An Automatic Technique for Optimizing Reed-Solomon Codes to Improve Fault Tolerance in Memories

Gustavo Neuberger, Fernanda Gusmão de Lima Kastensmidt, and Ricardo Reis
Federal University of Rio Grande do Sul

Editors' note:
Modern SoC architectures manufactured at ever-decreasing geometries use multiple embedded memories. Error detection and correction codes are becoming increasingly important to improve the fault tolerance of embedded memories. This article focuses on automatically optimizing classical Reed-Solomon codes by selecting the appropriate code polynomial and set of used symbols.

—Dimitris Gizopoulos, University of Piraeus; and Yervant Zorian, Virage Logic

THE TECHNOLOGICAL EVOLUTION of the IC fabrication process, consisting of device shrinking, power supply reduction, and increasing operating speeds, has significantly reduced the manufacturing yield and reliability of very deep-submicron (VDSM) ICs when various noise sources are present, as recent roadmaps (the *International Technology Roadmap for Semiconductors*, *Medea*, and *IEEE Design & Test*) have demonstrated. As a result, more and more applications must be robust in the presence of multiple faults. Consequently, fault tolerance in storage devices such as high-density and high-speed memories operating at low voltage, is a main concern nowadays and thus the focus of this work.

Faults can occur during the fabrication process, with direct consequences in terms of yield and memory operation. Using VDSM technologies increases the chances of manufacturing defects. It is important to design fault-tolerant mechanisms to ensure that a memory operates correctly, even in the presence of defects such as open and short gates and connections that can result in stuck-at faults or coupling faults in memory cells.[1]

During the memory's lifetime, another type of fault can affect its correct operation. This fault, known as a single event upset (SEU) or soft error, characteristically is a bit-flip in the memory cell as a result of a transient current pulse generated by ionization. Charged particles coming from sun activity and neutrons that collide with the material can cause this ionization.[2] More than one SEU can occur at the same time in a memory array because of the nanometer dimensions. This phenomenon, called multiple bit upset (MBU), can result from a high-energy particle (most likely causing double bit upsets) or a low incident angle (striking many cells in a row).[3] Experiments using proton and heavy-ion fluxes calculate the probability of a single ion provoking MBUs.[4-6]

Error detection and correction code (EDAC) is a well-known technique for protecting storage devices against transient faults because it is implementable in a high-level design step without changes in the mask process (that is, it has low nonrecurring engineering cost). There are examples of software[7] and hardware[8] that perform SEU mitigation using EDAC. An example of EDAC is the Hamming code, which is useful for protecting memories against SEU because of its efficient ability to correct single upsets per coded word with reduced area and performance overhead.[9] There are commercial memories that use Hamming code for high-

0740-7475/05/$20.00 © 2005 IEEE Copublished by the IEEE CS and the IEEE CASS

reliability applications (EDAC or error-correcting code memories) to increase yield or fault tolerance.[10] However, Hamming code is not able to ensure the reliability of VDSM memories when manufacturing defects and MBUs from the environment are present.

There are alternatives to EDAC, like Bose-Chaudhuri-Hocquenghem (BCH) and Reed-Solomon (RS) codes, based on finite-field (also known as Galois field) arithmetic, which can cope with multiple faults. BCH codes can correct a given number of bits at any position, whereas RS codes group the bits in blocks to correct them. Their disadvantages are that they have complex and iterative decoding algorithms, and use tables of constants in the algorithm. However, studies have proven that eliminating the tables of constants can simplify RS codes,[11] and that the decoding algorithm is simpler in the case of single-block correction. For these reasons, we chose RS instead of BCH and similar codes.

This article shows new techniques to further improve the RS core by individually optimizing the multipliers used in the RS algorithm. Recent publications in finite-field arithmetic show that the smallest multipliers are always the same for a given size of operands.[12] It's true that a best multiplier always exists for generic operands; but in the RS code algorithm, the multipliers are not generic because one of the operands is always a constant, which necessitates optimizing the multipliers for specific constants. This distinction completely changes the paradigm. The results we present here will show that the choice of the best multipliers is not generic, but depends on these constants and the target platform.

Basic concepts of RS code

RS is an error-correcting code designed to correct multiple errors. The specification of an RS code is *RS*(*n*, *k*). Each symbol (block of bits) has *s* bits. The total number of symbols used in the code is *n*, and the number of symbols used to store useful information is *k*. Consequently, the number of bits of information to be protected equals $s \times k$. The remaining symbols, $(n - k)$, are the parity symbols. Parameter *n* has a maximum value of $2^s - 1$ for a given symbol width. The code can correct number of symbols *t* equal to the number of parity symbols divided by two, where $2t = n - k$, as Figure 1 illustrates. If the code can correct only one symbol ($t = 1$), two symbols are necessary for parity ($n - k = 2$). Mathematically, the theory of RS codes is based on finite-field arithmetic.

The finite field is a sequence of numbers created from a generator polynomial (GP). It adheres to the following rules:[13]

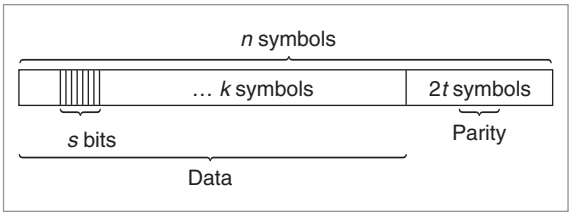

Figure 1. Reed-Solomon code word.

- This arithmetic executes all operations as modulo 2, which means that exclusive-OR (XOR) gates physically implement the addition and, consequently, subtraction operations.
- The finite field in which the operations take place bounds all operations.

GPs must be primitive. To be primitive, they must be irreducible. A polynomial is irreducible if you cannot factor it into nontrivial polynomials. For example, the polynomial $x^2 + x + 1$ is irreducible, but $x^2 + 1$ is not, because $(x + 1)(x + 1) = x^2 + 2x + 1 = x^2 + 1$ (modulo 2). However, not all irreducible polynomials are primitive. Their order must equal 2 raised to its degree minus 1, where the degree is the value of the most significant index of the polynomial, and the order is the smallest integer *e* for which the polynomial divides $x^e + 1$. For example, $x^2 + x + 1$ has order 3 and is primitive, because $(x^3 + 1) / (x^2 + x + 1) = x + 1$.

There are different ways to represent the polynomial, such as binary or decimal notations. All the notations in the following sequence denote the same polynomial:

$$x^5 + x^2 + 1 = 100101_2 = 37_{10}$$

The following example illustrates the RS algorithm with 4-bit symbols. The GP defines a finite field over which all the operations that are calculable in the following way take place: Taking a small field defined as $GF(2^4)$, use a polynomial—in this case, $x^4 + x + 1 = 0$. The term $GF(2^4)$ means that the finite field has 16 elements. The field's calculation starts with primitive element α, which in this case is 2 or 0010 (x). An increasing power of α represents each successive member of the field; hence, we call α the primitive root because its powers represent all the field's nonzero members. Table 1 shows the calculation of the field elements.

After defining the finite-field elements, it's useful to

Table 1. Calculation of the elements of GF(2^4).

Power	Calculation	Numeric value
$\alpha = x$	x	0010 (2)
$\alpha^2 = x \times x$	x^2	0100 (4)
$\alpha^3 = x \times x \times x$	x^3	1000 (8)
$\alpha^4 = \alpha \times \alpha^3$	$x^4 = x + 1$ (using the GP $x^4 + x + 1 = 0$)	0011 (3)
$\alpha^5 = \alpha \times \alpha^4$	$x^5 = x^2 + x$	0110 (6)
$\alpha^6 = \alpha \times \alpha^5$	$x^6 = x^3 + x^2$	1100 (12)
$\alpha^7 = \alpha \times \alpha^6$	$x^7 = x^4 + x^3 = x^3 + x + 1$	1011 (11)
$\alpha^8 = \alpha \times \alpha^7$	$x^8 = x^4 + x^2 + x = x^2 + 1$	0101 (5)
$\alpha^9 = \alpha \times \alpha^8$	$x^9 = x^3 + x$	1010 (10)
$\alpha^{10} = \alpha \times \alpha^9$	$x^{10} = x^4 + x^2 = x^2 + x + 1$	0111 (7)
$\alpha^{11} = \alpha \times \alpha^{10}$	$x^{11} = x^3 + x^2 + x$	1110 (14)
$\alpha^{12} = \alpha \times \alpha^{11}$	$x^{12} = x^4 + x^3 + x^2 = x^3 + x^2 + x + 1$	1111 (15)
$\alpha^{13} = \alpha \times \alpha^{12}$	$x^{13} = x^4 + x^3 + x^2 + x = x^3 + x^2 + 1$	1101 (13)
$\alpha^{14} = \alpha \times \alpha^{13}$	$x^{14} = x^4 + x^3 + x = x^3 + 1$	1001 (9)
$\alpha^{15} = \alpha \times \alpha^{14}$	$x^{15} = x^4 + x = 1$	0001 (1)

delineate the most important operations that are performable on finite-field elements. As we already mentioned, addition operations are the same as subtraction operations, and they work as XOR gates of the elements' numeric values. For example, $\alpha^5 + \alpha^6 = 0110$ XOR 1100 $= 1010 = \alpha^9$. Multiplication and division are only addition or subtraction of the elements' powers, remembering that $\alpha^{15} = 1$. For instance, $\alpha^2 \times \alpha^4 = \alpha^6$; $\alpha^{13} \times \alpha^9 = \alpha^{22} = \alpha^{15} \times \alpha^7 = \alpha^7$; $\alpha^4 / \alpha^2 = \alpha^2$.

Another fixed parameter to set in the RS code is t, the maximum number of symbols with errors that are correctable, which always equals one for this algorithm. For this example, the total number of symbols (n) is 15, and consequently, the number of information symbols (k) is 13. A single-symbol error-correcting code can correct any number of bit errors within a single symbol. When a single error occurs, there are two unknowns: the symbol position of the error within the message, and the error's bit pattern. Solving these two unknowns requires developing two simultaneous equations. In the example of GF(2^5), the block will comprise 15, 4-bit symbols. We call the first 13 symbols $A_0, A_1, A_2, \ldots$; they represent 52 bits of data (13, 4-bit symbols). We call the last two symbols R and S. These are our RS symbols, which are as yet unknown. We then generate two simultaneous equations:

$$A_0 + A_1 + A_2 + A_3 + \ldots + R + S = 0$$
$$\alpha A_0 + \alpha^2 A_1 + \alpha^3 A_2 + \alpha^4 A_3 + \ldots + \alpha^{14} R + \alpha^{15} S = 0$$

First, we solve the equations to find R and S:

$$R = \alpha A_0 + \alpha^5 A_1 + \alpha^{11} A_2 + \alpha^{13} A_3 + \ldots$$

$$S = \alpha^4 A_0 + \alpha^{10} A_1 + \alpha^{12} A_2 + \alpha^6 A_3 + \ldots$$

To decode and correct the possible received errors, we calculate the two syndromes, S_0 and S_1:

$$S_0 = A_0 + A_1 + A_2 + A_3 + \ldots + R + S$$

$$S_1 = \alpha A_0 + \alpha^2 A_1 + \alpha^3 A_2 + \alpha^4 A_3 + \ldots + \alpha^{14} R + \alpha^{15} S$$

If both S_0 and S_1 are 0, there are no errors. Otherwise, we can find error bit pattern ε and error location k as follows:

$$\varepsilon = S_0$$

$$k = S_1 / S_0$$

To correct the error, we add the pattern ε to the received symbol, in the location specified by k.

Previous work: The RS core and its application

An RS code can correct multiple faults, but its algorithm makes extensive use of tables. In our previous work,[11] we developed an efficient RS core in terms of area and timing for high-density memories. The solution removed the tables using a multiplier that multiplied numeric values instead of adding the powers of the elements. We divided the multiplication of numeric values into two steps:

- multiplying the numeric values modulo 2, and
- simplifying the result by substituting extra bits with the equivalent based on the GP.

The multiplier consists of a modulo-2 multiplier and extra AND gates and an XOR gate, depending on the polynomial used. The techniques we have developed to optimize the RS core are based on this implementation of the finite-field multiplier.

In that work,[11] we also proposed an application for the core—a memory that combines the developed RS core with Hamming code to protect 100% of double bit upsets and a large amount of MBUs. Although the results we achieved in that work were satisfactory for ICs, we noticed that we could improve the RS core with a more exhaustive optimization.

Optimization techniques

The main part of the RS encoding and decoding circuits is based on multiplications by several constants using the multiplier from our previous work.[11] Consequently, the effort to decrease the overhead in the overall circuit must pay special attention to the multipliers. There are two possible optimizations that are performable in the multiplier: choosing the most appropriate GP, and choosing the most appropriate constants for the multipliers.

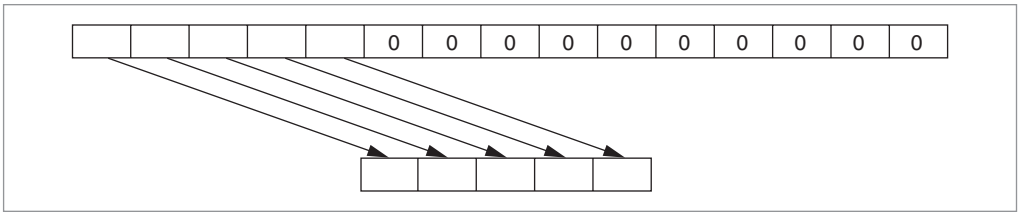

Figure 2. Simplified code without optimization.

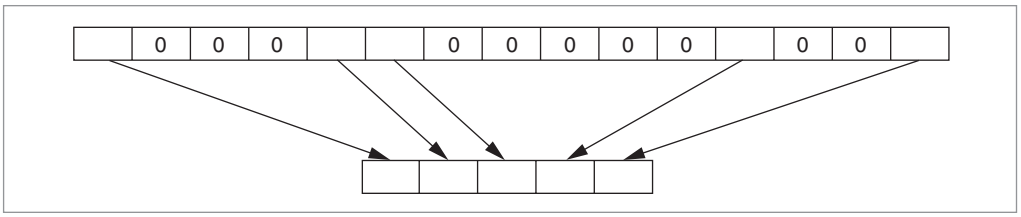

Figure 3. Simplified code with optimization.

Choosing the most appropriate GP

Assume that the user only needs to specify the number of bits to protect and the symbol width of the code. A possible optimization that a tool can perform automatically is to search all possible polynomials, create the corresponding multiplier for each polynomial, and evaluate the multipliers created in terms of area and performance. It's likely that this process will choose the simplest polynomial, because it will simplify the multiplier more than the others will. For example, to create a 5-bit multiplier, two usable polynomials are $x^5 + x^2 + 1$ and $x^5 + x^4 + x^3 + x^2 + 1$.

For the first polynomial, the constants are 00101, 01010, 10100, and 01101, whereas the constants in the second polynomial will be 11101, 00111, 01110, and 11100. The first option has a total of 11 zeros, whereas the second option has only seven. With more zeros, the first polynomial will result in a better simplification of the multiplier, because it will eliminate more AND gates.

Recent publications in finite-field arithmetic show that the smallest multipliers are based on trinomials (polynomials with fewer nonzero coefficients).[12] Chiou et al. evaluated the cost of a multiplier by the number of gates, finding the best option for a generic one. The problem with this approach is that the RS algorithm uses several multipliers, where one of the operands is always constant, and the chosen polynomial changes depending on these constants. The number of constants depends on the size of the word to be protected. In addition, the best choice of polynomial varies according to the platform target: FPGA or ASIC.

Choosing the most appropriate constants for the multipliers

The standard implementation of the RS code uses the maximum word size, $s \times n$ bits, to protect the information of $s \times k$ bits. Sometimes, however, the code word does not use all of the available $s \times n$ bits. For example, to protect 128 bits of information with a 5-bit symbol, there are 145 bits (29 symbols) available; but the code word uses only 26 of them to store the information. When a code with a smaller word is necessary, the user can use the original code and erase the extra symbols at the end of the word by replacing the exceeding bits with zeros, as Figure 2 shows.

For larger values of s, more symbols remain unused. This allows for choosing the best constant for the multipliers from the used symbols as well as the unused ones. Thus, instead of using the first symbols in order and replacing the others with zeros, the search must take into account the unused symbols to find the best constants that will generate multipliers with less area, as Figure 3 shows.

We will later show that this optimization technique by itself does not always generate the best results; it must work in conjunction with the technique for choosing the best polynomial (which we discussed earlier) to achieve the most optimized core. Because of the many possible combinations that can provide an optimal solution, we developed an optimization and generation tool, RS-OPGE, to evaluate the multipliers for

all possible symbols and to choose the ones with the best results.

Developing an automatic approach to optimizing and generating RS codes: The RS-OPGE tool

We implemented an automatic approach to optimizing RS cores using the techniques we discussed earlier. The RS-OPGE software allows the specification of the number of bits to protect, the number of bits per symbol, the polynomial to be used, and the target platform (ASIC or FPGA). The user can select from a list of possible polynomials for the given symbol width or allow the software to automatically calculate the best polynomial. Based on all parameters, the software builds the multiplier structures in the memory, propagates the possible constants to simplify them, and evaluates the circuits for an ASIC or FPGA. At the beginning of the evaluation, the software creates general multipliers for all possible polynomials for the given symbol width. It creates the multipliers using a cell-gate-net model. The cells represent logic cells like AND and XOR, as well as inputs and outputs. The nets represent wires interconnecting the cells. The gates don't have a physical representation, but they interface one cell with one net to facilitate manipulating the circuit in the program memory and to decrease the computation time.

After creating the general multipliers, RS-OPGE uses instances of them to propagate the available constants. It then evaluates the final circuits. For an ASIC target, it computes only the logic cells used, counting the number of transistors required to implement the logic. For example, for each 2-input AND cell in the circuit, the algorithm adds 6 transistors to the count; for a 3-input XOR cell, it counts 12 transistors, reducing the cell into two 2-input XORs. For each polynomial, the software chooses the constants that generate multipliers with the fewest transistors required for implementation. RS-OPGE uses the polynomial with the lowest final transistor count for generating VHDL code.

When the chosen target is FPGA instead of ASIC, the steps are almost the same, but the function that evaluates the multipliers with propagated constants is different. Instead of counting the number of transistors, it counts the number of 4-input lookup tables (LUT4s) used to implement the logic. The way to compute the cost for an FGPA has some peculiarities, because the implementation of a very simple logic can consume the same number of LUT4s as a very complex logic. This happens because a LUT4 can implement any logic consisting of 4 inputs and 1 output, regardless of the complexity. To compute it, the generator looks at the number of inputs and outputs in the multiplier circuits and computes the number of LUT4s to implement the logic, without evaluating the circuit's complexity.

The software then generates a VHDL description for the RS code implementation that is the most area efficient for the chosen target.

Results

The RS-OPGE tool allowed for more accurate comparisons between previously manually implemented cores and more complex optimized cores; this capability permits us to evaluate the proposed techniques' efficiency. We base our results on the synthesis of the generated VHDL code into a VirtexE 600 FPGA from Xilinx. We measured the area cost in lookup tables, the smallest logic units presented in the FPGA. The FPGA characterizes four LUTs combined with two flip-flops and a set of small glue logic as two slices presented in one configurable logic block (CLB). The performance estimation comes from the timing report. The delay depends on the number of slices (LUTs) in cascade and the routing.

RS code cost improvement based on proposed optimization techniques

The first experiment aims to show that the best choice of polynomial is not always based on the simplest trinomial—for example, the polynomial 131. We performed an experiment with different word sizes, using the same number of bits in the symbols, to check whether the chosen polynomial is the same in each case, and whether the polynomial optimization technique matters in the final result. Table 2 shows the chosen polynomials for each different word size to achieve the best result in terms of area. We obtained these results using RS-OPGE. In about half the cases, RS-OPGE chose the simplest trinomial (131) that generates the best cost for a generic multiplier. However, for many other word sizes, other polynomials can yield lower cost for the multipliers, depending on the word size. This leaves open the possibility of better solutions, including the search for the best polynomial.

As explained earlier, there are two optimization techniques that we can apply to RS code to reduce multiplier cost: finding the best polynomial, and finding the best constant for the multipliers. We applied these optimization techniques to case studies of RS codes that do not use the maximum word size, which leaves symbols unused and allows a higher flexibility of optimization. Table 3 shows the results.

The first technique shown in Table 3 uses the simplest trinomial (131), the second one chooses the best polynomial, the third one chooses the best constants among the symbols, and the last technique combines the previous two techniques (A and B). Some results show that using the simplest trinomial achieved the best area cost, whereas other cases show that combining techniques A and B (as shown in Table 3) achieved the best trade-off. Note that combining the techniques considerably reduces the area. Although it seems that using only the simplest trinomial achieved the best results in three of the eight cases (always in decoders), the tool always has the best results for the sum of encoder and decoder results of the same code, which is the optimization goal. This observation shows the efficiency of the proposed technique and the importance of having the RS-OPGE tool optimize and generate the RS core. In summary, there is a margin in each RS code implementation for optimization, and the RS-OPGE tool provides a way to easily test and generate the best approximate result, which manual methods could not achieve in most cases.

To show that the RS-OPGE tool's achieved results are correct and similar to a manual optimization, we compared previously manually optimized RS cores with the RS-OPGE results. Table 4 shows these implementation results in the case study of an RS code using 7-bit symbols in a 112-bit word. The first version of the RS code is the one presented in our previous work;[11] the results for this initial version appear in Table 3 of this earlier work, labeled as NEU03. We used this version in the final implementation of an MBU-tolerant memory and designed it manually, without optimizations. The sec-

Table 2. Polynomials chosen for different word sizes using a Reed-Solomon code with 7-bit symbols.

Word size (bits)	No. of constants	Polynomial
7	3	193
14	6	193
28	12	193
56	24	131
112	48	131
224	96	131
350	150	131
700	300	131
875	375	137

Table 3. Comparison of optimization techniques in Reed-Solomon codes using the RS-OPGE tool.

	Optimization techniques							
	Using simplest trinomials		Choosing polynomial (A)		Choosing constants (B)		Combining techniques A and B	
RS codes*	Polynomial	No. of LUTs/ Delay (ns)	Polynomial	No. of LUTs/ Delay (ns)	Polynomial	No. of LUTs/ Delay (ns)	Poly	No. of LUTs/ Delay (ns)
5, 32, ENC	37	45 / 7.0	41	41 / 6.0	37	33 / 5.7	37	33 / 5.7
5, 32, DEC	37	110 / 14.5	41	113 / 13.9	37	114 / 15.9	37	114 / 15.9
ENC + DEC (area only)	37	155 / NA	41	154 / NA	37	147 / NA	37	147 / NA
5,64, ENC	37	75 / 7.1	47	77 / 8.1	37	75 / 7.1	47	71 / 7.2
5,64, DEC	37	186 / 16.6	47	174 / 17.3	37	185 / 15.5	47	175 / 17.0
ENC + DEC (area only)	37	261 / NA	47	251 / NA	37	260 / NA	47	246 / NA
7, 64, ENC	131	98 / 9.1	137	90 / 10.6	131	65 / 6.0	131	65 / 6.0
7, 64, DEC	131	267 / 18.3	137	256 / 18.3	131	276 / 18.4	131	276 / 18.4
ENC + DEC (area only)	131	365 / NA	137	346 / NA	131	341 / NA	131	341 / NA
7, 128, ENC	131	151 / 11.3	137	167 / 10.5	131	150 / 7.0	137	153 / 7.2
7, 128, DEC	131	434 / 20.1	137	400 / 19.6	131	434 / 22.5	137	415 / 19.0
ENC + DEC (area only)	131	585 / NA	137	567 / NA	131	584 / NA	137	568 / NA

* RS codes are denoted by symbol width (bits), word size (bits), and mode (encoder, decoder, or both).

Table 4. Area and performance comparison between codes generated manually and automatically. We evaluate area in terms of the number of lookup tables (LUT4s) and the reduction percentage; we evaluate performance in terms of delay and the number of slices in the critical path.

	NEU03		Manually optimized		RS-OPGE tool		RS-OPGE tool (free choice of polynomial)	
Parameter	**Encoder**	**Decoder**	**Encoder**	**Decoder**	**Encoder**	**Decoder**	**Encoder**	**Decoder**
No. of LUT4s	215	538	140	402	134	392	129	370
Difference from NEU03 (%)	NA	NA	–35	–25	–37	–27	–39	–31
Delay (ns)	7.25	47.6	7	30.5	7	30.5	6.9	30.4
No. of cascaded slices	9	33	8	22	8	22	8	22

Table 5. Comparison between codes generated manually and automatically (using the RS-OPGE tool) for the maximum word size.

	Manually optimized				RS-OPGE tool			
	5 bits		7 bits		5 bits		7 bits	
Parameter	**Encoder**	**Decoder**	**Encoder**	**Decoder**	**Encoder**	**Decoder**	**Encoder**	**Decoder**
No. of LUT4s	194	358	2,161	2,809	184	347	2,152	2,832
Delay (ns)	9.2	20.7	9.6	29.9	9.2	20.7	9.46	30.5
No. of cascaded slices	7	13	8	22	7	13	8	22

ond version (also generated manually) evolved from the first one. We carefully chose the constants used, reducing area overhead using the concept that polynomials with more zero constants generate a more optimized core. We designed these two versions using the same polynomial (137). Using that same polynomial, the RS-OPGE tool automatically generated the third version. The constants that the tool chose were not the same as those for the manually designed second version; it seemed possible to achieve a reduction in area with the RS-OPGE tool. The fourth version was also automatically generated, but without the restriction of using the same polynomial. In this case, the tool chose 131 (the simplest trinomial). This choice represented the best trade-off between the polynomial and the constants for the multipliers in the RS code.

Table 5 shows the results of using the maximum word size. In this case, it is necessary to use all available constants, leaving the best-polynomial search as the only optimization. The table shows manually implemented versions with 5- and 7-bit symbols; they use the polynomials 37 and 137. The 5-bit version has 145 information bits, and the 7-bit version, 875. In this case, the only technique that you can use to optimize the code is to search all possible polynomials. The RS-OPGE has automatically chosen the polynomials 47 and 157.

The automatic approach successfully generated better results for both encoder and decoder blocks of the 5-bit version, as Table 5 shows. However, the manual optimization method achieved better results for the decoder block of the 7-bit version. This happened because the RS-OPGE tool accounts for only the multiplier costs, leaving the other circuitry out of the evaluation process. For the encoder block, this approximation is very good, because the only components unaccounted are XOR gates at the multiplier outputs, which have the same cost in all encoders. The decoder block needs one divider circuitry after these XOR gates, which can have different costs for each chosen polynomial. The RS-OPGE tool estimates a similar area for different chosen polynomials. Computing the divider circuitry in the optimization process can produce even better results. We will consider this in the next version of the tool.

Evaluating the time complexity of the RS-OPGE tool's algorithm

To evaluate the speed of the algorithms, we conducted two experiments. The first one changes the symbol width, which represents the order of the polynomial for a fixed word size; and the second one changes the word size for a fixed symbol width. Tables 6 and 7 show the execution times.

These tables show that the increase in execution time arises mainly from symbol width *s*. However, even for a

symbol width of 10 bits, the time is less than 20 minutes. That width can protect up to 10,000 bits per word—much more than current applications demand. The generation time is also considerably less than the time required for synthesizing the generated VHDL code in a commercial synthesis tool.

Table 6. Execution times for variable symbol width and fixed word size.

Symbol width (bits)	Time (s)
5	6
6	6
7	21
8	35
9	226
10	1,039

Table 7. Execution times for fixed symbol width and variable word size.

Word size (bits)	Time (s)
28	16
56	20
112	22
224	28
448	71
875	81

Comparing commercial RS code and RS-OPGE-generated cores

We compared a commercial RS core with the proposed algorithm implemented in the RS-OPGE tool. The commercial core is the Xilinx Reed-Solomon Encoder and Decoder Logicore version 4.1. The first difference between the cores is that the Xilinx core operates in serial mode (it receives one symbol per clock cycle and calculates the result using several clock cycles), whereas the developed core operates in parallel mode (it receives all data and calculates the result in the same cycle). Consequently, the Xilinx core has few changes in area for the different codes, but it has drastic increases in computation times. In the developed core, the area varies between the codes, but the computation time is almost the same, which represents a big benefit.

Table 8 shows a comparison of three different codes. In general, the Xilinx core optimizes area, and our solution optimizes performance. We evaluate performance by the total computation time each approach achieves, which in fact gives the total time required to encode or decode a given word. The delays in this case are only considered to measure the maximum clock cycle. For small RS codes with word sizes less than 200 bits, the proposed RS-OPGE-optimized RS code core has an area comparable to that of the commercial core and a much higher performance. For larger codes, there is a trade-off between area and performance. For example, in the last code presented in the table, 7-bit symbols and an 875-bit word, the proposed RS-OPGE-generated code has a 7 times larger area then the Xilinx core, but the computation time is more than 60 times faster.

FUTURE WORKS INCLUDE the development of techniques to compare the overhead for codes with different parameters, allowing a higher flexibility to the user, and also the inclusion of other EDAC codes for automatic generation and comparison with existing codes. Modern ICs will have to incorporate these complex codes to give the degree of reliability needed in secure systems designed in VDSM technologies. ■

Table 8. Comparison of codes automatically generated using the Xilinx code generator and the RS-OPGE tool. We measure area by number of LUTs and number of slices, and evaluate performance by computation time. To calculate CT, we multiply the delay of each cycle by the number of cycles for encoding and decoding.

	Xilinx core generator					RS-OPGE tool			
	Area				Computation	Area			
RS codes*	(no. of LUT4s)	(no. of slices)	Delay (ns)	No. of cycles	time (ns)	(no. of LUT4s)	(no. of slices)	Delay = CT (ns)	No. of cycles
5, 145, ENC	80	61	7.41	32	237.12	184	92	9.2	1
5, 145, DEC	348	213	12.33	50	616.5	347	174	20.7	1
7, 112, ENC	88	68	7.61	19	144.4	129	65	6.9	1
7, 112, DEC	469	292	13.31	37	492.47	370	185	30.4	1
7, 875, ENC	90	69	7.61	128	974.08	2,152	1,076	9.46	1
7, 875, DEC	456	281	13.31	146	1,943.26	2,832	1,416	30.5	1

* RS codes are denoted by symbol width (bits), word size (bits), and mode (encoder, decoder).

Acknowledgments

We thank CNPq Brazilian Agency for their support of this work.

References

1. S. Hamdioui, *Testing Multi-Port Memories, Theory and Practice*, PhD thesis, Dept. Computer Eng., Delft University of Tech., 2001.
2. A. Johnston, "Scaling and Technology Issues for Soft Error Rates," *Proc. 4th Ann. Research Conf. on Reliability*, Stanford Univ., 2000; http://parts.jpl.nasa.gov/docs/Scal-00.pdf.
3. R.A. Reed et al., "Heavy Ion and Proton Induced Single Event Multiple Upsets," *IEEE Trans. Nuclear Science*, vol. 44, no. 6, Dec. 1997, pp. 2224-2229.
4. F. Wrobel et al., "Simulation of Nucleon-Induced Nuclear Reactions in a Simplified SRAM Structure: Scaling Effects on SEU and MBU Cross Sections," *IEEE Trans. Nuclear Science*, vol. 48, no. 6, Dec. 2001, pp. 1946-1952.
5. K. Johansson et al., "Neutron Induced Single-Word Multiple-Bit Upset in SRAM," *IEEE Trans. Nuclear Science*, vol. 46, no. 6, Dec. 1999, pp. 1427-1433.
6. S. Buchner et al., "Investigation of Single-Ion Multiple-Bit Upsets in Memories on Board a Space Experiment," *IEEE Trans. Nuclear Science*, vol. 47, no. 3, June 2000, pp. 705-711.
7. P. Shirvani, N. Saxena, and E. McCluskey, "Software Implemented EDAC Protection Against SEUs," *IEEE Trans. Reliability*, vol. 49, no. 3, Sept. 2000, pp. 273-284.
8. G. Redinbo, L. Napolitano, and D. Andaleon, "Multibit Correcting Data Interface for Fault-Tolerant Systems," *IEEE Trans. Computers*, vol. 42, no. 4, Apr. 1993, pp. 433-446.
9. R. Hentschke et al., "Analyzing Area and Performance Penalty of Protecting Different Digital Modules with Hamming Code and Triple Modular Redundancy," *Proc. 15th Symp. Integrated Circuits and Systems Design* (SBCCI 02), IEEE CS Press, 2002, pp. 95-100.
10. K. Gray, "Adding Error-Correcting Circuitry to ASIC Memory," *IEEE Spectrum*, vol. 37, no. 4, Apr. 2000, pp. 55-60.
11. G. Neuberger et al., "Multiple Bit Upset Tolerant SRAM Memory," *ACM Trans. Design Automation Electronic Systems*, vol. 8, no. 4, Oct. 2003, pp. 577-590.
12. C. Chiou et al., "Low-Complexity Finite Field Multiplier Using Irreducible Trinomials," *Electronics Letters*, vol. 39, no. 24, Nov. 2003, pp. 1709-1711.
13. A.D. Houghton, *The Engineer's Error Coding Handbook*, Chapman & Hall, 1997.

Gustavo Neuberger is a PhD student at the Institute of Informatics of the Federal University of Rio Grande do Sul in Porto Alegre, Brazil. His research interests include fault tolerance, radiation effects, DFT, and SRAM memories. Neuberger has a BS in computer engineering from the Federal University of Rio Grande do Sul. He is a member of the ACM.

Fernanda Gusmão de Lima Kastensmidt is a professor at the Institute of Informatics of the Federal University of Rio Grande do Sul in Porto Alegre, Brazil. She is also a collaborative professor in the Department of Digital Systems Engineering at State University of Rio Grande do Sul. Her research interests include VLSI testing and design, fault effects, fault-tolerant techniques, and programmable architectures. Kastensmidt has a BS in electrical engineering, and an MS and a PhD in computer science and microelectronics from the Federal University of Rio Grande do Sul. She is a member of the IEEE.

Ricardo Reis is a professor at the Institute of Informatics of the Federal University of Rio Grande do Sul, and the Latin America liaison for *IEEE Design & Test*. His research interests include VLSI design, CAD, physical design, design methodologies, and fault-tolerant techniques. Reis has a BSc in electrical engineering from the Federal University of Rio Grande do Sul, and a PhD in computer science and microelectronics from the Institut National Polytechnique de Grenoble, France. He is a vice president of the International Federation for Information Processing and a member of the IEEE.

■ Direct questions and comments about this article to Gustavo Neuberger, Av. Venâncio Aires 177/605, Bairro Cidade Baixa, Porto Alegre – RS – Brazil, 90040-191; neuberg@inf.ufrgs.br.