Design of a configurable LDPC decoder for high-speed wireless.

Miquel Esteve Martin

June 2012
Abstract

The proposed solution for a LDPC decoder according the IEEE 802.11n standard is a multicode and multi rate solution, being capable to decode the codes defined in it and also it can be reused, after adding extra blocks, to decode turbo codes for example defined in 3GPP standard. Moreover the proposed solution uses very low memory compared with other existing architectures, without a relevant penalty in terms of performance and area, and has a very high throughput performance.
Acknowledgements

I would like to specially thank my supervisor Di Wu who help me a lot specially at the begining when I dindn’t know how to focus a project like this one. However after a deep study and reading a bunch of papers over and over, and the value advices of my supervisor always in the most critical moments when I was lost.

I would also like to mention Eric Tell, because he gave me the oportunity to work at Coresonic AB, implementing my solution in a profesional enviroment.

Before ending this section, I want to dedicate this thesis to my classmates and also the friends I had over all those years of hard study Juanjo, Xavi, Roger, Alex, Tery, Santi and Cristina, as well as the swedish people I go along well and really appreciate, therefore this thesis is also dedicated to you Erikk Hägglund, Madeleine Enghund, Per Fagrell and Björn Saläng.
# Contents

Contents ................................................. 4

1 Introduction and purpose of this work .................. 7

2 LDPC codes ........................................... 8
  2.1 Introduction ........................................ 8
  2.2 TPMP algorithm ..................................... 9
  2.3 Implementation constraints .......................... 11

3 Improved architecture ................................ 12
  3.1 Introduction ......................................... 12
  3.2 Trellis topology ..................................... 12
  3.3 Efficient implementation of $L(u_1 \oplus u_2)$ ........ 15
  3.4 Structured LDPC codes ................................. 17
  3.5 Soft-input soft-output concept ....................... 19
  3.6 TDMP decoding algorithm ............................. 21
  3.7 TPMP versus TDMP algorithm ......................... 23

4 LDPC decoder architecture proposed ................. 25
  4.1 LDPC decoder performance ........................... 25
  4.2 LDPC decoder architecture ........................... 27
  4.3 Comparison with other architectures .................. 32
  4.4 Conclusions ......................................... 36
  4.5 Future work ......................................... 36

A Throughput specifications .............................. 40

B Parity-check matrices definitions .................... 42
## List of Figures

1. Trellis graph representation for every check-node . . . . . . . . 13
2. $g(x) = \ln(1 + e^{-|x|})$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ...
Glossary

LDPC (Low-Density Parity-Check)
H (Parity-Check Matrix)
Z (Permutation matrix size)
R (Code rate)
$L_{CW}$ (Codeword length)
TPMP (Two-Phase Message-Passing)
BP (Belief Propagation)
SPA (Sum-Product Algorithm)
LUT (Look-Up Table)
TDMP (Turbo-Decoding Message-Passing)
LLR (Log-Likelihood Ratio)
APP (A Posteriori Probability)
SISO (Soft-Input Soft-Output)
BER (Bit Error Rate)
FER (Frame Error Rate)
PER (Packet Error Rate)
DFU (Decoding Function Unit)
BR (Barrel Shifter)
MUX (Multiplexer)
IFS (Inter-Frame Space)
MCS (Modulation and Coding Scheme)
GI (Guard Interval)
$N_{ss}$ (Number of spatial streams)
1 Introduction and purpose of this work

This thesis is separated in three parts. The first one explains the basic concepts of LDPC codes, such as the graph representation of a code and the BP or TPMP decoding algorithm, as well as the problems to implement a random LDPC codes.

After this brief introduction to the basic concepts, an alternative representation for an LDPC code is presented. This in combination with an the efficient serial computation of a parity operations, will allow the implementation of the check-node update messages with very efficient and small area without a throughput, performance or area penalty. Additionally, a kind of pseudo-random or structured codes which are used in the standard are presented. This kind of codes turn the LDPC decoding problem into a turbo decoding problem, and therefore a faster convergence speed is achieved. This section ends explaining the TDMP decoding algorithm for structured LDPC codes.

In the last section the proposed hardware architecture is described and compared with other existing architectures, proving that the proposed design in this work is a good solution.
2  LDPC codes

2.1  Introduction

An LDPC code is a linear block code defined by a sparse parity-check matrix $H = [h_{ij}]_{M \times N}$, which can be represented by a bipartite graph. A low-density parity-check matrix is typically be very large, and as its name indicates there are few ones randomly placed, which are spread out over the matrix. In order to show the bipartite graph representation for a parity-check matrix, a reduced matrix is proposed as an example, so considering the following parity-check matrix:

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

The rows of $H$ represent the check-nodes, and the columns the symbol-nodes. Therefore, the bipartite graph has $M$ check-nodes $\{c_1, c_2, ..., c_M\}$ and $N$ symbol-nodes $\{b_1, b_2, ..., b_N\}$, corresponding to the number of the rows and the columns of the parity-check matrix $H$ respectively.

The relation between the parity-check matrix $H$ and the bipartite graph is the following. A symbol node $b_j$ is connected to the check node $c_i$ if the entry $h_{ij}$ of $H$ is 1.

The bipartite graph shown is an example of a regular $(c,r)$-LDPC code, where $c$ is the symbol node degree (number of check-nodes connected into a symbol-node) and $r$ is the check node degree (number of symbol-nodes connected into a check-node).

The key point is that the symbol-nodes connected into a check-node must satisfy one condition. This is the check-node constraint, which is simply that all those bits must have an even parity. This will be explained in details later, but just to introduce the idea, this is what a check-node computes (check-to-symbol-node message) for every symbol-node.
2.2 TPMP algorithm

The TPMP algorithm (Two-Phase Message-Passing Algorithm), also known as BP (Belief Propagation) or SPA (Sum-Product Algorithm), is described in this section. Basically, it consists of two types of messages between the symbol-nodes and the check-nodes (symbol-to-check-node messages and check-to-symbol-node messages).

Then taking into account the trellis structure of a LDPC matrix $H_{MxN}$, and considering the transmitted codeword $u = [u_1, ..., u_N]$, and the received codeword $y = [y_1, ..., y_N]$, the following notation is introduced before to present formally the decoding algorithm:

- $\mathcal{M}(n)$ set of check-nodes connected to the symbol-node $n$
- $\mathcal{M}(n) \setminus m$ excluding the $m$-th check-node
- $\mathcal{N}(m)$ set of symbol-nodes connected to the check-node $m$
- $\mathcal{N}(m) \setminus n$ excluding the $n$-th symbol-node
- $\lambda_{n \to m}(u_n)$ symbol-node to check-node message
- $\Lambda_{m \to n}(u_n)$ check-node message to symbol-node

Before explaining the meaning of $\lambda_{n \to m}(u_n)$ and $\Lambda_{m \to n}(u_n)$, must be defined the LLR (log-likelihood ratio) of a binary random variable $U = \{0,1\}$, this is:

$$L(U) = \ln \left( \frac{P(U = 0)}{P(U = 1)} \right)$$

(1)

Then, knowing that $q_{n \to m}(x)$ is the message that the symbol-node $n$ sends to check-node $m$ indicating the probability of being 0 or 1 of symbol $n$ and $r_{m \to n}(x)$ the message from check-node $m$ to symbol node $n$ indicating the probability of symbol $n$ being 0 or 1, the meaning of $\lambda_{n \to m}(u_n)$ and $\Lambda_{m \to n}(u_n)$, will be clarified. Both are defined as the LLR as follows:

$$\lambda_{n \to m}(u_n) = \ln \left( \frac{q_{n \to m}(0)}{q_{n \to m}(1)} \right)$$

(2)

$$\Lambda_{m \to n}(u_n) = \ln \left( \frac{r_{m \to n}(0)}{r_{m \to n}(1)} \right)$$

(3)
Then the formal description of the TPMP algorithm in log domain consist in the following steps:

Step 0) Initialization
First of all the symbol-node are initialised to the received soft bits. Therefore the symbol-to-check node and the check-to-symbol node are:

\[
\lambda_{n \rightarrow m}(u_n) = L(u_n) \\
\Lambda_{m \rightarrow n}(u_n) = 0
\]

Step 1) Check-node update \( \forall m, n \in N(m) \)
The check-nodes generate the incremental soft information for every symbol-node, computing the check-node constraint. This is what the following expression means:

\[
\Lambda_{m \rightarrow n}(u_n) = 2 \cdot \tanh^{-1} \left\{ \prod_{n' \in N(m) \setminus n} \tanh \left( \frac{\lambda_{n' \rightarrow m}(u_{n'})}{2} \right) \right\}
\] (4)

Step 2) Symbol-node update \( \forall n, m \in M(n) \)
Every check-node receives the check-to-symbol-node messages, and adds this incremental information to the received soft bits:

\[
\lambda_n(u_n) = L(u_n) + \sum_{m \in M(n)} \Lambda_{m \rightarrow n}(u_n)
\] (5)

Then the symbol-to-check-node message is calculated to start a new decoding iteration. At this point, one decoding iteration has been completed, which means that steps 1 and 2 will be executed as many times as desired decoding iterations.

\[
\lambda_{n \rightarrow m}(u_n) = L(u_n) + \sum_{m' \in M(n) \setminus m} \Lambda_{m' \rightarrow n}(u_n)
\] (6)

Step 3) Hard decision
When the final decoding iteration is reached, it is time for the hard decision of the symbol-node information. Going back to the LLR definition, it’s easy to see that, the hard decision consist in taking the sign bits of each soft bit.

\[
\hat{u}_n = \begin{cases} 
0 & \text{if } \lambda_n(u_n) \geq 0 \\
1 & \text{if } \lambda_n(u_n) < 0
\end{cases}
\]
2.3 Implementation constraints

The implementation of the TPMP algorithm of randomly constructed LDPC codes implies an important hardware complexity in terms of interconnect and memory overhead.

One possible way to do that, is implementing the bipartite graph, this is a set of symbol-nodes and check-nodes, and interconnect them. Due the random structure of the parity-check matrix, the interconnection between the nodes is random too. Therefore, for a large number of symbol and check-nodes the interconnection problem becomes significant, in terms of placement, routing and timing. This can be, approximately, only for routing a chip area usage around 50%.

It is also possible to communicate the symbol and check-nodes through memories instead of through an interconnection matrix, and then saving the random interconnection problem. Unfortunately, using this technique a large memory overhead will be required.
3 Improved architecture

3.1 Introduction

In this section, a new way to compute the check-node operations will be introduced, as well as an efficient hardware implementation for those operations. This is performed by using an alternative representation for the bipartite graph, also referred as the trellis graph representation. Additionally, the structured codes (turbo-like codes) will be presented, which avoids the randomness of the conventional codes, will turn the LDPC decoding problem into a turbo-decoding problem. This kind of codes, are defined and used in the IEEE 802.11n standard, so the architecture proposed will try to achieve an efficient implementation to be used as a decoder for that standard. Moreover, the turbo-decoding algorithm for LDPC codes, or TDMP algorithm (Turbo-Decoding Message-Passing algorithm) will converge faster, as it will be explained later on.

To sum up, the theoretical background concepts to understand an efficient architecture for the IEEE 802.11n, will be exposed, as well as the TDMP algorithm. Once this concepts are explained, the reader will be able to understand the proposed architecture.

3.2 Trellis topology

There is an alternative representation for an LDPC code instead of the bipartite graph. Taking into account that the j-th row of $H_{M \times N}$, has $d_c$ ones at positions $\{j_1, \ldots, j_c\}$, then taking the bits $\{u_{j_1}, \ldots, u_{j_c}\}$ of the codeword $u$ will have to satisfy the parity-check constraint of each row has to have even parity:

$$u_{j_1} \oplus u_{j_2} \oplus \ldots \oplus u_{j_c} = 0 \quad (7)$$

If the LLR values are used, the parity-check constraint can be expressed as:

$$L(u_{j_1} \oplus u_{j_2} \oplus \ldots \oplus u_{j_c}) = 0 \quad (8)$$

This is also expressed using the boxplus or Hagenauer’s operator $\boxplus$ as:

$$L(u_{j_1}) \boxplus L(u_{j_2}) \boxplus \ldots \boxplus L(u_{j_c}) = 0 \quad (9)$$

In other words, all elements connected to a check-node will have to satisfy an even parity. Then the trellis representation for every a check-node (Figure 1)
Figure 1: Trellis graph representation for every check-node

consist in representing of the parity states for all possible combinations of each symbol-node connected to a check-node, starting from zero (leftmost node) and ending to zero too. Nodes on top represents even parity states and nodes on bottom the odd parity states.

Then utilising the even parity constraint of each row is possible to obtain the check-node update messages $\Lambda_{m\rightarrow n_i}(u_{n_i})$, knowing the incoming messages $\lambda_{n_i\rightarrow m}(u_{n_i})$. The key point is how $\boxplus$ operations are computed. This is if the boxplus operations are started from the left (forward recursion) and from the right (backward recursion) in groups of two, the forward and backward sets can be defined as:

<table>
<thead>
<tr>
<th>Forward recursion ($\alpha$’s)</th>
<th>Backward recursion ($\beta$’s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha_1 = L(u_{n_1})$</td>
<td>$\beta_{d_c} = L(u_{n_{d_c}})$</td>
</tr>
<tr>
<td>$\alpha_2 = \alpha_1 \boxplus L(u_{n_2})$</td>
<td>$\beta_{d_c-1} = L(u_{n_{d_c-1}}) \boxplus \beta_{d_c}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\alpha_{d_c-1} = \alpha_{d_c-2} \boxplus L(u_{d_c-1})$</td>
<td>$\beta_2 = L(u_{n_2}) \boxplus \beta_3$</td>
</tr>
</tbody>
</table>

With those sets for forward and backward recursions the computation of the check-node update messages can be finally calculated by:

$$
\begin{align*}
\Lambda_{m\rightarrow n_i}(u_{n_i}) &= L(\alpha_{i-1} \oplus \beta_{i+1}), \quad \forall i \neq 1, d_c \\
\Lambda_{m\rightarrow n_1}(u_{n_1}) &= L(\beta_2), \\
\Lambda_{m\rightarrow n_{d_c}}(u_{n_{d_c}}) &= L(\alpha_{d_c-1})
\end{align*}
$$

(10)

The serial computation of the check-to-symbol-node messages through the trellis graph is advantageous than the check-to-symbol-node computation in BP algorithm (4). In the next subsection, an efficient implementation for $\boxplus$ operations will be presented, which simplifies the check-node calculations,
compared with the two hyperbolic tangents that must be carried out in (4), that implies using a LUT for tanh and its inverse, becoming significant when representing a wide range of values. Moreover, a compact hardware implementation will be introduced through a simple approximation. Later on will also shown that its performance is very close to the floating point case, and at the end of the report, a comparison with other hardware architectures will be done, proving that the proposed solution stands out in many aspects, and this will imply a reduced core area.
3.3 Efficient implementation of $L(u_1 \oplus u_2)$

The trellis representation of a LDPC code was presented in the previous section, and was exposed how $\Lambda_{m \rightarrow n_i}(u_{n_i})$ can be expressed as LLR of the XOR of forward and backward recursion $L(\alpha_{i-1} \oplus \beta_{i+1})$. In this section, an efficient computation of XOR operations will be presented.

Let’s start from the formal definition of $L(u_1 \oplus u_2)$. It can be demonstrated, that for statistically and independent random variables $u_1$ and $u_2$:

$$L(u_1 \oplus u_2) = \ln\left(\frac{1 + e^{L(u_1)} \cdot e^{L(u_2)}}{e^{L(u_1)} + e^{L(u_2)}}\right)$$  \hspace{1cm} (11)

Note that:

$$L(u) \oplus \pm \infty = \mp \infty \quad \text{and} \quad L(u) \oplus 0 = 0$$

In order to reduce the computation complexity of $L(u_1 \oplus u_2)$ the expression will be approximated. First of all, rewriting the:

$$L(u_1 \oplus u_2) = \ln(1 + e^{L(u_1)} \cdot e^{L(u_2)}) - \ln(e^{L(u_1)} + e^{L(u_2)})$$

The following expression for $L(u_1 \oplus u_2)$ is purposed in [10], using the Jacobian algorithm for the numerator and the denominator obtaining:

$$L(u_1 \oplus u_2) = \max(0, L(u_1) + L(u_2)) + \ln(1 + e^{-|L(u_1) + L(u_2)|})$$
$$- \min(L(u_1), L(u_2)) - \ln(1 + e^{-|L(u_1) - L(u_2)|})$$  \hspace{1cm} (12)

where the Jacobian algorithm approximates this expression:

$$f(x, y) = \ln(e^x + e^y)$$  \hspace{1cm} (13)

to the following one:

$$f(x, y) \approx \max(x, y) + \ln(1 + e^{-|x-y|})$$  \hspace{1cm} (14)

Note that the max terms in the expression will be very easy to implement in hardware, but there are two other logarithms in the expression. Taking a look at the graphical representation of $g(x) = \ln(1 + e^{-|x|})$ in Figure 2, can be
Figure 2: $g(x) = \ln(1 + e^{-|x|})$

observed that the maximum values is less than 1, and the function decreases very fast (exponentially), being very close to zero for $x > 4$.

The implementation of the logarithm term is typically carried out by a LUT. The quantization levels in the LUT needed to implement this can vary, but the one proposed here, approximates the function with only two values:

$$g(x) = \begin{cases} 
0.0 & \text{if } x > 0 \\
0.5 & \text{if } x \leq 0
\end{cases}$$

This approximation works without significantly affecting the performance. The comparison between the exact calculation and the approximation done through a LUT is shown in Figure 2. Therefore the necessary logic needed to implement $L(u_1 \oplus u_2)$, can be small since this operation will be done with a small logic and a extremely reduced LUT.
3.4 Structured LDPC codes

As it was introduced before, the random structure of the parity-check matrix implies an implementation challenge. However, another kind of parity-check matrices can be used, pseudo-randomly constructed, with a BER performance close to random-constructed codes. This are called structured codes.

A structured LDPC code, consist in decomposing the parity-check matrix $H_{M \times N}$ in smaller $Z \times Z$ submatrices, having $n = N/Z$ block columns, and $m = (1 - R) \cdot n$ block rows. The $Z \times Z$ submatrices can be the identity matrix shifted a certain number of positions, or a null matrix.

The IEEE defines three different codeword lengths, as well as four different code rates for each codeword. Note that the number of block columns is fixed to 24 for all parity-check matrices in the standard, therefore:

<table>
<thead>
<tr>
<th>Codeword (bits)</th>
<th>Z</th>
<th>Code rate (R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>648</td>
<td>27</td>
<td>1/2, 2/3, 3/4, 5/6</td>
</tr>
<tr>
<td>1296</td>
<td>54</td>
<td>1/2, 2/3, 3/4, 5/6</td>
</tr>
<tr>
<td>1944</td>
<td>81</td>
<td>1/2, 2/3, 3/4, 5/6</td>
</tr>
</tbody>
</table>

This codes have the advantage of a hardware-efficient implementation, and a very compact definition, which means that can be stored using a relative small memory. This is easy to see taking a look at one parity-check matrix definition in the standard (Figure 3), only the $Z \times Z$ matrices have been represented, indicating the number of positions that the identity matrix has to be rotated, or “-1” to denote a null $Z \times Z$ matrix.

By the other hand, having the parity-check matrix decomposed into different block rows (equivalent to supercodes for turbo codes), which can be
viewed as a concatenation of $m$ submatrices, and the trellis representation of an LDPC code, can turn the LDPC decoding problem into a turbo decoding problem. Therefore a new decoding algorithm, known as TDMP (Turbo-Decoding Message-Passing) algorithm will be used to decode structured LDPC codes.

Viewing the parity-check matrix as a turbo-like code, have some advantages. First of all, the communication between adjacent supercodes is done through an interleaver or permuter. Taking into account that the non-zero $Z \times Z$ submatrices are the identity matrix shifter a certain number of positions instead a random submatrix, the permuter becomes simply in a barrel shifter.

Additionally, another advantage of this turbo-like codes is that the Turbo-Decoding Message-Passing algorithm converges faster than the traditional algorithm.

These concepts will be probably understood better when the TDMP algorithm is explained, later on.
3.5 Soft-input soft-output concept

For communications engineering, the Baye’s theorem [14] is very useful for hypothesis testing, and estimate the probability of the transmitted data when an AWGN channel may corrupt the information. Then, considering the transmitted data ‘d’ and the received data plus noise ‘x’ (continuous and observable value):

\[
P(d = i|x) = \frac{p(x|d = i)P(d = i)}{p(x)}
\]

where:

- \(P(d = i|x)\) represents the a posteriori probability (APP).
- \(p(x|d = i)\) is the probability density function of the continuous-valued data-plus-noise signal ‘x’.
- \(P(d = i)\) is the a priori probability.
- \(p(x)\) is the probability density function of the received signal ‘x’.

and ‘i’ is a set of M classes, but taking into account the binary case for a BSPK modulation \(i = \{-1, +1\}\).

Once this was introduced, if the LLR definition is applied to the Baye’s rule, and \(L'(\hat{d})\) is used to denote the LLR of the APP:

\[
L'(\hat{d}) = \ln \left( \frac{P(d = +1|x)}{P(d = -1|x)} \right)
\]

Then the LLR of the Baye’s rule, the APP can be expressed as an addition of LLR, therefore the soft decision output of a detector is shown in the following equation as:

\[
L(d|x) = L(x|d) + L(d) \quad (16)
\]

\[
L'(\hat{d}) = L_e(x) + L(d) \quad (17)
\]

There is another term called extrinsic information \(L_e(\hat{d})\), that provides incremental information gained in each decoding iteration. This information generated at one decoding iteration is used as an intrinsic information for
the next decoding iteration. Then considering extrinsic information, the APP LLR is:

\[ L(\hat{d}) = L_c(x) + L(d) + L_e(\hat{d}) \]  

(18)

Note that assuming an AWGN channel, the LLR channel value or \( L_c(x) \) is calculated as:

\[
L_c(x) = \ln \left( \frac{p(x|d = +1)}{p(x|d = -1)} \right) = \ln \left( \frac{\frac{1}{\sigma \sqrt{2\pi}} e^{\left(-\frac{1}{2} \cdot \frac{(x+1)^2}{\sigma^2}\right)}}{\frac{1}{\sigma \sqrt{2\pi}} e^{\left(-\frac{1}{2} \cdot \frac{(x-1)^2}{\sigma^2}\right)}} \right) = \frac{2}{\sigma} x
\]

The iterative siso decoding scheme is illustrated in Figure 4. For the first decoding iteration the data is supposed to be equally likely, and therefore \( L(d) = 0 \). After every decoding iteration the extrinsic information generated is going to be used as a priori information for the next iteration.
3.6 TDMP decoding algorithm

Once the soft decoding was introduced, the turbo decoding principle is going to be used to decode LDPC codes. This algorithm is described in [2] and [3]. Before exposing the algorithm, the notation will be presented. Taking into account the soft decoding parameters introduced in the previous section, this is the a priori and channel value as inputs, and extrinsic and posterior message as outputs, will be denoted as:

- Received channel LLR value $\delta$.
- A priori or intrinsic information $\lambda$.
- Extrinsic information $\Lambda$.
- Output posterior messages $\Gamma$.

Additionally $\Pi_i$ and $\Pi_i^{-1}$ will be used to denote the permutation and inverse permutation of a full block row. That means that the LLR values will be sorted according to the permutation matrices in $H$. The way to do it is the following. Consider the $j$-th block row of the LDPC matrix $H$, with $c$ permutation matrices $\{\Pi_{j1},...,\Pi_{jc}\}$ in that block row of size $S \times S$, where each permutation matrix is a cyclic permutation of the identity matrix. Then if $S$ LLR messages have to be permuted according to one of those permutation matrices, implies that $\Pi_{ji}$ operation will perform a circular left shift and $\Pi_{ji}^{-1}$ a circular right shift as much as the permutation matrix indicates.

After introducing the notation, the algorithm will be presented below, and consist in the following 3 steps.

1) First of all, the posterior memory $\gamma$ is initialized to the received channel value $\delta$ the intrinsic memory $\lambda$ is set to zero.

$$\gamma = \delta$$

After the initialization, the steps 2 and 3 are going to be done for all constituent codes in the LDPC matrix $H_{MxN}$, from $i = 1$ to $D$ to complete one decoding iteration, where $D$ is the number of block rows in $H_{MxN}$.

2) Then the decoder input is computed as the following equation states. Note that in order to avoid the correlation between the messages the $\lambda_i$ is subtracted. This is the same that was done in the TPMP algorithm to compute the symbol to check-node update messages.

$$\rho = \Pi_i(\gamma) - \Lambda_i$$
3) Finally, the SISO decoder generates the extrinsic message output $\Lambda_i$. This is going to be used at the next iteration as an intrinsic information, and therefore it is written back to the intrinsic memory replacing the old intrinsic value $\lambda_i$. The decoder output is also used to generate the posterior information $\Gamma$, which is going to be written back to the posterior memory $\gamma$.

$$\Gamma = \rho + \Lambda_i$$

$$\gamma = \Pi_i^{-1}(\Gamma)$$

When steps 2 and 3 are done for all block rowsm this is D time, a single decoding iteration is completed. After that, it can be done for a certain number of iterations.
3.7 TPMP versus TDMP algorithm

In this section the performance comparison between the TPMP and TDMP algorithm will be exposed.

Previously, the advantages of TDMP algorithm were introduced. BER and FER simulations are shown in order to illustrate the faster convergence speed of the TDMP compared with the TPMP algorithm. To understand why this happens, the way that check-nodes are computed has to be considered. In other words, in the BP algorithm, to complete a decoding iteration, the symbol-nodes send the symbol-to-check-node messages to the check-nodes and then the check-nodes reply to the symbol-nodes with a check-to-symbol-node message. That means that the incremental information is only generated at the end of one iteration.

On the other hand, the TPMP algorithm the decoding process consists in decoding the block rows one by one. This implies that the check-nodes are decoded by groups or clusters of \( Z \) check-nodes, and therefore, before finishing a decoding iteration or in other words after decoding a single block row,
the symbol-nodes have new incremental information.
The simulations are performed for a codeword length of 1944 bits, a code rate of 1/2 and at 5 iterations. The TDMP algorithm is represented in blue and BP algorithm in red. It is easy to see that TDMP algorithm is at least one order of magnitude better than BP algorithm at 5 iterations. Note that this difference is reduced for more decoding iterations. Even though, the TDMP algorithm performance is always better.
4 LDPC decoder architecture proposed

4.1 LDPC decoder performance

In this section the FER performance of the proposed hardware will be shown, through a C++ model simulations, as well as the way to calculate the throughput that it can achieve.

First of all, going back to the efficient computation section, a LUT was used to approximate the logarithm terms. This approximation forces to work with one fractional bit. Additionally, the minimum number of bit to represent a soft bit is 7 bits (6 integer bits and 1 fractional bit). As it is shown in Figure 7, can be observed that the FER performance for the floating point case (in blue) and for the hardware (in red) are almost the same, and therefore it can be considered a good approximation.

Note that the extrinsic information generated is saturated to a 5-bit arithmetic (4 integer bits and 1 fractional bit), reducing significantly the intrinsic memory size, as it will be shown when comparing the proposed architectures.
with other existing designs.

Then, knowing that the number of cycles needed to compute one block row is two times the check-node degree, a single iteration needs to decode m block rows, the throughput can be expressed with the following formula:

$$\Theta = \frac{L_{cw} \cdot R \cdot f_{clk}}{N_{iter} \cdot 2 \cdot \sum_{i=1}^{m} c_i}$$

where:
- $L_{cw}$ is the codeword length.
- $R$ is the code rate.
- $f_{clk}$ is the clock frequency.
- $N_{iter}$ is the number of decoding iterations performed.
- $2 \cdot c_i$ are the cycles to decode a block row ($c_i$ is the check-node degree).
- $\sum_{i=1}^{m} c_i$ is the addition all block row check-node degree.

Note that the latency (in cycles), is:

$$\ell = N_{iter} \cdot 2 \cdot \sum_{i=1}^{m} c_i$$

<table>
<thead>
<tr>
<th>$L_{cw}$ (bits)</th>
<th>Code rate</th>
<th>$\sum_{i=1}^{m} c_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>648</td>
<td>1/2, 2/3, 3/4, 5/6</td>
<td>88, 88, 88</td>
</tr>
<tr>
<td>1296</td>
<td>1/2, 2/3, 3/4, 5/6</td>
<td>86, 88, 88, 85</td>
</tr>
<tr>
<td>1944</td>
<td>1/2, 2/3, 3/4, 5/6</td>
<td>86, 88, 85, 79</td>
</tr>
</tbody>
</table>
4.2 LDPC decoder architecture

The decoder architecture is shown in Figure 8. It connects the APP memory (received codeword) to Z SISO decoding units (referred as SISO units) in parallel through a permuter. This permuter is reused to inverse permute when writing back into the APP memory. The reason for that is explained later on, when SISO unit is described (Figure 10), but basically it is because after reading all posterior memories from the memory, the extrinsic outputs and therefore the new posterior messages start to be generated, which means that the permuter can be reused.

The number of SISO units needed to be used in parallel is related with the permutation matrix size, in the parity-check matrices definitions. This can be $Z = \{27, 54, 81\}$ when $L_{CW} = \{648, 1296, 1944\}$, and is equivalent to say the number of check-nodes to be executed in parallel, for computing a block row. The worst case is 81, so 81 SISO units in parallel will be needed.

Note that for $Z = \{27, 54\}$ some SISO units will be inactive, but if the number of SISO units is reduced to 27 the hardware usage is optimized but the throughput and latency will be affected, and therefore the standard specifications will not be accomplished.
- SISO unit

The SISO unit is basically composed by 3 DFUs (Decoding Function Units), one for calculating the \( \alpha \) values, one for \( \beta \) and the last one is used to calculate the extrinsic information (\( \lambda \) values). In other words, this computes extrinsic values for the trellis of a single check-node (Figure 9).

Additionally, two stacks are required, and the reason is easy to understand when looking at the message flow of a single SISO unit (Figure 10). The posterior values are provided serially, and when the first value is received the forward recursion or \( \alpha \) values can start to be computed. However, the backward recursion can not start until the last value is received. Once it starts, it needs to take the inputs starting from the last one and going back to the previous received values. This is the reason that one stack is required at the beginning to store the inputs.

Additionally, the \( \alpha \) values have to be stored because the extrinsic values will not start to be generated until the backward recursion starts. Note that the first and the last extrinsic values only depends on the \( \beta_2 \) and \( \alpha_{c-1} \), and the time needed to generate all extrinsic values is exactly \( 2 \) times the check-node degree. This is important to understand how the throughput was calculated, see equation (19).
The Decoding Function Unit (Figure 11) is the responsible of ⊕ operations. Therefore it is used to calculate the forward, backward and extrinsic values. Then the computation of \( x \oplus y = Q(x, y) \), has a simple implementation due the max terms and the approximation for the logarithm explained before, see Figure 2. The computation of \( Q(x, y) \) will be separated into the max terms and the logarithm terms.

\[
Q(x, y) = \max(0, x + y) + \ln(1 + e^{-|x+y|}) - \max(x, y) - \ln(1 + e^{-|x-y|})
\]

In other words, these terms are computed simultaneously:

1) \( \max(0, x + y) - \max(x, y) \)
2) \( \ln(1 + e^{-|x+y|}) - \ln(1 + e^{-|x-y|}) \)

The delay of both branches, is as much one adder opeation (apart of some combinational logic delay), when both branches the max terms result are added to the logarithm terms. This implementation let the DFU be executed at 500 MHz without any problems. This is the component that restricts the clock frequency to be used. Anyway the decoder performance is evaluated at 400 MHz, and the maximum throughput provided is at this clock frequency instead of 500 MHz.
Every siso unit has its intrinsic memory, and it has to store the extrinsic values generated for each block row. Taking a look at the parity-check matrices, a single intrinsic memory will have to store as much 88 extrinsic values. This number corresponds to the no null elements in H, taking into account the worst case.

Note that a saturation is applied when storing the values, from a 7-bits arithmetic to a 5-bits arithmetic, saving an important amount of memory. Additionally when reading from the intrinsic memory a bit alignment must be done, from 5 to 7 bits.

- Permuter

The permuter is basically a right or a left shifter. The right shift operation corresponds to the inverse permute operation or in other words when writting into APP memory, and left shift when reading.

The typical way to do it is through a barrel shifter, but the number of ele-
ments to shift right or left from APP memory is not a power of two, it’s 81, and therefore the implementation of the shifter is done by using two barrel shifters (BR1 and BR2), as it is shown in Figure 12.

Each barrel shifter is a 128 element shifter, and are connected to the the 81 soft values from the APP memory. The other inputs will not be used. If it want to shift 81 elements c positions right or left, BR1 and BR2 will be shifted c1 and c2 positions, and at the end a selection network is the responsible to select a range of k elements from BR1, and the others from BR2. Considering that $Z_{MAX} = 128$, the control signals are generated like this:

<table>
<thead>
<tr>
<th>operation</th>
<th>c1</th>
<th>c2</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>left shift</td>
<td>$Z_{MAX} - Z + c$</td>
<td>$c$</td>
<td>$Z - c + 1$ to $Z_{MAX}$</td>
</tr>
<tr>
<td>right shift</td>
<td>$Z_{MAX} - c$</td>
<td>$Z - c$</td>
<td>1 to $Z - c$</td>
</tr>
</tbody>
</table>

Figure 12: Permuter implementation
4.3 Comparison with other architectures

To conclude the report, the proposed decoder has to be compared with other existing architectures, in order to analyze the pros and cons, and determine if the proposed implementation is a good design. Therefore an honest comparison will be done, and it will be demonstrated that the LDPC decoder proposed, is better in some aspect with existing designs, see references [6], [8] and [9].

It will be observed that other proposals have a large intrinsic memory overhead compared with the intrinsic memory in this work. By the other hand, the core area is significantly lower compared with [6] and [8], having a throughput performance high enough to fulfil the IEEE 802.11n requirements, being faster that other proposals and having a very low latency.

- Intrinsic and posterior memory coparison

The most remarkable feature of the proposed architecture is its reduced intrinsic memory compared with other existing designs. The intrinsic memory is one of largest parts in the LDPC decoders, and it is significantly reduced in this work by simply doing a saturation from a 7 bits arithmetic to a 5 bit arithmetic when writting into the intrinsic memory.

Knowing that each single siso unit needs to store as much 88 elements of 5 bits, then 88x5x81 bits will need to be stored, since 81 single siso units are used in parallel.

<table>
<thead>
<tr>
<th></th>
<th>This work</th>
<th>8</th>
<th>9</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ-memory (bits)</td>
<td>35640</td>
<td>47520</td>
<td>55224</td>
<td>62208</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>0.00⁰</td>
<td>+25.0</td>
<td>+35.46</td>
<td>+42.71</td>
</tr>
</tbody>
</table>

The posterior memory is not very far from [9] that is using less number of bits, as it is shown in the following table.

<table>
<thead>
<tr>
<th></th>
<th>This work</th>
<th>8</th>
<th>9</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP memory (bits)</td>
<td>13608</td>
<td>20736</td>
<td>12960</td>
<td>15552</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>0.00⁰</td>
<td>+34.38</td>
<td>-5.00</td>
<td>+12.50</td>
</tr>
</tbody>
</table>
The extra amount of memory required in % is based on the proposed design, which is taken as a reference.

- Area comparison

The other part in the LDPC decoder that can be significantly large is the LDPC core area. This is the decoding process unit, or in other words the siso decoders used in parallel. In this work 81 siso units are used in parallel.

<table>
<thead>
<tr>
<th>Area (kGE)</th>
<th>This work</th>
<th>[9]</th>
<th>[8]</th>
<th>[6]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>81²</td>
<td>60.8</td>
<td>195</td>
<td>217</td>
</tr>
<tr>
<td>Total</td>
<td>N/A³</td>
<td>349.5</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Note that the two internal stacks are not included in this block, instead of that, they were implemented through dedicated memories outside the siso unit, in order to reduce its area. Note that the core equivalent area (kGE) is an expected value that must be verified yet, through a synthesis. The estimation is based on the area of a single DFU.

The global architectrue is not evaluated yet, and therefore the total decoder area in ($mm^2$) can not be provided. Even though, it is expected to be very small since the intrinsic memory was significantly reduced compared with other works, and the core area is also very small compared with [6] and [8] and close to the best impletation [9].

- Throughput performance (Mbps) at 5 iterations

The throughput performance of the proposed decoder in this work is compared for all codeword lengths and code rates with the proposed in [8] and [9]. The [6] does not provide this information. Instead of that, the maximum throughput is given, and will be also compared with the maximum through-
put achieved with the different architectures.

As it can be observed, the proposed decoder in this work is faster than [8] and [9] for codeword lengths of 1296 and 1944 bits, and is only slower than [8] at 648 bits. Anyway, it will be explained later on that [8] is not fast enough to cover all the standard requirements, since the $\Theta_{MAX}$ defined in the standard [13] is 600 Mbps when 4 spatial streams are used.

<table>
<thead>
<tr>
<th>$L_{cw}$</th>
<th>R</th>
<th>This work</th>
<th>[9]</th>
<th>[8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>648</td>
<td>1/2</td>
<td>147</td>
<td>134</td>
<td>246</td>
</tr>
<tr>
<td></td>
<td>2/3</td>
<td>196</td>
<td>190</td>
<td>330</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>221</td>
<td>212</td>
<td>349</td>
</tr>
<tr>
<td></td>
<td>5/6</td>
<td>245</td>
<td>222</td>
<td>410</td>
</tr>
<tr>
<td>1296</td>
<td>1/2</td>
<td>301</td>
<td>286</td>
<td>236</td>
</tr>
<tr>
<td></td>
<td>2/3</td>
<td>393</td>
<td>373</td>
<td>330</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>442</td>
<td>419</td>
<td>349</td>
</tr>
<tr>
<td></td>
<td>5/6</td>
<td>508</td>
<td>471</td>
<td>389</td>
</tr>
<tr>
<td>1944</td>
<td>1/2</td>
<td>452</td>
<td>415</td>
<td>178</td>
</tr>
<tr>
<td></td>
<td>2/3</td>
<td>589</td>
<td>565</td>
<td>233</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>686</td>
<td>656</td>
<td>259</td>
</tr>
<tr>
<td></td>
<td>5/6</td>
<td>820</td>
<td>780</td>
<td>292</td>
</tr>
</tbody>
</table>

The maximum throughput that can be achieved with all architectures is presented in the following table. As it can be observed, the proposed architecture is faster than [8] and [9].

<table>
<thead>
<tr>
<th>$\Theta_{MAX}$ (Mbps)</th>
<th>This work</th>
<th>[9]</th>
<th>[8]</th>
<th>[6]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>820</td>
<td>780</td>
<td>410</td>
<td>1000</td>
</tr>
</tbody>
</table>

Before concluding the throughput comparison, the throughput requirements defined in the IEEE 802.11n must be explained. According to the standard [13], a maximum throughput of 600 Mbps will be needed, when four spatial streams moded is used. In this case, the architecture proposed in [8] will not be enough to satisfy the standard requirements.
- Latency comparison (µs) at 5 decoding iterations

Apart of the throughput performance, there is another design parameter that must be taken into account, the decoder latency. According to the standard [13] the time interval between frames or IFS is fixed to 10 or 16 µs for a 2.4 and 5 GHz band mode respectively. A common assumption [12], is that a 6 µs time slot is reserved for the channel decoder, and therefore the latency constraint is \( \ell \leq 6\mu s \).

Running the hardware at a clock frequency of 400 MHz, can achieve this goal, without using a lot of the channel decoder time slot.

The following table shows the comparison between the latency at 5 iterations with [8], which is the only reference that provides this information.

<table>
<thead>
<tr>
<th>(L_{cw})</th>
<th>R</th>
<th>This work</th>
<th>[8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>648</td>
<td>1/2</td>
<td>2.20</td>
<td>5.67</td>
</tr>
<tr>
<td></td>
<td>2/3</td>
<td>2.20</td>
<td>5.65</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>2.20</td>
<td>5.98</td>
</tr>
<tr>
<td></td>
<td>5/6</td>
<td>2.20</td>
<td>5.68</td>
</tr>
<tr>
<td>1296</td>
<td>1/2</td>
<td>2.15</td>
<td>5.90</td>
</tr>
<tr>
<td></td>
<td>2/3</td>
<td>2.20</td>
<td>5.65</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>2.20</td>
<td>5.97</td>
</tr>
<tr>
<td></td>
<td>5/6</td>
<td>2.13</td>
<td>5.96</td>
</tr>
<tr>
<td>1944</td>
<td>1/2</td>
<td>2.15</td>
<td>5.75</td>
</tr>
<tr>
<td></td>
<td>2/3</td>
<td>2.20</td>
<td>5.87</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>2.13</td>
<td>5.95</td>
</tr>
<tr>
<td></td>
<td>5/6</td>
<td>1.98</td>
<td>5.85</td>
</tr>
</tbody>
</table>
4.4 Conclusions

At this point, a detailed description of the decoder and its performance was exposed. Additionally, it was compared with other existing architectures, in terms of memory usage, core area, and latency.

To sum up, the intrinsic memory saturation from a 7-bits arithmetic to a 5-bit arithmetic, allows the decoder reduce significantly the intrinsic memory size compared with the other architectures without an important distortion. This is probably the main merit of this work, because the intrinsic memory is probably one of the biggest components in the decoder. This important reduction compensates the stacks area. The core uses this stacks in order to reuse the architecture for any other kind of codes and standards.

The core area is also another big part in the LDPC decoder, and the approximation done was in a very compact and fast implementation. It was shown that is not that far from [9], which is probably the most compact implementation, but as it was said before the proposed architecture in this work is intended to be for multi-code and a multi-standard solution.

To end this section, the the proposed architecture, is shown to be fast enough to satisfy the standard requirements, and also one of the fastest implementations, only slower than [6].

Therefore, all these aspects contribute to a compact, fast and the possibility to extend it to other codes and standards. Looking at the results, the proposed architecture is not the best in terms of throughput, and area, but in all cases its performance is very close to the best one. The best proposal taking into account all aspects is [6], which offers a very good throughput performance and a very small core area.

By the other hand [8] and [9] have a very large core area, but even that throughput can be higher, is more important to provide a throughput that can satisfy the standard requirements, and try to reduce the area instead of focus the design in provide a massive throughput.

For all these reasons, it can be concluded that the architectures proposed in this work can be considered a very good solution.

4.5 Future work

The full architecture has to be evaluated yet, that means that the memories, the permuter, the decoder core and the control unit have to be connected, and parameters such as the total area and the power consumption must be
analyzed, even though the area is expected to be small.

In addition, once the full architecture is completed, the next step is to add the extra part to make the solution reusable for other codes and standards. The SISO unit was designed taking into account this. In [7], a turbo-LDPC decoder architecture is introduced, which uses the same structure, so to extend the proposed architecture in this work, this is a good point to get started.
References


A  Throughput specifications

Data rates up to 600 Mbit/s are achieved only with the maximum of four spatial streams. Different modulation schemes and coding rates (1/2, 2/3, 3/4, 5/6) are defined by the standard and are represented by a MCS index value. The table below shows the relationships between the variables that allow for the maximum data rate, and it doesn’t represent all cases defined in the standard. For further details, and a complete specification for all possible cases defined, see pages from 345 to 354 in [13].

<table>
<thead>
<tr>
<th>MCS index</th>
<th>$N_{ss}$</th>
<th>Modulation</th>
<th>R</th>
<th>Data rate (Mbps)</th>
<th>20 MHz channel</th>
<th>40 MHz channel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>800 ns GI</td>
<td>400 ns GI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>800 ns GI</td>
<td>400 ns GI</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>BPSK</td>
<td>1/2</td>
<td>6.50</td>
<td>7.20</td>
<td>13.50</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>QPSK</td>
<td>1/2</td>
<td>13.00</td>
<td>14.40</td>
<td>27.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>QPSK</td>
<td>3/4</td>
<td>19.50</td>
<td>21.70</td>
<td>40.50</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>16-QAM</td>
<td>1/2</td>
<td>26.00</td>
<td>28.90</td>
<td>54.00</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>16-QAM</td>
<td>3/4</td>
<td>39.00</td>
<td>43.30</td>
<td>81.00</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>64-QAM</td>
<td>2/3</td>
<td>52.00</td>
<td>57.80</td>
<td>108.00</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>64-QAM</td>
<td>3/4</td>
<td>58.50</td>
<td>65.00</td>
<td>121.50</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>64-QAM</td>
<td>5/6</td>
<td>65.00</td>
<td>72.20</td>
<td>135.00</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>BPSK</td>
<td>1/2</td>
<td>13.00</td>
<td>14.40</td>
<td>27.00</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>QPSK</td>
<td>1/2</td>
<td>26.00</td>
<td>28.90</td>
<td>54.00</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>QPSK</td>
<td>3/4</td>
<td>39.00</td>
<td>43.30</td>
<td>81.00</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>16-QAM</td>
<td>1/2</td>
<td>52.00</td>
<td>57.80</td>
<td>108.00</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>16-QAM</td>
<td>3/4</td>
<td>78.00</td>
<td>86.70</td>
<td>162.00</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>64-QAM</td>
<td>2/3</td>
<td>104.00</td>
<td>115.60</td>
<td>216.00</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>64-QAM</td>
<td>3/4</td>
<td>117.00</td>
<td>130.00</td>
<td>243.00</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>64-QAM</td>
<td>5/6</td>
<td>130.00</td>
<td>144.40</td>
<td>270.00</td>
</tr>
</tbody>
</table>
The table continues here from the MCS index 16 to 31. As it was stated before, the maximum throughput requirements (600 MHz) is at the last case for a 40 MHz channel and a GI of 400 ns.

<table>
<thead>
<tr>
<th>MCS index</th>
<th>(N_{ss})</th>
<th>Modulation</th>
<th>(R)</th>
<th>Data rate (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20 MHz channel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>800 ns GI</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>BPSK</td>
<td>1/2</td>
<td>19.50</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>QPSK</td>
<td>1/2</td>
<td>39.00</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>QPSK</td>
<td>3/4</td>
<td>58.50</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>16-QAM</td>
<td>1/2</td>
<td>78.00</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>16-QAM</td>
<td>3/4</td>
<td>117.00</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>64-QAM</td>
<td>2/3</td>
<td>156.00</td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>64-QAM</td>
<td>3/4</td>
<td>175.50</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
<td>64-QAM</td>
<td>5/6</td>
<td>195.00</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td>BPSK</td>
<td>1/2</td>
<td>26.00</td>
</tr>
<tr>
<td>25</td>
<td>4</td>
<td>QPSK</td>
<td>1/2</td>
<td>52.00</td>
</tr>
<tr>
<td>26</td>
<td>4</td>
<td>QPSK</td>
<td>3/4</td>
<td>78.00</td>
</tr>
<tr>
<td>27</td>
<td>4</td>
<td>16-QAM</td>
<td>1/2</td>
<td>104.00</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td>16-QAM</td>
<td>3/4</td>
<td>156.00</td>
</tr>
<tr>
<td>29</td>
<td>4</td>
<td>64-QAM</td>
<td>2/3</td>
<td>208.00</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td>64-QAM</td>
<td>3/4</td>
<td>234.00</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
<td>64-QAM</td>
<td>5/6</td>
<td>260.00</td>
</tr>
</tbody>
</table>
B Parity-check matrices definitions

The full H matrices are represented in this annex. For each possible codeword length \{648, 1296, 1944\}, four H matrices are defined for every possible code rate \{1/2, 2/3, 3/4, 5/6\}.

Additionally, every entry represents a ZxZ matrix, which is the identity matrix shifted as much positions as the number indicates or a null matrix when “-1”.

![Table of parity-check matrices]

Figure 13: \( L_{CW} = 648, Z = 27 \) and \( R = \{1/2, 2/3, 3/4, 5/6\} \)
Figure 14: $L_{CW} = 1296$, $Z = 54$ and $R = \{1/2, 2/3, 3/4, 5/6\}$
Figure 15: $L_{CW} = 1944$, $Z = 81$ and $R = \{1/2, 2/3, 3/4, 5/6\}$