Decoder with Bit-Flipping Post-Processing for 6G wireless

Michel Cancalon, *EL-BA6 Student*
Rodrigue de Guerre, *EL-BA6 Student*

***Abstract*—In the realm of emerging 6G communication networks, optimizing decoding algorithms for error correction codes is crucial in order to achieve high throughput and reliability under ultra-low latency constraints. With tranfer speed being above 100Gbps, area efficiency between 10-100Gbps/mm2 and an energy requirement of 1pJ/bit, 6G wireless needs a more powerful decoder. But such decoder can introduce quantization overhead and increase the error-rate; issues we aim to reduce through a second Bit-flipping decoder. As it happens, hard-decision algorithms offer low-complexity post-processing, which makes them a fitting choice to attempt to address these said challenges. This paper presents a study of hard-decision decoding algorithms tailored for Spatially-Coupled Low-Density Parity-Check (SC-LDPC) codes, suited for the Additive White Gaussian Noise (AWGN) channel. Unlike traditional LDPC codes, Spatially-Coupled LDPC codes introduce dependencies between the neighboring blocks, enhancing decoding performance by leveraging a mutual help of the blocks during decoding. We evaluated various hard-decision algorithms, in order to identify the most effective approach in terms of Bit Error Rate (BER) and computational efficiency. The optimal algorithm was then implemented in hardware, with a focus on parallelizing operations in order to maximize throughput and minimize latency. Additionally, we worked on optimizing memory usage within the hardware to further enhance the decoder's efficiency. Our results offer a first step towards a possible post processing decoder for 6G wireless communications.**

***Index Terms*—Bit-Flipping, SC-LDPC, AWGN, BER,**

I. Introduction

Low-Density Parity-Check (LDPC) codes, also known as Gallager codes, first introduced in 1960 by Robert G. Gallager, are a class of error-correcting codes that have become pre-eminent in modern communication networks due to their capability to approach the Shannon limit. During the evolution of LDPC codes, one proposal was to combine them with tradditional convolutional codes, which was proven to have better performance. With slighlty modified LDPC convolutional codes and a change in the construction, we now call them Spatially-Coupled LDPC (SC-LDPC) codes.

This evolution is closely tied to the development of wireless communication standards, as IEEE stipulates that LDPC codes are a mandatory part of Wi-Fi 802.11ax (Wi-Fi 6). The transition from 4G to 5G is also a driving factor in these improvements, as 5G NR uses LDPC codes for the data channels.

In this paper, we will explore the adaptation of already existing hard-decision decoding algorithms to SC-LDPC codes as a possible means of enhancing post-processing for 6G wireless communications. The following sections will detail the development of the decoding algorithm, followed by the simulation results of such an algorithm, then we'll detail the design of the hardware architecture. Finally, we will present the results of the implementation onto an FPGA and discuss the potential for future improvements.

II. Background and related works

A. An overview of LDPC codes

Low-Density Parity-Check (LDPC) codes, or Gallager codes, discovered by Robert Gallager in 1960, are a type of linear error correcting code (or linear block code). They are characterized by a sparse parity-check matrix $\mathbf{H}$, which is the key to their efficiency, offering great error-correction performance while maintaining manageable computational complexity.

The particular advantage of LDPC codes lies in their ability to perform well under a variety of channel conditions, including the AWGN in wireless communications. Their capability to approach the Shannon limit makes them particularly suitable for the high data rate and reliability demands of emerging wireless networks.

B. Introduction to SC-LPDC codes and benefits over traditional LDPC codes

Spatially-Coupled LDPC (SC-LDPC) codes, originate from LDPC convolutional codes and involve structuring the parity-check matrix in such a way that blocks of the matrix not only check the bits within their block but also bits in adjacent blocks. An SC-LDPC code and the corresponding Tanner graph can be constructed from a regular LDPC code, by introducing a spatial dimension t. To do so, consider a spatial chain of L codes ($t = 1, ..., L$). Then consider each edge of the Tanner graph at spatial position t and connect it randomly to a check node of spatial positions $t, t+1, ..., t+w-1$, with probability $1/w$, such that the local degree distributions are preserved. At each boundary insert $w-1$ virtual spatial positions at $-w+2, ..., 0$ (left boundary) and $L+1, ..., L+w$ (right boundary) to fulfill the degree distributions at the boundaries. The code bits of these virtual positions are fixed to "0" and need not be transmitted. As the code bits are "0", the edges can then be removed at the end of the coupling process.

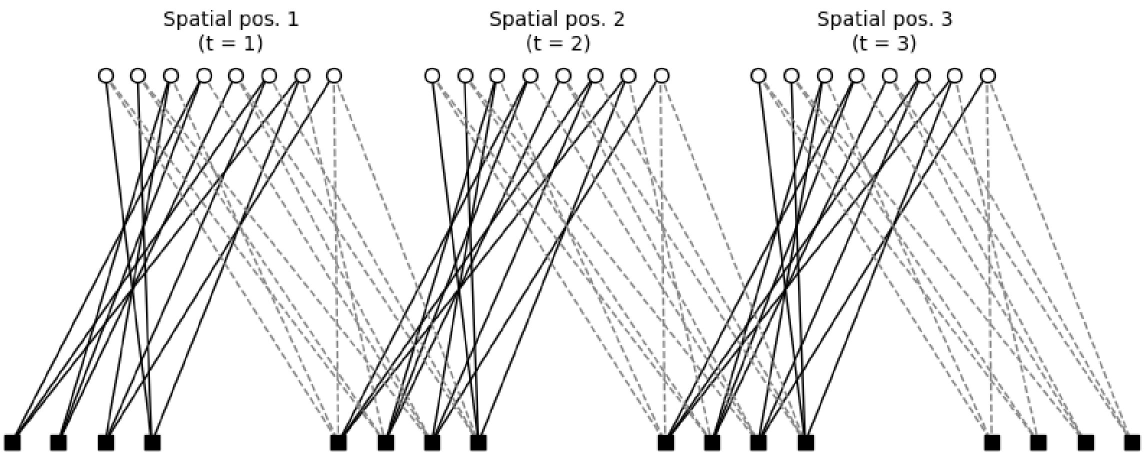


Fig. 1. Tanner graph reprensentation of an example SC-LDPC code, with $L = 3$.

Different decoding options are available for SC-LDPC codes, the most common being the windowed decoder that operates on principles of belief propagation. It is a soft-decision decoder that offers good performance at the cost of a higher computational complexity. In this paper, we will focus on hard-decision decoders, which offer lower complexity and are more suitable for hardware implementation.

Hard-decision decoders (or bit-flipping decoders) are a class of decoding algorithms that leverage the parity check matrix to correct errors in the received codeword. These decoders operate on the principle of flipping bits in the codeword to minimize the error syndromes, thereby converging to the correct codeword. The simplicity and lower computational complexity of hard-decision decoders make them an attractive choice for hardware implementations, especially in scenarios with stringent latency and energy constraints.

III. Preliminaries

Throughout this paper, we adhere to the following definitions. Boldface lowercase letters, such as $\mathbf{u}$, represent vectors, where $\mathbf{u}[i]$ denotes the i-th element of $\mathbf{u}$. Boldface uppercase letters, such as $\mathbf{B}$, represent matrices, where $\mathbf{B}[i][j]$ denotes the element at the i-th row and j-th column of $\mathbf{B}$. Blackboard bold letters, such as $\mathbb{S} = \{\cdot\}$, denote sets, with $|\mathbb{S}|$ representing the cardinality of the set $\mathbb{S}$.

We will denote $\bigoplus_{i=1}^{n} a_i$ as the XOR sum of all a_i.

For a matrix $\mathbf{H}$, we define $M(j) = \{i \mid \mathbf{H}[i][j] = 1\}$ and $N(i) = \{j \mid \mathbf{H}[i][j] = 1\}$.

The hard decision function is defined as:

$$HD(x) = \begin{cases} 1 & \text{if } x < 0 \\ 0 & \text{if } x \geq 0 \end{cases} \tag{1}$$

The sign function is defined as:

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \end{cases} \tag{2}$$

We will also employ the following notations:

- $\mathbf{c}_t$: Sent codeword at spatial position t
- $\mathbf{x}_t$: Binary to bipolar conversion of $\mathbf{c}_t$ (i.e., 0 to 1 and 1 to -1)
- $\mathbf{\hat{c}_t}$: Decoded codeword at spatial position t
- $\mathbf{\hat{x}_t}$: Binary to bipolar conversion of $\mathbf{\hat{c}_t}$
- $\mathbf{y}_t$: Received channel information at spatial position t
- $\mathbf{w}$: Noise vector with zero mean and variance σ^2

The channel studied in this paper is the Additive White Gaussian Noise (AWGN) channel, which is characterized by the following equation:

$$\mathbf{y_t} = \mathbf{x_t} + \mathbf{w} \tag{3}$$

The goal of the decoding process is to recover the sent codeword $\mathbf{c}_t$ from the received channel information $\mathbf{y}_t$.

We will also consider the following parameters for the SC-LDPC codes used in our report:

- $w = 2$: Spatial coupling factor
- $L = 50$: Number of batches per frame
- Quasi cyclic LDPC matrices: $\mathbf{H}_1$ and $\mathbf{H}_2$
- Lifting matrices (Basegraphs): $\mathbf{A}_1$ and $\mathbf{A}_2$
- $Z = 16$: Lifting factor
- $\mathbf{H}^{sc-ldpc}$: Parity-check matrix for the whole frame
- $\mathbf{H} \equiv \begin{bmatrix} H_2 & H_1 \end{bmatrix}$: Parity-check matrix for two successive codewords
- $\mathbf{A} \equiv \begin{bmatrix} A_2 & A_1 \end{bmatrix}$: Lifting matrix of $\mathbf{H}$
- $(m, n) = (8, 80)$: Basegraph size

With these parameters, we can find a codeword length of $\frac{n}{w} \times Z = 640$ and a code rate of $R \approx 0.8$. A full frame is composed of $L \times 640 = 32000$ bits.

A. SC-LDPC decoding overview

In the following report, we will consider a spatial coupling of $w = 2$ and $L = 50$ batches per frame. Because of the spatial coupling of two, the parity-check matrix $\mathbf{H}^{sc-ldpc}$ can be entirely described by two quasi cyclic LDPC matrices H_1 and H_2, each constructed from their respective lifting matrices A_1 and A_2.

Let us consider $\mathbf{c}_t$, $t = 1, 2, ..., L$ the sent codewords at different spatial positions. Encoding of the codeword resumes itself in setting the redundancy bits of each codeword such as the following equation is respected:

$$\mathbf{H}^{sc-ldpc} \cdot \mathbf{c} = \begin{bmatrix} \mathbf{H}_1 & & & & \\ \mathbf{H}_2 & \mathbf{H}_1 & & & \\ & \mathbf{H}_2 & & & \\ & & \ddots & \mathbf{H}_1 & \\ & & & \mathbf{H}_2 & \mathbf{H}_1 \\ & & & & \mathbf{H}_2 \end{bmatrix} \cdot \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \\ \vdots \\ \mathbf{c}_{L-1} \\ \mathbf{c}_L \end{pmatrix} = \mathbf{0} \tag{4}$$

Given the shape of the parity-check matrix, we can rewrite the equation for $t = 1, 2, ..., L-1$:

$$\mathbf{H} \cdot \begin{pmatrix} \mathbf{c}_t \\ \mathbf{c}_{t+1} \end{pmatrix} = \begin{bmatrix} \mathbf{H}_2 & \mathbf{H}_1 \end{bmatrix} \cdot \begin{pmatrix} \mathbf{c}_t \\ \mathbf{c}_{t+1} \end{pmatrix} = \mathbf{0} \tag{5}$$

As such, we can make the following observations:

- We need two successive codewords to check the correctness of a single one.
- Additionally, we only need the matrix $\mathbf{H} = \begin{bmatrix} \mathbf{H}_2 & \mathbf{H}_1 \end{bmatrix}$ to check the correctness of a codeword.

This essentially means that we can use standard LDPC decoding algorithms to decode the SC-LDPC codes, the only requirement is to consider the matrix $\mathbf{H}$ and to do the decoding on two successive codewords instead of one.

We will denote $\hat{\mathbf{c}} = \begin{pmatrix} \hat{\mathbf{c}}_t \\ \hat{\mathbf{c}}_{t+1} \end{pmatrix}$ the codeword used in the decoding process.

Decoding SC-LDPC codes will use the following algorithm:

Algorithm 1 SC-LDPC Decoding Algorithm

```
Inputs: Parity-check matrix $\mathbf{H}$, received word $\mathbf{y}$, number of batches $L$
  Set $\mathbf{y} = \begin{pmatrix} \mathbf{0} \\ \mathbf{y_1} \end{pmatrix}$ and $\hat{\mathbf{c}} = \begin{pmatrix} \mathbf{0} \\ -\mathbf{HD}(\mathbf{y_1}) \end{pmatrix}$
  for $t = 2$ to $L$ do
    Set $\mathbf{y} = \begin{pmatrix} \mathbf{y_{t-1}} \\ \mathbf{y_t} \end{pmatrix}$
    Shift $\hat{\mathbf{c}}$ and do the hard decision $\hat{\mathbf{c}} = \begin{pmatrix} \hat{\mathbf{c}}_{t-1} \\ -HD(\mathbf{y_t}) \end{pmatrix}$
    Decode $\hat{\mathbf{c}}$ using the parity-check matrix $\mathbf{H}$, the channel information $\mathbf{y}$, and a LDPC decoding algorithm.
    return $\hat{\mathbf{c}}_{t-1}$
  end for
```

B. *Decoding performance without post-processing*

Before looking at hard decision algorithms, we started by analysing the performance of the already exisiting windowed soft-decoder, in order to find a potential error pattern.

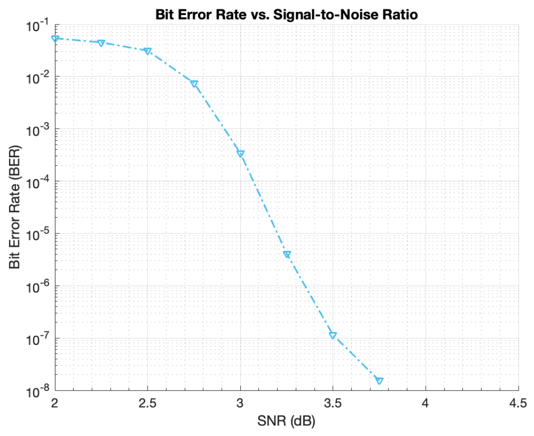


Fig. 2. BER performance of a 6G windowed decoder for SC-LDPC codes without post-processing.

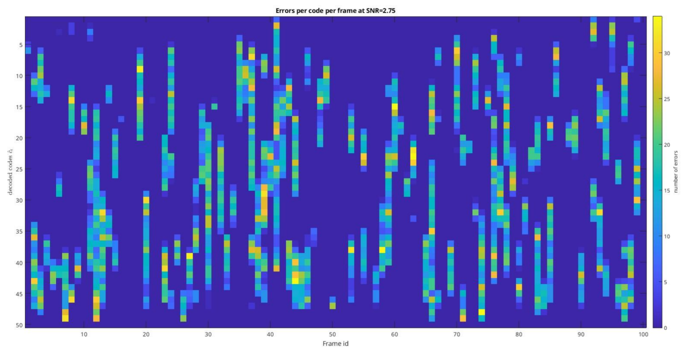


Fig. 3. Errors per code code per frame at SNR=2.75.

As shown in Figure 2, the Bit Error Rate (BER) performance of the 6G windowed decoder for SC-LDPC codes without post-processing is quite good. However, despite this strong performance, Figure 3 reveals that there is still a remaining error pattern, indicating that there is room for further improvement in the decoding process.

This error pattern is characterised by visible error propagation through strings of codewords. Few errors in a codeword lead to a high number of errors in the following codewords. This is a downside of SC-LDPC codes that is not present on regular LDPC codes. This led our motivation for the implementation of hard decision decoders, which could then potentially be re-used as post-processing for a soft decoder, effectively improving the performance SC-LDPC decoding with a reasonable increase in complexity.

IV. Theoretical results

A. *Weighted Bit-Flipping (WBF)*

As a first improvement over a regular bit flipping algorithm, we tried to implement the weighted bit flipping (WBF) algorithm. It has a simple structure, that consists of beginning by calculating a syndrome, making a hard decision on the input codeword. It then computes the reliability of the bits associated with each check nodes of the parity check matrix **H**. After that it computes an inversion function based on those previously associated bits, which will be used to determine the position of the bits to flip. Finally, the hard decision is effectuated on the memorized bits. But after working on this implementation and continuing our research, we soon realized that it would have poor results.

Algorithm 2 Weighted Bit Flipping Algorithm (WBF)

```
Inputs: $i_{max}$, $\mathbf{H}$, $\mathbf{y}$
  Initialization: Set $\hat{\mathbf{c}}$ =HD($\mathbf{y}$)
  $i \leftarrow 0$
  while $i \neq i_{max}$ do
    Step 1 For each $m \in \{0, 1, \ldots, M-1\}$, compute
        $$s_m = \bigoplus_{n \in N(m)} \hat{c}_n \cdot \mathbf{H}_{mn}$$
    Step 2 Compute the weights
        $$w_m = \min\{\sum_{n \in N(m)} \mathbf{H}^T_{mn} \cdot \mathbf{y}\}$$
    Step 3 Update the set of bits to flip
    Step 4 Flip $\hat{c}_n$ at all the required positions
    if $\sum s_m = 0$ then
      break
    end if
    $i \leftarrow n + 1$
  end while
  return $\hat{\mathbf{c}}$
```

B. *Low Complexity Hybrid Weighted Bit-Flipping (LC-HWBF)*

In order to improve the performance of the WBFA, several research have been made towards modifying the original algorithm, to make it more reliable and flip bits more accurately. Among these, algorithms such as the Modified WBF (MWBF) or the Improved MWBF (IMWBF) were proposed, but we found the approach of the Low Complexity Hybrid Weighted Bit-Flipping (LC-HWBF) to be particularly interesting. It leverages a blend of soft and hard decision, while maintaining a hard decision at the output. But even though from the research point, results where promising, our implementation of the algorithm was unfortunately unsuccessful and presented poor (close to none) and inconsistent results.

Algorithm 3 Low Complex Hybrid Weighted Bit Flipping Algorithm (LC-HWBF)

```
1: Initialization:
2: for $m = 0$ to $M-1$ do
3:   for $n = 0$ to $N-1$ do
4:     Calculate $w_m = \min_{n \in N(m)}\{|y_n|\}$
5:     Calculate $w'_{m,n} = \min_{i \in N(m)\setminus\{n\}}\{|y_i|\}$
6:   end for
7: end for
8: Set initial iteration $k = 0$ and maximum iterations $k_{max}$.
9: Step 1: Check Node Update Process
10: while $k < k_{max}$ do
11:   for $m = 0$ to $M-1$ do
12:     Compute syndrome bits $S_m = \sum_{n=0}^{N-1} \hat{c}_n^{(k)} \cdot H_{m,n}$
13:   end for
14:   if all $S_m = 0$ then
15:     Stop decoding and output $\hat{\mathbf{c}}^{(k)}$
16:   end if
17:   Step 2: Variable Node Update Process
18:   for $n = 0$ to $N-1$ do
19:     Compute $e_n^{(k)} = \frac{1}{|y_n|}\sum_{m \in M(n)}(2S_m - 1) \cdot w'_{m,n} \cdot \theta_{\text{attn}}$
20:   end for
21:   Update decision for each $n$: $\hat{c}_n^{(k+1)} = \begin{cases} 0 & \text{if } e_n^{(k)} \leq 0 \\ 1 & \text{if } e_n^{(k)} > 0 \end{cases}$
22:   Flip bit with maximum $e_n^{(k)}$ to obtain $\hat{\mathbf{c}}^{(k+1)}$
23:   $k \leftarrow k + 1$
24: end while
25: Output $\hat{\mathbf{c}}^{(k)}$ as the decoded codeword.
```

C. *Gradient Descent Bit-Flipping (GDBF)*

The Gradient Descent Bit-Flipping (GDBF) algorithm is an innovative approach to decoding low-density parity-check (LDPC) codes that combines the simplicity of traditional bit-flipping methods with the optimization power of gradient descent. By formulating the decoding process as the maximization of a non-linear objective function, the GDBF algorithm flips bits to improve the correlation between the received channel information and a valid codeword while minimizing error syndromes. This dual focus enables the GDBF algorithm to navigate the solution space more effectively, reducing the likelihood of decoding failures and significantly enhancing error correction performances compared to conventional bit-flipping algorithms.

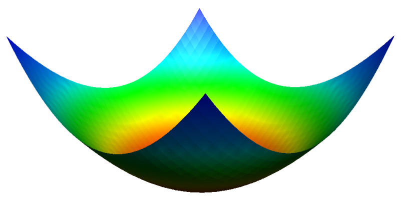


Fig. 4. Gradient descent 3D representation.

As it happens, it represents a more modern and advanced approach compared to its predecessors, the Weighted Bit-Flipping (WBF) and Modified Weighted Bit-Flipping (MWBF) algorithms. The GDBF algorithm leverages the principles of gradient descent to systematically improve the decoding performance of LDPC codes. Unlike traditional bit-flipping methods, which often struggle with local minima and sub-optimal performance, the GDBF algorithm introduces a non-linear objective function that guides the bit-flipping process. The main goal of the GDBF algorithm is to maximise this said objective function:

$$f(\hat{\mathbf{x}}) \triangleq \sum_{i=1}^{n} \hat{x}_i y_i + \sum_{i=1}^{m} \prod_{j\in N(i)} \hat{x}_j \tag{6}$$

It is composed of a correlation term: $\sum_{i=1}^{n} \hat{x}_i y_i$ between $\hat{x}_i$, the estimated codeword and y_i, the channel information. The second term: $\prod_{j\in N(i)} \hat{x}_j$ represents the parity term, which indicates how close the codeword is to respecting the parity check equation. From this function we can compute its gradient $\frac{\partial}{\partial x_k} f(\mathbf{x})$, which gives us a rule to choose the position of the bit to flip. We call it the inversion function:

$$\Delta_k^{(GD)}(\hat{\mathbf{x}}) \triangleq \hat{x}_k y_k + \sum_{i\in M(k)} \prod_{j\in N(i)} \hat{x}_j \tag{7}$$

Here we can retrieve a common calculation for the parity term, which may be interesting for a future hardware implementation. Overall, the algorithm's goal being to maximize this function, it would actually make more sense to call it the Gradient Ascent algorithm.

Algorithm 4 Gradient Descent Bit Flipping (GDBF) Algorithm

```
Inputs: Parity-check matrix $H$, received word $\mathbf{y}$, maximum
    iterations $L_{\max}$, threshold $\epsilon = 0.5$
Ensure: Decoded codeword $\hat{\mathbf{x}}$
 1: Initialize $\hat{\mathbf{x}} = \text{sign}(\mathbf{y})$
 2: for $\ell = 1$ to $L_{\max}$ do
 3:     Calculate the syndrome $\mathbf{s} = H\hat{\mathbf{x}} \mod 2$
 4:     if $\mathbf{s} = \mathbf{0}$ then
 5:         return $\hat{\mathbf{x}}$
 6:     end if
 7:     for each bit $\hat{x}_k$ in $\hat{\mathbf{x}}$ do
 8:         Compute the inversion function $\Delta_k^{(GD)}(\hat{\mathbf{x}}) = \hat{x}_k y_k + \sum_{i\in M(k)} \prod_{j\in N(i)} \hat{x}_j$
 9:     end for
10:     Set $k^* = \arg\min_k \Delta_k^{(GD)}$
11:
12:     if $\Delta_{k^*}^{(GD)}(\hat{\mathbf{x}}) < \epsilon$ then
13:         Flip the bit $\hat{x}_{k^*} = -\hat{x}_{k^*}$
14:     end if
15: end for
16: return $\hat{\mathbf{x}}$
```

With this algorithm, we were able to get considerable improvements compared to the previous ones especially with its improved versions that we will detail below.

1) Single Gradient Descent Bit-Flipping (SGDBF): The first thing that comes in mind when implementing this algorithm is to flip one single bit at a time in order to be sure to converge to a local maximum on the objective function, which may correspond to the actual transmitted codeword.

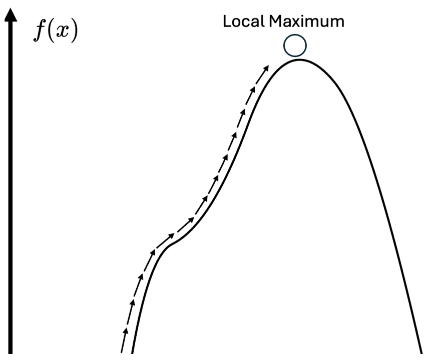


Fig. 5. Objective function representation for the Single GDBF.

But this method is quite slow and requires a lot of iterations in order to attain to the local maximum.

Hence, building on the foundation of the original SGDBF algorithm, we developed an improved version that enhances its performance further. While still employing single bit flipping to maximize the objective function effectively, in order to accelerate the decoding, we introduced an ϵ threshold, allowing the algorithm to escape quicker when close enough to the maximum. As we can see in Figure 6, where we plotted the BER for various ϵ values and even though we realized it did not have such an important impact, we determined that the optimal threshold was $\epsilon = 0.001$.

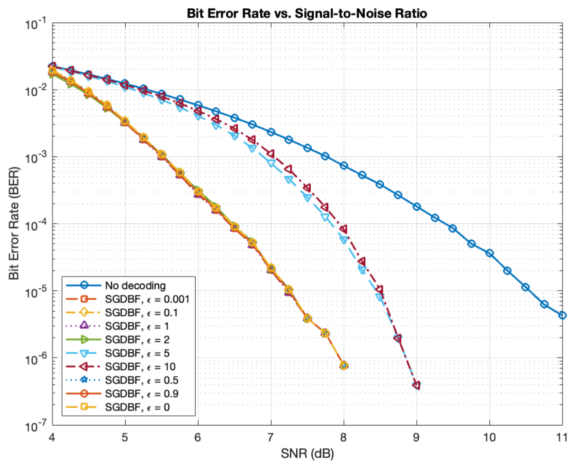


Fig. 6. Comparaison of results for a variation of the escaping threshold ϵ for the Single GDBF.

The advantage of this threshold is that it balances the need for a quicker escape while maintaining convergence towards the correct solution. The inversion function also plays a crucial role in this process by determining which bits to flip, ensuring that each iteration of the algorithm moves closer to minimizing errors. These refinements have demonstrated significant performance gains in our simulations, solidifying the SGDBF algorithm as a superior choice for modern error correction needs.

2) Multi Gradient Descent Bit-Flipping (MGDBF): One of the improvements we considered for the Gradient Descent Bit-Flipping (GDBF) algorithm was to flip multiple bits simultaneously, aiming to achieve faster convergence to a local maximum.

This approach, known as the Multi Gradient Descent Bit-Flipping (MGDBF) algorithm, demonstrated that it converges faster and reaches a higher maximum compared to the SGDBF algorithm. The faster convergence is achieved by flipping multiple bits simultaneously, which allows the algorithm to make larger steps towards the optimal solution at each iteration.

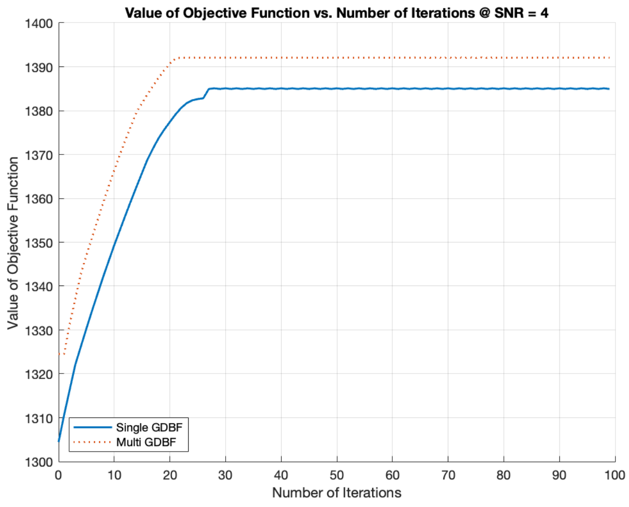


Fig. 7. Single vs. Multi GDBF objective function at SNR=4.

After plotting the objective function of the SGDBF and of the MGDBF at a fixed SNR, we can indeed confirm from Figure 7, that the MGDBF converges faster and to a higher maximum than the SGDBF.

However, as shown in Figure 10, we observed that this method could cause the objective function to fall in an oscillating motion and fail to ever reach the desired maximum. This undesired oscillation occurs because flipping multiple bits at once can sometimes push the solution away from the optimal path, leading to repeated over-corrections. To mitigate this issue, we introduced a dynamic switching mechanism that reverts the algorithm to single bit flipping when necessary.

To implement this mechanism effectively, we introduced a threshold parameter θ, which plays a crucial role in determining when to switch from multi-bit flipping to single bit flipping. Our experiments revealed that this threshold significantly impacts the BER performances of the algorithm, as shown in Figure 8. After analysis, we determined that the best configuration was with $\theta = -0.5$ and we adopted this value in all subsequent calculations and experiments.

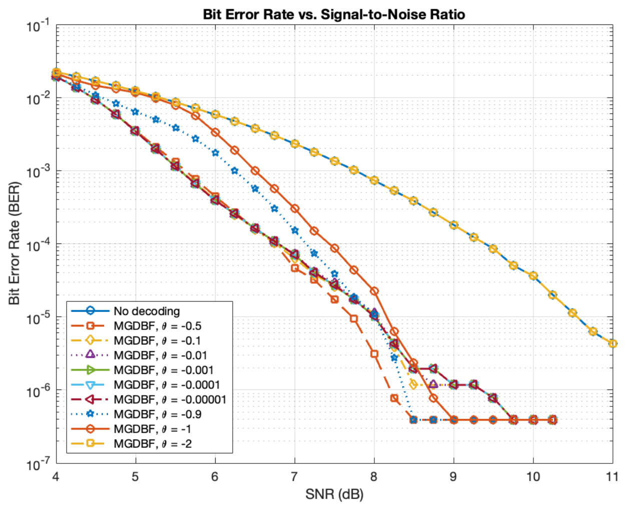


Fig. 8. BER vs. SNR for several escaping threshold θ.

Now in order to further improve the MGDBF algorithm, most of our work was focused on the escaping techniques in order to properly break out of the previously mentioned oscillating motion and dynamically switch to the single bit flipping mode. In order to do so, one must first detect a downward movement, as shown in Figure 9. After a downward move, one of the ways to escape the said oscillation could be to just move on to another local maximum, but during our research, this often turned out to lead to a trapped search point, instead of the actual transmitted codeword.

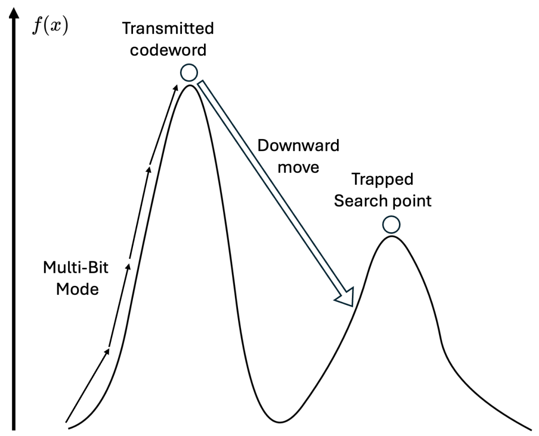


Fig. 9. First idea of escape process.

Hence another idea that struck our mind was to just go backwards in single bit flipping mode after detecting a downward move. But as shown in Figure 11, this didn't turn out to be so efficient either, as it often implied a lot of iterations backwards, before finding the local maximum.

The final and most effective escape technique we implemented is shown in Figure 12. We found that by keeping the last successful step in memory, the algorithm could go back

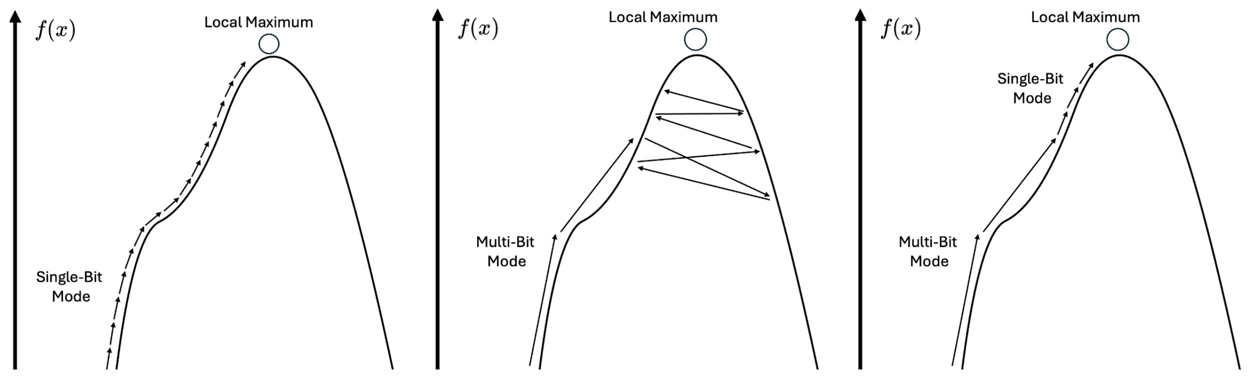


Fig. 10. Convergence behavior of a search point in GDBF decoding process

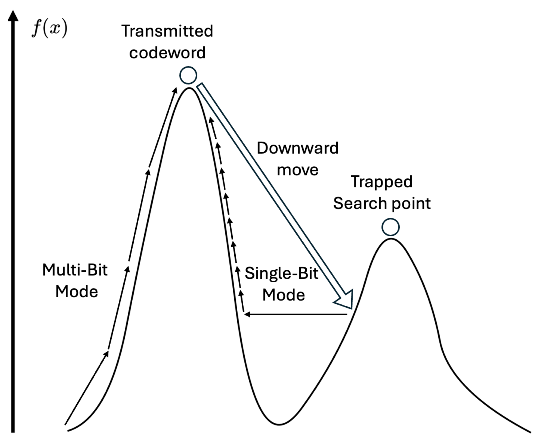


Fig. 11. Second idea of escape process.

to this step upon detecting a downward move and switch back to single bit flipping mode there. This method provided the best results in terms of convergence speed and accuracy.

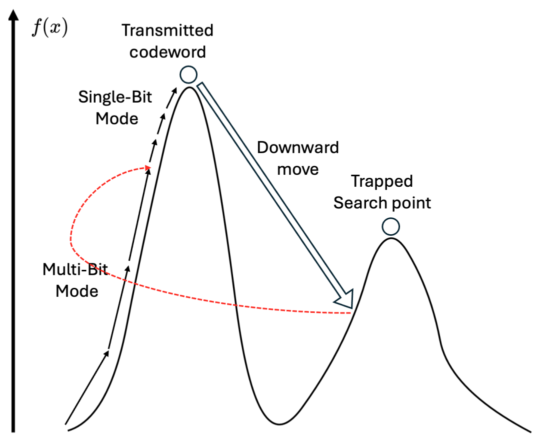


Fig. 12. Final idea of escape process.

Through these refinements, the MGDBF algorithm demonstrated significant improvements in error correction performance. The ability to dynamically adjust the bit-flipping strategy based on the algorithm's progress allowed for more efficient navigation of the solution space. By avoiding the pitfalls of oscillation and ensuring more reliable convergence to the correct codeword, the enhanced MGDBF algorithm showed a crucial improvement in Bit Error Rate BER performance and overall decoding efficiency.

After finding the best escaping technique, we also worked on an improved objective function. From the papers we read, we found that we could improve it by adding to static coefficients α and β:

$$\tilde{f}(\hat{\mathbf{x}}) \triangleq \alpha \sum_{i=1}^{n} \hat{x}_i y_i + \beta \sum_{i=1}^{m} \prod_{j \in N(i)} \hat{x}_j \tag{8}$$

This also of course led to the following modified inversion function:

$$\Delta_k^{(GD)}(\hat{\mathbf{x}}) \triangleq \alpha \hat{x}_k y_k + \beta \sum_{i \in M(k)} \prod_{j \in N(i)} \hat{x}_j \tag{9}$$

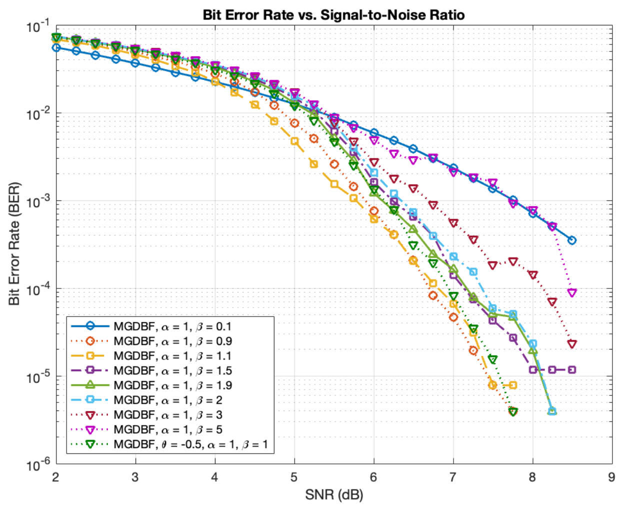


Fig. 13. BER vs. SNR with improved objective function for a fixed α.

We then tried a number of different configurations, first by fixing the α coefficient in order to find the best β; then doing the same the other way around.

In Figure 13, we can see that, even though changing β doesn't bring the best BER results, it can still have a significant impact depending on its choice. From our results, it turns out that $\beta = 0.9$ and $\beta = 1.1$ show the best configuration.

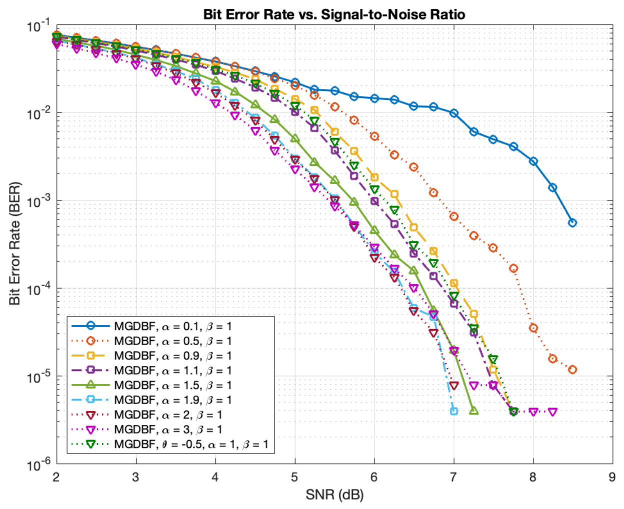


Fig. 14. BER vs. SNR with improved objective function for a fixed β.

Then in Figure 14, we can see straight away that α has a more significant impact. Depending on its choice, it can have a critical impact on the BER. Again, from our results, we find that $\alpha = 1.9$ and $\alpha = 2$ are best fitted.

D. Final theoretical results

After implementing the previously mentioned algorithms and their improved versions, we plotted their respective BER in order to clearly determine the one with the best performances, to then implement it in hardware.

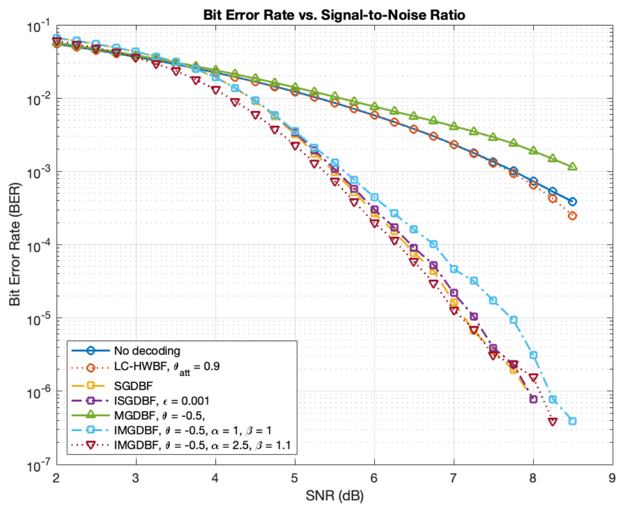


Fig. 15. BER vs. SNR of different hard decision algorithms.

Algorithm 5 Improved Multi Gradient Descent Bit Flipping (IMGDBF) Algorithm

```
Inputs: Parity-check matrix $H$, received word $\mathbf{y}$, maximum
    iterations $L_{\max}$, thresholds $\alpha, \beta, \theta$
Ensure: Decoded codeword $\hat{\mathbf{x}}$
    $\hat{\mathbf{x}} = \text{sign}(\mathbf{y})$
    mode = 0
    for $\ell = 1$ to $L_{\max}$ do
        Calculate the syndrome $\mathbf{s} = H\hat{\mathbf{x}} \mod 2$
        if s = 0 then
            return x
        end if
        Initialize empty vector flip_pos
        lowestIdx = 0, lowestValue = $\infty$
        for each $x_k$ in $\hat{\mathbf{x}}$ do
            $\Delta_k^{(GD)}(\hat{\mathbf{x}}) = \alpha \hat{x}_k y_k + \sum_{i \in M(k)} \beta \prod_{j \in N(i)} \hat{x}_j$
            if $\Delta_k^{(GD)}(\hat{\mathbf{x}}) <$ lowestValue then
                lowestIdx = $k$
                lowestValue = $\Delta_k^{(GD)}(\hat{\mathbf{x}})$
            end if
            if $\Delta_k^{(GD)}(\hat{\mathbf{x}}) < \theta$ then
                Append $k$ to flip_pos
            end if
        end for

        if mode = 0 then
            $\tilde{f}_1(\hat{\mathbf{x}}) \triangleq \alpha \sum_{i=1}^{n} \hat{x}_i y_i + \beta \sum_{i=1}^{m} \prod_{j \in N(i)} \hat{x}_j$
            for $k$ in flip_pos do
                $\hat{\mathbf{x}}[k] = -\hat{\mathbf{x}}[k]$
            end for
            $\tilde{f}_2(\hat{\mathbf{x}}) \triangleq \alpha \sum_{i=1}^{n} \hat{x}_i y_i + \beta \sum_{i=1}^{m} \prod_{j \in N(i)} \hat{x}_j$
            if $f_2 < f_1$ then
                for $k$ in flip_pos do
                    $\hat{\mathbf{x}}[k] = -\hat{\mathbf{x}}[k]$
                end for
                mode = 1
            end if
        else
            $\tilde{f}_1(\hat{\mathbf{x}}) \triangleq \alpha \sum_{i=1}^{n} \hat{x}_i y_i + \beta \sum_{i=1}^{m} \prod_{j \in N(i)} \hat{x}_j$
            $\hat{\mathbf{x}}[\text{lowestIdx}] = -\hat{\mathbf{x}}[\text{lowestIdx}]$
            $\tilde{f}_2(\hat{\mathbf{x}}) \triangleq \alpha \sum_{i=1}^{n} \hat{x}_i y_i + \beta \sum_{i=1}^{m} \prod_{j \in N(i)} \hat{x}_j$
            if $f_2 < f_1$ then
                $\hat{\mathbf{x}}[\text{lowestIdx}] = -\hat{\mathbf{x}}[\text{lowestIdx}]$
                break
            end if
        end if
    end for
    return $\hat{\mathbf{x}}$
```

As expected we can see in Figure 15 that both the (I)SGDBF and IMGDBF have the best results. But then one might ask why not just use the SGBDF. Figure 16 answers this rightful question by showing the number of iterations of both algorithm. As we can see it is way lower for the MGDBF, which makes it faster in convergence. This makes sense as the goal of both algorithm is to maximize the same objective function, but one achieves this flipping multiple bits at once, hence reducing drastically the number of iterations. But this comes at a certain cost, as shown in Figure 15, the "out of the box" MGDBF has terrible BER performances and required a lot of optimization in order to get performances close to the SGDBF. For low SNRs we even managed to have better results for the MGDBF, which makes it an ideal choice for the hardware implementation.

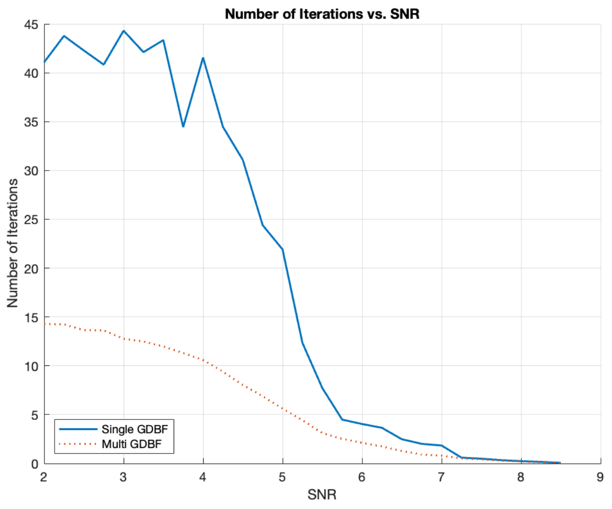


Fig. 16. Number of iterations vs. SNR for Single and Multi GDBF.

Additionally, we can also note that the MGDBF requires very little additional hardware, while still bringing a low quantization overhead due to Hard decision.

V. Decoder hardware architecture

A. Preliminaries

The encoding of the channel information y_i is done Q3.5 signed fixed point format.

In an effort to keep the same notation as the IMGDBF algorithm 5, we will from now on use $\hat{x}_i$ to represent c_i, this is due to the fact that the hardware naturally works with bits rather than with $\{-1, 1\}$ values.
This will induce changes in the objective function, inversion function and parity check equations that will be explained in section V-D.
We may also define the following function

$$\sigma(x) = \begin{cases} 1 & \text{if } x = 1 \\ -1 & \text{if } x = 0 \end{cases}$$

B. Top level overview

The full decoder is expected to receive the channel information y_i sequentially and output the decoded codeword bits $\hat{x}_i$ in the same sequential manner.
However, given the fact that the decoder operates on whole vectors rather than on individual bits, we require the hardware to incorporate a Serial to Parallel Unit (**SPU**) at the input and a Parallel to Serial Unit (**PSU**) at the output.
Figure 17 shows the high level description of the decoder, the red wire color represent busses used for the transfer of fixed point values while the black wire color represent busses used for the transfer of binary or non fixed point values.

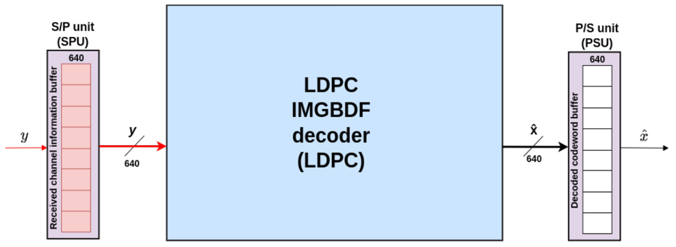


Fig. 17. High Level SC-LDPC decoder description.

The decoder operates by receiving consecutive batches of channel information that each have the length of a single codeword.
Given the fact that our SC-LDPC codes have a spatial coupling of 2, the decoder initially needs to receive two of these batches before it can start decoding the first codeword. The moment the first decoding run has started, we immediately start filling up the **SPU** with the next batch. This will allow the decoder to start it's next decoding run without having to wait as the **SPU** will be full by the time the decoder needs it.
This overlapping scheduling arrangement is permitted by the fact that the decoding will always take longer than filling up the **SPU** and outputting the whole codeword from the **PSU**.
Figure 18 shows the scheduling of the top level architecture, each color represents a single batch of information from when it's inputted (**SPU**), when it's decoded (LDPC) and when it's outputted (**PSU**).

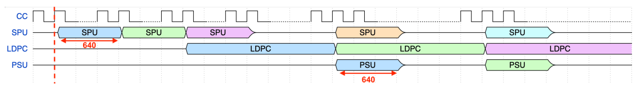


Fig. 18. Top Level Timing schedule.

C. LDPC decoder overview

Contrary to a standard LDPC decoder, the LDPC decoder used in the context of SC-LDPC must incorporate a shift register to store the $\hat{\mathbf{x}}_t$ codeword at the $(t-1)th$ spatial decoding iteration before it can be pushed out at the $(t)th$ spatial decoding iteration.
This essentially means that the codeword $\hat{\mathbf{x}}_t$ is decoded two times and only on the second spatial iteration can it be pushed out to the PSU.

We introduce the Shift Buffer x ($\mathbf{SBx}$) and Shift Buffer y ($\mathbf{SBy}$) that will hold the 2 consecutive codewords and channel information vectors for the decoding to take place on.
The buffer operations for decoding two consecutive codewords can be written as:

Spatial decoding iteration $t-1$:

1) Init:
$\hat{\mathbf{x}}_t := HD(\mathbf{y}_t)$

$$\mathbf{SBx} := \begin{pmatrix} \hat{\mathbf{x}}_{t-1} \\ \hat{\mathbf{x}}_t \end{pmatrix}$$

$$\mathbf{SBy} := \begin{pmatrix} \mathbf{y}_{t-1} \\ \mathbf{y}_t \end{pmatrix}$$

2) Decode $\mathbf{SBx}$ and output $\hat{\mathbf{x}}_{t-1}$

Spatial decoding iteration t:

1) Init:
$\hat{\mathbf{x}}_{t+1} := HD(\mathbf{y}_{t+1})$

$$\mathbf{SBx} := \begin{pmatrix} \hat{\mathbf{x}}_{t} \\ \hat{\mathbf{x}}_{t+1} \end{pmatrix}$$

$$\mathbf{SBy} := \begin{pmatrix} \mathbf{y}_{t} \\ \mathbf{y}_{t+1} \end{pmatrix}$$

2) Decode $\mathbf{SBx}$ and output $\hat{\mathbf{x}}_t$

So to summarize, we must first shift the buffers and load the top half of the buffers with the new information before we can start the LDPC decoding process on the whole buffers.

The main module of the LDPC decoder responsible for the calculations is called the Metric Calculation Unit (MCU), it requires buffers $\mathbf{SBx}$ and $\mathbf{SBy}$ as inputs and it gives the LDPC decoder these following outputs:

1) Does $\mathbf{SBx}$ respect the parity check ?
2) The value of the objective function calculated from $\mathbf{SBx}$ and $\mathbf{SBy}$
3) The index of the bit to flip in $\mathbf{SBx}$

The parity check result and objective function value are used by the decoder as control signals in the decoder finite state machine (FSM).
The bit index output is used for flipping the bits in $\mathbf{SBx}$, the decoder does that by reading the index value and by flipping the corresponding index in a bit mask ($\mathbf{BM}$) to 1.
We then flip the $\mathbf{SBx}$ bits by doing:

$$\mathbf{SBx} := \mathbf{SBx} \oplus \mathbf{BM}$$

The figure 19 shows the $\mathbf{SBx}$ and $\mathbf{SBy}$ buffers as well as the flipping operation.

The functioning of the decoder can be described using an FSM, as seen in figure 20.

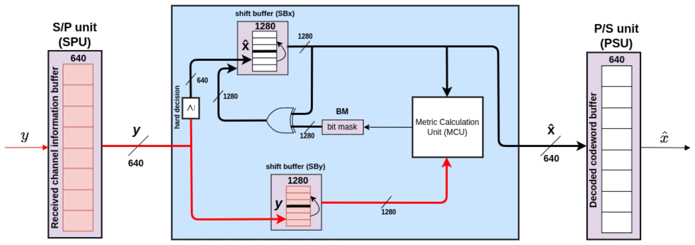


Fig. 19. LDPC decoder description.

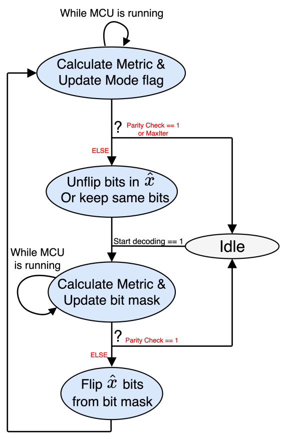

While MCU is running
Calculate Metric & Update Mode flag
? Parity Check == 1 or MaxIter
ELSE
Unflip bits in $\hat{x}$ Or keep same bits
Start decoding == 1
Idle
While MCU is running
Calculate Metric & Update bit mask
? Parity Check == 1
ELSE
Flip $\hat{x}$ bits from bit mask

Fig. 20. Decoding FSM diagram.

We note that there are two main steps per decoding iteration. In the first step, we run the MCU and flip the bits in the $\mathbf{BM}$ that the MCU tells us to flip and finally flip the bits in $\mathbf{SBx}$.
In the second step, we don't update the $\mathbf{BM}$ and we just run the MCU again.
This way, if in the second step we realise that the objective function is actually worse with the bits that were flipped in the first step, then we could reuse the $\mathbf{BM}$ from the first step and flip the bits back to their original values.

Figure 21 shows in the same color each of these two steps:

blue : step 1 of iter 1
green : step 2 of iter 1
purple: step 1 of iter 2
⋮

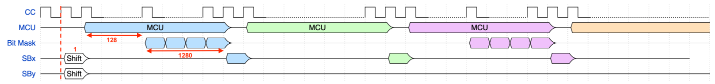


Fig. 21. LDPC decoder timing schedule.

D. Metric Calculation Unit (MCU) overview

The MCU is the main module of the decoder, it is responsible for calculating the objective function, the parity check and the bits to flip (given by the inversion function).
To describe the MCU, we must first define the calculations it

must do. Given that the bits $\hat{x}_i$ in $\mathbf{SBx}$ now represent binary values instead of bipolar values, the objective function and the inversion function can be rewritten as:

$$f(\hat{\mathbf{x}}) = \alpha \sum_{i=1}^{n} \sigma(\hat{x}_i) \cdot y_i - \beta \sum_{i=1}^{m} \sigma(\bigoplus_{j \in N(i)} \hat{x}_j) \qquad (10)$$

$$\Delta_k^{(GD)}(\hat{\mathbf{x}}) = \alpha \cdot \sigma(\hat{x}_k) \cdot y_k - \beta \sum_{i \in M(k)} \sigma(\bigoplus_{j \in N(i)} \hat{x}_j) \qquad (11)$$

And the the condition for parity check can be written as :

$$0 = \sum_{i=1}^{m} \bigoplus_{j \in N(i)} \hat{x}_j \qquad (12)$$

We can quickly notice that the $\bigoplus_{j \in N(i)} \hat{x}_j$ is common to 10, 11 and 12.
The module responsible for calculating this term is called the Common Calculation Unit (CCU) and will store the results for every i inside the CCU Buffer ($\mathbf{CCUB}$).
Once the CCU is done, we can then calculate the rest of the terms while being careful to reuse as much hardware as possible, notably :

- The $\alpha \cdot \sigma(\hat{x}_k) \cdot y_k$ term can be reused between 10 and 11.
- We can decompose the calculation of the objective function between it's parity term and it's correlation term, the parity term can directly give the result of the parity check.

This leads us to introduce these following modules and their respective mathematical operations:

name	operation	cycles
CCU	$\mathbf{CCUB}(i) = \bigoplus_{j \in N(i)} \hat{x}_j$	128
PTU	$\sum_{i=1}^{m} \sigma(\mathbf{CCUB}(i))$	128
CTU	$\sum_{i=1}^{n} \sigma(\hat{x}_i) \cdot y_i$	1280
IU	$\alpha \cdot \sigma(\hat{x}_k) \cdot y_k + \beta \sum_{i \in M(k)} \sigma(\mathbf{CCUB}(i))$	1

Fig. 22. Metric calculation unit modules

Figure 23 shows the scheduling of the MCU, it starts of with the Common Calculation Unit (CCU) and once it is done, it calculates the Parity Term Unit (PTU), the Correlation Term Unit (CTU), and the Inversion Unit (IU) in parallel.
We note that the inversion unit (IU) is able to calculate the inversion function in a single cycle. As such, for every cycle, after that the CCU is done, the MCU will output whether the bit needs to flip or not up until the end of the $\mathbf{SBx}$ buffer.

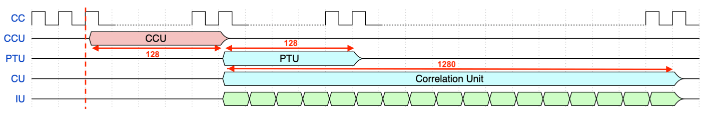


Fig. 23. Metric calculation unit description timing schedule.

Figure 24 shows the MCU and the different modules that it is composed of, we note that the LDPC matrix $\mathbf{H}$ is stored in it's basegraph form $\mathbf{A}$ inside the MCU, the details of this implementation will be discussed in the next section.

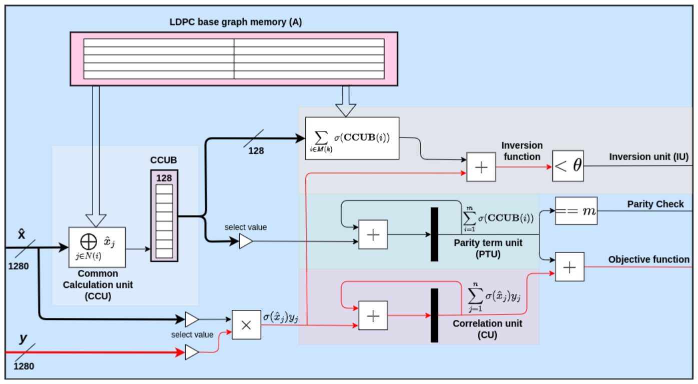

LDPC base graph memory (A)
$\sum_{i \in M(k)} \sigma(\mathbf{CCUB}(i))$
128
Inversion function
$< \theta$
Inversion unit (IU)
CCUB
128
$\hat{\mathbf{x}}$
1280
$\bigoplus_{j \in N(i)} \hat{x}_j$
Common Calculation unit (CCU)
select value
$\sum_{i=1}^{m} \sigma(\mathbf{CCUB}(i))$
Parity term unit (PTU)
$== m$
Parity Check
Objective function
$\sum_{j=1}^{n} \sigma(\hat{x}_j) y_j$
Correlation unit (CU)
$\sigma(\hat{x}_j) y_j$
select value
y
1280

Fig. 24. Metric calculation unit description.

E. Calculation details

In the last part we explained how only the basegraph is stored in the MCU, this allows us to store 8×80 5 bit values which makes up a total of 3200 bits.
If we stored the whole spare LDPC matrix, we would need 128×1280 1 bit values which would be a total of 163840 bits, so about 50 times more bits.
However, it brings some intricacies in the indexing of the matrix, thankfully they don't translate in a lot of added hardware as the operations are mainly done on powers of 2.

To calculate the CCU ($\bigoplus_{j \in N(i)} \hat{x}_j$) using the basegraph, we can use the following formula for each line i:

$$\mathbf{CCUB}(i) = \bigoplus_{j=1}^{n} \begin{cases} 0 & \text{if } \mathbf{A}(j/Z, i) = -1 \\ \mathbf{SBx}((\mathbf{A}(j/Z, i) + (j \bmod Z)) \bmod Z + j \cdot Z) & \text{otherwise} \end{cases}$$

Because $Z = 16$ in our case, the operations are simple.
We note that this calculation takes a single cycle to complete (per i), this is because we XOR the values in parallel.
XORing $n = 80$ values in total using a tree like XOR arangement could be done in $log_2(80) = 7$ stages and using 80 XOR gates.
Depending on the tool used for systesis, the program might optimize this arangement for an FPGA implementation, as such, we left the implementation of the XOR tree to the tool [1].

To calculate $\sum_{i \in M(k)} \sigma(\mathbf{CCUB}(i))$, using the basegraph, we can use formula 13 for each column k. This formula is calculated in 1 cycle (per k), this gives us the ability to determine at every cycle if the bit k should be flipped or not.
The calculation takes the sum of $m = 8$ values. Using a tree like structure of adders, it can be done in $log_2(8) = 3$ stages and using 8 adders.

[1]Vivado was used for the synthesis and implementation of the whole decoder

$$\sum_{i \in M(k)} \sigma\left(\mathbf{CCUB}(i)\right) = \sum_{j=1}^{m} \begin{cases} \sigma\left(\mathbf{CCUB}((Z - \mathbf{A}(j,k/Z) + (k \mod Z)) \mod Z + j \cdot Z)\right) & \text{if } \mathbf{A}(j/Z,i) \neq -1 \\ 0 & \text{if } \mathbf{BG}(j,k/Z) = -1 \end{cases} \tag{13}$$

However, to obtain a better critical path, this sum was manually pipelined (with one stage) as the tool couldn't optimise sufficiently enough.

VI. Hardware results

The decoder was implemented in VHDL on a ZYNQ7000 FPGA, the implementation and testbench simulations were done using Vivado.

A. *Testing and validation*

To compare the validity of the hardware implementation, we must compare it to our theoretical testbench results (our ground truth).
The testing is done by reusing the same channel information for the theoretical decoder (C code) and hardware decoder (VHDL code) and by analysing the output of both decoders.

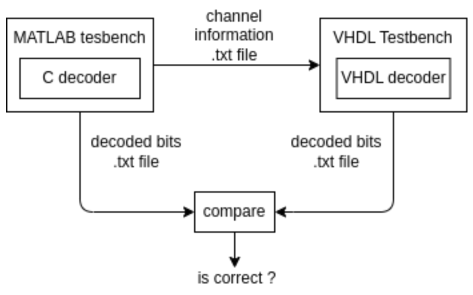


Fig. 25. Validation flow for the hardware decoder

The testing is done on a per frame basis, multiple different frames at different SNR's were tested and the results were compared to make sure the HW decoder was working as expected.

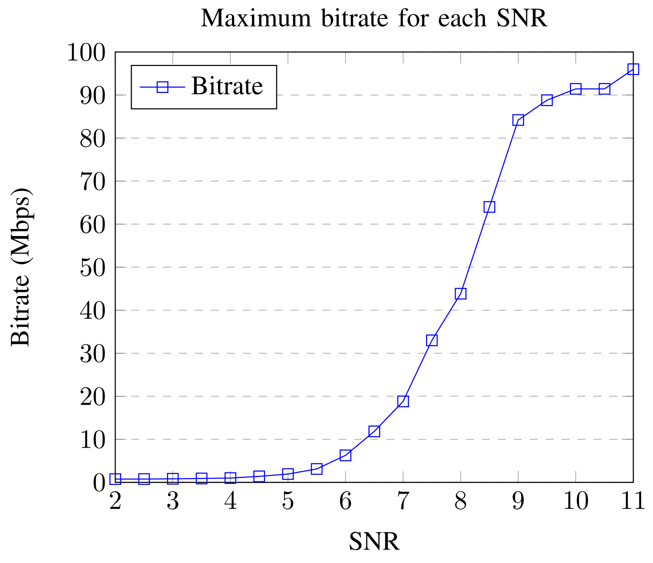
It is worth noting that in the MATLAB testbench, the channel information was truncated to the nearest $Q3.5$ fixed point value before passing it to both decoders, this was done to make sure that no approximation errors were made between the two decoders.

B. *Timing results*

The timing constraints for the FPGA were set for a 100MHz clock, with the current design, the max frequency achieved on the FPGA was 100,928542 MHz.
With a 100MHz clock, the decoding of a frame (32000 bits) takes the following amount of time per SNR for a particular frame:

SNR	time per frame
2	41.87 ms
2.5	41.67 ms
3	38.33 ms
3.5	35.06 ms
4	31.62 ms
4.5	22.89 ms
5	16.48 ms
5.5	10.25 ms
6	5.09 ms
6.5	2.71 ms
7	1.70 ms
7.5	0.97 ms
8	0.73 ms
8.5	0.5 ms
9	0.38 ms
9.5	0.36 ms
10	0.35 ms
10.5	0.35 ms
11	0.33 ms

It is worth noting that $SNR = 11$ is is the channel with which there are no hard decision errors.
If we plot the maximum bitrate for each SNR, we get the following graph:



Theoritcally, the upper limit for the decoder bitrate with a clock of $100MHz$ should be $100Mbps$, this is due do the sequantial nature of bits streaming in and out of the decoder. In practice, we reached $\approx 97Mbps$ in the perfect channel, this is good as it means that the decoder was able to reach the maximum theoritical speed (the small difference is due to the latency of the decoder and the finite frame length). This behaviour is expected as the MCU was designed to exit as

soon as it finds out the parity check is respected. This can be done in 128 cycles while the **SPU** and **PSU** buffers take 640 cycles to be ready, this means that the bottleneck are the input and output buffers (which can not be optimised further) rather than the MCU.
However, when the parity check is not immediately respected in the first iteration, the decoder must do 1408 cycles per iteration which slows down the bitrate substantially.

The critical path of the design is in the CCU which calculates $\bigoplus_{j\in N(i)} x_j$ by XORing 80 values in a single cycle.
Vivado reports it is taking 8 logic levels and 9.600ns to complete, however we note that out of this time the logic delay only takes 1.647ns while the net delay takes 7.953ns.
This large net delay may be due to the hefty amount of nets in the design, the large net density may force certain components to be placed far from each other, as such, the net delay may be larger than the logic delay (see fig 26).

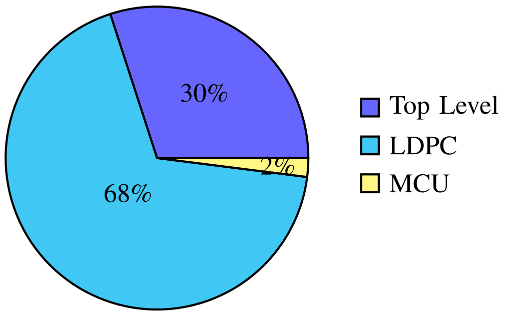
The big improvements that could be made are in the scheduling, mainly rewriting the LDPC decoder FSM and adding a second **BM** so that the MCU can continuously flip bits in the **BM** without being in states where the MCU calculates it's different functions without flipping bits.
Another even bigger improvement could be in the MCU itself, as it is now, the MCU parallelises some calculations like the CCU and the IU. For example in the CCU, the decoder is able to read a whole line of the basegraph (80 values), index the correct values inside the **SBx** buffer, and XOR them all up, all of it in a single cycle. This would probably take a lot of cycles for a normal CPU to achieve.
However, there is evidently room to parallelize even more. Calculating two consecutive values in the CTU at once and doing two IU calculations in parallel could already reduce the number of cycles per iteration by almost two (1408 $\rightarrow$ 768 cycles).
This is in our opinion, one of the most promising areas to explore if we want to increase the speed of the decoding.
However, in an effort to keep the area, resources and power consumption low, we haven't implemented these improvements.

C. Area results

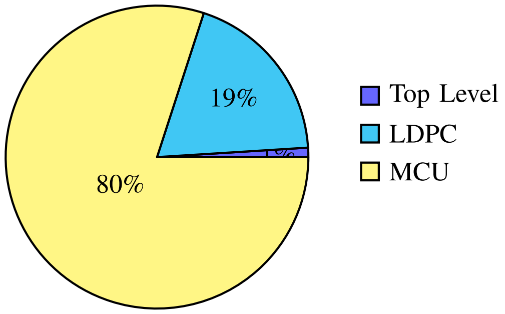
The ressources used on the FPGA are as follows:

Ressource	Used
LUT	8778 (16.50%)
FF	18966 (17.83%)
DSP	0 (0%)
BRAM	0 (0%)

We can also report the number of resources used per Module



FF's used per module



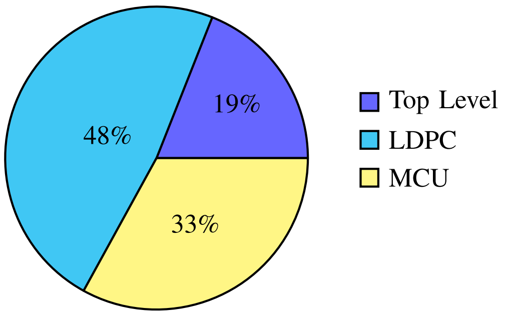
LUT's used per module

This makes sense as the MCU does most calculations and as such uses the most LUT's. The LDPC decoder on the other hand is mostly composed of shift registers, hence the heavy use of FF's.

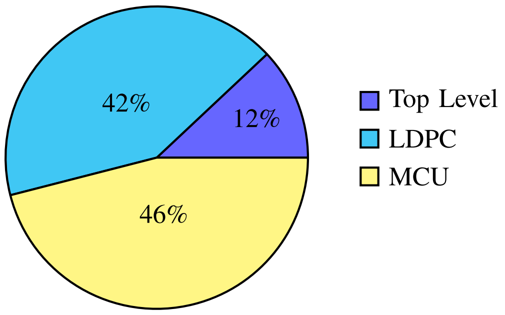
Another area of interest is the number of leaf cells and nets used in the design. Leaf cells are the smallest logic elements in the FPGA, they can either be FF's or LUT's. Knowing the number of leaf cells can be used to give an idea of the surface used by the design.

Leaf cells	30729
Nets	51459

Per module this gives us the following results:



Leaf cells used per module



Nets used per module

We can also visualize the FPGA layout of the design in figure 26.

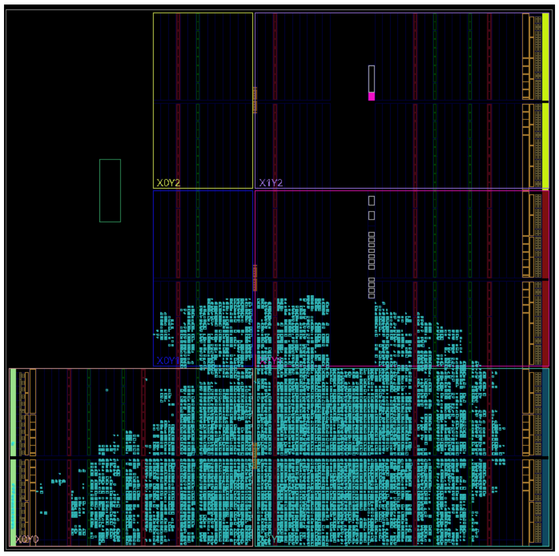


Fig. 26. Design implementation on ZYNQ7000 FPGA.

Overall we can notice that the design uses a lot of nets. This is due to the fact that the design has some very wide buffers and matrices that are interconnected inside the MCU.
Here is a summary of all the buffers used in the current design:

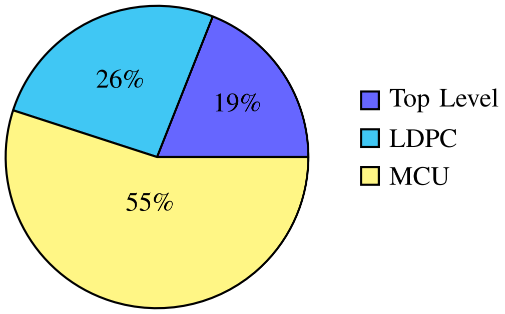
Buffer	Lenght	Width (bits)
SPU	640	8
PSU	640	1
SBx	1280	1
SBy	1280	8
BM	1280	1
CCUB	80	1
A	8×80	5

D. Power results

We got the following power predictions from Vivado:

Dynamic Power	Clocks	0.048W
	Signals	0.089W
	Logic	0.046W
	I/O	$<$ 0.001W
Static Power		0.107W
Total		0.291W

The total (predicted) power consumption is $0.291W$ which is within the limits of the ZYNQ7000 FPGA.
If we look at the dynamic power used per module, we obtain the following proportions (out of $0.184W$):



Dynamic power used per module

The specific part that consumes the most dynamic power is the internal counter in the MCU, this makes sense as this signal is used as an index in many calculations and it is constanly updated. The slice containing that counter has the biggest fanout (number of nets connected to it).
However that part still only consumes about 1% of the total power, which doesn't make it an exaggerated share.

It is worth noting that Vivado implements strategies to drastically reduce power consumption, notably by using clock enable signals to reduce the dynamic power.
By default, if we don't specify any timing constraints, the tool will assume that all the buffers and logic are updated every cycle. This makes dynamic power consumption grow exponentially large, in our example, Vivado reported about 94W of dynamic power used almost exclusively by signals and logic.
Specifying timing constraints allows the tool to automatically implement clock enable signals on most of the large buffers and logic (mainly the logic inside the MCU), and it probably also allows the tool to calculate the power used more precisely.

E. Conclusion on the Hardware

While the design is able to fit inside an FPGA and uses what we think is a reasonable amount of power and resources, it is unfortunately quite slow.

The main issue of an FPGA compared to an ASIC is that the FPGA has a lot of interconnects that can slow down the design, mainly by increasing the net delay. Plus, given that an FPGA is made to be re-configurable, the layout and mapping of the hardware onto the FPGA might not be optimal.

An ASIC on the other hand, with some good floor planning, could avoid congestion issues that an FPGA has. This could drastically decrease net delays and allow for much higher clock speeds.

Of course, leveraging more parallelism inside of the MCU could also be a good way to increase the speed of the decoder, this would of course be at the cost of increased power consumption and area, however an ASIC could quite certainly be able to handle some of these increased requirements.

Finally, a better scheduling of the decoder could certainly help, mainly in the LDPC FSM.

VII. Conclusion

This paper presented the development and implementation of a hard-decision decoder, tailored for SC-LDPC codes. After analysing several algorithms, the MGDBF with modified escaping conditions, proved to be the pre-eminent choice. Its hardware implementation on a ZYNQ7000 FPGA, configurable for different SC-LDPC codes, showed reasonable area and power consumption results. For future improvements, we would seek towards better scheduling, more parallelization and an ASIC implementation, that could lead to enhanced decoding speed.

References

[1] Alexandre Graell i Amat, Laurent Schmalen, “Forward Error Correction for Optical Transponders”, B. Mukherjee et al. (Eds.), Springer Handbook of Optical Networks, Springer Handbooks, pp. 177–248, April 2020.

[2] M. K. Roberts, A. Parthibaraj, “A Comparative Review of Recent Advances in Decoding Algorithms for Low-Density Parity-Check (LDPC) Codes and Their Applications,” Archives of Computational Methods in Engineering, vol. 28, pp. 2225–2251, 2021, https://doi.org/10.1007/s11831-020-09466-6.

[3] A. E. Pusane, A. J. Feltström, A. Sridharan, M. Lentmaier, K. Sh. Zigangirov, and D. J. Costello, Jr., “Implementation Aspects of LDPC Convolutional Codes,” IEEE Transactions on Communications, vol. 56, no. 7, pp. 1060–1069, July 2008.

[4] A. R. Iyengar, M. Papaleo, P. H. Siegel, J. K. Wolf, A. Vanelli-Coralli, G. E. Corazza, “Windowed Decoding of Protograph-Based LDPC Convolutional Codes Over Erasure Channels,” IEEE Transactions on Information Theory, vol. 58, no. 4, pp. 2303–2316, April 2012.

[5] I. Ali, J.-H. Kim, S.-H. Kim, H. Kwak, J.-S. No, “Improving Windowed Decoding of SC LDPC Codes by Effective Decoding Termination, Message Reuse, and Amplification,” IEEE Access, vol. 6, pp. 9336–9346, 2018, https://doi.org/10.1109/ACCESS.2017.2771375.

[6] T. Wadayama, K. Nakamura, M. Yagita, Y. Funahashi, S. Usami, I. Takumi, “Gradient Descent Bit Flipping Algorithms for Decoding LDPC Codes,” IEEE Transactions on Communications, vol. 58, no. 6, pp. 1610–1613, June 2010.

[7] T. C.-Y. Chang, Y. T. Su, “Dynamic Weighted Bit-Flipping Decoding Algorithms for LDPC Codes,” IEEE Transactions on Communications, vol. 63, no. 11, pp. 3950–3953, November 2015.

[8] Y. Ren, H. Harb, Y. Shen, A. Balatsoukas-Stimming, A. Burg, “A Generalized Adjusted Min-Sum Decoder for 5G LDPC Codes: Algorithm and Implementation,” IEEE Transactions on Circuits and Systems I: Regular Papers, 2024.

[9] C. Yue, V. Miloslavskaya, M. Shirvanimoghaddam, B. Vucetic, Y. Li, “Efficient Decoders for Short Block Length Codes in 6G URLLC,” 2022, arXiv:2206.09572.

[10] M. K. Roberts and R. Jayabalan, “An Improved Low Complex Hybrid Weighted Bit-Flipping Algorithm for LDPC Codes,” *Wireless Personal Communications*, vol. 82, pp. 327–339, 2015, doi: 10.1007/s11277-014-2210-4.

[11] T. Brack et al., “Low Complexity LDPC Code Decoders For Next Generation Standards”, Design, Automation and Test in Europe Conference, pp. 328–331, November 2010.

[12] N. Deak, T. Gyrfi, K. Mrton, L. Vacariu, and O. Cret, “Highly Efficient True Random Number Generator in FPGA Devices Using Phase-Locked Loops,” 20th International Conference on Control Systems and Computer Science, pp. 453–458, May 2015.

[13] Rinu Jose and Ameenudeen Pe, “Analysis of Hard Decision and Soft Decision Decoding Algorithms of LDPC Codes in AWGN,” 2015 IEEE International Advance Computing Conference (IACC), pp. 430–435, 2015.

[14] Namrata P. Bhavsar and Brijesh Vala, “Design of Hard and Soft Decision Decoding Algorithms of LDPC,” International Journal of Computer Applications, vol. 90, no. 16, pp. 1–6, March 2014.