Concurrent Error Detection in Finite Field Arithmetic Operations using Pipelined and Systolic Architectures

Siavash Bayat-Sarmadi and M.A. Hasan

University of Waterloo, Ontario, Canada

November 6, 2007

Abstract

In this work we consider mainly detection of errors in polynomial, dual and normal bases arithmetic operations. Error detection is performed by recomputing with shifted operands method while the operation unit is in use. This scheme is efficient for pipelined architectures, particularly systolic arrays. Additionally, One semi-systolic multiplier for each of the polynomial, dual, type I and type II optimal normal bases is presented in this work. The results show that having better or similar space and time overheads compared to a number of related previous work, the multipliers have generally a high error detection capability, e.g., the error detection capability of the scheme for the single and multiple stuck-at faults in a polynomial basis multiplier are 99.08% and 100%, respectively. Finally, we also comments on how RESO can be used for concurrent error correction to deal with transient faults.

Index Terms

Finite field operations, concurrent error detection (CED), concurrent error correction (CEC), polynomial basis, dual basis, normal basis, pipelined architectures, systolic arrays.

I. Introduction

Achieving an acceptable security level for hardware implementations of large digital systems specially with critical applications has received significant attention recently. During the use of these digital systems, faults in the circuits may occur either due to natural causes or deliberate fault injection by an attacker (see for example [5], [6], [7], [10]). Faulty circuits are likely to

generate erroneous results that may make the whole system unusable or may help the attacker to break the system. As a result, error correction and detection are important for these systems.

This paper investigates mainly the detection and/or correction of random errors in polynomial, dual and normal bases arithmetic operations, which have applications in deep space channel coding [33], VLSI testing [26], and cryptography [22], [11]. Although the schemes proposed in this paper may not cure the problem of deliberately injected faults completely, they can reduce the possibility of an attack. This is because the number of faults that can be injected by an attacker is reduced to the number of faults that cannot be detected by the scheme.

Furthermore, some random errors for example in a number of hardware based cryptosystems or their arithmetic accelerators can be detected at an upper level operation, e.g., in elliptic curve cryptography (ECC), if a point leaves the curve it can be easily detected by point verification [5], [10]. This is, however, not always possible. In the case of ECC, a fault may move a point to another point without leaving the curve and this has been exploited in the so-called sign change fault attack [6]. As a result, some kind of mechanisms for error detection in finite field operations can still be quite important.

A number of schemes have been recently proposed in this regard in the recent past (see [4], [8], [16]). Some of the techniques that are used in these schemes can be categorized as follows:

- Using parity bits: In this approach, basically, the parity of the output is predicted and compared against its actual party. Some examples are [3], [2], [13], [27], [28], [29].
- Scaling techniques: The second approach, which is used in [1], [14], is for example to scale the inputs of a multiplier by a factor and at the end of the multiplication the correctness of the result is checked by one or two divisions.
- Nonlinear techniques [15]: This approach is expensive in terms of area and time and in turn may not be very efficient for detecting random errors.
- Time redundancy based techniques such as recomputing with shifted operands method: A number of schemes for polynomial basis multipliers can be found in the recent past literature, including [9], [21].

This paper presents two schemes for detecting and correcting errors concurrently in polynomial, dual and normal bases arithmetic operations. The schemes presented in this paper are based on recomputing with shifted operands technique and are efficient for pipelined architectures such as systolic arrays. To investigate more on this scheme, one finite field semi-systolic multiplier is

presented for each of the polynomial, dual, type I and type II optimal normal bases. Then the CED scheme is applied to them. Additionally, the space and time complexities of these multipliers are compared against a number of systolic and/or semi-systolic multipliers previously published in the literature. Furthermore, the capability of error detection of each multiplier is evaluated by simulation-based fault injection. The results show that having better or similar space and time overheads compared to a number of related previous work, the multipliers have generally a high error detection capability, e.g., the error detection capability of the scheme for the single and multiple stuck-at faults in a polynomial basis multiplier are 99.08% and 100%, respectively.

The organization of this article is as follows. In Section II, some preliminaries about polynomial, dual and normal bases as well as RESO method are reviewed. The concurrent error detection strategy is presented in Section III. General pipelined architectures, which are suitable for these schemes, along with an overhead analysis are given in Section IV. In Section V, the CED scheme is investigated with more details for polynomial, dual and normal bases multipliers. The error detection capability of the scheme is then evaluated in Section VI. In Section VII, some comments on the concurrent error correction strategy are given. Finally, Section VIII gives a few concluding remarks.

II. Preliminaries

This section briefly reviews polynomial, normal and dual bases, and RESO method.

A. Polynomial, Dual, Normal Bases

Let $F(x) = \sum_{i=0}^{m} f_i x^i$ be an irreducible polynomial over $GF(2)$ of degree m. Polynomial (or canonical) basis is defined as the following set:

$$\left\{1, x, x^2, \cdots, x^{m-1}\right\}.$$

Each element A of $GF(2^m)$ can be represented using the polynomial basis (PB) as $A = \sum_{i=0}^{m-1} a_i x^i$ where $a_i \in GF(2)$.

Moreover, suppose the dual basis (DB) of the polynomial basis defined above is represented as $\{y_0, y_1, \cdots, y_{m-1}\}$, where $y_i \in GF(2^m)$, then we have:

$$\mathrm{Tr}(y_i x^j) = \begin{cases} 0 & \text{if } i \neq j; \\ 1 & \text{otherwise,} \end{cases} \tag{1}$$

where the trace function of an element $a \in GF(2^m)$ is:

$$\mathrm{Tr}(a) = \sum_{i=0}^{m-1} a^{2^i}.$$

Furthermore, given that $x \in GF(2^m)$, the following set is a normal basis (NB) if the set members are linearly independent.

$$\{x, x^2, x^{2^2}, \cdots, x^{2^{m-2}}, x^{2^{m-1}}\}$$

B. RESO Method

REcomputing with Shifted Operands (RESO) is a technique for concurrent error detection (CED) in arithmetic and logic units introduced by Patel and Fung in [24], [25]. This technique is based on time redundancy. Suppose x and $f(x)$ are the input and output of a computation unit f, respectively. Also, suppose E and D are two functions such that $D(f(E(x))) = f(x)$. Now, we store the result of the computation of $f(x)$ (first step) in a register and compare it with the result of the computation of $D(f(C(x)))$ (second step). Any difference between results of these two steps indicates an error. The functions E and D are referred to as encoding and decoding functions, respectively, and they can be usually chosen such that $D = E^{-1}$. It is worth mentioning that for integer operands, E and D are simple shifts and since this method was initially used for such operands, it is referred to as RESO.

III. Concurrent Error Detection Strategy

Errors may be caused by different types of faults such as open faults, short (bridging) faults, and/or stuck-at faults. Furthermore, the faults can be transient or permanent. We assume that locations of these faults, occurred naturally or injected by an attacker, are random.

In this section, we use RESO method to concurrently detect errors in arithmetic operations over the field $GF(2^m)$. For the polynomial, normal and dual bases, the encoding and decoding functions are chosen in a way that the overhead costs (in terms of area and time) are fairly low. Additionally, in this paper, the arithmetic operations addition/subtraction, multiplication, inversion, division, and exponentiation are considered. Figure 1 shows a general architecture of an operation with concurrent error detection. In the figure, two encoding functions of the inputs are E_1 and E_2 and the decoding function of the output is D. Clearly, for inversion the second

input should not be considered. Also, for exponentiation the exponent is a non-negative integer number and is not an extension field element. Therefore, this input of exponentiation is not considered as well.

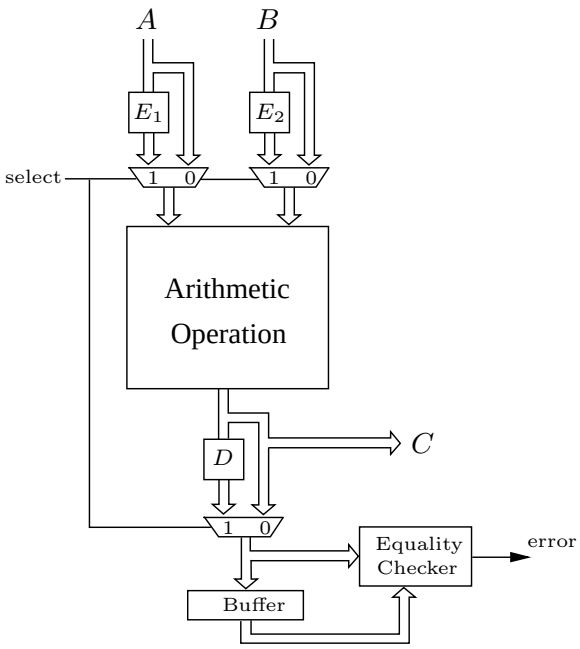


Fig. 1. General architecture for the arithmetic operations with CED

Let us assume that the arithmetic operation performs f function. Then we have:

$$C = f(A, B).$$

Also, let $A_{E_1} = E_1(A)$ and $B_{E_2} = E_2(B)$. Considering that C' is the result of the second computation after decoding, we have:

$$C' = D(f(A_{E_1}, B_{E_2})).$$

As a result,

$$\text{error} = \begin{cases} 0 & \text{if } C = C', \\ 1 & \text{if } C \neq C'. \end{cases}$$

In the following, the above-mentioned concurrent error detection strategy for each basis is investigated. The overhead cost of the CED scheme will be discussed in Section IV.

A. CED for Polynomial Basis (PB) Arithmetic Operations

Let $x \in GF(2^m)$ be the root of the field defining polynomial $F(x)$. A proper candidate for encoding and decoding functions in PB representation of the elements of the field is multiplication of an element by x or x^{-1}. Clearly, all arithmetic operations are modulo $F(x)$. Particularly, elements x^m and x^{-1} modulo $F(x)$ are as follows:

$$\begin{aligned} x^m &= 1 + \sum_{i=1}^{m-1} f_i x^i \bmod F(x), \\ x^{-1} &= x^{m-1} + \sum_{i=0}^{m-2} f_{i+1} x^i \bmod F(x). \end{aligned} \tag{2}$$

The multiplication of x and an arbitrary element A of $GF(2^m)$ is performed as follows:

$$\begin{aligned} xA &= \sum_{i=0}^{m-1} a_i x^{i+1} \bmod F(x) = \sum_{i=0}^{m-2} a_i x^{i+1} + a_{m-1} x^m \bmod F(x) \\ &= (0, a_0, \cdots, a_{m-3}, a_{m-2}) + a_{m-1}(1, f_1, \cdots, f_{m-2}, f_{m-1}). \end{aligned} \tag{3}$$

Similarly the multiplication of x^{-1} and A is performed as follows:

$$\begin{aligned} x^{-1}A &= \sum_{i=0}^{m-1} a_i x^{i-1} \bmod F(x) = a_0 x^{-1} + \sum_{i=1}^{m-1} a_i x^{i-1} \bmod F(x) \\ &= (a_1, a_2, \cdots, a_{m-1}, 0) + a_0(f_1, f_2, \cdots, f_{m-1}, 1). \end{aligned} \tag{4}$$

Hereafter, the former and the latter are referred to as (forward) scaling and inverse scaling, respectively. Additionally, both scalings are very inexpensive in hardware implementations. An overhead analysis will be given in Section IV-A.

Note that multiplication of an element with x^i or x^{-i} can be considered as i consecutive scalings or inverse scalings, respectively. In the following, encoding and decoding functions are determined for each operation. Also, we show the procedure of CED in each PB arithmetic operation, assuming that $A, B, C \in GF(2^m)$.

1) Addition/Subtraction: $E_1 = x$, $E_2 = x$, $D = x^{-1}$.
   a) Compute $A + B = C$; Store in a register;
   b) Compute $Ax + Bx = (A + B)x = Cx$; Inverse scaling; Compare this result with that of a).
2) Multiplication: $E_1 = x$, $E_2 = x$, $D = x^{-2}$.
   a) Compute $A \times B = C$; Store in a register;

b) Compute $Ax \times Bx = (A \times B)x^2 = Cx^2$; Two inverse scalings; Compare this result with that of a).

3) Inversion: $E_1 = x$, $D = x$.
   a) Compute $\frac{1}{A} = C$; Store in a register;
   b) Compute $\frac{1}{Ax} = (\frac{1}{A})x^{-1} = Cx^{-1}$; Forward scaling; Compare this result with that of a).

4) Division: $E_1 = x$, $E_2 = x^{-1}$, $D = x^{-2}$.
   a) Compute $\frac{A}{B} = C$; Store in a register;
   b) Compute $\frac{Ax}{Bx^{-1}} = \frac{A}{B}x^2 = Cx^2$; Two inverse scalings; Compare this result with that of a).

5) Exponentiation: $E_1 = x$, $D = x^{-n}$, where n is a non-negative integer.
   a) Compute $A^n = C$; Store in a register;
   b) Compute $(Ax)^n = A^n x^n = Cx^n$; n inverse scaling; Compare this result with that of a).

For large n, to speed up the exponentiation, one can pre-compute and store x^{-n} for $1 \leq n \leq 2^m - 1$. Alternative encodings and decodings for multiplication and division are as follows:

- Multiplication: $E_1 = x$, $E_2 = x^{-1}$, No decoding.
  1) Compute $A \times B = C$; Store in a register;
  2) Compute $Ax \times Bx^{-1} = (A \times B) = C$; Compare this result with that of a).
- Division: $E_1 = x$, $E_2 = x$, No decoding.
  1) Compute $\frac{A}{B} = C$; Store in a register;
  2) Compute $\frac{Ax}{Bx} = \frac{A}{B} = C$; Compare this result with that of a).

Although this is more efficient for implementation, it may result in a lower error detection capability. For example, a permanent single-bit fault at the end of an arithmetic operation cannot be detected, since such faults change the results of both runs in a same manner and generates identical results even in the presence of the faults.

B. CED for Dual Basis (DB) Arithmetic Operations

Similar to PB arithmetic operations, a suitable candidate for encoding and decoding functions in DB representation of the elements of the field is multiplication of an element by x or x^{-1}.

This multiplication is considered in Lemma 1.

Lemma 1: Let $A = (a'_0, a'_1, \cdots, a'_{m-1}) \in GF(2^m)$ be represented in dual basis. Let $F(x) = \sum_{i=0}^{m} f_i x^i$ be the field defining polynomial. Then the (forward) scaling and inverse scaling can be performed as follows:

$$\begin{aligned} xA &= (a'_1, a'_2, \cdots, a'_{m-1}, \sum_{i=0}^{m-1} f_i a'_i) \\ x^{-1}A &= (\sum_{i=0}^{m-1} f_{i+1} a'_i, a'_0, a'_1, \cdots, a'_{m-2}) \end{aligned} \tag{5}$$

Proof: The proof for forward scaling can be found in [32]. Similarly, the inverse scaling can be proved as follows. Assume that the PB representation of A is $(a_0, a_1, \cdots, a_{m-1})$. Then according to Section II, we have:

$$A = \sum_{i=0}^{m-1} a_i x^i, \qquad \text{(PB representation)}$$

$$A = \sum_{i=0}^{m-1} a'_i y_i. \qquad \text{(DB representation)}$$

Moreover, according to [32], we have:

$$\text{Tr}(x^j A) = a'_j.$$

Therefore, for $1 \leq j \leq m-1$, we have:

$$\begin{aligned} (x^{-1}A)'_j &= \text{Tr}\left(x^{-1}Ax^j\right) = \text{Tr}\left(Ax^{j-1}\right) \\ &= a'_{j-1}. \end{aligned}$$

Also,

$$\begin{aligned} (x^{-1}A)'_0 &= \text{Tr}\left(Ax^{-1}\right) = \text{Tr}\left(A\sum_{i=0}^{m-1} f_{i+1}x^i\right) = \sum_{i=0}^{m-1} f_{i+1}\text{Tr}\left(Ax^i\right) \\ &= \sum_{i=0}^{m-1} f_{i+1} a'_i. \end{aligned}$$

Therefore, $x^{-1}A = (\sum_{i=0}^{m-1} f_{i+1}a'_i, a'_0, a'_1, \cdots, a'_{m-2})$. ■

Note that for low wight $F(x)$, the hardware implementation of DB scalings requires only a few gates (see Section IV-B). Moreover, functions E_1, E_2 (if applicable), and D can be chosen same as those chosen for PB representation in Section III-A and similar CED procedure can be performed.

C. CED for Normal Basis (NB) Arithmetic Operations

Let $A = \sum_{i=0}^{m-1} a_i \alpha^{2^i}$ be the NB representation of A. Then, considering that the arithmetic is performed in characteristic 2, we have:

$$\begin{aligned} A^2 &= \sum_{i=0}^{m-1} \left(\hat{a}_i \alpha^{2^i}\right)^2 = \sum_{i=0}^{m-1} \hat{a}_i \alpha^{2^{i+1}} \\ &= (\hat{a}_{m-1}, \hat{a}_0, \hat{a}_1, \cdots, \hat{a}_{m-2}). \\ A^{\frac{1}{2}} &= \sum_{i=0}^{m-1} \left(\hat{a}_i \alpha^{2^i}\right)^{\frac{1}{2}} = \sum_{i=0}^{m-1} \hat{a}_i \alpha^{2^{i-1}} \\ &= (\hat{a}_1, \cdots, \hat{a}_{m-2}, \hat{a}_{m-1}, \hat{a}_0). \end{aligned} \tag{6}$$

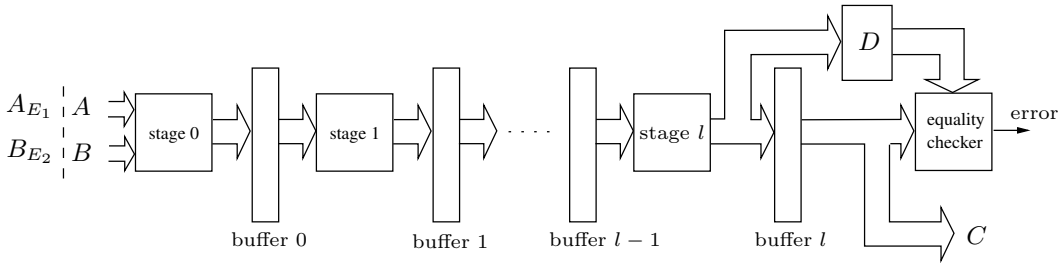
The hardware implementations of NB squaring and taking square root have no cost (see Section IV-C). Therefore, in NB arithmetic operations, proper choices for encoding and decoding functions are squaring and taking square root. Moreover, the procedures of CED in NB arithmetic operations are more uniform since the encoding function(s) and the decoding function are always squaring and taking square root, respectively. The CED procedures follow, assuming that $A, B, C \in GF(2^m)$ and n is a non-negative integer.

1) Addition/Subtraction:
   a) Compute $A + B = C$; Store in a register;
   b) Compute $A^2 + B^2 = (A + B)^2 = C^2$; Take square root; Compare this result with that of a).
2) Multiplication:
   a) Compute $A \times B = C$; Store in a register;
   b) Compute $A^2 \times B^2 = (A \times B)^2 = C^2$; Take square root; Compare this result with that of a).
3) Inversion:
   a) Compute $\frac{1}{A} = C$; Store in a register;
   b) Compute $\frac{1}{A^2} = (\frac{1}{A})^2 = C^2$; Take square root; Compare this result with that of a).
4) Division:
   a) Compute $\frac{A}{B} = C$; Store in a register;
   b) Compute $\frac{A^2}{B^2} = \frac{A}{B}^2 = C^2$; Take square root; Compare this result with that of a).

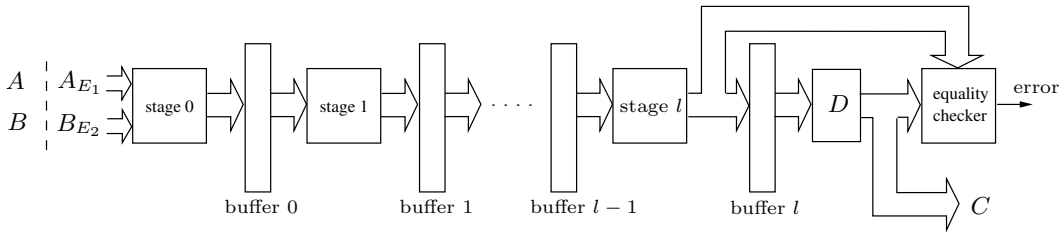
5) Exponentiation:
   a) Compute $A^n = C$; Store in a register;
   b) Compute $(A^2)^n = (A^n)^2 = C^2$; Take square root; Compare this result with that of a).

IV. Pipeline Architecture and Overhead Analysis

The proposed CED scheme is based on time redundancy. A straightforward implementation causes more than 100% time redundancy which may not be desirable. An efficient architecture that can reduce the time overhead significantly is the pipelined architecture. Additionally, this architecture has a moderate area overhead. An example for the pipelined architecture is the systolic array, which is used for efficient arithmetic operations. As shown in Figure 2.a, one buffer is added to the end of the pipeline architecture to store the result of the (first) computation of the arithmetic operation. Then the result of the second computation after decoding will be compared against the content of the last buffer of the pipeline. Another possibility is to start performing the operation with encoded inputs first and then perform the normal computation. In this case, decoder should be placed after the added buffer as shown in Figure 2.b.



(a)



(b)

Fig. 2. General pipelined architecture of an arithmetic operation with the proposed CED

Architecture	t'_{clk}	Clock period overhead (Δ_t)	Latency overhead (Γ_t)
(a)	$\geq$ Max $\{t_e + t_0, \text{Max}\{t_i\}, t_c,$ min$\{t_l + t_d - t_b, t_d - t_b + t_c\}\}$	$t'_{clk} - t_{clk}$	$l \times \Delta_{t_{(a)}} + t_c$
(b)	$\geq$ Max $\{t_e + t_0, \text{Max}\{t_i\}, t_d + t_c\}$	$t'_{clk} - t_{clk}$	$l \times \Delta_{t_{(b)}} + t_d + t_c$

TABLE I

THE TIME OVERHEADS FOR THE DIFFERENT PIPELINED ARCHITECTURES

Suppose that the number of pipeline stages is l. Let the propagation delays of the encoding function, the decoding function, the i^{th} stage of the pipeline (including a buffer), the buffer, the equality checker, and one XOR gate be t_e, t_d, t_i, t_b, t_c, and t_X, respectively. Let t'_{clk} and t_{clk} be the clock period of the pipelined arithmetic operation with and without CED, respectively. Clearly, $t_{clk} \geq \text{Max}\{t_i\}$ for $0 \leq i \leq l$. Also, $t_{clk} \geq t_X$; if this is not the case, let us assume that. For each pipeline architecture of Figure 2, t'_{clk}, the clock period and latency overheads are given in Table I. One can choose one of the above-mentioned architectures which has smaller latency overhead.

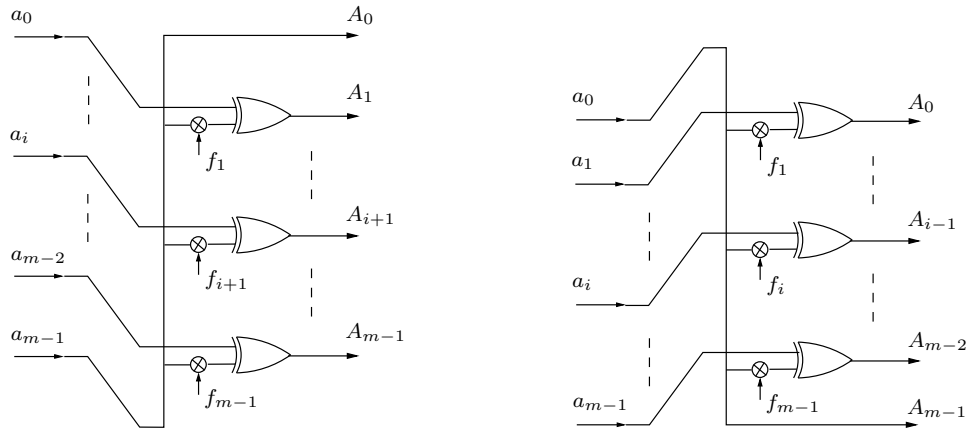
It is worth mentioning that in some pipeline architectures such as systolic arrays, the delay of the equality checker (t_c) can be larger than other delays mentioned in the second column of Table I under t'_{clk}. In this case, one may be able to reduce t_c using a suitable method such as pipelining the checker. This will be addressed with more details in Section V-D.

In the following, the time and area overhead costs of the proposed architectures for each basis are investigated.

A. Time and Area Overheads in PB Operations

The hardware implementations of the PB scaling or inverse scaling are very inexpensive since they need a cyclic shift to right or left, which is free in hardware, and $\omega - 2$ XOR gates, where ω is the Hamming weight of $F(x)$. Figure 3 shows the implementations of both scalings. As shown in the figure, the propagation delay for one PB scaling or inverse scaling is t_X, since there is one level of XOR gates in the implementation.

As mentioned before, the multiplication of a finite field element with x^i or x^{-i} can be implemented by i scalings or inverse scalings, respectively. Therefore, the maximum number



(a) (b)

Fig. 3. (a) Scaling and (b) inverse scaling in PB operations

of XOR gates required for the implementation is $i(\omega - 2)$.

To implement a PB arithmetic operation with CED, we need two encoding functions at maximum. Each of them consists of one scaling or inverse scaling. Additionally, we need one decoding function that consists of two scalings or inverse scalings at maximum, except for exponentiation. Therefore, $4(\omega - 2)$ XOR gates are needed for encoding and decoding functions for a PB arithmetic operation other than exponentiation. We also have:

$$t_e \approx t_X$$

$$t_d \leq 2t_X$$

As an example let us consider a pipelined operation that its clock period with or without CED are exactly equal to the maximum propagation delay of the stages (including the delay of a buffer) and $t_c \leq \text{Max}\{t_i\}$. Then we have:

$$t_{clk} = \text{Max}\{t_i\} \text{ for } 0 \leq i \leq l,$$

$$t'_{clk} = \begin{cases} \text{Max}\{t_i\} + 2t_X & \text{in architecture } (a), \\ \text{Max}\{t_0 + t_X, 2t_X + t_c\} & \text{in architecture } (b). \end{cases}$$

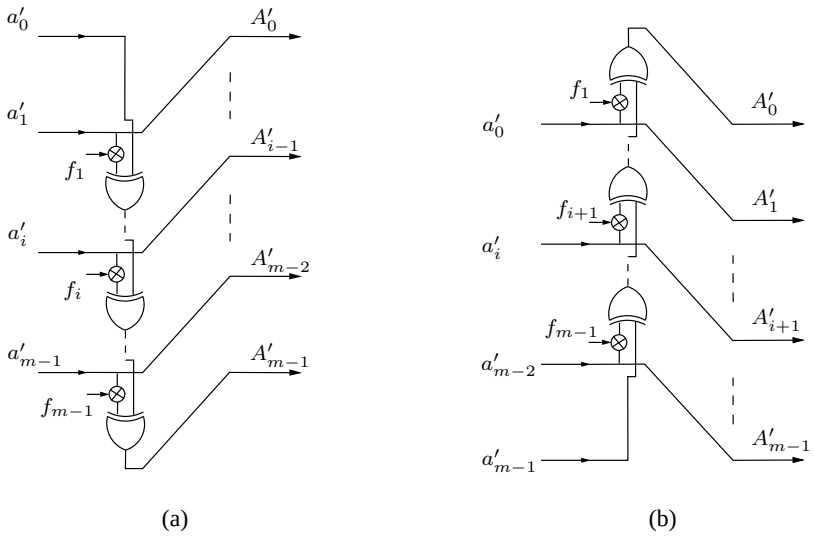
Clearly, $t_i \geq t_X$. Then we have $t'_{clk} \leq \text{Max}\{t_i\} + 2t_X$. Therefore,

$$\Delta_t \leq 2t_X,$$

$$\Gamma_t \leq l(2t_X) + 2t_X + t_c = (2l + 2)t_X + t_c.$$

B. Time and Area Overheads in DB Operations

The hardware implementations of the DB scalings need a cyclic shift to right or left and $\omega-2$ XOR gates, where ω is the Hamming weight of $F(x)$ (see Figure 4). As shown in the figure, the propagation delay for one DB scaling or inverse scaling is $(\omega-2)t_X$ due to the propagation delay of the least significant bit of a scaling or the most significant bit of an inverse scaling.



(a) (b)

Fig. 4. (a) Scaling and (b) inverse scaling in DB operations

Similar to PB arithmetic operations, for a DB arithmetic operation other than exponentiation, $4(\omega-2)$ XOR gates are needed for encoding and decoding functions at maximum. We also have:

$$t_e \approx (\omega-2)t_X$$

$$t_d \leq 2(\omega-2)t_X$$

Considering similar assumptions and arguments in Section IV-A, we have:

$$\Delta_t \leq 2(\omega-2)t_X,$$

$$\Gamma_t \leq [2l(\omega-2)+2]t_X + t_c.$$

C. Time and Area Overheads in NB Operations

Squaring and taking square root of an element represented in NB just needs a cyclic shift to right or left (see Figure 5).

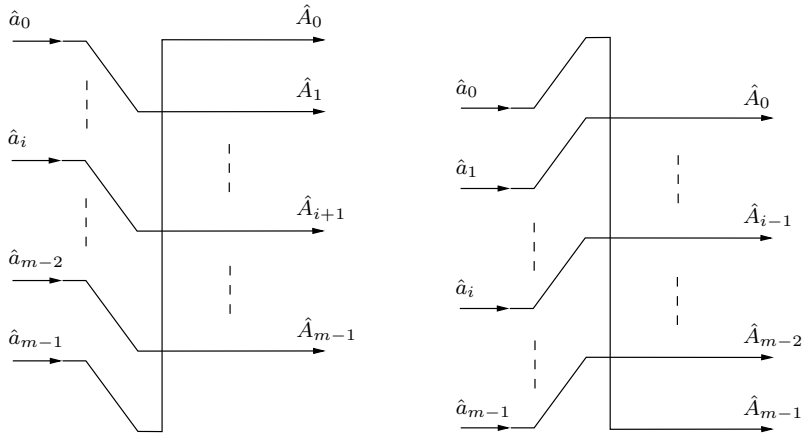


Fig. 5. (a) Squaring and (b) taking square root in NB operations

Therefore, squaring and taking square root have no area and time overhead in hardware implementation and we have:

$$t_e = t_d = 0$$

Furthermore, considering similar assumptions and arguments in Section IV-A, we have:

$$t'_{clk} = t_{clk},$$

$$\Delta_t \approx 0,$$

$$\Gamma_t \approx t_c.$$

The area and time overheads for a pipeline implementation of PB, DB and NB $GF(2^m)$ arithmetic operations with the proposed CED are summarized in Table II. It is worth mentioning that the added buffer is m bits long. Also, m XOR gates and one m-input OR gate are needed in the equality checker unit. Note that for all m, either a trinomial $(\omega = 3)$ or a pentanomial $(\omega = 5)$ can be found as the field defining polynomial.

V. A Closer Look at Polynomial, Dual and Normal Bases Multipliers with CED

In this section, two semi-systolic multipliers for PB and DB bases and two such multipliers for NB basis are presented. Then the time and area complexities of each of them with or without CED are given.

		PB[1]	DB[1]	NB
Maximum area overhead	m-bit Buffer	1	1	1
	2-input XOR	$(m+4\omega-8)$	$(m+4\omega-8)$	m
	m-input OR	1	1	1
Maximum clock period overhead $(\Delta_t)^2$		$2t_X$	$2(\omega-2)t_X$	0
Maximum latency overhead $(\Gamma_t)^2$		$(2l+2)t_X+t_c$	$[2l(\omega-2)+2]t_X+t_c$	t_c

TABLE II

THE TIME AND THE AREA OVERHEADS OF PB, DB AND NB ARITHMETIC OPERATIONS WITH THE PROPOSED CED

[1]The exponentiation operation is not considered.

[2]Assuming that the clock period of the pipelined operation is equal to the maximum propagation delay of the stages (including a buffer) and $t_c \leq \text{Max}\{t_i\}$.

A. A Systolic PB Multiplier with CED

Let $A, B, C \in GF(2^m)$. Then the result of their PB multiplication is as follows:

$$\begin{aligned} C &= A.B \bmod F(x) \\ &= b_0 A + b_1 xA + \cdots + b_{m-1}x^{m-1}A \bmod F(x). \end{aligned}$$

Let $A^{(0)} = A$ and $A^{(i)} = xA^{(i-1)} \bmod F(x)$. Then we have:

$$C = b_0 A^{(0)} + b_1 A^{(1)} + \cdots + b_{m-1}A^{(m-1)} \bmod F(x).$$

Considering that $C = \sum_{j=0}^{m-1} C_j x^j$, we have:

$$C_j = \sum_{i=0}^{m-1} b_i A_j^{(i)}. \tag{7}$$

Expression (7) can be written in a recursive manner as follows:

$$C_j^{(i)} = C_j^{(i-1)} + b_i A_j^{(i)}, \tag{8}$$

where $0 \leq i \leq m-1$ and $c_j^{(-1)} = 0$. Also according to (3) for $1 \leq i \leq m-1$, we have:

$$\begin{aligned} A^{(i)} &= (0, a_0^{(i-1)}, a_1^{(i-1)}, a_{m-2}^{(i-1)}) + a_{m-1}^{(i-1)}(f_0, f_1, f_2, \cdots, f_{m-1}) \\ &= \sum_{j=0}^{m-2} a_j^{(i-1)} x^{j+1} + a_{m-1}^{(i-1)} \sum_{j=0}^{m-1} f_j x^j. \end{aligned}$$

Therefore,

$$A_j^{(i)} = \begin{cases} a_j; & i = 0, \\ a_{j-1}^{(i-1)} + a_{m-1}^{(i-1)} f_j; & 1 \leq i \leq m-1. \end{cases} \tag{9}$$

where $a_{-1}^{(i)} = 0$. Substituting (9) in (8), we have:

$$C_j^{(i)} = \begin{cases} b_0 a_j; & i = 0, \\ C_j^{(i-1)} + b_i \left(a_{j-1}^{(i-1)} + a_{m-1}^{(i-1)} f_j \right); & 1 \leq i \leq m-1. \end{cases} \tag{10}$$

where $0 \leq j \leq m-1$ and $a_{-1}^{(i)} = 0$.

Figure 6 shows a general view of an arbitrary cell of a semi-systolic PB multiplier based on expression (10). The reason that we call it semi-systolic is that each cell may receive a number of input signals from non-adjacent cells or output some signals to them.

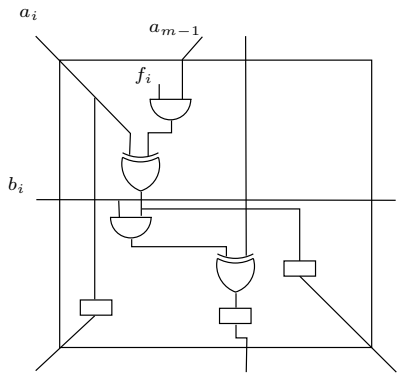
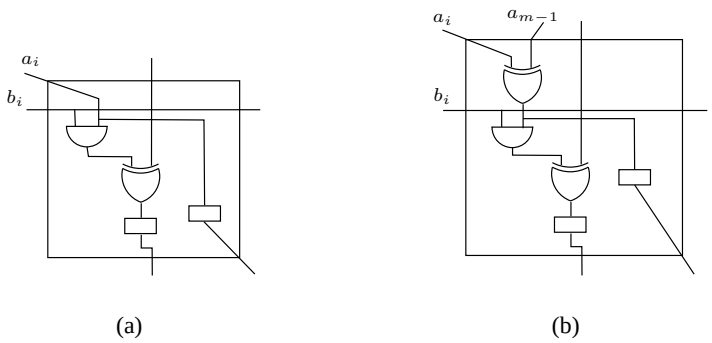


Fig. 6. General cell architecture for a semi-systolic PB mutliplier

It is worth-mentioning that except for f_0 an f_m, the number of non-zero f_i's for $1 \leq i \leq m-1$ is $\omega - 2$, where ω is the Hamming weight of $F(x)$. Therefore, we can have two different cells, one for those that have $f_i = 0$ (type 1) and another for those that have $f_i = 1$ (type 2) as shown in Figure 7.

Let us refer to type 1 and type 2 cells as PBT1 and PBT2. Figure 8 shows the semi-systolic PB multiplier.

In the figure, it is assumed that f_i, f_j and f_k are not zero. Consequently, the cells of the columns i, j and k are type 2 (PBT2). Generally, we have $(m - w + 2)$ PBT1 and $(\omega - 2)$ PBT2 in each row. Furthermore, PBT1 and PBT2 contain (1 AND, 1 XOR, 2 Latches) and (1



(a) (b)

Fig. 7. (a) Type 1 cell and (b) type 2 cell of a semi-systolic PB multiplier

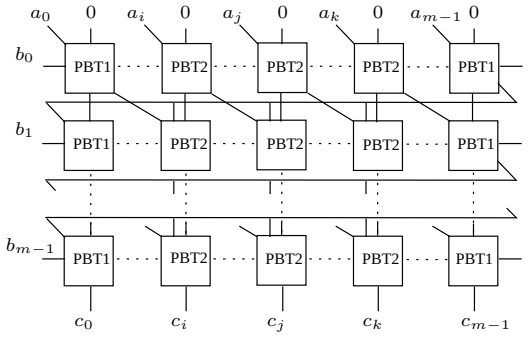


Fig. 8. A semi-systolic PB mutliplier

AND, 2 XORs, 2 Latches), respectively. Table III(a) presents the total number of required gates or latches for a semi-systolic PB multiplier. It is worth-mentioning that an irreducible trinomial $(\omega = 3)$ or pentanomial $(\omega = 5)$ can be found for all m. Also, the computation time for each type of cells is presented in Table III(b).

(a)

AND_2	m^2
XOR_2	$m(m+\omega-2)$
Latch_1	$2m^2$

(b)

cell	computation time per cell
PBT1	$T_A + T_X + T_L$
PBT2	$T_A + 2T_X + T_L$

TABLE III

SPACE AND TIME COMPLEXITIES OF THE SEMI-SYSTOLIC PB MULTIPLIER

For the purpose of error detection, we applied the method discussed in Section III-A. In other

words, each encoding function is one forward scaling and the decoding function is two inverse scaling. Table IV shows the area and time complexities of this work along with a number of previously published systolic or semi-systolic PB multipliers without CED capability and with CED capability if applicable.

In [21] and [9], two checkers (comparators) are used. One is to detect errors at the output of the circuit and another is to detect errors on global lines. These lines are horizontal lines that connect a bit of one of the inputs (say input B) to all cells in a row of the multiplier. However, in our proposed scheme since both inputs are encoded, error in the global lines can also be detected.

According to Table IV, the space complexities and latencies of the multipliers with or without CED presented in this work seems to be better compared to the other multipliers mentioned in the table. Note that $\omega = 3$ or $\omega = 5$ and the latency of the multiplier without CED in [21] is same as our work but that multiplier is not general. Apparently, the cell time complexity of our work is not among the best. However, this may not be considered as a drawback in multipliers with CED because the bottle neck for determining the minimum clock period is usually the propagation delay of equality checkers, not the cell time complexity. This will be further investigated in Section V-D.

B. A Systolic DB Multiplier with CED

Suppose that $A, B, C \in GF(2^m)$ and $C = A.B \mod F(x)$. The formulation for DB multiplication is similar to PB one except for the following. Let $A^{(0)} = A$. Then according to Lemma 1 for $1 \leq i \leq m-1$, we have:

$$\begin{aligned} A^{(i)} &= (a_1^{'(i-1)}, a_2^{'(i-1)}, \cdots, a_{m-1}^{'(i-1)}, \sum_{k=0}^{m-1} f_k a_k^{'(i-1)}) \\ &= \sum_{j=1}^{m-1} a_j^{'(i-1)} y_{j-1} + \sum_{k=0}^{m-1} f_k a_k^{'(i-1)} y_{m-1}. \end{aligned}$$

Therefore,

$$A_j^{(i)} = \begin{cases} a'_j; & i = 0, 0 \leq j \leq m-1, \\ a_{j+1}^{'(i-1)}; & 1 \leq i \leq m-1, 0 \leq j \leq m-2, \\ \sum_{k=0}^{m-1} f_k a_k^{'(i-1)}; & j = m-1, 1 \leq i \leq m-1. \end{cases} \quad (11)$$

Multipliers	[21]		[9]		[31]	[34]	[19]	This work	
Generating polynomial	AOP		General		General	General	Trinomial	General	
CED	No	Yes	No	Yes	No	No	No	No	Yes
Cell no.	m^2	m^2+3m	m^2+m	m^2+2m	m^2	m^2	m^2	m^2	m^2
Space complexity	[(1)]								
XOR_2	$2m^2$	$2m^2+7m$	—	$2m$	—	$2m^2$	m^2+m	$m(m+\omega-2)$	$m^2+(\omega-1)m+4(\omega-2)$
XOR_3	—	—	m^2+m	m^2+2m	m^2	—	—	—	—
AND_2	$2m^2$	$2m^2+4m$	$2m^2+2m$	$2m^2+4m$	$2m^2$	$2m^2$	m^2	m^2	m^2
AND_3	—	m	—	—	—	—	—	—	—
$Latch_1$	$2m^2$	$2m^2+6m$	$3.5m^2+3.5m$	$3.5m^2+7.5m+1$	$7m^2$	$7m^2$	$3.5m^2+m$[(2)]	$2m^2$	$2m^2+m$
OR_m	—	2	—	2	—	—	—	—	1
2-1 Switch	—	—	—	—	—	—	m	—	—
Min. CLK period	$2T_A+2T_X+T_L$	[(3)]	$T_A+T_{3X}+T_L$	[(3)]	$T_A+T_X+2T_L$	$T_A+T_X+2T_L$	$T_A+T_X+T_L$	$T_A+2T_X+T_L$	[(3)]
Latency	m	$m+2$	$m+1$	$m+5$	$3m$	$3m$	$m+k$	m	$m+1$

[(1)]The space complexity of this work has to be more than the complexity mentioned in the table, because the corresponding encoding and decoding functions were not considered.

[(2)]Each cell needs 3 or 4 latches. Hence, we estimate that $3.5m^2$ latches are needed for all cells as well as m extra latches at the end of computation.

[(3)]Should be similarly computed according to Table I and Max$\{t_i\}$ is same as that of the multiplier without CED.

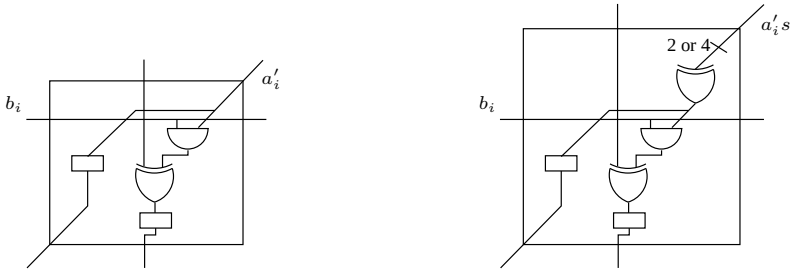
TABLE IV

SPACE AND TIME COMPLEXITIES OF A NUMBER OF SYSTOLIC OR SEMI-SYSTOLIC PB MULTIPLIERS

Substituting (11) in (8), we have:

$$C_j^{(i)} = \begin{cases} b_0 a'_j; & i = 0, 0 \leq j \leq m-1, \\ C_j^{(i-1)} + b_i a_{j+1}^{'(i-1)}; & 1 \leq i \leq m-1, 0 \leq j \leq m-2, \\ C_j^{(i-1)} + b_i \sum_{k=0}^{m-1} f_k a_k^{'(i-1)}; & j = m-1, 1 \leq i \leq m-1. \end{cases} \quad (12)$$

Considering Expression (12), we can consider two types of cells for semi-systolic DB multipliers as shown in Figure 9. Except for the last column of the multiplier ($j \neq m-1$), Type 1 cells are used (see Figure 9(a)). These cells requires one 2-input AND gate, one 2-input XOR gate and two 1-bit latches.



(a) (b)

Fig. 9. (a) Type 1 cell and (b) type 2 cell of a semi-systolic DB multiplier

Type 2 cells (Figure 9(b)) are used in the last column. In addition to the gates and latches needed for Type 1 cells, Type2 cells require one extra 2-input or 4-input XOR gate depending on whether the defining polynomial of the underlying field is a trinomial ($\omega = 3$) or pentanomial ($\omega = 5$), respectively. Figure 10 shows a semi-systolic DB multiplier.

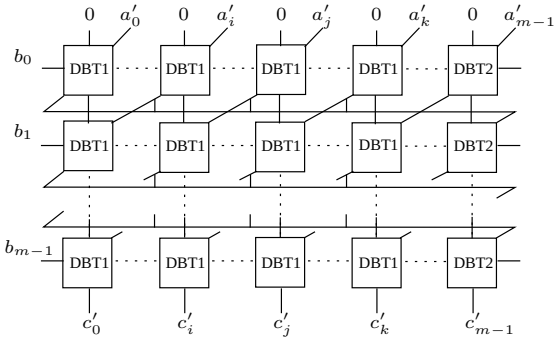


Fig. 10. A semi-systolic DB multiplier

The error detection scheme presented in Section III-B is applied to this multiplier. Hence, the encoding and decoding functions are one forward scaling and two inverse scaling, respectively. Table V summarizes the time and space complexities of this work and a number of previously published multipliers with and without CED capability as appropriate.

According to Table V the multipliers (with and without CED) presented in this work can be considered as the best ones in terms of space complexity and latency. It is worth-mentioning that latches are relatively more area consuming components and hence the multiplier in [19] requires more space than our work. The cell time complexity of our work is not better than the other multipliers, however, this does not imply that the minimum clock period (MCP) of our work with CED is larger than another multiplier with CED. This case mostly happens when the other delay parameters for determining MCP (according to Table I), such as propagation delay of the equality checker, is larger than the cell time complexity.

C. A Systolic NB Multiplier with CED

In this section two multipliers for the optimal normal bases of type I and type II are presented.

1) Type I Optimal Normal Basis (ONB1): Suppose that $m+1$ is a prime number and 2 is primitive in $GF(m+1)$. Then the field defining polynomial, can be chosen to be $F(x)=\sum_{i=0}^{m} x^i$, which is an all-one irreducible polynomial over $GF(2^m)$. Let x be the root of $F(x)$. Since $F(x)|(x^{m+1}-1)$, then we have $x^{m+1}\equiv 1$. Therefore, the set of normal basis presented in Section II-A, can be reduced accordingly. The resulting set has m linearly independent elements [23] as follows and is referred to as type I optimal normal basis:

$$\left\{x, x^2, \cdots, x^{m-1}, x^m\right\}.$$

It is worth mentioning that the order of the elements in the above set is different from the conventional representation of a normal basis. Therefore, we define the following permutation functions that basically change the order of the coefficients in the normal basis representations:

$$\Gamma_1 : NB \Longrightarrow ONB1,$$

$$\Gamma_1^{-1} : ONB1 \Longrightarrow NB.$$

Suppose that the NB and ONB1 representations of $A \in GF(2^m)$ are $A=\hat{a}_0 x+\hat{a}_1 x^2+\hat{a}_2 x^{2^2}+ \ldots+\hat{a}_{m-1}x^{2^{m-1}}$ and $A=a_1x+a_2x^2+a_3x^3+\ldots+a_mx^m$, respectively. Also, let us assume that

Multipliers	[12]	[20]		[19]	This work			
CED	No	No	Yes	No	No		Yes	
Cell no.	m^2	m^2	m^2+m	m^2	m^2		m^2	
Space complexity					$\omega=3$	$\omega=5$	$\omega=3$	$\omega=5$
XOR_2	$2m^2$	$2m^2$	$3m^2+3m-2$	m^2+m	m^2+m	m^2-m	m^2+2m+4	m^2+12
XOR_4	—	—	—	—	—	m	—	m
AND_2	$2m^2$	$2m^2$	$3m^2-m$	m^2	m^2		m^2	
$Latch_1$	$7m^2$	$5m^2$	$10m^2-2m-4$	$3.5m^2+m$	$2m^2$		$2m^2+m$	
OR_m	—	—	1	—	—		1	
2-1 Switch	—	—	—	m	—		—	
Min. CLK period	$T_A+T_X+T_L$	$T_A+T_X+T_L$	((1))	$T_A+T_X+T_L$	$T_A+2T_X+T_L$	$T_A+T_X+T_{4X}+T_L$	((2))	
Latency	$3m$	$3m$	$3m+1$	$m+k$	m		$m+1$	

((1))Should be computed according to Table I, where Max$\{t_i\}=T_A+2T_X+T_L$.

((2))Should be similarly computed according to Table I and Max$\{t_i\}$ is same as that of the multiplier without CED.

TABLE V

SPACE AND TIME COMPLEXITIES OF A NUMBER OF SYSTOLIC OR SEMI-SYSTOLIC DB MULTIPLIERS

after permutation we have $a_j = \hat{a}_i$, Then:

$$j = 2^i \bmod (m+1).$$

Now, suppose that $A, B, C \in GF(2^m)$ are represented in ONB1. Hence, $A = \sum_{i=0}^{m} \hat{a}_i x^i$ and $B = \sum_{i=0}^{m} \hat{b}_i x^i$, where $\hat{a}_0 = \hat{b}_0 = 0$. Therefore, considering previous notations we have:

$$\begin{aligned} C &= A.B \bmod F(x) \\ &= \hat{b}_0 A^{(0)} + \hat{b}_1 A^{(1)} + \cdots + \hat{b}_{m-1} A^{(m-1)} + \hat{b}_m A^{(m)} \bmod F(x). \end{aligned} \tag{13}$$

Expression (13) is very similar to PB multiplication except that this multiplication is $(m+1)$ bits long. Additionally, for $1 \leq i \leq m$ we have:

$$\begin{aligned} A^{(i)} &= \sum_{j=0}^{m} \hat{a}_j^{(i-1)} x^{j+1} \bmod (x^{m+1} - 1) \\ &= \hat{a}_m^{(i-1)} x^{m+1} + \sum_{j=0}^{m-1} \hat{a}_j^{(i-1)} x^{j+1} \bmod (x^{m+1} - 1) \\ &= \sum_{j=0}^{m} \hat{a}_{<j-1>}^{(i-1)} x^j. \end{aligned} \tag{14}$$

where $< j-1 >= j-1 \bmod m+1$. In fact, $A^{(i)}$ is one bit rotation of $A^{(i-1)}$ in such a way that the MSB of $A^{(i-1)}$ shifts out and rotates back to the LSB position. Clearly, $A^{(0)} = A$.

Now, similar to expression (8) for $1 \leq i, j \leq m$, we have:

$$C_j^{(i)} = C_j^{(i-1)} + \hat{b}_i A_j^{(i)},$$

where $C_j^{(-1)} = 0$. Therefore,

$$C_j^{(i)} = \begin{cases} \hat{b}_0 \hat{a}_j; & i = 0, \\ C_j^{(i-1)} + \hat{b}_i \hat{a}_{<j-1>}^{(i-1)}; & 1 \leq i \leq m-1. \end{cases} \tag{15}$$

where $0 \leq j \leq m-1$ and $< j-1 >= j-1 \bmod m+1$.

Note that $C^{(m)}$ is not necessarily in ONB1 representation, since $C_0^{(m)}$ may not be zero. To resolve this issue, one can zero out $C_0^{(m)}$ as follows:

$$C = C^{(m)} + C_0^{(m)}.F(x).$$

Figure 11 shows a cell of a semi-systolic ONB1 multiplier.

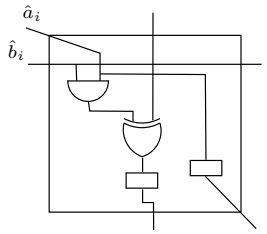


Fig. 11. The cell of a semi-systolic ONB1 multiplier

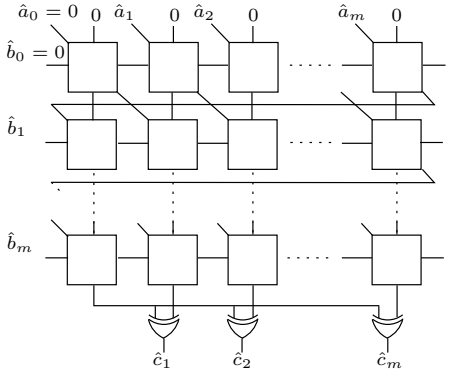


Fig. 12. A semi-systolic ONB1 multiplier

It is worth mentioning that since $F(x)$ is an all-one-polynomial, one can simply add the LSB of $C^{(m)}$ with all other bits of $C^{(m)}$ in the hardware implementation. Figure 12 shows a semi-systolic ONB1 multiplier.

Furthermore, as shown in Figure 12, $b_0 = 0$ is one input of all AND gates of the cells in the first row. Therefore, the first row can be omitted and then a_i's should be fed into the first row after one rotation (see Figure 13). Clearly, in this way the space and latency of the multiplier are slightly reduced.

The error detection scheme presented in Section III-C is applied to this multiplier. The encoding and decoding functions are one or two shifts to the left or right. Apparently, the encodings should be performed before the permutation Γ_1 and the decoding should be performed after the inverse permutation Γ_1^{-1}. The time and space complexities of this work with and without CED capability are presented in Table VI.

According to Table VI, the ONB1 multiplier presented here is better than that in [19] in terms of space complexity and latency and they are the same in terms of cell time complexity.

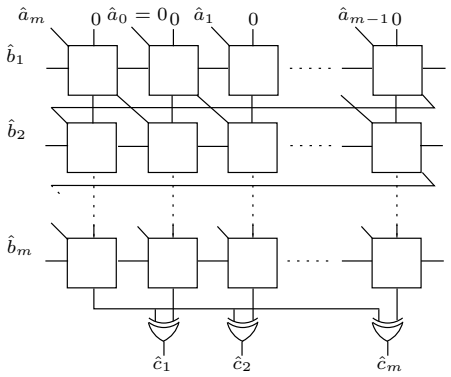


Fig. 13. A semi-systolic ONB1 multiplier with m rows

Furthermore, to the best of our knowledge this is the first work that addressed the CED in the NB multipliers.

2) Type II Optimal Normal Basis (ONB2): Suppose that $2m+1$ is a prime number and either of the following conditions holds:

- 2 is primitive in $GF(2m+1)$, or
- $2m+1 = 3$ (mod 4) and the multiplicative order of 2 modulo $2m+1$ is m.

Then, the field $GF(2^m)$ can be constructed using the normal element $\gamma+\gamma^{-1}$ [23] and the basis for field representation is referred to as type II optimal normal basis as follows:

$$\left\{\gamma+\gamma^{-1}, \gamma^2+\gamma^{-2}, \cdots, \gamma^{2^{m-1}}+\gamma^{-2^{m-1}}\right\},$$

where γ is a primitive $(2m+1)^{th}$ root of unity. Hence, for $1 \leq i \leq 2m-1$, $\gamma^i = 1$ only when $i = 2m+1$. It is worth mentioning that the above set can be rewritten as follows [30]:

$$\left\{\gamma+\gamma^{-1}, \gamma^2+\gamma^{-2}, \gamma^3+\gamma^{-3}, \cdots, \gamma^m+\gamma^{-m}\right\}.$$

Similar to ONB1, a permutation function is needed to convert an NB representation to an ONB2 representation and vise versa. Hence,

$$\Gamma_2 : NB \Longrightarrow ONB2,$$

$$\Gamma_2^{-1} : ONB2 \Longrightarrow NB.$$

Suppose that the NB and ONB2 representations of $A \in GF(2^m)$ are $A = \hat{a}_0 x + \hat{a}_1 x^2 + \hat{a}_2 x^{2^2} + ... + \hat{a}_{m-1} x^{2^{m-1}}$ and $A = a_1(\gamma+\gamma^{-1}) + a_2(\gamma^2+\gamma^{-2}) + a_3(\gamma^3+\gamma^{-3}) + \cdots + a_m(\gamma^m+\gamma^{-m})$,

respectively. Also, let us assume that after permutation we have $a_j = \hat{a}_i$. Then:

$$j = \begin{cases} k; & 1 \le k \le m, \\ (2m+1) - k; & m+1 \le k \le 2m. \end{cases},$$

where $k = 2^i \mod (2m+1)$.

Now, suppose that $A, B, C \in GF(2^m)$ are represented in ONB2, e.g., $A = \sum_{i=1}^{m} \hat{a}_i(\gamma^i + \gamma^{-i})$. Then following [30], we have:

$$\begin{aligned} C = A.B &= \sum_{i=1}^{m}\sum_{j=1}^{m} \hat{a}_i\hat{b}_j(\gamma^i + \gamma^{-i})(\gamma^j + \gamma^{-j}) \\ &= \sum_{i=1}^{m}\sum_{j=1}^{m} \hat{a}_i\hat{b}_j \left[(\gamma^{(i-j)} + \gamma^{-(i-j)}) + (\gamma^{(i+j)} + \gamma^{-(i+j)})\right] \\ &= \sum_{i=1}^{m}\sum_{j=1}^{m} \hat{a}_i\hat{b}_j \left[\gamma^{(i-j)} + \gamma^{-(i-j)}\right] + \sum_{i=1}^{m}\sum_{j=1}^{m} \hat{a}_i\hat{b}_j \left[\gamma^{(i+j)} + \gamma^{-(i+j)}\right] \\ &= C_1 + C_2. \end{aligned}$$

$$\begin{aligned} C_1 &= \sum_{i=1}^{m}\sum_{j=1}^{i} \hat{a}_i\hat{b}_j \left[\gamma^{(i-j)} + \gamma^{-(i-j)}\right] + \sum_{i=1}^{m}\sum_{j=i+1}^{m} \hat{a}_i\hat{b}_j \left[\gamma^{(i-j)} + \gamma^{-(i-j)}\right] \\ &= C_{11} + C_{12}. \end{aligned}$$

$$\begin{aligned} C_2 &= \sum_{i=1}^{m}\sum_{j=1}^{m-i} \hat{a}_i\hat{b}_j \left[\gamma^{(i+j)} + \gamma^{-(i+j)}\right] + \sum_{i=1}^{m}\sum_{j=m-i+1}^{m} \hat{a}_i\hat{b}_j \left[\gamma^{(i+j)} + \gamma^{-(i+j)}\right] \\ &= C_{21} + C_{22}. \end{aligned}$$

Now, we adjust the power of the basis for C_{11}, C_{12}, C_{21} and C_{22} by changing the variables as follows:

1) Let $j - i = -k$. Then we have:

$$\begin{aligned} C_{11} &= \sum_{i=1}^{m}\sum_{k=0}^{i-1} \hat{a}_i\hat{b}_{i-k} \left(\gamma^{-k} + \gamma^{k}\right) \\ &= \sum_{i=1}^{m}\sum_{k=1}^{i-1} \hat{a}_i\hat{b}_{i-k} \left(\gamma^{k} + \gamma^{-k}\right). \end{aligned}$$

Note that for $i = 1$, the upper bound of the second summation becomes negative and the result is all zero.

2) Let $j-i=k$. Then we have:

$$C_{12}=\sum_{i=1}^{m}\sum_{k=1}^{m-i}\hat{a}_i\hat{b}_{i+k}\left(\gamma^k+\gamma^{-k}\right).$$

3) Let $j+i=k$. Then we have:

$$C_{21}=\sum_{i=1}^{m}\sum_{k=i+1}^{m}\hat{a}_i\hat{b}_{k-i}\left(\gamma^k+\gamma^{-k}\right).$$

4) For C_{22}, the powers of the basis is larger than m. Hence using $\gamma^{2m+1}=1$, we have:

$$C_{22}=\sum_{i=1}^{m}\sum_{j=m-i+1}^{m}\hat{a}_i\hat{b}_j\left[\gamma^{2m+1-(i+j)}+\gamma^{-(2m+1)+(i+j)}\right].$$

Now, let $2m+1-(j+i)=k$. Then we have:

$$C_{22}=\sum_{i=1}^{m}\sum_{k=m-i+1}^{m}\hat{a}_i\hat{b}_{2m+1-(i+k)}\left(\gamma^k+\gamma^{-k}\right).$$

To derive a single closed form for the multiplication, we have:

$$\begin{aligned}C_I&=C_{11}+C_{21}\\&=\sum_{i=1}^{m}\sum_{k=1}^{m}\hat{a}_i\hat{b}_{|i-k|}\left(\gamma^k+\gamma^{-k}\right),\end{aligned}$$

where $|i-k|$ is the absolute value of $(i-k)$ and $\hat{b}_0=0$. Also, we have:

$$\begin{aligned}C_{II}&=C_{12}+C_{22}\\&=\sum_{i=1}^{m}\sum_{k=1}^{m}\hat{a}_i\hat{b}_{||i+k||}\left(\gamma^k+\gamma^{-k}\right),\end{aligned}$$

where $||i+k||=\begin{cases}i+k; & i+k\leq m,\\(2m+1)-(i+k); & i+k>m.\end{cases}$

Finally, we have:

$$C=C_I+C_{II}=\sum_{i=1}^{m}\sum_{k=1}^{m}\hat{a}_i\left(\hat{b}_{|i-k|}+\hat{b}_{||i+k||}\right)\left(\gamma^k+\gamma^{-k}\right),$$

where $\hat{b}_0=0$. Therefore, each coefficient of C can be computed as:

$$\hat{c}_k=\sum_{i=1}^{m}\hat{a}_i\left(\hat{b}_{|i-k|}+\hat{b}_{||i+k||}\right).$$

To have a recursive (rolled) form which is suitable for systolic arrays, we can write the above expression as follows:

$$\begin{aligned} C_k^{(i)} &= C_k^{(i-1)} + \hat{a}_i \left(\hat{b}_{|i-k|} + \hat{b}_{||i+k||} \right) \\ &= C_k^{(i-1)} + \hat{a}_i \hat{b}_{|i-k|} + \hat{a}_i \hat{b}_{||i+k||}, \end{aligned} \tag{16}$$

where $1 \leq i, k \leq m$, $C_k^{(-1)} = 0$, $\hat{c}_k = C_k^{(m)}$ and $\hat{b}_0 = 0$.

Figure 14 shows two possible implementations for a cell of the ONB2 semi-systolic multiplier according to Expression 16. One can choose one of the above-mentioned implementations based on space and/or time complexities.

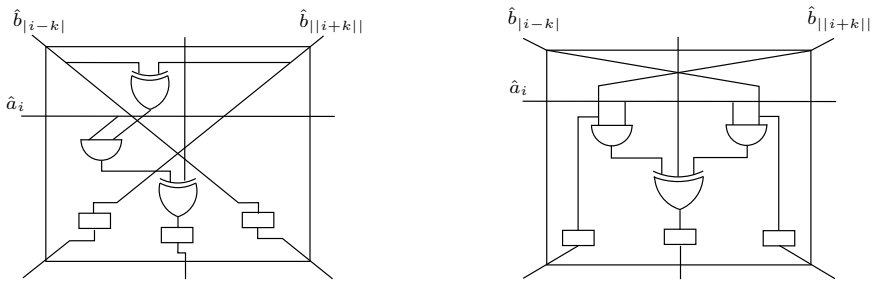


Fig. 14. Two cells of a semi-systolic ONB2 multiplier with same functionality

In this work, we have chosen the cell shown in Figure 14(b). A complete semi-systolic ONB2 multiplier is shown in Figure 15.

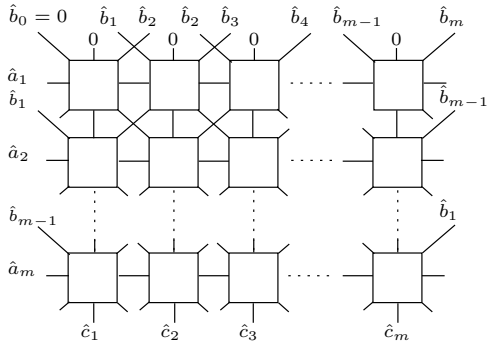


Fig. 15. A semi-systolic ONB2 multiplier

Similar to the ONB1 multiplier, the error detection scheme presented in Section III-C is applied to this multiplier. The encoding and decoding functions are one or two shifts to the left

or right. It is worth-mentioning that the encodings and the decoding should be performed before the permutation function Γ_2 and after inverse permutation function Γ_2^{-1}, respectively. The time and space complexities of this work for ONB1 and ONB2 along with a number of other related previous work with and/or without CED capability are presented in Table VI.

According to Table VI, the ONB2 multiplier presented here can be considered among the best in terms of space complexity and is the best in terms of latency. Additionally as mentioned earlier, to the best of our knowledge this is the first work addressing the CED in the NB multipliers.

D. Some Notes About Delays of Cells and Equality Checkers

The clock rate of a pipeline architecture can be determined according to a number of parameters presented in Table I (second column). The propagation delays of the stages of some pipeline architectures such as systolic or semi-systolic arrays are small. Particularly for these architectures, one important parameter to determine the clock rate is the delay of the equality checker. In the following this issue is investigated.

The equality checker is basically one level of XOR gates to compare the equality of the bits of two inputs and one OR unit to determine the final error signal as shown in Figure 16.

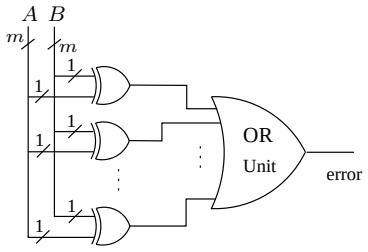


Fig. 16. An m-bit equality checker

A straightforward method to design the OR unit is to use 2-input OR gates. Then m such gates in $\lceil \log_2 m \rceil$ levels are needed. Therefore, $t_c = t_X + \lceil \log_2 m \rceil t_{OR_2}$. Alternatively, one can construct the m-input OR unit using $\lceil \log_n m \rceil$ levels of n-input OR gates. To determine the efficient one, we performed a number of simulations in Cadence^{TM} at switch-level (transistor-level). For $m = 163$, we constructed 163-input OR unit in the following ways:

- 1-level 163-input OR
- 2-level 13-input OR

Multipliers	[19]		[18]	This work			
ONB Type	I	II	II	I		II	
CED	No	No	No	No	Yes	No	Yes
Cell no.	$(m+1)^2$	m^2	m^2	$m(m+1)$	$m(m+1)$	m^2	
Space complexity							
XOR_2	$(m+1)^2+m$	m^2+m	m	$m(m+1)$	m^2+2m	—	m
XOR_3	—	—	m^2	—	—	m^2	m^2
AND_2	$(m+1)^2$	m^2	$2m^2+m$	$m(m+1)$	$m(m+1)$	$2m^2$	$2m^2$
$Latch_1$	$3.5(m+1)^2$	$3.5m^2+m$	$5m^2$	$2m(m+1)$	$2m^2+3m$	$3m^2$	$3m^2+m$
OR_m	—	—	—	—	1	—	1
Min. CLK period	$T_A+T_X+T_L$	$T_A+T_X+T_L$	$T_A+T_{3X}+T_L$	$T_A+T_X+T_L$	((1))	$T_A+T_{3X}+T_L$	((1))
Latency	$m+1$	$m+1$	$m+1$	m	$m+1$	m	$m+1$

((1))Should be computed according to Table I and Max$\{t_i\}$ is same as that multiplier without CED.

TABLE VI

SPACE AND TIME COMPLEXITIES OF A NUMBER OF SYSTOLIC OR SEMI-SYSTOLIC ONB MULTIPLIERS

- 3-level 6-input OR
- 4-level 4-input OR
- 8-level 2-input OR

For the purpose of simulation, gates were modeled using *ratioed* logic that uses only one PMOS transistor in pull-up network. We used $0.18\mu m$ technology. Also, we initialized all inputs of the gates with zero and after a while we changed the value of only one of them to one. The result of the transient response simulation is shown in Figure 17.

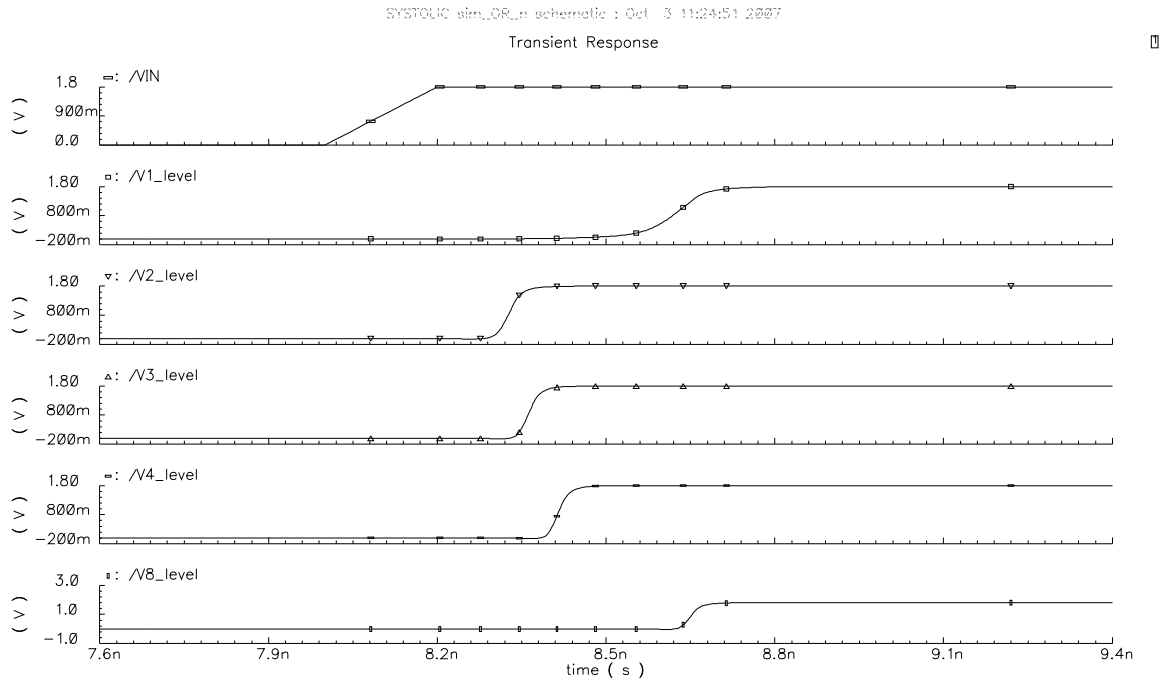


Fig. 17. The propagation delays for different ways of implementing an m-input OR unit

According to the figure, the best propagation delay is for 2-level 13-input OR design. Furthermore, both 8-level 2-input OR and 1-level 163-input OR designs are significantly slower. The reason that the 1-level 163-input OR design is slow is that all 163 NMOS transistors of the pull-down network are connected to each other in parallel. This produces a large parasitic capacitance at the output of the gate, which is time-consuming to be discharged when one NMOS transistor turns on.

It is worth-mentioning if one needs to use the standard cells of a library, the best choice is 4-level 4-input OR design because the 13-input OR gate are often unavailable in the standard

cells and it has the smallest propagation delay after 2-level 13-input design.

Moreover, if after designing the equality checker, t_c becomes larger than the other parameters for determining the clock rate of the pipeline, and decreasing the clock rate is not desirable, one can implement the equality checker in a pipeline manner. In other words, the equality checker can be divided into two or more stages such that the propagation delays of its stages become smaller than the desired clock period. Clearly, this approach results in a larger latency in terms of the required clock cycles.

VI. Error Detection Capability

In this section the capabilities of the error detection schemes discussed earlier are evaluated. With regard to the duration of faults, we consider two categories for faults in our simulations as follows:

- Transient faults: These faults are assumed to occur only in one of the two runs.
- Permanent (or intermittent) faults: These faults occur in both runs.

The fault coverage for the transient faults is 100%, because either these faults make the output erroneous or they are masked. In the first case, the result of the first run and the second run are different. Hence, the fault causing errors is detected. It is worth-mentioning that fault coverage is the percentage ratio of the number of detected and masked faults over the number of all injected faults. Therefore, the fault coverage is 100%.

For permanent (or intermittent) faults, we performed a number of simulation-based fault injections on the PB, DB, ONB1 and ONB2 multipliers presented in Section V. Fault injections were performed in a C model of the multiplier. We injected stuck-at faults (both stuck-at 1 and stuck-at 0) at the input and output pins of the gates of the multiplier. In the proposed scheme, same faults are injected in the same locations of the circuits in both runs. Fault injection in a complete multiplier with a field degree of approximately 163 is extremely time consuming. Therefore, faults were injected in only one randomly chosen row of cells of the semi-systolic multipliers. We performed the fault injections in two phases as discussed below.

1) Single-Bit Stuck-at Faults: In this experiment, only one-bit stuck-at fault was injected during the multiplication. As mentioned earlier, the location of a fault can be at the input and/or output pins of gates. Hence, to perform fault injection, a multiplexer can be placed at the fault location, where the control signal of the multiplexer selects between the original value of that

point and the fault. Moreover, the fault value can be chosen to be 1 or 0. This is shown in Figure 18.

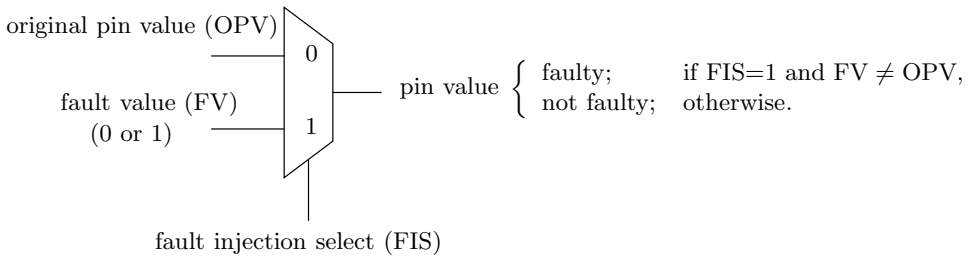


Fig. 18. Conventional fault injection at a gate pin

The number of faults that can be injected to a multiplier for each set of inputs depends on the number of AND gates and XOR gates of that multiplier (see Table IV, Table V and Table VI). It is worth-mentioning that the output pins of AND gates in each cell, which are direct inputs of the XOR gates, were not injected. In this experiment, we simulated the fault injection for PB, DB, ONB1 and ONB2 multipliers. Each multiplier was simulated for one million random inputs and for every input, all the above mentioned single-bit stuck-at faults were injected. Table VII shows the number of injected single-bit stuck-at faults for each set of inputs as well as the error detection capability (fault coverage) for each multiplier. Note that the field $GF(2^{163})$ cannot be represented using ONB1 or ONB2. Therefore, we chose other fields of close degrees instead.

Type of the multiplier	No. of stuck-at faults	No. of random inputs	Error detection capability
$GF(2^{163})$ PB	1650	1000000	99.08%
$GF(2^{163})$ DB	1642	1000000	100%
$GF(2^{162})$ ONB1	1632	1000000	97.37%
$GF(2^{173})$ ONB2	2770	1000000	99.98%

TABLE VII

ERROR DETECTION CAPABILITY OF THE SCHEME FOR A $GF(2^{163})$ PB MULTIPLIER AGAINST STUCK-AT FAULTS

2) Multiple-Bit Stuck-at Faults: To inject multiple-bit stuck-at faults, the locations for the injections were randomly selected from the above mentioned single-bit fault locations. Then one stuck-at 0 or stuck-at 1 was randomly injected there. We injected 500 multiple-bit stuck-at

faults per each set of inputs. Furthermore, one million random sets of inputs were simulated in each experiment. For example for a $GF(2^{163})$ PB multiplier, there are 825 single-bit stuck-at fault locations. Therefore, the number of accesses to those locations, whether or not any fault is injected, is $(1000000 * 500 * 825 =)$ 412.5 billions. All simulations for PB, DB, ONB1 and ONB2 multipliers result in 100% error detection capability.

VII. Concurrent Error Correction Strategy

The RESO method can also be used for error correction. Let us assume that the faults are transient. Also, we similarly assume that locations of these faults, occurred naturally or injected by an attacker, are random. This scheme cannot correct the permanent faults with a significant probability. For the purpose of error correction, we should have N computations, where $N \geq 3$. Therefore, we should have $N-1$ sets of encoding functions, where each set contains two encoding functions for both inputs of an operation (if applicable). Moreover, $N-1$ decoding functions are needed. Let $C_0 = f(A, B)$, $A_{E_{i,1}} = E_{i,1}(A)$ and $B_{E_{i,2}} = E_{i,2}(B)$, where $E_{i,1}$ and $E_{i,2}$ are two encoding functions of a set. Considering that D_i and C_i are the i^{th} decoding function and the result of the i^{th} computation, respectively, we have:

$$C_i = D_i(f(A_{E_{i,1}}, B_{E_{i,2}})),$$

where $1 \leq i \leq N-1$.

Now, let us assume that we need to have $M < N$ identical results from N computations to output one of them as the correct result. This is similar to an N-of-M system and needs an N-of-M majority voter since at least M computations should be exactly same (see [17]). The voter performs the following tasks:

- If at least M C_i's are identical, for $0 \leq i \leq N-1$, output one of them and there is no error.
- Otherwise, an error detected but the output cannot be corrected.

One well-known N-of-M system is 2-of-3 one, which is considered for CEC in this work. Figure 19 shows a general architecture of an operation with concurrent error correction scheme based on RESO method and using a 2-of-3 majority voter. The error signal in the figure is set to one if all of three computations are different.

In the following, the above-mentioned concurrent error correction strategy for each basis are investigated.

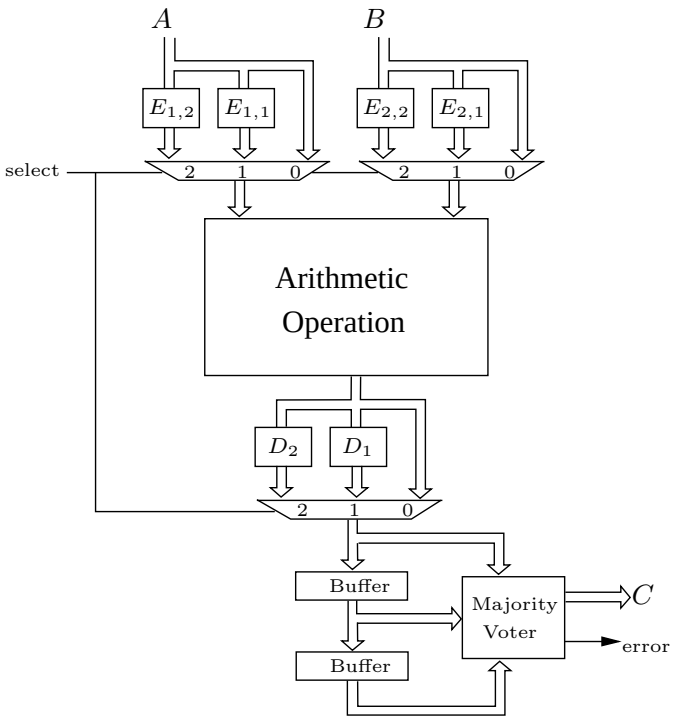


Fig. 19. General architecture for the arithmetic operations with CEC (using a 2-of-3 system)

A. CEC for Polynomial, Dual and Optimal Normal Bases Arithmetic Operations

In this section, the encoding and decoding functions for CEC of each basis are given as follows:

- Encoding and decoding functions for PB and DB
  1) Addition/Subtraction:

     $E_{1,1} = x$, $E_{1,2} = x$, $D_1 = x^{-1}$

     $E_{2,1} = x^{-1}$, $E_{2,2} = x^{-1}$, $D_2 = x$
  2) Multiplication:

     $E_{1,1} = x$, $E_{1,2} = x$, $D_1 = x^{-2}$

     $E_{2,1} = x^{-1}$, $E_{2,2} = x^{-1}$, $D_2 = x^2$
  3) Inversion:

     $E_{1,1} = x$, $D_1 = x$

     $E_{2,1} = x^{-1}$, $D_1 = x^{-1}$
  4) Division:

$E_{1,1} = x$, $E_{1,2} = x^{-1}$, $D_1 = x^{-2}$

$E_{2,1} = x^{-1}$, $E_{2,2} = x$, $D_2 = x^2$

5) Exponentiation:

$E_{1,1} = x$, $D_1 = x^{-n}$

$E_{2,1} = x^{-1}$, $D_2 = x^n$

- Encoding and decoding functions for NB

1) Addition/Subtraction/Multiplication/Division:

$E_{1,1} = E_{1,2} =$ squaring, $D_1 =$ taking square root,

$E_{2,1} = E_{2,2} =$ taking square root, $D_2 =$ squaring.

2) Inversion/Exponentiation:

$E_{1,1} =$ squaring, $D_1 =$ taking square root,

$E_{2,1} =$ taking square root, $D_2 =$ squaring.

VIII. Conclusions

This paper presents a number of schemes, which are efficient for pipelined architectures and are based on recomputing with shifted operands (RESO) method. These schemes have been developed to concurrently detect errors in polynomial, dual, type I and type II optimal normal bases arithmetic operations. We have also presented one semi-systolic multiplier for each of above-mentioned bases and applied the CED scheme to them. We have compared these multipliers with a number of previously published systolic and/or semi-systolic ones. The results show that this scheme can be considered among the best. Also, a simulation-based fault injection has been performed for each of the multipliers. Results of the simulations for single stuck-at faults show 99.08%, 100%, 97.37% and 99.98% detection capability for polynomial, dual, type I and type II bases multipliers, respectively. The simulations also show that this scheme can detect multiple stuck-at faults with 100% probability for the above-mentioned multipliers. Finally, we also commented on how RESO can be used for concurrent error correction to deal with transient faults.

Acknowledgments

The work was supported in part by an NSERC grant awarded to Dr. Hasan. The authors also would like to thank Dr. Miguel F. Anjos for letting them run part of the simulations on his com-

puter. The authors are also grateful for being able to use some of the computing facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET:www.sharcnet.ca).

REFERENCES

[1] S. Bayat-Sarmadi and M. A. Hasan. Run-time error detection of polynomial basis multiplication using linear codes. In *Proceedings of the 18th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 102–110, Monterey, CA, 2005.

[2] S. Bayat-Sarmadi and M. A. Hasan. Detecting errors in a polynomial basis multiplier using multiple parity bits for both inputs. In *proceeding of the 25th IEEE International Conference on Computer Design (ICCD)*, pages 368–375, Lake Tahoe, CA, 2007.

[3] S. Bayat-Sarmadi and M. A. Hasan. On concurrent detection of errors in polynomial basis multiplication. *IEEE Trans. VLSI*, 15(4):413–426, April 2007.

[4] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri. Error analysis and detection procedures for a hardware implementation of the Advanced Encryption Standard. *IEEE Trans. Comp.*, 52(4):1–14, April 2003.

[5] I. Biehl, B. Meyer, and V. Muller. Differential fault attacks on elliptic curve cryptosystems. In *Proc. 20th Int'l Conf. CRYPTO*, pages 131–146. Springer-Verlag, 2000.

[6] J. Blomer, M. Otto, and J.-P. Seifert. Sign change fault attacks on elliptic curve cryptosystems. Cryptology eprint archive, Report 2004/227, 2004. http://eprint.iacr.org/2004/227.

[7] D. Boneh, R. Demillo, and R. Lipton. On the improtance of checking cryptographic protocols for faults. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, pages 37–51. Springer-Verlag, 1997.

[8] C. W. Chiou. Concurrent error detection in array multipliers for $GF(2^m)$ fields. *Electronics Letters*, 38(14):688–689, July 2002.

[9] C.-W. Chiou, C.-Y. Lee, A.-W. Deng, and J.-M. Lin. Concurrent error detection in montgomery multiplication over $GF(2^m)$. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E89-A(2):566–574, 2006.

[10] M. Ciet and M. Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Designs, Codes and Cryptography*, 36(1):33–43, July 2005.

[11] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Info. Theory*, 31(4):469– 472, July 1985.

[12] S. Fenn, M. Benaissa, and O. Taylor. Dual basis systolic multipliers for $GF(2^m)$. *Computers and Digital Techniques, IEE Proceedings*, 144(1):43–46, January 1997.

[13] S. Fenn, M. Gossel, M. Benaissa, and D. Taylor. Online error detection for bit-serial multipliers in $GF(2^m)$. *J. Electronics Testing: Theory and Applications*, 13:29–40, 1998.

[14] G. Gaubatz and B. Sunar. Robust finite field arithmetic for fault-tolerant public-key cryptography. In *Proc. 3rd Workshop on Fault Tolerance and Diagnosis in Cryptography (FTDC)*, pages 196–210, 2006.

[15] G. Gaubatz, B. Sunar, and M. G. Karpovsky. Non-linear residue codes for robust public-key arithmetic. In *Proc. 3rd Workshop on Fault Tolerance and Diagnosis in Cryptography (FTDC)*, pages 173–184, 2006.

[16] N. Joshi, K. Wu, and R. Karri. Concurrent error detection for involutional functions with applications in fault-tolerant cryptographic hardware design. *IEEE Trans. CAD/ICAS*, 25(6):1163– 1169, June 2006.

[17] I. Koren and C. Krishna. *Fault-Tolerant Systems*. Morgan-Kaufman, San Francisco, CA, 2007.

[18] S. Kwon. A low complexity and a low latency bit parallel systolic multiplier over $GF(2^m)$ using an optimal normal basis of type II. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, Washington, DC, USA, 2003. IEEE Computer Society.

[19] C.-Y. Lee, Y.-H. Chen, C.-W. Chiou, and J.-M. Lin. Unified parallel systolic multiplier over $GF(2^m)$. *Journal of Computer Science and Technology*, 22(1):28–38, January 2007.

[20] C.-Y. Lee, C. W. Chiou, and J.-M. Lin. Concurrent error detection in a bit-parallel systolic multiplier for dual basis of gf(2m). *J. Electron. Test.*, 21(5):539–549, 2005.

[21] C.-Y. Lee, C. W. Chiou, and J.-M. Lin. Concurrent error detection in a polynomial basis multiplier over $GF(2^m)$. *J. Electron. Test.*, 22(2):143–150, 2006.

[22] D. A. McGrew and J. Viega. The Galois/Counter mode of operation (GCM). NIST Modes of Operation for Symmetric Key Block Ciphers, 2005.

[23] A. J. Menezes, I. F. Blake, G. XuHong, R. C. Mullin, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields*. Springer-Verlag, 1993.

[24] J. H. Patel and L. Y. Fung. Concurrent error detection in ALU's by REcomputing with Shifted Operands. *IEEE Transactions on Computers*, C-31(7):589– 595, July 1982.

[25] J. H. Patel and L. Y. Fung. Concurrent error detection in multiply and divide arrays. *IEEE Transactions on Computers*, C-32(4):417– 422, April 1983.

[26] D. Pradhan and M. Chatterjee. GLFSR-a new test pattern generator for built-in-self-test. In *Proc. Int'l Test Conf.*, pages 481–490, 1994.

[27] A. Reyhani-Masoleh and M. A. Hasan. Error detection in polynomial basis multipliers over binary extension fields. In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 515–528. Springer-Verlag, 2002.

[28] A. Reyhani-Masoleh and M. A. Hasan. Towards fault-tolerant cryptographic computations over finite fields. *ACM Trans. Embedded Comp. Sys.*, 3(3):593–613, August 2004.

[29] A. Reyhani-Masoleh and M. A. Hasan. Fault detection architectures for field multiplication using polynomial bases. *IEEE Transactions on Computers-Special Issue on Fault Diagnosis and Tolerance in Cryptography*, 55(9):1089–1103, September 2006.

[30] B. Sunar and C. Ko. An efficient optimal normal basis type II multiplier. *IEEE Trans. Computers*, 50(1):83–87, January 2001.

[31] C.-L. Wang and J.-L. Lin. Systolic array implementation of multipliers for finite fields $GF(2^m)$. *IEEE Trans. Circuits and Systems*, 38(7):796–800, July 1991.

[32] M. Wang and I. F. Blake. Bit serial multiplication in finite fields. *SIAM J. Discret. Math.*, 3(1):140–148, 1990.

[33] S. B. Wicker and V. K. Bhargava, editors. *Reed-Solomon Codes and Their Applications*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[34] C.-S. Yeh, I. Reed, and T. Truong. Systolic multipliers for finite fields $GF(2^m)$. *IEEE Trans. Computers*, C-3315(4):357–360, April 1984.