Received: 4 March 2020 | Revised: 30 July 2020 | Accepted: 1 September 2020

IET Computers & Digital Techniques

DOI: 10.1049/cdt2.12009

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH PAPER

RTL development of a parameterizable Reed–Solomon Codec

Mateus G. Silva | **Guilherme L. Silvano** | **Ricardo O. Duarte**

Graduate Program in Electrical Engineering, Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais, Brazil

Correspondence

Ricardo O. Duarte, Graduate Program in Electrical Engineering, Universidade Federal de Minas Gerais, Av. Antonio Carlos 6627, 31270-901, Belo Horizonte, MG, Brazil.
Email: ricardoduarte@ufmg.br

Funding information

CNPq, CAPES, and FAPEMIG

Abstract

Error correction coding (ECC) methods have been considered essential constituents of data transmission systems. Reed–Solomon (RS) codes are a core ECC technique that have been adopted in numerous applications and standards. Several register-transfer level (RTL) architectures for RS codecs have been proposed to address specific demands and overcome scalability challenges in speed and area. However, the influence of the main RS codec parameters on the corresponding hardware design has been undervalued by literature. The authors propose an open access intellectual property (IP) of a parameterizable RS codec and explore key aspects of its RTL development using IEEE 802.15.7 standard as illustration. Herein, it is demonstrated that formal verification has the potential to be solely used to attest the correctness of the developed IP for the RS codec configurations specified by IEEE 802.15.7. Furthermore, synthesis reports for the target field-programmable gate array devices indicate that the proposed IP is able to cope with throughput requirements in IEEE 802.15.7.

1 | INTRODUCTION

In modern communications and storage systems, the demand for solutions that reconcile high data exchange rates with reliability constraints using an economic viable hardware has been increased to fulfil the latest technological needs. Error correction coding (ECC) methods play an important role to achieve these opposite requirements. They add redundancy to the payload and restrict the characteristics of the encoded signal to augment the decoder's capability to extract the original data transmitted in unreliable and noisy communication channels [1]. Reed–Solomon (RS) codes are one of the most popular ECC methods that use a block-by-block basis to correct burst errors and erasures in data. The adequate compromise between effectiveness and implementation complexity guarantees the widespread use of RS codes in many applications until today.

The article entitled 'Polynomial Codes over Certain Finite Fields' [2] published in 1960 was the first formal publication of RS codes. At this time, a search for the most optimal and mechanizable ECC emerged after Claude Shannon's theory defined the upper bound of the channel capacity at which is possible to transmit error-free information using a proper coding method. Richard Hamming developed the first practical work on ECC algorithms using linear algebra in early 1950s, and Bose, Ray-Chaudhuri and Hocquenghem theorized his experiments in 1959—known as BCH codes. Finally, Irving Reed and Gustave Solomon introduced a special case of BCH codes (RS codes) that employs an efficient decoding algorithm which reduced implementation complexity of ECC systems [3].

Many technologies and standards have adopted RS codes with a variety of parametric configurations. The Compact Disc (CD) system was the first consumer mass application to employ it [4], and since then engineering solutions such as satellite, mobile systems, barcodes, and digital television have used RS codes. Literature presents many custom hardware implementations of RS codecs that encompass only the parametric configurations required by the related target application. Hardware Description Languages (HDL)—Verilog and VHDL—already provide ways to express parameterizable register-transfer level (RTL) architectures; however, previous works have not completely explored RTL solutions of RS codecs along those lines. Moreover, RTL verification of generic RS Codecs is challenging and requires a proper methodology to assure the correctness of the implementation. Most articles that cover RTL architectures of RS codecs only provide a limited validation of the implementation without analysing metrics and results that ensures verification quality.

As a novelty, this work devises a parameterizable, reliable and open-source IP core for RS codec available for academic and research community and applicable to communication hardware designs, as it has not been found any IP with such characteristics. For instance, many standards under the IEEE 802 refer to RS codecs with different parametric configurations. The functional behaviour of the proposed IP is attested by formal verification, which is an emerging technology that has become widely adopted to check RTL designs. Finally, the field-programmable gate array (FPGA) synthesis of the IP is analysed in terms of timing and resource utilization. To illustrate the RTL development process of the IP, an in-depth analysis is carried out for IEEE 802.15.7 (Short-Range Optical Wireless Communications) standard [5], which specifies RS codecs with adequate parametric variability and complexity to demonstrate the experimental results of this work.

Herein, Section 2 introduces theoretical aspects of RS codes, Section 3 analyses related works, Section 4 presents the proposed RTL architecture of RS codec, Section 5 examines and exhibits results of the adopted verification methodology, Section 6 explores FPGA synthesis aspects of the implemented IP and the paper conclusion is presented in Section 7.

2 | THEORETICAL ASPECTS

Irving S. Reed and Gustave Solomon briefly defined the principles of RS codes in few sentences:

> They are simply sets of algebraic curves defined by polynomials with a limited range of degrees. These curves when graphed are a set of discrete points—the abscissas and ordinates are values in a Galois field (GF). The degree limitation allows recovery of the complete curve even when the graph is assumed to be smudged and erased at many points. The relation of this idea to error control on noise-corrupted digital communication channels is immediate [6].

In a simplified way, it is possible to observe how RS codes work. It basically relies on 'fitting' points in a plane by the polynomial of smallest degree that passes through these points [7]. Basic polynomial mathematics affirms that 2 points can be fitted by a straight line, 3 points by a parabola, and so on. If some redundancy is added by oversampling a given polynomial function, the curve can be correctly estimated even though some recorded points move unintentionally. For instance, assume that a parabola, which can be represented by $k = 3$ points, is recorded in $n = 7$ points (Figure 1a). If 2 points are moved vertically, the original polynomial can still be recovered by finding a single parabola that fits as many as possible of the 7 points (Figure 1b). However, the original parabola may not be found anymore when 3 or more points are moved (Figure 1c).

(a) Parabola sampled in $n = 7$ points,
(b) Parabola recovered even though two points are erroneously moved,
(c) Wrong parabola recovery because maximum capacity of error correction was exceeded

If the number of moving points, represented by t, is greater than half the number of extra points, an incorrect polynomial function may fit the most number of points. Therefore, $t = (n - k)/2$ corresponds to the maximum capacity of error correction. RS codes are represented as a set of length n vectors (codewords) where their elements (symbols) consist of m binary digits (Figure 2). The original message takes k vector positions, then there are $n - k$ redundant symbols (parity). An RS codec configuration is usually referred by the notation RS(n, k).

RS codes are linear, every codeword comes from the sum of all other codewords, and cyclic, a shift rotated codeword of any number of symbols represents another codeword [8]. These two properties are guaranteed due to the use of GF as arithmetic framework for codeword encoding and decoding processes. It enables the application of Linear Feedback Shifter Registers (LFSR) for RS codec implementation, which has a convenient hardware realization [1]. GF defines an algebraic field with finite number of elements so that any arithmetic operation between them results in an element within the established restricted set. A primitive element α, which is root of a primitive polynomial ($p(x)$), is used as reference to generate the field elements. In the digital world, $\alpha = 2$ is chosen since it represents the binary base. For such scenario, the total number of elements in a GF is equal to 2^m. Table 1 is an example of GF(2^3) where $\alpha = 2$ is root of $p(x)$. Then, $\alpha^3 + \alpha + 1 = 0$, or equivalently $\alpha^3 = \alpha + 1$ (additions and subtractions are interchangeable in field arithmetic).

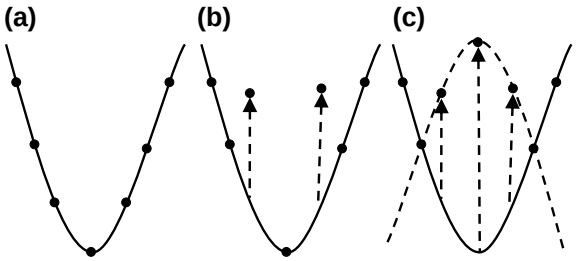


FIGURE 1 Parabola recovery using sampled points

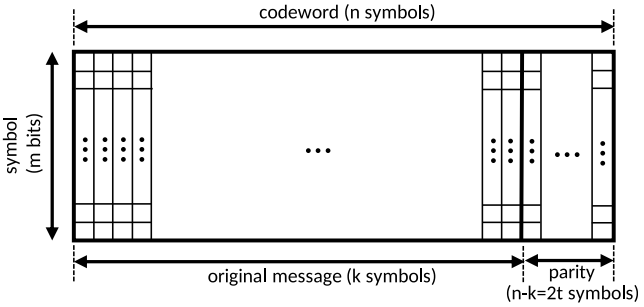


FIGURE 2 Generic parametric configuration of RS codes (figure adapted from Fig. 1s in [8])

TABLE 1 GF(2^3) representation using $p(x) = x^3 + x + 1$

Index form	Polynomial form	Decimal form
0	0	0
α^0	1	1
α^1	α	2
α^2	α^2	4
α^3	$\alpha + 1$	3
α^4	$\alpha^2 + \alpha$	6
α^5	$\alpha^2 + \alpha + 1$	7
α^6	$\alpha^2 + 1$	5

The original approach to generate RS codes published in [2] considers a sequence of symbol information, $\{i_{k-1}, i_{k-2}, \ldots, i_1, i_0\}$, to build the polynomial $I(x) = i_{k-1}x^{k-1} + i_{k-2}x^{k-2} + \cdots + i_1x + i_0$. If $n \leq \alpha^m$ elements from GF(α^m) are evaluated in $I(x)$, then the result is a system of n linear equations in k variables (1) [3]. This system can be solved for all $\binom{n}{k}$ combinations of equations. If the number of corrupted symbols $t \leq (n-k)/2$, the most occurring solution of $\binom{n}{k}$ possible linear systems corresponds the original codeword. This method clearly requires much computational resources to calculate all system solutions.

$$\begin{cases} I(0) = i_0 \\ I(\alpha) = i_{k-1}\alpha^{k-1} + \cdots + i_2\alpha^2 + i_1\alpha + i_0 \\ I(\alpha^2) = i_{k-1}\alpha^{2(k-1)} + \cdots + i_2\alpha^4 + i_1\alpha^2 + i_0 \\ \ldots \\ I(\alpha^{2^m-1}) = i_{k-1}\alpha^{(2^m-1)(k-1)} + \ldots + \\ \qquad i_2\alpha^{(2^m-1)2} + i_1\alpha^{(2^m-1)} + i_0 \end{cases} \quad (1)$$

The Generator Polynomial (GP) (2) [8] comes to light to decrease the complexity of decoding RS codes by acting as a keycode to produce the encoded message $C(x)$. GP is characterised by a polynomial of order $2t$, where its roots are the first $2t$ elements of the defined GF (b is usually chosen to be 0). The main assumption is that $C(x)$ must be divisible by GP, and it rules the values that parity symbols of $C(x)$ will hold. This simple constraint for $C(x)$ enables the most commonly used process to encode and decode RS codes, which will be covered in the next topics. Most explanation relies on [8], and it should be examined for more details.

$$G(x) = (x + \alpha^b)(x + \alpha^{b+1}) \ldots (x + \alpha^{b+2t-1}) \quad (2)$$

2.1 | Encoding process

The encoding process is based on the polynomial division of $I(x)$ by $G(x)$ in a GF(α^m), as demonstrated by Equation (3). $I(x)$ is shifted by $n - k$ positions (multiplication by x^{n-k}) to select the lowest $n - k$ coefficients of $C(X)$ to be the parity check symbols. $Q(x)$ and $R(x)$ are the quotient and the remainder of the division operation. If $\{i_{k-1}, i_{k-2}, \ldots, i_1, i_0\}$ is assigned to be the k highest coefficients of $C(x)$, then $R(x)$ has the missing coefficients to shape $C(x)$ in such way that it is divisible by $G(x)$. Equation (4) shows the result of the encoding process.

$$\frac{x^{n-k}I(x)}{G(x)} = Q(x)G(x) + R(x) \quad (3)$$

$$\begin{aligned} C(x) &= x^{n-k}I(x) + R(x) \\ &= i_{k-1}x^{n-1} + \cdots + i_0x^{n-k} \\ &\quad + r_{n-k-1}x^{n-k-1} + \cdots + r_0 \end{aligned} \quad (4)$$

2.2 | Decoding process

Errors might be introduced to $C(x)$ during its transmission through a noisy channel, and $E(x)$ corresponds to the polynomial that models the errors introduced in $C(x)$. Then, the received codeword $D(x)$ is the sum of $C(x)$ and $E(x)$. The number of non-zero coefficients of $E(x)$ must be less than or equal to t to guarantee the recovery of $C(x)$ after decoding $D(x)$. The following sequential steps are required for the decoding process:

1. Compute syndrome terms.
2. Solve the key equation to find error locator and evaluator polynomials.
3. Find the right positions where errors occurred.
4. Calculate the correction values of the errors.
5. Correct the erroneous positions.

2.2.1 | Syndrome calculator

The syndrome $S(x)$ characterises a particular error pattern in the received codeword. It is the remainder of the division between $D(s)$ and $G(x)$, which can be expressed by (5). If $S(x) = 0$, the received codeword $D(x)$ is error free.

$$S(x) = \sum_{i=b}^{b+2t-1} S_i x^{i-b}, \ \textit{where } S_i = D(\alpha^i) \quad (5)$$

2.2.2 | Key equation solver

This step aims to find out the error locator $\Lambda(x)$ (6) and evaluator $\Omega(x)$ (7) polynomials based on $S(x)$, which is a polynomial with $2t$ coefficients. The relation between $\Lambda(x)$, $\Omega(x)$, and $S(x)$ is known as key Equation (8). Peterson-Gorenstein-Zierler (PGZ) algorithm [9,10] is the first known solver for (8), and it solely relies on linear algebra [11]. PGZ is only

cost-effective for $t \leq 3$ because of its single correction capability—it requires different hardware units for each value of t [12]. It does not scale when t is large, and, for such scenarios, iterative solver methods such as Berlekamp–Massey [13,14] and Euclidean algorithms [15] are recommended.

$$\Lambda(x) = 1 + \Lambda_1 x + \Lambda_{t-1} x^{t-1} + \Lambda_t x^t \quad (6)$$

$$\Omega(x) = \Omega_{t-1} x^{t-1} + \cdots + \Omega_1 x + \Omega_0 \quad (7)$$

$$\Omega(x) = \Lambda(x) S(x) \bmod x^{2t} \quad (8)$$

2.2.3 | Error locator

The coefficients of $D(x)$ with errors are determined by calculating the roots L_j of $\Lambda(x)$, which has at most t degrees. All finite field elements are substituted into the equation to obtain the roots—method known as Chien's search [16].

2.2.4 | Magnitude discovery

After obtaining the $D(x)$ coefficients that requires correction, the next step is to find which value C_j should be added to the corrupted symbol as a means to correct the introduced error. Forney's method (9) calculates these values.

$$C_j = L_j^{1-b} \frac{\Omega(L_j^{-1})}{\Lambda'(L_j^{-1})} \quad (9)$$

2.2.5 | Codeword correction

The last step of the decoding process is the final correction of $D(x)$, which is done by simply adding C_j for $D(x)$ coefficient terms in the positions defined by L_j.

2.3 | RS codes in IEEE 802 standards

The IEEE 802 group specifies Local Area Network (LAN) and Metropolitan Area Network (MAN) standards and is mainly focussed on Medium Access Control (MAC) and Physical (PHY) layers of the ISO Reference Model for Open Systems Interconnection (OSI). RS codecs are prescribed in many subitems of IEEE 802 standards, mostly related to specification of PHY layers. Table 2 enumerates their main occurrences and demonstrates how varied are the parametric definitions for RS codecs.

The definition of n depends on the intended data rate of the related network interface, whereas k is dictated by the required reliability level given the channel characteristics such as noise and bandwidth. For instance, RS(544, 514) is applied in high frame rate technologies (e.g. 100 Gigabit Ethernet), as RS(15, 2) is present in low data rate communication (e.g. some operation modes of IEEE 802.15.7).

TABLE 2 Parametric configurations of RS codecs in IEEE 802

Standard	RS config.	GF
IEEE 802.3 [17]	RS(15,11)	2^4
	RS(144,128)	2^8
	RS(192,186)	2^8
	RS(255,239)	2^8
	RS(255,223)	2^8
	RS(450,406)	2^9
	RS(528,514)	2^{10}
	RS(544,514)	2^{10}
IEEE 802.15.3 [18]	RS(255,239)	2^8
IEEE 802.15.3c [19]	RS(33,17)	2^4
	RS(56,48)	2^4
	RS(52,44)	2^8
	RS(255,239)	2^8
IEEE 802.15.7 [5]	RS(15,11)	2^4
	RS(15,7)	2^4
	RS(15,4)	2^4
	RS(15,2)	2^4
	RS(64,32)	2^8
	RS(160,128)	2^8
IEEE 802.16 [20]	RS(46,26)	2^6
	RS(255,239)	2^8

Abbreviation: GF, Galois field; RS, Reed–Solomon.

Energy per bit-to-noise power spectral density ratio (E_b/N_0) is a parameter that allows comparing different coding schemes because it normalises Signal-to-noise (SNR) performance per bit. E_b/N_0 can be used to compare bit error rate (BER) among several RS configurations. To illustrate how the number of parity symbols ($n - k$) affects BER performance, Figure 3 shows different RS code configurations from IEEE 802.15.7 where $n = 15$. As expected, the BER increases when k is closer to n.

3 | RELATED WORK

Several proposals of RTL architectures for RS codecs are depicted in literature. The majority of them are designed for specific applications such as wireless networks [21,22], digital television [23], and space data systems [24]. They are usually regulated by standards that specify which parametric configurations must be used in the RS codec implementation. Hence, these works are limited to specific values for n, k, and m parameters in RS codec implementations that are similar to each other. Another aspect is that they present results of timing and

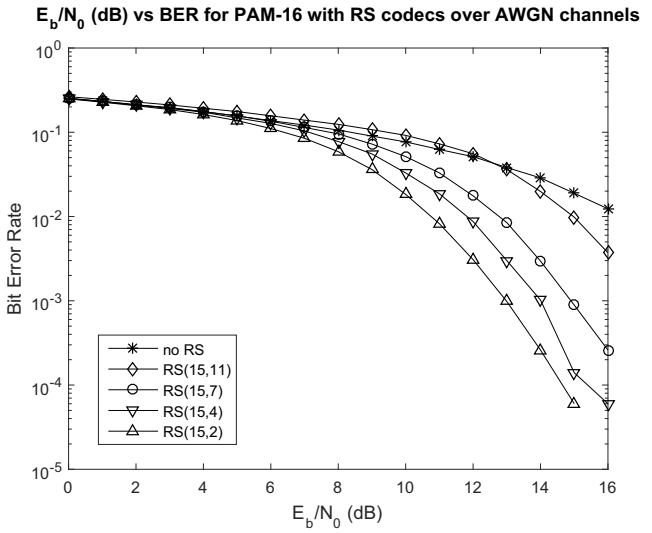


FIGURE 3 Transmission efficiency for Pulse-Amplitude Modulation with 16 discrete levels (PAM-16) over Additive White Gaussian Noise (AMGN) channels using RS codecs with $n = 15$ and $k = 11, 7, 4, 2$ and E_b/N_0 varying from 0 to 16 in a MATLAB® simulation of 10^6 sampling transmissions

device utilization from synthesis reports, but most of them lack verification analysis of the proposed design.

The most complex piece of a RS codec is the key equation solver (KES) (2.2.2) of the decoding process. It comprises the critical path of the decoder RTL design. As it is the most significant bottleneck to enable high-throughput communication systems, many papers focus on optimising KES blocks. Wu reviews implementations of both Euclidean and Berlekamp–Massey (BM) schemes considered as milestones in literature [25]. His work also proposes a BM-based architecture with less number of process elements (PE) compared with existing formulations. Ji et al. attempted to enhance Wu's work by using a single PE that is multiplexed during the calculation of Equations (6) and (7) [26]. This approach could be employed because other steps in 2.2 have far more latency than the KES. A pipelined GF multiplier was also used to improve the clock frequency, and the resulting decoder architecture reached a throughput of 4.6 Gb/s with 12.950 gates. Ji's design fundamentals are used by Zhang et al. [27], which claims that some PEs could be replaced by a Compensation Simplified (CS) unit responsible for computing some edge cases of the BM algorithm. Their proposed RS decoder has throughput of 5,1 Gb/s with 12.963 gates.

There are very few articles on the development of parameterizable IPs for RS Codecs. Smith et al. explored a configurable RS Codec IP using VHDL and focussed their contribution on how gate count of RS decoder blocks are influenced by design parameters [28]. Timing analysis was neglected in that work, an important metric that defines whether a given architecture meets or not throughput requirements of the targeted application. Park et al. covered such an aspect by proposing a Soft IP compiler for RS decoders that is capable to search a RTL architecture which is closest to user area and throughput needs [29]. It concentrates more on the algorithm used to generate the IP, and the aspects of the RTL development of parameterizable RS codecs were not a centrepiece in that work. These facts suggest that there is room to explore a generic architecture of a RS codec and its parameter implications in verification and synthesis phases.

4 | RTL DESIGN

The IP that implements a parameterizable architecture of a RS codec was developed in VHDL-2008. The parameters and ports of both RS encoder and decoder interfaces are equivalent. The IP user may select any combination of $RS(n, k)$ and GF order (2^m) as showed in Table 3. Furthermore, the IP supports shortened versions of the original $RS(n, k)$ setup, then the input message length might vary between 1 and k—but the number of parity symbols inserted into the encoded codeword is always $n - k$. Also, GF is bound to 2^{10} ($m = 10$) because GP coefficients for each $RS(n, k)$ combination are precomputed in a Look-Up Table (LUT) format contained in a VHDL package (*RS_CONSTANTS*). The limit of 2^{10} has been chosen based on the current industry demands which is captured by what is in the IEEE 802 standards (Table 2). A *python* script calculates GP coefficients relying on a set of primitive polynomials provided by the user and generates *RS_CONSTANTS* package.

The port interface of the proposed IP is listed by Table 4. A codeword is transmitted or received in a serial mode, so there are input and output ports for a single symbol along with their respective validation ports. Also, both input and output codewords must have delimiters to identify the first and the last symbol of them. The IP informs its readiness to receive a new valid input symbol, and external availability to consume output symbols is notified to the IP. There is an error indicator that is asserted when the following operating principles are violated:

- A *i_end_cw* pulse without a previous *i_start_cw* pulse.
- Two *i_start_cw* pulses without a *i_end_cw* pulse in between.
- Number input symbols exceeding the maximum codeword length.

The data path of RS codec depends on finite field arithmetic operators, which were implemented based on the usual approach seen in literature [8]: adders and subtractors are implemented by a simple XOR (only valid when $\alpha = 2$); constant multipliers operate by retrieving precomputed multiplication results from LUTs; full-multipliers are built using a

TABLE 3 Parameter interface of RS codec

Name	Description	Range
n	Length of the codeword	2 to 1023
k	Number of message symbols	1 to $n - 2$
m	GF order	2 to 10

Abbreviation: GF, Galois field; RS, Reed–Solomon.

TABLE 4 Port interface of RS codec

Name	I/O	Description
clk	I	System clock pin
rst	I	System reset pin
i_consume	I	Readiness to consume *o_symbol*
i_start_cw	I	Delimiter of input codeword start
i_end_cw	I	Delimiter of input codeword end
i_valid	I	Validity of input symbols
i_symbol	I	Input data symbol
o_start_cw	O	Delimiter of output codeword start
o_end_cw	O	Delimiter of output codeword end
o_in_ready	O	Readiness to accept new input symbols
o_valid	O	Validity of output symbols
o_error	O	Error indicator
o_symbol	O	Output data symbol

Abbreviation: RS, Reed–Solomon.

combination of bitwise AND and XOR operations on their operands; and dividers use a full-multiplier in conjunction with an inverter, which is implemented via LUT.

4.1 | Encoder

According to Section 2.1, the codeword that resulted from encoding process consists of the own input data symbols (*i_symbol*) plus the result of the *mod* operation between $x^{n-k}I(x)$ and $G(x)$. The top level architecture has a control and a processing unit as seen in Figure 4. The processing unit is responsible for computing the *mod* operation and the generation of the output symbols (*o_symbol*). The control unit manages the initialisation, interruption and output control of the processing component. The input symbol (*i_symbol*) is stored in a register because the control unit requires one-cycle delay to decide the processing status.

4.1.1 | Processing unit

The *mod* operation described in 2.1 is performed by an LFSR. The remainder is incrementally computed for the first k input symbols, which represents the coefficients of $C(x)$ from the highest to the lowest order. The size of the LFSR corresponds to the number of parity symbols, which is $n—k$. Figure 5 shows the RTL schematic of the processing unit. *A_DIV* is responsible for computing the quotient of the division step. Then, it is multiplied by the GP coefficients, with the exception of the highest order coefficient (x^{2t} according to Equation (2)), which is already cancelled in the current division step. The result is summed (or subtracted) with the previous accumulated value, and the registers are updated. After k steps, the remainder ($R(x)$) is completed and transmitted by selecting the

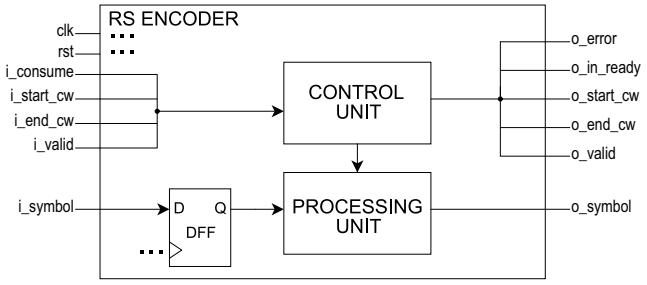


FIGURE 4 Top level architecture of Reed–Solomon encoder

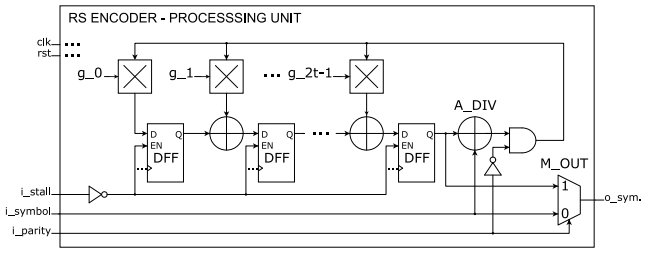


FIGURE 5 Register-transfer level schematic of the processing unit of Reed–Solomon encoder

first input of *M_OUT*. Every register has an enable port to retain its state when the mod calculation is stalled (*i_stall* = '1').

4.1.2 | Control unit

It implements the Finite State Machine (FSM) showed in Figure 6. *WAIT SYMBOL* (WS) is the initial state and scans for the first valid input symbol. Once it is detected, the next state is *START CODEWORD* (SC), which allows the processing unit to receive the input symbols to be encoded and also asserts *o_start_cw*. If the input codeword has more than one symbol to encode, *PROCESS SYMBOLS* (PS) takes over and supervises the input symbols until the end of the input codeword. Since the RS Encoder does not have a buffer to hold multiple symbols, a received symbol must be consumed to leave room for the next one. When the last received input symbol is consumed, the FSM is ready to go to *GENERATE PARITY* (GP). This state controls the transmission of the parity symbols stored into the registers of the processing unit. It provides $n - k - 1$ parity symbols, consumed by the block that reads *o_symbol*. The last parity symbol is handled by the *END CODEWORD* (EC), which asserts *o_end_cw*. Finally, the control unit returns to WS or SC, depending on the availability of a new valid symbol. *ERROR* (E) state is reached when any of the violations explained at the beginning of this section occurs, represented by dashed lines in Figure 6, and this state requires the encoder reset.

4.2 | Decoder

The top level architecture of the RS decoder is drawn in Figure 7. According to 2.2, five steps are required to complete

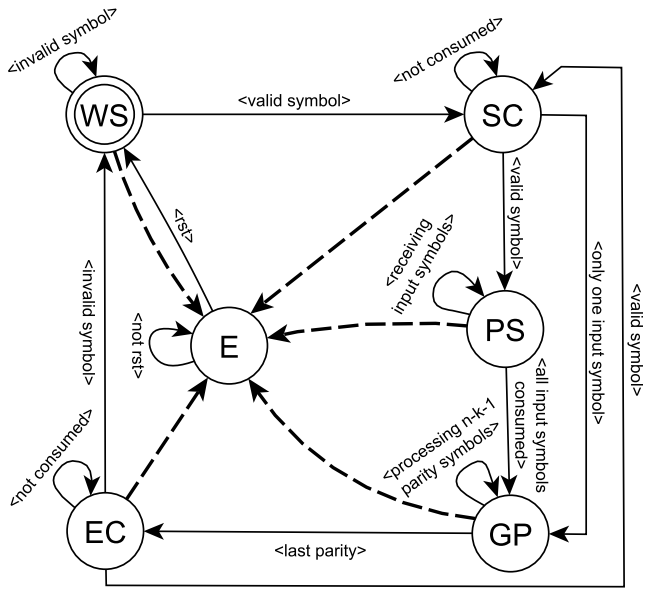


FIGURE 6 Finite state machine implemented by the control unit of Reed–Solomon encoder

the decoding process. They are implemented into three pipeline stages. The first stage consists of the syndrome calculator (2.2.1), which computes $S(x)$ and stores it into the pipeline register. Input control signals go to the control unit of the *Syndrome* block, so that it owns multiple roles: coordination of the syndrome processing unit, error signalling (*i_error* = '*1*') in case of unexpected behaviour of control inputs, and controlling of *SYMBOL FIFO* and *CW LENGTH FIFO*. The second stage implements the KES (2.2.2) and uses the BM algorithm for that purpose. The results of this process, $\Lambda(x)$ (6) and $\Omega(x)$ (7), are stored into the pipeline registers. Finally, the third stage combines Chien's search (2.2.3), Forney's method (2.2.4), and codeword correction (2.2.5) because they are interdependent steps. The control unit of this stage coordinates them, drives output control signals and consumes *SYMBOL FIFO* and *CW LENGTH FIFO* accordingly. The three pipeline stages will be detailed in subsequent sections.

4.2.1 | First pipeline stage

Equation (5), which calculates $S(x)$, can be rearranged to Equation (10). This procedure is known as Horner's rule, and Equation (8) has a simpler hardware implementation. Basically, the coefficients of $S(x)$ can be calculated by N successive operations of multiplications and additions. The initial value is set to $D_{n-1}\alpha^i$. Then, it is added to the next $D(x)$ coefficient and finally multiplied by α^i. This procedure is carried out recursively until last symbol of the received codeword (D_0) is added to the accumulated result.

$$\begin{aligned} S_i = D(\alpha^i) &= D_{n-1}(\alpha^i)^{n-1} + D_{n-2}(\alpha^i)^{n-2} + \\ &\quad \ldots + D_1\alpha^i + D_0 \\ &= (\ldots(D_{n-1}\alpha^i + D_{n-2})\alpha^i + \cdots + D_1)\alpha^i + D_0 \end{aligned} \tag{10}$$

Processing Unit

Each Syndrome coefficient (S_i) is computed by an independent processing unit (*Syn Term Unit*) as showed in Figure 8. It implements the Horney's rule using constant multipliers and adders. Two registers store the accumulated result and the S_i value. The unit is frozen by *i_stall* when *i_symbol* is invalid (*i_valid* = '*0*'), and *i_zero* is asserted at the first cycle of the calculation process to reset the accumulated value to $D_{n-1}\alpha^i$. The number of parallel *Syn Term Units* corresponds to the number of parity symbols of the adopted RS configuration ($n - k$).

Control unit

It implements a FSM displayed by Figure 9. *WAIT SYMBOL* (WS) is the initial state and scans for the first valid input symbol. When it comes up, the next state is *START SYNDROME* (SS), which asserts *i_zero* to reset all *Syn Term Units* to their initial values. If the upcoming *i_symbol* is valid, the *COMPUTE SYNDROME* takes control and monitors the remaining codeword symbols. At this state, whenever a *i_symbol* is valid (*i_valid*), a write operation is sent to *SYMBOL FIFO*. An internal counter keeps track of the number of received symbols, and it should neither exceed the maximum number of input symbols (n) nor be less than $n - k + 1$. Otherwise, *ERROR* (E) state is reached. Whenever *i_valid* is de-asserted during SS and CS states or FIFOs become full, *STALL* (S) assumes to freeze all *Syn Term Units* until *i_valid* is re-enabled and FIFOs are free. The signal *o_in_ready* is de-asserted whenever FIFOs are full. The FSM goes to *END SYNDORME* (ES) state when *i_end_cw* is asserted during CS, and it saves last valid input symbol and store the total number of received symbols into the *CW LENGTH FIFO*. Finally, the last state is *REGISTER RESULT* (RR), which stores $S(x)$ result into the pipeline register. If it is already occupied by a previous $S(x)$, then RR waits until it becomes free and de-asserts *o_in_ready* meanwhile. After that, the next state might be WS or SS, depending on whether a new valid input symbol is available or not. E is reached when any of the violations explained at the beginning of this section occurs, represented by dashed lines in Figure 9, and this state requires the decoder reset.

4.2.2 | Second pipeline stage

The BM algorithm is an iterative procedure to resolve (6). The algorithm steps [?] are shortly represented below:

- Initial values:

$$\Lambda^{(0)}(x) = 1,\ B^{(0)} = 1,\ L^{(0)} = 1,\ d^{(0)} = 1$$

- In every kth iteration:

$$d^{(k+1)} = S_k + \sum_{i=1}^{L^{(k)}} \Lambda_i^{(k)} S_{k-i} \tag{11}$$

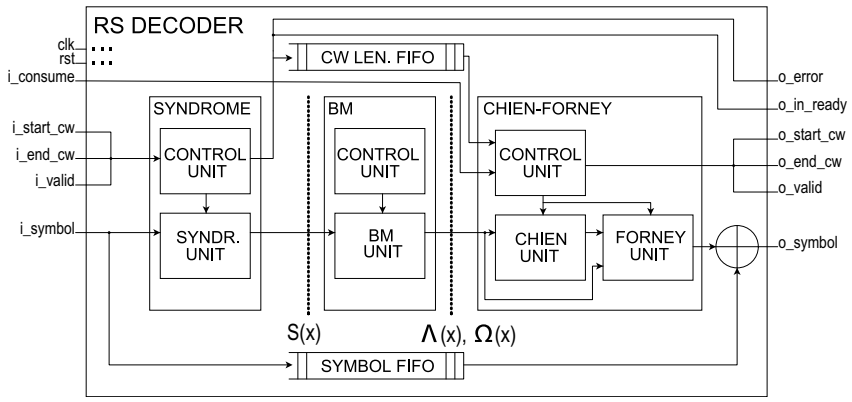


FIGURE 7 Top level architecture of the Reed–Solomon decoder

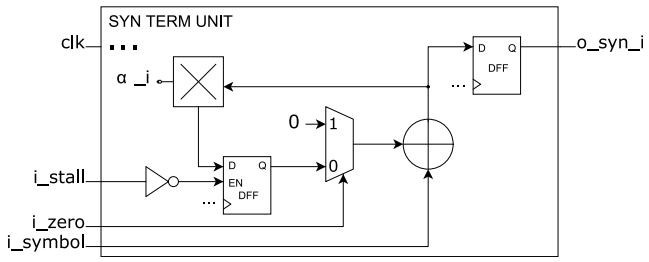


FIGURE 8 Register-transfer level schematic of *Syn Term Unit*

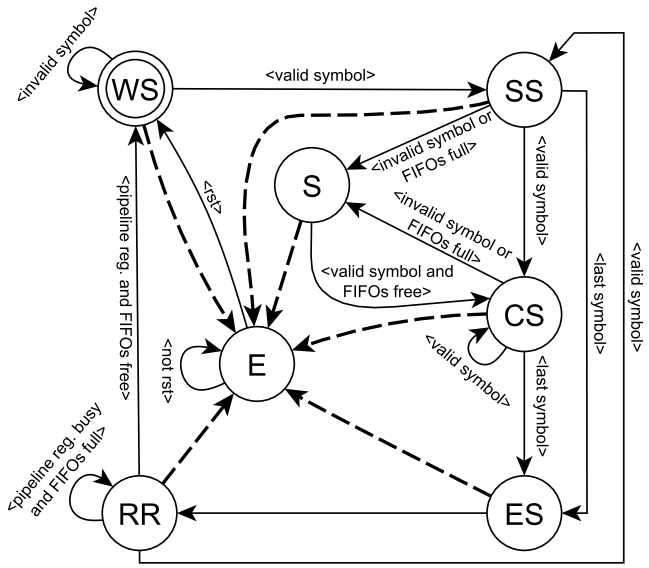


FIGURE 9 Finite state machine implemented by the control unit of *Syndrome*

$$\Lambda_{(k+1)}(x) = \Lambda^{(k)}(x) - \frac{d^{(k+1)}}{d^{(k)}} B(x)x \tag{12}$$

$$B^{(k+1)}(x) = \begin{cases} xB^{(k)}(x) \text{ if } d^{(k+1)} = 0 \text{ or } 2L^{(k)} \geq k \\ \Lambda^{(k)}(x) \text{ if } d^{(k+1)} \neq 0 \text{ and } 2L^{(k)} < k \end{cases} \tag{13}$$

$$L^{(k+1)} = \begin{cases} L_{(k)} \text{ if } d^{(k+1)} = 0 \text{ or } 2L^{(k)} \geq k \\ (kL) \text{ if } d^{(k+1)} \neq 0 \text{ and } 2L^{(k)} < k \end{cases} \tag{14}$$

- Stop condition:

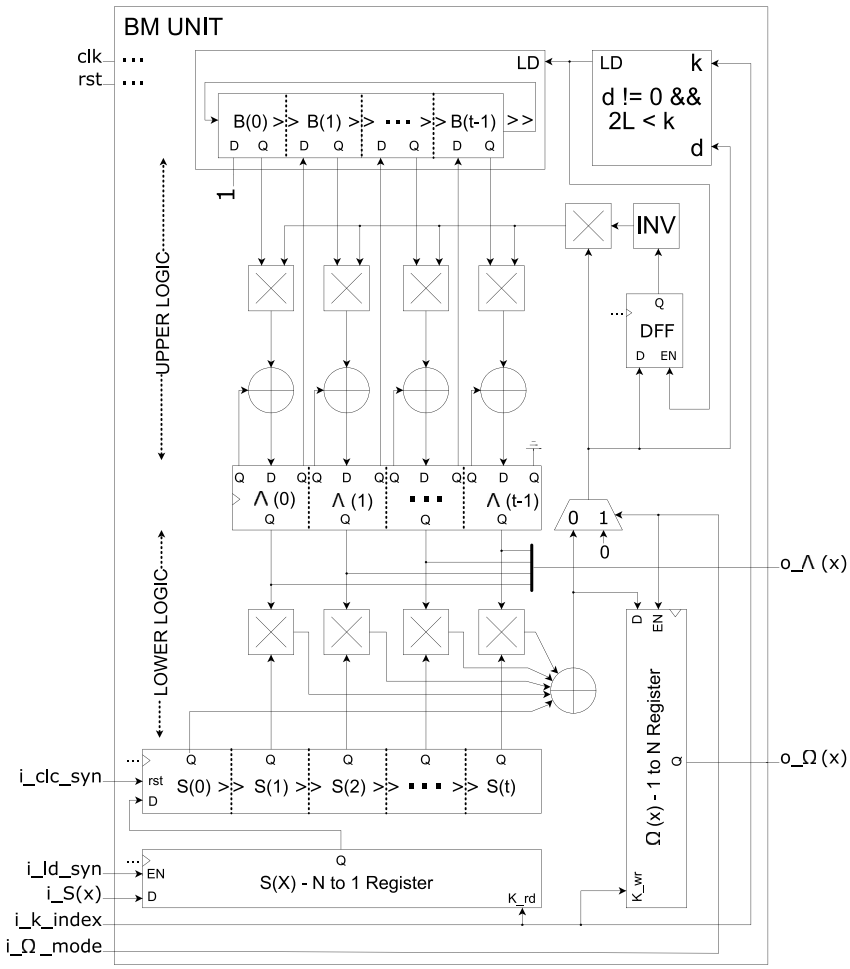
$$k = 2t$$

The error-locator polynomial $\Lambda(x)$ is decremented every iteration (or incremented) by the ratio between the past and current discrepancy (d) and multiplied by an auxiliary polynomial $B(x)$ shifted by one position (12). L is a reference number used in conjunction with $d^{(k)}$ as criteria to define what is the update of $B(x)$. After $2t$ iterations, $\Lambda(x)$ result is ready, then error-evaluator polynomial $\Omega(x)$ can be computed. It is composed of the first t coefficients of the resulting polynomial from $\Lambda(x) \times S(x)$. This operation takes t iterations of a polynomial multiplier.

The hardware architecture of the described BM algorithm was first proposed by Massey [16], which is the classical approach present in literature. The main issue of the original BM algorithm is the division operation present in (12) because its hardware implementation is complex and time-consuming. An inverter and a full-multiplier is usually employed to carry-out a division operation. For $m < 8$, inverters are performed by LUTs because they are more efficient than the hardware implementation of inversion algorithm [18]. BM algorithm has been reformulated to avoid the inversion operation [?], and some enhancements in the hardware implementation have been proposed, as referred in Section 3. However, this article still explores Massey's proposal because it aims to present a starting point of a parameterizable RS codec that will be reference for further enhancements and comparative analysis in future works.

Processing unit

It implements an LFSR of t stages (Figure 10). Once $S(x)$ is available, it is stored into the 'N to 1 Register', which drives a single $S(x)$ indexed by the current iteration k. The 'lower logic' computes the summation referred in (11), whereas the 'upper logic' implements (12). Λ_0 register is omitted because it is always 1. $B(x)$ is placed in a shift register, and $\Lambda^{(k)}(x)$ is loaded to it according to the conditions in (13). A simple logic was implemented to update $L_{(k+1)}$, $B_{(k+1)}$, and $d_{(k+1)}$ according to the condition 'd ! = 0 && $2L < k$'. The 'lower logic' used to calculate d which operates as a polynomial multiplier. Hence, it can also compute $\Omega(x)$ as well. During this time the 'upper logic' is disabled to keep the result of $\Lambda(x)$ (i_Ω_mode = '1'), and the result of each $\Omega(x)$ term is stored in a '1 to N Register'.



The critical path of this unit—and also for the whole RS decoder—connects the 'lower logic' with the 'upper logic', and it has three multipliers, an adder (XOR), a multiplexer, and a summation operator (XOR-reducer).

Control unit

It implements a FSM of six states (Figure 11). It starts with *WAIT SYNDROME* (WS), and it looks for a valid $S(x)$ in the pipeline register. Once it is found, the next state is *START BM* (SBM), which consumes $S(x)$ from the pipeline register and initialises the Processing Unit. In the next cycle, *COMPUTE LAMBDA* (CL) assumes to calculate $\Lambda(x)$ during the next $2t$ cycles. Once it is finished, *CLEAR SYNDROME* (CS) is the subsequent state responsible for resetting the syndrome shift register with S_0. Then $\Omega(x)$ is calculated after t cycles in *COMPUTE OMEGA*(CO) state. Finally, $\Lambda(x)$ and $\Omega(x)$ are ready to be stored into the pipeline register. *END BM* (EBM) takes care of it and moves to WS or SBM depending on whether a new $S(x)$ is available or not.

4.2.3 | Third pipeline state

With $\Lambda(x)$ and $\Omega(x)$, it is possible to use Chien's search (2.2.3) and Forney's method (2.2.4) to find out which D_i has error and

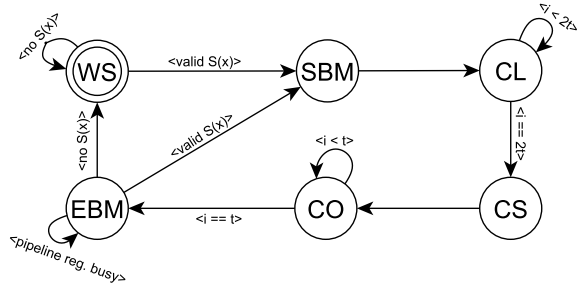


FIGURE 10 Register-transfer level schematic of Berlekamp–Massey processing unit

FIGURE 11 Finite state machine implemented by the control unit of Berlekamp–Massey

the correcting value needed to recover the original symbol. Since these steps are highly tied, they have the same controller. Chien's Search notifies if a given D_i has error or not. It also informs what is the value of $\Lambda'(L_j^{-1})/L_j$. When $b = 0$, it corresponds to the summation of the odd terms of $\Lambda(x)$ evaluated in L_j [8]. Therefore, (9) is obtained when $\Lambda'(L_j^{-1})/L_j$ is inverted and multiplied by the result of $\Omega(x)$ evaluated in L_j.

Chien unit

The roots of $\Lambda(x)$ indicate the positions of symbol corruption in $D(x)$. All GF elements are tested into $\Lambda(x)$ to find these roots. *Term Evaluator Unit* (Figure 12) does cumulative

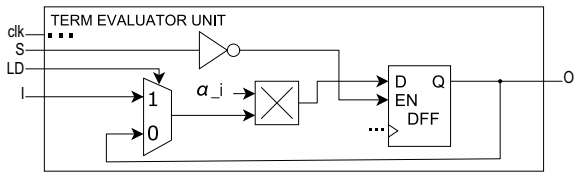


FIGURE 12 Register-transfer level schematic of *Term Evaluator Unit* (TEU)

multiplication of α^i by its respective $\Lambda(x)$ terms every iteration j from 1 to $\alpha^m - 1$. Therefore, each iteration obtains the evaluation of a $\Lambda_i(x)^i$ term from Equation (6) in j. Then, results of each *Term Evaluator Unit* are summed up to obtain $\Lambda(j)$ (Figure 13). Odd and even terms are added separately first since the odd side corresponds to $\Lambda'(L_j^{-1})/L_j$, which is a value needed by *Forney Unit*.

Forney unit

Forney method requires the polynomial evaluation of $\Omega(x)$, similar to the procedure done by Chien's search. Therefore, the only difference between *Chien* and *Forney Units* (Figures 13 and 14) is that the latter has an inverter and a multiplier to compute (7) and does not require the summation of the odd terms. The resulting value is only considered when Chien Unit detects an error in a given D_i term. Otherwise, the block output is 0. Finally, the step 2.2.5 is carried out by adding the output of this block and the D_i term stored into the *SYMBOL FIFO*.

Control unit

It implements FSM with five states (Figure 15). *WAIT BM* (WBM) checks for $\Lambda(x)$ and $\Omega(x)$ in the pipeline registers. When they are detected, the next state may *START CHIEN-FORNEY (SCF)* or *ADVANCE VOID POSITIONS* (AVP). AVP is reached when $D(x)$ has less than 2^m terms. Then, since *Chien Unit* evaluates all terms from GF(2^m), results for unused terms must be ignored. The subtraction between 2^m and the length of $D(x)$ determines which cycle SCF will be reached to assert *o_start_cw* and transmit the first valid output symbol. *COMPUTE CHIEN-FORNEY* (CCF) takes care of the subsequent processing iterations. Finally, *END CHIEN-FORNEY* (ECF) sends the last codeword symbol of this process and asserts *o_end_cw*. During SCF, CCF, and ECF, terms are only advanced when *i_consume* is asserted, and *CHIEN* and *FORNEY UNITs* are stalled otherwise. After *o_valid* is asserted, *SYMBOL FIFO* will be consumed in each subsequent cycle where *i_consume* is enabled until $D(x)$ is completely processed. If the pipeline registers already have the next $\Lambda(x)$ and $\Omega(x)$ to be processed, the FSM restart the computing process in SCF or AVP. Otherwise, it returns to WBM.

5 | VERIFICATION

The verification of a parameterizable RS codec is a challenging task because every combination of its parameters determines a unique design and its complexity depends on them. For example, if the parameter n is set to 15, k to 11, and m to 4,

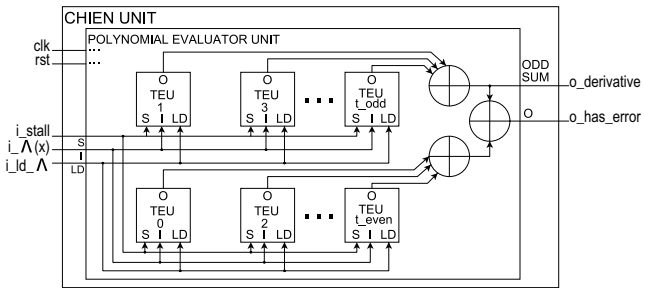


FIGURE 13 Register-transfer level schematic of *Chien Unit*

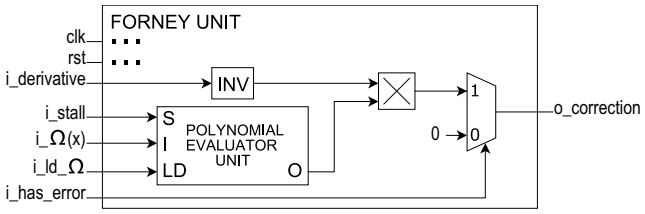


FIGURE 14 Register-transfer level schematic of *Forney Unit*

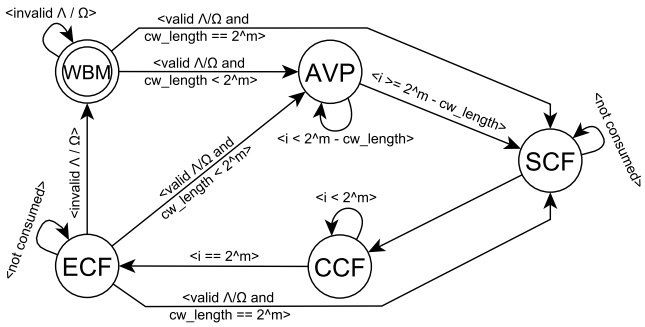


FIGURE 15 Finite state machine implemented by the control unit of *Chien-Forney*

each symbol may assume 16 different values and the input codeword may have up to 11 symbols, making the number of different input codewords be $(2^4)^{11}$. A similar analysis can be done for the number of parity symbols, which corresponds to $n - k$. Hence, n, k and m make a great impact on the complexity of inputs or outputs of the IP.

Moreover, n, k and m also dictate the state space of the model that represents the RTL design and the number of tests required to comprehensively verify it. The state space of a RTL design consists of the value of each input and state element at a particular time. The state elements include flops, latches and memories. The issue lies with the fact that an RTL design with i state elements, theoretically, would have 2^i possible configurations. These state elements in conjunction with the number of input bits (j)—and their combinations 2^j—forms the complete design state that could be conceived: 2^{i+j}.

The main reason for increasing the number of states in a instantiated RS codec is t since it is directly related to the

number of registers in the data path of the design as pointed out by Section 4. The parameter m must be taken into consideration as well since it determines the width of these registers. Therefore, a systematic approach to check a parameterizable RTL design is mandatory to guarantee the quality of the verification effort.

5.1 | Methodology

Dynamic Assertion-Based Verification (ABV) such as directed tests, constrained-random simulation, and hardware acceleration have been the traditional approach to design verification [30]. Simulation-based methods rely on HDL testbenches, which defines a set of input vector stimuli for the Design Under Test (DUT). Such approach is non-exhaustive in nature and not effective for critical systems, whose desired unsafe scenarios are difficult to be assessed via test benches [31]. However, another approach has gained acceptance in recent years with widely recognized advantages in finding corner case bugs in complex designs: Formal Verification (FV). This method uses tools that mathematically analyse the state space of a RTL design, rather than computing results for particular values [32]. Hence, if a design behaviour is proven using FV, there is no sequence of input vectors that is capable to expose a corner-case bug.

FV is more effective on control logic or data transport blocks than data transformation blocks, since the later involves RTL designs with complex mathematical manipulations that result in a large state space. This case applies to RS codecs as explained previously. However, recent technological advances in satisfiability solvers employed by FV tools have enabled their use in more complex problems [33]. Moreover, certain techniques can be used to reduce the problem complexity for FV tools and still ensure an acceptable quality level of verification. One of these techniques is the possibility of scaling down the design without affecting its key aspects that must be verified. Proving desired behaviours in a scaled down design often provides a significant indicator about the correctness of the original design [32]. Another aspect is that the functional design requirements can be broken down into multiple behavioural checks (case split) that demand lower computational resources and can together attest the original requirement. These two techniques have been successfully applied for verifying an ECC block in [33], and they are used in this work as well.

Since it is not realistic to check all possibilities of n, k, and m, this article focuses on the RS configurations listed on IEEE 802.15.7 (Table 2). It provides an adequate set of configurations that exercises variation in k with constant n, large differences in n, and contrasting values for m. The machine model used to run the experiments is a Intel® Xeon® Bronze 3106 1.7 GHz with a RAM memory of 8 GB. The FV tool adopted in this work is JasperGold® Formal Verification Platform from Cadence® Verification Suite 2019.12v. It requires inputs and provides outputs as described in the following:

Inputs to FV tool:

- An RTL model
- Check requirements (Assertions): Capture the design behaviour to be proved. These set of behavioural checks are divided in two: end-to-end assertions and interface assertions. The end-to-end assertions express the required core functionality of the design, and should be described only in terms of block interface signals. The interface requirements specify the control rules expected by neighbour blocks and are defined solely over a single interface [34].
- Generate requirements (Assumptions): Assumptions restricts the set of admissible input value sequences, preventing illegal input stimuli from causing spurious property violations. Incorrect assumptions over-restrict the input stimuli to a subset of possible behaviours and hide real property violations. Therefore, it is important to validate the constraints.
- Cover requirements (Covers): Coverage is a measure of verification completeness. It provides confidence that the DUT has been fully verified in a deterministic, and measurable way, within a non-over-constrained environment.

Outputs from FV tool:

- List of proven asserts and unreachable covers.
- Waveforms showing Counter Examples (CEx) of assertions or reachability traces of covers.
- List of assertions and covers that are undetermined. Even though a full-proof is not obtained for these properties, their achieved bound during formal analysis might be enough to validate some requirement (bounded-proof).

The requirements of RS codec are classified into three categories: Reset, Interface Control and Functional. Reset specifies the initial state of the design and output ports; Interface Control defines what is the procedure required to communicate with the RS codec; and Functional certifies the correctness of the encoded or decoded codeword process. Each requirement is mapped into a set of SystemVerilog Assertions (SVA) properties that can be analysed by the FV tool. For each written assertion, there is a cover with the same expression to guarantee that the desired behaviour can be reached at least once and prevent bogus proof results.

5.2 | Encoder

RS encoder requirements for Reset and Interface Control categories were verified by using white-box or code based approach. It requires knowledge of DUT implementation to create tests to ensure that internal components operate according to the design specification [35]. In this method, testing is based on coverage of code statements, branches, paths or conditions of the DUT. For Reset requirements, SVA properties were written to check the initial state of the control unit and the states of the registers instantiated within the processing

unit. For Interface Control requirements, all states and transitions of the FSM implemented by the control unit (Figure 6) were captured into SVA properties. Thirty-two asserts were created for these requirements. After fixing all CExs found during the verification process, all assertions could be proven in less than 1 min for each target RS configuration.

The functional requirements of RS encoder comprise the fundamentals of the encoding process. As explained in Section 2.1, $C(x)$ is composed of a sequence of input symbols ($I(x)$), which has length of 1 up to k, and the parity calculated by the RS encoder. The primary goal is to verify the correctness of $C(x)$; however, there is not a convenient way to describe a SVA property that checks it. A viable approach to certify the correctness of $C(x)$ is to use the syndrome calculator, which is the first step of the decoding process 2.2.1, to validate the encoding process (Figure 16). As $C(x)$ directly drives the *Syndrome Unit*, it implies that $S(x)$ should be 0 once the encoding process ends. This behaviour is captured by the following check:

```
ENC_FUNC_REQ:
assert property (o_end_cw |=> not(o_syn))
```

It is very difficult to obtain a full-proof of this property because it covers every valid sequence of encoded codewords starting from the reset state. If the independence between the codewords can be proven, then it is enough to check the correctness of a single encoded codeword. It means that all registers present in the processing unit of RS encoder must be in reset state, when the end of the codeword is reached (*o_end_cw = '1'*) with its last symbol consumed by external blocks. A checker for such a scenario was created and proven with low effort. To check a single encoded codeword, assumptions were created to model a contiguous input codeword of 1 up to k symbols. These assumptions excludes the case in which the input sequence is interrupted by invalid symbols (*i_valid = '0'*), and *i_consume* is always enabled. These constraints were imposed to reduce proof complexity of *ENC_FUNC_REQ*; however, the stall mechanism, triggered by *i_consume* or *i_valid* de-assertion, was separately checked with low effort by specific assertions that attest the stability of the data path registers when required.

Table 5 shows the proof results of for *ENC_FUNC_REQ* for each RS codec configuration specified in IEEE 802.15.7. The time limit set for each proof was 24 h and the results are from the JasperGold® engine 'Ht'. For RS configurations in which $n = 15$ some seconds are enough to prove *ENC_FUNC_REQ* for all possible input codewords. However, *RS*(64, 32) and *RS*(160, 128) with $m = 8$ could only be proven for input codewords of two symbols. It demonstrates how proof complexity increases with n, m, and t. *RS*(64, 32) with $m = 7$ and *RS*(63, 31) with $m = 6$ were also analysed to check if proof bound can be increased when m is decreased, and it did not cause any great impact on that. Then t was assigned to its minimum to observe proof results. *RS*(64, 62) and *RS*(160, 158) with $m = 8$ could be proven for all possible input codewords. Even though it was not be possible to fully prove *RS*(64, 32) and *RS*(160, 128), the results indicate an adequate level of assurance for them since scaled down scenarios were fully proven and shorter input codewords are able to exercise the complete data path because it only affects the space state of the input stimuli.

5.3 | Decoder

As opposed to the RS encoder verification, a black-box approach was employed to check Reset and Interface Control requirements of the RS decoder. Then, requirements were translated into SVA properties relying solely on the port interface of the DUT. The reason for that change is the difficulty in binding internal design behaviours to the original requirements because of the complexity of RS decoder. Reset and Interface Control requirements were mapped into 15 assertions and covers. After fixing all RTL bugs, most of them took less than a minute to prove, and the only exception was a

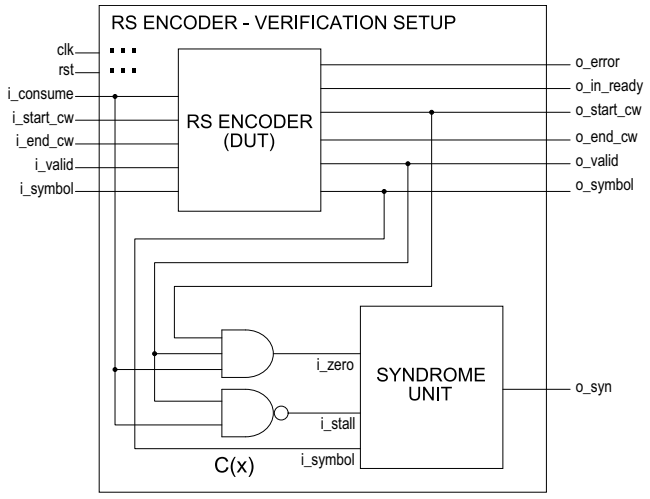


FIGURE 16 Verification scheme used to check the functional requirements of Reed–Solomon encoder

TABLE 5 Proof results of *ENC_FUNC_REQ*

RS Config.	*M*	Time(s)	Max. input Codeword Length
RS(15, 11)	4	1	11
RS(15, 7)	4	6	7
RS(15, 4)	4	32	4
RS(15, 2)	4	2	2
RS(64, 32)	8	9803	2
RS(64, 32)	7	1606	2
RS(63, 31)	6	7641	3
RS(64, 62)	8	34	62
RS(160, 128)	8	13,985	2
RS(160, 158)	8	521	158

property which checks that the output codeword has the same length as its corresponding input codeword. For the RS configurations where $n = 15$, it took about 1 h to fully prove this property, whereas it required 9 h prove for RS(64,32). The same property could not be fully proven for RS(128,160); however, it was possible to cover input codewords from 1 to 74 symbols in 22 h.

Similarly, for the RS encoder, it is not possible to write a simple checker for verifying the correctness of the decoded codeword without adding helper RTL blocks, as described by Figure 17. The received codeword ($D(x)$) must be a valid encoded codeword with the number of errors less than the maximum error capacity (t). As explained in 5.2, the encoded codeword is valid when $S(x)$ is 0 for it, then a syndrome processing unit was added to check this precondition. Also, an adder and a counter were introduced to corrupt the input symbol and keep track of the number of errors included in $C(x)$ to generate $D(x)$. A FIFO is needed to store $C(x)$, which will be compared with the symbol output of the decoder. Therefore, the following checker attests the correctness of the decoded codeword:

```
DEC_FUNC_REQ:
assert property (o_valid and not (o_syn) and (o_num_errors <= t) |-> (o_original_symbol = o_symbol))
```

Due to the complexity of RS decoder, to obtain a full-proof of *DEC_FUNC_REQ* is even harder than *ENC_FUNC_REQ*. Hence, the stall mechanism in Syndrome Unit to handle invalid input symbols, consumption handling in Chien-Forney controller, and the independence between subsequent codewords must be checked to simplify the proof of *DEC_FUNC_REQ* for a single contiguous $D(x)$ with *i_consume* always enabled. Some key registers present in the datapath of three pipeline stages of RS decoder must be checked to certify these functional aspects. Then, a bounded-proof of *DEC_FUNC_REQ* that covers the number of cycles required by all pipeline stages for processing a single codeword, as specified by Figure 18, should be enough to prove the correctness of the decoded codeword for all possible $D(x)$.

After several bug fixes in the RS decoder, Table 6 shows the total time spent to prove *DEC_FUNC_REQ* for RS codec configurations that could be proven using JasperGold® engine 'Ht' for a time limit of 72 h. The only RS configuration specified in IEEE 802.15.7 that could be proven was RS(15, 11) after more than 56 h. The other RS configurations could not even be proven for input codewords with the minimum length. It demonstrates that RS decoder is complex to formally verify due to its large state space even for simple RS configurations. An alternative is to scale down the parameters and check how the design behaves when varying t and m. Then, *RS*

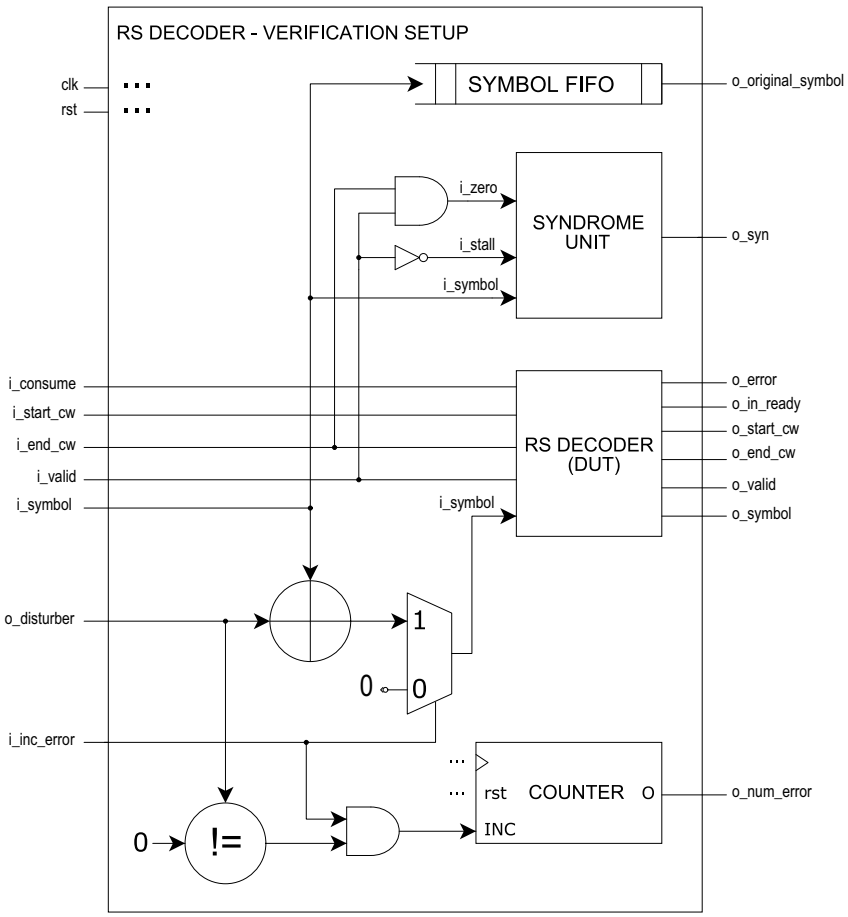


FIGURE 17 Verification scheme used to check the functional requirements of RS decoder

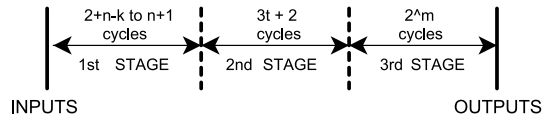


FIGURE 18 Timeline representing the number of cycles spent in each pipeline stage of the Reed–Solomon decoder

TABLE 6 Proof results of *DEC_FUNC_REQ*

RS Config.	*m*	*Time(s)*	Max. input codeword length
RS(7, 5)	3	1	7
RS(7, 3)	3	13,623	7
RS(7, 1)	3	887	7
RS(7, 5)	4	2	7
RS(7, 3)	4	156,893	7
RS(7, 1)	4	114,832	7
RS(7, 5)	8	3588	7
RS(7, 3)	8	–	0
RS(7, 1)	8	–	0
RS(15, 11)	4	199,617	15

(7, 5), *RS*(7, 3), and *RS*(7, 1) with m assuming 3, 4, and eight were used for verification analysis. The proof effort abruptly increases with m according to the results, and only *RS*(7, 5) could be proven for $m = 8$. Furthermore, state space and prove complexity certain scales with t; however, when k is very low, the reduced number of input combinations may compensate the increase of t. It explains why *RS*(7, 3) spends more proof time than *RS*(7, 1).

Even though *DEC_FUNC_REQ* could not be proven for many RS configurations specified by IEEE 802.15.7, the results for scaled down designs in conjunction with the proof of Reset and Interface Control properties are a relevant indicator about the correctness of the RS decoder. The RS configurations described by Table 6 obtained at least one proof for each targeted m, which confirms that the basic operators are performing properly. Incremental values of t (1, two and 3) were checked, which indicates that the data path and control units are correctly arranged when error correction capacity is increased. Moreover, there is still room to improve proof convergence in FV. For example, abstract models may replace complex design structures and helper assumptions may be included to assist formal solvers.

6 | SYNTHESIS

The interface parameters n, k, and m affect area and throughput of RS codec. As m dictates GF order, it determines the complexity of arithmetic operators and the width of data buses. The error-correction capacity (t) is another relevant factor because it specifies the number of elementary processing units in *Syndrome*, *Chien* and *Forney Units* and the size of the LFSR architectures used in the RS encoder and *BM Unit*. Table 7 lists the utilization of the main basic RTL blocks in data path entities of RS codec: Registers (R), Muxes (M), Adders (A), Constant Multipliers (CM), Full Multipliers (FM) and Inverters (I). The control units are almost static in relation to the RS codec parameters, and their internal counters are the only exception to it.

TABLE 7 Utilization of basic elements in data path units of the RS codec

Data path unit	R	M	A	CM	FM	I
Encoder	$2t$	$2t + 2$	$2t$	$2t$	–	–
Syndrome	$4t$	$4t$	$2t$	$2t$	–	–
BM	$4t + 3$	5	$2t + 1$	–	$2t$	1
Chien	$t + 1$	$t + 2$	$t + 2$	$t + 1$	–	–
Forney	$t + 1$	t	t	T	1	1

FPGA devices can be used to observe how logic utilization and maximum allowed frequency (F_{max}), intrinsically related to area and throughput, are affected by variations of the parametric configurations of the RS codec. EP4CE6E22C6 device from the family Cyclone IV E was selected to illustrate such analysis in Quartus® Prime Lite Edition 18.1.0. Since the RTL design of RS codec mostly depends on m and t—the only exception is the *SYMBOL FIFO* which varies only with n, synthesis reports were generated for $m = 4, 5, 6, 7, 8$ and t bound to 16. The highest possible value for n was used to obtain t to obtain the worst case scenario due to *SYMBOL FIFO* dependency on n. Figures 19 and 20 report the number of Logic Elements (LE), Register Elements (RE), and F_{max} for RS encoder and decoder with t varying from 1 to 16.

The synthesis results for logic elements and registers reported for both RS encoder (Figure 19a,b) and decoder (Figure 20a,b) show curves that are linear with t and their coefficients are determined by m. It confirms the O(N) complexity inferred from Table 7. As expected, the RS decoder has about 10 times more LEs and 8 times more REs than the RS encoder. Timing analysis was carried out using the most pessimistic operating condition: 'Slow °C Timing Model' [36]. For the RS encoder, Figure 19c does not indicate a clear correlation between F_{max} and design parameters. It means that unknown aspects related to 'Place & Route' algorithm employed by the FPGA tool has relevant predominance on F_{max} results of RS encoder, which has very low occupation in the target device (<10% for all cases). On the other hand, F_{max} results of the RS decoder (Figure 20c) exhibit curves that approximate to an exponential function decayed with t and scaled by m.

In the context of IEEE 802.15.7, RS codecs are specified for six different parametric configurations (Table 2), which are used among 38 operating modes of its three PHY layers. To evaluate the synthesis results of the proposed IP, LE, RE, Memory Bits (MB) and F_{max} numbers are compared against Altera® 'Reed Solomon II' IP Core [37], using the same device (EP4CE6E22C6) and analogous parametric configuration.

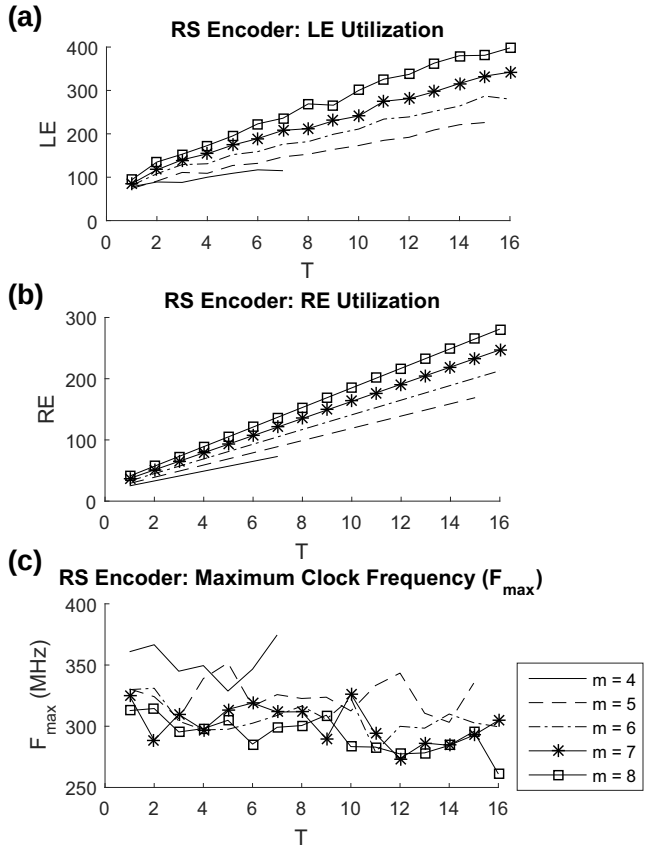


FIGURE 19 Synthesis results for the RS encoder (a) LE versus t, (b) RE versus t, (c) F_{max} versus t

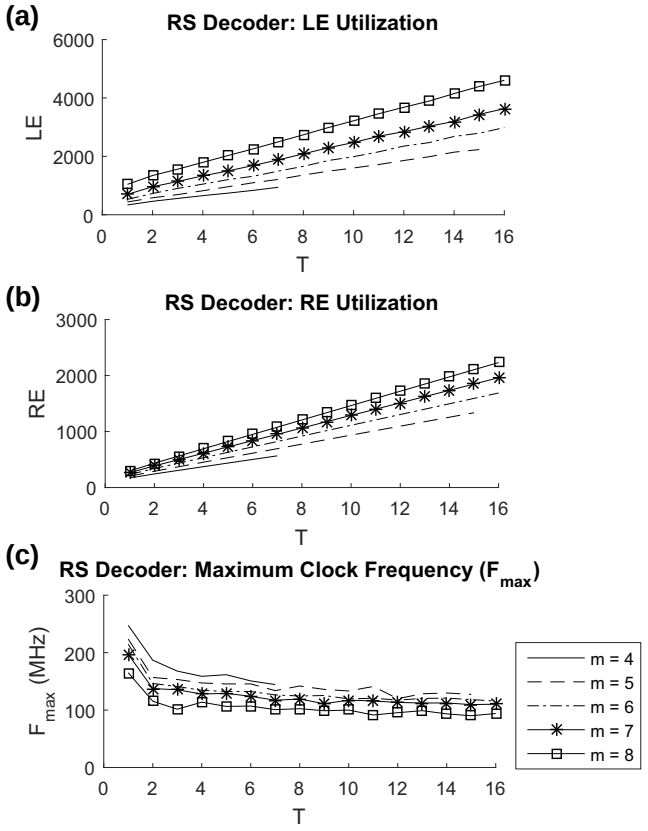


FIGURE 20 Synthesis results for the RS decoder (a) LE versus t, (b) RE versus t, (c) F_{max} versus t

Table 8 displays results for RS encoder (in grey) and RS decoder (in black) for the proposed IP (left) and Altera 'Reed Solomon II' (right). For the RS encoder, LE, RE, and MB are very similar to the commercial IP; however, F_{max} is more than 2 times as high in the proposed IP. For RS decoder, LE and RE is up to 20% lower and MB is superior in the commercial IP. As confirmed by Figure 20, F_{max} has a significant drop when m is increased, and such a behaviour is not apparent in the commercial IP. It means that there is room to reduce the critical path of RS decoder, as highlighted in 4.2.2. These results confirm that the proposed IP has synthesis numbers that are comparable to other available parameterizable IPs.

Each operating mode present in IEEE 802.15.7 defines an optical clock rate to transmit and receive data. Then, the throughput of RS encoder (T_{RSE}) or decoder (T_{RSD}) must be greater than the operation mode throughput (T_{OM}) determined by the selected Optical Clock Rate (OCR). This assumption guarantees that buffers used to store input and output symbols of the RS codec shall not have overflow or underrun.

Since an optical clock period codes a single bit in the modulation schemes adopted by IEEE 802.15.7, T_{OM} is the OCR itself (15). T_{RSE} and T_{RSD} requires the F_{max} of the related parametric configuration and the ratio between the number of bits of the input codeword and the number of cycles required to process it. Given that, T_{RSE} and T_{RSD} are represented by ((16)) and (17). The RS encoder requires n cycles to process k symbols of m width. The RS decoder processes a codeword of n symbols in every 2^m cycles, which is how many cycles the third pipeline stage takes. It is assumed that the initial latency for the pipeline filling is handled accordingly. Table 9 displays T_{OM}, T_{RSE} and T_{RSD} for each RS codec configuration present in IEEE 802.15.7 using the highest optical clock (OC_{max}) defined by a related operating mode. T_{RSE} and T_{RSD} are greater than T_{OM} in all scenarios. Therefore, the proposed IP core for RS codec, synthesised in the targeted FPGA device, fulfils the throughput requirements in IEEE 802.15.7.

$$T_{OM} = OCR \tag{15}$$

$$T_{RSE} = F_{max}\frac{mk}{n} \tag{16}$$

$$T_{RSD} = F_{max}\frac{mn}{2^m} \tag{17}$$

7 | CONCLUSION

In this article, the development process of a parameterizable IP to perform RS codes has been presented. It underlines how a single architectural description can handle different configurations of the intended functionality with the costs of writing a generalized HDL code. Since RS codec is an ECC method that aims to enhance reliability of data transmission systems, the

TABLE 8 Proposed IP versus Altera 'Reed–Solomon II' for EP4CE6E22C6

RS Config.	*m*	*LE*	*RE*	*MB*	F_{max}(MHz)
RS(15, 11)	4	89/43 452/345	33/34 243/236	0/0 188/192	367/125 181/153
RS(15, 7)	4	100/57 645/498	49/51 373/339	0/0 188/192	350/136 150/127
RS(15, 4)	4	115/75 752/590	61/64 454/396	0/0 188/192	330/141 160/150
RS(15, 2)	4	126/79 853/662	69/72 518/444	0/0 188/192	343/105 159/125
RS(64, 32)	8	408/339 4566/3934	279/285 2211/1764	0/0 1552/5120	243/122 94/132
RS(160, 128)	8	402/339 4611/3941	281/285 2228/1769	0/0 3856/5120	273/124 95/121

TABLE 9 T_{OM}, T_{RSE}, and T_{RSD} based on F_{max} in EP4CE6E22C6 for each RS configuration

RS Config.	*M*	OC_{max}(MHz)	T_{OM}(Mbps)	T_{RSE}(Mbps)	T_{RSD}(Mbps)
RS(15, 11)	4	0.4	0.4	1077	679
RS(15, 7)	4	0.4	0.4	653	563
RS(15, 4)	4	0.4	0.4	352	600
RS(15, 2)	4	0.4	0.4	183	596
RS(64, 32)	8	120	120	972	188
RS(160, 128)	8	120	120	1747	475

correctness of its hardware implementation is a key aspect that should not be ignored. However, verification of a RS codec is a laborious task, since its RTL model potentially has a huge state space, which results in a heavy computational problem for verification tools. Nevertheless, FV could be able to test target RS configurations captured from IEEE 802.7.15. It still faces significant challenges when parametric values are increased, but it could be mitigated by using design knowledge to create a set of properties that is adequate for FV and guarantees certain level of verification quality. Reports from synthesis analysis confirmed the expected changes in area and timing for parametric variations. Furthermore, the throughput requirements of IEEE 802.7.15 was achieved by the proposed architecture synthesised in the target FPGA device (EP4CE6E22C6). The resulting IP with its related HDL files, scripts, and RTL schematics can be found in https://github.com/mateusgs/rs_codec and is available for general use and further collaboration improvements.

ACKNOWLEDGEMENTS

The present work was supported by the following Brazilian organisations: CNPq, CAPES, and FAPEMIG. The authors thank Cadence Design Systems for providing software licences and product assistance via the University Software Programme.

REFERENCES

1. Geisel, W.A: Tutorial on Reed-Solomon error correction coding, Technical Memorandum 102162. NASA (1990)
2. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields, J. Soc. Ind. Appl. Math. 8(2), 300-304 (1960)
3. Wicker, S.B., Bhargava, V.K.: An introduction to Reed-Solomon codes, In: Wicker, S.B. (ed.) Reed-Solomon Codes and Their Applications, 1st ed., pp. 1–16 Wiley-IEEE Press (1994)
4. Immink, K.: Reed-Solomon codes and the compact disc. In: Wicker, S.B. (ed.) Reed-Solomon Codes and Their Applications, 1st ed., pp. 41–59. Wiley-IEEE Press (1994)
5. IEEE standard for local and metropolitan area networks, Part 15.7: Short-range wireless optical communication using visible light. (2011)
6. Reed, I.S., Solomon, G.: Reed-Solomon codes: a historical overview, In: Wicker, S.B. (ed.) Reed-Solomon Codes and Their Applications, 1st ed., pp. 17–24. Wiley-IEEE Press (1994)
7. Wolf, J.K.: An introduction to Reed-Solomon codes. http://pfister.ee.duke.edu/courses/ecen604/rspoly.pdf. Accessed January 2020
8. Clarke, C.K.P.: Reed-Solomon error correction, BBC R&D white paper, WHP, 31 (2002)
9. Peterson, W.: Encoding and error-correction procedures for the Bose-Chaudhuri codes, IRE Trans. Inf. Theor. 6(4), 459–470 (1960)
10. Gorenstein, D., Zierler, N.: A class of error-correcting codes in p^m symbols, Soc. Ind. Appl. Math. 9(2), 207–214 (1961)
11. Giacomelli, I.: Improved decoding algorithms for Reed-Solomon codes. (2013). arXiv preprint arXiv:1310.2473
12. Hsu, H.Y., Wang, S.F., Wu, A.Y.A.: A novel low-cost multi-mode Reed–Solomon decoder design based on Peterson-Gorenstein-Zierler algorithm, J. VLSI Signal Proc. Syst. Signal. 34(3), 251–259 (2003)
13. Berlekamp, E.: Binary BCH codes for correction multiple errors, In Berlekamp, E. (ed.) Algebraic Coding Theory, Revised ed,. pp. 176–199. World Scientific Publishing Co. (2015)
14. Massey, J.: Shift-register synthesis and BCH decoding, IEEE Trans. Inf. Theor. 15(1), 122–127 (1969)
15. Sugiyama, Y., et al.: A method for solving key equation for decoding Goppa codes, Inf. Contr. 27(1), 87–99 (1975)
16. Chien, R.: Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes, IEEE Trans. Inf. Theor. 10(4), 357–363 (1964)
17. IEEE Standard for Ethernet. IEEE Std 802.3–2018 (2018)
18. IEEE Standard for High Data Rate Wireless Multi-Media Networks. IEEE Std 802.15.3–2016 (2016)
19. IEEE standard for information technology– local and metropolitan area networks– specific requirements– Part 15.3: Amendment 2: Millimetre-wave-based alternative Physical layer extension (2009)
20. IEEE standard for air interface for broadband wireless access systems, IEEE Std 802.16–2017 (2017)

21. Dayal, P., Patial, R.K.: Implementation of Reed-Solomon CODEC for IEEE 802.16 network using VHDL code. In International Conference on reliability Optimization and Information Technology, Faridabad, India, pp 452–455 (February 2014)
22. Samanta, J., Bhaumik, J., Barman, S.: FPGA based area efficient RS (23, 17) codec. Microsyst. Technol. 23(3), 639–650 (2017)
23. Kamar, S., et al.: FPGA implementation of RS codec with interleaver in DVB-T using VHDL. Int. J. Eng. Technol. 6(4), 171-180 (2017)
24. Garg, D., et al.: High throughput FPGA implementation of Reed-Solomon encoder for space data systems. In: Nirma University International Conference on Engineering, Ahmedabad, India, pp 1–5 (2013)
25. Wu, Y.: New scalable decoder architectures for Reed–Solomon codes, IEEE Trans. Commun. 63(8), 2741–2761 (2015)
26. Ji, W., et al.: High-efficient Reed–Solomon decoder design using recursive Berlekamp–Massey architecture. IET Commun. 10(4), 381–386 (2016)
27. Liu, Y., et al.: Area-efficient reed–solomon decoder using recursive Berlekamp–Massey architecture for optical communication systems, IEEE Commun. Lett. 21(11), 2348–2351 (2017)
28. Smith, S., Taylor, D., Benaissa, M.: Design automation of Reed-Solomon codecs using VHDL, Microelectron. J. 29(12), 977–982 (1998)
29. Park, J.K., Kim, J.T.: Soft IP compiler for a Reed-Solomon decoder, ETRI J. 25(5), 305–314 (2003)
30. Yeung, P.: Getting started with static verification. In: 47th Design Automation Conference, Anaheim, United States, pp. 1–10 (2020)
31. Armstrong, R.C., et al.: Survey of existing tools for formal verification', SANDIA REPORT SAND2014-20533. (2014)
32. Seligman, E., Schubert, T., Kumar, M.A.K.: Formal verification: an essential toolkit for modern VLSI design, 1st ed., Morgan Kaufmann (2015)
33. Servadei, L., et al.: Formal verification methodology in an industrial setup. In: Euromicro Conference on Digital System Design (DSD), Kallithea, Greece, pp. 610–614 (August 2019)
34. Foster, H., et al.: Guidelines for creating a formal verification testplan. In: Design & Verification Conference, San Jose, US, pp. 1–8 (February 2006)
35. Nidhra, S., Dondeti, J.: Black box and white box testing techniques-a literature review, Int. J. Embed. Syst. Appl. 2(2), 29–50 (2012)
36. Intel. Guaranteeing Silicon performance with FPGA timing models, Report No. WP-01139-1. vol. 1 (2006)
37. A Reed–Solomon II IP Core User Guide, IP Manual. (2016)

How to cite this article: Silva MG, Silvano GL, Duarte RO. RTL development of a parameterizable Reed–Solomon Codec. *IET Comput. Digit. Tech.* 2021;15:143–159. https://doi.org/10.1049/cdt2.12009