FPGA Implementation Of An LDPC Decoder And Decoding Algorithm Performance

BY

LUIGI PEPE
B.S., Politecnico di Torino, Turin, Italy, 2011

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2013

Chicago, Illinois

Defense Committee:
David Borth, Chair and Advisor
Dan Schonfeld
Giuseppe Vecchi, Politecnico di Torino

ACKNOWLEDGMENTS

I would like to thank my UIC advisor David Borth, without whom this work would not have been possible, for his guidance, his suggestions and his availability to constantly meet me. I really appreciate the way he helped me in getting this work done by the end of April 2013.

I would also like to thank my Politecnico di Torino advisor, Giuseppe Vecchi, for reaching me in Chicago and assisting me during the defence of this thesis, and also for having been of great help in giving explanations about the TOP-UIC program.

Thanks to Lynn Thomas, for her incredible patience and her ability to answer any kind of question I have had during this Academic Year.

A big thank to my fellow Antonello who has always been there for any kind of help or advise I asked for.

I have to thank my parents, who have always supported me and stimulated me to do my best. I'm sure that without them I would not have completed this thesis.

Also, I want to thank all my friends from Italy, the ones with whom I have been constantly in touch with and who made me feel closer to home (especially Lorenzo and Simone) and also the ones that I heard a few times, but that supported me ever since I was still in Italy and I was not even sure to move to Chicago for this Academic Year. I really can not cite all of them because they would be too many.

Thanks to all my friends in Chicago, American and international, who lived this wonderful experience with me. We had so much fun together!

ACKNOWLEDGMENTS (continued)

Thank also to my mates Federico, Paolo, Francesco and Davide, with whom I have spent a lot of time living fantastic experiences in Chicago.

Thank to Maurizio, my oldest friend, who gave me advices and who has been a landmark for me during these months.

Finally, a special thank to Marshall Bruce Mathers III and to his best citation:"...you can do anything you set your mind to, man...", which helped me a lot and made me believe in myself!

LP

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF FIGURES

LIST OF FIGURES (continued)

FIGURE **PAGE**

LIST OF FIGURES (continued)

FIGURE **PAGE**

SUMMARY

In this work the Low Density Parity Check (LDPC) codes have been introduced and described as very powerful *error correcting* codes. The whole Thesis has been divided in 5 Chapters: in the first Chapter a generical introduction to Block Codes has been provided, followed by a more deeply description about LDPC codes. In the second Chapter the attention has been posed on the decoding algorithms and decoding architectures that are mostly used in practical cases. Then in the third Chapter a new kind of LDPC decoding architecture has been proposed, while in the fourth Chapter several MATLAB (see (1)) simulations results are shown to explain the behaviour of the different decoding algorithms and their performances. The last Chapter is about the conclusions and eventual improvements to both the presented decoder implementation and to the decoding algorithms used.

CHAPTER 1

COMMUNICATION CODES

1.1 Channel Coding

A general communication system transmits information data from a source to a destination, through a specific channel or medium, like air (Figure 1). Of course the received data at the destination could be different with respect to the original data sent at the source, and this because of the channel distortion or the external noise. From Shannon's theorem (see (2, p.22)) we know that we can achieve reliable transmissions if the data rate is lower than the capacity of the channel.

Theorem 1 (Shannon's Theorem) *For any $\epsilon > 0$, if the data rate R is lower than the capacity of the Channel C, and the length n of the codewords is sufficiently large, then there exists a code with error probability $p_e < \epsilon$.*

In other words we can say that the channel capacity is the maximum rate at which the communication is reliable.

To avoid data distortion a more complex structure is used, which includes also an encoder and a decoder. The channel encoder introduces redundancy to the source information such that the transmission is more reliable. The channel decoder recovers (or tries to) the original information data from the received data. In other words the encoder transforms a sequence of

Figure 1. General Communication System

information symbols into a *codeword*, and the decoder retrieves the original data sequence from the received *codeword*.

There are two main kinds of *channel coding techniques*. The first kind is called Automatic Repeat Request (ARQ) and in this case the receiver requests for a retransmission of the data if the received codeword is unreliable. The second kind is called Forward Error Correction (FEC), where the channel tries to correct eventual errors and estimates the original codeword from the received data. The FEC technique is used when a high thoroughtput is required by the system, while the ARQ technique is more suitable when the channel is unknown or when we need to be sure that the received data are correct. Today FEC channel coding is more common and it generally exploits two different codes: convolutional codes and block codes; in this thesis we will focus our attention on block codes (since LDPC are block codes), which are introduced in the following.

In coding theory there are two main ways to model a binary channel. The first one is called Binary Symmetric Channel (BSC). It assumes that the transmitter sends a bit (0 or 1) to the receiver, but this bit can be flipped in the channel with a probability p, *crossover probability*, that is usually small (Figure 2). The second model is the Binary Erasure Channel (BEC) in

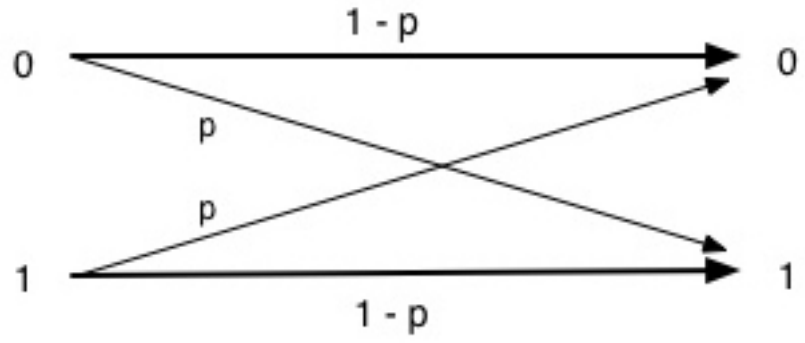

Figure 2. Binary Symmetric Channel

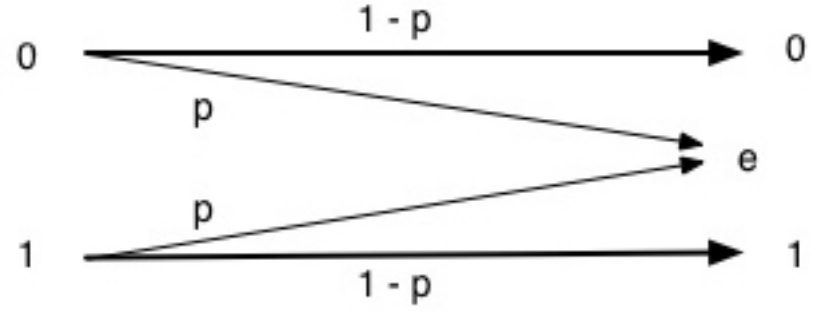

Figure 3. Binary Erasure Channel

which the transmitter sends a bit and the receiver receives the same (correct) bit, or it receives an "erasure" message, that means the bit was erased (Figure 3). A more general channel model is the AWGN (Additive White Gaussian Noise) channel in which a Gaussian noise is added to the signal in the channel.

1.2 Linear Block Codes

The block codes are very common today and they consist in encoding and decoding a sequence (block) of information symbols instead of symbols considered one by one. Usually an information block of length k is denoted by $\mathbf{u}$, where, if we are dealing with a binary code, each information symbol u_i can be 0 or 1. A block code is a mapping between any vector $\mathbf{u}$ and a codeword $\mathbf{c}$ of length n, where again, each code symbol can take on 0 or 1. If the information block has length k this means that there are 2^k possible codewords, each of length n, and k is the *dimension* of the code. A linear (n, k) block code is called regular iff its codewords form a k-dimensional subspace of the vector space spanned by all the n-tuples (3, p.66). In other words any linear combination of two codewords must be still a codeword.

If we want to transmit a block of k symbols, actually we will send through the channel a codeword of n symbols; from this we define the **rate** of the code as $r = \frac{k}{n}$. Furthermore we define the **weight** of a codeword as the number of its non-zero elements and the **minimum distance** of a code as the minimum weight of the codewords of which it consists. The distance between two codewords is the number of elements in which they differ.

A linear block code can be used for detection and correction of a code, and its *error correcting capability* depends on the minimum distance of the code itself. If the minimum distance is d then the code can correct $\lfloor \frac{d-1}{2} \rfloor$ using a Maximum Likelihood decoding, where $\lfloor \cdot \rfloor$ indicates the largest previous integer and provided that the error probability is stricly less than 0.5. If the d is even then the code can correct $\lfloor \frac{d-1}{2} \rfloor$errors and detect $\frac{d}{2}$ errors (2, p.10).

1.2.1 Encoding A Block Code Using Parity-Checks

One easy encoding method consists in setting the first part of the n-length codeword equal to the message we want to send and filling the second part of the codeword with n - k check symbols. Suppose we are sending a message $\mathbf{u} = \{u_1, u_2, \ldots n_k\}$, then the encoder will produce a codeword $\mathbf{c} = \{c_1, c_2, \ \ldots c_n\}$ where

$$c_1 = u_1, \ c_2 = u_2, \ \ldots \ c_k = u_k$$

and

$$c_{k+1}, \ c_{k+2}, \ \ldots c_n$$

will be the check symbols. The corresponding parity check matrix will be put in a systematic form, that is

$$H = [A|I_{n-k}]$$

where A is some fixed matrix, while I_{n-k} is the identity matrix. The codewords $\mathbf{c}$ will be chosen in such a way that

$$H(c_1, c_2, \ \ldots c_n)^T = 0$$

If H is a binary matrix, then the linear code relative to H is the set of all codewords which satisfy

$$H\mathbf{c}^{\mathsf{T}} = 0$$

In order to find a codeword $\mathbf{c}$ from the original message $\mathbf{u}$ we need to use the so called generator matrix G, which in its systematic form is defined as

$$G = [I_k|A^{\mathsf{T}}]$$

and we have

$$\mathbf{c} = \mathbf{u}G$$

The generator matrix and the parity-check matrix are related by

$$GH^{\mathsf{T}} = 0$$

which means that the row space of G is orthogonal to the row space of H.

1.2.2 Error Detection And Correction

Now suppose we are sending a message $\mathbf{u} = \{u_1, u_2, \ldots n_k\}$ from a source to a destination. The original message is mapped to a codeword $\mathbf{c} = \{c_1, c_2, \ \ldots c_n\}$ and after the transission, at

the receiver we get the vector $\mathbf{y} = \{y_1, y_2, \ldots y_n\}$. We define the *error vector* $\mathbf{e} = \{e_1, e_2, \ldots e_n\}$ as

$$\mathbf{e} = \mathbf{y} - \mathbf{c}$$

Each element e_i can be 0 with probability (1 - p) (probability that no error occurred on i^{th} symbol) or 1 with probability p (error probability over i^{th} symbol). (1 - p) is in general greater than p so it is more likely to have an error vector $\mathbf{e}$ with a low number of 1s (if $\mathbf{e}$ is $00\ldots0$ it means that the decoder received the correct codeword, with probability $(1-p)^n$).

Usually we need to distinguish between two kinds of decoders, according to whether they attempt to correct errors or just to detect them. The **error detection** decoder consists simply in detecting that an error occurred, and it is achieved by calculating the syndrome of $\mathbf{y}$

$$\mathbf{s} = H\mathbf{y}^T$$

and if the $\mathbf{s}$ is not zero than the received vector $\mathbf{y}$ is not a codeword. Sometimes the original codeword could be turned into a vector $\mathbf{y}$ corresponding to another codeword and in that case the error would be undetectable, because the result of the syndrome $\mathbf{s}$ would be zero even if $\mathbf{y}$ and $\mathbf{c}$ are different. For this reason it is important to consider the minimum distance of the code and say that a code with minimum distance d can detect t errors iff $t < d$ (3, p.78). This is true because, for example, if the minimum distance is d=2 then a codeword can be turned into

another one only if at least two symbols are flipped. So in case only one error occurred it would be certainly detected by the decoder. The **error correction** is possible only if the decoder is capable to recover the original codeword from the received vector $\mathbf{y}$. There are two types of decoders which are mostly used: the *maximum likelihood* (ML) decoder and the *maximum a posteriori* (MAP) decoder.

The ML decoder chooses the codeword which is most likely to have produced the received bit sequence $\mathbf{y}$, or in formula, it chooses the recovered codeword according to:

$$\hat{\mathbf{c}} = \underset{c \in \mathsf{C}}{\operatorname{argmax}}\ p(\mathbf{y}|\mathbf{c})$$

So it chooses the codeword $\mathbf{c}$ which maximizes the probability that $\mathbf{y}$ arrived from the channel if $\mathbf{c}$ was sent, that is, the codeword with minimum distance from $\mathbf{y}$ (the decoder assumes always $\mathbf{e}$ with the minimum weight possible, provided that the probability of error is less than 0.5, otherwise it should assume $\mathbf{e}$ with maximum weight possible (decoding with Standard Array, see (2, p.16))). In order to choose the right codeword, the decoder compares all the possible codewords with the received $\mathbf{y}$ and this turns out to be computationally very expensive.

On the other hand the MAP decoder chooses the codeword $\mathbf{c}$ according to

$$\hat{\mathbf{c}} = \underset{c \in \mathsf{C}}{\operatorname{argmax}}\ p(\mathbf{c}|\mathbf{y})$$

where $p(\mathbf{c}|\mathbf{y})$ is called *a posteriori* probability (see (4, p. 22)). If each codeword is equally

likely to be sent, then $p(\mathbf{c}|\mathbf{y}) = p(\mathbf{y}|\mathbf{c})$ and the two decoders result to be equal. Moreover, with the MAP decoder the computational cost can be very high for long original messages $\mathbf{u}$; for this reason several solutions have been found to ease the decoding, substituting these decoding methods with *iterative* algorithms.

1.3 LDPC Codes

LDPC codes were first introduced by Robert Gallager, a PhD student at MIT, in his PhD thesis in 1962 (5). At that time the decoding of such codes was too complex for the available technology and for this reason these codes remained unused for 35 years. They were rediscovered and started to be considered as very powerful codes only in the 1990s.

A Low-Density Parity-Check code is a block code with a particular H, whose elements are mostly 0s and only a few of them are 1s. This code is designed by constructing the H matrix first, with all the constraints due on it, and then by finding a generator matrix G. The main difference with respect to classical block codes is that LDPC codes are not decoded with the *maximum likelihood* algorithm but with an iterative method which uses a graphical representation of the parity-check matrix.

Considering that each row of H corresponds to a parity-check equation, an LDPC code is said to be *regular* if each code bit is contained in a fixed number w_c of equations and each equation contains a fixed number w_r of code bits. An example of parity-check matrix of a regular code is the following:

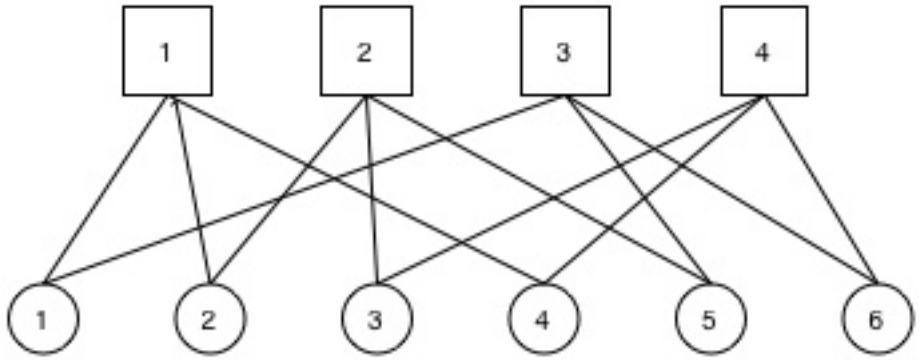

Figure 4. Tanner Graph: variable nodes are represented by circles and check nodes are represented by squares.

$$H = \begin{bmatrix} 1\,1\,0\,1\,0\,0 \\ 0\,1\,1\,0\,1\,0 \\ 1\,0\,0\,0\,1\,1 \\ 0\,0\,1\,1\,0\,1 \end{bmatrix}$$

where $w_c = 2$ and $w_r = 3$.

A graphical represetation of a parity-check matrix is given by a bipartite graph, called a Tanner Graph, in which all the nodes can be separated into two different types and each node of one type can be connected only with a node of the other type. The two nodes in a Tanner Graph can be *variable nodes* (VN), also called *bit nodes*, and *check nodes* (CN). A *variable node* j can be connected with a *check node* i only if the element h_{ji} of H is 1. The Tanner Graph of the parity-check matrix H seen in the previous example is shown in Figure 4. As we can

see from the figure each CN is connected to 3 VNs and each VN is connected to 2 CNs. We also say that the degree of each CN is 3 and the degree of each VN is 2. Furthermore the total number of edges in the graph is equal to the number of 1s in the parity-check matrix (see (6)).

An LDPC code is said to be irregular if w_c and w_r are not fixed but they are functions of columns and rows respectively. Accordingly not all CNs (or VNs) have the same degree. In the literature (7, p.204) two *degree-distribution polynomials* are defined for irregular codes. For VNs we have

$$\lambda(\chi) = \sum_{d=1}^{d_v} \lambda_d \chi^{d-1}$$

where λ_d is the fraction between the edges connected to VNs of degree d and all edges in the graph. For CNs we have

$$\rho(\chi) = \sum_{d=1}^{d_c} \rho_d \chi^{d-1}$$

where again ρ_d is the fraction between the edges connected to CN of degree d and all edges in the graph. d_v and d_c represent maximum VN and CN degree respectively.

In a Tanner Graph an important parameter to consider is the *cycle*, which consists in a path comprising v edges and that closes back on itself. The number of edges included in the path is the length of the cycle. The minimum cycle length is called *girth* of the graph and it is denoted

by γ. Generally small cycles have to be avoided because they degrade the performance of the iterative decoding algorithm that is used for LDPC codes (8).

1.3.1 LDPC Encoding

Given a low-density parity-check matrix H we can encode a message $\mathbf{u}$ by finding the generator matrix G. In order to do so we need to put H in the systematic form $H = [A|I_{\mathrm{n-k}}]$ by applying elementary row and column operations. Once we get H in the desired form we can obtain G as we have already seen in 1.2.1. The codewords will be obtained by

$$\mathbf{c} = \mathbf{u}G$$

The main drawback of this method is that G usually will not be so sparse as H and so the previous matrix multiplication will produce a lot of operations.

The encoded message is then sent through the channel and when it gets to the receiver it needs to be decoded in order to recover the original sequence of bits. In Chapter 2 several kinds of decoding algorithms will be discussed and compared.

CHAPTER 2

LDPC DECODING

Besides the encoding and the transmission of a message, another important part in a properly working communication system (Figure 5) is the decoding process. As already said, decoding a codeword means to recover the original message $\mathbf{u}$ from the received sequence $\mathbf{y}$. Suppose a codeword $\mathbf{c}$ has to be sent using a Bipolar Phase Shift Keying (BPSK) modulation, that is as a sequence of bipolar symbols $\mathbf{x} = (x_1 x_2 \ \ldots x_n)$, where

$$x_i = A(2c_i - 1)$$

and A is the amplitude of the pulses. After the channel the received block turns out to be $\mathbf{y} = (y_1 y_2 \ \ldots y_n)$ where each y_i is defined as

$$y_i = x_i + g_i$$

and g_i is the channel noise introduced by the Additive White Gaussian Noise (AWGN) channel model. So each symbol y_i is different from its correspondent symbol x_i and its value strongly depends on the gaussian noise. $\mathbf{y}$ is then turned into a binary sequence $\mathbf{r}$ by the hard-decision block, which converts positive values of $\mathbf{y}$ into 1s and negative values into 0s. With no added Gaussian Noise in the channel $\mathbf{r}$ would be the same as the original codeword $\mathbf{c}$, but most of

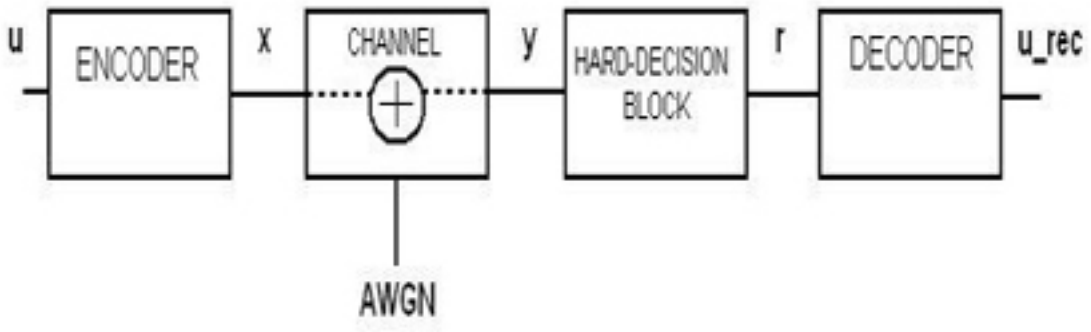

Figure 5. Communication system scheme: u is the original message; x is the codeword sent as a bipolar signal through the channel; y is the received sequence of symbols after the channel; r is the corrispondent binary message and *u_rec* is the recovered codeword

the times, since the noise is present, this is not true and that is why a forward correction algorithm is usually needed to recover the correct message. Being the main topic of this thesis about LDPC codes decoding, different decoding architectures and decoding algorithms will be discussed in the following.

2.1 Decoding Architectures

According to which architecture they use and to how they are implemented in hardware, the decoders can be categorised in **serial**, **parallel** and **partially parallel** decoders.

A *serial decoder* is the most simple to implement in hardware, because it usually only consists of a single Check Node Unit (CNU), a Variable Node Unit (VNU) and a memory (Figure 6). First the VNU updates the variable nodes, one at a time, and it stores the

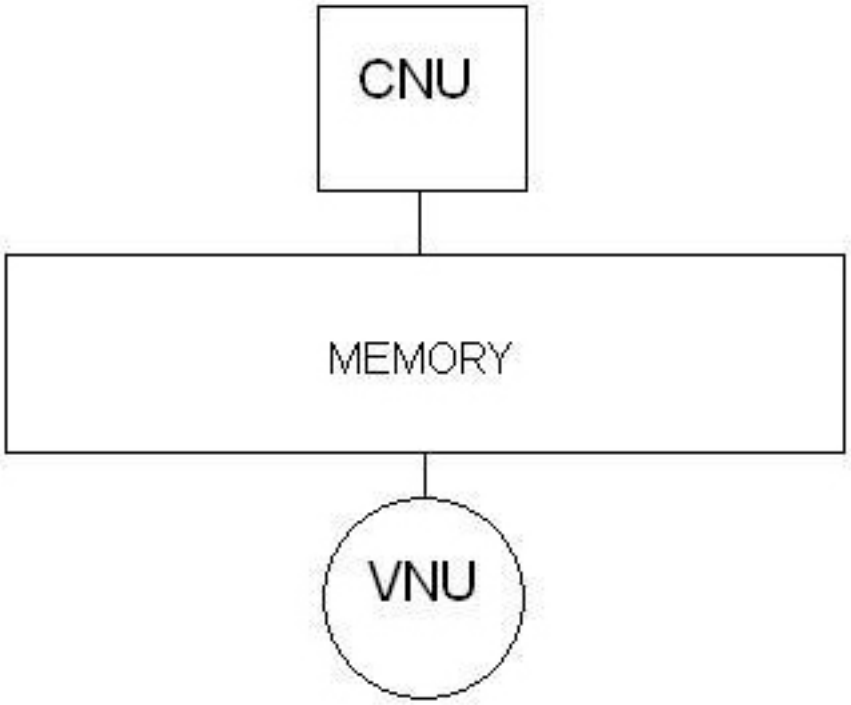

Figure 6. Serial decoder architecture

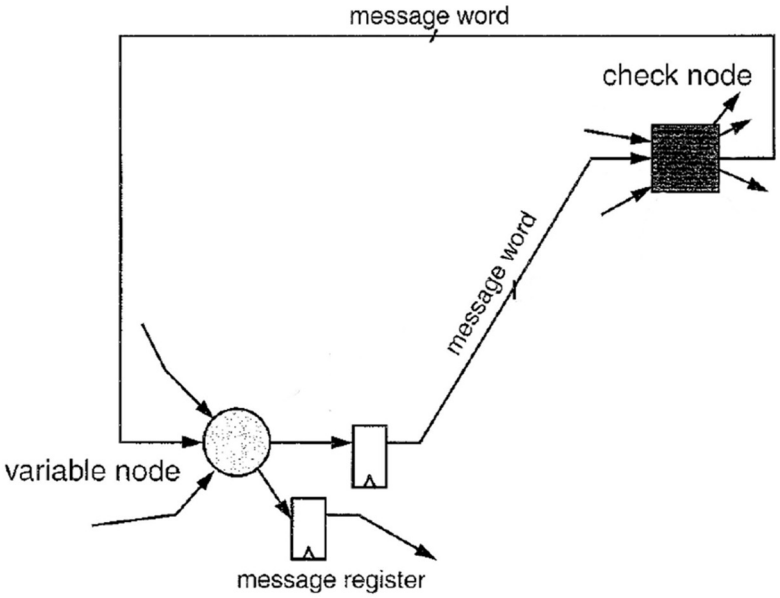

Figure 7. Parallel decoder architecture

variable nodes values to the memory; then the CNU updates the check nodes and stores the corresponding values to the memory as well. Since it all happens sequentially the serial decoder turns out to be very slow for real systems. On the other hand its main advantage is that it is extremely flexible and it can support several codes, with different code rates and block sizes.

A *parallel decoder* is very complex to implement in hardware, because for each check node in the Tanner Graph there is a CNU, and for each variable node there is a VNU (Figure 7). This means that in case of huge H matrixes with thousands of entries, the decoder needs very many CNUs and VNUs, and so its architecture turns out to be very complex. The main advantage

of a parallel architecture though is that it best exploits the parallelism of the message-passing algorithms used to decode LDPC codes. In fact basically all the variable nodes are updated at the same time, in only one clock cycle or in a few of them, and all the check nodes in the following one. This allows the parallel architecture to reach the highest speed possible among all types of decoders.

A *partially parallel decoder* represents a trade-off between complexity and speed. In this case there are multiple CNUs and VNUs which sequentially update the values of multiple variable and check nodes. This approach implies that many variables nodes share the same VNU and many check nodes share the same CNU, as in serial architecture, but it also allows to exploit the paralllism, since multiple CNUs and VNUs run in parallel. This kind of decoder is usually preferible to the *fully parallel* because it is more flexible and supports different codes. The main issue is about memory collisions, since in the same clock cycles more units could try to access the same memory, causing a collision.

2.2 Decoding Algorithms

The algorithms used to decode LDPC codes are called *message-passing* or *iterative-decoding* algorithms because messages go back and forward between check nodes and bit nodes. These messages can be binary numbers or probabilities values; in case they are binary numbers we have a hard-decision algorithm, otherwise in case of probability values we have a soft-decision algorithm. Both types of algorithms will be analysed and discussed in the following and in particular the implementation of three different algorithms will be shown.

2.2.1 Bit-flipping Decoding

The bit-flipping decoding is a common hard-decision algorithm in which all the messages are 0s or 1s. It totally relies on the computation of the syndrome **s**. Considering a parity-check matrix H of size (m,n), and a received block **r** of size (1, n), the syndrome is calculated as

$$\mathbf{s} = H\mathbf{r}^{\mathsf{T}}$$

If the syndrome is 0 then all the parity-check equations are satisfied and so the received bit sequence is a valid codeword. If not, the algorithm takes into account all the symbols of $\mathbf{r} = (r_1, r_2, \ldots r_n)$ which are involved in more failed parity-check equations, or in other words, all the variable nodes connected to check nodes which correspond to the 1s in the syndrome vector. The bit in **r** involved in the greatest number of failed equations is flipped (negated). After each iteration a new bit is flipped, until a valid codeword is found or a maximum number of iterations has been reached.

2.2.2 Weighted Bit-flipping Decoding

The Weighted bit-flipping algorithm is defined as a partial soft-decision algorithm. In order to fully understand its decoding criterion we need to understand the meaning of the received block $\mathbf{y} = (y_1 y_2 \ldots y_n)$ after the channel. As already said $y_i = x_i + g_i$, so the amplitude of y_i totally depends on the noise g_i and the greater the amplitude $|y_i|$, the higher the reliability of the hard-decision symbol r_i. Let us now denote the set of all the r_i bits which are involved in the computation of the syndrome bit s_j as *N(j)*, that is, *N(j)* is the set of all the entries of *H*

matrix in j^{th} row. In the same way we can define the set *M(i)* of all parity checks in which r_i is involved, or in other words the set of all the entries of H in the i^{th} column (9, p.208). At this point we can compute the reliability of the syndrome components; we know that each element s_j of the syndrome $\mathbf{s}$ is obtained multiplying one row of H by the vector $\mathbf{r}$ after the decision block. In the computation of each syndrome component s_j different elements of $\mathbf{r}$ are used, according to the entries of H matrix, or more precisely to the set *N(j)*. Since the reliability of each r_i depends on the corresponding y_i we can consider the reliability of s_j as the lowest magnitude of the received vector $\mathbf{y}$. In formula we can write

$$|y|^j_{min} = \min_{i:i\in N(j)} |y_i| \text{ with } j = 1, 2, ..., m$$

Once all $|y|^j_{min}$ have been calculated, for each r_i bit we compute

$$E_i = \sum_{j\in M(i)} (2s_j - 1)|y|^j_{min} \text{ with } i = 1, 2, ..., n$$

and the bit r_i corresponding to the greatest E_i is then flipped. This procedure is repeated until a maximum number of iterations has been reached or a valid codeword is found, that is, the obtained $\mathbf{r}$ satisfies all the parity-check equations.

2.2.3 Sum-product Decoding

The sum-product algorithm is a soft decision algorithm, which means that all the messages passed between check and bit nodes are now probabilies and not bits. The input bit probabili-

ties are called *a priori* probabilities, which means that they are known in advance and do not depend on the decoding process. The probabilities (messages) returned by the check nodes, and also all the probabilities passed between nodes, are called *a posteriori* probabilities. All probability values are expressed as *log-likelihood ratios*: given a variable x and the probabilities $p(x = 0)$ and $p(x = 1)$ we call *log-likelihood ratio* (LLR) the value

$$\mathrm{L}(x) = \log\left(\frac{p(x=0)}{p(x=1)}\right)$$

As we can see the sign of $\mathrm{L}(x)$ tells us which one between $p(x = 0)$ and $p(x = 1)$ is larger, and the magnitude of $\mathrm{L}(x)$ tells us the difference between $p(x = 0)$ and $p(x = 1)$, and so the reliability of the decision. The reason why the LLR is preferable to the normal probability values is that in the logarithm domain all the products become summations, and these latter are easier to implement in hardware.

Before talking about how this algorithm works it is necessary to introduce the concept of *extrinsic* information. Let us suppose to have a system with many processors which exchange information between them and consider Figure 8 (see example in (7, p.210)). In the figure many soldiers are depicted, each of them can be thought of as a different processor. Now suppose that the goal of each soldier is to know how many soldiers there are in total. Each of them sends a message to its neighbors and receives messages from all of them. As we can see from Figure 8 the number that a soldier A passes to another soldier B is the sum of all incoming messages to soldier A (processor) plus the information about the soldier A itself, minus the message that

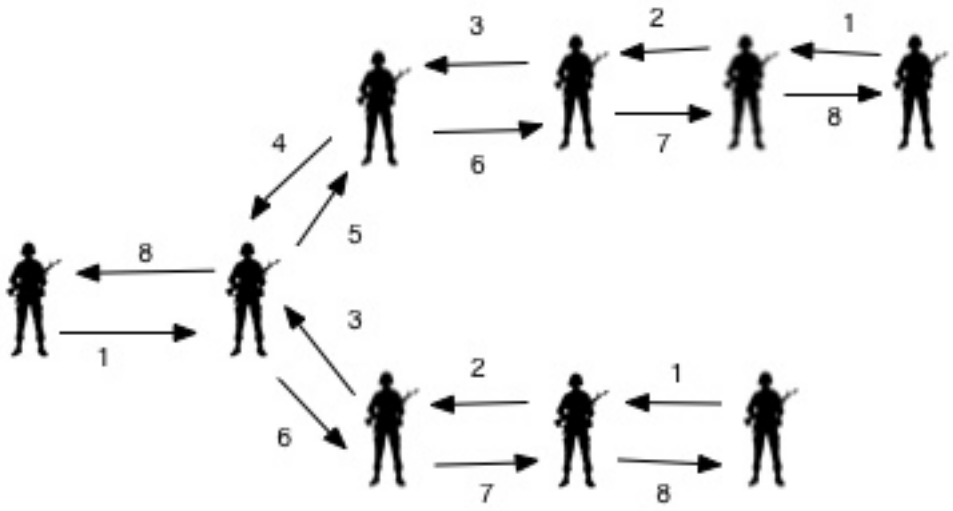

Figure 8. Soldiers distribuited as processors in a system, exchanging extrinsic information between them

soldier B had sent to soldier A. This comes from the fact that a soldier does not pass to its neighbor the information that this latter already has, so it passes only *extrinsic* information. This concept will be useful in the following. Note that the soldiers are disposed in such a way that no cycle (loop) is present, otherwise this kind of counting would not be possible because of a positive feedback. This explains why the message passing on a graph is considered optimal only if no cycles are present and also why 4-cycle free H matrixes a preferable.

Now to describe the behaviour of the Sum-product decoder we need to treat Bit nodes and Check nodes as two different kinds of processors (decoders), each of which decodes a different kind of code. In particular Bit nodes can be considered as Repetition decoders, which means that they receive messages from the channel and from the Check nodes and all these messages

are seen as symbols of a repetition code. Check nodes are instead considered as Single Parity Check Decoders, where all the incoming messages from the Bit nodes are checked to satisfy the parity check equation for each Check node.

Both the decoder types are MAP decoders, so for each codeword bit in the transmitted codeword $\mathbf{c} = \{c_1, c_2, \ldots c_n\}$ we want to find the *a posteriori* probability that it equals 1, given the received codeword $\mathbf{y} = \{y_1, y_2, \ldots y_n\}$, (see (7, p.213)). Considering only one codeword bit v_j and expressing the probability as LLR, we have

$$L(c_j|\mathbf{c}) = \log\left(\frac{p(c_j=0|\mathbf{y})}{p(c_j=1|\mathbf{y})}\right)$$

where, for an AWGN channel model, $p(c_j = b|y_j)$ can be expressed as $(1 + e^{\frac{-2by_i}{N_0}})^{-1}$ and N_0 is the variance (see (10)).

First of all let's consider a *Repetition Code* in which a binary symbol $c \in \{0, 1\}$ is transmitted over a channel k times, so that a vector $\mathbf{r}$ of size k arrives to the receiver. The MAP decoder computes the following

$$L(c|\mathbf{r}) = \log\left(\frac{p(c=0|\mathbf{r})}{p(c=1|\mathbf{r})}\right)$$

If the events $c = 0$ and $c = 1$ are equally likely, then we have $p(c|\mathbf{r}) = p(\mathbf{r}|c)$ and so we can write

$$\mathrm{L}(c|\mathbf{r}) = \log\left(\frac{p(\mathbf{r}|c=0)}{p(\mathbf{r}|c=1)}\right)$$

which in turn can be written as

$$\mathrm{L}(c|\mathbf{r}) = \log\left(\frac{\prod_{h=0}^{k-1} p(r_h|c=0)}{\prod_{h=0}^{k-1} p(r_h|c=1)}\right)$$

Since we are dealing with log functions we can substitute the previous equation with

$$\mathrm{L}(c|\mathbf{r}) = \sum_{h=0}^{k-1} \log\left(\frac{p(r_h|c=0)}{p(r_h|c=1)}\right)$$

$$= \sum_{h=0}^{k-1} \mathrm{L}(r_h|c)$$

The MAP decoder for a *Repetition Code* computes all LLRs for each r_h and adds them; the receiver decides $\hat{c} = 0$ if $\mathrm{L}(c|\mathbf{r}) \geq 0$ (which means that $c = 0$ is a more likely event), $\hat{c} = 1$ otherwise.

Now we can adapt the previous expression in case of LDPC decoding, to compute *the information* to be sent from VN j to CN i, that is

$$\mathrm{L}_{j\to i} = \mathrm{L}_j + \sum_{i' \in N(j)\setminus i} \mathrm{L}_{i'\to j}$$

As we can see L_j is the value computed from the channel sample y_j

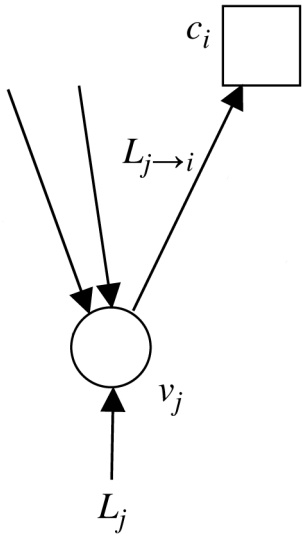

Figure 9. VN j receives the information from the channel and from all CNs except CN i and sends the result back to CN i

$$\mathsf{L_j} = \mathsf{L}(c_j|y_j) = \log\left(\frac{p(c_j=0|y_j)}{p(c_j=1|y_j)}\right)$$

and $\sum_{i'\in \mathsf{N}(j)\setminus i} \mathsf{L}_{i'\to j}$ is what comes from $\mathsf{L}(c|\mathbf{r})$ and takes into account all the values from the CNs connected to VN j except the one from CN i (only extrinsic information) (see Figure 9). The sum of all these probability values is finally sent to CN i.

Consider now the following **Lemma** (see (7, p.217)):

In a vector of d independent binary random variables $\boldsymbol{w} = \{w_0, w_1, \ldots w_{d-1}\}$, *where* $p(w_h = 1)$

is the probability that w_h is 1, the probability that $\boldsymbol{w}$ contains an even number of 1s is

$$\tfrac{1}{2} + \tfrac{1}{2}\prod_{h=0}^{d-1}(1 - 2p(w_h = 1))$$

while the probability that $\boldsymbol{w}$ contains an odd number of 1s is

$$\tfrac{1}{2} - \tfrac{1}{2}\prod_{h=0}^{d-1}(1 - 2p(w_h = 1))$$

This result is very useful to analyze the behavior of a CN. Suppose we are transmitting a d-length codeword over a memoryless channel and suppose the output of this channel is a vector $\mathbf{r}$. We are using a MAP decoder for *Single Parity Check Codes*. In order to solve a parity check equation we can impose a constraint over the codeword, that is, there must be an even number of 1s. Let us consider for now only one symbol of the codeword, say c_0. The MAP decision rule is

$$\hat{c_0} = \underset{b \in \{0,1\}}{\operatorname{argmax}}\ p(c_0 = b|\mathbf{r})$$

that is, c_0 is equal to the argument b which maximizes $p(c_0 = b|\mathbf{r})$. We could write

$$p(c_0 = 0|\mathbf{r}) = p\{c_1, c_2, \ldots c_{d-1} \text{ have an even number of 1s}|\mathbf{r}\}$$

$$= \frac{1}{2} + \frac{1}{2}\prod_{h=1}^{d-1}(1 - 2p(c_h = 1|r_h))$$

But we know that $p(c_0 = 0|\mathbf{r}) = 1 - p(c_0 = 1|\mathbf{r})$ so substituting and rearranging we have

$$1 - 2p(c_0 = 1|\mathbf{r}) = \prod_{h=1}^{d-1}(1 - 2p(c_h = 1|r_h))$$

What we want to do now is to express the probability as LLR, and for this reason we use the relation (not proved here) for a binary random variable with probabilities p_1 and p_0

$$1 - 2p_1 = \tanh\left(\frac{1}{2}\log\left(\frac{p_0}{p_1}\right)\right) = \tanh\left(\frac{1}{2}LLR\right)$$

Substituting in the previous equation we have

$$\tanh\left(\frac{1}{2}L(c_0|\mathbf{r})\right) = \prod_{h=1}^{d-1}\tanh\left(\frac{1}{2}L(c_h|r_h)\right)$$

from which we finally derive

$$L(c_0|\mathbf{r}) = 2\tanh^{-1}\left(\prod_{h=1}^{d-1}\tanh(\frac{1}{2}L(c_h|r_h)\right)$$

Also in this case the decoder decides $\hat{c_0} = 0$ if $L(c_0|\mathbf{r}) \geq 0$, $\hat{c_0} = 1$ otherwise.

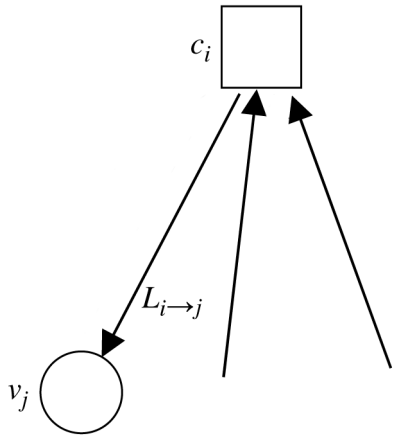

Figure 10. CN i receives the information from all VNs except VN j and sends the result back to VN j

In case of LDPC decoding we can adapt the previous expression to estimate *the information* sent from CN i to VN j, that is

$$\mathrm{L}_{i\to j} = 2\tanh^{-1}\left(\prod_{j'\in \mathrm{N}(i)\setminus j} \tanh\left(\tfrac{1}{2}\mathrm{L}_{j'\to i}\right)\right)$$

where j' takes into acount all the messages from VNs connected to CN i except the one from VN j (only extrinsic information) (Figure 10).

Generally we can say that a Sum-Product Algorithm decoder is mainly based on the constraints of *Single Parity Check Code* for CN messages and of *Repetition Code* for VN messages. What is still missing is an initialization step and a stopping criterion. To initialize the decoder we can set all VN messages $\mathrm{L}_{j\to i}$ to L_J that is

$$\mathsf{L}_j = \log\left(\frac{p(c_j=0|y_j)}{p(c_j=1|y_j)}\right)$$

while the stopping criterion most used is to stop the process when $\hat{\mathbf{c}}H^{\mathsf{T}} = 0$, where $\hat{\mathbf{c}}$ is the decoded codeword, or when a maximum number of iterations has been reached. The Sum-Product Algorithm can be summarized as follows:

1. **Initialization**: At the beginning we initialize all L_j according to

$$\mathsf{L}_j = \log\left(\frac{p(c_j=0|y_j)}{p(c_j=1|y_j)}\right)$$

   and set $\mathsf{L}_{j\to i} = \mathsf{L}_j$ for all j and i for which $h_{ij} = 1$

2. **Compute CN messages**: now we compute $\mathsf{L}_{i\to j}$ for all CNs as we have seen

$$\mathsf{L}_{i\to j} = 2\tanh^{-1}\left(\prod_{j'\in N(i)\setminus j}\tanh\left(\tfrac{1}{2}\mathsf{L}_{j'\to i}\right)\right)$$

   and messages are transmitted to VNs.

3. **Compute VN messages**: now we compute $\mathsf{L}_{j\to i}$ for all VNs as we have seen

$$\mathsf{L}_{j\to i} = \mathsf{L}_j + \sum_{i'\in N(j)\setminus i}\mathsf{L}_{i'\to j}$$

   and then send the messages back to CNs.

4. **Compute total LLR**: for all VNs we compute the total $\mathsf{L}_j^{\text{tot}}$ in which for each VN we consider all the incoming messages from all the CNs connected to it

$$\mathsf{L}_j^{\text{tot}} = \mathsf{L}_j + \sum_{i \in \mathsf{N}(j)} \mathsf{L}_{i \to j}$$

5. **Stopping Criteria**: We estimate the codeword symbols c_{j} for every j

$$\widehat{c_{\mathsf{j}}} = \begin{cases} 1, & \text{if } \mathsf{L}_j^{\text{tot}} < 0 \\ 0, & \text{else} \end{cases}$$

in such a way we finally obtain the recovered codeword $\hat{\mathbf{c}}$. If $\hat{\mathbf{c}} H^{\mathsf{T}} = 0$ then stop, otherwise we need to go back to step 2.

CHAPTER 3

VHDL IMPLEMENTATION OF LDPC DECODER

As already seen in Chapter 2, a parallel LDPC decoder presents the most complex architecture among all possible decoder architectures, because essentially for each bit node and check node there are independent dedicated portions of hardware. In this thesis a fixed hardware implementation of a Bit-Flipping decoder is presented, based on the VHDL (which stands for **V**HSIC **H**ardware **D**escription **L**anguage) used to describe an FPGA from the family Cyclone II, device EP2C35F672C6, by Altera (see (11)). The "fixed" implementation refers to the fact that it is valid only for a given Parity Check matrix and so only for a given code. To decode a different code a different architecture should be implemented. The (4x6) Parity Check matrix H taken into account describes a regular block code, which means it has a fixed column weight $w_c = 2$ and a fixed row weight $w_r = 3$; the H matrix is shown below:

$$H = \begin{bmatrix} 1\,0\,1\,1\,0\,0 \\ 0\,1\,1\,0\,1\,0 \\ 1\,0\,0\,0\,1\,1 \\ 0\,1\,0\,1\,0\,1 \end{bmatrix}$$

As we can see it is 4-cycle free and it describes a (6,3) block code, in fact the number of linearly independent rows is three, which means that in the matrix there is one row dependent on the others. The corresponding Tanner Graph is shown in Figure 11.

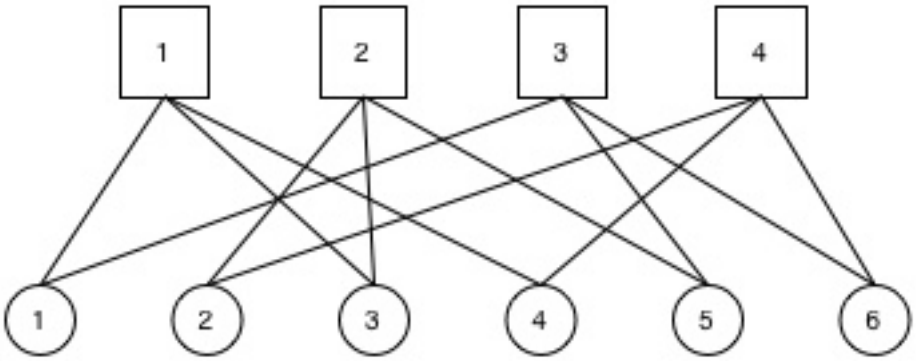

Figure 11. Tanner Graph of the Parity check matrix implemented in the decoder

3.1 Bit-Flipping Decoding Implementation

As already stated in Chapter 2 one of the most common Hard-Decision decoding algorithms is the Bit-Flipping one and in this thesis an implementation of a decoder using such an algorithm is provided. The idea behind this implementation relies on the fact that the H matrix is already known and it cannot change (it is fixed). Considering that we are talking about the Bit-Flipping algorithm the goal is to "flip" the value of the bit node (and so the incoming codeword bit) which is connected to the greatest number of Check nodes where the parity check equation is not satisfied. From the previous H matrix and from its Tanner Graph it results that there are four Check nodes and six Bit nodes, and each Check node is connected to three Bit nodes, while each Bit node is connected to two Check nodes (regular code $w_r = 3$, $w_c = 2$). The decoder implementation presented in the following consists in six Blocks which perform different tasks and nine different registers needed to store the intermediate results. Since registers are present

it is clear that the implemented circuit is a *sequential logic* circuit in which the outputs don't depend only on the present inputs but also on their history. The most common tool to describe a sequential logic circuit is a Finite State Machine (FSM), an abstract machine that can be in only one state at a time among a finite number of states and it can change state in correspondence of a triggering event or a condition. The behaviour of the whole machine is described in the Control Unit, that is one of the architecture Blocks. In the following all the Blocks are described in details.

3.1.1 Syndrome Block

As already seen the first thing the Bit-flipping algorithm does is to compute the syndrome **s** and if **s** is a zero vector then the algorithm stops, otherwise it looks for the bit to flip in the codeword. Computing the syndrome actually means calculating a product between the matrix H and the received sequence of bits after the channel and verifying that the result is all zeros vector. In this hardware implementation the syndrome is calculated by verifying that all the parity check equations are satisfied: for each matrix row (that is for each check node) a **xor** operation is made between all the codeword bits in the same positions of the ones in the row (or the codeword bits connected to the same check node). Since there are four Check nodes, four different equations are computed and the results are put together in an **or** equation. A 0 as final result means that a valid codeword has been found and the algorithm stops, a 1 means that not all parity check equations are satisfied (there is at least a wrong one) and so the algorithm goes on. The equations are all computed by means of a component called **Syndrome Block** in which there is a 12-bit input and a 1-bit output. The input comes from a MEMORY13 and

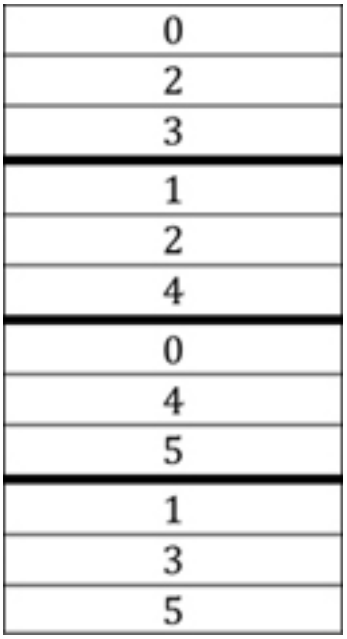

Figure 12. The 0 to 5 bits entering the Syndrome block are phisically connected to the memory in the way described in the picture, according to the H matrix

it is composed of the six codeword bits, each of them repeated twice, and all of them physically connected to the register slots in such an order that four groups of three bits each are formed, and in each group the bit nodes connected to a different check node are present. By numbering the codeword bits from 0 to 5 we can see the order in which they are sorted in MEMORY13 in Figure 12. This means that the first group contains codeword bits number 0, 2 and 3, that is, Bit nodes number 0, 2 and 3 are connected to the first Check node, and so on. The circuit implementation of the Syndrome Block is shown in Figure 13.

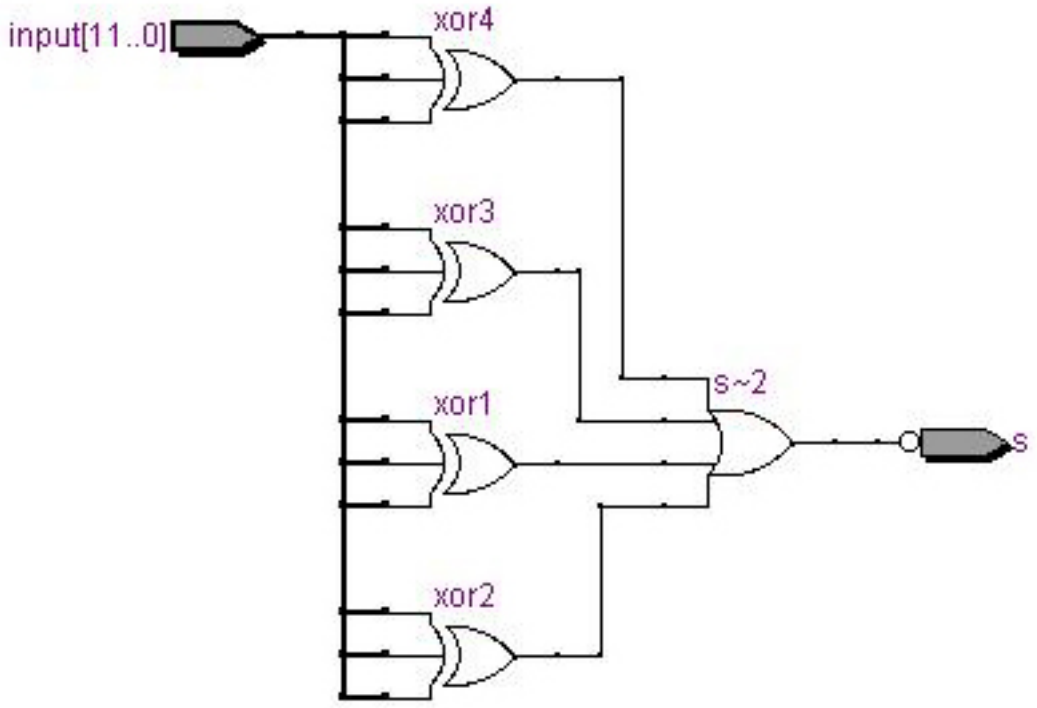

Figure 13. Implementation of the Syndrome block

3.1.2 Check-nodes Block

In case the Syndrome **s** is not a zero vector then the algorithm goes on and as already said it looks for the bit to flip. So the next step is the computation of the four parity check equations and it is performed by means of the **Check-node Block**, which is a simple circuit with a 12-bit input and a 12-bit output. The input bits come from MEMORY11 and they are the same as the ones of the Syndrome Block, and for each input bit the corresponding output bit is computed making a **xor** operation between the other two bits of the same group. In other words, since in each group there are three bits coming from the Bit-nodes (and so from the channel), three different **xor** operations will be performed and at each bit will be assigned the result coming from the **xor** operation between the other two in that group (see Figure 14). All the results are then stored into MEMORY2 and if one of the assigned bits resulting from the **xor** equation is equal to the corresponding one in MEMORY11 then the parity check equation is satisfied; note that if a parity check eqation is satisfied all the assigned bits in one group in MEMORY2 are equal to the ones in the corresponding group in MEMORY11. The bits in MEMORY2 will be used from the Comparator Block to decide which is the codeword bit connected to the greatest number of Check nodes with a not satisfied parity check equation. The circuit implementation of the Check-nodes Block is shown in Figure 15.

3.1.3 Comparator Block

The **Comparator Block** is one of the most important blocks in this decoder architecture, because it decides which bit has to be flipped. It receives two 12-bit inputs from two memories, the first input is equal to the Syndrome input and the Check-node input and comes from

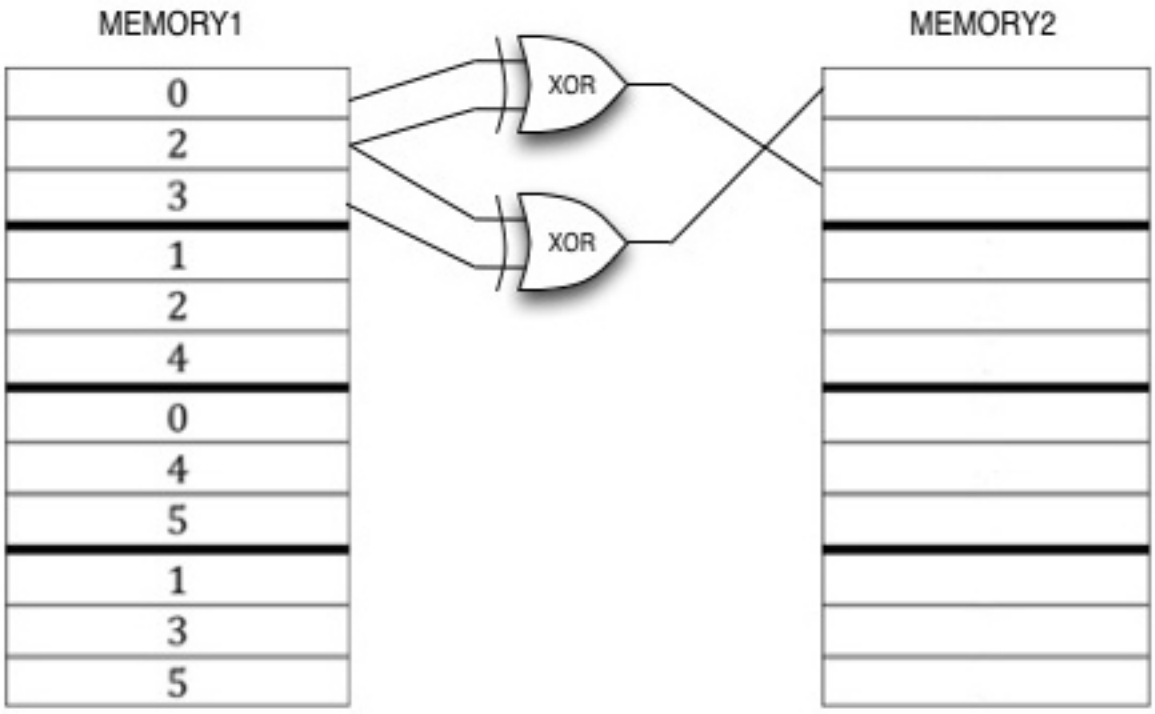

Figure 14. Every couples in each group is used to compute a xor operation and the result is stored in the slot of another memory corresponding to the third bit in the group

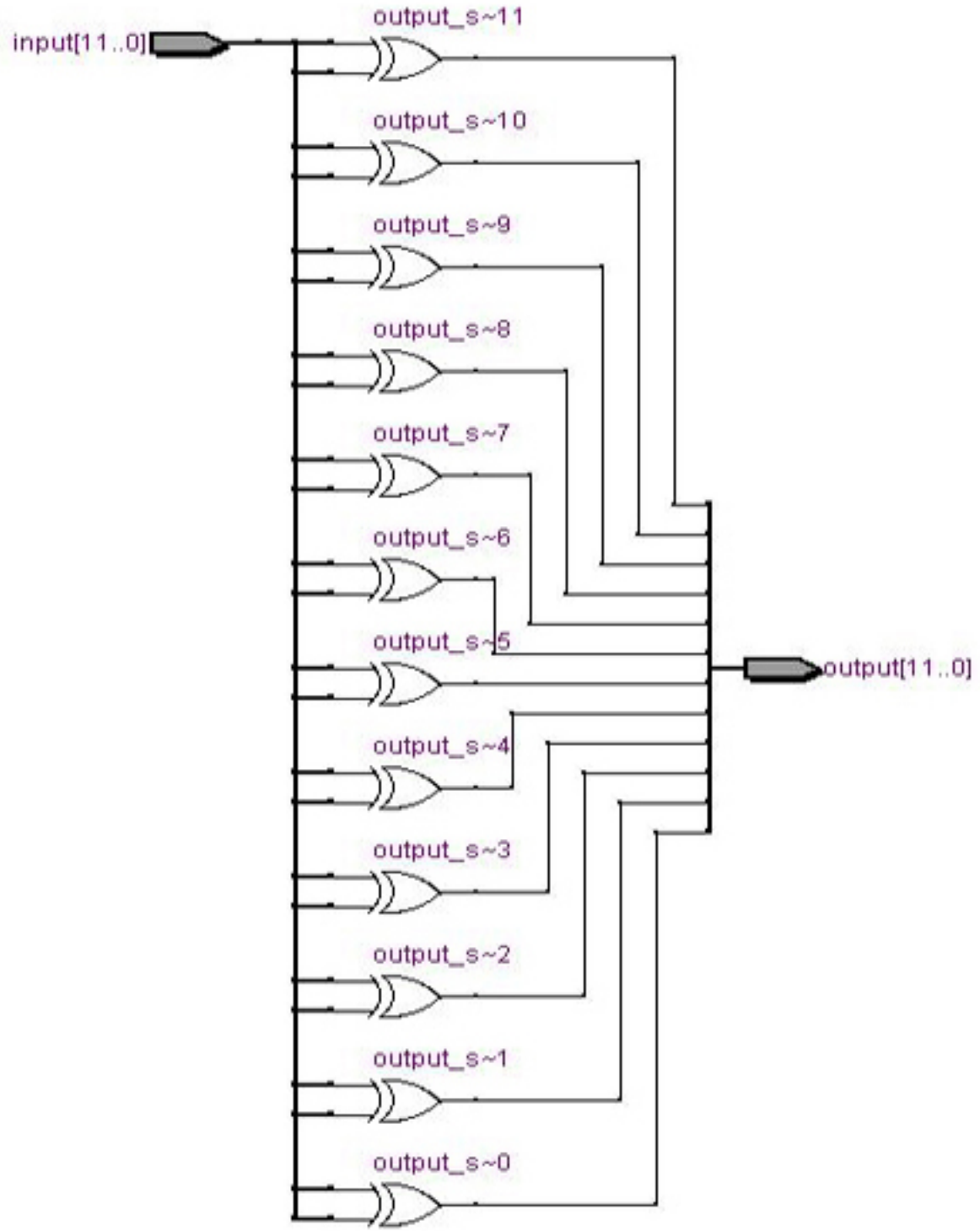

Figure 15. Implementation of the Check-node block

MEMORY12, while the second input is the one coming from the Check-node block that has been stored into MEMORY2. The idea is to use a **xor** operation between the bits in the same positions in the two registers and for each operation if the result is 1 (the two bits are different) then it means that the considered Bit-node is connected to a Check-node with a wrong parity check equation. Since the operations are twelve but the Bit-nodes are six, each single Bit-node is involved in two **xor** operations, according to the fact that each Bit-node is connected to two check-nodes (see Figure 16). The two results related to the same Bit-node are added together by means of an Half-Adder block and then all these summations are compared by means of the **Comp_if** components in order to see which one is the biggest. Each Comp_if component has four inputs and two outputs: two inputs are the results coming from the Half-Adders, while the other two inputs are two 6-bit vectors with all 0s and only one 1, related to the two considered Bit-nodes; for example if we are considering the Bit nodes number 1 and 2, then the four inputs of a Comp_if will be: *nu_er1*, that is the number of failed Check nodes to which Bit-node 1 is connected, *nu_er2*, that is the number of failed Check nodes to which Bit-node 2 is connected and the two vectors "100000" and "010000" in which the position of the 1 depends on the number of the Bit-node. The largest between *nu_er1* and *nu_er2* is output together with the corresponding 6-bit vector. So after the comparison between the numbers of failed Check nodes to which all the Bit-nodes are connected the 6-bit vector corresponding to the codeword bit to be flipped is output and it will be stored into MEMORY3. The implementation of the Comp_if component is showed in Figure 17 while the combinational part of the Comparator Block, without Comp_if components, is shown in Figure 18.

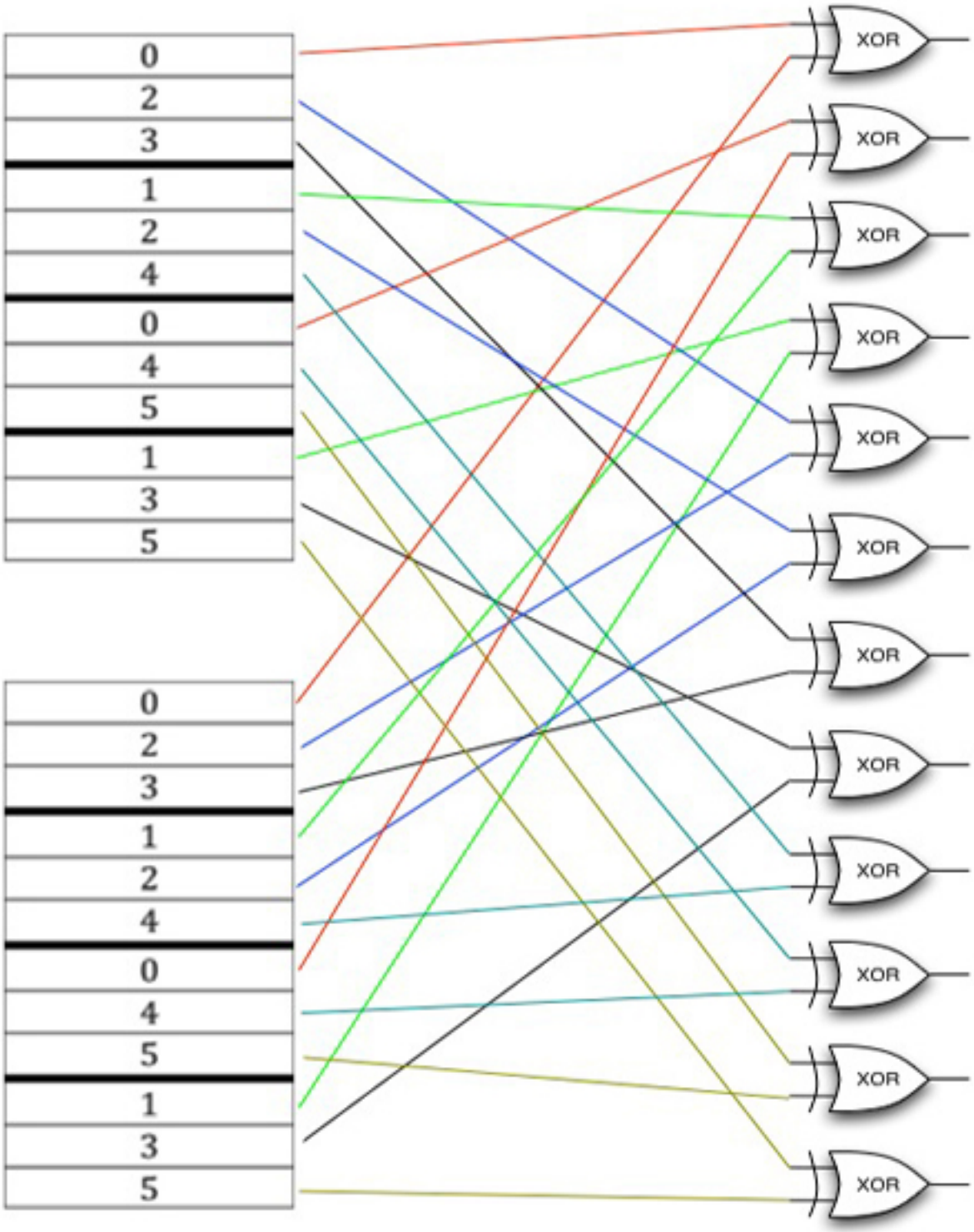

Figure 16. First part of Comparator Block

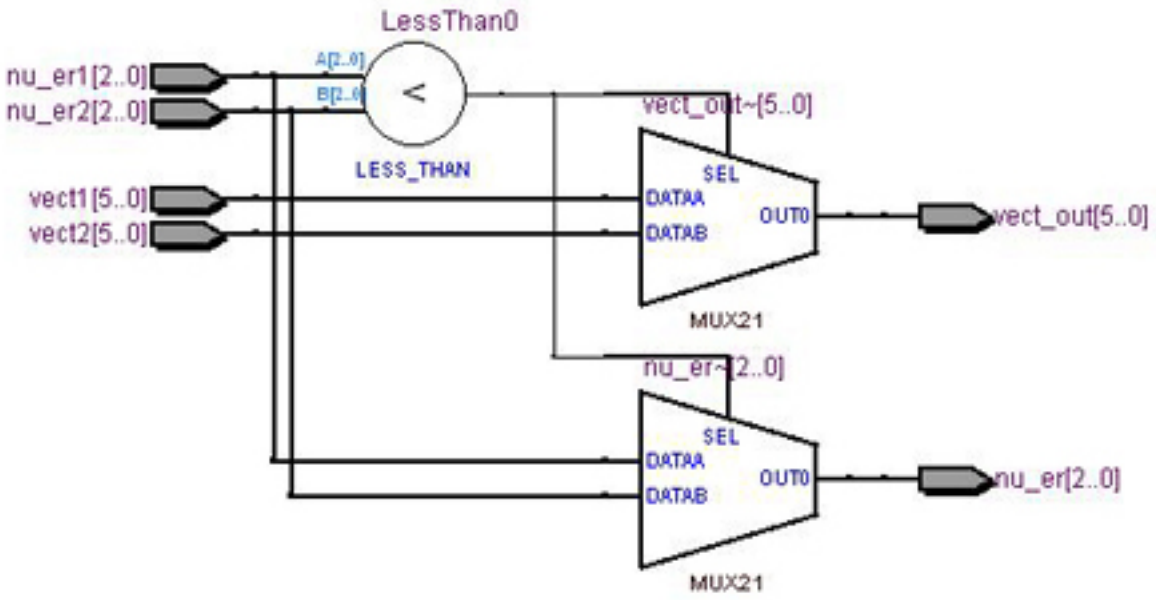

Figure 17. Implementation of the Comp_if component

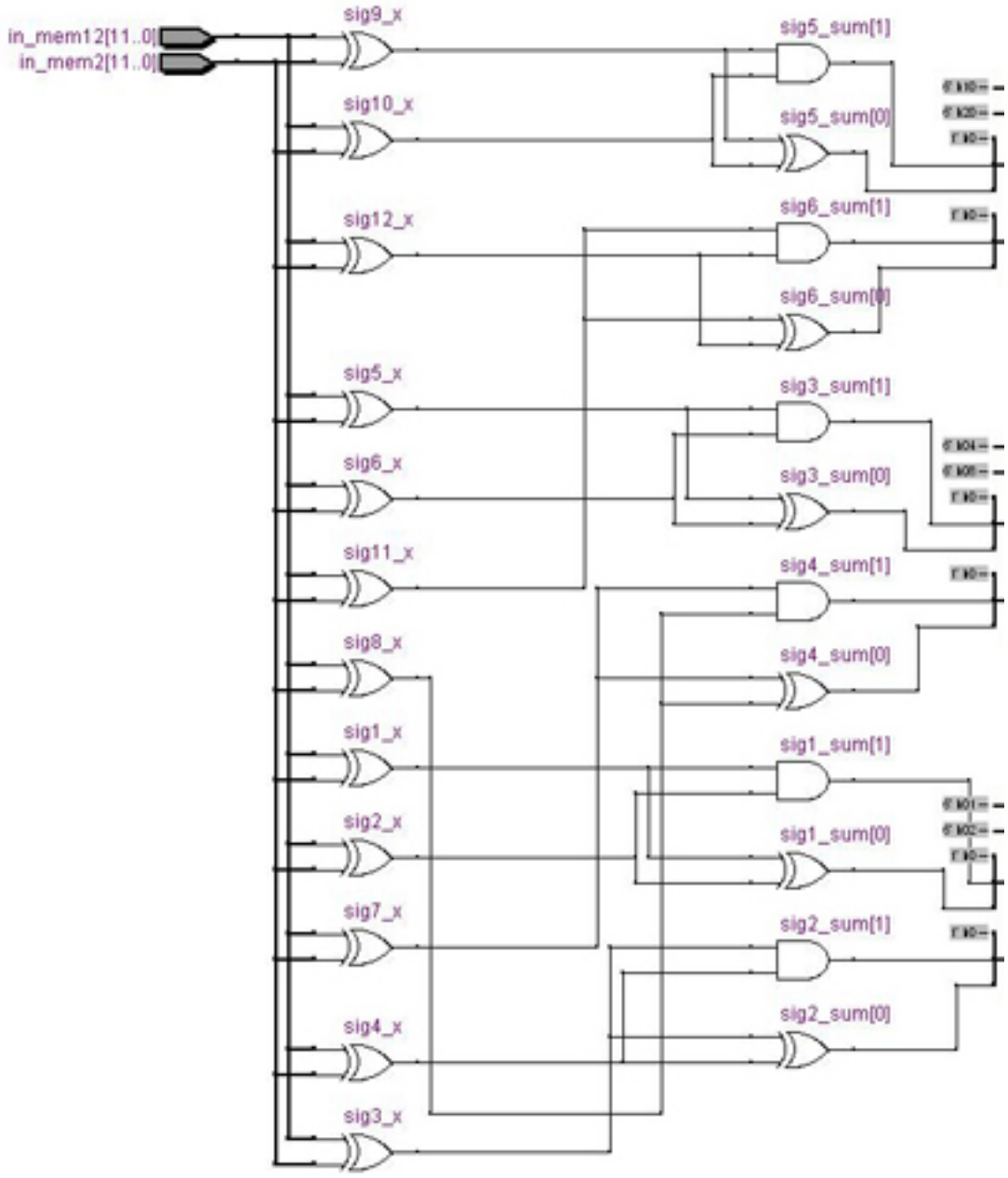

Figure 18. Implementation of the first part of the Comparator block. The outputs on the right are the inputs of three Comp_if components

3.1.4 Bit-nodes Block

If the Comparator Block decides which bit has to be flipped, the effective flipping happens by means of the **Bit-nodes Block**. The latter is a simple block which takes as input two 6-bit vectors, one of which comes from the Comparator Block and so from MEMORY3 and the other one is the received codeword from MEMORY14. Since the output of the Comparator is a vector with only a 1 in the position of the bit in the codeword that has to be flipped, then with a bit-to-bit **xor** between the two input vectors the new codeword is obtained. For instance if the output of the Comparator Block is "100000" then the first bit of the codeword is flipped and a new sequence is found. This new sequence is then stored into MEMORY5, waiting for the Syndrome Block to analyze it again. The implementation of the Bit-nodes Block is shown in Figure 19.

3.1.5 Counter Block

Until the Syndrome **s** is not a zero vector the algorithm goes on flipping new codeword bits and trying to find a valid codeword. In case it is not able to succede after a certain number of iterations it stops. In order to count the iterations a **Counter Block** is used. This latter outputs a high level signal every time it counts 5 and it receives a reset signal so that it restarts counting. The circuit implementation of the Counter Block is shown in Figure 20.

3.1.6 Control Unit Block

The **Control Unit Block** is the most important component of this circuit because it represents the "brain" of the decoder. This is the component in which the FSM is described and the one which controls all the other Blocks. The FSM implemented in this decoder architecture

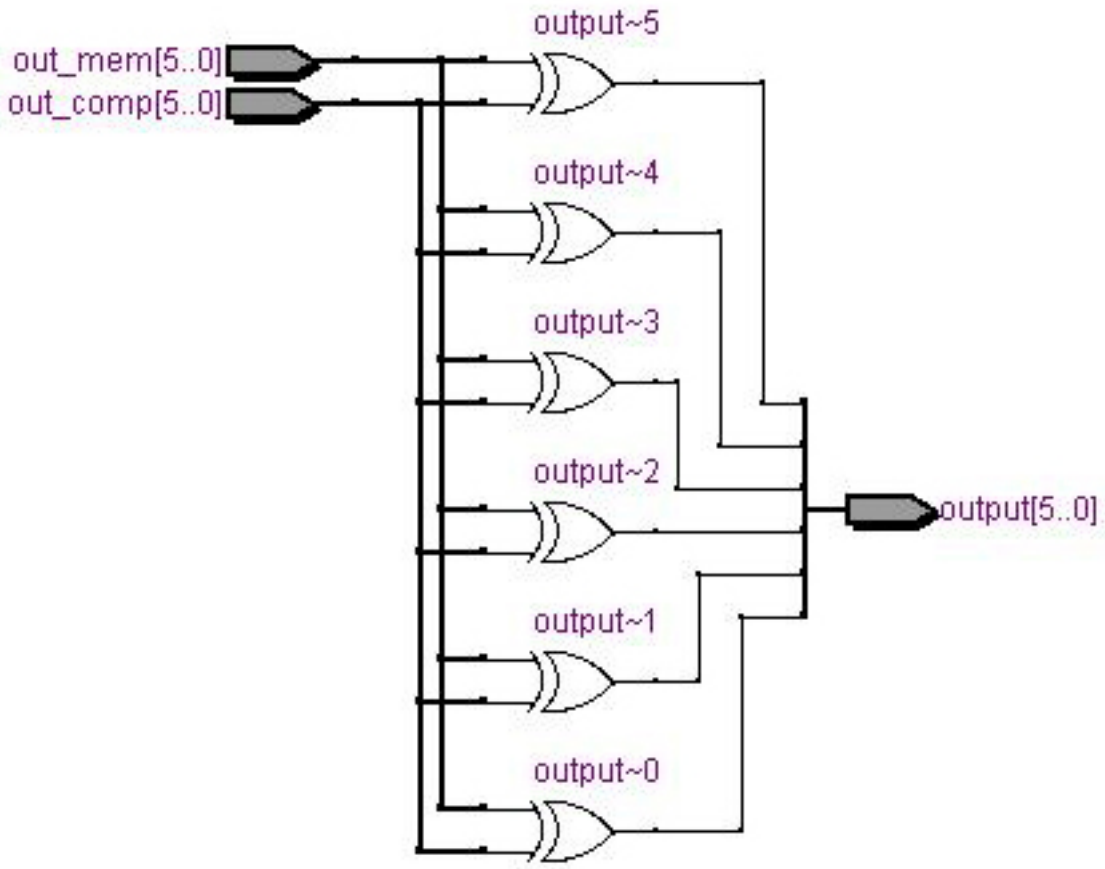

Figure 19. Implementation of the Bit-node Block

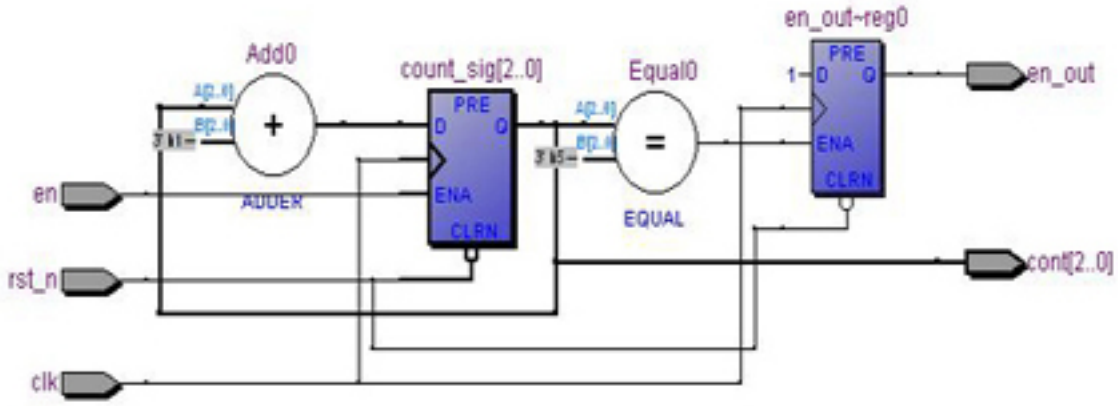

Figure 20. Implementation of the Counter Block

is shown in Figure 21. From the figure we can see that there are nine different possible states and only two of them are conditioned to a particular event, which can be an input from the outside (CODE) or a result coming from some other Blocks in the circuit (SYN_OUT or CONT_OUT). The machine starts from the START state and it remains in there until the input signal CODE becomes '1', that indicates a new codeword is ready to be analyzed. Then the machine goes into the CODE_IN state, in which the codeword is read and stored into a register (MEMORY0). After that there is a transition to the COMPUTE_S state, in which the decoder waits for the MEMORY13 to be acivated such that the Syndrome Block can take its input from this memory. The following state is the DECIDE state, in which the Control Unit checks the SYN_OUT and CONT_OUT inputs and if at least one of them is 1 then the

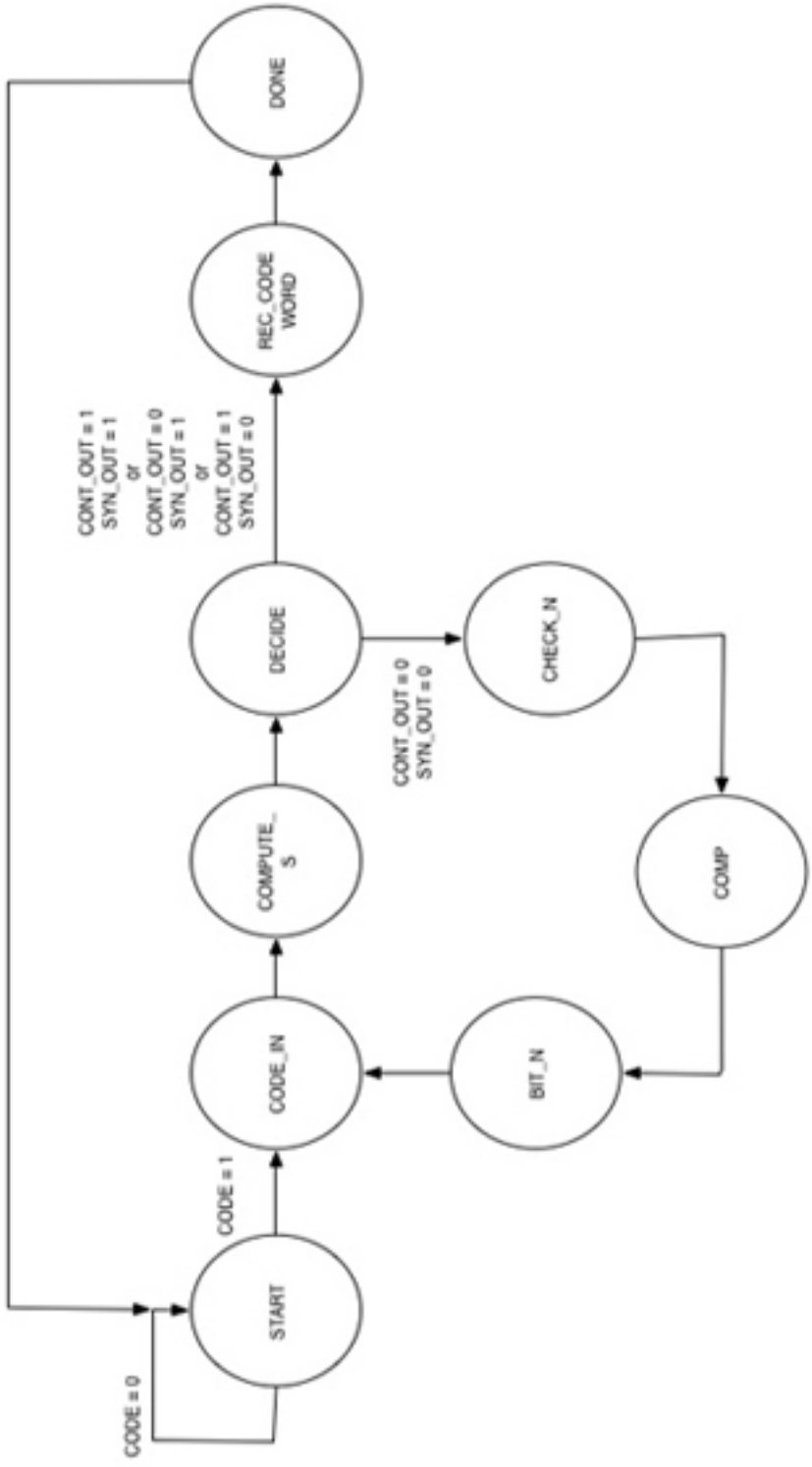

Figure 21. Finite State Machine described in the Control Unit Block

machine goes into the REC_CODEWORD state because a valid codeword has been found or the maximum number of iterations has been reached. The codeword found is then stored into MEMORYF from which it is output. The last state is the DONE state in which all the registers and the counter are reset. If in the DECIDE state both SYN_OUT and CONT_OUT are still 0 then the machine goes into the CHECK_N state (instead of the REC_CODEWORD) in which the Check-nodes Block operates as already seen, taking its input from MEMORY11. Then the machine goes to the COMP state and the BIT_N state, and all the comparisons are made and the bit to be flipped is found. In these two states all the registers MEMORY12, MEMORY3 and MEMORY14 are activated in turn. After the BIT_N state the machine goes back into the CODE_IN state and this time the new codeword, with the flipped bit, is considered by the Syndrome Block, and the algorithm starts again.

The LDPC decoder based on the Bit-flipping algorithm described so far uses a particular architecture, in which all the Check nodes are implemented in a single portion of Hardware and all the Bit nodes in another one. The complete schematic of the whole architecture is depicted in Figure 22. Since all Check nodes and Bit nodes are updated in one clock period, we can say that this architecture is a parallel-like architecture, even if there is not a specific hardware (processor) for each node. Furthermore this is a very simple architecture valid only for a specific H matrix; for parity check matrixes of the same size similar implementations could be thought, with different physical connections between the incoming codeword bits and the registers slots. For bigger size parity check matrixes the same architecture could be used but with bigger registers and different physical connections as well. As this is not a flexible

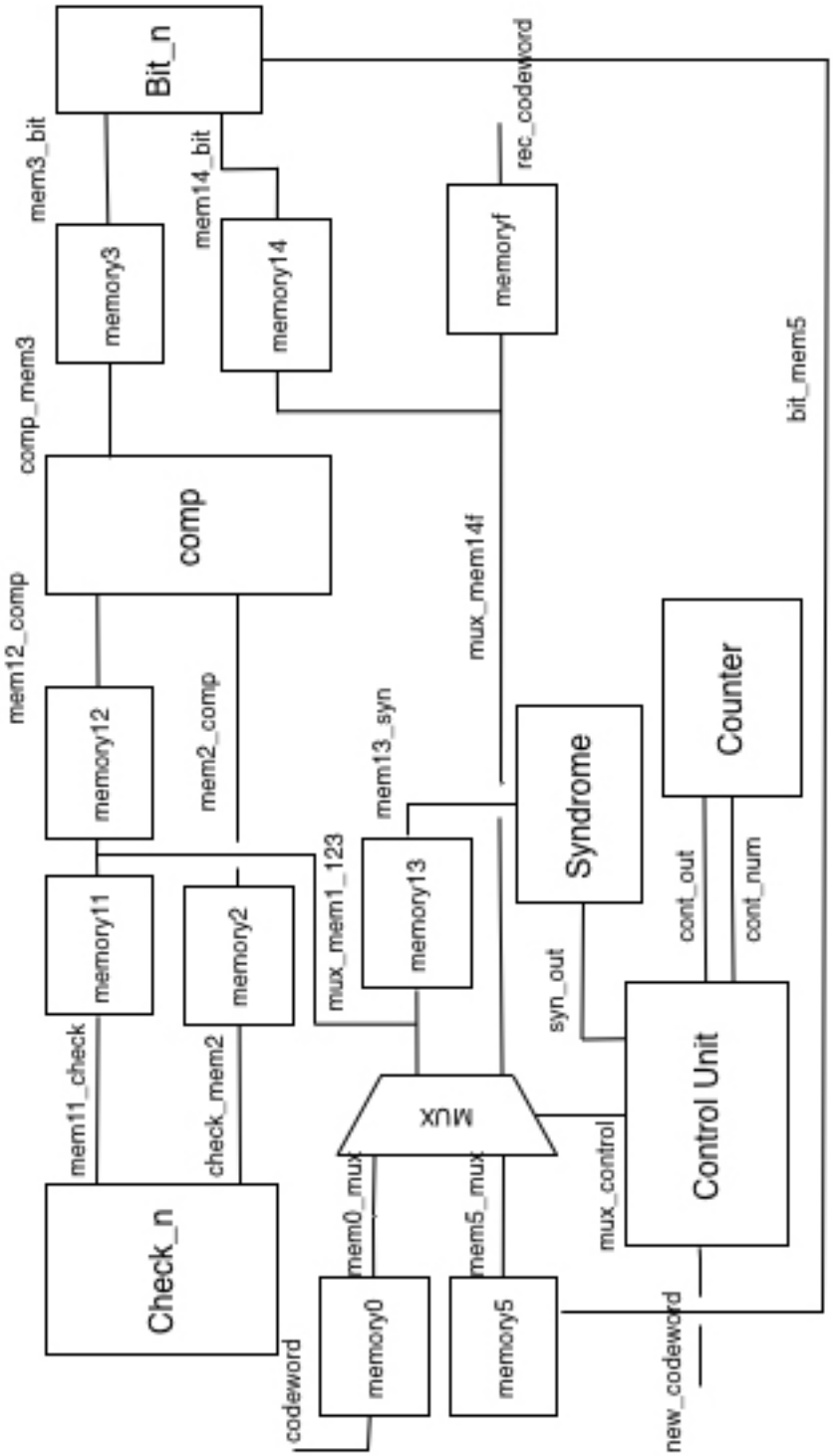

Figure 22. The complete schematic of the decoder is depicted: all the reset signals, enable signals and the clock signal have been hidden to avoid confusion with all the other signals

architecture, once the decoder is implemented only the code described from that H matrix can be decoded correctly. Furthermore, since this architecture implements a hard-decision decoding algorithm, it only deals with binary values; notice that when a soft-decision (or a partial soft-decision) algorithm is implemented, the decoder has to deal with decimal numbers (probability values) and not only 0s or 1s. In this case each node processor needs to store real numbers and, as in case of Sum-Product algorithm, more complex operations must be computed. For this reason the whole decoder architecture gets more complex, a greater number of Logic Elements of the FPGA are required and bigger memories are needed. So in general we can say that the decoding algorithm implemented in hardware is also index of the resulting architecture complexity.

3.2 Decoder Simulation

To verify the correct behaviour of the implemented decoder the MODELSIM software by ALTERA has been used (see (12)). A series of codewords have been put as input of the decoder together with the warning signal that a new codeword is ready to be analyzed. A clock signal at a frequency of 1GHz has been generated and used to syncronize the whole circuit. In Figure 23 the response of the circuit to the input codeword "000011" is shown. Notice that in Figure 23 all the bit sequences are in the opposite order with respect to the real one because all the registers slots have been sorted as "MSB downto LSB"; for this reason in Figure 23 the input codeword appears as "110000" instead of "000011". Furthermore all the messages exchanged between the Blocks are shown and also all the states in which the machine is in that moment or will be in the next future are. The output of the counter is also present to

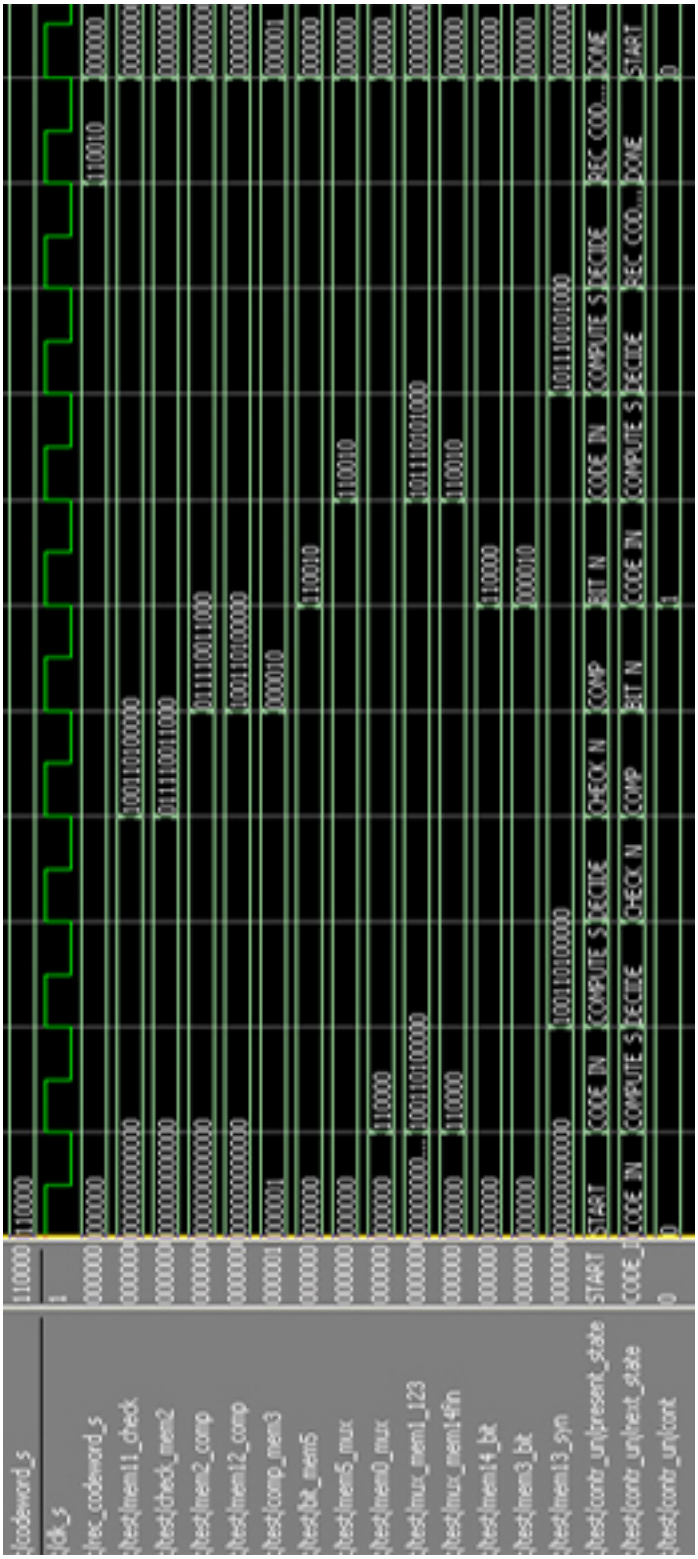

Figure 23. MODELSIM simulation result

show how many (if any) iterations the decoder needs in order to recover the correct codeword. Before a new codeword is ready to be analyzed the machine is stuck in the START state and all the memories are set to all zeros bit. When the new codeword is input into the decoder then the machine goes to the CODE_IN state and the MEMORY0 is loaded in such a way that the sequence "000011" is sent through the MUX to all the registers in the circuit, even if all these registers are still disabled. The first register to be enabled is the MEMORY13, from which the Syndrome Block gets its input bits "000001011001" and checks if the incoming sequence of symbols corresponds to a valid codeword (COMPUTE_S state) . As soon as the Syndrome Block analyzes its input it outputs a warning message that informs the Control Unit whether the codeword is valid or not. Since in this case the received sequence does not correspond to a valid codeword the Control Unit tells (in DECIDE state) the machine to go to the CHECK_N state in which the MEMORY11 is enabled and the Check-nodes Block evaluates its input and stores the output "000110011110" into MEMORY2. Then the machine goes to the COMP state in which MEMORY12 and MEMORY2 are enabled and the Comparator Block estimates which is the bit that has to be flipped. In this case the bit to be flipped is the second one, so the output "010000" is stored into MEMORY3. After that the machine goes into BIT_N state and here finally the bit is flipped by means of the xor operation between the content of MEMORY3 and the original codeword stored into MEMORY14 "000011". The output of Bit-nodes Block "010011" is finally stored into MEMORY5 and then the machine goes into CODE_IN state again, from where the algorithm starts again. Since this time the content of MEMORY5 corresponds to a valid codeword the Syndrome Block analyzes its input sequence

and informs the Control Unit that a valid codeword has been found. So from DECIDE state this time the machine goes to REC_CODEWORD state where the MEMORYF is enabled and the recovered codeword "010011" is available as output of the decoder. After that the machine goes into the DONE state from which it goes again to the START state, waiting for a new codeword to analyze and here all the registers are reset to all zero vectors.

For this implementation a total number of 97 Logic Elements have been used, that corresponds to less than 1% of the available Logic Elements on the FPGA used. Moreover the real maximum computing rate (maximum frequency) achievable due to the delays along the physical paths is 215.66MHz.

CHAPTER 4

DECODING ALGORITHMS AND THEIR PERFORMANCES

In Chapter 3 the implementation of a decoder based on the Bit-flipping algorithm has been shown, but as already seen in Chapter 2 Bit-flipping is not the only LDPC decoding algorithm. The goal in this new Chapter is to apply the three most common decoding algorithms (Sum-product, Weighted Bit-flipping and Bit-flipping) to different codes and then compare their performances and discuss the results. In order to analyze the algorithms behaviour several simulations on MATLAB software have been run.

4.1 Algorithms Code

The three decoding algorithms described in Chapter 2 have been implemented in three different MATLAB functions; the relevant code for all of them is as follows.

4.1.1 Bit-flipping Algorithm Code

```
function [codeword,r] = bitflip_func(H1,y)
%the 2 argument passed to the function are the parity check matrix H1 and
%the received sequence of bits alterated by the channel noise
 n=size(H1,2);
 m=size(H1,1);
 s=zeros(1,m);
 r=zeros(1,n);
```

```
for i=1:n
    if y(i)>=0  %y is the vector received after the channel, with noise
        r(i)=1;  %r is the received vector after the hard decision block
    end
end
codeword=r;%the recovered codeword is initialized to the output of the
           %hard decision block
%the syndrome is computed
s=mod(H1*r',2);
iter=1;
iteration=100; %maximum number of iterations allowed
q=ones(1,size(H1,1));
%if the max number of iterations has been reached or a codeword has been
%found the algorithm stops
while ((iter<=iteration) && q*s~=0)
    f=zeros(1, size(H1,2));
    %position of elements in syndrome different
    %from 0, which correspond to the to the check
    % nodes in which the parity equation is not satisfied
    ind_vect=find(s);
    %f is the vector of the number of failed syndrome bits for every symbol
```

```
    for i=1:length(ind_vect)
        for j=1:size(H1,2)
            if(H1(ind_vect(i),j)==1)
             f(j)=f(j)+1;
            end
        end
    end
    max=f(1);
   %ind_max is the position in f of the greatest number
    ind_max=1;
    for i=1:length(f)
        if f(i)>max
            max=f(i);
            ind_max=i;
%the bit connected to the greatest number of check
%nodes with wrong parity check equations will be flipped
        end
    end
    codeword(ind_max)=xor(codeword(ind_max),1); %the bit is flipped
    %computation of the new syndrome vector
    s=H1*codeword';
```

```
    for i=1:length(s)
        if(mod(s(i),2)==0)
            s(i)=0;
        else
            s(i)=1;
        end
    end
    iter=iter+1;
end
end
```

4.1.2 Weighted Bit-flipping Algorithm Code

```
function [codeword,r] = wbitflip_func(H1,y)
%the 2 parameters passed to this function are the parity check matrix H1
%and the bit sequence affected by noise after the channel
    n=size(H1,2);
    m=size(H1,1);
    N=zeros(m,n);
    r=zeros(1,n);
    M=zeros(n,m);
    y_min=zeros(1,m);
    E=zeros(1,n);
```

```
    iter=1;
    iteration=100; % maximum number of iterations allowed
    q=ones(1,size(H1,1));
for i=1:m
    qn=length(find(H1(i,:)));
    vect_indn=find(H1(i,:));
    for w=1:qn
        N(i,w)=vect_indn(w); % in each N row there are the
  % positions of the entries of each H row
    end
end
for j=1:n
    qm=length(find(H1(:,j)));
    vect_indm=find(H1(:,j));
    for w=1:qm
        M(j,w)=vect_indm(w); % in each M row there are the positions
   % of the entries of each H column
end
%for each row (check node) the minimum
%corresponding value of the y vector is found
for i=1:m
```

```
    min=abs(y(N(i,1)));
    for j=1:length(find(N(i,:)))
        if abs(y(N(i,j)))<min
            min=abs(y(N(i,j)));
        end
    end
    y_min(i)=min;
end
for i=1:n
    if y(i)>=0  %y is the vector received after the channel, with noise
        r(i)=1;  %r is the received vector after the hard decision block
    end
end
codeword=r;%the codeword is initialized to
    %the received sequence after
                    %the hard decision block
s=mod(H1*r',2);%s is the syndrome vector
%the algorithm goes on until a valid codeword
%has been found or the maximum
%number of iterations has been reached
while ((iter<=iteration) && q*s~=0)
```

```
    for i=1:n %rows
        sum=0;
        for j=1:length(find(M(i,:)))   %columns
            sum=sum+((2*s(M(i,j))-1)*y_min(M(i,j)));
        end
        E(i)=sum;
    end
    max=E(1);
    index=1;
    for i=1:n
        if (E(i)>max)
            index=i;
            max=E(i); %the bit in correspondance of
     %the max value of E will be flipped
        end
    end
    codeword(index)=xor(codeword(index),1); %flipping the bit
    %the new syndrome vector is computed
    s=mod(H1*codeword',2);
    iter=iter+1;
end
```

```
end
```

4.1.3 Sum-product Algorithm Code

```
function [c_rec,r] = sum_prod_func(H1,y)

%the parameters passed to the function are the parity check matrix H1 and

%the received vector after the channel y (with noise)

format long

[m,n]=size(H1);

N=zeros(m,n);

M=zeros(n,m);

c_rec=zeros(1,n);

N0=1; % this is the variance of the noise values

P1 = ones(size(y))./(1 + exp(-2*y./(N0)));

%P(ci=1|yj)-> the probability that ci is 1 if yi arrived

P0 = 1 - P1;  %P(ci=0|yj)-> probability that ci is 0 if yi arrived

L_ch=log(P0./P1); %LLR of the input vector

Lj_tot=zeros(1,n);

Lv_c=zeros(m,n); %from bit to check nodes

Lc_v=zeros(m,n); %from check to bit nodes

r=zeros(1,n);

%in each row there are the bit nodes

%connected with the coresponing check nodes
```

```
for i=1:m
    qn=length(find(H1(i,:)));
    vect_indn=find(H1(i,:));
    for w=1:qn
        N(i,w)=vect_indn(w);
    end
end
%in each row there are the check nodes
%connected to the corresponding bit nodes
for j=1:n
    qm=length(find(H1(:,j)));
    vect_indm=find(H1(:,j));
    for w=1:qm
        M(j,w)=vect_indm(w);
    end
end
iter=0;
iteration=100; %maximum number of iterations
for i=1:n
    if y(i)>=0  %y is the vector received after the channel, with noise
        r(i)=1;  %r is the received vector after the hard decision block
```

```
    end
end
c_rec=r;
%the recovered codeword is initialized to the vector got after the
%decision block
s=mod(H1*r', 2) ;
%s is the syndrome, if it is 0 then the received word is correct
k=ones(1,m);
%if the syndrome is all zero vector or the
%maximum number of iterations has
%been reached then the algorithm stops
while ((iter<iteration) && (k*s~=0))
    iter=iter+1;
    for i=1:m
        for j=1:length(find(N(i,:)))
            prod=1;
            for q=1:length(find(N(i,:)))
                if j~=q
                    if iter==1
                        prod=prod*(tanh(0.5*L_ch(N(i,q))));
                    else
```

```
                    prod=prod*(tanh(0.5*Lv_c(i,N(i,q))));
                end
            end
        end
        Lc_v(i,N(i,j))=2*(atanh(prod)); %messages from check to bit nodes
    end
end
for i=1:n
    for j=1:length(find(M(i,:)))
        sum=0;
        for q=1:length(find(M(i,:)))
            if j~=q
                sum=sum+Lc_v(M(i,q),i);
            end
        end
        Lv_c(M(i,j),i)=L_ch(i)+sum; %messages from bit to check nodes
    end
end
for i=1:n
    sum=0;
    for j=1:length(find(M(i,:)))
```

```
            sum=sum+Lc_v(M(i,j),i);
        end
        Lj_tot(i)=L_ch(i) + sum; %total LLR for all bit nodes
    end
    %if Lj_tot is positive then a 0 is decided, otherwise a 1 is decided
    for i=1:n
        if Lj_tot(i)<0
            c_rec(i)=1;
        else
            c_rec(i)=0;
        end
    end
    %check on the syndrome, if it is all zero vector the algorithm stops
    s=mod(H1*c_rec',2);
end
end
```

4.2 Algorithm Performance With A (6,3) Code

The first MATLAB simulation is about how the three algorithms behave when they are applied to a (6,3) block code. In particular the code is the one described by the H matrix used to implement the decoder in Chapter 3. The H matrix, here not presented in a systematic form, is

$$H = \begin{bmatrix} 1\,0\,1\,1\,0\,0 \\ 0\,1\,1\,0\,1\,0 \\ 1\,0\,0\,0\,1\,1 \\ 0\,1\,0\,1\,0\,1 \end{bmatrix}$$

and the generation matrix G for the same code is

$$G = \begin{bmatrix} 1\,0\,0\,1\,0\,1 \\ 0\,1\,0\,0\,1\,1 \\ 0\,0\,1\,1\,1\,1 \end{bmatrix}$$

from which all the codewords $\mathbf{c}$ can be obtained from the orignal messages $\mathbf{u}$ by

$$\mathbf{c} = \mathbf{u}\ G$$

In the simulation 1000 different $\mathbf{u}$ messages have been randomly created, and each of them has been coded by means of G. Then a Gaussian channel noise has been added to each codeword and its standard deviation has been varied in such a way to have a Signal to Noise Ratio (SNR) (defined as the ratio between the power of the signal and the power of noise) from 0dB to 6dB with a step of 0.25dB. The channel model used is the Additive White Gaussian Noise (AWGN) in which the noise has a constant spectral density and a Gaussian distribution of amplitude. Also, the noise has been created with the *randn* MATLAB function, which creates

a matrix or a vector of pseudorandom values with mean equal to 0 and standard deviation equal to 1. The standard deviation of such noise has been modified by multiplying it by a variable value A. The simulation results are shown in Figure 24 and Figure 25. In Figure 24 we can see the Bit Error Rate obtained for each implemented algorithm, that is the number of wrong bits after the decoding process over the total number of bits sent. In Figure 25 the effective number of bit errors is shown. In this case 1000 codewords for each SNR value have been analyzed, and for each codeword there are 6 bits, so the total number of bits sent for each SNR value is 6000. From the simulations it results that the Sum-product agorithm is the one with the best performance, in fact at SNR around 6dB no error occurred at all, with a BER tending to minus infinity on a logarithmic scale. At the same SNR level the Weighted Bit-flipping algorithm presents 9 errors out of 6000 with a BER = 0.0015. The same number of errors (9 out of 6000) has occurred using the Bit-Flipping algorithm at SNR = 6dB, but with worse performance in general for low noise values (it is clear from Figure 24 that the curve representing the BER using a Bit-Flipping algorithm always stays above the other curves, indicating a worse BER). Considering now lower values of SNR we can see that the Sum-product algorithm is still the best one, followed by the Weighted Bit-flipping and finally from the Bit-flipping. In general we can state that with small codes a low BER is obtained with all algorithms, even for low SNR values, in fact the highest BER value is found at 0dB (the power of the signal equal to the power of the noise) with a Bit-flipping algorithm and it is BER = 0.1205 (723 errors out of 6000 bits). At the same SNR a Sum-product performs far better, with only 477 errors and BER = 0.0795, while the Weighted Bit-flipping has BER =

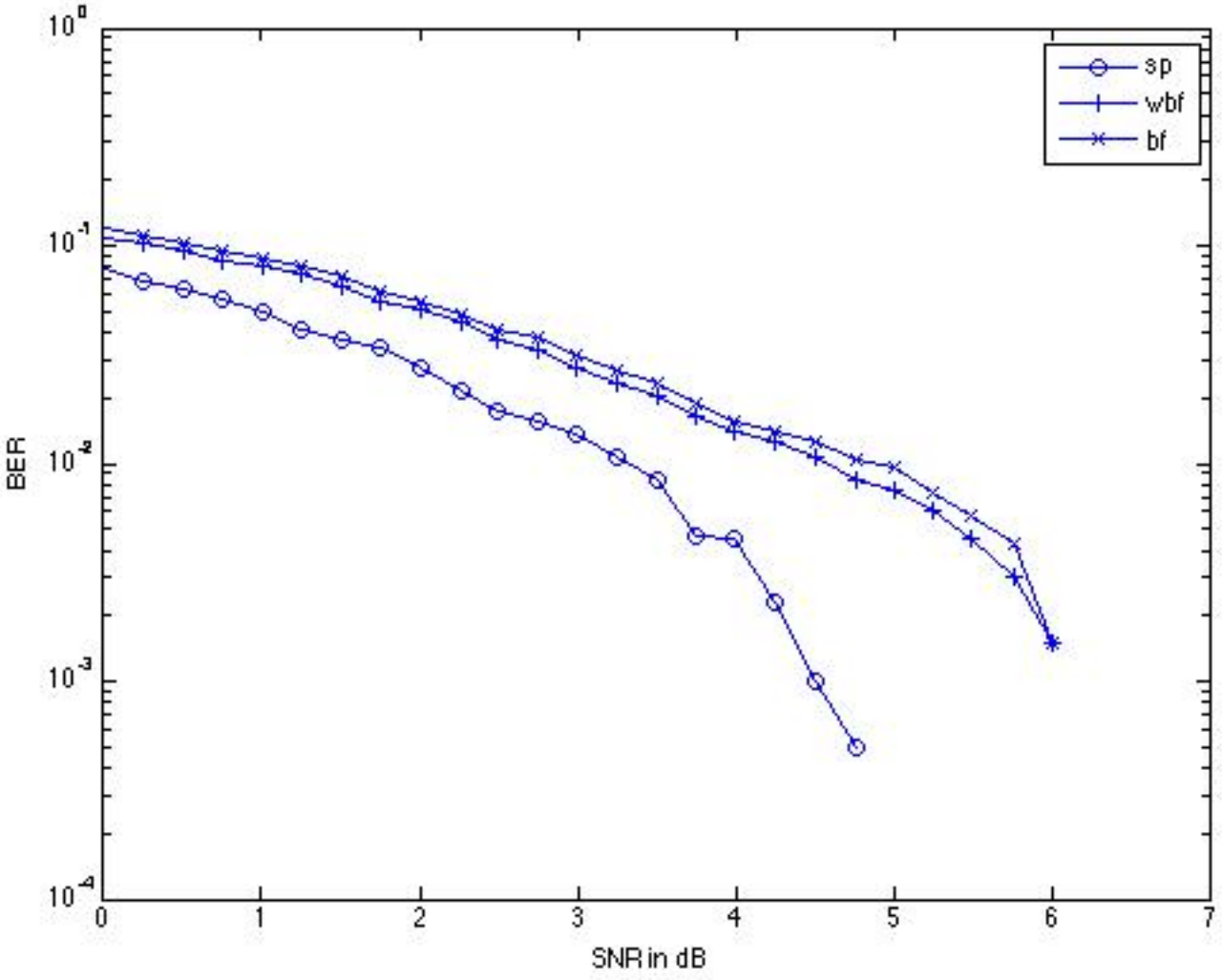

Figure 24. Bit Error Rate due to the channel noise relative to a (6,3) code

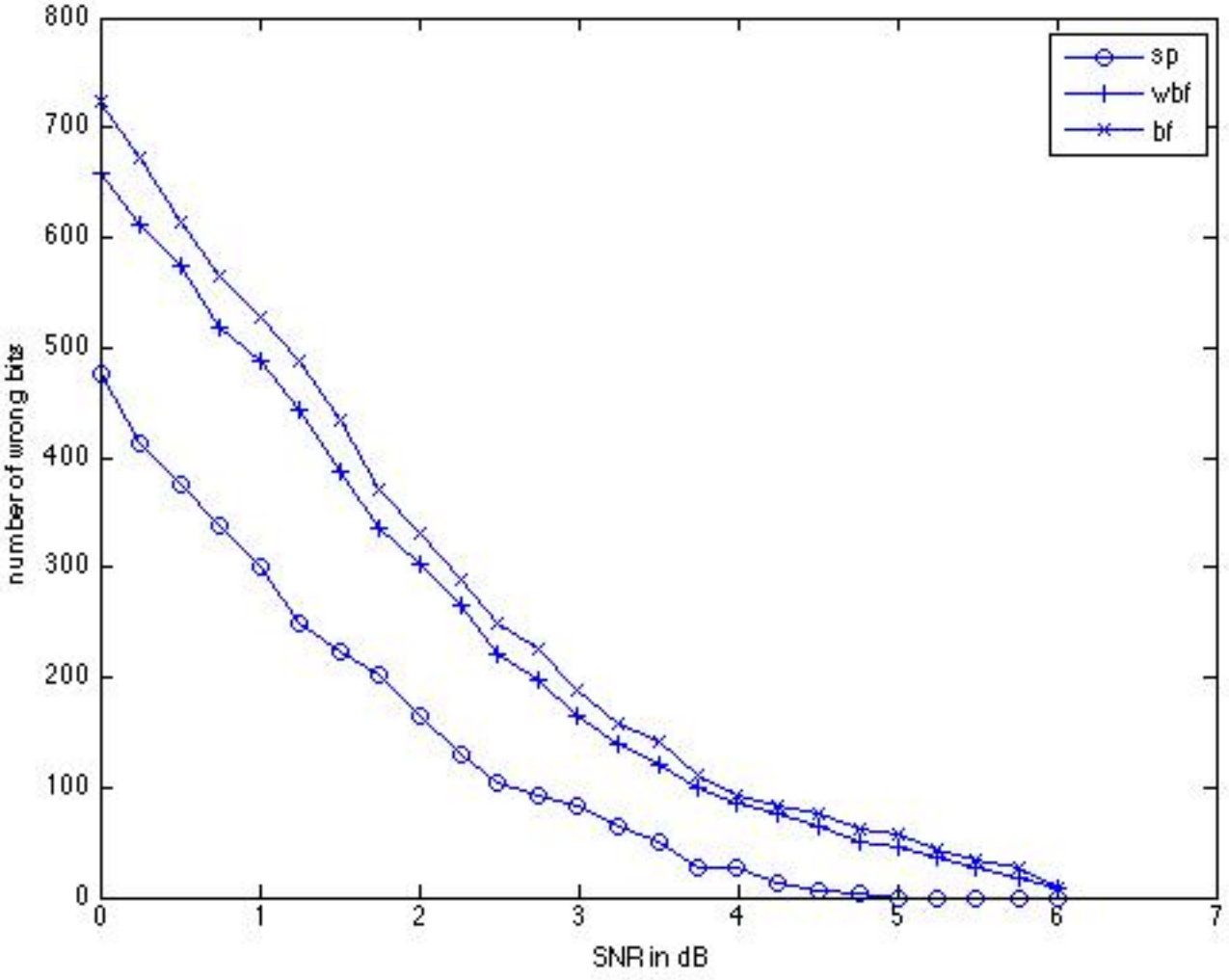

Figure 25. Total number of wrong bits after the decoding process, per each considered SNR value, with (6,3) code

0.1098 with 659 total errors. So we can say that for a code of small size the algorithm which performs better is the Soft decision Sum-product one, but that also an Hard decision algorithm keeps a low BER even for low SNR.

Besides all these data about the BER, another observation has to be done about how many codewords have been correctly recovered after the channel. The three algorithms have been applied for a total of 25000 times each, 10001 times of which the bit sequence after the channel had been alterated by the noise. At this point it turns out that, with the Sum-product algorithm the codeword is correctly recovered 8907 times out of 10001, with the Weighted Bit-flipping 8106 out of 10001 and with the Bit-flipping only 7946 out of 10001. Once again we see how the Sum-product is the algorithm with the best performance over this kind of code.

4.3 Algorithm Performance With A (55,33) Code

In the second MATLAB simulation a 55-length code with code rate $r = \frac{k}{n} = 0.6$ has been tested by the algorithms. The corresponding parity check matrix H and the generator matrix G are two low-density matrixes, depicted in Figure 26 and Figure 27, where a full point in the matrix represents an entry (a 1 element) and the blank spaces are instead 0s. As we can see this time H is a full-rank matrix, with 10% of non zero element and it is put in its systematic form.

As in the previous simulation, 1000 different codewords have been generated starting from 1000 different 33-bit random messages. A AWGN channel has been used to introduce noise to the transmitted sequence of symbols. The result of the decoding process with the three algorithms is shown in Figure 28 and Figure 29 where the BER and the total number of errors

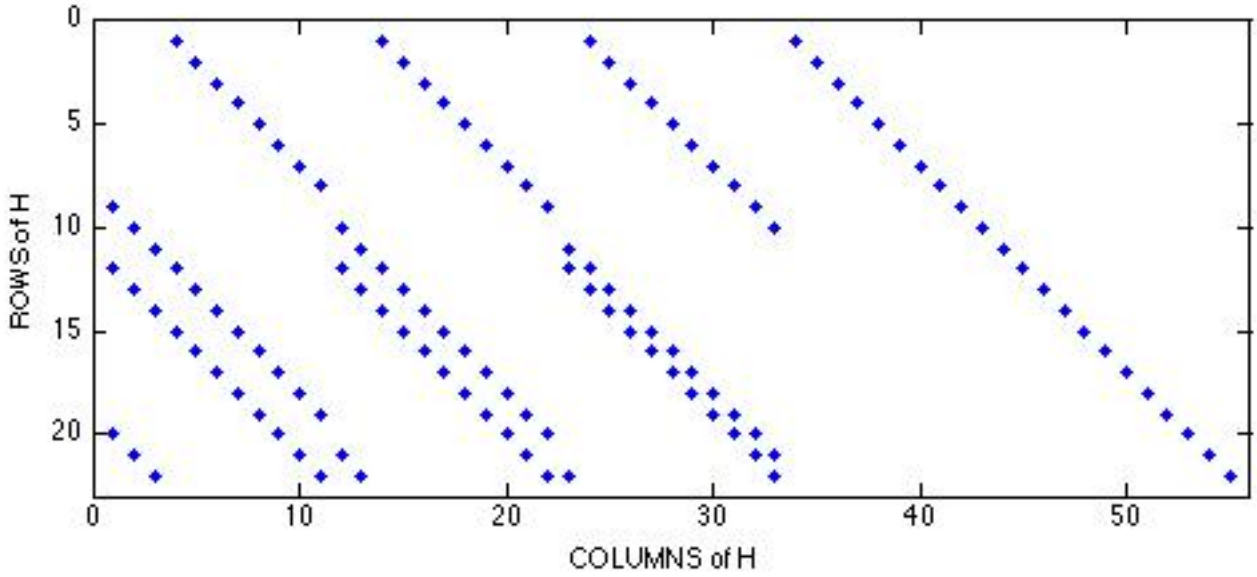

Figure 26. In the figure the parity check H matrix is shown: the full points represent 1s in the matrix

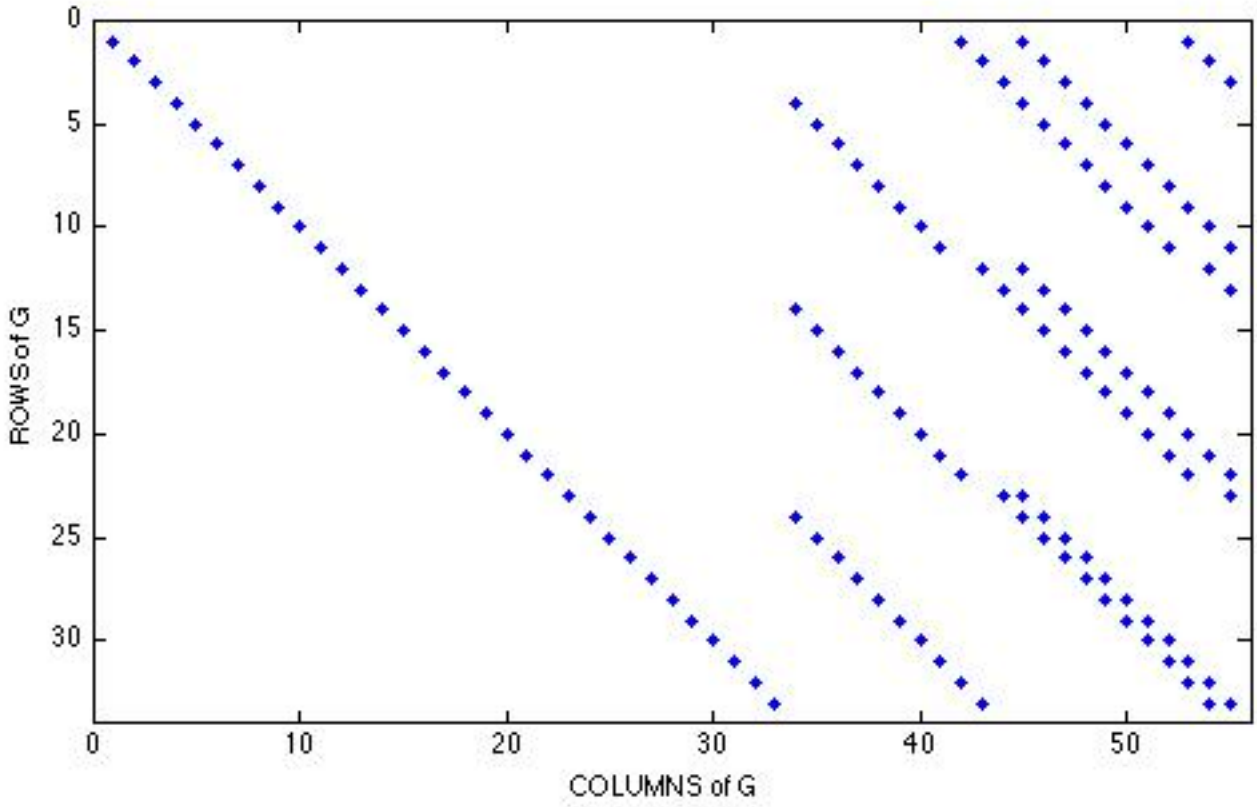

Figure 27. In the figure the generator matrix G of the (55,33) code is shown: the full points represent 1s in the matrix

are depicted. From the pictures it is clear that for high values of SNR the situation is similar to the one in the previous simulation with a small code. The Bit-Flipping algorithm is the worst one, followed by the Weighted one and the Sum-Product. At SNR=8dB for Sum-Product the BER is 0.000018, with only 1 error, due to one wrong bit out of 55000 total bits, found after the decoding process; at the same SNR value for a Weighted Bit-Flipping algorithm we have BER = 0.000145 and 8 bit errors out of 55000. For a simple Bit-Flipping algorithm the BER rises to 0.0022 with a total of 121 wrong bits out of 55000. Moving now to lower values of SNR we notice a difference with respect to the previous simulation: around 0dB of SNR the Weighted Bit-Flipping algorithm seems to performe slightly worse than the Bit-Flipping. This is shown in picture Figure 28 and, in numbers, we have for Weighted Bit-Flipping a BER = 0.1883 and 10360 wrong bits out of 55000, while for Bit-Flipping a BER = 0.1844 and 10143 wrong bits out of 55000. This phenomenon can be due to the fact that sometimes, in order to recover the original codeword, the Weighted Bit-flipping algorithm tries a particular correction path which results to be wrong and that instead of correcting the bit sequence, it introduces more errors. For example this can be the case in which the algorithm recovers a valid codeword and stops its iterations, but the recovered codeword is not the one which was sent. The Sum-Product algorithm remains the one with best performances also at low SNR, with a BER = 0.12 and only 6617 wrong bits out of 50000 total bits.

In this simulation 33000 different messages have been coded and 33000 codewords have been analyzed by the algorithms. The number of times the Sum-Product algorithm has been able to recover the right codeword after a non valid bit sequence arrived from the channel is 16409

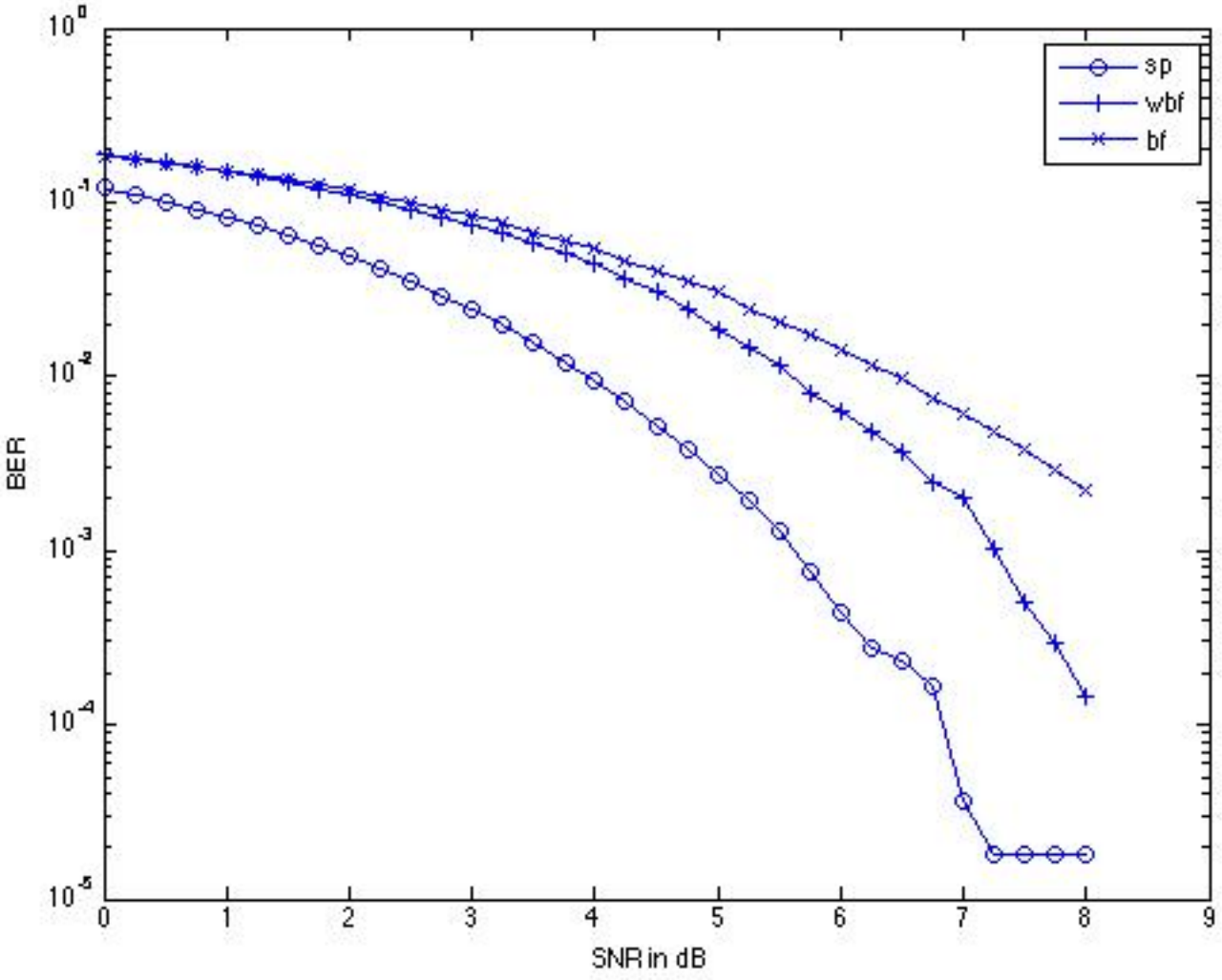

Figure 28. Bit Error Rate due to the channel noise relative to a (55,33) code

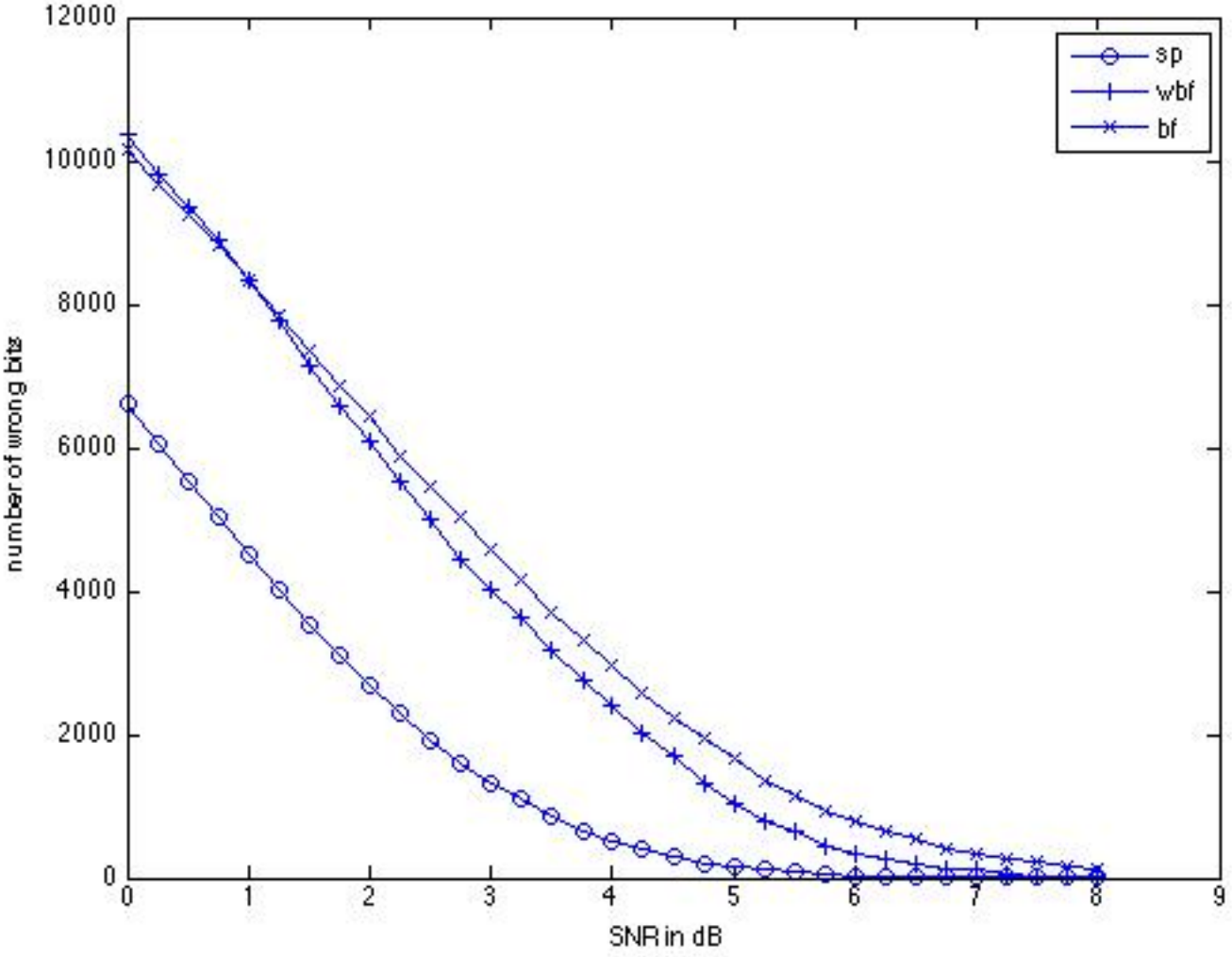

Figure 29. Total number of wrong bits after the decoding process, per each considered SNR value, with (55,33) code

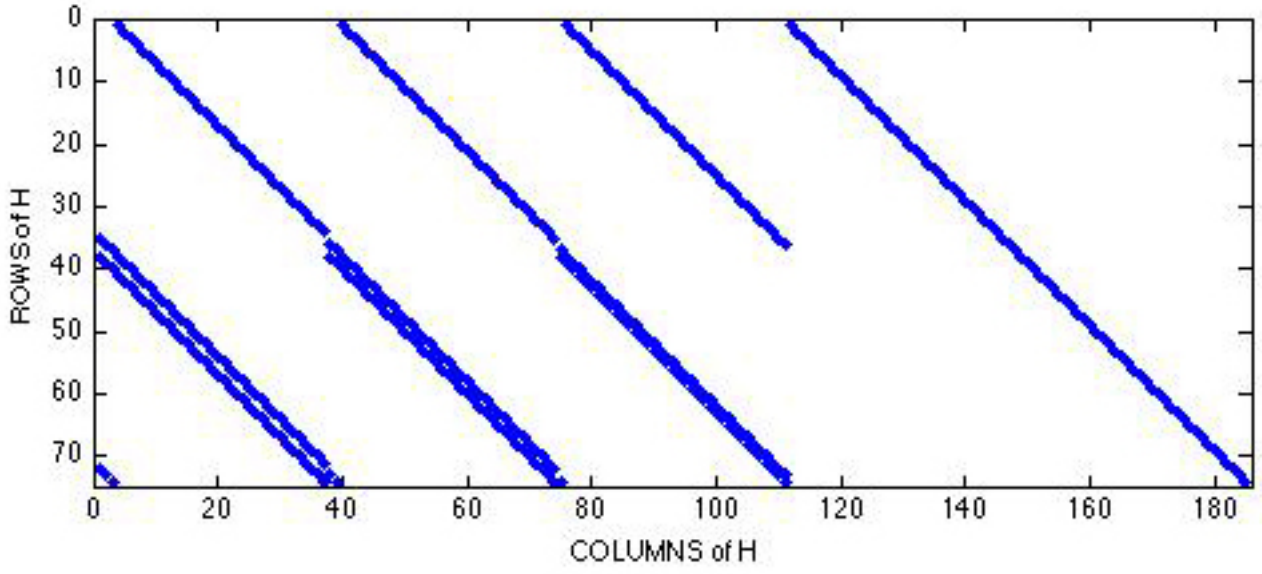

Figure 30. In the figure the parity check H matrix is shown: the full points represent 1s in the matrix

out of 27517 . This number decreases for the Weighted Bit-Flipping to 12921 out of 27517 and for the Bit-Flipping algorithm to 7476 out of 27517. From these data we clearly see that the Sum-Product algorithm is the best even for a code of length 55.

4.4 Algorithm Performance With A (185,111) Code

The same simulation described in the previous section has been repeated with a longer code of length 185 and code rate $r = \frac{k}{n} = 0.6$. The parity check H matrix and the generator G matrix for the used code are depicted in Figure 30 and Figure 31, while the results of the simulation are shown in Figure 32 and Figure 33.

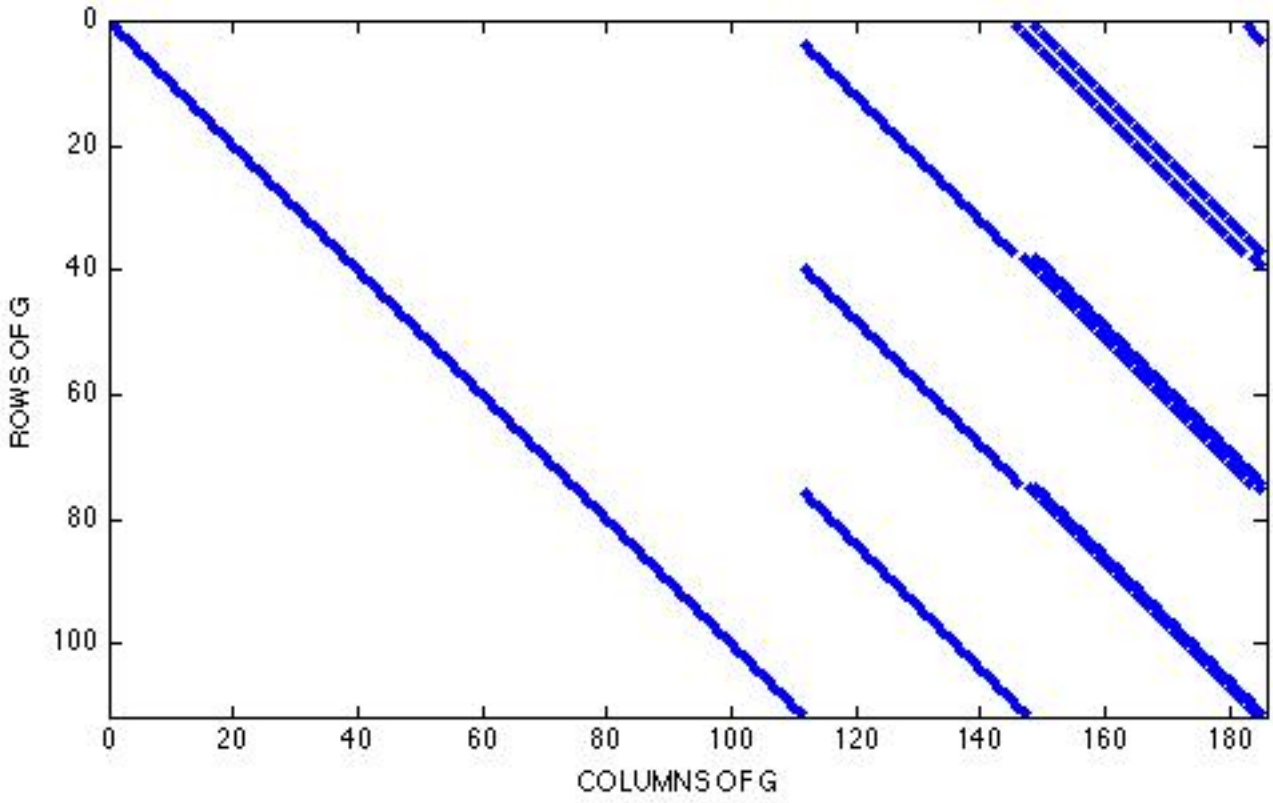

Figure 31. In the figure the generator matrix G of the (185,111) code is shown: the full points represent 1s in the matrix

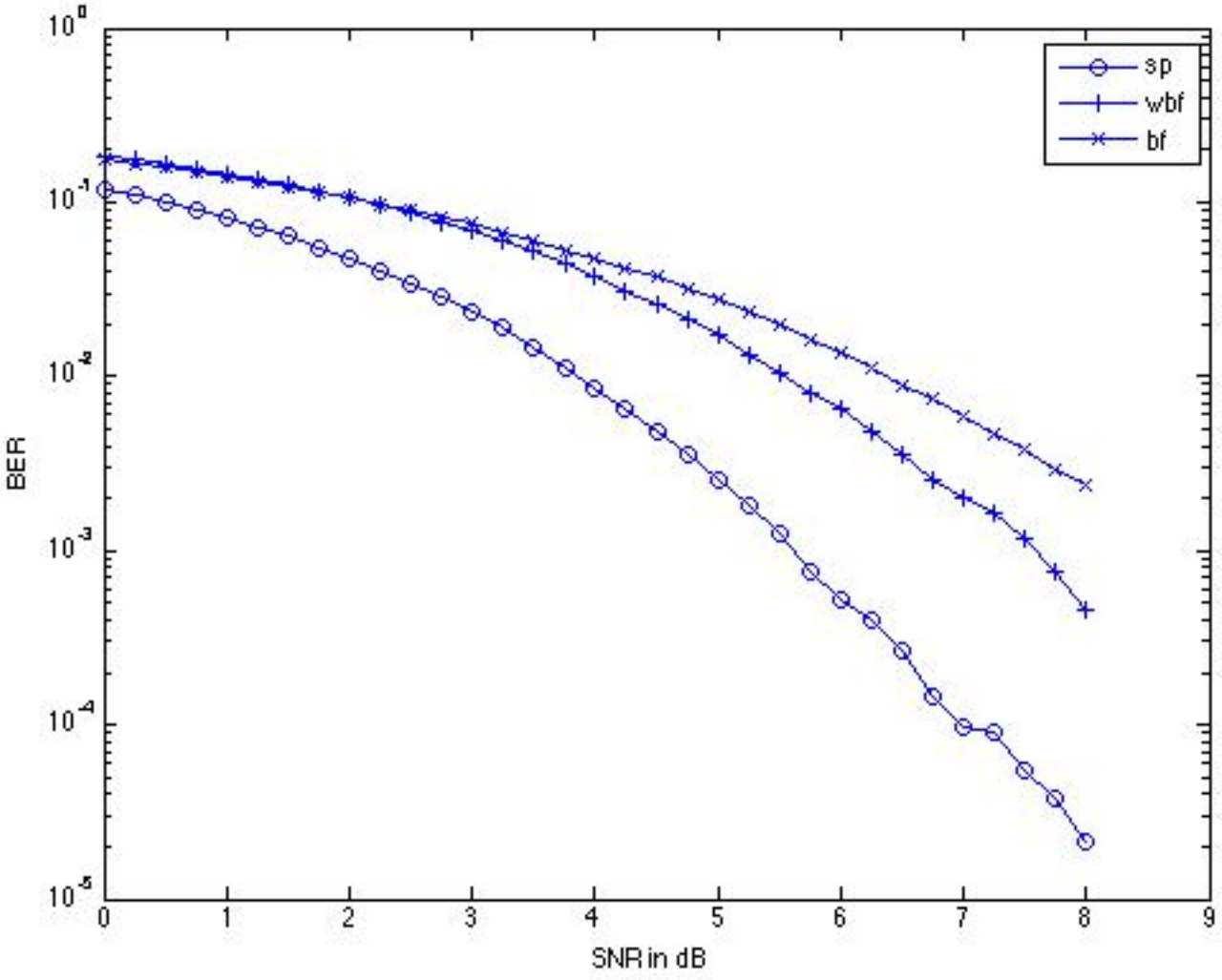

Figure 32. Bit Error Rate due to the channel noise relative to a (185,111) code

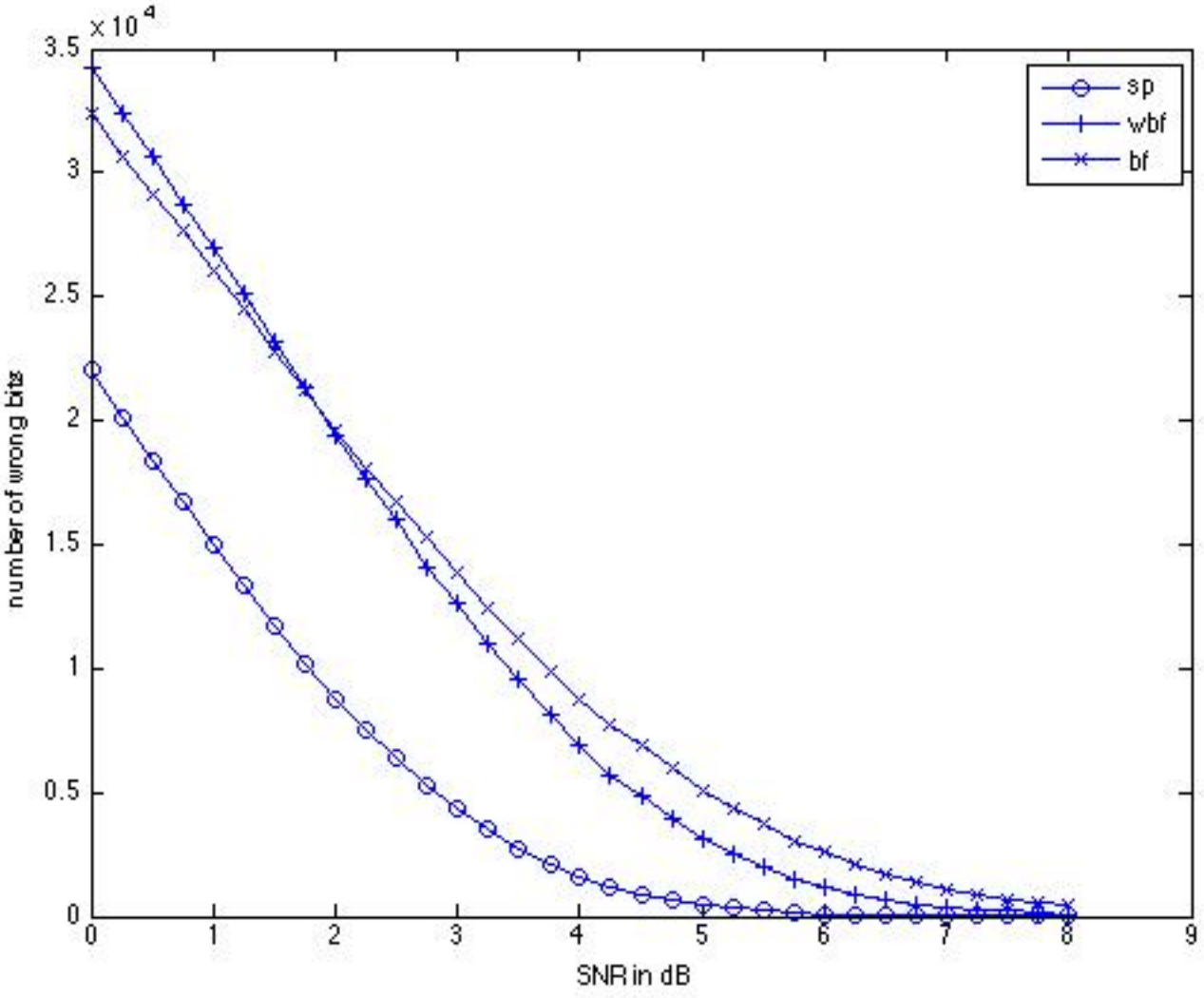

Figure 33. Total number of wrong bits after the decoding process, per each considered SNR value, with (185,111) code

The H matrix in this case is also full-rank with 2.43% of non-zero elements and it is shown in its systematic form. As we can see the three algorithms performances are similar to the ones in the previous simulations, especially for high values of SNR. In this case at SNR = 8dB with the Sum-Product algorithm we find a BER = 0.00002 with only 4 errors out of 185000, while using the Weighted Bit-Flipping algorithm we have a BER=0.00045 with 84 wrong bits out of 185000. Instead using a Bit-Flipping algorithm the corresponding BER found at SNR = 8dB is BER = 0.0023 and a total of 440 wrong bits out of 185000. For lower values of SNR the situation is exactly the same for the Sum-Product algorithm, which performs as the best one, but there is something to notice for the other two. The Weighted Bit-Flipping algorithm provides a greater BER with respect to the Bit-Flipping, as it happens for a 55-length code, but this time not only for very low values of SNR (in Figure Figure 33 it is possible to see how for SNR lower than 2dB the number of errors using Weighted Bit-Flipping algorithm is greater than the number of errors obtained with a simple Bit-Flipping algorithm). This happens because, as already said, when a great noise is present on the channel the Weighted Bit-Flipping tries to recover the original codeword but, being too many errors in the received sequence, it usually chooses the wrong correction path and instead of deleating errors it adds them to the codeword. So the bigger the length of the codewords, the greater this phenomenon shows up.

As in the previous simulation 33000 different situations have been analyzed and for 31806 times the original codeword has arrived alterated by the noise to the receiver. Using the Sum-Product algorithm 14686 valid codewords have been recovered out of 31806 bit sequences, while using the Weighted Bit-Flipping 10941 out of 31806 and with a simple Bit-Flipping only 4589

out of 31806. So once again we see how, even with bigger codewords the Sum-Product algorithm remains the best one, for all SNR values.

In both Figures Figure 28 and Figure 32 it is possible to see how the BER obtained using the Sum-Product algorithm never gets lower that 10^{-5} and this can be explained by the fact that a relative low amount of bits have been used in these simulations; using a greater amount of message bits an even lower BER could be found. At this point another consideration is needed about the speed and the kind of processor used to run all the simulations: in this thesis the simulations have been run with a 2.3 GHz Intel Core i5 single processor, with two cores and a RAM of 8Gb. The time needed to run the simulations about the (55,33) code has been of 5 minutes and 52 seconds, while the time required to run the simulations about the (185,111) code has been of 52 minutes and 10 seconds. The time required for each simulation increases exponentially with the codeword size and the number of messages sent through the channel, that is why the total number of bits has been kept low because longer sequences would have taken too much time to be decoded.

4.5 Sum-Product Algorithm Performance

Since the Sum-Product algorithm has been shown to be the best algorithm among all iterative-algorithms so far, we can also say it is the most used for very big codes. There are still two things that have not been discussed and analyzed yet about this kind of decoding process: how the maximum number of iterations allowed in the algorithm and the *code rate* can affect the decoding performance.

It is known that at each iteration more than one bit can be flipped at once, so in most cases using the Sum-Product algorithm the correct codeword can be recovered after a few iterations. Of course increasing the number of iterations also increases the chances to recover the correct codeword. A MATLAB simulation has been run to show how the maximum number of iterations affects the decoding performance. A LDPC code of length 150 and code rate 0.4 has been decoded four different times using the Sum-Product algorithm, and at each time the maximum number of iterations has been changed. 17000 codewords were analyzed and 16012 of these arrived to the decoder with at least one error. The results of the simulation with maximum iteration numbers equal to 5, 10, 20 and 50 are shown in Figure 34 and Figure 35

In the figures we can clearly see how the best performance is obtained with the biggest number of iterations allowed (50). In particular, using only 5 iterations the algorithm has been able to correct 8698 codewords out of 16028 bit sequences that had been alterated by the channel noise, while with 10 iterations 11581 codewords have been correctly recovered. So increasing the number of iterations an improvement has been detected. A better idea of that comes from the number of codewords corectly recovered using 20 iterations: 12162 codewords out of 16028. Furthermore we see how in all pictures the line representing the decoding with 5 iterations always stays above the others with a certain margine, while the other lines result to be closer to each other. So the higher the number of iterations, the better the performance of the decoder, but increasing this number too much brings a different result: many iterations provide just a little benefit or they don't provide improvement at all with respect to the case with less iterations. We can notice this issue considering that with 50 iterations allowed 12309

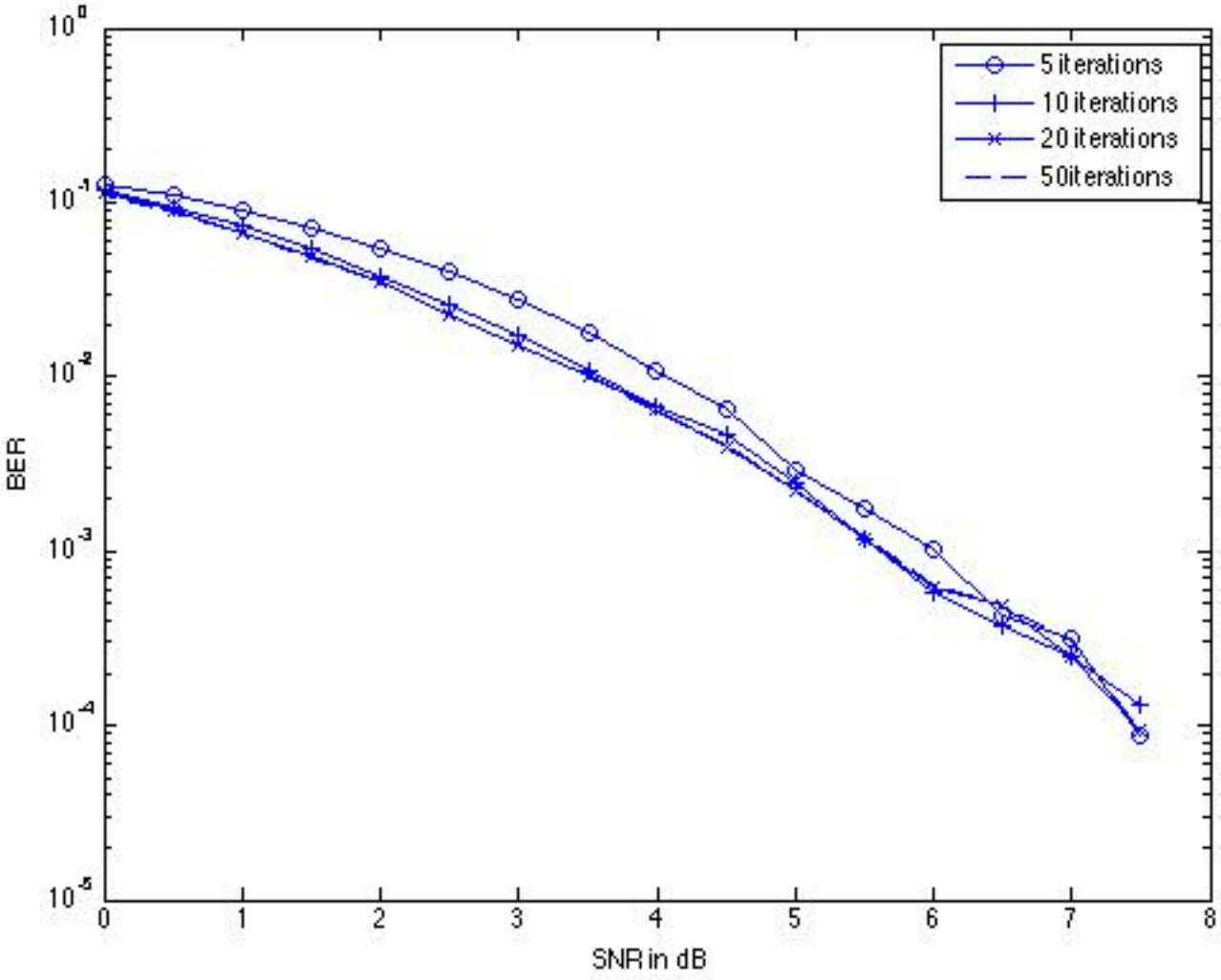

Figure 34. BER vs SNR graphs of a (150,60) code decoded using SP with 5, 10, 20 and 20 max iteration numbers

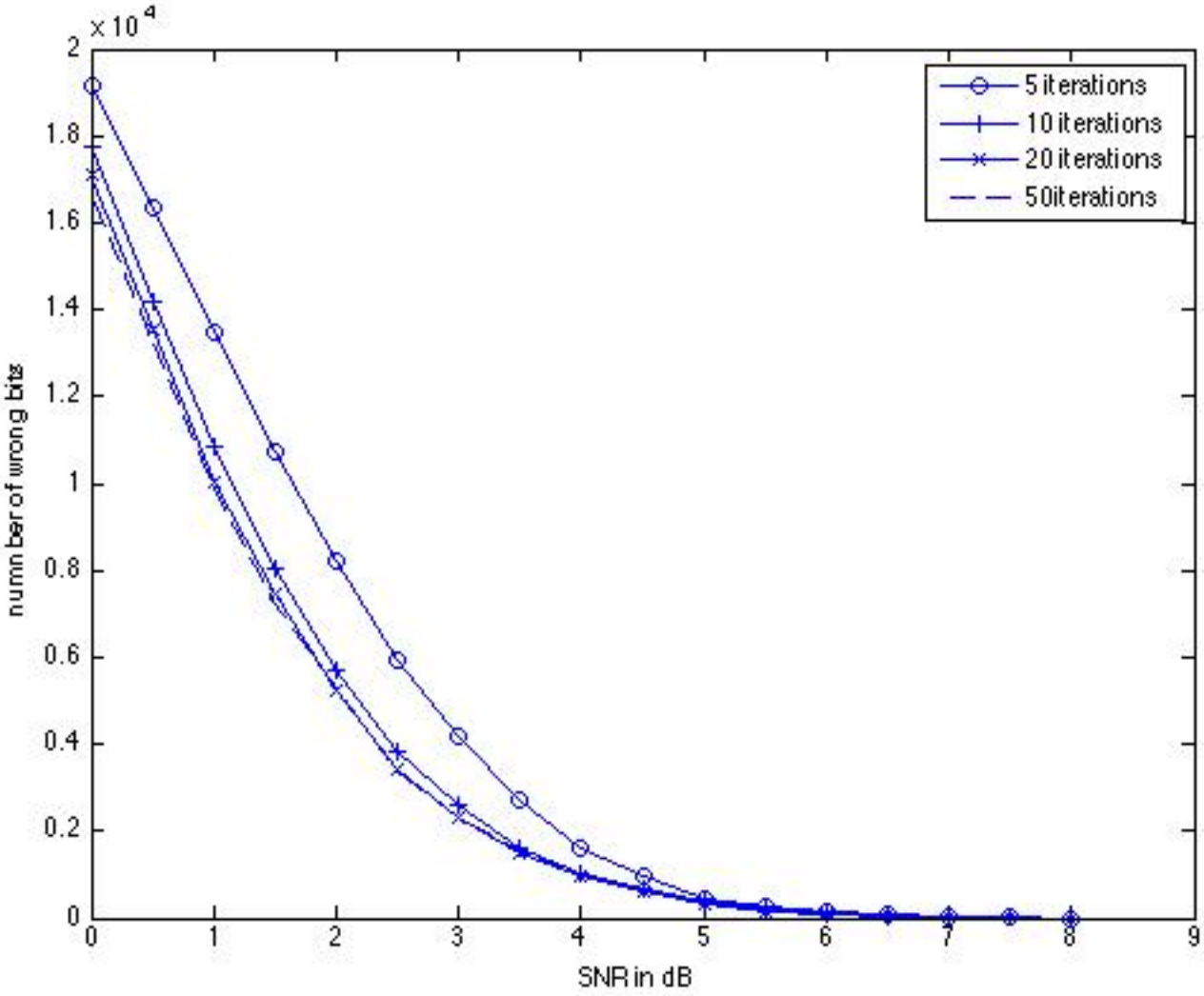

Figure 35. ERR vs SNR graphs of a (150,60) code decoded using SP with 5, 10, 20 and 20 max iteration numbers

out of 16028 codewords have been correctly recovered, that is only 147 more codewords than the same algorithm using 20 iterations. For this reason too many iterations are not very useful to improve the decoder performance, on the contrary they could just decrease the throughtput and so the communication speed.

Besides the iterations number, another important factor in error correction decoding performance is about the code rate $r = \frac{k}{n}$. As we know k is the number of bits in the original message, while n is the number of bits in the codeword, that is n - k bits have been added to the original message in order to be able to recover the original sequence after the channel. Usually a low code rate is good because it provides good performance, but at the same time it makes communication slower, because for each message there are many added bits that need to be sent as well. A MATLAB simulation has been run to show how the variation of the code rate affects the decoding performance. Four different codes with same length but different rates, have been decoded. The code length is equal to 95 bits, and the four diferent rates are 0.2, 0.4, 0.6, 0.8. The respective 4-cycles free H matrixes of all these codes are depicted in Figure 36, Figure 37, Figure 38 and Figure 39. In Figure Figure 36 a 19x95 matrix is shown, with $k = 76$ and a code rate $r = 0.8$. In Figure 37 a 38x95 matrix appears, with $r = 0.6$, while in Figure 38 a 57x95 matrix is presented, describing a code with $r = 0.4$. Figure 39 shows the last parity check matrix H which describes a code with $r = 0.2$. The results of the simulation are depicted in Figure 40 and Figure 41. It is clear from the pictures how the code with the lowest code rate ($r = 0.2$) performs far better than the others, providing the lowest number of bit errors and the maximum number of correctly recovered codewords. In

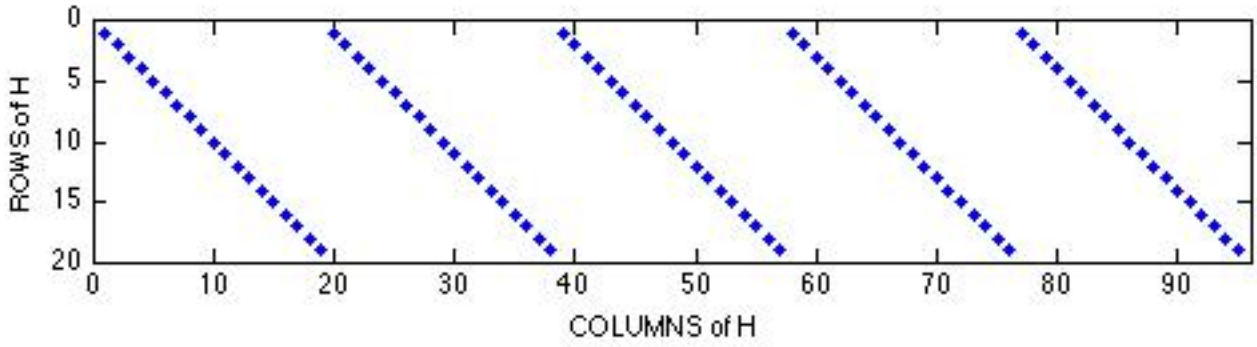

Figure 36. This is the H matrix of the code with coderate equal to 0.8: the full points represent 1s in the matrix

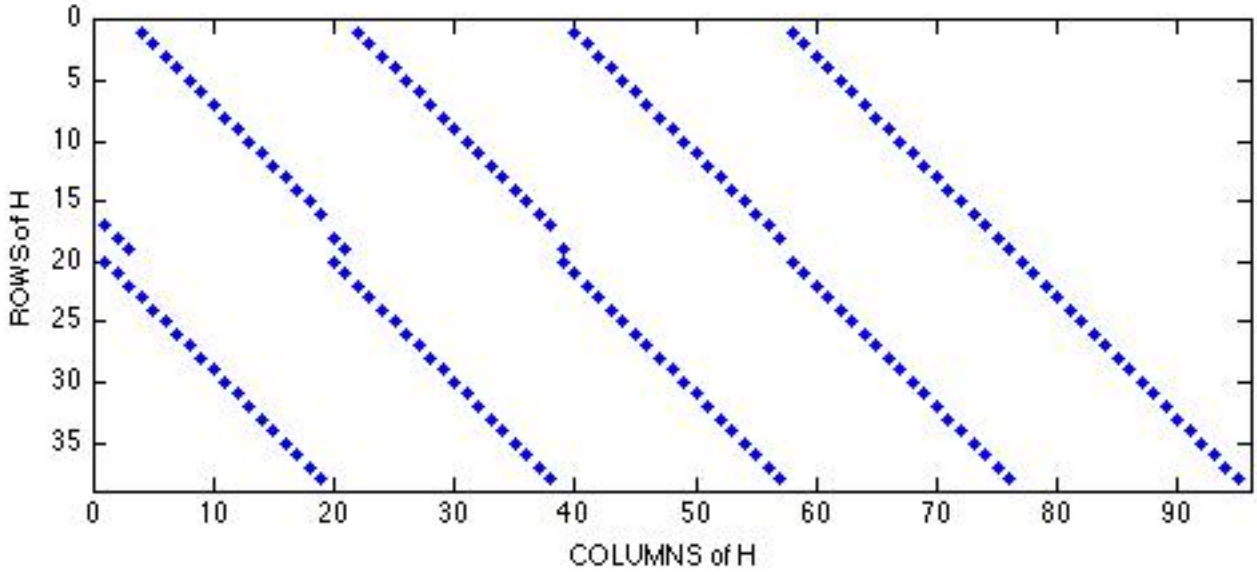

Figure 37. This is the H matrix of the code with coderate equal to 0.6: the full points represent 1s in the matrix

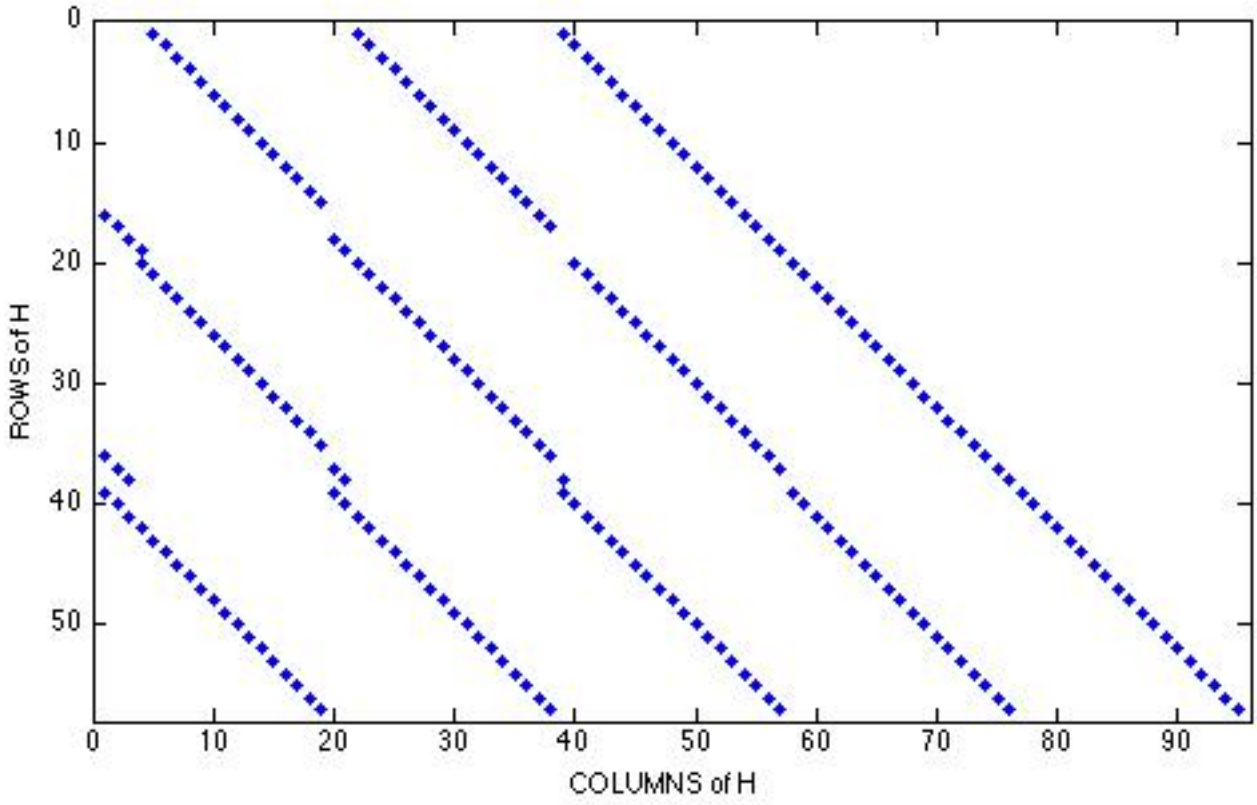

Figure 38. This is the H matrix of the code with coderate equal to 0.4: the full points represent 1s in the matrix

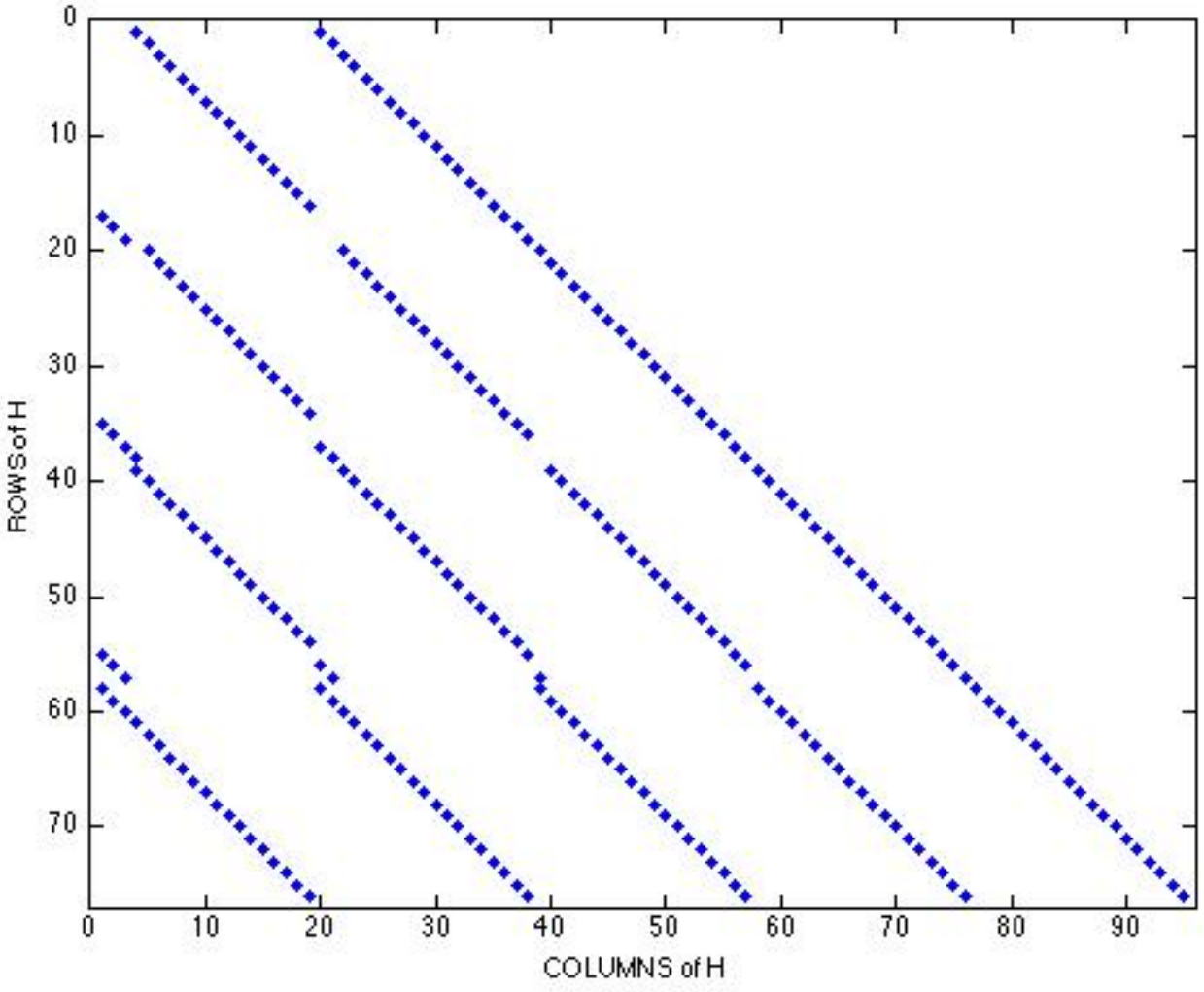

Figure 39. This is the H matrix of the code with coderate equal to 0.2: the full points represent 1s in the matrix

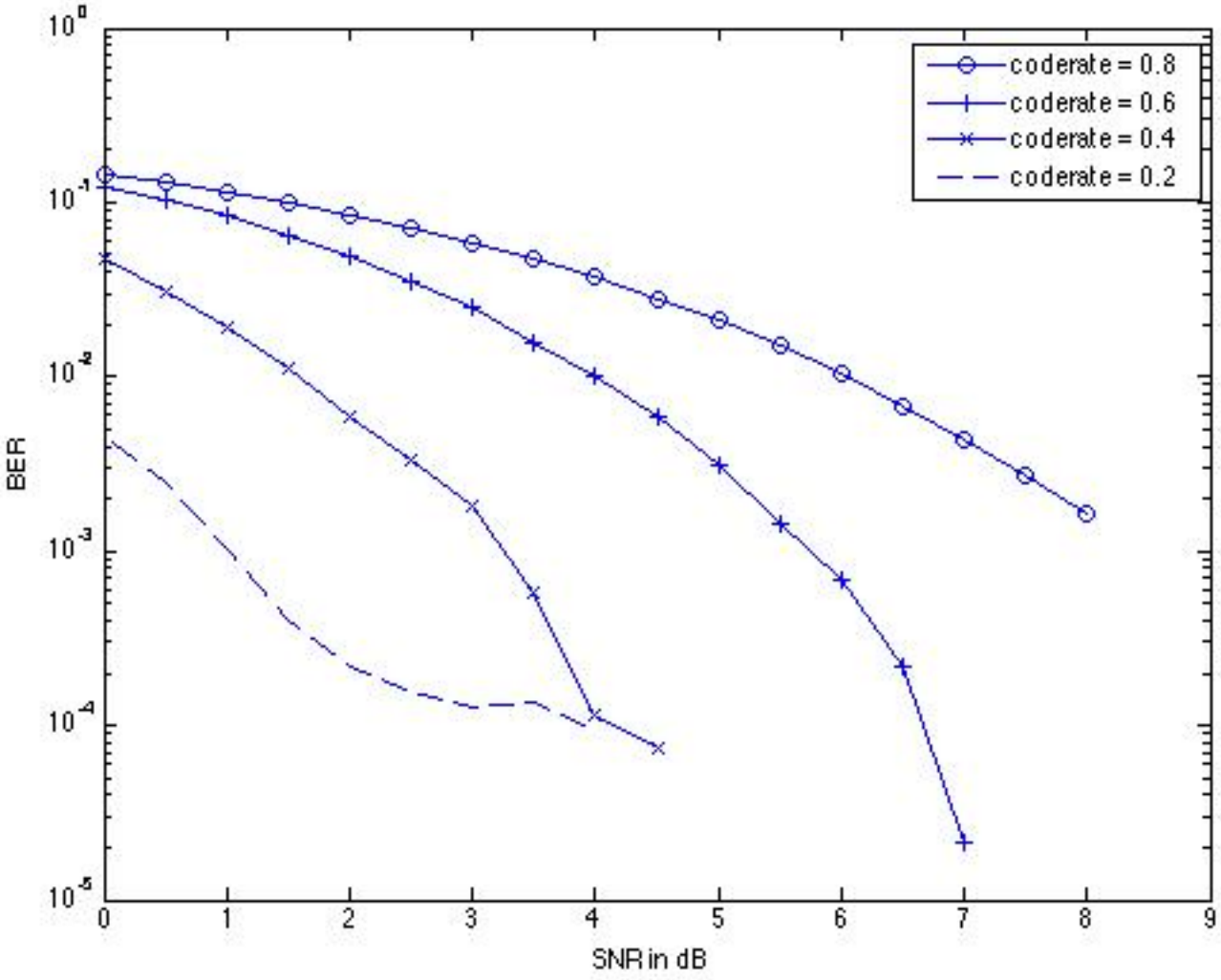

Figure 40. BER vs SNR of the four used codes with different code rates

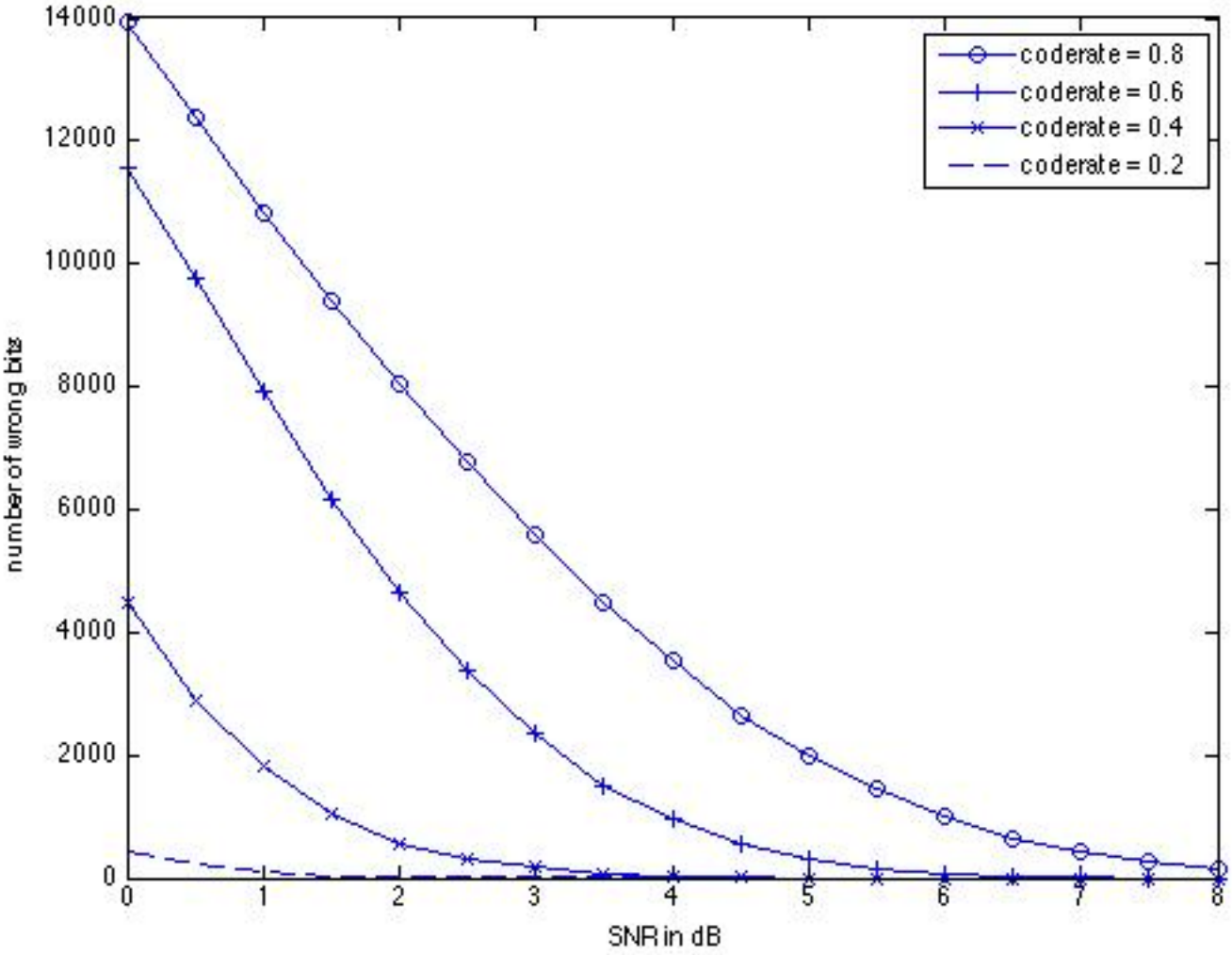

Figure 41. Total number of the wrong bits vs SNR obtained using four codes with different code rates

total 16898 codewords out of 17000 have been correctly recovered with this kind of code. By increasing the code rate we can notice how the performance decreases; in particular with a code rate equal to 0.4 the code has been able to recover 15360 codewords out of 17000. Moreover, the code with $r = 0.6$ has recovered 9607 codewords while the one with $r = 0.8$ has recovered 4048 codewords out of 17000. So we can conclude that by increasing the code rate we increase the transmission speed but decrease the decoding performance.

CHAPTER 5

CONCLUSIONS

In this thesis an example of a LDPC code decoder implementation has been presented together with a comparison between the most important decoding algorithms. Low-Density Parity-Check codes have been rediscovered in the '90s and since then they have been studied deeply and used in many different applications like cellular wireless or data storage. Nowdays they are one of the most used Block codes. In the following finally, a discussion about LDPC decoding algorithms complexity and LDPC decoder architectures is provided.

5.1 Algorithms Complexity

As already seen a LDPC code is completely described by its parity check matrix H, which defines the code size n, the code rate r and also the complexity of the decoding process. This latter issue also depends on which decoding algorithm is being used. Looking at the simulations results it is clear how the "soft" Sum-Product algorithm is the best among the three presented algorithms, even if it is the most complex either from a theoretical point of view and from an hardware implementation one. This is why in practical cases it is usually preferable to use the so called "Min-Sum" algorithm, which basically consists in a simplified version of the Sum-Product one. In the next paragraph the Min-Sum algorithm is introduced and its advantage on the Sum-Product is clarified.

5.1.1 Min-Sum Algorithm

In section 2.2.3 we have seen that the messages sent from CN to VN can be described by

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

but this expression is computationally challenging and for this reason a simpler expression is usually used. In order to find it we can split the $L_{j'\to i}$ in magnitude and sign, which means reliability of the bit and bit value (see (7)). So we can write:

$$L_{j\to i} = \alpha_{ji}\ \beta_{ji}$$

where $\alpha_{ji} = \mathrm{sign}(L_{j\to i})$ and $\beta_{ji} = |L_{j\to i}|$. The messages from CNs to VNs can be rewritten in the following form:

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

from which we get:

$$\tanh(\tfrac{1}{2}L_{i\to j}) = \prod_{j'\in N(i)\setminus j}\alpha_{j'i}\prod_{j'\in N(i)\setminus j}\tanh(\tfrac{1}{2}\beta_{j'i})$$

and so

$$L_{i\to j} = \prod_{j'\in N(i)\setminus j} \alpha_{j'i} \cdot 2\ \tanh^{-1}(\prod_{j'\in N(i)\setminus j}(\tfrac{1}{2}\beta_{j'i})) =$$

$$= \prod_{j'\in N(i)\setminus j} \alpha_{j'i} \cdot 2\ \tanh^{-1}\log^{-1}\log(\prod_{j'\in N(i)\setminus j}(\tfrac{1}{2}\beta_{j'i})) =$$

$$= \prod_{j'\in N(i)\setminus j} \alpha_{j'i} \cdot 2\ \tanh^{-1}\log^{-1}\sum_{j'\in N(i)\setminus j}\log(\tanh(\tfrac{1}{2}\beta_{j'i}))$$

Defining now

$$\phi(x) = -\log(\tanh(\tfrac{x}{2})) = \log(\tfrac{e^x+1}{e^x-1})$$

we can rewrite our expression as:

$$L_{i\to j} = \prod_{j'\in N(i)\setminus j} \alpha_{j'i} \cdot \phi(\sum_{j'\in N(i)\setminus j}\phi(\beta_{j'i}))$$

where it has been assumed that $\phi(x) = \phi^{-1}(x)$ when $x > 0$; this expression becomes the new expression for the messages sent from CNs to VNs. Looking at how ϕ is defined we clearly see that the largest term in the sum $\sum_{j'\in N(i)\setminus j}\phi(\beta_{j'i})$ corresponds to the smallest value of $\beta_{j'i}$, so assuming the entire sum is dominated by this term we can approximate

$$\phi(\sum_{j'\in N(i)\setminus j}\phi(\beta_{j'i})) \approx \phi(\phi(\min(\beta_{j'i}))) =$$

$$= \min_{j'\in N(i)\setminus j}\beta_{j'i}$$

So we can finally write the messages from CNs to VNs as:

$$\mathsf{L}_{i\to j} = \prod_{j'\in \mathsf{N}(i)\setminus j} \alpha_{j'i} \cdot \min_{j'\in \mathsf{N}(i)\setminus j} \beta_{j'i}$$

where the previous very complex expression has become much easier to compute. This is the expression used for messages from CNs to VNs on which the Min-Sum algorithm is based; its name derives from the fact that it basically performs two main operations: research of a minimum value and summation.

In practical hardware implementations the Min-Sum algorithm is usually preferred to the Sum-Product one because it requires a simpler architecture and less computation complexity. The Check-node processors only need to choose the minimum value among a set of incoming magnitude values and they do not need to perform really hard computations. On the other hand the Min-Sum algorithm usually provides worse BER and worse correcting performance and further simulations shoud be run to prove this. According to the kind of application the LDPC codes are being used for, the right algorithm should be implemented.

5.2 Architecture Complexity

For what concerns the architecture described in this work we see that it is different from a usual *serial* or *parallel* architecture because it is not based on a single processor per each node, but it presents a unique Block for all Check nodes and another one for all Bit nodes. So it can be thought of as a simple architecture, but as already said in Chapter 3, it only works

with a certain code, described by a fixed parity check matrix. Moreover it is simple only if the code size is short, otherwise too many physical connections should be implemented and the complexity of the circuit would increase a lot. In the practical decoder implementations it is a good habit to build flexible decoders that can decode several kinds of codes, with flexible length and code rate. For this reason the architecture developed in this work should be reconsidered for long codes.

CITED LITERATURE

1. MATLAB: version 7.14.0.739 (R2012a). Natick, Massachusetts, The MathWorks Inc., Feb. 2012. http://www.mathworks.com/products/matlab/.

2. MacWilliams, F. and Sloane, N.: The Theory of Error-Correcting Codes. New York, NY 10017, North-Holland Publishing Company, 1977.

3. Lin, S. and Costello, D.: Error Control Coding. Upper Saddle River, NJ 07458, Pearson Prentice Hall, 2004.

4. Johnson, S.: Iterative Error Correction. The Edinburgh Building, Cambridge CB2 8RU, UK, Cambridge, 2010.

5. Gallager, R.: Low-Density Parity-Check Codes. Doctoral dissertation, Massachusetts Institute of Technology, 1963.

6. Johnson, S.: Introducing low-density parity-check codes. Technical report, Department of Electrical and Computer Engineering, University of Newcastle, Australia. http://materias.fi.uba.ar/6624/index_files/outline_archivos/SJohnsonLDPCintro.pdf.

7. Ryan, W. and Lin, S.: Channel Codes Classical and Modern. The Edinburgh Building, Cambridge CB2 8RU, UK, Cambridge University Press, 2009.

8. Ryan, W.: An introduction to ldpc codes. Technical report, Department of Electrical and Computer Engineering, University of Arizona, 2003. http://tuk88.free.fr/LDPC/ldpcchap.pdf.

9. Wesolowski, K.: Introduction to Digital Communication Systems. The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom, Wiley, 2009.

10. Balatsoukas-Stimming, A.: Decoding of LDPC codes using the Sum-Product algorithm for the AWGN channel with BPSK modulation. Technical University of Crete, http://www.telecom.tuc.gr/ alex/lectures/lecture5.pdf, 2009.

CITED LITERATURE (continued)

11. Quartus II: Web Edition v11.1. San Jose, California, Altera Corporation, Nov. 2011. https://www.altera.com/download/software/quartus-ii-we/11.1.

12. ModelSim: Altera Starter Edition 10.0cb. San Jose, California, Altera Corporation, Nov. 2011. https://www.altera.com/download/software/modelsim-starter/11.1.

VITA

NAME: Luigi Pepe

EDUCATION: High School Leonardo da Vinci Certificate, Fasano BR, Italy 2008

B.Sc., Electronic Engineering, Politecnico di Torino, Italy, 2011

M.Sc., Electronic Engineering, Politecnico di Torino, Italy, 2013

M.Sc. Electrical and Computer Engineering, University of Illinois at Chicago, USA, 2013