belonging to higher deviation sets, such as D_3 and D_4 rarely contribute to the generation of the output. That allows us to limit the number of sub-sets to the ones with low amount of deviations.

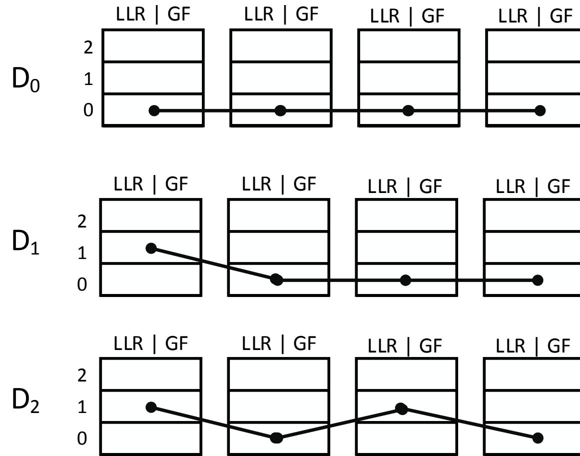

Figure 4.7: D_i Sets - Syndrome examples

Another parameter introduced is a maximum allowed distance d_i from the most reliable element for every D_i set. As we know that the higher position syndromes usually have low reliabilties we limit the sets to lower values. Another restriction is used, the higher the deviation i, the lower the maximum distance permitted from the most reliable element, that is $d_1 \geq d_2 \geq ... \geq d_{dc}$. The size of the D_i can be calculated as follows:

$$|D_i| = \begin{cases} \binom{d_c}{i} .(d_i)^i, & \text{if } d_i \geq 1; i > 0 \\ 1, & \text{if } i = 0 \\ 0 & \text{else} \end{cases} \tag{4.19}$$

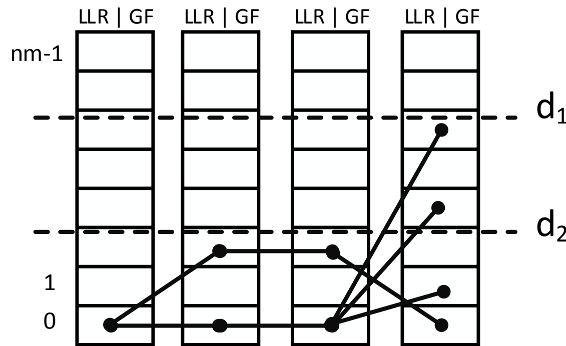

Figure 4.8: Distance technique

Combining both techiniques we are able to calculate only the most reliable syndromes and remove unreliable ones. Using some parametrization, and calculating only D_0, D_1, and D_2 with $d_0 = 0$, $d_1 = n_m - 1$, $d_2 = 2$, $d_c = 4$ and $n_m = 13$ for a $GF(64)$ code; we reduce the size of the set S from 28561 to 73.

For the case that the algorithm functionally is not clear yet, an illustrated example of a single iteration is presented in the appendix. Below we present an algorithmic version of the Syndrome algorithm:

```
1  procedure Decode(X)
2    It = 0                                  //Initialization
3    for n = 1:N do
4      for m = 1:d_v do
5        for x = 1:n_m do  // best n_m symbols
6          U_nm[x] = L_n(x)
7        end for
8      end for
9    end for
10
11   repeat
12     for m = 1:M do
13      for i = 1:d_c do                      // CN operation.
14           R^i_m(x_1,...,x_dc) = sum_{n=1}^{d_c} U_nm[x_i] - U_nm[x_i], x_i ∈ K_nm
15           E^i_m(x_1,...,x_dc) = sum_{n=1}^{d_c} x_i - x_i, x_i ∈ K_nm
16           S^i_m = {R^i_m, E^i_m}
17           Sort(S^i_m)
18           V_mi = best n_m of S^i_m
19      end for
20     end for
21
22     for n = 1:N do                         // Get most reliable
23      x_n = min_{x∈GF(q)}(L_n[x] + sum_{p=1} d_v V_pv[x])n
24     end for
25
26     if It = It_max or H.X^T = 0 then
27      Finished
28     else
29       for n = 1:N do
30          for m = 1:d_v do
31            for x = 1:n_m do              // VN operation
32              U_nm[x] = L_n(x) + sum_{t=1,t≠m} d_v V_mn[x]
33            end for
34          end for
35       end for
36       It = It + 1
37     end if;
38   until Finished
39 end procedure
```

Listing 4.2: Syndrome Algorithm

5 Hardware Architecture

Efficient hardware design is always a trade-off between communication performance, flexibility, area and power consumption. Considering a NB-LDPC decoder, the check node function accounts for the largest part of the decoders complexity.

5.1 Existing Solutions Analysis

Before the presentation of the hardware, we make an introduction to the pros and cons of the existing non-binary decoding solutions up to today:

- EMS Decoder: [13]
Advantages:
Good performance.
Performs even better than the BP decoder in the error floor region.

Disadvantages:
Bottleneck of the decoder complexity is the check node update, complex implementation.

- Min-Max Decoder: [14]
Advantages:
Throughput is improved compared to the full-complexity Non-Binary Min-Sum decoder
Possible to perform a block implementation

Disadvantages:
Throughput is only slightly better than EMS decoder.
Looses more performance than the EMS decoder in the waterfall region.
Lack on quantization robustness (good performance with fixed quantization).

- Symbol Flipping Decoder: [15]
Advantages:
Increadibly low complexity hard decoding.
Requires minimal storage for the message values.
Could be an alternative to hard decoding of Reed-Solomon codes.

Disadvantages:
Large performance loss in the waterfall region : 1.5 dB to 2.5 dB for very small block-lengths.
More research is needed to further improvement.

- Stochastic Decoder: [16]
Advantages:
Simple decoder.
Can achieve good performance.

Disadvantages:
Unpractical because of very high latency.
No hardware implementation proposed yet.

None of those algorithms scale well with a growing q for a fixed complexity. How to achieve that with a good performance is the *Holy Grail* of Non-Binary Decoding.

We need to find a way to reduce the check node computation complexity, maintaining the state-of-the-art performance, and achieving a reduce area architecture.

First the techniques and decisions used to reduce the hardware complexity will be presented, and in further sections, a detailed hardware overview of how every module was designed is given.

5.2 Optimizing Area

Analysing the architecture based on the SYN CN algorithm, some problems might be noticed. At the syndrome calculator module, we have a high register use to store all the calculated syndromes. Before updating the outputs we have many sorters, one for each connected VN, and the size of every one of those sorters is big, based on the number of syndromes we are calculating and getting the n_m best of them for each connected VN.

To reduce the number of sorters and registers, a scheme that uses only one sorter at the input and then calculate syndromes only for a n_s number of values can be used. The idea is shown in the picture below:

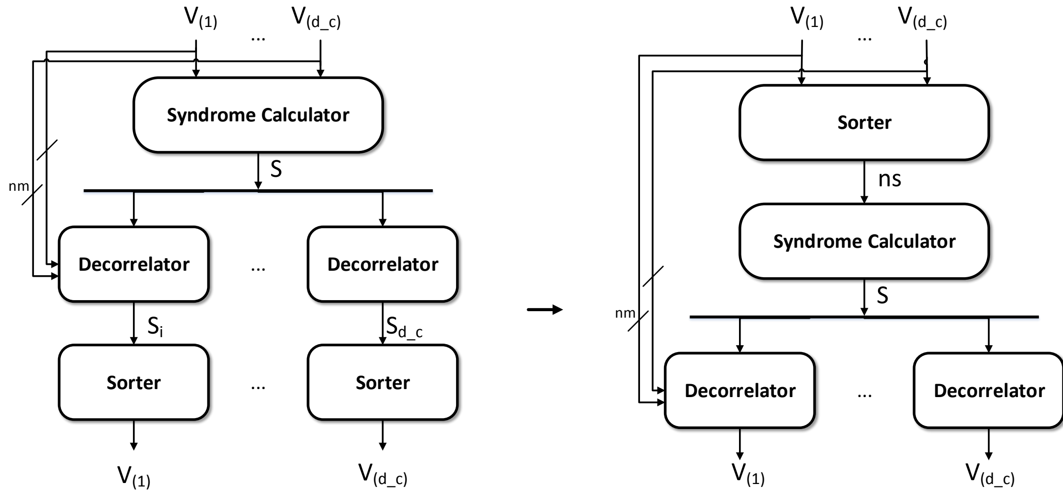

Figure 5.1: Moving the sorter

With that scheme the number of sorters is reduced to one, and the size of the syndrome calculator module is also reduced, because now it will have to calculate and store syndromes only based on already sorted LLR values.

However, if we analyse the decoder as a whole, we may see that an exact sorting is not needed, because once the VN receive all the *a posteriori* messages from the CNs, it will need to resort all of them anyway. Therefore we adapt the sorter to a sub-optimal sorting.
Based on this analisis, to reduce the size of this sorter module even further, we use a probe technique, distributing probes over the CN inputs and then sorting values based

on those probes. For a given input of n_m size, we distribute a certain number of probes over the input messages. With those probes, we build sets of messages, grouping the neightboring values to every probe. We sort all those sets using only the LLR value of the probe associated to the set.

If we reduce the parallelization of this sorter here, we can reduce the size even more simply by sorting only a single group from every VN input set on every clock cycle, as shown below. Combined with the probe scheme, we are able to reduce the sorting size from an initial $n_m.d_c$ to only d_c. In the picture, the grey blocks are the selected probes, and we can see that a group from every connected VN is analysed in every clock cycle.

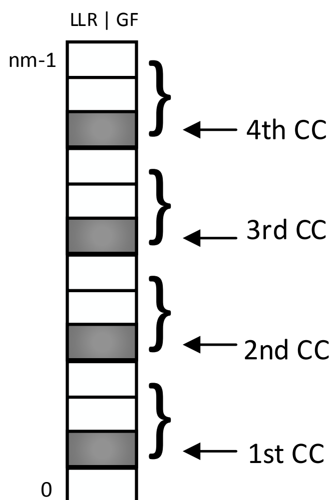

Figure 5.2: Probe scheme

In this stage the syndrome calculator is reduced, but it still calculate syndromes in parallel. One way to reduce the area of this, is to reduce the sorter input to only one message, sequentializing the syndrome calculator module. Here we have the throughput/area trade-off, but since we are optimizing the area, and we were still able to achieve a good throughput (see in the results section), it is worth making this change.

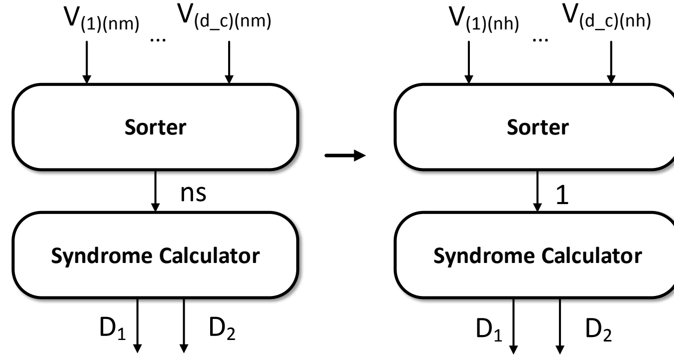

Figure 5.3: Sorter output reduce

Since we are now sequentializing the processment, we will have a D_1 value processed in every clock cycle. The D_2 values in counterpart, depend on two sorted values. We can not know in advance the actuall sorting order of D_2. For the case when a D_1 and a D_2 are calculated in parallel, we need an additional sorter at the output to get the best of those values first.

Analysing D_1 syndromes beeing calculated based on the same VN input m, it can be noticed that those calculated syndromes share the same values from the other inputs (the most reliable) and are beeing calculated again for every syndrome.

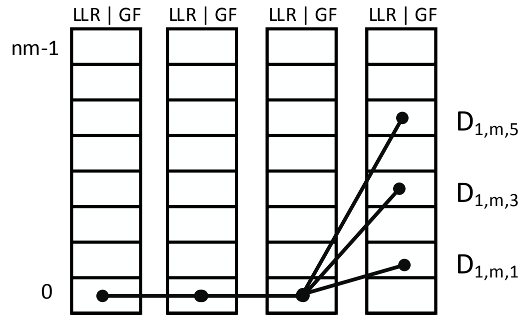

Figure 5.4: Common Syndromes

To remove those redundant calculations, we pre calculate them in a newly added module, and store this values to be used in the syndrome module. We pre-calculate d_c values for every D_1 set, and $\frac{d_c!}{(d_c-2)!.2}$ for all the D_2 possibilities. In that way we can reduce a little further the size of the syndrome calculator module.

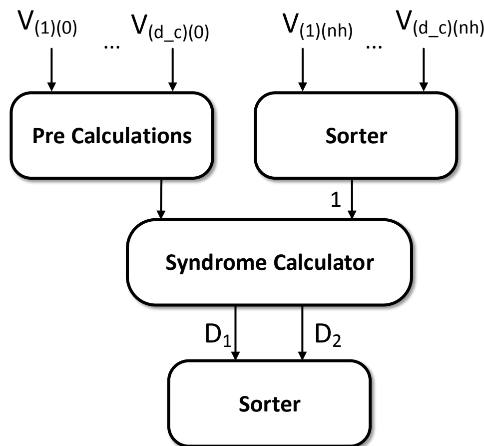

Figure 5.5: Pre Calculations Module

Combining that last change together with what we had, and the additional sorting at the output, we now have a new hardware architecture. If we analyse it, we can see that now we have two small sorters, comparing with the first architecture, a low area syndrome calculator with reduced register use, and reduced wiring needed to the decorrelator modules, because of the consideration of values based only in the most realiable one. With that we have addressed and solved all the problems we had in the first architecture.

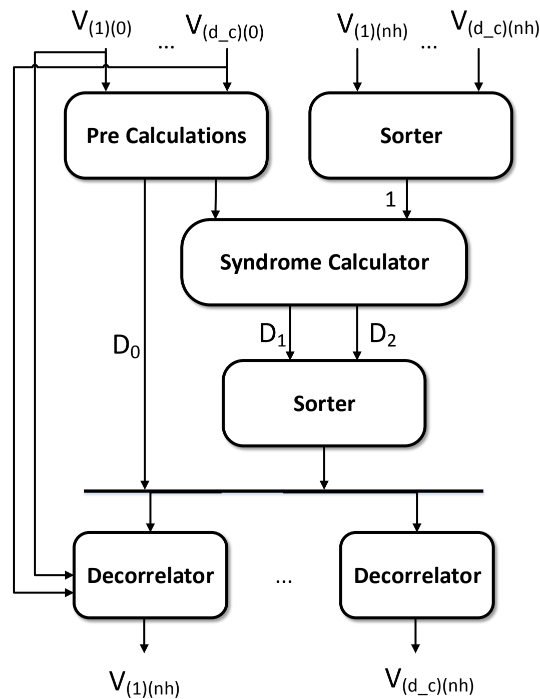

Figure 5.6: Final Hardware

Now we give a more detailed view about the tools used and how the simulation and implementation process were done.

5.3 CSE Simulation Environment

The *Creonic Simulation Environment* (CSE) [17] is a communication chain simulation environment developed by Matthias Alles and Timo Lehnigk-Emden. It was developed using a C++ programming language, for maximum portability and executing speed, along with the ITPP library for mathematical calculations. Basically, it simulates a communication chain and uses the object orientation paradigm to configurate it, letting the user create new modules, that are encapsulated classes, for every part of the chain. Every module is configured using a XML file, making it easier to make various simulations without the need to recompile the code. After a full iteration is done, the Statistic Error Rate module compares the input and the output files and generates a detailed XML file with the simulation results.

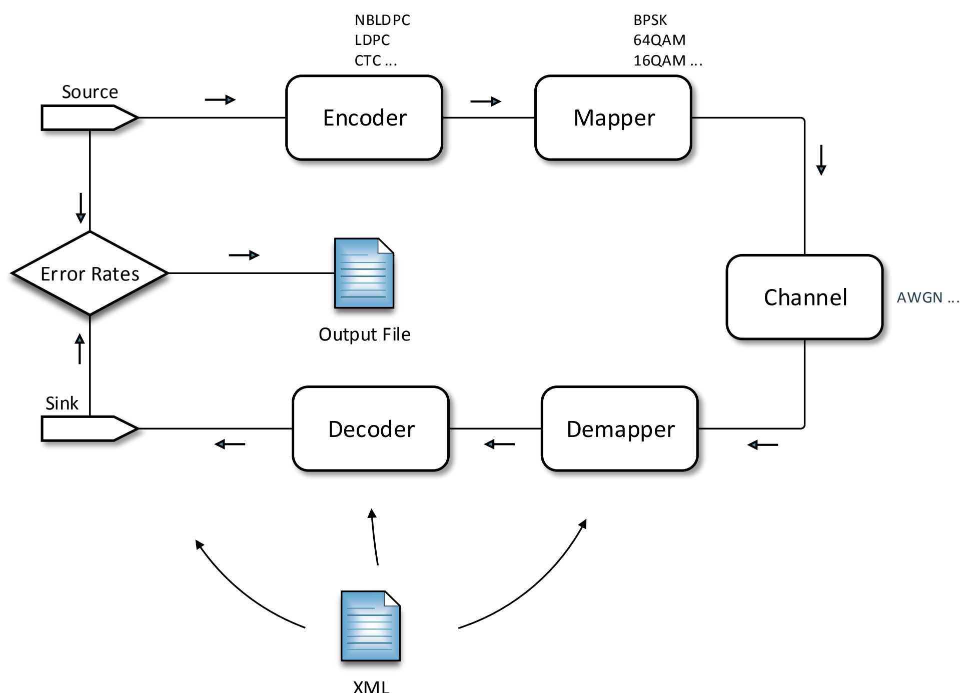

Figure 5.7: CSE Communication Chain

Accordingly, the first process before initiating a real hardware implementation was to develop a NB-LDPC decoder module for the CSE, based on the SYN algorithm. All ideas have been checked first on the software model before being implemented in hardware. Along with the SYN algorithm decoder module, we had already coded decoders based on other algorithms, to make a fair comparison between the proposed implementation and the already well-known state-of-the-art algorithms.

5.4 Hardware overview

The hardware implementation was done using the VHDL language and the Xilinx ISE Design Suite tools. In the figure below we present an overview of the hardware implementation based on the architecture described. The pipeline registers, the size of every bus used in the architecture and also the size of the inputs and outputs can be noticed

in the figure.

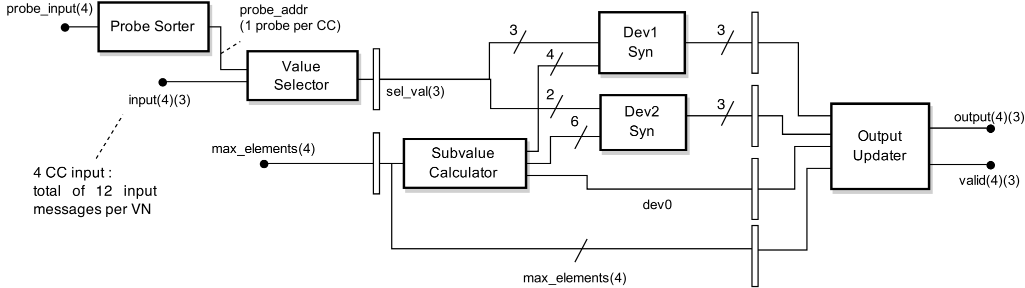

Figure 5.8: Hardware Overview

Before further explanation, we may first define the parameters used for this implementation, as we have optimized it to those. They are:

Code Parameters:
- $GF(64)$
- $d_c = 4$
- $d_v = 2$
- $n_m = 13$
- $it_{max} = 10$

Implementation and Syndrome Parameters:
- input/output of 3 elements per clock cycle
- best 6 probes sorted, with a sequential output
- 18 D_1 values
- 3 D_2 values

In the figure 5.8, 3 inputs and 2 outputs can be seen:
- the initial $n_m = 13$ value is divided here. We get the most reliable elements from every VN in a separated *max_elements* vector and the other values at the *input* vector. We consider an already sorted input, so in this other vector the values are already sorted based on the LLRs.
- the decision to use only an input of three messages per VN per clock cycle was based on a decision already presented in the Optimizing Area section.
- the *probe_input* vector is only a separated signal derived from the input, it is not part of the decoder input, for the sake of reducing size at the sorter and then passing to it only one probe from every VN.
- matching the input parallelization, at the output we also have three messages for every VN.
- additional to this messages, we also have a valid bit to every one of them, in order to signalize to every VN if a message is valid or not.

The basic functionally of the hardware goes as described:
In the first clock cycle, we get the 3 first input messages from every VN at the check node input. A separated input signal for the most realiable message of every VN was created.In the next three clock cycles, we do the same thing, until we receive all the messages. Every related probe from this messages will go to the *Probe Sorter* module and all the messages will be stored at the *Value Selector* module. Once the *Probe Sorter* sorts a probe, the address of this probe is passed to the *Value Selector* and then the messages related to the sorted probe goes further in the chain. The *Subvalue Calculator* module was shifted to the right just to reduce the number of pipeline registers needed. With the sub-calculations made by this module and the sorted messages, the modules *Dev1 Syn* and *Dev2 Syn* will be able to calculate the syndrome values. After that, those

values are passed to the *Output Updater* module, where they are re-sorted and validated based on which input has generated this messages (BP principle).

To optimize how many calculated syndromes should be used at the output, alot of testing were done. We do not want to have an excess of calculations, because that would lead to a huge hardware, but we also do not want to lose performance, due to a reduced number of calculations. The best trade-off we found was to use the best 6 probes, out of the 16 (remember, we are using 4 probes distributed over 4 inputs), and with those we build the 6 possible D_1 syndromes and 4 D_2 syndromes. Considering the 3 message output parallelization, chosen due to other optimizations, one may think of using all those messages resulting in a total of 10 clock cycles output. Instead of doing that, after analysing the contribution of the messages in the late clock cycles, we limit the output to 7 clock cycles, since after simulations it has been seen that the last 3 messages dont contribute that much. That results in a total of 21 syndromes.

5.4.1 Probe Sorter

To clarify the ideas behind this module we may first consider the full input set. Since we had setted $n_m = 13$ and $d_c = 4$ over a $GF(64)$, we have then, a total of $4 * 12 = 52$ inputs at the check node. Using the already presented probe scheme, we found that grouping the messages 3 to 3 was a good trade-off, between size of the sorter and resulting performance. That leads us to 4 probes per VN input, and a total of 16 probes to be sorted. Just to remember, we consider the VN inputs at the CN as sorted, then the first message from every VN is the most reliable element, which do not need to be sorted.

However, sorting 16 numbers at once result in an undesired sorter size. An analysis of the Batcher Odd-Even[18] sorting algorithm for sorting networks lead us to networks with depth $O(\log^2 n)$ and size $O(n.\log^2 n))$, and the following table:

n	Depth	Length
1	0	0
2	1	1
3	3	3
4	3	5
5	5	9
6	5	12
7	6	16
8	6	19
9	7	25
10	7	29
11	8	35
12	8	39
13	9	45
14	9	51
15	9	56
16	9	60

Table 5.1: Sorting complexity

After the hardware considerations, regarding number of registers through the sorting stages, number of comparators and total LUTs generated for each sorter size, we decided to use a $n = 4$ and then use 4 clock cycles to read all the inputs. With that, the wiring and routing at the check node input and outputs was also reduced, because instead of having 64 inputs, we are now using 12 instead. After the sorting initiation, we have a one clock cycle delay, and after that we output 1 sorted probe per clock cycle, until the total number of probes are sorted. Another implementation reducing the sorting even further and using 2 clock cycles per sorted probe was investigated, however, the throughput loss regarding that implementation was undesirable, so we decided not to use it.

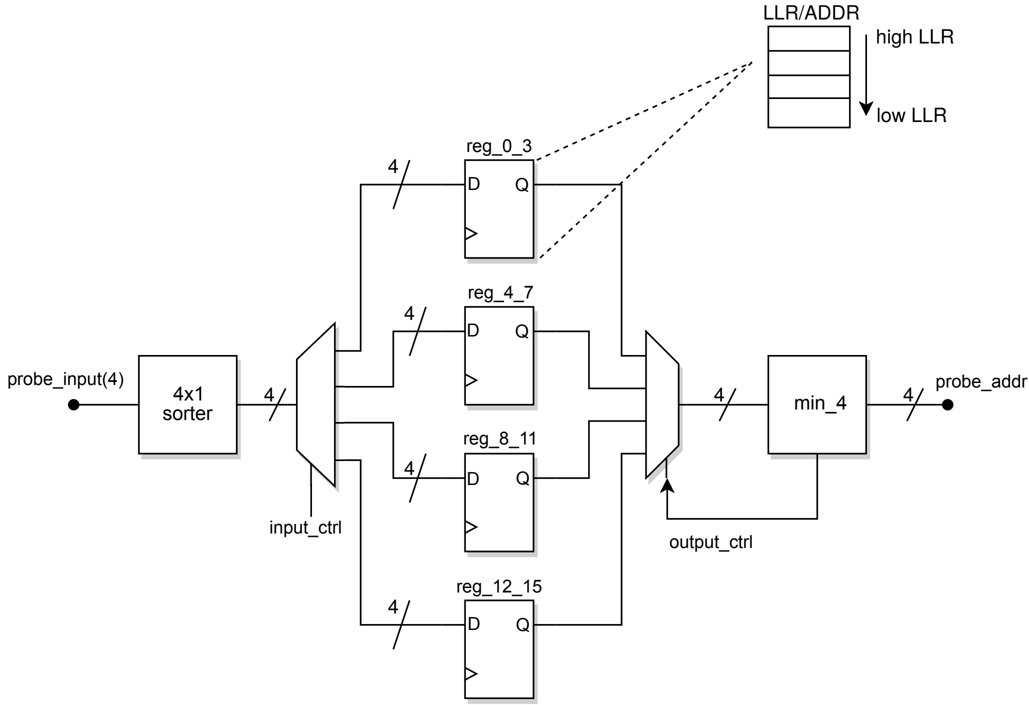

Figure 5.9: Probe Sorter

As shown in the figure 5.9 the 4 input messages are sorted, and then stored accordingly, from the lowest LLR (most reliable message) to the highest (less reliable). One message from every register is taken and then the most reliable from those is selected to the output. We must do this because we cannot garantee that the first probes from every VN are better than the other probes from the other VNs, it is usual that one VN can have real good messages and other real bad messages. At the output we have the address of the sorted probe, which will be sent to the *Value Selector* module.

5.4.2 Value Selector

The *Value Selector* module basically contains many registers to store all the input messages (LLRs and GF elements) and a selector, to output the desired messages related to the sorted probe address, coming from the *Probe Sorter* module. After 4 clock cycles all the registers received the input messages, and they remain there until the next decoding iteration starts. The probe address coming from the *Probe Sorter* module have two

fields: one says to which input the probe belongs, and the other which probe is that, from the 4 probes of this particular input. Once that selection is done, the three input messages related to this probe address are sent to the output.

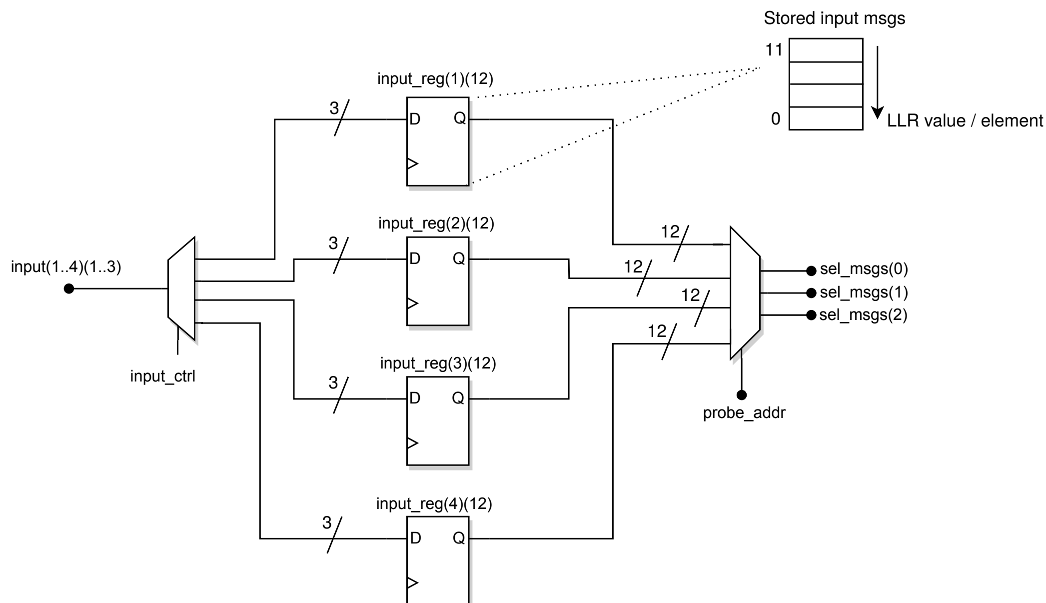

Figure 5.10: Value Selector

5.4.3 Subcalculations Module

As already explained, we have some syndromes that share the same calculations, therefore this module have been designed. It calculates subvalues for all possible combinations of deviations that we can have, based on the $d_c = 4$ input. That is 4 D_1 combinations, one for every input, and 6 D_2 combinations, using the inputs combined 2 to 2. This calculations are only based on the GF element of every message and are made using XOR combinations.

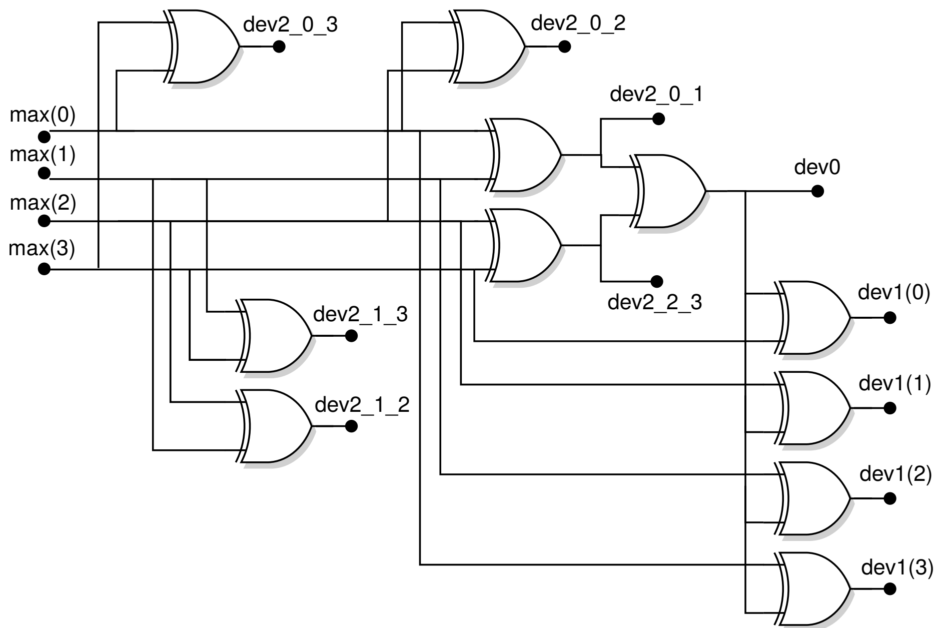

Figure 5.11: Subcalculations module

5.4.4 Dev1 Syndrom and Dev2 Syndrom

Those two modules calculate the syndromes, using the sorted messages and the subcalculations performed.

The *Dev1 Syndrome* module selects the subcalculated value to be used, based on which input the messages came from, and then with this value calculates the syndromes. This is the calculation for the GF element of each syndrome, the LLR values calculation however are not needed, because every D_1 syndrome is based in only one message deviating from the most reliable values ($LLR = 0$), so the LLR of this message is used directly.

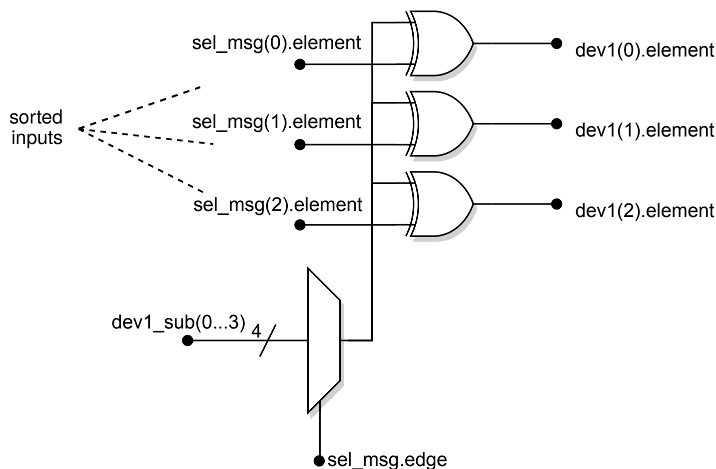

Figure 5.12: Dev1 Syndrome

With the D_2 syndromes, it is a little different, because every D_2 is based on two values that deviates from the most reliable ones. That means we need two sorted probes, before we can do any calculation. It can also be the case that the sorted probes are from the same input. If that is the case, a D_2 value calculated base on those probes is not valid, and must be not considered in the decoding. Having that in mind, we see that we can not be sure when will we have a valid D_2 to be outputed, if we have any, because it can be also the case that all the 4 considered probes come from the same input. Regardless of that, what we do is store the 4 considered probes, and output every combination of them, using valid bits to signalize the *Output Updater* when the D_2 value is valid or not. A smart way to do that is just use a XOR with the source address from both values, resulting in 0 in case they are from the same source.

As in the D_1 module, the subcalculations values are selected based on the source of every message (note that we have 2 different sources), and then the element calculations are done combining the subcalculation with the registered inputs. In this module we also need three adders, to calculate the resulting LLR value for every combination.

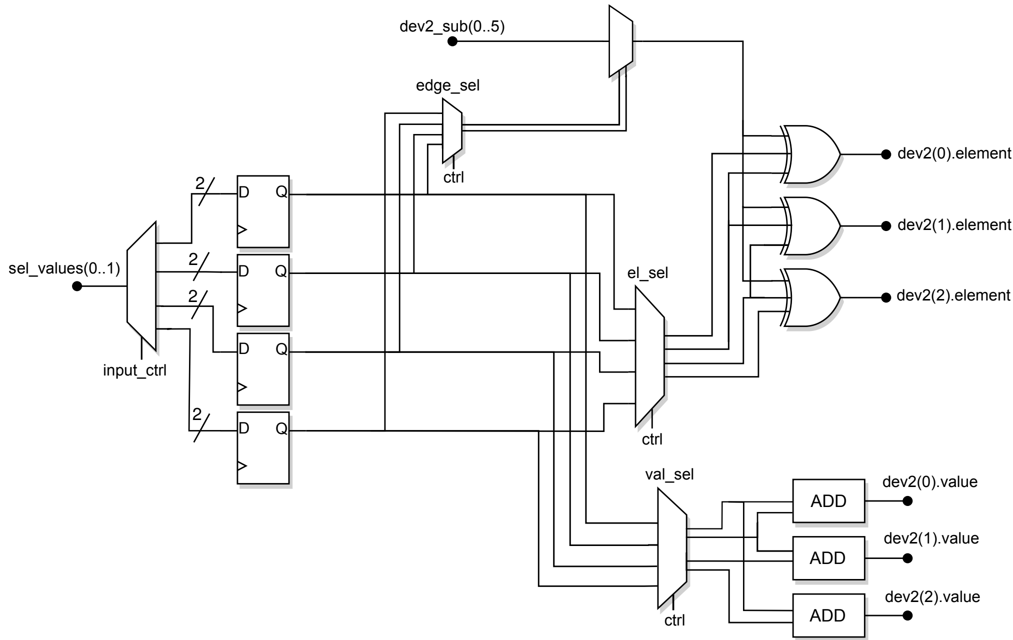

Figure 5.13: Dev2 Syndrome

5.4.5 Output Updater

The basic idea of the *Output Updater* is to check if the incoming syndromes are valid for each connected VN, decorrelate them regarding the BP principle, and output them accordingly. One may see that we can have D_1 and D_2 messages comming in parallel, so since we use only a 3 messages output parallelization it would be as simple as getting the best out of those 6 incoming messages. The problem, as seen in the *Dev2 Syndrome* module, is that we are not sure in which clock cycle will we have a valid D_2.

So the picture is: we have a D_1 every clock cycle, and we *may* also have D_2. Accordingly, this scheme was used: in case we have a valid D_2 at the input, we sort both D_1 and D_2, totalizing 6 syndromes, the best 3 of those we use as output in this clock cycle, the

other 3 we store in a register and sort with the next incoming messages. In case we have at two sequential clock cycles both valid D_1 and D_2 inputs, the D_1 is selected to be sorted with the rest at the second clock cycle, because it is probably more reliable than the D_2. Remember that we have to aim for the most reliable syndromes. Once we have at least one sorted value, the rest of this sorting will be always be resorted with the incoming values in the following clock cycles, to keep the most reliable messages being sequentially outputed.

Before outputting the syndromes, we must also decorrelate them, to keep the uncorrelated output, regarding the belief propagation principle. It is done by using the most reliable values from the particular VN as input of a XOR with the incoming messages. A message is valid for a particular VN if this VN has not contributed for its generation. To set the valid bits, we use the source bit of a syndrome as input of an inverter.

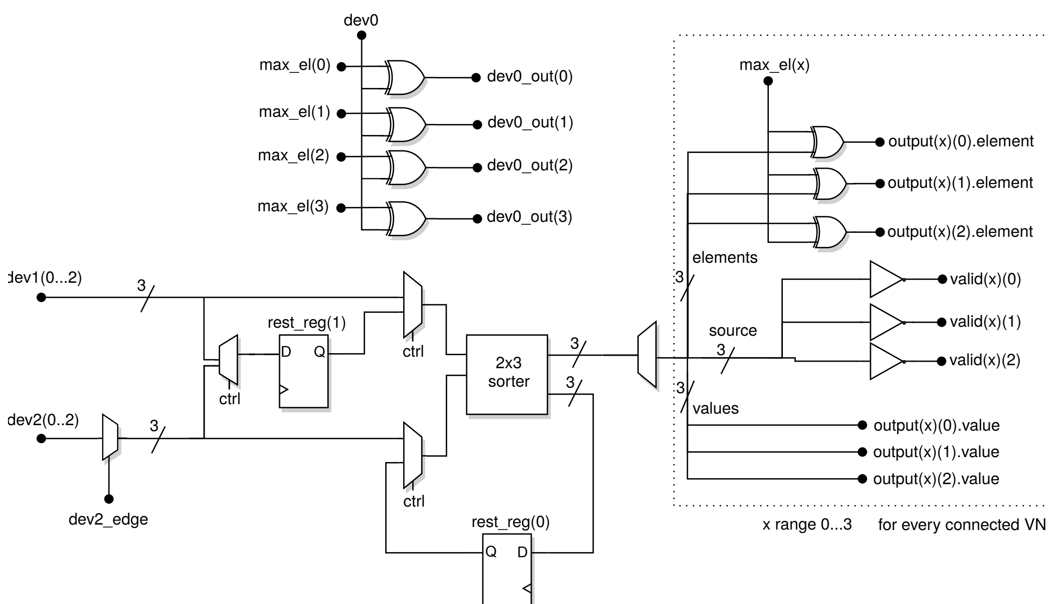

Figure 5.14: Output Updater

6 Results

In this section, we analyse the implementation results of the check node architecture detailed in the previous section and compare it to the state-of-the-art architectures used up to today. All the synthesis results presented here were made using a Xilinx Virtex 5 FPGA device XC5VLX50T with speed grade -3. All the results were obtained after Place and Route.

6.1 Communications Performance

In the figure below we analyse the communications performance of the developed check node compared to the state-of-the-art FWBW CN. To a fair comparison, the code parameters are exactly the same. As observed, for a signal-to-noise ratio from 2 to 4.5, we were able to achieve the same performance of the forward backward scheme. For a SNR of 5 and 5.5 we were able to achieve an even better performance. After this SNR range, the algorithms are in the so called error-floor range, so higher SNR values were not considered. The y axis of the figure refers to *Frame Error Rate* (FER)

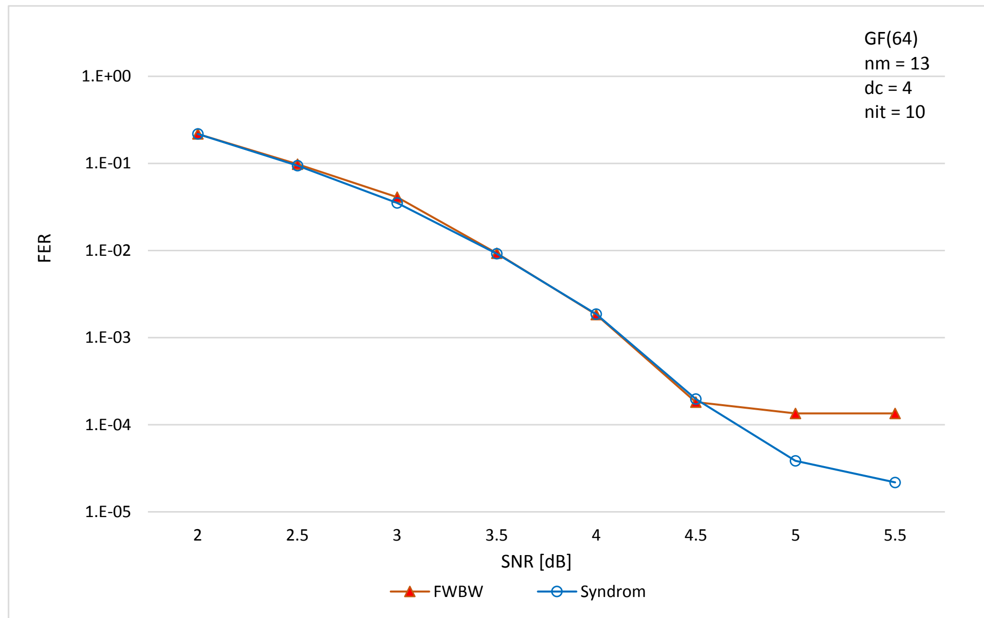

Figure 6.1: Communications Performance

6.2 Area Analysis

Regarding the area comparison, it must be said that the same number of quantization bits was the same in both implementations (6 bits for LLR values and 6 bits for element representation). The syndrome implementation uses 41% of the register area and 48% of the *Lookup Table* (LUT) area, comparing to the FWBW implementation.

	FWBW	Syndrome
Registers	2426	1002
LUTs	1872	898
Total area	4298	1900

Table 6.1: Area comparison

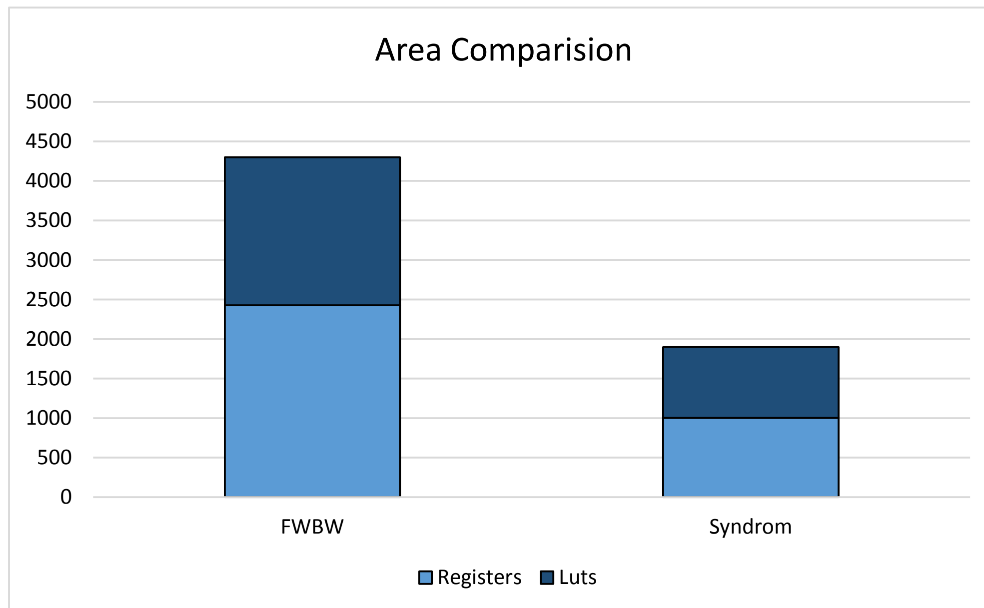

Figure 6.2: Area comparison

6.3 Throughput Analysis

In the throughput analysis, we first consider the raw element throughput, and second, the throughput per area unit achieved. In the first, our architecture surpasses the FWBW architecture in more than 6 times, and in the latter, the number goes over 14 times.

	FWBW	Syndrome
Frequency (Mhz)	123.183	259.336
Throughput (Mel/s)	123.1	778.68
Throughput/area (Kel/s/unit)	28.64	409.8

Table 6.2: Throughput comparison

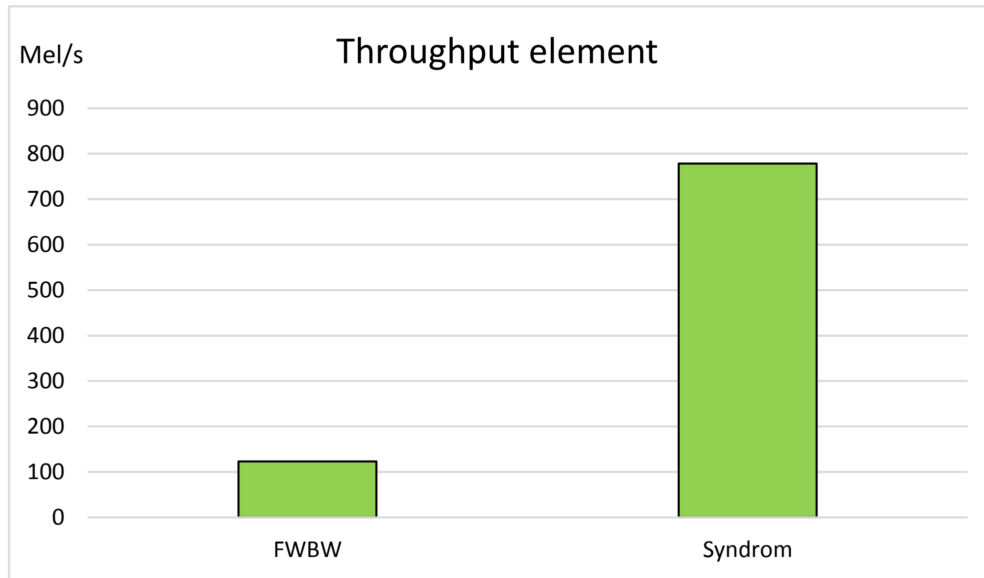

Figure 6.3: Throughput Comparison

6.4 Scaling the architecture

Last, but not least, another architecture has been develop, to check how the presented syndrome architecture scales in terms of area and throghput. In the results we found out that we can have an almost linear increase in throughput and area.

	Syndrome	Scaled Syndrome
Registers	1002	2200
Luts	898	2077
Frequency (Mhz)	259.336	263.351
Throughput (Mel/s)	778.68	1592
Throughput/area (Kel/s/unit)	409.8	371.2
Latency (CC)	12	6

Table 6.3: Sorting complexity

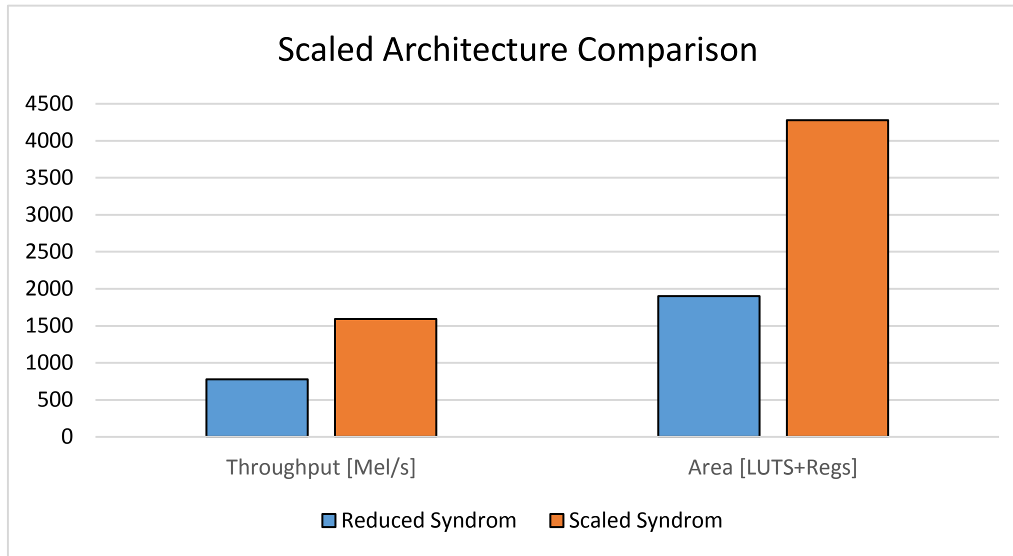

Figure 6.4: Scaled architecture

6.5 Final results

In the table, a brief overview of the obtained results is shown. We were able to develop a check node architecture with greatly reduced area, mantainig the good performance, and achieving a way higher throughput than the actual state-of-the-art architecture.

	EMS-FWBW	Syndrome	Scaled Syndrome
Registers	2426	1002	2200
Luts	1872	898	2077
Frequency (Mhz)	123.183	259.336	263.351
Throughput (Mel/s)	123.1	778.68	1592
Throughput/area (Kel/s/unit)	28.64	409.8	371.2
Latency (CC)	21	12	6

Table 6.4: Final results

7 Conclusion

In this thesis, a new non-binary LDPC check node hardware architecture has been presented. It is based on the SYN CN Algorithm, which has been shown to provide efficient hardware implementations, and is one alternative to the commonly used FWBW approach. This is the first approach for efficient parallel computation for a code based on a higher order field $q > 64$. It is area efficient, has a high throughput and low latency.

In a direct comparison with the FWBW scheme, the presented hardware has less than half the area, with over six times the throughput of the FWBW. Regarding the communications performance, our hardware deliver state-of-the-art performance, and for particular signal-to-noise ratio values, it can even exceed it.

For the presented architecture, the used parameters were chosen wisely, based on the code parameters. If any changes in the code parameters must be done, algorithmic and hardware modifications might be needed to achieve the same results presented in this thesis.

7.1 Future Work

Having in mind the upcoming technologies, higher modulation will have to be used in future communication schemes. Therefore it would be interesting to extend this idea and investigate a check node implementation considering even higher order fields, $q = 256$ for example.

As it has been said, this was a first approach to efficient high-order field computation, the performance of the syndrome algorithm must be further investigated for growing modulation and data rates.

Up to today, no algorithm that scale well with a growing q number for a fixed complexity was presented. A complexity grow in $\log q$ instead of q is what is desired. How to achieve that, keeping the state-of-the-art performance, is the *Holy-Grail of Non-Binary Decoding.*

8 Appendix

8.1 Galois Field Arithmetic

Finite field arithmetic is widely used in a variety of applications such as linear block codes and cryptography algorithms. It differs from standard arithmetic for the fact that there is a limited number of elements in that field, and every operation performed over it, result in an element within. The number of elements that limits those fields, must be given in the form p^n, where p is a prime number, and n is a positive integer. The integer n defines the dimension of the field, and the prime p the characteristic.

The finite field p^n is also called Galois Field - $GF(p^n)$, in honor to the founder of finite field theory, variste Galois. We are particular interested in the family of binary fields with size 2^m, due to the binary representation of numberic values used in our applications. Here we may see that for the case of a $GF(2)$, the addition will result in an exclusive OR (XOR) operation and the multiplication in an AND operation.

Ideas from this section are based on the work [17].

8.1.1 Definition

A polynomial $p(x)$ with grade $grad\, p(x)$ is coefficients out of a prime field $GF(p)$ is called irreductible over $GF(p)$ if it can not be constructed by multiplication of polynomials with lower grade also having only coefficients out of $GF(p)$.

We define an extension field $GF(p^m)$ over the prime field $GF(p)$: assume an irreductible polynomial $p(x)$, with $grad\ p(x) = m$, over $GF(p)$. Each of the $p^m - 1$ lower graded polynomials with coefficients out of $GF(p)$ has an inverse polynomial. Therefore the set of all these $p^m - 1$ polynomials inlcuding the zero element, with the two operations of addition and multiplication span a field $GF(p^m)$ containing p^m elements. For the sake of our applications we use $p = 2$, $GF(2^m)$.

8.1.2 Construction

An element α out of the field p^m is called root of p(x) if $p(\alpha) = 0$. The polynomial $p(x) = p_0 + p_1.x + ... + p_m.x^m$ is called a primitive polynomial if a element α is root of $p(x)$.

For each field $GF(p)$ and each number $m \in N$ at least one primitive polynomial $p(x) = p_0 + p_1.x + ... + p_m.x^m$ exists. If more than one primitive polynome exists, they span the same extension field, so knowing only one of them is enough.

Assuming $GF(2^4)$ and the primitive polynomial $p(x) = x^4 + x + 1$ with primitive element α:

$$p(\alpha) = \alpha^4 + \alpha + 1 := 0 => \alpha^4 = \alpha + 1$$

m	Primitive Polynomial
1	$x+1$
2	x^2+x+1
3	x^3+x+1
4	x^4+x+1
5	x^5+x^2+1
6	x^6+x+1
7	x^7+x+1
8	$x^8+x^6+x^5+x^4+1$

Table 8.1: Primitive polynomials for $GF(2^m)$

Step	Power of α	Corresponding Polynomial
1	$-\infty$	0
2	0	1
3	1	α
4	2	α^2
5	3	α^3
6	4	$\alpha^4 \ mod \ (\alpha^4+\alpha+1) = \alpha+1$
7	5	$\alpha^5 \ mod \ (\alpha^4+\alpha+1) = \alpha^2+\alpha$
8	6	$\alpha^6 \ mod \ (\alpha^4+\alpha+1) = \alpha^3+\alpha^2$
9	7	$\alpha^7 \ mod \ (\alpha^4+\alpha+1) = \alpha^4+\alpha^3 \ mod \ (\alpha^4+\alpha+1) = \alpha^3+\alpha+1$
10	8	$\alpha^8 \ mod \ (\alpha^4+\alpha+1) = \alpha^4+\alpha^2+\alpha \ mod \ (\alpha^4+\alpha+1) = \alpha^2+1$
11	9	$\alpha^9 \ mod \ (\alpha^4+\alpha+1) = \alpha^3+\alpha$
12	10	$\alpha^10 \ mod \ (\alpha^4+\alpha+1) = \alpha^4+\alpha^2 \ mod \ (\alpha^4+\alpha+1) = \alpha^2+\alpha+1$
13	11	$\alpha^11 \ mod \ (\alpha^4+\alpha+1) = \alpha^3+\alpha^2+\alpha$
14	12	$\alpha^12 \ mod \ (\alpha^4+\alpha+1) = \alpha^4+\alpha^3+\alpha^2 \ mod \ (\alpha^4+\alpha+1) = \alpha^3+\alpha^2+\alpha+1$
15	13	$\alpha^13 \ mod \ (\alpha^4+\alpha+1) = \alpha^4+\alpha^3+\alpha^2+\alpha \ mod \ (\alpha^4+\alpha+1) = \alpha^3+\alpha^2+1$
16	14	$\alpha^14 \ mod \ (\alpha^4+\alpha+1) = \alpha^4+\alpha^3+\alpha \ mod \ (\alpha^4+\alpha+1) = \alpha^3+1$

Table 8.2: Generating all elements for $GF(2^4)$

8.1.3 Representation

There are four different ways to represent such extension field elements: exponential, polynomial, components and decimal values.

Exponential representation: all elements except the zero element of an extension field can be generated by exponentiation of the primitive element α. If the primitive element α and the primitive polynomial $p(x)$ are known we can represent each element by the giving the related power of the element α.

Component representation: based on the polynomial representation, the unique coefficients of every polynomial is represented as a 1 if it is present or a 0 if it is not, using a binary representation of size m.

Decimal representation: changing the primitive element α in a polynomial representation to the prime number p of the field $GF(p)$, we can obtain an unique decimal value for each element.

Exponents	Polynomials	Components	Decimal Values
$-\infty$	0	0000	0
0	1	0001	1
1	α	0010	2
2	α^2	0100	4
3	α^3	1000	8
4	$\alpha + 1$	0011	3
5	$\alpha^2 + \alpha$	0110	6
6	$\alpha^3 + \alpha^2$	1100	12
7	$\alpha^3 + \alpha + 1$	1011	11
8	$\alpha^2 + 1$	0101	5
9	$\alpha^3 + \alpha$	1010	10
10	$\alpha^2 + \alpha + 1$	0111	7
11	$\alpha^3 + \alpha^2 + \alpha$	1110	14
12	$\alpha^3 + \alpha^2 + \alpha + 1$	1111	15
13	$\alpha^3 + \alpha^2 + 1$	1101	13
14	$\alpha^3 + 1$	1001	9

Table 8.3: Representation of $GF(2^4)$ elements

8.1.4 Addition and Multiplication

The mathematical operations needed to perform our calculations in non binary LDPC codes are addition and multiplication, thus, we define them as follows.
Addition $\oplus$ of extension field elements is based on the addition of the corresponding prime field elements out of $GF(p)$:

$$x \oplus y := (x_{m-1} \oplus y_{m-1}, ..., x_1 \oplus y_1, x_0 \oplus y_0)$$

In the case of the binary prime field $GF(2)$, an addition is a bitwise exclusive OR (XOR) of the two elements. Subtraction has the same result.
Multiplication $\otimes$ of extension field elements in exponenetial representation simply means the addition of the exponenets *modulo* $p^m - 1$:

$$x \otimes y := (x.y) mod p(\alpha)$$

8.2 Syndrome Algorithm Execution Example

In this appendix we present an execution example of the syndrome algorithm, to clarify how this algorithm works, for the case the explanation given in chapter 4 was not enough for a clear understanding.

Consider that are using a NB-LDPC code over $GF(64)$, with $n_m = 13$ and $d_c = 4$. In the figure below we can see the input of every connected VN at the check node, and the sorted initial LLRs provided from every connection. Have in mind that every one of this LLR values have with it a correspondent GF element, we are not showing them in

the figures because the algorithm decisions are based on the LLRs and therefore we can simplify the images.

	LLR	LLR	LLR	LLR
12	28	21	45	22
11	28	19	43	20
10	28	14	43	20
9	22	15	43	19
8	21	15	41	17
7	20	13	37	15
6	20	13	32	15
5	10	11	25	11
4	10	11	25	11
3	10	8	28	11
2	9	8	19	7
1	1	2	14	3
0	0	0	0	0

Figure 8.1: Input of the check node

One can remember that we use the probe scheme presented on chapter 5. We group the inputs messages with their neighbours and make the sorting based only on one value of each group to simply the sorting.

	LLR	LLR	LLR	LLR
12	28	21	45	22
11	28	19	43	20
10	28	14	43	20
9	22	15	43	19
8	21	15	41	17
7	20	13	37	15
6	20	13	32	15
5	10	11	25	13
4	10	11	25	11
3	10	8	28	11
2	9	8	19	7
1	1	2	14	3
0	0	0	0	0

Figure 8.2: Probe scheme

Based on this probes we select the most reliables of them (lowest LLR values). In this case we have chosen 8. It is also important to note from which input every sorted group came from.

In{0}	In{1}	In{3}	In{0}	In{1}	In{3}	In{1}	In{2}
10	8	11	20	13	15	15	28
9	8	7	10	11	13	15	19
1	2	3	10	11	11	13	14

Figure 8.3: Sorted groups of messages

We are considering here only values for deviation 1 and deviation 2. We calculate Dev1 values for every one of those sorted groups. As shown in chapter 4, a Dev1 value is calculated summing one of each sorted value with the most reliable values from the other inputs. Since we use $LLR = 0$ for the most reliable values, there is no need to make any sum for Dev1.

	LLR	LLR	LLR	LLR
12	28	21	45	22
11	28	19	43	20
10	28	14	43	20
9	22	15	43	19
8	21	15	41	17
7	20	13	37	15
6	20	13	32	15
5	10	11	25	11
4	10	11	25	11
3	10	8	28	11
2	9	8	19	7
1	1	2	14	3
0	0	0	0	0

Figure 8.4: How a dev1 is calculated

The deviation 2 calculation is a bit different. We use 2 sorted groups and calculate three values, as shown in the figure below, for every possible combination. We will have 2 inputs contributing for one Dev2 calculation, if 2 sorted groups are from the same input, their Dev2 calculation is not valid.

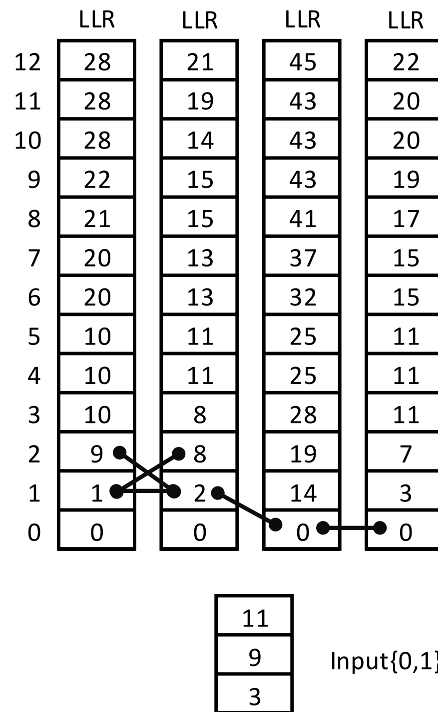

Figure 8.5: How a dev2 is calculated

In this case we considered 4 sorted groups to make Dev2 calculations. Therefore we will have 6 possible combinations. Below we show all the generated values:

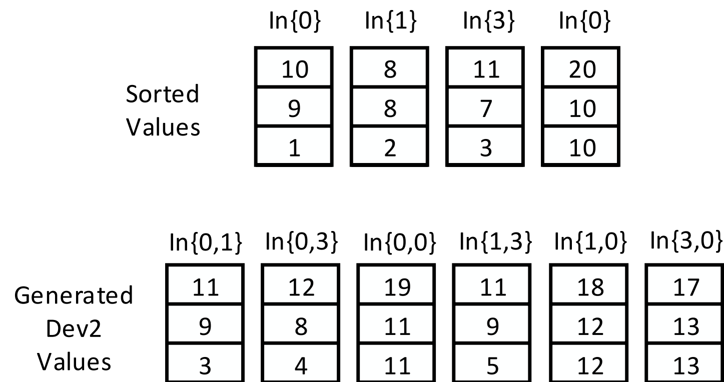

Figure 8.6: Generated Dev2 Syndromes

Based on the parameters used on this example, we have generated 8 groups of syndromes for Dev1 and 6 for Dev2. One may notice that there is a Dev2 syndrome with both base values from the first input, that means this syndrome is not valid.

	In{0}	In{1}	In{3}	In{0}	In{1}	In{3}	In{1}	In{2}
Dev1 Values	10	8	11	20	13	15	15	28
	9	8	7	10	11	13	15	19
	1	2	3	10	11	11	13	14

	In{0,1}	In{0,3}	In{0,0}	In{1,3}	In{1,0}	In{3,0}
Dev2 Values	11	12	19	11	18	17
	9	8	11	9	12	13
	3	4	11	5	12	13

Figure 8.7: All Generated Syndromes

After that, to update the output of every connection, we must select the n_m best values from those sorted sets, that are not based on the to-be-updated connection(belief propagation principle). This makes the decorrelation easy, because all the values we need to decorelate are based only on the most reliable value of this output. In the figure we show possible values for the first output.

In{1}	In{3}	In{1}	In{3}	In{1}	In{2}
8	11	13	15	15	28
8	7	11	13	15	19
2	3	11	11	13	14

In{1,3}
11
9
5

Figure 8.8: Possible values for $output_0$

After all the values are selected we have an output like this:

	LLR	LLR	LLR	LLR
12	11	11	9	11
11	11	11	9	11
10	11	11	8	10
9	11	10	8	10
8	11	10	8	10
7	9	10	7	9
6	8	9	5	9
5	8	8	4	8
4	7	7	3	8
3	5	4	3	3
2	3	3	2	2
1	2	1	1	1
0	0	0	0	0

Figure 8.9: Output

As the third output contributed with only one group of probes, it is likely that at his output it will have the lowest values comparing with the others.

8.3 Hardware Top Module Code

Listing 1: Top Module of the Check Node implementation

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.pkg_param.all;
use work.pkg_param_derived.all;
use work.pkg_types.all;
use work.pkg_components.all;

entity check_node_reduced_syn is
port(
        -- INPUTS
        rst : in std_logic;
        clk : in std_logic;
        en  : in std_logic;

        start        : in std_logic;
        max_elements : in t_max_vec;
        input        : in t_syn_gen_vec;

        -- OUTPUTS
        dev0_out : out t_dev0_vec;
        output   : out t_out_val_vec;
```

```
25          valid    : out t_valid_vec
26  );
27  end check_node_reduced_syn;
28
29  architecture behavioral of check_node_reduced_syn is
30
31          -- Probe sorter
32          signal probe_inputs : t_probe_input;
33          signal probe_addr_out: t_probe_addr;
34          signal probe_addr_out_reg: t_probe_addr;
35
36          -- Subvalue generator
37          signal dev0 : t_gf_element;
38          signal dev1_sub : t_sub_dev1_vec;
39          signal dev2_sub : t_sub_dev2_vec;
40
41          -- Pipeline regs
42          signal dev0_reg_1 : t_gf_element;
43          signal dev0_reg_2 : t_gf_element;
44
45          -- Values selector
46          signal dev_out : t_sub_syn_vec;
47          signal dev_out_reg : t_sub_syn_vec;
48
49          -- Dev1
50          signal dev1_in : t_syn_vec_dev1;
51          signal dev1_syn_val : t_syn_vec_dev1_out;
52          signal dev1_val_reg : t_syn_vec_dev1_out;
53
54          -- Dev2
55          signal dev2_in : t_syn_vec_dev2;
56          signal dev2_syn_val : t_syn_vec_dev2_out;
57          signal dev2_val_reg : t_syn_vec_dev2_out;
58
59          signal start_d1 : std_logic;
60          signal start_d2 : std_logic;
61          signal start_d3 : std_logic;
62          signal start_d4 : std_logic;
63          signal start_d5 : std_logic;
64
65          signal max_elements_d1 : t_max_vec;
66          signal max_elements_d2 : t_max_vec;
67          signal max_elements_d3 : t_max_vec;
68          signal max_elements_d4 : t_max_vec;
69          signal max_elements_d5 : t_max_vec;
70
71  begin
72          gen_sorter_in : for i in MAX_D_C-1 downto 0 generate
73                  probe_inputs(i) <= input(i)(0).value;
74          end generate;
```

```
75
76          sorter : probe_sorter
77          port map (
78                  -- INPUTS
79                  rst => rst,
80                  clk => clk,
81                  start => start,
82
83                  -- OUTPUTS
84                  probe_inputs => probe_inputs,
85                  dev_addr => probe_addr_out
86          );
87
88          subvalue : subvalue_generator
89          port map(
90                  -- INPUTS
91                  max_elements => max_elements_d3,
92
93                  -- OUTPUTS
94                  dev0 => dev0,
95                  dev1_sub => dev1_sub,
96                  dev2_sub => dev2_sub
97          );
98
99          selector : values_selector
100         port map (
101                 -- INPUTS
102                 rst => rst,
103                 clk => clk,
104                 start => start,
105                 probe_addr => probe_addr_out_reg,
106                 input => input,
107
108                 -- OUTPUTS
109                 dev_out => dev_out
110         );
111
112         dev1_in(0) <= dev_out_reg(0);
113         dev1_in(1) <= dev_out_reg(1);
114         dev1_in(2) <= dev_out_reg(2);
115
116         dev2_in(0) <= dev_out_reg(0);
117         dev2_in(1) <= dev_out_reg(1);
118
119         dev1_calc: dev1_syndrome
120         port map (
121                 -- INPUTS
122                 dev1_probe => dev1_in,
123                 dev1_sub_vec => dev1_sub,
124
```

```
125                 -- OUTPUTS
126                 dev1_syn => dev1_syn_val
127         );
128
129         dev2_calc: dev2_syndrome
130         port map(
131                 -- INPUTS
132                 rst => rst,
133                 clk => clk,
134                 start => start_d4,
135
136                 dev2_probe => dev2_in,
137                 dev2_sub_vec => dev2_sub,
138
139                 -- OUTPUTS
140                 dev2_out => dev2_syn_val
141         );
142
143         out_update: output_updater
144         port map(
145                 -- INPUTS
146                 rst => rst,
147                 clk => clk,
148                 start => start_d5,
149                 dev0 => dev0_reg_2,
150                 dev1_in => dev1_val_reg,
151                 dev2_in => dev2_val_reg,
152                 max_elements => max_elements_d5,
153
154                 -- OUTPUTS
155                 dev0_out => dev0_out,
156                 output => output,
157                 valid => valid
158         );
159
160         pr_dev_reg : process(rst, clk)
161         begin
162                 if rst = '1' then
163                         start_d1 <= '0';
164                         start_d2 <= '0';
165                         start_d3 <= '0';
166                         start_d4 <= '0';
167                         start_d5 <= '0';
168                 else
169                         if rising_edge(clk) then
170                                 dev_out_reg <= dev_out;
171                                 probe_addr_out_reg <=
                                       probe_addr_out;
172                                 max_elements_d1 <=
                                       max_elements;
```

```
173                                     max_elements_d2 <=
                                           max_elements_d1;
174                                     max_elements_d3 <=
                                           max_elements_d2;
175                                     max_elements_d4 <=
                                           max_elements_d3;
176                                     max_elements_d5 <=
                                           max_elements_d4;
177                                     dev0_reg_1 <= dev0;
178                                     dev0_reg_2 <= dev0_reg_1;
179                                     dev1_val_reg <= dev1_syn_val
                                           ;
180                                     dev2_val_reg <= dev2_syn_val
                                           ;
181                                     start_d1 <= start;
182                                     start_d2 <= start_d1;
183                                     start_d3 <= start_d2;
184                                     start_d4 <= start_d3;
185                                     start_d5 <= start_d4;
186                             end if;
187                     end if;
188             end process;
189
190     end architecture behavioral;
```

List of Figures

2.1 Communication Chain 6

2.2 Block code codeword 7

2.3 Convolutional coding 9

3.1 Tanner Graph Representation 11

3.2 Parallel Scheduling 12

3.3 Serial Scheduling 12

3.4 Comparison between LDPC codes of variable length 17

4.1 Map bits to symbols over GF(q) 18

4.2 Binary x Non Binary Comparison 18

4.3 Input Reduction 19

4.4 Performance comparison using different n_m values 20

4.5 Syndrome Algorithm 23

4.6 SYN CN Architecture 24

4.7 D_i Sets - Syndrome examples 25

4.8 Distance technique 25

5.1 Moving the sorter 29

5.2 Probe scheme 30

5.3 Sorter output reduce 31

5.4 Common Syndromes 31

5.5 Pre Calculations Module 32

5.6 Final Hardware 32

5.7 CSE Communication Chain 33

5.8 Hardware Overview 34

5.9 Probe Sorter 36

5.10 Value Selector 37

5.11 Subcalculations module 38

5.12 Dev1 Syndrome 38

5.13 Dev2 Syndrome 39

5.14 Output Updater 40

6.1 Communications Performance 41

6.2 Area comparison 42

6.3 Throughput Comparison 43

6.4 Scaled architecture 44

8.1 Input of the check node 49

8.2 Probe scheme 49

8.3 Sorted groups of messages 50

8.4 How a dev1 is calculated 50

8.5 How a dev2 is calculated 51

8.6 Generated Dev2 Syndromes 51

8.7 All Generated Syndromes 52

8.8 Possible values for $output_0$ 52

8.9 Output 53

List of Tables

5.1 Sorting complexity 35

6.1 Area comparison 42

6.2 Throughput comparison 42

6.3 Sorting complexity 43

6.4 Final results 44

8.1 Primitive polynomials for $GF(2^m)$ 47

8.2 Generating all elements for $GF(2^4)$ 47

8.3 Representation of $GF(2^4)$ elements 48

List of Listings

3.1 Algorithm Belief-Propagation Decoding [9] 15

3.2 Algorithm Bit-flipping Decoding [9] 16

4.1 Extended Min-Sum Algorithm 22

4.2 Syndrome Algorithm 27

1 Top Module of the Check Node implementation 53

List of Abbreviations

BF Belief Propagation

CN Check Node

CSE Creonic Simulation Environment

EMS Extended Min-Sum

FER Frame Error Rate

FWBW Forward Backward

GF Galois Field

LDPC Low Density Parity Check

LLR Log-likelihood Ratio

LUT Lookup Table

MAP Maximum a Posteriory

ML Maximum Likelihood

NB-LDPC Non-Binary Low Density Parity Check

PN Permutation Nodes

SNR Signal-to-noise Ratio

SP59 Sphere Packing Bound

SYN Syndrome-Based

VN Variable Node

References

[1] Robert G. Gallager. Low Density Parity Check Codes. *The Bell System Technical Journal*, 1963.

[2] Claude E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, July 1948.

[3] Matthew C. Davey and David MacKay. Low-Density Parity Check Codes over GF(q). *IEEE COMUNICATIONS LETTERS*, June 1998.

[4] L. Barnault and D. Declerq. Fast decoding algorithm for LDPC over GF(2q). *IEEE Information Theory Workshop*, March 2003.

[5] P. Schläfer, N. Wehn, M. Alles, T. Lehnigk-Emden, and E. Boutillon. Syndrome Based Check Node Processing of High Order NB-LDPC Decoders. *International Conference on Telecommunications*, 2015.

[6] R. W. Hamming. Error detecting and error correcting codes. *Bell Syst. Tech. J.*, April 1950.

[7] P. Elias. Coding for noisy channels. *IRE Conv. Rec*, March 1955.

[8] AJ. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, April 1967.

[9] Sarah J. Johnson. Introducing low-density parity-check. *University of Newcastle*.

[10] T.J Richardson, M.A. Shokrollahi, and R.L. Urbanke. Design of Capacity-Approaching Low Density Parity Check Codes. *IEEE Trans. Inf. Theory*, February 2001.

[11] S. Pfletschinger, A. Mourad, E. Lopez, D. Declercq, and G. Bacci. Performance evaluation of non-binary LDPC codes on wireless channels. *Proceedings of ICT Mobile Summit*, June 2009.

[12] A. Voicila, F. Verdier, M. Fossorier, P. Urard, and D. Declerq. Low-complexity decoding for non-binary LDPC codes in high order fields. *IEEE Transactions on Communications*, August 2007.

[13] D. Declercq and M. Fossorier. Decoding algorithms for nonbinary LDPC codes over GF. *IEEE Transactions on Communications*, April 2007.

[14] V. Savin. Min-Max Decoding for Non Binary LDPC Codes. *Proceedings of ISIT*, July 2008.

[15] B. Liu, J. Gao., G. Dou, and W. Tao. Weighted Symbol-Flipping Decoding for Nonbinary LDPC Codes. *Int. Conf. on Network Security, Wireless Communications and Trusted Computing*, April 2010.

[16] G. Sarkis, S. Mannor, and W. J. Gross. Stochastic Decoding of LDPC codes over GF(q). *Proceedings of IEEE ICC*, June 2009.

[17] Timo Lehnigk-Emden. *Implementation and Simulation Aspects of Advanced Non-Binary Iterative Coding Schemes*. PhD thesis, Microeletronic Systems Design Research Group - University of Kaiserslautern, 2011.

[18] K.E. Batcher. Sorting Networks and their Applications. *AFIPS Spring Joint Comput. Conference*, 1968.