POLITECNICO DI TORINO

Master of Science in Electronic Engineering

Master Degree Thesis

VLSI QC-LDPC Decoder for Post-Quantum Cryptography

Supervisor:
prof. Guido Masera

Co-Supervisor:
prof. Maurizio Martina

Candidate
Kristjane Koleci

Anno accademico 2018 – 2019

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The technological progress introduces new challenges and new problems to be handled and solved. The next big technological improvement is for sure the quantum computer, that in the next years is going to change deeply the concepts of bit, the computational capabilities of the devices and poses problems regarding security of currently employed cryptographic systems.

The asymmetric encryption systems employed nowadays are the RSA and ECC, the systems are considered secure against currently known attacks. The RSA is based on the problem of recovering prime factors of the key, this problem is hard to be solved by a classical computer but Shor developed an algorithm that if run on a quantum computer could retrive the keys in a small amount of time, the keys adopted are too small to be considered secure. The next technological revolution requires new and more complex problems to create safer cryptosystems.

The problems that could be suitable for this purpose are:

- Lattice-based cryptography: NTRU encryption.
- Code-based cryptography: McEliece cryptosystem.

In both cases the security is based on the harness of retrieving the secret key from the public key, they are called *NP-hard* problems since the computational cost increases exponentially with the dimension of the keys.
The quantum algorithms developed to break the systems do not succeed in their purpose, the result is just a reduction is the CPU cycles.

Goppa codes in the McEliece cryptosystem adopt large matrices, that can make the problem not suitable for embedded systems with limited memory, this issue requires different linear codes to be adopted. The Low-Density Parity Check codes are the best candidates. The McEliece cryptosystem for LDPC codes has a Public Key (PK) named $\mathbf{G}$ employed to add redundancy to the message $\mathbf{m}$ and the Secret Key (SK) that is used by the decoder to retrive the original message, it is $\mathbf{H}$. The encoding process add a bounded number of errors, $\mathbf{e}$, to the redundant message thus obtaining the ciphertext $\mathbf{x}$ There is an additional

important that is the syndrome, evaluated as $\mathbf{x} * \mathbf{L}^T$, if the vector is zero the message contains no errors.

$$\mathbf{x} = \mathbf{m} * \mathbf{G} + \mathbf{e} \tag{1.1}$$

The advantage in the use of LDPC codes is the reduced size of the matrices, they can be further compressed employing structured codes as QC-LDPC codes. The decoding capabilities of the codes is the same independently of the way they are constructed, the difference is in the possibility to further reduce their size and in the security threats since the information of the construction can be exploited by an attacker. The common ways to increase the security is to increase the size of the keys or increase the number of errors in the message.

The codes considered in the initial work are unstructured code and the error applied to the message is of BSC type, the errors are just flips of a number of bit, the probability to exchange a bit is given by a probability p. The decoders are divided in two categories: hard decoders and soft decoder. The hard decoding techniques are based on the bit flipping of the message, the criteria to select the bit depends on the decoder adopted. The most common ones are: Bit Flipping, Gradient Descent Bit Flipping and Probabilistic Gradiente Descent Bit Flipping.

The study of the algorithms clarifies the number of iterations required to converge that is important in order to select the best decoder and the BSC probability. The second study shifted the problem of lowering the iterations and increasing the error probability to code parameters, such as size, density and code regularity. Unfortunately the LDPC code parameters do not affect the performance of the decoder, they remain approximately the same a part from small changes. Due to this the attention moved to structured codes and a system in particular: LEDApck.

The LEDApck method is based on QC-LDPC codes, that can further reduce the size of the keys, and a Bit Flipping decoder able to correct the message in most case in a 3 or 4 iterations. The deep study of their security and ability to correct the code made the system the best candidate to be implemented in hardware.
The operations to be performed by the decoder are:

1. evaluate the Syndrome of the ciphertext and count the ones in the syndrome;
2. evaluate the Correlation;
3. evaluate the Threshold from the
4. find the positions of the Correlation that has a value higher than the threshold;
5. update the ciphertext;

Quasi Cyclic code studied in this thesis has Secret Matrix in format $\mathbf{L} = [\mathbf{L_0}, \mathbf{L}_1]$, each block is a square matrix of size $p = 15013$ and the first row is enough to store the complete matrix the next row is the previous shifted by 1. The fundamental evaluation in the

decoder is the vector by matrix multiplication both in $\mathbf{GF}(2)$ and in integer domain. The quantities to be evaluated are:

$$\mathbf{s} = \mathbf{x} * \mathbf{L}^T \tag{1.2}$$

$$Correlation = \mathbf{s} * \mathbf{L} \tag{1.3}$$

The equation 1.2 evaluates the Syndrome of the ciphertext is evaluated in $\mathbf{GF}(\mathbf{2})$ while the product in equation 1.3 is in integer domain. The length of the vector are: p for the Syndrome and $2*p$ for the Correlation. The multiplication is done only with square blocks, then the complete vectors are easily derived.

The object to be implemented is something that evaluates the product of a vector of size p, dense in general, by a square matrix of size p cyclic and sparse, the asserted positions are d that is the way the matrix is stored in memory, only by position. The cyclic nature of the matrix produces and elegant result in which the product in given by d vectors shifted by a quantity equal to the position and xored all together. The integer version of the result requires just to sum the shifted versions of the input vector.

The vector can not be stored in memory as a $1 \times p$ vector due to its size, but it is stored in rows of length N_b. The xor operation is performed on portions of this length among N_b bits of the input and N_b bits of the result. The bits of the input vector aligned with the row of the result are contained in two memory location, to produce the right set of values there are two consecutive rows of the input available, they are connected to a multiplexer that gives in output the consecutive N_b bits starting from the selection provided to the logic. The resulting vector is updated row by row instead of bit by bit. N_b is the parallelism of the architecture, it depends on the aspect ratio of the memory and can be sized considering the resources available.

The other units provided are simpler, to realized since its is required just to count the number of ones, match a value in a LUT and then flip the bit of a vector.

The other components implemented are simpler than this one, but the effort was to keep them parallelized too in order to avoid the presence of wasted cycles.

The solution is able to give a huge speed-up in the computation of the product, the time employed considering 8-bit parallelism is at most 29ms, that is lower than the 51 ms required by the RSA running on an FPGA. The parallelism can be increased and the time required by the decoder to converge is halved. The larger case tested is with 64-bit parallelism, it converges to a solution in 5ms, the penalty to pay is in terms of area but the results are promising.

The work can be improved introducing units that updates the Syndrome instead of computing it again and reconsidering the way the Correlation is computed, since splitting the product in two steps reduces the number of cycles even if the parallelism is low.

The other problem to address is the memory management since the presence of the Correlation requires a large memory and the parallelization can be limited by the number of

integers that can be provided at the same time, since wide memory are required.

The security is main concern in the use of Low density codes for cryptographic purposes, this aspect is not studied in this work, but in literature there are many works that introduced proper parameters to make the code secure against most of the known attack, coming both from classical computers and quantum computers.

Chapter 2

McEliece Cryptosystem

Cryptography studies the strategies to secure data in presence of an insecure communication channel. The basic principle is to transform the clear text in something more cryptic, thus the decryption without the proper key is too expensive in terms of resources or a code can take years to be broken.
The recent algorithms to secure data belong to two categories:

- Symmetric cryptosystems, one key is employed to encrypt and decrypt;
- Asymmetric cryptosystem, two keys are employed to encrypt and decrypt.

Asymmetric and Symmetric cryptosystems are employed for different purposes due to their practical differences and requirement. For example the asymmetric encryption and decryption take more time, but one key can be shared to everybody without affecting the security of the code. This is not true for a symmetric cryptosystem, just one key is used and it has to be kept secure.

Asymmetric key, for example, can be useful to exchange a symmetric key or to be able to carry out an authentication to a system. On the other hand the symmetric key, since it is in general faster, can be employed in a communications in which both parts can be trusted.

The system analyzed in this work is an asymmetric cryptosystem.

2.1 Asymmetric cryptosystems

The asymmetric cryptosystem is a scheme used to secure data that adopts two different keys, Public Key (**PK**) and Secret Key (**SK**). The scheme is known as Public-key cryptography.
The entity, Alice, that wants to receive secret messages through an insecure channel provides the Public Key but keeps secret the other one. The sender of a message, Bob, employees the PK to encrypt the message and send to Alice, who is able to decrypt the ciphertext through a decoding method based on SK.

The two keys are related one to the other, but:

- The method to generate the keys starts from SK and it is simple to generate PK;
- the asymmetry is in the fact that discovering SK from PK is a hard task, it is based on a problem that requires more computations to be solved.

The most common asymmetric key method employed nowadays are:

- **RSA**
  The public key of the RSA is a number derived from the product of two prime numbers, the factors selected are large in order to make the reverse problem even more difficult. The provided Public Key is the product of this two prime numbers and an additional value.

  The system employed is secure from attacks running on classical computers, for key of length 2048 and 4096 bit. But there is a class of algorithms developed for quantum computers that is able to factorize a number is $log(N)$.

  The algorithms is Shor's algorithm, developed in 1994 by Peter Shor.
- **ECC**
  The Elliptic Curve Cryptography is based on plane curves defined over a finite field, in GF(2) for example.
  The problem is the basis of key agreement schemes and digital signatures. The problem is secure against modern attacks, but Shor's algorithm is a threat for this systems, that is even less reliable than RSA[1].

2.1.1 RSA

The key generation process starts from the generation of two large and near prime numbers: p_1 and p_2. Then the steps are:

- Compute the product $q = p_1 * p_2$;
- Compute $\lambda(n) = lcm(p_1 - 1, p_2 - 1)$, the value is part of the Secret Key;
- Select an integer e without common dividers with $\lambda(n)$, but smaller then λ;
- Evaluate $d \equiv e^{-1} (mod\ \lambda(n))$

The set of Private Keys are n and e, given as public key exponent. The Secret Key is d the secret key exponent.

The encryption works starting from the clear text converted into an integer m, then the operation is:

$$c \equiv m^e\ (mod\ n) \tag{2.1}$$

The encryption is:

$$c^d \equiv (m^e)^d = m\ (mod\ n) \tag{2.2}$$

2.1.2 Shor's algorithm

The attack RSA method is the algorithm presented by P. Shor in 1994 [2], if it runs on a large enough quantum computer it can retrive the two prime numbers of SK and then evaluate SK.
The quantum factorization algorithm is organized in two parts:

- Classical part that reduces the problem;
- Quantum part, called period-finding subroutine

The classical part provides the starting point to the Quantum part, since a random number a, lower than N, is selected and tested as a divisor of the input N. The quantum part is called in case the random number is not a factor of N. The period-finding part searches a value r that is the period of the function describing the ciphertext:

$$f(x) = a^x \ mod \ N \rightarrow f(x+r) = a^{x+r} \ mod \ N \equiv a^x \ mod \ N \tag{2.3}$$

The resulting value of r is even a new a is selected, while if it is odd a new check has to be done:

$$a^{r/2} \equiv -1 \ (mod \ N) \tag{2.4}$$

in case r does not satisfy this condition one among $a^{r/2}+1$ and $a^{r/2}-1$ is a factor of N.

The speed up is given by the quantum part in the evaluation of a period, all the possible values are considered in a reasonably amount of time.

The time complexity of the algorithm is $log_2(N)$.

2.2 McEliece cryptosystem

The McEliece cryptosystem is an asymmetric encryption technique developed in 1978 by Robert McEliece[3]. The security of the system is based on a NP-complete problem, the problem of retrieving the linear code without any information on the structure of the code.

The original version of the system is based on a class of error correcting coded: Goppa codes. The design of the systems starts from the selection of n, the code length, and the number of error to be corrected t. Then it is selected an irreducible polynomial, $a(x)$ of degree t in $GF(2^m)$. From $a(x)$ the *Generator Matrix* $\mathbf{G}$ is derived, it has dimensions $k \times n$.

The code selected has dimensions (n, k) and has an efficient decoding algorithm that is able to correct t errors.
The next step is to design the other two components of the **The Public Key**. The additional matrices are:

- a $k \times k$ random and dense non-singular matrix $\mathbf{S}$, the "scrambling" matrix.
- a $n \times n$ permutation matrix $\mathbf{P}$,

These matrices are selected in order to hide the structure of the *Generator Matrix*, in this way it would be harder to exploit its structure. Then the public key is the matrix: $\mathbf{G'} = \mathbf{SGP}$. The operation generates a new code that keeps the same parameters of the original one.
The **Private key** is $(\mathbf{S}, \mathbf{G}, \mathbf{P})$.

Encryption The encryption procedure works as follows:

- The original message $\mathbf{m}$ is a *dataword* on k-bits, through $\mathbf{G'}$ a redundancy is added to it: $\mathbf{u} = \mathbf{mG'}$.
- The string obtained is a *codeword* $\mathbf{c}$ on n-bits.
- An intentional error vector $\mathbf{e}$, of weight t, is added to $\mathbf{u}$.

The **ciphertext** is:

$$\mathbf{y} = \mathbf{u} + \mathbf{e} = \mathbf{mG'} + \mathbf{e} \tag{2.5}$$

The encrypted message can be sent to an insecure channel.

Decryption The decryption phase tries to retrieve $\mathbf{m}$ from $\mathbf{y}$ in the following way:

- The inverse of $\mathbf{P}$ is computed.
- $\mathbf{y'} = \mathbf{y} * \mathbf{P}^{-1}$ is the string that has to be decoded.
- a decoding algorithm, for example the one developed by Patterson[4], is used to remove the errors in $\mathbf{y'}$ and obtain $\mathbf{mS^{-1}}$.
- the "scrambling" matrix is inverted.
- the original message is $\mathbf{m} = \mathbf{m'S}^{-1}$.

The use of Goppa codes makes the system secure against many types of attacks, but it has some important drawbacks. The use of this type of codes required large keys since all the elements has to be stored due to the randomness of the codes, this means that the system can not be adopted in devices where memory is limited.

The size proposed is not secure considering the performances of modern computer, in an article of 2008 [5] an attack running on a cluster with 200 computers broke the code in 7 days. The authors of the improved attack proposed two ways to increase the security:

- Increase the size of the matrices;
- Increase the number of errors to a higher value still correctable by the decoder, since 50 is a too conservative choice.

The explored sizes considering a security level of 256-bit for the proposed attack the sizes are (6624,5129) with 117 errors, the matrix to be stored has size considerably large: 0.91 MB.

The options to improve this systems relays in the use of different error correcting codes, but the challenge is to try to keep the security, while increasing the rate and decreasing the size of the keys.
The drawbacks of the Goppa codes can be overcame considering the use of a different family of error correcting codes.

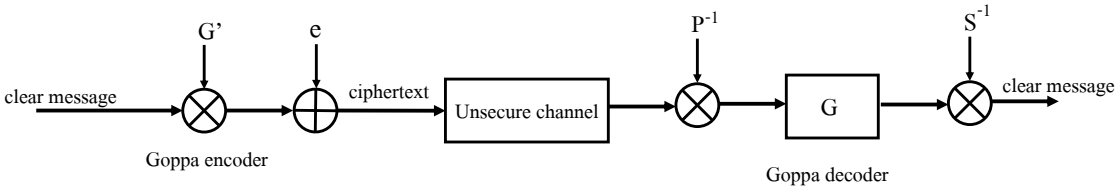


Figure 2.1: McEliece cryptosystem

The family of codes suitable for this systems must have some specific characteristics:

- a large set of possible matrices can be employed for specific dimensions and parameters, this aspect in order to avoid that an exhaustive search could retrive the original matrix;
- a fast decoding algorithm should be available;
- the public key should oscure information that can be exploited to derive **H**.

2.2.1 Security

The definitions that describes the security of a system are the following:

- **IND-CPA** is the *indistinguishability under chosen plaintext attack*
  the challengers selects a Private Key and a Public Key (this one is known to the adversary too), the adversary choose two plaintext of the same length and the challenger randomly selects one of them and encrypts it, the adversary is able to understand which message has been selected performing any encryption or computation on the ciphertext; the benefit is obtained if it is more probable to make a correct guess with additional operation than randomly choosing between the two plaintexts.
- **IND-CCA1** is *indistinguishability under non-adaptive chosen ciphertext attack*
  The initial conditions are the the same before. The additional help to the attacker is the presence of a *decryption oracle* that can be called just one time and perform operations and encryption to the message. The choice made at the end is correct. The system belong to the category if the advantage in discovering the right plaintext is non-negligible.

- **IND-CCA2** is *indistinguishability under adaptive chosen ciphertext attack*
  The only difference is that the oracle can be called multiple times for ciphertext different to the received one. The adversary wins if the guess is correct. The system belong to the category if the advantage in discovering the right plaintext is non-negligible.

The system is more secure if it is IND-CCA2, meaning that even considering the help of an oracle that can be called multiple times the effort to retrive the Secret Key is huge. The definition is referred to an asymmetric system, but can be adapted to the symmetric case.

The other definition of security that is given is usually in number of operations that the attacker executes to break the code. The security is defined with respect to a method adopted by the attacker.The security is given by a "n-bit" format, this means that the operations are 2^n. The lowest work factor to consider a system secure is 80-bit.

Information Set Decoding The most important attack on the code that has to be mentioned is the information set decoding (ISD), that is suitable to attack all other types of codes, and it tries to find low weight codewords. The starting point of the search is the presence of portions in the ciphertext that does not contain errors and corresponds to portions of the generator matrix that together produce an invertible sub matrix.
The detailed results of ISD with different algorithm on a McEliece cryptosystem based on Goppa (1024,524) code with 50 errors is summarized in Table 2.1

Table 2.1: Work Factor for Instruction Set Decoding with different algorithms

Algorithm	n-bit	Year
Stern	66.21[6]	1989
Canteaut-Chabanne	65.5[7]	1994
Canteaut-Sendrier	64.2 [8]	1998
Bernstein-Lange-Peters	58[5]	2008

The most recent article [5] describing the attack proposed the modifications in the dimension of the code in order to make Goppa codes secure against newest attacks based on Stern's searching algorithm with Grover's improvement (again a quantum algorithm). In Table 2.2 the different dimensions to obtain higher security.

Table 2.2: Work factor for different Goppa codes dimensions

Code	n-bit	Memory
(1632,1269,34)	80	56 kB
(2960,2288,57)	128	187 kB
(6624,5129,117)	256	1MB

The second possible attack is an exhaustive attack, but since there are many possibilities the algorithm will not be able to retrieve the keys in a a small amount of time.

2.3 LDPC codes

The candidates explored in this work are the *Low-Density-Parity-Check* codes, discovered by Robert G. Gallager in 1960 [9]. They are a channel capacity approaching error correcting codes and mainly used in Telecommunication in order to remove the error added by the communication channel. The big advantage of these class of codes is the low density matrix that can be exploited on order to save some space in memory, their structured version allows to save the matrix in a compressed format that takes even less space.

The big drawback is that they are not secure as the Goppa codes, but some work has been done in this sense.

The core of the thesis regards the LDPC codes employing in the McEliece cryptosystem.

Chapter 3

LDPC codes

3.1 Linear block codes and low density parity check codes

Linear Codes are a class of codes used in coding theory, their main advantage is the error-correcting capability that approaches channel capacity. The message to be transmitted is subdivided in blocks, it is encoded and then it is sent through a channel which introduces some error bit that a specific algorithm is able to remove.

The *data word* is mapped on k-bit, while the *codeword* is mapped on n-bit, where $n > k$. The *codeword* contains the same information of the *data word*, but with a redundancy. The additional bits are exploited later in order to be able to recover the clear message.

The linear code is described by a *Generator Matrix* $\mathbf{G}$ and the *Parity Check Matrix* $\mathbf{H}$.
The *codeword* from the original message is generated by:

$$\mathbf{x} = \mathbf{u} \cdot \mathbf{G} \tag{3.1}$$

The code is *systematic* if the first M-bit of it corresponds to the *dataword*, the following bit are the *parity bits*. The generator matrix for a systematic codes is:

$$\mathbf{G} = [\mathbf{I}_k|\mathbf{P}] \tag{3.2}$$

The dimensions of $\mathbf{P}$ are $k \times (n-k)$ and this is called the *Parity Matrix*, while $\mathbf{I}_k$ is the identity matrix of dimension $k \times k$.
The codeword is sent to the receiver and some errors due to the transmission are added, the received message is:

$$\mathbf{y} = \mathbf{x} + \mathbf{e} \tag{3.3}$$

The correct value of $\mathbf{x}$ can be evaluated with the *parity check matrix* $\mathbf{H}$, this is constructed in this way:

$$\mathbf{H} = [\mathbf{P}^T|I_{n-k}] \tag{3.4}$$

$$\mathbf{H} = \left[\begin{array}{cccccccc|cccccc} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{array}\right]$$

The received *word* is a *codeword* when the syndrome is:

$$\mathbf{s} = \mathbf{y}\mathbf{H}^T = \mathbf{0} \tag{3.5}$$

The condition 3.5 is valid for more than one code on n-bit: there are 2^n possible codes, among them the codewords are 2^k.

The following decoding process has to remove the errors added in the original message $\mathbf{x}$. The LDPC codes are iterative linear block codes, with a low density *Parity Check matrix*, which means a low presence of $'1'$s, for this reason the are *Low Density*.

3.1.1 Tanner Graph

The $\mathbf{H}$ can be represented by a Tanner Graph, a type of bipartite graph: one set of *variable nodes* with n elements, one for each column of the matrix, and a set of *check nodes* with $n - k$ elements, one for each row of the matrix, the connections among these groups corresponds to a $'1'$ in the *Parity Check matrix*.
In terms of graph theory the low density means that the degree of its variable nodes (d_v) is much smaller than $n - k$ and the check node degree(d_s) is much smaller than n.
The check nodes correspond to the bits of the syndrome of the message, while the variable nodes correspond to the bits of the encoded message.

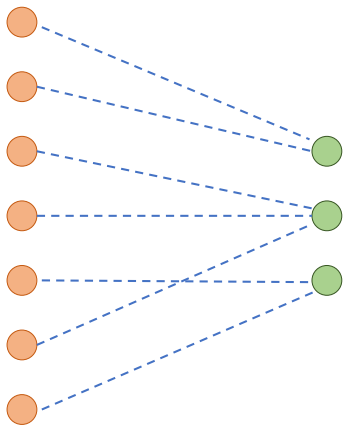


Figure 3.1: Regular (3,8) LDPC code with $n = 15$ and $r = 6$

The representation is useful in order to understand more clearly some decoders explained later.

3.1.2 Notations and definitions

The quantity $d_v(i)$, with $i = 1\ to\ N$, is the number of 1s in the i^{th} column, while $d_s(j)$, with $j = 1\ to\ K$,is the number of 1s in the j^{th} row.
The set of variable nodes is called N_v and the set of check nodes is called N_s.
The group of variable nodes connected to the j^{th} check node, $j = 1\ to\ n-k$, up to depth l is $N_{s_j}^l$, while $N_{v_i}^l$ is the group of check nodes connected to the i^{th} variable nodes, with $i = 1\ to\ n$. The complementary set of these groups is: $\overline{N_{s_j}^l}$ and $\overline{N_{v_i}^l}$.
The important distinction that can be done regards the *regular* and *irregular* codes:

- *regular*: $d_s = \gamma_s\ \forall s \in N_s$ and $d_v = \gamma_v; \forall v \in N_v$
- *irregular*: d_s and d_v has a variable degree along the nodes.

The matrix has low density if $d_s \ll N$ and $d_v \ll K$.
The graph is characterized by another important parameter: the girth. The parameter defines the minimum length of the cycles in a graph, the girth is important when it comes to design codes.
The minimum distance is the minimum Hamming distance between all the codewords that satisfies the syndrome relation with **H**.

3.2 Code construction

The code construction refers to the construction of the *Parity-Check matrix*. Different algorithms are available, in the following they are going to be described in detail.

Parity-check matrix derivation

The LDPC codes belong to two categories.

- Structured codes: **H** has a predefined structure that allows to reduce the elements to be stored.
- Unstructured codes: **H** has a structure that can not be compressed since there is no relation between the asserted position of the matrix.

The best choice is the one that generates matrices with a few (or none) small cycles, a suitable algorithm that can generate good matrices is the Progressive Edge Growth [10], which generates unstructured codes.
Structured codes are, for example, QC-LDPC codes[11] or codes based on Pseudo Difference Families[11].

Generator matrix derivation

The *Generator Matrix* is derived from the *Parity Check Matrix* in this way.
The matrix should be in the from

$$\mathbf{H} = [\mathbf{AB}] \tag{3.6}$$

Where $\mathbf{A}$ is a $r \times r$ invertible matrix, the permutation of the columns does not affect the properties of $\mathbf{H}$, then some columns exchange is done in order to have a suitable structure. In this way the systematic form of $\mathbf{H}$ can be obtained by: $\mathbf{H'} = [\mathbf{AB}] * \mathbf{A^{-1}} = [\mathbf{P}, \mathbf{I_r}]$.
The row exchange in order to produce a suitable matrix introduces additional time complexity to the code generation, in order to check the rank of the sub-matrix, find the non-independent rows and exchange it with a suitable row.
The *Generator Matrix* can be obtained:

$$\mathbf{G} = [(\mathbf{BA^{-1}})^{\mathbf{T}}, \mathbf{I_k}] \tag{3.7}$$

The step changes $\mathbf{H}$, but allows to have $\mathbf{G}$ in systematic form.
The decoder has to use $\mathbf{H'}$ to recover the message without errors; the use of $\mathbf{H}$ is not going to give the right message unless a permutation is applied on the resulting vector.

3.3 Code construction Algorithms

The algorithms described are:

- Progressive Edge Growth
- QC-LDPC

Algorithms have to generate $\mathbf{H}$ with full rank in order to be able to generate $\mathbf{G}$, the proposed methods does not include this step. The condition is more probable to be satisfied with specific parameters of the code otherwise a new run of the construction is required.

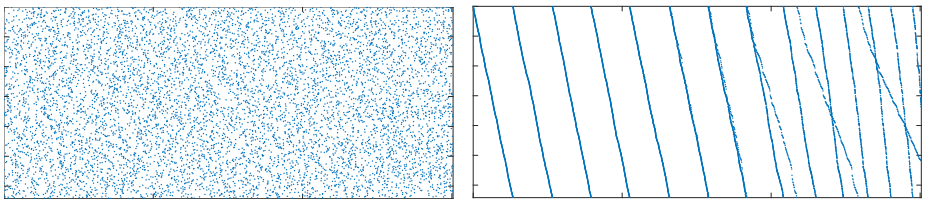
3.3.1 Progressive Edge Growth

The algorithm is able to generate codes without a relation between the asserted positions and that has a large girth, it is described in detail in article [10].
The method inserts at each step a new connection that does not include the previous paths. The pseudo-code is in Algorithm 1.
The central operation of the algorithm is to find the connections of each node and introduce a new connection among far nodes. The only change that can be introduced is in the selection of the new connection if more than one node is suitable.
The codes can be both regular and irregular, the type of matrix depends on the degree selected at the beginning: the value of d_s can be the constant $\forall k$ or not, moreover the d_v is decided by this choice.
The problem is in the generation of $\mathbf{G}$ because it can happen that $\mathbf{H}$ is not full rank, in this case the process has to be iterated again or the degree of the rows and columns has to be changed. The check to be done is to compute the rank of $\mathbf{H}$, this value has to be r, otherwise an invertible matrix $\mathbf{A}$ cannot be derived. In general the suggestion is to avoid generation of regular codes with even d_s or with one of the dimensions even.

The variations that can be added on the selection of the new node leads to matrices with different shapes. The different cases are in Fig. 3.2

Data: **c** : *check nodes*,**v** : *variable nodes*
Result: **H**
initialization: ;
for $j = 0$ *to* $n - 1$ **do**
  for $k = 0$ *to* $d_{s_j} - 1$ **do**
    if $k = 0$ **then**
      $E^0_{s_j} \leftarrow$ edge (c_i, s_j), where $E^0_{s_j}$ is the first edge connected to s_j, and c_i is a check node with the lowest check degree in the current graph arrangement $E_{s_0} \bigcup E_{s_1} \cdots \bigcup E_{s_{j-1}}$
    else
      expanding a tree from a symbol node s_j up to depth l under the current graph arrangement such that $\overline{N}^l_{s_j} \neq \emptyset$ but $\overline{N}^{l+1}_{s_j} \neq \emptyset$, or the cardinality of the $N^l_{s_j}$ stops increasing but is less than m, then $E^k_{s_j} \leftarrow$ edge (c_i, s_j), where $E^k_{s_j}$ is the k-th edge incident to s_j and c_i is one check node selected from the set $\overline{N}^l_{s_j}$ having the lowest check-node degree. ;
    end
  end
end

Algorithm 1: PEG



(a) Example of random selection for the new node of **H**

(b) Example of random fixed selection for the new node of **H**

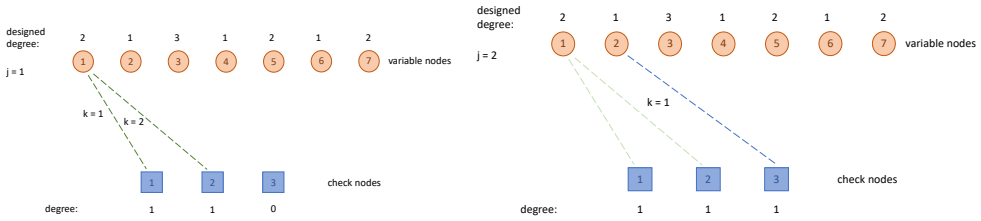
Figure 3.2: Different *Parity-Check matrices*

PEG example

The PEG algorithms works as follows:

- The variable nodes are: $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ with degree $d_s = \{2,1,3,1,2,1,2\}$.
- The check nodes are: $\{c_1, c_2, c_3\}$ with degree that depends on the construction.

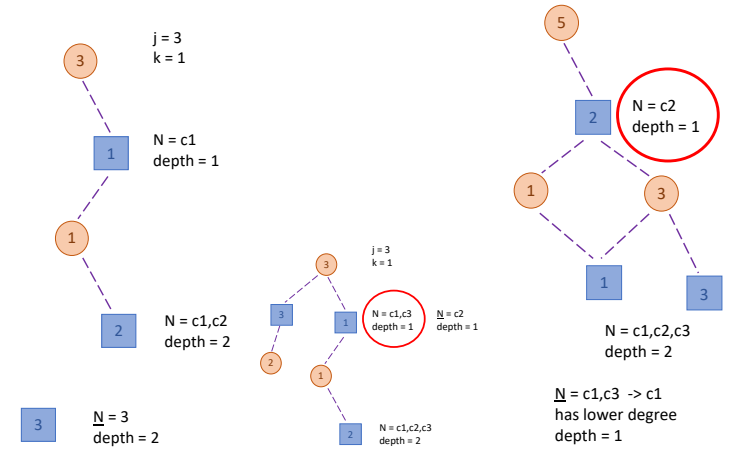
The first variable node has degree 2 and the check nodes has no connections, thus the first arc connect v_1 with c_1. The second arc from v_1 can be connected to both c_2 or c_3, c_2 is randomly selected. At this point the arrangement in Figure 3.3(a) in obtained.



(a) Variable node v_1 and arc 1,2 (b) Variable node v_2 and arc 1

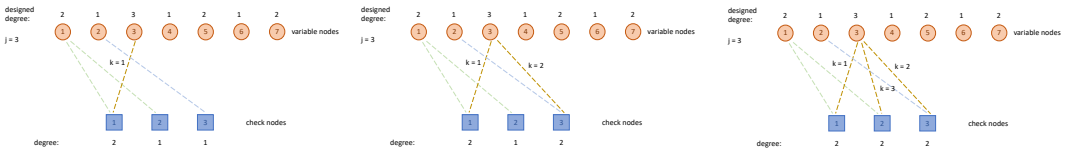
Figure 3.3: Step 1,2

The following variable node, v_3, is selected then the choice of the check node is among $\{c_1, c_2, c_3\}$, c_1 is selected since it is the first one of the list. The second arc, $k = 2$, is selected and a path starting from v_3 is expanded as in Figure 3.4(a), the remaining check node is only c_3 then this one is selected. The next arc, with $k = 3$, requires again the expansion of the tree starting from the variable node v_3, as in Figure 3.4(b) but the situation is different now because the set of possible the expansion stops at depth $l = 2$. The new connection is with check node c_2.



(a) Expansion of node v_3 with $k = 1$ (b) Expansion of node v_3 with $k = 1,2$ (c) Expansion of node v_5 with $k = 1,2$

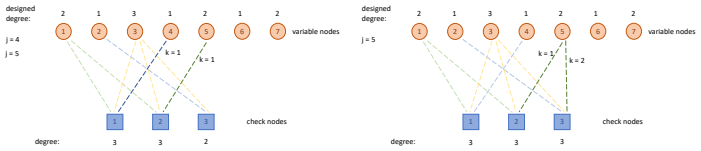
Figure 3.4: Three expansions



(a) Variable node v_3 and arc 1 (b) Variable node v_3 and arc 2 (c) Variable node v_3 and arc 3

Figure 3.5: Variable node v_3 connections.

The connections for v_4 and v_5 with $k = 1$ are straightforward since no expansion is needed: the check node selected is the first one with lower degree, the step is in Figure 3.6(a). The selection of the second connection of v_5 requires an expansion of the node(see Figure 3.4(c)), since the expansion has to be stopped before the maximum cardinality of the check node is reached there are two possible candidates c_2 and c_3, the next arc is between v_5 and one of the previous nodes.



(a) Variable nodes v_4 and v_5 con-(b) variable node v_5 connections nection.

Figure 3.6: Variable node v_3 connections.

The final arrangement of the graph is in Figure 3.7

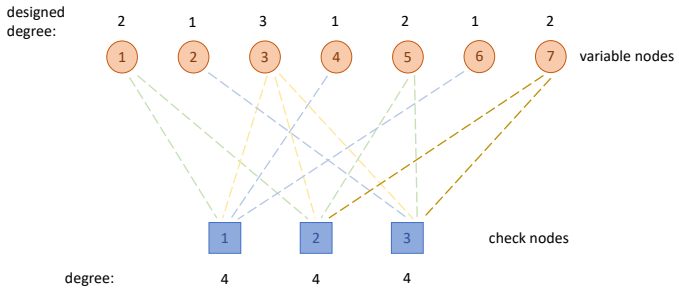


Figure 3.7: Final arrangement

PEG security The structure of the *Parity-Check Matrix* is unpredictable then the attacker can not exploit any information to recover **H** from **G**. They are still insecure against other types of attacks due to their sparsity, the only solution is to increase the size of the code but the code generation could be time consuming at this point since the rank and

the inverse of the matrix have to be computed. The study regarding their security is not available, then they can be used just to test the performances of the decoding algorithms.

3.3.2 Quasi Cyclic construction

The Quasi-Cyclic construction for LDPC codes has been presented the first time in 1967 and in [12]. The codes generated are structured and a recent work on the McEliece Cryptosystem showed that they can be adopted in order to reduce the key size[13].
Quasi-cyclic codes have the following structure:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H_{0,0}} & \mathbf{H_{0,1}} & \cdots & \mathbf{H_{0,n0-1}} \\ \mathbf{H_{1,0}} & \mathbf{H_{1,1}} & \cdots & \mathbf{H_{1,n0-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{H_{r0-1,0}} & \mathbf{H_{r0-1,1}} & \cdots & \mathbf{H_{r0,n0-1}} \end{bmatrix} \tag{3.8}$$

The block are $p \times p$ circulant matrices, in which only the first row has to be specified. The dimensions of $\mathbf{H}$ is $r \times n$, where $r = pr0$ and $n = pn0$; the dimensions of $\mathbf{G}$ are $n \times k$, where $k = pk0$. The code rate is: $k0/n0$, it depends on the number of blocks in the matrix.

The *Parity Check Matrix* for this particular case is:

$$\mathbf{H} = [\mathbf{H_0}, \mathbf{H_1}, \ldots, \mathbf{H_{n0-1}}] \tag{3.9}$$

each $\mathbf{H_i}$ block is a circular matrix with size $p \times p$.
The *Generator Matrix* is:

$$\mathbf{G} = \begin{bmatrix} & \mathbf{H_{n0-1}^{-1}}^{\mathbf{T}} * \mathbf{H_0} \\ & \mathbf{H_{n0-1}^{-1}}^{\mathbf{T}} * \mathbf{H_1} \\ \mathbf{I}, & \cdots \\ & \mathbf{H_{n0-1}^{-1}}^{\mathbf{T}} * \mathbf{H_{n0-2}} \end{bmatrix} \tag{3.10}$$

In order to be able to generate $\mathbf{G}$, $\mathbf{H}_{n0-1}$ must be invertible. The random choice of the first row can lead to a singular matrix.
The important advantage of the construction is in the computation of the inverse since it requires less operation compared to the other case, even if larger matrices are involved.The derivation of the *Generator Matrix* is less computationally expensive.

Moreover only $n0$ rows have to be saved, since the full matrix can be derived from them, the problems related to the space in memory is easily overcame. The main drawback relays in the fact that some common attacks on the code could discover easily the secret key from the public key, but some modifications on the encoder allow to reduces this risk.

The article that presents this options suggests the use of large (order of thousands) p values and small n_0 (usually between 2 and 4 circulant blocks). The rate for this arrangement is $r = (n_0 - 1)/n_0$[14].
The shape of the *Parity-Check Matrix* in Quasi-Cyclic case is in Figure 3.8.
The method produces suitable codes, but there are some problems in terms of security since the unknowns to find are less then the one of the PEG codes, unless some counter measurements are not applied the *QC* structure can be easily exploited by an attacker.

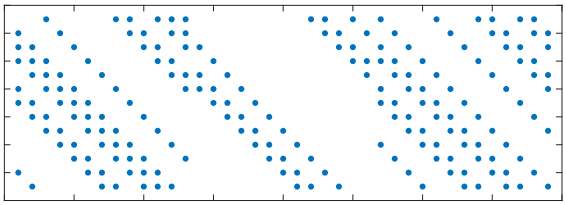


Figure 3.8: *Parity-Check Matrix with quasi-cyclic construction*

3.3.3 Girth reduction

The main purpose of the PEG construction is to reduce the number of cycles in the Tanner Graph. The minimum length of the cycle present is called *girth*. Small cycles (ex. girth 4,6,8) do not allow a proper correction of the code in most of the cases the decoder is not able to converge to the solution since it is trapped in a cycle as it is explained later.

The article[10] relates the girth of PEG codes with the dimensions of the matrix, in Fig. 3.9 it can be seen that with larger codes the maximum girth increases, the presence of small cycles is avoided.

The work presents also a lower bound to the girth, which appears to be related with codes

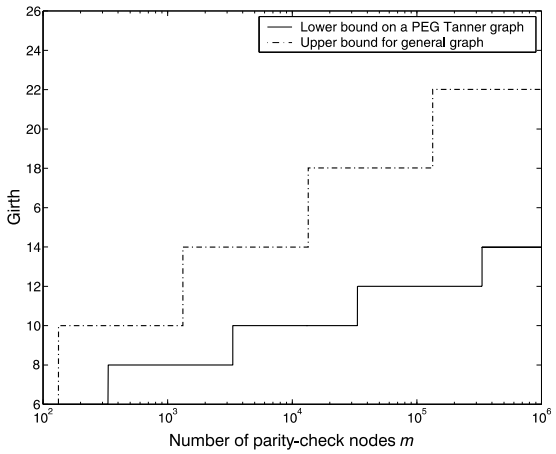


Figure 3.9: Size of the matrix and girth[10]

parameters (both for regular and irregular codes):

$$g > 2(floort + 2) \tag{3.11}$$

where

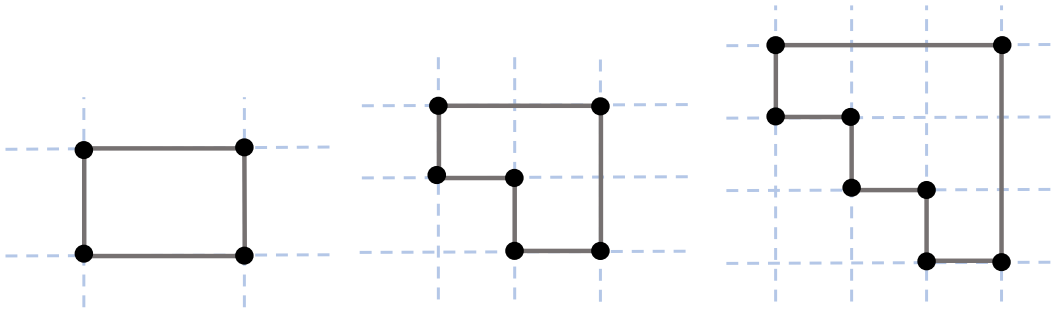
$$t = \frac{log(kd_c^{max} - \frac{kd_c^{max}}{d_v^{max}} - k + 1)}{log(d_v^{max} - 1)(d_c^{max} - 1)} - 1 \tag{3.12}$$

In case of PEG constructed codes the minimum girth can be increased changing the degree of the nodes and the size of the matrices.

The QC-LDPC, studied in [13], requires two conditions to construct matrices with high girth.

- **Lemma 1**: A circulant matrix with row (column) weight equal to 2 has length-4 cycles only if its size p is even and the difference between the positions of two ones in each row is $p/2$.
- **Lemma 2**: A circulant matrix with row (column) weight higher than 2 has local girth at most equal to 6.

The girth can be graphically represented in this way:



(a) Example of girth-4 (b) Example of girth-6 (c) Example of girth-8

Figure 3.10: Girth examples[11]

3.4 LDPC codes encoding

The encoding process requires to multiply the message by the public key and then add a bounded error on it.
The error added is supposed to be the one of a Binary Symmetric Channel (BSC) with probability p_{error}, which only flips the bit of the redundant message, $\mathbf{m} * \mathbf{G}$, with that probability.
The encoded message has the following form:

$$\mathbf{x} = \mathbf{m} * \mathbf{G} \oplus e \tag{3.13}$$

3.4.1 Minimum distance and maximum error

The probability of the BSC has to be chosen carefully since it will affect the decoding process. The codeword corresponding to the original message is unique given that $\mathbf{G}$ and $\mathbf{H}$ are defined, but there is more than one *codeword* for the matrix $\mathbf{H}$, the minimum distance between two *codewords* gives a bound to the maximum error that can be applied. The codes involved in the following analysis are huge and an exhaustive search of the minimum distance would be too time expensive, the solution adopted is to simulate the various decoding algorithms with different probabilities, then the $p_{error,max}$ could be selected.

The second problem of LDPC codes and the choice of the probability is that the error correction capabilities of codes with similar characteristics are not the same, to overcome this issue the choice is to change the construction: the QC-LDPC codes based on Difference Families.

The minimum distance problem is not analyzed in this thesis, but mentioned just for completness.

3.5 LDPC decryption

The first step of the decoding process for the McEliece cryptosystem requires a multiplication for the permutation matrix:

$$message\ to\ decoder = x * \mathbf{S} * \mathbf{G}' + e * \mathbf{P}' \tag{3.14}$$

The LDPC/QC-LDPC codes employe does not require this operation because the Private Key does not include a permutation to be generated.

The error is removed through an LDPC decoder, two classes of them are available, both based on different kind of decisions and information about the added error:

- **Soft decision algorithm**: soft decoders consider the stream of informations not only with they specific value, but also their probability to assume value $'1'$ or $'0'$, the informations on the errors introduced by the channel are taken into account.
- **Hard decision algorithm**: hard decoders consider only the stream of the data and do not include any information from the channel, the flipping decision in based mainly on the computation of the syndrome.

The following steps are the ones already explained in the general description of the McEliece cryptosystem.

3.5.1 Decoder characteristics

The decoding algorithms and codes can be classified and compared considering some figure of merit:

- **BSC**: the algorithm has to be suitable and optimized for this type of error.
- **Waterfall Region**: this is the region where the algorithm starts to correct error, it depends on the structure of the matrix and not on the decoding algorithm, but it should be analyzed if the starting point changes.
- **Error Floor Region**: the decoder correction capability stops to increase at a certain $p_{error,max}$.
- **Complexity**: the operations involved should not be complex (such as divisions, multiplications), it's better to use basic boolean operations. The complexity can be reduced not only adopting simpler decoders, but also dealing with structured codes.

- **Correction**: the error correction capability should be as higher as possible, which means that $p_{error,max}$ has to be high.
- **Iterations**: algorithms that require, in general, a lower number of iterations to converge are preferred.

The meaning of the *Waterfall* and *Error Floor* regions for the BER vs. S/N_0 is in Figure 3.11. The first region is between 30 and 45 dB, in this part the decoder starts to correct, the second region starts from 45 dB in this case the BER is more flat (in the Figure is not much clear but the variation of BER vs S/N_0 changes).
In general these are basic requirements of a decoder used in Telecommunication, while in Cryptography the requirements are different: the key point is to have messages is always successfully decrypted, it is useful to know the maximum number of errors that can be applied and the failure rate of the decoder with that value. Unfortunately the LDPC codes have a decoding rate that can not be predicted in advance.

The value can not be found with a huge number of iterations since these decrypting procedure is not meant to work on devices with a high computational capability. The parameters that may be useful are: the degree of the variable nodes and check nodes and the size of the code. Always taking into account that in many cases the maximum number of iteration is not that high, so the error correcting capability could be reduced.

The most important class od decoders are the *Belief Propagation*, based on a soft decision, and the *Bit Flipping*, based on a hard decision. The other decoding solutions are mainly based on this two algorithms.

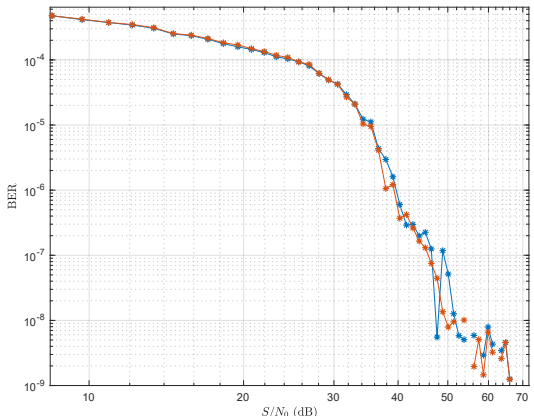


Figure 3.11: Waterfall and Error Floor region for two different methods

3.5.2 Soft Decoding algorithms

The soft decoding algorithms are all based on message passing between check nodes and variable nodes. The stream of bits considered are $'1'$ or $'0'$, with an additional value that includes the reliability of that value. The most famous one is the *Belief Propagation*

Algorithm developed by Gallager, the other soft decoding algorithms are a semplification of this one.[9]

Belief Propagation

The two set of nodes exchange the $T_{n,m}$ and $E_{n,m}$ messages:

- $T_{n,m}$ is the partial message produced by the $i-th$, with $i = 1\ to\ N$, variable node and sent to the check nodes connected to it;
- $E_{n,m}$ is the partial message produces by the $j-th$, with $j = 1\ to\ N-M$,check node and sent to the variable node connected to it.

The variable nodes produce a total information of the node, T_n, that includes the message coming from the arrival node.
On the other hand $T_{n,m}$ and $E_{n,m}$ doesn't have the total information, this is the key point of the Gallager algorithm and makes the *Belief Propagation* the best correcting algorithm.

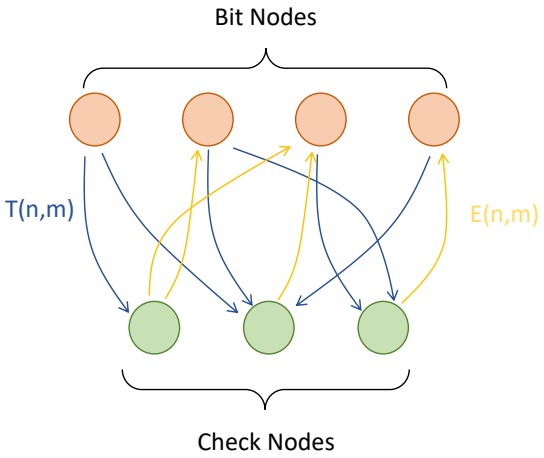


Figure 3.12: Graphic representation of the structure

The update rule for each node is:

	Check nodes update rule	Bit nodes update rule
$T_{n,m}$ from bn_n to cn_m	$E_{n,m}^{(i)} = 2tanh^{-1} \prod_{n' \in N(m) \backslash n} tanh \frac{T_{n',m}^{(i)}}{2}$	$T_{n,m}^{i} = I_n + \sum_{m' \in M(n) \backslash M} E_{n,m'}^{(i-1)}$
T_n from vn_n to cn_m	$E_{n,m}^{(i)} = 2tanh^{-1} \prod_{n' \in N(m) \backslash n} tanh \frac{T_{n'}^{(i)} E_{n',m}^{(i-1)}}{2}$	$T_{n}^{i} = I_n + \sum_{m' \in M(n)} E_{n,m}^{(i-1)}$

Table 3.1: Updating

Sum Product Algorithm

The previous algorithm considered products and hyperbolic tangents that implemented in hardware can be some time consuming tasks, in order to prevent this the *likelihood ratio* can be substituted with its logarithm, the *log-likelihood ratio* and the operations to be done are simpler.
The new notation is:

- k is the current iteration;
- Q_{ji} is the message from BN_j to CN_i;
- R_{ij} is the message from CN_i to BN_j;
- $C[j]$ is the set of incoming messages for BN_j;
- $R[i]$ is the set of incoming messages for CN_i.

The first step initializes BN with the specific γ_i, then CN receives the information and at each iteration the message from CN is:

$$R_{ij}^{(k)} = \psi^{-1}\Big[\sum_{j' \in R[i]\setminus j} \psi(Q_{j'i}^{(k)})\Big] \cdot \delta_{ij'} \tag{3.15}$$

where

$$\delta_{ij'} = (-1)^{|R[j]|} \cdot \prod_{j' \in R[i]\setminus j} sign(Q_{j'i}) \tag{3.16}$$

while ψ is a non linear function with $\psi(x) = \psi(x)^{-1}$ for $x >= 0$

$$\psi = -ln(tanh\Big|\frac{x}{2}\Big|) = ln\frac{1+e^{-|x|}}{1-e^{-|x|}} \tag{3.17}$$

The equation 3.15 can be rewritten in

$$R_{ij}^{(k)} = \psi[PS_i - \psi(Q_{ji}^{(k)})] \cdot \delta_{ij'} \tag{3.18}$$

Where PS_i is the sum of all incoming message to c_i with ψ applied.

$$Q_{ji}^{(k)} = \lambda_j + \sum_{i' \in C(j)\setminus i} \tag{3.19}$$

3.5.3 Hard Decoding algorithms

Hard decoding algorithms consider the hard information of the bit stream: $'0'$ or $'1'$. The following algorithms presented here are suitable when each single bit does not include an additional information related to the error added by the channel. The simplest hard algorithm for LDPC decoding is the Bit Flipping. The syndrome is evaluated and the number of unsatisfied check nodes connected to a variable nodes is counted and compared to a predefined threshold, the algorithms ends when the syndrome is 0 or a maximum number

of iterations is reached.

The algorithm is also the worst in terms of performance, since the threshold remains always the same and it's not able to correct a huge number of errors. The main benefit is the complexity since only some basic operations are required.

The improvements of the Bit Flipping introduces an higher computational complexity, but their performance are highly improved.

Gallager's algorithm

The hard decoding technique developed by Gallager[9] is a type of Bit Flipping. Two versions has been studied, for regular codes the threshold δ:

- **Gallager A**: $\delta \leq \gamma_v - 1$.
- **Gallager B**: $\gamma_v/2 \leq \delta \leq \gamma_v - 1$

The B-version works better, while both has the same behavior if the degree node is 3. The main issue of the BF solution is its determinism since the threshold remains constant during the iterations. The main improvement of this solution can be achieved by including a way to change δ in case no additional error is corrected, otherwise the decoder will iterate each time the same approximated codeword without being able to correct any new error. The computation stops when a maximum number of iterations is reached or the syndrome is 0, in the second case the message is decoded.

Data: $\mathbf{y}$: *message encoded*
Initialization: $\forall v \in V : \ \hat{x}_v^{(0)} \leftarrow y_v$
$\mathbf{s}^{(0)} \leftarrow \mathbf{x}^{(0)} H^T$
$l = 0$;
while $\mathbf{s}^{(0)} \neq \mathbf{0}$ *and* $l \leq L$ **do**
  $\forall v \in V : \ \text{Compute } \psi_{us}^l(v)$
  for $v = 1$ *to* N **do**
    if $\psi_{us}^l(v) > \frac{\gamma_v}{2}$ **then**
      $\hat{x}_v^{(l+1)} \leftarrow 1 \oplus \hat{x}_v^{(l)}$
    else
      $\hat{x}_v^{(l+1)} \leftarrow \hat{x}_v^{(l)}$
    end
  end
  $\mathbf{s}^{(l+1)} \leftarrow \mathbf{x}^{(l+1)} H^T$
  $l = l + 1$
end
Result: $\hat{\mathbf{x}}_v^l$: *message decoded*

Algorithm 2: Parallel Bit Flipping over BSC

The decoder was tested for many cases, but the result are not enough to consider it as a possible candidate to be implemented in hardware, the structure is simple but the residual error is not acceptable.

Gradient Descent Bit Flipping for BSC

The algorithm is an improvement of the Bit Flipping, the threshold changes as the algorithm goes on. The version presented is an adaptation of BSC error of the original GDBF[15].
The Gradient Descent Bit Flipping tries to solve the problem in a different way: the search of the correct codeword can be seen as the process to minimize a function (since we want to find the codeword that have a zero-vector as syndrome), this is called the objective-function. It is derived considering two different decoding processes:

- *Maximum likelihood* decoding process: the goal of this step is to find an approximation with the highest correlation with the original encoded message, the ciphertext.

- *Sum of syndromes of the approximated codeword*: this counts the number of unsatisfied check nodes connected to each variable node (as in the Bit Flipping).

The GDBF for BSC works as follows:

- Evaluate the syndrome of the current approximation of the codeword;

- Evaluate for each variable node its inverse function in case of γ-variable-regular codes

$$\Lambda_v^{(l)}(\hat{\mathbf{x}}_v^{(l)}, \mathbf{y}) = \hat{\mathbf{x}}_v^{(l)} \oplus + \sum_{c \in N_v} \bigoplus_{u \in N_c} \hat{\mathbf{x}}_u^{(l)} \tag{3.20}$$

  or more in general

$$\Lambda_v^{(l)}(\hat{\mathbf{x}}_v^{(l)}, \mathbf{y}) = \hat{\mathbf{x}}_v^{(l)} \oplus + \sum_{c \in N_v} \bigoplus_{u \in N_c} \hat{\mathbf{x}}_u^{(l)} - 0.5 \cdot d_v \tag{3.21}$$

The quantity is the inverse function it is an integer vector with the same length as ciphertext, it has to be evaluated at each new iteration. Then the maximum is evaluated and the corresponding variable node with that value are flipped.

The algorithm works better than bit flipping since at each iteration a new threshold is evaluated, but with small cycles a problem arises: the approximation of the original message changes only between two incorrect values and it is not able to escape from this condition and successfully correct other errors or jump into another bit configuration. The only solution is to construct matrices without small cycles.

Data: $\mathbf{y}$: *message encoded*
Initialization: $\forall v \in V : \hat{x}_v^{(0)} \leftarrow y_v$
$\mathbf{s}^{(0)} \leftarrow \mathbf{x}^{(0)} H^T$
$l = 0$;
while $\mathbf{s}^{(0)} \neq \mathbf{0}$ *and* $l \leq L$ **do**
  $\forall v \in V :$ Compute $\Lambda_v^{(l)}(\hat{\mathbf{x}}, \mathbf{y})$
  $b^{(l)} \leftarrow max_v(\Lambda_v^l(\hat{\mathbf{x}}, \mathbf{y}))$
  for $v = 1$ *to* N **do**
    if $\Lambda_v^{(l)}(\hat{\mathbf{x}}, \mathbf{y}))b^{(l)} = b^{(l)}$ **then**
      $\hat{x}_v^{(l+1)} \leftarrow 1 \oplus \hat{x}_v^{(l)}$
    else
      $\hat{x}_v^{(l+1)} \leftarrow \hat{x}_v^{(l)}$
    end
  end
  $\mathbf{s}^{(l+1)} \leftarrow \mathbf{x}^{(l+1)} H^T$
  $l = l + 1$
end
Result: $\hat{\mathbf{x}}_v^l$: *message decoded*

Algorithm 3: Gradient Descent Bit Flipping over BSC

The decoding behavior of GDBF for a (400,200) code with different values of BSC probability are in Figure 3.13, the requirement to have the errors completely cleared is achieved only considering a number of iterations of 100 or 200, at least. Even if the residual error in the message is lower, there are still some cases that does not allow complete correction. The code considered is constructed by PEG.

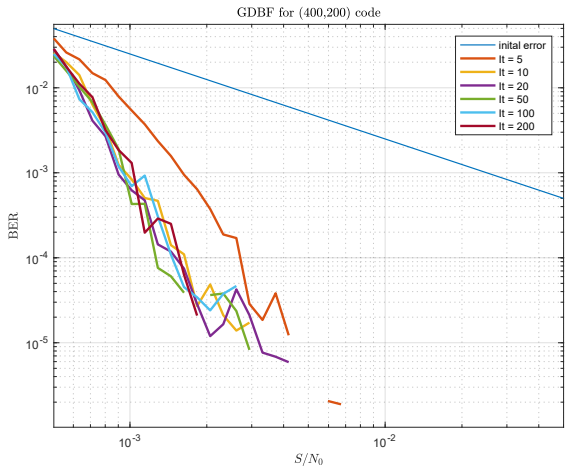


Figure 3.13: GDBF

Probabilistic Gradient Descent Bit Flipping

The PGDBF is an improvements of the GDBF[15]: in the first case all the bits that satisfy the flipping condition are changed, here they are changed with a certain probability p_{PGDBF}. The difference is that :

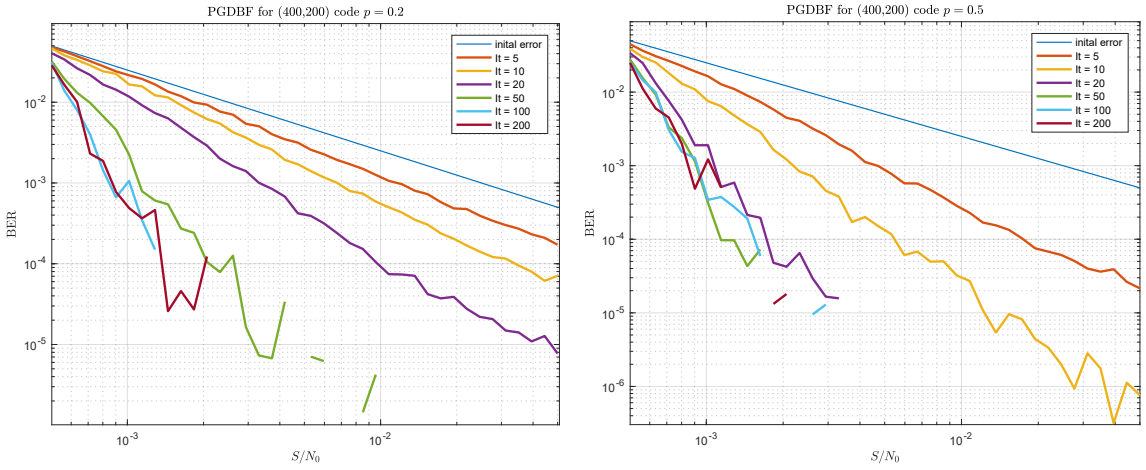
- a Bernoulli random variable a_v is generated with $Pr(a_v = 1) = p$
- the new approximation is evaluated as

$$\hat{\mathbf{x}}_v^l = a_v \oplus \hat{\mathbf{x}}_v^l \tag{3.22}$$

  for the values that have the maximum value od inverse function

The algorithm works better than the previous one, since all the error are corrected up to a $p_{err,max} = 10^{-2}$ for a code (151,61), for $p_{PGDBF} = 0.8$[15].
The complexity is higher with respect to the previous case, since it is required the evaluation of a quantity considering a Bernoulli distribution.
The main advantage is that increasing the number of iterations and the dimensions of the code $p_{err,max}$ can be slightly increased, but it has again the same problems of the GDBF, because the errors not corrected by the PGDBF are the ones that the GDBF is able to correct, sizing successfully the p_{PGDBF} can overcome this problem.

The decoder is simulated for the same conditions of the GDBF for two values of p. The behavior is in Figures 3.14 that consider two probabilities, it is suggested to increase it in order to exploit at most the algorithm.



(a) PGDBF with $p = 0.2$

(b) PGDBF with $p = 0.5$

Figure 3.14: PGDBF

Multiple Decoding attempts and Random re-Initialization

The PGDBF behavior shows that the erroneous corrections depends on the first flipping conditions evaluated at the beginning of the process, a new idea is to restore the original

condition when there is a specified number of iteration and the syndrome is not **0**.

The algorithm is widely described in [15]. The performance is studied below compared to other decoders.

Probabilistic Parallel Bit Flipping

The Probabilistic Parallel Bit Flipping is a technique suitable for regular codes.
The last adopted technique is based again on a different phylosophy. The variable to evaluate for each bit of the approximated codeword at each iteration is the energy:

$$E_n^{(k)} = v_n^{(k)} \oplus y_n \ + \sum_{c_m \in N_{v_n}} c_m^k \tag{3.23}$$

where $c_m^{(k)}$ is:

$$c_m^{(k)} = \oplus_{v_n \in N(c_m)} v_n^k \tag{3.24}$$

The quantity is the same evaluated for the previous algorithms, instead of considering the maximum value there is a predefined flipping probability for each of the possible E_n. The values that can be assumed are $0 \leq E_n^k \leq d_v + 1$, for each of them there is a vector $\mathbf{p} = (p_0, p_1, \ldots, p_{d_v+1})$. The vector $R_n^{(k)}$ is the flipping condition evaluated for each bit. The vector $\mathbf{p}$ is evaluated with an exaustive simulation.

Data: $\mathbf{y}$: *message encoded*
Initialization: $\forall v \in V : \ \hat{x}_v^{(0)} \leftarrow y_v$
$\mathbf{s}^{(0)} \leftarrow \mathbf{x}^{(0)} H^T$
$\mathbf{p} = (p_0, p_1, \ldots, p_{d_v+1})$
$l = 0$;
while $l \leq L$ **do**
  for $1 \leq m \leq M$ **do**
    Compute $c_m^{(k)}$
  end
  if $c^{(k)} = 0$ **then**
    exit while
  end
  for $1 \leq n \leq N$ **do**
    Compute $E_n^{(k)} = v_n^{(k)} \oplus y_n \ + \ \sum_{c_m \in N_{v_n}} c_m^k$
    Generate $R_n^{(k)}$ from $B(p(E_n^{(k)}))$
    if $R_n^{(k)} = 1$ **then**
      $v_n^{(l+1)} = v_n^{(l)} \oplus 1$
    end
  end
  $l = l + 1$
end
Result: $\hat{\mathbf{x}}_v^l$: *message decoded*

Algorithm 4: PPBF

The algorithm can be adapted to irregular codes in two different ways:

- modify $E_n^{(k)}$ to adapt it to different degrees, this one is more similar to the PGDBF, but it avoid the evaluation of the maximum;
- use different $\mathbf{p}$ depending on the degree of the node, this is for sure the most complex option since in case od high degree of the nodes many possibilities are present.

The first option is adopted. The most relevant part of the algorithms is the selection of the probabilities, since many combinations are possible.
The simulations considered the case of (200,60) codes, with variable node degree 5. The resulting vector $\mathbf{p}$ is $\mathbf{p} = [0\ 0.001\ 0.003\ 0.005\ 0.3\ 0.7\ 1]$, which worked similarly for a (200,100) code with the same variable node degree.
The same probability vector is kept and tested for larger codes, in all the cases it worked similarly obtaining promising results. The only drawback is the huge number of iterations required, but it is not required the computation of the maximum.
The increasing number of iterations increases the error correction capabilities of the algorithm, the same does not happen for the other algorithms.
The algorithm has been tested comparing different code sizes, while keeping the same variable node degree (5 in out case). The probability vector remains the same and the results are in Figures 3.15.

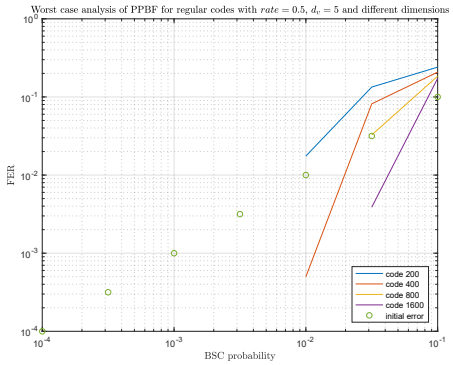


Figure 3.15: PPBF with codes with different dimension

Q-Decoder

The Q_Decoder is a special type of decoder adapted to the case of QC-LDPC codes that includes a $\mathbf{Q}$ matrix in order to add density to the original key. The decoder is developed in order to exploit the structure of $\mathbf{Q}$ and its effect on the error vector since it is able to 'move' the errors on some bits. This last point does not create any security issue since the matrix $\mathbf{Q}$ is only known to the decoder.

The main peculiarity of the algorithm is that increasing the number of iterations the error correcting capabilities are not improved. The $It_{max} = 4$ produces outstanding results since only in some few cases the decoder is not able to converge. The only drawback of the proposed solution is that the results are obtained adopting too large codes, which means that the resources employed are huge. The requirement of 'large codes' is important since

the codes tested in this work are smaller and the results obtained are not the same at all since even with a small number of errors per each message the decoder has some residual error.
The Q_Decoder has been tested even with PEG generated matrices, since the QC-LDPC codes seems to behave approximately in the same way as the other, but something has to be noted since the results are less promising in that case.

3.6 Code parameters to achieve high p_{BSC} with complete error correction

The code has some variables that can be modified in order to be able to increase p_{BSC} without decreasing the decoding performances. The increase in the error added to the message can improve the security of codes or the number of iterations can be reduced to obtain a faster decoder.

The matrices employed in the study are constructed with PEG, the study on the cryptographic use of this codes is wide as is it the one of QC-LDPC codes, then they are selected for this purpose.

The parameters under attention are:

- **Matrix dimensions**: the values of n and r of the **H** matrix, keeping constant the other parameters may affect the decoding phase;
- **Code Rate**: the second value that could be changed is r, since this means that more equations to verify are present in **H** and there are more connections among the various bits of the encoded message;
- **Density of the matrix**: the parameter could be important since it is the measure of the connections of the nodes, but this parameter has an upper bound since codes are low density;
- **Regular/Irregular codes**: this aspect has been already explored in literature and Irregular appeared to be the best ones;
- **Variable nodes degree**: the maximum value of the variable nodes degree could affect the decoding performance.

The decoder considered in order to simulate the error correction capability are the GDBF and the PGDBF with $I_{max} = 100$, the reason of this choice is that, as stated by the group who developed the algorithms, the probabilistic one has better performances but fails to correct patterns that the GDBF is able to solve, then it's meaningful to study codes with both methods.

The total number of errors considered varies during the simulations, initially 1000 error are corrected but in order to have a more precise study the number has been increased up to 2000. The same happened for the number of matrices studied for each configuration,

they are 20 for most of the case, increased only if some results seemed to be too optimistic and in these case it is specified. The iterations considered are 100, unless a more precise analysis has to be done or another algorithm is take into account.

The plots presented in the following figures has p_{BSC} on the x-axis and on the y-axis is present the *maximum error/message length*, the initial error introduced is plotted too in order to have a comparison between the quantities. The Figures present a "worst case analysis" of the decoders.

3.6.1 Matrix dimensions

The first study takes into account the following code length:

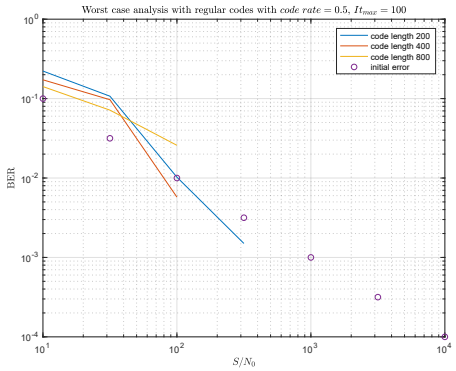
- 200
- 400
- 800

with a code rate varying from: 0.5, 0.6, 0.7. The study considers just the dimensions of the matrix, but in order to understand its impact on the error correcting capabilities it is important to test it with different arrangements.

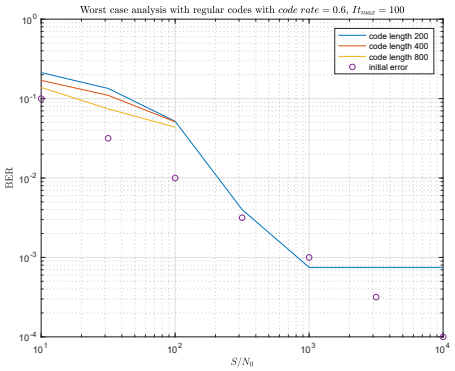
The dimensions of the matrices appears to be relevant in the error correcting capabilities, since as shown in Figures 3.16 a larger code is able to correct more errors, despite the code rate. The reason of this may be from two causes:

- The construction of larger codes creates less cycles and thus it is less probable to be in a trapping set. The construction algorithm considered, the PEG, links the dimensions of the matrix to the minimum size of the cycle and larger codes has larger minimum cycles.
- The density is approximately the same among the matrices, then the degree of the nodes is higher as the matrix becomes larger, it is possible that the presence of more connections for a single node has some benefits. This option can be denied comparing the performances of regular codes with different degree, another proof could be an analysis related to the density of the codes: denser (but still with a low density of 1's) codes could behave better.

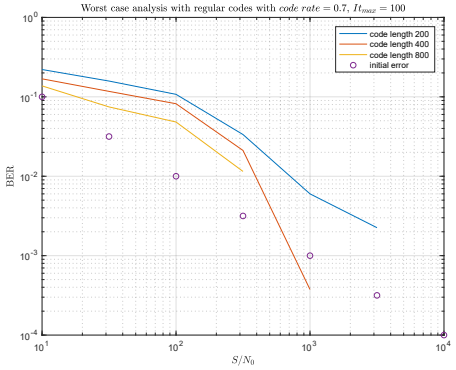
The codes with *code length* = 200 fails to correct errors even if there is only one error added, which is the case of $p_{BSC} = 10^{-4} - 10^{-3}$ and this is for sure linked to the presence of cycles, because just one erroneous bit can cause the decoder to be trapped into a cycle.
The decoding process has the same trend both for regular and irregular codes: larger codes has better error correcting capabilities since in some cases the errors are completely corrected.
The decoder considered is the GDBF, the results with the PGDBF decoder has approximately the same trend.
The generation of larger matrices allows to have an higher p_{BSC}, from the analysis seems that also codes with length 400 are enough. This point has to be clarified taking into



(a) Code rate = 0.5

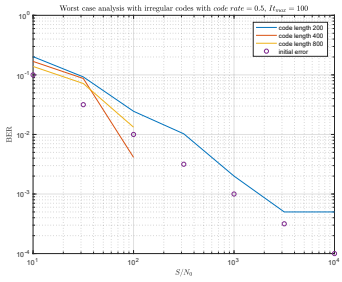


(b) Code rate = 0.6

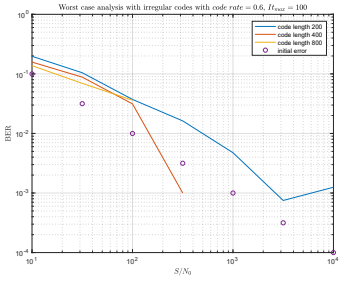


(c) Code rate = 0.7

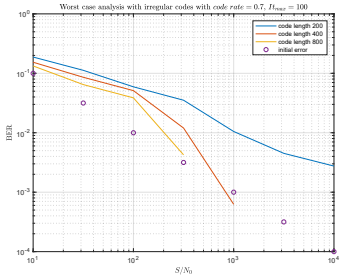
Figure 3.16: Regular codes error correction with different dimensions, with GDBF decoder



(a) Code rate = 0.5



(b) Code rate = 0.6



(c) Code rate = 0.7

Figure 3.17: Irregular codes error correction with different dimensions, with GDBF decoder.

account an higher number of matrices with approximately the same characteristics.
The trend is the same both with regular and irregular matrices, then irregular codes with length 400 and 800 are compared, the rate considered is 0.7. Considering 40 matrices the result obtained is in Figure 3.18

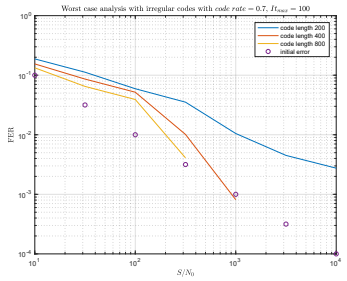


Figure 3.18: Irregular codes correction with GDBF doubling the number of matrices considered.

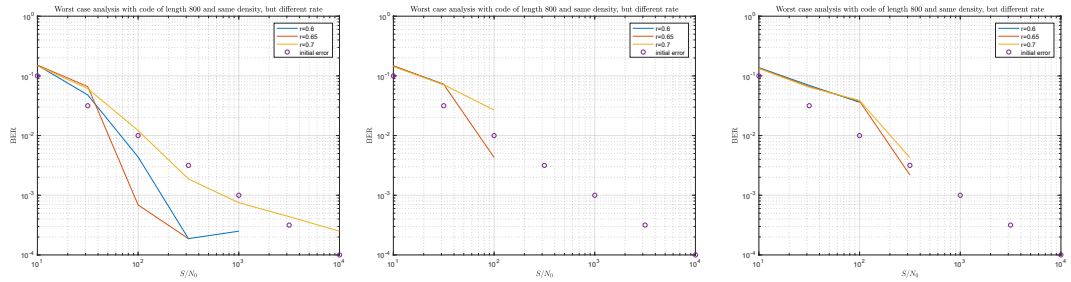
Conclusion

The result including more matrices does not change, this means that the assumptions made before are correct. The following simulations and analysis will be done considering codes with length 800, since working with larger matrices has a positive impact.

3.6.2 Code Rate

The rate is defined as k/n, as this value becomes higher the redundancy added decreases. It is clear that as the redundancy is higher the error correcting capabilities of the code increases, but having code rates approximately of 0.5 means that half of the ciphertext is dedicated to the redundancy and it is not the message that has to be sent, obviously the space dedicated to the redundancy has to be limited, then solutions with higher rate are explored.
The simulations presented in Figures 3.16 show clearly that lower code rates produces a lower residual error in the corrected message. The QC-LDPC codes works usually with the rates mentioned above, rates between 0.6 and 0.7 are going to be explored. The test will consider irregular codes with length 800 and different densities of the *Parity-Check Matrix*. The densities considered for the matrices are: 1%, 3%, 7% of 1's.
The different conditions simulated clearly show the dependence on the rate of the output of the decoder, this is clear since having more redundant bit increases the possibilities of the decoder and thus it is possible to have more connections between variable nodes and check nodes in the bipartite graph. This behavior is constant even if the density has changed and even if there are different dimensions of the matrices (considering the same density of 1's).

The best option seems to select a code with rate 0.7, in order to have less space occupied by the matrices, and further tune the correcting capabilities with the following parameter that is going to be explored: the density of $\mathbf{H}$.



(a) Density = 1 (b) Density = 3 (c) Density = 7

Figure 3.19: Irregular codes error correction capability in different code rates/density conditions, with GDBF decoder.

Conclusion

The code rate is important for the space occupied by the matrix and the message to be sent, but since the most important thing is to have the message to be corrected this aspect will be for sure preferred, then is no meaningful result will be obtained with the other parameters the choice will be to select.

3.6.3 Density of the Matrix

The density of the *Parity-Check Matrix* is directly linked to the degree of the nodes. Different cases are explored, in the analysis the code rate is kept equal to 0.7. The considerations done previously on the density are verified in this step, in addition an higher number of matrices with the same density are generated and then simulated. The result is in Figure 3.20.

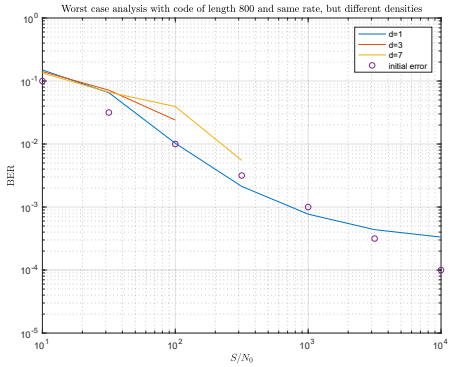


Figure 3.20: Irregular code error capabilities for different densities

The different densities has been generated with a different degree of the nodes, for each density the *variable node degree* and *check node degree* has been evaluated in this way:

- *Density* = 1:

  - `degree_variable_node= randi([2 7]),1,code_length)`
  - `degree_check_node= randi([7 24]),1,redundancy)`

- $Density = 3$:
  - `degree_variable_node= randi([4 14]),1,code_length)`
  - `degree_check_node= randi([14 48]),1,redundancy)`

- $Density = 7$:
  - `degree_variable_node= randi([8 28]),1,code_length)`
  - `degree_check_node= randi([28 96]),1,redundancy)`

The result obtained shows that it is possible to have a complete correction of the errors in a range between $3 * 10^{-3}$ and $*10^{-2}$, which means that a number of errors among 8 and 2. The following simulations will consider the errors in that range to find the approximately exact point. In Figure 3.21. The additional consideration that can be done on the error is

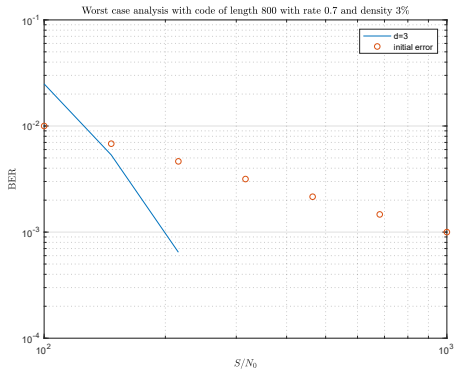


Figure 3.21: Code with density 3% with errors corrected in a more detailed range.

linked to the construction algorithm adopted for the codes and the lower bound of the girth given by its authors. The parameters of the code can be used in 3.11 and 3.12 in order to know the lowest possible girth. In the last case considered, for the different densities the lower bound of the girth is in Table 3.2.

Table 3.2: PEG parameters for different cases

	$Density = 1\%$	$Density = 2\%$	$Density = 7\%$
t	0.88	0.57	0.38
g_{min}	4	4	4

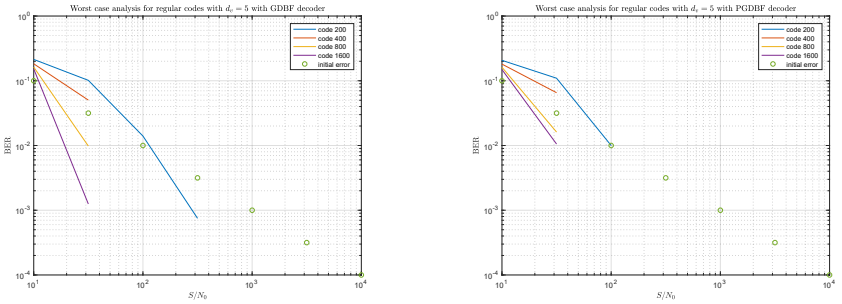
The result clearly shows that the most relevant parameter in the evaluation of the expected minimum girth are the maximum degree of the variable and check node: lower values of d_c^{max} and d_s^{max} with large values of k allows to have larger cycle in the graph. But this point is in contradiction with the simulations that has been done in the last

step since is seems clear from Figure 3.20, that codes with a lower density behave worse than the ones with a medium density. This point has to be analyzed more in detail, since the article in the PEG[10] consider the generation of the degree of the nodes in a different ways and even lower numbers are included, in this description the initial degree is higher. Moreover it could be possible that considering the degree of nodes in a small range, i.e. considering `degree_variable_node= randi([2 28]),1,code_length)` instead of `degree_variable_node= randi([8 28]),1,code_length)`, could be the reason of this mismatch. If this is true the relevant parameter is not the density, but the degree of the nodes, which is not actually linked to the density but directly affect the performances.

3.6.4 Variable nodes degree

The last analysis that can be done take into account matrices with different dimension but the same degree of the node (or the same distribution for irregular codes), in order to verify the impact of this parameters.
The code length considered are: 200,400,800,1600.

Regular codes with $d_v = 5$,11,21



(a) GDBF decoder (b) PGDBF decoder

Figure 3.22: Regular codes with $d_v = 5$.

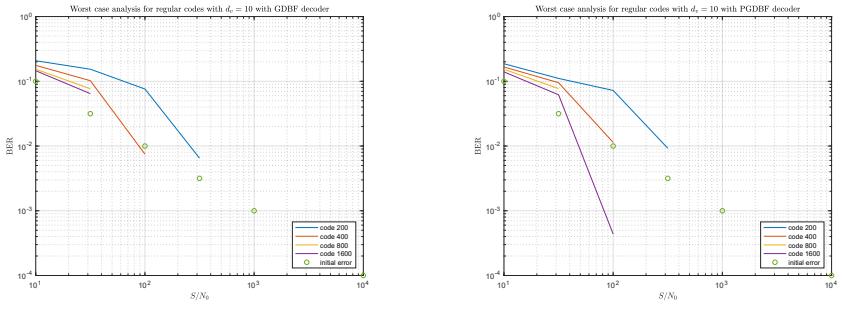
The degree is doubled in this case, the trend is the same.

Irregular codes with $d_v^{max} = 5$,11,21

The result seems strange at first sight, since in the previous sections the irregular codes seems to behave better then their regular version, but in this case the vector containing the degrees of each node has numbers from 2 to 5, each of them selected with the same probability. The same simulations has been done also in other arrangements, i.e. with different rate, with the degree of the nodes that cover a wider range, from 2 to higher values, and even in that case the results were not promising. This behavior can be linked to the not promising behavior of codes with low density and small length since in those cases nodes with low degree where included. Further simulations will be done.
The following simulations consider a code with length 400, but different distributions of the vector containing the variable node degree. The only possibility that cannot be tested is the one that has only even numbers as variable nodes degree, since in that case the *Generator Matrix* cannot be derived. The result consider degrees of variable nodes of

- for $d_v^{max} = 5 : 2, 3, 4, 5$.
- for $d_v^{max} = 7 : 4, 5, 6, 7$.
- for $d_v^{max} = 10 : 5, 6, 7, 8, 9, 10$.

The vector with the degree of variable nodes is generated selecting each value with a specific probability. The different cases taken into account are:



(a) GDBF decoder

(b) PGDBF decoder

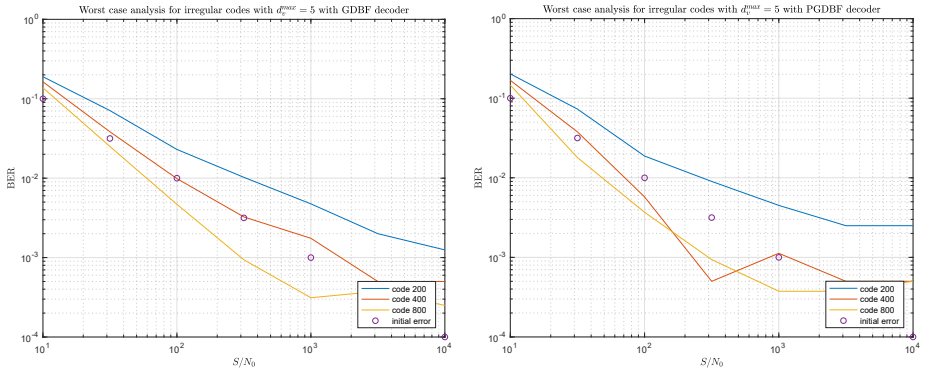
Figure 3.23: Regular codes with $d_v = 11$.

- *less even degree*: it is more probable to have odd numbers as degree, 2 and 4 has an higher probability to be selected;
- *less higher degree*: it is more probable to have small numbers as degree, 2 and 3 has an higher probability to be selected.
- *less odd degree*: it is more probable to have even numbers as degree, 3 and 5 has an higher probability to be selected.
- *more higher degree*: it is more probable to have large numbers as degree, 4 and 5 has an higher probability to be selected.
- *odd degree*: only odd numbers as degree, 3 and 5 are the only ones selected.

The result shows that including an odd number of degree nodes has some benefits in the error correcting capabilities since the curves with lowest residual error are from the *odd degree* case. The worst choice that can be done is the one that includes mainly small and even values for the degree. This aspect can be linked to the results obtained previously on the density, since matrices with quite low density appeared to behave better the the ones with a really small number of ones.
The comparison of the results obtained here and the section dedicated to the density gives some limits to the selection of the variable nodes degree.

Conclusions

The simulations done up to now shows that considering a low degree for the variable node could improve the performances even if regular code are adopted, in an important work proposed by a group that studied in detail LDPC codes for cryptographic applications the following result has bean already found, since the realized that BF based decoders convergence is improved if the elements of the codeword are included in a small number of equations[11].This is true for regular codes, but appeared to fail in case there are regular codes with low density, which is linked to a low degree of the nodes.



(a) GDBF decoder (b) PGDBF decoder

Figure 3.24: Irregular codes with $d_v = 11$.

3.6.5 General conclusions

The result with different structure of the code has not a huge impact on the correction capabilities, since the p_{BSC} can not be increased too much. Moreover there is an important problem related to the cryptographic use of these codes: nobody has studied their security increasing their dimensions, larges codes requires more computations to be broken, but this is true also for the generation of the key pair because of the computation of the inverse. The choice to realize a Decoder for PEG was abandoned mainly for this reason.

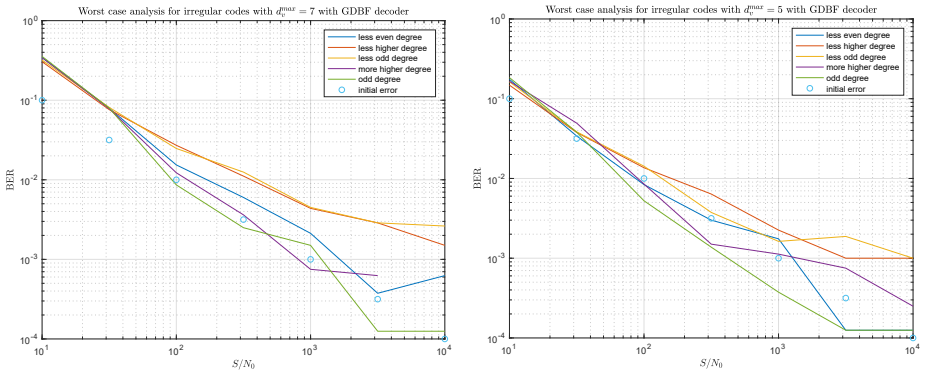
The other aspect considered is the memory required to store them since the small codes taken into account requires kB of memory to store the matrix and their random structure does not allow simplification in the decoder.
PEG codes has advantages because there is not a structure in them, but the limitations that they introduce are huge and their employment in the cryptographic field is not suggested.

The best choice are the QC-LDPC codes widely studied in order to make them secure enough for cryptographic standards, moreover the memory required to store them is less the PEG ones. There is an huge work dedicated to the cryptographic use of these cyclic codes, the following work is entirely dedicated to it.

3.7 Architecture of Decoder and type of codes

The choice of a good decoder is important to consider other aspects, since the architecture has to be implemented in VHDL and then the huge dimensions of the code has to be taken into account in order to adopt the best choices. The best decoder in this sense is the *Bit Flipping* algorithm with a *QC-LDPC* code for two reasons:

- *QC-LDPC* codes required less memory to be stored (even if they are huge) and their structure allow further simplification in the computation;



(a) $d_v^{max} = 7$ (b) $d_v^{max} = 5$

Figure 3.25

- The security of these codes has been studied in detail;
- the *Bit-Flipping* with a LuT table that allows to reduce the iterations to 4 or 5, which is better then the 100 required for the $GDBF$ or the BF.

The decoder presented previously behave similarly in the worst case since there is no guarantee that a message, even with few errors, is always successfully corrected. Then a high complexity in the decoder does not improve the correcting capabilities of the decoder, then a simpler decoder is adopted and the decoding performances are improved selecting properly the threshold, for example.
The simulations of the soft decoding algorithms done in previous works showed the same issue, they require a huge number of iterations to converge and the complexity is even higher compared to the GDBF and PGDBF or the others.
The work done on the *Bit-Flipping* decoder for $QC - LDPC$ that has a variable threshold is huge and allows to reduce the time to compute the clear message because there are less computations to be performed and it converges in less iterations, there are still some cases in which the message is not corrected but this is a problem of all LDPC codes since their decoding radius is unknown.

Chapter 4

LEDA Algorithm

The LEDA algorithm is a method developed by an Italian research group that studied the a way to generate the keys and the security of this kind of codes[14].
The LEDA has two versions:

- LEDApkc: developed on the basis of the McEliece cryptosystem with QC-LDPC codes;
- LEDAkem: developed on the Niederreiter cryptosystem with QC-LDPC code.

The choice is obviously the LEDApkc that states for ***L****ow-d****E****nsity parity-check co****D****e-b****A****sed* ***P****ublic* ***K****ey* ***C****ryptosystem.*

LEDA overcomes the limits of the LDPC codes and a faster *Bit-Flipping* decoder is studied. The other big advantage is that less memory is required to store the matrices, this thanks to the cyclic structure of the keys.

4.1 Key generation

4.1.1 Public Key

The generation of the private and public key has as a starting point the generation of the private key, called **L**. In LEDApkc the key is the result of the multiplication among two matrices:

- **H**: the secret key;
- **Q**: the transformation matrix.

The structure of the code is *QC*, *Quasi-Cyclic*, the secret matrix contains n_0 square matrices, of size p, that are circulant and the first row (or column) is sufficient to describe the entire matrix. In the LEDA case the code is:

$$\mathbf{H} = [\mathbf{H}_0, \mathbf{H}_1, \ldots, \mathbf{H}_{n_0-1}] \tag{4.1}$$

The $\mathbf{H_i}$ matrix has a cyclic structure:

$$\mathbf{H}_i = \begin{bmatrix} h_0 & h_1 & \dots & h_{n_0-1} \\ h_{n_0-1} & h_0 & \dots & h_{n_0-2} \\ \vdots & \vdots & \ddots & \vdots \\ h_1 & h_2 & \dots & h_0 \end{bmatrix} \tag{4.2}$$

The set positions in the first row $\mathbf{h}_i = [h_0 h_1 \dots h_{n_0-1}]$ are stored.

The transformation matrix has a slightly different structure, but it is still QC. The block contained are $n_0 \times n_0$, but each sub-matrix is $p \times p$.

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{0,0} & \mathbf{Q}_{0,1} & \dots & \mathbf{Q}_{0,n_0-1} \\ \mathbf{Q}_{1,0} & \mathbf{Q}_{1,1} & \dots & \mathbf{Q}_{1,n_0-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{Q}_{n_0-1,0} & \mathbf{Q}_{n_0-1,1} & \dots & \mathbf{Q}_{n_0-1,n_0-1} \end{bmatrix} \tag{4.3}$$

The product between the two is still a circular matrix:

$$\mathbf{L} = \mathbf{HQ} = [\mathbf{L}_0, \mathbf{L}_1, \dots, \mathbf{L}_{n_0-1}] \tag{4.4}$$

The codes involved in this method are regular, then the weight, d_v, of each sub-block in $\mathbf{H}_i$ is the same. The weights of sub-block in the transformation matrix are defined by another QC matrix:

$$w(\mathbf{Q}) = \begin{bmatrix} m_0 & m_1 & \dots & m_{n_0-1} \\ m_{n_0-1} & m_0 & \dots & m_{n_0-2} \\ \vdots & \vdots & \ddots & \vdots \\ m_1 & m_2 & \dots & m_0 \end{bmatrix} \tag{4.5}$$

The sum of all m_i is $m = \sum_{i=0}^{n_0-1} m_i$, which is again constant among all the rows (or columns).
The second key is generated starting from the secret key, in particular it is required to have an invertible sub-block. This condition is reached when two conditions hold:

- d_v and m must be odd;
- $\mathbf{Q}$ has to be full rank.

The second point introduces some limitations in the choice of the weights: it can be shown that $\mathbf{Q}$ is invertible if the permanent of $w(\mathbf{Q})$ is odd and lower than p.
The permanent of a matrix is the same as the determinant, but the summation is with plus sign.
The matrix $\mathbf{Q}$ is required in order generate a denser key, in this way the resulting matrix $\mathbf{L}$ is denser than $\mathbf{H}$. The additional sparsity is required in order to make the LDPC codes more secure, their limits are due to sparsity of the matrices.

The public key can be further compressed introducing a Deterministic Random Bit Generator (DRBG), the starting bit of this generation comes from another module that generates a random number.

4.1.2 Public key

The steps to generate the public code were already explained in the above sections, the algorithm is the same here but less computations are required in order to have the secret matrix in the correct form, the construction generated an invertible sub-block.

The public key is generated starting from the inverse of $\mathbf{L}_{n_0-1}$. The inverse of a matrix requires a huge amount of time if it is evaluated in the canonical way, but thanks to the structure of QC matrices the operations can be strongly reduced.

The generation starts SK:

$$\mathbf{L} = \mathbf{LQ} = [\mathbf{L_0}, \mathbf{L_1}, ..., \mathbf{L}_{n0-1}] \tag{4.6}$$

The *Generator* matrix is:

$$\mathbf{M} = \mathbf{L}_{n_0-1}^{-1}\mathbf{L} = [\mathbf{M}_l, \mathbf{I}_p] \tag{4.7}$$

The block $\mathbf{M_l}$ is the result of a matrix multiplication.
The inverse has to be performed just on one block, not all the square matrices.

Multiplicative inverse

The first row of the matrix is sufficient to define a QC matrix, this row can be written in another form, the polynomial $a(x)$ in GF(2), then the inverse of the matrix is the multiplicative inverse of $a(x)$, there is a theorem to compute this value, it is a method developed by Euclides. The proposed version employed to invert these matrices is in article [16]. In algorithm 5 the pseudo-code of the VLSI implementation.

The starting polynomial are $\mathbf{a}(\mathbf{x})$, that describes the first row of the cyclic matrix, and $\mathbf{g}(\mathbf{x})$, an irreducible polynomial of degree p.
The value of $\mathbf{g}(\mathbf{x})$ for the block sizes employed is $\mathbf{g}(\mathbf{x}) = x^{p-1} + 1$.

Data: $g(x)$,$a(x)$
$s(x) := g(x);\ v(x) := 0$
$r(x) := a(x);\ u(x) := 1$
$\delta = 0;$
for *i=1 **to** 2m **do*** **do**
  if $r_m = 0$ **then**
    $r(x) := x \times r(x);$
    $u(x) := x \times g(x);$
    $\delta := \delta + 1$
  else
    if $s_m = 1$ **then**
      $s(x) := s(x) - r(x);$
      $v(x) := v(x) - u(x);$
    end
    $s(x) := x \times s(x);$
    if $\delta = 0$ **then**
      $r(x) := s(x);\ s(x) := r(x)$
      $u(x) := x \times v(x);\ v(x) := u(x)$
      $\delta = 1$
    else
      $u(x) := u(x)/x;$
      $\delta = \delta - 1$
    end
  end
end
Result: $u(x) = a(x)^{-1}$

Algorithm 5: Euclidean Inversion for hardware implementation [16]

4.2 Encryption

The encryption is done in three steps:

- the message $\mathbf{u}$ is multiplied by $\mathbf{M}^T$, the output is $\mathbf{m} = [\mathbf{u}, \mathbf{r}]$ where $\mathbf{r}$ is the redundancy;
- the error vector $\mathbf{e}$ is generated.
- the ciphertext is $\mathbf{x} = \mathbf{m} \oplus \mathbf{e}$

The number of errors to be added is a parameter of the system and depends on the error correcting capability of the code and the decoder. The value can not be predicted in advance since the correcting radius of this kind of LDPC codes is unknown, but a huge number of simulations allows to find the maximum number of errors that allows correctable in most of the cases.
The result is $\mathbf{x}$ can be sent to an insecure channel.

4.3 Q-Decoder

The decoding algorithm employed is the Q-Decoder. The algorithm consists in a *Bit Fllipping* with a LuT to reduce the iterations. The decryption executes the following operations in each iteration:

- *Sigma* is : $\sum^{(l)} = s^{(l-1)}\mathbf{H}$
- *Correlation* is : $R^{(l)} = \sum^{(l)} \mathbf{Q} = [\rho_1^{(l)}, \rho_2^{(l)}, \ldots, \rho_n^{(l)}]$
- The Correlation threshold is evaluated based on the sum of Sigma: $b^{(l)}$
- The position in Correlation that have a value higher than the threshold are saved: $P = \{v \in [1, n] | \rho_v^{(l)} = b^{(l)}\}$
- The current error is: $\mathbf{e}_{new}^{(l)} = [e_1^{(l)}, e_2^{(l)}, \ldots, e_n^{(l)}]$ and $e_j = 1 \; for \; j \in P$, updated to $\mathbf{e}^{(l)} = \mathbf{e}^{(l-1)} \oplus \mathbf{e}_{new}^{(l)}$
- Syndrome is updated: $\mathbf{s}^{(l)} = \mathbf{s} + \mathbf{e}^{(l)} * \mathbf{L}$
- The sum of the Syndrome vector is evaluated and the iteration updated, if $\mathbf{s}$ is 0 or $It = It_{max}$ the decoding is stopped.

The LuT with threshold is required in order to avoid the evaluation of the maximum that requires to find its value and then to store the positions that contain it, this results in more operation involved. Moreover the algorithm requires more iterations to reach a syndrome equal to 0. The LuT is a way to obtain a suitable threshold depending on the value of the *Syndrome* at that iteration.

The choices adopted by the authors works fine for a software implementation, but the size of the problem requires some changes to translate the code to hardware.

4.4 Code parameters

The $QC - LDPC$ code involved in the method have some parameters to be defined, their performances and security have been studied in detail for the ones listed in 4.1.
The values presented are:

- Category is related to the dimension of the code;
- n_0 is the number of circulant block;
- p is the dimension of each block;
- d_v and m the weight of the rows in $\mathbf{H}$ and $\mathbf{Q}$, respectively;
- t number of errors;
- DFR is the decoding failure of the code for that error, the simulations involved 10^8 decoding attempts.

Table 4.1: Code parameters[14]

Category	n_0	p	d_v	m	t	DFR
1	2	27779	17	[4,3]	224	$\approx 8.3 \cdot 10^{-9}$
	3	18701	19	[3,2,2]	141	$\lesssim 10^{-9}$
	4	17027	21	[4,1,1,1]	112	$\lesssim 10^{-9}$
2	2	57557	17	[6,5]	349	$\lesssim 10^{-8}$
	3	41507	19	[3,4,4]	220	$\lesssim 10^{-8}$
	4	35027	21	[4,3,3,3]	175	$\lesssim 10^{-8}$
3	2	99053	19	[7,6]	474	$\lesssim 10^{-8}$
	3	72019	19	[7,4,4]	301	$\lesssim 10^{-8}$
	4	60509	23	[4,3,3,3]	239	$\lesssim 10^{-8}$

4.5 Security

The security of LEDApck and of QC-LDPC codes more in general has been studied widely by the research group that developed the system, in the article [14] all the details are present.

The important aspect that arise in the work is that the regularity of the codes provides a speed-up in the key or clear message retrieving, the improvement introduced is at most p. Apparently this seems not a good news, but considering the exponential complexity of the attacks having a polynomial reduction is not much relevant. The security of QC-LDPC codes is not lower that the PEG codes. The security issues are still related to their sparsity and not their cyclic nature.

The attacks developed recently on this type of system are presented in the following.

- **Squaring Attack**
  The attack is paired with an Information Set Decoding method, the article [17] proposes a method that simplifies the ISD if applied before it. The system is intended to introduce a huge improvement, but it works just for matrices size of the power of 2.
  LEDA overcome the problems since, as can be seen in Table 4.1, the size of the circulant blocks is even and a prime number. The choice of p prime number in order to further simplify the inverse evaluation, otherwise for an odd number in general the inverse evaluation takes more time.

- **Instruction Set Decoding**
  The ISD attack is based on the search of low weight codewords, the most promising one that works better than the others is Stern's algorithm with a quantum part proposed by Grover's that introduces a speed-up in the computation. The results for McEliece cryptosystem based on Goppa codes are in Table 2.1. The LDPC codes, due to their sparse nature, are even more easier to break with this method, but LEDApck huge dimensions overcome the problem.

- **Attack on the Dual Code**
  The attack is based on the search of low wight codeword in the Public matrix, since PK is derived from SK. In this case the task can be simplified by the sparse nature of the codes.

The Table 4.1 has to be extended including the security level of the code. The complete results are in Table 4.2. The security estimate is given in log_2 number of operations on Quantum machine (QM) and Classical machine (CM) both for ISD and Dual Code Attack (DC). The work factor (WF) is in the "n-bit" format.

Table 4.2: Code parameters [14]

Category	$\mathbf{n_0}$	**p**	$\mathbf{WF}_{cm}^{DualCode}$	$\mathbf{WF}_{qm}^{DualCode}$	$\mathbf{WF}_{cm}^{ISD}$	$\mathbf{WF}_{qm}^{ISD}$
1	2	27779	223.66	134.84	217.45	135.43
	3	18701	219.84	133.06	216.42	135.63
	4	17027	230.61	139.29	216.86	136.11
2	2	57557	358.16	204.84	341.53	200.47
	3	41507	351.57	200.95	341.61	200.44
	4	35027	351.96	201.40	343.36	200.41
3	2	99053	478.67	267.00	467.24	265.38
	3	72019	484.48	270.18	471.67	265.70
	4	60509	480.73	268.03	473.38	265.48

The last class of attacks is the **Reaction Attack**. It is based on a a study of the failure of the decoder, from this study additional informations can be obtained on the structure of the code. This limits in time the use of a key-pair.

The property of indistinguishability is reached if a conversion is applied on the codes, this is described in article [18], it is called KI-γ conversion. The applied obfuscation allows the system to reach semantic security (IND-CCA2) if employed in the given key-pair lifetime or if the decoder has no failures.

Chapter 5

LEDA Architecture

The algorithm of the Q-Decoder has been implemented in hardware, but some modifications are introduced to make the evaluation faster, with less modules and to avoid too heavy computations.

The parameters considered in the design are listed in Table 5.1. The values are not included in Table 4.1, but the architecture can be easily adapted to codes with different p, while codes with different n_0 require more modification, but they are still feasible. The considerations that will be done can be applied to any code of this kind and length with only some slightly changes.

The second change is that $\mathbf{L}$ is stored in memory and the computation of $\mathbf{HQ}$ is not implemented in order to reduce the logic to be added and unnecessary computations. The same happens for $\mathbf{L^T}$, it is required for the evaluation, but since the memory required is small it is not computed in the decoder, but already present.

Table 5.1: Architecture parameters

Category	$\mathbf{n_0}$	**p**	$\mathbf{d_v}$	**m**	**t**	It_{max}
1	2	15013	9	[5,6]	143	4

5.1 Memory organization

The length of the message is huge, it occupies at least $2kB$ of memory if stored bit-by.bit. It is supposed to be divided in two memories, one for each block of length $\mathbf{p}$ of the ciphertext, the manipulation of the data is easier in this way. The same happens for the partial result, they are all stored in block, a part from the Syndrome that requires just one block.
The matrices are saved by position, the different blocks are stored in different memories. There are:

- Message : Msg_1 and Msg_2 both of length 15013bit;

- Public key : Ltr_1 and Ltr_2 both of length 81 bits and width 16 bit;

- Syndrome : Syn of length 15013 bits.

- Correlation : UPC_1 and UPC_2 of length 15013 bits and width 8 bit;

- Error Position : Pos_1 and Pos_2 of variable length and with width 16bit.

The variables of the algorithm can be seen as vectors in software, but when considering an hardware implementation it is not feasible to have a memory that in output has 15013 bit since they are too much and in the datapath this amount of FlipFlops is required in order to be able to temporary store a value for the computation.

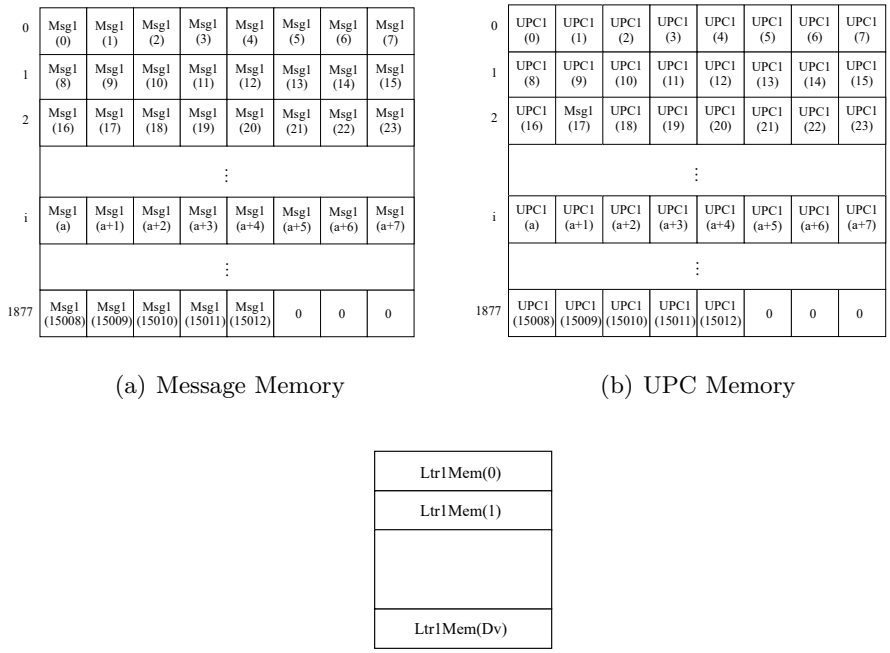
The memories, instead of long vectors, are seen ad matrices of different width. In particular:

- *MsgMem1,2* and *Syn* are N_b bit wide and 1877 high;

- *UpcMem1,2* are $N_b * 8bit$ wide and 1877 high;

The memories containing positions have a different structure. Less rows are required to store them, 81 at most for this particular case. The other categories present in Table 4.1 require different sizes that are present in the table, in particular the number of rows to store the matrix is given by $d_v * m$, that can be a huge number in case the circulant block taken into account are 3 or 4.

Figures 5.1 represents how data are stored in memories for $N_b = 8$. UPC Memory and Msg Memory have the same format, but each location has different size. UPC requires 8 bit for one value, while Msg requires just 1 bit for a single value. The Syndrome has the same format as Msg Memory.



(a) Message Memory

(b) UPC Memory

(c) Ltr Memory

Figure 5.1: Memories

The message and the syndrome are saved as bit-by-bit vector, but those vectors can be stored by position. The second option requires less memory for sparse vectors, the assumption is valid only for the syndrome when the decoder is in the last iteration.

5.2 Q-Decoder simplification

The algorithm requires some little changes in order to reduce the overall computations.

- The *Correlation* can be directly obtained by $s^{l-1}\mathbf{L}$, this is done in order to avoid the storage of a partial result;
- The sum of the rows in $\mathbf{Q}$ is not required to update the error, it is sufficient to take the flipped bit in the message;
- The new syndrome can be both obtained updating its current value with $\mathbf{eL}$ of evaluating it by $\mathbf{x} * \mathbf{L}$.

The choice to consider just $\mathbf{L}$ as secret key requires a bit more memory to store it, but this is saved since Σ is not stored.

The second change requires to choose between two different approaches in the update of the error vector.

- The error vector can be updated with a xor with its previous value and the message is updated only at the end of the decoding process, this required to work by position in the update of the syndrome.
- The error vector is applied to the message at each iteration and the syndrome is evaluated again at each iteration.

The first solution was explored, but not implemented because it requires an additional unit and can not be easily parallelized, but working by positions in some steps of the evaluation lowers the cycles required.

5.2.1 Vector by Matrix Product

The decoder requires an optimization of the product of a vector by a matrix, both in GF(2) and in integer domain, because it is required in two steps of the algorithm.
The units developed for this purpose are:

- Vector By Circulant: to optimize the product of a dense vector by a circulant matrix,
- Sparse Vector By Circulant: to optimize the product of a sparse vector by a circulant matrix,

Vector By Circulant

The unit that is the most important since is a big change in the way the multiplications between vector and matrix are performed.
The starting point to explain it are a vector $\mathbf{m} = [m_0\ m_1\ \dots m_{n-1}]$ of length n and a circulant matrix $\mathbf{B}$ of size $n \times n$ with the first column $\mathbf{b}_0 = [b_0\ b_1 \dots b_{n-1}]$ that defines the whole matrix.
The product is:

$$\begin{bmatrix} m_0 & m_1 & \dots & m_{n-1} \end{bmatrix} \begin{bmatrix} b_0 & b_{n-1} & \dots & b_1 \\ b_1 & b_0 & \dots & b_2 \\ \vdots & \ddots & \vdots & \\ b_{n-1} & b_{n-2} & \dots & b_0 \end{bmatrix} \tag{5.1}$$

The result of the product in $\mathbf{GF(2)}$ is the *xor* operation among the following vectors:

$$\begin{matrix} m_0 \otimes b_0 & m_1 \otimes b_1 & \dots & m_{n-1} \otimes b_{n-1} \\ m_0 \otimes b_1 & m_1 \otimes b_0 & \dots & m_{n-1} \otimes b_2 \\ \dots & \dots & \dots & \dots \\ m_0 \otimes b_{n-1} & m_1 \otimes b_{n-2} & \dots & m_{n-1} \otimes b_0 \end{matrix} \tag{5.2}$$

Since the matrices involved are highly sparse the asserted positions are less than n which means that some elements b_i in 5.2 are missing, in our case the remaining elements are just a few compared to the initial one.

The example below shows the remaining elements if a vector by a circular and sparse matrix are multiplied.
The original message is $\mathbf{m} = [m_0\ m_1\ m_2\ m_3\ m_4\ m_5\ m_6\ m_7\ m_8]$, the first column of the matrix $\mathbf{B}$ is $\mathbf{b_0} = [b_0\ b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8]$.

The result of the product is in general:

$$\begin{array}{ccccccccc}
m_0 \otimes b_0 & m_1 \otimes b_1 & m_2 \otimes b_2 & m_3 \otimes b_3 & m_4 \otimes b_4 & m_5 \otimes b_5 & m_6 \otimes b_6 & m_7 \otimes b_7 & m_8 \otimes b_8 \\
m_0 \otimes b_8 & m_1 \otimes b_0 & m_2 \otimes b_1 & m_3 \otimes b_2 & m_4 \otimes b_3 & m_5 \otimes b_4 & m_6 \otimes b_5 & m_7 \otimes b_6 & m_8 \otimes b_7 \\
m_0 \otimes b_7 & m_1 \otimes b_8 & m_2 \otimes b_0 & m_3 \otimes b_1 & m_4 \otimes b_2 & m_5 \otimes b_2 & m_6 \otimes b_4 & m_7 \otimes b_5 & m_8 \otimes b_6 \\
m_0 \otimes b_6 & m_1 \otimes b_7 & m_2 \otimes b_8 & m_3 \otimes b_0 & m_4 \otimes b_1 & m_5 \otimes b_2 & m_6 \otimes b_3 & m_7 \otimes b_4 & m_8 \otimes b_5 \\
m_0 \otimes b_5 & m_1 \otimes b_6 & m_2 \otimes b_7 & m_3 \otimes b_8 & m_4 \otimes b_0 & m_5 \otimes b_1 & m_6 \otimes b_2 & m_7 \otimes b_3 & m_8 \otimes b_4 \\
m_0 \otimes b_4 & m_1 \otimes b_5 & m_2 \otimes b_6 & m_3 \otimes b_7 & m_4 \otimes b_8 & m_5 \otimes b_0 & m_6 \otimes b_1 & m_7 \otimes b_2 & m_8 \otimes b_3 \\
m_0 \otimes b_3 & m_1 \otimes b_4 & m_2 \otimes b_5 & m_3 \otimes b_6 & m_4 \otimes b_7 & m_5 \otimes b_8 & m_6 \otimes b_0 & m_7 \otimes b_1 & m_8 \otimes b_2 \\
m_0 \otimes b_2 & m_1 \otimes b_3 & m_2 \otimes b_4 & m_3 \otimes b_5 & m_4 \otimes b_6 & m_5 \otimes b_7 & m_6 \otimes b_8 & m_7 \otimes b_0 & m_8 \otimes b_1 \\
m_0 \otimes b_1 & m_1 \otimes b_2 & m_2 \otimes b_3 & m_3 \otimes b_4 & m_4 \otimes b_5 & m_5 \otimes b_6 & m_6 \otimes b_7 & m_7 \otimes b_8 & m_8 \otimes b_0
\end{array}$$

The asserted bits of this example are only $[b_2\ b_5]$ then some values of each equation are not useful. The resulting vector $\mathbf{r}^T = [r_0\ r_1\ r_2\ r_3\ r_4\ r_5\ r_6\ r_7\ r_8]$ is then equal to the *xor* operation the rows:

$$\begin{array}{ccccccccc}
0 & 0 & m_2 & 0 & 0 & m_5 & 0 & 0 & 0 \\
0 & 0 & 0 & m_3 & 0 & 0 & m_6 & 0 & 0 \\
0 & 0 & 0 & 0 & m_4 & 0 & 0 & m_7 & 0 \\
0 & 0 & 0 & 0 & 0 & m_5 & 0 & 0 & m_8 \\
m_0 & 0 & 0 & 0 & 0 & 0 & m_6 & 0 & 0 \\
0 & m_1 & 0 & 0 & 0 & 0 & 0 & m_7 & 0 \\
0 & 0 & m_2 & 0 & 0 & 0 & 0 & 0 & m_8 \\
m_0 & 0 & 0 & m_3 & 0 & 0 & 0 & 0 & 0 \\
0 & m_1 & 0 & 0 & m_4 & 0 & 0 & 0 & 0
\end{array} = \begin{bmatrix} m_2 \oplus m_5 \\ m_3 \oplus m_6 \\ m_4 \oplus m_7 \\ m_5 \oplus m_8 \\ m_6 \oplus m_0 \\ m_7 \oplus m_1 \\ m_8 \oplus m_2 \\ m_0 \oplus m_3 \\ m_1 \oplus m_4 \end{bmatrix} \tag{5.3}$$

The analysis of the result shows that each equation is a circular shift of the initial vector, those shifts are given by the values asserted in the matrix.

m : *binary vector*
PosB : *positions asserted in B*
$i = 0$
$\mathbf{r} = \mathbf{0}$
while $i \leq PosBSize$ **do**
  $\mathbf{r} = circularshift(m, PosB_i) \oplus \mathbf{r}$
  $i = i + 1$
end
Result: **r** : *result*

Algorithm 6: VectorbyCirculantBinary

The algorithm described is suitable for the integer case too. The vector and matrix must be binary, but in case the result has to be an vector of integer values the *xor* operation is substituted by an addition.

The integer result is:

$$\begin{bmatrix} 0 & 0 & m_2 & 0 & 0 & m_5 & 0 & 0 & 0 \\ 0 & 0 & 0 & m_3 & 0 & 0 & m_6 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_4 & 0 & 0 & m_7 & 0 \\ 0 & 0 & 0 & 0 & 0 & m_5 & 0 & 0 & m_8 \\ m_0 & 0 & 0 & 0 & 0 & 0 & m_6 & 0 & 0 \\ 0 & m_1 & 0 & 0 & 0 & 0 & 0 & m_7 & 0 \\ 0 & 0 & m_2 & 0 & 0 & 0 & 0 & 0 & m_8 \\ m_0 & 0 & 0 & m_3 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 & m_4 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} m_2 + m_5 \\ m_3 + m_6 \\ m_4 + m_7 \\ m_5 + m_8 \\ m_6 + m_0 \\ m_7 + m_1 \\ m_8 + m_2 \\ m_0 + m_3 \\ m_1 + m_4 \end{bmatrix} \tag{5.4}$$

m : *binary vector*
PosB : *positions asserted in B*
$i = 0$
$\mathbf{r} = \mathbf{0}$
while $i \leq PosBSize$ **do**
  $\mathbf{r} = circularshift(m, PosB_i) + \mathbf{r}$
  $i = i + 1$
end
Result: **r** : *result*

Algorithm 7: VectorbyCirculantInteger

The serial version of *VectorByCirculant* is the Algorithm 8.

m : *binary vector*
PosB : *positions asserted in B*
$i = 0$
$\mathbf{r} = \mathbf{0}$
while $i \leq PosBSize$ **do**
  $j = 0$
  $p = \mathbf{PosB_i}$
  while $j \leq n$ **do**
    $k = mod(p + j, n)$
    $r_j = m_k \oplus r_j$
    $j = j + 1$
  end
  $i = i + 1$
end
Result: **r** : *result*

Algorithm 8: VectorbyCirculant

The effect of the algorithm on the product has been tested in Matlab to check the correctness and the reduction of the computational time. The result is promising because the time required to compute the result is two order of magnitude lower than the classical product and the result is always correct.
The big advantage is due to the parallelization of the operations, because each partial result requires just a shift that in software does not require much time.

The version in Algorithm 8 employees the modulo operation in order to be able to restart from the beginning of the vector when the end is reached. The input vector is $\mathbf{a}^T =$

$[a_0\ a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8]$ and the position is $p = 3$. The inner while in Algorithm 8 starts from $j = 0$, the first value of $\mathbf{a}$ is in position $k = mod(j + 3{,}9) = 3$, then as j is incremented the value of k is incremented, until the sum $j+i$ is higher than 9. This means that the position to match is the first one in the input vector. The inner while stops when the last bit of the result is reached. The partial result is:

$$\mathbf{b}^T = [a_3\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$$
$$\mathbf{b}^T = [a_3\ a_4\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$$
$$\dots$$
$$\mathbf{b}^T = [a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_0\ a_1\ a_2]$$

Sparse Vector By Circulant

The second way to compute the product is to consider the result by positon, but is instead and evaluation by position of the product among a vector and a matrix. It is suitable in case the input vector is sparse. The method is referred as *SparseVectorByCirculant*.

The basic idea of the algorithm is exposed in the example below. The product of the previous case has some important reduction if the vector is sparse. The vector $\mathbf{m} = [m_0\ m_1\ m_2\ m_3\ m_4\ m_5\ m_6\ m_7\ m_8]$ has, for example, only $[m_1\ m_3]$ asserted. The result vector for $\mathbf{m} * \mathbf{B}$ is:

$$\begin{array}{ccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & m_3 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & m_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & m_3 & 0 & 0 & 0 & 0 & 0 \\
0 & m_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
=
\begin{bmatrix}
0 \\ m_3 \oplus 0 \\ 0 \\ 0 \\ 0 \\ 0 \oplus m_1 \\ 0 \\ 0 \oplus m_3 \\ m_1 \oplus 0
\end{bmatrix}
=
\begin{bmatrix}
0 \\ m_3 \\ 0 \\ 0 \\ 0 \\ m_1 \\ 0 \\ m_3 \\ m_1
\end{bmatrix}
\tag{5.5}$$

The resulting positions are: $[r_1\ r_5\ r_7\ r_8]$. The evaluation of this result is basically the difference in modulo n between the asserted position of the vector and of the matrix. It may happen that some repetitions occur while performing the difference.
The algorithms works better then the other in case the vector is highly sparse, otherwise the two are comparable.

The result is a list of position of asserted bits. The problem of the solution is that some repetitions occur resulting in more than one position to be stored, for this reason a check is required, but the cycles required to remove unwanted positions is huge.

The cycles needed to adjust the result can be huge, the method is included in the software version of the decoder and the merge operation is included, the simulations showed that millions of cycles are needed to perform this operation. In general this is not suggested and the problem has to be handled in other ways or it can happen that this does not produce a big impact in the further computations.

PosM : *positions asserted in* m
PosB : *positions asserted in* B
PosR : *positions asserted in* r $i = 0$
$j = 0$
$r = 0$
while $i \leq PosBSize$ **do**
  $CurrentBPos = PosB_i$ **while** $j \leq PosMSize$ **do**
    $CurrentMPos = PosM_j$
    $Difference = CurrentBPos - CurrentMPos$
    if $Difference \leq 0$ **then**
      $CurrentNewPos = n + 1 + Difference$
    else
      $CurrentNewPos = Difference + 1$
    end
    $PosR_r = CurrentNewPos$
    $r = r + 1$
  end
end
Result: PosR : *positions asserted in* r

Algorithm 9: SparseVectorbyCirculant

5.2.2 Comparison between the algorithms

The plot in Figure 5.2 shows that if the vector contains approximately 1000 ones the *SparseVectorByCirculant* with *BinaryMerge* algorithm is more convenient. The second point that has to be taken into account is the memory occupied by the two algorithms, since the position required more bit to be stored.

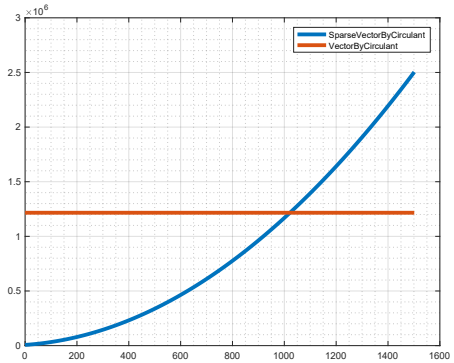


Figure 5.2: Comparison of the two algorithms with respect to the density of the input vector

The modules adopted must take into account this plot in order to be able to select the best algorithm for the computation.

5.3 Vector By Circulant Architecture

The basic unit implements Algorithm 8. The relevant operations is to create a shifted version of the input vector, the Message for example. The approach handled takes into account how data are stored in memory and the meaning of the position vector. Version 2 is an improvement of Version1.

5.3.1 Version 1

The first version translates the algorithm without any improvement.
The flow of the operations is:

- The first position in Ltr memory is read and the position of the first bit to be xored extracted;
- Ltr points the address row of MsgMemory and the specific bit of the line.
- The proper message row is read and at its output a multiplexer is inserted.
- The syndrome is read starting from the first row and at its output another multiplexer is present;
- The two multiplexers starts to count from a value specified by Ltr and from 0 for MsgSel and SynSel respectively;
- MsgSel is incremented until the end is reached, at this point MsgAdx is incremented and a new read performed;
- SynSel is incremented until the end is reached and the new syndrome row is stored, the SynAdx is incremented and a new read from SynMemory is performed;
- The endpoint of MsgMemory requires the MsgSel to be stopped before the end, the same happens when the endpoint of SynMemory is reached.

The flow of the algorithms requires particular attention when the end is reached, in this case the mux selections has to stop before the end and the counter of the message has to be cleared.
At the end of the syndrome evaluation the mux of the result has to end before its maximum value.
The integer version of the component has ad accumulator instead of a xor.
The DataPath requires 5 counters:

- *MsgAdxCounter*: the adx of the MsgMemory;
- *MsgSelCounter*: the selection bit of the Msg mux;
- *SynAdxCounter*: the adx of the SynMemory;
- *SynSelCounter*: the selection bit of the Syn mux;

- *LtrAdxCounter*: the adx of the LtrMemory.

In Figure 5.3 there is the Datapath of the Version1 for the binary case.

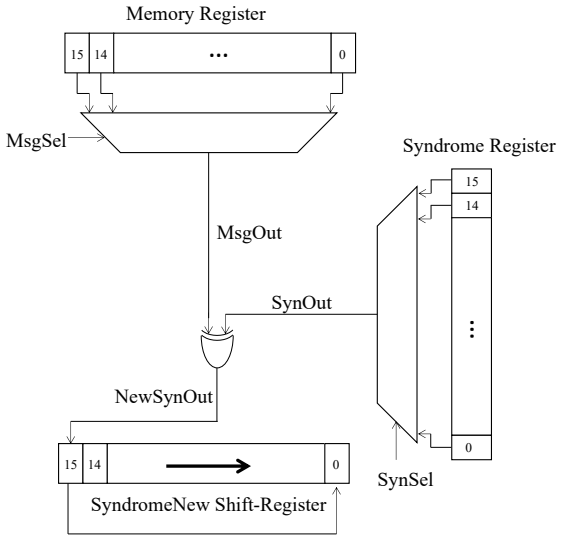


Figure 5.3: VectorByCirculantBinary DataPath

The iterations required by the unit to complete one product is huge, it is:

$$N_{cycles} = p * d_v + 3 * (p/n_b) * d_v + d_v \tag{5.6}$$

The case studied has the parameters of Table 5.1 and $n_b = 8$, then the cycles required to compute just one product is: $1.3 * 10^6$ cycles. The cycles can be reduced increasing the size of a row, but they cannot be lower then $p * d_v$ that is $1.2 * 10^6$. The architecture requires a modification in order to be able at least to parallelize the product.

The best option would be the possibility to have a row of the syndrome produced in a single cycle, the limit to implement this is due to the fact that the row of the message that matches a row of the syndrome is contained in two rows of the message memory.

5.3.2 Version 2

The shifted versions of the initial Message can be generated bit by bit, but the goal is to generate them in groups of more than one bit.

Version 2 aims to generate a row of N_b bits of the shifted version of the message, in order to be able to xor them with a row of the Syndrome, with no need to perform the xor bit by bit.
The Memory alignement between the Message (input) and the Syndrome (result) is represented in Figure 5.4.

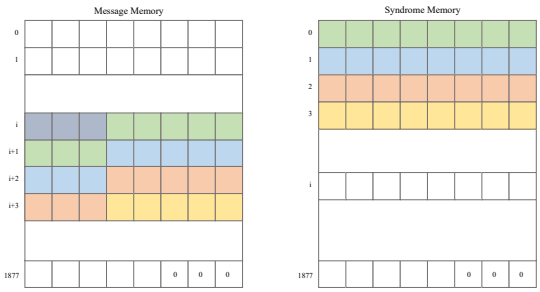


Figure 5.4: Message and Syndrome Memory

The Logic that is able to produce the shifted versions of the message requires two registers and a multiplexer that at its inputs has the possible rows, the selection bit choses the amount of the shift. In Figure 5.5 the basic logic block implemented for $N_b = 4$, this block has in input vectors of N_b length that are the all the possible rows to be xored with the Syndrome, the selection bits selects right one.

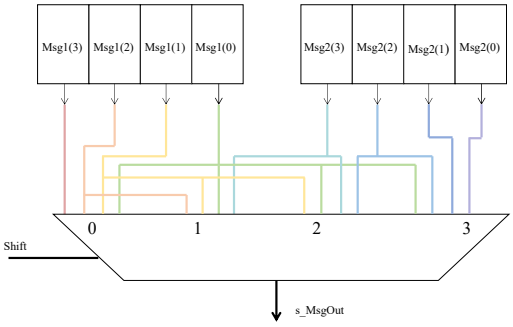


Figure 5.5: Vector By Circulant Multiplexer

The registers *Msg1* and *Msg2* contain consecutive rows from memory. The first rows of the Message is stored in *Msg2*, in the following iteration the value is loaded in *Msg1*, the arrangement at this point is the desired one, the following cycles requires just to repeat the process and store the new row in *Msg2*.

The complete DataPath and ControlUnit of this new solution are more complex than the Version1, but a huge improvement is achieved.

The central operation performed by the unit is to read the result vector from the memory, generate the shifted version of the input vector, compute the xor operation between the two vectors and store in memory the output. The memory location is the same as the result read at the beginning, the updated value is stored here. In some points of the memory there are some small changes to be performed, those modifications produce the complex ControlUnit in Figure 5.11.
The basic steps in the evaluation of the product are:

- **Load_Ltr** : the Ltr Address points the current position to be read;

- **Load_Msg**: the position read gives the initial Address of the Memory, this is read and loaded is register Msg2 and the Counter of the Message Address is updated;
- **Load_Msg_Syn**: the Message Memory is again read, the vector in Msg2 is saved in Msg1 and Msg2 is loaded with the new value read from memory, the Syndrome Message is read and Syndrome Register is loaded;
- **Cmp**: the shifted version of the Message is generated, this is aligned with the value in Syndrome Register, the xor operation between the two vectors is store in the SyndromeNew Register;
- **Store**: the updated Syndrome row is stored in memory, the next operation depends on the current Addresses of the Memories.

The first row of the Syndrome is produced at this point, the following operation requires to cycles between:

- **Load_Msg_Syn**;
- **Cmp**;
- **Store**.

In Figures 5.15 there is the content of the registers during the execution of the algorithm, in green the group of bits that is xored with the value of the Syndrome already present in memory. The example contains two cases.

The behavior changes just in some points in order to be able to have always a group of bits aligned with the result. In the following the additional checks are explained in detail.

The end is reached when the last row of the Syndrome is read, in the first cycle of the execution the value store in memory is the circular-shift of the original message by the position defined in Ltr. The next time the end is reached the xor among two shifted version of the message are store in the Syndrome Memory.

The memory that contains the positions of the matrix can be seen in the same format as for Version1, in Figures 5.7 and 5.6 the meaning of each block, but there is a little change to be introduced: the MSB contains a 0 or a 1 depending of the position of the shift with respect to a quantity called *Difference*, the ControlUnit has two different paths depending on that bit.

The last row of the memory contains an additional row full of zeros, this means that only part of the resulting row is evaluated it is called *PartialStore*, the following row to be read is the first one and its first bit has to be connected with the last one. The partial store is required since in any case the last row contains zeros not belonging to the message, their presence does not change the value already present in memory for those positions, but the result row has to be completed.

The last rows requires different operations, then the flow of the computation is the same explained before, there is just a change in the shift.

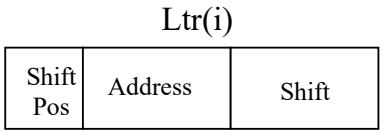


Figure 5.6: Ltr Position

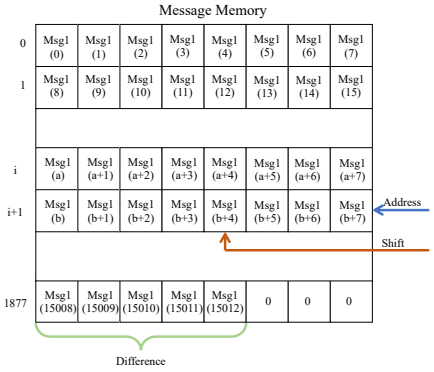


Figure 5.7: Message Memory correspondence with Ltr

The position of the last bit of the last row in the Message Memory is called *difference*. The *shift* of each Ltr iteration can be higher or lower than *Difference*.
The case with *Difference* < *shift* is named as *S0*, the other one *Difference* > *shift* is named as *S1* in the ControlUnit. There is another condition to be mentioned since *Difference* can be equal to *shift*, this case is included in *S0*.
The correct generation of the result requires an additional row, full of zeros, to be added to the Message Memory.

Difference > shift The arrangement of this condition when the last row is read is in Figure 5.8. The output includes some zeros, not belonging to the message, this means that this is just a partial generation of the syndrome, the complete one requires to have in the DataPath the bits from the first row of the Message Memory.

The portion of bits at the output of the multiplexer are in green. The color changes with a different *shift* to be applied to the multiplexer.

The condition has the *shift* lower than the difference, the last value from the multiplexer is stored in Syndrome Memory, the address of the Syndrome is not updated, the next value generated has a different Shift that is sized in order to have zeros in the positions where

the previous read has relevant values of the message .

The new shift, named as *Shift2*:

$$Shift2 = N_{bit} - Difference + Shift \tag{5.7}$$

In this case it is mandatory to have the last row full of zeros.

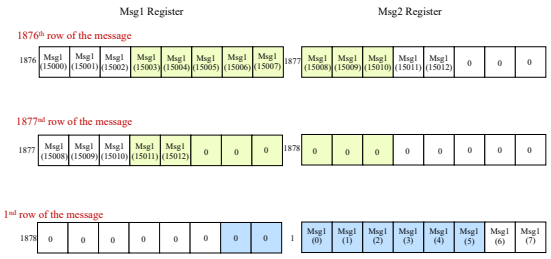


Figure 5.8: n^{th} cycle for Condition 1

Difference < shift The case is in Figure 5.9. The last bits of the message are read one cycle in advance with respect to the previous case, the condition to follow this branch is to have the *ShiftPosition* bit equal to 1 and that the Address of the message is the last useful one, 1877 with this parallelism.

It is required to connect the last bit with the first one: $Msg1(15012)$ with $Msg1(0)$. The last partial row is stored in memory, the address of the Message Memory is updated but without saving any group of bits since they are all zero. The new row to be stored is generated for address 1, the second part of the vector is saved at this point. The last row full of zeros is not required for this condition.

The operation is done correctly if the shift is changed, the new one is called *Shift3*.

$$Shift3 = Shift - Difference \tag{5.8}$$

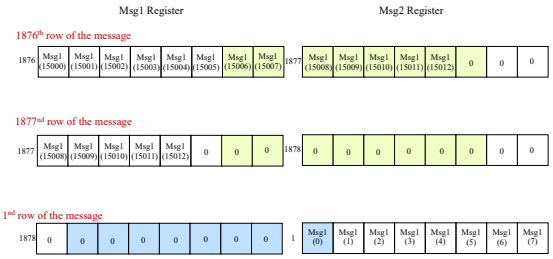


Figure 5.9: n^{th} cycle for Condition 2

Difference = shift The last row is not a partial result, but it is the complete result. The following cycle contains only zeros and it is considered as a partial result, the new Shift is always the maximum possibile since it has to provide in output the first row of the message.

This choice is taken in order to avoid a too complex ControlUnit, since three different paths are possible and an additional bit has to be introduced in order to be able to know the shift. In this case two updates of the Message address has to be done and the Shift is 0. In order to avoid this an useless read and compute is done and the Shift is N_b.

The condition does not require the last row of the Message Memory full of zeros.
Figure 5.10 represents this condition

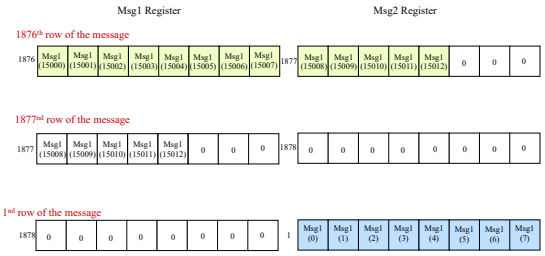


Figure 5.10: n^{th} cycle for Condition 3

The detailed ControlUnit is in Figure 5.11. The operations performed are basically the one listed before, the relevant point is in the choice of the path, selected thanks to the selection bit. The ControlUnit in its integer version is basically the same, the only difference is in the name of the sates.
The DataPath is in Figure 5.12 for the binary version, while the integer version is in Figure 5.13.
The only difference among the two version is the presence of an adder instead of a xor port. The generation of the new Shift is done in the same way since they are computed in the in the DataPath and the selection bit comes from the ControlUnit.

The values useful throughout the computation are stored in registers in order to be available in any moment. The different shifts are not stored in registers, but generated from the output of the initial Shift register.

The input of the *Msg1* is connected to the output of *Msg2*, each time is is enable the value is loaded there.

The multiplexer is able to provide the N_b consecutive bits starting from the one pointed by the Shift. The evolution, cycle by cycle, in in Figures 5.15(a) and 5.15(b), the Shift is kept constant during the execution and it changes just one time.

The integer version of the ControlUnit has the same operations as the Binary one, the only difference is in the same of the states.

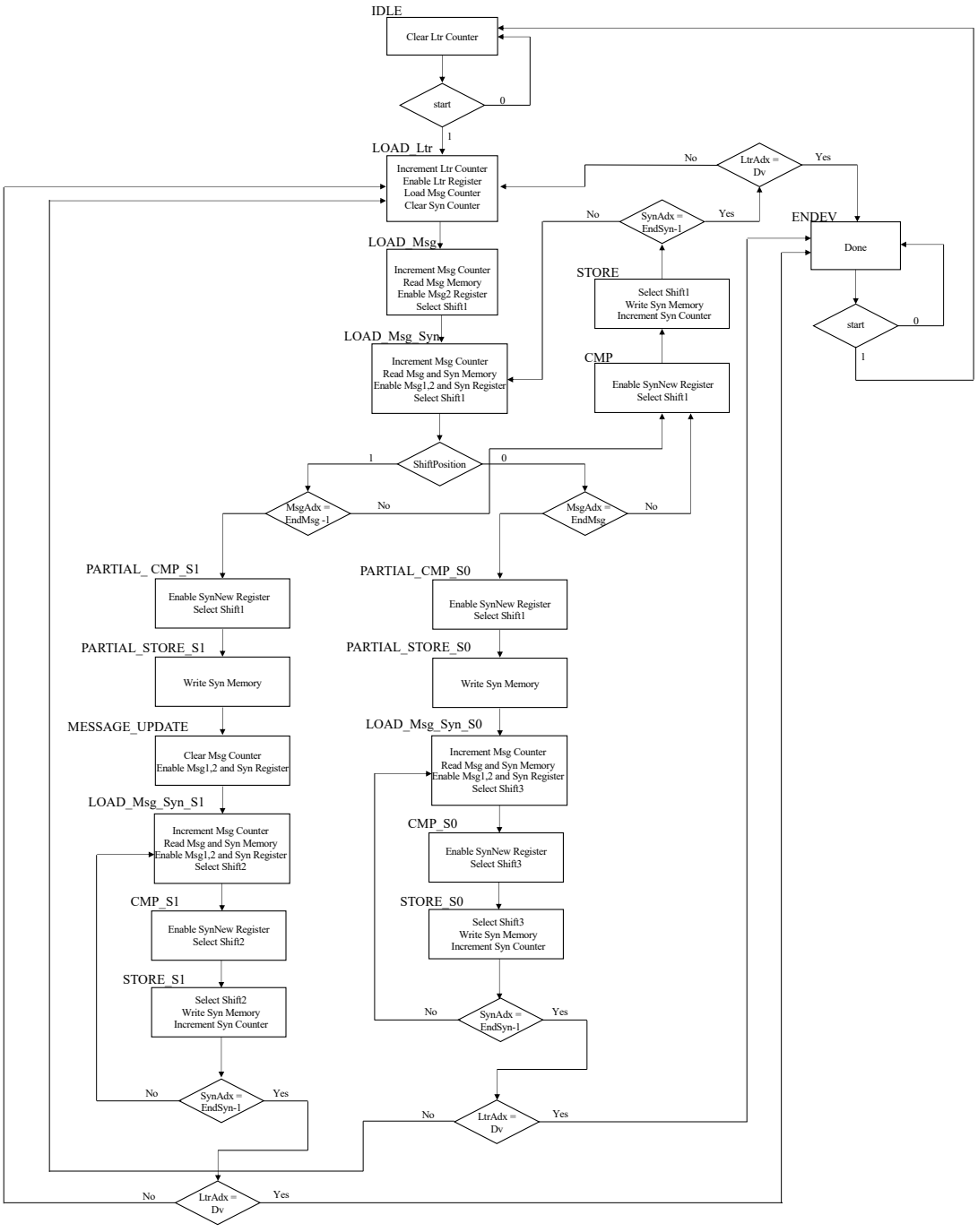


Figure 5.11: Vector By Circulant ControlUnit

The address generation is contained in the DataPath, in Figure 5.14 the Counter of the Message is driven by the value of the position, this is necessary in order t point the correct

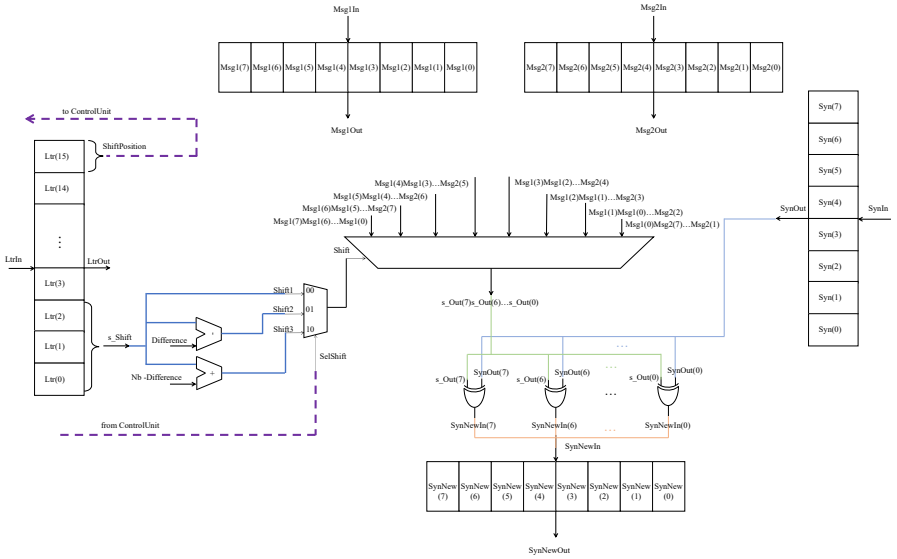


Figure 5.12: Vector By Circulant DataPath

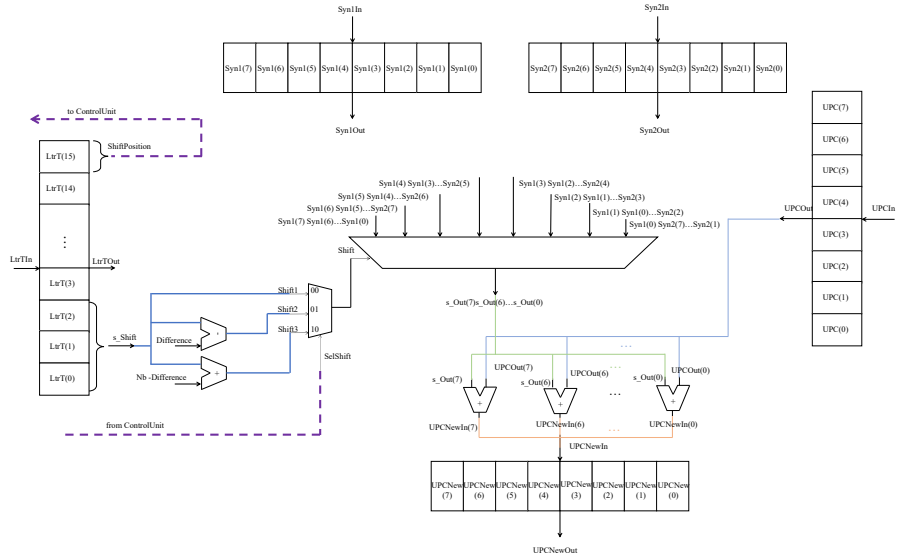


Figure 5.13: Vector By Circulant DataPath in Integer version

memory location.
The integer version of the architecture has LtrT instead of Ltr and the Syndrome Address counter that loaded by the shift present in memory.

The counters included are:

- *Ltr Address Counter*
- *Syn Address Counter*

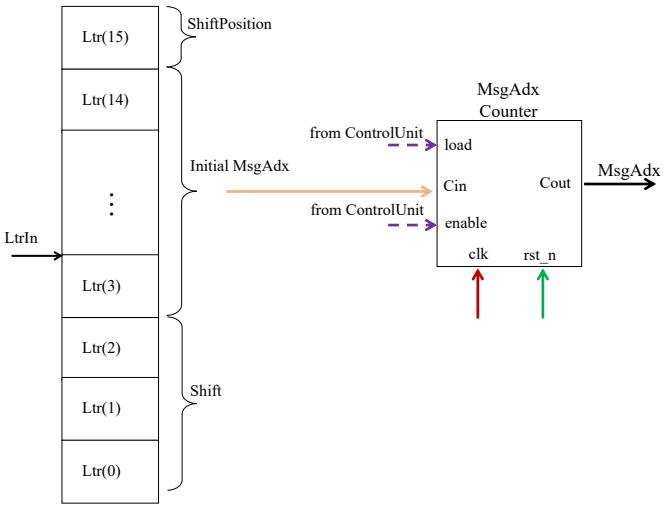
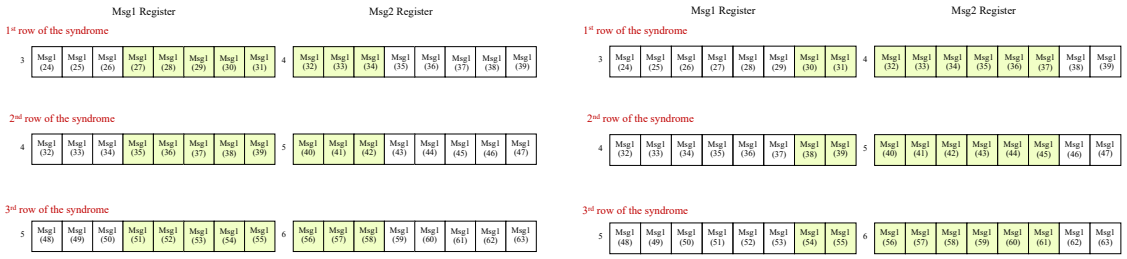


Figure 5.14: Message Address Counter



(a) Condition 1

(b) Condition 2

Figure 5.15: Initial cycles

These counters are driven in a more simpler way.
The ControlUnit provides the *enable*, *clk* and *rst_n* to the Counters and Registers present in the architecture. The only counter loaded by a value is the *Message Address Counter*.

The Version 2 ControlUnit and DataPath can be further modified and reduced in complexity if some conditions are satisfied. The length of the registers is in general N_{bit}, the number is supposes to be a power of 2, this can lead a simplification. The modulo N_{bit} can be obtained just performing the operation $Shift = ShiftMem - Difference$ and taking the last $log_2(N_{bit})$ bits. The implications of this solution has not been studied in detail, but there is a change to be performed in the ControlUnit since the condition with $Shift = Difference$ requires another path, different from the previous two since the three conditions mentioned above require different processes of data, moreover it is required to include in the Position a bit that states if the Shift is equal to the Difference because the path is different.

The second change is in the DataPath, there is just a multiplexer with two entries instead of three. The positions in Ltr Memory has still to be saved in the format mentioned above with a small change, because there is a different behavior in the two cases.

The only problem of Version is in the last iteration. The last row of the Syndrome is read and xored with the last portion of Message in output of the multiplexer, but in Syndrome after *Difference* there is no zero filling but instead some bits that has to be considered as noise and not take into account. This can be a problem is Syndrome has to be considered as input vector of the *VectorByCirculant* unit because in that case it is mandatory to have bits equal to 1. This problem arises in the following modules where the method is employed. The condition is in Figure 5.16 and the memory content at the end is in Figure 5.17.

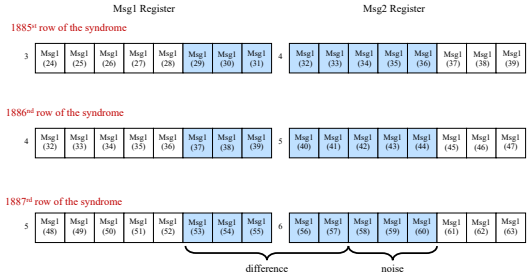


Figure 5.16: Last row generation

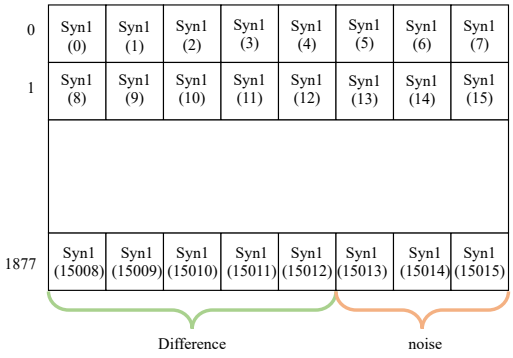


Figure 5.17: Syndrome Memory

Version 1 and Version 2 require the same amount of memory to store the values, but for sure Version2 works better since it is enough to change some parameters to highly increase its parallelism.

5.4 Architecture Modules

The important units involved are:

- *Syndrome Evaluation unit*: the Syndrome is evaluated by Algorithm 6 working on the two blocks of the matrix;
- *Syndrome Weight Evaluation*: the weight is the sum of each bit in the syndrome;
- *Correlation Evaluation unit*: the Correlation is evaluated by Algorithm 7;
- *Threshold Evaluation*: the syndrome weight points a specific threshold;
- *Error Evaluation*: the threshold is compare to each value of the Correlation and if higher or equal it is stored as error bit;
- *Message Update unit*: the bits of the message pointed by the error vector are flipped;
- *Syndrome Update unit*: the syndrome bit to be changed are evaluated with the error vector and the key, then they are flipped.

The Decoder included all the units in the DataPath, they are connected to the memories for the input result and provides the values to be stored and the addresses. The connections among units involves only three parameters. The DataPath is in Figure 5.18. The signals not present are the start for each module that is provided by the ControlUnit, while the done is used for the execution.

The ControlUnit is in Figure 5.19. The flow of the decoder follows the steps in the algorithm.

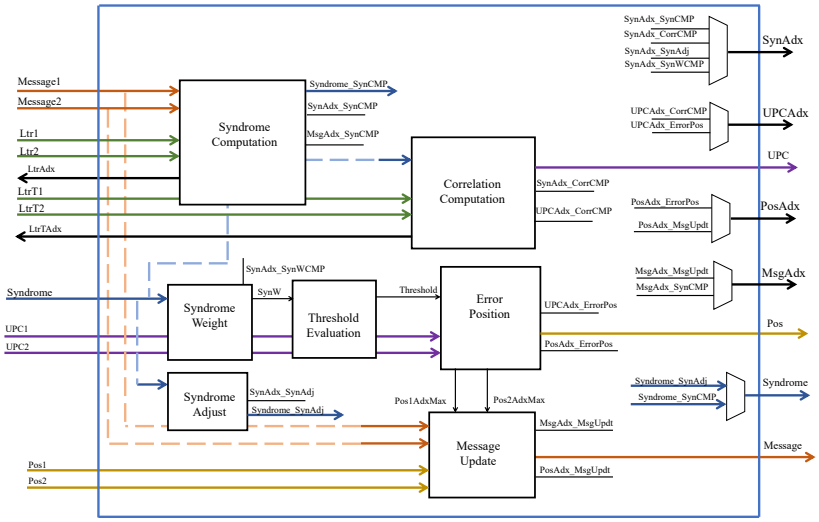


Figure 5.18: Decoder DataPath

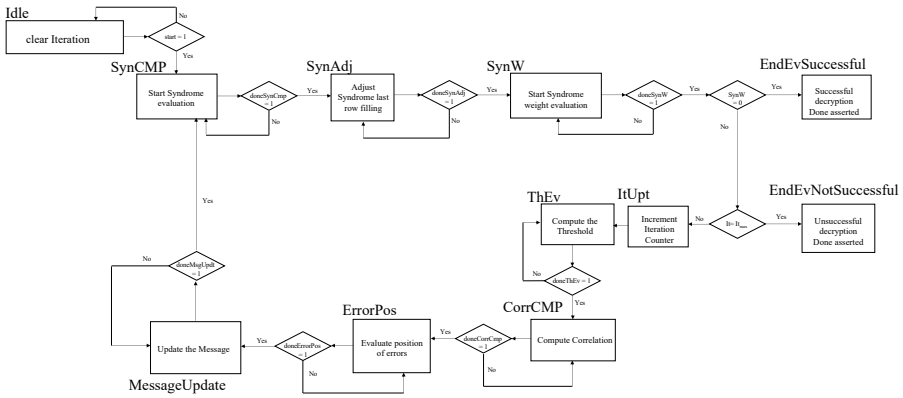


Figure 5.19: Decoder ControlUnit

5.5 Syndrome and Correlation Computation

The units presented here evaluated the Syndrome and Correlation requires the product of vector by square circulant matrix, the block employed in the *VectorByCirculant* component in both the two versions.

The inner block is modified depending on the variable to be computed, because different requirement has to be met.

5.5.1 Syndrome Evaluation

The Syndrome is:

$$MsgEncoded * \mathbf{L}^T \tag{5.9}$$

The codes adopted contains n_0 ($n_0 = 2$) circulant blocks with size $p \times p$ ($p = 15013$), then $\mathbf{L}$ is $p \times n_0 * p$ and $\mathbf{m}$ is $1 \times n_0 * p$, the message can be divided in two equal parts m_0 and m_1 of size $1 \times p$ each.

$$\begin{bmatrix}\mathbf{m}_0 & \mathbf{m}_1\end{bmatrix} \begin{bmatrix}\mathbf{L}_0 \\ \mathbf{L}_1\end{bmatrix} = \begin{bmatrix}\mathbf{m}_0 * \mathbf{L}_0 \\ \oplus \\ \mathbf{m}_1 * \mathbf{L}_1\end{bmatrix} = \begin{bmatrix}\mathbf{s_0} \\ \oplus \\ \mathbf{s_1}\end{bmatrix} \tag{5.10}$$

The DataPath of the unit has the *VectorByCirculant* module that evaluates $\mathbf{s}$. The memories with $\mathbf{m}_0$, $\mathbf{m}_1$, $\mathbf{L}_0$ and $\mathbf{L}_1$ are provided at the input of the component, the ControlUnit selects the right one depending on the partial sindrome to be evaluated.

The partial syndrome $\mathbf{s}_0$ is evaluated as a progressive xor among different vectors, then it is required to have only zeros stored in memory or not take into account its content in the

first run. The option selected is the second one, in order to avoid a clear of the memory that is a waste of time.

The result **s** is the xor among shifted versions of the partial message.

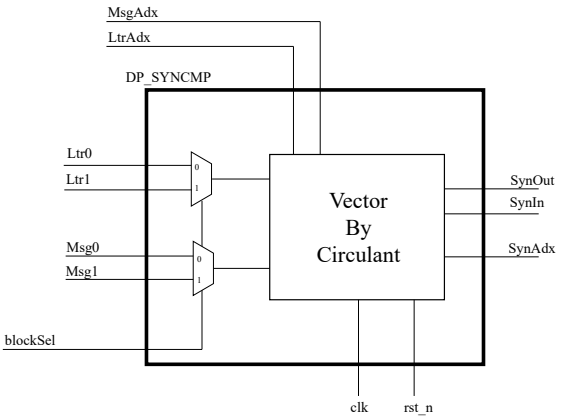


Figure 5.20: DataPath of the Syndrome Computation Module

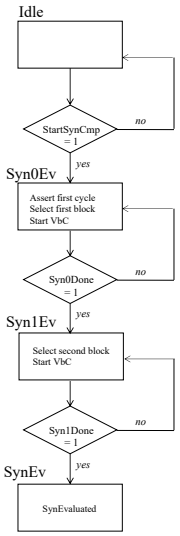


Figure 5.21: ControlUnit of the Syndrome Computation Module

The ControlUnit has to select the correct input for the module that evaluates the product. The unit can be adapted to matrices with 3 and 4 blocks. It is required to add one or two states to be able to selected the other blocks. The operations in the other states are the same, only the first one is different. The syndrome present in memory at the beginning may not be '0', then a signal has to be provided to the logic in order to be sure that during the first load the syndrome register contains only zeros and the xor among it and

the message shifted is not affected by unwanted values.

The *VectorByCirculant* units can be modified in two ways: the ControlUnit can be replicated considering a "first position" execution; the second is set a signal in one state that enable the clear signal to the syndrome register. The second option has been selected in order to avoid too many states to be inserted. The version required the addition of two states :

- **Load_Ltr_FirstCycle**: this is required in order to assert the flag that select *ClearSynReg*= 1, the state is reached after **Idle** only for **Syn0Ev** for the other ones **Load_Ltr** in the first state.

- **End_Ltr**: this state is reached every time the operation with a Ltr value has been completed and it is required to modify the value of the flag that in this case asserts *ClearSynReg*= 0.

The execution after the first time **End_Ltr** is reached has the flow described before. Despite the presence of a condition in a state this is still a Moore Machine since the signal is provided synchronously and not in a state transition.

The Syndrome is computed and a *done* signal is asserted. The result is in Syndrome Memory that is the input of the following unit that is still based on the same method, for this reason it is required to remove the noise from the last row. The Decoder has an additional unit, *SyndromeAdjust*, that has to remove the noise in the last row. The clear of the noise is realized in parallel in order to take just 3 clock cycles since the unwanted bits are just masked with by predefined vector, otherwise the number of cycles required is:

$$N_{clr\ cycles} = N_b - Difference \tag{5.11}$$

5.5.2 Correlation Evaluation

The Correlation is:

$$Syndrome * \mathbf{L} \tag{5.12}$$

The Syndrome is a $1 \times p$ vector and the matrix is the same as before, the computation is:

$$\mathbf{s} * \begin{bmatrix} \mathbf{L}_0 & \mathbf{L}_1 \end{bmatrix} = \begin{bmatrix} \mathbf{s} * \mathbf{L}_0 & \mathbf{s} * \mathbf{L}_1 \end{bmatrix} \tag{5.13}$$

The CorrelationEvaluation unit has almost the same structure as the previous one. The input vectors are the transpose of Ltr (again stored by position), the two blocks of the Correlation vector. and the Syndrome vector. The component provides at each iteration an update version of the Correlation.

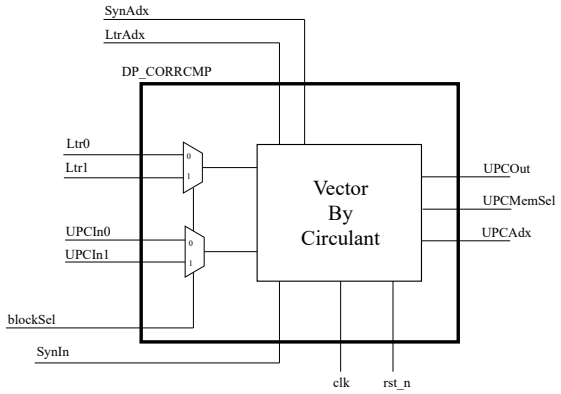


Figure 5.22: DataPath of the Correlation Computation Module

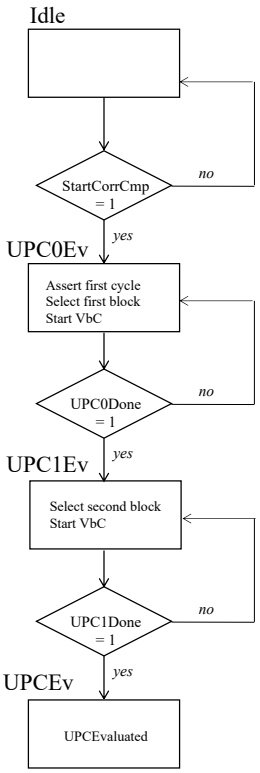


Figure 5.23: ControlUnit of the Correlation Computation Module

The parallelism introduced makes this block the one with more interconnections. The Correlation is a vector of length $2 * 15013$ of values that require at least 8 bits to be stored. Then if there are 8 values read at each iteration there are 64 bits for one memory, in total there are $3 * 64$ for UPCIn1,UPCIn2 and UPCOut. This can be a limit to the implementations in devices with a few resources.

The changes in the *VectorByCirculant* ControlUnit are different for the integer version because it is required to work with a cleared register each time a new UPC (both UPC1 and UPC2) is selected, because this vector is organized in blocks. The condition flag to state the first run to *VectorByCirculant* can be asserted while in **Idle** because the state is reached each time the method is called, the **End_Ltr** is still required to set the flag to zero.

5.6 Threshold Evaluation

The threshold evaluation module requires firstly to evaluate the weight of the syndrome. The SynW register stores this value, that is updated at each read of the syndrome memory. The number of 1s in one row is counted and summed to the previous value in SynW.
The DataPath of the is in Figure 5.24, the ControlUnit is simple and then not included here since it has just to read the Syndrome rows and update the register, the only requirement is to clear the register each time it starts the evaluation.

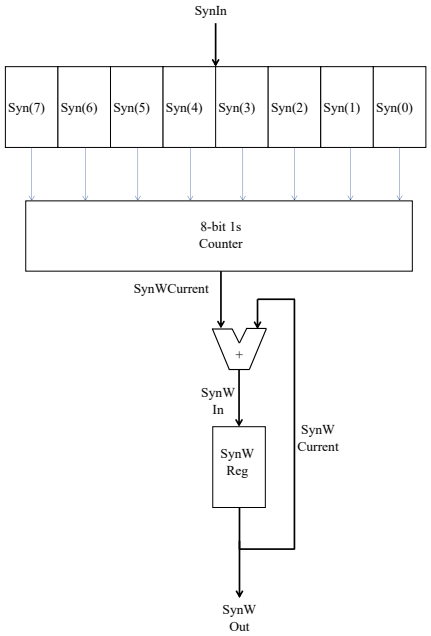


Figure 5.24: DataPath of Syndrome Weight module

The *SyndromeAdjust* is required to be performed before the evaluation of the Syndrome weight in order to not have the correct value from the computation.

The different strategy that can be employed here is to update SynW reading the Syndrome bit by bit. The solution requires at least 15013 cycles to update the weight bit by bit. The effect on the whole systems is an increase in the time to compute it, but the benefit is that instead of a unit that sums in parallel a group of bits the operation is done by an

accumulator, thus reducing the critical path.

The Threshold is given by ranges of SynW, then the difference among SynW and the value in the range is evaluated, as in Figure 5.25. There is a sign pattern coming from those differences that points the threshold.
The specific UPC value is pointed considering the specific error pattern.

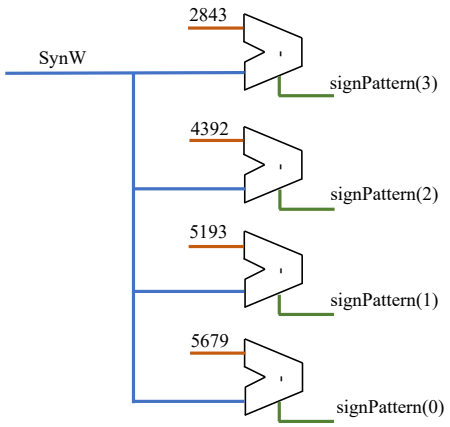


Figure 5.25: Sign Pattern Generation

The output is given by another LUT, describe in Table 5.2

Table 5.2: Threshold Look-Up table

Sign Pattern	Threshold
1111	42
0111	43
0011	44
0001	45
0000	46

5.7 Message Update

The message update is made considering the threshold evaluated and the UPC vector.
The row of UPC is read and subtracted to the threshold, then each value of the subtraction is read and if it is higher than the threshold it is saved as a position.
The positions that satisfy this condition are a few compared to the length of the vector, to exploit this fact the end of the difference bit an or port is present, if they are all 0 the row is skipped.
The process follows a different path if the row is 1, in this case a match is present. The unit checks each bit of the difference pattern and if there is a value higher than the threshold

it is stored in memory.

The MessageUpdate is performed reading from memory the corresponding row and flipping the error bit, then the row is stored again in memory.
The units introduces at this step are two:

- *ErrorPosition Unit*;
- *MessageUpdate Unit*.

The blocks are organized as the other units, they can be easily adapted to cases with a different number of circulant blocks.
The details of the ControlUnit and DataPath are described in the following.

5.7.1 ErrorPosition Unit

The inner block included in this unit is the *VectorPosFlip* component. The DataPath and ControlUnit are included in Figures 5.26 and 5.27.

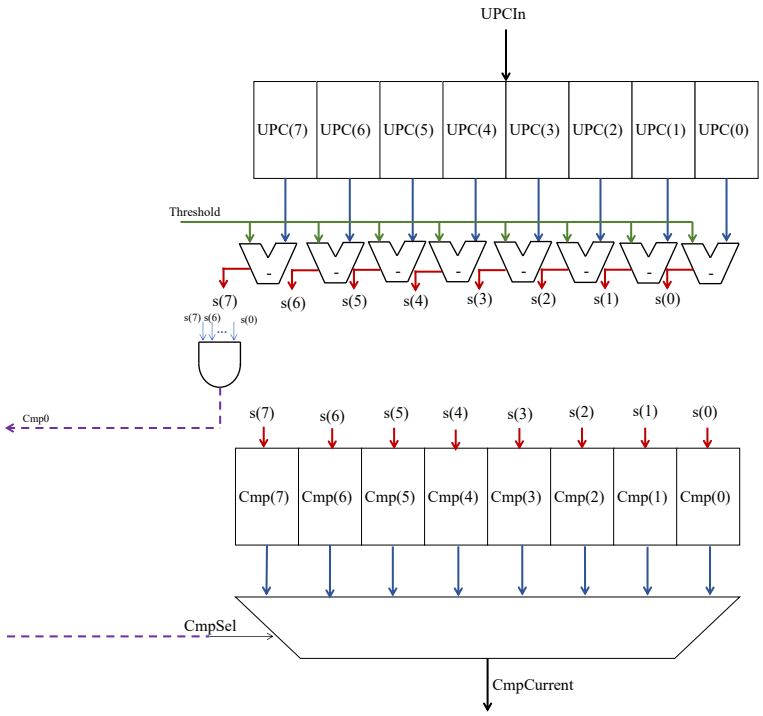


Figure 5.26: ErrorPosition DataPath

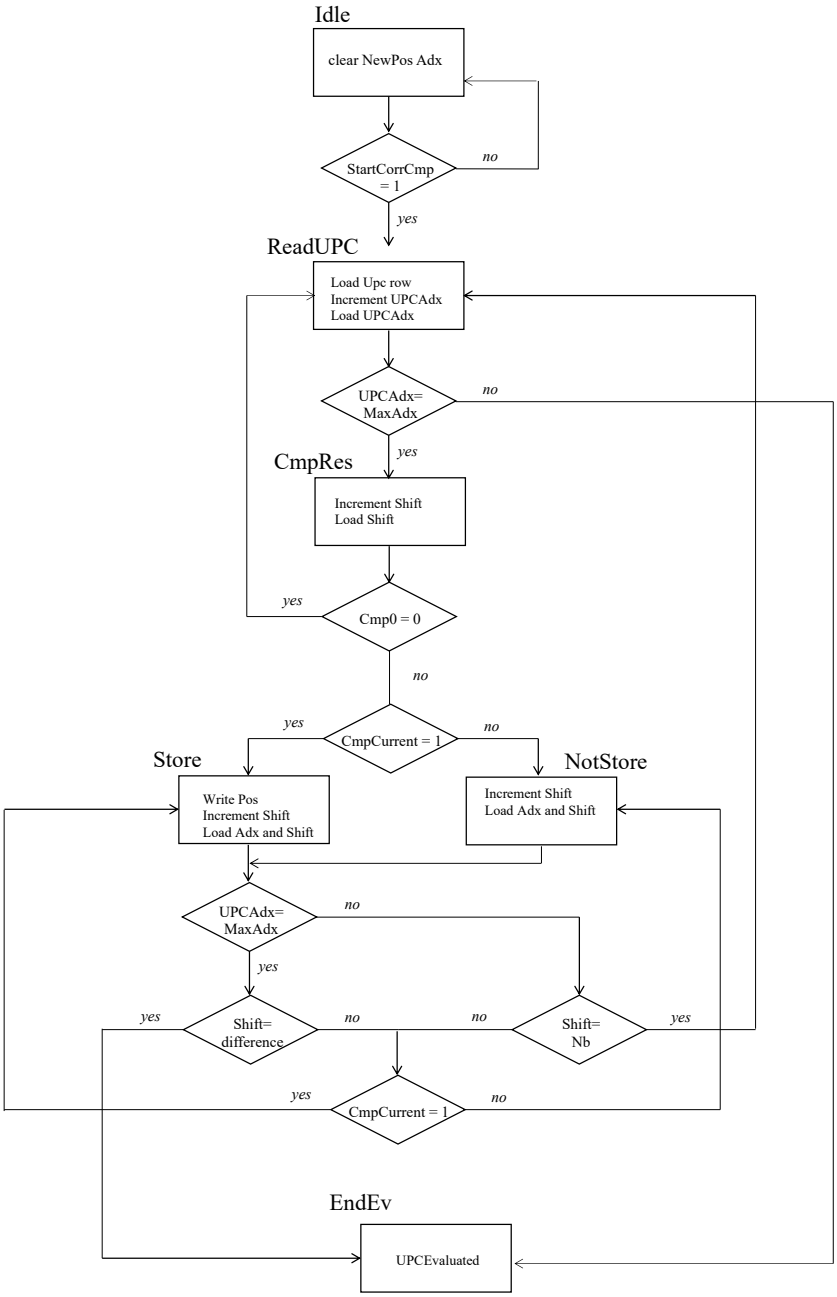


Figure 5.27: ErrorPosition ControlUnit

The ControlUnit requires the *Cmp0* signal in order to drive the execution and to know if the rows has to be skipped or not.

The second useful signal is *CmpCurrent* if it is 1 the value of *Message Address* and *CmpSel* has to be stored since it is the position of the error. *Cmp0* is different from 0 if at least one value of the row is higher than the threshold, in this case each sign bit is passed and when

equal to 1 (when a match is present) the value of Address and Shift present in a specific register are saved in memory. The operation is repeated until the end is reached. The unit provides the size of the vector too, this is because it changes during the decoding process.

The vectors generated contains the positions of both UPC1 and UPC2 that matched the threshold, the positions are saved in two separate memories.

The parallelization of the error position search has a big benefit due to the skip of the row in case no relevant position are present in a row. The improvement that can be introduced is in the the case *Cmp0* is equal to 1, a number of cycles equal to the parallelism is necessary to find the all the ones in a row, the increasing parallelism increases the number of cycles required. In example if the at most 80 errors are all in different rows and the parallelism is 64 the total cycles are $64 * 80 = 5120$, for Pos1 and Pos2.

The solution is to save in parallel the positions that satisfy the condition. The idea can be to use a *LeadingZero Architecture* in case the position to be saved is only one in one row. The study of the occurrence of an error in one row of the message saved in memory clarify this point. The simulations shows that the condition to have more than 3 positions per row are unlikely. Then it is required some logic to generate the position if it is only one or instead save multiple positions, it is better to introduce only this two possibilities since the more probable one is the condition of a single position.

The simulations showed that just 2 or 3 vector per cycle have two error position in the same row, this means that the cycles saved with a different unit are $(80 - 6) * 64 = 4736$. The problem can be due to the presence of a complex unit that could increase the critical path. The modification is not added to the Decoder because the number of cycles required by the unit is irrelevant compared to the *VectorByCirculant* module.

5.7.2 MessageUpdate Unit

The idea is to consider the positions as the values included in Ltr Memory, the last bit are decoded and stored in a register the output is decoded and stored in another register.

The second register present in the DataPath contains the row of the Message Memory with the erroneous bit. This is pointed by the first part of the position and loaded in a register.

The output of the two register is xored and the result is the bit flipping of that specific bit. Then the message is saved in memory.

The DataPath in Figure 5.28 shows the operation performed and the content of the registers for a specific case. The result is the Message with just one bit flipped.

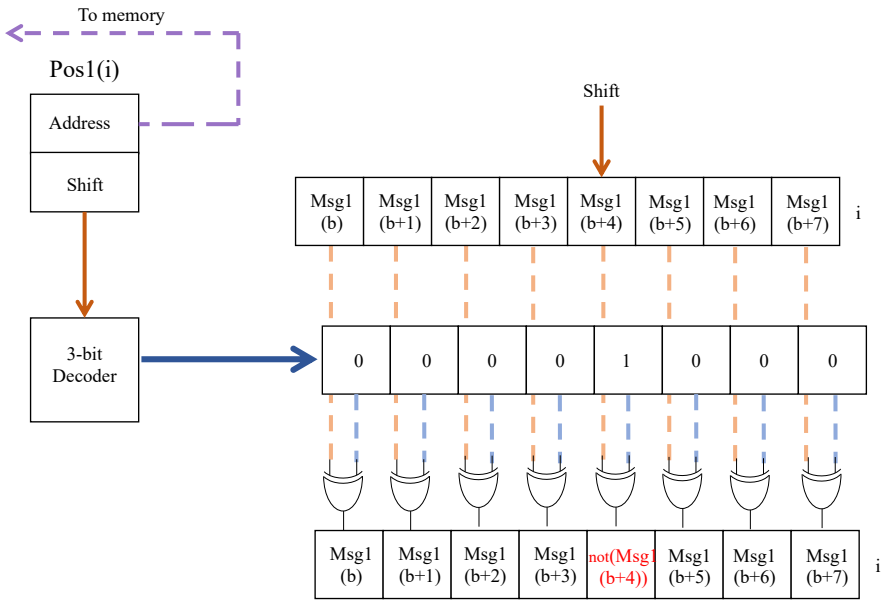


Figure 5.28: MessageUpdate operation

Chapter 6

Architecture Synthesis and Simulation

The synthesis result refers to the following parallelisms:

- *8 bit*
- *16 bit*
- *32 bit*
- *64 bit*

The higher parallelism requires more resources to be implemented, but depending on the device it can be feasible. The main problem regards the connections among the logic and the memory since increasing working with more bits requires wider memory or smaller but more in number.

6.1 Modelsim

The Simulation has been done on Modelsim, the cycles are referred to the worst case in which the convergence is reached in 3 iterations. The algorithms works fine, but in some cases more than 2 iteration are necessary. The cases in which the decoder is not able to converge are for sure the worst ones, but this is unlikely to happen.

The number of cycles required by the different parallelism are:

Table 6.1: Simulation results

Parallelism	Cycles $It = 4$	Cycles $It = 3$
8	6500000	4500000
16	3300000	2300000
32	1630000	1200000
64	824000	600000

The relevant number of cycles is the one employed by each unit to evaluate its result. The cycles required are summarized in Table 6.2 for each parallelism selected.

Table 6.2: Modelsim Simulations for each Module

Parallelism	Module	Cycles	%
8	Syndrome Computation	913240	49.6%
	Correlation Computation	913240	49.6%
	Syndrome Weight	3756	0.2%
	ErrorPos ($It = 0$)	8644	0.46%
	Message Update ($It = 0$)	578	0.03%
	Total:	**1839450**	
16	Syndrome Computation	457430	49.5%
	Correlation Computation	457430	49.5%
	Syndrome Weight	1880	0.2%
	ErrorPos ($It = 0$)	5780	0.6%
	Message Update ($It = 0$)	530	0.05%
	Total:	**923050**	
32	Syndrome Computation	229534	49.2%
	Correlation Computation	229534	49.2%
	Syndrome Weight	942	0.2%
	ErrorPos ($It = 0$)	5600	1%
	Message Update ($It = 0$)	514	0.1%
	Total:	**466124**	
64	Syndrome Computation	115247	48%
	Correlation Computation	115247	48%
	Syndrome Weight	472	0.2%
	ErrorPos ($It = 0$)	8244	3%
	Message Update ($It = 0$)	538	0.2%
	Total:	**239748**	

The result is useful in order to clarify the effect of some choices and presence of all the modules. The *SyndromeWeight* and *ErrorPos* unit could be included in the *SyndromeComputation* and *SyndromeComputation* modules, the effect of this change is that a more

complex ControlUnits and DataPath are required, the original *VectorByCirculant* module has to be further modified. The additional complexity could create a longer critical path that requires the addition of registers in order to cut the path and have the same values of the modular case, more states means that the benefit of the "all in one" strategy are lost. The solution could be not to include the small modules inside the larger ones, but to assert the "start" in the larger control unit and be able to read the same value in memory without the need to read it from memory two times. This prevents 81 useless read from memory, that compered to all the read and write present are just a few.

The Table 6.2 has the percentage of the units that should remain the same while increasing the parallelism, a part from the *MessageUpdate* unit, but analyzing in detail the result the cycles of the *ErrorPos* unit does not scale. The cycles required are only 1% (at most) of the total number of cycles required, then the solution to limit its effect (as explained in the dedicated section) are not necessarily required. The adoption of a different unit should be taken into account in case the parallelism is further increased, in that case the effect of the non optimized unit could be more relevant.

The last change to be mentioned required an additional unit to be inserted and an additional memory. The *SyndromeUpdate* unit evaluated just the positions of the Syndrome that has to be flipped, they are at most 12000. The new unit requires $2 * 12000$ cycles to compute the position, 12000 cycles to read the initial position and the same amount to store them, the update unit involves $3 * 12000$ cycles in total. The amount of cycles required (referred to the second iteration) are: 84000. The difference among this and the unit described is:

- *8 bit*: $913240 - 84000 = 829240$
- *16 bit*: $457430 - 84000 = 373430$
- *32 bit*: $229534 - 84000 = 145534$
- *64 bit*: $115247 - 84000 = 31247$

The improvement is huge in terms of cycles required. The Decoder DataPath and ControlUnit has to be modified in order to include the new unit, the unit is not included in the Decoder but its presence can be tested in future works.

The module was designed, but not tested inside the Decoder. It is the architectural derivation of Algorithm 9. The position of the syndrome is evaluated and flipped in order to avoid the use of an additional memory to save positions.

The other change that can be introduces regards the initial choices and modification of the Q-Decoder. The modification involved the evaluation of the Correlation, the original description involved both $\mathbf{Q}$ and $\mathbf{H}$ computing the integer value in two steps. The solution requires an additional memory to be connected and to store the two matrices of the private key, but in this case the cycles to evaluate the correlation. The reduction is due to the structure of the algorithm because it generates shifted versions of a vector for each asserted bit in the matrix, the dimensions of $\mathbf{H}$ and $\mathbf{Q}$ are 27 in total, on the other side the asserted bits in $\mathbf{L}$ are 81, the cycles are reduced by a factor of 4.5.

6.2 Synopsys

The critical path for each unit is the same it comes from the *presentState* of the ControlUnit in the VectorByCirculantInteger and goes to *SelShift*, in the accumulator and then to the output register.

The synthesis does not took into account the presence of the memories.

Table 6.3: Synthesis results

Parallelism	t_{cp}	Comb. Area	Non Comb. Area	Total Cell Area	Static Power
8	4.58 ns	4254 µm^2	5024 µm^2	9279 µm^2	0.2631 µW
16	5.30 ns	6324 µm^2	7882 µm^2	14103 µm^2	0.4057 µW
32	5.57 ns	10366 µm^2	13654 µm^2	24021 µm^2	0.6927 µW
64	6.09 ns	31602 µm^2	25226µm^2	56768 µm^2	1.2777 µW

The results are referred only to the decoder without memories, but the one containing positions are small and can be synthesized as Flip-Flops.

The result of the area has some anomalies. Then a detailed report is presented in order to understand clearly origin of these values.

The important result is the time employed by the Decoder to converge with different parallelisms. In Table 6.4 the comparison of the T_{ev} combining the values in Table 6.3 and Table **??**.

Table 6.4: Convergence time

Parallelism	T_{ev} $It = 4$	ratio	T_{ev} $It = 3$	ratio
8	29.8 ms		20.6ms	
16	17.5 ms	0.6	12.2ms	0.6
32	9.1 ms	0.52	6.7 ms	0.54
64	5.01 ms	0.55	3.6 ms	0.53

The time is almost halved at each time the parallelism is doubled, the reduction is not 0.5 but a bit more because the increase in the t_{cp} while the parallelism increases.

Convergence time reduction The introduction of the modifications mentioned above can further reduce the number of cycles, while the critical path should remain the same (the larger port is an adder).
The parallelism considered is 8-bit. The presence of the *SyndromeUpdate* unit considering $It = 4$ is applied in three cycles, then the cycles are reduce of 829240 per iteration:

$$T_{total} - T_{syn} = 6500000 - 3 * 829240 \approx 4100000 \tag{6.1}$$

The reduction is of 5ms, the converge time is comparable to the case with one iteration less.
The second improvement that splits the evaluation of the Correlation produces a reduction on every cycle, the amount is:

$$T_{CorrCmp}/4.5 = 913240/4.5 \approx 202843 \tag{6.2}$$

The difference is 710297 valid for three cycles, then the total reduction is:

$$T_{total} - 3 * T_{diff,CorrCmp} = 6500000 - 3 * 710297 \approx 4400000 \tag{6.3}$$

The time saved is 9ms.

The introduction of both the improvements saves in total 14ms.

6.2.1 Area analysis

The detailed report of the most relevant units is useful, in Table 6.5 and 6.6 there is the area of each module of the decoder. The tables reports in red the ratio between the current area and the one with half the parallelism.

Table 6.5: Modules area

Parallelism	CorrelationCmp	ratio	ErrorPos	ratio	MessageUpdate	ratio
8	3355 µm^2		2049 µm^2		891 µm^2	
16	5672 µm^2	1.69	3321 µm^2	1.62	1239 µm^2	1.40
32	10439 µm^2	1.84	5825 µm^2	1.75	1921 µm^2	1.55
64	26495µm^2	2.54	10805 µm^2	1.75	3282 µm^2	1.70

Table 6.6: Modules area

Parallelism	SyndromeCmp	ratio	SyndromeAdjust	ratio	SyndromeWeight	ratio
8	1580 µm^2		227 µm^2		624 µm^2	
16	2126 µm^2	1.34	406 µm^2	1.78	762 µm^2	1.22
32	3349 µm^2	1.57	769 µm^2	1.89	1098 µm^2	1.44
64	12337µm^2	3.68	1487 µm^2	1.93	1715 µm^2	1.56

The area of the ThresholdEvaluation module is fixed to 187 µm^2, the same for the iteration counter that is 50 µm^2.

The ratio should be constant increasing the parallelism but in most of the cases there are small changes, this is due to the fact that the ControlUnit of the modules are included in the computation, their dimension is constant then it is an offset in the complete area evaluation. The deeper analysis of the area shows that the size of the DataPath doubles for all the modules, just some small changes are present in the units that store positions since the increasing parallelism does not change these registers.

The *Syndrome Computation* and *Correlation Computation* has a different behavior that explains the difference in the complete area. The components with parallelism 64 and 32 are analyzed more in detail in the following, the two values has been selected since they have high ratios.

Table 6.7: Syndrome Computation

Parallelism	DataPath	ratio	ControlUnit
8	1545 μm^2		35 μm^2
16	2091 μm^2	1.35	35 μm^2
32	3314 μm^2	1.58	35 μm^2
64	12302μm^2	3.71	35 μm^2

The Syndrome Computation Unit includes a register in order to store the position, its size is an offset then the variation from 8 to 16 bit does not have a huge increase in the area. The effect of the offset is less relevant doubling further the size of the registers. There is something more to be taken into account in this analysis, since the unexpected area increase is between 32 to 64.

Table 6.8: Syndrome Computation DataPath components

Parallelism	Counters	ratio	Registers	ratio	Multiplexer	ratio
8	198 μm^2		91 μm^2	102		
16	178 μm^2	0.90	181 μm^2	2	259	2.5
32	162 μm^2	0.91	362 μm^2	2	664	2.5
64	144 μm^2	0.90	730 μm^2	2	7949	12

The area increase is unexpected for the 64 bit case. The idea to solve the problem is to find another way to implement the multiplexer, due to the sizes of the problem it is implemented in a behavioral way, but a description near to the hardware could solve the problem, another solution could be to use 32 bit multiplexer and one two way multiplexer to implement the whole device.

The same decoder is implemented in FPGA where the are increased by the same factor, but the critical path is longer. This can lead to a second way to reduce the gates, the clock

frequency can be increased in order to relax the specifications.

The first option tried on Synopsys was to run a second `compile` of the design, this option solved the previous problem. The second compile running on all the designs produced a strong reduction in the area, the critical path. The t_{cp}, area and power results after the second compile are reported in Table 6.9

Table 6.9: Synthesis after a second `compile`

Parallelism	t_{cp}	Total Cell Area	Static Power
8	3.65 ns	2591 µm^2	0.1126 µW
16	3.68 ns	3567 µm^2	0.1566 µW
32	3.66 ns	5166 µm^2	0.2137 µW
64	4.45 ns	23201µm^2	0.2714 µW

The reduction in the critical path makes the decoder even more faster. The critical path with different parallelisms is almost the same, this is due to the fact that the critical path is moved to the *MessageUpdate* unit. The complex multiplexer is reduced to a less complex unit that does not occupy a lot of area as previously.

6.2.2 Timing analysis

The path length distribution is analyzed for the 64 bit version.
The second longest path in the architecture is in the Syndrome Weight unit, the time required is approximately the same as the longest one. The variations that could be adopted to speed up the architecture can be applied on this two units, but this will result in an higher number of cycles required by the computation in this modules, the effect on Syndrome Weight is not relevant since it occupies a few cycles, but applied on the Correlation Computation unit this requires an additional cycle.
The critical path of the two units is cut with a register in the middle, the new synthesized design has a critical path of $T_{cp} = 5.53$ns and placed in the unit that evaluates the erroneous bits. This can be further cut, but the other DataPath of the Decoder has the same blocks, the path length is the same along the architecture.

The version generated by the second `compile` was not studied with a cut on the new critical path, but if this is the longest path of the design it can be easily reduced with a register in the middle, thus further reducing the critical path. The unit works for just a reduce number of cycles, then an additional state to include the presence of the register does not increase in an appreciable way the time required.

6.2.3 Memories required

The parallelism in the DataPath took into account the width of the memories, since they are usually provided with the dimensions power of 2 the sizes has been approximated in order to match this condition.

The memories required to store the positions are 6 and with size 16∗81 bit, this is the aspect ratio that they could have, independently of the parallelism. Since they are connected separately to the decoder, six distinct memories with that size are required. The total storage required is: 972 Byte.
The memories are:

- *Ltr1*,*Ltr2*;
- *LtrT1*,*LtrT2*;
- *Pos1*,*Pos2*.

The binary vector stored requires memories of different aspect ratio depending on the selected parallelism. In general the memory required is 6 kB. The aspect ratios are:

- *8 bit* : 8 $bit \times 1887$ $addresses$;
- *16 bit*: 16 $bit \times 938$ $addresses$;
- *32 bit*: 32 $bit \times 470$ $addresses$;
- *64 bit*: 64 $bit \times 235$ $addresses$;

The Correlation requires memories of:

- *8 bit* : 64 $bit \times 1887$ $addresses$;
- *16 bit*: 128 $bit \times 938$ $addresses$;
- *32 bit*: 256 $bit \times 470$ $addresses$;
- *64 bit*: 512 $bit \times 235$ $addresses$;

The memory required is 22kB, but with different aspect ratios.

6.3 FPGA implementation

The synthesis in a Xilinx FPGA took into account the presence of the Memories that are included in the design. The clock period considered is 10ns and was always met.

The most critical point are the connections, but even for 64 bit parallelism there was no problem in instantiating the decoder.
The FPGA selected belongs to the Artix-7 family, the characteristics are in Figure 6.2. The part considered is the XC7A15T, with CPG236 package.
The Spartan-7 family has even less resources, despite there is no possibility to synthesize the decoder for them it is possible to compare the result obtained for other families with the resources available for this one. The characteristics of the devices are provided in Figure 6.1.

Spartan-7 FPGAs

		I/O Optimization at the Lowest Cost and Highest Performance-per-Watt (1.0V, 0.95V)					
	Part Number	**XC7S6**	**XC7S15**	**XC7S25**	**XC7S50**	**XC7S75**	**XC7S100**
Logic Resources	Logic Cells	6,000	12,800	23,360	52,160	76,800	102,400
	Slices	938	2,000	3,650	8,150	12,000	16,000
	CLB Flip-Flops	7,500	16,000	29,200	65,200	96,000	128,000
Memory Resources	Max. Distributed RAM (Kb)	70	150	313	600	832	1,100
	Block RAM/FIFO w/ ECC (36 Kb each)	5	10	45	75	90	120
	Total Block RAM (Kb)	180	360	1,620	2,700	3,240	4,320
Clock Resources	Clock Mgmt Tiles (1 MMCM + 1 PLL)	2	2	3	5	8	8
I/O Resources	Max. Single-Ended I/O Pins	100	100	150	250	400	400
	Max. Differential I/O Pairs	48	48	72	120	192	192
Embedded Hard IP Resources	DSP Slices	10	20	80	120	140	160
	Analog Mixed Signal (AMS) / XADC	0	0	1	1	1	1
	Configuration AES / HMAC Blocks	0	0	1	1	1	1
Speed Grades	Commercial Temp (C)	-1,-2	-1,-2	-1,-2	-1,-2	-1,-2	-1,-2
	Industrial Temp (I)	-1,-2,-1L	-1,-2,-1L	-1,-2,-1L	-1,-2,-1L	-1,-2,-1L	-1,-2,-1L
	Expanded Temp (Q)	-1	-1	-1	-1	-1	-1

Package[1]	Body Area (mm)	Ball Pitch (mm)	**Available User I/O**: 3.3V SelectIO™ HR I/O					
CPGA196	8x8	0.5	100	100				
CSGA225	13x13	0.8	100	100	150			
CSGA324	15x15	0.8			150	210		
FTGB196	15x15	1.0	100	100	100	100		
FGGA484	23x23	1.0				250	338	338
FGGA676	27x27	1.0					400	400

Figure 6.1: Spartan-7 Family[19]

6.3.1 64 bit

The result obtained for the resources employed if the only constraint applied is the clock period are:

Table 6.10: 64 bit parallelism

Resource	Utilization	Utilization (%)
LUT	3579	34%
FF	2459	12%
BRAM	21	84%
IO	5	5%
BUFG	1	3%

The worst negative slack for the specified clock period is : $T_{wns} = 1.813$ns, for the implemented decoder. Considering a clock period of $T_{clk} = 10$ns the critical path is:

$$T_{cp} = T_{clk} - T_{wns} = 8.187\text{ns} \tag{6.4}$$

The Total On-Chip Power is: 0.138W.

The type of memories required are block RAMs with the following sizes:

Artix-7 FPGAs

		Transceiver Optimization at the Lowest Cost and Highest DSP Bandwidth (1.0V, 0.95V, 0.9V)							
	Part Number	XC7A12T	XC7A15T	XC7A25T	XC7A35T	XC7A50T	XC7A75T	XC7A100T	XC7A200T
Logic Resources	Logic Cells	12,800	16,640	23,360	33,280	52,160	75,520	101,440	215,360
	Slices	2,000	2,600	3,650	5,200	8,150	11,800	15,850	33,650
	CLB Flip-Flops	16,000	20,800	29,200	41,600	65,200	94,400	126,800	269,200
Memory Resources	Maximum Distributed RAM (Kb)	171	200	313	400	600	892	1,188	2,888
	Block RAM/FIFO w/ ECC (36 Kb each)	20	25	45	50	75	105	135	365
	Total Block RAM (Kb)	720	900	1,620	1,800	2,700	3,780	4,860	13,140
Clock Resources	CMTs (1 MMCM + 1 PLL)	3	5	3	5	5	6	6	10
I/O Resources	Maximum Single-Ended I/O	150	250	150	250	250	300	300	500
	Maximum Differential I/O Pairs	72	120	72	120	120	144	144	240
Embedded Hard IP Resources	DSP Slices	40	45	80	90	120	180	240	740
	PCIe® Gen2[1]	1	1	1	1	1	1	1	1
	Analog Mixed Signal (AMS) / XADC	1	1	1	1	1	1	1	1
	Configuration AES / HMAC Blocks	1	1	1	1	1	1	1	1
	GTP Transceivers (6.6 Gb/s Max Rate)[2]	2	4	4	4	4	8	8	16
Speed Grades	Commercial Temp (C)	-1, -2	-1, -2	-1, -2	-1, -2	-1, -2	-1, -2	-1, -2	-1, -2
	Extended Temp (E)	-2L, -3	-2L, -3	-2L, -3	-2L, -3	-2L, -3	-2L, -3	-2L, -3	-2L, -3
	Industrial Temp (I)	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L

	Package[3], [4]	Dimensions (mm)	Ball Pitch (mm)	**Available User I/O**: 3.3V SelectIO™ HR I/O (GTP Transceivers)							
	CPG236	10 x 10	0.5		106 (2)		106 (2)	106 (2)			
	CPG238	10 x 10	0.5	112 (2)		112 (2)					
	CSG324	15 x 15	0.8		210 (0)		210 (0)	210 (0)	210 (0)	210 (0)	
	CSG325	15 x 15	0.8	150 (2)	150 (4)	150 (4)	150 (4)	150 (4)			
	FTG256	17 x 17	1.0		170 (0)		170 (0)	170 (0)	170 (0)	170 (0)	
	SBG484	19 x 19	0.8								285 (4)
Footprint Compatible	FGG484[5]	23 x 23	1.0		250 (4)		250 (4)	250 (4)	285 (4)	285 (4)	
	FBG484[5]	23 x 23	1.0								285 (4)
Footprint Compatible	FGG676[6]	27 x 27	1.0						300 (8)	300 (8)	
	FBG676[6]	27 x 27	1.0								400 (8)
	FFG1156	35 x 35	1.0								500 (16)

Notes:
1. Supports PCI Express Base 2.1 specification at Gen1 and Gen2 data rates.
2. Represents the maximum number of transceivers available. Note that the majority of devices are available without transceivers. See the Package section of this table for details.

Figure 6.2: Artix -7 Family[19]

- *VectorMemory*: with size 64 bit by 235, three of them are required;
- *UPCMemory*: with size $64 * 8$ bit by 235, two of them are required;
- *PosMemory*: with size 16bit by 81, six of them are required.

The utilization of the BRAM in this case is higher compared to other cases. The 64bit parallelism requires a memory with width of 512bit, that is not available then this is divided in two separated memories that since are block memories are available in predefined sizes. The high value of utilization come from the fact that wide memories are required.

6.3.2 32 bit

The result obtained for the resources employed is:

Table 6.11: 32 bit parallelism

Resource	Utilization	Utilization (%)
LUT	1994	19%
FF	1309	6%
BRAM	12.50	50%
IO	5	5%
BUFG	1	3%

The worst negative slack for the specified clock period is : 2.153ns, for the implemented decoder.

$$T_{cp} = T_{clk} - T_{wns} = 7.848\text{ns} \tag{6.5}$$

The Total On-Chip Power is: 0.109W.

The type of memories required are block RAMs with the following sizes:

- *VectorMemory*: with size 32 bit by 471, three of them are required;
- *UPCMemory*: with size $32 * 8$ bit by 471, two of them are required;
- *PosMemory*: with size 16bit by 81, six of them are required.

6.3.3 16 bit

The result obtained for the resources employed is:

Table 6.12: 16 bit parallelism

Resource	Utilization	Utilization (%)
LUT	1160	11%
FF	762	4%
BRAM	12.50	50%
IO	5	5%
BUFG	1	3%

The worst negative slack for the specified clock period is : 2.953ns, for the implemented decoder.

$$T_{cp} = T_{clk} - T_{wns} = 7.04\text{ns} \tag{6.6}$$

The Total On-Chip Power is: 0.092W.

The type of memories required are block RAMs with the following sizes:

- *VectorMemory*: with size 16 bit by 938, three of them are required;
- *UPCMemory*: with size $16 * 8$ bit by 938, two of them are required;
- *PosMemory*: with size 16bit by 81, six of them are required.

6.3.4 8 bit

The result obtained for the resources employed is:

Table 6.13: 8 bit parallelism

Resource	Utilization	Utilization (%)
LUT	838	8%
FF	493	2%
BRAM	12.50	50%
IO	5	5%
BUFG	1	3%

The worst negative slack for the specified clock period is : 4.189ns, for the implemented decoder.

$$T_{cp} = T_{clk} - T_{wns} = 5.811\text{ns} \tag{6.7}$$

The Total On-Chip Power is: 0.089W.

The type of memories required are block RAMs with the following sizes:

- *VectorMemory*: with size 32 bit by 471, three of them are required;
- *UPCMemory*: with size $32 * 8$ bit by 471, two of them are required;
- *PosMemory*: with size 16bit by 81, six of them are required.

6.3.5 Comments on FPGA implementation

The code synthesized for FPGA has not the same issues of the ASIC version, the increase in the are is approximately the same and even the larger version does not creates problems.

The results are tested varying the clock applied to the decoder, but the values of area and critical path remains the same. The utilization of the logic inside is not a problem, only a small part of the array is occupied. the comparison of Figures 6.2 and 6.1 allows to have an idea of the possible implementations on smaller devices.
The LUT required are at most 3600 and the Flip Flops are 2500 for 64-bit parallelism, the decoder can be implemented in smaller FPGAs as the Spartan-7 XC7S6, the only limit is in the number of memory blocks required because this parallelism has an utilization of 84% of the memory, smaller devices could not be enough. The idea can be to reduce the parallelism to 32-bit, in this case the memory required is less and even smaller devices can be enough.

6.4 Comparison with RSA implementations

The smallest Decoder can correct a message in at most 29ms with an ASIC implementation and in 37ms for the FPGA implementation.
The results of an FPGA implementation of RSA are present in the PhD Thesis [15] that developed one of the Decoders explained above. The time employed by the RSA employing 1024 bit long keys is 51ms.
The result obtained with this architecture is huge, since even the unoptimized version is faster than commonly used decoders. Moreover the RSA decoding time is huge if more secure keys are employed, nowadays it is suggested to use keys of 2048 or 4096 bit, the decoding time dramatically increases.

Chapter 7

Conclusions

The selection of a method to construct LDPC codes and the decoder explored different possibilities. The structured and unstructured codes behave in almost the same way, then the candidates selected are the QC-LDPC codes in order to reduce the size of the keys.

The method discovered that satisfied the cryptographic requirements is LEDApck because it adopted quasi cyclic matrix as keys and the choice of the Q-Decoder is the most suitable for a hardware implementation, the structure of the code combined with the simplicity of the BitFlipping allowed to design a decoder that in some milliseconds can provide the clear text.

The multiplication is parallelized because the result is computed by N_b bit at time and not by single bit, the parallelization does not increase that much the area occupied, since it can be employed even in small FPGAs, but the improvement in the evaluation time is significant.

7.1 Future work

The future work that can be done can cover many aspects of the hardware implementation. The possibilities are listed below.

- The *SyndromeUpdate* unit can be introduced in order to reduce the number of cycles required by the decoder to converge. The effect of the unit in the critical path length and area is relevant since if it increases too much the combinatorial path the reduction in the number of cycles can not be enough to save time. The new unit has different impact for the various parallelisms studied, in some cases it may not be suggested.
- The second change to be introduced is in the *CorrelationCmp*, this change requires a more invasive change in the decoder because an additional memories to be connected, this may limits its employing in smaller devices. The way to reduce the number of memory connected can be to evaluate the circulant matrix product in the decoder, they are not kept in memory but in FlipFlops inside the architecture. The effect of this changes has to be studied.

The security was a topic non studied in this thesis, only the work done by others is presented. The big improvement in the architecture can be to introduce some modifications to make the design secure against power analysis attacks since in the computation of the multiplication is present an asymmetry in the ControlUnit, this can be exploited by a possible attacker.

The thesis describes possible decoders and one implementation, other studies can cover the implementation of both the encoder and decoder.

Bibliography

[1] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter, "Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms", in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds., Cham: Springer International Publishing, 2017, pp. 241–270, ISBN: 978-3-319-70697-9.

[2] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring", in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[3] R. J. McEliece, "A Public-Key Cryptosystem Based On Algebraic Coding Theory", *Deep Space Network Progress Report*, vol. 44, pp. 114–116, Jan. 1978.

[4] N. J. Patterson, "The Algebraic Decoding of Goppa Codes", *IEEE Transactions on Information Theory*, vol. IT-20, no. 2, 1975.

[5] D. J. Bernstein, T. Lange, and C. Peters, "Attacking and Defending the McEliece Cryptosystem", in *Post-Quantum Cryptography*, J. Buchmann and J. Ding, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 31–46, ISBN: 978-3-540-88403-3.

[6] J. Stern, "A method for finding codewords of small weight", in *Coding Theory and Applications*, G. Cohen and J. Wolfmann, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 106–113, ISBN: 978-3-540-46726-7.

[7] A. Canteaut and H. Chabanne, "A further improvement of the work factor in an attempt at breaking McEliece's cryptosystem", Jan. 1994.

[8] A. Canteaut and N. Sendrier, "Cryptanalysis of the Original McEliece Cryptosystem", in *Advances in Cryptology — ASIACRYPT'98*, K. Ohta and D. Pei, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 187–199, ISBN: 978-3-540-49649-6.

[9] R. Gallager, "Low-density parity-check codes", *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962, ISSN: 0096-1000. DOI: 10.1109/TIT.1962.1057683.

[10] and E. Eleftheriou and D. Arnold, "Progressive edge-growth Tanner graphs", in *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No.01CH37270)*, vol. 2, 2001, 995–1001 vol.2. DOI: 10.1109/GLOCOM.2001.965567.

[11] M. Baldi, *QC-LDPC Code-Based Cryptography*. Jan. 2014, ISBN: 978-3-319-02555-1. DOI: 10.1007/978-3-319-02556-8.

[12] R. Townsend and E. Weldon, "Self-orthogonal quasi-cyclic codes", *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 183–195, 1967, ISSN: 0018-9448. DOI: 10.1109/TIT.1967.1053974.

[13] M. Baldi, F. Chiaraluce, R. Garello, and F. Mininni, "Quasi-Cyclic Low-Density Parity-Check Codes in the McEliece Cryptosystem", in *2007 IEEE International Conference on Communications*, 2007, pp. 951–956. DOI: 10.1109/ICC.2007.161.

[14] F. C.G.P.P. S. M. Baldi A. Barenghi, "Low-Density Parity-Check Code-based public-key cryptosystems", p. 58, 2017. [Online]. Available: https://www.ledacrypt.org/documents/LEDApkc_spec_latest.pdf.

[15] O. A. Rasheed, "Low complexity decoding algorithms suitable for application in asymmetric cryptosystems", PhD thesis, University of Belgrade, 2015. [Online]. Available: https://bit.ly/2UtVXQT.

[16] K. K.Kobayashu N.Takagi, "An Algorithm for Inversion in GF(2^m) Suitable for Implementation usgin a Polynomial Multiply Instruction on GF(2)",

[17] M. Koochak Shooshtari, M. Ahmadian-Attari, T. Johansson, and M. Reza Aref, "Cryptanalysis of McEliece cryptosystem variants based on quasi-cyclic low-density parity check codes", *IET Information Security*, vol. 10, no. 4, pp. 194–202, 2016, ISSN: 1751-8709. DOI: 10.1049/iet-ifs.2015.0064.

[18] K. Kobara and H. Imai, "Semantically Secure McEliece Public-Key Cryptosystems -Conversions for McEliece PKC -", vol. 1992, Feb. 2001, pp. 19–35. DOI: 10.1007/3-540-44586-2_2.

[19] [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf.