LDPC Encoder

A generic FEC encoder looks like this

$$\underline{u} \longrightarrow \boxed{G} \longrightarrow \underline{c}$$

The message word (or message vector) has dimensions 1×4096, in the case of this LDPC code. Each position in this vector stores one bit

$$\underline{u} = [u_0, u_1, u_2, \cdots, u_{4095}]$$

For this LDPC code, the code word (or code vector) has dimensions 1×6144, where each position stores a bit

$$\underline{c} = [c_0, c_1, c_2, \cdots, c_{6143}]$$

In a brute force implementation, the codeword is generated with a matrix-vector multiplication, where the arithmetic is computed in a modulo-2 fashion

$$\underset{1 \times 6144}{\underline{c}} = \left(\underset{1 \times 4096}{\underline{u}} \cdot \underset{4096 \times 6144}{G}\right)_{\text{mod-2}}$$

The required dimensions of the generator matrix are implied by the rules of linear algebra.

Systematic Generator Matrix

When a generator is in systematic form, it can be partitioned into the identity matrix and the parity generator matrix

$$G = \left[\underset{4096 \times 4096}{I} \;\vdots\; \underset{4096 \times 2048}{W} \right]$$

Because of this structure, the codeword has the following format

$$\underset{\substack{1 \times 6144 \\ \text{code bits}}}{\underline{c}} = \left[\underset{\substack{1 \times 4096 \\ \text{message} \\ \text{bits}}}{\underline{u}} \;\vdots\; \underset{\substack{1 \times 2048 \\ \text{parity} \\ \text{bits}}}{\underline{p}} \right]$$

Therefore, the real job of a systematic encoder is to generate the parity bits. Again, a brute force implementation is

$$\underset{1 \times 2048}{\underline{p}} = \left(\underset{1 \times 4096}{\underline{u}} \cdot \underset{4096 \times 2048}{W} \right)_{\text{mod-}2}$$

Quick Definitions of Mod-2 Arithmetic

Multiplication

$a \cdot b$

a	b	$a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

This is easy to recognize as the truth table for the logical AND

$$a \cdot b = a \text{ AND } b$$

Addition

$a \oplus b$

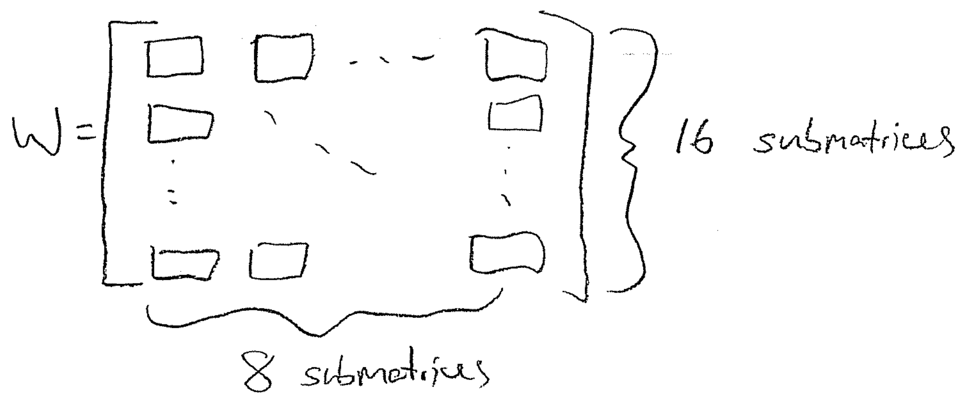
a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

This is easy to recognize as the truth table for the logical XOR

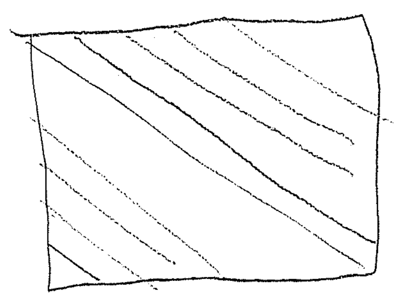
$$a \oplus b = a \text{ XOR } b$$

Circulant Structure of the JPL LDPC Code

The W submatrix of the generator matrix G for the JPL code has its own submatrices



Each of these submatrices has dimensions 256×256 and each is "circulant," which means that the second row/column is a circular (barrel) shift of the first row/column, and the third row/column is a circular (barrel) shift of the second row/column, and so on. Thus, only the first row/column needs to be stored in memory and the other 255 rows/columns can be derived by properly shifting the first



By exploiting this structure, the entire W matrix can be stored with $8 \cdot 16 \cdot 256 = 32{,}768$ (2^{15}) bits, instead of the brute force $4096 \cdot 2048 = 8{,}388{,}608$ (2^{23}) bits.

Encoder implementation # 2

Here is one way to implement the encoder. This might be the one you thought of first (which is why I'm explaining it first) but it is not the one recommended by JPL (which is why I call it #2).

This implementation requires all 4096 message bits before it starts, and it computes the 2048 parity bits one at a time

The first parity bit is

$$p_0 = \underline{u} \cdot \begin{bmatrix} \text{first column of } W \end{bmatrix}$$

$\underline{u}$ is 1×4096, the column is 4096×1, so the result is a scalar (1×1). This computation requires 4096 AND operations, and 4095 XOR operations

The second parity bit is

$$p_1 = \underline{u} \cdot \begin{bmatrix} \text{second column of } W \end{bmatrix}$$

If the 1^{st} column of W was pulled out of ROM and loaded into some registers, clearly the 2^{nd} column (all the way up to the 256^{th} column) can be obtained by properly shifting the bank of 4096 registers

... Continue computing parity check bits. Once every 256 steps we need to load in a fresh column of W into the bank of 4096 registers, the next 255 steps we do the shifting. The output is

$$\underline{c} = [\underline{u} \,\vdots\, \underline{p}]$$

Encoder Implementation #1

Here is a non-obvious (which is why I present it second) implementation of the encoder that is recommended by JPL (which is why I call it #1)

This implementation processes the message bits one at a time and arrives at the final value of the parity word (vector) all at once.

Consider the computation

$\underline{TEMP} = u_0 \cdot$ [first row of W] , this is a scalar-vector multiplication that requires 2048 AND operations. The result (the vector $\underline{TEMP}$) is 1×2048. Next, consider the computation

$\underline{TEMP} = \underline{TEMP} \oplus u_1 \cdot$ [second row of W]

This requires 2048 AND operations followed by 2048 XOR operations. As we already know, the 2^{nd} row of W (all the way up to the 256^{th} row) can be obtaind by properly shifting the bank of 2048 registers that were used to store the 1^{st} row after it was loaded from ROM

... Continue with this recursive computation of $\underline{TEMP}$. Once every 256 steps we need to load in a fresh row of W into the bank of 2048 registers, the next 255 steps we do the shifting. After a total of 4096 steps we have $\underline{p} = TEMP$ and $\underline{c} = [\underline{u} \vdots \underline{p}]$

Block Diagram of Implementation #1

(this is taken from the CCSDS document published in September 2007, commonly refered to as "the Orange Book" with the official title "Low Density Parity Check Codes for Use in Near-Earth and Deep Space Applications")