POLITECNICO DI TORINO

Master's Degree
in Electronic Engineering

Master's Degree Thesis

Exploring the impact of RCQ approach in an unrolled LDPC decoder

Supervisors
Prof. Guido Masera
Prof. Maurizio Martina
Dr. Vincenzo Petrolo

Candidate
Mario Porcaro

Academic Year 2023-2024

Summary

This thesis addresses the implementation of a high-performance Low-Density Parity-Check (LDPC) code decoder, crucial for achieving Ultra Reliable Low Latency Communication (URLLC) as envisioned by the 3GPP consortium. Future technologies like the tactile internet, autonomous vehicles, and telemedicine require robust Forward Error Correction (FEC) methods, and LDPC codes are ideal due to their rapid convergence and reliability in noisy environments. However, high-throughput LDPC decoders often face challenges related to area efficiency and routing congestion.

To overcome these limitations, this work introduces a fully reconfigurable, fully unrolled, and parallel LDPC decoder using the Reconstruction-Computation-Quantization (RCQ) paradigm. The architecture supports Offset-Min-Sum (OMS) and Min-Sum (MS) decoding algorithms, optimizing the use of fine-grain pipelining to achieve higher throughput. RCQ effectively reduces hardware area with minimal Frame Error Rate (FER) degradation, utilizing unique quantization parameters for each iteration.

Implemented in 65 nm CMOS technology, the proposed architecture uses a (648,540)-LDPC code from IEEE 802.11n, applying 3-bit quantization for min-sum messages across ten iterations with three distinct RCQ parameter sets: one for initial LLR quantization and reconstruction, one for the first five iterations and one for the last five.

The results demonstrate substantial improvements in both throughput and area efficiency compared to existing designs. This compact, high-throughput LDPC decoder provides a practical and scalable solution for next-generation communication systems, balancing performance and resource utilization to meet the stringent demands of modern multi-user scenarios.

Acknowledgements

I would like to offer my heartfelt thanks to the people who supported me in this important period of my life and helped me grow both professionally and personally.

Firstly, I am grateful to Prof. Guido Masera for his invaluable guidance and encouragement in bringing this work to fruition. I am also immensely thankful to Dr. Vincenzo Petrolo who guided me through every stage of the thesis work, from selecting the project's parameters to the testing phase of the completed work.

I would like to express my heartfelt gratitude to my parents for their unwavering emotional and financial support. Without their encouragement and belief in me, I would not have been able to complete my studies. Thank you for standing by me and supporting my decisions.

I would also like to express my deepest gratitude to my girlfriend, Maria, for her constant support. Your presence has been a source of inspiration and comfort, and I will always cherish the moments we have shared, from the carefree days spent together to the hours of intense study sessions by each other's side.

I would like to extend my heartfelt thanks to my dear friend Matteorocco for his steadfast support and for constantly motivating me to pursue self-improvement each and every day. Having a brother from another mother makes my life a lot more enjoyable.

I am sincerely grateful to my friends Stefano and Giuseppe for supporting me during a particularly challenging time and for introducing me to a new hobby. While our views on truth and justice may differ, our afternoon workouts are always something I eagerly look forward to.

I would like to express my gratitude to all the friends and colleagues met in these two years for the wonderful experiences we shared during my time in Turin. I had the privilege of meeting wonderful people like all of you.

Finally, a special thank you goes to both my father and my mother for teaching me the value of tough work and for showing me firsthand what resilience is: having the strength to fight and not lose yourself in the face of life's challenges.

MP

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Achieving Ultra Reliable Low Latency Communication is a critical goal in the development of next-generation communication systems, as URLLC plays a foundational role in supporting emerging applications that demand exceptional levels of reliability and responsiveness. For example, tactile internet requires real-time haptic feedback to enable users to interact with remote objects. Similarly, autonomous vehicles depend on ultra-reliable and low-latency communication to exchange data with other vehicles and infrastructure, ensuring safety-critical decisions are made in milliseconds to prevent accidents. Also telemedicine, particularly remote surgery, demands precise communication to transmit high-definition video, control signals, and feedback data, where any delay or error could have severe consequences.

In order to address these requirements, robust Forward Error Correction methods are essential to ensure that data can be transmitted accurately over noisy and unreliable channels. The prime candidates to meet all these requirements are LDPC codes, factoring in their ability to provide strong error-correction performance while maintaining high efficiency.

Unfortunately, the two main LDPC algorithm implementations, flooded and layered, each come with their own set of trade-offs. On one hand, flooded decoding provides high throughput but suffers from poor area efficiency and routing congestion. On the other hand layered decoding provides better area efficiency but can't reach high throughput due to its reduced parallelism, control complexity and memory bottlenecks, making it less suitable for URLLC applications.

In recent years, researchers have explored the application of new techniques to LDPC decoders, one of them being the Reconstruction-Computation-Quantization framework. RCQ employs dynamic, non-uniform quantization to maintain strong error performance with low message precision and limiting area and power consumption. In particular, the layer-specific RCQ decoding structure provides better FER performance and requires a smaller number of iterations than the original RCQ structure with the same bit width.

Inspired by that, this thesis work proposes the application of RCQ paradigm to a fully reconfigurable, fully unrolled and parallel LDPC decoder. The objective is retaining the advantages of the unrolled structure for flooded algorithm, like its high throughput and control simplicity, while mitigating its downsides, like high area consumption.

Chapter 2

Background

This chapter presents all the topics necessary for the complete comprehension of the work.

2.1 Low-Density Parity-Check codes

In information theory, a low-density parity-check (LDPC) code [3] is a linear error correcting code, used for message transmission over a noisy channel. LDPC codes are a special class of linear block codes. A binary LDPC code is represented by a sparse parity check matrix H counting M rows and N columns:

$$H = \begin{bmatrix} h_{0,0} & h_{0,1} & h_{0,2} & \dots & h_{0,n-1} \\ h_{1,0} & h_{1,1} & h_{1,2} & \dots & h_{1,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{m-1,0} & h_{m-1,1} & h_{m-1,2} & \dots & h_{m-1,n-1} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & \dots & 0 \\ 1 & 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & \dots & 1 \end{bmatrix}$$

N is the length of the codeword, and M is the number of parity bits, such that each element h_{mn} is either 0 or 1.

Sometimes the parity check matrix H is generated from the expansion of another matrix called base matrix H_{BASE}, defined as:

$$H_{BASE} = \begin{bmatrix} \prod_{0,0} & \prod_{0,1} & \prod_{0,2} & \cdots & \prod_{0,N_b-1} \\ \prod_{1,0} & \prod_{1,1} & \prod_{1,2} & \cdots & \prod_{1,N_b-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \prod_{M_b-1,0} & \prod_{M_b-1,1} & \prod_{M_b-1,2} & \cdots & \prod_{M_b-1,N_b-1} \end{bmatrix}$$

If the expansion factor (or lifting factor) is equal to z and the base matrix H_{BASE} has M_b rows and N_b columns, the expanded matrix H has $M_b \cdot z$ rows and $N_b \cdot z$ columns. The H_{BASE} matrix is expanded by replacing each entry $\prod_{i,j}$ with a z×z permutation matrix: these permutation matrices are obtained by a series of right shifts of the z×z identity matrix, where the shift amount is determined by $\prod_{i,j}$.

The complete H matrix can best be described by a Tanner graph (figure 2.1), a graphical representation of associations between code bits and parity checks. Each row of H

corresponds to a check node (CN), while each column corresponds to a variable node (VN) in the graph. An edge on the Tanner Graph connects a VN_j with CN_i only if the corresponding element h_{ij} is a 1 in H.

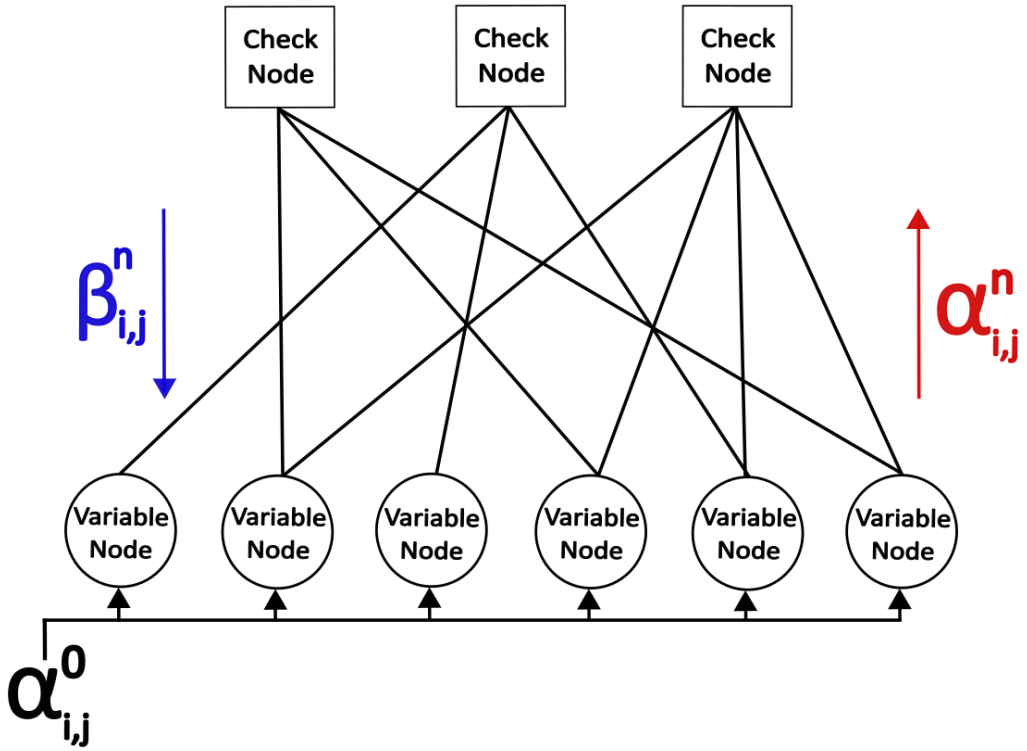


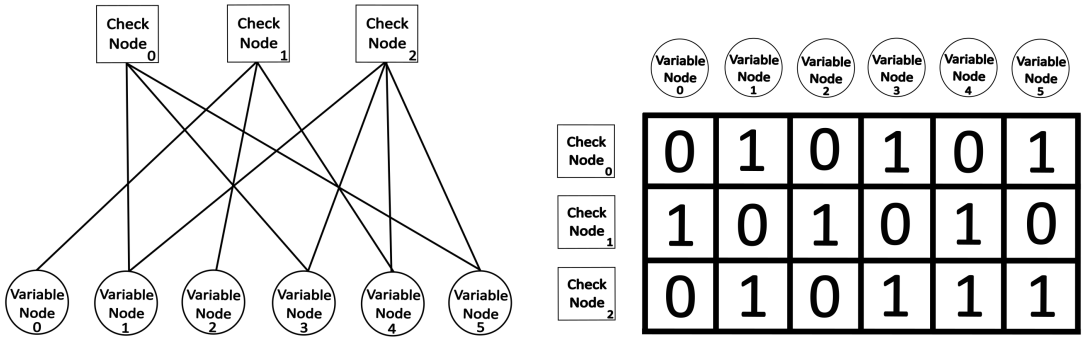
Figure 2.1: Tanner Graph representation

Check nodes represent a parity check equation, while variable nodes are representative of the elements in the codeword, meaning that each H matrix row introduces one parity check constraint on the input data vector $x = x_0, x_1, ..., x_{N-1}$:

$$H_i \cdot x^T = 0 \pmod 2.$$

Decoding in Low-Density Parity-Check codes works by using iterative message-passing algorithms on the Tanner graph. In order to better understand each step, the equivalence between Tanner graph representation and expanded H matrix is shown in figure 2.2.

For AWGN channels, messages are real-valued log-likelihood ratios (LLRs) representing the probability that a bit is 0 or 1 given the noise level (σ) in the channel. The mechanism is called Belief Propagation (BP), a general term used for the message-passing algorithm in LDPC decoding: it operates on the belief (likelihood) that a bit is either 0 or 1, updating these beliefs as messages are passed back and forth between variable and check nodes, aiming at refining the MAP estimations of the transmitted codeword.



(a) Tanner Graph (b) H matrix

Figure 2.2: Comparison between Tanner graph representation and expanded H matrix

2.2 Two-Phase Message Passing Algorithm

The received word is indicated as $y = [y_0, y_1, ..., y_{N-1}]$ meanwhile the transmitted codeword is $c = [c_0, c_1, ..., c_{N-1}]$.

This algorithm [1] can be described by the following 4 rules:

Initialization rule

At the start, the decoder receives the real-valued log-likelihood ratios from AWGN (y vector). They represent the probability that a bit is 0 or 1 given the noise level in the channel, so they can be described as:

$$\alpha_{i,j}^0 = ln\frac{P(VN_j = 0|y_i)}{P(VN_j = 1|y_i)} = \frac{2y_i}{\sigma^2} \tag{2.1}$$

These values are passed to the Check Nodes according to the Tanner Graph edges and the decoder is now ready to start belief propagation. The decoding process is done iteratively by passing messages between the variable nodes and check nodes along the edges of the Tanner graph, until a certain number of iterations is reached.

At the n-th iteration, let:

- $\alpha_{i,j}^n$ be the message coming from VN_j directed to CN_i ;
- $\beta_{i,j}^n$ be the message coming from CN_i directed to VN_j ;
- $M(j) = \{i : H_{ij} = 1\}$ be the set of Check nodes connected to VN_j;
- $N(i) = \{j : H_{ij} = 1\}$ be the set of Variable nodes connected to CN_i;

According to the standard Two-Phase Message-Passing (TPMP) algorithm, each node updates its value based on the messages it receives from its neighbors in the graph and according to the following rules.

Check Node Update Rule

Check Node update consists of a magnitude update and a sign update:

- Sign update is the same for every decoding algorithm
- Magnitude update depends on the type of decoding algorithm (ex. SP,MS,NMS,OMS)

for this work the decoding algorithm chosen is the Offset Min-Sum algorithm, which reduces the values of $\beta_{i,j}^n$ of a positive constant offset, giving improved performances compared to Sum-Product (SP) algorithm and Min-Sum (MS) algorithm.

The Check Node update rule for the OMS algorithm is:

$$\forall CN_i, i \in \{1, ..., M\} : \beta_{i,j}^n = sgn(\beta_{i,j}^n) \cdot |\beta_{i,j}^n| \tag{2.2}$$

with:

$$sgn(\beta_{i,j}^n) = \prod_{j' \in N(i) \setminus j} sgn(\alpha_{i,j'}^{n-1}) \tag{2.3}$$

$$|\beta_{i,j}^n| = max\{min_{j' \in N(i) \setminus j}\{|\alpha_{i,j'}^{n-1}|\} - \beta,0\} \tag{2.4}$$

where β is the offset.

This means that for each output message $\beta_{i,j}^n$:

- its sign is the product of the signs of every input message except the one with the same index as the output message itself
- its magnitude is the minimum of the magnitudes of every input message except the one with the same index as the output message itself, minus the positive offset β and capped to 0 in case of negative result.

The message exchange mechanism is depicted in figure 2.3 for each Check Node.

Variable Node Update Rule

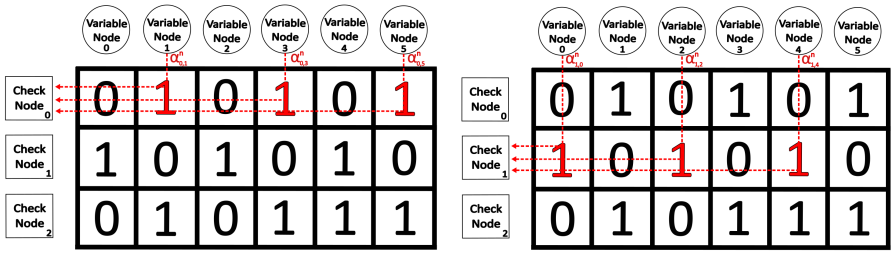
The Variable Node update rule is:

$$\forall VN_j, j \in \{1, ..., N\} : \alpha_{i,j}^n = \alpha_{i,j}^0 + \sum_{i' \in M(j) \setminus i} \beta_{i',j}^n \tag{2.5}$$

So each output message $\alpha_{i,j}^n$ its calculated as the sum of:

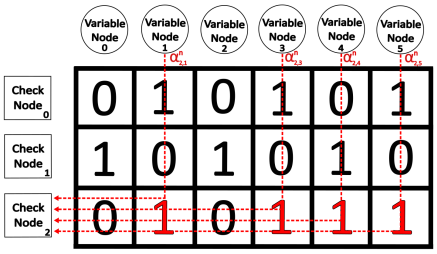
- the corresponding channel LLR
- every input message except the one with the same index as the output message itself

The message exchange mechanism is depicted in figure 2.4 for each Variable Node.



(a) Input messages for CN_0

(b) Input messages for CN_1



(c) Input messages for CN_2

Figure 2.3: Input messages collection for each Check Node

Decoding

At last iteration, for each codeword bit the a-posteriori LLR are computed as:

$$\alpha_j^n = \alpha_{i,j}^0 + \sum_{i' \in M(j)} \beta_{i',j}^n \tag{2.6}$$

In other word each a-posteriori LLR is obtained as the sum of the a-priori LLR plus every message coming from the connected Check Nodes.

From each a-posteriori LLR the reconstructed codeword bits $\hat{c} = [\hat{c}_0, \hat{c}_1, \hat{c}_2, ..., \hat{c}_{N-1}]$ are obtained as:

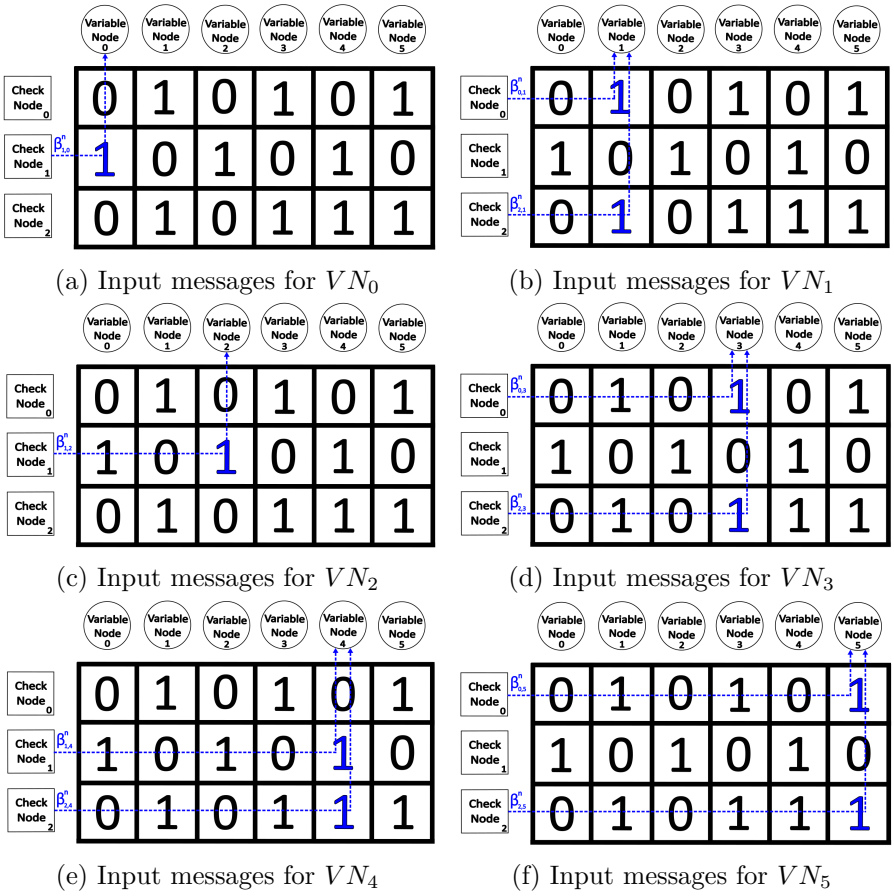
$$\hat{c}_j = \begin{cases} 0 & \text{if } \alpha_j^n > 0 \\ 1 & \text{else} \end{cases} \tag{2.7}$$

2.2.1 Clipping

Variable Node updated messages $\alpha_{i,j}^n$ are calculated using equation 2.5, so their dynamic range can be very large. In order to avoid overflow, a technique must be chosen. One of the possible approaches is called message clipping (MC) [8] and consists of decreasing the second term of the sum described by VN update rule, according to:

$$clip(r,t) = max\{min\{r, Q_{max} - t\}, -Q_{max} - t\} \tag{2.8}$$

which ensures that $\alpha_{i,j}^n \leq Q_{max}\ \forall(n,i,j)$. Q_{max} denotes the MC parameter and different values lead to different results.



(a) Input messages for VN_0

(b) Input messages for VN_1

(c) Input messages for VN_2

(d) Input messages for VN_3

(e) Input messages for VN_4

(f) Input messages for VN_5

Figure 2.4: Input messages collection for each Variable Node

2.3 LDPC Decoding Schedules

2.3.1 Flooded decoding

The flooded decoding schedule is the direct translation of the Tanner graph representation in a decoding algorithm: in a round of computation, all messages are updated and exchanged between Check Nodes and Variable Nodes simultaneously. Consequently, each iteration requires the input messages of all nodes to be readily available and decoding is completed in n iterations.

2.3.2 Layered decoding

Layered decoding algorithm is obtained [11] modifying the Variable Node update rule such as it can be merged with the Check Node update rule. The H matrix is then represented as the concatenation of l layers (called block rows), in practice dividing the single iteration into l sub-iterations (figure 2.5). In each sub-iteration only a subset of the Check nodes is

updated, such as a complete iteration is a full cycle through all the sequential parity checks in the graph. This means that Layered and Unrolled approaches have similar decoding arithmetic complexity, but Layered structure requires more time for computation and less memory to store intermediate messages.

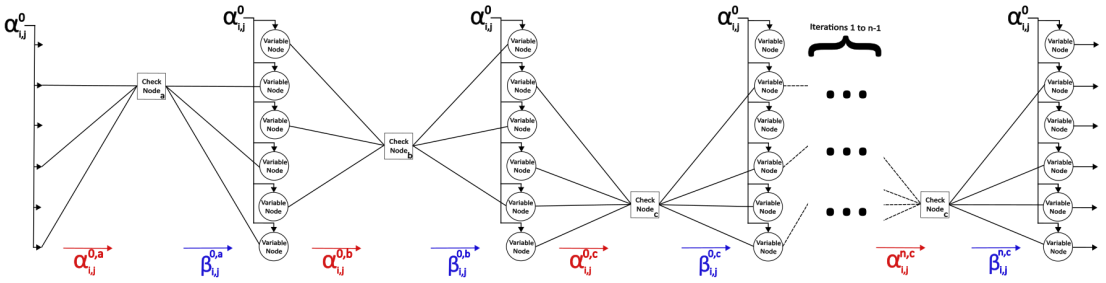


Figure 2.5: Layered Graph representation for n iterations, with $l = 3$ and each sub-iteration processing only one CN

2.4 Unrolled structure

Tanner graph representation supports the TPMP algorithm, that outputs a-posteriori LLRs after n iterations. The objective is transforming the Tanner Graph so that the computation can be done in one go, traversing n iteration of Check Node and Variable Node layers pair. These transformation is called Unrolling and consists of instantiating all the Check Nodes and Variable Nodes required by the computation and connecting them so that each Node is traversed only one time (figure 2.6). In this structure all the messages α and β traverse the graph in the same direction, meaning that timing optimization techniques can be applied, such as pipelining and retiming.

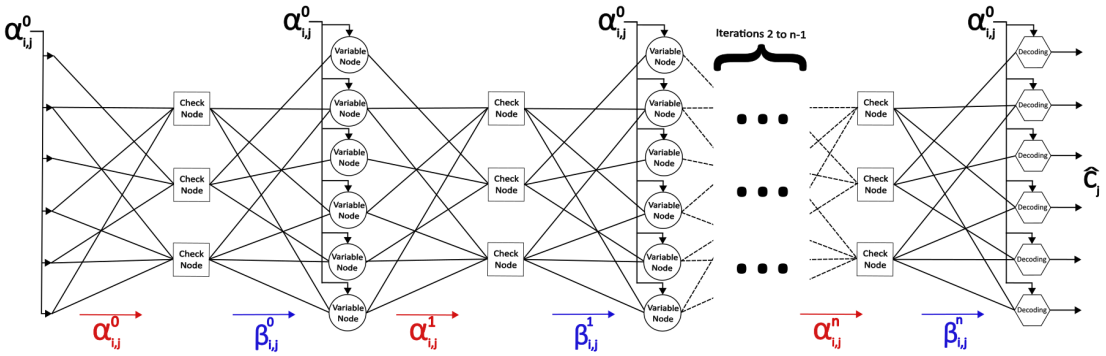


Figure 2.6: Unrolled Graph representation for n iterations

2.5 Reconstruction-Computation-Quantization (RCQ)

In standard message-passing decoding algorithm, messages exchanged between nodes in the LDPC code's Tanner graph are often quantized to reduce complexity and resource usage, especially in hardware implementations like FPGA or ASICs. Reconstruction-Computation Quantization (RCQ) [10] is a technique used to decode Low-Density Parity-Check (LDPC) codes by performing dynamic non-uniform quantization of messages during the decoding process.

Using a simple uniform quantization can result in performance degradation, particularly at low bit widths: applying the same quantization and reconstruction for all layers of an iteration leads to frame error rate (FER) degradation and a high average number of iterations. However, RCQ enables dynamic non-uniform quantization: the layer-specific RCQ decoding paradigm optimizes quantization and reconstruction for each layer of each iteration, significantly improving FER and reducing the number of decoding iterations compared to the original RCQ method.

Taking into account RCQ, the updating procedure of message passing algorithms becomes:

(1) Collection of input quantized messages

(2) Reconstruction of input messages

(3) Computation of output messages according to update rule

(4) Quantization of output messages

(5) Distribution of output quantized messages

Therefore, the generalized RCQ unit consists of 3 modules, represented in figure 2.7 :

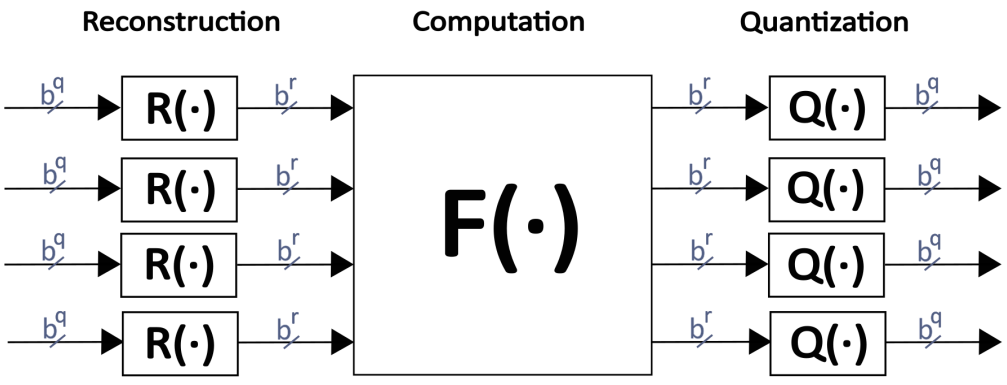


Figure 2.7: General scheme of RCQ elaboration unit

Reconstruction Module

Let the quantized message be represented by b^q bits and the reconstructed message be represented by b^r bits. The Reconstruction Module applies a function $R(\cdot)$ to each incoming quantized message to produce the reconstructed message, which has an higher bit length ($b^r > b^q$). When using layer-specific RCQ decoding, reconstruction function $R(\cdot)$ is different depending on layer, iteration and node type: $R_c^{(t,r)}$ and $R_v^{(t,r)}$ are the reconstruction functions of check node and variable node for layer r at iteration t. The reconstruction functions are mappings of the input external messages to log-likelihood ratios (LLR) that will be used by the node [9]. Under the assumption of a symmetric channel, a symmetric quantization function is implemented:

$$R(d) = [d^{MSB} R^*(\bar{d})] \tag{2.9}$$

where d is the quantized message, d^{MSB} its sign and $\bar{d}$ its magnitude.

Computation Module

The Computation module uses the outputs of reconstruction module to compute CN and VN updated messages according to their respective rules.The update rules are denoted as $F_c(\cdot)$ and $F_v(\cdot)$ for CN and VN respectively.

Quantization Module

Using the same notation for bit length, the quantization module applies the function $Q(\cdot)$ that quantizes the incoming b^r bits input messages into a b^q bits message, with $b^r > b^q$. Under the assumption of a symmetric channel, a symmetric quantization function is implemented:

$$Q(h) = [h^{MSB} Q^*(\bar{h})] \tag{2.10}$$

where h is the reconstructed message, h^{MSB} its sign and $\bar{h}$ its magnitude. The magnitude quantization function $Q^*(\bar{h})$ is based on a set of thresholds values $\{\tau_0, \tau_1, \tau_2 ... \tau_{max}\}$ with $0 \leq \tau_j \leq 2^{b^r-1} - 1$. These threshold are used to select one of the $2^{b^q-1} - 1$ possible indexes corresponding to a certain quantized value $j \in \{0,1, ...,2^{b^q-1} - 2\}$ following the rule:

$$Q^*(\bar{h}) = \begin{cases} 0, & \bar{h} \leq \tau_0 \\ j, & \tau_{j-1} < \bar{h} \leq \tau_j \\ 2^{b^q-1} - 1 & \bar{h} > \tau_{max} \end{cases} \tag{2.11}$$

with $0 < j \leq j_{max}$ and $j_{max} = 2^{b^q-1} - 2$. Both quantization and reconstruction rules are assumed to be monotonic without loss of generality.

Chapter 3

Related Work

LDPC codes are known for their strong error-correction performance, decoding using belief-propagation algorithms, and near-capacity performance. However, decoder's hardware implementation remains complex due to high computation demands and routing congestion, especially with longer code lengths. To tackle this problem, researchers are pushing toward new solutions regarding both type of coding schedules, flooded and layered. This chapter provides an overview of several approaches, followed by an in-depth exploration of the RCQ method.

3.1 Flooded schedule

The inherently advantage of the flooded algorithm over layered one its his suitability for hardware implementation where massive parallelism is feasible, as all nodes can be updated simultaneously.

In [7] the implemented decoder is capable of supporting multiple code rates (1/2, 2/3, 3/4, and 5/6) and achieves a throughput of up to 941.8$Mbps$ with a latency of less than 6μs. It simplifies the parallel decoding process using blockwise partitioning and regularity in the parity-check matrix of Structured LDPC codes: codes are partitioned into sub-blocks of size $P = 27$, where each non-null sub-block is a rotation of the diagonal identity matrix. Both Variable Node Processors and Check Node Processors are designed to operate in a serial fashion. Serial VNs update messages using a sequential accumulation, incorporating delay sections to handle inputs based on node degrees and minimize idle cycles during continuous decoding. Serial CNs employ a double-recursion scheme with forward and backward processing branches to compute check-to-variable messages, reducing operations by utilizing shared intermediate results. A programmable Message Vector Switch, implemented using a Banyan network, performs rotations on message vectors to facilitate efficient memory access during processing. Additionally, a paging factor of $G = 6$ divides memory into 36 pages, enabling high-speed decoding.

However, throughput is still limited due to serial processing used for Variable Node and Check Node update functions. Considering that unrolled implementations have no hardware reuse, they can achieve extremely high decoding speeds by completing all iterations

in a single hardware pass. This makes them ideal for applications demanding ultra-low latency, such as ultra-reliable low-latency communications (URLLC). An example of the high throughput of fully unrolled fully parallel structures is shown in [5]: all decoding iterations of the MS decoding scheme are instantiated in hardware, eliminating iterative control and enabling simultaneous processing across iterations and graph nodes. Using a 4-bit fixed point quantization for both input LLRs and internal messages, this decoder is able to achieve ultra-high throughput of $550Gbps$, but it's main downside is its significant hardware area, reaching $4.98mm^2$ even using $28nm$ CMOS technology.

3.2 Layered schedule

The main advantage of the layered decoding schedule hardware implementations is the significantly fewer hardware resources required, making them more suitable for resource-constrained platforms like FPGAs. Moreover, asynchronous updates between layers adapts the number of iterations dynamically based on convergence criteria, optimizing latency and energy consumption.

The architecture implemented in [6] is a full row-based layered architecture with frame-interleaved scheduling that achieves ultra-high throughput while maintaining hardware efficiency. The decoder supports multiple coding rates and implements various optimization strategies: it uses compressed and super-compressed layers, reducing the number of effective layers, and enhances pipeline utilization by processing multiple frames simultaneously (frame-interleaved scheduling). The pipeline consists of 3 stages: reading input messages from storage, Check Node update function and message update for storage.

However, area can be further reduced through the use of alternative strategies. The solution implemented in [9] achieves multi-Gbps throughput while maintaining a small hardware footprint, offering significant improvements in area efficiency and energy efficiency compared to existing solutions. It uses a fully pipelined structure, using barrel shifters to rotate messages between variable nodes and check nodes efficiently. It reduces area overhead with 2 strategies: processing longer frames in multiple cycles using the barrel shifter of the smallest frame and reduces the quantization bit-width from 5 to 4 bits for check node computations.

Another possible solution is reducing the high latency associated with layered decoders by modifying the LDPC decoding algorithm , as described in [12]. The authors proposed a new type of algorithm called Reweighted Offset Min-Sum (ROMS), which distinctive trait is dynamically adjusting message values during decoding to suppress unreliable messages caused by short cycles in the LDPC code structure. This new algorithm reduces the number of iterations required for decoding, significantly lowering latency without increasing hardware complexity. The message values adjustment is done introducing a reweighting factor, determined based on message's connection to short cycles, which tells the processing units (both CNs and VNs) to update their messages prioritizing inputs from constructive nodes and suppress overconfident ones. The new architecture is further optimized to:

- reduce number of multipliers, applying reweighting only to selected messages in the minimum function (ROMSO1)

- minimizing hardware overhead , storing reweighting activation indicators at check-node level (ROMSO2)

The fully optimized ROMSO2 decoder minimizes complexity while maintaining high performance, achieving faster convergence with minimal hardware overhead.

3.3 Reconstruction-Computation-Quantization

The paper [10] introduces Reconstruction-Computation-Quantization (RCQ) as a framework for efficient decoding with reduced bit width. RCQ employs dynamic, non-uniform quantization to maintain strong error performance with low message precision and limiting area and power consumption.

The paper presents the following concepts:

3.3.1 Layer-specific RCQ Decoding structure

The main difference between the original RCQ approach and the layer-specific RCQ decoder is that this approach quantization and reconstruction parameters are designed for each layer of each iteration independently. The layer-specific RCQ decoder provides better FER performance and requires a smaller number of iterations than the original RCQ structure with the same bit width, with the cost of an increased number of parameters that need to be stored in the hardware.

3.3.2 Layer-specific RCQ Parameter Design

The layer-specific RCQ parameters are designed using hierarchical dynamic quantization (HDQ). For each layer of each iteration, layer-specific HDQ discrete density evolution separately computes the Probability Mass Function (PMF) of the messages, distinguishing quantizers and reconstructions. HDQ offers a low-complexity method to design quantizers that maximize mutual information.

3.3.3 FPGA-based RCQ Implementations

The implementation section presents 3 different alternatives to distribute RCQ parameters efficiently in an FPGA:

- **Lookup**: Utilizes pre-stored mappings, but requires substantial memory.
- **Broadcast**: Centrally stores parameters and distributes them dynamically, reducing memory demands.
- **Dribble**: Incrementally updates parameters, balancing memory and communication overhead.

Results showed that a 3-bit layer-specific RCQ decoder reduced LUTs and routed nets by over 10%, and register usage by 6%, compared to a 5-bit offset Min-Sum decoder, with similar performance.

3.3.4 RCQ Decoding Structure

As described in previous chapter, the RCQ framework incorporates three modules:

- **Reconstruction**: Converts low-bit external messages to higher-bit internal messages using mappings derived from density evolution.
- **Computation**: Applies the arithmetic update function, which depends on node's type and algorithm
- **Quantization**: Reduces the internal message back to lower-bit external representation for transmission to neighboring nodes.

Layer-specific RCQ optimizes these processes to improve decoding performance and minimize hardware demands. In particular, the authors applied RCQ to 2 different structures:

IEEE 802.11 Standard LDPC Code (Flooding Schedule):

The investigate structure was a 4-bit RCQ decoder with a flooding schedule using IEEE 802.11n LDPC codes, with N=1296 and k=648 and a maximum of 50 iterations. It showed superior performance over Min-Sum and comparable performance to floating-point belief propagation (BP) at high signal-to-noise ratios. So the RCQ decoder defensively reduced error floors by slowing the convergence of message magnitudes.

QC-LDPC Code (Layered Schedule):

The code considered was a quasi-regular LDPC code, with all VNs having degree 4 and CNs having degree 29 and 30, with N=9472, k=8192 and number of layers equal to 10. Among all the layered schedule decoders analyzed, the layer-specific RCQ decoder significantly outperformed traditional RCQ in both FER and iteration count and solved the "representation mismatch problem," ensuring better alignment of message distributions across layers.

Chapter 4

Motivation

In this chapter we highlight the motivation and the contributions of this work.

4.1 LDPC decoders implementation problems

Both flooded and layered decoding strategy implementation suffer of some sort of intrinsic issues.

On the one hand flooded decoders have:

- **High Latency**, since they require multiple iterations to converge
- **Hardware Complexity**, requiring significant parallelism and suffering routing congestion to simultaneously update all nodes

On the other hand layered decoders have:

- **Limited Parallelism and Throughput**, since layers are processed sequentially
- **Control Complexity**, as layered schedules require precise management of dependencies between layers
- **Memory Access Overheads**, because stored LLRs are updated frequently

As shown in previous chapter, RCQ decoding and in particular Layer-specific RCQ are able to maintain strong error performances, limiting area and power consumption at the same time.

Therefore, the goal is to apply the RCQ strategy to a fully unrolled, fully parallel structure, a novel approach that has not been explored before. Implementing RCQ in an unrolled structure is particularly compelling because every iteration is instantiated, making a layer-dependent approach highly practical. Additionally, one of the key benefits of RCQ is area reduction, addressing the major drawback of unrolled structures: their massive area requirements. By combining RCQ with an unrolled structure, the advantages of the unrolled design are retained, such as high throughput, the absence of a complex control unit, the limited number of memory accesses and a deep pipeline, while also improving code performance and significantly reducing area and latency.

Chapter 5

Methodologies

This chapter outlines the methodologies employed in the design and implementation of the proposed decoder. The two main sections detail the development of a high-level functional model in C++ for simulating and validating the decoding algorithm, and its subsequent translation into RTL code using SystemVerilog.

5.1 C++ code

This section describes the C++ model of the decoder, providing the algorithmic representation of its operations and enabling validation of its functionalities.

5.1.1 decoder.cpp structure

The decoder is modeled as a class and the main methods are:

- the *constructor*, which reads all significant information from a file to initialize the decoder instance
- the *decode* method, which performs the TPMP algorithm reading input LLRs from a vector and writing decoded bits into another one.

The input LLRs are $[HorizontalIterations \cdot LiftingFactor]$ integers with bit width *LLRChannelBits* and their number is equal to the number of Variable Nodes. On the other hand the Check Node number is $[NumLayers \cdot LiftingFactor]$.

The main data structure used to model the message exchange mechanism is the *Messages* matrix, which has $[NumLayers \cdot LiftingFactor]x[HorizontalIterations \cdot LiftingFactor]$ dimensions and each cell is an integer with bit width *DWIDTH*.

Both structures are initialized by the constructor, which reads from a file the following data:

- on the first row are indicated in this order the quantities *HorizontalIterations*, *NumLayers*, *LiftingFactor*.

- on the following rows is represented the H_{base} matrix, so a matrix with dimensions [*NumLayers*]x[*HorizontalIterations*] and whose elements are integers with value between *0* and *LiftingFactor-1*

The H_{base} matrix is stored in the *qc_matrix*. The input LLRs are copied from the given vector to a local vector which expands their bit width from *LLRChannelBits* to *DWIDTH*, meanwhile the *Messages* matrix is initialized at all zeros.

The *decode* method contains all the phases of the TPMP described before. In detail:

Initialization

During the first iteration, input LLRs written in the local vector are passed to the *Messages* matrix. In figure 5.1 is depicted an example using a matrix of small dimensions: each α_j is written in all corresponding $M_{i,j}$ if the corresponding element of the expanded matrix $H_{i,j}$ is equal to 1. Otherwise the cell value is left initialized at 0.

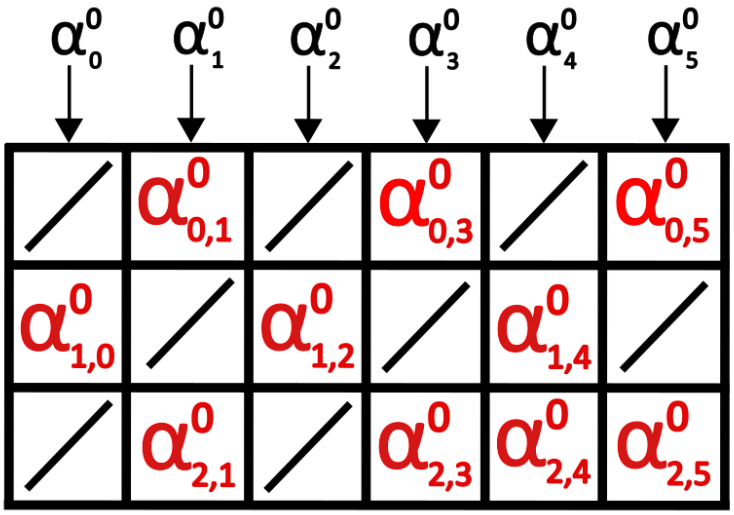


Figure 5.1: Messages matrix initialization

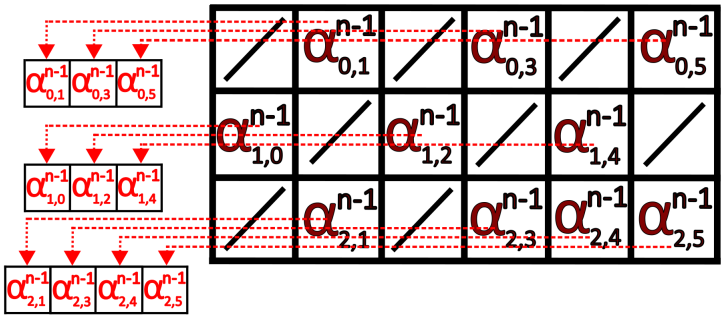
Check Node Update

At each iteration, for every Check node, the Check node update rule is implemented in 3 phases (figure 5.2):

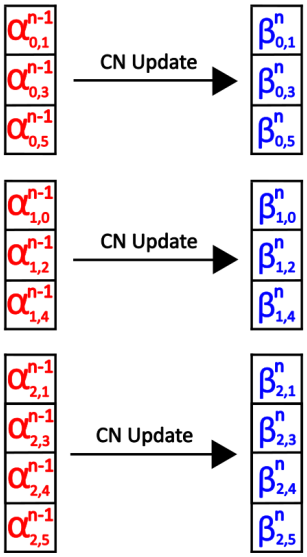
1. All input messages to the same CN are collected in an array: for every column of the expanded matrix, if $H_{i,j} = 1$ (connection present) the corresponding message stored in the *Messages* matrix is pushed in the array

2. The vector containing all messages is updated using the *cnUpdate* method: it searches minimum and subminimum applies the equations 2.2, 2.3 and 2.4.

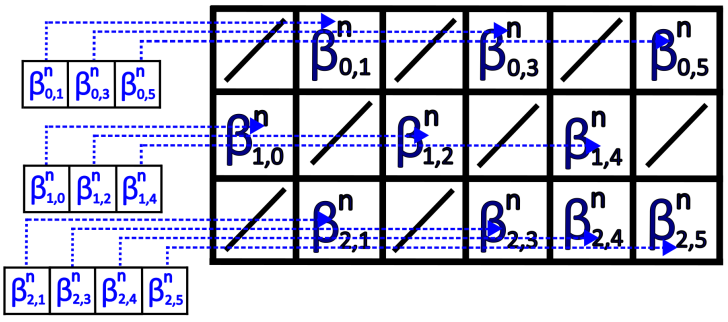
3. Output messages are now distributed in the same cells visited before, popped back in reverse order (to preserve the same indices)



(a) Messages collection



(b) Messages update



(c) Messages distribution

Figure 5.2: CN update rule implementation

Variable Node Update

At each iteration excluding the last one, for every Variable node, the Variable node update rule is implemented in 3 phases (figure 5.3):

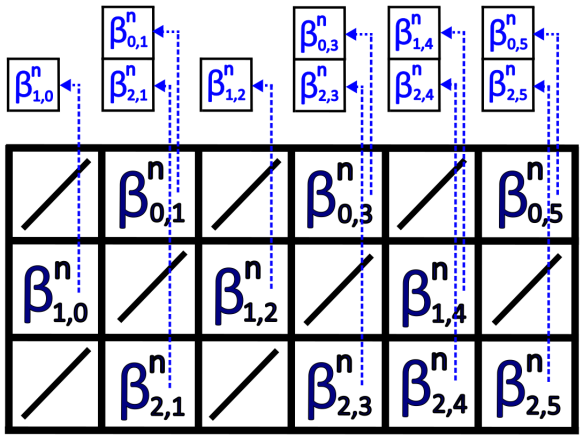
1. All input messages to the same VN are collected in an array: for every row of the expanded matrix, if $H_{i,j} = 1$ (connection present) the corresponding message stored in the *Messages* matrix is pushed in the array

2. The vector containing all messages is updated using the *vnUpdate* method: it sums all input messages and for each output subtracts the corresponding input. Lastly it adds each partial sum to the corresponding input LLR applying the clipping rule, according to equation 2.5.

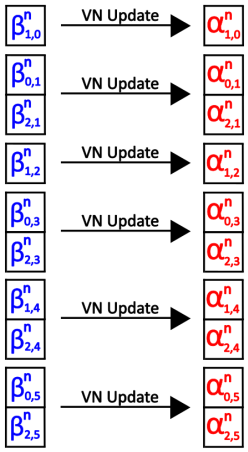
3. Output messages are now distributed in the same cells visited before, popped back in reverse order (to preserve the same indices)

Decoding

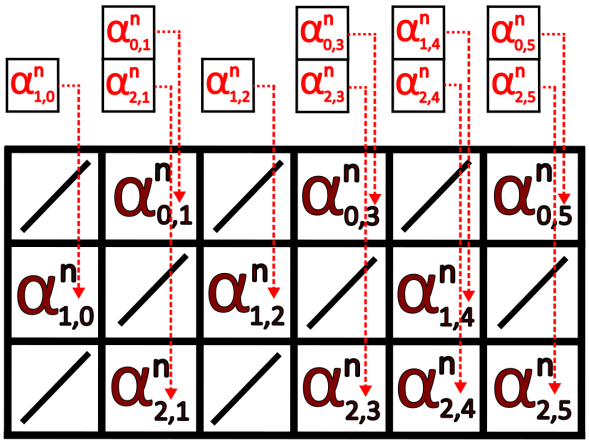
At last iteration a-posteriori LLRs are computed: for each VN, all input messages to the same VN are added together, and each partial sum is added to the corresponding input LLR using clipping rule. A-posteriori LLRs are used to update the initial LLR vector, meanwhile decoded bits are determined from their signs, according to equation 2.7.



(a) Message collection



(b) Messages update



(c) Messages distribution

Figure 5.3: VN update rule implementation

Algorithm improvement

Code shown before can be improved exploiting the base matrix, stored in *qc_matrix*. As explained in previous chapter, the expanded matrix can be obtained from the base matrix, where each number represents the amount of consecutive right shifts to apply to the identity matrix with dimensions equal to the lifting factor. The cells containing -1 are expanded as all zeros. An example of expansion is depicted in figure 5.4 where the $LiftingFactor = 3$.

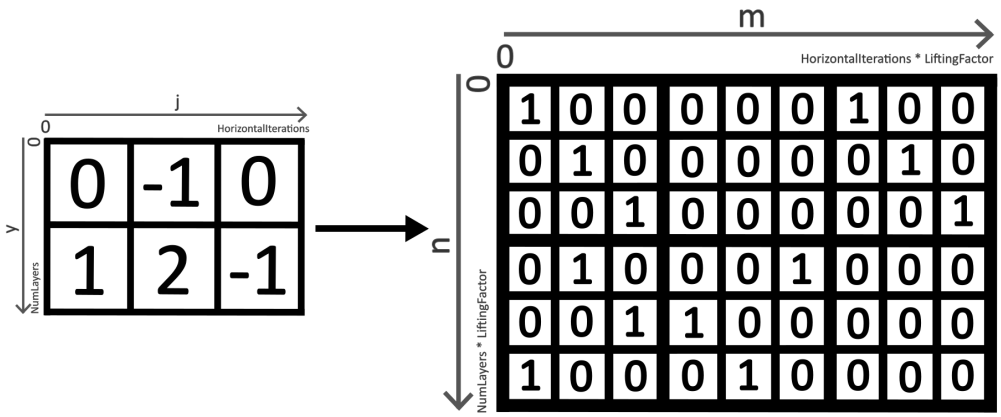


Figure 5.4: Base matrix expansion

Knowing that each expanded cell is a permutation of the identity matrix, each Variable node is connected to either 0 or 1 Check node of each expanded cell and the same goes for each Check node with respect to Variable nodes. Exploiting this piece of information it's possible to collect and distribute messages in a more efficient way: instead of visiting each cell of the expanded matrix, it's sufficient to visit only the cells of the base matrix, and where there's a connection ($H_{base-a,b} \neq -1$) calculate the right index.

Considering the m-th Check node, its index can be written as $m = j \cdot LiftingFactor + z$ where j is the index used to iterate on the Base matrix rows (from 0 to $NumLayers - 1$) and z goes from 0 to $LiftingFactor - 1$. For each column y of the base matrix (from 0 to $HorizontalIterations - 1$) it's necessary to check if a connection is present, so check if $qc\_matrix[j][y] \neq -1$. If so, the number contained in that specific cell is used to calculate the right column index n:

$$n = y \cdot LiftingFactor + ((z + qc\_matrix[j][y])\%LiftingFactor)$$

So the message that needs to be collected is located in cell $Messages[m][n]$, in other words:

$$Messages[j \cdot LiftingFactor + z][y \cdot LiftingFactor + ((z + qc\_matrix[j][y])\%LiftingFactor)]$$

An example can help to better understand the mechanism: suppose to have the same base matrix depicted in figure 5.4 and to be interested in collecting all messages directed

to Check node with index 4. Because $LiftingFactor = 3$ and $m = j \cdot LiftingFactor + z$, $j = 1$ and $z = 1$ are deduced. Next step is to iterate on base matrix columns with index y:

- (y=0) $\longrightarrow$ $qc\_matrix[1][0] = 1$ so message is located in $Messages[1 \cdot 3 + 1][0 \cdot 3 + ((1 + 1)\%3)] = Messages[4][2]$
- (y=1) $\longrightarrow$ $qc\_matrix[1][1] = 2$ so message is located in $Messages[1 \cdot 3 + 1][1 \cdot 3 + ((1 + 2)\%3)] = Messages[4][3]$
- (y=2) $\longrightarrow$ $qc\_matrix[1][2] = -1$ so there is no connection

Checking on the expanded matrix it's true that Check Node 4 is connected to Variable Nodes 2 and 3. Using this method the number of comparisons done for each Check node update decreases from $HorizontalIterations \cdot LiftingFactor$ to just $HorizontalIterations$.

Same upgrade can be applied to VNs: considering the n-th Variable node, its index can be written as $n = y \cdot LiftingFactor + z$ where y is the index used to iterate on the Base matrix columns (from 0 to $HorizontalIterations - 1$) and z goes from 0 to $LiftingFactor - 1$. For each column y of the base matrix (from 0 to $NumLayers - 1$) check if a connection is present ($qc\_matrix[j][y] \neq -1$). If so, the number contained in that specific cell is used to calculate the right row index n:

$$m = j \cdot LiftingFactor + ((z - qc\_matrix[j][y])\%LiftingFactor)$$

So the message that needs to be collected is located in cell $Messages[m][n]$, in other words:

$$Messages[j \cdot LiftingFactor + ((z - qc\_matrix[j][y])\%LiftingFactor)][y \cdot LiftingFactor + z]$$

To correct problems with the modulus (%) operator, the expression

$$((z - qc\_matrix[j][y])\%LiftingFactor)$$

is replaced with

$$((LiftingFactor + z - qc\_matrix[j][y])\%LiftingFactor)$$

The operation remains the same, but it's a mandatory correction in order to avoid to pass a negative number to modulus, which forces it to output 0 in any case.

Using the same base matrix as before (figure 5.4), in this case the objective is to collect all messages directed to Variable node with index 2. Knowing that $LiftingFactor = 3$ and $n = y \cdot LiftingFactor + z$, $y = 0$ and $z = 2$ are deduced. Next step is to iterate on base matrix rows with index j:

- (j=0) $\longrightarrow$ $qc\_matrix[0][0] = 0$ so message is located in $Messages[0 \cdot 3 + (3 + 2 - 0)\%3][0 \cdot 3 + 2] = Messages[2][2]$
- (j=1) $\longrightarrow$ $qc\_matrix[1][0] = 1$ so message is located in $Messages[1 \cdot 3 + (3 + 2 - 1)\%3][0 \cdot 3 + 2] = Messages[4][2]$

Checking on the expanded matrix it's true that Variable Node 2 is connected to Check Nodes 2 and 4. Using this method the number of comparisons done for each Variable node update decreases from $NumLayers \cdot LiftingFactor$ to just $NumLayers$.

5.2 RTL

This section outlines the methodology and reasoning applied in implementing the decoder using SystemVerilog HDL.

5.2.1 Configuration files

In order to completely define the decoder, some configuration files are necessary: the main configuration file is *configs.sv* and is modified by the user, while the other configuration files are generated using Python scripts from user given text files.

configs.sv

This SystemVerilog file contains a set of parameters used for decoder definition, both for its structure and for its internal operations. All the significant parameters can be divided in 3 groups and they are here reported with the value they assume for this particular work.

The parameters defining decoder's structure are:

- $LiftingFactor = 27$
- $HorizontalIterations = 24$
- $NumLayers = 4$
- $MaxIterations = 10$
- $CodeLength = HorizontalIterations * LiftingFactor = 648$
- $NumRows = NumLayers * LiftingFactor = 108$
- $InformationBits = CodeLenght - NumRows = 540$

The parameters defining decoder's internal operations are:

- $OFFSET = 0$, which is the offset used by OMS check node update rule. In this particular case the algorithm becomes MS.
- $DWIDTH = 7$
- $LLRChannelBits = 7$
- $QMaxPos = 2^{DWIDTH-1} - 1 = 63$, positive threshold used in clipping.
- $QMaxNeg = -2^{DWIDTH-1} + 1 = -63$, negative threshold used in clipping.

The parameters defining the RCQ approach are:

- $LLRChannelBitsQuant = 3$, defining the bit lenght used for the first Quantization and Reconstruction of LLRs coming from AWGN channel

- *LLRQuantVersion* = 1, defining the version of the table used for the first Quantization and Reconstruction of LLRs
- *RCQ_bits*[25] = '{3,3,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,4,4,4,4,4}
- *RCQ_table_sel*[25] = '{2,2,2,2,2,3,3,3,3,3,1,1,1,1,1,1,1,1,1,1,1,1,1}

The last two vectors decide for each iteration how many bits and which version of quantization and reconstruction tables are used. So, for example, iteration 2 used the quantization and reconstruction tables 7 $\longrightarrow$ 3 bits and 3 $\longrightarrow$ 7 bits respectively, both with version 2 of the tables. For this work, the code supports a maximum number of iteration of 25, 3 different versions for 3 bits Q/R tables and 1 version for 4 bits Q/R tables. However, as stated before, only 10 iterations are considered, so:

- The first Q/R of channel LLRs is performed on 3 bits using version 1 of the tables
- Q/R for iterations 1 to 5 is done on 3 bits using version 2 of the tables
- Q/R for iterations 6 to 10 is done on 3 bits using version 3 of the tables

These versions of the tables were the ones available for those specific bit-length Q/R at the time this project was developed, and their configuration was optimized to minimize both BER and FER (more details in Experimental results chapter).

Python scripts

Some python script are used to create SystemVerilog files useful for decoder definition, enabling complete adaptability to any parameters, H matrix and RCQ tables.

init_params.py

This script generates SystemVerilog files used for decoder's nodes definition and connection. It reads the text file containing the base matrix, in this case *seq_mem_wifi_648_56.txt*, and generates:

- *matrix.sv* : defines the base matrix (*qc_matrix*) and 2 vectors containing the indexes of the CN connected to each VN sector and vice-versa (*CNs_connections* and *VNs_connections*)
- *CN_degrees.sv* : defines the vector of all CN degrees (*CNs_DEG*) , the vector with the incremental sums of the CN degrees (*CNs_limits*) and the vector of all CN internal structure sizes (*CNs_SIZE*).
- *VN_degrees.sv* : defines the same vectors described before but for VNs, so *VNs_DEG*, *VNs_limits* and *VNs_SIZE*.

One the one hand, all these vectors are necessary to connect CN and VN layer using specific blocks called C2V_connection, V2C_connection and Initial layer. On the other hand, only "DEG" and "SIZE" vectors are necessary to instantiate all the nodes with the right degree and right calculation network size. All the elements in the SIZE vector are derived from the elements of DEG vector, finding for each the nearest power of 2 greater than the corresponding DEG element. More details are provided in subsequent sections.

table_generator.py and table_rom_generator.py

These two scripts are used to generate SystemVerilog files that define the Quantization and Reconstruction modules as simple ROM memories.

table_generator.py takes as input the file "roms_content.txt", which is made of rows of reconstructed values for R tables and rows of quantization thresholds for Q tables. These values are used to write text files called *table_[IN]to[OUT]_[V].txt* with:

- IN = input bits, for the memory are the address bits
- OUT = output bits, for the momery are the data bits
- V = version of the table

For this work 8 tables are generate, 6 for R/Q at 3 bits and 2 for R/Q t 4 bits, but only the first 6 are actually used in 10 iterations. Each table file includes all the values to be written to each ROM, organized in rows and arranged in the correct order.

table_rom_generator.py takes all the *table_[IN]to[OUT]_[V].txt* files and writes their content into the corresponding Verilog file *rom_[In]to[OUT]_[V]_generated.v* that describes a ROM using block RAM resources. All Verilog files implementing a reconstruction ROM are stored in the "R_layer" folder, while those defining a quantization ROM are stored in the "Q_layer" folder.

5.2.2 Check Node

The generic Check Node of degree DEG has DEG input messages and computes the corresponding DEG output messages using a certain LDPC decoding algorithm.

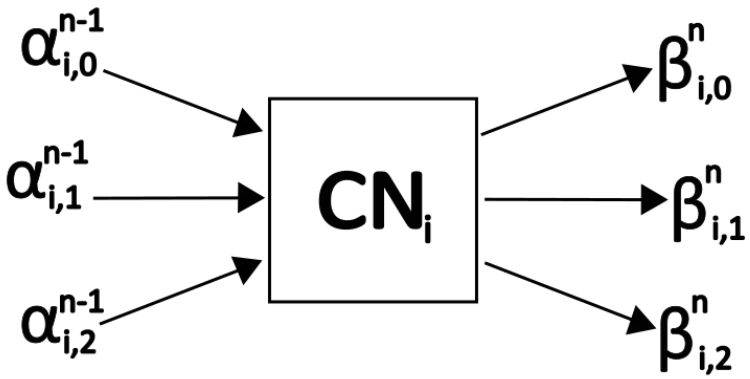


Figure 5.5: Check Node entity

As explained in previous chapter, for this work the operation performed by Check nodes is:

$$\forall CN_i, i \in \{1, ..., M\} :$$
$$\beta_{i,j}^n = sgn(\beta_{i,j}^n) \cdot |\beta_{i,j}^n|$$

with:

$$sgn(\beta_{i,j}^n) = \prod_{j' \in N(i) \setminus j} sgn(\alpha_{i,j'}^{n-1})$$
$$|\beta_{i,j}^n| = max\{min_{j' \in N(i) \setminus j}\{|\alpha_{i,j'}^{n-1}|\} - \beta, 0\}$$

and it's called OMS (Offset Min-Sum) Algorithm where β is the offset.

Magnitude calculation

For each output message it's necessary to evaluate the quantity $min_{j' \in N(i) \setminus j}\{|\alpha_{i,j'}^{n-1}|\}$, which is the minimum among every input messages except the one with the same index as the output message. It's trivial to notice that, having N inputs, N-1 outputs will have the value of the minimum, and the one with the same index of to the minimum input message will have the value of the sub-minimum. Therefore, two approaches are viable:

- (1) calculate the N minima of the N subsets of inputs obtained removing one different element each time, The index of the removed inputs corresponds to the index of the output.
- (2) calculate minimum and sub-minimum of the N inputs, then search for the index of the minimum, then feed minimum and sub-minimum to the right output.

For both the approaches a parallel calculation is used, as the serial calculation would take N-1 clock cycles (storing a temporal variable and compare it with each element), and the goal is to evaluate Check Node's output messages in a combinatorial manner.

First approach

For the first approach no control structure is used, instead the objective is extracting the N minima of the N subsets of inputs obtained removing one different element each time, using an architecture as simple as possible. In order to find a regular structure circuits for number of even inputs are analyzed.

For N=2 the structure (figure 5.6) consist of only 2 wires, because the 2 subsets contain just 1 element each.

For N=4 the architecture (figure 5.7) has 2 layers: the first layer computes the 2 minima of the subsets containing 2 inputs each, meanwhile the second one computes the 4 minima of the subsets of 3 elements.

By looking at the structure for N=6 and N=8, depicted in figure 5.8, it's possible to understand how a generic structure would work: a first section computes the minima

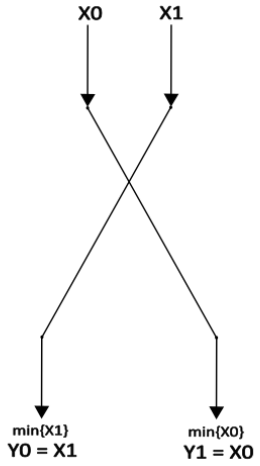


Figure 5.6: First approach architecture for N=2

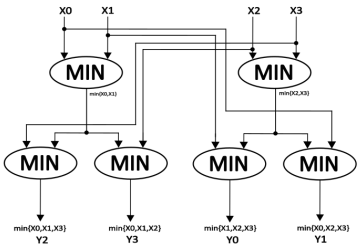


Figure 5.7: First approach architecture for N=4

of subsets of inputs, a second section combines the partial results obtained in the first section to complete the computation. For even numbers that are powers of 2 the structure becomes much more regular: the first section becomes 2 trees of min blocks that compute the minima of half inputs each, meanwhile the second section becomes an inverse tree that combines all the partial results. For this reason, from now on only $N = 2^k$ number of inputs will be considered, as every other number of inputs can be obtained by using the nearest power of 2 structure and feeding the max representable value to unused inputs (so that computation won't be affected).

For N=16 (figure 5.9) it's possible to highlight the regularity of the architecture, which can be obtained from the previous power of 2 by adding a top layer and a bottom layer of min blocks.

To compare this solution with the other one, the critical path delay and area will be calculated.

The critical path delay can be expressed as:

$$t_1 = (log_2(N) - 1) \cdot t_{min} + (log_2(N) - 1) \cdot t_{min} = \\ = 2(log_2(N) - 1) \cdot t_{min}$$

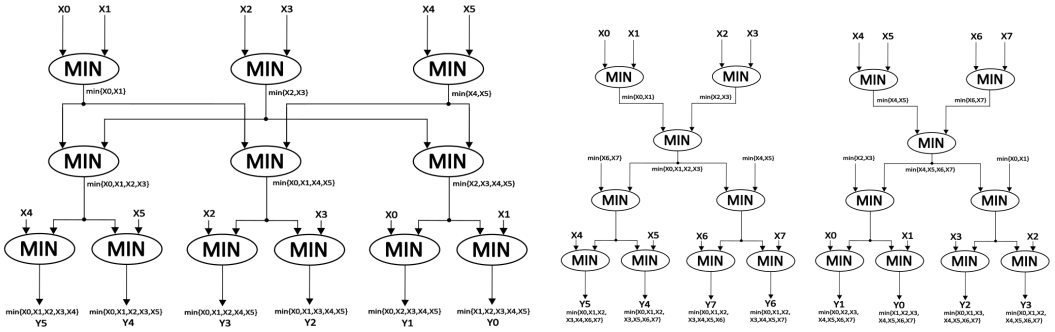


Figure 5.8: First approach architecture for N=6 and N=8

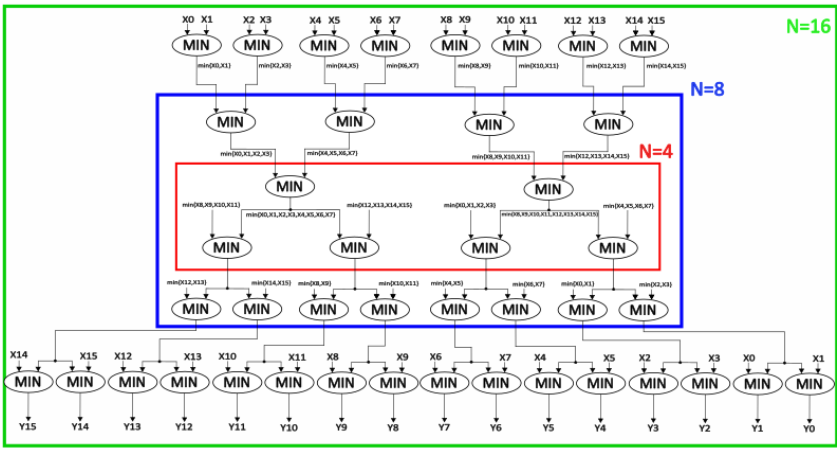


Figure 5.9: First approach architecture for N=16

Knowing that $t_{min} = t_{comp} + t_{mux}$, the critical path delay is:

$$t_1(N) = 2(log_2(N) - 1) \cdot (t_{comp} + t_{mux}) \quad \text{with} \quad N = 2^k \tag{5.1}$$

The total area can be evaluated as the sum of areas of the elementary blocks that compose the total architecture (ignoring net area):

$$\begin{aligned} A_1 &= \left(\sum_{i=1}^{k-1} \frac{N}{2^i}\right) \cdot A_{min} + \left(\sum_{i=0}^{k-2} \frac{N}{2^i}\right) \cdot A_{min} = \left[2\left(\sum_{i=0}^{k-1} \frac{N}{2^i}\right) - N - \frac{N}{2^{k-1}}\right] \cdot A_{min} = \\ &= \left[2N\left(2 - \left(\frac{1}{2}\right)^{k-1}\right) - N - \frac{N}{2^{k-1}}\right] \cdot A_{min} = \left[(2^{k+2} - 4) - 2^k - 2\right] \cdot A_{min} = \\ &= \left[4 \cdot 2^k - 2^k - 6\right] \cdot A_{min} = \left[3 \cdot 2^k - 6\right] \cdot A_{min} = (3N - 6) \cdot A_{min} \end{aligned}$$

Knowing that $A_{min} = A_{comp} + A_{mux}$, total area is:

$$A_1(N) = (3N - 6) \cdot A_{comp} + (3N - 6) \cdot A_{mux} \quad \text{with} \quad N = 2^k \tag{5.2}$$

Second approach

In order to find a regular structure that can extract minimum and sub-minimum from a certain set of numbers, circuits for small powers of 2 are analyzed.

For N=2 the structure (figure 5.10) is the same as in the first approach because having only 2 inputs one is the minimum and the other is the sub-minimum.

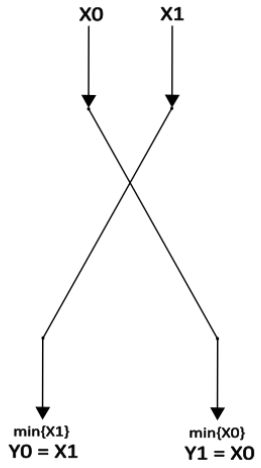


Figure 5.10: Second approach architecture for N=2

For N=4 the structure becomes more complex: in order to avoid unnecessary comparisons, the inputs can be divided in pairs, knowing that in a set of 2 elements one is the minimum and the other is the sub-minimum. To distinguish between the two a swap block is implemented: as shown in figure 5.11, the swap block feeds the inputs to the output ports based on a comparison of the two.

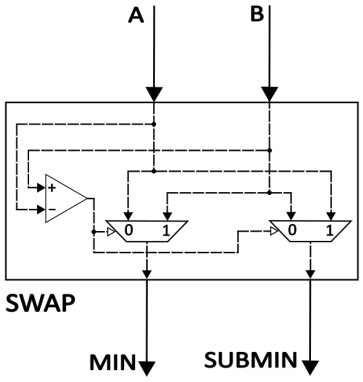


Figure 5.11: Swap block structure

By using 2 swap blocks on the four inputs, two pairs of minima and sub-minima are obtained, from which the global minimum and sub-minimum are to be calculated. For this

task a MIN/SUB block is implemented: as shown in figure 5.12, the global minimum is the smallest between the 2 input minima, meanwhile the sub-minimum must be obtained from the cross comparison of opposing input minimum and sub-minimum.

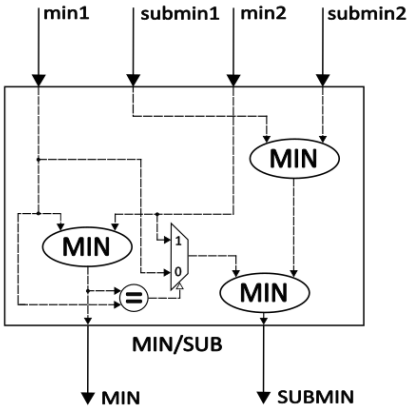


Figure 5.12: MIN/SUB block structure

Lastly, the 2 numbers obtained must be fed to the right output, and this is done comparing the global minimum to each input, finding the right position for the sub-minimum. The total structure is portrait in figure 5.13.

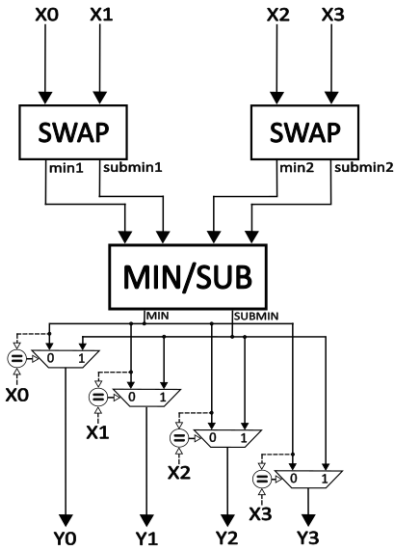


Figure 5.13: Second approach architecture for N=4

For N=8 the architecture is shown in figure 5.14, from which is easy to extract direct formulas for both delay and area as functions of the number of inputs.

Assuming $N = 2^k$ is the number of inputs, the total structure is composed of:

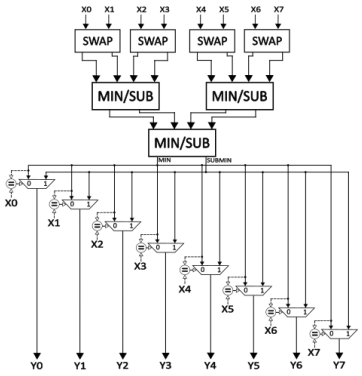


Figure 5.14: Second approach architecture for N=8

- a single layer of N/2 swap blocks;
- a k-1 layer tree of min/sub blocks;
- a layer of N comparators and MUXs pairs;

So the critical path delay can be expressed as:

$$\begin{aligned} t_2 &= t_{swap} + (log_2(N) - 1) \cdot t_{min/sub} + t_{comp} + t_{mux} = \\ &= t_{comp} + t_{mux} + (log_2(N) - 1) \cdot (2 \cdot t_{min} + t_{comp} + t_{mux}) + t_{comp} + t_{mux} \end{aligned}$$

Knowing that $t_{min} = t_{comp} + t_{mux}$, the critical path delay is:

$$t_2(N) = (3 \cdot log_2(N) - 1) \cdot (t_{comp} + t_{mux}) \quad \text{with} \quad N = 2^k, k \geq 2 \tag{5.3}$$

The total area can be evaluated as the sum of areas of the elementary blocks that compose the total architecture (ignoring net area):

$$\begin{aligned} A_2 &= \frac{N}{2} \cdot A_{Swap} + \left(\sum_{i=0}^{k-2} 2^i \right) \cdot A_{Min/Sub} + N \cdot (A_{comp} + A_{mux}) \\ &= \frac{N}{2} \cdot (A_{comp} + 2 \cdot A_{mux}) + (2^{k-1} - 1) \cdot A_{Min/Sub} + N \cdot (A_{comp} + A_{mux}) \\ &= \frac{3N}{2} \cdot A_{comp} + 2N \cdot A_{mux} + (2^{k-1} - 1) \cdot (3 \cdot A_{min} + A_{comp} + A_{mux}) \end{aligned}$$

Knowing that $A_{min} = A_{comp} + A_{mux}$, total area is:

$$A_2(N) = \frac{3N}{2} \cdot A_{comp} + 2N \cdot A_{mux} + \left(\frac{N}{2} - 1 \right) \cdot 4 \cdot (A_{comp} + A_{mux})$$

$$A_2(N) = \left(\frac{7N}{2} - 4 \right) \cdot A_{comp} + (4N - 4) \cdot A_{mux} \quad \text{with} \quad N = 2^k, k \geq 2 \tag{5.4}$$

Final choice

N	t_1	t_2	A_1	A_2
4	$\mathbf{2 \cdot t_{min}}$	$5 \cdot t_{min}$	$\mathbf{6 \cdot A_{comp} + 6 \cdot A_{mux}}$	$10 \cdot A_{comp} + 12 \cdot A_{mux}$
8	$\mathbf{4 \cdot t_{min}}$	$8 \cdot t_{min}$	$\mathbf{18 \cdot A_{comp} + 18 \cdot A_{mux}}$	$24 \cdot A_{comp} + 28 \cdot A_{mux}$
16	$\mathbf{6 \cdot t_{min}}$	$11 \cdot t_{min}$	$\mathbf{42 \cdot A_{comp} + 42 \cdot A_{mux}}$	$52 \cdot A_{comp} + 60 \cdot A_{mux}$
32	$\mathbf{8 \cdot t_{min}}$	$14 \cdot t_{min}$	$\mathbf{90 \cdot A_{comp} + 90 \cdot A_{mux}}$	$108 \cdot A_{comp} + 124 \cdot A_{mux}$
64	$\mathbf{10 \cdot t_{min}}$	$17 \cdot t_{min}$	$\mathbf{186 \cdot A_{comp} + 186 \cdot A_{mux}}$	$220 \cdot A_{comp} + 252 \cdot A_{mux}$

Table 5.1: Comparison between the 2 approaches in term of delay and area

Table 5.1 sums up the result obtained analysing the two approaches for some values of N: it's clear that the first approach is the best one, having both a smaller critical and a smaller area.

The only issue is managing all the connection between minimum blocks, which can get difficult at coding level. This problem is solved introducing a new basic block called "Minimum_unit", which contains 3 min blocks organized as shown in figure 5.15.

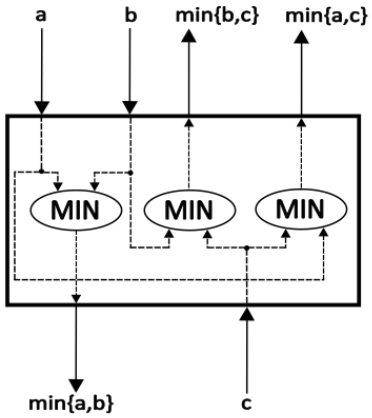


Figure 5.15: Minimum unit internal architecture

Using this unit as a basic block, the scheme is reorganized and is represented in figures 5.16 and 5.17 for the cases N=4, N=8 and N=16;

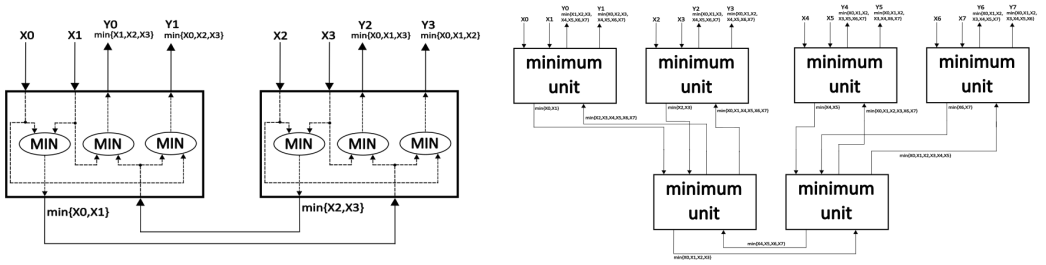


Figure 5.16: Final architecture of minimum network for N=4 and N=8

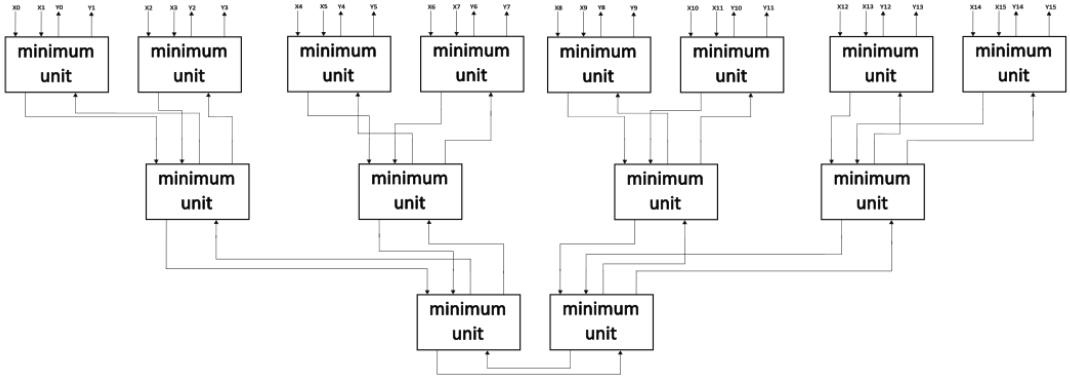


Figure 5.17: Final architecture of minimum network for N=16

Now the minimum network is organized as a tree that is traversed in both directions:

- The forward path computes the 2 minima of each N/2 inputs sub-sets
- The backward path combines the 2 minima with each "local" partial minimum, feeding the right output to the corresponding input index

This structure is totally equivalent to the one shown in first approach, and it can be easily demonstrated.

The critical path delay is:

$$t_{crit} = t_{forward} + t_{backward} = (log_2(N) - 1) \cdot t_{min} + (log_2(N) - 1) \cdot t_{min} = \\ = 2(log_2(N) - 1) \cdot t_{min} = t_1$$

The total area is:

$$A = 3 \cdot \left(\sum_{i=1}^{k-1} \frac{N}{2^i} \right) \cdot A_{min} = 3N \cdot \left(1 - \left(\frac{1}{2} \right)^{k-1} \right) \cdot A_{min} = \\ = 3 \cdot (2^k - 2) \cdot A_{min} = (3N - 6) \cdot A_{min} = A_1$$

To sum up:

$$t_{crit}(N) = 2(log_2(N) - 1) \cdot (t_{comp} + t_{mux}) \quad \text{with} \quad N = 2^k \tag{5.5}$$

$$A(N) = (3N - 6) \cdot A_{comp} + (3N - 6) \cdot A_{mux} \quad \text{with} \quad N = 2^k \tag{5.6}$$

Using the OMS algorithm, the results obtained from the minimum network are then added to the parameter $-\beta$, defined in *configs.sv*, and then eventual negative values are capped to 0 using a set of maximum blocks.

Sign calculation

Computing the output signs means calculating the product of all signs of inputs except for the one with the same index as the output. So the approach is the same used for magnitude calculation, using a different basic operation in spite of the $min\{\}$ operation.

x	1	-1
1	1	-1
-1	-1	1

XOR	0	1
0	0	1
1	1	0

Table 5.2: Comparison between product of signs and XOR operations

As shown in table 5.2 , it's easy to find a correlation between the XOR operation and the product between numbers that are either 1 or -1: the product of signs is just a XOR operation in which a "positive" corresponds to 0 and "negative" corresponds to 1. Using a XOR gate is much more efficient in terms of delay and area than using a multiplier, so the selected approach is taking the sign bit of each number and feed it to the XOR network, whose basic blocks are implemented like shown in figure 5.18.

A XOR unit is simply a minimum unit in which the minimum block is replaced by a XOR gate. The total architecture is identical to the magnitude calculator: an example of is shown in figure 5.19, that represents the sign calculator for N=4.

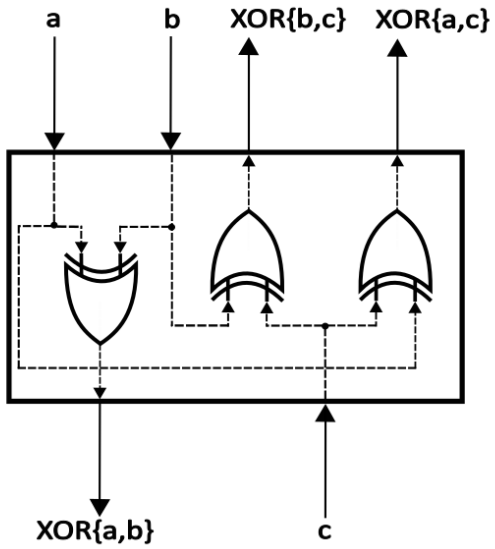


Figure 5.18: XOR unit implementation

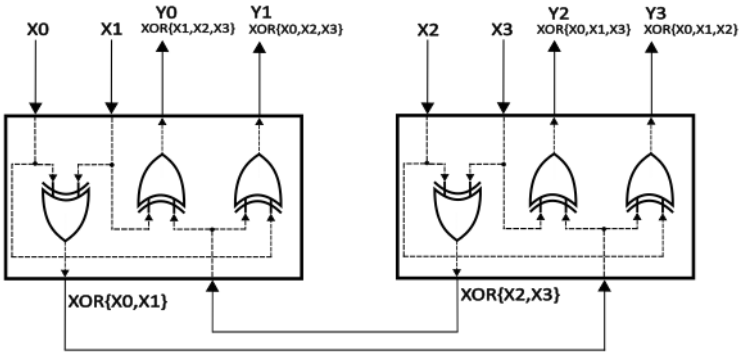


Figure 5.19: XOR network for N=4

Implementation of CN

The final structure of the CN is shown in figures 5.20 and 5.21, using as examples DEG=3 and DEG=5. It's possible to notice that both Minimum and XOR network have a size equal to the nearest power of 2 greater than the number of inputs ($DEG = 3 \longrightarrow SIZE = 4$ and $DEG = 5 \longrightarrow SIZE = 8$). The OMS algorithm is implemented calculating minima's magnitudes with the Minimum network, then feeding each output to an adder that sums the offset $-\beta$ (defined as an internal parameter in *configs.h*) and then to a max block that caps eventual negative values at 0. Finally, magnitude and sign are matched together and sent to the right output port.

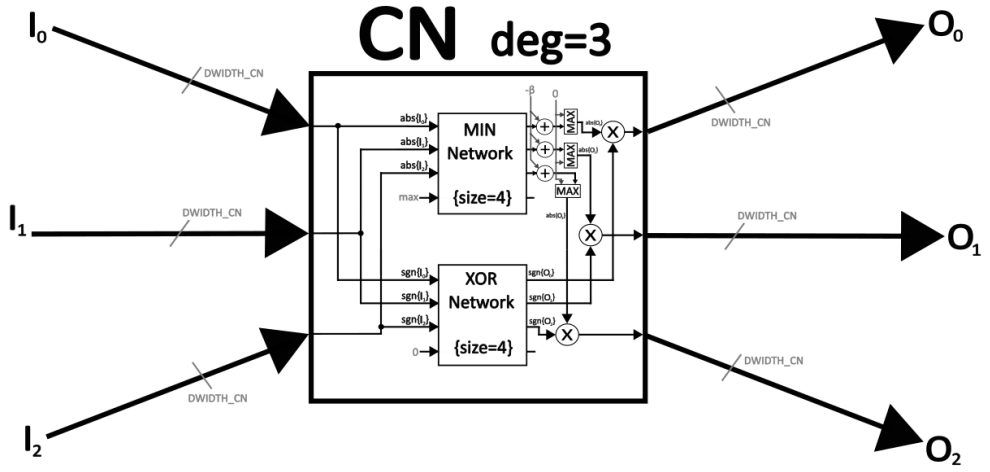


Figure 5.20: Check Node architecture for DEG=3

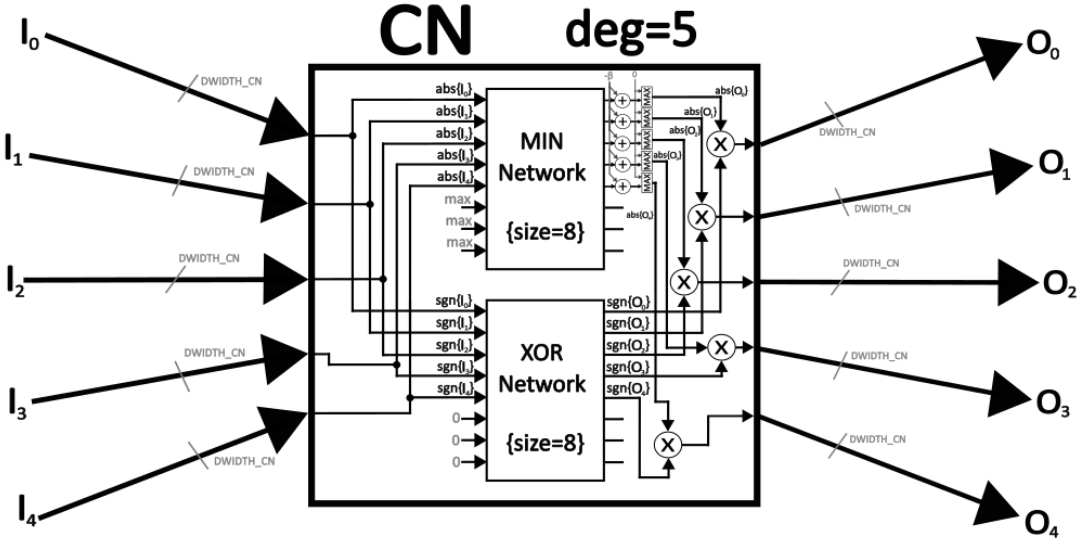


Figure 5.21: Check Node architecture for DEG=5

5.2.3 Variable Node

The generic Variable Node of degree DEG takes as inputs DEG messages coming from the Check Nodes and one LLR coming from the channel and computes the corresponding DEG output messages.

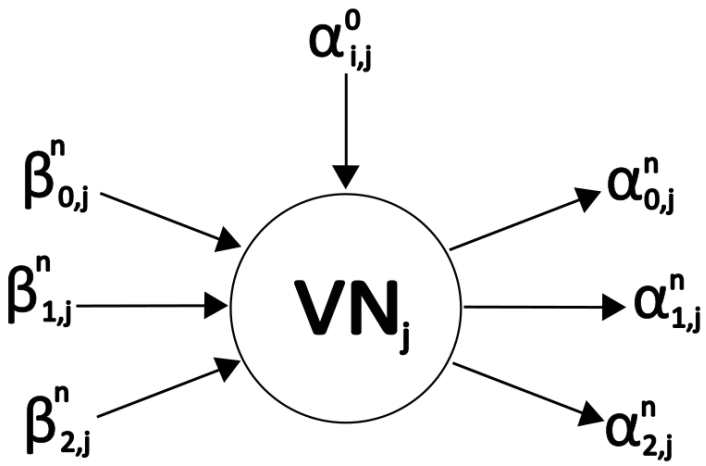


Figure 5.22: Variable Node entity

As explained in previous chapter, for the operation performed by Variable nodes is:

$$\forall VN_j, j \in \{1, ..., N\} :$$
$$\alpha^n_{i,j} = \alpha^0_{i,j} + \sum_{i' \in M(j) \setminus i} \beta^n_{i',j}$$

Considering the j-th VN, the generic output message with index i is computes as the sum between every input messages except the i-th and the j-th LLR coming from AWGN channel.

Exploiting the similarities between the operation done by Check and Variable nodes, the minimum network implemented before can be adopted with slight modifications. In particular, to compute the N sums of N-1 input messages it's sufficient to modify the basic component of the network, which becomes the adder unit (figure 5.23).

The adder unit consists of 3 adders with different bit width, depending on their position in the adder network. Figure 5.24 shows an example of adder network with N=4:

In general, having $N = 2^k$ inputs, the adder network presents k-1 layers, meaning that the critical path delay is:

$$t_{crit} = t_{forward} + t_{backward} = (k-1) \cdot t_{add} + (k-1) \cdot t_{add} =$$
$$= 2(k-1) \cdot t_{add} = 2(log_2(N) - 1) \cdot t_{add}$$

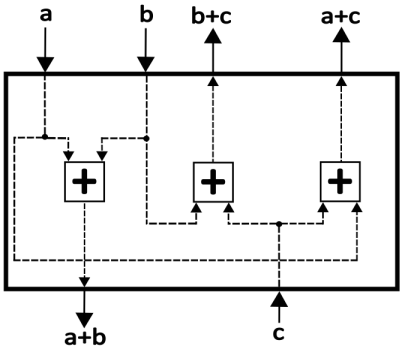


Figure 5.23: Adder unit internal architecture

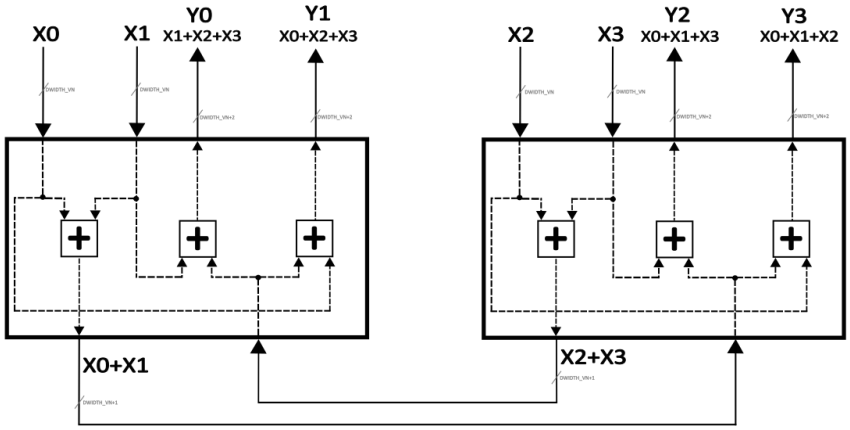


Figure 5.24: Adder network structure for N=4

Each layer is made of a number of adder units halved with respect to the previous one, so the total area is:

$$A = \left(\sum_{i=1}^{k-1} \frac{N}{2^i}\right) \cdot A_{AdderUnit} = 3 \cdot \left(\sum_{i=1}^{k-1} \frac{N}{2^i}\right) \cdot A_{add} =$$
$$= 3N \cdot \left(1 - \left(\frac{1}{2}\right)^{k-1}\right) \cdot A_{add} = 3 \cdot (2^k - 2) \cdot A_{add} = (3N - 6) \cdot A_{add}$$

As expected, area and delay expressions are similar to the one found for the min network (equations 5.5 and 5.6):

$$t_{crit}(N) = 2(log_2(N) - 1) \cdot t_{add} \quad \text{with} \quad N = 2^k \tag{5.7}$$

$$A(N) = (3N - 6) \cdot A_{add} \quad \text{with} \quad N = 2^k \tag{5.8}$$

Now each adder network outputs has to be added to the channel LLR corresponding to the j-th VN. The bit length of the channel LLR is $LLRChannelBits$, meanwhile the bit length of the partial sum is $DWIDTH + 2 \cdot (log_2(N) - 1)$. Output messages bit width must be the same as input messages, that is, $DWIDTH$, meaning that some sort of approximation has to be done.

Clipping

Since $\alpha_{i,j}^n$ messages are calculated by adding channel LLRs with a subset of $\beta_{i,j}^n$ messages, their dynamic range can be very large. The approach selected is called message clipping (MC) and consists of clipping the partial sum of $\beta_{i,j}^n$ messages instead of clipping the $\alpha_{i,j}^n$ messages, according to:

$$clip(r,t) = max\{min\{r, Q_{max} - t\}, -Q_{max} - t\} \tag{5.9}$$

which ensures that $\alpha_{i,j}^n \leq Q_{max}\ \forall(n,i,j)$ and Q_{max} denotes the MC parameter. For this work the MC parameter is chosen to be dependent on messages bit length instead of being constant:

$$Q_{max} = 2^{DWIDTH-1} - 1 \tag{5.10}$$

This choice ensures that $\beta_{i,j}^n$ messages are representable by $DWIDTH$ bits, as expected.

Implementation of VN

The final structure of the VN is shown in figures 5.25 and 5.26, using as examples DEG=3 and DEG=5. As for minimum and XOR networks in CNs, adder network has a size equal to the nearest power of 2 greater than the number of inputs ($DEG = 3 \longrightarrow SIZE = 4$ and $DEG = 5 \longrightarrow SIZE = 8$). Each adder network output is then clipped (CLIP block) and added to the corresponding channel LLR (called old LLR).

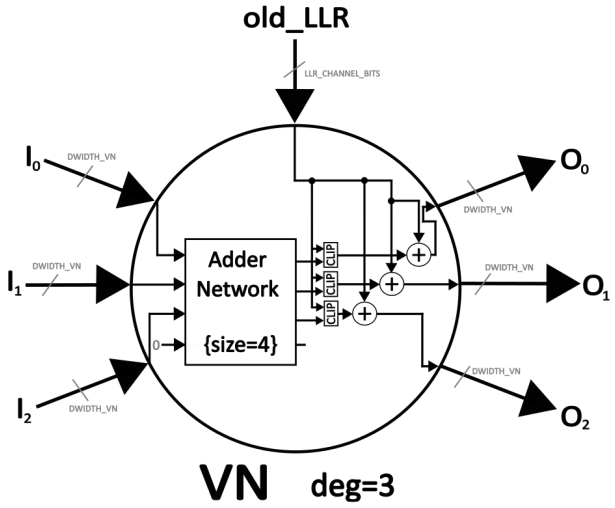


Figure 5.25: Variable Node architecture for DEG=3

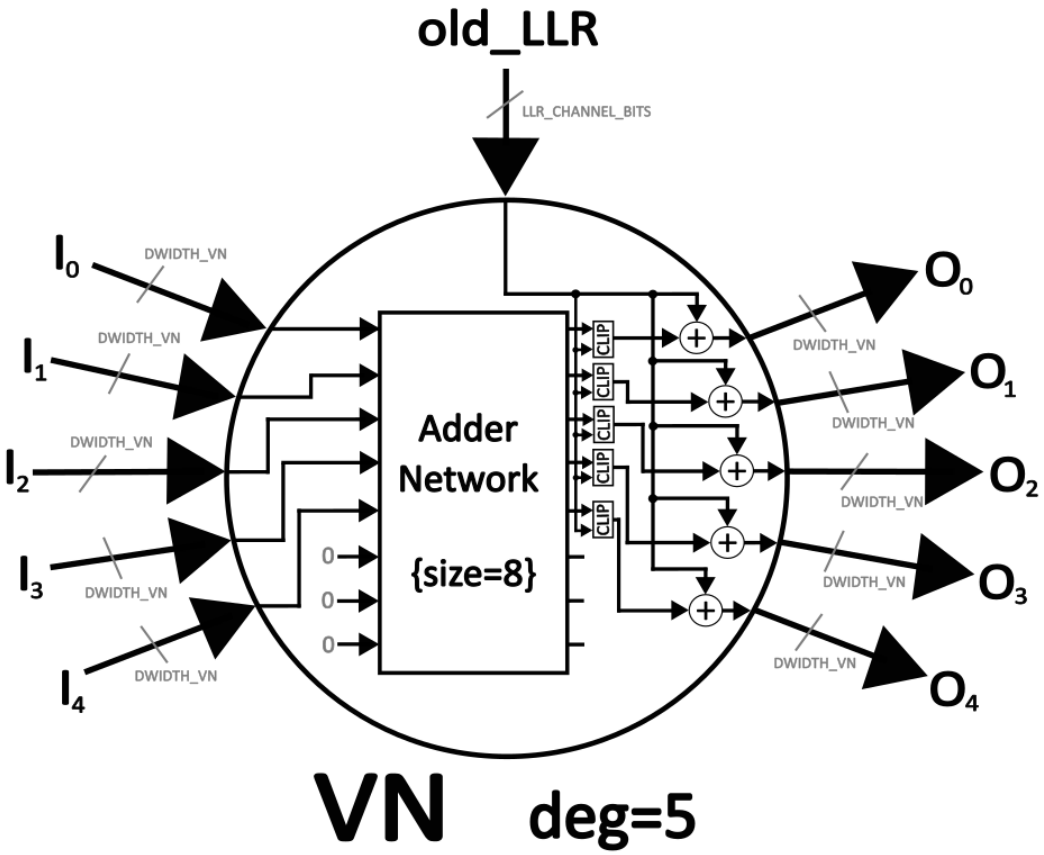


Figure 5.26: Variable Node architecture for DEG=5

5.2.4 OutLLR Node

In the final stage of LDPC decoding, for each bit the a-posteriori LLR needs to be calculated. As explained in previous chapter, the a-posteriori LLR is computed as:

$$\alpha_j^n = \alpha_{i,j}^0 + \sum_{i' \in M(j)} \beta_{i',j}^n$$

This formula is slightly different for the one used for the Variable Node update: in this case all the input messages are added together along with the corresponding channel LLR. For this purpose a new block is defined, called "OutLLR calculator": a generic OutLLR calculator of degree DEG takes as inputs DEG messages coming from the Check Nodes and one LLR coming from the channel and computes the corresponding (single) output message. The generic j-th OutLLR calculator entity is represented in figure 5.27.

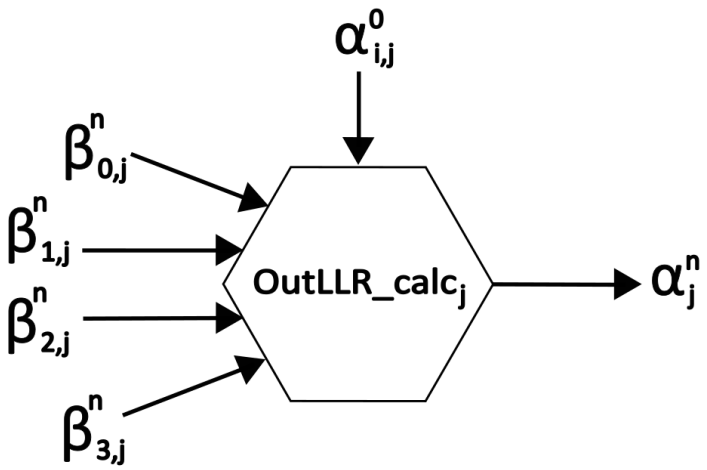


Figure 5.27: OutLLR calculator entity

Implementation of OutLLR calculator

Starting from the internal architecture of the Variable Node (figure 5.25), the adder network block has to be replaced with a simple adder tree in order to combine all the input messages. The partial sum so obtained is now clipped using the same strategy as for VNs and added to the corresponding channel LLR. The final OutLLR calculator architecture is depicted in figure 5.28 for the case DEG=4.

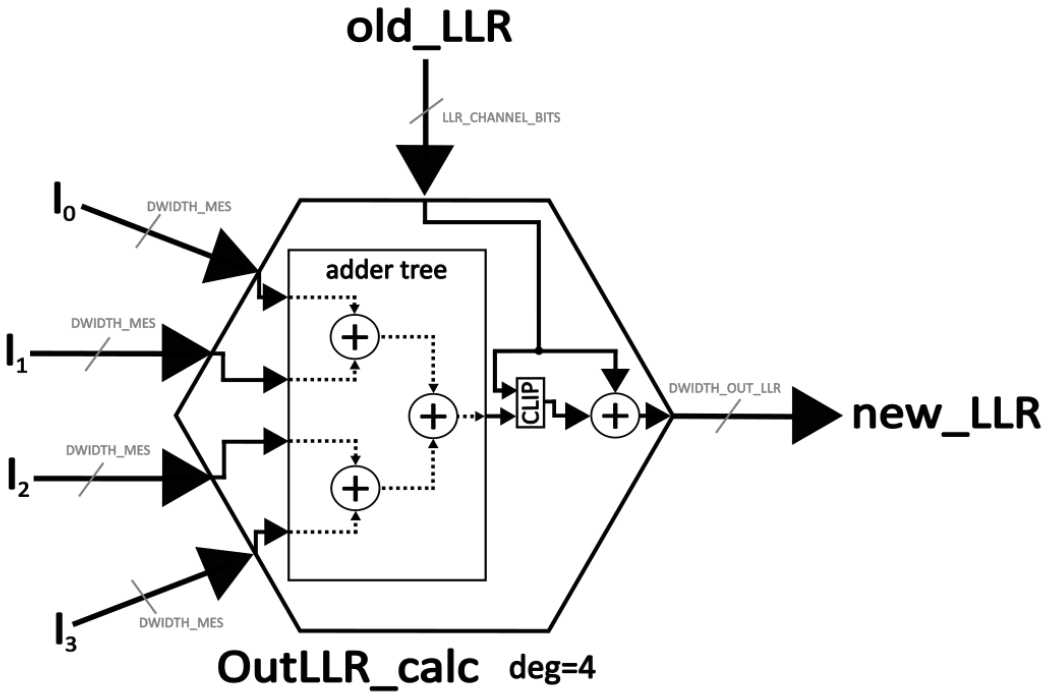


Figure 5.28: OutLLR calculator architecture for DEG=4

5.2.5 Node wrappers

Both Check Nodes and Variable nodes are inserted in their own wrapper. This structure is created for two purposes:

- easily add or remove registers in order to apply retiming strategy
- insert R and Q Modules around the Variable Node instance

Retiming is applied at synthesis level, adding a certain number of registers and then using the command *compile_ultra -retime*. The results obtained for Check Nodes and Variable Nodes are summed up into tables 5.3 and 5.4 respectively.

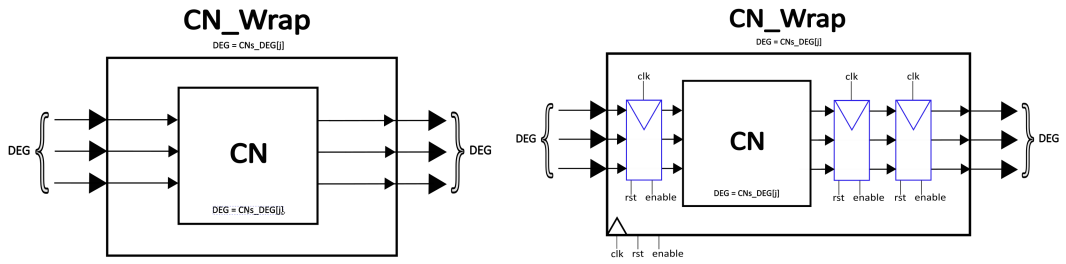
$bits$	total registers	$t_{crit}[ns]$	$area[\mu m^2]$
5	0	1.9	6364
5	1	1.5	9815
5	2	1.3	14588
5	3	1.0	18842
5	4	1.1	18080

Table 5.3: Retiming for Check Node

$bits$	RCQ	total registers	$t_{crit}[ns]$	$area[\mu m^2]$
5	NO	0	1.5	1473
5	NO	1	1.4	2148
5	NO	2	1.1	2775
5	NO	3	1.1	3421
5	YES	0	1.3	1135
5	YES	1	1.3	1790
5	YES	2	1.0	2384
5	YES	3	1.0	2561

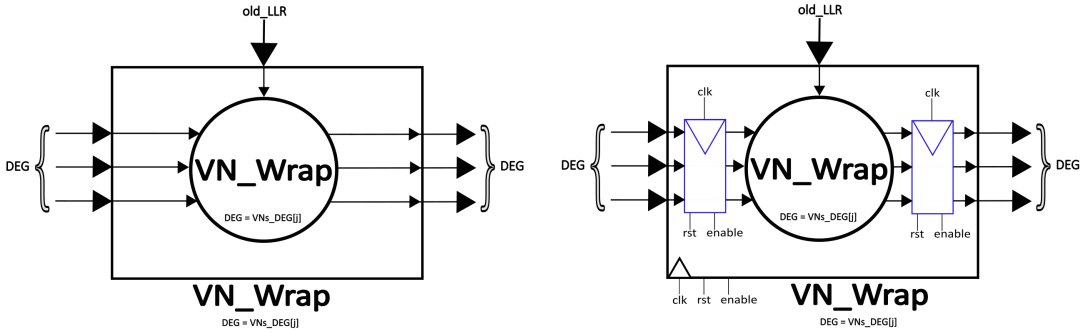
Table 5.4: Retiming for Variable Node

The optimal number of retiming registers to reach peak throughput without increasing area uselessly is 3 for Check Nodes and 2 for Variable Nodes (both with and without RCQ). The complete architectures are shown in figures 5.29 and 5.30.



(a) CN Wrap with optimal latency (b) CN Wrap with optimal throughput

Figure 5.29: CN Wrap internal structure



(a) VN Wrap with optimal latency (b) VN Wrap with optimal throughput

Figure 5.30: VN Wrap internal structure

The *-retime* option is the only way to reduce critical path of all instantiated nodes because nodes with different degree have also different tree height for both minimum network (CN key component) and adder network (VN key component), meaning that a pipeline applied to to each tree level will lead to de-synchronization of results. However, retiming is a burdensome task for the synthesis tool factoring in the enormous number of instantiated nodes in 10 iterations. The synthesis tool can't handle *-retime* option, so in the decoder's total structure retiming registers are removed from wrappers and R and Q modules are moved to their own layers, called R_layer and Q_layer.

5.2.6 Initial layer

The decoder receives the log-likelihood ratios as inputs and they should traverse a first layer of Variable Nodes with all the Check-to-Variable node messages set to 0.

$$\alpha_{i,j}^{0} = ln\frac{P(VN_j = 0|y_i)}{P(VN_j = 1|y_i)} = \frac{2y_i}{\sigma^2}$$

However, considering the VN update rule (equation 2.5), the result is that each Variable node has its own LLR message as output and no sums are performed. In order to save area, instead of instantiating a VN layer, the best approach is taking the input LLR and feeding them to the right Check Node using a simple layer of connection, called **Initial layer** to distinguish it from the other connection layers.

The input LLRs are *(Horizontal_Iterations * Lifting_factor)* and need to be sorted in *(Num_Layers * Lifting_factor)* sets of maximum *Horizontal_Iterations* messages. Connection are performed using a generate statement that connects an output to:

- the corresponding input calculated using the values contained in *CN_connections*, *CN_limits* and *qc_matrix* vectors, if the index $0 \leq count < Horizontal\_Iterations$ is smaller than Node's degree (found in CNs_DEG vector)
- 0, if the index *count* is greater than Node's degree

The resulting structure is depicted in figure 5.31.

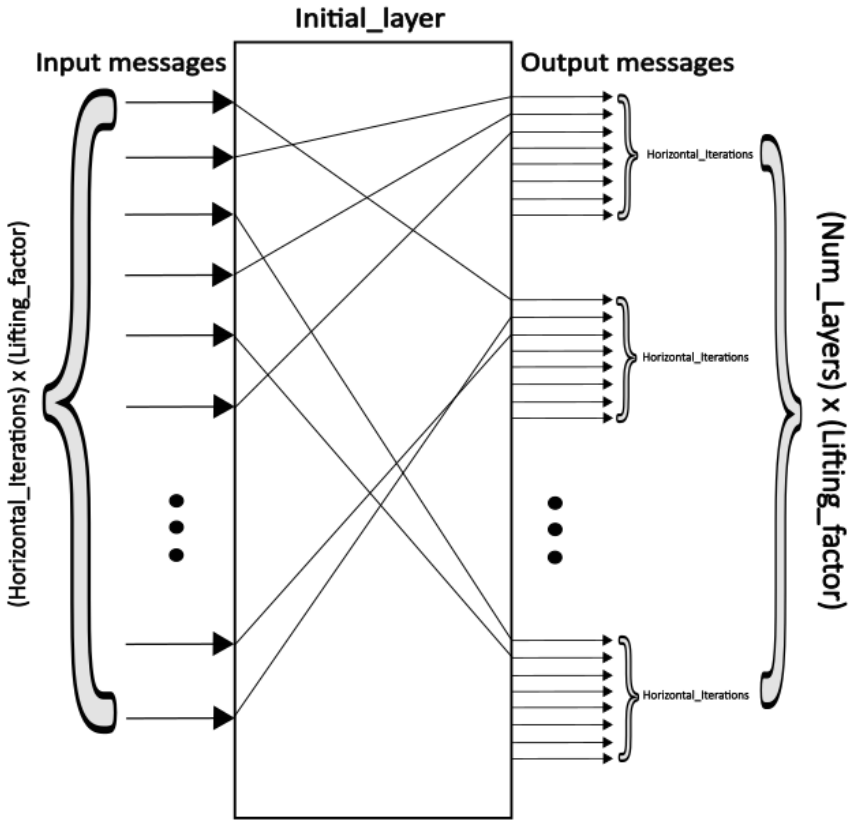


Figure 5.31: Initial layer structure

5.2.7 CN and VN layers

Both Check Node and Variable Node wrappers are instantiated in parallel in their own layer.

The CN_Layer structure (figure 5.32) contains a total of *Num_Layers * Lifting_factor* CN_Wrappers, with each group of *Lifting_factor* nodes having the same degree. There are *Num_layers* different degrees (contained in *CNs_DEG* vector), corresponding to *Num_layers* different minimum network sizes (contained in *CNs_SIZE* vector), and all wrappers are instantiated in the same generate statement and connected to their own input and output ports.

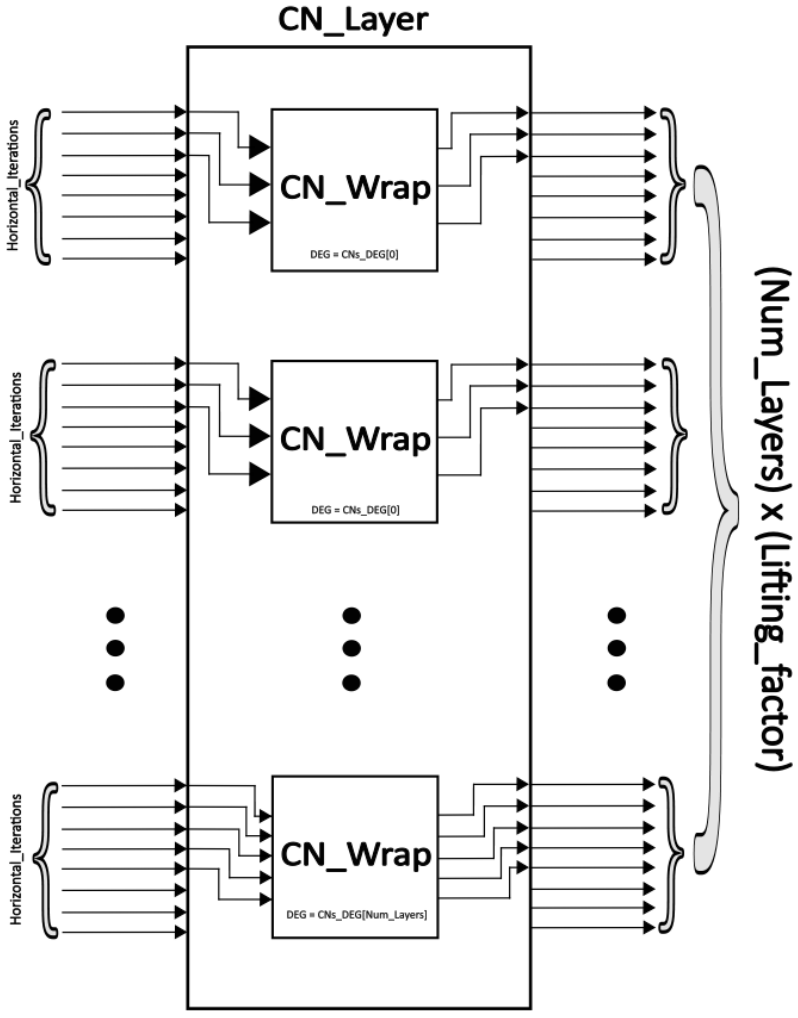


Figure 5.32: Check Node layer structure

The VN_Layer structure (figure 5.33) contains a total of *Horizontal_Iterations * Lifting_factor* VN_Wrappers, with each group of *Lifting_factor* nodes having the same

degree. In this case the different degrees and sizes of calculation network are *Horizontal_Iterations* and they are collected in *VNs_DEG* vector and *VNs_SIZE* vector respectively. A generate statement instantiates all the wrappers, connecting them to their corresponding input LLRs (equal in number to the wrappers) and to their respective input and output message ports.

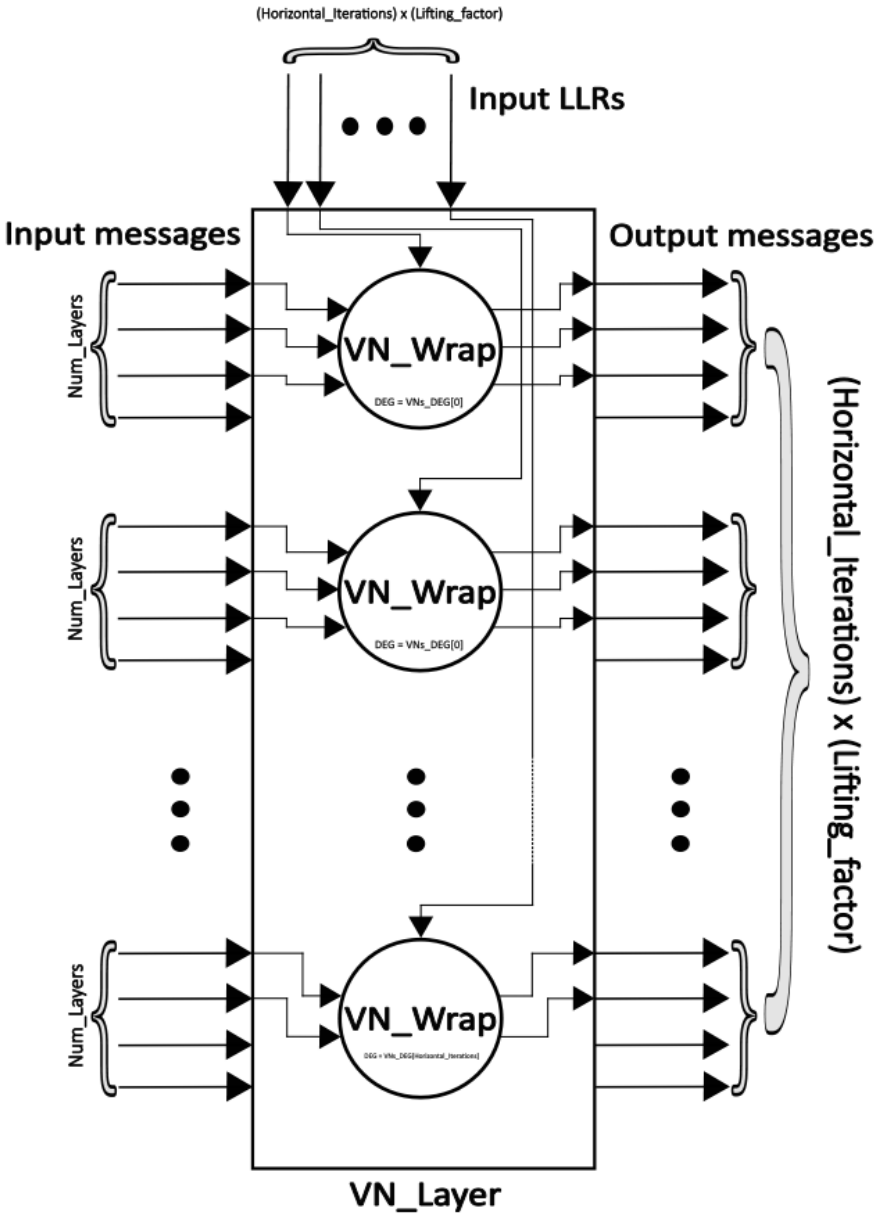


Figure 5.33: Check Node layer structure

5.2.8 C2V and V2C layers

Connection between layer of Check Nodes and Variable nodes are implemented using 2 different blocks called *C2V_connection* (figure 5.34) and *V2C_connection* (figure 5.35), using a mechanism totally equivalent to the message passing algorithm implemented in C++ code.

Both structures use an internal matrix of signals with dimensions *[NumLayers*LiftingFactor-1:0]* and *[HorizontalIterations*LiftingFactor-1:0]*, totally equivalent to the message matrix used in section 5.1 .

For *C2V_connection*, input messages are connected to the matrix cells row by row, while output messages are connected to the matrix cells column by column. The indexes of connected CNs and connected VNs are written into the configuration vectors *CNs_connections* and *VNs_connections*. Only a subset of cells are eventually connected, because CN and VN can have degree smaller than the maximum possible. Wires that are not connected are automatically removed during synthesis.

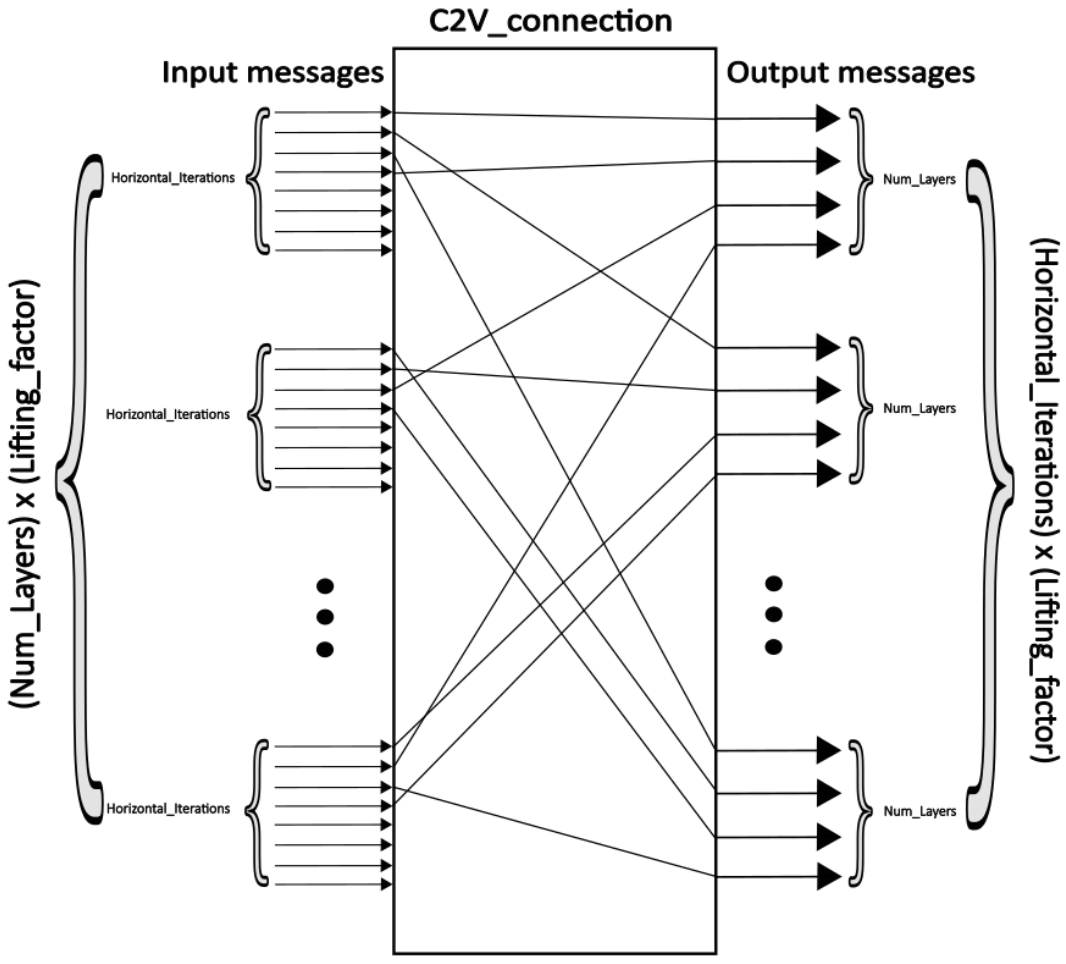


Figure 5.34: Check Node to Variable Node connection structure

For *C2V_connection* the same mechanism is applied in reverse, so input messages are connected to the matrix cells column by column, while output messages are connected to the matrix cells row by row. Wires not connected are removed during synthesis.

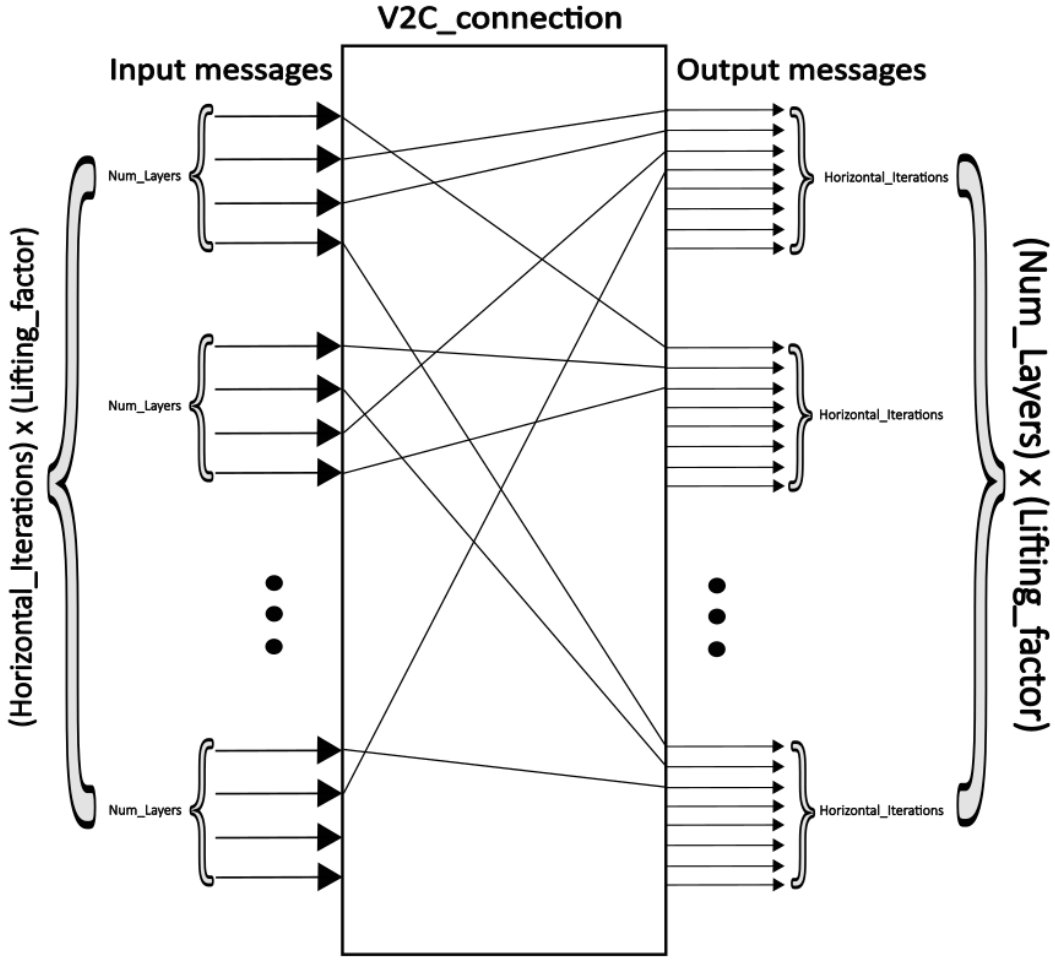


Figure 5.35: Variable Node to Check Node connection structure

5.2.9 Final layer

For the last iteration VN_layer is replaced by a layer containing all the Out_LLR_Wrap instances, in order to compute the output LLRs according to decoding algorithm. Each wrapper has a maximum of $NumLayers$ input messages, one input channel LLR and a single output message, meaning that the layer containing all wrapper instances is made of $(HorizontalIterations \cdot LiftingFactor)$ entities, with a total of $(HorizontalIterations \cdot LiftingFactor)$ a-posteriori LLRs. The final architecture is shown in figure 5.36.

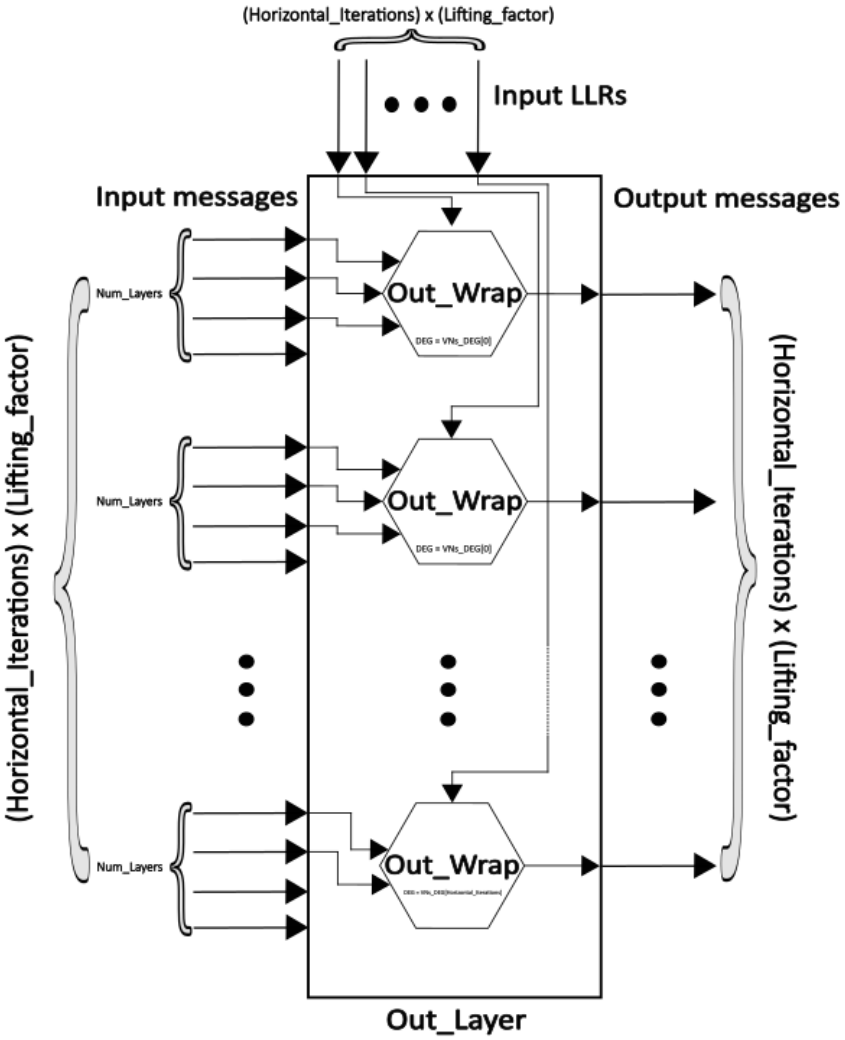


Figure 5.36: OutLLR layer structure

5.2.10 Conversion layer

Lastly, each a-posteriori LLR coming from OutLLR layer is used to estimated the corresponding bit of the estimated codeword, using the expression: $\hat{c}_j = \begin{cases} 0 & \text{if } \alpha_j^n > 0 \\ 1 & \text{else} \end{cases}$

In other terms, the sign bit of the a-posteriori LLRs determines the value of the estimated codeword. Not all a-posteriori LLRs are used for computation, because the codeword has a bit lenght of $InformationBits$, which is defined in *configs.sv* as the quantity:

$$InformationBits = (HorizontalIterations - NumLayers) \cdot LiftingFactor$$

The conversion layer structure is depicted in figure 5.37.

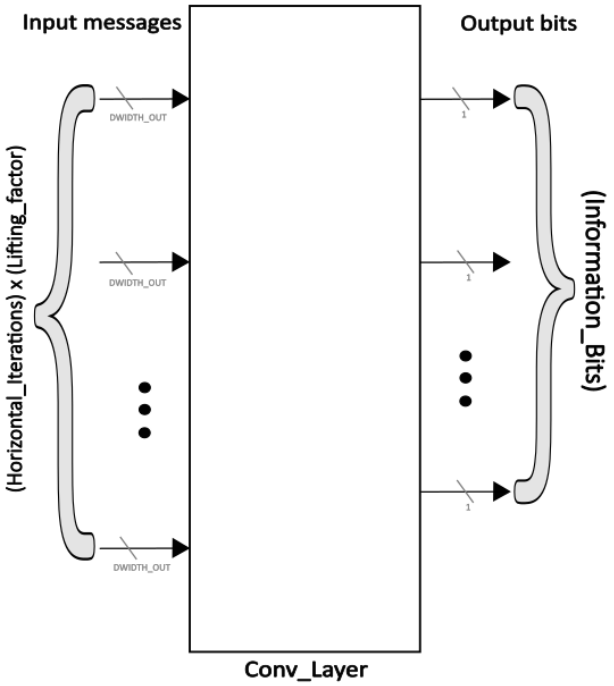
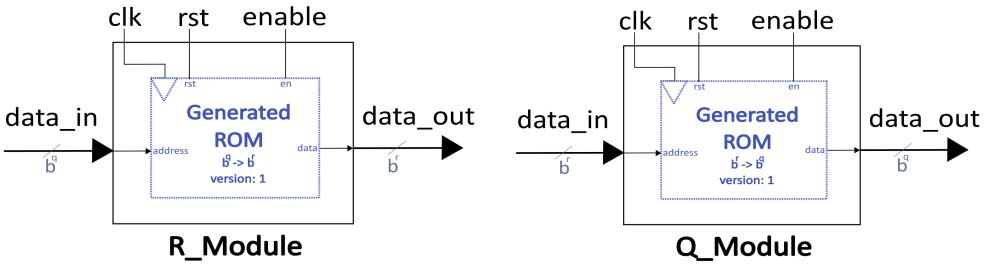


Figure 5.37: Conversion layer structure

5.2.11 R and Q Modules

R and Q modules are responsible of Reconstruction and Quantization operations, core mechanic of the RCQ paradigm. The ROM memories containing all the correct values for Q/R are generated using python scripts (section 5.2.1), so the task Q and R modules use an if-generate statement to instantiate the right ROM memory (figure 5.38) using 3 parameters: input bit length (*DWIDTH_IN*), output bit length (*DWIDTH_OUT*) and table version (*VERSION*). If there is no generated table that matches these parameters, a simple resize of inputs is performed.

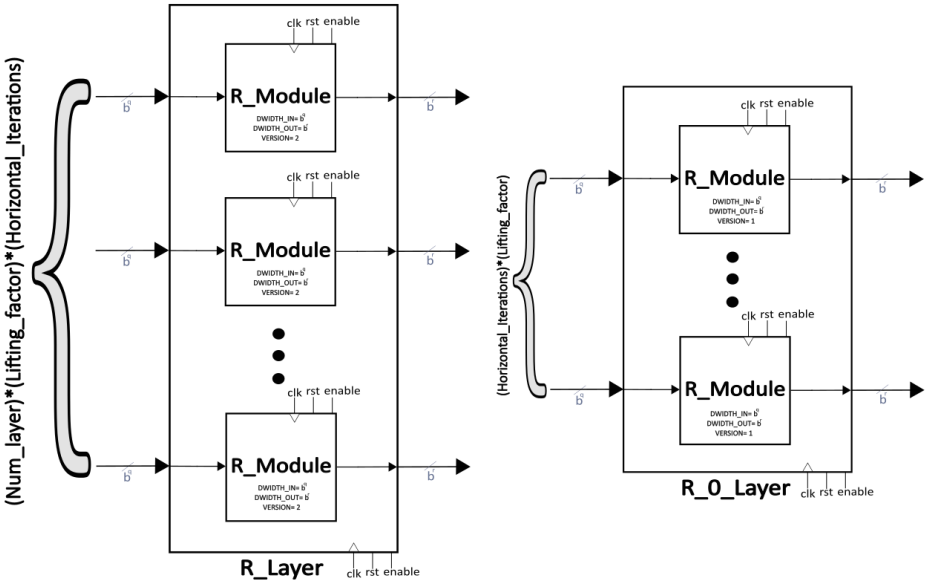


(a) R_module architecture (b) Q_module architecture

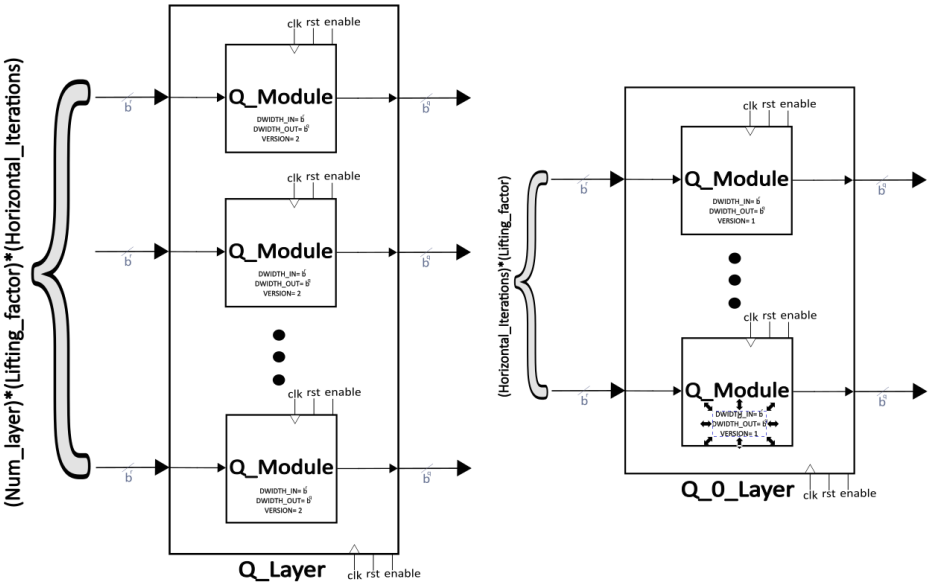
Figure 5.38: Modules for Reconstruction and Quantization

5.2.12 R and Q Layers

R and Q modules are instantiated in parallel using 2 types of layers for each type of module (figure 5.39). The only difference between *R_0_layer* and *R_layer* is the number of modules: the first has $HorizontalIterations \cdot LiftingFactor \cdot NumLayer$ modules, the second $HorizontalIterations \cdot LiftingFactor$. *R_0_layer* and *Q_0_layer* are used for Q/R of input LLR, so they are placed right after input registers. *R_layer* and *Q_layer* are used to reconstruct input messages to *VN_layer* and quantize its output messages, meaning that every other layer is instantiated with an inferior number of bits with respect to *VN_layer*. The number of bits after quantization used for each iteration is defined into the configuration vector *RCQ_bits*, as well as the type of quantization used, defined into *RCQ_table_sel* vector.



(a) R_layer and R_0_layer architecture



(b) Q_layer and Q_0_layer architecture

Figure 5.39: Layers for Reconstruction and Quantization

5.2.13 Total Datapath

All the layers are combined together to obtain decoder's total datapath, depicted in figure 5.41. The section within the green square represents a single iteration of the central component, shown in figure 5.40. This section is replicated $N-1$ times to create a decoder with N iterations. Pipeline is applied considering that:

- datapath has its input and output registers
- each iteration has the same number of pipe registers
- CN_layer presents the highest combinatorial delay, due to the minimum network structure
- all connection layers are assumed to have zero load, so they do not factor into the delay calculation

Therefore each iteration is sliced in 3 parts, being careful to isolate *CN_layer* block. To correctly apply pipeline strategy, also quantized input LLRs are delayed (they belong to the same feed-forward cut-set of inter-node messages). The computation requires a number of clock cycles equal to:

$$Cycles = 1 + 3 \cdot (N-1) + 2 + 1 = 3 \cdot N + 1$$

obtained considering input register, pipe registers for the first $N-1$ iterations, pipe registers for last iteration and output registers.

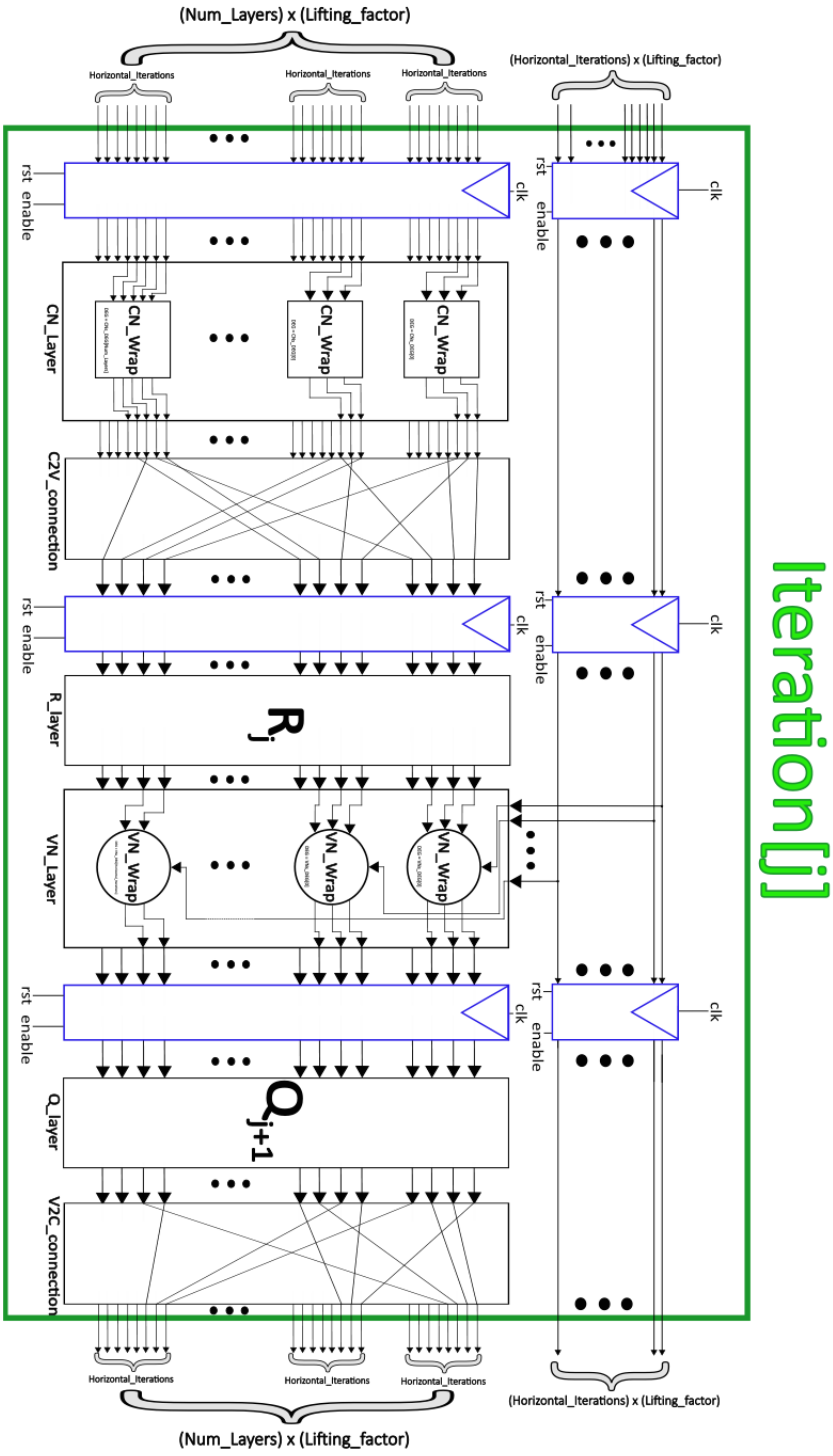


Figure 5.40: Internal structure of the j-th iteration

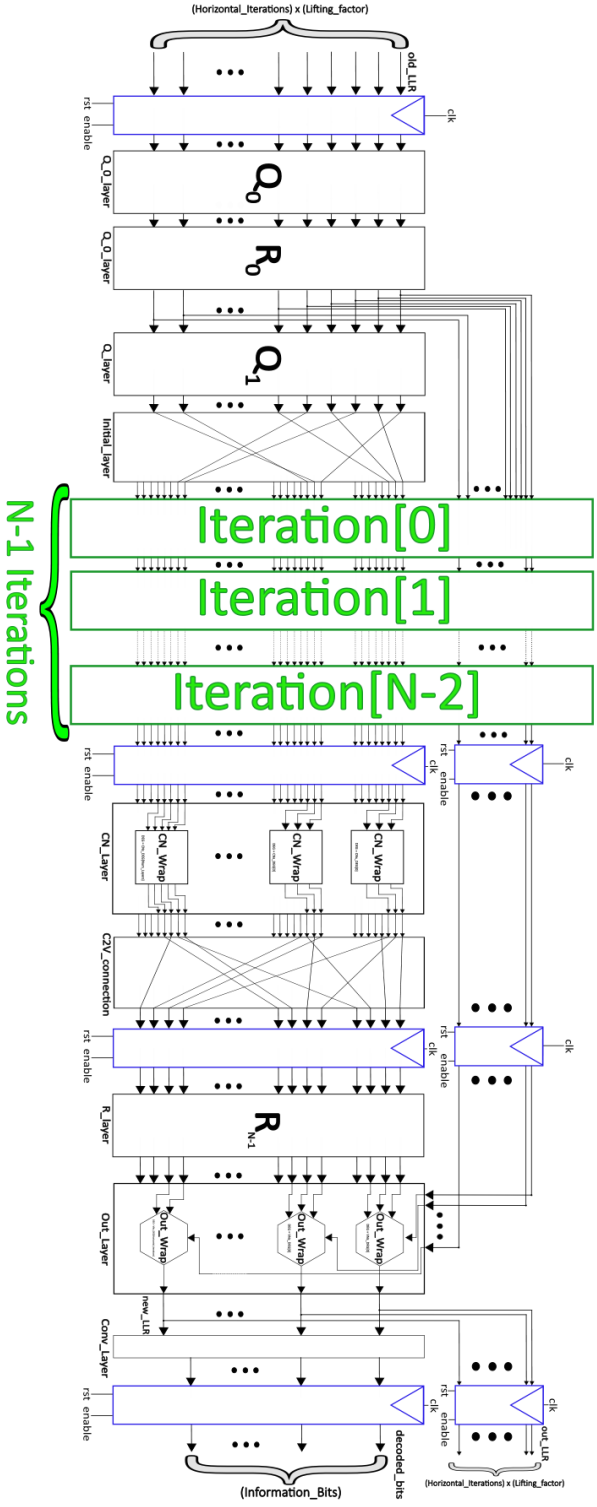


Figure 5.41: Total Datapath

5.2.14 Control Unit

Decoder's Control Unit implements a ready/valid protocol, a widely used handshaking mechanism that ensures reliable data transfer between two components. It is assumed that the decoder:

- receives channel LLRs from a producer block. Input data is valid when *in_valid* is asserted
- sends decoded bits and output LLRs to a receiver block. Output data is valid when *out_valid* is asserted

Therefore, the Control Unit (figure 5.42) has the following input signals:

- *receiver_ready*: is asserted when receiver is ready to sample output data
- *in_valid*: tells the decoder when input data can be sampled to start decoding operation

and the following output signals:

- *decoder_ready*: tells the producer that decoder is ready to receive channel LLRs
- *out_valid*: tells the receiver that the decoding operation is completed and output LLRs are ready to be sampled
- *enable* : controls all datapath's pipe registers, stalling all operations when needed.

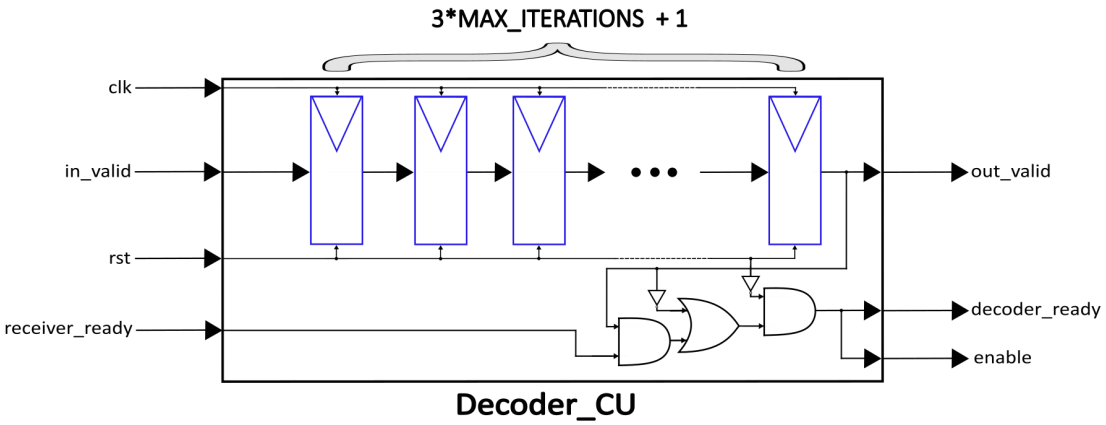


Figure 5.42: Control Unit architecture

The Control Unit is responsible of stalling Datapath's pipeline negating the enable signal. Pipeline can advance when there is no output data ready(*out_valid* == "0") or there is output data ready and the receiver can sample it (*out_valid* == "1" && *receiver_ready* == "1").

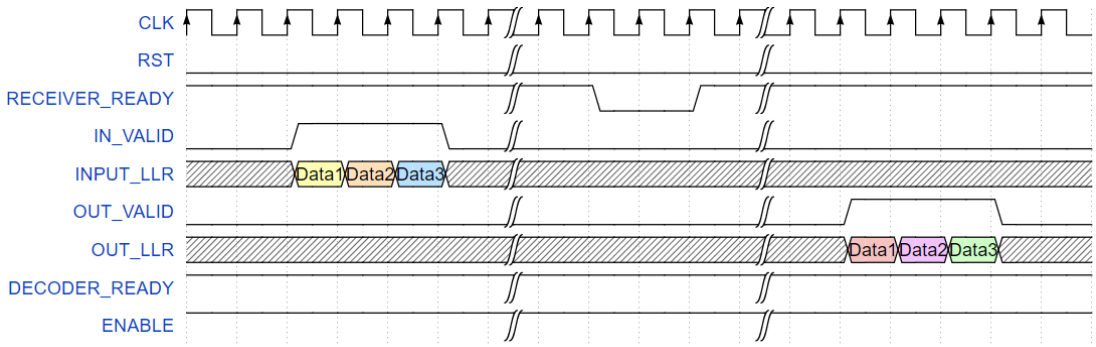


Figure 5.43: CU timing diagram without stall

The following timing diagrams illustrate the behavior of the Control Unit under the two main operating conditions. In the first case (figure 5.43) *receiver_ready* is negated when there is no output data available, so there is no need to stall the pipeline.

In the second case (figure 5.44) *receiver_ready* is negated right before decoding ends, requiring the decoder to stall when output data becomes available until the receiver block is ready again.

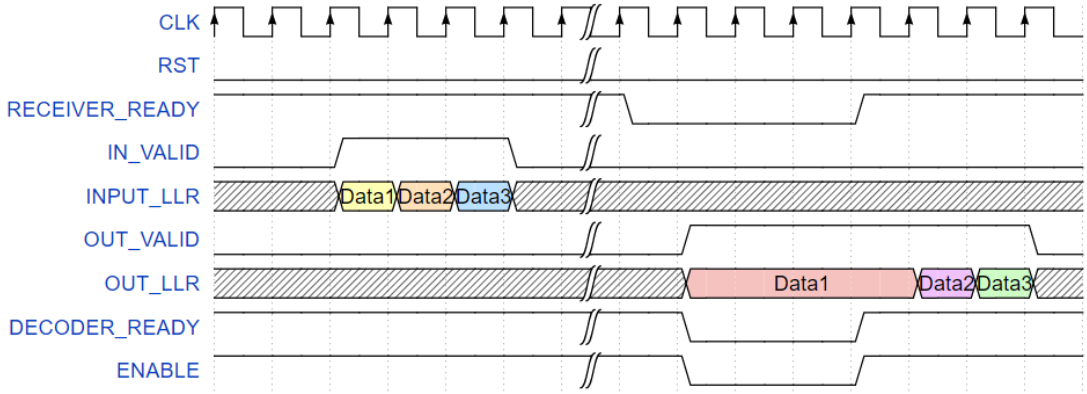


Figure 5.44: CU timing diagram with stall

5.2.15 Final structure

The final structure is obtained simply instantiating Datapath and Control Unit, with a single enable signal connecting them (figure 5.45.

However, as consequence of some problems encountered during synthesis, enable signal slows down the circuit due to its high fan-out. The solution adopted consists in instantiating a number of Control Units equal to the number of pipe levels, so that each layer of pipe registers has its own enable signal connected. They are, of course, identical to one another and function as before; however, this modification is essential to eliminate the

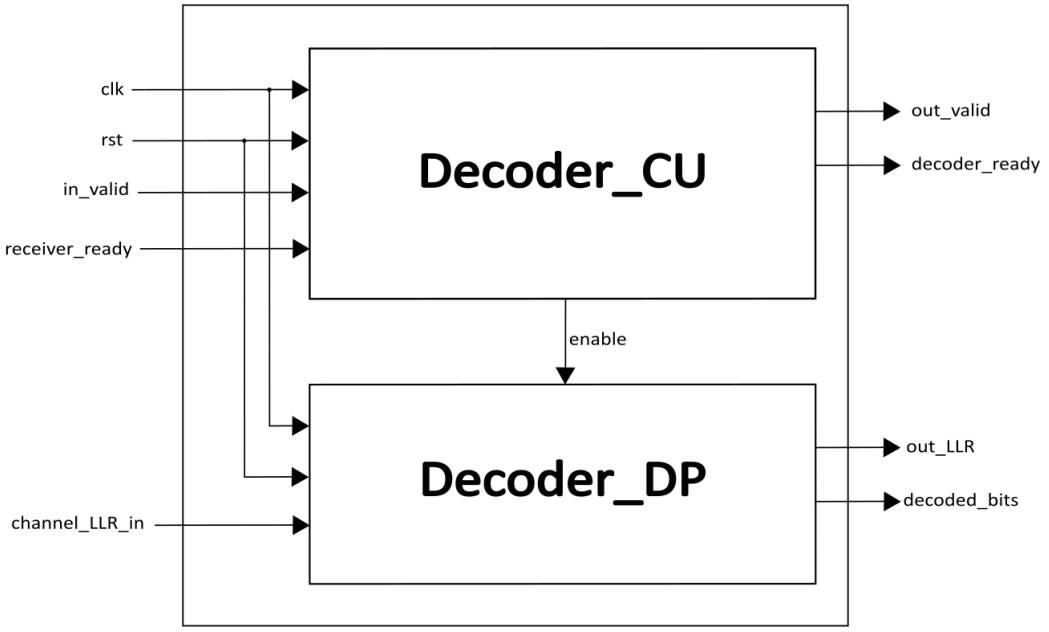


Figure 5.45: Total structure

slowing issue caused by a high fan-out net (figure 5.46).

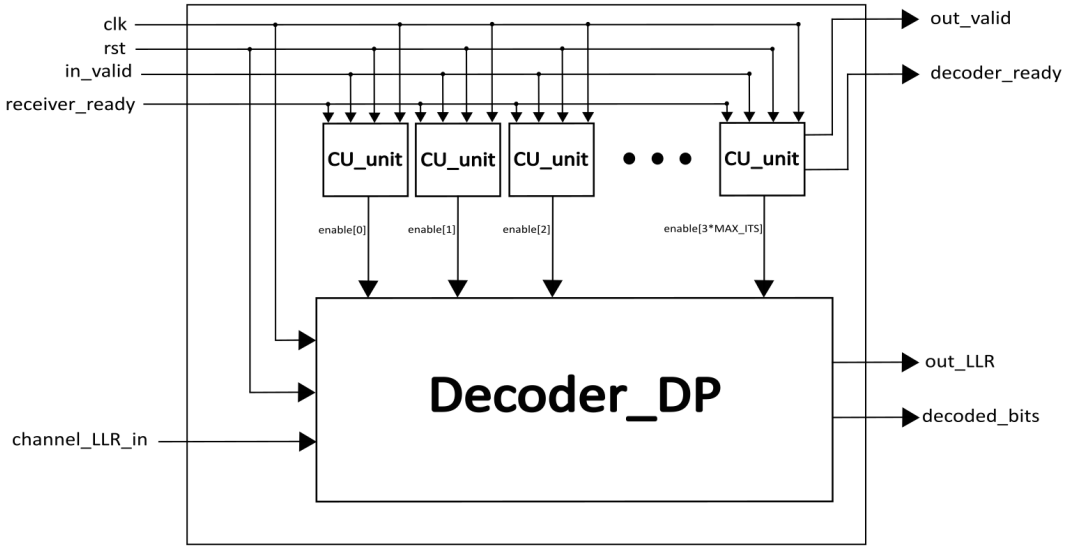


Figure 5.46: Total structure with reduced fanout enable signals

Chapter 6

Experimental Setup

This section outlines the experimental setup used to evaluate the decoder, focusing on both the algorithm's performance and the implementation's efficiency.

6.1 Testing C++ model

Decoder's C++ model (decoder.cpp) is tested using a top level called *sim.cpp*. Its internal structure is depicted in figure 6.1 and consists of:

- a Source that generates a set of random bits, called *reference bits*
- an Encoder that applies an LDPC encoding using the same parameters and the same H matrix as the decoder
- a Modulator that applies Bi-Phase Shift Keying (BPSK) modulation
- the AWGN channel which adds a noise with a certain N_0 and σ^2
- a Demodulator that converts noisy symbols coming from channel in LLRs
- the developed LDPC Decoder
- a Monitor that compares *reference bits* and *decoded bits*

The configuration file *configs.h* is used to define all decoder and simulation parameters, like H matrix, code rate R, number of iterations *MAX_ITERATIONS* and so on.

The simulation campaign is designed as a sweep over a range of E_b/N_0 values, where the level of AWGN noise added by AWGN channel module depends on E_b/N_0. For each E_b/N_0 value, the simulation processes up to a maximum of 1,000,000 frames, terminating earlier if 50 frame errors are detected. Once the simulation for a given E_b/N_0 value is complete, whether due to reaching the frame limit or hitting the error threshold, the results are saved to a CSV file. This procedure is carried out for each E_b/N_0 value in the range, which spans from 0.0 dB to 5.0 dB in increments of 0.25 dB.

The CSV generated file consists of rows, each containing 4 values:

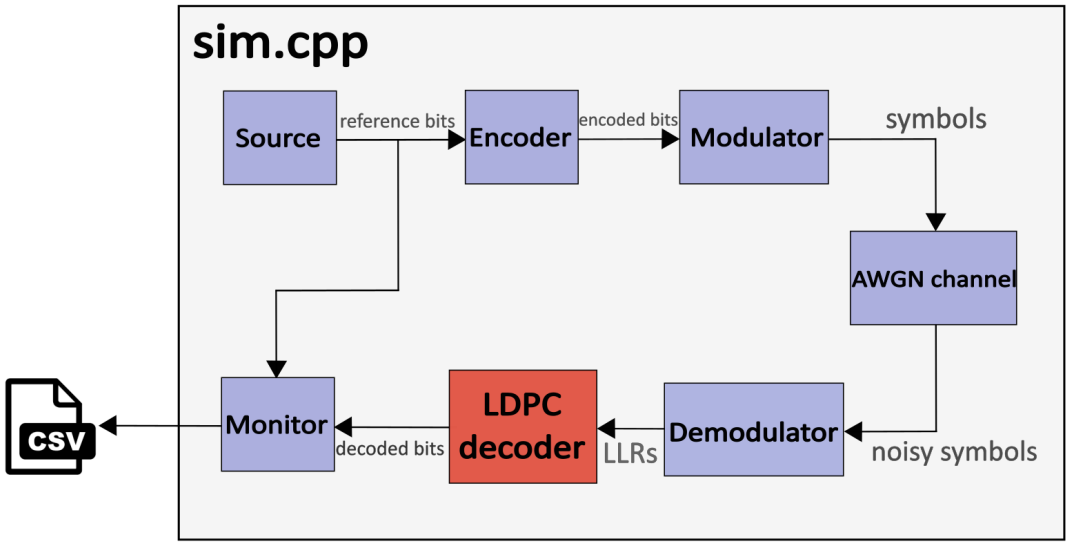


Figure 6.1: sim.cpp structure

- E_b/N_0 value
- number of frame errors
- number of bit errors
- total number of frames tested

So each row is used to place a point on the (FER,E_b/N_0) graph and on the (BER,E_b/N_0) graph. The python script *plot_bfer.py* is used to extract all rows from all CSV generated files and depict the complete graphs (FER,E_b/N_0) and (BER,E_b/N_0).

In these graphs, each line corresponds to a different type of simulated decoder. Specifically, decoding performance was evaluated for various numbers of iterations and compared against decoders with the same (648,540) code.

6.2 RTL code organization

The decoder consists of multiple blocks, as described in the previous chapter: to manage dependencies effectively, *.core* files are used.

.core files are configuration files often used in design and simulation flows to define filesets, dependencies, and associated commands. They provide a structured way to manage source files and build instructions, simplifying the integration and compilation process, particularly when using tools like Makefiles. Each *.core* file includes an *rtl* fileset that lists all the source *.sv* files along with their module dependencies. The dependency structure obtained is depicted in figure 6.2.

This structure allows the creation of distinct subsets for each module, enabling compilation and simulation with various tools and testbenches. In particular:

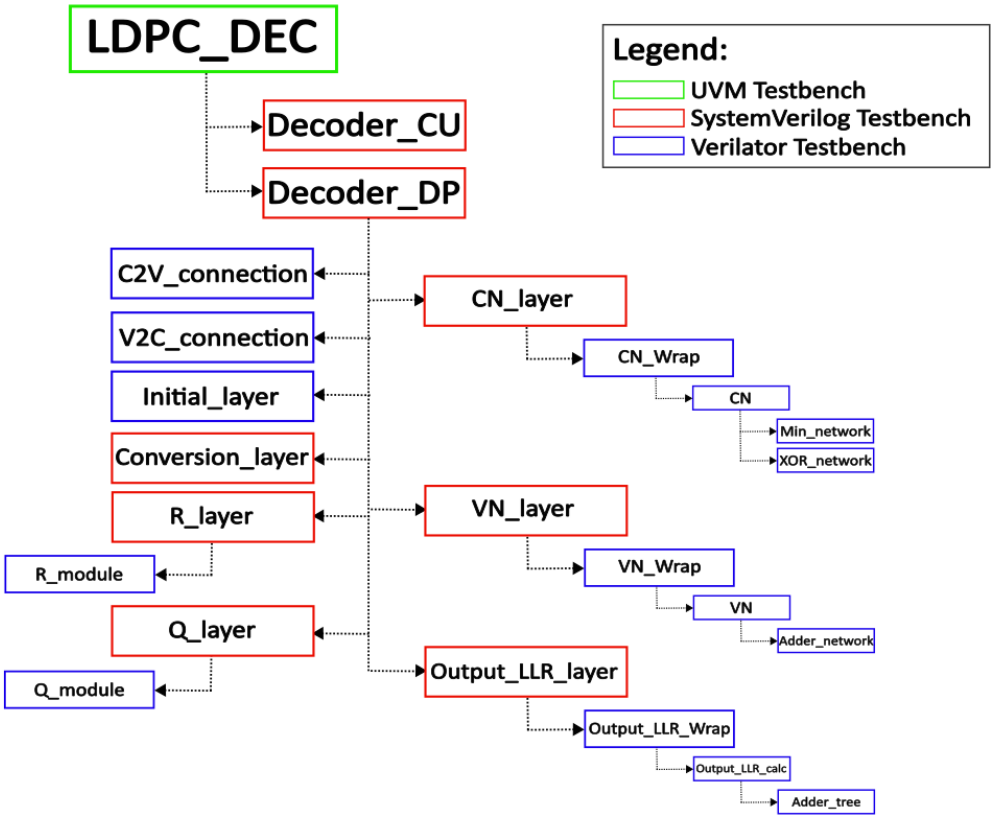


Figure 6.2: Dependencies hierarchy of .core files

- Blocks are compiled using Verilator and tested with a SystemC testbench (blue box)
- Layers are compiled using Modelsim and tested using a SystemVerilog testbench (red box)
- the whole decoder is compiled using Modelsim and simulated using an UVM testbench (green box), both pre-synthesis and post-synthesis

6.3 Testing RTL blocks

Blocks and their components are compiled using Verilator, an open-source simulator designed for the cycle-accurate simulation of hardware described in Verilog or SystemVerilog. Verilator works by converting HDL code into a cycle-accurate SystemC model, which can then be compiled and executed. This SystemC component is included in a SystemC testbench organized as shown in figure 6.3.

The testbench instantiates and connects all the modules and generates the clock signal. The generator module is responsible for producing inputs for the DUT, which are randomly

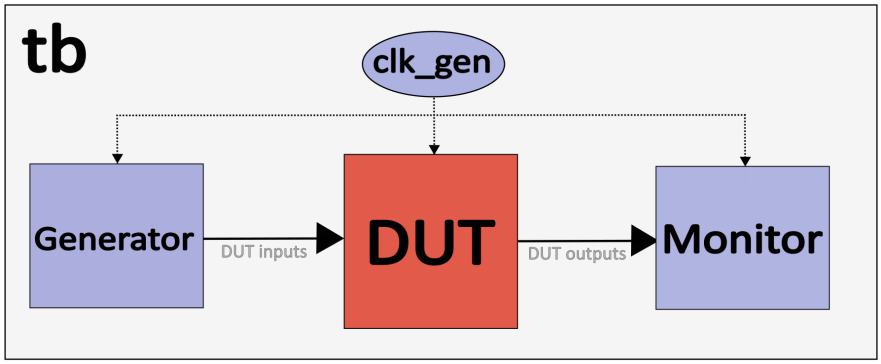


Figure 6.3: Architecture of the generic SystemC testbench

generated. The monitor module captures the outputs from the DUT and displays them on the terminal, allowing for comparison and analysis.

6.4 Testing RTL layers

When compiling layers, which are modules consisting of a large number of smaller block instances, Verilator takes an excessive amount of time, and displaying all the output on the terminal becomes impractical. For this reasons, layers are compiled using Modelsim and tested using a SystemVerilog testbench.

The SystemVerilog testbench instantiates the DUT, provides it with inputs read from a text file (which differs in name and size depending on layer's type), and writes the outputs to an output text file.

6.5 Synthesis scripts

Synthesis is performed using Synopsis Design Compiler, mapping the decoder on standard cells from 65 nm technology libraries. The synthesis script used is *synth.tcl*, which sets clock frequency to 3.3 ns and uses the command *compile_ultra -timing -retime*. This command forces the compiler to apply the retiming technique, which involves repositioning the flip-flops within a circuit without altering its functional behavior. Retiming redistributes these registers to optimize critical paths, reducing the longest delay between any two flip-flops, which in turn increases the circuit's maximum clock speed. This technique is particular effective on the unrolled structure of the decoder because the pipeline register can be easily moved back and forth to reduce the critical path delay.

However, synthesis cannot reach its completion because the decoder is composed of too many cells and Design Compiler requires too much RAM memory to perform optimizations. Using this script only 3 iterations can be synthesized, which is an insufficient number.

6.5.1 Bottom-Up Compilation strategy

In order to synthesize the complete decoder (consisting of 10 iterations), the Bottom-Up Compilation strategy, commonly employed for large designs, must be applied.

In the Bottom-Up strategy, individual subdesigns are constrained and compiled separately and then incorporated into the top-level design. Each layer of the decoder is synthesized independently using the *synth_subsystem.tcl* script and the same clock period, which is determined by the longest delay among all the layers. After successful compilation, each layer is exported in a .ddc file, which contains detailed information about the design's structure, logic hierarchy, gate-level netlist, constraints, and optimization details. All the .ddc files are collected into the *ps_netlists* folder, ready to be used to compile the whole decoder using the *synth_modules.tcl* script: the .ddc files are loaded into top-level design and each layer is assigned the *dont_touch* attribute to prevent further changes during decoder's synthesis. This method is necessary to compile large designs because Design Compiler does not need to load all the uncompiled subdesigns into memory at the same time. However, using bottom-up compilation prevents the synthesizer from performing cross-block optimizations, resulting in a lower achievable maximum clock frequency (minimum clock period is 4.1 ns).

6.6 Testing complete structure: UVM testbench

To test the final SystemVerilog architecture and the post-synthesis netlist is used an UVM testbench. UVM (Universal Verification Methodology) employs a layered, object-oriented approach used to design a modular and reusable verification environment. This paradigm involves separating concerns between the testbench architecture and the specific tests to be performed. Multiple tests can be executed on the same architecture by simply modifying the input stimulus sequence (and constraints) and selecting different properties of the Design Under Test (DUT) to observe.

As depicted in figure 6.4, the UVM environment comprises several key components, each fulfilling a specific role, as detailed in the following sections. All major testbench components are derived from the corresponding base class.

6.6.1 Configuration file

The configuration file *uvm_tb_configs.sv* is used to define types and parameters. In this particular case, the only parameters specified are the paths to the two files:

- *recLLRvectors*, containing a vector of channel LLRs for each row
- *decLLRvectors*, containing the vectors of output LLRs corresponding to the channel LLRs of the other file

The testing mechanism involves feeding the decoder with the channel LLRs from one file and comparing its output with the output LLRs from the other file.

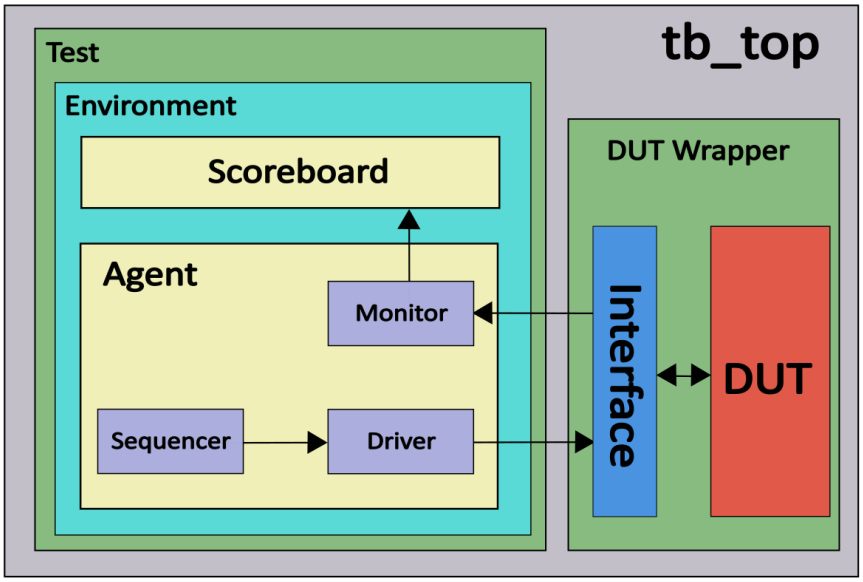


Figure 6.4: UVM testbench architecture

6.6.2 DUT Interface and DUT Wrapper

A key feature of UVM environment is the Dual-Top architecture and consists of splitting the architecture into a verification environment (TB), based on classes that rely on TLM (Transaction Level Modeling) protocols, and the HDL domain containing the DUT and its interface, used with Drivers and Monitors to translate transactions into pin signals and vice-versa. The interface class *dut_if.sv* models the connection between the DUT instance and driver/monitor using 2 ports. The wrapper class, *dut_wrap.sv*, instantiates the DUT and the interface, connecting them while exposing the remaining port to the outside, which will be linked to driver/monitor. Considering that both DUT's SystemVerilog top entity and post-synthesis netlist entity have same name and same I/O signals, the same UVM testbench is used to conduct tests on both of them.

6.6.3 Packet

The purpose of this class is to model a packet of information. The packet class (*packet.sv*) is an extension of the uvm_sequence_item class and will be exchanged between driver, DUT and monitor.

In this specific case the packet class consists of 2 structures:

- the *LLR_payload* vector, consisting of $LiftingFactor \cdot HorizontalIterations$ integers
- the *eof* bit, that tells the driver wether the *recLLRvectors* file has reached EOF or not

If the packet is received by the driver, then it carries the channel LLRs that are going to be decoded. If the packet is received by the scoreboard, then it carries the output LLRs to compare with the golden reference (*decLLRvectors* file).

6.6.4 Sequencer

The sequencer manages the flow of transactions between the stimulus generation and the driver. *sequencer.sv* contains a constructor method that ensures that input file *recLLRvectors* is opened correctly and a get_next_item method that reads an input vector from file each time it is called until EOF is reached.

6.6.5 Driver

The driver interacts with the DUT through its port, playing a critical role in translating high-level transaction-based stimuli into low-level signal activities. In this specific case, the *driver.sv* class resets the DUT, receives packets from sequencer and executes the *drive_packet* method for each valid packet (*eof*=1):

- waits for the DUT to be ready (*decoder_ready*=1) for 2 clock cycles
- loads the received LLRs into the DUT using the interface's write method and asserts the valid signal (*in_valid*=1) for 1 clock cycle
- waits for the decoding process to be done (*out_valid*=1)
- waits for the DUT to be ready again (*decoder_ready*=1)

6.6.6 Monitor

This monitor is in charge of detecting the packets, record them and send to the scoreboard. The signals coming from the DUT are converted into transaction-level data. In particular the *monitor.sv* class waits until a decoding operation is completed (*out_valid*=1),then:

- copies the output LLRs coming from DUT into a packet
- sends the packet to the scoreboard unit
- waits for the *out_valid* to be negated

6.6.7 Agent

The agent is in charge of instantiating the components described before: the sequencer, the driver and the monitor. Moreover, it connects sequencer with driver and monitor with a specific port that communicates with scoreboard.

6.6.8 Scoreboard

The scoreboard is responsible for verifying the correctness of the DUT's output. Compares actual outputs from the DUT with expected results, called golden results. In this specific case *scoreboard.sv* opens the file containing golden LLRs (*decLLRvectors*) and, for every packet received, it compares the carried output LLRs with a row of golden LLRs from file. Comparison is done element by element, If an error occurs, the scoreboard halts and reports the issue to the user, specifying the differing LLRs and their index.

6.6.9 Environment

The environment is the core of the UVM testbench, encapsulating other components and providing a structure for communication. Instantiates the agent (that contains monitor, driver and sequencer) and the scoreboard and connects them.

6.6.10 Test

The test is the top-level entity that configures the testbench for specific scenarios. It sets up the stimulus, environment, and DUT properties. Additionally, *test.sv* includes a method to count the number of lines in the input file, ensuring the testing sequence is repeated the correct number of times.

6.6.11 TB_top

The top entity instantiates both the test structure and the DUT wrapper (that contains the DUT and its interface), connecting them. Being the top entity it's also responsible of initializing clock, reset and other signals. In this case it sets $receiver\_ready = 1$, so tests are conducted without stalling the decoder. It uses an "always" block to generate a clock signal with period $10ns$.

Chapter 7

Experimental Results

In this chapter, the previously described testbenches are employed to test the implemented decoder: the C++ testbench is used to obtain information regarding decoding performances of the algorithm in terms of BER and FER; synthesis reports are used to collect data on area, latency, and throughput; the UVM testbench is used to verify that the results from the C++ implementation and simulation are consistent.

These results are then compared with the state of the art to highlight the decoder's strengths and weaknesses.

7.1 Decoding performances

The *sim.cpp* testbench is employed to evaluate the performances of the decoding algorithm for different numbers of iterations, in particular 5,8,10 and 25. The generated .csv files are then read by the python script and BER and FER are plotted in function of E_b/N_0. Plots are depicted in figures 7.1 and 7.2 respectively.

Obviously BER and FER decrease as the number of iterations increases, because each iteration refines the exchanged messages between nodes.

In order to evaluate the performance of the implemented decoding algorithm, it is compared with the BER plots of two other decoders that implement the same (648,540) code:

- Work [5] describes an high-throughtput unrolled decoder, with 4-bit parallelism and 8 iterations
- Work [9] described a low-power layered decoder. with a 7 bit parallelism and 8 iterations

As shown, the decoding algorithm used in this work is capable of reaching better BER of both considered decoders even with a reduced number of iterations and a 3-bit quantization and reconstruction at CN level.

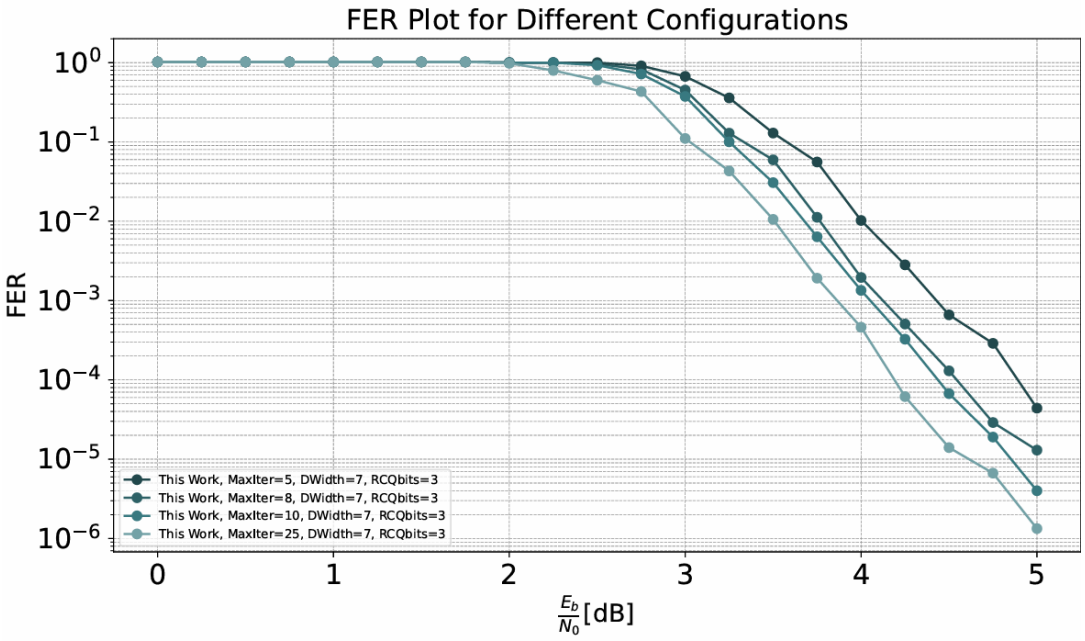


Figure 7.1: FER for different number of iterations

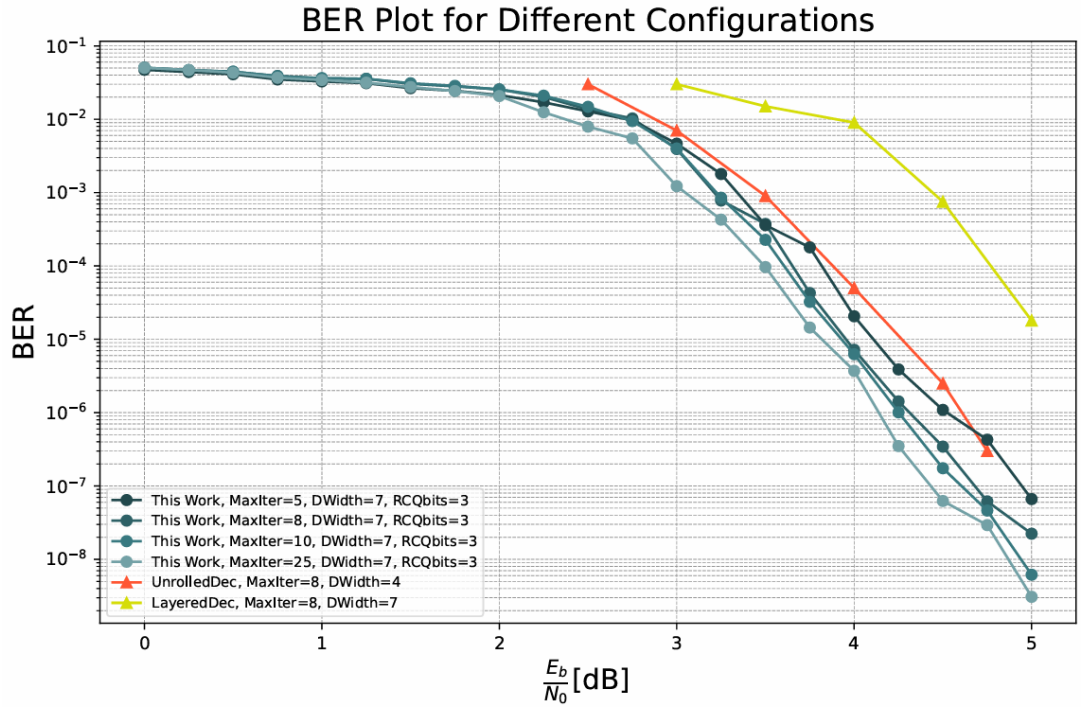


Figure 7.2: BER comparison with other (648,540) decoders

7.2 Synthesis results

To gain a clearer understanding of how various parameters are impacted by different design choices, the decoder is synthesized multiple times using different sets of parameters. However, in all cases described in table 7.1, the technology library used is $65nm$ and the LDPC code considered is the (648,540).

$bits$	n° iterations	pipe registers	RCQ	Bottom-Up syn	$t_{clk}[ns]$	$area[mm^2]$
5	2	7	NO	NO	2.6	3.18
5	3	10	NO	NO	2.6	4.99
6	3	10	NO	NO	3.0	6.04
7	3	10	NO	NO	3.4	7.37
7	3	10	YES	NO	3.3	5.42
7	3	10	YES	YES	4.1	4.22
7	6	19	YES	YES	4.1	8.98
7	10	31	YES	YES	4.1	15.27
7	25	76	YES	YES	4.1	39.07

Table 7.1: Synthesis results with different parameters

By comparing the results with one another, several considerations can be made:

- t_{clk} is not affected by iteration number, since the use of pipeline isolates the critical path and increasing iteration number just replicates that same path more times
- bit-length increases both t_{clk} and area, for obvious reasons
- RCQ introduction reduces both t_{clk} and area
- Bottom-up synthesis increases t_{clk} but decreases area

The final two points on the list require additional explanation. Regarding the effect of RCQ introduction:

- t_{clk} is slightly reduced because the critical path changes: before RCQ introduction the critical path was the one traversing the 7-bit CN layer instance; with RCQ introduction all the CN layers are synthesized with less bits (3 bits for the first 10 iteration) so their delay is reduced, and the new critical path becomes the one traversing the Reconstruction table and the 7-bit VN layer, which happens to be slightly faster than the former one.
- area is reduced because the introduction of RCQ reduces all Check Node and all inter-layer connection parallelism from 7 to 3 bits. RCQ introduces R and Q tables, but their area overhead is compensated by the massive area reduction of CN layers.

Regarding Bottom-up compilation and synthesis, this technique is mandatory for a number of iterations greater than 3, because Design compiler is unable to manage the optimization of such a massive number of gates (further details are explained in previous

chapter). Each layer is synthesized considering the clock period of the slower component, which is VN layer, so:

- all other blocks are synthesized with a less stringent constraint on timing, resulting in smaller sizes and contributing to an overall area reduction
- the critical path is still passing through Reconstruction layer and VN layer, but there is no cross-block optimization to reduce delay even more, resulting in a t_{clk} increase

The decoder used for comparison with other implementations is synthesized with 10 iteration and the post-synthesis netlist is tested using UVM testbench. The inputs and golden results used are taken from the C++ model, and the UVM report summary is depicted in figure 7.3.

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :    8
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [UVMTOP]     1
# [uvm_test_top]     3
# ** Note: $stop    : ../../../uvm_tb/top.sv(45)
#    Time: 361 ns  Iteration: 60  Instance: /tb_top
# Break in Module tb_top at ../../../uvm_tb/top.sv line 45
# Stopped at ../../../uvm_tb/top.sv line 45
#  quit -code [expr [coverage attribute -name TESTSTATUS -concise] >= 2 ? [coverage attribute -name TESTSTATUS -concise] : 0]
# Saving coverage data in coverage_ucdb ...
# End time: 00:58:47 on Nov 23,2024, Elapsed time: 3:07:00
# Errors: 0, Warnings: 200593
```

Figure 7.3: UVM report summary

Since C++ and synthesized RTL results coincide, all the considerations done before on decoding performances are valid also for the synthesized decoder. Now it's possible to compare all characteristic of the decoder with other similar decoders. However some decoders and synthesized using different technology libraries, so to make a proper comparison area and clock frequency need to be scaled. The scaling policy applied is the Constant Field Scaling (CFS): being the scaling factor $K = \frac{\text{channel length 1}}{\text{channel length 2}}$, area scales quadratically as $1/K^2$ meanwhile clock frequency scales linearly as K. Results are shown in table 7.2, with scaled results marked by an asterisk (*).

Compared to other decoders, the solution presented in this work achieves the lowest BER among the other (648,540) code implemented and is able to compete with higher N codes. Moreover, area occupation is comparable with other hardware solution, still maintaining a low latency. The throughput is comparable to other solutions, even better factoring in that N=648 is a smaller code length with respect to other solutions; however, it could be higher, as in the final stage of the work, it is negatively affected by bottom-up compilation caused by Design Vision's memory limitations. As described in table 7.1, when the synthesizer does the proper optimizations, the clock period achieved is $3.3ns$,

corresponding to a $300MHz$ clock frequency. Applying CFS to the 6-iteration decoder, the Coded Throughput obtained is $451Gbps$, which is 22% higher than the one reported into table 7.2 .

	[4]	[2]	[5]	This work*	This work
Technology	28nm FD-SOI	28nm FD-SOI	28nm	28nm	65nm
Implementation	Place & Route	Place & Route	Synthesis	Synthesis	Synthesis
Coded throughput	588 Gbps	455 Gbps	682 Gbps	372 Gbps	160 Gbps
Inf. throughput	494 Gbps	400 Gbps	568 Gbps	310 Gbps	133 Gbps
LDPC Code	(2048,1723)	(60000,53570)	(648,540)	(648,540)	(648,540)
Code rate	0.841	0.88	0.8333	0.8333	0.8333
Algorithm	Finite alphabet, unrolled, parallel	Adaptive degeneration	Min-sum, unrolled, parallel	OMS, unrolled, parallel, RCQ	OMS, unrolled, parallel, RCQ
Iterations	5	49	6	6	10
Quantization	3 bit	5 bit	4 bit	7 bit,3 bit RCQ	7 bit,3 bit RCQ
Area	16.2 mm^2	7.49 mm^2	1.346 mm^2	1.665 mm^2	15.3 mm^2
Clock frequency	862 MHz	373 MHz	1818 MHz	574 MHz	247 MHz
Latency	69.6 ns	134 ns	11 ns	33.1 ns	127.1 ns
SNR *(@BER=1e-6)*	–	4.50 dB	4.70 dB	4.46 dB	4.24 dB
SNR *(@BER=1e-7)*	4.95 dB	4.55 dB	4.85 dB	4.84 dB	4.60 dB

Table 7.2: Comparison of implemented decoder with the state-of-art

Chapter 8

Conclusion

In this work, it was presented the implementation of the Reconstruction-Computation-Quantization framework applied to a fully reconfigurable, fully unrolled, and parallel LDPC decoder. The architecture supports Offset-Min-Sum and Min-Sum decoding algorithms, and the objective was retaining the advantages of the unrolled structure for flooded algorithm (high throughput and control simplicity) while mitigating its downsides (high area consumption).

To tackle this challenge, a high-level functional model was developed in C++ in order to simulate and validate the decoding algorithm. Then the algorithm was implemented using SystemVerilog HDL, organizing the structure as a concatenation of different type of layers.

Both decoding performance and implementation metrics were evaluated: a C++ testbench was employed to evaluate the performances of the decoding algorithm for different numbers of iterations, plotting BER and FER in function of SNR; Synthesis was performed using a $65nm$ technology library and with different sets of parameters, using area and timing reports to observe the impact of each variation. Moreover the synthesized decoder (with 10 iterations) was tested using an UVM testbench, to make sure that high-level functional model and RTL implementation are consistent.

Compared to other decoders with same LDPC code (648,540), the solution presented achieves the lowest BER among them all and is even able to compete with higher code length codes. Additionally, RCQ reduces area consumption by more than 26% compared to a solution that does not implement it, making its area footprint less significant to that of state-of-the-art decoders, still maintaining a low latency. The throughput is comparable to other implementations, which is a good result considering the smaller code length.

While the decoder achieves the expected results, it is important to acknowledge its limitations. The main issue encountered during synthesis was the excessive RAM usage by Design Vision to perform all optimizations, forcing the use of a Bottom-Up compilation strategy. While this approach reduced RAM consumption, it limited cross-block optimizations, resulting in a clock period of $4.1ns$ instead of the potential $3.3ns$. This limitation prevented a 22% increase in throughput.

A possible improvement to this decoder is applying layer-specific RCQ structure using even more quantitation and reconstruction parameters . This work uses only 3 type of

Q/R tables, one for the input LLRs, one for the first five iterations and one for the last five ones. Layer-specific RCQ structure is capable of supporting different Q/R table for each iteration, improving even further decoding performances.

Therefore, future works should focus on searching for better RCQ parameters, which are different for each type of code. The fully reconfigurable structure of this decoder enables it to support such research, as it can implement every H_{base} matrix described LDPC code and allows for easy modification of RCQ tables and their contents using configuration files.

In conclusion, this work presented the development of a novel solution regarding the application of RCQ paradigm to an unrolled LDPC decoder. This compact, high-throughput LDPC decoder provides a practical and scalable solution for next-generation communication systems, balancing performance and resource utilization to meet the stringent demands of modern multi-user scenarios.

Bibliography

[1] Muhammad Awais and Carlo Condo. Flexible LDPC Decoder Architectures. *VLSI DESIGN. - ISSN 1065-514X. - STAMPA. - 2012:(2012), pp. 1-16*, 2012.

[2] K. Cushon, P. Larsson-Edefors, and P. Andrekson. Low-Power 400-Gbps Soft-Decision LDPC FEC for Optical Transport Networks. *J. Lightw. Technol., vol. 34, no. 18, pp. 4304–4311*, 2016.

[3] R. G. Gallager. Low-density parity-check codes. *IRE Trans. Inf. Theory, vol. 8, no. 1,*, 1962.

[4] R. Ghanaatian, A. Balatsoukas-Stimming, T. C. M¨uller, M. Meidlinger, G. Matz, A. Teman, and A. Burg. A 588-Gb/s LDPC Decoder based on Finite-Alphabet Message Passing. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 26, no. 2, pp. 329–340*, 2017.

[5] A. Hasani, L. Lopacinski, G. Panic, and E. Gras. 550 Gbps Fully Parallel Fully Unrolled LDPC Decoder in 28 nm CMOS Technology. *2022 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit): Components and Microelectronics (CME)*, 2022.

[6] Meng Li, Veerle Derudder, Claude Desset, Andy Dewilde, André Bourdoux, and Yanxiang Huang. A 100 Gbps LDPC Decoder for the IEEE 802.11ay Standard. *2018 IEEE 10th International Symposium on Turbo Codes & Iterative Information Processing*, 2018.

[7] Massimo Rovini, Nicola E. L'Insalata, Francesco Rossi, and Luca Fanucci. Vlsi design of a high-throughput multi-rate decoder for structured ldpc codes. *Proceedings of the 2005 8th Euromicro conference on Digital System Design*, 2005.

[8] C. Studer, N. Preyss, C. Roth, and A. Burg. Configurable high-throughput decoder architecture for quasi-cyclic ldpc codes. *Asilomar 2008*, 2008.

[9] Saleh Usman and Mohammad M. Mansour. An Optimized VLSI Implementation of an IEEE 802.11n/ac/ax LDPC Decoder. *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020.

[10] Linfang Wang, Caleb Terrill, Maximilian Stark, Zongwang Li, Richard D. Wesel, Sean Chen, Chester Hulse, Calvin Kuo, Gerhard Bauch, and Rekha Pitchumani.

Reconstruction-Computation-Quantization (RCQ): A Paradigm for Low Bit Width LDPC Decoding. *IEEE TRANSACTIONS ON COMMUNICATIONS, VOL. 70, NO. 4, APRIL 2022*, 2022.

[11] Engling Yeo, Payam Pakzad, Borivoje Nikolić, and Venkat Anantharam. High throughput low-density parity-check decoder architectures. *Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE Volume: 5*, 2001.

[12] Sangbu Yun, Dongyun Kam, Jeongwon Choe, Byeong Yong Kong, and Youngjoo Lee. Ultra-Low-Latency LDPC Decoding Architecture Using Reweighted Offset Min-Sum Algorithm. *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020.