ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

FPGA Implementation of encoders for CCSDS Low-Density Parity-Check (LDPC) codes.

Δημήτριος Κ. Θεοδωρόπουλος

Επιβλέπων: **Αντώνιος Πασχάλης,** Καθηγητής

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2015

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

FPGA Implementation of encoders for CCSDS Low-Density Parity-Check (LDPC) codes

Δημήτριος Κ. Θεοδωρόπουλος
Α.Μ.: Μ1321

Σεπτέμβριος 2015

ΠΕΡΙΛΗΨΗ

Η παρούσα διπλωματική εργασία παρουσιάζει την υλοποίηση με τεχνολογία FPGA αλγορίθμων κωδικοποίησης καναλιού που έχουν προτυποποιηθεί από τον οργανισμό CCSDS για χρήση σε διαστημικές επικοινωνίες.

Ο CCSDS προτείνει δύο κατηγορίες κωδίκων για εφαρμογές τηλεμετρίας: μία για επικοινωνίες στο εγγύς (near-earth) διάστημα (π.χ. δορυφορικές επικοινωνίες) και άλλη μια για επικοινωνίες βαθέος διαστήματος (deep-space), με χαρακτηριστικά η κάθε μία βελτιστοποιημένα ως προς το πεδίο εφαρμογής τους. Και στις δύο περιπτώσεις, οι κώδικες είναι γραμμικοί μπλοκ κώδικες με μεγάλο μέγεθος μπλοκ και πίνακα ισοτιμίας με χαμηλή πυκνότητα (LDPC).

Στην περίπτωση των κωδίκων near-erth, η προδιαγραφή αφορά σε ένα κώδικα LDPC (8160,7136) με ρυθμό 7/8, βασισμένο σε ευκλείδεια γεωμετρία, ενώ για τους κώδικες deep-space προδιαγράφονται 9 κώδικες που προκύπτουν από 3 συνδυασμούς μεγέθους μπλοκ (1024,4096, 16384 bits) με 3 ρυθμούς (½, 2/3, 4/5). Οι κώδικες αυτοί μοιράζονται κοινή μαθηματική περιγραφή, γεγονός που καθιστά εφικτή την περιγραφή με τη γλώσσα VHDL ενός κοινού κωδικοποιητή για όλους.

Στην παρούσα εργασία, γίνεται εκμετάλλευση της δομής των πινάκων-γεννητόρων των κωδίκων deep-space προκειμένου να μεγιστοποιηθεί η απόδοση. Προκύπτουν δύο ειδών παραλληλίες στη δομή των εν λόγω πινάκων, η ταυτόχρονη αξιοποίηση των οποίων οδηγεί σε βελτίωση των επιδόσεων με ελαχιστοποίηση των καταναλισκόμενων πόρων. Το τίμημα βέβαια της βελτιστοποίησης αυτής είναι κάποια αύξηση στην απόκριση (latency) ανάλογα με τις επιλογές παραλληλίας,που ωστόσο αντιμετωπίζεται με την λειτουργία του διαύλου της διεπαφής εξόδου με διοχέτευση (pipelining) Η περιγραφή στη γλώσσα VHDL είναι γενική και επιτρέπει την εύκολη παραμετροποίηση των βασικών χαρακτηριστικών του κώδικα (μέγεθος μπλοκ, ρυθμός), των βαθμών παραλληλίας για κάθε μια από τις δύο κατηγορίες και του εύρους των διαύλων εισόδου-εξόδου.

Αντίστοιχα στην περίπτωση του κώδικα near-earth, περιγράφεται μια αποδοτική μέθοδος στη σχεδίαση των επί μέρους οντοτήτων του κυκλώματος που βελτιστοποιεί την αξιοποίηση των πόρων, σε σχέση με γνωστές λύσεις. Ο κωδικοποιητής σε αυτή την περίπτωση είναι σχεδιασμένος για διαύλους εισόδου-εξόδου μεγέθους 16 bit.

Και στις δύο περιπτώσεις η είσοδος και έξοδος δεδομένων γίνεται από δύο αντίστοιχες διεπαφές συμβατές με το πρωτόκολλο AMBA AXI4-Stream, γεγονός που επιτρέπει την εύκολη διασύνδεσή τους σε μια σχεδίαση SoC ή μια διεπαφή FIFO. Η λειτουργία των κωδικοποιητών είναι βέλτιστη από την άποψη ότι παράγουν μια (σχεδόν) αδιάκοπη ροή δεδομένων στη διεπαφή εξόδου-χωρίς να είναι απαραίτητοι αδρανείς κύκλοι.

Η περιγραφή των κωδικοποιητών σε VHDL επαληθεύεται ως προς την ορθή της σχεδίαση με προσομοιώσεις για όλες τις υποστηριζόμενες περιπτώσεις, όπου απαιτείται η μέγιστη κάλυψη κώδικα (code coverage). Τέλος, το σχέδιο επαλήθευσης περιλαμβάνει την επίδειξη λειτουργίας σε ένα ενσωματωμένο σύστημα υλοποιημένο στην κάρτα XUPV505-LX110T, όπου καταγράφονται και οι πραγματικές επιδόσεις του συστήματος, όπου βρίσκονται στην περιοχή των μερικών Gbps. Η παρούσα υλοποίηση προκύπτει ότι είναι η ταχύτερη για την συγκεκριμένη οικογένεια LDPC κωδικών.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ψηφιακή Σχεδίαση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: LDPC, CCSDS, FPGA, near-earth, deep-space

ABSTRACT

The FPGA implementation of LDPC encoders for channel codes standardized by CCSDS for space communication applications is described in this work.

CCSDS suggests two classes of channel codes for telemetry applications: one for near-earth and another for deep-space communications, each one optimized for the demands of the specific field. In both cases, the specification concerns linear block codes with large block size and sparse generator matrices.

Regarding near-earth codes, the specification describes a Euclidean geometry based (8160,7136) LDPC code at rate 7/8, while in the deep-space case, 9 codes are defined which are the combination of thee block lengths (1024,4096,16384 bits) with three rates (½, 2/3, 4/5), sharing a common mathematical description. This fact enables the VHDL description of a common encoder for all of them.

The generator matrices of these codes possess considerable structure which facilitates implementation. Concerning deep-space codes generator matrices, parallelism extends over two dimensions, which can be exploited concurrently to optimize timing performance and at the same time minimize resource utilization. The price to be paid however is increased latency, which can be mitigated by the pipelined operation of the output interface. VHDL description of the encoder is generic, allowing the easy modification of the code parameters (block size, rate), the amount of parallelism in each dimension and the input-output bus width, leading to different performance-latency balances.

Also in the case of the near-earth code, an efficient design of the encoder's sub-entities is described, leading to resources utilization optimizations, compared to existing implementations. The encoder in this case is designed for 16-bit input-output bus.

All described encoders input-output is performed on AMBA AXI-4 Stream compliant interfaces, facilitating their integration in an embedded system's design and communication with standard FIFO interfaces. The encoders' operation is optimal in that an uninterrupted flow of data is provided on the output interface, without idle cycles. The only exception is the near-earth encoder for which just one idle cycle every 513 is inserted.

The correctness of the VHDL description's is validated by functional simulation for all supported cases, where 100% code coverage is demanded. The verification plan includes also the demonstration of real-time operation of the encoders in an integrated system implemented on a XUPV505-LX110T development board, where the actual performance of the encoders is recorded and lies in the multi-Gbps range. Finally, the proposed encoders are shown to be the fastest stream-oriented implementations for the specified family of LDPC codes, with minimal resource utilization.

SUBJECT AREA: Digital Design

KEYWORDS: LDPC, CCSDS, FPGA, near-earth, deep-space

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervising professor, Dr. A. Paschalis for the advice and patience exerted and also to his associates, A. Tsigkanos, N. Kranitis and G. Theodorou for their invaluable advice and assistance.

ΠΕΡΙΕΧΟΜΕΝΑ

FIGURES INDEX

TABLES INDEX

PREFACE

The work presented in this thesis was done in partial fulfillment of the requirements for the post-graduate program of Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens and was supported by the Digital Systems & Computer Architecture Laboratory (DSCAL). Apart form its other areas of activity and research, DSCAL exhibits interest in the development of applications for space systems, including its active involvement is ESA's PROBA-3 space mission.

This work attempts to provide an efficient implementation of communication channel codes already standardized for use in space communications by CCSDS, a multi-national forum for the development of communications and data systems standards for spaceflight, based on the expertise of communication experts from its participant nations. Implementations are already parts of launched space missions (Cibola Flight Experiment, MSL-MRO proximity link etc) and are expected to culminate in space communications in future missions, while at the same time gathering attention from disparate application fields, like mobile terrestrial communications (U.S.A.F. LCOT program).

For the development of the encoders for this work, a Xilinx XUPV505-LX110T development board was used, granted by DSCAL. Implementation was performed on Xilinx ISE design suite using VHDL, although considerable effort was made so that the code can be ported to other FPGA vendors without any modifications: it has been verified to be synthesizable also in Altera Quartus software and Microsemi's Libero suite. Simulations were executed in Mentor Graphics Modelsim .

1. INTRODUCTION TO LDPC CODES

The purpose of this chapter is to provide briefly the theoretical background necessary for understanding the encoder implementation. Relevant information theory topics are described to the minimum extent required for the adequate description of the application and by no means intended to explain theoretical topics from a mathematically or information theory concrete point of view.

1.1 Noisy Channel Coding Introduction

Noise is an inherent element of every communication system. A simplified version of one such system is displayed in figure 1. Noise in space communications channels (not accounting for weather effects at least), is modeled in almost perfect approximation by the Additive White Gaussian Noise (AWGN) model and the channel most commonly considered is the Binary Symmetric Channel (BSC) for a digital communications system. In this model, noise can be represented as a binary vector n , added to the binary sequence t transmitted on the channel, resulting in the received vector $\hat{t}$. The purpose of the encoding process is to receive a binary sequence s and transform it into another binary sequence t of greater length, which should depict the necessary features to mitigate the result of the addition of the noise vector n . Considering AGWN over a BSC, this noise vector n contains a '1' in each bit position corresponding to a flipped bit of the finally received sequence, and this occurs with a constant probability value “f”.The result of this encoding process is that a corresponding decoder at the other end of the communication channel is able to provide an estimate $\hat{s}$ of the initially transmitted vector s , which is as close to it as possible. This maximum probability of correct inference will be refined later in this chapter.

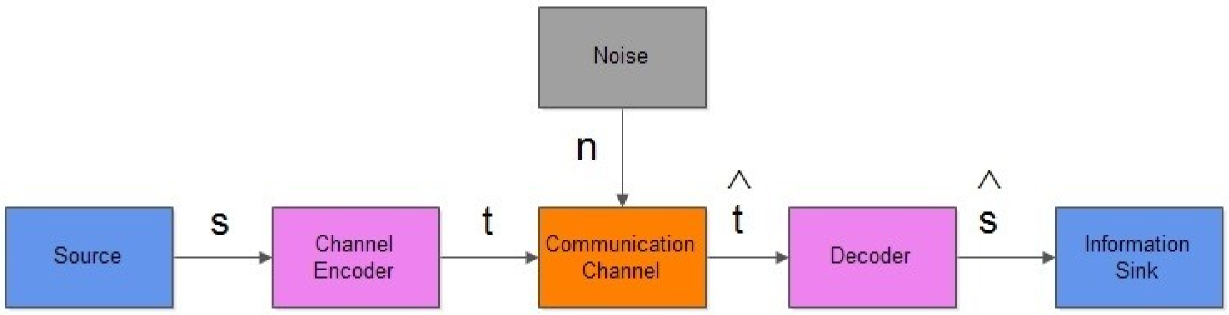


Figure 1: A simplified binary noisy communication system

Always considering AWGN over BSC, the probability of a bit flipping as a result of the noise vector contribution to the received codeword (i.e. $P(n_i=1)=f$) is expressed by a single metric for an individual bit, namely the bit error rate (BER), or equivalently the error rate for a whole frame (Frame Error Rate, FER). For a sequence of N independent bits, generally $FER=1-(1-BER)^N$, although for more interesting codes as our case the frame contains an error if $s_i\neq \hat{s}_i$ instead of $t_i\neq \hat{t}_i$. The value of this probability for a single bit ($P(n_i=1)=f$) is a channel parameter, which for the communication model described herein is related to the received signal energy per information bit to the (one sided) spectral density of the white Gaussian noise, commonly referred as E_b/N_0 or Signal to Noise Ratio (SNR).

The encoded sequence t contains redundant information, decreasing thus the rate at which actual communication occurs. In a trivial encoding example, replication codes transmit multiple sequences of the same source bits and decoding is performed on a majority basis of the value of the received sequence, at the same time though by dividing proportionally the actual information data rate. More interesting codes exploit redundancy more efficiently but generally there seems to be a trade-off between the (decoded) bit error probability and the communication rate.

Before 1948, it was believed that a vanishingly small BER for given channel characteristics (i.e. constant E_b/N_0 value) requires proportionally decreasing rate. All this changed by Claude E. Shannon in his Phd thesis [1], in which the fundamental limits on the performance of all codes (for a given rate) were set.

In particular, Shannon associated with each channel a quantity called capacity C, up to which reliable communication can occur with arbitrarily small BER. This quantity is a channel feature and an equivalent interpretation in the AWGN channel is that for a specific rate, there is a minimum E_b/N_0 for which communication can occur error-free. Figure 2 depicts this relationship for the binary input AWGN channel for a number of communication rates.

Shannon's calculations assume an asymptotically infinite code-block length. In practical applications though this is obviously not feasible and the theoretical capacity limit is lower than that of fig. 2. The effect of block size on code performance is studied further in [3].

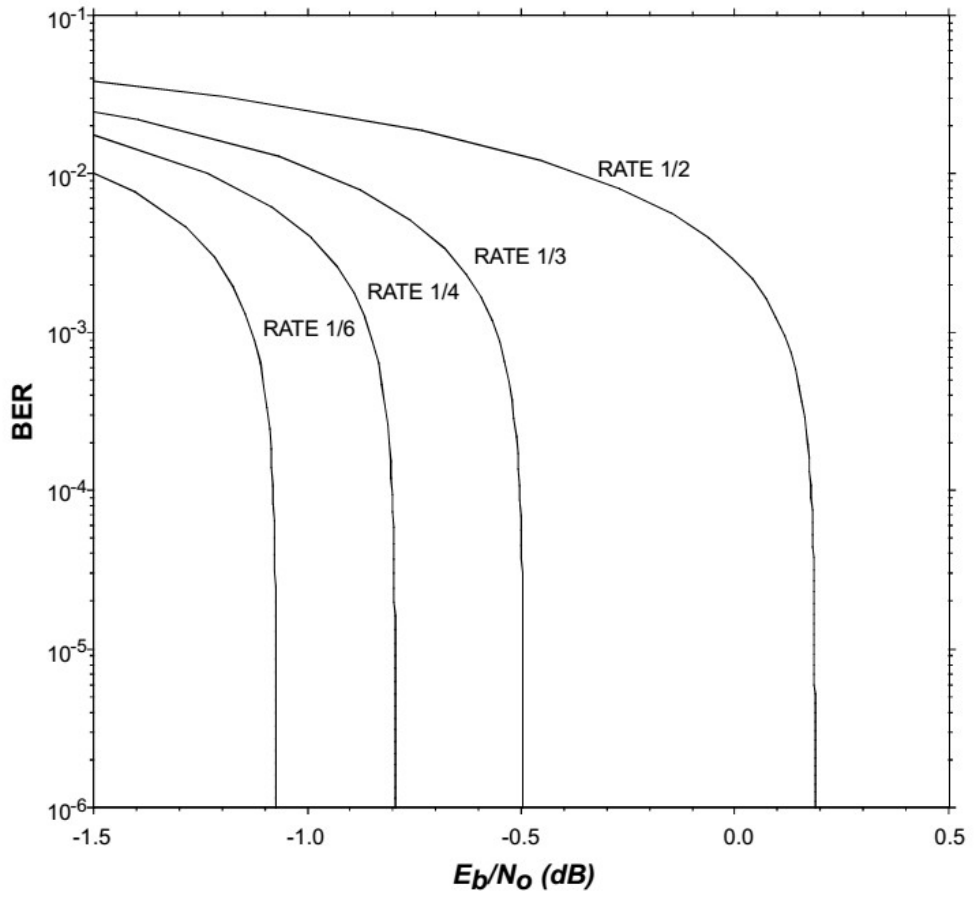


Figure 2: Capacity limits for the AWGN channel over a selection of data rates (image from [2])

The above results only prove the feasibility of such codes. The design of the actual codes themselves is nevertheless a different issue, out of Shannon's work scope but a very interesting field of considerable research towards capacity approaching codes. The channel codes implemented in this thesis belong to one such class.

1.2 Low Density Parity Check codes

Channel codes can be divided into two major types depending on the grouping of input information in constant size packets (block codes) and the encoding of a continuous stream of data (convolution codes). The codes examined herein belong to the first class. Block codes are further divided into linear and non-linear ones, the latter having never been used in practice. Consequently, we are interested in linear block codes.

1.2.1 Linear Block Codes

For a block code, information source bits are grouped into blocks of k bits. The encoding process transforms these into a n -bit codeword, where $n>k$, adding thus $n-k$ bits of redundant information. For the code to be considered linear, the ensemble of 2^k possible codewords should form a k-dimensional subspace in the vector space F_2^n . According to this definition, if $s\in F_2^k$ and $t\in F_2^n$ are row vectors corresponding to the information block and encoded codeword respectively, there exists a $k*n$ binary matrix, the rows of which are k linearly independent codewords and $t\in F_2^n$ is a linear combination of them. Consequently, the encoding process can be described as the operation $t\in F_2^n$ (performed in GF_2). The matrix G is known as the generator matrix and the code itself as a (n,k) linear block code.

The code can be alternatively described by the null space of a different binary matrix H, such that for every valid codeword $t\in F_2^n$: $tH^T=0$ (zero vector). The H matrix dimensions are $(n-k)*n$, assuming full rank (rank deficient matrices are also possible as it will be the case for one of the CCSDS codes of interest) and $GH^T=0$. The H matrix is called the parity check matrix and because it is the null space of the code, it can be perceived as the expression of the constraints an arbitrary binary sequence should satisfied in order to be considered as a valid codeword.

Codes in which the first k bits of the codewords are the uncoded source information bits are called systematic. The codes implemented in this thesis are all systematic. This feature facilitates decoding and other optimizations in the receiver.

1.2.2 LDPC description

LDPC codes where introduced in 1960 by Gallager [4], but generally ignored in the following years due to the current era's technology limitations, which could not allow their implementation at a reasonable cost.

These codes are generally characterized by a sparse parity check matrix H, i.e. a matrix with a very low density of “1s”. An absolute definition of sparsity in not defined in literature, but densities up to 1% qualify for the characterization [5]. This sparsity of H matrix coefficients is a key feature for reduced complexity implementation.

One exception to the oblivion in which LDPC codes succumbed after their invention was the work of Tanner in 1981 [6]. Among other things, he introduced a graphical representation of these codes in what are currently widely known as Tanner graphs. The Tanner graph is a bipartite graph in which node fall into two categories: check nodes and variable nodes, the former expressing the constraints on codewords and the latter the received encoded bits. A connection between a variable node v_i and a check node c_j is drawn if the corresponding element h_{ij} in the parity check matrix is

1. The representation is valid for all linear block codes and a Tanner graph for the well known (7,4) Hamming code is displayed in fig. 3. For LDPC codes this representation facilitates the description of the decoder based on a message passing algorithm between the nodes of the Tanner graph [7].

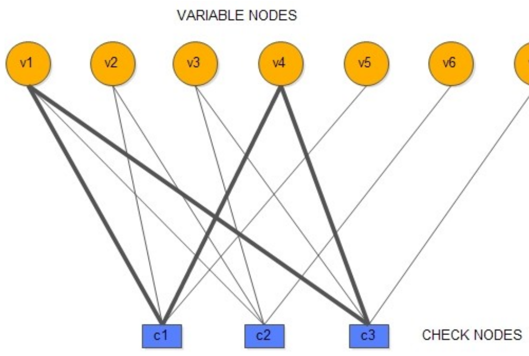


Figure 3: Tanner graph for the (7,4) Hamming code. Bold lines mark a cycle.

1.2.3 LDPC features

An LDPC code is described as **regular** when its parity check matrix has a constant column weight, say γ and constant row weight ρ. Such a code is said to be (γ,ρ) regular. In contrast, **irregular** codes have multiple weights. The CCDS codes are all regular.

The parity check matrices of initial Gallager codes possessed no other structure except being linear block codes. The problem is that implementation complexity makes their application prohibitive. A desirable structure to facilitate implementation is that of **cyclic** codes: each row of the parity matrix H is a cyclic shift of the previous one. Since each check equation is related to the previous in a very specific way, encoder complexity is substantial: it is built by simple elements around a shift register. More interesting codes though are built using a viable compromise between complexity and performance, using a **quasi-cyclic (QC) structure**. The parity check matrix of these codes consists of an array of juxtaposed cyclic submatrices called **circulants**. The general form of such a matrix is the following:

$$H=\begin{bmatrix} A_{11} & A_{12} & . & . & .A_{1N} \\ A_{21} & A_{22} & . & . & .A_{2N} \\ . & . & . & . & . \\ A_{M1} & A_{M2} & . & . & .A_{MN} \end{bmatrix}$$

Each sub-matrix A_{ij} is a cyclic matrix with a very low density of ones. The implemented codes in this thesis belong to this class of LDPC codes.

The design of the code for this class of LDPC codes is consequently reduced to the task of defining the optimum position of 1s in the parity check matrix. Several techniques and mathematical tools are employed and considerable research is always hot on these topics. Generally, design techniques are classified in two big categories: a) **random** or **pseudo-random** codes, which use computer-based algorithms or methods and b) **algebraic** codes which use mathematical or combinatorial tools such as finite geometries and combinatorial designs. The codes implemented here belong to both categories. In particular, AR4JA codes were generated by a pseudo-random algorithm based on a design entity known as a **protograph.** This is simply a Tanner graph with a low number of nodes, which is repeated (consider placing a number of such graphs side-by side). Connection lines between variable and check nodes of the expanded super-graph are permuted in a pseudo-random manner. The number of repetitions and the permutations pattern is a result of advanced techniques (density evolution, progressive edge growth). An introduction to this topic is included in [8].

A common feature shared among the most interesting LDPC codes is the Row-Column **(RC) constraint**: no two rows or two columns are allowed to have a '1' in more than one position at the same time: for example codewords “01001101...” and “11101010..” belong to a non-conformant code because there is a '1' at positions 2 and 5 (at least). The presence of this constraint ensures that the minimum distance of a (γ,ρ) irregular code is at least γ+1. Moreover, it precludes cycles of length 4 in the Tanner graph of the code. Cycles in a graph are structures in the form of a path in a graph from one node back to itself. One such path in displayed with bold lines in fig. 4 for the Hamming code. These structures jeopardize the code performance, as it will be mentioned in next paragraph.

1.2.4 LDPC performance

LDPC codes are the most promising solution towards capacity approaching performance. The most obvious performance metric is the BER or FER performance, or in other words, how close to the Shannon capacity limit these codes can approach. In fig. 4, simulation results for the AR4JA rate 3/4, block length 1024 bits LDPC code are presented. Decoding is performed in a software (MATLAB) implementation of the iterative Sum of Products Algorithm (SPA). For the number of simulation iterations specified in this test, BER was zero for E_b/N_0 grater than roughly 2,5 dB.

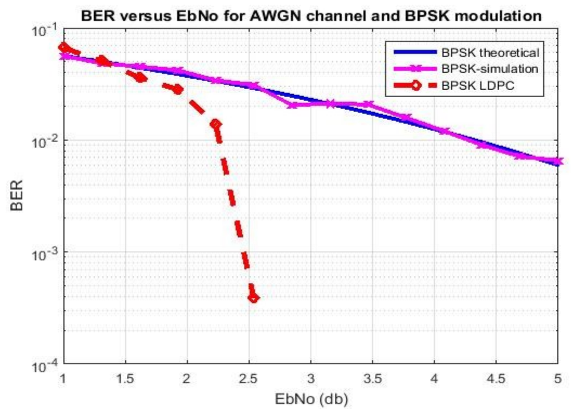


Figure 4: BER performance for the rate 4/5 k=1024 CCSDS LDPC code

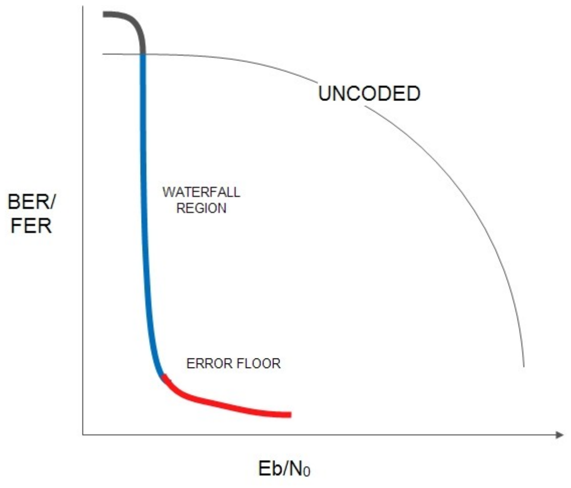


Figure 5: General form of the BER curve for most LDPC code

The BER curve of a code is generally partitioned into three regions displayed on fig. 5: the non-performing region (black), the waterfall region (blue) and the floor region (red). For extremely low SNR values, there is no point to introduce a channel code since the errors are so many that the decoder will try to converge most probably towards another completely different codeword (with smaller distance to the erroneously received). The code is non-operational in this region. Above a certain limit of E_b/N_0 value called the **threshold**, the **waterfall region** begins: this is recognized as an abrupt slope int the plot and it is the area where the code performs optimally. There is however a point where this abrupt transition halts or even the curve remains constant. This is known as the **error floor** and it is the point where the code becomes inefficient. This weakness is a common feature of all LDPC codes and it is caused mainly by undesirable structures in the Tanner graph of the code such as **trapping sets** or **stopping sets**. For reasons that remain unknown, there seems to be a trade-off between low threshold and low error floor [9].

Decoder complexity is another performance parameter for LDPC codes. Generally, decoding is performed by an iterative belief propagation algorithm in which variable nodes and check nodes exchange messages conveying likelihood information. The number of steps required for decoding may limit the actual data rate of the communication and this is especially important for high data rates.

Generally, the code performance is a still unknown function of a number of code parameters and structures in the Tanner graph and the LDPC codes exhibit a wide diversity of characteristics. The area is open for research. An introduction to performance considerations can be found in [9].

1.3 Encoder architectures for Quasi-Cyclic codes.

Encoding process of a linear block code is in essence nothing more than a matrix multiplication over GF_2 . The encoded vector is $t = s * G$, where G is the generator matrix and s the input vector.

The circulant structure of the parity check matrix can be exploited to facilitate the encoding process. With suitable transformations, it is possible to calculate the Generator matrix in systematic circulant form and limit the encoder's complexity to being just linear with the block length [10]. The circulants though of the resulting generator matrix from this process are dense circulant matrices. Figure 6 displays the shape of such a systematic circulant matrix for one of the codes of interest. A '1' in this image is represented by a white pixel, whereas a '0' is black.

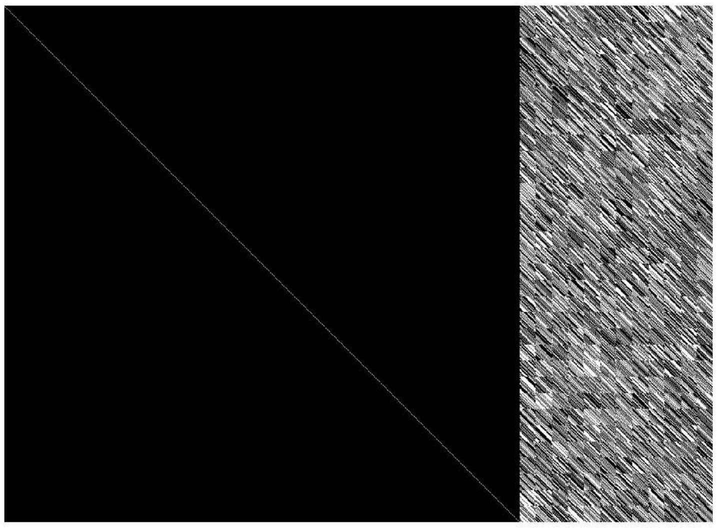


Figure 6: Generator matrix of AR4JA LDPC code with k=1024, rate 4/5 (punctured bits included)

The systematic output of the encoder with a generator matrix in this form is the direct output of the input bits. The non-systematic part of the output can be implemented by simple shift registers, as shown below.

1.3.1 Straightforward implementation

A straightforward solution for the multiplication of a row vector with a systematic circulant matrix is displayed on fig. 7. Codeword length is n, input block size is k and circulant size is m

Initially the last n-k bits of the Generator matrix are stored in the cyclic shift registers at the top of the image. These bits correspond to the first rows of the first row of circulants in the generator matrix. The first information bit to arrive is ANDed with this vector, and the resulting vector is XORed with the current value of the accumulator at the lower end of the image. The accumulator stores the result of this XOR operation. Then the shift registers containing the circulants of the generator matrix are cyclically shifted one position to construct the second row of the generator matrix; the result is multiplied by the next message bit and added to the accumulator. This process is repeated *m* times to complete the first row of circulants in the generator matrix.

After the AND-XOR operation corresponding to the last row of circulants is completed, the shift registers do not perform a shift operation but a load instead: the next row is loaded and the above steps are repeated until all information bits have arrived at the encoder. During all these steps, arriving input bits form the systematic part of the output.

This straightforward implementation requires *2(n-k)* storage elements (flip-flops) and *k(n-k)* AND-XOR operations.

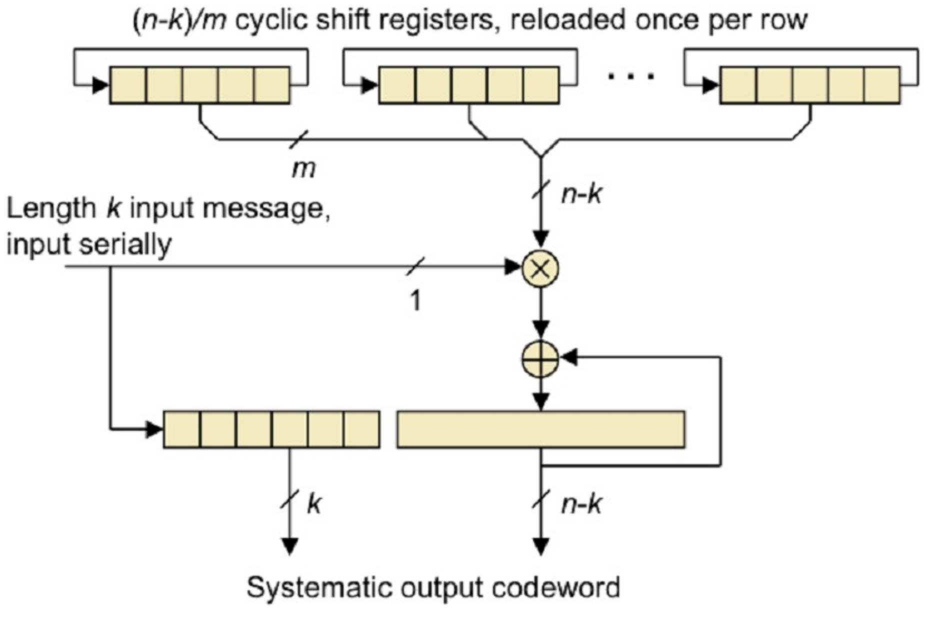

(n-k)/m cyclic shift registers, reloaded once per row
m
n-k
Length k input message, input serially
1
k
n-k
Systematic output codeword

Figure 7: A straightforward encoder implementation for a QC LDPC code

1.3.2 RCE implementation

A more efficient approach is given in [11], henceforth mentioned as RCE encoder. The encoder's architecture is displayed in Figure 8 and it makes use of the ideas applied to the design of convolutional codes, namely the encoder structure of a Recursive Convolutional Encoder (RCE).

The idea behind this implementation is to keep the circulants values stationary and cyclically shift the accumulator bits instead. The generator matrix values become combinational functions of an input message bits counter.

During the information bits input, the (systematic output of the encoder is these input bits: the selectors in the RCEs are set to perform the cyclic shift and the output selector is set to the upper position to select the input. Upon completion of the calculation the selectors positions are switched and the calculated parity bits are simply shifted out of the RCEs.

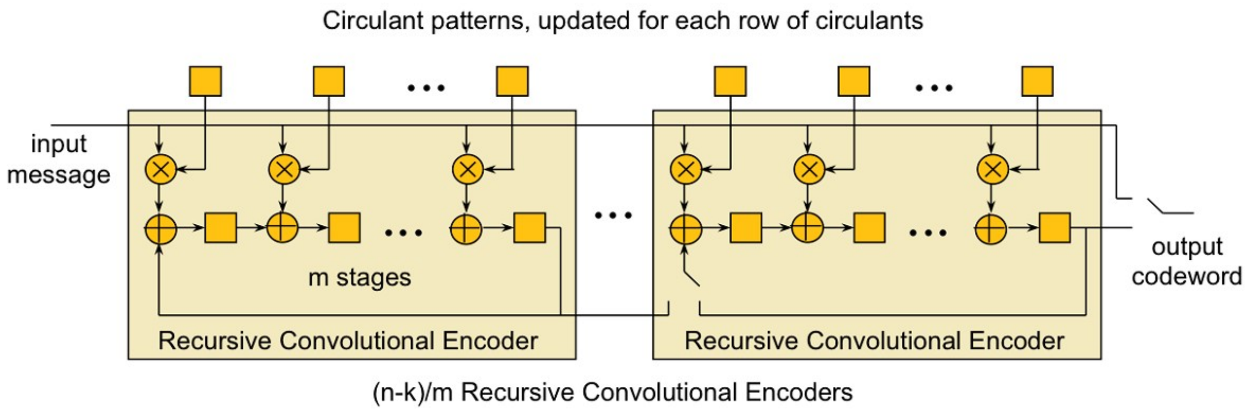

Circulant patterns, updated for each row of circulants
input message
m stages
Recursive Convolutional Encoder
Recursive Convolutional Encoder
output codeword
(n-k)/m Recursive Convolutional Encoders

Figure 8: RCE encoder

1.3.3 RU encoder

In 2001 Thomas Richardson and Rüdiger Urbanke demonstrated a reduced complexity encoder for LDPC codes [12]. As a first step, the parity check matrix is rearranged into in an approximately lower triangular form through reordering of rows and columns. The resulting matrix has the general shape of fig. 9.

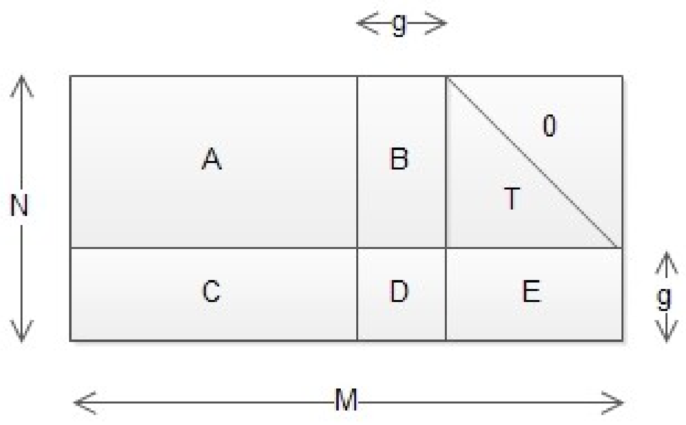
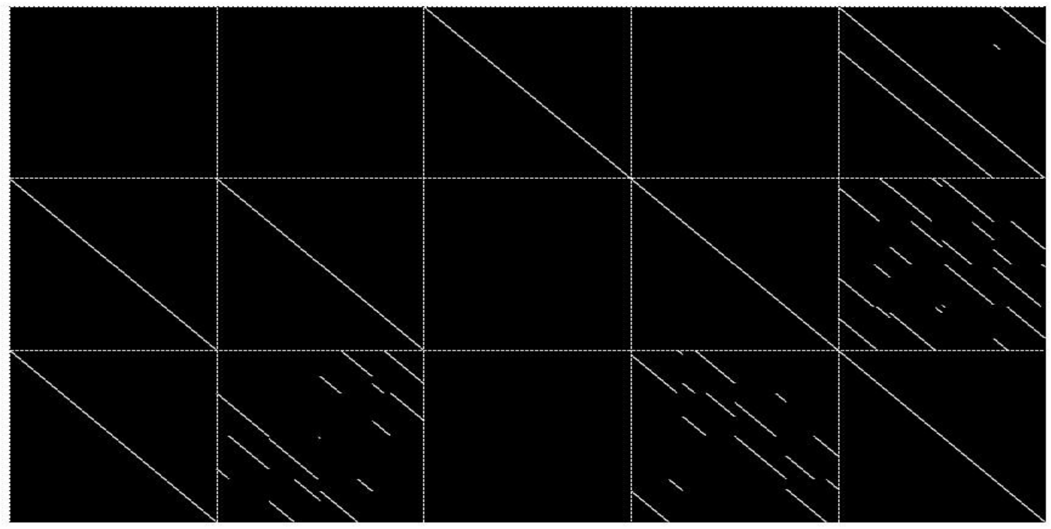


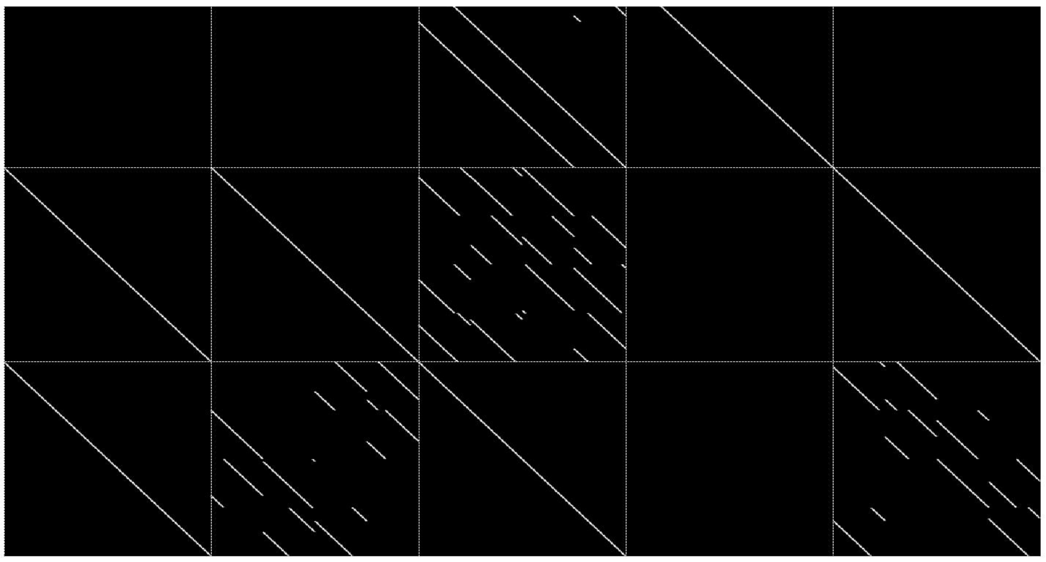
Figure 9: The parity check matrix transformed into lower triangular form

Since the original matrix is sparse, the sub-matrices A, B C, D are also sparse. The elements of matrix T are all zero above a certain diagonal. Without describing the mathematical details of the work in [12], we use only their result, keeping the notation though consistent. For a systematic code, the k=M-N input vector s is encoded as a systematic codeword $t=[s\,p_1\,p_2]$, where parity bits are partitioned in two sub-vectors p_1 p_2 . The steps are outlined below.

Firstly, calculate $\varphi=ET^{-1}B+D$, which is a dense g x g matrix. The first parity bit vector is $p_1^T=\varphi^{-1}(ET^{-1}A+C)s^T$ and the second is $p_2^T=T^{-1}(As^T+B\,p_1^T)$. This calculation involves several sparse matrices and for several interesting codes (including the CCSDS AR4JA examined here), T is the identity matrix. As a result, the above equations are further simplified into the following: $\varphi = EB + D$, $p_1^T = \varphi^{-1}(EA+C)s^T$ and $p_2^T = As^T + Bp_1^T$. The dense matrix φ can be precomputed in advance, while other operations on sparse matrices can be calculated using simplified hardware.

As an example, the corresponding matrices for one of the codes implemented in this thesis are displayed in fig. 10.





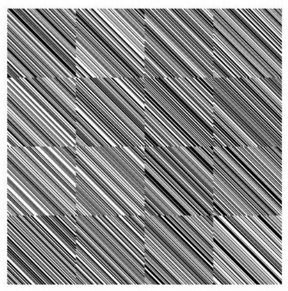


Figure 10: From top to bottom: The parity check matrix of rate ½ k=1024 CCSDS LDPC code, the same matrix transformed in lower triangular form and the inverse matrix φ^{-1} . Each rectangular is 512x512 bits. The φ^{-1} matrix is also 512x512, consisting of 64x64 submatrices.

The dense matrix operations are calculated using a usual encoder such as those described in previous chapters. For sparse matrix operations, a simplified architecture is displayed in fig. 11.

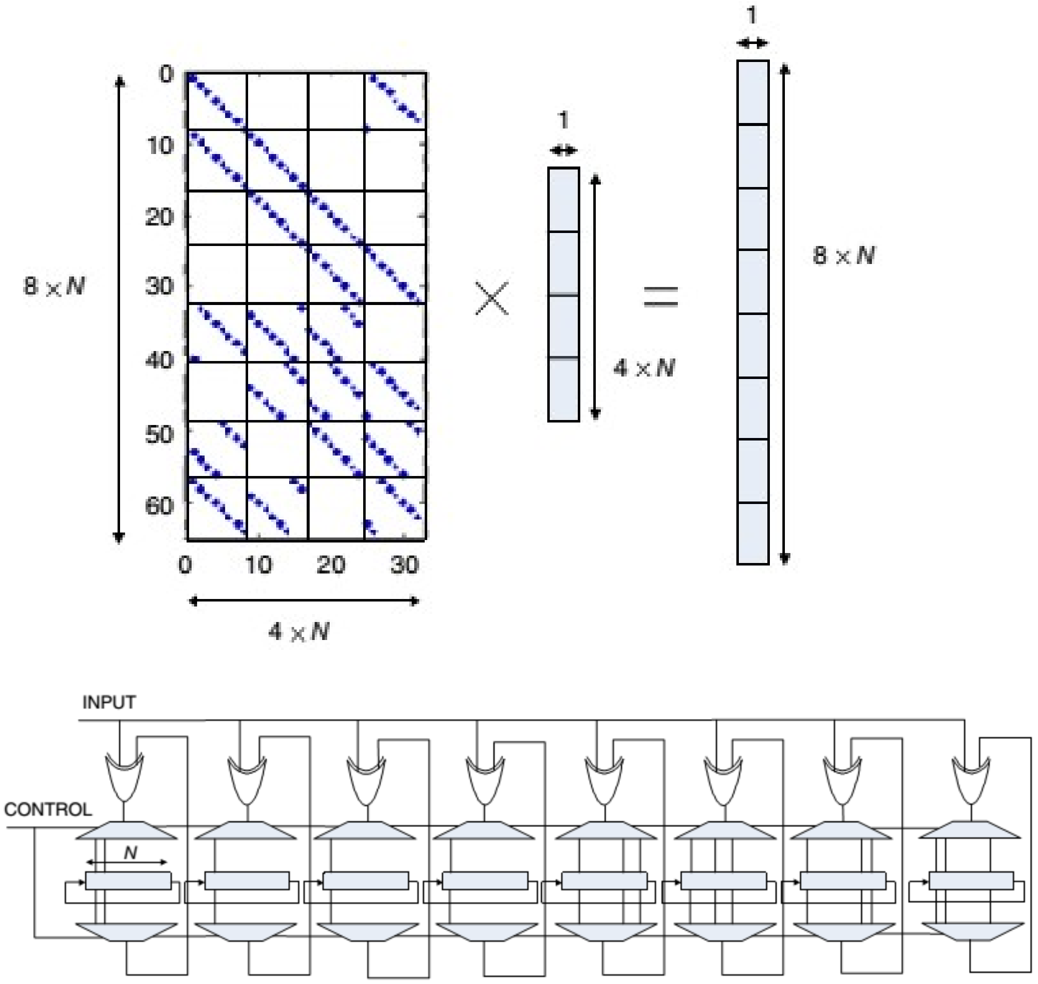


Figure 11: Multiplication of a sparse matrix with a vector and the corresponding hardware[13]

The sparse matrix is constructed from circulant submatrices of size *N*. The circuit on the figure performs a multiplication of a vector of size *4N* with a sparse matrix *4Nx8N*. As incoming information bits arrive, the multiplexors below the shift registers select which bit of the shift register is going to be subject to modification by the XOR gate. This is easy to understand since from the *N* bits of the circulant, only one is going to take part in the parity calculation at each step. A (hopefully) small memory controls the MUX operations. More XOR gates are needed in case the sparse matrix is not a rotated identity matrix like those in the given example.

The iterative encoder calculates parity bits directly from the parity-check matrix instead of the generator. This can lead to high performance parallel encoders, provided that the corresponding LDPC codes are amenable to the modifications described in Richardson-Urbanke work (very low value of g). This is the case with LDPC codes employed in DVB-T and 802.16ac standards for which encoders have been proposed in the multi-gigabits per second range speed [14] [15].

1.3.4 Iterative encoder

In the special case where g=0 (fig. 9), the parity check matrix is simplified into a lower-triangular form, the structure of which can be exploited in order to create especially optimized high-throughput encoders. Like in RU method, parity bits can be directly calculated from the parity-check matrix using back-substitution: Let c = [m p] be a codeword block, where m and p indicate the information bit sequence and the parity bit sequence, respectively and H=[H_1 H_2] the parity-check matrix partitioned into two sub-matrices H_1 and H_2 of suitable size to correspond to the multiplication operations detailed below. From the property that the correct codeword satisfies the parity check equation, the parity bit sequence p can be derived as follows:

$$H\cdot c^T = H_1\cdot m^T + H_2\cdot p^T \quad \Rightarrow \quad p^T = H_2^{-1} H_1 \cdot m^T$$

Matrix H_1 is sparse in all LDPC codes but this is not always the case for H_2^{-1} matrix, which is generally sparse. LDPC codes designed with encoding efficiency as a primary goal contain significant structure in these codes. Efficient encoders for the applicable codes can take advantage of these structures to maximize throughput while keeping resource utilization at a minimum.

Examples of such codes are LDPC codes for IEEE 802.11ac and DVB standards. In the former case, H_2^{-1} matrix consists of rotated identity submatrices, while in DVB-S2 it is an upper triangular matrix (fig. 12). In all cases, the sparse matrix operation $H_1 \cdot p^T$ between the sparse matrix H_1 and the vector *m* can be performed in a highly parallel way, which can even be performed in just one clock cycle. For 802.11ac, the last multiplication with H_2^{-1} can be performed in parallel with shift registers and back-forward accumulation [16] in just a few clock cycles (depending on the rotated identity matrix size in H_2^{-1}). In a similar way, the structure of H_2^{-1} in DVB-S2 corresponds to a trivial forward substitution operation [17]. Encoders have been proposed for such codes in [16], [17], [18], [19] with performance in the multi-Gbps range.

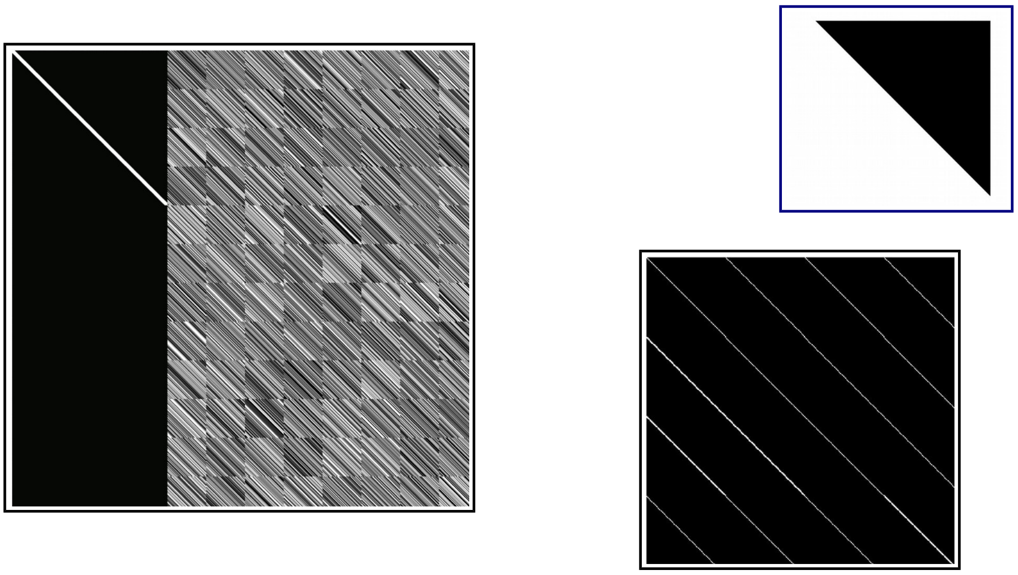


Figure 12: H_2^{-1} matrices for various LDPC codes: AR4JA code (left),DVB-S2 (right-top), IEEE802.11ac (right bottom). Left image identity matrix stressed for emphasis.

These highly parallel architectures however are not suitable for the codes considered in the current work. Considering fig. 12, this is obvious. Indeed, the matrix for AR4JA codes (rate ½, k=1024 in the case depicted) consists of a 512×512 identity matrix on the left and a 12×8 array of 128x128 dense circulants. This matrix is apparently more complicated than the generator matrix of the code and there is absolutely no benefit if this algorithm is followed. Instead, hardware requirements are expected to be even larger because of the calculation of the (unnecessary) omitted punctured parity bits. Similar assumptions hold also for other examples of LDPC codes whose parity-check matrix exhibits similar structure.

2. CCSDS STANDARDS

The Consultative Committee for Space Data Systems was found in 1982 by the major space agencies of the world and it is a multinational forum for the development of communication and data systems standards for spaceflights.

CCSDS protocols, collectively known as Space Communication Protocol Specifications (SCPS) are generally based on well-known Internet protocols, with the necessary modifications and extensions to cope with the specific space demands. For an introduction to CCSDS space communication protocols, reference [20] is a starting point.

CCSDS standards follow a color code according to which, “yellow” publications start as experimental and are finally standardized as “blue” books, which is the color of the recommended publications. Books colored “green” are information reports, generally providing the rationale behind the adoption of each standard or other information of general interest.

In this work, the focus is on the data link layer, in which four Space Data Link Protocols (SDLPs) have been developed:

- Telemetry SDLP (TM-SDLP) is used mainly by spacecraft systems for the emission of sensor data and systems readings.
- Telecommand SDLP (TC-SDLP) for commands from a ground station (or another spacecraft) to a spacecraft.
- Advanced Orbiting Systems SDLP (AOS-SDLP) is an extension to TM-SDLP for bidirectional exchange of on-line information like audio and video.
- Proximity-1 Space Link Protocol for short-range bidirectional links between fixed probes, landers, rovers, orbiting constellations and orbiting relays. Proximity-1 is an altogether different protocol stack from the previous one (SCPS) but obviously has a data link layer.

The data link layer's lowest functions are synchronization of upper layer Protocol Data Units (PDUs), called **Transfer Frames (TF)**, randomization and channel coding. These functions belong to a sublayer of data link layer called synchronization and channel coding sublayer. TM and AOS SDLPs share the same synchronization and channel coding sublayer specification.

The excellent performance characteristics of LDPC codes led CCSDS to adopt them in synchronization and channel coding sublayer of TM-SDLP and recently in Proximity-1 data link layer.

Two classes of LDPC codes were adopted for use in TM-SDLP: one class of codes optimized for deep-space applications (AR4JA) and another for near-earth (C2). Interestingly though, one particular code of the AR4JA family (k=1024 rate ½) was selected for Proximity-1 coding and synchronization sublayer.

In the first case, for deep space communications, good Eb/N_0 performance is more important than high data rates. Communication data rates are lower and the bandwidth expansion caused by lower channel code rates can be tolerated. Low error floors are an important parameter defining the SNR performance of the code and for the codes of this family, they achieve a fair combination of low threshold and low error-floor. The CCSDS standards define nine LDPC codes for this family, sharing a common mathematical description. The total number of nine codes is the result of the combination of three block length sizes of 1024, 4096 and 16384 bits over three code rates: 1/2, 2/3 and 4/5.

In the second case, for Near Earth communications, data is transmitted at hundredths of Mbps in 375Mhz restricted band. Higher data rates are important in this case, together with fast convergence of the (iterative) decoder. Error floor should be very low (<10^{-10} BER) also. For these reasons, CCSDS adopted a (8176,7156) LDPC code for these communications, henceforth described as C2 after [21].

The selected code rates are 1/2, 2/3, 4/5, and approximately 7/8, which are about uniformly spaced by 1 dB on the rate-dependent capacity curve for the binary-input AWGN channel [2]. Near rate 1/2, a one-percent improvement in bandwidth efficiency costs about 0.02 dB in power efficiency; near rate 7/8, a one-percent improvement in bandwidth efficiency costs 0.1 dB in power efficiency.

Within the AR4JA family, the selected block lengths (k=1024, 4096 and 16384) are about uniformly spaced by 0.6 dB on the sphere-packing bound at WER=10 -8. By choosing to keep k constant among family members, rather than n (codeword length), the spacecraft's command and data handling system can generate data frames without knowledge of the code rate. To simplify implementation, the code rates are exact ratios of small integers, and the choices of k are powers of two.

Since the LDPC codes implemented in this thesis are already a standard publicly available in [22], the description that follows is limited to the necessary features for the better understanding of the implementation section that follows. Further insight on the performance characteristics of the adopted codes is provided in [2].

Real-life implementations of the proposed standards exist and they are continuously growing. NASA adopted the AR4JA for the MSL(Curiosity) to MRO link. Proximity-1 LDPC code was also the choice for all links of the Constellation program. Code C2 (rate 7/8) was the choice for LDCM (Landsat 8) and NOAA's geostationary satellite GOES.

2.1 AR4JA LDPC code family

TF (information block) length for each of the 9 codes is given on Table 1.

An important feature of this family is that the codes are **punctured**, meaning that not all of the encoded bits are transmitted. Parity check matrices include additional linearly dependent rows.

Table 1: Codeword lengths for supported AR4JA codes (in bits

Transfer Frame length (k)	Codeword length (n)		
	1/2	2/3	4/5
1024	2048	1536	1280
4096	8192	6144	5120
16384	32768	24576	20480

The parity check matrix of this code is a juxtaposition of circulant sparse M×M submatrices. The value of the parameter M is given on Table 2.

Table 2: Submatrix size and K parameter for supported codes

Rate		1/2	2/3	4/5
		Submatrix size M		
Transfer Frame length (k)	**1024**	512	256	128
	4096	2048	1024	512
	16384	8192	4096	2048
K parameter		2	4	8

The positions of 1 in the parity check matrix are provided by the standard as a formula and they can be easily implemented in MATLAB. One pictorial example of a parity check matrix was displayed on fig. 10 for the k=1024 rate ½ member.

The generator matrix for each member of the family has the form $G=[I_{MK}\,W]$, where I_{MK} is the MKxMK identity matrix and W is a dense matrix of size MKx3M. Matrix W is calculated in systematic-circulant form, according to a methodology provided in [23]. The punctured bits can be omitted from the generator matrix during the encoding process and the matrix W can be simplified to MKx2.

The submatrix W is also an array of juxtaposed circulants. The parameter m describes the circulant size of the generator submatrix W and its value is for all members m=M/4. It follows that the submatrix W is consequently a 4Kx8 array of m×m circulants. Note that parameter K is related only to the code rate and is independent of block length! For better intuition into the structure of the generator matrix, fig. 13 displays the W submatrices for all k=1024 codes. Also, the code parameters defined so far are of high importance for the encoder's design. Table 3 summarizes them briefly. These parameters and images are important for the description of encoder operation.

Table 3: Summary of the most important parameters for AR4JA family

Parameter	Description
k	Transfer Frame block length
n	Codeword length
M	H matrix circulant size, depends on rate and k
m	G matrix circulant size, equals M/4
K	Describes G matrix vertical dimension as a function of m, equals k/M

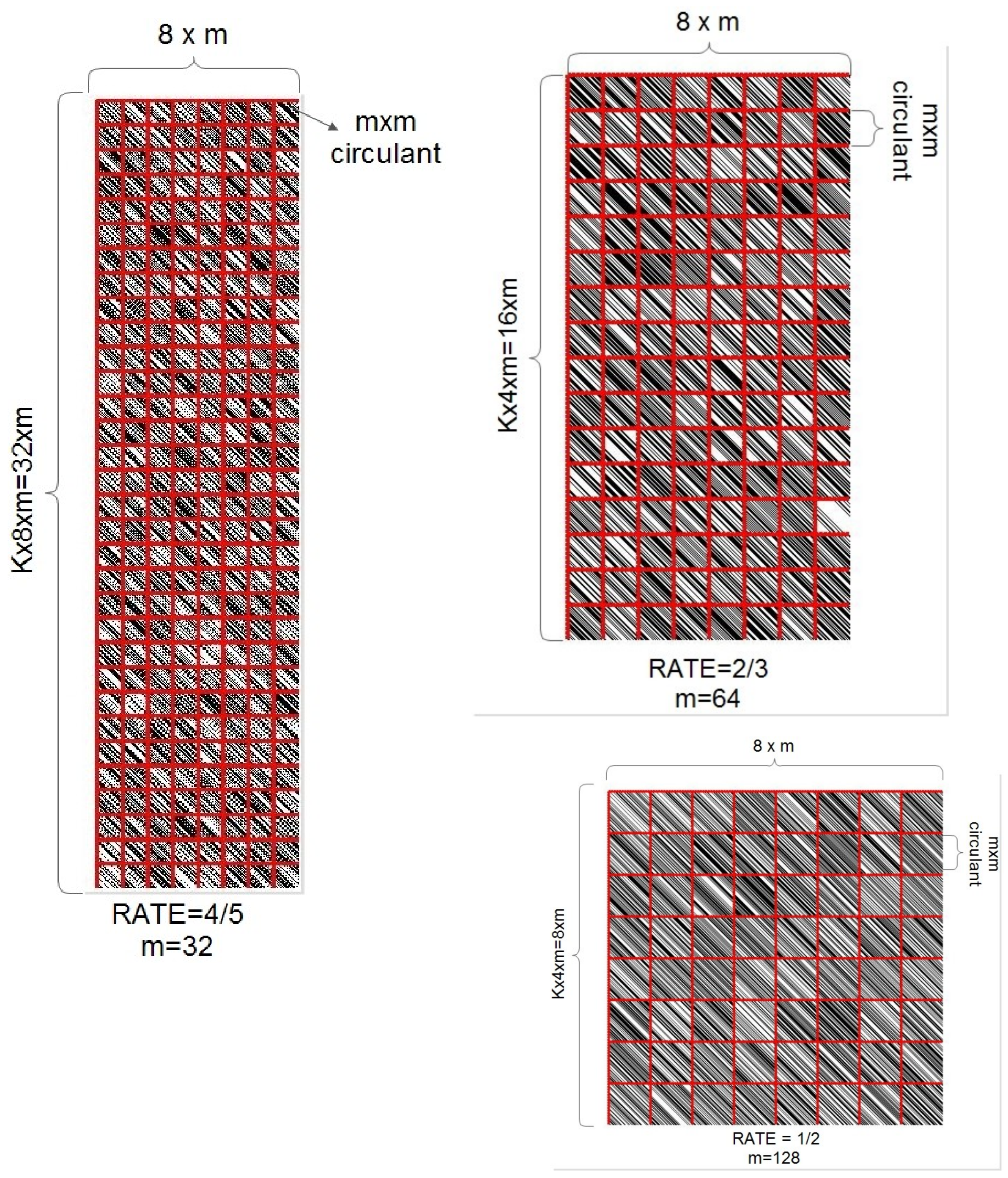


Figure 13: Submatrices W of generator matrices for codes of k=1024.

It is very important to note that for all members of the family, there are always 8 circulant columns. Also note that the number of circulant rows depends only on the code rate and not the block length (k). These notes will enable the development of a single parametric VHDL model for the encoder to cover all the members of the family.

2.2 C2 code for near-earth applications

According to the standard, (8160,7136) C2 LDPC code is an expurgated, shortened, and extended version of a basic (8176,7156) LDPC code, based on Euclidean geometry. The important features of the code needed for the design of a suitable encoder are the following:

- A TF of 7136 bits is provided for encoding, to which 18 zero bits are prepended. The reason for this is to ensure that incoming information block is divided by 8 and 16, which is the word length of many microprocessor buses.
- The 18 prepended zeros take part in the encoding process, but they are not transmitted as a part of the systematic output of the encoder. To ensure though that the output is also divided by 8 and 16, two filling zero bits are appended to the final codeword to produce an 8160 bit output.
- Parity check matrix is quasi-cyclic: is consists of a 2×14 array of 511×511 sparse circulants (image 13). Generator matrix in systematic circulant form is provided in the standard. The non-systematic part of it is a 14×2 array of 511×511 dense circulants.

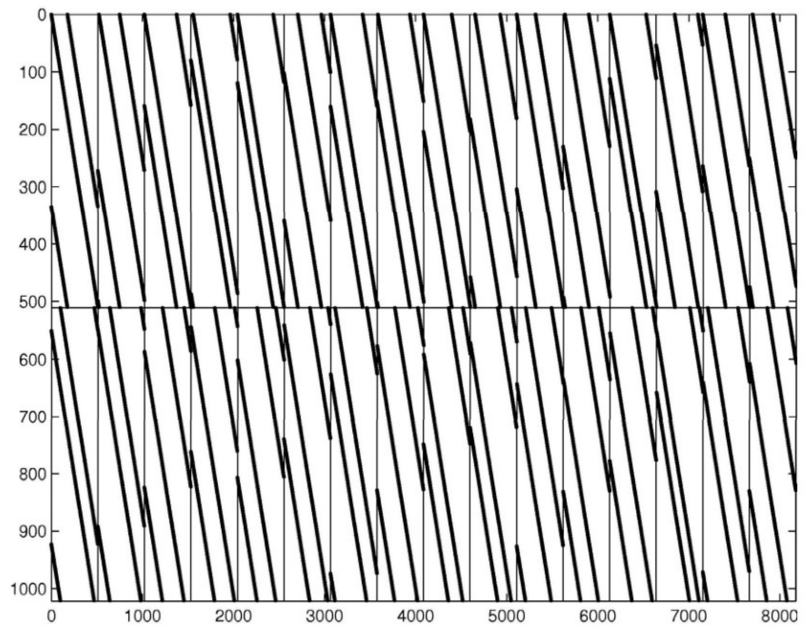


Figure 14: Scatter chart of the parity-check matrix of C2 LDPC code

2.3 Frame Synchronization and CADU structure

At the receiving end, a method is required for discerning the boundaries of codewords in the received stream of code symbols, or else decoding process would fail: the decoding algorithm would be applied to the wrong sequence of received bits.

CCSDS standards require that LDPC codewords shall be synchronized with a specially designed bit sequence, called Attached Sync Marker (ASM). For AR4JA codes, this sequence is 64 bit, while for C2, a 32-bit sequence has been adopted. Note that the 64-bit ASM is the same for TM SDLP and Proximity-1.

The ASM patterns in hexadecimal notation are the following:

AR4JA: 034776C7272895B0

C2: 1ACFFC1D

ASM sequence is prepended to the encoded codeword to form a data unit called Channel Access Data Unit (CADU).

2.4 Randomization

The correct operation of the receiver requires that incoming data should contain adequate transition density of received symbols. Transitions help receiver maintain symbol synchronization with the coded symbol boundaries in the received signal. In addition, short periodic data patterns generate spurious frequencies which impair receiver's performance. The absence of randomization in the encoded data has been the source of several unexpected problems with the telemetry links of a number of projects [20]. Consequently, randomization is highly recommended by CCSDS standards, although not mandatory.

Randomization is assured through the bit-wise addition of the codeword data with a pseudo-random sequence generated by the polynomial $h(x) = x^8 + x^7 + x^5 + x^3 + 1$. This polynomial can be implemented an 8-bit Linear Feedback Shift Register (LFSR). A possible implementation of the LFSR provided in the standard is displayed on fig. 15 with a Fibonacci LFSR. At the beginning of each codeword, the LFSR is initialized to all 1s. The pseudo-random sequence is repeated every 255 bits until the end of the codeword.

It is important to note that the ASM sequence defined in previous paragraph is already optimized for transition density and should not be subject to randomization. Also, the specified randomization sequence remains the same for all protocols.

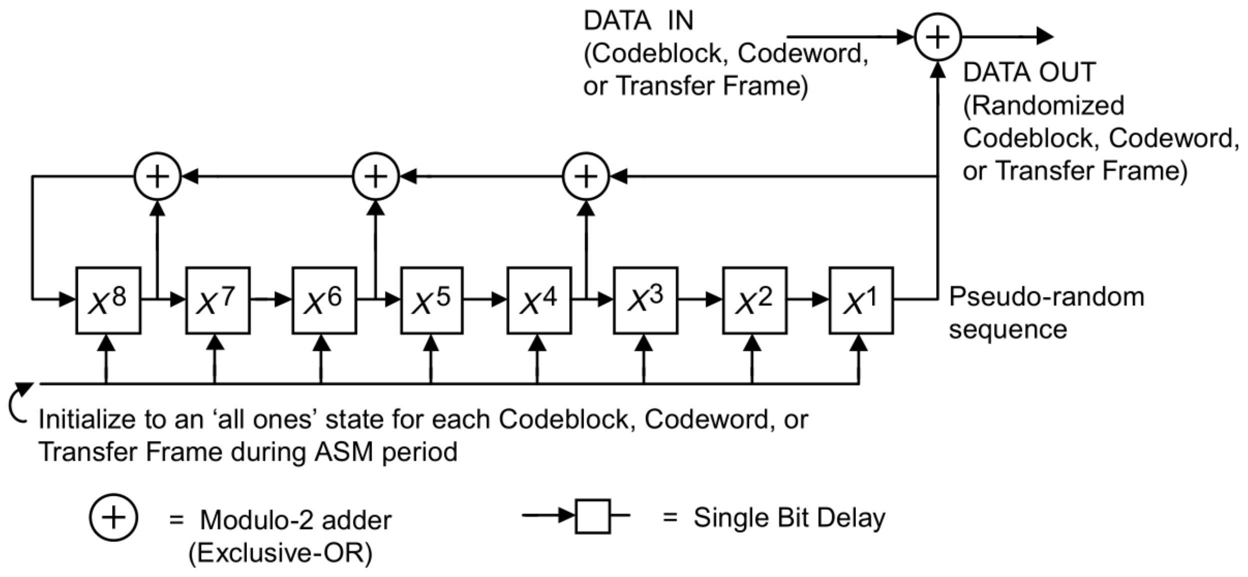

DATA IN
(Codeblock, Codeword, or Transfer Frame)
DATA OUT
(Randomized Codeblock, Codeword, or Transfer Frame)
x^8
x^7
x^6
x^5
x^4
x^3
x^2
x^1
Pseudo-random sequence
Initialize to an 'all ones' state for each Codeblock, Codeword, or Transfer Frame during ASM period
= Modulo-2 adder (Exclusive-OR)
= Single Bit Delay

Figure 15: A possible implementation of a CCSDS pseudo-random sequence generator[22].

3. ENCODER DESIGN

Having provided the necessary background for the description of the implementation, this chapter moves on to the implementation itself. A single encoder top entity is designed for all members of CCSDS LDPC codes and the corresponding diagram is displayed on fig. 16.

The encoder receives a continuous stream of data and produces a stream of CADUs. The receiving (slave) and transmitting (master) interfaces conform to AMBA 4 AXI4-Stream protocol [24] and are built according to the simplest possible configuration allowed by the protocol specification.

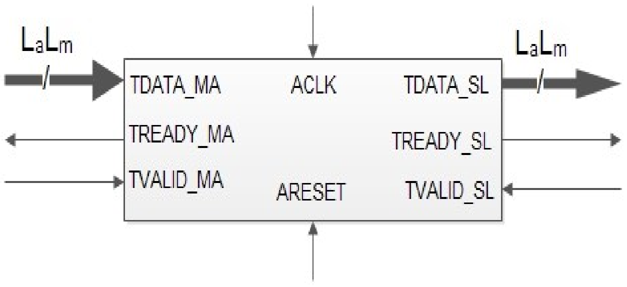


Figure 16: Encoder's top level diagram

The data buses are $L_m \times L_a$ bits wide. The meaning of the parameters L_m and L_a is to be clarified in this chapter. Valid data are framed by TVALID signal and a TREADY signal signifies the availability of the interface. According to the specification, for a transfer to occur both the TVALID and TREADY signals must be asserted at a rising clock edge.

No other signaling triggers the encoder to initiate the synthesis of a CADU other than the presence of valid data on the slave bus and the boundaries between successive TFs are not marked by any handshake signals but are kept by the encoder's counters instead. For maximum performance, the master (output) interface should be busy 100% of the time and this means that idle cycles should be imposed on the receiving (slave) interface through the AXI-4 Stream handshake signals. The timing of input and output data is displayed on fig. 17.

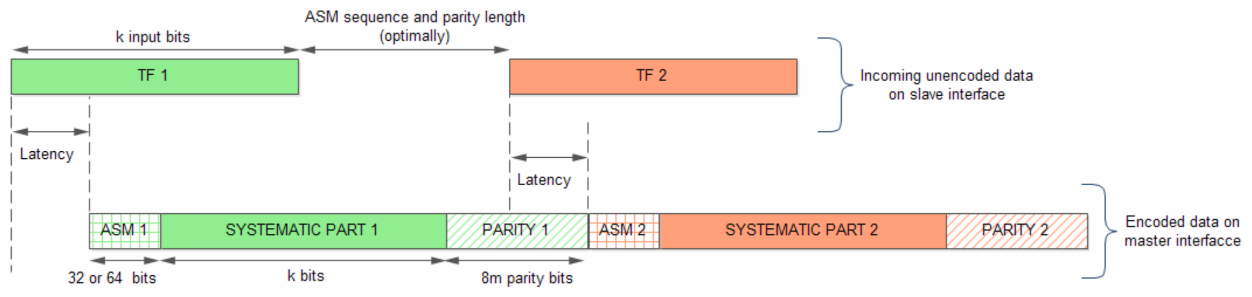


Figure 17: Timing of data on encoder's interfaces

3.1 Encoder Architecture selection

In §1.3 the general encoder architectures were briefly presented. Among the proposed architectures, the one that is most suitable for the characteristics of CCSDS codes needs to be selected for implementation.

The advantages of RCE encoder over the straightforward implementation are self-evident and have already been amply described in [11] and [13]. Consequently, the straightforward implementation is not subject to further investigation. Following the analysis for iterative encoders in §1.3.4, they are also excluded from investigation. The choice is to be made among the two remaining options, namely the RCE and RU encoder, based on the combination of resources and the performance each method can achieve. This analysis is similar to the work in [13], with the difference that a real AR4JA code is taken as an example here instead of the (small) example code considered in that paper. In addition, this work focuses on FPGA implementation possibilities.

3.1.1 RCE encoder resources

For all members of the AR4JA family, the generator matrix is composed of eight circulant columns for which eight RCEs similar to those displayed on fig. 8 can be used. Each RCE is implemented with m flip-flops (F/F), m 2-input AND operations and m 2-input XOR operations. The circulants are implemented by m function generators of *ceil(log$_2$(k/m))* inputs. For the interconnection of the RCEs, a negligible amount of resources are needed. In particular, 8 2-input multiplexors to select the desired input for the RCE (switches in fig. 8) and a small amount of control logic to activate them, not taken into consideration. The necessary resources are listed on Table 4.

Table 4: Bill of materials for RCE-based encoding

Resource	# needed	Example for k=1024 rate 1/2
AND	8xm	1024
XOR	8xm	1024
F/F	8xm	1024
ceil(log$_2$(k/m)) input function generators	8xm	1024 (3-input)
2-input multiplexors	8	8

3.1.2 Iterative encoder resources

The calculations based on Richardson-Urbanke work [12] can be executed according to the flow diagram of fig. 18 for maximum parallelism. The partition of parity-check matrix into submatrices is repeated from figure 9 for easier understanding.

The encoding process entails one dense matrix multiplication and two sparse. The partial products are added in the end. For the dense matrix multiplication in the first stage, RCE encoders shall be employed. The dense matrix J is quasi-cyclic: it is a

4×(k/m) array of m×m circulants. Following an analysis similar to the previous paragraph, the resources for the RCEs corresponding to the column J stage of Table 5 are calculated.

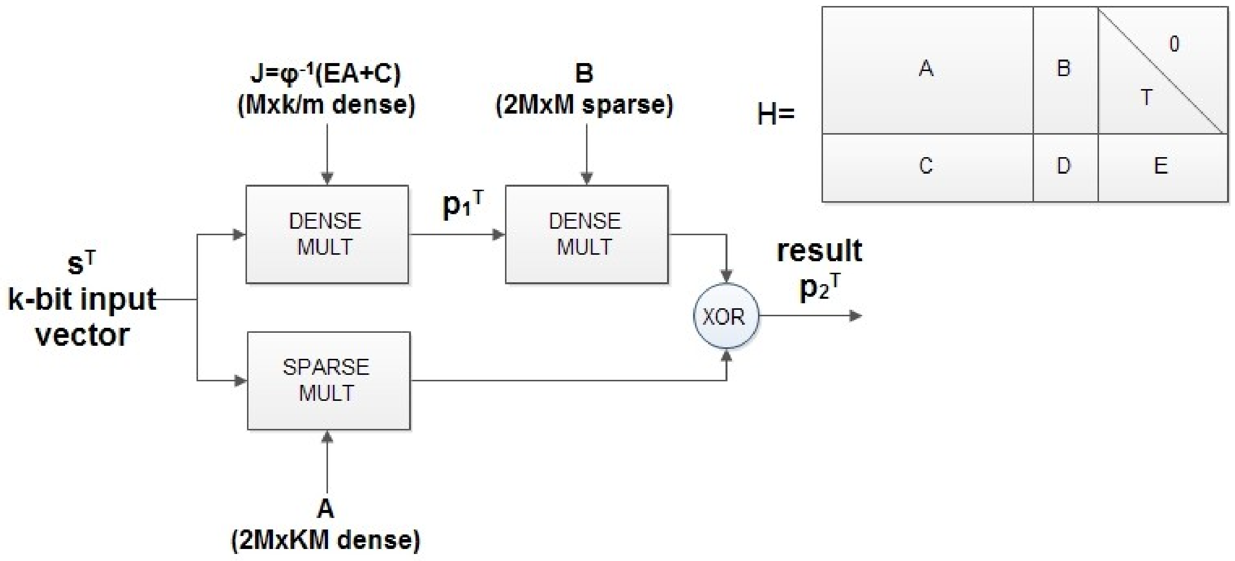


Figure 18: Iterative calculation of parity bits

Sparse matrices multiplications can be efficiently executed by the circuit of figure 11. For each circulant of size m this configuration needs 1 XOR operation, m F/Fs for implementation of the shift register, 1 m-input encoder and m OR operations between the elements of the shift registers to multiplex the input to the F/Fs between the value of the previous register and the input from the decoder. The last requirement for the shift registers was not taken into account in [13]. The control logic for x circulant rows can be implemented by a function generator of ceil($\log_2(x)$).

Matrix A is an *8×4K* array of m×m circulants. The first 4 rows are always zero, so the function generator required for the control logic can be simplified to 2 inputs. Column.

Similarly, matrix B is an 8×4 array of m×m circulants. Resource reuse between stages A and B cannot be established for pipeline operation, so independent hardware should be allocated to these two stages. Table 5 summarizes the results.

A comparison between the two matrices justifies the assumption that the iterative encoder –at least at this form- is not an appealing proposal. The simplification of the functions generators and the reduced number of AND and XOR functions of the iterative encoder implementation are balanced by the increased number of flip-flops, the addition of the OR functions amidst the elements of the shift registers and the large multiplexors and decoders Another important drawback of the iterative encoder is the higher latency introduced. B stage operations cannot start before all bits of the TF have been processed by stages A and J.

As stated in [13], modifications could be made to the standard's code design without significant impact on the BER performance of the code. This however is an area of considerable interest and research of information theory and future developments are heavily anticipated.

In this thesis, the design based on the RCE is selected for the implementation of the CCSDS LDPC codes. Optimizations of the basic design are presented in next paragraph

Table 5: Bill of materials for iterative encoding

Resource	# J stage	#A stage	#B stage	TOTAL (example for k=1024 rate 1/2)
AND	4xm	-	-	512
XOR	4xm	4xK	32	512+8+32=552
F/F	4xm	4xKxm	4xm	512+1024+512=2048
1 input function generators	4xmxK	-	-	1024
3 input function generator	-	-	4	4
2 input function generators		4xK	-	8
2-input multiplexors	k/m	-	-	8
m-input multiplexor	-	4xK	4	12 (128-input)
m-input decoder	-	4xK	4	12 (128-input)
OR	-	4xKxm	4xm	1024+512=1536

3.1.3 Parallel RCE implementation

The RCE described so far (fig. 8) is capable of serial (one bit at a time) output of the calculated parity bits. A parallel output of a number of L_m bits can be produced with a modification of the basic RCE according to fig. 19. In the image, L_m=4 for easier understanding. Note that shift operations have a step of L_m bits to the right, instead of one in the shift register of figure 8.

Alternatively, the input at each register can be conceived as a function generator of the L_m information bits and the *$log_2(k/m)$* bits of each function generator f_i. It follows that increasing the L_m parallelism leads to larger combinational paths in the design but at the same time it increases throughput.

Another source of parallelism can arise from the structure of the generator matrix. For all members of the family, it consists of k/m circulants of size m×m each. Two or more circulants (generally L_a) can be processed at the same time, provided that the corresponding input information bits are available. The partial products of the multiple circulants are XORed. Figure 20 is a simplified diagram showing this possibility. Different colors are employed to show the parts of the generator matrix for which each branch of AND-XOR operations is responsible.

The two sources of parallelism are are thus described by two corresponding degrees: L_m describes the successive bits parallelism and L_a multiple circulants parallelism.

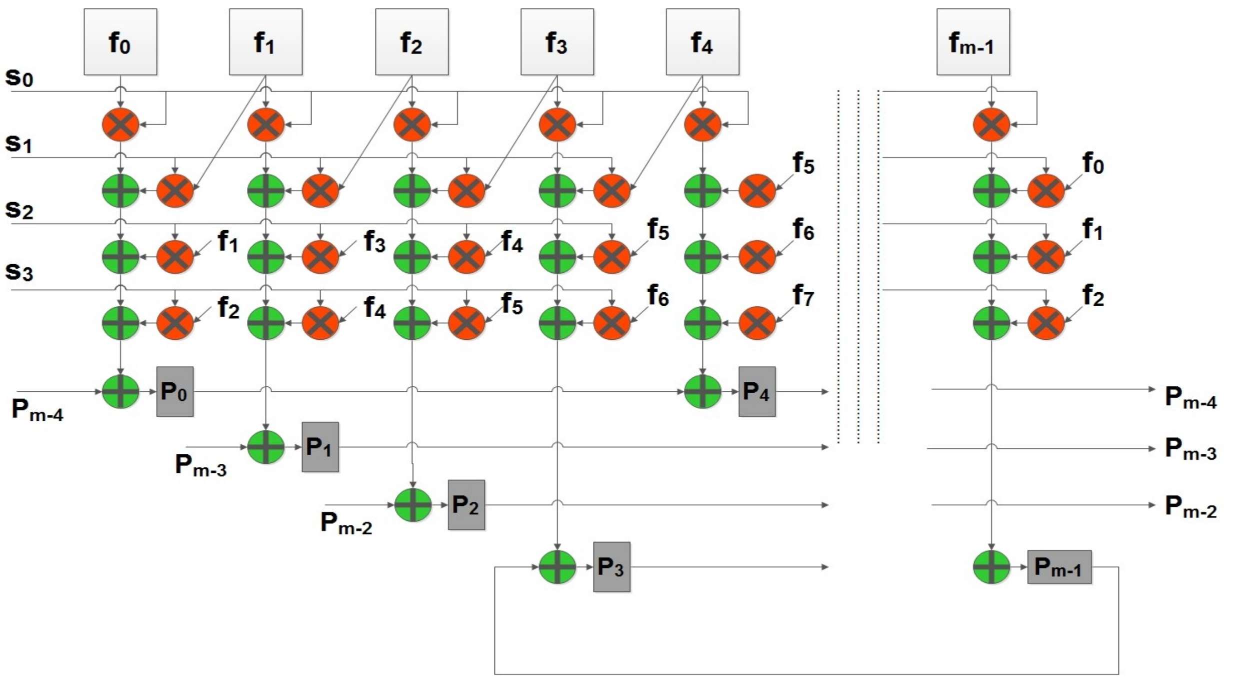


Figure 19: A RCE module for parallel processing of L_m=4 bits of mxm circulants.

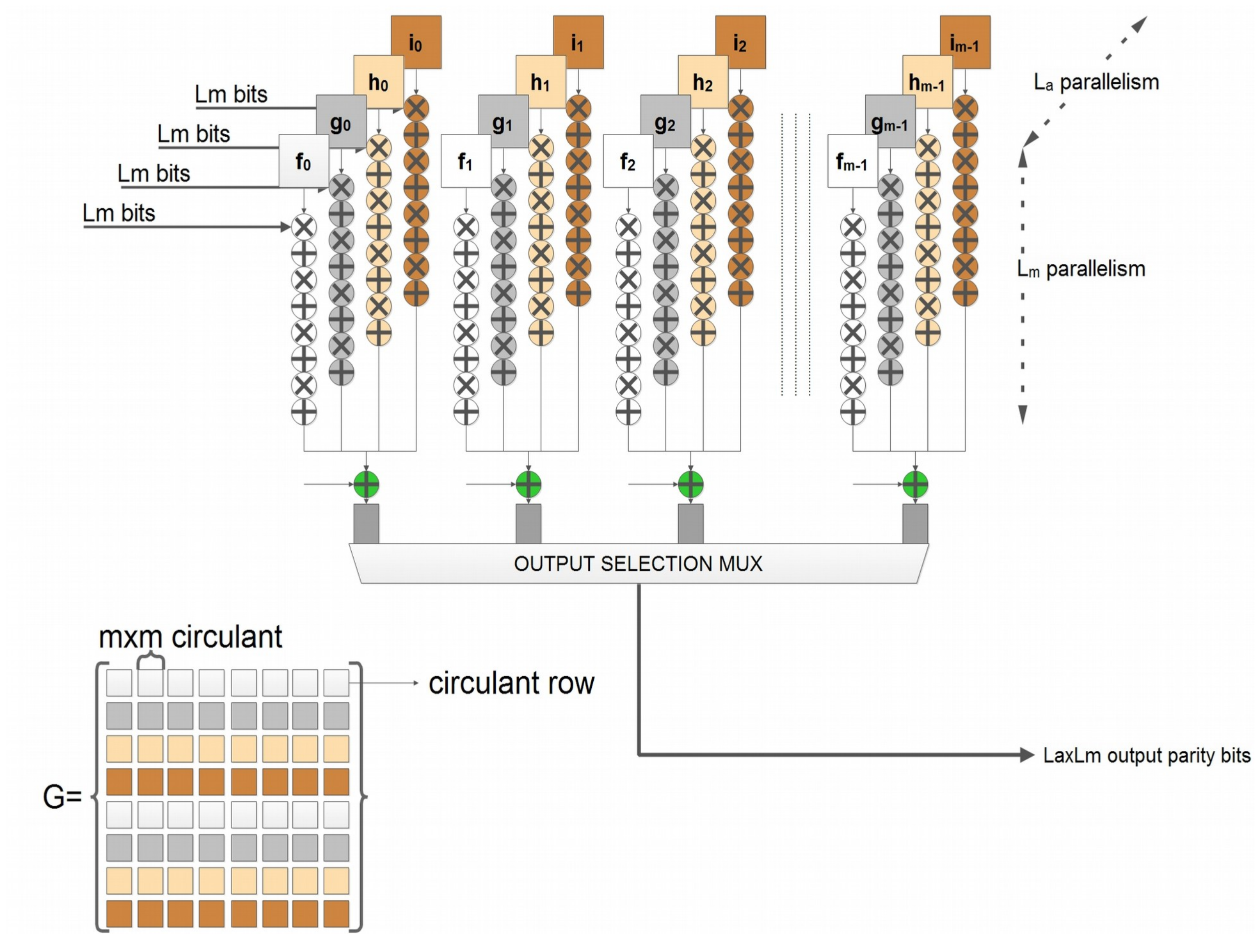


Figure 20: Simplified view of a parallel RCE with both sources of parallelism: L_a and L_m.

Increasing L_a parallelism comes with the advantage of simplification of the function generators (f_i, g_i, h_i, i_i in fig. 20) for the values of the generator matrix. In the example given in this figure, the 3-input functions generators necessary for 8 rows of circulants of the generator matrix are simplified to just 1-input. Since this simplification affects all the 8xm functions generator for all the parity bits of the code, considerable amounts of resources can be saved, leading to more efficient encoders.

On the other hand, the information bits corresponding to a number of L_a circulants need to be available for the RCE operation to begin. In cases where the information bits arrive at the encoder as a stream of data, buffer structures are necessary to save them until the required amount for processing has arrived. These buffer structures and the control logic required for their operation place demands on the resources budget of the design. In fact, the amount of memory required is not just mL_a bits, but double this ($2\times m\times L_a$) for uninterrupted operation. In addition, the commencement of parity calculations for a given TF has to be delayed for at least $(L_a\text{-}1)\times(m/L_m)/L_a$ clock cycles, introducing an equal amount of latency.

The product of these two sources of parallelism ($L_m\times L_a$) can give the total combined degree of parallelism of the encoder which describes the total number of input bits to the encoder. Generally, this number should be a power of two, so as to match the width of computer buses. For optimal performance, calculated parity bits should also be output in the same number. For $L_a>1$ and optimal performance (i.e. L_axL_m bits output), it is not possible to just shift out the calculated parity bits, like the encoder of fig. 8 or fig.19. The $L_a\times L_m$ output parity bits are selected each time from a multiplexor shown in fig. 20. The multiplexor and associated control logic are another source of complexity as a result of an encoder selection with $L_a>1$.

For AR4JA codes, the encoders described in this thesis implement every reasonable combination of L_a and L_m parallelism for a given amount of total parallelism ($L_a\times L_m$), leading to different compromises between latency, speed and resource utilization. These results are provided in a subsequent chapter. The degrees of parallelism are design parameters statically defined for each individual implementation. An obvious limitation in the value of L_a is that it cannot exceed the number of the circulant rows of the generator matrix. For example, for all rate ½ AR4JA codes it cannot exceed the value of 8.

For C2 code, the circulant size is 511 bits. Any value of L_a parameter other than one would impose very high latency and at the same time require a large number of memory for FIFO and PRCE structures, so it avoided and only the case of $L_a=1$ is considered for this code. L_m parallelism is constant at 16 bits for this encoder.

3.2 Components Description

This paragraph describes the implemented encoder a block diagram of which is displayed on fig. 21. The parallel RCE of fig 20 has been incorporated and can be recognized in the figure.

The Control and Buffer unit implements the receiving interface and accumulates incoming data until the necessary amount has arrived for RCE operation to start. For a given value of the L_a parameter, the parallel RCE of fig. 20 is not able to operate (efficiently) until all the L_a branches of the PRCE tree have data to process. In particular, $(L_a\text{-}1)m+L_aL_m$ bits need to have arrived to the Control and Buffer unit for the calculation of parity bits to begin. The L_m bits of the L_a circulants concurrently being processed by the parallel RCE are applied to it through *s_feed* signal. The shift registers of the encoder are controlled by two signals issued from the Control Unit: *mac_en* signal, which is the clock enable of the corresponding registers and *reset_prce,* which re-initializes the contents of the registers to all zeros after parity calculation and export of each CADU.

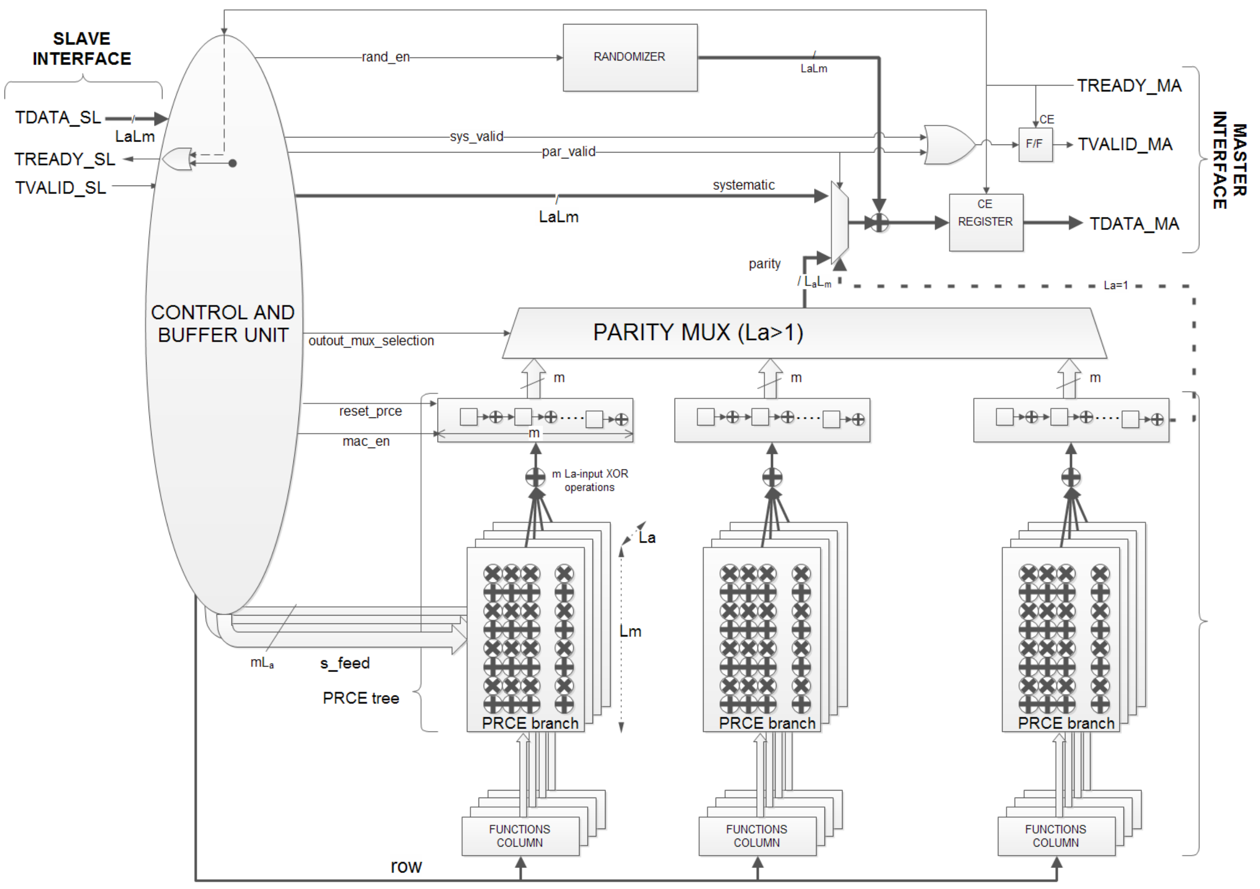


Figure 21: Encoder Block Diagram.

At the same time, incoming information bits form the receiving interface form the systematic part of the CADU. The systematic output, in which the ASM sequence is included, is multiplexed with the parity bits by the MUX at the upper part of the image, to select the output vector.

If randomization has been selected, the bits of the CCSDS pseudo-random sequence generator are XORed with the output of the MUX. The operation of the randomization circuit is controlled by the *rand_en* signal.

Output data are valid during either the systematic or during parity output where *sys_valid* and *par_valid* signals are correspondingly asserted. The master interface validity signal is consequently the result of the OR operation on these signals. Incoming TREADY_MA signal on the master interface is the clock enable of the output registers of the encoder and is also routed to the Control Unit to halt the operation of the encoder. In fact TREADY_MA becomes directly (through a gated combinatorial path) the TREADY_SL signal of the receiving interface, as indicated by the OR gate in the control unit in the above diagram. In cases where the encoder is to be used in an embedded system where this arrangement could significantly jeopardize performance, the circuit of fig. 22 could be used. In applications considered in this work however, no critical paths were presented along this path and the this solution was not necessary.

When the encoder is ready, TDATA and TVALID signals from the transmitting unit are selected from the multiplexors. Registers D1, D2 keep the last value of these signals when the output from D3 is asserted. When TREADY_SL is de-asserted, the values saved by the delay elements D1, D2 are maintained in the registers. During the first cycle after TREADY_SL assertion, they are provided to the corresponding encoder's inputs. The timing diagram of fig. 23 is an effort to clarify this. Only TDATA is displayed but the timing of TVALID is identical.

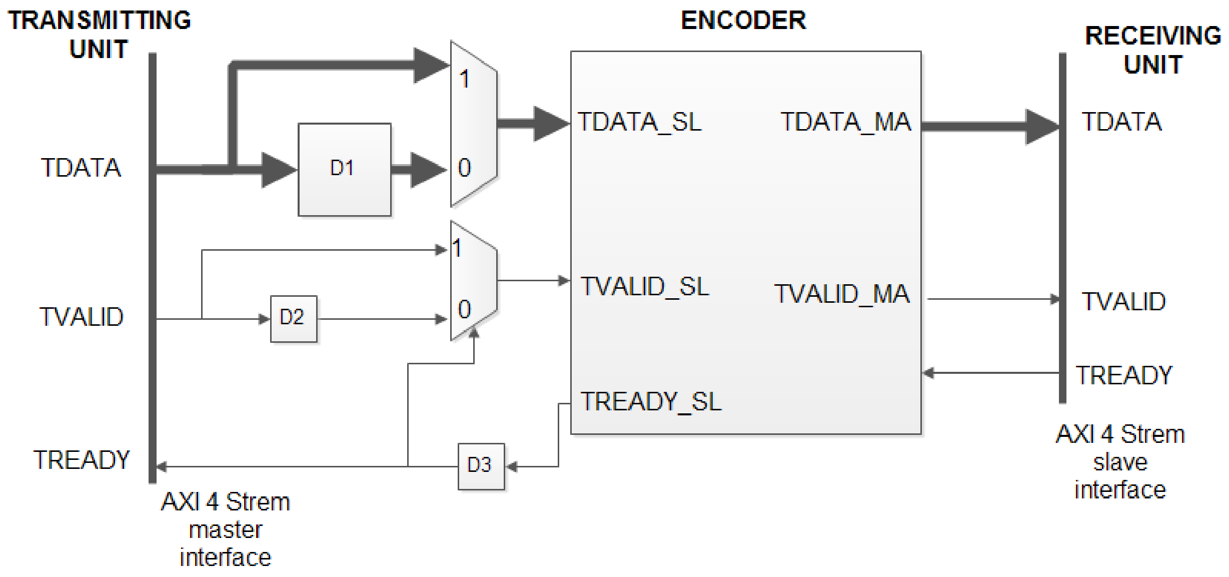


Figure 22: A solution to the non-registered output TREADY_SL

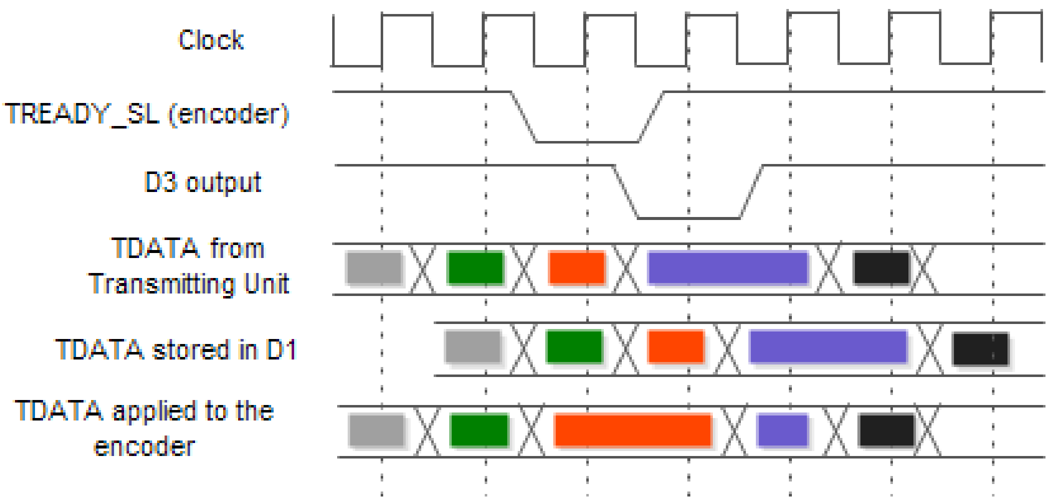


Figure 23: Timing example of the proposed solution

Figure 21 also describes the simplifications allowed in the case where L_a parameter is 1. In particular, the PARITY MUX can be omitted in that case and L_m parity bits can be simply shifted out of the shift registers, as described in previous paragraph, to form *parity* signal.

Each FUNCTION COLUMN block in the figure describes m function generators for one branch of the parallel RCE each. For example, regarding Figure 20, all f_i functions are described by one such block.

The description of the building blocks of the encoder continues with the randomizer and control and buffer units.

3.2.1 RANDOMIZER unit

Randomizer unit implements the polynomial mentioned in § 2.4. Since the output of the encoder is not one bit at a time but $L_a \times L_m$, a parallel implementation is needed. The solution adopted for this design is displayed on fig. 24.

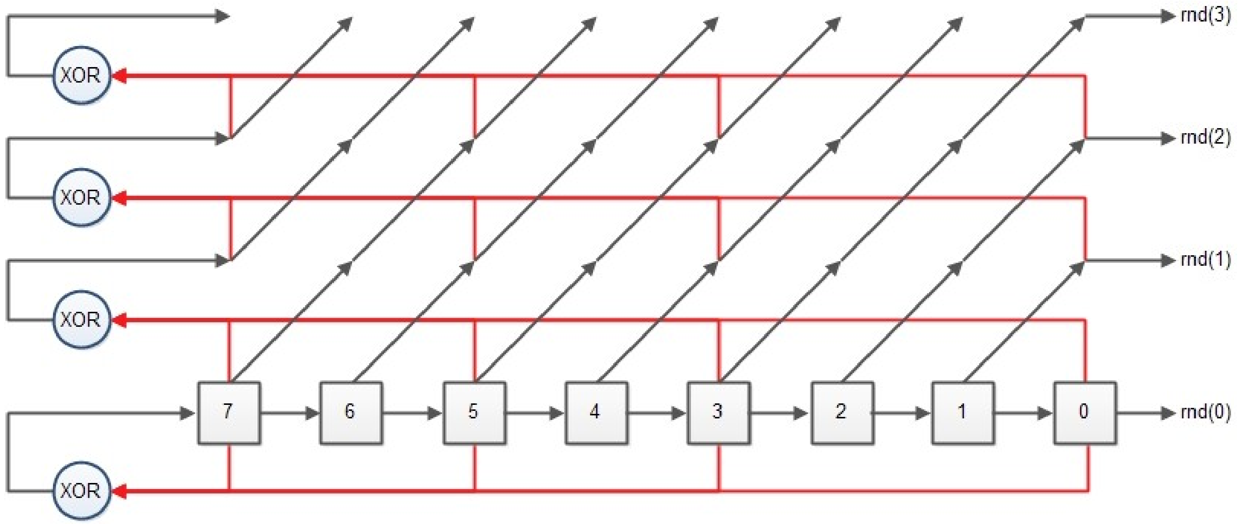


Figure 24: Implementation of a parallel CCSDS pseudo-random sequence generator.

In the figure, the LFSR at the bottom implements the polynomial, with the output coming from the rightmost register (numbered 0). A number of N bits of the pseudo-random sequence can be obtained if the bit sequence in the LSFR is expanded to an array of N vectors of 8 bit each according to the following algorithm:

- Considering an array of vectors, *lfsr_array*, the first element of the array (*lfsr_array(0)*) is the LFSR itself.
- Foreach vector from 1 to N, the bit position i takes the value of the bit position *i+1* of the previous vector, except from bit position 0.
- Bit position 0 takes the value of the XOR operation corresponding to the polynomial over the bits of the previous vector.

It is evident that the for the first eight bits of the produced result no additional XOR gates are needed.

3.2.2 Control and Buffer Unit: the general case

The general case for considered first is the control unit for an encoder with a “reasonable” value of L_a>1. For such an encoder, the latency introduced by the L_a parameter is adequately small so that the next TF arrives to the encoder when it (the encoder) outputs the parity bits of the current TF. This is the case depicted on fig. 17 and the corresponding part of fig. 25. Note the difference between the term Latency in fig. 17 and Systematic Latency in fig. 25. The former was introduced during a high level description of the encoder and refers to the latency from receiving unit's perspective, while the latter is the number of cycles it takes to begin the output of the systematic part of the CADU. Systematic Latency is caused by the L_a parameter and it is the time it takes to accumulate enough data for the operation of all branches of the parallel RCE.

Obviously this latency does not exist for L_a=1. Also shown in the figure is the case of very high latency, such that the next TF arrives to the encoder while the systematic output of the current TF has not finished. This situation calls for a different FSM of the control unit examined later.

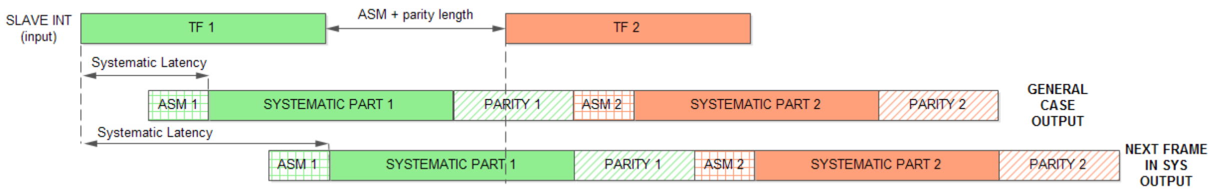


Figure 25: Timing of data on encoder's interfaces for the general and high latency cases

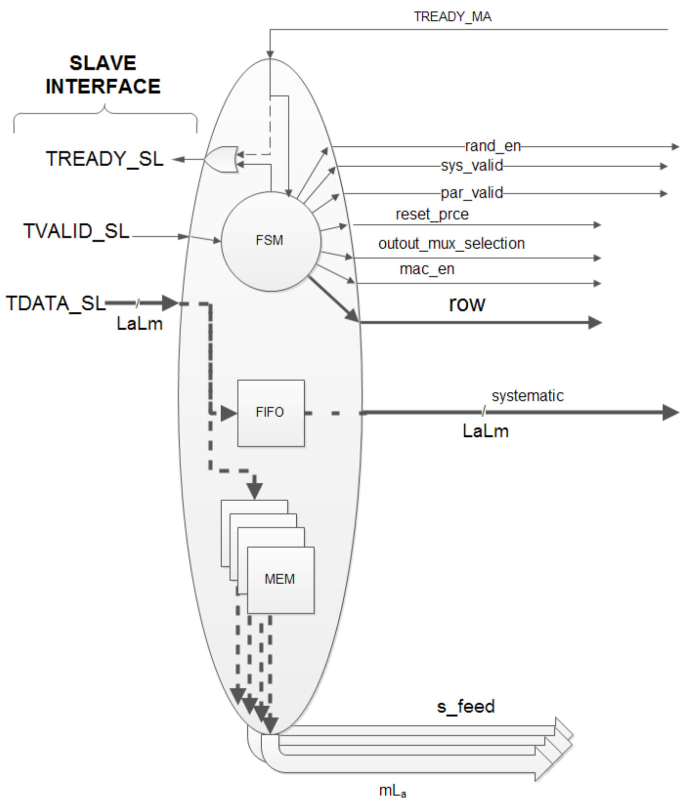


Figure 26: Control and Buffer Unit

Apart from the generation of all control signals for the encoder's operation, the control and buffer unit includes also the necessary memory structures to buffer incoming data on slave interface. The first such structure is a FIFO for the systematic part of the output. Until enough data have gathered for RCE operations for all branches of the parallel RCE, incoming data arriving at the encoder at a rate of $L_a L_m$ bits in every cycle, are queued in a FIFO for subsequent output. The size of this FIFO is therefore such that all bits arriving during the latent cycles can be stored. Since the Systematic Latency period is $(L_a\text{-}1)(m/L_m)/L_a$ clock cycles, the FIFO should accommodate for this number of words of $L_a L_m$ bits.

The second memory structure necessary for the Parallel RCE operation rearranges incoming "packets" of $L_a L_m$ bits into pages of m bits each, where each page contains input bits referring to the same circulant of the Generator matrix. Due to L_a parallelism, L_m inputs bits of L_a circulants are concurrently processed by the PRCE. Considering uninterrupted operation of the sender, one clock cycle after the L_a-*1* of these pages have been filled with data, L_aL_m bits for the last page are received by the encoder. This condition fulfills the prerequisites for commencement of RCE operations. For fully pipelined operation however, the double amount of memory is required for the structure to work as a double buffer. In the actual implementation, resource sharing between these two memory structures exploits the same resources for both memories.

Control signals generation and routing of data to the described memory structures are orchestrated by a FSM. The top-level diagram of the unit with a simplified pictorial representation of its constituent structures is provided on fig. 26, while fig. 27 displays the transition diagram of the FSM. The diagram of fig. 27 does not intend to provide a detailed description of the FSM, but to assist in the better understanding of the functionality of the control unit. All the hardware for the Control and Buffer Unit is described by a FSMD model at a high level of abstraction and amply documented in-line with the code.

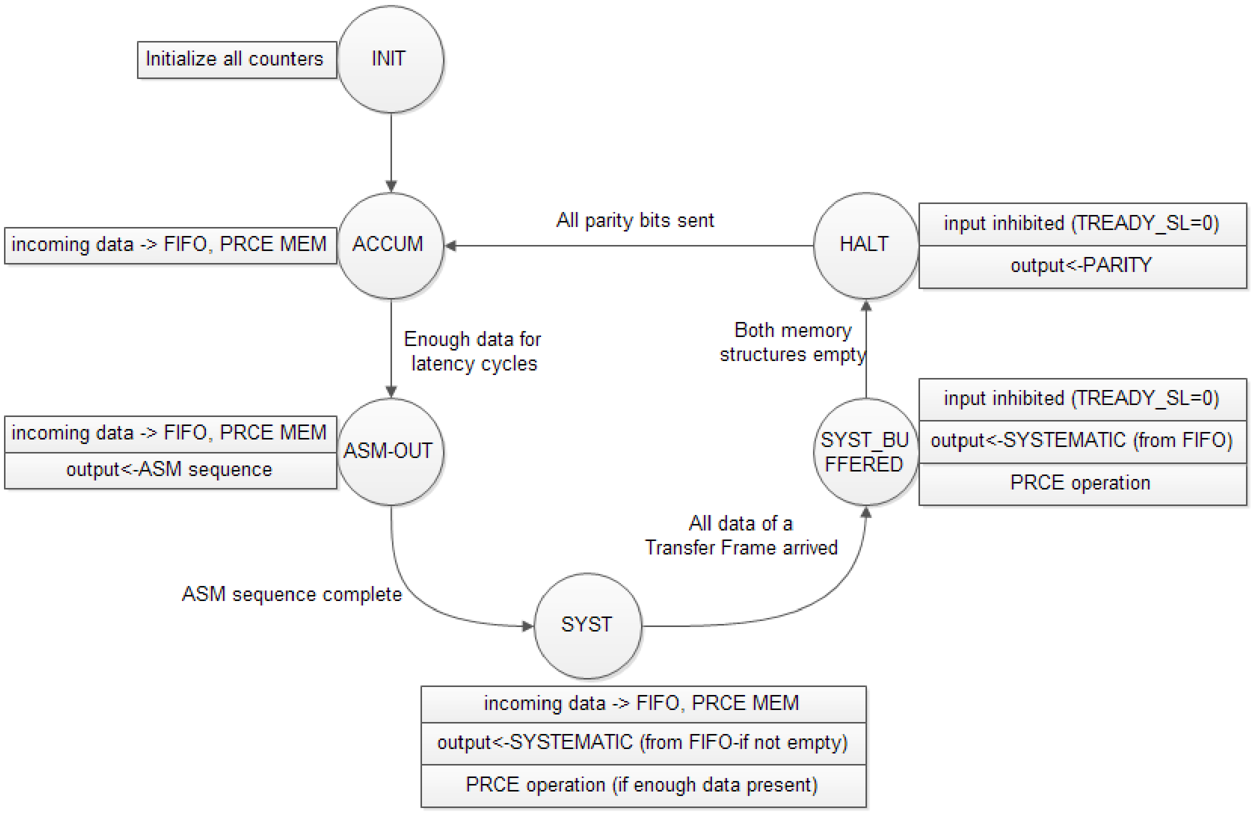

Initialize all counters
INIT
incoming data -> FIFO, PRCE MEM
ACCUM
All parity bits sent
HALT
input inhibited (TREADY_SL=0)
output<-PARITY
Enough data for latency cycles
Both memory structures empty
incoming data -> FIFO, PRCE MEM
output<-ASM sequence
ASM-OUT
SYST_BUFFERED
input inhibited (TREADY_SL=0)
output<-SYSTEMATIC (from FIFO)
PRCE operation
All data of a Transfer Frame arrived
ASM sequence complete
SYST
incoming data -> FIFO, PRCE MEM
output<-SYSTEMATIC (from FIFO-if not empty)
PRCE operation (if enough data present)

Figure 27: Simplified state transition diagram of the FSM of the Control and Buffer Unit

3.2.3 Control and Buffer Unit: L_a=1.

Considerable simplifications can be made to the Control and Buffer unit of the previous paragraph in the case where L_a parameter is 1, the most important of them being the elimination of the two memory structures (FIFO and PRCE memory), but also the FSM can be considerably simplified.

The FSM in this case becomes that of fig. 28 and as expected, it comprises fewer states than that of fig. 27. SYST_BUFFERED state is not necessary here. Another important difference of this simplified FSM has to do with the behavior of the slave interface.

During IDLE and ASM_OUT states, the encoder keeps TREADY_SL signal in low state. A sender unit however should keep the TVALID_SL signal high if it has data to send and this is in fact the event that triggers IDLE→ASM_OUT transition. This behavior is compliant with the protocol specification, according to §2.2.1 of [24], which explicitly allows a slave to wait for TVALID to be asserted before asserting the corresponding TREADY. Another important difference is that contrary to the previous case, input is inhibited during ASM output.

At the end of HALT state, when all parity bits have been transmitted through the master interface, the presence of an asserted TVALID signal on the slave interface initiates a transition directly to the ASM_OUT state, instead of IDLE. Since the FSM receives indication that the sender has more data to send (a new TF), one cycle of latency is saved by this transition. Like before, input from sender is inhibited by a de-asserted TREADY_SL signal until the FSM reaches SYST state.

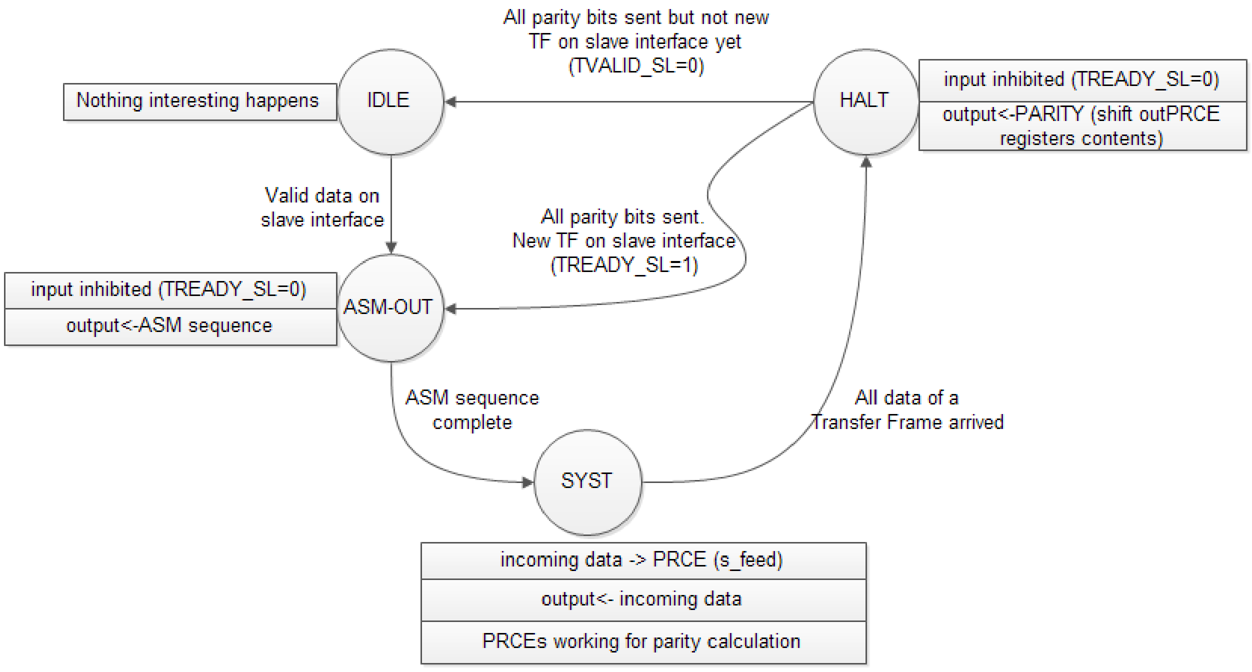

All parity bits sent but not new TF on slave interface yet (TVALID_SL=0)
Nothing interesting happens
IDLE
HALT
input inhibited (TREADY_SL=0)
output<-PARITY (shift outPRCE registers contents)
Valid data on slave interface
All parity bits sent. New TF on slave interface (TREADY_SL=1)
input inhibited (TREADY_SL=0)
output<-ASM sequence
ASM-OUT
ASM sequence complete
All data of a Transfer Frame arrived
SYST
incoming data -> PRCE (s_feed)
output<- incoming data
PRCEs working for parity calculation

Figure 28: Simplified state transition diagram of the FSM of the Control and Buffer Unit for L_a=1

3.2.4 Control and Buffer Unit: High latency case

The latency caused by high values of L_a parameter calls for a separate FSM which should handle the co-existence of two consecutive TFs in the encoder. The memory structures size (FIFO and PRCE memory) are the same as the general case, but extra counters are needed to index the boundaries of the two Transfer Frames into these structures.
The FSM is more complicated and its state transition diagram is provided in fig. 29. Figure 30 displays the state of the FSM on the timing diagram of successive CADUs. During ACCUM state in this case, parity data of the previous CADU are transmitted on the master interface, while the encoder is receiving and buffering the current TF. The machine exits SYST_BUFFERED state when the number of data in the systematic FIFO structure indicate that the necessary number of latent cycles have elapsed. If a new TF has arrived on slave interface, the FSM moves to ACCUM_OLDSYS state in which the current and previous TF co-exist in the memory structures, while the systematic bits of the previous CADU continue to be transmitted by master interface. Special counters record the boundaries between the two TFs in the memory structures.

When all the systematic bits of the previous CADU have been transmitted, the machine switches to ACCUM state in which the calculated parity bits of the previous CADU are transmitted on the master interface and at the same time the new TF is received. The ACCUM state is also the state at which the machine moves when no new TF has arrived after SYST_BUFFERED state and consequently the special counters for separation of two TFs in the memory structures are not necessary.
The condition that needs to be satisfied so that the high latency FSM is employed is that latency should be higher than the sum of the number of cycles needed for systematic output and ASM sequence.

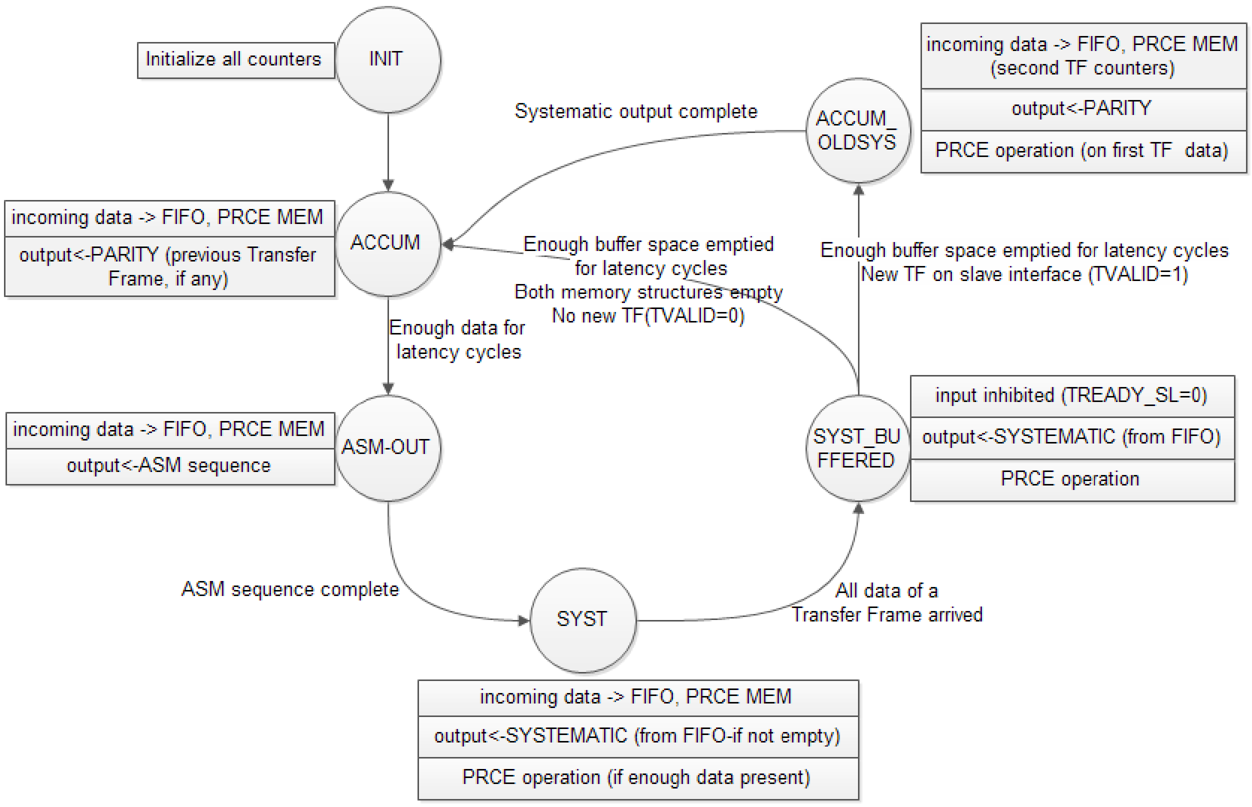

Initialize all counters
INIT
incoming data -> FIFO, PRCE MEM (second TF counters)
output<-PARITY
PRCE operation (on first TF data)
Systematic output complete
ACCUM_OLDSYS
incoming data -> FIFO, PRCE MEM
output<-PARITY (previous Transfer Frame, if any)
ACCUM
Enough buffer space emptied for latency cycles
Both memory structures empty
No new TF(TVALID=0)
Enough buffer space emptied for latency cycles
New TF on slave interface (TVALID=1)
Enough data for latency cycles
input inhibited (TREADY_SL=0)
output<-SYSTEMATIC (from FIFO)
PRCE operation
SYST_BUFFERED
incoming data -> FIFO, PRCE MEM
output<-ASM sequence
ASM-OUT
ASM sequence complete
SYST
All data of a Transfer Frame arrived
incoming data -> FIFO, PRCE MEM
output<-SYSTEMATIC (from FIFO-if not empty)
PRCE operation (if enough data present)

Figure 29: Simplified state transition diagram of the FSM of the Control and Buffer Unit for high latency case.

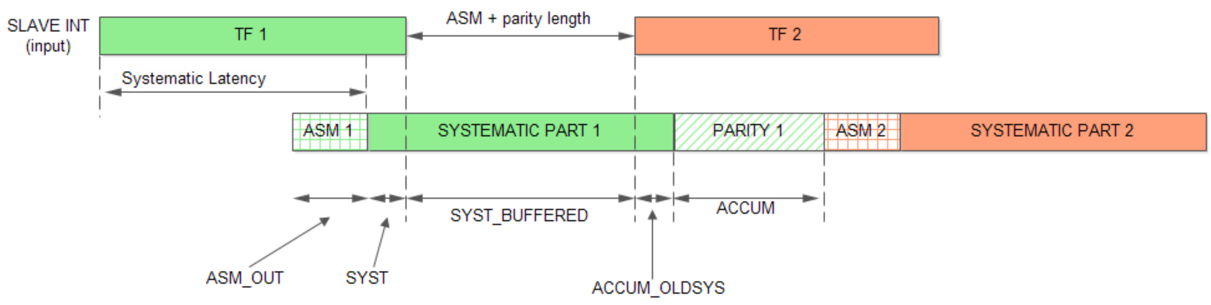


Figure 30: Timing diagram of the FSM states for high latency case.

3.2.5 Control and Buffer Unit: Very small Latency

A fourth version of the FSM is necessary when the latency is so small that it equals the number of cycles necessary for the transmission of the ASM sequence. This situation itself and the corresponding FSM transition diagram bear significant resemblance to the case of L_a=1: although FIFO and PRCE memory structures exist in this case. Valid data on slave channel trigger the INIT→ ASM_OUT transition but contrary to the L_a=1 case, TREADY_SL is asserted during ASM output and the incoming data are stored to the memory structures. Similar behavior to the L_a=1 FSM is exhibited by the transitions from the HALT state: if TVALID is asserted at the end of parity output indicating that a new TF is arriving on slave interface, the machine moves to ASM_OUT state, while in the opposite case it moves to INIT.

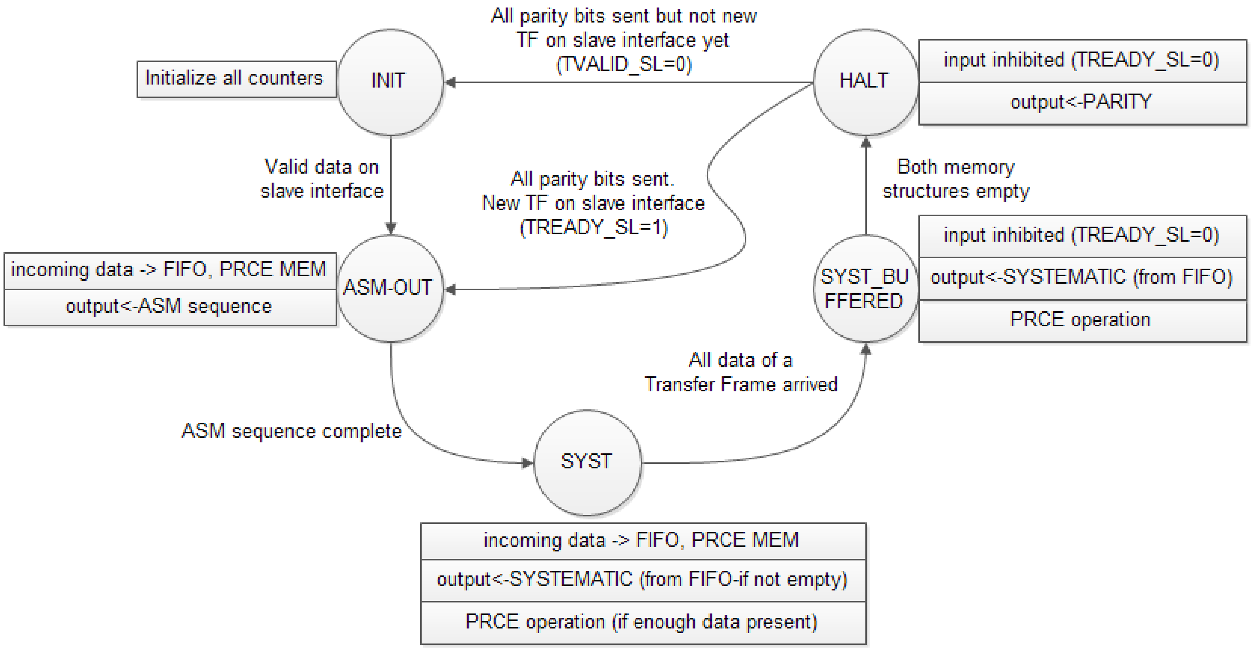

All parity bits sent but not new TF on slave interface yet (TVALID_SL=0)
Initialize all counters
INIT
HALT
input inhibited (TREADY_SL=0)
output<-PARITY
Valid data on slave interface
All parity bits sent. New TF on slave interface (TREADY_SL=1)
Both memory structures empty
input inhibited (TREADY_SL=0)
output<-SYSTEMATIC (from FIFO)
PRCE operation
incoming data -> FIFO, PRCE MEM
output<-ASM sequence
ASM-OUT
SYST_BUFFERED
All data of a Transfer Frame arrived
ASM sequence complete
SYST
incoming data -> FIFO, PRCE MEM
output<-SYSTEMATIC (from FIFO-if not empty)
PRCE operation (if enough data present)

Figure 31: Timing diagram of the FSM states for very low latency case.

3.2.6 Control and Buffer Unit: no HALT state

It is possible for some configurations that the latency has such value that it is not necessary to have a state such as HALT, in which input is inhibited and output comes from the calculated parity bits of the PRCEs registers.

Indeed, if latency has such a value that the parity output is complete at the exact cycle in which the encoder begins to output the ASM sequence for the next TF, the HALT state would be a source of suboptimal operation by inserting an idle cycle on the output bus. The FSM therefore of the control and buffer unit can be simplified according to the simplified diagram of fig.32.

The necessary condition for use of this FSM is that the sum latency plus two cycles (to account for the input-output buffers of the encoder) is greater than the sum of parity output cycles plus ASM sequence cycles. This latency value is an intermediate stage between the general case and the high latency case documented in previous paragraph.

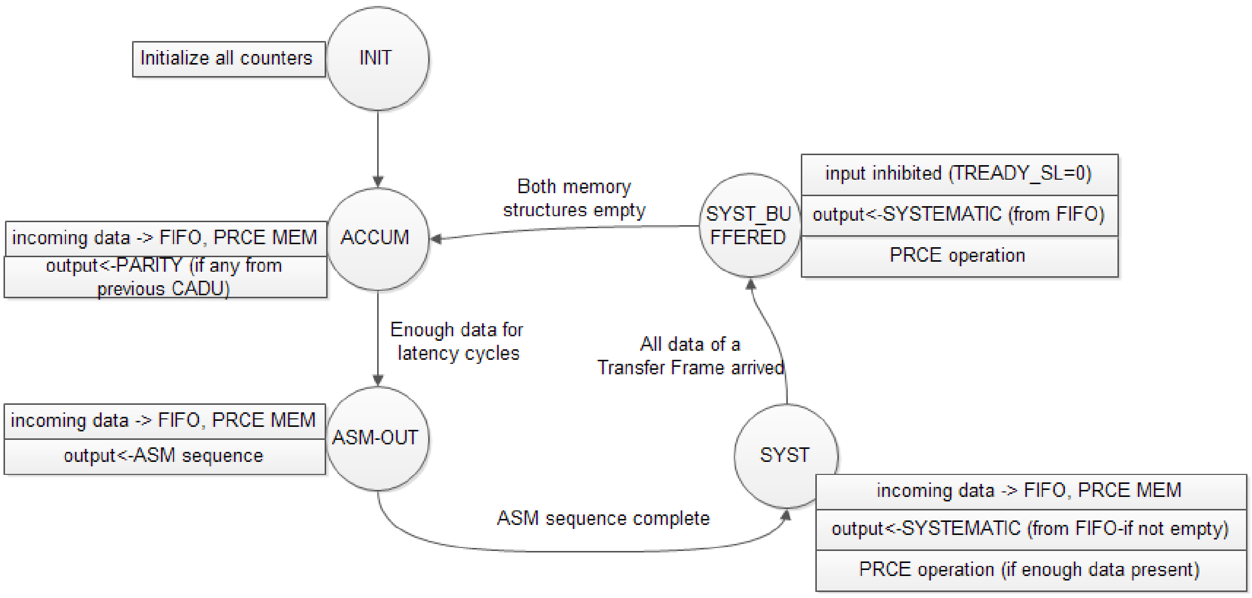

Initialize all counters
INIT
incoming data -> FIFO, PRCE MEM
output<-PARITY (if any from previous CADU)
ACCUM
Both memory structures empty
SYST_BUFFERED
input inhibited (TREADY_SL=0)
output<-SYSTEMATIC (from FIFO)
PRCE operation
Enough data for latency cycles
All data of a Transfer Frame arrived
incoming data -> FIFO, PRCE MEM
output<-ASM sequence
ASM-OUT
SYST
ASM sequence complete
incoming data -> FIFO, PRCE MEM
output<-SYSTEMATIC (from FIFO-if not empty)
PRCE operation (if enough data present)

Figure 32: Timing diagram of the FSM states for the case of no HALT state

3.2.7 Control and Buffer Unit: C2 code

For C2 code, the circulant size is 511 bits. Any value of L_a parameter other than one would impose very high latency and at the same time require a large number of memory for FIFO and PRCE structures, so it avoided and only the case of L_a=1 is considered for this code.

Another complication of this code is that the circulants size is not a power of two. Despite the manipulation described in §2.2 to make the input and output block lengths divisible by 8 or 16, the multiplication of the input vector with the generator matrix is problematic at the boundaries of the circulants. For the encoders of this work, input bus (slave interface) width is equal to L_aL_m, or just L_m since L_a=1 and L_m is a power of two. To mitigate this, L. Miles and S. Whitaker in [25] propose a method of packing input data on a 16-bit input bus in groups of 21 bits and then unpacking them to groups of 7 bits. Each multiplication operation is performed against 3 such unpacked groups of data (21 bits) and an equal number of elements of the generator matrix. For any circulant, the first 24 3-tuples are multiplied with the corresponding elements of the current circulant. Since 24×21=504, at the boundaries of the circulants, the first of these 3 groups is multiplied with the last 7 bits of the current circulant and the other with the corresponding elements of the next circulant.

The disadvantage of this solution is that although multiplication operations are performed on 21 bits at the same time, data flow into the encoder in a 16-bit bus, introducing thus a number of idle cycles in the operation of the MAC modules. This is apparently a waste of resources.

Another source of sub-optimality in the proposed encoder is that the 18 zeros which are prepended to the Transfer Frame before encoding according to the standard have an a-priori known result, since the result of multiplication with zero is always zero. An optimal encoder does not need to waste calculation cycles for these prepended bits but should directly incorporate the effect they have on the final codeword, knowing that they always have 18 zeros as the result. The control unit of the C2 encoder proposed in this work is designed according to this optimizations.

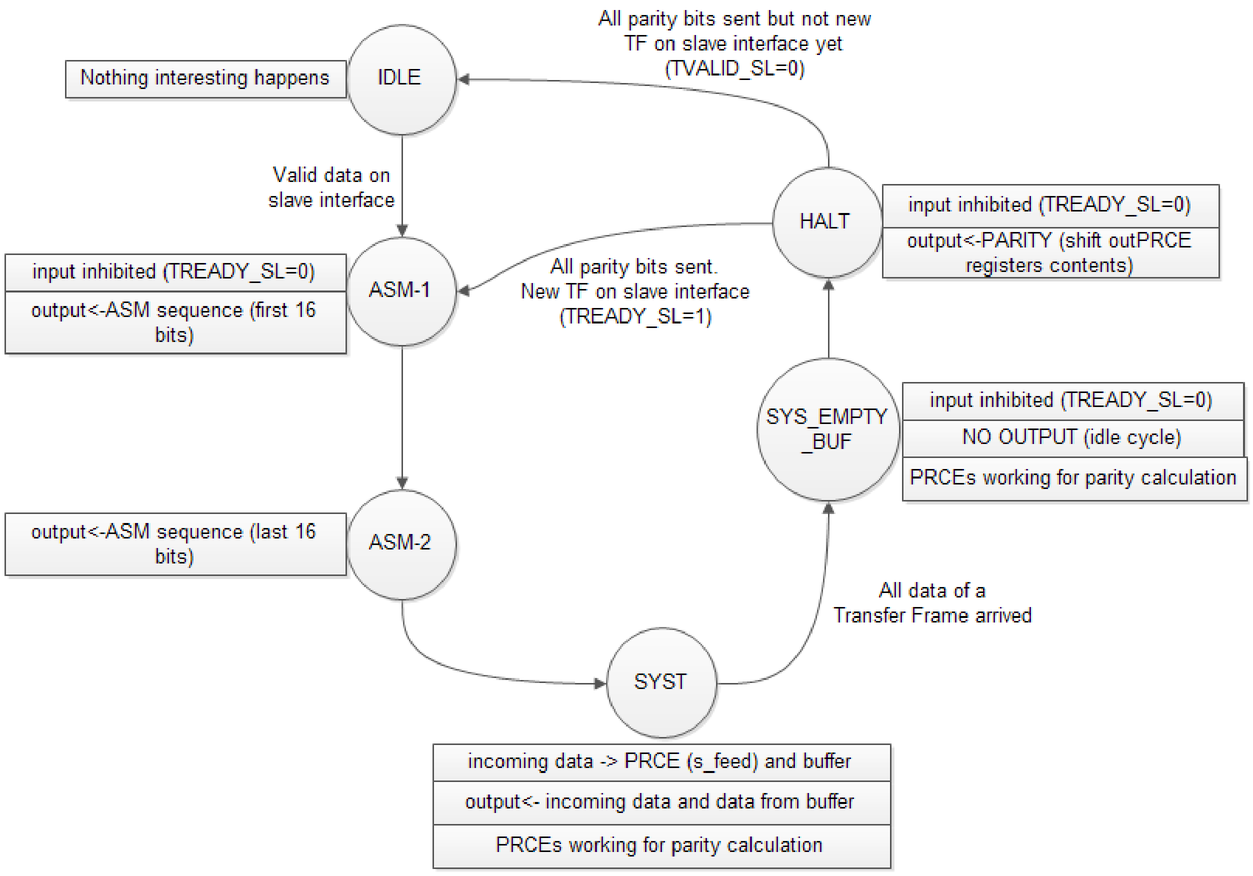

All parity bits sent but not new TF on slave interface yet (TVALID_SL=0)
Nothing interesting happens
IDLE
Valid data on slave interface
HALT
input inhibited (TREADY_SL=0)
output<-PARITY (shift outPRCE registers contents)
input inhibited (TREADY_SL=0)
output<-ASM sequence (first 16 bits)
ASM-1
All parity bits sent. New TF on slave interface (TREADY_SL=1)
SYS_EMPTY_BUF
input inhibited (TREADY_SL=0)
NO OUTPUT (idle cycle)
PRCEs working for parity calculation
output<-ASM sequence (last 16 bits)
ASM-2
All data of a Transfer Frame arrived
SYST
incoming data -> PRCE (s_feed) and buffer
output<- incoming data and data from buffer
PRCEs working for parity calculation

Figure 33: Timing diagram of the FSM states for C2 code.

The value of parallelism selected in this work is constant at 16 bits. Mismatch at the boundaries of each circulant occurs at every 32th group of 16 bits of the information block sequence. To handle this, the encoder of this work utilizes a different technique. A variable length buffer is used which saves a number of bits from the current input. Let N be the size in bits of this buffer at a given instance. Instead of sending the 16-bit input sequence to the PRCE for parity calculation directly, the control unit sends the N bits saved in the buffer in the previous cycle and the 16-N bits of the current input sequence. This value increments at the boundary of each circulant to accommodate for the 1 bit by which each circulant is short of 512, which is the number of bits received after 32 cycles of input. Especially for the last of these 32 groups of incoming information block bits, the control unit adds a zero to the sequence as the 16th bit, before incrementing the value N of the buffer. The 16-bit input sequence is thus converted to a 15-bit one, making the total number of bits corresponding to a circulant equal to $31\times16+15=511$.

The encoder saves one execution cycle by truncating the 18 prepended zeros to the code. For the first 16 of these 18 bits, the technique explained in next paragraph is employed, which does not require special handling from the control unit. For the last 2 of these 18 bits however, it is necessary to initialize the size of the input buffer to 2.

At the last (14th) circulant, the number of the buffer has reached the value 15, meaning that an extra cycle is needed for the buffer to empty its contents upon the PRCE. Also, after parity output, the two appended zeros are appended to the CADU.

The simplified state diagram of the FSM for the control unit of this encoder is provided in fig. 33. Similarly to the previously described FSMs, valid data on slave interface initiate a CADU. Note that for performance reasons, the ASM output is controlled by two states, the difference between them being the assertion of TREADY_SL signal by the last, so as to allow the sender transmit the next group of 16 bits in the next cycle (in SYST state). The SYS_EMPTY_BUF state is introduced in order to provide an extra cycle in which the bits stored in the buffers used for the alignment to the boundaries of circulants are all consumed. Otherwise, the FSM is similar to that of L_a=1 for AR4JA codes.

3.2.8 Function generators

The function generators for AR4JA codes simply provide the first row of the code circulant based on the value of “row” input which selects the current circulant. These parameters can be easily calculated following the methodology in [22] using a software like MATLAB. Note also that if L_a parameter is equal to the number of circulant rows in the generator matrix (e.g. 8 for k=1024 and rate ½), these function generators are simplified to constants.

For C2 codes however the situation is complicated by the fact that the selection of the row has to serve two more functions described here.

- In the steps described in previous paragraph for the alignment of the 16 bit input to the boundaries of 511-bits circulants, the last bit of the 16 bits applied to the PRCE during the 32th multiply operation is forced to zero, in order to describe a 15-bit multiplication, since the result of multiplying with zero is also a zero. The shift operation however executed by the shift register is always 16-bits. This discrepancy can be compensated for by a left shift by one position of the parity register bits. Since however the circulants of C2 code are right circulants, it is equal to providing to the function generator the last line of the next circulant, instead of the first, because he last line is the left cyclic shift of the first.
- Following the same reasoning, the effect of the multiplication of the 18 prepended zeros is simply a cyclic shift operation, which is equivalent to the cyclic rotation of the circulants. Taking into account the last shift operation of the shift register during parity calculations, which is equivalent to a rotation by 16 of the circulants, rotation of the circulants by two positions to the left is what it takes to simulate the multination by zero.

In short, the combined effect of the above factors is that for the first circulant, line 510 is the value of the function generator, for the second line 509 etc. Shift operations are thus executed by proper selection of circulants rows, without adding extra complexity to the encoder.

4. IMPLEMENTATION

4.1 Code design and parametrization

The encoder components analyzed in previous chapter are described in VHDL. Provided code is amply documented and all language structures are justified using state-of-the-art Doxygen documentation system and in-line comments.

Encoder parameters are globally defined in corresponding package file (DEFS.vhd), with the purpose of being easily modifiable from a central location. All these parameters are statically defined for a given implementation of the encoder (i.e. they cannot change after synthesis with a configuration register for example).

For AR4JA, these parameters are the following:

- Desired rate: selection among R12, R23 and R45
- Transfer Frame length (k). Valid values for CCSDS AR4JA LDPC codes are 1024, 4096 and 16384. All members of the code share the same mathematical description and consequently the encoder could operate efficiently for all the specified block lengths. For practical reasons however having to do with software synthesis, implementation and simulation runtimes, only 1024-bits block length is used for this implementaion. The only requirement for addition of the other members of AR4JA family is the addition of the corresponding VHDL files to describe the function generators (first rows of circulants) of these members. The MATLAB scripts to produce these matrices are provided, along with the matrices themselves. For k=16384 however, it was not possible to even execute them due to high computer memory requirements. A limited simulation of the encoder operation for k=4096 is however included in the accompanying code.
- Circulants size (m). It is defined in the specification but a small memo is also provided in-line with the code.
- Parallelism parameters L_a, L_m. According to previous chapter, they also define the width of the encoder's interfaces. For the purposes of examining the impact each parameter has on the encoder's performance features, all combinations resulting inL_a, L_m=16 and 32 where used and extensively simulated.
- Randomization option. Set to true if the highly recommended randomization is selected.

Based on the selections made on these parameters, the encoder's top-level entity selects the suitable components for the specific implementation. This is especially important for the function generator entities, which are different for each member of the family, as well as for the different versions of the FSMs of the control and buffer unit detailed in §3.2.

For C2 code, there are no modifiable parameters in the package file, other than the randomization option. Code parameters are constant and the control unit is designed specifically for L_m=16.

4.2 Core synthesis

The VHDL code was synthesized targeting a Xilinx Virtex 5 XC5VLX110T FPGA on Xilinx software for each configuration. Synthesis should run without problem, provided that the selected parameters defined in package files are valid. Block RAM resources were excluded using suitable synthesis options in order that code could be portable to different FPGA vendors.

4.3 Performance

The encoders synthesized previously are implemented on the FPGA, using a constraints file specifying a target clock speed. Performance and resource utilization and maximum speed are extracted from post place and route reports and listed in table 6 for all the combinations of L_a and L_m giving 16 as the product. Code rate ½ and k=1024 is considered and minimum run-time was set as the design strategy.

For AR4JA codes, important results can be deduced concerning the selection of parameters L_a and L_m and the impact the have on performance and resources budget of the encoder, for a constant total parallelism product of L_a, and L_m. As L_a parameter increases, more latency is introduced. Because of the pipelined operation however, there is no practical impact on the throughput of the encoder. From the results on the table, it is apparent that the reduced complexity of the function generators for higher values of L_a parameter is reflected on the diminishing LUT utilization. On the other hand, higher latency is introduced and the demand for larger memory structures of the control and buffer unit places higher demands on slice registers. The reduce complexity of the function generators also leads to better timing performance, which can considerably increase throughput.

For L_a=1 case for both codes (AR4JA and C2), the only source of latency is from the input and output buffers.

The critical path in almost all cases was through branches of the PRCE tree, starting either from the memory structures (*s_feed* signal on fig. 21) or the FSM state logic to define the value of function generators (*row* signal on same figure) and towards the parity registers.

Note that the parameters leading to the employment of the high latency case control unit appear to result in sub-optimal in terms of resource utilization and speed. It is possible that the extra logic necessary to handle the co-existence of two TFs at the same time in the control and buffer unit may significantly burden the design.

The control unit without HALT state described in previous chapter is not used in this case, since targets other cases (L_aL_m=32).

Table 6: Resouces and speed

	Parameters	**Slice Regs**	**Slice LUTs**	**Occ. Slices**	**Avg. fanout**	**Max. Freq. (MHz)**	**Systematic Latency (cycles)**	**Control unit used[1]**
AR4JA 1k r=1/2	*La=1, Lm=16*	1120	7844	2252	5,77	155,84	2	CU LA1
	La=2, Lm=8	1657	5164	1496	4,98	180,96	10	GENERAL
	La=4, Lm=4	2173	3250	849	6,05	191,24	26	GENERAL
	La=8, Lm=2	2176	3258	849	4,43	217,16	58	GENERAL
AR4JA 1k r=2/3	*La=1, Lm=16*	623	4911	1357	5,61	160,95	2	CU LA1
	La=2, Lm=8	884	4785	1753	5,93	180,83	6	GENERAL
	La=4, Lm=4	1144	3808	1497	5,57	196,31	14	GENERAL
	La=8, Lm=2	1661	3134	870	5,97	241,25	30	GENERAL
	La=16, Lm=1	2696	3855	1009	6,51	211,19	62	NEXT IN SYS
AR4JA 1k r=4/5	*La=1, Lm=16*	359	2453	780	5,57	196,46	2	CU LA1
	La=2, Lm=8	495	3179	930	5,93	211,37	4	LAT.EQ. ASM
	La=4, Lm=4	632	2625	728	6,09	251,70	8	GENERAL
	La=8, Lm=2	888	2386	644	5,70	271,89	16	GENERAL
	La=16, Lm=1	1409	2583	696	6,19	241,96	32	NEXT IN SYS
C2	*La=1, Lm=16*	1159	9789	2998	5,62	190,91	2	C2

1. Control unit used based on the configuration:

CU LA 1: La=1 case, GENERAL: default unit, NEXT IN SYS: high latency case when next TF arrives during systematic output of the current, LAT. EQ. ASM.: unit used when latency is equal to the number of cycles necessary for ASM sequence output. C2: unit for C2 code

5. VERIFICATION AND VALIDATION

5.1 General

The first step towards verification of the core is the formulation if the requirements which the encoder should be able to satisfy.

The requirements therefore for the encoders are the following:

- Produced CADUs should comply with the CCSDS definitions.
- Input and output interfaces should comply with the AMBA AXI-4 Stream protocol for transmission of an infinite continuous aligned stream.
- The number of idle cycles on output (master) interface should be minimal. As already described in §3.2, for all cases of AR4JA codes there are no idle cycles at all. For C2 code however, just one idle clock cycle per CADU is introduced between the systematic and parity output for reasons detailed in §3.2.6.

The verification plan for this work includes the following two actions:

- Validation through functional simulation of the HDL description compliance to the previously cited requirements are met. Code coverage metrics are expected to report 100% coverage in all cases and for all types of coverage. This action is accomplished by a suitable testbench and a tcl script which automates the entire procedure.
- Validation of the correct operation of the final implemented netlist. The encoder will be embedded into a suitable system which is then going to be implemented in the actual hardware. The purpose of the integrated system is to provide the necessary stimuli and record the encoder's responses, all these while the encoder operates at the specified clock frequency. Encoder's responses will be examined to verify correct operation on the actual hardware.

Details of these two actions are elaborated in following paragraphs. The results provide strong evidence that the encoder satisfies the requirements.

5.2 Functional simulation

For the functional simulation of the code, stimulus data are needed. Two MATLAB scripts (one for each code family) generate text files with the hexadecimal representation of a number of TFs, making use of MATLAB's *rnd* function. The MATLAB scripts also generate the expected CADUs which the encoders should produce with the specified input vectors.

Simulation is executed in Mentor Graphics Modelsim by a calling tcl script, which performs the following operations:

i. Compiles the necessary sources for the particular configuration selected in definitions package. For C2 code this is only the inclusion/exclusion of the randomizer entity but for AR4JA it has to select also a number of other files, like the suitable control and buffer unit corresponding to the selected L_a and L_m parameters or the output multiplexor, excluded for L_a=1.
ii. Selects and compiles the suitable testbench among a range of options corresponding to the different control units described in §3.2 and executes it with coverage option on. Coverage data and simulator's console output are logged in a suitable file. As explained later in this paragraph, during its execution, the

testbench records the encoder's responses to the applied test vectors into suitable files.

iii. Saves coverage database and generates coverage (text) report.

iv. Compares the files generated by the testbench and contain the encoder's responses with the expected values calculated in advance with MATLAB. If any differences are found, an error message is displayed on the console.

For AR4JA codes, the above process needs to be repeated for a number of L_a-L_m combinations over different family members. Consequently, the above steps are included in a tcl macro, which is then called for each parameters combination that needs to be verified. For C2 code of course, the parameters are static and there is no need for a macro. The provided tcl scripts are documented extensively. In all cases, simulation logs are automatically saved in corresponding text files.

Simulation for k=4096 is provided from a separate location, using a different VHDL file for the encoder and uses a significantly smaller test dataset, not always leading to code coverage. The reason for this is the prodigious amount of simulation time that would be required if the two block lengths were simulated with the same number of TFs. This however does not compromise the validity of the simulation results, since code family members across different block lengths share exactly the same mathematical description.

5.2.1 Testbench description

The testbench has to apply the stimuli created in advance to the Unit Under Test (UUT), while at the same time do this in such way that:

- Interface protocol operation (AXI4-Stream) is verified.
- 100% code coverage is ensured.
- Optimality of the encoder as described in the requirements formulation is verified as well.

In addition, UUT responses are recorded and written in a file and coverage data are collected.

Based on the above requirements, the testbench comprises the following parts:

i. Instantiate and initialize UUT.

ii. Full throttle operation validation. The testbench validates that the encoder exhibits optimal operation, i.e. there are no idle cycles (AR4JA) or one on output interface. It provides TF data to the encoder at the documented rate, without waiting for TREADY_SL.

iii. Protocol validation. TREADY_MA and TVALID_SL are de-asserted at random instances controlled by a pseudo-random VHDL function (uniform). This simulates the possibly bursty behavior of the transmitting and receiving units. As the simulation time elapses, the probability that TVALID_SL or TREADY_MA increases, so that the control unit's FSM traverses all states and transitions.

iv. Any other necessary steps to ensure 100% code coverage by the testbench, according to specific needs of the utilized control unit (§3.2). One such example is the correct reinitialization of the FSM following a reset signal from all the FSM states. Another case is the possibility that the FIFO of the control and buffer unit empties during the systematic output. This would require a very long simulation runtime if the testbench resorted solely to the method of the previous step, but can be easily reached using signal spy library to monitor related signals and insert the necessary TVALID_SL=0 cycles on slave interface.

As the simulation progresses, a special process in the testbench records UUT's responces in a suitable file, which is going to be examined by the calling tcl script after the end of the simulation.

All testbenches are fully documented and the details of their operation not covered at this paragraph can be easily tracked with the help of in-line comments.

5.2.2 Simulation results

For AR4JA codes, a number of 5000 TFs were used as stimulation data for k=1024. Simulation was successful for all valid combinations of L_a and L_m parameters for data bus width equal to 16 and 32 and corresponding coverage reports indicated that 100% coverage was accomplished for all types of coverage. The only exception is the case of L_a=32, L_m=1, where the latency is so high that the it covers entirely the number of cycles for the input of the next TF and extends over the second subsequent TF, making this choice impractical.

Due to the longer simulation runtime by reason of the very high block length, C2 code simulation is performed with a significantly smaller test data set (1000 TFs instead of 5000). Results are coverage are equally successful.

5.3 Implementation validation

The implemented design is integrated in an embedded system to verify the encoder's operation in real time and on the actual hardware. All hardware tests described in this paragraph were performed on a XUPV5-LX110T development system.

The embedded system used for the test should provide for the following:

- Necessary hardware for generation or external input of test vectors.
- Necessary hardware for recording the UUT responses.
- Necessary hardware for display of results
- Control and time-scheduling over the test process.

The tests performed at this stage use two sources of test vectors:

- Input TFs are inserted by the board's UART.
- Pseudo-random TFs are generated by a LFSR.

UUT's responses are compacted by a Multiple-Input Shift-Register (MISR). The MISR employed is based on a 64-bit Fibonacci LFSR using the primitive polynomial: $h(x)=x^4+x^3+x+1$.

Compacted output (i.e. the MISR signature) needs to be provided in human readable form. On completion of the test, specially designed hardware converts the MISR binary value into its ASCII representation in hexadecimal form and returns it to the operator of the test through the board's serial port. Another output to the operator's console is the number of clock cycles needed to encode the provided TFs as well as the number of these Tfs.

The above procedure is controlled by a specialized entity's FSM. This control unit defines the initiation of the test, based on operator's input, detects the end of the test and reports results back to the operator.

The two versions of the implementation test for the two types of TF input are subsequently described. In both cases, the embedded systems were designed around a data bus for the encoder which is equal to 16-bits, meaning that for AR4JA encoders, only those with L_aL_m=16 can be tested by this design.

5.3.1 Embedded system description: UART input

For the first of the two tests, the input to the encoder comes from the operator's console through the development board's UART. Following their input, the TFs are temporarily stored in a FIFO, large enough to accommodate the desired number of them. The increased size of this FIFO dictates that it should be implemented in Block RAM resources on the FPGA. When the slow process of uploading test vectors to the FIFO completes, the encoding process is automatically initiated. Detection of the completion of uploading procedure is done by the integrated circuit's control unit which counts incoming TF data, up to the point where they reached a standard fixed number. The encoder core responses are routed to the MISR for signature compaction and display. When the control unit detects that the encoder has finished encoding all the TFs, it initiates the results display procedure.

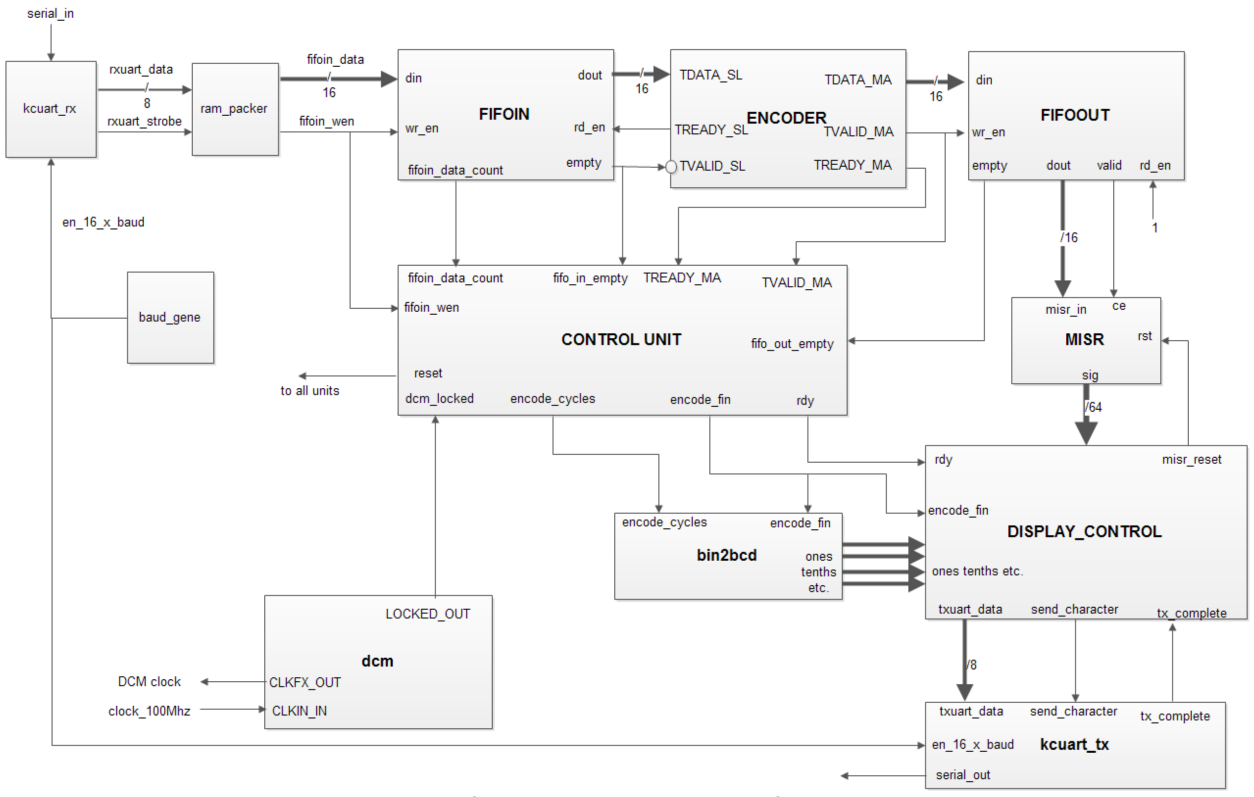
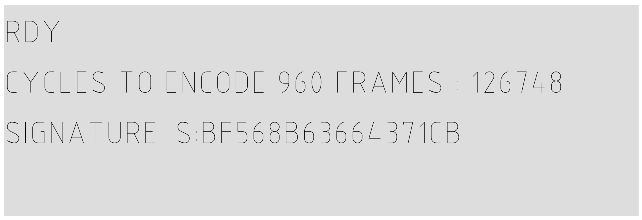


Figure 34: Block diagram of the embedded system for the test with UART input.

The block diagram of the circuit created for the test is displayed in fig.34. An operator's terminal is assumed to be connected to the corresponding *serial_in* and *serial_out* ports. After initialization, a RDY message is provided on it to indicate that the system is ready to start receiving data from the console. Data are received as binary with the help of a UART generated with the corresponding macro used for Picoblaze applications. A packer unit packs concatenates two incoming input bytes into a 16-bit word which is stored in the FIFO. FIFO includes a counter to keep track of the amount of incoming data (*fifo_datain_count*). When this number reaches a fixed number (960 TFs here), hardwired into the control unit's design as a constant, the control unit asserts TREADY_MA signal and the encoder begins the processing of data in the input FIFO. Generated responses are written to the output FIFO, which is in turn connected to the MISR. The control unit keeps track of the clock cycles elapsing from the moment when encoding begins in a special counter.

When both input and output FIFOs are empty, the control unit asserts a signal (encode_fin) which triggers the display of result data on the connected operator terminal. The display data include the number of cycles necessary for the encoding and the hexadecimal representation of the MISR value. The control unit's cycles counter value is converted to Binary Coded Decimal (BCD) and analyzed to decimal divisions (units, tenths, hundredths etc). A specialized display control unit has the role of organizing the information into a user-friendly human readable form. A sample of the generated output is displayed on fig. 35.

The desired frequency of operation for the entire system is generated by a DCM unit. *LOCKED_OUT* output of the DCM triggers the control unit to transit to a state where RDY message is displayed and the test execution can begin. The DCM frequency can be set to the maximum achievable value and this is going to be the claimed operation frequency of the encoder.



```
RDY
CYCLES TO ENCODE 960 FRAMES : 126748
SIGNATURE IS:BF568B63664371CB
```

Figure 35: Sample terminal output

The number of cycles depends on the LDPC code and the latency introduced by input-output FIFOs and the system's control unit logic. The calculation of the expected value of the MISR can be performed using a provided (with accompanying code data) testbench, in which the UUT is the MISR and the expected CADUs (calculated in MATLAB) are applied to it. The final claimed encoder performance takes FIFO delays into account, so that the reported performance is the actual being experienced when the encoders of this work have been incorporated in a real embedded system.

In all cases for all encoders, the integrated system was simulated before implementation in hardware. Corresponding simulation testbenches and input sources are provided in accompanying code files.

The entities displayed on fig. 34 contain a significant number of details. This is especially true for the two control units (main and display). These details include a lot of information pertaining the operation of the FSMs of the components their constituent entities etc, the thorough description of which would distract the current document from its main subject. Their VHDL code however is documented.

5.3.2 Results with UART input

For AR4JA codes, the configurations selected for the test where those with the best timing performance for each family member (see Table 6).

Tests were performed with 960 input TFs provided with the accompanying documentation and they were all successful. The maximum operation frequency of the encoder achieved in each case is displayed on table 7, where the MISR signature value is that which corresponds to the input data provided with the accompanying code.

Test results are listed on table 7 below. Implementation options were tuned to optimize timing and the maximum frequency was reached through successive implementations in ISE design suite.

Due to the significantly higher TF length for C2 codes, the test was run over 127 of them and was also successful.

Table 7: Implementation test results with UART input data (AR4JA:960, C2:127 TFs)

	Parameters	MISR value	No. of cycles	DCM frequency (MHz)
AR4JA r=1/2	*La=8, Lm=2*	BF568B63664371CB	126780	230
AR4JA r=2/3	*La=8, Lm=2*	8C51CA585AF9203D	96032	240
AR4JA r=4/5	*La=8, Lm=2*	6C13FC1220D91C53	080658	250
C2	*La=1, Lm=16*	37EFE1DBCB5F86B2	65157	200

5.3.3 Embedded system description: LFSR input

The second of the two implementation tests generates test vectors from a LFSR, instead of receiving them from an external source. The advantage of this method is the significantly higher number of TFs that the test can use, since it is not limited to the amount of FPGA Block RAM resources. The block diagram of the embedded system is displayed on fig. 36.

The input UART and the packer unit have been replaced by the LFSR, which fills up the input FIFO with TFs after initialization of the FPGA. The operator receives a RDY message on the connected terminal and initiates the test by pressing a properly debounced push-button on the development board. The next steps are similar to the previous case with externally provided TF data.

The LFSR operation is simulated in software to calculate the correct value of the MISR signature at the end of the test. A suitable testbench records LFSR output sequence in corresponding files and the recorded data are applied to the MISR using the testbench of the previous paragraph.

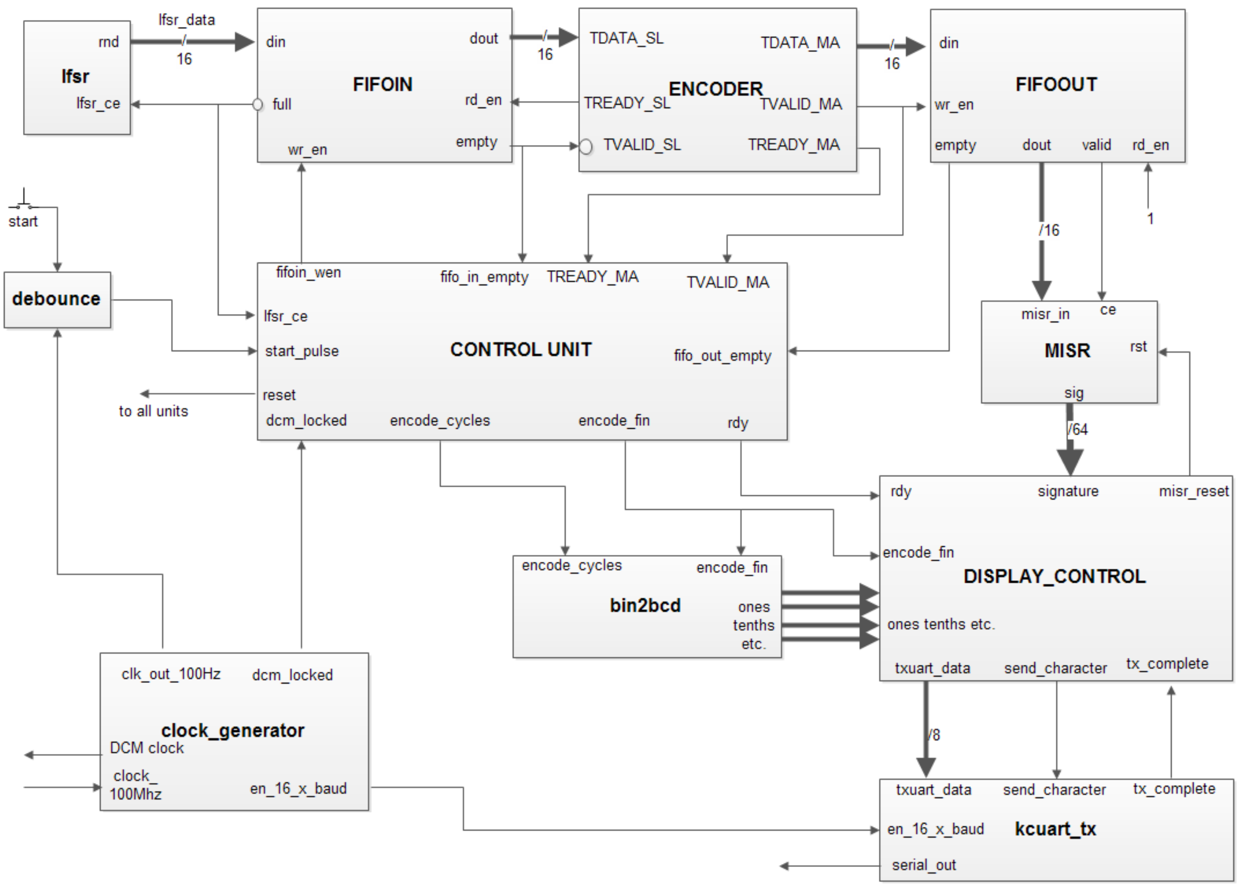


Figure 36: Block diagram of the embedded system for the test with LFSR generated data.

5.3.4 Results with LFSR input

Comparable results were received to the previous case with UART input of test data, albeit maximum operation frequencies were smaller than those achieved with serial input of TFs, mainly because of the debouncing circuit for test initiation. Correspondingly to Table 7, the results are displayed on Table 8.A number of 5000 TFs was used for AR4JA codes and 1000 for C2.

Table 8: Implementation test results with LFSR generated input data (5000 TFs)

	Parameters	MISR value	No. of cycles	DCM frequency (MHz)
AR4JA r=1/2	*La=8, Lm=2*	5152CC221167879B	660059	220
AR4JA r=2/3	*La=8, Lm=2*	F38F5B29AFF499A2	500031	240
AR4JA r=4/5	*La=16, Lm=1*	10A15E1FD603048D	420032	240
C2	*La=1, Lm=16*	D32DFF71836DB4F6	525317	180

6. RESULTS

Implementation results for the fastest configurations are summarized in Table 9, in which the claimed encoding speed is the speed demonstrated by the implementation test, taking into account the latency introduced by input and output FIFO and is the actual speed expected to be experienced from channel coding in real applications. The formula of calculation of the speed is:
*SPEED = Number of CADUs x CADU size / Encode cylces * Clock speed*

Results on this table should be interpreted in conjunction with Table 6 for a concrete viewpoint on the performance characteristics of the encoders in the current work. Performance for other members of AR4JA family is expected to exhibit similar behavior, in regard to the effect of increasing L_a parallelism.

Table 9: Summary of demonstrated performance characteristics of the fastest implementations

	Parameters	**Register utilization**	**LUT utilization**	**Slice utilization**	**Claimed encoding speed (Gbps)**	**Latency (ns)**
AR4JA r=1/2	*La=8, Lm=2*	3,148%	4,714%	4,913%	3,678	252,17
AR4JA r=2/3	*La=8, Lm=2*	2,4%	4,53%	870	3,839	125,01
AR4JA r=4/5	*La=8, Lm=2*	1,28%	3,4%	644	3,999	64
C2	*La=1, Lm=16*	1,67%	14,16%	3165	3,056	10

6.1 Comparison to other implementations: commercial products.

The implemented encoders' performance is compared to existing solutions in this paragraph, according to available parameters in corresponding published product briefs. Such solutions are currently available only for C2 code.

CREONIC GmbH has made available an encoder core for the (8160,7154) C2 code [26]. According to the information provided in the product brief, coding throughput at 200MHz operation is 1,6Gbps, while encoding latency for the same clock speed is 40ns. Compared to the C2 encoder of the present work, throughput at the specified clock speed is almost the half (3,193 vs 1,6 Gbps). ASM output and randomization are not implemented.

Small World Communications has created a core also for CCSDS C2 code (LCE01C). Product specification sheet [27] states that encoding rate can reach up to 1,75 Gbps at a clock speed up to 250MHz on a XC6VLX75T–3 FPGA. Data buses are 8-bit and it consumed 9.2 K Virtex-5 LUTs. Encoding is initiated by a start pulse and it employes double function generators for the coefficients of the generator matrix: one for normal operation and one at the boundaries of the 511-bit wide circulants. ASM sequence is not generated in this case and extra logic is necessary for the interface of the core with a RAM unit and randomization possibility.

Iprium also sells an encoder/decoder IP core for C2 code [28]. The core is designed for serial input/output of one bit at a time. For encoding to start, a special signal needs to be asserted at the beginning of each TF.

In addition, the patented design in [25] has already been reviewed in §3.2.7 and proven to be susceptible to criticism for inefficient use of the 21-bit MAC module during idle cycles. Although F/F count is lower than current encoder, logic resources required, expressed in gate count since the encoder was implemented in ASIC, are significant.

Above results are summarized in table 10. While encoding performance is inferior in all cases, none of the alternative encoders provides a standardized interface for input of TFs and output of CADUs, nor randomization is an option. If the absence of ASM sequence output is added, the presented encoder of this work is the only complete core for generation of CADUs according to the standard.

Table 10: Comparison of Implementations for C2 code

	Bus width	**Resources**	**Max. clock freq. (MHz)**	**Enc. Speed (Gbps)**	**Latency (cycles)**	**Randomi zation**	**ASM**
CREONIC	*N/A*	N/A	200	1,6	8	NO	NO
LCE01C	*8*	9,2K LUTs (Virtex5)	250 (XC6VLX75T–3)	1,75	3	NO	NO
IPrium	*1*	290 Slices (Virtex 6)	418	0,418	N/A	NO	NO
[25]	*16*	1492 F/Fs 30680 gates (ASIC)	128	2	9	NO	NO
This work	*16*	9,8K LUTs 7412 F/Fs 3165 Slices (Virtex5)	200	3,19	2	YES	YES

6.2 Comparison to other implementations: literature.

There is significant scientific interest on the development of efficient LDPC encoders. As already mentioned, interesting results can be found in [14], [15], [16], [17], [18] and [19]. All of these solutions are targeted to LDPC codes which are characterized for encoding efficiency, which is not the case for the codes of this work. A comparison

however can be made to the extent applicable and all cited implementations are going to be commented-out and compared to the current.
At first, none of the cited encoders provide CCSDS ASM framing and randomization functions. Furthermore, none of the encoders cited above includes flow control so that the core can handle streams of input-output data. Apart from [19] and this work, all other implementations are iterative encoders and write output parity data in output memory elements. In most cases, optimizations in the code design described in §1.3.4 make feasible the calculation of all the parity bits at once.
As already mentioned, [14] and [15] are based on Richardson-Urbanke encoding algorithm, which is not efficient for CCSDS codes. In both cases the entire information block needs to be available for the parallel parity calculation process, adding thus an amount of latent cycles in case of a stream-oriented encoder, which in the case of [14] is expected to be significant. Especially the impressive performance encoder of [15] is designed for a particular class of encoding-friendly codes (B-LDPC), adopted for IEEE 802.11 and 802.16 standards and a direct comparison with the present encoders cannot be made. In addition, [15] only implements the parity bits generation, without taking care for the concatenation with systematic input bits or the serialized process to write them into a memory.
The work in [16] is about a 7,7 Gbps encoder for IEEE 802.11ac QC-LDPC codes. It is suitable for streaming operation, since the entire information block is not necessary for the encoding process to begin and output of calculated parity data seems to be serialized from output buffer. The algorithm calculates parity bits directly from the (sparse) parity-check matrix according to the procedure described in §1.3.4.
Similarly, [17] is another example of high throughput encoder for LDPC codes designed for encoding efficiency, like those used in DVB-S2 standard, for which the specified encoder can reach a throughput up to 29Gbps. In this case also, the claimed throughput refers only to the parity bits generation and does not take into account serialization of input data and the number of cycles needed to store the input vector into memory. The proposed encoding algorithm bears significant resemblance to [16] and calculates the parity bits directly from the (sparse) parity-check matrix. The extremely simple structure of H_2^{-1} matrix in this case simplifies the multiplication of that matrix with a the result of H_1m^T(following notation of §1.3.4) into a recursive XOR operation.
Reference [18] describes an efficient encoding implementation, also for IEEE 802.16, which again takes advantage of the special structure of the parity-check matrix of the code to calculate the parity bits directly from the (sparse) parity-check matrix, using back-substitution and consequently requiring the parity-check matrix to be in lower-triangular form. The multi-Gbps claimed performance refers to the internal encoding core operation when all information block bits are available and is limited to 422 Mbps when serialization of incoming data needs to be taken into account.
The encoder architecture in [19] follows a different approach in that, like the encoders of the current work, it calculates parity bits from the generator (G) instead of the parity-check matrix (P) and consequently can be generalized for the CCSDS LDPC codes. The entire information block needs however to have already been accumulated in order that the parity calculations can begin. Moreover, it introduces a very large critical path in the XOR operations which add the results of the information block sequence to the corresponding column of the parity-check matrix (for the calculation of one parity bit). This approach is not expected to scale well for increased block lengths-other than IEEE 802.16 LDPC family block lengths, especially the 16384 of AR4JA or 8160 of C2. Finally, even for the simple case of the maximum information block of the WiMax LDPC codes, the memory requirements are prodigious, at the same time reaching a maximum performance of 360 Mbps.

Although the direct comparison of the cited implementations with the current work cannot be made because of the code characteristics, performance and resource utilization data are however summarized in Table 11. From the solutions available for implementation of CCSDS codes, it is evident that proposed encoders

Table 11: Comparison of various LDPC Encoder Implementations

	[14]	**[15]**	**[16]**	**[17]**	**[18]**	**[19]**	**This work[3]**
Demonstrated Application Field	*General*	*802.11n*	*802.11ac*	*DVB-S2* *DVB-T2*	*802.16e*	*802.16e*	*CCSDS*
Codeword length	*2000*	*1944*	*1944*	*64800*	*1920*	*2304*	*2048*
Rate	*1/2*	*5/6*	*1/2*	*5/6*	*1/2*	*1/2*	*1/2*
Resources/ technology	*870 Slices* *19 Block RAMs* *Virtex 2*	*1782 LUTs* *2187 F/Fs* *Virtex5*	*96K equiv. Gates* *ASIC 130nm CMOS*	*32734 LE* *126,6k F/Fs* *STRATIX-2*	*8924 LEs* *STRATIX*	*11430 LEs* *3,9M F/Fs* *STRATIX*	*849 Slices* *2176 F/Fs* *Virtex 5*
Algorith[1]	*RU*	*RU*	*BS*	*BS*	*BS*	*Direct*	*Direct*
Clock speed	*143MHz*	*290MHz*	*100MHz*	*320MHz*	*149 MHc*	*60MHz*	*230MHz*
Claimed throughput	*44Mbps[2]*	*117,45 Gbps[2]*	*7,7 Gbps*	*29Gbps[2]*	*3,32 Gbps[2]* *422 Mbps (serialized)*	*119,7 Mbps*	*3,19 Gbps* *(stream)*
Applicable to CCSDS codes	*YES* *(with different results)*	*NO*	*NO*	*NO*	*NO*	*YES*	*YES*
Special Notes	*1. RU: Richardson-Urbanke, BS: Back-substitution, Direct: multiplication with Generator matrix* *2. Not serialized output* *3. Data in parenthesis refer to highest supported code rate (¾)*						

7. CONCLUSIONS

LDPC codes were initially considered impossible to implement. Advances in VLSI technology however have entirely reverted this image and this work is towards this direction. The encoders implemented in this thesis occupy only a small percentage of the area of XCV5-LX110T FPGA, while at the same time reaching mutli-Gbps encoding performance.

It has been shown that it is possible to improve the timing performance and resource utilization of the standard RCE-based encoders by processing incoming information bits corresponding to multiple circulants at the same time. The price however that has to be paid in this case is increased latency. Due to the pipelined operation of the control unit however, this latency is not translated into performance degradation, since almost no idle cycles exist on the output interface. The only exception is the encoder for the C2 code, for which only one idle cycle per CADU is necessary, that is only one cycle in 514 is wasted.

For AR4JA encoders, the provided VHDL code provides a description that is the same across all members of the AR4JA family and the parameters are centrally defined in one package file, simplifying the selection of the configuration which meets the performance-latency target and also being able to adapt to different information block sizes (parameter k of the code) and bus sizes up to 64 bits.

The encoder for C2 code uses fixed 16-bit buses for input and output and introduces no additional latency, other than for its input-output buffering. Through a suitable selection of circulant rows and an advanced control unit design, it manages to align the input data arriving in packets of 16 bits on input interface to the boundaries of the 511-bit circulants of the generator matrix, without wasting resources.

All encoders interface to AMBA AXI4-Stream buses, providing thus a solution which can be readily incorporated in a SoC design and enabling the encoders of this work to be characterized as complete practical cores.

Compared to the existing solutions in literature and in market, the proposed encoders reach unprecedented performance for the specified code family, while at the same time keeping resource utilization at a minimum.

ACRONYMS-ABBREVIATED TERMS

AOM	Advanced Orbiting Systems
AR4JA	Accumulate-Repeat 4-Jagged Accumulate
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BCD	Binary Coded Decimal
BSC	Binary Symmetric Channel
CADU	Channel Access Data Unit
CCSDS	Consultative Committee for Space Data Systems
DCM	Digital Clock Manager
DVB	Digital Video Broadcast
FER	Frame Error Rate
F/F	Flip Flop
FSM	Finite State Machine
Gbps	Gigabits per second
LDPC	Low-Density Parity-Check
LFSR	Linear Feedback Shift Register
PRCE	Parallel Recursive Convolutional Encoder
QC	Quasi-Cyclic
RCE	Recursive Convolutional Encoder
SDLP	Space Data-Link Protocol
SNR	Signal to Noise Ratio
TF	Transfer Frame
TM	TeleMetry
UUT	Unit Under Test

REFERENCES

[1] C. E. Shannon, “A mathematical theory of communication,” Bell Systems Tech. J., vol. 27, pp. 379–423, 623–656, Jul., Oct. 1948.

[2] *TM Synchronization and Channel Coding-Summary of Concepts and Rationale*, CCSDS 130.1-G-2 Green book, Nov. 2012.

[3] S. Dolinar, D. Divsalar, and F. Pollara, “Code Performance as a Function of Block Size,” IPN Progress Report 42-133, JPL, May 1998.

[4] R. G. Gallager, “Low density parity-check codes,” *IRE Trans. Information Theory*, vol. 8, no. 1,pp. 21–28, Jan. 1962.

[5] W. E. Ryan and S. Lin, “Low-Density Parity-Check Codes” in *Channel Codes Classical and Modern*, ed. Bew York, Cambridge University Press, 2009, pp. 202

[6] R. M. Tanner, “A recursive approach to low complexity codes,” *IEEE Trans. Inf. Theory*, vol. IT-27, no. 5, pp. 533–547, Sep. 1981.

[7] F. Kschischang, B. Frey, and H.-A. Loeliger, “Factor Graphs and the Sum-Product Algorithm,” *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 498–519, Feb. 2001.

[8] J. Thrope, “Low-Density Parity-Check (LDPC) Codes Constructed from Protographs,” IPN Progress Report 42-154, JPL, Aug. 2003.

[9] K. Andrews *et. al.*, “Design of Low-Density Parity-Check (LDPC) Codes for Deep-Space Applications,” IPN Progress Report 42-159, JPL, Nov. 2004.

[10] Z. Li *et.al.*, “Efficient Encoding of Quasi-Cyclic Low-Density Parity-Check Codes,” *IEEE Trans. Commun.,* vol.54, no. 1, pp.71-81, Jan. 2006

[11] K. Andrews, S. Dolinar, and J. Thorpe, “Encoders for Block-Circulant LDPC Codes,” in *Proceedings IEEE International Symposium on Information Theory*, Adelaide, SA, pp. 2300–2304, Sept. 2005.

[12] T. Richardson and R. Urbanke, “Efficient Encoding of Low-Density Parity-Check Codes,” *IEEE Trans. Inf. Theory*, vol.47, no. 1, pp. 638–656, Feb. 2001

[13] J. Perez, K. Andrews, “Low-Density Parity-Check Code Design Techniques to Simplify Encoding,” IPN Progress Report 42-171, JPL, Nov. 2007.

[14] D.U. Lee, W. Luk, “A Flexible Hardware Encoder for Low Den-sity Parity Check Codes,” 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 101–111, April 2004.

[15] G. Tzimpragos *et.al.,* “A Low-Complexity Implementation of QC-LDPC Encoder in Reconfigurable Logic” in *23rd. Int. Conf. on Field Programmable Logic and Applications*, Porto, 2013, pp. 1-4.

[16] Y. Jung *et.al.,* “7.7 Gbps Encoder Design for IEEE 802.11ac QC-LDPC Codes,” *Journal of Semiconductor Technology and Science,* vol.14, no.4, pp.419-426, Aug. 2014.

[17] Al Hariri *et.al.,* “A High Throughput Configurable ParallelEncoder Architecture for Quasi-Cyclic Low-ensity Parity-Check Codes, In 19th Int. On-Line Testing Symp., Chania, Jul. 2013.

[18] R. Kopparthi and D. Bruenbacher. “Implementation of a Flexible Encoder for Structured Low-Density Parity-Check Codes”. In IEEE Pacific Rim Conf. Communications, Computers and Signal Processing.

[19] H. Yasotharan, A. Chan Carusone, “A Flexible Hardware Encoder for Systematic Low-Density Parity-Check Codes,” in IEEE Int.Midwest Symp. Circuits and Systems (MWSCAS’09), 2009.

[20] *Overview of Space Communications Protocols*, CCSDS 130.0-G-3 Green book, Jul. 2014.

[21] W. Fong, “White Paper for Low Density Parity Check (LDPC) Codes for CCSDS Channel Coding Blue Book.” CCSDS P1B Channel Coding Meeting, Houston, TX, Oct. 2002; https://standards.nasa.gov/documents/viewdoc/3315856/3315856. [Accessed 10/8/15].

[22] *TM Synchronization and Channel Coding*, CCSDS Standard 131.0-B-2 Blue book, Aug. 2011.

[23] *Low Density Parity Check Codes for Use in Near-Earth and Deep Space Applications*, CCSDS Experimental Specification 131.1-O-2, Sep. 2011.

[24] AMBA 4 AXI3-Stream Protocol v.1.0, ARM Specification, 2010.

[25] L. Miles and S. Whitaker, “Low-density parity-check (LDPC) encoder,” U.S. Patent 3,754,212 B2, Jun. 2, 2009.

[26] *CCSDS (8160,7136) LDPC Encoder and Decoder Product Brief*, CREONIC GmbH Product Brief; http://www.creonic.com/images/product_briefs/PB_Creonic_CCSDS_LDPC_FEC_IP.pdf [Accessed 3/9/2015].

[27] *LCE01C CCSDS (8160,7136) LDPC Encoder,* Small World Communications Product Specification, Mar. 2013; http://www.sworld.com.au/pub/lce01c.pdf [Accessed 3/9/2015].

[28] *LDPC NASA Encoder/Decoder IP Core*, Iprium Specification, r1091, Sept. 2014.