Design of Quasi-cyclic LDPC decoder for 5G NR

A Project Report

submitted by

VIGNESH SUNDARESHA

in partial fulfilment of the requirements
for the award of the degree of

DUAL DEGREE
(BACHELOR AND MASTER OF TECHNOLOGY)

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JULY 2021

THESIS CERTIFICATE

This is to certify that the thesis titled **Design of Quasi-cyclic LDPC decoder for 5G NR**, submitted by **Vignesh S**, to the Indian Institute of Technology, Madras, for the award of the degree of **Dual degree (B.Tech and M.Tech)**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Andreas Peter Burg
Research Guide
Associate Professor
STI IEL TCL
EPFL, Switzerland

Prof. Janakiraman Viraraghavan
Research Guide
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Lausanne

Date: 28th July 2021

ACKNOWLEDGEMENTS

Firstly, I would like to express my immense gratitude to both my guides Prof. Janakiraman Viraraghavan and Prof. Andreas Burg. I was inspired to pursue research in VLSI thanks to the excellent teaching of Prof. Janakiraman in his Digital IC design course. Through my interactions with him I have gained a lot knowledge and insights about the subject. His flexibility, cooperativeness and understanding has helped me complete this project. I would like to thank Prof. Andreas Burg for giving me an opportunity to work in his lab and do my thesis here. His way of going about the research, solving the problems with us and using different tools to do things efficiently has helped me become a better researcher. I would also like to thank Prof. Nitin Chandrachoodan for being in the panel and reviewing my work. I want to thank all the professors at IIT Madras for taking their time and teaching me the courses.

I would like to thank Hassan Harb for clarifying all my doubts and guiding me to complete the objective of the project. Then I want to thank Karthikeyan M, my partner in the glitch minimization project for working with me during the pandemic to finish the project. I want to thank Ioanna Paniara and Francesca De Simone for helping me come to Switzerland safely during the pandemic. I also want to thank Rajat, Shruthi, my family members and all my friends for making my stay at IIT Madras a wonderful experience. I want to extend my gratitude to Samsung IITM Pravartak Technologies Foundation Fellowship and the EPFL Excellence in Engineering Fellowship for supporting my project.

Lastly, I want to thank my father, my sister and most importantly my mother for being there for me through the ups and downs and always supporting me.

ABSTRACT

KEYWORDS: 5G NR, QC-LDPC, decoding, code rates, lifting sizes

With the advent of the 5th generation New Radio (5G NR) communication standard, we expect to see massive connectivity and high data rates being delivered on battery-powered devices. One of the error-correction schemes used in the 5G NR standard is the Quasi-cyclic Low-Density Parity-Check (QC-LDPC) codes, which offer good performance in terms of average frame error rates. The 5G LDPC codes cover a wide range of lifting sizes and code rates that make the design of a flexible decoder challenging. This thesis outlines the various modifications that were executed on the reference architecture (built for the IEEE 802.11n standard) to make it functional and fully compliant with the 5G NR standard. The modifications are performed with the consideration of retaining the reference optimizations.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS **i**

ABSTRACT **ii**

LIST OF TABLES **v**

LIST OF FIGURES **vi**

ABBREVIATIONS **vii**

NOTATION **viii**

1 INTRODUCTION **1**

2 LDPC CODES AND DECODING **2**

2.1 Overview of Linear codes 2

2.2 System model 3

2.3 LDPC codes 4

2.3.1 Decoding - Tanner graphs 4

2.3.2 Decoding LDPC codes - Message passing 4

2.3.3 QC-LDPC codes 5

2.4 5G QC-LDPC codes 7

2.5 Decoding algorithms 7

2.5.1 Sum Product algorithm (SPA) 7

2.5.2 Min Sum (MS) and Offset Min Sum (OMS) 9

2.5.3 Layered v/s flooding 10

2.5.4 Layered - Offset Min Sum (L-OMS) 10

2.5.5 Adapted L-OMS 11

2.6 Algorithmic optimizations 14

2.6.1 Specific Message clipping 15

2.6.2 Early termination 15

3 EAGLE - REFERENCE DECODER ARCHITECTURE 17

3.1 Overview 17
3.2 Decoder scheduling 17
3.3 Control unit 19
3.4 Clock control unit 20
3.5 NCU pool 20
3.5.1 MIN unit 20
3.5.2 SEL unit 21
3.6 Memories 21
3.7 Cyclic shifter 21
3.8 Interface 22

4 5G EAGLE - PROPOSED DECODER ARCHITECTURE 23

4.1 IEEE 802.11n v/s 5G NR requirements 23
4.2 Control unit 23
4.2.1 FSM 24
4.2.2 Sequence memory 25
4.3 Clock control unit 26
4.4 NCU pool 26
4.5 Memories 27
4.6 Cyclic shifter 28
4.7 Interface 28
4.8 Summary of changes 28
4.9 Results 29
4.10 Further optimizations and future work 32

5 CONCLUSION 35

LIST OF TABLES

4.1 802.11n v/s 5G NR 23

4.2 Sequence word bit-structure 25

4.3 Command bit-structure 26

4.4 Memory sizes 27

4.5 Architectural modifications 29

4.6 Throughput results 31

LIST OF FIGURES

2.1 System model 3

2.2 Example representation of Tanner graph with CNs and VNs [1] . . . 5

2.3 Base graph structures for 5G 9

2.4 Layered OMS pictorial representation of message passing [2] 12

3.1 Top-level architecture of Eagle [2] 18

3.2 Variable node conflict resolving in scheduling [3] 18

3.3 Interface block diagram 22

4.1 Control FSM in state diagram 24

4.2 Control FSM out state diagram 25

4.3 NCU pool block diagram 27

4.4 SNR v/s FER 30

4.5 clk cycles v/s CR and NZ 33

4.6 clk cycles v/s Z and iters 34

ABBREVIATIONS

SISO	Single Input Single Output
SNR	Signal to Noise Ratio
AWGN	Additive White Gaussian Noise
VLSI	Very Large Scale Integration
QC-LDPC	Quasi-Cyclic Low Density Parity Check
SPA	Sum Product Algorithm
MS	Min-Sum
OMS	Offset Min-Sum
CN	Check Node
VN	Variable Node
MC	Message clipping
NZ	Non-zero
MCC	Macro computation cell
CMD	Command
FSM	Finite State machine
5G NR	5th generation New Radio
IEEE	Institute of Electrical and Electronics Engineers
LUT	Look up table
FER	Frame Error Rate
CR	Code Rate

NOTATION

$\mathbb{P}(Er)$	Probability of error
$\mathbf{K}$	Message length
$\mathbf{N}$	Block length
$\mathbf{H}$	Parity check matrix
$\mathbf{G}$	Generator matrix (dimension : K x N)
$\mathbf{m}$	Message vector (dimension : 1 x K)
$\mathbf{c}$	Codeword vector (dimension : 1 x N)
$\mathbf{H}_p$	Prototype parity check matrix
$\mathbf{L}_s$	Number of commands in sequence memory
$\mathbf{B}_R$	Quantization bits for R-message
$\mathbf{B}_Q$	Quantization bits for Q-message
$\mathbf{B}_T$	Quantization bits for T-message
$\mathbf{Z}_{max}$	Maximum lifting size
$\mathbf{N}_{B,max}$	Maximum non-zero blocks
$\mathbf{N}_{non-zero\ block}$	Number of non-zero blocks
$\mathbf{N}_p$	Maximum number of columns of $\mathbf{H_p}$
$\mathbf{M}_p$	Maximum number of rows of $\mathbf{H_p}$
L_j	Intrinsic information of VN
$L_{i\to j}$	CN update
$L_{j\to i}$	VN update
Z	Lifting size

CHAPTER 1

INTRODUCTION

In the current age where smartphones and internet are extensively used, it is imperative that we have high data rates, massive connectivity and reliable communication. The 5th generation New Radio (5G NR) [4] standard aims to support these features by using Quasi-Cyclic Low Density Parity Check (QC-LDPC) codes as one of the error correction schemes. The LDPC codes [5] have good performance compared to existing linear codes and the QC-LDPC codes [6] reduce the hardware complexity of the decoder significantly, thus making LDPC codes realizable and one of the error correction schemes used for 5G NR. 5G LDPC codes consist of two prototype matrices with 51 lifting sizes varying between 2 and 384 and code rates ranging from 0.2 to 0.9. Due to the wide range of lifting sizes and code rates, 5G LDPC codes are suitable for various applications each with its own specific requirements (small/modern/high code length and code rates).

The state-of-the-art is rich with LDPC decoder designs ([7][2] to name a few). The authors in [2] proposed, what is called, Eagle decoder for IEEE 802.11n and 802.16e standards. The features in terms of power consumption and throughput rate of the Eagle architecture makes it a suitable starting point for our work.

Following this chapter the report is organized as follows: chapter 2 introduces the LDPC codes and some of the decoding algorithms used for them. Chapter 3 talks about the reference architecture Eagle [2]. The steps to update the Eagle architecture to 5G Eagle are presented in chapter 4. Finally the chapter 5 concludes the report.

CHAPTER 2

LDPC CODES AND DECODING

2.1 Overview of Linear codes

In real world both noise and fading affect the message being transmitted thus resulting in the reception of erroneous messages. In the mid 20th century Claude Shannon came up with a theory [8] which stated that at a given signal-to-noise ratio (SNR) it is possible to achieve nearly error-free (probability of error : $\mathbb{P}(Er)$, tends to zero) transmission up to a maximum rate which is called capacity. The talk of rate - which is the fraction of message bits among the transmitted bits - arises due to the addition of redundant bits that do not carry any information. These redundant bits are computed in a manner which enables the received erroneous message to be decoded. This sort of encoding and decoding occurs not only in communication systems, but also in various other practical systems such as Compact Disks (CDs), data storage systems and so on.

There are various methods for channel coding [9]. However the linear block codes are widely used for practical purposes due to minimal hardware complexity when compared to other methods. They are called block codes since the redundant bits are added to a block of message bits and not individual bits. Linear block codes mean that the combination of two or more codewords results in a codeword belonging to the same code. Therefore, linear block codes can be represented by a generator matrix $\mathbf{G}$ and parity check matrix $\mathbf{H}$. The generator matrix is used at the encoding side where the codeword is generated (see equation 2.1a) and the parity check matrix defines the set of constraints to be satisfied at the decoding side (see equation 2.1b). The generator matrix is used while encoding the message while parity check matrix plays a crucial role in decoding the message. The following equations hold,

$$\mathbf{c} = \mathbf{G}\mathbf{m} \tag{2.1a}$$

$$\mathbf{H}\mathbf{c}^{\mathbf{T}} = 0 \tag{2.1b}$$

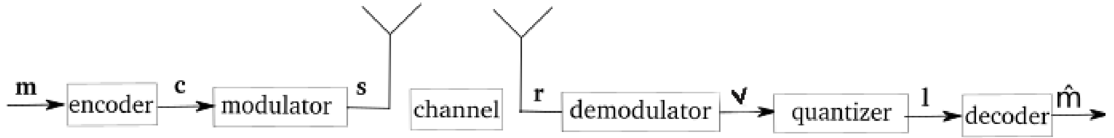


Figure 2.1: System model

There are several linear block codes [1] that have been used in the past for communication systems. Repetition codes, Hamming codes [10], Bose-Chaudhuri-Hochquenghem (BCH) codes [11], Reed-Solomon (RS) codes [12] are some of the most popular linear block codes. All these codes have been constructed with a specific structure to the parity check matrix that allows for simpler decoding. In recent times the Low-density parity-check codes (LDPC) has become more predominant. However, unlike the former set of codes, the LDPC codes do not have a well-defined structure beforehand. As the name suggests, the only constraint is that the parity check matrix should be sparse. As a result of this lack of structure, the encoding and decoding becomes more complex. The reason for the usage of LDPC codes, the decoding complexity and ways to overcome this hurdle will be explained in the coming sections.

2.2 System model

The model of the employed communication system is shown in figure 2.1. Throughout the whole process we use SISO communication system with an AWGN channel. The input to the encoder is K uniform i.i.d. message bits. The output of the encoder - which is the codeword - is sent as input to the modulator that performs BPSK modulation. This is then sent across the AWGN channel and the received vector is passed through the demodulator. The demodulator converts the received vector into LLR information, which is then sent to the quantizer (as in equation 2.2). The quantized LLRs are sent to the decoder which performs the decoding and gives the estimate of the bits that were transmitted.

$$\begin{aligned} v_n &\triangleq \log\left(\frac{P(c_n = 0|\mathbf{r})}{P(c_n = 1|\mathbf{r})}\right) \\ l_n &= \mathbf{Q}(v_n) \end{aligned} \tag{2.2}$$

2.3 LDPC codes

The LDPC codes were originally introduced by Gallager [5] in 1962. But the lack of advancement in the VLSI technology required for the decoding prevented people from investigating the topic further at that time. The main motivation for reusing the LDPC codes was their capability to achieve performance very close to capacity when compared to the prevailing linear block codes [13]. One of the justifications given for this improved performance of LDPC codes is the “similarity" in their construction with that of Shannon’s [8]. Shannon uses a random code while proving his theorem and unlike the specificity in the structure of the parity check matrices as in [11] [12], the LDPC codes have a fairly random construction for their parity check matrix [14] (we show later how we constrain this with structure to help encoding and decoding).

2.3.1 Decoding - Tanner graphs

Apart from looking at the parity check matrix as a set of equations that need to be satisfied, another intuitive way would be to look at them from a graph theory perspective. If we consider the parity check matrix as the adjacency matrix of a graph, then the rows become the check nodes (CN) and the columns become the variable nodes (VN). We can do so, since there are no internal connections between the CNs or VNs thus making this a bipartite graph. A more common terminology for this used in the field of communications is the Tanner graph. Every non-zero entry on the parity check matrix indicates a connection/edge between the corresponding CN and VN, i.e. an ith CN is connected to a jth VN if,

$$[\mathbf{H}]_{i,j} = 1$$

Thus the sum (in the binary field : GF(2)) of all the VNs connected to a particular CN should be 0. This should hold true for all the CNs which translates to 2.1b.

2.3.2 Decoding LDPC codes - Message passing

The decoding for the LDPC codes uses the iterative decoding algorithm. This algorithm is efficient for decoding when the connections are sparse which is the case with LDPC codes. The iterative decoding algorithm works on the principle of message passing. As

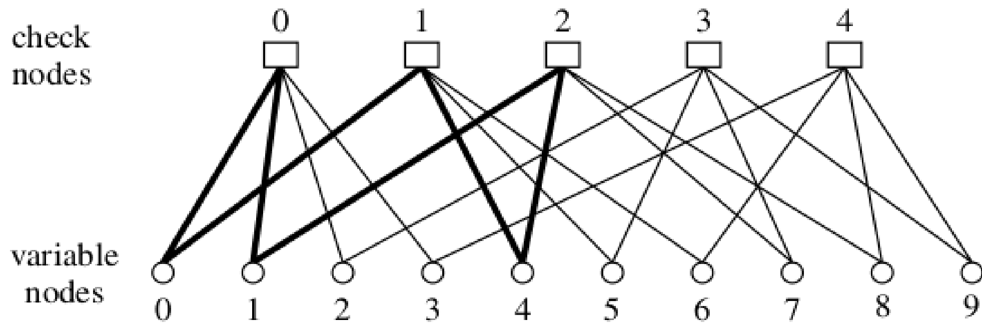


Figure 2.2: Example representation of Tanner graph with CNs and VNs [1]

the name suggests, message passing means the transfer of message/information from one node to its connected nodes. In case of communication systems this information is passed in the form of LLRs. In the first step the decoder receives LLRs from the channel - the prior probability information - which is sent to the VNs. The next step - which is the iterative step - is where the LLR information from the VNs is transferred to the CNs. Here each CN - connected to d_c VNs - sends back the information to all of its connected VNs by estimating the parity of that VN using the remaining $d_c - 1$ VNs. Each CN performs this for all the VNs connected to it. After this, the VNs update their values based on the parity information they receive from all their connected CNs. These two updates keep on going until a valid codeword is obtained, or maximum number of iterations is reached. Let the length of a cycle in the Tanner graph be indicated as the number of passed edges starting from a node and coming back to the same node. It is observevd that the short cycles in the graph negatively affect the performance.

2.3.3 QC-LDPC codes

Though the performance of LDPC codes are very good, the encoding and decoding becomes extremely difficult due to lack of structure. Thus, the goal is to strike a balance by not affecting the performance significantly, but still giving the parity check matrix some structure that will reduce the encoder and decoder complexities. Thus effectively, we try to make the code pseudorandom. The cyclic codes - in which any codeword belonging to the code can be circularly shifted to obtain another word that belongs to the same code - have simpler encoder-decoder hardware compared to most other linear codes. Their encoding and decoding can implemented using feedback shift registers. But they provide lot of structure to the parity-check matrix and remove the randomness

needed. Thus we use Quasi-Cyclic (QC) codes, which do not significantly harm the performance [14], but aid enormously by reducing the hardware complexity and allowing for parallelism. The parity check matrix for these QC-LDPC codes can be represented compactly using prototype matrix. This prototype matrix $\mathbf{H}_p$ (as shown below) is of size $M_p \times N_p$ contains entries (called shift amounts) whose value ranges from 0 to Z along with "–", where Z is the lifting size.

$$\mathbf{H_p} = \begin{bmatrix} 307 & 19 & 50 & \dots & \dots & - \\ 76 & - & 76 & \dots & \dots & - \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ - & 135 & - & \dots & \dots & 0 \end{bmatrix}_{M_p \times N_p}$$

Each entry in the prototype matrix is replaced by a $Z \times Z$ cyclic shift matrix $\mathbf{P}^c$, where $c\ (> 0)$ represents the value at that particular entry in $\mathbf{H}_p$. This cyclic shift matrix is defined as follows,

$$\mathbf{P}^c = \prod_{i=1}^{c} \mathbf{P}^1 \tag{2.3a}$$

$$\mathbf{P}^0 = \mathbf{I}_{Z \times Z} \tag{2.3b}$$

$$\mathbf{P}^- = \mathbf{0}_{Z \times Z} \tag{2.3c}$$

$$[\mathbf{P}^1]_{i,j} = \begin{cases} 1, & j = i \pmod{Z} + 1 \\ 0 & \text{else} \end{cases} \tag{2.3d}$$

This choice of cyclic shift matrix has sparsity same as the identity matrix thus making the overall blown-up parity check matrix low density. However, the code would not be a LDPC code if the cyclic shift matrix chosen was denser. Each block in the parity check matrix corresponding to each entry in the prototype matrix is just a cyclic shift of the identity matrix as it can be seen in equation 2.4a. Hence we call these codes as 'Quasi' cyclic LDPC codes.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.4a}$$

$$\mathbf{P}^1 \times \mathbf{P}^1 = \mathbf{P}^2 \tag{2.4b}$$

2.4 5G QC-LDPC codes

In case of 5G LDPC codes, there are two main divisions for the prototype matrices which are called Base Graphs (BG) 1 and 2. The BG 1 is of size 46×68 while the BG 2 is of size 42×52. These are the maximum sizes of the base graphs which corresponds to code rates of 0.33 and 0.2 respectively. However, it is possible to shorten or puncture the code and thus obtain different code rates. There are 51 possible lifting sizes ranging from 2 to 384. The structure of base graphs 1 and 2 are shown in figure 2.3.

2.5 Decoding algorithms

There are several algorithms over time which are variants of the iterative decoding algorithm. The goal of the successive algorithms have been to reduce the hardware complexity by trading off some performance (FER v/s SNR).

2.5.1 Sum Product algorithm (SPA)

The most basic form of the iterative decoding algorithm that can be used for the LDPC decoding is the Sum-Product algorithm (SPA). The hardware complexity for SPA is high. Below (algorithm 1) we show only an algorithmic perspective of the SPA
The notations for L_j, $L_{i \to j}$ and $L_{j \to i}$ are the input LLRs, the CN update from CN i to VN j and VN update from VN j to CN i respectively (see chapter notation).

Algorithm 1: Sum-product algorithm [1]

1 **Initialization :**

2 The input LLRs are computed using the received vector from the channel and is sent to all the VNs

3 $L_{j \to i} = L_j$

4 **CN update :**

5 The message going from ith CN to jth VN is computed by estimating the parity of the jth VN using the information from the other VNs connected to the ith CN

6 $L_{i \to j} = 2 \tanh^{-1} \left(\prod_{j' \in N(i) - \{j\}} \tanh \left(\frac{1}{2} L_{j' \to i} \right) \right)$

7 **VN update :**

8 The message going from jth VN to ith CN is computed

9 $L_{j \to i} = L_j + \sum_{i' \in N(j) - \{i\}} L_{i' \to j}$

10 **LLR total :**

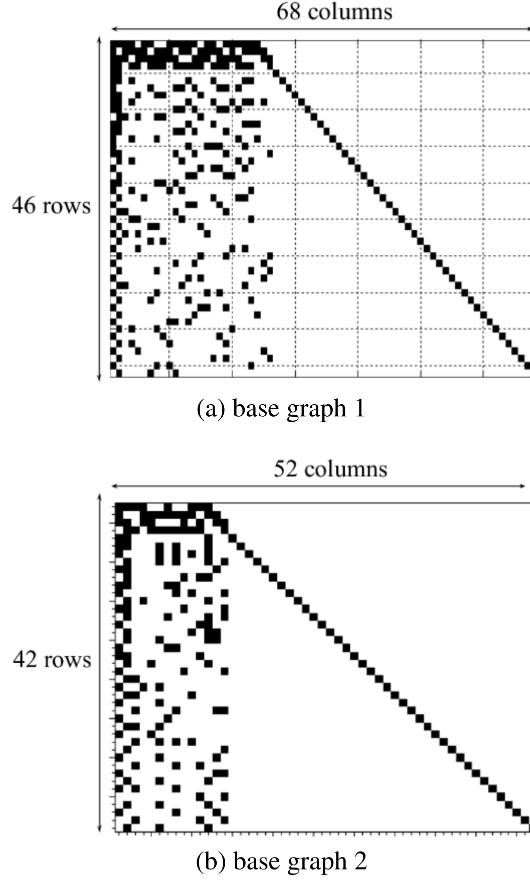
11 Compute the total LLR that is used to determine the codeword

12 $L_j^{total} = L_j + \sum_{i \in N(j)} L_{i \to j}$

13 **Decode and stop :**

14 Using the BPSK criteria we convert the LLR into corresponding bits and obtain the word. We then check if the word belongs to the code

15 $\mathbf{Hc^T} = 0$



(a) base graph 1

(b) base graph 2

Figure 2.3: Base graph structures for 5G

2.5.2 Min Sum (MS) and Offset Min Sum (OMS)

As it can be seen in algorithm 1 : line 6, the functions involved increase the hardware complexity significantly. Thus the goal is to obtain similar functionality which can minimize the hardware, but does not degrade the performance significantly. The expression in line 6 of 1 can be separated into sign and magnitude, and the magnitude component can be replaced with a simple minimum function of the LLRs of all the connected VNs. The below expression is the min-sum approximation of the SPA.

$$
\begin{aligned}
L_{i\to j} &= \prod_{j'\in N(i)-\{j\}} \alpha_{j'i} . \min_{j'\in N(i)-\{j\}} \beta_{j'i} \\
\alpha_{ji} &= sign(L_{j\to i}) \\
\beta_{ji} &= |L_{j\to i}|
\end{aligned}
\tag{2.5}
$$

This is very simple in terms of hardware complexity, but it sacrifices the performance considerably. Thus we use a modified version of this min-sum algorithm called the offset min-sum (OMS). The equation 2.5 is modified as,

$$L_{i\to j} = \prod_{j'\in N(i)-\{j\}} \alpha_{j'i}.\max\{\min_{j'\in N(i)-\{j\}} \beta_{j'i} - c_{\text{offset}}, 0\} \tag{2.6}$$

The CN update for the MS has a very optimistic extrinsic information [1] since we are replacing the entire product term by the maximum term (negative of minimum). Thus OMS compensates for this by subtracting a fixed offset to give better performance.

2.5.3 Layered v/s flooding

The algorithm mentioned in 1 is what is called the flooding schedule. To motivate the use of the layered scheduling we look more closely at the flooding operation. In case of the flooding algorithm, all the VNs send their LLRs to all the CNs at once. The CNs compute the required CN updates and send them back to the VNs all at once. But it is not necessary that all these nodes send their information simultaneously.

This is where the layered scheduling comes in, where initially, all the VNs connected to the first CN send their information to that. Once the first CN computes its updates, it sends back the information to its connected VNs. After this, the next set of VNs connected to the second CN send their information to that and the process goes on until the last CN sends its information back to the VNs. At this point, one iteration is completed (see figure 2.4). Since the later CNs receive information which are well-updated, it is natural that the layered schedule takes lesser iterations to complete the decoding process. However, it should be noted that each iteration itself, takes significantly extra time compared to the flooding schedule. This trade-off provides a designer to choose the schedule based on his/her application.

2.5.4 Layered - Offset Min Sum (L-OMS)

The layered-OMS algorithm [15] has good performance and fast convergence along with being hardware efficient. The algorithm 2 is described below.

Algorithm 2: Layered-OMS [15]

```
1  Initialization :
2      R-messages are initialized to 0, Q-messages are initialized to input LLR
         from the channel
3  loop iters : 1, .., I
4      loop layers : 1, .., M_p
5          Initialize min 1 & 2, index and sign
6          MIN phase :
7              Read LLRs from Q-mem and perform shift
8              compute temp vector
9              compute sign, 1st, 2nd min and index from temp vector
10         SEL phase :
11             Read LLRs from Q-mem and perform shift
12             compute temp vector
13             Obtain data (sign,mins,index) from pipeline registers
14             compute R-message w/ MC
15             compute Q-message w/ MC
16             Shift the Q-messages back using cyclic shifter
```

2.5.5 Adapted L-OMS

Though the layered-OMS is good, it is not fully efficient when implemented on hardware. For instance, the L-OMS in algorithm 2 needs three cyclic shifters which can be easily reduced. The temporary vector is being computed twice. It requires a double clocking systems to read from the Q and R memories. These limitations can be minimized by simple algorithmic and architectural changes which are mentioned below in the algorithm 3.

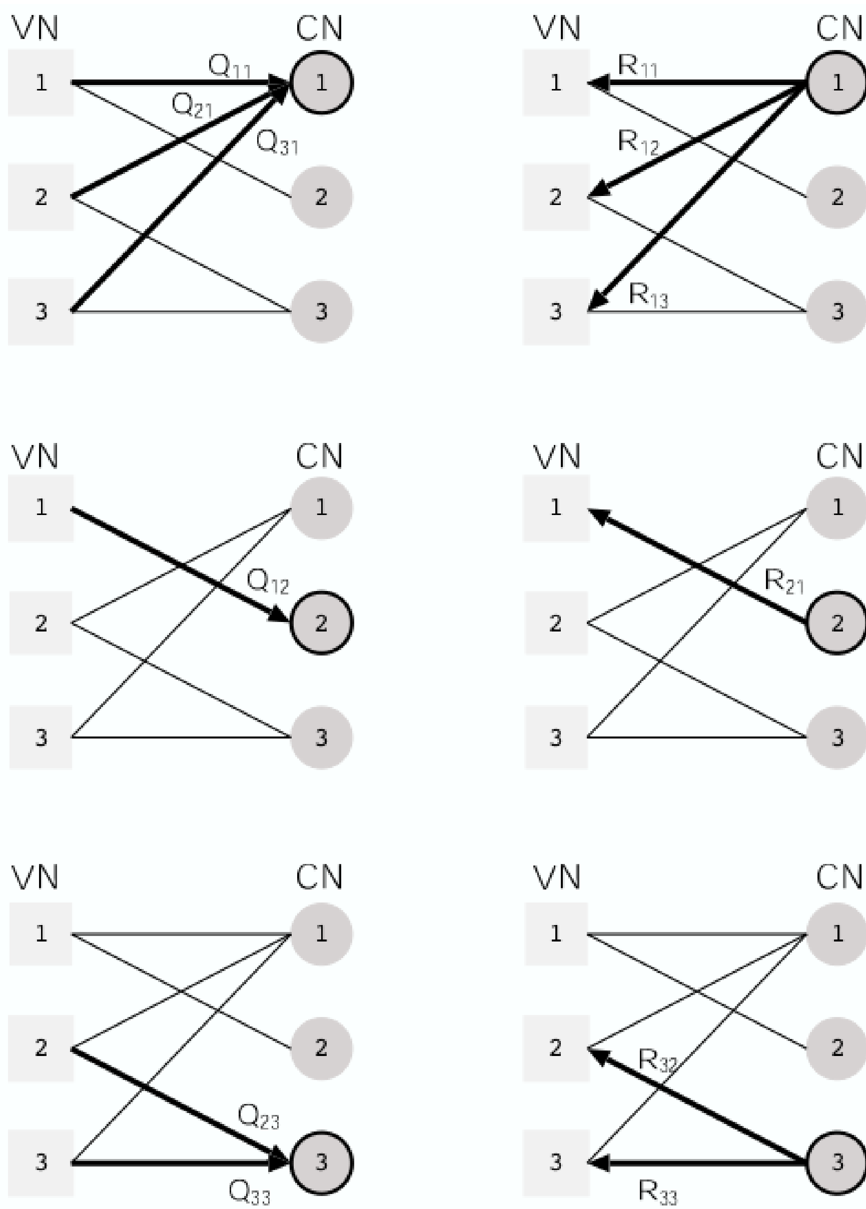


Figure 2.4: Layered OMS pictorial representation of message passing [2]

Algorithm 3: Adapted layered-OMS [2]

1 **Initialization :**

2 R-messages are initialized to 0, Q-messages are initialized to input LLR from the channel : $\mathbf{r^0_{m,n}} = \mathbf{0},\ \mathbf{q_n} = \mathbf{l_n}$

3 **loop iters :** $1, .., I$

4 **loop layers :** $1, .., M_p$

5 Initialize min 1 & 2, index and sign : $\mathbf{m_1}, \mathbf{m_2} \leftarrow \infty.\ \mathbf{I_{Z\times 1}},\ \mathbf{s} \leftarrow \mathbf{I_{Z\times 1}}$

6 **MIN phase :**

7 **loop VN** $n : 1, .., N_p$ **and** $\neq$ ” $-$ ”

8 compute delta shift : $c_\Delta = [Z - c_n + [\mathbf{H}_p]_{m,n}]$(mod Z)

9 compute temp vector & store in T-mem : $\mathbf{t}_n \leftarrow \mathbf{P}^{c_\Delta}\mathbf{q}_n - \mathbf{r}^{i-1}_{m,n}$

10 compute 1st min and index : $[\mathbf{m_1}, \mathbf{x}, \mathbf{v}] \leftarrow \min\{\mathbf{m_1}, |\mathbf{t_n}|\}$

11 compute 2nd min : $\mathbf{m_2} \leftarrow \min\{\mathbf{m_2}, \mathbf{x}\}$

12 compute sign : $\mathbf{s} \leftarrow \mathbf{s}$.sign($\mathbf{t_n}$)

13 **SEL phase :**

14 **loop VN** $n : 1, .., N_p$ **and** $\neq$ ” $-$ ”

15 compute abs min : $\mathbf{m} \leftarrow$ sel$(\mathbf{m_1}, \mathbf{m_2}, \mathbf{v}, n)$

16 compute R-message : $\mathbf{r}^{i-1}_{m,n} \leftarrow \mathbf{s}$.sign$(\mathbf{t_n}).\max\{\mathbf{m} - \beta\}$

17 compute Q-message $\mathbf{q_n} \leftarrow (\mathbf{t_n} + \mathbf{r^i_{m,n}}), c_n = [\mathbf{H_p}]_{m,n}$

18 **loop VN** $n : 1, .., N_p$ **and** $\neq$ ” $-$ ”

19 $c_\Delta = [Z - c_n](mod\ Z), \mathbf{q_{out}} \leftarrow \mathbf{P}^{c_\Delta}\mathbf{q}_n$

The above algorithm is what is used in the reference architecture [2]. The algorithm can be explained as follows,

Initialization

The R-messages are all initialised to 0 and the Q-messages are all initialised to the input LLR. The minimums 1 and 2 along with sign are initialized to infinity and ones respectively (algorithm 3 lines 1 to 5). The operations are looped across the number of iterations and the number of layers. When considering the loops over the columns of

the prototype matrix, we only consider the "non-zero blocks (NZ)", i.e. those which do not have a "–" in the matrix entry as indicated in line 7 of algorithm 3.

MIN phase

In the MIN phase initially the delta shift is computed (algorithm 3 line 8), by which the cyclic shifter has to rotate the Q-messages. After the obtaining the shifted Q-messages, the R-messages are subtracted to obtain the temporary vector which is stored in the T-memory (algorithm 3 line 9). As it can be seen in lines 10-12, the first and second minimums are computed iteratively for each layer along with the first minimum index. Then the product of all the signs is taken in a particular layer is computed. All these computations (first and second minimum, index and signs) are stored in pipeline registers so that they can be used later in the SEL phase.

SEL phase

In the SEL phase, the absolute minimum values for all the VNs are computed using the "sel" function. The sel function assigns the first minimum for the all VNs except that VN which was the first minimum, and replaces that with the second minimum (line 15). After this we compute the R-messages and Q-messages (lines 16,17). While computing the R-messages (CN updates) we subtract the fixed offset value β (hyperparameter) and multiply the sign to obtain the extrinsic information to be sent to each VN. While computing the Q-message, we just add the extrinsic information from the VN and CN to obtain the total LLR. The circular-shift value is also updated. In the end, after processing all the layers and iterations, we undo the cyclic shift so that the Q-messages return to their original value.

2.6 Algorithmic optimizations

A few algorithmic optimizations are done in [2] to improve the throughput and reduce the hardware complexity.

2.6.1 Specific Message clipping

Reducing the number of bits used to represent the LLRs at the Q , R and T memories as well as the inputs is necessary to reduce the hardware complexity. This done by using the message clipping operation. The idea behind this operation is, since the LLRs keep getting added to the Q-memory, it grows larger and larger. As a result we need more bits to represent it. In case we just reduce the number of bits used in the representation of Q-messages, it will significantly impact the performance. So we try to control the maximum value of the R-messages, and in turn control the range of the Q-message value. The message clipping operation is shown below,

$$\text{clip}(\mathbf{r}, \mathbf{t}) = \max\{\min\{\mathbf{r}, Q_{max} - \mathbf{t}\}, -Q_{max} - \mathbf{t}\} \tag{2.7}$$

where $Q_{max} = 2^{B_Q - 1} - 1$ and B_Q represents the bit width of the Q-messages. Thus we can use this message clipping as trade-off to reducing the memory consumption with error-rate performance

2.6.2 Early termination

Early termination is an optimization that is incorporated to save power and increase the throughput of the decoder. The main idea of early termination is to reduce the average number of iterations that is performed to decode the codewords. There are two primary reasons why we would want to do early termination. Firstly, when the SNR is high, it takes very few iterations for the decoder to reach the correct codeword, so we do not want to waste the remaining iterations (each iteration corresponds to at least 88 clock cycles for 802.11n and 316 clock cycles for 5G, when implemented at the minimum code rate) just performing redundant computations. It not only reduces the throughput but also wastes power. Second case is when the SNR is very low, then we know that even if we take the maximum iterations (15 in our case) we are not able to reach the desired codeword. So we decide to discard this frame and go to the next frame. The early termination method for high SNR is straightforward, we just use the equation 2.1b and implement it efficiently on the hardware. However, the low SNR termination is the tricky part, and it is best use a single method that works for both these SNR regimes.

This is what is done in the reference architecture [2]. They employ a statistical metric using which they decide both the high and low SNR early termination. To give a broad idea of this metric, they observe the LLR values of the Q-messages with the increasing iterations and it is seen that in case of high SNR, these values separate into peaks very early, and in case of low SNR, there is almost no separation. The early termination technique is something that is not incorporated in the current version of the 5G Eagle decoder since we are looking for hardware efficient implementations for the same. Thus the derivation of the algorithm for early termination used in [2] is beyond the scope of this report.

CHAPTER 3

EAGLE - REFERENCE DECODER ARCHITECTURE

3.1 Overview

In this chapter we talk about the reference architecture [2], [16] along with the hardware optimizations that were implemented to improve it. The reference architecture - Eagle - was designed to work for both the IEEE 802.11n and 802.16e standards. This was also an improvement done to an older reference design [7] which was slower and lacking power efficiency. The main units of the decoder include the control unit, Q-memory, cyclic shifter and the NCU pool as seen in figure 3.1. These units are explained in more detail in the subsequent sections.

3.2 Decoder scheduling

The decoder is designed in such a way that the Z NCUs can function in parallel to deliver the output. Thus the blocks are processed in groups of Z as seen in the prototype matrix. The MIN and SEL units are decoupled by pipelining and hence can process in parallel. While the MIN unit performs the computations for the $(m+1)$th layer, the SEL unit performs the computation for the mth layer. Since we are doing the layered scheduling, we need the previous layer's LLR updates before computing the MIN computation of the current layer. Thus, in case of VN conflicts as shown in figure 3.2, the blocks are reordered so that the SEL unit performs the computation on these conflicted VNs first and then the MIN unit can use those updated LLRs. In case the conflicts are not resolvable by reordering then the MIN unit is stalled until all the conflicted VNs are updated by the SEL unit.

Assuming there are no stalled cycles, i.e. all the conflicts are resolved by reordering, then we get the following expression for the number of cycles needed to decode a codeword,

$$clk\ cycles = N_{iters} \times N_{non-zero\ blocks} \tag{3.1}$$

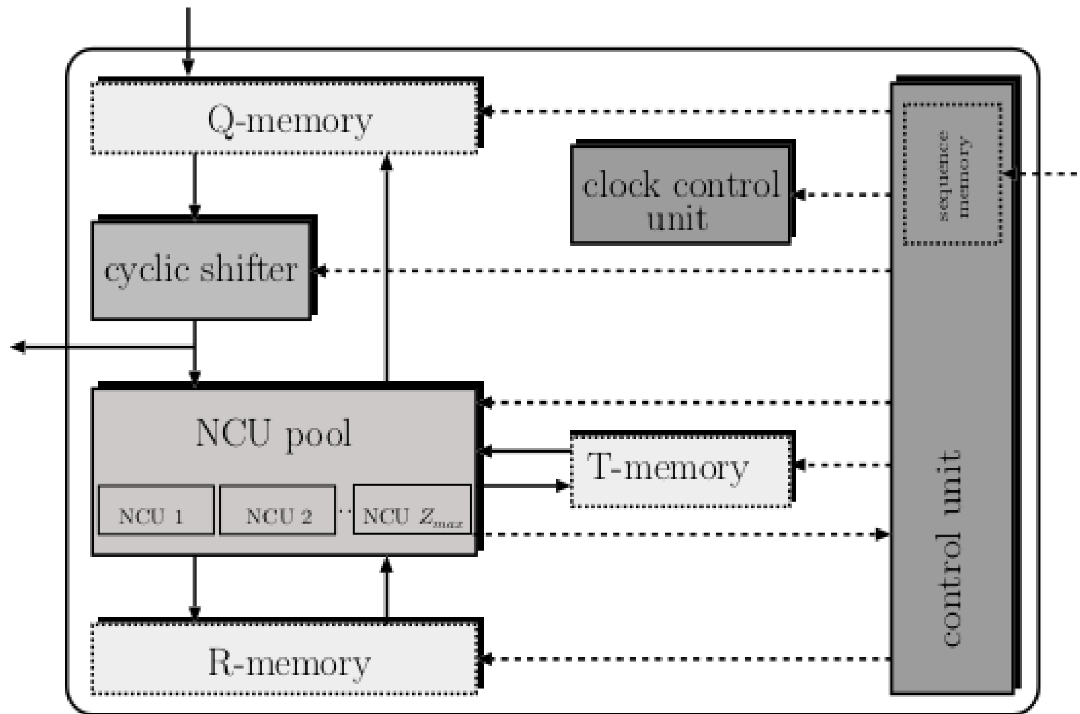


Figure 3.1: Top-level architecture of Eagle [2]

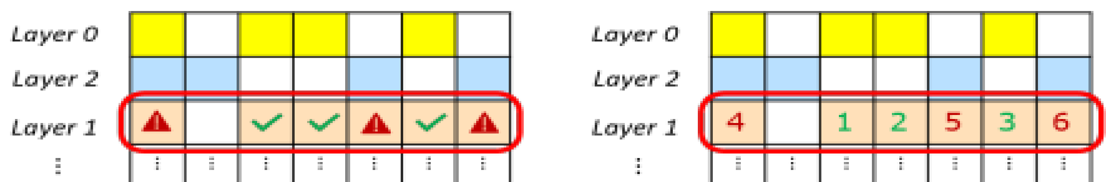


Figure 3.2: Variable node conflict resolving in scheduling [3]

where, N_{iters} stands for the number of iterations on average the decoder needs to complete the decoding process, $N_{non-zero\,blocks}$ stands for the number of non-zero blocks in the prototype matrix. This can be seen from the figure 2.4 where we first go through all the VNs connected to the first CN, and then to the second CN and so on. As a result, the number of times we will be processing the Z NCUs are equal to the number of non-zero blocks in the prototype matrix. Going through all these non-zero blocks needs to performed for many iterations until the expected codeword is obtained.
However, the expression 3.1 ignores the stalling effects. Another thing to keep in mind is the effect pipelining, which also requires some additional cycles to flush out the existing outputs. Taking these effects into account the expression becomes,

$$clk\ cycles = N_{iters} \times (N_{non-zero\,blocks} + N_{stall}) + N_{pipeline} \tag{3.2}$$

where, N_{stall} indicates the number of stalling cycles and $N_{pipeline}$ indicates the number of cycles needed to flush out the pipeline. The architecture is extensively pipelined, i.e. between any two units in figure 3.1 there is pipelining, thus helping all the operations to function in parallel. This pipelining can cause latency and thus complicate the data dependency of the VNs. This also needs to be considered while resolving the clashes during scheduling.

3.3 Control unit

The control unit is the brain of the decoder as it sends all the necessary commands to synchronize different units to work as expected. The control unit consists of the sequence memory which contains the order in which MIN and SEL units have to process the VN blocks. These sequence words which are stored in the sequence memory contain the shift amounts in the prototype matrix along with the VN (Q-memory address) and CN (R-memory address) on which the operation needs to be performed. The table 4.2 shows the attributes and the number of bits used for each of them in the sequence command. The sequence memory has 128 words, to incorporate the 88 non-zero blocks plus any stalling.

The prototype matrix $\mathbf{H_p}$ is converted into the set of commands using MATLAB - where all the scheduling and resolving of clashes is taken care of. The number of commands L_s - which is equivalent to $N_{non-zero} + N_{stall}$ - determines the throughput of the decoder. Thus lower the value of L_s higher the throughput. The sequence commands are stored only for one iteration, so the same set of commands are repeated for each iteration. These sequence commands are loaded into the memory during the configuration phase.

3.4 Clock control unit

It gets the main clock input and enable signals from the control unit, and using this, it generates the gated clocks. These gated clocks are used to control different groups of NCUs and memories, and based on the lifting size it can either enable or disable the groups.

3.5 NCU pool

The NCU pool is the heart of the decoder where all the computations that are necessary to do message passing happen. The entire NCU pool is divided into 3 groups of 27 NCU units. This is done so that when the lifting size is not maximum, some of the groups can be clock gated, thus saving power. Each of these 3 groups contain 3 Macro computation units (MCCs) each having 9 NCUs, a T-memory and a R-memory unit. So in total there are 9 such MCCs. Each NCU contains a MIN unit and a SEL unit to perform the MIN and SEL phase as described in the algorithm 3

3.5.1 MIN unit

The input to the MIN unit is the circularly shifted Z LLR values from the cyclic shifter. In the MIN unit we perform the MIN phase operations (lines 9-12 in algorithm 3). We first compute the temporary vector, and then use that to iteratively compute the first and second minimum along with the indices. We then move on to compute the sign of the entire layer. The temporary vector is stored in the T-memory while the other values are

stored in the pipeline registers - all of which are passed onto the SEL unit.

3.5.2 SEL unit

The SEL unit performs the operations in the SEL phase (lines 15-17 in algorithm 3). The absolute minimums are assigned to all the VNs and then the R-message is computed by subtracting the offset. This is then written into R-memory. But before doing so, the Q-message is also computed.

3.6 Memories

All the memories have been instantiated with a memory wrapper. The sizes of these memories and the their quantization is shown in table 4.4. To improve the throughput (by reducing the L_s) we do memory forwarding for Q and T memories. In case of the forwarding for Q-memory, the output of the SEL unit is directly fed to the cyclic shifter, thus saving the cycles of writing and reading into Q-memory. This is done, only if the Q-memory read address is equal to the Q-memory write address in the same cycle. In case of T-memory, the output of the MIN unit is directly fed to the input of the SEL unit (instead of writing and reading into T-memory), thus saving cycles. At the beginning of each decoding phase, it is important to set the R-memory to zero.

3.7 Cyclic shifter

The cyclic shifter circularly shifts the Z LLRs from the Q-memory by the amount specified in the shift memory. The Eagle architecture uses only one cyclic shifter that performs the delta shift,i.e. instead of circularly shifting the LLRs by the absolute value in the prototype matrix, it shifts by the difference in shift value from the previous non-zero block. The expression for this is seen in line 9 of algorithm 3. Along with the shift memory there is a shift buffer and this loads the shift amount value into the shift memory only after the Q-messages are updated.

3.8 Interface

The interface unit designed here acts as mediator between the chip and the top level decoder. It mainly separates the combined input sent from the chip input pads into LLR information and CMD information so that this can be sent separately to the Q-memory and control unit of the decoder. It does this by using 2 Finite State Machines (FSMs). These FSMs, apart from including the loading of input and unloading of the output, also include request and acknowledgement handshakes. These handshakes are done with the chip and the top decoder module.

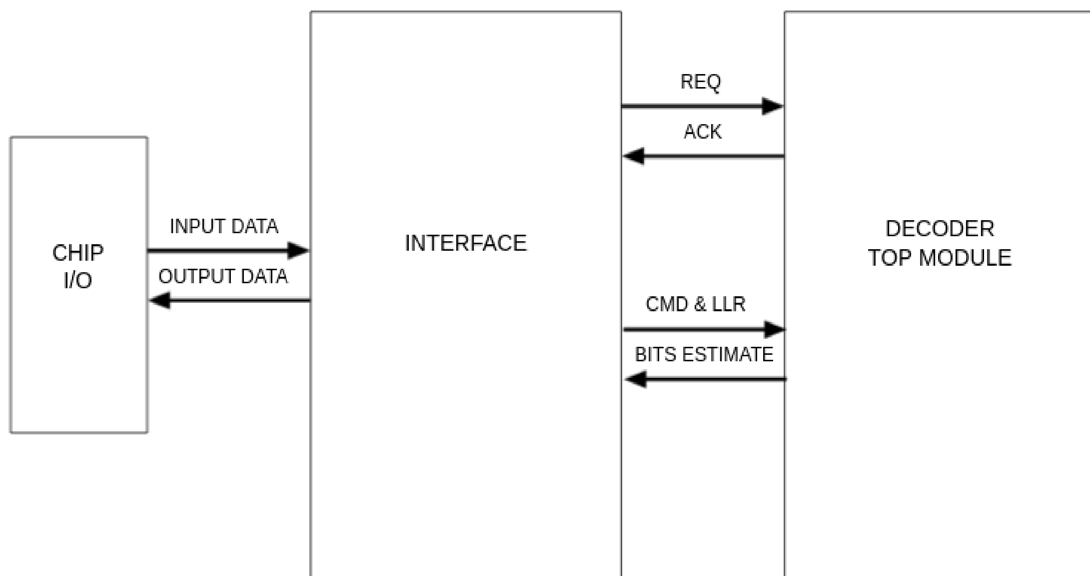


Figure 3.3: Interface block diagram

CHAPTER 4

5G EAGLE - PROPOSED DECODER ARCHITECTURE

4.1 IEEE 802.11n v/s 5G NR requirements

Attributes	IEEE 802.11n	5G NR
Base-graphs	1	2
Lifting sizes (Z)	3	51
Z_{max}	81	384
Z_{min}	27	2
Code rates	4	more than 50
Columns of H_p	fixed - 24	variable
Maximum columns	24	68
Maximum rows	12	46

Table 4.1: 802.11n v/s 5G NR

The reference architecture [2] is used for the IEEE 802.11n standard. As we saw this architecture was optimized very well for power, area and throughput. Thus it would be very convenient if we can retain these optimizations and make the decoder functional for 5G NR. However, as we can see in the table 4.1, there are several differences between the 802.11n standard and 5G NR that prevent us from directly using the Eagle [2] architecture. In the next few sections we explain the major architectural changes that were performed in order to adapt the Eagle architecture to 5G NR.

4.2 Control unit

The main functioning of the control unit of the 5G Eagle is similar to the Eagle [2]. However, a few modifications have been done to accommodate for all the large number of lifting sizes and code rates for 5G.

It should be noted that, unlike the 802.11n standard we cannot divide the NCU/memory groups into the exact lifting size values since there is no progression/sequence in the

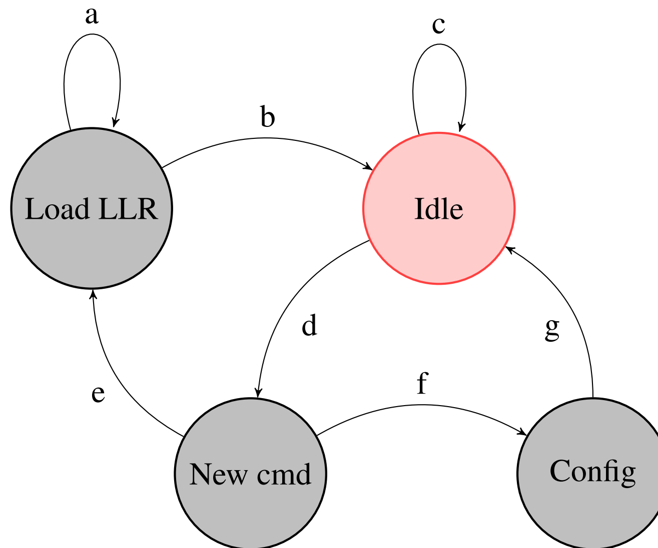


Figure 4.1: Control FSM in state diagram

lifting size values for 5G. Thus if we want to retain the granularity feature of the NCU/memories, we need to group the bunch of lifting sizes (we receive the exact value of the lifting size as input) into bins of 24 so that these groups can be clock gated. The justification for using 16 groups of 24 units is explained in a later section. This grouping is done in the control unit, which sends the enable signals to the clock control unit that generates these gated clocks. There is a clock look up table that is used to map the lifting size to the number of clocks that needs to be clock gated. Using this, the enable signals are sent to the clock control unit. For example all the lifting sizes below 24 : 2,3,..22,24 - come under one group, while all lifting sizes below 48 and above 24 : 26,...48 - come under another group and so on.

4.2.1 FSM

The control unit has three FSMs - FSM in, FSM control and FSM out - that help synchronize the functioning of the decoder. The functioning of the FSM control is unchanged compared to the Eagle architecture. However, the FSMs in and out have to modified to adapt multiple code rates for 5G. This is because these i/o FSMs depend on the value of the number of columns to decide how long they want to stay in the “load LLR" state or the “o/p LLR" state. To provide this value to the i/o FSMs, we use CMD register which stores the number of columns of the prototype matrix (which we obtain as input).

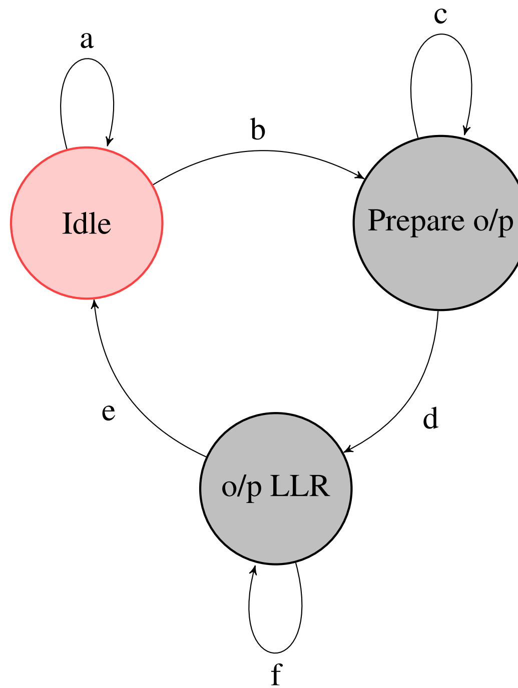


Figure 4.2: Control FSM out state diagram

4.2.2 Sequence memory

The function of the sequence memory is same as in the case of [2]. With the increase in the maximum number of non-zero blocks (from 88 to 316) in the prototype matrix, it is necessary to increase the sequence memory size to 512 words. The changes in the sequence word and command word is as shown in tables 4.2 and 4.3 respectively. The address lines for the sequence memory are also changed from 7 bits to 9 bits.

Attributes	Reference architecture (bits)	Proposed architecture (bits)
Q-mem address	5	7
T-mem address	5	7
R-mem read address	7	9
R-mem write address	7	9
Shift amount	7	9
Min-Sel stall	2	2
row-iter-seq end	3	3
Last Q block	1	1
Total bits	37	47

Table 4.2: Sequence word bit-structure

Attributes	Reference architecture (bits)	Proposed architecture (bits)
Standard	1	1 (unused)
Mode (Z)	2	6
Code rate	2	2 (unused)
No. of columns	0	7
Code attribute	1	1 (unused)
Max-iters	4	4
Early termination (high-low)	2	2
Output mode (bits/LLR) end	1	1
Operation	1	1
Beta	3	3
ETMaxWinSize	5	5
ETMaxNoProgrInRow	5	5
Total bits	27	38 (34)

Table 4.3: Command bit-structure

4.3 Clock control unit

As mentioned before this unit generates the gated clocks using the input clock and enable signals form control unit. We have 16 clocks, that are going to be gated based on the enable signals from control unit.

4.4 NCU pool

The NCU pool is divided into 16 groups of 24 NCUs. Each group has 3 MCCs each containing 8 NCUs, a T-memory block and R-memory block. The architecture and functionality of the internal MIN and SEL units remain unchanged. The reasoning behind this split of NCU groups is, firstly, 16 times 24 is 384 which is equal to the maximum lifting size in 5G. We do not want to split the groups into very small blocks of NCUs (let us say 196 groups of 2) since the memory overhead will be too much. Thus to find a balance and also retain the granularity feature (thus retaining the clock gating feature) we chose 16 groups of 24 units. See figure 4.3 for the block diagram of the NCU.

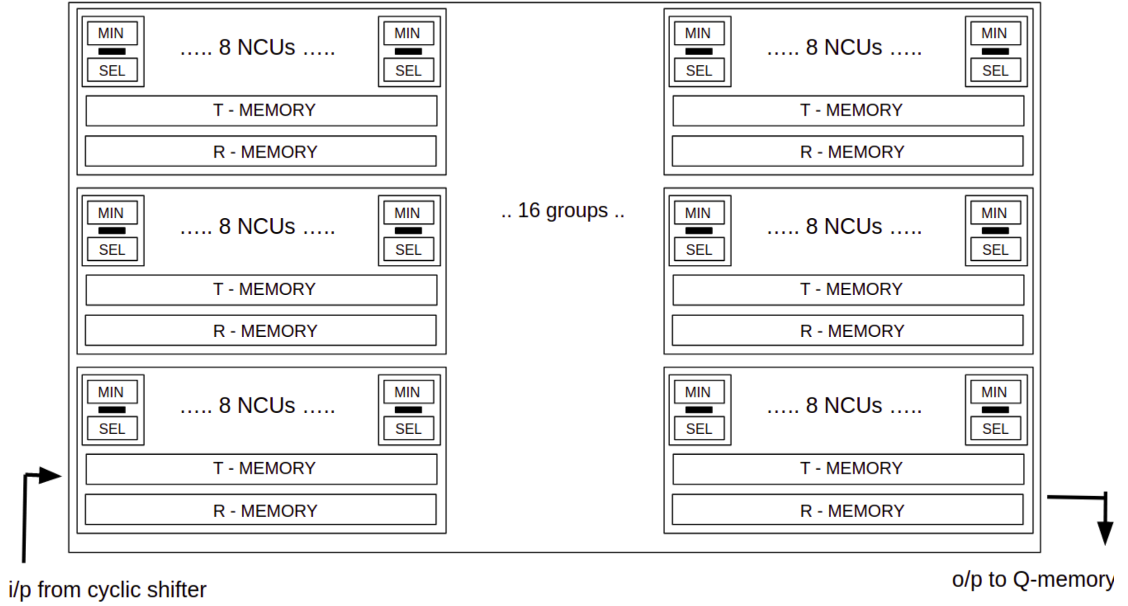


Figure 4.3: NCU pool block diagram

4.5 Memories

The functionality of the memory architecture (along with the forwarding) remains same as Eagle. With the change in the maximum number of columns in the prototype matrix, the maximum number of non-zero blocks and the maximum lifting size, the sizes of the memories have to be changed. This is depicted in table 4.4. It should also be noted that the number of quantization bits used for Q and T memories have been increased from 5 to 7. This was done based on the simulation results, where it was observed that the 7-bit quantization are much closer to the floating point performance as compared to the 5-bit quantization.

Memory units	Expressions	Reference architecture (bits)	Proposed architecture (bits)
R-memory	$B_R N_{B,max} Z_{max}$	$5{\times}88 \times 81$	$5\ {\times}316 \times 384$
T-memory	$B_Q N_P Z_{max}$	$5{\times}24 \times 81$	$7{\times}68 \times 384$
Q-memory	$B_T N_P Z_{max}$	$5{\times}24 \times 81$	$7{\times}68 \times 384$
Total bits	$(B_Q + B_T) N_P Z_{max}$ $+ B_R N_{B,max} Z_{max}$	55080	972288

Table 4.4: Memory sizes

4.6 Cyclic shifter

The functioning of the cyclic shifter also remains similar to the Eagle architecture. The major changes here are, first, the cyclic shifter units are divided into 16 groups of 24 - similar to the NCU blocks. Second, the address (from 2 to 6) and data lines (from 7 to 9) for the shift memory have to updated to accommodate the maximum shift that is 384.

4.7 Interface

Excluding the control unit, interface is the next unit which undergoes the maximum number of changes. The architecture needs to be changed to accommodate both the increased lifting sizes and code rates.

In case of adapting it for the code rates, similar to the control unit, even the interface has FSM in and FSM out (but with a lot more states to check for the requests and acknowledgements). Here as well, in the acknowledgement state and while waiting for the top module, it needs the actual value of the number of columns. This value is stored in the command buffer and is assigned to an internal signal in the interface that is used by the FSM.

To accommodate all the 51 lifting sizes, the interface unit needs to be modified. This is mainly because the it is necessary to map the mode (lifting size) to the LLR input and output handshakes. To perform this, we use look up tables. These look up tables list out all the options which are a function of the Z value and number of input and output pads. Apart from the cyclic shifter unit, this is the only unit which requires the exact value of the lifting size (the group information is not enough). Thus we modify the look up tables to accommodate for the 5G lifting sizes.

4.8 Summary of changes

Listed below is the summary of changes that were done to adapt the Eagle [2] to 5G NR. In case of increase in number of words in the memory, the address bus size also changes. This information is not included in the table.

* in the table indicates that the power of 2 is considered for the number of words in

Units	Reference architecture	Proposed architecture
NCU/memory division	3 groups of 27	16 groups of 24
Quantization of input LLR, Q & T memories	5 bits	7 bits
Q & T memory sizes	24×81 words	68×384 words
R-memory size	88 words	316 words
Sequence memory size	128 words	512* words
CR flexibility	Needs only CR value	Needs the value of no. of rows and columns of H
CMD and SEQ length	27 and 37 bits	34 and 47 bits
Interface	LUTs for 3 Zs	LUTs for 51 Zs

Table 4.5: Architectural modifications

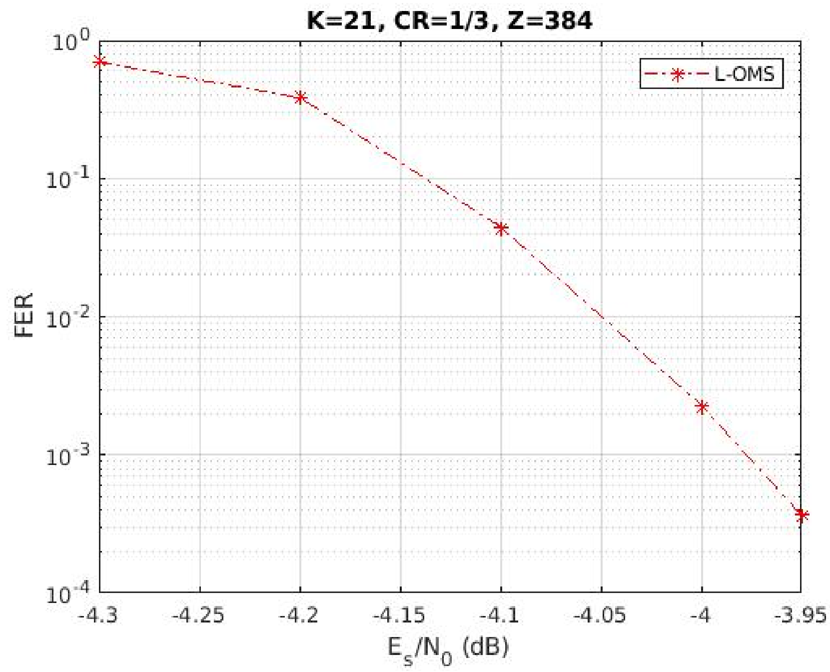
the sequence memory. However, the maximum number of words needed for the biggest 5G BG is only 347.

4.9 Results

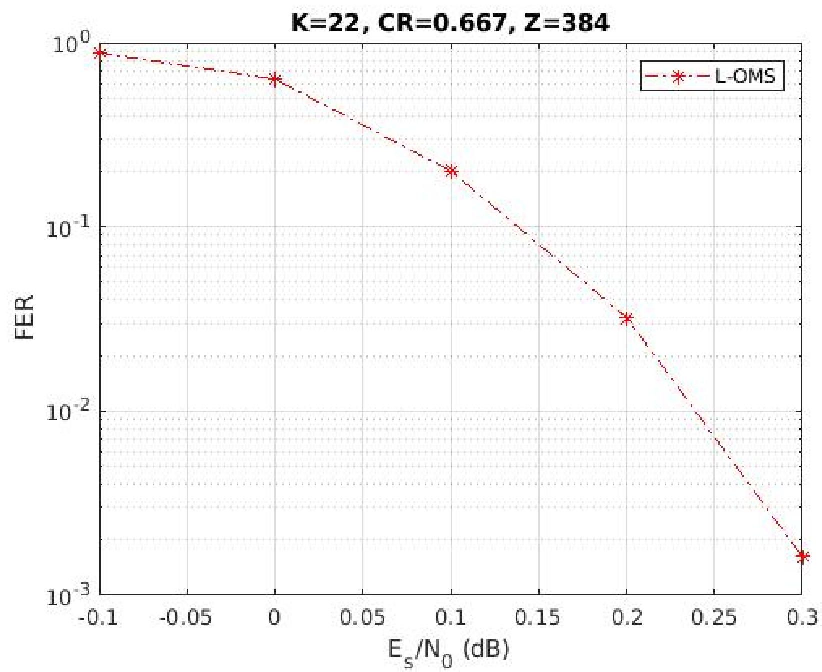
Coming to results, the 5G Eagle architecture was simulated on ModelSim SE-64 10.7c, with 10ns as clock period for the simulation. The scheduling was performed on MATLAB and the required sequence of commands was generated and stored in a file named stimuliHL.asc. The stimuliHL.asc file also contained the configuration in which the decoder has to operate (all the CMD word information 4.3 - lifting size, number of columns, number of codewords, output mode, operation mode etc.) and the input LLRs that the decoder receives from the channel (all generated using MATLAB). It also needs an expresp.asc file that contains the expected response (from MATLAB) after decoding is complete (which are in terms of bits).

The expected response generated from MATLAB matched the VHDL/RTL code response upon simulating for various lifting sizes, code rates and iterations. The performance curve is shown in figure 4.4,

Apart from the performance results we also obtain throughput results which we estimate by calculating the number of clock cycles needed by the decoder to complete the decoding for a single codeword. This will also give us the average number of clock cycles needed for any codeword with the same configuration since we are not using early termination, and all decoding is performed till the maximum number of iterations. The results are shown in table 4.6.



(a) K = 21, CR = 0.33



(b) K = 22, CR = 0.67

Figure 4.4: SNR v/s FER

Z	CR	N_{iters}	$N_{non-zero\ blocks}$	**clk cycles (simulation)**	clk cycles (analytical)	% extra cycles
384	0.33	15	316	5230	4740	10.34
288	0.33	15	316	5230	4740	10.34
176	0.33	15	316	5230	4740	10.34
60	0.33	15	316	5230	4740	10.34
4	0.33	15	316	5230	4740	10.34
352	0.2	15	197	3284	2955	11.13
256	0.2	15	197	3284	2955	11.13
112	0.2	15	197	3284	2955	11.13
28	0.2	15	197	3284	2955	11.13
5	0.2	15	197	3284	2955	11.13
5	0.33	15	316	5230	4740	10.34
5	0.4	15	265	4405	3975	10.81
5	0.5	15	210	3350	3150	12.70
5	0.55	15	188	3205	2820	13.65
5	0.67	15	144	2515	2160	16.43
5	0.55	13	188	2781	2444	13.79
5	0.55	11	188	2357	2068	13.94
5	0.55	9	188	1933	1692	14.24
5	0.55	7	188	1509	1316	14.67

Table 4.6: Throughput results

The equation 4.9 is again written below

$$clk\ cycles = N_{iters} \times (N_{non-zero\ blocks} + N_{stall}) + N_{pipeline}$$

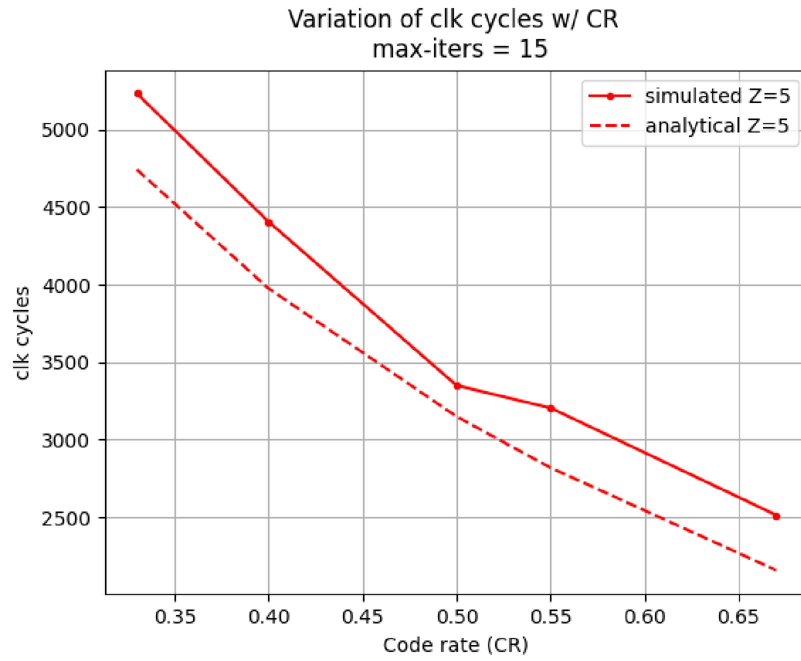
It can be seen from the plots that the number clock cycles depicts the equation 4.9 very closely. The throughput increases (clk cycles reduces) with increase in code rate due to the reduction in the number of non-zero blocks as the parity-check matrix shrinks (see figure 4.5). Due to parallel processing of the Z NCUs, the number of clock cycles remain constant and independent of lifting size. The throughput however, decreases with increase in number of maximum iterations (see figure 4.6).

The percentage of extra cycles needed compared to the analytical method is seen in the table 4.6. For changing lifting sizes, the percentage of extra cycles remains the same for reasons explained above. The percentage of extra cycles increases with the increase in code rate. This is mainly because, the number of stalling cycles will be higher for the initial layers of the base graphs (see figure 2.3) due to the higher density of the non-zero blocks in the initial layers. As a result of this, as the code rate increases, even though

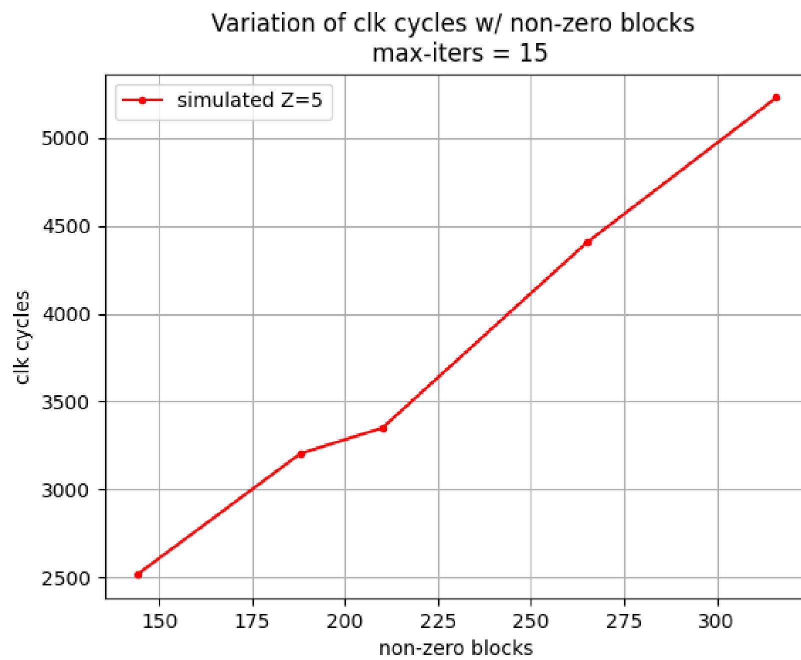
the number of non-zero blocks reduces, the percentage reduction in the stalling cycles is not much. Thus we see the increasing trend of percentage extra cycles. The increase in the percentage extra cycles with decrease in number of iterations - though minimal - is due to constant amount of cycles needed flushing, even though the number of iterations reduces.

4.10 Further optimizations and future work

We now have a functional QC-LDPC decoder for 5G, which includes most of the optimizations that were done in the Eagle [2] architecture. However, 5G NR standard is unique in its own way and poses several problems along with options to optimize further. Some of the optimizations that we will be performing in the near future are listed below.
As seen in the figure 4.5 and 4.6 the simulation throughput results are a little far from the analytical results from equation 3.1. We can bring these curves closer by modifying the scheduling and reducing the number of stalling cycles. One of the ways to achieve this is to reorder the way in which we process the layers so that we face minimum clashes, which can resolved further by reordering the VN processing. We can replace the cyclic shifter with a more efficient architecture that can enable higher degree of parallelism. As mentioned above, the current 5G Eagle setup does not incorporate the early termination scheme since it requires the value of the number of rows in the prototype matrix. This can range between 0 and 46 and this will need us to have 6 bits dedicated for that. Thus, we want to find ways to perform the early termination in more hardware-efficient manner. Lastly, we also want to find a way to reduce the number of NCUs and still maintain the same throughput.

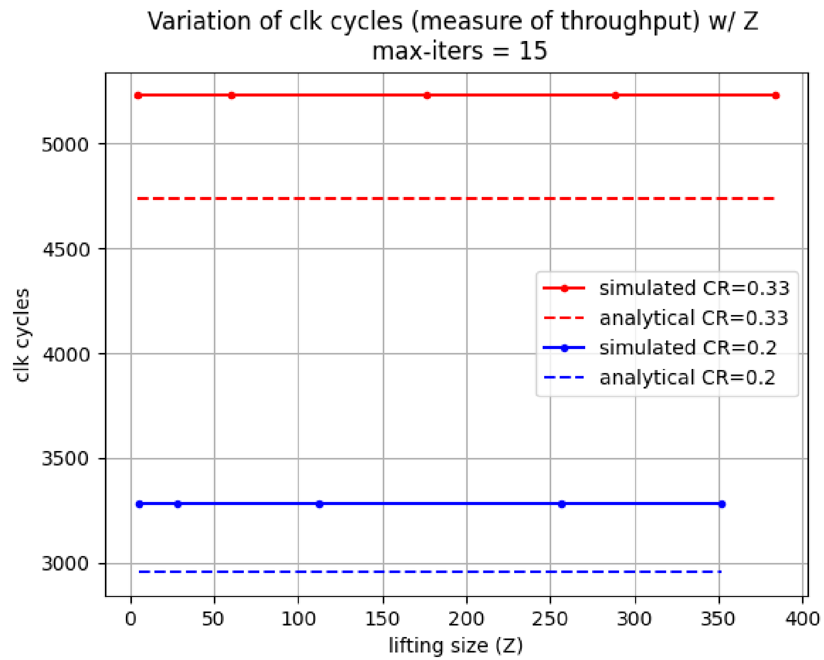


(a) clk cycles v/s CR

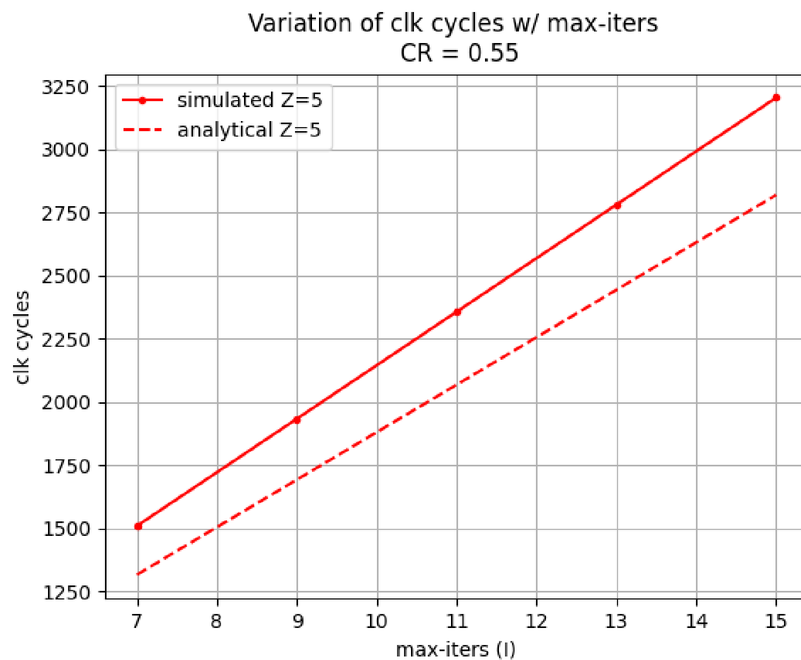


(b) clk cycles v/s NZ

Figure 4.5: clk cycles v/s CR and NZ



(a) clk cycles v/s Z



(b) clk cycles v/s iters

Figure 4.6: clk cycles v/s Z and iters

CHAPTER 5

CONCLUSION

The reference architecture Eagle [2] - with all its optimizations - provides a good starting point to develop the QC-LDPC decoder for 5G. However, due to the sporadic and large number of lifting sizes and code rates, we had to modify most of the units -mainly to accommodate the maximum lifting size, maximum number of non-zero blocks and maximum number of columns in $\mathbf{H_p}$ - to make it functional for 5G. We were able retain most of the optimizations performed, mainly the clock gating of groups to save power. The quantization of Q and T memories had to be increased by 2 bits to ensure good performance. The RTL code outputs agree with MATLAB version, and the performance is as expected. The throughput results look promising; however several optimizations can be further done to improve the efficiency in terms of power, area and throughput which were stated above.

REFERENCES

[1] William Ryan and Shu Lin. *Channel codes: classical and modern*. Cambridge university press, 2009.

[2] Christoph Roth. Design and vlsi implementation of a low-power quasi-cyclic ldpc decoder. *Integrated Systems Laboratory, ETH Zurich*, 2009.

[3] Chieh-Yu Lin, Li-Wei Liu, Yen-Chin Liao, and Hsie-Chia Chang. A 33.2 gbps/iter. reconfigurable ldpc decoder fully compliant with 5g nr applications. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[4] Jung Hyun Bae, Ahmed Abotabl, Hsien-Ping Lin, Kee-Bong Song, and Jungwon Lee. An overview of channel coding for 5g nr cellular communications. *APSIPA Transactions on Signal and Information Processing*, 8, 2019.

[5] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[6] Marc PC Fossorier. Quasicyclic low-density parity-check codes from circulant permutation matrices. *IEEE transactions on information theory*, 50(8):1788–1793, 2004.

[7] Christoph Studer, N Preyss, C Roth, and A Burg. Configurable high-throughput decoder architecture for quasi-cyclic ldpc codes. In *2008 42nd Asilomar Conference on Signals, Systems and Computers*, pages 1137–1142. IEEE, 2008.

[8] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[9] Wikipedia contributors. Wikipedia error detection schemes. `https://en.wikipedia.org/wiki/Error_detection_and_correction#Error_detection_schemes`. Accessed: 2021-07-19.

[10] Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

[11] Raj Chandra Bose and Dwijendra K Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and control*, 3(1):68–79, 1960.

[12] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[13] Jonghong Kim, Junho Cho, and Wonyong Sung. Error performance and decoder hardware comparison between eg-ldpc and bch codes. In *2010 IEEE Workshop On Signal Processing Systems*, pages 392–397. IEEE, 2010.

[14] Thomas J Richardson and Rüdiger L Urbanke. The capacity of low-density parity-check codes under message-passing decoding. *IEEE Transactions on information theory*, 47(2):599–618, 2001.

[15] Eran Sharon, Simon Litsyn, and Jacob Goldberger. An efficient message-passing schedule for ldpc decoding. In *2004 23rd IEEE Convention of Electrical and Electronics Engineers in Israel*, pages 223–226. IEEE, 2004.

[16] Christoph Roth, Pascal Meinerzhagen, Christoph Studer, and Andreas Burg. A 15.8 pj/bit/iter quasi-cyclic ldpc decoder for ieee 802.11 n in 90 nm cmos. In *2010 IEEE Asian Solid-State Circuits Conference*, pages 1–4. IEEE, 2010.