FPGA Implementation of an LDPC Encoder

BY

ANTONELLO TARTAMO
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, University of Illinois at Chicago
Giuseppe Vecchi, Politecnico di Torino

To my parents for their endless support and encouragement that guided me in the walk of life.

ACKNOWLEDGMENTS

I wish to thank my UIC Advisor David Borth for his time and patience throughout this entire work. If this thesis work has been possible I owe it to him. I would also like to thank my Politecnico di Torino advisor, Giuseppe Vecchi, for all the information he gave me about this double degree project with the UIC. A special thanks to Lynn Thomas for her continuous support during all my experience at UIC. Also, I want to thank my fellow Luigi Pepe, with whom I shared this great experience. I must express my gratitude to all my friends, whose words of encouragement have taught me that no hurdle is too hard to overcome. Finally, I would like to thank the UIC staff and the administrative office of the Politecnico di Torino for the great assistance offered.

AT

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

FIGURE **PAGE**

SUMMARY

This thesis work is organized as follows. Chapter 1 introduces linear codes and low-density parity-check (LDPC) codes. Chapter 2 describes all LDPC families that will be used throughout this thesis. Chapter 3 shows all possible encoding techniques for LDPC codes. Chapter 4 describes two FPGA encoder implementations: a block circulant quasi-cyclic encoder and an IEEE 802.11n encoder. The former is used as reference to make a complexity comparison with the latter. Finally, chapter 5 presents my conclusions.

CHAPTER 1

INTRODUCTION TO LOW DENSITY PARITY CHECK CODES

1.1 Linear block codes

Channel coding is employed to correct errors due to a noisy communication channel. In this scenario linear codes add a certain amount of redundancy to the messages by means of extra bits that have to be sent in order to correct or detect possible errors. From now on we will concentrate our attention on binary linear codes, but the reader has to know that also nonbinary alphabets can be employed in the development of linear codes. A message made of *k bits* $\mathbf{u} = [\mathbf{u_1}\ \mathbf{u_2}\ \ldots\ \mathbf{u_k}]$ will be encoded to form a codeword $\mathbf{c} = [\mathbf{c_1}\ \mathbf{c_2}\ \ldots\ \mathbf{c_n}]$ with $\mathtt{n} \geq \mathtt{k}$. As shown in Figure 1, this codeword will be sent through the channel and the received message $\mathbf{r} = [\mathbf{r_1}\ \mathbf{r_2}\ \ldots\ \mathbf{r_n}]$ will be decoded in order to retrieve the original message. This decoded message **u'** could be different from the sent one due to the noisy channel.

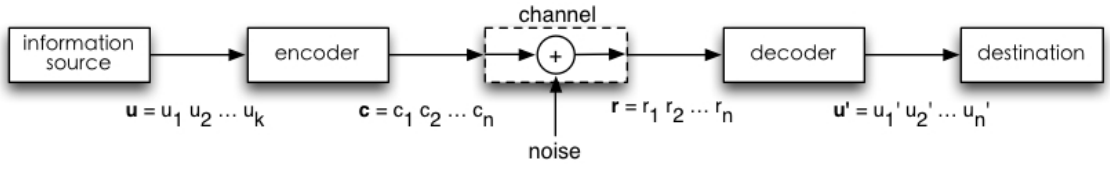

Figure 1. Communication system

We are going to describe *linear systematic codes*, in which the first k bits of the codeword correspond to the *message bits*

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

and the other *n-k* bits correspond to the so-called *check bits*

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

For $\mathbf{c} = [\mathbf{c_1}\ \mathbf{c_2}\ \ldots\ \mathbf{c_n}]$ to be a codeword the equation $\mathbf{Hc^T} = \mathbf{0}$ must be satisfied, where H is a $(n-k)$ x n *parity check matrix*:

$$H \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = Hc^T = 0 \tag{1.1}$$

with

$$\mathbf{H} = [\mathbf{A} \quad \mathbf{I_{n-k}}]$$

where A is a $(n-k)$ x k matrix and I_{n-k} is the $(n-k)$ x $(n-k)$ identity matrix. The parameter m is usually used instead of $(n-k)$. The arithmetic operations in Equation 1.1 have to be performed in *modulo-2*, that is $1+0=1,\ 1+1=0,\ -1=1$. These codes are

called linear because the *linearity* property is respected, that is, given two codewords $\mathbf{a}$ and $\mathbf{b}$, $H(\mathbf{a}+\mathbf{b})^T = H\mathbf{a}^T + H\mathbf{b}^T = 0$.

For the sake of clarity we are going to describe the easiest channel coding technique, the single parity check code (SPC). In this code only one additional bit is employed. There are two possible configurations: even parity, where the check bit is set so that there is an even number of 1s, and odd parity, where the total number of 1s has to be odd. Serial communication is one of the applications of this code, where, messages made of 7 or 8 bits plus other control bits are sent through the channel. Using the even parity configuration and a message of 7 bits, the final codeword will be:

$$\mathbf{c} = [c_1\ c_2\ c_3\ c_4\ c_5\ c_6\ c_7\ c_8]$$

where the extra bit c_8 is one of the control bits and it has to be chosen in order to satisfy the so called *parity check equation*:

$$c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 = 0 \tag{1.2}$$

where all additions are performed modulo-2.

This code is only able to detect an odd number of bit inversions. A codeword with an even number of errors will satisfy the Equation 1.2 causing a decoding error. In addition, this code cannot correct any bit errors. This is the reason why more check bits and additional parity check equations are needed.

Example 1.1 For instance, given the following parity check equations

$$c_1 + c_2 + c_4 = 0$$

$$c_2 + c_3 + c_5 = 0$$

$$c_1 + c_3 + c_6 = 0$$

these can be rewritten in a matrix form

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

where the obtained matrix is the parity-check matrix, with

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

All the codewords that satisfy these parity-check equations form a code $\mathcal{C}$ with k = 3 and n = 6.

The original message $\mathbf{u} = [u_1\ u_2\ u_3]$ will be encoded as $\mathbf{c} = [c_1\ c_2\ c_3\ c_4\ c_5\ c_6]$, where $c_1 = u_1,\ c_2 = u_2$ and $c_3 = u_3$. The other three check bits will be chosen accordingly to the equation $Hc^T = 0$,

$$c_4 = -c_1 - c_2$$

$$c_5 = -c_2 - c_3$$

$$c_6 = -c_1 - c_3$$

If the message is $u = [0\ 0\ 1]$, then $c_1 = 0,\ c_2 = 0,\ c_3 = 1$ and

$$c_4 = 0, \quad c_5 = -1 = 1, \quad c_6 = -1 = 1$$

Therefore the final codeword is $\mathbf{c} = [0\ 0\ 1\ 0\ 1\ 1]$. In every code, there are 2^k possible codewords. Thereby, in this example there are $2^3 = 8$ codewords:

000000	100101
001011	101110
010110	110011
011101	111000

A more formal definition (1) of a linear code is:

Def. Let H be any binary matrix. The linear code $\mathcal{C}$ [n,k] with parity check matrix H consists of all vectors $\mathbf{c}$ such that $H\mathbf{c}^T = 0$ (where this equation has to be interpreted modulo-2).

At the same time, $\mathbf{c} = [c_1\ c_2\ \ldots\ c_n]$ is a codeword if and only if $H\mathbf{c}^T = 0$. If the form of H is $[A \quad I_{n-k}]$, the codeword will be

$$\mathbf{c} = [\underbrace{c_1 \ldots c_k}_{\text{message bits}} \qquad \underbrace{c_{k+1} \ldots c_n}_{\text{check bits}}]$$

Given a message $\mathbf{u} = [\mathbf{u_1}\ \mathbf{u_2}\ \ldots\ \mathbf{u_k}]$ we want to generate the codeword $\mathbf{c} = [\mathbf{c_1}\ \mathbf{c_2}\ \ldots\ \mathbf{c_n}]$. To do this we can use the equation

$$\mathbf{c} = \mathbf{uG} \tag{1.3}$$

where

$$\mathbf{G} = [\mathbf{I_k} \quad \mathbf{A^T}]$$

is the *generator matrix*, where I_k is the k x k identity matrix and A^T is the k x $(n-k)$ transpose matrix of A. The steps to get the form of the matrix G from the parity check matrix H are shown in (1) (page 5). The Equation 1.3 means that a codeword $\mathbf{c}$ can be obtained as a linear combination of the rows contained in the generator matrix G. Furthermore, from the previous properties $HG^T = 0$ and $GH^T = 0$.

A code can have different generator matrices and any maximal set of linearly independent codewords forms a generator matrix. In the same way, any maximal set of linearly independent parity check equations can be employed as a parity check matrix H. It is important to remember

that the bits of a parity check equation have to sum modulo-2 to zero. In addition, a parity check matrix can have any number of parity check equations, but only n-k will be linearly independent, where n-k is the rank of the parity check matrix H, that is the number of linearly independent rows.

Example 1.2 Referring to the example 1.1, the generator matrix is:

$$G = [I_3 \quad A^T] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right]$$

The messages are encoded using $\mathbf{c} = \mathbf{u}G$, that is

$$\mathbf{c} = u_1 G_{row_1} + u_2 G_{row_2} + u_3 G_{row_3}$$

Let's encode the message $\mathbf{u} = [1\ 0\ 1]$,

$$\mathbf{c} = G_{row_1} + G_{row_3} = [1\ 0\ 0\ 1\ 0\ 1] + [0\ 0\ 1\ 0\ 1\ 1] = [1\ 0\ 1\ 1\ 1\ 0]$$

where all additions are performed modulo-2.

1.2 Parameters of a linear code

1. First of all, the *rate* or *efficiency* of a code $\mathcal{C}$ [n,k] is equal to

$$R = \frac{k}{n}$$

recalling that k is the number of message bits and n is the length of the codeword $\mathbf{c}$.

Def. Given two binary vectors $\mathbf{a} = [a_1 a_2 \dots a_n]$ and $\mathbf{b} = [b_1 b_2 \dots b_n]$, the *Hamming distance* is the number of places where they differ.

E.g. [1 0 1 1 0 1] and [0 1 1 1 0 0], dist = 3

Def. Given a binary vector $\mathbf{a} = [a_1 a_2 \dots a_n]$, the *Hamming weight* is the number of non-zero bits.

E.g. [1 0 1 1 0 1], weight = 4

In addition, the distance between two binary vectors is equal to the weight of their difference: $\mathbf{dist}(\mathbf{a}, \mathbf{b}) = \mathit{weight}(\mathbf{a} - \mathbf{b})$.

2. Given the previous definitions, an important parameter is the *minimum distance* of a code $\mathcal{C}$ (1):

$$d_{min} = \min \mathbf{dist}(\mathbf{c}_x, \mathbf{c}_y) = \min \mathit{weight}(\mathbf{c}_x - \mathbf{c}_y) \qquad \mathbf{c}_x \in \mathcal{C}, \mathbf{c}_y \in \mathcal{C}, \mathbf{c}_x \neq \mathbf{c}_y$$

where $\mathbf{c_x}$ and $\mathbf{c}_y$ are two codewords from code $\mathcal{C}$.

It is not necessary to determine the distance between every pair of codewords, because if both $\mathbf{c}_x$ and $\mathbf{c}_y$ belong to the same code, the modulo-2 difference $\mathbf{c}_x - \mathbf{c}_y$ is also a codeword, therefore

$$\mathbf{d_{min}} = \min_{c \in \mathcal{C}, c \neq 0} weight(\mathbf{c})$$

This means that the minimum distance of a code is the minimum weight between all non-zero codewords.

3. The error correction capability (1) of a code $\mathcal{C}$ with minimum distance d_{min} is of

$$\left\lfloor \frac{1}{2}(d_{min} - 1) \right\rfloor \quad \text{errors}$$

where $\lfloor x \rfloor$ is the floor function. If d_{min} is even, $\frac{1}{2}(d_{min} - 2)$ errors can be corrected and $\frac{1}{2}d_{min}$ errors can be detected.

Example 1.3 The modulo-2 difference between two codewords from example 1.1:

[1 0 0 1 0 1] - [0 1 0 1 1 0] = [1 -1 0 0 -1 1] = [1 1 0 0 1 1]

is another codeword. In this case, $d_{min} = 3$, hence only one error can be corrected.

A linear code can be represented graphically using a *Tanner graph* (2), which is a bipartite graph where the two kinds of vertices are bit nodes and check nodes. Every row in the parity check matrix H is represented by a check node, while every column of H is represented by a bit node. An edge connects a check node with a bit node if $H_{r,c} = 1$,

that is the bit c in the parity check equation r is 1. Obviously, the number of edges is equal to the number of 1s in the parity check matrix H.

4. A *cycle* is a path that starts from one node and ends on the same node, passing through each other vertex only once. The number of edges contained in a path is the *cycle length* and the minimum cycle length in a Tanner graph is called *girth*. In literature the term n-cycle is used to indicate a cycle of length n. Moreover, the *degree of a node* is the number of edges that are connected to that node.

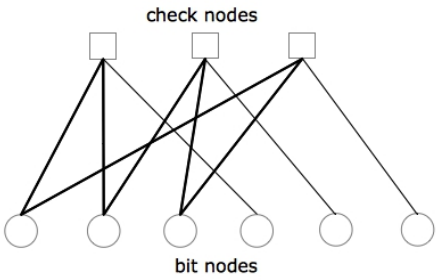

Figure 2. Tanner graph example 1.1

Example 1.4 The Tanner graph of the example 1.1 can be seen in Figure 2, where the path in bold is a 6-cycle.

1.3 Error detection

Let's assume the message $\mathbf{u} = [\mathbf{u_1}\ \mathbf{u_2}\ \ldots\ \mathbf{u_k}]$ is encoded into the codeword $\mathbf{c} = [\mathbf{c_1}\ \mathbf{c_2}\ \ldots\ \mathbf{c_n}]$. This codeword is sent through a noisy channel, and the received codeword $\mathbf{r} = [\mathbf{r_1}\ \mathbf{r_2}\ \ldots\ \mathbf{r_n}]$ could be different from $\mathbf{c}$.

We can define the *error vector* as

$$\mathbf{e} = \mathbf{r} - \mathbf{c} = [e_1\ e_2\ \ldots\ e_n]$$

and p as the error probability of having $e_i = 1$. This happens when sending a bit b the reversed value $\overline{b}$ is received, with b = 0 or 1.

To check if a received vector $\mathbf{r}$ contains errors we can compute the $(n - k)$ vector:

$$\mathbf{S} = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{n-k} \end{pmatrix} = H\mathbf{r}^T$$

that is called *syndrome* of $\mathbf{r}$. By means of this vector we can know which parity check equations are not satisfied. If $H\mathbf{r}^T = \mathbf{0}$, $\mathbf{r}$ is a correct codeword in $\mathcal{C}$.

Given $\mathbf{r} = \mathbf{c} + \mathbf{e}$ with $\mathbf{c} \in \mathcal{C}$

$$S = H\mathbf{r}^T = H(\mathbf{c} + \mathbf{e})^T$$

and thanks to the linearity property

$$S = Hc^T + He^T$$

where by definition $Hc^T = 0$, so

$$S = He^T = \sum_i e_i \mathbf{H}_i$$

with $\mathbf{H}_i$ equal to the i-th column of the parity-check matrix. In plain words, the syndrome is the sum of the parity-check columns where the errors happen.

If the probability of error is lower than $\frac{1}{2}$, an error vector of lower weight is more likely to be obtained than one with higher weight, therefore a possible way to decode is to choose the codeword with the lowest Hamming distance (the nearest one) with respect to the received codeword. This decoder is called *maximum likelihood (ML)* decoder. The simplest decoding scheme is to use a brute force approach. That is, given a received vector, we can take as decoded codeword the closest vector between all possible 2^k codewords of the code. The problem is that this method becomes unfeasible as k becomes larger.

A fundamental concept, related to a noisy communication system, is *Shannon's channel coding theorem*, which states that if the code rate $R = k/n$ is less than the capacity of the channel, an error correction code that is able to make the probability of error arbitrarily small exists. In other words, we could also say that given a noise level n_{th} such that the code rate $R = C(n_{th})$, n_{th} represents the noise threshold for all codes with rate R and it is usually called the *Shannon limit*. This theorem is important because it proves the existence of good codes,

which can transmit information over a noisy channel with no errors (provided that the code rate is less than the channel capacity). However, the original paper by Shannon failed to describe any explicit codes achieving channel capacity. The capacity of the channel is the upper bound on the amount of information that can be sent through a channel and it is measured in bits per second.

1.4 LDPC codes

Low density parity check (LDPC) codes are a particular kind of linear codes in which the number of 1s in the parity check matrix H is very small. It is this characteristic that allows to have near-capacity performance and low complexity encoders/decoders.

An LDPC code is *regular* if its parity check matrix $H_{m\,x\,n}$ contains w_c 1s in each column and w_r 1s in each row, that is every bit is just present in w_c parity check equations and w_r bits are present in each parity check equation. Thereby, there will be $nw_c = mw_r$ ones in the parity check matrix.

An *irregular* LDPC code has not a fixed number of 1s. In this case the *degree distribution* has to be defined: we indicate with λ_i the fraction of edges connected to degree-i bit nodes with respect to the total number of edges and with ρ_i the fraction of edges connected to degree-i check nodes with respect to the total number of edges.

The polynomials that describe the degree-distribution are:

$$\lambda(x) = \sum_{i=1}^{d_c} \lambda_i x^{i-1}$$

$$\rho(x) = \sum_{i=1}^{d_r} \rho_i x^{i-1}$$

where d_c indicates the maximum bit node degree, while d_r is the maximum check node degree. It is easy to show that $\sum_{i=1}^{d_c} \lambda_i = 1$ and $\sum_{i=1}^{d_r} \rho_i = 1$.

Example 1.5 For the irregular code of example 1.1,

$$\lambda(x) = \lambda_1 + \lambda_2 x, \quad \text{with } \lambda_1 = \frac{3}{9} = \frac{1}{3} \quad \lambda_2 = \frac{6}{9} = \frac{2}{3}$$

and

$$\rho(x) = \rho_3 x^2 = x^2, \quad \rho_3 = 1$$

Another approach to determine the degree-distribution is to compute ν_i, the fraction of columns of weight i, and h_i, the fraction of rows of weight i. In this case there are $n(\sum_i i \cdot \nu_i) = m(\sum_i i \cdot h_i)$ ones in the parity check matrix.

Example 1.6 An irregular code with parity check matrix

$$H = \left[\begin{array}{cc|ccc} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{array}\right]$$

has $\nu_1 = \frac{3}{5}$, $\nu_2 = \frac{2}{5}$ and $h_2 = \frac{2}{3}$, $h_3 = \frac{1}{3}$.

1.5 LDPC code-construction

There are different approaches to construct a LDPC code. We can choose between algorithmic and mathematics techniques. The first class includes all computer-based methods, while the second one is about graph theory, combinatorial design or finite geometries.

We will now concentrate on algorithmic methods. First of all, we have to know that the LDPC concept was first introduced by Gallager in his PhD thesis (3) at MIT in 1963. His original definition of a regular code is based on the following matrix structure:

$$H = \begin{bmatrix} H_1 \\ H_2 \\ \vdots \\ H_{w_c} \end{bmatrix}$$

where H is the usual m x n matrix with w_c ones in each column and w_r ones in each row. Each of the submatrices H_i, $i = 1 \ \ldots \ w_c$ is a r x rw_r matrix with row weight w_r, unitary column weight and where r is just a constant greater than one.

$\mathbf{H_1}$ has to be created as follows: the j-th row ($j = 1,\ 2,\ \ldots,\ r$) contains all the w_r 1s in the positions from $(j-1)w_r + 1$ to jw_r. The other submatrices are obtained from random column permutations of H_1.

Example 1.7 A parity check matrix with $w_r = 3$, $w_c = 2$ and $r = 3$:

$$H = \left[\begin{array}{ccccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ \hline 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{array}\right]$$

Another construction method is the one invented about 35 years later by *MacKay* (4) and *Neal*. In this case the parity check matrix is created adding one column at a time from left to right depending on the degree distribution we want to obtain. The positions of 1s in any new column are chosen randomly between the rows that are still free.

Example 1.8 A parity check matrix with $w_r = 3$ and $w_c = 2$, generated by means of the MacKay and Neal algorithm could be:

$$H = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

where adding the 8th column we can choose between 2nd, 3rd, 5th and 6th rows. In this example the 2nd and 6th rows have been chosen.

Another approach is to work directly on the Tanner graph. A possible algorithm is the *Progressive Edge Growth* (PEG) that maximizes the local girth in order to guarantee good iterative decoding performance. It starts with a list of variable nodes, check nodes and the desired degree distribution, then the algorithm adds progressively one edge at a time in order to connect a variable node with a check node trying to maximize the local cycles length. Another alternative is the *bit filling* algorithm, that is similar to the PEG algorithm, but in this case an edge will be added if the minimum cycle length requirement is satisfied without maximizing the local girth.

A different technique is to start from a small matrix H_b of size m_b x n_b and then expand it to obtain a larger matrix of size Zm_b x Zn_b. Each item of the base matrix H_b is replaced

with a Z x Z submatrix. If each submatrix is a weight-1 circulant matrix, a quasi-cyclic code is obtained. The base matrix is equivalent to a *protograph* graph, which is a bipartite graph that is copied for a certain number of times and whose edges are permuted according to the degree distribution to obtain the expanded graph. In a protograph there can be more than one edge between a variable node and a check node, these *parallel* edges will disappear in the final expanded graph, for example creating a weight-π circulant matrix for a set of π parallel edges.

Example 1.9 A common base matrix is the one of the Accumulate Repeat 3 Accumulate (AR3A) code:

$$H_b = \begin{bmatrix} 1 & 1 & 2 & 0 & 0 \\ 0 & 2 & 1 & 2 & 0 \\ 0 & 1 & 2 & 0 & 2 \end{bmatrix}$$

Its protograph can be seen in Figure 3. Another base matrix is the one of the AR4A code:

$$H_b = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 3 & 1 & 2 & 0 \\ 0 & 1 & 3 & 0 & 2 \end{bmatrix}$$

where the corresponding protograph is shown in Figure 4.

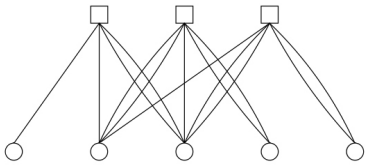

Figure 3. AR3A protograph

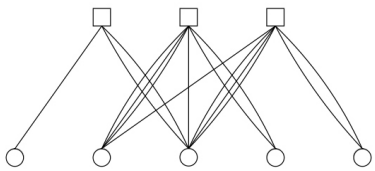

Figure 4. AR4A protograph

1.6 Iterative Decoding

1.6.1 Message passing

The decoding algorithms used for LDPC codes are based on the principle of the *message passing*. In order to make this concept clear I will use the soldier counting problem as in (5) (chapter 5). This is further described in MacKay's book (6).

Referring to Figure 5, each soldier has to respect the following rules:

- If there are not soldiers in front or behind you pass the number 1 to the only soldier close to you. This is the case of the first and last soldiers in the line.
- If a number is received from another soldier, add 1 to it and pass the result to the soldier in front or behind you.

Obviously, a meaningful result will be obtained only at the end of the line.

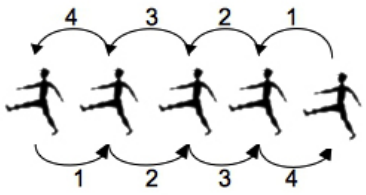

Figure 5. Soldiers line

Let's consider now the Figure 6. In this case the message that a soldier A has to pass to a neighbor soldier B is the sum of all incoming messages, plus one for itself, minus the message that the soldier B has passed to A. We can summarize this process as:

$$M_{A\to B} = \sum_{X\in N(A)} M_{X\to A} + M_A - M_{B\to A} = \sum_{X\in N(A)-B} M_{X\to A} + M_A \tag{1.4}$$

where N(A) is the set of all neighbors of the soldier A, $M_{A\to B}$ is the *extrinsic information*, that is information that a node does not already have and M_A is the *intrinsic information*. In this example M_A is equal to one.

For instance, the message from the soldier *S3* to the soldier *S2* in Figure 6 will be

$$M_{S3\to S2} = (M_{S4\to S3} + M_{S6\to S3} + M_{S2\to S3}) + M_{S3} - M_{S2\to S3} = (M_{S4\to S3} + M_{S6\to S3}) + M_{S3}$$

$$M_{S3\to S2} = (2+2+2) + 1 - 2 = (2+2) + 1 = 5$$

It is easy to show that if a cycle is present the message will be added infinitely and so the algorithm will not work.

1.6.2 Gallager Sum-Product Decoding

The goal of this decoding algorithm is to compute the *maximum a posteriori probability* (MAP) that a specific bit in the transmitted codeword $\mathbf{c} = [c_1\ c_2\ \ldots\ c_n]$ is equal to 1, given the received codeword $\mathbf{r} = [r_1\ r_2\ \ldots\ r_n]$. The codeword $\mathbf{c}$ is the one that has to be sent

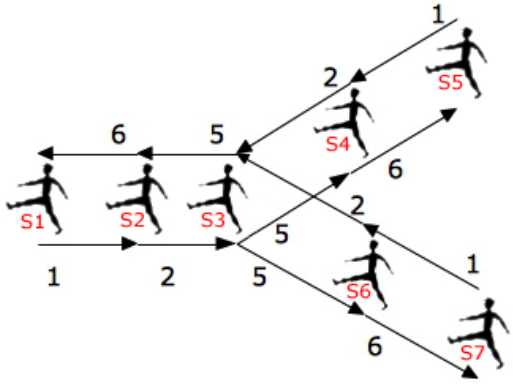

Figure 6. Soldiers tree

through the channel, while $\mathbf{r}$ is the one received at the output of the channel. The a posteriori probability for a generic codeword bit c_i can be defined as:

$$P(c_i = 1|\mathbf{r})$$

while the *log-likelihood ratio* (LLR) is computed as

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

where log is the natural logarithm. If $P(c_i = 0|\mathbf{r}) > P(c_i = 1|\mathbf{r})$, $L(c_i|\mathbf{r})$ is positive. Instead if $P(c_i = 1|\mathbf{r}) > P(c_i = 0|\mathbf{r})$, $L(c_i|\mathbf{r})$ is negative. Furthermore, the greater one term is compared to the other, the higher the magnitude of $L(c_i|\mathbf{r})$ will be. In this way the sign of the log-likelihood

ratio will tell us the value that the bit c_i has to take on and its magnitude represents the reliability on taking that decision.

This algorithm is a message-passing algorithm because the messages are sent along the edges of a Tanner graph and it is iterative because it requires a certain number of steps to reach the final result.

Each bit node and each check node can be thought as independent decoders. The bit node decoder (Figure 7) has an edge L_j coming directly from the channel. L_j represents the intrinsic information and it is called the *a priori probability*, because it is known in advance due to its strict dependence from the channel parameters. As in Equation 1.4, the extrinsic information $L_{j\to i}$ is computed only using the messages coming from the neighbor check nodes and the channel. The message $L_{i\to j}$ is indeed not used, since it should otherwise be subtracted.

The same happens for the check node decoder (Figure 8). Indeed, $L_{j\to i}$ is not needed in the computation of the extrinsic information $L_{i\to j}$.

The log-likelihood ratios returned by this algorithm are correct only if the Tanner graph is cycle free. If this is not possible, it will be just an approximation of the a posteriori probability for each bit. This is due to the *independence assumption*, according to which all log-likelihood ratios received at one node are independent. This assumption is not more respected when the intrinsic information L_j reaches the corresponding bit node b_j by means of a cycle in the Tanner graph. This condition occurs if the number of iterations is equal or greater than half of the graph girth.

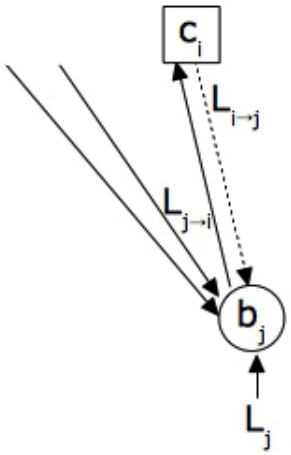

Figure 7. Bit node

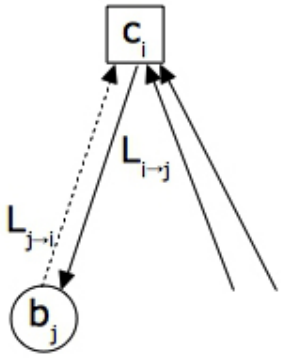

Figure 8. Check node

According to (5) a bit node can be seen as a *repetition code*. This code has a single information bit that is repeated n times. Thus, only two codewords are possible: $[0\ 0\ \ldots\ 0]$ and $[1\ 1\ \ldots\ 1]$.

Let's assume that a length d vector $\mathbf{r}$ contains the bits received after a bit c has been sent d times through a memoryless channel. The log-likelihood ratio is

$$L(c|\mathbf{r}) = \sum_{l=0}^{d-1} L(r_l|c)$$

where $L(r_l|c) = \log \frac{P(r_l|c=0)}{P(r_l|c=1)}$. The decision will be $\overline{c} = 0$ if $L(c|\mathbf{r}) \geq 0$, and $\overline{c} = 1$ if $L(c|\mathbf{r}) < 0$.

In our case, the previous expression has to be rewritten as

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

that has the same structure of the message passing Equation 1.4. The extrinsic information $L_{j\to i}$ from a bit node b_j to a check node c_i is equal to the sum of the LLRs coming from the neighbor check nodes excluded c_i and the intrinsic information. It means that if the majority of LLRs are positive, the final value of $L_{j\to i}$ will be positive and so we are dealing with a zero bit. Similarly, if the majority of LLRs is negative, the corresponding bit will be one.

The intrinsic information L_j is computed as

$$L_j = L(c_j|r_j) = \log\left(\frac{P(c_j = 0|r_j)}{P(c_j = 1|r_j)}\right)$$

At the end of the algorithm the total LLR has to be found as:

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

Let's now model a check node as a single parity check (SPC) code, in which there is just one parity bit. First of all, a result present in the Gallager's PhD thesis states that given a vector of m independent binary digits with $P_l = P\{l^{th}$ bit in the vector is $1\}$, the probability that an even number of digits is 1 can be computed as

$$\frac{1}{2} + \frac{1}{2}\prod_{l=1}^{m}(1 - 2P_l)$$

Let's consider the transmission of a SPC codeword $\mathbf{c}$ of length n through a memoryless channel and define with $\mathbf{r}$ the received vector. There must be an even number of 1s in the codeword $\mathbf{c}$. The probability that a generic bit c_l in the codeword $\mathbf{c}$ is equal to 0 is

$$P(c_l = 0|\mathbf{r}) = P\{\{\mathbf{c} - \{c_l\}\} \text{ contains an even number of 1s}|\mathbf{r}\}$$

it means that if c_l is equal to 0 all the other bits have to contain an even number of 1s to respect the SPC constraint and thanks to Gallager's result we can write that as

$$P(c_l = 0|\mathbf{r}) = \frac{1}{2} + \frac{1}{2}\prod_{j,\, j\neq l}(1 - 2P(c_j = 1|r_j))$$

and using the equivalence $P(c_l = 0|\mathbf{r}) = 1 - P(c_l = 1|\mathbf{r})$

$$1 - P(c_l = 1|\mathbf{r}) = \frac{1}{2} + \frac{1}{2}\prod_{j,\, j\neq l}(1 - 2P(c_j = 1|r_j))$$

$$1 - 2P(c_l = 1|\mathbf{r}) = \prod_{j,\, j\neq l}(1 - 2P(c_j = 1|r_j))$$

Then thanks to the relation

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

we can write

$$\tanh\left(\frac{1}{2}L(c_l|\mathbf{r})\right) = \prod_{j,\, j\neq l}\tanh\left(\frac{1}{2}L(c_j|r_j)\right)$$

or better

$$L(c_l|\mathbf{r}) = 2\tanh^{-1}\left(\prod_{j,\, j\neq l}\tanh\left(\frac{1}{2}L(c_j|r_j)\right)\right)$$

The bit c_l will be equal to 0 if $L(c_l|\mathbf{r}) \geq 0$ and $c_l = 1$ if $L(c_l|\mathbf{r}) < 0$. The final equation can be adapted for a LDPC code obtaining the extrinsic message that will be sent from a check node c_i to a bit node b_j

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

The decoding algorithm can be summarized in the following steps:

1. **Initialization step:** for all $j = 1, 2, \ldots, n$, initialize L_j as follows

$$L_j = L(c_j|r_j) = \log\left(\frac{P(c_j = 0|r_j)}{P(c_j = 1|r_j)}\right)$$

   and for each pair (i,j) for which the corresponding parity check matrix item $H_{i,j}$ is equal to 1 set $L_{j\to i} = L_j$.

2. **Check messages:** for each check node compute

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

3. **Bit messages:** for each bit node compute

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

4. **Total LLR:** for j=1, 2, ..., n compute

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

5. **Test:** for $j = 1, 2, \ldots, n$, determine the decoded codeword $\mathbf{d}$ setting $d_j = 0$ if $L_j^{total} \geq 0$ and $d_j = 1$ otherwise. If $\mathbf{Hd^T} = \mathbf{0}$ ($\mathbf{d}$ is a valid codeword) or the maximum number of iterations is reached stop the algorithm, otherwise go back to step 2.

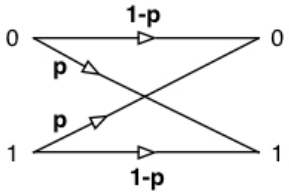

Figure 9. Binary symmetric channel

Example 1.10 The initialization process depends on the employed channel model. For the binary symmetric channel (BSC) in Figure 9, we have probability of error

$$p = P(\text{receive } 1 | \ 0 \text{ was sent}) = P(\text{receive } 0 | \ 1 \text{ was sent})$$

Thus, assuming that the values of c_j are equally likely ($P(c_j = 0) = P(c_j = 1)$) the LLR for $r_j = 1$ will be

$$L(c_j|r_j = 1) = \log\left(\frac{P(c_j = 0|r_j = 1)}{P(c_j = 1|r_j = 1)}\right)$$

Using the Bayes theorem

$$P(c_j = 0|r_j = 1) = \frac{P(r_j = 1|c_j = 0)P(c_j = 0)}{P(r_j = 1)}$$
$$P(c_j = 1|r_j = 1) = \frac{P(r_j = 1|c_j = 1)P(c_j = 1)}{P(r_j = 1)}$$

Therefore,

$$L(c_j|r_j = 1) = \log\left(\frac{P(r_j = 1|c_j = 0)}{P(r_j = 1|c_j = 1)}\right) = \log\left(\frac{p}{1-p}\right)$$

Similar operations are done for $r_j = 0$

$$L(c_j|r_j = 0) = \log\left(\frac{P(c_j = 0|r_j = 0)}{P(c_j = 1|r_j = 0)}\right) = \log\left(\frac{P(r_j = 0|c_j = 0)}{P(r_j = 0|c_j = 1)}\right) = \log\left(\frac{1-p}{p}\right)$$

CHAPTER 2

LDPC CLASSES

There are three main classes of LDPC codes: random, cyclic and quasi-cyclic. The random codes have the best decoding performance, but they have not a predefined structure. Hence, these codes are really hard to encode and decode. Some construction techniques for random codes have been shown in section 1.5.

2.1 Quasi-cyclic codes

In a quasi-cyclic code a cyclic shift of a codeword by x positions gives another codeword of the code. A cyclic code is a subclass of quasi-cyclic codes where x is equal to 1. In quasi-cyclic codes the parity check matrix H is made of an array of circulants.

$$H = \begin{bmatrix} C_{1,1} & \dots & C_{1,n} \\ \vdots & \vdots & \vdots \\ C_{m,1} & \dots & C_{m,n} \end{bmatrix}$$

Each circulant $C_{i,j}$ is a square matrix created such that a row r is a cyclic shift of the above row *r-1* and the first row is a cyclic shift of the last row.

2.1.1 Row-circulant QC codes

For the sake of simplicity, we are going to introduce a quasi-cyclic code with only one row of circulants:

$$H = [C_1\ C_2\ \dots\ C_l]$$

where $C_1,\ C_2,\ \dots C_l$ are Z x Z circulant matrices.

A circulant matrix can be represented by the polynomial

$$a(x) = a_0 + a_1 x + \cdots + a_{Z-1} x^{Z-1}$$

and so the matrix H is characterized by the polynomials

$$a_1(x)\ a_2(x)\ \dots\ a_l(x).$$

Let's assume that C_l is invertible, but it is sufficient that at least one of these circulant matrices is invertible. Thus, the generator matrix is:

$$G = \begin{bmatrix} & (C_l^{-1}C_1)^T \\ I_{Z(l-1)} & (C_l^{-1}C_2)^T \\ & \vdots \\ & (C_l^{-1}C_{l-1})^T \end{bmatrix} \tag{2.1}$$

The resulting code $\mathcal{C}$ [n, k] has $n = Zl$ and $k = Z(l-1)$. For further information see (7) (page 90).

Example 2.1 A quasi-cyclic code with Z = 4 and l = 2. The first circulant matrix is represented by $a_1 = 1 + x^3$ and the second circulant by $a_2 = 1 + x$.

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

This results in a [8,4] code with a rate $R = \frac{4}{8} = \frac{1}{2}$.

2.1.2 Block-circulant QC codes

In this part of the section I will just give a hint about block circulant quasi cyclic codes. A possible implementation is the *array code* (8). The parity check matrix of this code is the following:

$$H = \begin{bmatrix} I & I & I & \dots & I \\ I & I_1 & I_2 & \dots & I_{q-1} \\ I & I_2 & I_4 & \dots & I_{2(q-1)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ I & I_{r-1} & I_{(r-1)2} & \dots & I_{(r-1)(q-1)} \end{bmatrix}$$

where q is a prime number, r is a positive number such that $r \leq q$ and I_x is a cyclic shift to the right by x positions of a L x L identity matrix with $0 \leq x < L$. The parity check matrix will have size $m = L\,r$ and $n = L\,q$ with $L = q$. The code rate is $R = \frac{k}{n} = \frac{n-m}{n} = 1 - \frac{m}{n} \geq 1 - \frac{r}{q}$ due to linear dependence between the rows of H.

This is a regular code because the parity check matrix contains r-weight columns and q-weight rows, since each circulant has weight-1 rows/columns and there are r circulants in each column and q circulants in each row. The identity matrix can be seen as a circulant I_0.

Another way (9) to create a quasi cyclic code is to choose a prime number m, two positive numbers a and b, $a < m$, $b < m$, with multiplicative orders j and k, respectively. The multiplicative order is the smallest positive number k such that $a^k = 1 \bmod m$. The parity check matrix will be a j x k matrix

$$H = \begin{bmatrix} I_1 & I_a & I_{a^2} & \dots & I_{a^{k-1}} \\ I_b & I_{ab} & I_{a^2b} & \dots & I_{a^{k-1}b} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ I_{b^{j-1}} & I_{ab^{j-1}} & I_{a^2b^{j-1}} & \dots & I_{a^{k-1}b^{j-1}} \end{bmatrix}$$

where, as before, I_x is a cyclic shift to the right by x positions of a m x m identity matrix. Also in this case, we are dealing with a regular LDPC code, because of the k-weight rows and the j-weight columns. The code rate is $R \geq 1 - \frac{j}{k}$ due to the linear dependence between the rows of H.

In order to make this code irregular (9) we can start from a regular j x k parity check matrix and then for all the rows except the last two rows $j-2$ and $j-1$ we replace the last $j-i-1$ circulant matrices with m x m zero matrices, where i is the row on which we are working on and $0 \leq i \leq j-3$.

For an efficient encoding, a modified array code (10) has been proposed, which has the following parity check matrix:

$$\begin{vmatrix} I_{(j-1)(k-j)} & I_{(j-1)(k-j-1)} & I_{(j-1)(k-j-2)} & \dots & I_{j-1} & I & 0 & 0 & 0 & \dots & 0 \\ I_{(j-2)(k-j+1)} & I_{(j-2)(k-j)} & I_{(j-1)(k-j-1)} & \dots & \dots & I_{j-2} & I & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \dots & \dots & \dots & \dots & I & 0 & \dots & 0 \\ I_{2(k-3)} & I_{2(k-4)} & I_{2(k-5)} & \dots & \dots & \dots & \dots & I_2 & I & 0 & 0 \\ I_{(k-2)} & I_{(k-3)} & I_{(k-4)} & \dots & \dots & \dots & \dots & \dots & I_1 & I & 0 \\ I & I & I & \dots & \dots & \dots & \dots & \dots & \dots & \dots & I \end{vmatrix} \tag{2.2}$$

where I_x is a L x L identity matrix cyclically shifted to the right by x positions. A prime number q has to be chosen and $q \geq k \geq j$. This is an irregular LDPC code with L = q and without 4-length cycles. The parity check matrix $H_{m \times n}$, with $m = q\,j$ and $n = q\,k$, is full rank because of its lower triangular form and the code rate is $R = 1 - \frac{j}{k}$.

To make the matrix creation clearer I have created a simple MATLAB® (11) script:

```
q = 30;
```

```
v_p = primes(q);
% q represents the circulant size and it has to be a prime
% take the closest prime to q
q = v_p(end);
% n = k
k = q-1;
% m = n-k = j
% R = k/n = (n-m)/n = 1-j/k
R = 2/3;
j = floor((1-R)*k);

% NaN = all-zero matrix
% create the base matrix with shift amounts
Hb = nan(j,k);
r1 = 1;
x = 0;
for r = j-1:-1:1
    c1 = 1;
    for c = k-j+x:-1:0
        Hb(r1,c1) = r*c;
        c1 = c1+1;
    end
    x = x+1;
    r1 = r1+1;
end
Hb(j,:) = 0;

% create parity check matrix
H = zeros(q*j,q*k);
for r = 1:j
```

```
    for c = 1:k
        if isnan(Hb(r,c))
            H(q*(r-1)+1:q*r,q*(c-1)+1:q*c) = zeros(q,q);
        else
            H(q*(r-1)+1:q*r,q*(c-1)+1:q*c) = circshift(eye(q),[0 Hb(r,c)]);
        end
    end
end
```

An example of a 348 x 696 parity check matrix obtained with this algorithm is shown in Figure 10.

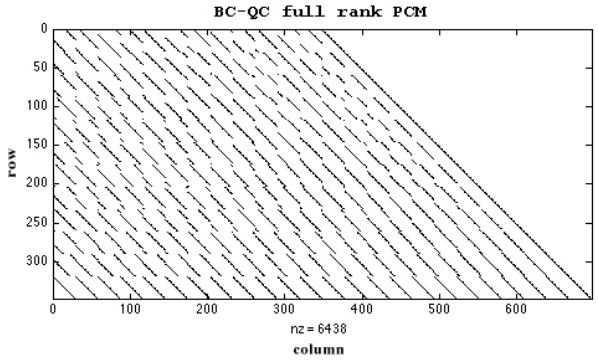

Figure 10. BC-QC full rank PCM

The generator matrix G can be easily computed from a full rank parity matrix $H_{(n-k)\ x\ n}$ in the following way: first of all, a matrix M is created with the columns from $k+1$ to n of the matrix H, then the inverse matrix of M is computed. Finally, the systematic version of the matrix H is found as $H_{sys} = M^{-1}H$. The matrix H_{sys} has the form $[A_{(n-k)\ x\ k} \quad I_{(n-k)}]$, thereby, the systematic form of the generator matrix is $G_{sys} = [I_k \quad A^T]$.

2.2 Repeat-Accumulate codes

Repeat Accumulate (RA) codes are another kind of LDPC codes, where the parity check matrix has the form

$$H = [H_1\ H_2]$$

where H_1 is a m x k sparse matrix and H_2 is a m x m matrix with $(m-1)$ weight-2 columns (only the last column has unitary weight) arranged in a staircase pattern as the following 5 x 5 matrix:

$$H_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

A regular RA (c,a) code has fixed weight-c columns and fixed weight-r rows in the matrix H_1. Nevertheless, the matrix H will be LDPC irregular due to the last column of weight-1 in the matrix H_2. An irregular RA will have a matrix H_1 with an irregular degree distribution.

Example 2.2 A (2,2) regular RA parity check matrix for a [8,4] code could be:

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

The first four columns are related to the message bits. The parity check equations are:

$$c_5 = c_1 + c_3 \qquad c_6 = c_5 + c_2 + c_4$$

$$c_7 = c_6 + c_2 + c_3 \qquad c_8 = c_7 + c_1 + c_4.$$

Thanks to the dual-diagonal form of the matrix H_2, each parity check bit can be computed using only the message bits and the previously computed parity check bit in a recursive manner.

To make it clearer, given the matrix

$$H = [H_1 \mid H_2] = \left[\begin{array}{cccc|cccc} h_{11,1} & h_{11,2} & \dots & h_{11,k} & 1 & 0 & \dots & 0 \\ h_{12,1} & h_{12,2} & \dots & h_{12,k} & 1 & 1 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & 0 & \ddots & \ddots & 0 \\ h_{1m,1} & h_{1m,2} & \dots & h_{1m,k} & 0 & \dots & 1 & 1 \end{array}\right]$$

and using the equation $\mathbf{H}\mathbf{c}^{\mathsf{T}} = \mathbf{0}$ the parity check bits can be found as

$$p_1 = h_{11,1}u_1 + h_{11,2}u_2 + \cdots + h_{11,k}u_k$$

$$p_2 = h_{12,1}u_1 + h_{12,2}u_2 + \cdots + h_{12,k}u_k + p_1$$

$$\vdots$$

$$p_m = h_{1m,1}u_1 + h_{1m,2}u_2 + \cdots + h_{1m,k}u_k + p_{m-1}$$

with all additions performed modulo-2.

CHAPTER 3

ENCODING TECHNIQUES

We have seen in Chapter 1 that when the parity check matrix H is in the systematic form $H_{sys} = [A_{(n-k)\ x\ k} \quad I_{n-k}]$, the generator matrix is equal to $G_{sys} = [I_k \quad A^T]$. Given any parity check matrix H, the systematic form can be found using Gauss-Jordan elimination. This operation has to be done only once and then the encoder will work on the generator matrix, while the decoder will work on the parity check matrix. After that the encoding can be done using the well known expression $\mathbf{c} = \mathbf{u}G$. The main disadvantage is that most of the time G is not sparse and the matrix multiplication will have order n^2 complexity.

3.1 Richardson/Urbanke Encoding Algorithm

The goal of this method (12) is to encode a message using the parity-check matrix instead of the generator matrix. First of all, the matrix is put into *approximate lower triangular form* H_{alt} (Figure 11) by means of row and column permutations. The matrix T is a lower triangular matrix with all ones on the main diagonal and all zeros above the diagonal. The g rows at the bottom of the matrix are called the *gap* of the matrix.

After that, the matrix E is transformed in an all-zero matrix by means of Gauss-Jordan elimination. This operation can also be done multiplying H_{alt} by another matrix as follows:

$$H_{salt} = \begin{bmatrix} I_{m-g} & 0 \\ ET^{-1} & I_g \end{bmatrix} H_{alt} = \begin{bmatrix} A & B & T \\ C_1 & D_1 & 0 \end{bmatrix}$$

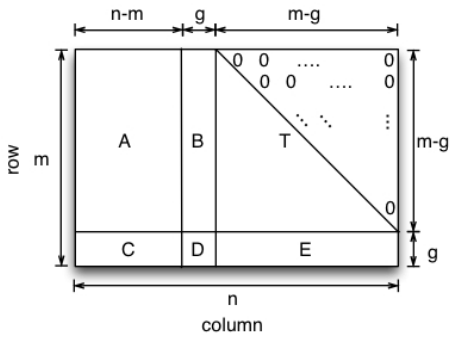

Figure 11. Approximate lower triangular form

with $C_1 = ET^{-1}A + C$ and $D_1 = ET^{-1}B + D$. Actually, it should be $-ET^{-1}$, but the minus can be ignored due to modulo-2 operations. This new matrix form is called *systematic approximate lower triangular*. All the matrices remain unchanged, except for the matrices C_1 and D_1 that could not be sparse.

The codeword is made up of three parts

$$\mathbf{c} = [\mathbf{u} \quad \mathbf{p_1} \quad \mathbf{p_2}]$$

where $\mathbf{u} = [u_1\ u_2\ \ldots\ u_k]$ is the message with $k = n - m$, $\mathbf{p_1} = [p_{11}\ p_{12}\ \ldots\ p_{1g}]$ is a vector with the first g parity bits and $\mathbf{p_2} = [p_{21}\ p_{22}\ \ldots\ p_{2m-g}]$ is another vector with the other $m-g$ parity bits. Using the equation $H_{salt}\mathbf{c}^T = 0$

$$\begin{bmatrix} A & B & T \\ C_1 & D_1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p_1} \\ \mathbf{p_2} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}$$

the following two equations can be found

$$A\mathbf{u} + B\mathbf{p_1} + T\mathbf{p_2} = \mathbf{0}$$

$$C_1\mathbf{u} + D_1\mathbf{p_1} = \mathbf{0}$$

If D_1 is invertible, p_1 can be computed as:

$$\mathbf{p_1} = {D_1}^{-1}C_1\mathbf{u}$$

and p_2 can be found from p_1 as

$$\mathbf{p_2} = T^{-1}(A\mathbf{u} + B\mathbf{p_1})$$

The parity vector p_2 can also be computed using *back-substitution*, thanks to the lower triangular form. It means that each parity bit p_{2i} can be found as a function of the message bits and previously computed parity bits.

The complexity order of this algorithm is $\mathcal{O}(n + g^2)$, and therefore the smaller the gap g the lower the encoding complexity will be.

3.2 ALT form using a greedy algorithm

This is a modified version of the algorithm (13) that is used to put a generic parity check matrix $H_{m\,x\,n}$ into the approximate lower triangular (ALT) form shown in Figure 11.

First of all, in the *initialization step* the column with the smallest `positive` number k_0 of 1s is found. In case there are more columns with the same number of 1s, randomly pick one. After that, swap this column with the n-th column. At this point, the k_0 rows with a one in the new n-th column have to be moved to the bottom of the matrix H. To avoid to excessively modify the matrix a row r_i will be swapped only if its row number r_i is lower than $m - k_0 + 1$. This means that if the 1s are already at the bottom of the matrix the corresponding rows will not be moved. If $r_i < m - k_0 + 1$ the first "free" row starting from the m-th row, that is the row with a zero in the n-th column, will be found and then the row r_i will be swapped with this "free" row. For the rows left, the search for the "free" row will start from the one on the top of the last swapped row. After all the rows are properly moved the gap is set to $g = k_0 - 1$ and two new variables $p = n - 1$ and $j = 1$ are created. At this point the first element on the main diagonal of the matrix T, starting from the bottom, is in the n-th column at the $(m - g)$-th row.

After that, the *step 1* starts and the column with the smallest `positive` number of 1s k_j will be found between the columns from 1 to p taking into account only the rows from 1 to $m - g - j$. As before, if more columns have the same number of 1s, randomly pick one. Next

swap the found column with the p-th column. If there is just one 1 or one of the 1s in the new column p is already on the main diagonal do nothing, otherwise swap the corresponding row with the $m-g-j$-th row. If more than one 1 are present append the remaining rows k_j-1 to the bottom of H and update g with $g=g+k_j-1$.

Then update $j=j+1$ and $p=p-1$ and go back to the step 1 until $m-g-j\geq 1$. At the end of this step the matrix will be in ALT form. At this point the Richardson/Urbanke algorithm can be applied.

```
% find column with smallest number of 1s
% between columns from 1 to c_lim
% and rows from 1 to r_lim
function [m,min_col] = min1s_col(H,c_lim,r_lim)
    cnz = zeros(1,c_lim);
    for i=1:c_lim
        cnz(i) = nnz(H(1:r_lim,i));
    end
    cnz(~cnz) = NaN;
    [m,min_col] = min(cnz);
```

```
[m, n] = size(H);


%% initialization
% find column with smallest number of 1s (k0)
[k0,min_col] = min1s_col(H,n,m);
```

```
% swap min_col and the n-th col
H(:,[min_col,n]) = H(:,[n,min_col]);
% move k0 rows with a 1 in the n-th col to the bottom of H
% find row indexes where there are these 1s
pos = find(H(:,n));
s = m;
for i = 1:size(pos,1)
    % if i-th row is already below the m-k0+1 row do not move it
    if (pos(i) < m-k0+1)
        %otherwise find the first free row where to move the i-th row
        while(H(s,n) == 1)
            s = s - 1;
        end
        % move the row
        H([pos(i),s],:) = H([s,pos(i)],:);
        s = s - 1;
    end
end

% gap
g = k0-1;

%% Step 1
p = n-1;
j = 1;

while (m-g-j >= 1)
    % n_ones = number of 1s on or above the main diagonal
    [n_ones,min_col] = min1s_col(H,p,m-g-j);
```

```
% swap min_col and the p-th col
H(:,[min_col,p]) = H(:,[p,min_col]);

% find row indexes where there are the 1s
pos = find(H(1:m-g-j,p));

% swap pos(1)-row with the row with index m-g+j (on the diagonal)
if ismember(m-g-j,pos)
    % if one of the items is already on the diagonal
    % move it in pos(1), so it will be not taken into account
    % in the following step

    % find index of the m-g-j-th item in pos
    index = find(pos == m-g-j);
    pos([index,1]) = pos([1,index]);
else
    % swap pos(1)-th row with the row on the diagonal
    H([pos(1),m-g-j],:) = H([m-g-j,pos(1)],:);
end

if n_ones > 1
    % swap the remaining r_j rows to the bottom of H
    for i=2:n_ones
        % append pos(i)-th row at the bottom of the matrix
        H(m+1,:) = H(pos(i),:);
        % remove pos(i)-th from the old position
        H(pos(i),:) = [];
        % subtract all the indeces of vector pos, because
        % all items have been moved up of 1 position
        pos = pos-1;
```

```
            % increase the gap
            g = g+1;
        end
    end

    j = j+1;
    % the next search will be done on (1:p-1) columns
    p = p-1;
end
```

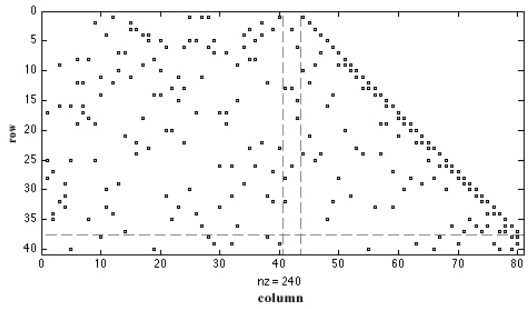

Figure 12. ALT form via greedy algorithm

An example of the parity check matrix obtained after this algorithm has been executed is shown in Figure 12. This is a 40 x 80 matrix with $g = 3$ and where there are 240 non-zero elements. Then the Richardson/Urbanke algorithm is applied on this matrix giving as result

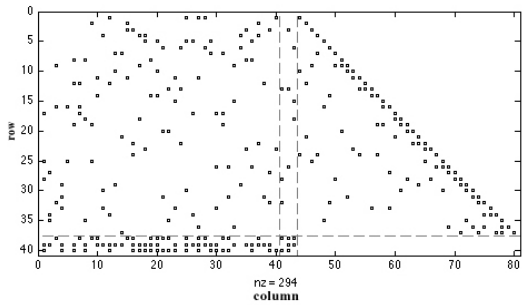

Figure 13. Matrix after Richardson/Urbanke algorithm

the matrix in Figure 13. As it is possible to see the number of ones increases to 294 and the matrices C and D are no more sparse.

3.3 Adaptive Message Length Encoding

The adaptive message length encoding (14) represents a valid alternative to the Richardson/Urbanke algorithm. This method is called *adaptive* because the dimension of the matrix A is not fixed and depends on the size of the matrix T, which changes according to the randomly generated parity check matrix. As before, the matrix H is put into the lower triangular form (Figure 14), for example by means of the previous algorithm. The obtained matrix has the following structure:

$$H_{alt} = \begin{bmatrix} A & T \\ B & C \end{bmatrix}$$

where A is ν x $(n-\nu)$, T is ν x ν, B is $(m-\nu)$ x $(n-\nu)$ and D is $(m-\nu)$ x ν.

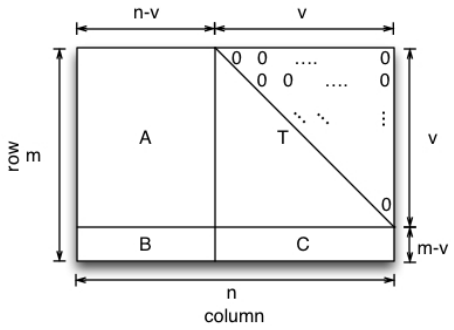

Figure 14. Lower triangular form for AML encoding

The codeword is divided in two parts

$$\mathbf{c} = [\mathbf{u} \quad \mathbf{p}]$$

where $\mathbf{u} = [u_1\ u_2\ \ldots\ u_k]$ is the message and $\mathbf{p} = [p_1\ p_2\ \ldots\ p_v]$ is the parity vector. Using the equation $H_{alt}\mathbf{c}^T = 0$

$$\begin{bmatrix} A & T \\ B & C \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}$$

the following two equations are obtained

$$A\mathbf{u}^T + T\mathbf{p}^T = \mathbf{0}$$

$$B\mathbf{u}^T + C\mathbf{p}^T = \mathbf{0}$$

From the first equation the parity bits can be computed as

$$\mathbf{p}^T = T^{-1} A \mathbf{u}^T$$

This new method has two big advantages over the Richardson/Urbanke algorithm. The former is that if we first compute $A\mathbf{u}^T$ and then we multiply the result by T^{-1}, the overall complexity is $\mathcal{O}(n)$ instead of $\mathcal{O}(n + g^2)$ as in the Richardson/Urbanke algorithm. The latter is related to the memory required to store the parity check matrix. Indeed, the matrices B and C can be ignored and the matrix to be stored becomes

$$H = [A \quad T]$$

3.4 Quasi-Cyclic encoding

3.4.1 Row Circulant QC encoding

As we have seen in section 2.1.1 a circulant matrix can be represented by a polynomial. Given the generator matrix in Equation 2.1, the polynomials

$$c_l^{-1} c_1, \quad c_l^{-1} c_2, \quad \ldots, \quad c_l^{-1} c_{l-1}$$

can be used to create an encoder made of $(l-1)$ left cyclic shift registers, each with L bits. All the parity bits will be ready after m clock cycles.

The example 3.1 will make all clear.

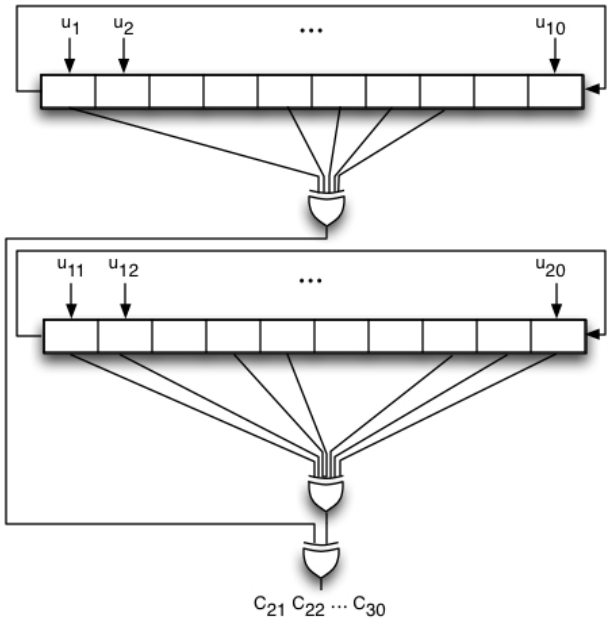

Figure 15. Row circulant QC encoder example 3.1

Example 3.1 Given the following polynomials:

$$c_1(x) = 1 + x^2 + x^6$$

$$c_2(x) = x + x^5 + x^7$$

$$c_3(x) = 1 + x^4 + x^9$$

a code with $l = 3$ and $L = 10$ has been created, where l is the number of circulants and each circulant has size L x L. The code rate is $R = \frac{2}{3}$ and the final code length is $n = 30$. The parity check matrix is $H = [C_1\ C_2\ C_3]$. The polynomial $c_3(x)$ is invertible and it is equal to

$$c_3(x)^{-1} = x^2 + x^7 + x^9$$

Therefore,

$$c_3(x)^{-1}c_1(x) = 1 + x^4 + x^5 + x^6 + x^7$$

$$c_3(x)^{-1}c_2(x) = 1 + x + x^3 + x^4 + x^7 + x^8 + x^9$$

At this point the generator matrix is

$$G = \begin{bmatrix} & (C_3^{-1}C_1)^T \\ I_{20} & \\ & (C_3^{-1}C_2)^T \end{bmatrix}$$

and the corresponding encoder implementation is shown in Figure 15. A new parity bit will be computed at each clock cycle starting from c_{21} up to c_{30}.

3.4.2 Block Circulant QC encoding

The encoding for a block circulant QC code requires a generator matrix in the systematic form. A possible parity check matrix is the one of Equation 2.2 from which the systematic form

of the generator matrix G_{sys} can be found as seen in section 2.1.2. The general structure of a systematic generator matrix is:

$$G_{sys} = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_{k_b} \end{bmatrix} =$$

$$\underbrace{\begin{bmatrix} I & 0 & \dots & 0 \\ 0 & I & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & I \end{bmatrix}}_{I} \underbrace{\begin{bmatrix} G_{1,1} & G_{1,2} & \dots & G_{1,m_b} \\ G_{2,1} & G_{2,2} & \dots & G_{2,m_b} \\ \vdots & \vdots & \vdots & \vdots \\ G_{k_b,1} & G_{k_b,2} & \dots & G_{k_b,m_b} \end{bmatrix}}_{A}$$

where each $G_{i,j}$ is a circulant matrix of size L x L and $m_b = n_b - k_b$. The matrix G_{sys} is of size Lk_b x Ln_b, and it contains a matrix A of size Lk_b x Lm_b and an identity matrix I_{Lk_b}.

As shown in (15), the message to be encoded is divided in blocks of L bits

$$\mathbf{u} = [\mathbf{u_1}\ \mathbf{u_2}\ \dots\ \mathbf{u_{k_b}}] = [u_1\ u_2\ \dots\ u_{Lk_b}]$$

and the corresponding codeword is

$$\mathbf{c} = [\mathbf{u_1}\ \mathbf{u_2}\ \dots\ \mathbf{u_{k_b}}\quad \mathbf{p_1}\ \mathbf{p_2}\ \dots\ \mathbf{p_{m_b}}]$$

where each $\mathbf{p_i}$ contains L bits. Assume that $g_{i,j}$ is the last row of a circulant $G_{i,j}$ and $g_{i,j}{}^{x}$ the left cyclic shift by x positions of $g_{i,j}$. The codeword is obtained using the equation

$$\mathbf{c} = \mathbf{u}G_{sys} = \mathbf{u_1}G_1 + \mathbf{u_2}G_2 + \cdots + \mathbf{u_{k_b}}G_{k_b}$$

where $\mathbf{u_i} = [u_{(i-1)L+1},\ u_{(i-1)L+2}\ u_{(i-1)L+3},\ \ldots\ u_{iL}]$ with $1 \leq i \leq k_b$.

The j-th block of parity bits, with $1 \leq j \leq m_b$, is computed as

$$\mathbf{p_j} = \mathbf{u_1}G_{1,j} + \mathbf{u_2}G_{2,j} + \cdots + \mathbf{u_{k_b}}G_{k_b,j} \tag{3.1}$$

where for $i = 1,\ 2,\ \ldots,\ k_b$

$$\mathbf{u_i}G_{i,j} = u_{(i-1)L+1}g_{i,j}{}^{(L-1)} + u_{(i-1)L+2}g_{i,j}{}^{(L-2)} + \cdots + u_{iL}g_{i,j}{}^{0} \tag{3.2}$$

The encoder implementation (15) to compute a block of parity bits p_j is shown in Figure 16.

The information bits are shifted out one at a time starting from the last bit u_{Lk_b}. The encoding process consists of the following steps:

- the left cyclic shift register B is initialized with $\mathbf{g_{k_b,j}}^{0} = \mathbf{g_{k_b,j}}$, that is the last row of the circulant (no shifted) and the content of accumulator register A is set to zero;
- the information bit u_{Lk_b} is shifted out and the product $u_{Lk_b}\mathbf{g_{k_b,j}}$ is computed by the AND gates;

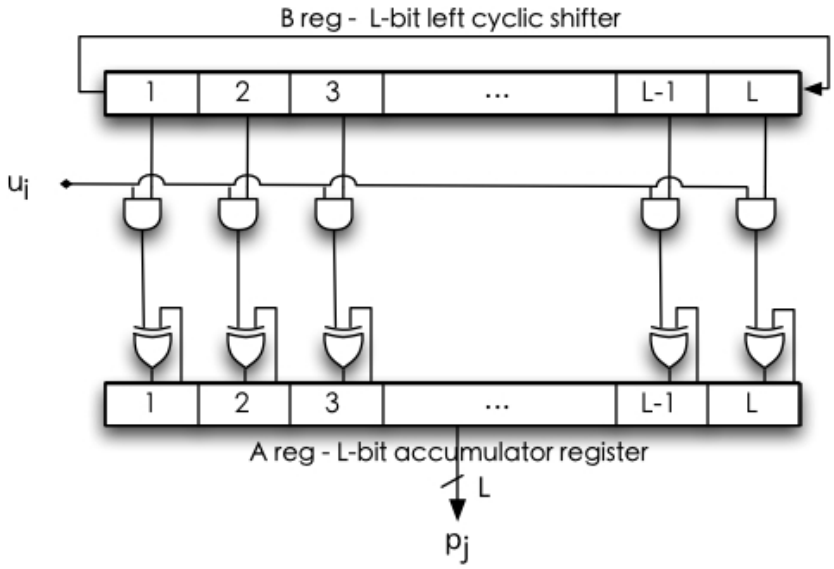

Figure 16. CSRAA encoder

- the result of the AND gates is added by means of the XOR gates to the vector of zeros stored in the register A.
- After that, the register B is cyclically shifted to the left by 1 position and its new value is $\mathbf{g_{k_b,j}}^1$;
- at this point the bit u_{Lk_b-1} is shifted out and the product $u_{Lk_b-1}\mathbf{g_{k_b,j}}^1$ is performed;
- the result at the output of the ANDs is added to the previous vector stored in the register A as shown in Equation 3.2. The final result is $u_{Lk_b-1}\mathbf{g_{k_b,j}}^1 + u_{Lk_b}\mathbf{g_{k_b,j}}$.
- These operations continue until the last bit $u_{L(k_b-1)}$ of the block $\mathbf{u_{k_b}}$ is shifted out. At this time the register A stores the partial sum $\mathbf{u_{k_b}}G_{k_b,j}$ of Equation 3.1.

- The next thing to do is to store in B the following circulant generator $\mathbf{g_{k_b-1,j}}^0 = \mathbf{g_{k_b-1,j}}$;
- then the previous operations are repeated until all the bits of the block u_{k_b-1} are shifted out. At this point the content of the register A is $\mathbf{u_{k_b-1}} G_{k_b-1,j} + \mathbf{u_{k_b}} G_{k_b,j}$.
- All these steps are iterated until the last information bit u_1 is shifted out and the content of the register A is exactly the j-th block of parity bits $\mathbf{p}_j$.

This implementation is called *cyclic shift register adder accumulator* (CSRAA). In order to compute all the parity blocks, m_b CSRAA will be employed as will be shown in the next chapter.

3.5 Repeat-Accumulate encoding

As we have seen in section 2.2, the matrix H_1 is related to the information bits u_i. Let's assume we are dealing with a regular RA code, that is a code with a matrix H_1 with a fixed row weight a and a fixed column weight q. Then it means that each message bit u_i will be repeated q times and a of these repeated bits will be summed together modulo-2 in order to create a check node. The connection between the repeated bits and the check nodes will be decided by a component called the *interleaver*. The form of the matrix H_2 indicates that the parity bits have to be added two at a time, except for the first parity bit. The component in charge of doing this operation is called the *accumulator*.

The encoder structure is shown in Figure 17.

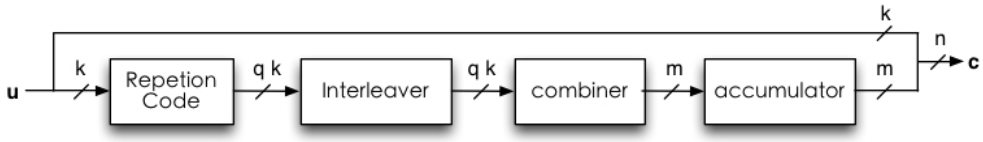

Figure 17. Repeat-Accumulate systematic encoder

As it is described in (7), the message $\mathbf{u} = [u_1\ u_2\ \ldots\ u_k]$ is repeated q times and the output of the repetition code has the form:

$$\mathbf{r} = [r_1,\ r_2,\ \ldots,\ r_{qk}] = [\underbrace{u_1,\ u_1,\ \ldots\ u_1}_{q},\quad \underbrace{u_2,\ u_2,\ \ldots\ u_2}_{q},\quad \ldots,\quad \underbrace{u_k,\ u_k,\ \ldots\ u_k}_{q}]$$

After this step, a permutation of the bits in the vector $\mathbf{r}$ is found following the content of the interleaver vector $\mathbf{\Pi} = [\Pi_1,\ \Pi_2,\ \ldots,\ \Pi_{qk}]$, where each Π_i is a positive number indicating the new position in the vector $\mathbf{i}$ of the bit r_i. The output of the interleaver is a vector

$$\mathbf{i} = [i_1,\ i_2,\ \ldots,\ i_{qk}] = [r(\Pi_1),\ r(\Pi_2),\ \ldots,\ r(\Pi_{qk})]$$

where $r(\Pi_i)$ is the entry of the vector $\mathbf{r}$ at the position Π_i. At this point, the combiner adds modulo-2 "a" bits at a time in the following way

$$x_j = i_{(j-1)a+1} + i_{(j-1)a+2} + \cdots + i_{ja}$$

with $j = 1,\ 2,\ \ldots,\ m$ and $m = \frac{qk}{a}$, giving as output the vector $\mathbf{x} = [x_1,\ x_2,\ \ldots\ x_m]$. In the last step the parity bits $\mathbf{p} = [p_1,\ p_2,\ \ldots\ p_m]$ are found by means of the expressions:

$$p_1 = x_1 \qquad p_i = p_{i-1} + x_i \quad \text{with} \quad i = 2,\ 3,\ \ldots,\ m$$

The systematic codeword at the output of the encoder is $\mathbf{c} = [u_1\ u_2\ \ldots\ u_k\ p_1\ p_2\ \ldots\ p_m]$.

Example 3.2 Assume our initial message is made of 4 bits $\mathbf{u} = [u_1\ u_2\ u_3\ u_4] = [1\ 0\ 1\ 0]$, $q = 3$ and $a = 2$. So, $\mathbf{r} = [1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0]$. Let's write the bits of the vector $\mathbf{r}$ row-wise in a 3 x 4 matrix, and read it out column wise, as follows:

$$\begin{bmatrix} r_1 & r_2 & r_3 & r_4 \\ r_5 & r_6 & r_7 & r_8 \\ r_9 & r_{10} & r_{11} & r_{12} \end{bmatrix}$$

thus the interleaver vector is $\mathbf{\Pi} = [1\ 5\ 9\ 2\ 6\ 10\ 3\ 7\ 11\ 4\ 8\ 12]$. At this point

$$\mathbf{i} = [r(\Pi_1),\ r(\Pi_2),\ \ldots,\ r(\Pi_{12})] = [r_1\ r_5\ r_9\ r_2\ r_6\ r_{10}\ r_3\ r_7\ r_{11}\ r_4\ r_8\ r_{12}]$$

$$\mathbf{i} = [1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0]$$

and

$$\mathbf{x} = [1+0,\ 1+1,\ 0+0,\ 1+1,\ 0+0,\ 1+0] = [1\ 0\ 0\ 0\ 0\ 1]$$

With $p_1 = x_1 = 1$,

$$p_2 = p_1 + x_2 = 1 + 0 = 1 \qquad p_3 = p_2 + x_3 = 1 + 0 = 1$$

$$p_4 = p_3 + x_4 = 1 + 0 = 1 \qquad p_5 = p_4 + x_5 = 1 + 0 = 1$$

$$p_6 = p_5 + x_6 = 1 + 1 = 0$$

and therefore $\mathbf{p} = [1\ 1\ 1\ 1\ 1\ 0]$. The equivalent parity check matrix can be found looking at the interleaver pattern $\mathbf{\Pi}$. In the 6 x 4 matrix H_1 there should be $a = 2$ 1s in each row and $q = 3$ 1s in each column. For example the first row is represented by the first two numbers 1 and 5 of the vector $\mathbf{\Pi}$, that indicate the bits $\lceil \frac{1}{q} \rceil = \lceil \frac{1}{3} \rceil = 1$ and $\lceil \frac{5}{3} \rceil = 2$, the second row by the following two numbers 9 ($\lceil \frac{9}{3} \rceil = 3$rd bit) and 2 ($\lceil \frac{2}{3} \rceil = 1$st bit), etc.

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

The chosen interleaver is really simple and it will affect the decoding performance due to all the 4-cycles introduced in the matrix H. The 1s in bold in the matrix H are just two 4-cycles. Better interleaver implementations can be found in (16).

3.6 Encoding using erasure decoding

The encoding can also be done using the *erasure decoding* algorithm (7) (page 52) as it has been shown in (17). In this algorithm the messages are binary numbers and not probabilities as in the sum product decoding algorithm. The message bits are put in place of k linearly independent bit nodes and the other $\mathsf{n} - \mathsf{k}$ bit nodes are set as erased bits. The encoding will work properly if no *stopping sets* are present in the erased bits. A stopping set is a set of code bits such that a check node that checks on one bit in this set, checks on at least two bits of this set. Otherwise, if the bit nodes in the stopping set are erased bits, the decoding algorithm will fail. Indeed, in this case the parity check equations cannot be solved because at least two erased bits are included in the corresponding parity check equations. However, a big advantage of this implementation is to save hardware resources, because the encoding and the decoding operations could be done with the same circuit.

CHAPTER 4

ENCODER IMPLEMENTATIONS ON ALTERA FPGA

The software used to analyze and synthesize the VHDL code is the *Quartus® II Web Edition v12.1 Service Pack 1* (18). This is a free version from Altera® that allows to compile a HDL design for a specific device. I have used an Altera® FPGA of the family Cyclone® II, whose name is *EP2C35F672C6*. It has 33216 logic elements, 483840 memory bits, 70 embedded multipliers and 475 I/O pins. Furthermore, the software to simulate the design using different stimuli is the *ModelSim®-Altera Starter Edition 10.1b* (19), which is also provided by Altera®.

4.1 Cyclone FPGA

The Cyclone II structure is shown in Figure 18. As described in (20), it contains programmable logic components called logic array blocks (LABs) which are linked together via reconfigurable interconnects. Furthermore, embedded memory blocks and embedded multipliers are present. All these elements are arranged according to a row-column structure.

The Cyclone II FPGA also presents a global clock network and a maximum of four phase-locked loops (PLLs). The M4K memory blocks are dual-port memory with 4K of data bits plus other 4608 parity bits. These memory blocks can be used both as dual-port and single-port memory.

The logic arrays consist of LABs, where each LAB contains 16 logic elements (LEs). The logic element is the smallest unit of logic inside this FPGA and it used to efficiently implement

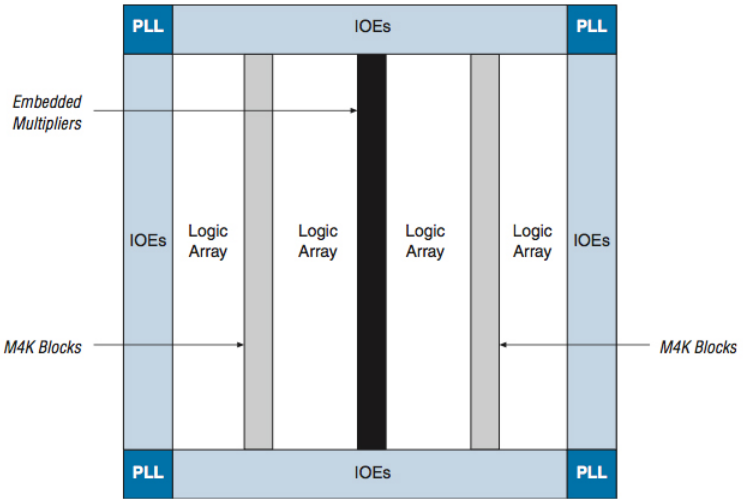

Figure 18. Cyclone II device structure

logic functions. As shown in Figure 19, each logic element contains: a four-input look-up table (LUT), which can implement any logic function of four variables, a programmable register and other components used for the interconnection with other LEs. It is important to note that in case of combinational logic, the LUT output bypasses the register going directly to the LE outputs.

4.2 Block Circulant QC encoder

Given a block circulant QC code with a generator matrix $G_{sys} = [I_k\ A_{k\,x\,m}]$ made of k_b x n_b circulants, a possible encoder implementation is shown in Figure 20. The components required are m_b *CSRAA*, whose structure is described in detail in section 3.4.2, a parallel input serial output (PISO) shift register for the k-bit message and a ROM memory made of k_b words, each

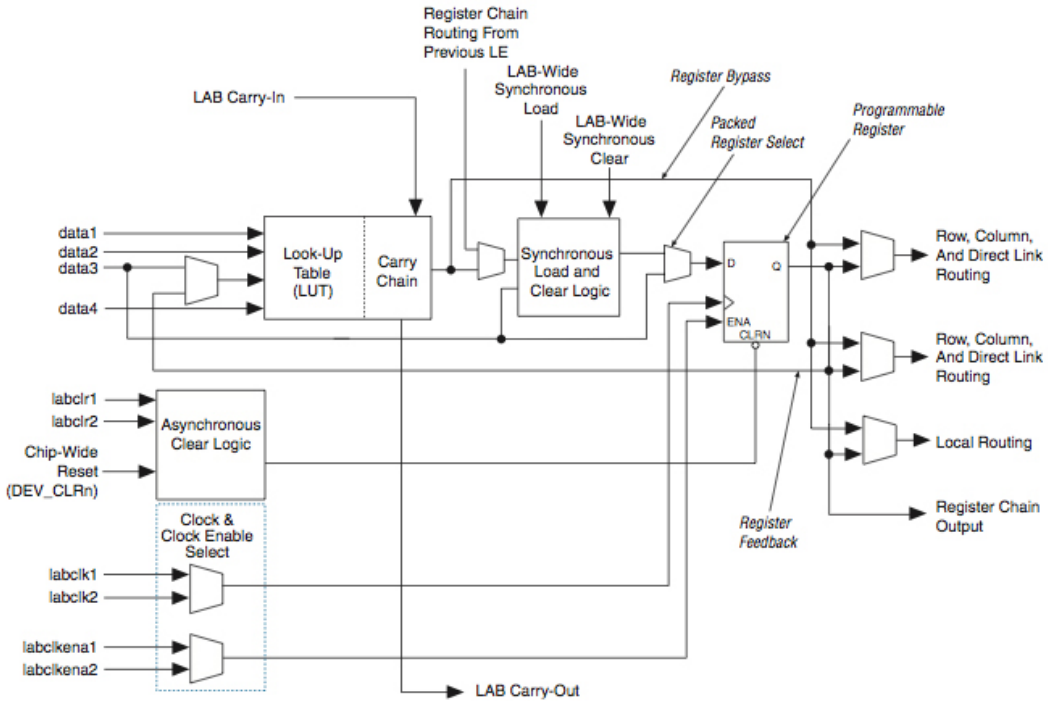

Figure 19. Cyclone II logic element structure

containing m bits. Given a generator matrix as the one in section 3.4.2, a row of the ROM memory *g_rom* contains the L-th row of all the circulants $G_{i,j}$ with $j = 1, 2, \ldots, m_b$ starting from the last row of circulants ($i = k_b$) up to the first row of circulants ($i = 1$).

For instance, given a code with rate $R = 1/2$ and $L = 4$

$$G = \begin{bmatrix} I & 0 & G_{1,1} & G_{1,2} \\ 0 & I & G_{2,1} & G_{2,2} \end{bmatrix} = \left[\begin{array}{cccccccc|cccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{array}\right]$$

the ROM will be made of the two 8-bit rows in bold, starting from the bottom

1st row 1 1 0 0 0 1 1 0

2nd row 1 0 0 1 0 1 0 1

In addition, the 2nd 4-bit vectors (1001 and 1100) are written before the 1st ones (0101 and 0110), because of the *big-endian* notation used in the VHDL design, according to which the $(N-1)$-th bit is the leftmost bit (MSB), while the 0-th bit is the rightmost bit (LSB). As we can see only two out of eight rows have to be saved with an enormous storage saving.

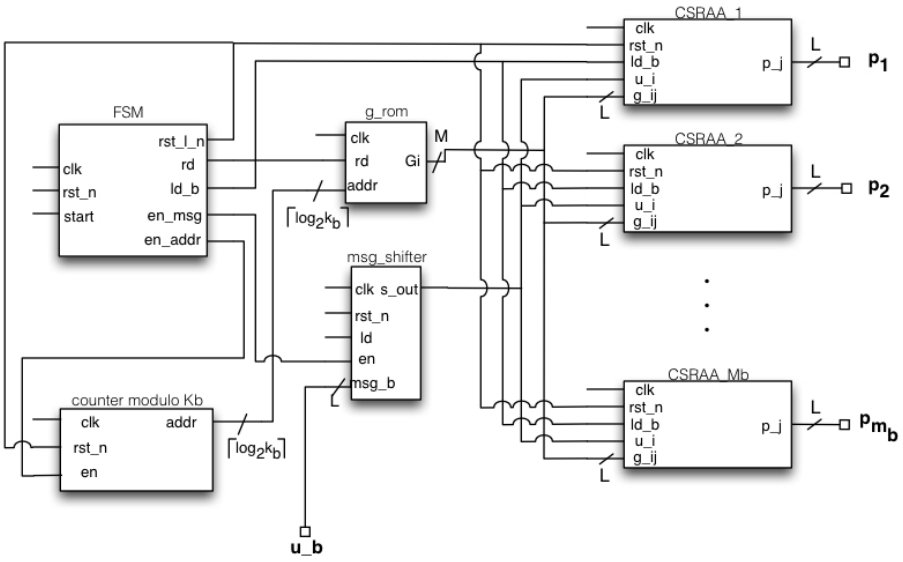

Figure 20. Block circulant QC encoder

After the ROM content is correctly initialized, the encoding process starts loading the message **u** into the *msg_shifter* component. The FPGA contains a limited number of I/O pins (475), thus, I decided to load L message bits at a time. Indeed, when the *ld* signal of the *msg_shifter* is set to one the content of the register is shifted to the right by L bits and the message block $\mathbf{u_b}$ is loaded in the first L bits. This step finishes when all the k_b message blocks are loaded inside the k-bit *msg_shifter*, which after the initialization process will right shift out one bit at each clock cycle.

After that, the start signal is enabled and the first row of the ROM memory is read out and it is divided in m_b blocks of L bits ($G_{i,j}$), one for each circulant matrix ($j = 1, 2, \ldots, m_b$).

These blocks are the inputs of the registers B inside each CSRAA circuit. All the operations performed in in the CSRAA components are described in detail in section 3.4.2.

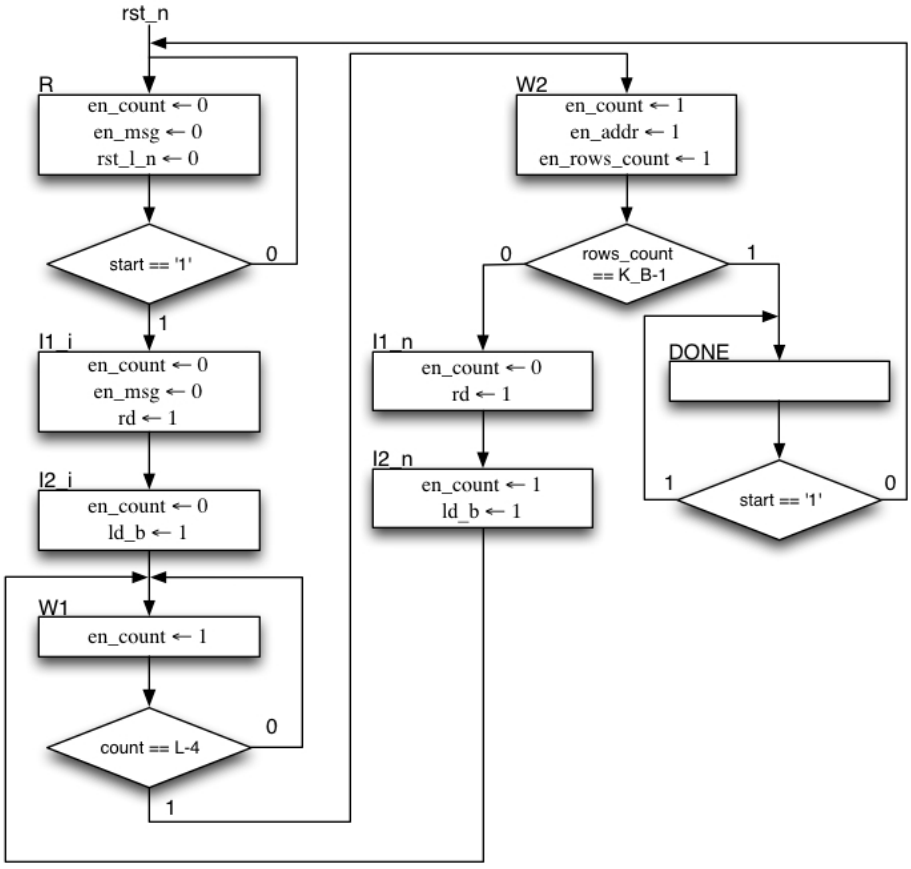

Figure 21. ASM QC encoder

The algorithmic state machine (ASM) chart can be seen in Figure 21, where each rectangular box represents a state in the finite state machine. All the signals not specified in the state boxes are set to zero, except for the *en_msg* and the local reset signal *rst_l_n* that are initialized to one.

As seen in section 3.4.2, the content of the register B has to be shifted L times before a new $\mathbf{g}_{\mathrm{i,j}}$ can be loaded. Nevertheless, in the state *W1* the condition *count = L-4* is present rather than *count = L-1*, where *count* starts from 0. This happens because 3 clock cycles are needed to obtain the new values $\mathbf{g}_{\mathrm{i,j}}$ from the ROM memory. More precisely, the involved states are *W2*, *I1_n* and *I2_n*, where in *W2* the following ROM address is generated, in *I1_n* the row with the new $\mathbf{g}_{\mathrm{i,j}}$ values is read from the memory and finally in *I2_n* the values $\mathbf{g}_{\mathrm{i,j}}$ are written in the respective registers B. Meanwhile, at each clock cycle every register B continues to cyclic left shift its content and every register A loads a new input coming from the XOR gates. These two registers do not need any enable signal to perform their operations. This is done to have less control signals in the FSM. The operations done in the states *I1_i*/*I2_i* and *I1_n*/*I2_n* are the same except for the fact that the first two states are used only once at the beginning. In addition, the state *I1_i* has the control signal *en_msg* set to zero to synchronize the message bits with the values $\mathbf{g}_{\mathrm{i,j}}$. Indeed, without the state *I1_i* the first message bit is shifted out one clock cycle before the $\mathbf{g}_{\mathrm{k_b,j}}$ values are loaded in the registers B.

Two counters (Figure 22) are used to keep track of the number of times the registers B has been shifted (*count*) and how many rows are read out from the ROM memory (*rows_count*).

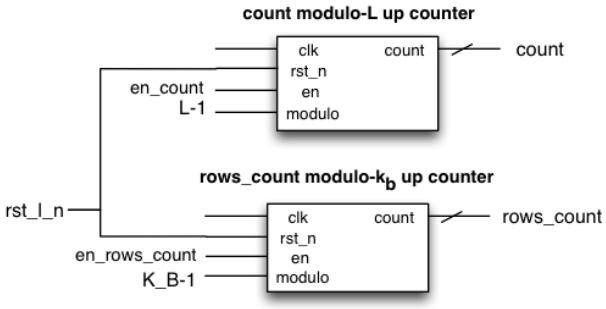

Figure 22. Internal counters of the QC encoder

The circuit finishes its operation when all k_b rows are read from the ROM memory, that is when the counter *rows_count* $= K_B - 1$.

At this point, *DONE* becomes the new state and a kind of handshaking protocol is used to restart the encoding process. In fact, the signal *start* has to be reset to zero before a new message can be encoded setting again this signal to one.

This encoder implementation requires $[2Lm_b + k]$ FFs for the registers, Lm_b ANDs and XORs, a ROM memory with k_b rows each with m bits, a modulo-L up counter ($\lceil \log_2 L \rceil$ FFs) and a modulo-k_b up counter ($\lceil \log_2 k_b \rceil$ FFs). In addition, k clock cycles are required to encode a message without considering the initialization step.

A simulation of the BC-QC encoder has been done using a code with $n = 28$ bits and rate $R = 1/2$. The message to encode is $\mathbf{u} = [1\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0]$ and the content of the ROM memory related to the generator matrix is:

1st row	0	0	1	0	1	0	0	1	0	1	0	0	1	0
2nd row	0	1	0	0	1	0	0	0	0	1	0	0	1	1

The waveform window of the *ModelSim* simulation can be seen in Figure 23. The resulting parity blocks are $\mathbf{p}(0) = [1\ 0\ 0\ 1\ 1\ 0\ 0]$, the first parity block and $\mathbf{p}(1) = [1\ 1\ 1\ 1\ 1\ 1\ 0]$, the second parity block.

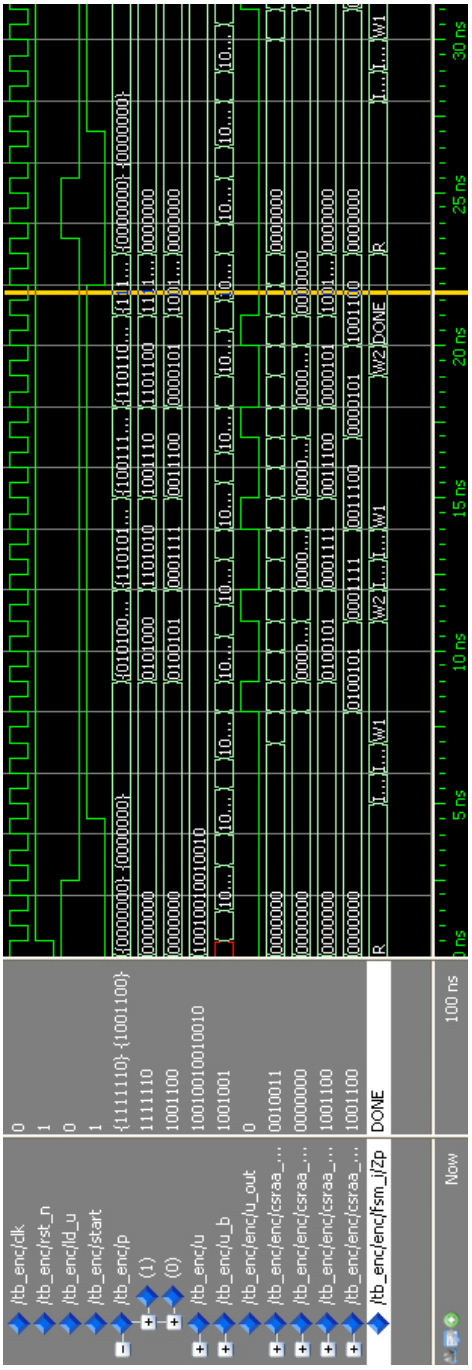

Figure 23. Simulation QC encoder

4.3 IEEE 802.11n encoder

4.3.1 Encoding algorithm

A *quasi cyclic irregular repeat accumulate* (QC-IRA) code is employed in the IEEE 802.11n standard (21). The parity check matrix presents a block circulant structure, where each circulant can be a zero matrix or a cyclic right shift of an identity matrix. The matrix H can be described by a base matrix H_b, as we have seen in section 1.5. An example of a base matrix for a code with $n = 648$ bits and $R = \frac{1}{2}$ is shown in Figure 24. Each circulant has size $Z = 27$ and the character "-" represents an all-zero matrix. The parity check matrix consists of two submatrices, the matrix H_1 that is related to the message bits and H_2, which is related to the parity bits. The second matrix H_2 has a *dual-diagonal* form and its first column presents the following structure

$$[1 \text{ - } \ldots \text{ - } 0 \text{ - } \ldots \text{ - } 1]^T$$

for all the matrices defined in the standard (21). This property allows to significantly reduce the encoder complexity (22). The standard defines 12 different codes, with three different code lengths N=648, 1296 and 1944 and four possible code rates for each code length (R = 1/2, 2/3, 3/4 and 5/6).

As seen in (22), the message bits are divided in k_b blocks of length Z, $\mathbf{u} = [\mathbf{u_0}\ \mathbf{u_1}\ \ldots\ \mathbf{u_{k_b-1}}]$, while the parity bits are divided in m_b blocks of length Z, $\mathbf{p} = [\mathbf{p_0}\ \mathbf{p_1}\ \ldots\ \mathbf{p_{m_b-1}}]$. Therefore, $\mathbf{c} = [\mathbf{u} \quad \mathbf{p}]$. Using the equation $H\mathbf{c}^T = 0$

0	-	-	-	0	0	-	-	0	-	-	0	**1**	0	-	-	-	-	-	-	-	-	-	-
22	0	-	-	17	-	0	0	12	-	-	-	-	0	0	-	-	-	-	-	-	-	-	-
6	-	0	-	10	-	-	-	24	-	0	-	-	-	0	0	-	-	-	-	-	-	-	-
2	-	-	0	20	-	-	-	25	0	-	-	-	-	-	0	0	-	-	-	-	-	-	-
23	-	-	-	3	-	-	-	0	-	9	11	-	-	-	-	0	0	-	-	-	-	-	-
24	-	23	1	17	-	3	-	10	-	-	-	-	-	-	-	-	0	0	-	-	-	-	-
25	-	-	-	8	-	-	-	7	18	-	-	**0**	-	-	-	-	-	0	0	-	-	-	-
13	24	-	-	0	-	8	-	6	-	-	-	-	-	-	-	-	-	-	0	0	-	-	-
7	20	-	16	22	10	-	-	23	-	-	-	-	-	-	-	-	-	-	-	0	0	-	-
11	-	-	-	19	-	-	-	13	-	3	17	-	-	-	-	-	-	-	-	-	0	0	-
25	-	8	-	23	18	-	14	9	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
3	-	-	-	16	-	-	2	25	5	-	-	**1**	-	-	-	-	-	-	-	-	-	-	0

H_1 H_2

Figure 24. Standard IEEE 802.11n matrix R=1/2 N=648 Z=27

$$\begin{bmatrix} \mathbf{h_{0,0}} & \mathbf{h_{0,1}} & \dots & \mathbf{h_{0,k_b-1}} & 1 & 0 & - & - & \dots & \dots & \dots & - \\ \mathbf{h_{1,0}} & \mathbf{h_{1,1}} & \dots & \mathbf{h_{1,k_b-1}} & - & 0 & 0 & - & - & \dots & \dots & - \\ \vdots & \vdots & \dots & \vdots & \vdots & \dots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{h_{x,0}} & \mathbf{h_{x,1}} & \dots & \mathbf{h_{x,k_b-1}} & 0 & - & - & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & & \dots & \vdots & - & \vdots & \vdots & \dots & \ddots & \ddots & \vdots & \vdots \\ \vdots & & \dots & \vdots & \vdots & \vdots & \vdots & \vdots & \dots & \ddots & \ddots & \vdots \\ \mathbf{h_{m_b-1,0}} & \mathbf{h_{m_b1,1}} & \dots & \mathbf{h_{m_b-1,k_b-1}} & 1 & - & \dots & - & - & - & - & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u_0} \\ \mathbf{u_1} \\ \vdots \\ \mathbf{u_{k_b-1}} \\ \mathbf{p_0} \\ \mathbf{p_1} \\ \vdots \\ \mathbf{p_{m_b-1}} \end{bmatrix} = 0$$

where the x-th row is the one with the 0 entry (no shifted identity matrix) in the first column of H_2, the following equations are obtained:

$$\sum_{j=0}^{k_b-1} \mathbf{h}_{0,j}\mathbf{u}_j + \mathbf{p}_0{}^1 + \mathbf{p}_1 = \mathbf{0} \qquad \text{0-th equ.}$$

$$\sum_{j=0}^{k_b-1} \mathbf{h}_{i,j}\mathbf{u}_j + \mathbf{p}_i + \mathbf{p}_{i+1} = \mathbf{0} \qquad i=1,\ \ldots,\ x\text{-}1,\ x\text{+}1,\ \ldots,\ m_b\text{-}2$$

$$\sum_{j=0}^{k_b-1} \mathbf{h}_{x,j}\mathbf{u}_j + \mathbf{p}_0 + \mathbf{p}_x + \mathbf{p}_{x+1} = \mathbf{0} \qquad \text{x-th equ.}$$

$$\sum_{j=0}^{k_b-1} \mathbf{h}_{m_b-1,j}\mathbf{u}_j + \mathbf{p}_0{}^1 + \mathbf{p}_{m_b-1} = \mathbf{0} \qquad (m_b\text{-}1)\text{-th equ.}$$

where $\mathbf{p}_0{}^1$ is a circular left shift of $\mathbf{p}_0$ by one position. If we sum all the previous equations together, we have

$$\mathbf{p_0} = \sum_{i=0}^{m_b-1} \sum_{j=0}^{k_b-1} \mathbf{h_{i,j}u_j}$$

This thanks to the fact that all the matrices of the IEEE 802.11n standard have an even number of rows and that the sum of two identity matrices, both no shifted or shifted, is equal to the all-zero matrix. Indeed, the only item that remains in the matrix H_2 after the sum of all the rows is the no shifted identity matrix corresponding to the p_0 parity block (Figure 25). In addition, the sum of the rows of the matrix H_1 is exactly $\sum_{i=0}^{m_b-1} \sum_{j=0}^{k_b-1} \mathbf{h_{i,j}u_j}$.

If we define $\boldsymbol{\lambda}_i = \sum_{j=0}^{k_b-1} \mathbf{h_{i,j}u_j}$,

$$\mathbf{p_0} = \sum_{i=0}^{m_b-1} \lambda_i \tag{4.1}$$

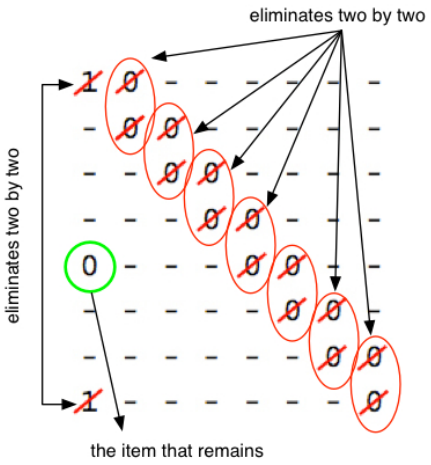

Figure 25. Matrix H_2 from code N=648 R=2/3

and

$$\mathbf{p}_1 = \boldsymbol{\lambda}_0 + \mathbf{p}_0{}^1 \tag{4.2}$$

$$\mathbf{p}_{m_b-1} = \boldsymbol{\lambda}_{m_b-1} + \mathbf{p}_0{}^1 \tag{4.3}$$

After we have computed $\mathbf{p}_1$ and $\mathbf{p}_{m_b-1}$, we can find the other parity blocks as follows

$$\mathbf{p}_i = \mathbf{p}_{i-1} + \boldsymbol{\lambda}_{i-1} \qquad i=2, \ldots, x\text{-}1 \tag{4.4}$$

$$\mathbf{p}_i = \mathbf{p}_{i+1} + \boldsymbol{\lambda}_i \qquad i=m_b\text{-}2, \ldots, x\text{+}1 \tag{4.5}$$

starting from the 2nd parity block and going forwards for the first equation, while starting from the $(m_b - 2)$-th parity block and going backwards for the second equation. After that, when we reach the x-th position, the corresponding parity block can be obtained from

$$\mathbf{p}_x = \mathbf{p}_0 + \mathbf{p}_{x+1} + \boldsymbol{\lambda}_x \tag{4.6}$$

4.3.2 Encoder architecture

The term $\mathbf{h}_{i,j}\mathbf{u}_j$ indicates a left circular shift of the message block $\mathbf{u}_j$ by as many positions as defined in the base matrix item $\mathbf{h}_{i,j}$. For example if $u_j = [1\ 0\ 0\ 1]$ and $\mathbf{h}_{i,j} = 3$, $\mathbf{h}_{i,j}\mathbf{u}_j = [1\ 1\ 0\ 0]$, that is the left cyclic shift of $\mathbf{u_j}$ by 3 positions.

After this observation, we can start to analyze the encoder architecture of Figure 26, which numbers on the wires are related to an encoder for a code with n=648 bits, rate R=1/2 and Z=27, which, from now on, will be used as example. Obviously, this encoder structure will work for all matrices of the IEEE 802.11n standard. This is possible using a conditional compilation, that allows to remove some hardware components that are not needed for the code rates $R = 3/4$ and $R = 5/6$.

1. First of all, the ROM *sha_rom* contains all the items $\mathbf{h}_{i,j}$ converted in binary present in the matrix H_1. These values $\mathbf{h}_{i,j}$ are the shift amounts previously seen. I've assigned $Z_w = \lceil \log_2(Z) \rceil$ bits for each item of the matrix H_1. Thanks to the fact that the value Z is not present in any of the standard matrices, the item "-" of the base matrix, corresponding to the all zero matrix, is saved in the ROM just as the value Z. Our base matrix (Figure 24) has the following

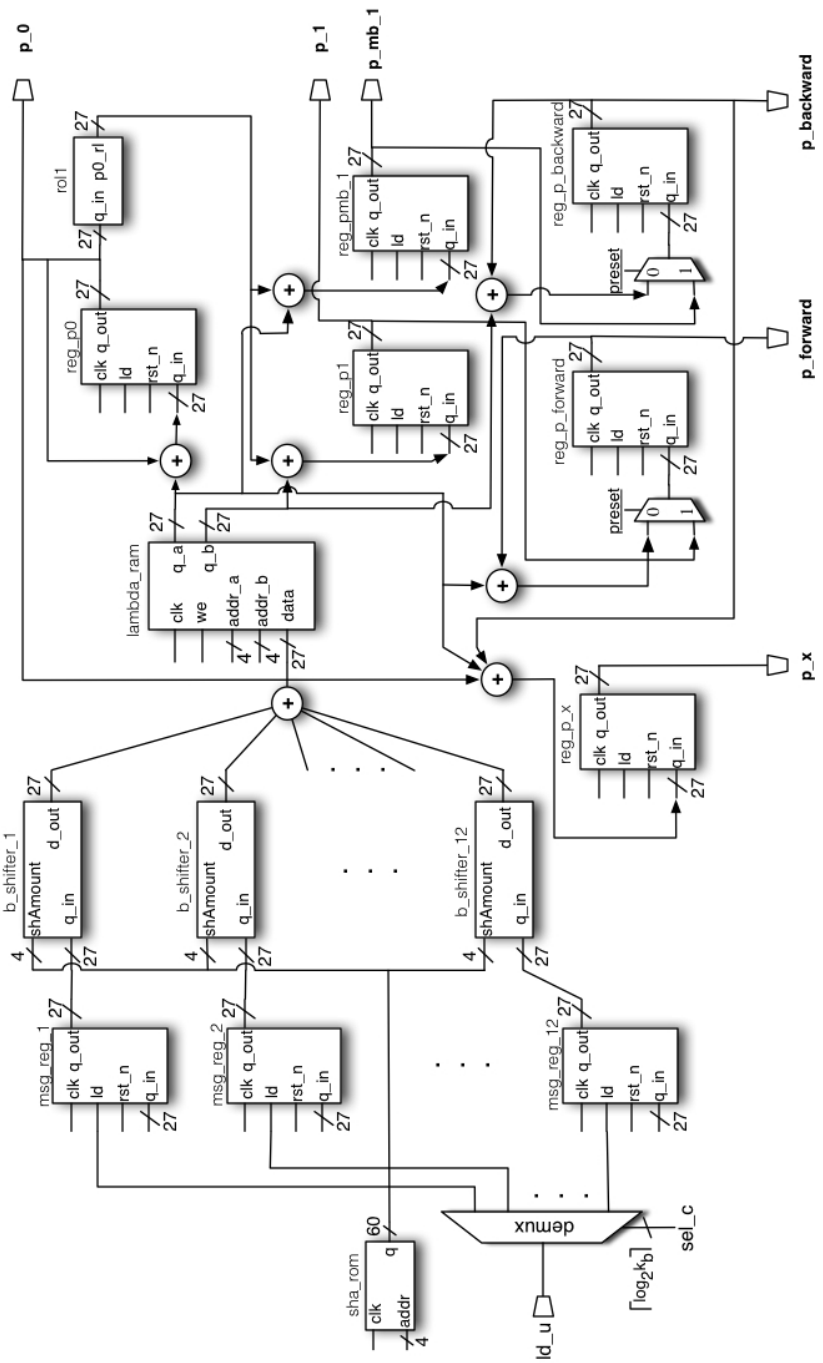

Figure 26. IEEE 802.11n encoder for N=648 Z=27 R=1/2

parameters $n_b = 24$, $k_b = 12$ and $m_b = 12$, therefore each row of the ROM memory contains $\lceil \log_2(Z) \rceil k_b = 5 \cdot 12 = 60$ bits and there will be m_b rows. The output B of the ROM memory also has $\lceil \log_2(Z) \rceil k_b$ bits.

2. The encoding of a message $\mathbf{u} = [\mathbf{u_0}\ \mathbf{u_1}\ \ldots\ \mathbf{u_{k_b-1}}] = [u_1\ u_2\ \ldots\ u_k]$ with $k = k_b Z$, begins saving all the message blocks $\mathbf{u_i}$ with $i = 0,\ 1,\ \ldots,\ k_b - 1$ in the Z-bit registers *msg_reg_i* one block at a time. During this operation the control signal *ld_u* has to be equal to one until all the message blocks are loaded and the demux selector *sel_c* has to be set to the corresponding message block number we want to select $(0,\ 1,\ \ldots,\ k_b - 1)$. Then the signal *ld_u* has to be set to zero in order not to modify the content of the registers *msg_reg_i* and the signal *start* of the FSM has to be put to one in order to let the encoding process begin.

3. After that, the first row of the ROM memory is read out and the message blocks are left cyclic shifted by the components *b_shifter_i* by as many positions as specified in the bits $[B_{iZ_w},\ B_{iZ_w+1},\ \ldots,\ B_{(i+1)Z_w-1}]$ at the output of the ROM memory *sha_rom*, with $i = 0,\ 1,\ \ldots,\ k_b - 1$. In case an all zero item is found (value Z=27), the output of the *b_shifter_i* is a Z-bit all-zero vector. At this point all the outputs of the *b_shifter_i* are added together modulo-2 and the result ($\boldsymbol{\lambda}_0$) is saved in the *lambda_ram* at the i-th address. The address of the shift amounts memory (*sha_rom*) is generated by the modulo-M_B *shAddr up counter*, while the address of the *lambda_ram* comes from the modulo-M_B *lambdaAddr up counter* with the *d_addr* selector set to zero as can be seen in Figure 28.

4. The operations of step 3 are repeated for all the rows in the ROM memory until all the $\boldsymbol{\lambda}_i$ are saved in the RAM memory. This happens when the *lambdaAddr up counter* reaches the

value $M_B - 1$. To save time, each time a value $\boldsymbol{\lambda}_i$ is saved in the memory, it is also read out and added to the content of the accumulator register *reg_p0*, initially set to zero. In this way at the end of these two steps (3 and 4) we have both all lambda values and the first parity block $\mathbf{p}_0$ as seen in Equation 4.1.

5. Then the parity block $\mathbf{p}_1$ is obtained from the sum of the value $\boldsymbol{\lambda}_0$ coming from the RAM memory with the left cyclic shift of $\mathbf{p}_0$ by one position coming from the component *rol1* just as it is shown in Equation 4.2. To speed up the circuit the lambda RAM has two output ports, therefore the parity block $\mathbf{p}_{m_b-1}$ is computed at the same time summing $\boldsymbol{\lambda}_{m_b-1}$ coming from the second port of *lambda_ram* with the cyclic shifted parity block $\mathbf{p}_0$ at the output of *rol1* as indicated in Equation 4.3. These two parity blocks will be saved into different registers, *reg_p1* and *reg_pmb_1* respectively. To get the values $\boldsymbol{\lambda}_0$ and $\boldsymbol{\lambda}_{m_b-1}$, the direct addresses *d_lambdaAddr1* and *d_lambdaAddr2* (Figure 28) are set to 0 and $M_B - 1$ respectively. In the meanwhile, *d_addr* is set to one, while *sel_dir_n* is set to zero such that the signal *lambdaAddr_1* is connected to *d_lambdaAddr1*, while the signal *lambdaAddr_2* is connected to *d_lambdaAddr2*. The signals *lambdaAddr_1/2* are connected to the address ports of the *lambda_ram* memory.

6. After that, the register *reg_p_forward* is initialized with the parity vector $\mathbf{p}_1$, while the register *reg_p_backward* is initialized with the parity vector $\mathbf{p}_{m_b-1}$. At this point, as seen in Equation 4.4 and Equation 4.5, the parity vectors $\mathbf{p}_i$ with $i_{forward} = 2, \ldots, x - 1$ and $i_{backward} = m_b - 2, \ldots, x + 1$ are obtained from the modulo-2 sum between the previously computed parity blocks and the lambda vectors coming from the *lambda_ram*. Also this time, the forward and backward parity blocks are computed at the same time thanks to the double

output RAM. At each clock cycle two new parity blocks are present at the outputs *p_forward* and *p_backward*. These two values are used to compute the following parity vectors in a recursive fashion. The addresses of the $\boldsymbol{\lambda}_i$ values are generated by the *f_addr up counter* for the forward parity blocks which counts from 1 to $X-1$ and the *b_addr down counter* for the backward parity bits which counts from $M_B - 2$ to $X + 1$.

7. Finally, the parity block $\mathbf{p}_x$ is found as shown in Equation 4.6, where the vector $\mathbf{p}_{x+1}$ comes from the last computed *p_backward* vector, $\mathbf{p}_0$ comes from the register *reg_p0* and $\boldsymbol{\lambda}_x$ is obtained setting *d_lambdaAddr1* equal to X. At this point, as in the QC encoder the new state becomes *DONE*. The encoder remains in this state until the signal *start* is one, when it becomes zero the FSM returns in the state *R*, where all the components are opportunely reset.

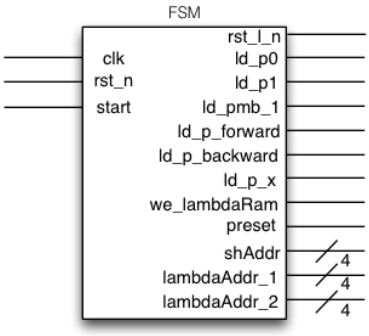

Figure 27. FSM component

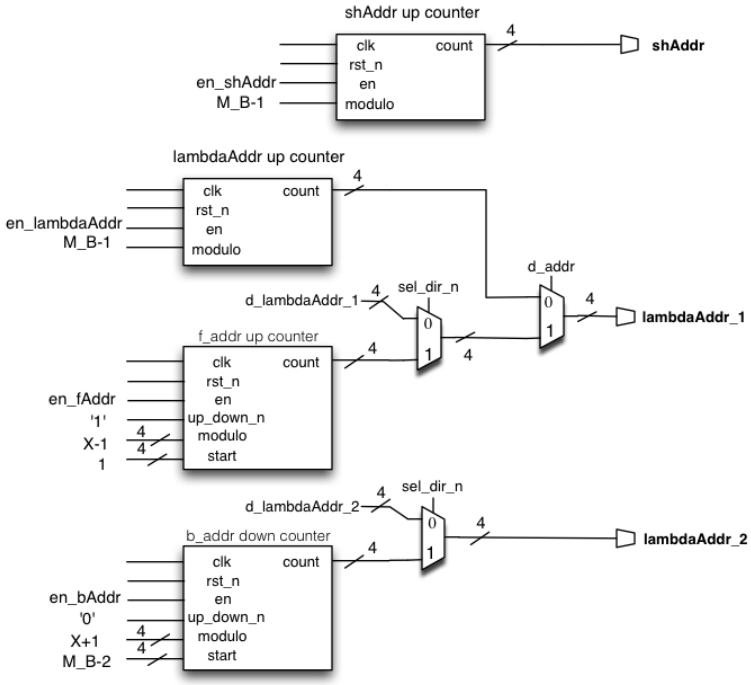

Figure 28. Internal counters of the IEEE 802.11n encoder

The datapath of the encoder in Figure 26 is managed by the finite state machine (FSM) shown in Figure 27. This component generates all the control signals and the addresses needed by the *sha_rom* SHift Amounts ROM and the *lambda_ram*.

The ASM chart can be seen in Figure 29. All the signals not present in the rectangular boxes are set to zero, except for the *rst_l_n* which is set to one as default. The addresses are generated by means of different counters (Figure 28), whose control signals are also managed by the FSM.

This encoder structure requires $[(k_b+6)Z]$ FFs for the registers, (k_b+1) Z-bit cyclic shifters (Z FFs), $[(k_b+6)Z]$ XORs, one ROM memory with $2^{\lceil \log_2 m_b \rceil}$ rows, each with $\lceil \log_2 Z \rceil k_b$ bits, one RAM memory with $2^{\lceil \log_2 m_b \rceil}$ rows, each with Z bits and two modulo-m_b counters ($\lceil \log_2 m_b \rceil$ FFs). The counters for the forward and backward parity vectors computation are needed only for the code rates $R = \frac{1}{2}$ and $R = \frac{2}{3}$ ($\log_2 (\frac{m_b-4}{2})$ FFs).

The number of clock cycles required is $[6 + 2m_b + (m_b - 4)/2]$ for code rates R=1/2, 2/3 and 3/4, and $[6 + 2m_b]$ for R=5/6.

The *ModelSim* simulation of this encoder is shown in Figure 30. The employed code has $n = 648$ bits and rate $R = 1/2$. The message to encode is 324 bit long, and it is divided in 12 message blocks, where the leftmost block is defined as *msg_g(11)*, while the rightmost one is defined as *msg_g(0)*. The resulting parity blocks: p_0, p_1, p_{m_b-1}, p_x, the four vectors $p_{forward}$ and the other four vectors $p_{backward}$ are shown in Figure 31. The first values of $\mathbf{p}_{forward}$ and $\mathbf{p}_{backward}$ in the clock cycle from 42 ns to 43 ns are just the initialization values $\mathbf{p}_1$ and $\mathbf{p}_{m_b-1}$.

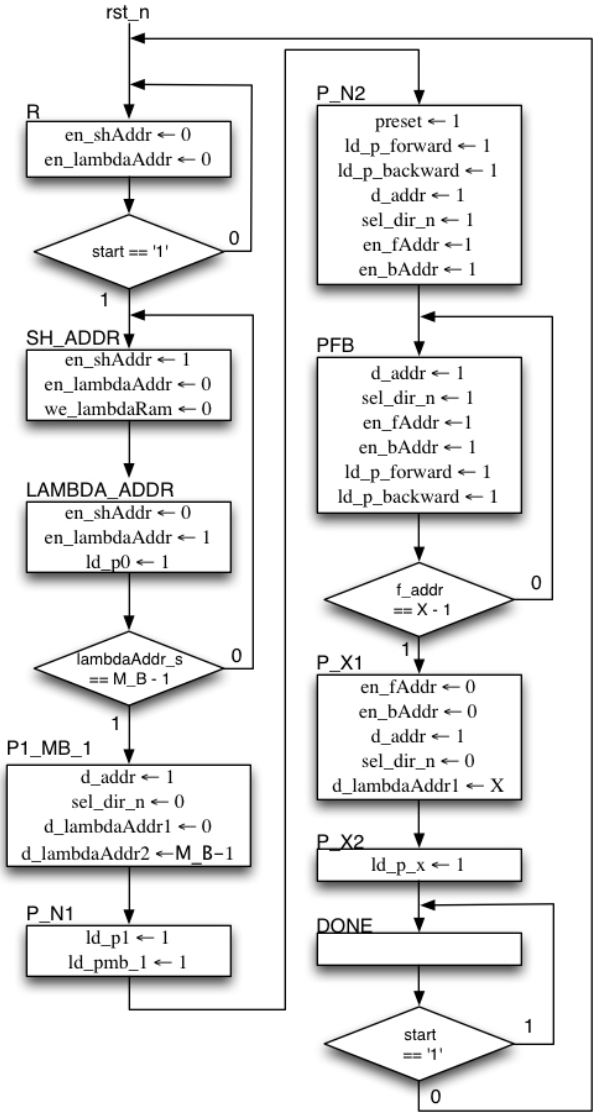

Figure 29. ASM IEEE 802.11n encoder

In addition, $\mathbf{p}_x$ is still zero in the clock cycle from 47 ns to 48 ns, because in this extra cycle the value λ_x is read from the memory.

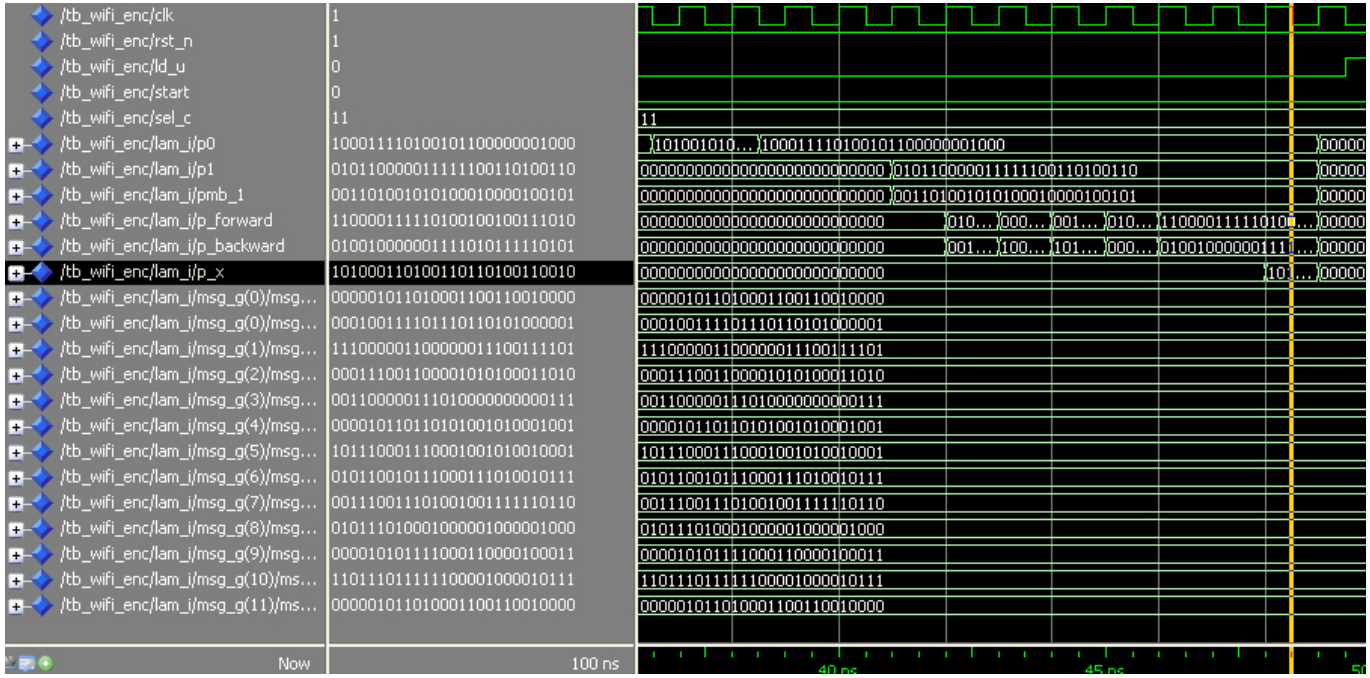

Figure 30. Simulation IEEE 802.11n encoder message blocks

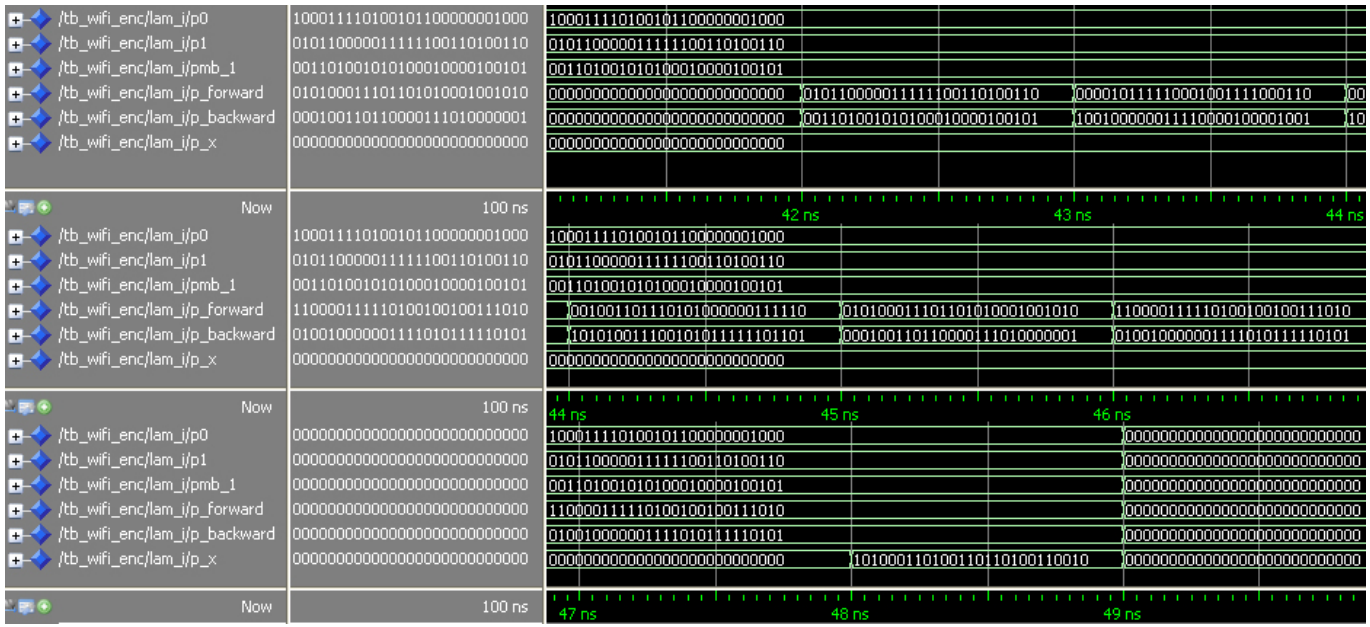

Figure 31. Simulation IEEE 802.11n encoder parity vectors

4.4 Comparison

The two encoder implementations have been synthesized using the same code parameters. I have decided to use the twelve possible code parameters defined in the standard IEEE 802.11n both for the block circulant QC encoder and the IEEE 802.11n encoder in order to make the comparison more straightforward. This standard defines three different code lengths (648, 1296 and 1944), each one with four possible code rates (1/2, 2/3, 3/4 and 5/6). The circulant sizes Z for these code lengths are 27, 54 and 81, respectively. The circulant size L in the QC encoder has been set equal to Z.

The number of FPGA logic elements (LEs) is shown in Figure 32. It is clear that the IEEE 802.11n encoder requires a higher number of LEs for all the code lengths. The strange behavior of the LEs curves of the IEEE 802.11n encoder at the rate $R = 5/6$ depends on the fact that at this rate the hardware required for the computation of the forward and backward parity blocks is not more needed. In this manner the encoder architecture for this rate is significantly simplified.

The same trend can be seen in the registers comparison of Figure 33. However, the number of registers of the two encoders related to the code rate $R = 1/2$ is really close. This is due to the fact that during the synthesis of the IEEE 802.11n encoder for this code rate, the Quartus Compiler is able to infer a RAM with a size compatible with the RAM blocks present inside the Altera FPGA, thereby reducing the number of registers required. Also in this case the number of registers for the IEEE 802.11n encoder with code rate $R = 5/6$ decreases because of the simplied HW structure.

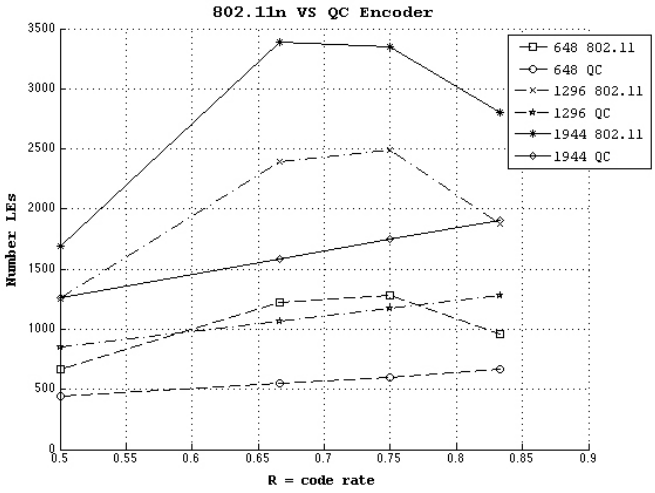

Figure 32. Logic elements comparison

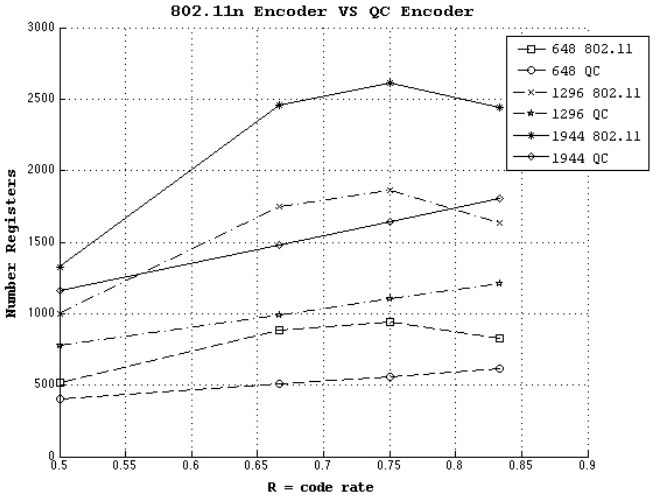

Figure 33. Registers comparison

The number of combinational functions can be seen in Figure 34. The IEEE 802.11n encoder has a lower or equal number of combinational logic at rates $\mathsf{R} = 1/2$ and $\mathsf{R} = 5/6$, but it presents a higher number at rates $\mathsf{R} = 2/3$ and $\mathsf{R} = 3/4$ with respect to the QC encoder.

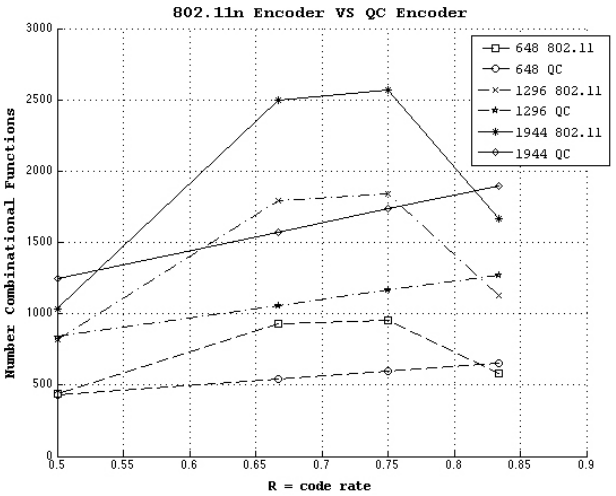

Figure 34. Combinational functions comparison

The parallel structure of the block circulant QC encoder allows an average frequency of 346 MHz, while the IEEE 802.11n encoder presents an average frequency of 159 MHz. The computation time in μs of the IEEE 802.11n encoder is shown in Figure 35, while the one of the QC encoder is in Figure 36.

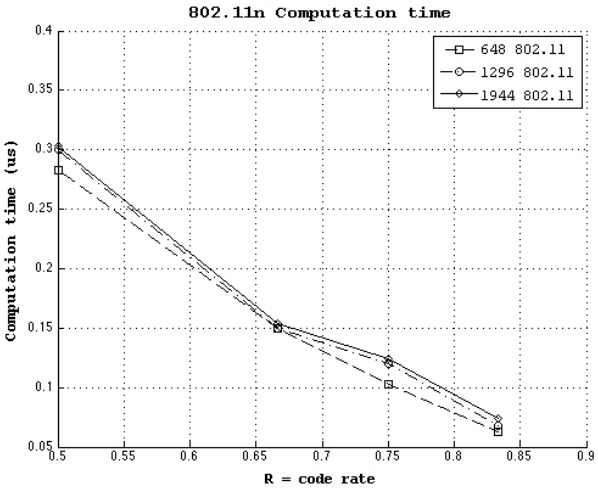

Figure 35. Computation time IEEE 802.11n encoder

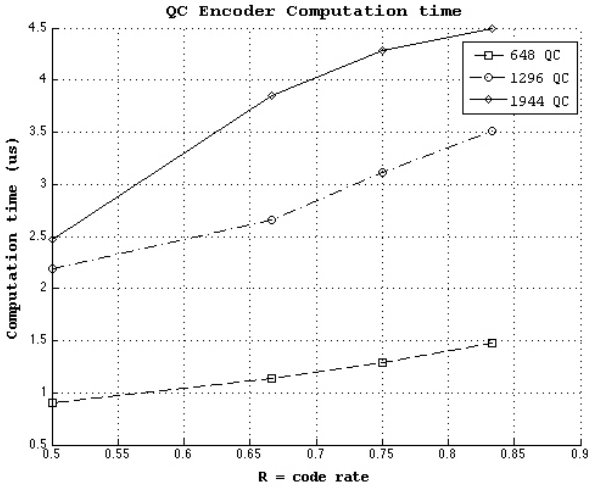

Figure 36. Computation time QC encoder

The computation time of the IEEE 802.11n encoder depends on the number of base matrix rows m_b and so it decreases as higher code rates are used. On the other hand the computation time of the QC encoder increases if a higher code rate is employed because of its dependence on the number of information bits k. In addition, the compution time of the IEEE 802.11n encoder is one order of magnitude lower than the one of the QC encoder.

The initialization time of the message to encode is not taken into account in the computation time graphs because it is equal for the two encoders as I've decided to load L/Z message bits at a time. This is due the limited number of I/O pins present in my FPGA. Obviously the initialization step can be speeded up loading more than L/Z bits at a time.

For what concerns the decoding performance the IEEE 802.11n QC-IRA codes behave better than the BC-QC codes with a parity check matrix in the form of Equation 2.2 as it is possible to see in Figure 37 and Figure 38. The decoding performance has been found using a code with $n = 648$ bits with rate $R = 1/2$, a BPSK modulation and a Additive White Gaussian Noise (AWGN) channel model. The message vector is first encoded, then the BPSK modulation has been applied mapping each zero to the value $A_s = 1$ and each one to the value $-A_s = -1$. Then, the received vector is computed adding a White Gaussian Noise with zero mean and standard deviation σ_n to the modulated signal. Finally, the received signal is decoded using the sum-product algorithm. These operations are repeated for different values of signal-to-noise ratio (SNR_{dB}) (0 to 10 dB with increments of 0.5 dB) and for each SNR_{dB}, 2000 frames (message

vectors) are used. Then for each value of SNR_{dB}, the total number or errors over all frames is saved in the variable *errors* and the BER for a value of SNR_{dB} is found as

$$BER = \frac{errors}{frames \cdot n} = \frac{errors}{2000 \cdot 648}$$

with

$$SNR_{dB} = 10 \log_{10} \left(\frac{A_s}{\sigma_n} \right)^2 = 10 \log_{10} \left(\frac{1}{{\sigma_n}^2} \right)$$

where $A_s = 1$.

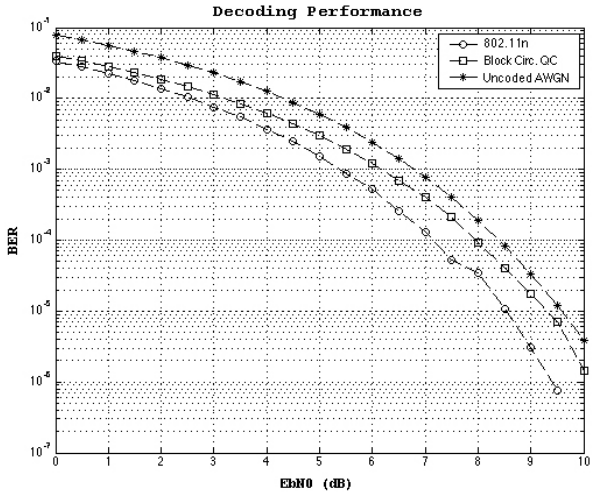

Figure 37. Decoding performance n=648 R=1/2

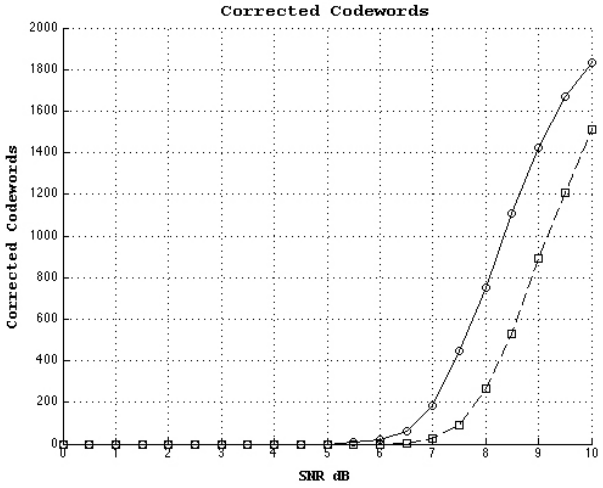

Figure 38. Number of correctly decoded codewords n=648 R=1/2

Finally, both the BC-QC encoder and the IEEE 802.11n encoder require a very low storage space. Indeed, for the BC-QC code, only k_b out of k rows, each with m bits, must be saved in the memory. While for the IEEE 802.11n code the shift amounts, each with $\lceil \log_2 Z \rceil$ bits, have to be saved instead of the Z x Z circulants. Therefore, the BC-QC encoder is able to save

$$\left[\left(1 - \frac{k_b}{k}\right) 100\right] \%$$

of the storage space, while the IEEE 802.11n encoder is able to save

$$\left[\left(1 - \frac{\lceil \log_2 Z \rceil}{Z^2}\right) 100\right] \%$$

of the required memory.

CHAPTER 5

CONCLUSION

5.1 Final Results

Block-Circulant Quasi-Cyclic (BC-QC) codes and Repeat-Accumulate (RA) codes are the most used LDPC codes, because it has been shown that they have really good decoding performance and very low complexity encoders. In this thesis, two different encoder implementations have been analyzed. The BC-QC encoder has been used as reference and it has the following advantages:

- it works with a general BC-QC matrix in the systematic form;
- it has a very parallel and uniform architecture;
- it saves 96-99% of storage space;
- it has good decoding performance.

Its main drawback is related to the fact that the generator matrix is required to encode a message vector. This means that the parity check matrix has to be full rank in order to easily find a generator matrix still in the BC-QC form.

The IEEE 802.11n encoder works with matrices in the standard form as seen in section 4.3.1 and it presents the following advantages:

- it has really low computation time (one order of magnitude less than the QC encoder);

- it saves 99% of storage space;
- it has better decoding performance than the BC-QC code employed in this thesis.

The main disadvantage is that it requires more HW area on the FPGA as it is shown in Figure 39. Nevertheless, a maximum of about 5.5% extra LEs seems a reasonable price to pay for the improved performance. The percentage of extra LEs is computed as the difference between the LEs of the IEEE 802.11n encoder and the LEs of the QC encoder over the 33216 LEs present in the employed FPGA.

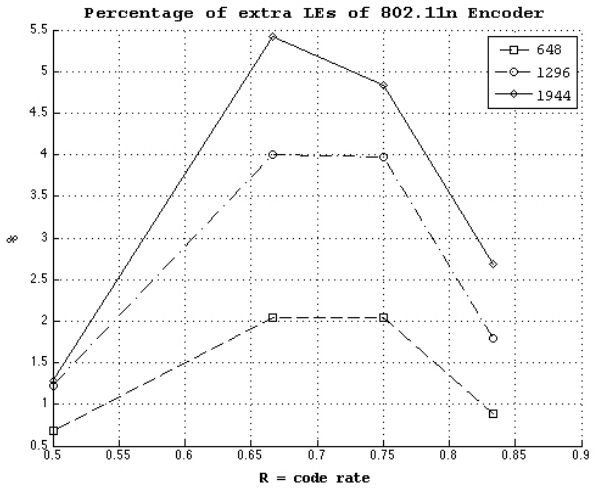

Figure 39. Percentage extra LEs for IEEE 802.11n encoder

This encoder structure can be easily adapted to work with the IEEE 802.16 standard (23). This standard supports 19 code lengths (from 576 bits to 2304 bits), and for each code length the same code rates of the IEEE 802.11n standard are defined (R=1/2, 2/3, 3/4 and 5/6). Nevertheless, the IEEE 802.16 standard defines two different code matrices for the code rates R=2/3 (A and B) and R=3/4 (A and B).

As in the IEEE 802.11n standard, the base matrix size is fixed to $n_b = 24$ for all the codes. This base matrix is divided in two parts:

$$H_b = [H_{1\,m_b \times k_b} \quad | \quad H_{2\,m_b \times m_b}]$$

where H_1 corresponds to the message bits, while H_2 corresponds to the parity check bits.

The submatrix H_2 has the following form:

$$H_2 = [\mathbf{h_b} | H_2{}'] = \left[\begin{array}{c|cccccc} h_{b0} & 0 & - & \cdots & \cdots & \cdots & - \\ - & 0 & 0 & - & \cdots & \cdots & - \\ \vdots & - & 0 & \ddots & - & \cdots & - \\ h_{bx} & - & - & \ddots & \ddots & - & - \\ - & \cdots & \cdots & - & \ddots & \ddots & - \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ h_{b\,m_b-1} & - & \cdots & \cdots & \cdots & - & 0 \end{array}\right]$$

The two items h_{b0} and h_{bm_b-1} have to be equal (same shift amount), while h_{bx} must have a different shift value. In the IEEE 802.11n standard $h_{b0} = h_{bm_b-1} = 1$ and $h_{bx} = 0$ for all the code matrices. The character "-" represents an all-zero matrix. Each number $h_{bi,j}$ in the base matrix represents a right cyclic shift by $h_{bi,j}$ positions of an identity matrix of size $Z = \frac{n}{24}$, where n is the length of the code. All the "0" items in the matrix H_b are no shifted identity matrices.

The standard defines a base matrix only for the code length $n = 2304$ of each code rate. All the other matrices are directly determined using these matrices. The circulant size of the code $n = 2304$ is defined as $Z_0 = \frac{2304}{24} = 96$.

At this point, the shift amounts $h_{bi,j}$ of the base matrix for all other code lengths are obtained as

$$h_{bi,j} = \left\lfloor \frac{h_{bi,j} Z}{Z_0} \right\rfloor \tag{5.1}$$

for code rates 1/2, 2/3 B, 3/4 A/B and 5/6, where $Z = \frac{n}{24}$, and with

$$h_{bi,j} = \mathrm{mod}(h_{bi,j}, Z) \tag{5.2}$$

for code rate 2/3A.

The all-zero matrices indicated by the "-" character remain zero matrices in the expanded matrix. These matrices are indicated with the value -1 in the standard (23). An example of the base model matrix for the rate $R = 1/2$ is in Figure 40. As said before, this matrix is

defined for the code $n = 2304$, therefore the base matrices for the other code lengths have to be computed scaling the values $h_{b_{i,j}}$ in this matrix with Equation 5.1.

-1	94	73	-1	-1	-1	-1	-1	55	83	-1	-1	7	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	27	-1	-1	-1	22	79	9	-1	-1	-1	12	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	24	22	81	-1	33	-1	-1	-1	0	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1
61	-1	47	-1	-1	-1	-1	-1	65	25	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1
-1	-1	39	-1	-1	-1	84	-1	-1	41	72	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	46	40	-1	82	-1	-1	-1	79	0	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1
-1	-1	95	53	-1	-1	-1	-1	-1	14	18	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1
-1	11	73	-1	-1	-1	2	-1	-1	47	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1
12	-1	-1	-1	83	24	-1	43	-1	-1	-1	51	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1
-1	-1	-1	-1	-1	94	-1	59	-1	-1	70	72	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1
-1	-1	7	65	-1	-1	-1	-1	39	49	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0
43	-1	-1	-1	-1	66	-1	41	-1	-1	-1	26	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0

Figure 40. IEEE 802.16 base model matrix for n=2304 R=1/2

As we can see in Table I the values h_{b0} and $h_{b m_b - 1}$ are different for all the code rates, while $h_{b_x} = 0$ for all code rates except of rate $R = 3/4B$. Hence, the only component that has to be modified in the structure of Figure 26 is *rol1*, that has now to left cyclic shift by $h_{b0}/h_{b m_b - 1}$ positions the parity vector $\mathbf{p}_0$ to properly compute $\mathbf{p}_1$ and $\mathbf{p}_{m_b - 1}$. The rest of the architecture can be left as before. This change will work for all code rates except of $R = 3/4B$. For this rate

$\mathbf{p}_0$ has to be left cyclic shifted by h_{bx} to compute $\mathbf{p}_x$. In this case Equation 4.2, Equation 4.3 and Equation 4.6 become:

$$\mathbf{p}_1 = \boldsymbol{\lambda}_0 + \mathbf{p}_0$$

$$\mathbf{p}_{m_b-1} = \boldsymbol{\lambda}_{m_b-1} + \mathbf{p}_0$$

$$\mathbf{p}_x = \mathbf{p}_0{}^{h_{bx}} + \mathbf{p}_{x+1} + \boldsymbol{\lambda}_x$$

where $\mathbf{p}_0{}^{h_{bx}}$ is a circular left shift of $\mathbf{p}_0$ by h_{bx} positions.

TABLE I

H_B VALUES

rate \ $h_{i,j}$	$h_{b0}/h_{b\,m_b-1}$	h_{bx}
1/2	7	0
2/3 A	1	0
2/3 B	95	0
3/4 A	48	0
3/4 B	0	80
5/6	80	0

The right part of a modified architecture is shown in Figure 41 and it is able to work with all rates of the IEEE 802.16 standard. The only component that has been added is the multiplexer inside the circle. The rest of the structure is the same of Figure 26. If the selector *p0_sh* of

the new multiplexer is set to zero the output is the no shifted parity vector $\mathbf{p}_0$, otherwise the output is the vector $\mathbf{p}_0$ shifted by h_{b0}/h_{bm_b-1} positions for all rates except $R = 3/4B$ and by h_{bx} positions for the code rate 3/4B. Furthermore, for all code rates except $R = 3/4B$ the signal *p0_sh* has to be set to one while the parity vectors $\mathbf{p}_1$ and $\mathbf{p}_{m_b-1}$ are being computed. After that, it has to be set to zero to compute $\mathbf{p}_x$. As for the code rate $R = 3/4B$, the signal *p0_sh* has to be first set to zero during the computation of $\mathbf{p}_1$ and $\mathbf{p}_{m_b-1}$ and then to one in order to determine $\mathbf{p}_x$.

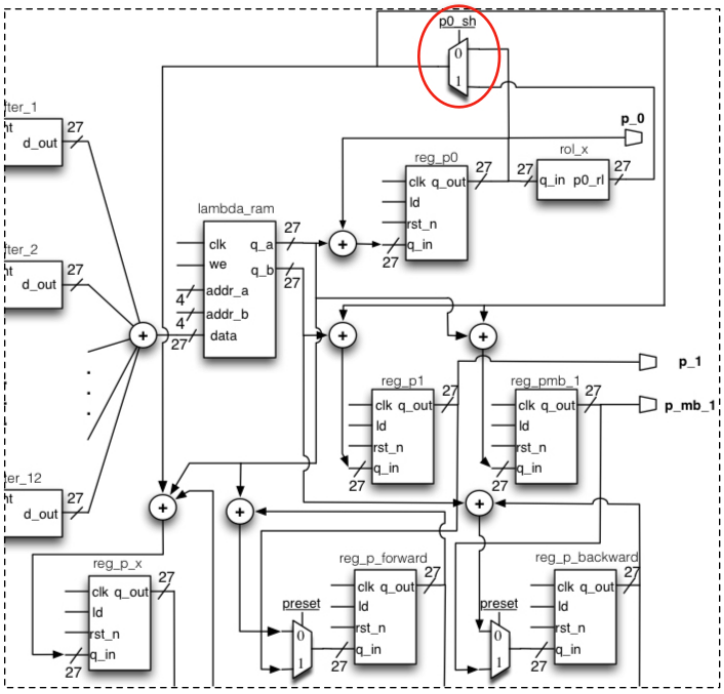

Figure 41. Right part of the modified IEEE 802.11n encoder

5.2 Applications

So far, we have seen the LDPC codes used in both the IEEE 802.11n standard for wireless local area network (WLAN) and the IEEE 802.16 standard for wireless metropolitan area network (WMAN). However, LDPC codes are employed in a lot of applications.

Another example are the *DVB-S2* and *DVB-T2* standards. The former is the Digital Video Broadcasting – Satellite 2nd generation (DVB–S2) standard also known as ETSI EN 302 307 (24) and the latter is the Digital Video Broadcasting – Terrestrial 2nd generation (DVB–T2) standard or ETSI EN 302 755 (25) used for the digital terrestrial television.

Irregular Repeat Accumulate (IRA) codes are used in these two standards. The IRA parity check matrix has the form $H = [H_{1\,m\,x\,k} \quad H_{2\,m\,x\,m}]$, where H_1 is a matrix with an irregular degree distribution and H_2 is a dual-diagonal matrix. More information about RA codes can be found in section 2.2. The codeword of this IRA code has a systematic form, that is $\mathbf{c} = [\mathbf{u}\ \mathbf{p}]$.

The DVB-S2 standard supports two different code lengths (16200 bits and 64800 bits), each with different code rates (R=1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6 and 8/9). The rate 9/10 is only defined for the code length $n = 64800$ bits. As before, the DVB-T2 standard defines two code lengths: 16200 bits and 64800 bits. The former with code rates R=1/4, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 1/3, 2/5 and the latter with code rates R=1/2, 3/5, 2/3, 3/4, 4/5, 5/6.

The message bits in the matrix H_1 are divided in group of $M = 360$ bits. All the bit nodes in a group x must have the same weight w_x. The connection between the first bit node of a group and its w_x check nodes is determined in a pseudo-random way. Let's define $\mathbf{i} = [i_1\ i_2 \cdots_{w_x}]$ as

the positions of the check nodes connected to the first bit node. Then the connection between the other M-1 bit nodes in a group and the corresponding check nodes is determined using

$$(i_1 + j\,q) \mod m, \qquad (i_2 + j\,q) \mod m, \quad \ldots, \quad (i_{w_x} + j\,q) \mod m$$

with $j = 1 \ldots M - 1$ and $q = m/M$. Thus, in the matrix H_1 a new column pattern appears each 360 columns and the columns in the middle are just cyclic shift of the new column to the bottom by q positions.

For instance using a DVB-S2 code with $n = 64800$, $R = 1/4$ and $q = 135$ the first column (j=0) of the matrix H_1 is defined in the Annex B of the standard (24) and has ones in the positions

$$\mathbf{i} = [540\ 1140\ 6226\ 18148\ 18510\ 20879\ 23606\ 23802\ 28859\ 36098\ 42014\ 47088]$$

while the second column (j=1) has ones in the positions $(i_x + q) \mod m = (i_x + 135) \mod 48600$, where i_x is a generic element of the vector $\mathbf{i}$ with $x = 1,\ 2,\ \ldots\ 12$. This means that the ones in the second column are in the following positions

$$[675\ 1275\ 6361\ 18283\ 18645\ 21014\ 23741\ 23937\ 28994\ 36233\ 42149\ 47223].$$

The Tanner representation of these IRA codes is shown in Figure 42, where the edge connections between the check nodes and the parity nodes corresponds to the dual diagonal form of the matrix H_2. Two encoder implementations for DVB-S2 codes can be found in (26) and (27).

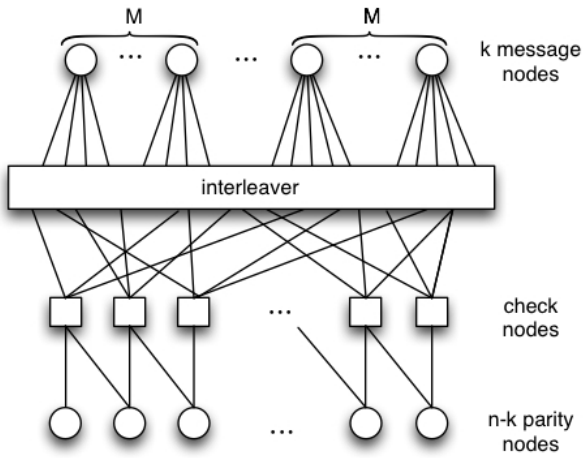

Figure 42. Tanner graph for DVB IRA codes

A LDPC code with $k = 1723$ and $n = 2048$ is also used in the IEEE 802.3an standard for ethernet communications. Cryptography represents another possible application of LDPC codes. One example is a modified McEliece Cryptosystem Based on QC-LDPC Codes (28).

5.3 Future Directions on the Design of LDPC Encoders

A possible enhancement is to make the IEEE 802.11n encoder programmable to let it work with all kind of code rates avoiding to recompile the circuit for a new set of code parameters. Furthermore, the two encoder implementations could be synthesized using the Synopsys® DC Ultra™ tool, which allows one to compile a RTL (Register Transfer Level) design optimizing timing, area and power consumption. Obviously, the ASIC (application-specific integrated circuit) obtained using *Design Compiler* will have a better performance with respect to a FPGA implementation. A FPGA requires a larger area because it contains a really high amount of logic elements. The reason is that a FPGA should be general purpose and reusable. Similarly, a FPGA circuit consumes more power due to its extra components. In addition, an ASIC design can have a higher clock rate because of the custom interconnections. Nevertheless, a higher time to market is required for ASIC designs and the overall cost will be reasonable only with large-volume sales.

CITED LITERATURE

1. MacWilliams, F. and Sloane, N.: The Theory of Error-Correcting Codes. North Holland publishing company, 1977.

2. Tanner, R.: A recursive approach to low complexity codes. Information Theory, IEEE Transactions on, pages 533–547, Sep. 1981.

3. Gallager, R. G.: Low-Density Parity-Check Codes. MIT Press, 1963.

4. MacKay, D. J. C.: Good error-correcting codes based on very sparse matrices. Information Theory, IEEE Transactions on, pages 399–431, Mar. 1999.

5. Ryan, W. E. and Lin, S.: Channel codes, classical and modern. Cambridge University Press, 2009.

6. MacKay, D. J. C.: Information Theory, Inference & Learning Algorithms. New York, NY, USA, Cambridge University Press, 2002.

7. Johnson, S. J.: Iterative Error Correction. Cambridge University Press, 2009.

8. Fan, J. L.: Array Codes as Low-Density Parity-Check Codes. 2nd Int. Symp. Turbo Codes Brest France, pages 543–546, 2000.

9. Tanner, R., Sridhara, D., Sridharan, A., Fuja, T., and Costello, D.: LDPC block and convolutional codes based on circulant matrices. Information Theory, IEEE Transactions on, pages 2966–2984, Dec. 2004.

10. Eleftheriou, E. and Olcer, S.: Low-density parity-check codes for multilevel modulation. Proc. IEEE Int. Symp. Information Theory, page 442, Jun. 2002.

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

12. Richardson, T. J. and Urbanke, R.: Efficient Encoding of Low-Density Parity-Check Codes. IEEE Trans. Inform. Theory, vol. 47, pages 638–656, Feb. 2001.

CITED LITERATURE (continued)

13. Qi, H. and Goertz, N.: Low-Complexity Encoding of LDPC Codes: A New Algorithm and its Performance. Retrieved April 7, 2013, from http://publik.tuwien.ac.at/files/PubDat_166941.pdf, Jul. 2010.

14. Islam, M. R. and Kim, J.: Linear encoding of LDPC codes using approximate lower triangulation with postprocessing. IEEE 20th Int. Symp. Indoor and Mobile Radio comm., pages 1791–1795, Sept. 2009.

15. Li, Z., Chen, L., Zeng, L., Lin, S., and Fong, W. H.: Efficient encoding of quasi-cyclic low-density parity-check codes. IEEE Trans. Communications, vol. 54, pages 71–81, Jan. 2006.

16. Johnson, S. J. and Weller, S. R.: Practical Interleavers for Systematic Repeat-Accumulate Codes. VTC 2006-Spring IEEE 63rd, pages 1358–1362, May 2006.

17. Andrews, K., Dolinar, S., and Thorpe, J.: Encoders for Block-Circulant LDPC Codes. ISIT, Sept. 2005.

18. Quartus II: Web Edition v12.1 Service Pack 1. San Jose, California, Altera Corporation, Feb. 2013. https://www.altera.com/download/software/quartus-ii-we.

19. ModelSim: Altera Starter Edition 10.1b. San Jose, California, Altera Corporation, Nov. 2012. https://www.altera.com/download/software/modelsim-starter.

20. Altera Corporation: Cyclone II Device Handbook, Volume 1. San Jose, California, Feb. 2008. Retrieved April 12, 2013, from http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf.

21. IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007), pages 1–2793, March 2012. Retrieved April 7, 2013, from http://standards.ieee.org/getieee802/download/802.11-2012.pdf.

CITED LITERATURE (continued)

22. Cai, Z., Hao, J., Tan, P., Sun, S., and Chin, P. S.: Efficient encoding of IEEE 802.11n LDPC codes. Electronics Letters vol. 42, pages 1471–1472, Dec. 2006.

23. IEEE Standard for Air Interface for Broadband Wireless Access Systems. IEEE Std 802.16-2012 (Revision of IEEE Std 802.16-2009), pages 1–2542, Aug. 2012. Retrieved April 7, 2013, from http://standards.ieee.org/getieee802/download/802.16-2012.pdf.

24. Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2). ETSI EN 302 307 V1.3.1 (2013-03), March 2013. Retrieved April 7, 2013, from http://www.etsi.org/deliver/etsi_en/302300_302399/302307/01.03.01_60/en_302307v010301p.pdf.

25. Digital Video Broadcasting (DVB); Frame structure channel coding and modulation for a second generation digital terrestrial television broadcasting system (DVB-T2). ETSI EN 302 755 V1.3.1 (2012-04), pages 1–188, April 2012. Retrieved April 7, 2013, from http://www.etsi.org/deliver/etsi_en/302700_302799/302755/01.03.01_60/en_302755v010301p.pdf.

26. Yokokawa, T., Nakane, M., and Kan, M.: A Low Complexity and Programmable Encoder Architecture of the LDPC Codes for DVB-S2. 6th International ITG-Conference on Source and Channel Coding, 2006 4th International Symposium on, pages 1–6, April 2006.

27. Gomes, M., Falcao, G., Sengo, A., Ferreira, V., Silva, V., and Falcao, M.: High throughput encoder architecture for DVB-S2 LDPC-IRA codes. Microelectronics, 2007. ICM 2007. Internatonal Conference on, pages 271–274, Dec. 2007.

28. Baldi, M., Bianchi, M., and Chiaraluce, F.: Security and complexity of the McEliece cryptosystem based on QC-LDPC codes. CoRR, pages 1–22, Sep. 2011. Retrieved April 8, 2013, from http://arxiv.org/abs/1109.5827.

VITA

Personal information

Name	Antonello
Surname	Tartamo
e-mail	antonellotartamo@gmail.com

Education

Dates	**15 Sep 03 – 30 Jun 08**
Title of qualification	Industrial Technical Certificate
Organization	ITIS G.B. Pentasuglia (Italian secondary school)
Dates	**15 Sep 2008 - 15 Sep 2011**
Title of qualification	BSc in Electronics Engineering
Organization	Politecnico di Torino
Dates	**10 Oct 2011 - Present**
Title of qualification	MSc in Embedded Systems (Politecnico di Torino) MSc in Electrical and Computer Engineering (University of Illinois at Chicago)
Organizations	Politecnico di Torino University of Illinois at Chicago

VITA (continued)

Personal skills

Organisational skills	Through my cooperation with an NGO called AIESEC I further enhanced my ability to work as a part of a team, organize events and manage time scheduling.
Computer skills	Programming languages known: C, C++, Visual Basic, PHP, HTML, XHTML, CSS, Bash(scripting), Java HDL languages known: VHDL, SystemC, ECL