ECC Design for Fault-Tolerant Crossbar Memories: A Case Study

Nor Zaidi Haron[1,2] Said Hamdioui[1] Zaiyan Ahyadi[1]
[1] Computer Engineering Laboratory, Delft University of Technology, The Netherlands
[2] Faculty of Electronics and Computer Engineering, Univeristi Teknikal Malaysia Melaka, Malaysia
{N.Z.B.Haron, S.Hamdioui}@tudelft.nl, zaidi@utem.edu.my

***Abstract*— Crossbar memories are promising memory technologies for future data storage. Although the memories offer trillion-capacity of data storage at low cost, they are expected to suffer from high defect densities and fault rates impacting their reliability. Error correction codes (ECCs), e.g., Redundant Residue Number System (RRNS) and Reed Solomon (RS) have been proposed to improve the reliability of memory systems. Yet, the implementation of the ECCs was usually done at software level, which incurs high cost. This paper analyzes ECC design for fault-tolerant crossbar memories. Both RS and RRNS codes are implemented and experimentally compared in terms of their area overhead, speed and error correction capability. The results show that the encoder and decoder of RS requires 7.5× smaller area overhead and operates 8.4× faster as compared to RRNS. Both ECCs has fairly similar error correction capability.**

I. Introduction

The quest for new memory technology that can provide further scalability, yet able to tolerate reliability failures has made fault tolerance as one of the key requirements [1]–[6]. Crossbar memory is one of the emerging new memory technologies able to offers trillion-capacity of data storage at low power consumption and reduced fabrication cost. However, these advantages do not come for free as several challenges need to be resolved [3]. One of the challenges is that the memories are likely to suffer from high defect densities and fault rates impacting their reliability.

In order to improve the reliability of crossbar memories, *error correction codes (ECCs)* such as Hamming, Low-Density Parity-Check (LDPC) and Bose-Hocquengham-Chaudhuri (BCH) codes [7]–[11] have been proposed. According to [5], [6], defects and faults in crossbar memories tend to induce *cluster errors*; hence, ECCs able to correct such errors, such as RS [12] and RRNS [13]–[15], are required. Traditionally, these ECCs have been implemented using software resulting in low performance; this make such implementation unsuitable for scalable yet unreliable crossbar memories.

This paper studies ECC design for fault-tolerant crossbar memories. The encoder and decoder of both RS and RRNS are designed and implemented. An evaluation in terms of their area overhead and decoding speed as well as error correction capability is carried out. The evaluation shows that the encoder and decoder of RS requires smaller area and operates faster as compared to that of RRNS. Moreover, both ECCs can correct almost equivalent numbers of errors.

The rest of the paper is organized as follows. Section II gives the background of crossbar memories and error correction codes. Section III presents the theory of RS and RRNS that are used in our work. Section IV explains the design of the encoder and decoder for both ECCs. Section V analyzes and compares the area overhead, speed and error correction capability of the considered ECCs. Section VI concludes this paper.

II. Background

This section gives the background required to further understand the paper. It starts with explaining crossbar memories, thereafter error correction codes.

A. Crossbar Memories

Figure 1(a) shows one of the crossbar memories referred to as *CMOS/Molecular (CMOL)* memory [7], [8]. CMOL memory provides the utmost data storage capacity as huge as 1Tbit/cm^2, which is about three magnitude denser than the existing semiconductor memories. In addition to the data storage, the novelties of this hybrid memory are: (i) the memory array is stacked above the peripheral circuits (3D stacking IC instead of planar IC), and (ii) the memory array are formed by *non-CMOS* devices instead of CMOS and/or capacitor.

The memory array consists of nanowire crossbars with reconfigurable two-terminal nanodevices embedded at each crosspoint. Because non-CMOS-based devices are incapable to perform the periphery tasks (e.g., sensing, amplification, etc.), nanoscale CMOS is required to structure the peripheral circuits [7], [8]. Two sets of CMOS-to-nano (CtN) interface pins connect the memory array to the peripheral circuits; see Fig. 1(b). These CtN interface pins are different in height such that the short pins connect the lower nanowires, while the tall pins connect the upper nanowires.

In order to write to and read from the memory, a sufficient voltage is biased across the targeted two-terminal nanodevices (memory cells) from the CMOS-based peripheral circuits through the CtN interface pins to the corresponding nanowires [7], [8]. For writing, the voltage must be larger than the threshold voltage of the two-terminal devices to turn them on (represent 1) and smaller to turn them off (represents 0). For reading, a smaller voltage is used. Note that the value of the voltages depends on the two-terminal nanodevices used as the memory cells [7].

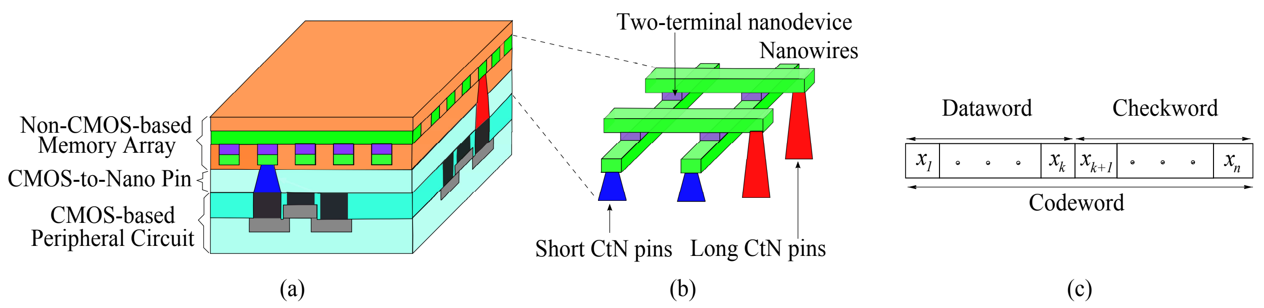


Fig. 1. Schematic of (a) CMOL memory architecture (b) CMOL memory array (c) ECC

B. *Error Correction Codes*

Error correction codes are formed by a group of codewords. As shown in Fig. 1(c), a codeword *C*=$\{x_1,...,x_k,x_{k+1},...,x_n\}$ comprises of a k-element of dataword and $(n-k)$-element of checkword where n and k are integer [17]. Here, the element x_i; $1 \leq i \leq n$, can be either a number of bits (for bit-oriented ECCs) or a number of symbols (for symbol-oriented ECCs); a symbol is a set of bits. The dataword represents the input data, whereas the checkword denotes the required extra elements for error detection or/and correction. Generally, the number of elements required for correction is twice as many as that for detection.

Depending on their types, whether bit-oriented or symbol-oriented, ECCs can be classified into two groups. Bit-oriented means that the ECCs operate in *bit by bit* basis during the encoding and decoding. Because of the bit-oriented characteristic, these ECCs are suitable to tolerate *random* faults. The ECCs that belong to this class include Hamming, BCH and LDPC [17]. Contrarily, symbol-oriented ECCs operate in *a group of bits by a group of bits* basis during the encoding and decoding. Due to the symbol-oriented characteristic, these ECCs are suitable to correct *cluster* faults, which are the case for crossbar memories. The ECCs that fall into this group are RS and RRNS. While RS composes of *fixed-length* symbols, RRNS consists of *varied-length* symbols. More descriptions on these two ECCs will be given in the next section.

III. ECCs for Crossbar Memories

This section explains the encoding and decoding theory of RS and RRNS ECCs.

A. *RS Code Theory*

An n-symbol RS codeword consists of k-symbol dataword and $(n-k)$-symbol checkword where n and k are integer [12], [17]. Each symbol is generated based on *Galois Field* GF(2^m) where m is the number of bits in each symbol. The correction capability of this code is defined as $t=\frac{(n-k)}{2}$. For example, two symbols are appended as the checkword to correct a single erroneous symbol.

To encode RS code, input data X is multiplied with a primitive polynomial. This primitive polynomial is selected in such a way that it cannot be factorized into smaller polynomial to ensure the encoding and decoding consistency (the unique relationship between input/output data and RS codeword). The resulting product is then appended to X to produce an RS codeword *C*.

To decode RS code, the read codeword is validated by checking syndrome S_i; it can be expressed as [12]:

$$S_i = \sum_{j=0}^{n-1} C_j (\alpha^v)^j \tag{1}$$

where C_j is the codeword symbol, α^v is the primitive polynomial roots and $1 \leq v \leq 2t$. If S_i=0, then the RS codeword is error-free and is read out. In contrast if $S_i \neq$0, then the read codeword has errors and requires correction.

Several algorithms can be used to correct errors in RS codeword, e.g., *Peterson-Gorenstein-Zierler (PGZ), Berlekamp-Massey, etc* [13], [17]. PGZ provides a low computational complexity for small t values as compared to other algorithms, e.g., *Berlekamp-Massey*, which are preferable for large t values [12]. In this work PGZ will be used as it suits our experiment.

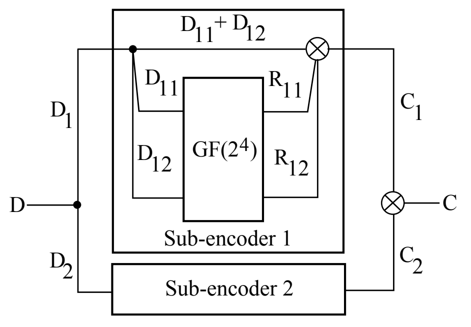
B. *RRNS Code Theory*

RRNS code has a similar structure and the same error correction capability as RS code; yet in theory, the symbol is usually referred to as *residue* [13], [18]. The residues in RRNS code might have different bit length b, depending on the *moduli* used, i.e., $b=\lfloor log_2(moduli-1)+1 \rfloor$ bits.

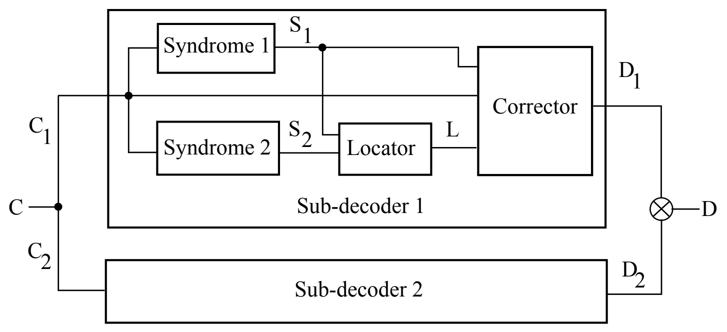
To encode RRNS code, input data X is divided by a set of moduli m_i where $1 \leq i \leq n$; n is the number of symbols of the codeword. The remainder of the division of X by the moduli results in the dataword and checkword. In contrast to RS that relies on Galois Field for encoding and decoding consistency, RRNS depends on three different rules. Briefly, these three rules are: (i) the moduli set must be mutually co-prime, (ii) the succeeding modulus must be larger than its preceding, and (iii) their product must be larger than the operating legitimate range of 2^d-1 where d is input data length [14], [18].

To decode RRNS code, a similar steps as for RS is performed. The read codeword is first validated by checking its syndromes. Two algorithms can be utilized for decoding: *Mixed-Radix Conversion (MRC)* or *Chinese Remainder Theorem (CRT)* [13], [18]. MRC is used in this work because it require simple design and easier to be optimized. The calculated codeword referred to as *syndrome* can be expressed as follows [13], [18]:

$$S_i = | \left(\left(\left(\left(x_i - S_1\right) \times g_{1i}\right) \ldots - S_{(i-1)}\right) \times g_{(i-1)i}\right) |_{m_i} \tag{2}$$



(a)



(b)

Fig. 2. Block diagram of RS (a) encoder (b) decoder

where $g_{(i-u)i}$ is the multiplicative inverse of $m_{(i-u)}$ with respect to m_i defined as $|m_{(i-u)}g_{(i-u)}|_{m_i}$=1; 2$\leq i \leq n$ and 1$\leq u \leq n-$1.

After reading, if S_{imax}=0 where $imax$ refers to the largest syndrome in each iteration (more detail in Section IV), then the read codeword is error-free and is converted into binary prior to read out; otherwise a correction takes place. As in RS, the correction procedure for RRNS is quite complex, [13], [18] can be referred for further theory and explanation.

IV. Encoder and Decoder Design

This section explains the design of the encoder and decoder of RS and RRNS ECCs.

RS encoder and decoder design

The RS encoder is designed in such a way that, e.g., 16-bit input data D, will be encoded into two-symbol datawords D_1 and D_2; each consist of 8 bits. For this purpose, GF(2^8) is chosen, meaning that each symbol comprises of 8 bits. Moreover, the decoder is set to correct one erroneous symbol t=1, or a maximum of 8 bits cluster error. Therefore, the RS codeword needs $(n-k)$=2t=2 symbols as the checkword.

However, because GF(2^8) may results in a complex conversion from binary to GF element and the way around, GF(2^4) is used instead; this will result in a simple design without impacting the error correction capability [16]. This means that each 8-bit data is further divided into two sub-group; each composes of 4 bits. D_1 becomes two sub-datawords D_{11} and D_{12}, while D_2 turns into another two sub-datawords D_{21} and D_{22}. This is also applied to the checkword; R_1 becomes two sub-checkwords R_{11} and R_{12}, while R_2 turns into another two sub-checkwords R_{21} and R_{22}.

The used primitive polynomial for encoding is $GF(2^4)$=x^4+s^3+1. This $GF(2^4)$ consists of a successive power of the polynomial roots α^v, i.e., $\{0, \alpha^0, \alpha^1, \alpha^2, ..., \alpha^{14}\}$ [17]. Each α^v has its binary representation that can be pre-calculated using polynomial generator.

The following algorithms are used to design the encoder of RS [19]:

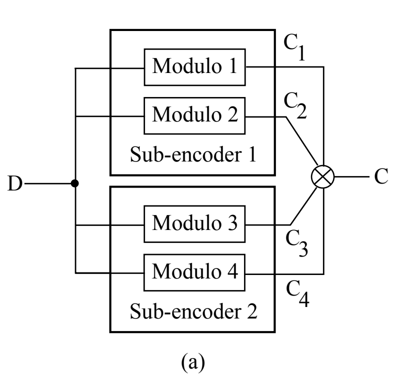
1) Generate $GF(2^4)$ elements.
2) Split input data D, into 4-bit sub-group.
3) Calculate R_{11} and R_{12} by solving two orthogonal equations as follows:
$$D_{11} + D_{12} + R_{11} + R_{12} = 0$$
$$\alpha^1 D_{11} + \alpha^2 D_{12} + \alpha^3 R_{11} + \alpha^4 R_{12} = 0$$
4) Append R_{11} and R_{12} to D_{11} and D_{12} to form the sub-codewords C_1.
5) Repeat the third and fourth steps to form the sub-codewords C_2.

Figure 2(a) shows the block diagram of the RS encoder in which two sub-encoders operate in parallel [16]. Each sub-encoder receives D_1 and D_2, respectively, which are further split into a smaller group of 4-bit data. For example the first sub-encoder, the split data becomes D_{11} and D_{12}; at the same time, it is multiplied to $GF(2^4)$ roots α^v, resulting in R_{11} and R_{12}. These sub-datawords and sub-checkwords are concatenated producing a sub-codeword C_1. A similar process is performed by the second sub-encoder. Finally, both sub-codewords C_1 and C_2 are concatenated yielding a b-bit (or n-symbol) RS codeword C.

The following algorithms are used to design the decoder of RS (see 2(b)) [19]:

1) Split the read C into two sub-codewords C_1 and C_2.
2) Calculate both syndromes S_1 and S_2 as follows:
$$S_1 = D_{11} \oplus D_{12} \oplus R_{11} \oplus R_{12}$$
$$\mathrm{S}_2 = \alpha^1 D_{11} \oplus \alpha^2 D_{12} \oplus \alpha^3 R_{11} \oplus \alpha^4 R_{12}$$
3) Define error location by dividing S_1 by S_2, i.e., L=$\frac{S_1}{S_2}$.
4) If L≠1 or L≠2, both D_{11} and D_{21} are error-free.
5) If L=1, then D_{11} is erroneous; correction is performed by XORing D_{11} with S_1.
6) if L=2, then D_{12} is erroneous; correction is performed by XORing D_{12} with S_2.

Figure 2(b) illustrates the block diagram of the RS decoder in which two sub-decoders operate in parallel [16]. Each sub-decoder comprises of two syndrome units *Syndrome1* and *Syndrome2*, an error locator unit *Locator*, and a corrector unit *Corrector*. The syndrome units, which validates the read codeword, are formed by an array of XOR gates. Their outputs



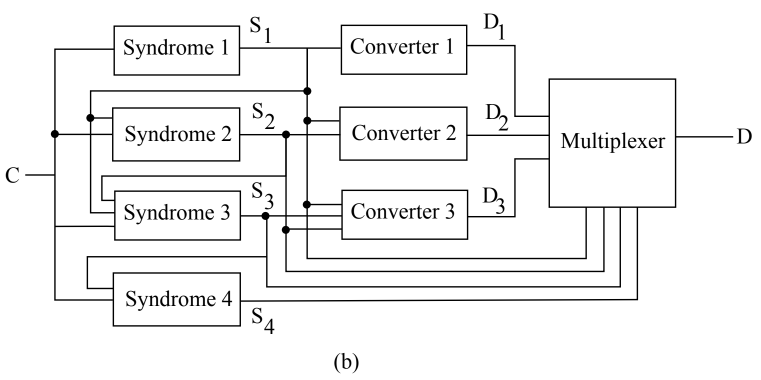


Fig. 3. Block diagram of RRNS (a) encoder (b) decoder

become the inputs to *Locator*, which is structured by a look up table (LUT) storing a pre-calculated *GF* roots α^v. The outputs of *Locator* and the outputs of *Syndrome1* unit then become the inputs to the corrector, which is structured by *XOR* gates and multiplexer. Finally, the outputs of the sub-decoders are concatenated creating the output data.

RRNS encoder and decoder design

The RRNS encoder and decoder are designed based on four moduli m_i=$\{2^{\frac{d}{2}}, 2^{\frac{d}{2}}+1, 2^{\frac{d}{2}+1}-1, 2^{\frac{d}{2}+1}+1\}$ where d is input data length. The moduli set comprises of *low-cost moduli*, which realizes small and fast RNS-based arithmetic circuits [13]. Such moduli are selected because the resulting residues have a fairly similar codeword length to that of RS symbols. E.g., for d=16 bits, the dataword length is b=$\lfloor log_2(m_1-1)+1\rfloor + \lfloor log_2(m_2-1)+1\rfloor = \lfloor log_2(256-1)+1\rfloor + \lfloor log_2(257-1)+1\rfloor$=17 bits. Two residues are set as the checkword to provide a single residue correction.

The following algorithms are used to design the encoder of RRNS (see Fig. 3(a)) [20]:

1) Calculate C_1 by taking the $\frac{d}{2}$ least significant bits of D.
2) Calculate C_2 by first dividing D into two groups B_1 and B_2; B_1 is the $\frac{d}{2}$ least significant bits, whereas B_2 is the $\frac{d}{2}$ most significant bits. Then, C_2 is obtained from B_1-B_2 if ($B_2<B_1$), else x_2=$(2^{\frac{d}{2}}+1)+B_1-B_2$.
3) Calculate C_3 by first dividing D into two groups B_1 and B_2 as in Step 2. Then, C_3 is obtained from B_1+B_2.
4) Calculate C_4 by first dividing D into two groups B_1 and B_2 as in Step 2. Then, C_4 is obtained from B_1-B_2 if ($B_2<B_1$), else $(2^{\frac{d}{2}+1}+1)+B_1-B_2$.

Figure 3(a) shows the block diagram of the RRNS encoder where two sub-encoders operate in parallel [16]. The first sub-encoder consists of two modulo units; each is based on $2^{\frac{d}{2}}$ and $2^{\frac{d}{2}}+1$. The second sub-encoder also comprises of two modulo units; each is based on $2^{\frac{d}{2}+1}-1$ and $2^{\frac{d}{2}+1}+1$. The modulo units may consist of either simple buffer, or more complex circuits (e.g., adders and multiplexers) depending on the moduli set they operate. E.g., the first modulo circuit is formed by a $\frac{d}{2}$-bit buffer. The third modulo is structured of adders and multiplexers. However, the second and fourth modulo units require additional subtracters besides adders and multiplexers. The modulo units runs in parallel producing the corresponding residues, which in turn are concatenated producing a b-bit (n-symbol) RRNS codeword *C*.

The following algorithms are used to design the decoder of RRNS [20]:

1) Calculate S_1, S_2 and S_3 by discarding x_4 and using the remaining residues x_1, x_2 and x_3.
$S_1 = x_1$
$S_2 = |(x_2 - S_1) \times g_{12}|_{m_2}$
$S_3 = |((x_3 - S_1) \times g_{13} - S_2)) \times g_{23}|_{m_3}$
If S_3=0, calculate output data X=S_1+S_2×m_1+S_3×m_1×m_2, otherwise
If $S_3\neq 0$, go to next step
2) Calculate S_1, S_2 and S_4 by discarding x_3 and using the remaining residues x_1, x_2 and x_4
$S_1 = x_1$
$S_2 = |(x_2 - S_1) \times g_{12}|_{m_2}$
$S_4 = |((x_4 - S_1) \times g_{14} - S_2) \times g_{24}|_{m_4}$
If S_4=0, calculate output data X=S_1+S_2×m_1+S_4×m_1×m_2, otherwise
If $S_4\neq 0$, go to next step
3) The other two iterations is calculated based on the similar calculation as the above, but with their corresponding residues, moduli and multiplicative inverses.

Because RRNS decoder is based on MRC, which operates sequentially, a modification has been performed to improve the speed [16]. Instead of checking the residues one by one, some of them are checked in parallel. For example S_1, S_2 and S_2×m_1 are calculated twice, i.e., both in the first and second steps. Thus, by calculating these common syndromes once and sharing it to all required calculations, a faster decoding can be obtained. However, it is worth noting that this might incurs extra circuitries (e.g., multiplexer) and additional routing; hence, larger overhead area.

Figure 3(b) illustrates the RRNS decoder that comprises of four syndrome units, three converter units and a multiplexer. All syndrome units operate concurrently to validate the read codeword. They generally comprise of subtracter, adder, multiplier and multiplexer. As mentioned before common syndromes are shared by the units. These are realized by the

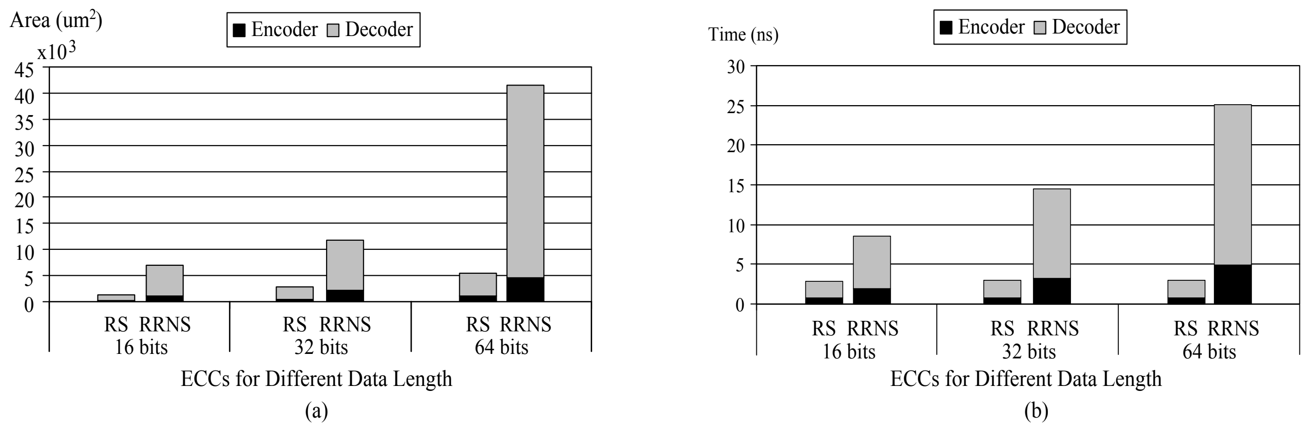


Fig. 4. (a) Encoder and decoder area overhead (b) encoder and decoder time delay

feedback connections from, e.g., *Syndrome1* to *Syndrome2*, etc. The output of the syndrome units become the input to the converter units, which produce binary data D_1, D_2 and D_3. Consequently, the binary data is compared to the operating legitimate range, which can be hardwired. Finally, the multiplexer selects the valid output data.

V. Experimental Results and Discussion

This section presents the experimental results and discussion. First, it presents the results of the hardware implementation of encoder and decoder for both ECCs, the analytical evaluation of the memory cell array overhead and the analytical evaluation of error correction capability. Finally, it discusses the experimental results.

A. Encoder/Decoder Area and Speed

To analyze the implementation cost, the encoder and decoder of both ECCs were designed using Xilinx Design Suite and synthesized using Synopsys Design Compiler based on 90nm CMOS.

Figure 4(a) illustrates the area overhead of the encoder and decoder for both ECCs. It shows that the area overhead for RS is smaller than that of RRNS irrespective of the data length. For example, the encoder and decoder of RS occupies 5× smaller area overhead than that of RRNS for 16-bit data. As the data length increases, the area overhead for RS slightly enlarges, while that for RRNS escalates.

Figure 4(b) depicts the speed of the encoder and decoder for both ECCs represented by the critical path time delay. It shows that RS operates faster than that of RRNS irrespective of the data length. For example, the encoder and decoder of RS is 3× faster than that of RRNS for 16-bit data and is 8.4× for 64-bit data. As the data length increases, RS time delay is quite the same; however, that of RRNS increases fairly linear (becomes slower).

B. Memory Cell Array Area Overhead

The area overhead of the memory cell array depends on the bit length of the codeword. Thus, the overhead of memory cell array when using both ECCs can be estimated analytically. Note that no real hardware synthesis can be carried out for *non-CMOS devices* because there is no available design tool for such devices yet.

Figure 5(a) depicts the required codeword length to correct clustered faults in RS and RRNS codewords stored in crossbar memories at different lengths of input data. Although both RS and RRNS codeword length increases as the size of input data enlarges, the difference in the required number of bits for the codeword becomes severe. For example at 64-bit input data, RRNS requires about 1.7× more bits than RS. Translating these numbers into the memory cell array area means that RS requires smaller area than RRNS for a fixed input data capacity. The difference becomes greater as the input data length increases. For example, it is about 1.4× greater for 64-bit input data encoded into both ECCs as compared to that of 16-bit.

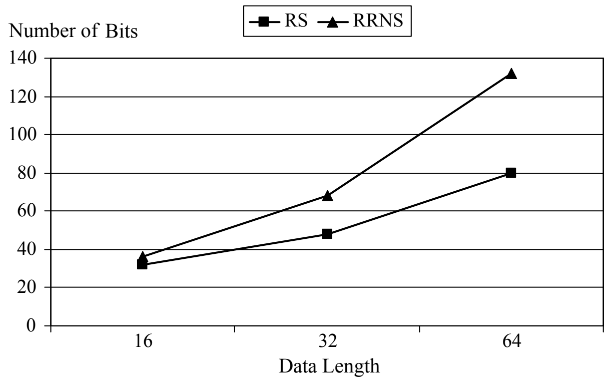
C. Error Correction Capability

In terms of correcting cluster errors, both ECCs has quite similar capability. Theoretically, RRNS scores slightly better than RS in case of the cluster errors exceeding the size of RS symbols. For example, consider a fault that induces 34-bit cluster errors at the third symbol of both ECCs as shown in Fig. 5(b). In this scenario, RS cannot correct them because the errors impact two symbols, which is beyond its single residue correction capability. However, for RRNS the errors only corrupt the third symbol, which is still can be corrected.

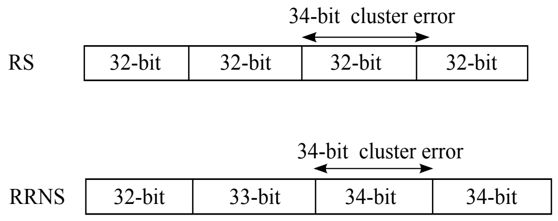
D. Discussion

With respect to hardware implementation and the associated cost, RS performs better than RRNS because of the followings:

- RS symbols are based on Galois Field elements for which all symbols have equal bit length. However, RRNS symbols are based on the residues generated from mutually co-prime moduli, each might has different bit length. Moreover, the redundant residues (checkword) must be bigger than non-redundant residues (dataword). Hence, the total bit length of RRNS codeword is larger than that of RS. Clearly, larger bit length implies bigger area and longer execution time of the encoder and decoder as well as greater memory cell array area.



(a)



(b)

Fig. 5. (a) Codeword length for different data length (b) cluster errors impacting the ECCs

- The RS encoder and decoder comprises of simple XOR gates and LUT-based error corrector, whereas that of RRNS consists of adders, subtracters and multiplier besides ROM-based moduli and moduli inverses. Obviously, XOR gates is smaller and faster than adders, subtracters and multiplier.
- RS decoder requires only two syndromes in validating the read codeword. On the other hand, RRNS decoder needs to compute three syndromes for the same purpose. Even though the MRC-based RRNS decoding has been parallelized, the time latency is still worst than RS. On the other hand, the parallel execution incurs bigger area overhead.
- Above all, ECCs in essence depend on consistency rules to have a unique relationship between data and codeword. For RS, the ECC requires a single consistency rule, i.e., Galois Field, whereas RRNS needs three consistency rules (see Section III-B). Intuitively, lesser rules realize simpler algorithm and implementation.

VI. Conclusion

This paper has presented a case study of two symbol-oriented ECC designs for fault-tolerant crossbar memories. The encoder and decoder of two ECCs, Reed Solomon and Redundant Residue Number System, have been implemented and experimentally compared. The results show that RS requires smaller area overhead and operates faster than the RRNS. In terms of correcting cluster errors, both ECCs posses quite similar capability. It can be concluded that RS offers better performance at lower cost than RRNS because the former can be implemented mainly using simple logic gates, whereas the latter needs more complex logic circuitries such as adder, multiplier and multiplexer. Moreover, RS relies on *one* encoding and decoding consistency rule; on the other hand, RRNS depends on *three* consistency rules.

References

[1] The International Technology Roadmap for Semiconductors 2009. Available: http://www.itrs.net/Links/2009ITRS/Home2009.htm

[2] G.S. Rose et al., "Design Approaches for Hybrid CMOS/Molecular Memory Based on Experimental Device Data", in *Proceedings of Great Lakes Symposium on VLSI*, pp. 2–7, 2006.

[3] A. DeHon and K.K. Likharev, "Hybrid CMOS/Nanoelectronic Digital Circuits: Devices, Architectures, and Design Automation", in *Proceedings of IEEE/ACM International Conference on Computer-aided design*, pp. 375–382, 2005.

[4] M. Mishra and S. C. Goldstein, "Defect Tolerance at the End of the Roadmap", in *Proceedings of International Test Conference*, pp. 1201–1211, 2003.

[5] A. Orailoglu, "Nanoelectronic Architectures: Reliable Computation on Defective Devices", in *Digest of Workshop on Dependable and Secure Nanocomputing*, 2007.

[6] P. Lincoln, "Challenges in Scalable Fault Tolerance", in *Proceedings of IEEE/ACM International Symposium on Nanoscale Architectures*, pp. 13–14, 2009.

[7] D.B. Strukov and K.K. Likharev, "Prospects for Terabit-scale Nanoelectronic Memories", *Journal Nanoscience and Nanotechnology*, vol. 16, no. 1, pp. 137–148, 2005.

[8] D.B. Strukov and K.K. Likharev, "Defect-Tolerant Architectures for Nanoelectronics Crossbar Memories", *Journal Nanoscience and Nanotechnology*, vol. 7, no. 1, pp. 151–167, 2007.

[9] C.M. Jeffery and R.J.O. Figueiredo, "Hierarchical Fault Tolerance for Nanoscale Memories", *IEEE Transactions on Nanotechnology*, vol. 5, no. 4, pp. 407–414, July 2006.

[10] F. Sun and T. Zhang, "Defect and Transient Fault-Tolerant System Design for Hybrid CMOS/Nanodevice Digital Memories", *IEEE Transactions on Nanotechnology*, vol. 6, no.3, pp. 341–351, May 2007.

[11] H. Naeimi and A. DeHon, "Fault Secure Encoder and Decoder for NanoMemory Applications", *IEEE Transactions on Very Large Scale Integration Systems*, vol.17, no.4, pp. 473–486, April 2009.

[12] S.L. Ngoc and Z. Young, "An Approach to Double Error Correcting Reed-Solomon Decoding Without Chien Search", in *Proceedings of the 36th Midwest Symposium*, pp. 534–537, 1993.

[13] L. Yang and L. Hanzo, "Coding Theory and Performance of Redundant Residue Number System Codes". Available: http://www-mobile.ecs.soton.ac.uk/lly/papers/RRNS-code.pdf

[14] N.Z. Haron and S. Hamdioui, "Residue-based Code for Reliable Hybrid Memories", in *Proceedings of IEEE/ACM International Symposium on Nanoscale Architectures*, pp. 27–32, 2009.

[15] N.Z. Haron and S. Hamdioui, "Using RRNS Codes for Cluster Faults Tolerance in Hybrid Memories", in *Proceedings of IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, pp. 85–93, 2009.

[16] Z. Ahyadi, *Experimental Analysis on ECC Schemes for Fault-Tolerant Hybrid Memories*. MSc Thesis, Delft University of Technology, 2009.

[17] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*. 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 2004.

[18] F. Barsi and P. Maestrini, "Error Correcting Properties of Redundant Residue Number Systems", *IEEE Transactions of Computers*, vol. 22, no. 3, pp. 307–315, 1973.

[19] A. Houghton, *The Engineers Error Coding Handbook*. Chapman and Hall, 1997.

[20] A. Omondi and B. Premkumar, *Residue Number System: Theory and Implementation*. Imperial College Press, 2008.

[21] N. Szabo and R. Tanaka, *Residue Arithmetic and its Application to Computer Technology*. New York: McGraw-Hill, 1967.