ReRAM Based In-Memory Computation of Single Bit Error Correcting BCH Code

Yaswanth Tavva, Debjyoti Bhattacharjee, Anupam Chattopadhyay, Swagata Mandal

ReRAM based In-Memory Computation of Single Bit Error Correcting BCH Code

Swagata Mandal[1,*], Yaswanth Tavva[2], Debjyoti Bhattacharjee[2], and Anupam Chattopadhyay[2]

[1] Department of Electronics and Communication Engineering,
Jalpaiguri Government Engineering College (Autonomous), India
[2] School of Computer Science Engineering,
Nanyang Technological University, Singapore
Email*:swaga89@gmail.com

Abstract. Error resilient high speed robust data communication is the primary need in the age of big data and Internet-of-things (IoT), where multiple connected devices exchange huge amount of information. Different multi-bit error detecting and correcting codes are used for error mitigation in the high speed data communication though it introduces delay and their decoding structures are quite complex. Here we have discussed the implementation of single bit error correcting Bose, Chaudhuri, Hocquenghem (BCH) code with simple decoding structure on a state-of-the art ReRAM based in-memory computing platform. ReRAM devices offer low leakage power, high endurance and non-volatile storage capabilities, coupled with stateful logic operations. The proposed lightweight library presents the mapping for generation of elements on Galois field (GF) for computation of BCH code, along with encoding and decoding operations on input data stream using BCH code. We have verified the results for BCH code with different dimensions using SPICE simulation. For (15,11) BCH code, the number of clock cycles required for element generation, decoding and encoding of BCH code are 103, 230 and 251 respectively, which demonstrates the efficacy of the mapping.

Keywords: Error Correcting Code, BCH code, In memory Computing, ReRAM

1 Introduction

In the age of big data and IoT, error resilient data storage, analysis and transmission are very crucial in different fields like social media, health care, deep space exploration and underwater surveillance etc. Even though the chances of data corruption in the silicon based semiconductor memory has grown with the shrinking of technology node, semiconductor based storage devices like random access memory (RAM), read only memory (ROM) and flash memory popularly used in the memory industry still have large footprint [1]. In order to prevent data corruption in the semiconductor memories, various traditional error mitigation techniques like triple modular redundancy (TMR), concurrent error detection

(CED) [2] and readback with scrubbing [3] are generally used. The above mentioned methods consume large area, power and are not suitable for real time applications. Sometimes interleaving is used for error mitigation in memory but it increases the complexity of the memory and is not useful for small memory devices.

In order to alleviate the drawbacks of TMR, CED or scrubbing, various error detecting and correcting (EDAC) codes are used for error mitigation in the data memory as well as in the communication channels. In general, single bit errors in the memory are corrected by using single bit error correcting code such as Hamming or Hisao code. In order to correct multiple erroneous bits, multi-bit error correcting block codes like Bose, Chaudhuri, Hocquenghem (BCH) code [4], Reed-Solomon code [5] are used. They have greater decoding complexity and large overhead due to the presence of more number of redundant bits compared to single bit error correcting code. Data in the memory is arranged as a matrix. Hence, different product codes are used for error mitigation in the memory where two low complexity block codes are used as component codes. Product codes formed using only Hamming codes as component codes [6] or Hamming code and parity code as component codes [6], are used to correct multi-bit upset in the SRAM based semiconductor memory. Error detection capability of different complex EDAC codes can be concatenated with Hamming code to generate low complexity multi-bit error correcting code, such as RS code concatenated with Hamming code [7] and BCH code concatenated with Hamming code [8]. In addition to block code, memory based convolutional codes [9] are also used for error mitigation in the storage devices.

Error detection and correction methods discussed so far are implemented separately that read data from memory, perform encoding and decoding operation and finally write back data into the memory. With the rise of emerging technologies, computing can be performed in the memory itself, alongside storage of data unlike traditional von Neumann computing models [10]. Redox based Random Access Memory (ReRAM) is one of the non-volatile storage technology which supports such in memory computing [11]. Due to high circuit density, high retention capability and low power consumption, ReRAM technology is capable of being used as an alternative of NAND or NOR flash in the industry. Unlike CMOS or TTL based semiconductor memory technology, ReRAM uses different dielectric materials to develop its crossbar structure. ReRAM demonstrates good switching characteristics between high and low resistance state compared to other emerging memories like magnetic random access memory (MRAM), ferroelectric random access memory (FRAM) [12], etc. ReRAM based memory technology is compatible with conventional CMOS based design flow and provides inherent parallelism due to its crossbar structure. The working principle of ReRAM technology involves formation of low resistance conducting path through dielectric material by applying a high voltage across it. The conducting path arises due to multiple mechanisms like metal defect, vacancy, etc [13]. The conducting tunnel through insulator can be controlled by an external voltage source for performing SET or RESET operations on the device.

Several in-memory computation platforms have already been proposed using ReRAMs, such as, general purpose in memory arithmetic circuit implementations [14], neuromorphic computing platforms [15] and general purpose Programmable Logic-in-Memory (PLiM) [16]. Apart from these general purpose applications, ReRAM based computation platforms are also used to implement different domain specific algorithms like machine learning [17,18], encryption [19] or compression algorithm [20].

Authors in [21] proposed efficient hardware implementation of BCH code. Further, hardware implementation of non-binary BCH code or RS code is also proposed by authors in [22]. The basic building blocks of error correcting code is the finite field arithmetic. The hardware implementation of high throughput finite field multiplier circuit on field programmable gate array (FPGA) and application specific integrated circuit (ASIC) are discussed by authors in [23]. Recently, ReRAM based in memory computation of Galois field (GF) arithmetic is described by authors in [24]. In this work, we propose the first in-memory BCH encoding and decoding operation library. Specifically, our contributions are as follows :-

- This work presents the first in-memory implementation of encoding and decoding operation of BCH code using ReRAM crossbar array.
- The proposed mapping harnesses the bit-level parallelism offered by ReRAM crossbar arrays and supports a wide variety of crossbar dimensions.
- In order to perform matrix multiplication during encoding and decoding operations, we have proposed a new method of implementation of binary matrix multiplication using ReRAM crossbar array. We refer the method as BiBLAS-3, since it is a level-3 binary basic linear algebra subprogram.
- The proposed implementation has a very low footprint in terms of devices required as well as energy, which makes it suitable for use as building blocks for different applications.

The rest of the paper is organized as follows. Section 2 presents the fundamentals of GF arithmetic, basics of encoding and decoding operations using BCH code along with a succinct introduction to ReVAMP, a state-of-the-art ReRAM based in-memory computing platform. Section 3 presents detailed implementation of element generation of GF, encoding and decoding operations for the ReVAMP plaftorm using BiBLAS-3. Experimental results are described in section 4, followed by conclusion in section 5.

2 Preliminaries

In this section, we present the fundamentals of encoding and decoding operation using BCH code. We introduce the preliminaries of logic operation using ReVAMP architecture. The encoding and decoding operations of the BCH code will be performed on binary GF, that we describe briefly.

2.1 Galois Field Arithmetic

A field is a set of elements on which basic mathematical operations like addition and multiplication can be performed without leaving the set. Hence, these basic operations must satisfy *distributive*, *associative* and *commutative* laws [25]. The order of a field is the number of elements in the field. A field with finite number of elements is known as GF. The order of the GF is always a prime number or the power of a prime number. If p be a prime number and m be a positive integer, then GF will contain p^m elements and can be represented as $GF(p^m)$.

For $m = 1, p = 2$, the elements in GF will be {0,1} and this is known as binary field. Here, we will consider GF of 2^m elements from the binary field $GF(2)$ where $m > 1$. If U be the set of the elements of the field and α be an element of $GF(2^m)$, then U can be represented by equation (1).

$$U = [0, \alpha^0, \alpha^1, \alpha^2, \alpha^3, ..., \alpha^{2^m-1}] \tag{1}$$

Let $f(x)$ be a polynomial over $GF(2^m)$ and it is said to be irreducible if $f(x)$ is not divisible by any other polynomial in $GF(2^m)$ with degree less than m, but greater than zero [26]. The irreducible polynomial is a primitive polynomial, if the smallest positive integer q for which $f(x)$ divides $x^q + 1$, where $q = 2^m - 1$. For each value of m, there can be multiple primitive polynomials, but we will use the primitive polynomial with least number of terms for computation over GF.

(a)

$GF(2^m)$	**Primitive Polynomial**
2^2	$x^2 + x + 1$
2^3	$x^3 + x + 1$
2^4	$x^4 + x + 1$
2^5	$x^5 + x^2 + 1$
2^6	$x^6 + x + 1$
2^7	$x^7 + x^3 + 1$

(b)

Power Repr.	**Polynomial Repr.**	**4-Tuple Repr.**
0	0	(0, 0, 0, 0)
1	α^0	(0, 0, 0, 1)
α	α^1	(0, 0, 1, 0)
α^2	α^2	(0, 1, 0, 0)
α^3	α^3	(1, 0, 0, 0)
α^4	$\alpha + 1$	(0, 0, 1, 1)
α^5	$\alpha^2 + \alpha$	(0, 1, 1, 0)
α^6	$\alpha^3 + \alpha^2$	(1, 1, 0, 0)
α^7	$\alpha^3 + \alpha + 1$	(1, 0, 1, 1)
α^8	$\alpha^2 + 1$	(0, 1, 0, 1)
α^9	$\alpha^3 + \alpha$	(1, 0, 1, 0)
α^{10}	$\alpha^2 + \alpha + 1$	(0, 1, 1, 1)
α^{11}	$\alpha^3 + \alpha^2 + \alpha$	(1, 1, 1, 0)
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	(1, 1, 1, 1)
α^{13}	$\alpha^3 + \alpha^2 + 1$	(1, 1, 0, 1)
α^{14}	$\alpha^3 + 1$	(1, 0, 0, 1)

Fig. 1: *(a) Primitive polynomial for various order GF. (b) Representation of elements in* $GF(2^4)$.

The list of primitive polynomials for different values of m is shown in Figure 1a. These primitive polynomials are the basis of computation using the

Table 1: *Variation of dimension of single bit error correcting BCH code with the order of GF.*

Order of $GF(m)$	**Dimension of BCH code**	α^k
3	(7, 4)	$\alpha^k = \alpha^{k-2} + \alpha^{k-3}$
4	(15, 11)	$\alpha^k = \alpha^{k-3} + \alpha^{k-4}$
5	(31, 26)	$\alpha^k = \alpha^{k-3} + \alpha^{k-5}$
6	(63, 57)	$\alpha^k = \alpha^{k-5} + \alpha^{k-6}$
7	(127,120)	$\alpha^k = \alpha^{k-4} + \alpha^{k-7}$

elements of GF. For the generation of elements of GF, we will start from two basic elements 0, 1 and another new element α.

In this paper, we have discussed encoding and decoding operation of single bit error correcting BCH code on $GF(2^m)$ where m varies from 3 to 7. As α is an element of $GF(2^m)$, it must satisfy the primitive polynomial corresponding to $GF(2^m)$. With the variation of m, not only primitive polynomial changes but also dimension of BCH code changes as shown in Table 1. If α be an element in $GF(2^m)$, α^k (where k is an positive integer and $k > 2$) is also be an element of $GF(2^m)$ and the recursive expression that will be used to calculate α^k for different values m in $GF(2^m)$ are shown in Table 1. Here in Figure 1b we have illustrated the power, polynomial and 4-Tuple representation of all the elements of $GF(2^4)$ are shown in Fig 1b. Based on the elements of GF, the encoding and decoding operations of BCH code will be performed.

2.2 Basics of BCH encoding and decoding operation

BCH is a powerful random error correcting cyclic code which is basically general purpose multi-bit error correcting Hamming code. Given two integers m and t such that $m > 3$ and $t < 2^m - 1$, then there exists a binary BCH code whose block length will be $n = 2^m - 1$ with the number of parity check bits equal to $(n - k) \leq mt$ and the minimum distance will be $d_{min} \geq (2t + 1)$. This will represent t error correcting BCH code. If α be a primitive element in $GF(2^m)$, then generator polynomial $g(x)$ of t error correcting BCH code of length $2^m - 1$ will be lowest degree polynomial over $GF(2)$ and α, α^2,...,α^{2t} will be its root. Hence, the equation (2) must satisfy.

$$g(\alpha^i) = 0 \quad \forall i \in \{1, 2, \ldots, t\} \tag{2}$$

If $\phi_i(x)$ be the minimal polynomial of α^i, then $g(x)$ will be formed using the equation (3).

$$g(x) = LCM\{\phi_1(x), \phi_2(x), \ldots, \phi_{2t}(x)\} \tag{3}$$

As α^i and $\alpha^{i'}$ (where $i = i'2^l$, i' is odd and $l > 1$) are conjugate to each other $\phi_i(x) = \phi_{i'}(x)$. Hence, $g(x)$ will be formed using the equation (4).

$$g(x) = LCM\{\phi_1(x), \phi_3(x), \ldots, \phi_{2t-1}(x)\} \tag{4}$$

Since we will use single bit error correcting BCH code, the generator polynomial $g(x)$ for $GF(2^4)$ is given by

$$g(x) = \phi_1(x) = x^4 + x + 1$$

The degree of $g(x)$ will be at most mt and the number of parity bits will be $(n-k)$. After the generation $g(x)$, the encoding operation will involve multiplication of input data $D(x)$ with $g(x)$, *i.e*, $C(x) = D(x) \times g(x)$.

The decoding operation of BCH code will involve the following steps:

1. Syndrome computation.
2. Determine the error locater polynomial λ from the syndrome components S_1, S_2,...,S_{2t}.
3. Find the error location by solving the error locater polynomial $\lambda(x)$.

Let $r(x) = r_0 + r_1x + r_2x^2 + \ldots + r_{n-1}x^{n-1}$ be the received data and $e(x)$ be the error pattern, then $r(x) = D(x) + e(x)$. For t error correcting BCH code, the parity check matrix will be

$$H = \begin{bmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \ldots & \alpha^{(n-1)} \\ 1 & \alpha^3 & (\alpha^3)^2 & (\alpha^3)^3 & \ldots & (\alpha^3)^{(n-1)} \\ 1 & \alpha^5 & (\alpha^5)^2 & (\alpha^5)^3 & \ldots & (\alpha^5)^{(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{(2t-1)} & (\alpha^{(2t-1)})^2 & (\alpha^{(2t-1)})^3 & \ldots & (\alpha^{(2t-1)})^{(n-1)} \end{bmatrix}$$

The syndrome is a $2t$-tuple $S = (S_1, S_2, \ldots, S_{2t}) = r \times H^T$ where H is the parity check matrix. Since we are considering single bit error correcting BCH code, t will be equal to 1 and $S = S_1 = r \times H^T$.

In the next step, from the syndrome values $2t$ nonlinear equations are formed which will be solved using either Berlekamp-Massey or Euclid's algorithm [27] and an error locater polynomial is formed using the roots obtained by solving the $2t$ nonlinear equations. Finally, the roots of the error locater polynomial is solved using Chien search algorithm [27]. Single bit error correcting BCH code generate only one syndrome whose value can directly locate the position of the erroneous bit and hence, we have not discussed the detailed implementation of step 2 and step 3 of the decoding of BCH code.

2.3 In-memory computing using ReRAM

In this subsection, we describe the ReRAM-based in-memory computing platform — ReVAMP, introduced in [28]. The architecture, presented in Figure 2 utilizes ReRAM crossbar with lightweight peripheral circuitry for in-memory computing. The ReRAM crossbar memory is used as data storage and computation memory (DCM). This is where in-memory computation using ReRAM devices takes place. A ReRAM crossbar memory consists of multiple 1-Select 1-Resistance (1S1R) ReRAM devices [29], arranged in the form of a crossbar [30].

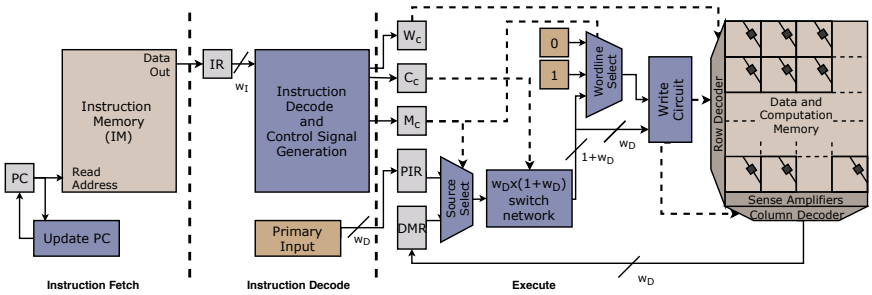


Fig. 2: *ReVAMP architecture.*

Read *wl*

Apply *wl s ws wb* (v val_{w_D-1}) ... (v val_0)

Fig. 3: *ReVAMP instruction format.*

A V/2 scheme is used for programming the ReRAM array. Unselected lines are kept to ground. In a readout phase, the presence of a high current ($\approx 5\ \mu A$) is considered as logic '1' while presence of a low current ($< 2\ \mu A$) is interpreted as logic '0'. Like conventional RAM arrays, ReRAM memories are accessed as w_D-bit wide words. Each ReRAM device has two input terminals, namely the wordline *wl* and bitline *bl*. The internal resistive state Z of the ReRAM acts as a third input and stored bit. The next state of the device Z^n can be expressed as Boolean majority function with three inputs, where the bitline input is inverted.

$$Z^n = M_3(Z, wl, \overline{bl}) \tag{5}$$

This forms the fundamental logic operation that can be realized using ReRAM devices. Using the intrinsic function Z^n, inversion operation can be realized. Since majority and inversion operation form a functionally complete set, any Boolean function can be realized using the Z^n.

The ReVAMP architecture has a three-stage pipeline with Instruction Fetch, Instruction Decode and Execution stages, as shown in Fig. 2. The instruction memory (IM) can be a regular memory or a ReRAM memory, with the program counter being used to address and fetch the stored instructions in the Instruction Fetch stage.

The architecture supports two instructions — *Read* and *Apply*, as presented in Fig. 3. *Read* instruction reads a specified word, *wl* from the DCM and stores it in the Data Memory Register (DMR). The read out word, available in the DMR, can be used as input by the following instructions. The *Apply* instruction is used for computation in the DCM. The address *wl* specifies the word in the DCM that will be computed upon. A bit flag *s* chooses whether the inputs will be from primary input register (PIR) or DMR. Two-bit flag *ws* is used to select the wordline input – 11 selects '1', 10 selects '0', 00 selects *wb* bit within the chosen data source for use as wordline input while 01 is an invalid value for *ws*. The pairs (v, val) are used to specify individual bitline inputs. The bit flag *v*

indicates if the input is NOP or a valid input. Similar to *wb*, the bits *val* specify the bit within the chosen data source for use as bitline input.

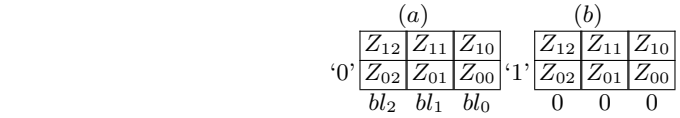


Fig. 4: *A* 2×3 *ReRAM crossbar, i.e., a crossbar with two rows and three bitlines.* Z_{ij} *represents the state of device at wordline* i *and bitline* j*. (a) Computation on* 0^{th} *row with '0' and* $\{bl_0, bl_1, bl_2\}$ *as the wordline and bitline inputs respectively. Valid inputs can be either* $\{$*'1'* $(+2.4V)$ *or '0'* $(-2.4V)$ $\}$*. (b)* 0^{th} *row is being read out, by setting wordline to '1'* $(+2.4V)$ *and the bitlines to 0* $(0V)$*.*

.

Fig. 4 shows a 2×3 ReRAM crossbar array, which can act as the DCM. The operation in Fig. 4 (a) can be expressed as an *Apply* instruction,

Apply 0 00 00 00 1 00 1 01 1 10

and the PIR contents are set to bl_0 bl_1 bl_2. The operation in Fig. 4 (b) can be expressed as Read 0. From here on, we express the in-memory compute operations in the crossbar representation.

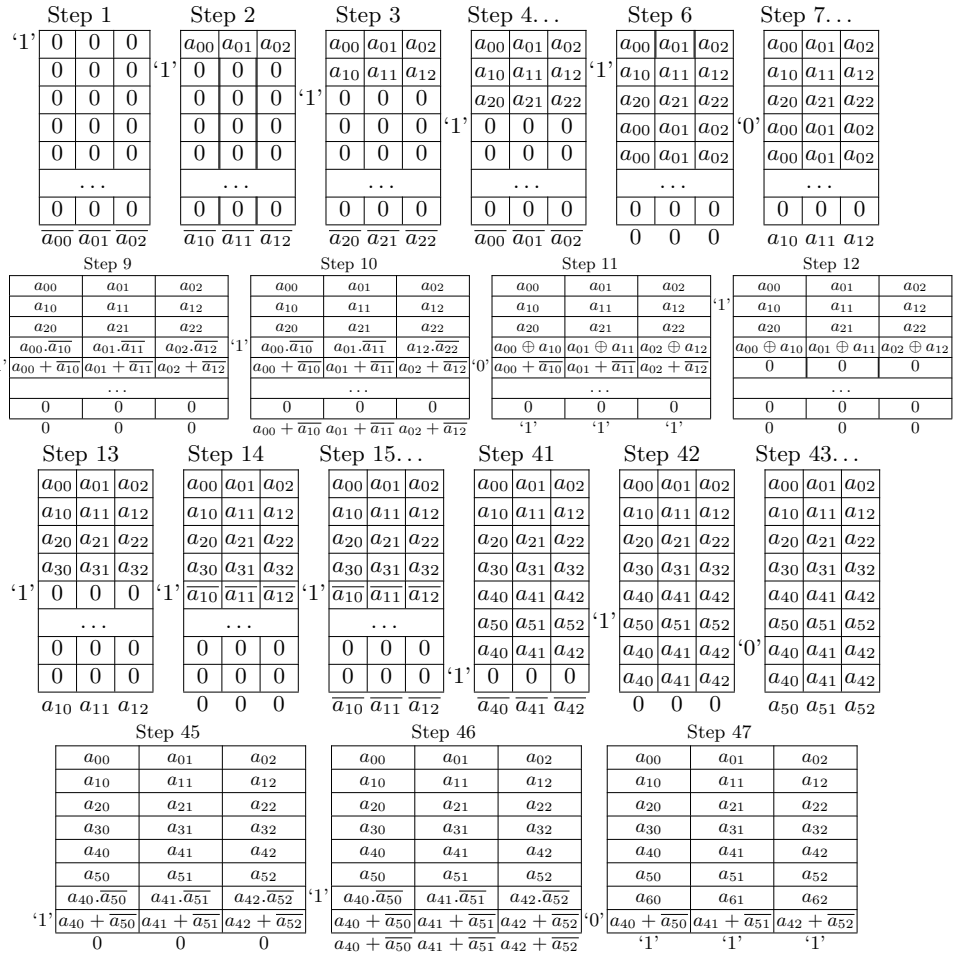
3 Methodology

The ReVAMP architecture performs in-memory computing operation using ReRAM devices capable of computing three-input Boolean majority with a single input inverted. Boolean majority with inverter is a functionally complete set. Therefore, it can be used for computation of arbitrary Boolean functions. The ReVAMP allows simultaneous computation on all devices that share a common wordline. A signal is required to read out the contents of a word. A recent work demonstrated multiple mathematical operation on the elements of GF using ReVAMP architecture [24]. In this section, we present the mapping of encoding and decoding operation of BCH code using these mathematical operation on the ReVAMP architecture. It involves three steps: generation of GF elements, encoding using BCH code and decoding using BCH code. The encoding and decoding operation using BCH code basically involves matrix multiplication which we will implement using ReRAM crossbar with the help of the BiBLAS operations proposed in [31]. We use the terms wordline and rows interchangeably. Similarly, the terms bitline and columns are used interchangeably.

3.1 Generation of GF elements

Here, we will illustrate the generation of elements of $GF(2^3)$ as an example. As each element in $GF(2^3)$ is a three tuple, the number of the columns in the

Table 2: *Generation operation for elements in* $GF(2^3)$ *using* 8×3 *DCM.*



DCM should either three or a multiple of three. For this purpose, we need DCM having 8 wordlines and 3 bitlines. Table 2 presents the intermediate state of the DCM and the inputs used for the generation of elements of GF. In Step 1, '1' is applied on 7^{th} wordline and $\overline{a_{00}}$, $\overline{a_{01}}$, $\overline{a_{02}}$ are applied to the bitlines. Here, ($\overline{a_{00}}$, $\overline{a_{01}}$, $\overline{a_{02}}$) represent 1, 1 and 0 respectively (from the 3-tuple representation shown in Fig. 1b). Similarly, 0, 0 and 1 are loaded into the 7^{th} row which basically represents α^0. In the next two steps, α and α^2 are loaded into the sixth and fifth row applying ($\overline{a_{10}}$, $\overline{a_{11}}$, $\overline{a_{12}}$) and ($\overline{a_{20}}$, $\overline{a_{21}}$, $\overline{a_{22}}$) to the bitlines respectively that represent (1,0,1) and (0,1,1).

Now α^3 will be calculated by modulo-2 addition of elements in 7^{th} and 6^{th} row. Modulo-2 addition between $a_{i,j}$ and $a_{(i+1),j}$ can be broken down into two op-

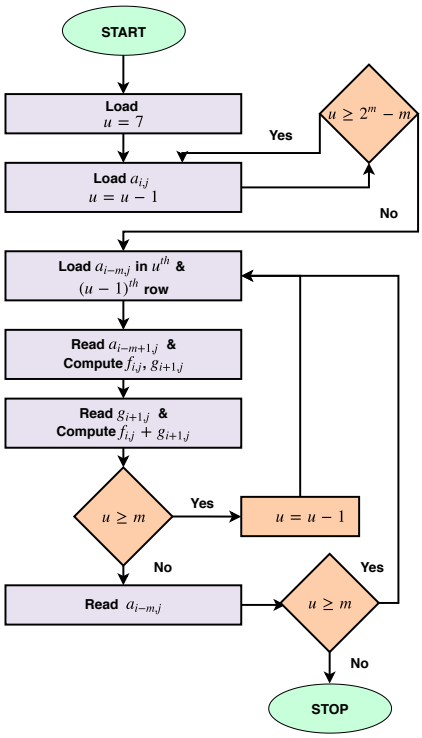


Fig. 5: *Flowchart for the generation of elements of* $GF(2^m)$.

erations.

$$a_{i,j} \oplus a_{(i+1),j} = a_{i,j}.\overline{a_{(i+1),j}} + (\overline{a_{i,j} + \overline{a_{(i+1),j}}}) \tag{6}$$

$$= f_{i,j} + \overline{g_{i+1,j}} \tag{7}$$

To compute $f_{i,j}$ and $g_{i+1,j}$, we require two copies of $a_{i,j}$. Here i represents power of α and j represents postion of a bit when α^i is expressed in 3-tuple format. Hence, in Step 5 and Step 6, we have loaded a_{00}, a_{01} and a_{02} in 4^{th} and 3^{rd} row. $f_{0,j}$ and $g_{1,j}$ are calculated by applying '0' in 4^{th} row, '1' in 3^{rd} row and 0 along all the bitlines in Step 7 and Step 8 respectively. In order to do OR operation between $f_{0,j}$ and $\overline{g_{1,j}}$, $g_{1,j}$ is read from the 3^{rd} row in Step 9 and then in Step 10, apply $g_{1,j}$ along all the bitlines and '1' in wordline of 4^{th} row. Finally, 4^{th} row will store the value α^3. Obeying the same procedure, α^4, α^5 and α^6 will be calculated and stored in the 3^{rd}, 2^{nd} and 1^{st} row of the crossbar respectively. The flowchart shown in Figure 5 describes the generation of elements for $GF(2^m)$ using DCM with u wordlines and m bitlines.

Table 3: *BiBLAS-3 parameters.*

Parameter	Description
m_1, n_1	Dimensions of Matrix A
m_2, n_2	Dimensions of Matrix B
m_1, n_2	Dimensions of Matrix C
r	Number of rows in crossbar
c	Number of columns in crossbar

3.2 Encoding and Decoding Operations

The encoding and decoding operations primarily involve binary matrix multiplications. During the encoding operation, the input data will be multiplied with the generator matrix which will give the encoded data. Similarly, during the decoding operation, the received data will be multiplied with the transpose of parity check matrix to generate the syndromes which helps to locate the error in the received data. Even though the decoding of BCH codes involve multiple steps as presented in section 2, we only require syndrome computation for achieving single-bit error correction.

Suppose A and B are two matrices with dimensions 2×2 and C be the matrix obtained after multiplication of A and B.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} \oplus a_{12}b_{21} & a_{11}b_{12} \oplus a_{12}b_{22} \\ a_{21}b_{11} \oplus a_{22}b_{21} & a_{21}b_{12} \oplus a_{22}b_{22} \end{bmatrix}$$

We will explain mapping of BiBLAS-3 on ReRAM crossbar using 2×2 matrix. The parameters involved in the BiBLAS-3 implementation are described in Table 3. We consider the following configuration for the BiBLAS-3 implementation.

- Matrices A,B are available in within the crossbar and the product matrix C is stored crossbar after computation.
- First five rows(r_1,r_2,...,r_5) are reserved for storage and last four rows ($reserve1$,...,$reserve4$) are used for computation.

The minimum dimensions of the matrix in mapping for BiBLAS-3 should be 1×2. Algorithm 1 shows step by step mapping of BiBLAS-3. In step 1, the elements of product matrix C is divided into a groups of size c. In this example, C is of 2×2 size with a total of 4 product terms. As the column size of choosen crossbar is 3, product terms are divided into two groups with elements 1, 3, 2 in group 1 and element 4 in group2. In step 2, the terms from matrices A and B, responsible for the formation of each product term in a group, are obtained. Each product constitutes a series of XORed dot products. The first and second dot products of each element in group 1 is computed in *reserve1* and *reserve2* rows respectively. Before computing third dot product if there are *reserve1, reserve2* they will be XORed first and followed by a clear operation on *reserve2*. Likewise, the following dot products are computed in *reserve2* and

Algorithm 1 BiBLAS-3

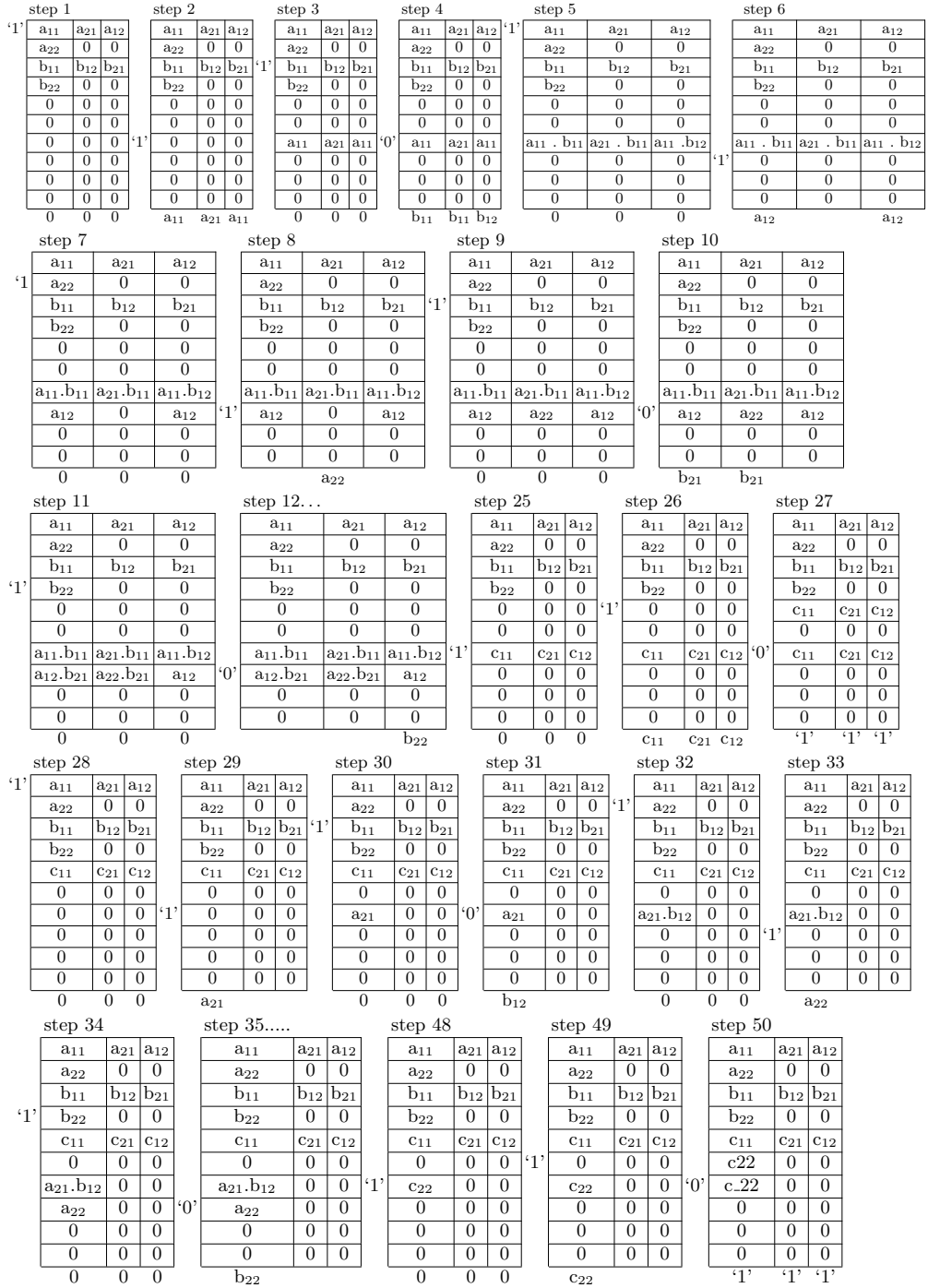
1: $groups \leftarrow$ GETGROUPS$(m1, n2, c)$ ▷ divides m1×n2 product terms into groups, with c being maximum size of each group
2: **for each** *groups* **do**
3: $partialproducts \leftarrow$ GETPARTIALPRODUTS$(group)$
4: **for each** *partialproducts* **do**
5: **if** first partial product **then**
6: LOADELEFROMA$(r1)$ ▷ load elements from matrix A present in partial product to row r1
7: ANDELEFROMB$(r1)$ ▷ load elements from matrix B present in partial product and dot product with row r1
8: **else**
9: LOADELEFROMA$(r2)$
10: ANDELEFROMB$(r2)$
11: XOR$(r1, r2)$. ▷ xor rows r1 and r2
12: RESET$(r2)$ ▷ reset row r2
13: **end if**
14: **end for**
15: COPY$(r1, x)$ ▷ copy r1 to an empty row x in crossbar
16: RESET$(r1)$ ▷ reset row r1
17: **end for**

accumulated to *reserve1* by XOR. This is repeated till all the dots products in the choosen group are completed. By the end of step 2, *reserve1* contains the products are of group 1. In step 3, the products from *reserve1* are copied to free memory and then *reserve1* is cleared. steps 2 and 3 are iterated till all the groups are computed on the crossbar. A detailed example is shown in Table 4.

4 Experiment

In this section, we analyze the performance of in-memory computation of encoding and decoding operations of BCH code on GF for various order m and crossbar dimensions. The experiments were performed using Cadence Virtuoso, using device-accurate model of ReRAM devices [29]. The dimension of single bit error correcting BCH code varies with change in the order m of GF. Figure 6 and Figure 7 show the delay in terms of number of clock cycles and area in terms of the number of ReRAM devices required to perform encoding and decoding operations, along with element generation of GF elements respectively using ReVAMP. The encoding operation involves generation of the generator matrix from parity check matrix and multiplication of the input data with the generator matrix whereas the decoding operation involves multiplication of the received data with the transpose of parity check matrix. Hence, the number of cycles required for encoding operation are more than the number of cycles required for decoding operation, that can be verified from Figure 6. Generation of generator matrix from the parity check matrix and multiplication of the input data with the generator matrix are sequential operations, so the same ReRAM

Table 4: *BiBLAS-3 computation with the input matrices A and B to compute the product matrix C.*



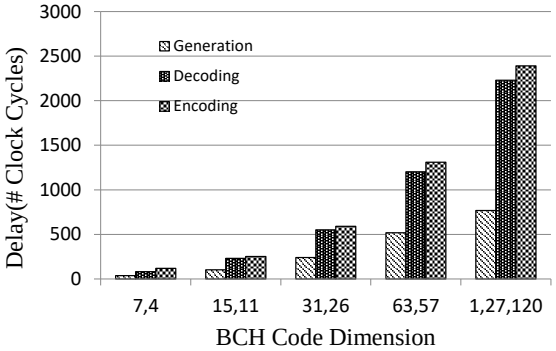


Fig. 6: *Delay of mapping for various operations on different dimension of single bit error correcting BCH code.*

devices will be used for both the operations. Thus, the area required for both encoding and decoding operations are basically ReRAM devices used for matrix multiplication. Hence, the area required for encoding and decoding operations are same as shown in Figure 7.

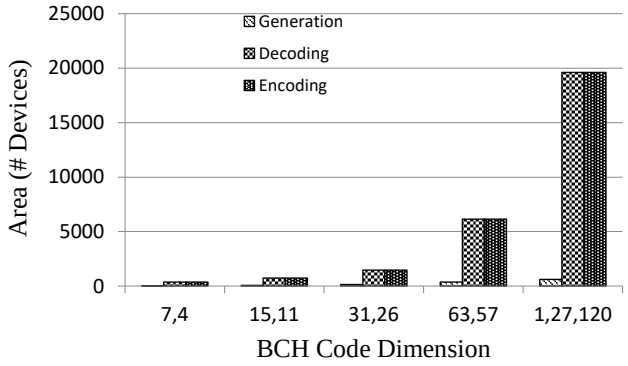


Fig. 7: *Area or the number of ReRAM devices required for generation of the elements of GF, encoding and decoding operation of BCH code on that GF.*

The increase in dimension of BCH code increases the order of GF, *i.e*, the delay as well as area requirements for computation of BCH code grow exponentially. Specifically, the number of instructions I_m required for generating all the elements of $GF(2^m)$ can be expressed by equation (8).

$$I_m = m + 11(2^m - m - 1) \tag{8}$$

Similarly, the number of instructions required for encoding (I_{en}) and decoding operation (I_{de}) of BCH code can be calculated by equation (9) and equation (10)

respectively, where n and k are the length of encoded data and input data respectively.

$$I_{en} = 2n + (n-1)12 + (k * (n-k)) \tag{9}$$

$$I_{de} = 2n + (n-1)12 \tag{10}$$

Addition operation involves XORing the individual bits of the input operands, which is done in parallel using the rows of ReRAM crossbar. Hence, the delay of performing XOR between any number of elements in two rows of crossbar operation remain constant, even with increase in the number of bitlines. The change in the length of encoded data n and input data k leads to change in the delay as well the number of ReRAM devices required for mapping.

The number of bit-level parallel operations on ReRAM crossbar arrays is dependent on the number of bitlines present in the crossbar. Figure 8 demonstrates the impact of number of bitlines on the mapping delay for operations in (15,11) BCH code. With the increase in the number of bitlines, the delay of operations reduce, which demonstrate the effectiveness of the proposed mapping in harnessing the bit-level parallelism offered by the ReRAM crossbar array.

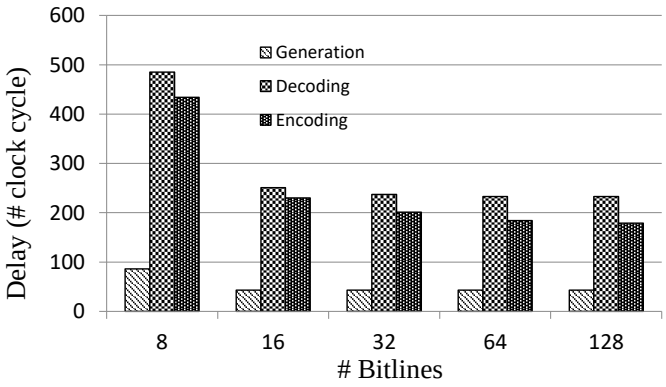


Fig. 8: *Impact of the number of bitlines on delay for computation of (15,11) BCH code.*

Figure 9 presents the mean energy required for element generation and encoding with decoding operation. Due to the large runtime of device-accurate simulations for all possible input combinations, we only report energy number for operations till BCH code with dimension (63,57). With the increase in dimension of BCH code, the dimensions of parity check and generator matrix also increases and hence, the total number of multiplications also increase. This drastically increases delay, area and energy consumption with the increase in dimension of BCH code.

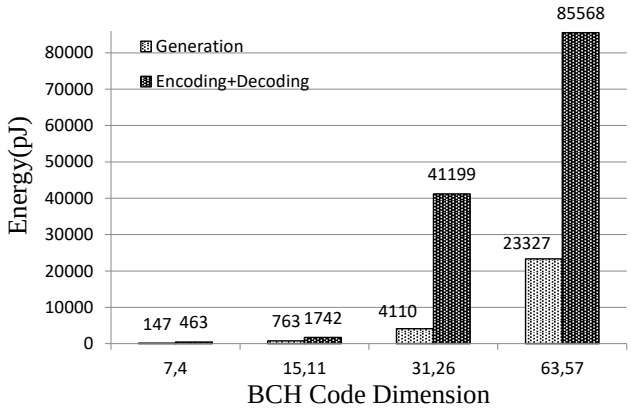


Fig. 9: *Impact of BCH code dimensions on energy required for computation.*

Even though the contemporary technologies such as ASIC and FPGA cannot be directly compared with ReRAM based implementation, we report a coarse comparison for the sake of completeness in Table 5. For ReRAM, we assume mature ReRAM technology with 1 ns read/write times [32]. The DCM column presents the size of the DCM used for in-memory computation, in terms of number of bits. Hence the size of DCM is calculated by multiplying number of wordline with the number of bitlines. The Instruction Memory (IM) column in the Table 5 presents the number of bits required for instruction storage in the IM of the ReVAMP architecture. The FPGA implementation (synthesized at 100 MHz) is on Kintex-7 evaluation board. The ASIC implementation (synthesized at $\approx$ 1GHz) was performed using Synopsys Design Compiler with TSMC 65nm technology library. As computing using ReRAM is inherently sequential, increase in the dimension of BCH code leads to direct increase in delay, along with corresponding increase in area (DCM size). For ASIC and FPGA, the increase in delay is relatively lesser. The main advantage of ReRAM is low area requirement in terms of number of devices required for implementation. For example, encoding and decoding operation of (31,26) BCH code requires $\approx$ 8k GE for ASIC, 510 LUT for FPGA but only 590 devices for ReVAMP.

5 Conclusion

In this work, efficient mapping of encoding and decoding operation of single bit error correcting BCH code was proposed on state-of-the-art ReRAM based in-memory computing platform. Further, we have devised a technique for efficient in-memory realization of BiBLAS-3 operations using ReRAM crossbar array. We have explored multiple configurations for the crossbar dimensions and demonstrated performance trade-offs while varying the dimensions of BCH code. The proposed implementation has a low energy footprint and shows good improve-

Table 5: *Comparison of area and delay for BCH computation on ReVAMP, ASIC and FPGA.*

Op.	BCH Dimension	ReVAMP Delay *ns*	ReVAMP DCM *bits*	ReVAMP IM *bits*	ASIC Delay *ns*	ASIC Area *GE*	FPGA Delay *ns*	FPGA Area *#LUT*
Generation	(7, 4)	36	8	154	4.975	2283	50	266
	(15, 11)	103	24	918	9.09	3260	90	316
	(31, 26)	239	64	2992	18.19	4619	170	341
	(63, 57)	519	160	10528	36.96	6883	330	390
	(127, 120)	768	384	20498	74.1	11028	650	456
Encoding	(7, 4)	118	400	531	6.88	3538	82	379
	(15, 11)	251	1600	33885	11.52	5126	91	421
	(31, 26)	590	3200	122130	22.36	8096	98	510
	(63, 57)	1310	5400	609150	40	9876	108	567
	(127, 120)	2391	6200	2418717	80	16078	119	625
Decoding	(7, 4)	81	400	3645	6.38	3050	77	377
	(15, 11)	230	1475	31050	10.83	5056	89	417
	(31, 26)	550	2890	113850	22.37	7820	97	510
	(63, 57)	1202	5400	558930	34.45	9201	106	568
	(127, 120)	2230	102	2325890	71.65	15890	117	627

ments in terms of area requirements compared to traditional ASIC and FPGA based design. In future, the work can be extended for in-memory implementation of multi-bit error correcting BCH code.

References

1. Ibe, E., Taniguchi, H., Yahagi, Y., Shimbo, K., Toba, T.: Impact of scaling on neutron-induced soft error in srams from a 250 nm to a 22 nm design rule. IEEE Transactions on Electron Devices **57**(7) (July 2010) 1527–1538
2. Krasniewski, A.: Concurrent error detection in sequential circuits implemented using fpgas with embedded memory blocks. In: Proceedings. 10th IEEE International On-Line Testing Symposium. (July 2004) 67–72
3. Asadi, G.., Tahoori, M.B.: Soft error mitigation for sram-based fpgas. In: 23rd IEEE VLSI Test Symposium (VTS'05). (May 2005) 207–212
4. Reviriego, P., Argyrides, C., Maestro, J.A.: Efficient error detection in double error correction bch codes for memory applications. Microelectronics Reliability **52**(7) (2012) 1528 – 1530 Special Section Thermal, mechanical and multi-physics simulation and experiments in micro-electronics and micro-systems (EuroSimE 2011).
5. Chen, B., Zhang, X., Wang, Z.: Error correction for multi-level nand flash memory using reed-solomon codes. In: 2008 IEEE Workshop on Signal Processing Systems. (Oct 2008) 94–99
6. Park, S.P., Lee, D., Roy, K.: Soft-error-resilient fpgas using built-in 2-d hamming product code. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **20**(2) (Feb 2012) 248–256
7. Neuberger, G., de Lima, F., Carro, L., Reis, R.: A multiple bit upset tolerant sram memory. ACM Trans. Des. Autom. Electron. Syst. **8**(4) (October 2003) 577–590
8. Poolakkaparambil, M., Mathew, J., Jabir, A.M., Mohanty, S.P.: Low complexity cross parity codes for multiple and random bit error correction. In: Thirteenth International Symposium on Quality Electronic Design (ISQED). (March 2012) 57–62

9. Jacobvitz, A.N., Calderbank, R., Sorin, D.J.: Writing cosets of a convolutional code to increase the lifetime of flash memory. In: 2012 50th Annual Allerton Conference on Communication, Control, and Computing (Allerton). (Oct 2012) 308–318
10. Chen, B., Cai, F., Zhou, J., Ma, W., Sheridan, P., Lu, W.D.: Efficient in-memory computing architecture based on crossbar arrays. In: 2015 IEEE International Electron Devices Meeting (IEDM). (Dec 2015) 17.5.1–17.5.4
11. Chen, Y., Petti, C.: Reram technology evolution for storage class memory application. In: 2016 46th European Solid-State Device Research Conference (ESSDERC). (Sep. 2016) 432–435
12. Yu, S., Chen, P.: Emerging memory technologies: Recent trends and prospects. IEEE Solid-State Circuits Magazine **8**(2) (Spring 2016) 43–56
13. Zhu, L., Zhou, J., Guo, Z., Sun, Z.: An overview of materials issues in resistive random access memory. Journal of Materiomics **1**(4) (2015) 285 – 295
14. Siemon, A., Menzel, S., Waser, R., Linn, E.: A complementary resistive switch-based crossbar array adder. IEEE Journal on Emerging and Selected Topics in Circuits and Systems **5**(1) (March 2015) 64–74
15. Sah, M.P., Kim, H., Chua, L.O.: Brains are made of memristors. IEEE Circuits and Systems Magazine **14**(1) (Firstquarter 2014) 12–36
16. Gaillardon, P., Amar, L., Siemon, A., Linn, E., Waser, R., Chattopadhyay, A., Micheli, G.D.: The programmable logic-in-memory (PLiM) computer. In: 2016 Design, Automation Test in Europe Conference Exhibition (DATE). (March 2016) 427–432
17. Song, L., Qian, X., Li, H., Chen, Y.: Pipelayer: A pipelined reram-based accelerator for deep learning. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). (Feb 2017) 541–552
18. Wang, Z., Karpovsky, M.G., Kulikowski, K.J.: Replacing linear hamming codes by robust nonlinear codes results in a reliability improvement of memories. In: 2009 IEEE/IFIP International Conference on Dependable Systems Networks. (June 2009) 514–523
19. Bhattacharjee, D., Pudi, V., Chattopadhyay, A.: SHA-3 implementation using ReRAM based in-memory computing architecture. In: 2017 18th International Symposium on Quality Electronic Design (ISQED). (March 2017) 325–330
20. Bhattacharjee, D., Chattopadhyay, A.: In-memory data compression using ReRAMs. In: Emerging Technology and Architecture for Big-data Analytics. Springer (2017) 275–291
21. Haroussi, M.E., Chana, I., Belkasmi, M.: Vhdl design and fpga implementation of a fully parallel bch siso decoder. In: 2010 5th International Symposium On I/V Communications and Mobile Network. (Sep. 2010) 1–4
22. Khan, M.A., Afzal, S., Manzoor, R.: Hardware implementation of shortened (48,38) reed solomon forward error correcting code. In: 7th International Multi Topic Conference, 2003. INMIC 2003. (Dec 2003) 90–95
23. Xie, J., Meher, P.K., Mao, Z.: High-throughput finite field multipliers using redundant basis for fpga and asic implementations. IEEE Transactions on Circuits and Systems I: Regular Papers **62**(1) (Jan 2015) 110–119
24. Mandal, S., Tavva, Y., Chattopadhyay, D.B.A.: ReRAM-based In-Memory Computation of Galois Field arithmetic. In: 2019 IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2019. (2019) 1–6
25. Couveignes, J.M., Edixhoven, B.: Computational aspects of modular forms and Galois representations. Princeton University Press (2011)

26. Kyuregyan, M.K.: Recurrent methods for constructing irreducible polynomials over fq of odd characteristics. Finite Fields and Their Applications **9**(1) (2003) 39 – 58
27. Joiner, L.L., Komo, J.J.: Decoding binary bch codes. In: Proceedings IEEE Southeastcon '95. Visualize the Future. (March 1995) 67–73
28. Bhattacharjee, D., Devadoss, R., Chattopadhyay, A.: ReVAMP: ReRAM based VLIW architecture for in-memory computing. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2017. (March 2017) 782–787
29. Siemon, A., Menzel, S., Marchewka, A., Nishi, Y., Waser, R., Linn, E.: Simulation of taox-based complementary resistive switches by a physics-based memristive model. In: 2014 IEEE International Symposium on Circuits and Systems (ISCAS). (June 2014) 1420–1423
30. Linn, E., Rosezin, R., Tappertzhofen, S., Bttger, U., Waser, R.: Beyond von neumannlogic operations in passive crossbar arrays alongside memory operations. Nanotechnology **23**(30) (2012) 305205
31. Bhattacharjee, D., Merchant, F., Chattopadhyay, A.: Enabling in-memory computation of binary BLAS using ReRAM crossbar arrays. In: 2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC). (Sep. 2016) 1–6
32. : Emerging research devices (ERD) report. In: International Technology Roadmap for Semiconductors (ITRS). (2013)