3. Executive Summary

In many cryptographic schemes and DSPs, operations based on finite fields are of prime importance. Not surprisingly, a lot of research have gone into developing fast and efficient schemes for Finite Field Arithmetic Processing. Coming up with an architecture is a task by itself as these systems can get incredibly complicated given the mathematics involved in them. The implementation of just a bit-parallel multiplier can contain millions of gates. Given that these systems can get incredibly complicated, faults in the logic is not unexpected. These faults can give us erroneous results and the thought that these systems find their applications in extremely critical fields, gives us no room to negotiate with them. The aim of this project is to investigate design methodologies for reliable low power Galois Field Arithmetic Processor.

The focus will on the various aspects of hardware reliability and testing, while optimizing criteria such as power consumptions, area overhead, and reducing delay overhead. Particular focus will be given to the efficient designs of hardware specifically for certain types of algebraic structures, such as groups, rings, and fields, as these types of hardware have particular applications for cryptography and data security and also error control coding

In particular the the new structure should tolerate specific faults in on-line computation. For example parity prediction has been used in certain arithmetic designs but the area and power overhead can exceed that of duplication.However when parity prediction is used in conjunction with duplication it has been observed that the power performance can be better than duplication.The overall objective is to explore some example designs for Galois field operations and demonstrate under what circumstances parity prediction can provide superior performance.

The main contributions and achievements:

- *I created a High Performance Galois Field Arithmetic Processor with Addition and Multiplication Blocks. The Algorithm implemented is a version of [34].*
- *I implemented four versions of a Digit Serial Multiplier, optimized the area and critical paths.*
- *I synthesized them in 180nm and 90nm technology nodes using Synopsys® Tools. I also implemented them on various FPGA devices to compare the area and power. I also placeed and routed the designs by using SoC Encounter to compare the area. Observations are stated in Section(11).*
- *The designs were then made Fault Tolerant, using a version of the Algorithm specified in [19].*
- *Error Correction Schemes were provided to the design and up to 40 stuck at faults can be corrected using my method. Given that a single bit error can permeate into multiple bits in a field multiplier, this feature is clearly an important achievement. Further, the aims and objectives laid out during the conception of the project, all have been achieved and the evaluation of each one of them is given in the Section(12) and (13).*

Contents

List of Figures

List of Tables

4. Introduction

Finite field arithmetic has been the topic of research ever since its application in cryptographic systems was found. The operations such as inversion and doubling can be efficiently performed using repeated multiplication over a finite filed. Addition is no harder. Arithmetic operations over the Galois Field GF(2^m) is thoroughly discussed in literature, ex. [21], and [16].

A finite field multiplier is quite complex as compared to the integer or floating-point multipliers. Given the size of the field used in cryptographic systems, there can be millions of gates in the implementation and surprisingly faults are not unexpected. Since cryptographic systems are really important for data security and protection against malicious data and fault attacks, there reliability cannot be compromised. The importance of cryptographic systems being error free is discussed in many articles like, [6], [12]. Many error detection and correction schemes for such systems have been proposed previously. Parity prediction was found to be a recurring theme in most of these research papers. It is not surprising given its advantages over other fault tolerant schemes. Cryptographic systems follow AES (Advanced Encryption Standards). In AES a message is encrypted or decrypted with a key and the entire operation consists of 11 rounds and has four operations. [12]. A message is divided into a number of bytes and each byte is an element of a given GF(2^m) generated by an irreducible polynomial for a value of m. During each round of the algorithm a set of operations are performed on the GF(2^m). In [5, p.492-505] parity predicted for each byte is compared to the actual parity, specifically in their scheme 16 parity bits are produced and verified at various stages of operation(rounds of operation). Further more, many low cost methods for concurrent error detection is also proposed, [37]. The cryptographic systems need to fault tolerant because if presence they make the systems vulnerable to attacks which can result in the loss of important information.

In this project, many of these fault tolerant implementations of GF Arithmetic Processors are investigated for low power and cost efficient improvements without compromising their functionality and reliability they provide for these crypto-systems.

In the Section(6), the mathematics that binds all these systems together and the intricacies of it are discussed in length. A careful understanding of these is required for designing and comprehending these complex systems. Then different architectures that forms the basis of design process that followed, are also discussed (Section(7)).

The project is divided into two development stages,

- Designing an High Performance Galois Field Multiplier
- Introducing Fault Tolerance to the design

The Report follows the same structure throughout. In the Section(10), the design of the multiplier and the adder blocks are discussed. In the Section that follows, how fault tolerance is introduced to them are discussed. The Analysis of the results obtained and evaluation of the objectives achieved are discussed in the Sections (12) and (13) respectively and finally Section(14), briefly discusses the scope for future improvements to the project.

5. Aims and Objectives

The aim of this project is to investigate design methodologies for reliable low power Galois Field Arithmetic Processor. Designing efficient implementations of hardware specifically for certain types of algebraic structures discussed before will be the main focus. The design methodologies for the Galois Field are to be carefully analyzed in order to come up with a new design and the reliability of which is to be carefully tested. The main objectives of the project can be itemized into four.

(1) **Investigate design methodologies for reliable low power Galois Field Arithmetic Processor.**
An efficient design for the processor cannot be done without understanding the mathematics behind it. The design should be then converted into hardware logic so that it can be incorporated into various other systems, be it Cryptography or DSP and the further stages of the design would then involve as little of Mathematical complexity as possible. That is the level of abstraction aimed. As a starting point for the research into this field, as most of the other authors [31], [13], [32], the one proposed in [21] and [22] is considered. The structures given in this scheme is the most basic. In [13], this design is fine tuned and reduces the area and delay.
In [13], the possibility of improvising the Reduction Matrix and thus a better implementation is proposed. This is the same area that will be explored in the project. The aim of the project is to investigate these designs and consider the advantages weighed against the complexities that making them fault tolerant will pose. The produced design should be reliable, and fault tolerant.

(2) **Consider the potential for delay overhead that may be introduced by redundant logic.**
An efficient design is a myth. Depending on what the constraint is, one can always come up with a superior one. This is the philosophy that drives the entire electronic design technology. The better - faster - smaller seems to be the driving force. Delay in systems is unavoidable. One has to deal with it and make their designs as fast as possible. Introduction of the concurrent testing circuits, parity prediction, we are introducing a redundant logic unit. This can further increase the delay of the circuit. The parity prediction algorithm is explained before. The architecture that is proposed for this parity prediction will be further investigated so that the delays caused by the extra hardware can be decreased to a minimum.

(3) **Investigate new methods for fault tolerant architectures for GF arithmetic processor**
This is the main objective of the project, which is the amalgamation of both the points stated before. A good GF Arithmetic processor, with excellent parity prediction system and thus tolerant to faults. The possibility of SEC/DED is explored. As a single bit parity can detect only single bit/odd number of faults, error detection/correction algorithms will be investigated and implemented in the system.

(4) **Analyze the overhead (area, power, delay) for new structure and reliability** As discussed before, there is always an area/power/delay overhead while

designing fault tolerant systems. But thats the price paid for attaining fault tolerance. Earlier only memory was made fault tolerant, and a lot of research has already been made and efficient fault tolerant systems are already in place. The faults in logic was always neglected. In the age were the devices are becoming so small and the efficiency of the systems cannot be compromised at all, logic systems need to be fault tolerant as well. Designing the extra hardware for fault tolerance is as complex as the design itself. The area/power/delay characteristics of this new addition or changes made to the original system in order to include fault tolerance, need to be considered and reduced to a minimum, so that the system has the advantage of being resilient to faults at the same time have least changes to the area/power/delay characteristics of the original.

6. Preliminary Mathematical Definitions and Theorems

The ideas of groups, rings, fields, congruences are continually referred through out this report. The academic references for the following sections are [30] and [16]. The ideas and definitions are purely mathematical.

6.1. **Congruences.**

Theorem 1. *Given any two integers,* a *and* b*, and another positive integer* n*,* a *is said to be* congruent *to* y modulo n *if* (a - b) *is divisible by* n.
It is mathematically represented as:

$$a \equiv b \ (mod\ n)$$

It should be noted that congruence is an equivalence relation. [33]
A relation is said to be equivalence if it is *reflexive*, *symmetric* and *transitive.* i.e.

- Every element is related to itself.
- Every two elements share a commutative relationship.
- Every three elements share an associative relationship.

6.2. **Multiplicative Inverse.**

Theorem 2. *Two elements* a *and* b *of* Z_n*,(set of* n *integers) such that*

$$\text{ab mod n} = 1$$

then, b *is said to be the* multiplicative inverse *of* a*. It is represented as:*

$$b = a^{-1} \ mod\ n$$

6.3. **Fermat's Theorem.**

Theorem 3. *[16] Let* p *be a prime number. Any integer* x *that satisfies*

$$x^p \equiv x \ (mod\ p),$$

and any integer x *not divisible by* p *satisfies,*

$$x^{p-1} \equiv 1 (mod\ p).$$

Proof:
If x is not divisible by p and if $ix \equiv jx(modp)$, that is $(i-j)x = qp$, then $i \equiv j(modp)$. Thus

$$(1x)(2x)...((p-1)x) \equiv 1.2.....(p-1)(\text{mod p})$$

As p *-1* above multiples of x are distinct and non-zero, they must be congruent to 1,2,3, .. , p - 1 in some order. So,

$$(p-1)!x^{p-1} \equiv (p-1)!\text{mod p}$$

or

$$(p-1)!(x^{p-1}-1) \equiv 0 \text{mod p}$$

As p does not divide *(p-1)!*,

$$(x^{p-1}-1) \equiv 0 (\text{mod p})$$

that is,

$$x^{p-1} \equiv 1(modp) \qquad \text{and} \qquad x^p \equiv x(modp)$$

If x is divisible by p, then $x^p \equiv x \equiv 0(modp)$. Corollary:
Let p be a prime. If x is not divisible by p and if $r \equiv s(modp-1)$, then

$$x^r \equiv x^s \text{ (mod p)}$$

6.4. **Sets.** According to Cantor, who is considered as the father of set theory, a set is a collection of distinguishable objects or items. For example, set of all integers,
$Z = \{...,3, 2, 1, 0, 1, 2, 3, ...\}$

Theorem 4. *Depending on the assumption that defines a set we can determine whether an object belongs to the set or not. [33]*

6.5. **Relations.**

Theorem 5. *A relationship that can be defined on a particular set S is called a relation. A binary relation, which is relevant for this paper, is defined as a relation between any two objects considered in a definite order.*

6.6. **Groups.**

Theorem 6. *A set* G *along with a binary operation '' is called a* Group*, if it comply with the following axioms,*

(1) *The operation over G is associative*

$$\forall a, b, c \in G, a * (b * c) = (a * b) * c$$

(2) *Identity with respect to the operation under consideration exists*

$$\forall a \in G, \exists i \in G, \mid \ a * i = i * a = a$$

(3) *There exists an inverse for each element in the group w.r.t the operation under consideration*

$$\forall a \in G, \exists a^{-1} \in G, \mid \ a * a^{-1} = a^{-1} * a = i$$

Example: The set of integers Z forms a group under $+$. The additive identity is 0.

6.7. **Rings.**

Theorem 7. *A set* R *along with two binary operations + and '' is called a* Ring*, if it comply with the following axioms,*

(1) *(R, +) forms a commutative group with identity element 0*
(2) *The set R is associative.*

$$\forall x, y, z \in R, x * (y * z) = (x * y) * z$$

(3) *The operations are distributive on each other.*

$$\forall x, y, z \in R, x * (y + z) = (x * y) + (x * z) and (x + y) * z = (x * z) + (y * z)$$

Example: The set Z_n is a ring with respect to addition and multiplicative modulo n *operations.*

6.8. **Fields.**

Theorem 8. *A set* F *along with two binary operations + and '' is called a* Field*, if it comply with the following axioms,*

(1) *(F, +,*) forms a commutative ring.*

$$\forall x, y \in F, x * y = y * x$$

(2) *There exists a multiplicative inverse for all non-zero elements in the set F.*

Example: The set Z_n is a ring with respect to addition and multiplicative modulo n *operations.*

6.9. **Characteristic.**

Theorem 9. *The characteristic of a field is the least positive integer* m*, such that*

$$\sum_{i=1}^{m} 1 = 0$$

Example: The set Z_p forms a Field with respect to the operations addition and multiplication modulo p, if and only if p is a prime number. If p is prime, then the characteristic of the field is p.
Example: Addition and Multiplication over Z_5

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

*	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Theorem 10. *A field* S *is a subfield of* F *if* S *itself is a field with respect to the same operations of* F *and* F *is said to be an* extension field *of* S*.*
If $S \neq F$, then S *is called a* proper subfield *of* F*.*

6.10. **Finite Fields.**

Theorem 11. *A field containing finite number of elements in it is called a* finite field*. It is also known as* GALOIS FIELD *named in the honor of Évariste Galois.*

Theorem 12. *The number of elements in a Galois Field is called the* order *of the field. If a Galois Field has* q *elements then it is denoted as* GF(q)*.*
The order of a non-zero element $a \in GF(q)$ is the smallest positive integer $n \in Z_+$ such that

$$a^n \equiv 1$$

In a Galois Field,

(1) $$Characteristic\ of\ GF(q) = \ q\ if\ q\ is\ prime$$

(2) $$= \ p\ if\ q\ is\ a\ power\ of\ q$$

Theorem 13. *For any $a \in GF(q), a^{q-1} \equiv 1$*

6.11. **Polynomials.**

Theorem 14. *A polynomial of degree* n *over the Galois Field,* GF(p) *is of the following form,*

$$F(x) = \sum_{i=0}^{n} a_i * x^i = a_0 + a_1 x + ... + a_n x^n$$

where $a_i \in GF(p)$ *and* n*, is a positive integer.*

The Polynomial *F(x)* is called a *monic polynomial if* $a_n = 1$
If it cannot be written as a product of polynomials of smaller degree it is called an *irreducible polynomial.*
The order of the polynomial is the least positive number t, such that $x^t - 1$ divides *F(x).* The polynomial is said to be *primitive* if its order is $p^m - 1$. If all the co-efficients of the polynomial is one then its called *AOP, all-ones polynomial*

6.11.1. *Field Defining Polynomial.*

Theorem 15. *Let* F(x) *be an irreducible polynomial of degree m over GF(p) . Then all polynomials over GF(p) of degree less than m form a finite field GF(p^m) of order p^m if addition and multiplication are performed modulo F(x).*
F(x) is defined as the Field-defining polynomial. [31]

Since *GF(p^m)* is an extension field of *GF(p)*, we can consider it as a vector space of dimension m over *GF(p).* Thus the *basis* of the vector space could be any set of m linearly dependent elements, e.g. $x_0, x_1, x_2, ..., x_{m-1}$. A linear combination of these elements is as follows,

$$A = a_0 x_0 + a_1 x_1 + ... + a_{m-1} x_{m-1}$$

where $a_i \in GF(p)$.
There are some well known bases used for representing these extension fields,

(1) Canonical bases
(2) Normal bases
(3) Dual bases
(4) Triangular bases

Representations and further implementations of the operations are done over the polynomial bases(canonical), hence it is explained below.

6.11.2. *Canonical Basis.*

Theorem 16. *Let r be the root of an irreducible polynomial F(x) over GF(p) of degree m, i.e F(r) = 0, Canonical basis is defined by the following set: [30]*

$$1, r, r^2, r^3, ..., r^{m-1}$$

If $a \in GF(p^m)$, its trace is defined as,

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

6.11.3. *Dual Basis.* The dual basis of the polynomial basis is a basis $y_0, y_1, ..., y_{m-1}$, where $y_i \in GF(2^m)$, such that, [31] [30]

$$Tr(y_i x^j) = 0, i \neq j \tag{3}$$

$$= 1, otherwise \tag{4}$$

6.11.4. *Normal Basis.* The normal basis is of the following form,

$$z, z^p, z^{p^2}, z^{p^3}, ..., z^{p^{m-1}}$$

where $z \in GF(p^m)$

In order to further understand the applications of such representations, we look at the following, [13]
Let $F(x) = x^m + \sum_{i=0}^{m-1} f_i x^i$ be a degree m, monic, irreducible polynomial defined over *GF(2)*, such that, $f_i \in GF(2)$ for all values of i from *0 to m-1.* Let $\alpha \in GF(2^m)$ be a root of the polynomial. Then the standard/polynomial basis can be written as $\{1, \alpha, \alpha^2, ..., \alpha^{m-1}\}$ and each element in the Field, $GF(2^m)$ can be written with respect to the elements in the polynomial basis.
Let $A \in GF(2^m)$, it can be written as, $A = \sum_{i=0}^{m-1} a_i \alpha^i, a_i \in GF(2) = \{0, 1\}$.

7. Arithmetic Operations over Fields

Arithmetic operations like addition, subtraction and multiplication can be performed over these fields. Further operations like inversion, exponentiation and division can be performed by repeated multiplication, as shown in the sections given below. Polynomial basis for representation of elements of binary extension field *GF(2^m)* is used for calculations. Algorithms for implementing these operations are discussed below.

7.1. **Addition.** As trivial as it is, the addition and subtraction are the most basic of operations that is to be performed by any processor. It can be hardly boasted as a functionality. Nevertheless, it is an integral part of the processor. Addition for example, is used in performing other operations like multiplication, inversion, division, exponentiation and also point-additions and point-doubling in elliptic curve groups. [27] [4]. In Finite Field Arithmetic hardware implementation, *addition* is implemented by *XOR* gates and *D* flip flops.
Let the field defining polynomial which defines the finite field *GF(2^m)* is given by,

$$F(x) = x^m + f_{m-1}.x^{m-1} + ... + f_2.x^2 + f_1.x + 1 \tag{5}$$

where the co-efficients $f_i \in GF(2)$. As defined by the Theorem: 16, it generates a PB(Polynomial Basis), given by,

$$1, \alpha, \alpha^2, \alpha^3, ..., \alpha^{m-1}$$

α is a root of the equation *F(x)*.
Let A and B be any two elements in *GF(2^m)*. They can be represented by the polynomial basis in the form of polynomials of degree *m-1* as defined in 16. Also, as mentioned before, addition is performed by bit-by-bit XOR operations of the pair of operand words. Addition of any two field elements is given by,

Algorithm 1. Addition of $A, B \in GF(2^m)$
Require: $A = \sum_{i=0}^{m-1} a_i\alpha^i, a_i \in GF(2) =$ **{0, 1} and** $B = \sum_{i=0}^{m-1} b_i\alpha^i, b_i \in GF(2)$
Ensure: $C \equiv A + B \ \ mod \ \ f(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i, c_i \in GF(2)$**.**

(1) $C \leftarrow 0$
(2) for i = 0 to m-1 do
(3) $c_i = a_i \oplus b_i$
(4) end for
(5) Return C

$\oplus$ **operator is XOR.**

7.2. **Polynomial Basis Multiplication over** $GF(2^m)$ **and Summary of Previous Work.** Let $F(x) = x^m + f_{m-1}.x^{m-1} + ... + f_2.x^2 + f_1.x + 1$ be the degree m, monic, irreducible polynomial that defines the field $GF(2^m)$ and $f_i \in GF(2)$ for all the values of i from *0* to *m - 1*. If α is the root of the polynomial then we know that the polynomial basis is given by $1, \alpha, \alpha^2, \alpha^3, ..., \alpha^{m-1}$.
From Section: **??**basisdef), we know that every element in $GF(2^m)$ can be represented by this basis. Let $A \in GF(2^m)$ can be represented as, [13]

$$A = \sum_{i=0}^{m-1} a_i\alpha^i, a_i \in GF(2) = \{0, 1\},$$

Multiplication of any two such elements A and B in $GF(2^m)$ is given by,

$$C \equiv A.B \ \ mod \ \ f(x) \tag{6}$$

This operation is a two step process,

(1) Polynomial Multiplication
(2) Reduction Modulo using the irreducible polynomial

STEP 1: Polynomial Multiplication:
The product is a polynomial D of degree not less than *2(m-1)* obtained by direct multiplication of the polynomial basis representations of the elements. [13] [16] i.e.

$$D = (\sum_{i=0}^{m-1} a_i \alpha^i)(\sum_{j=0}^{m-1} b_j \alpha^j) = \sum_{k=0}^{2(m-1)} d_k \alpha^k, \tag{7}$$

It can be obtained as follows,

$$\begin{vmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ . \\ . \\ . \\ d_{m-2} \\ d_{m-1} \\ d_m \\ d_{m+1} \\ . \\ . \\ . \\ d_{2m-2} \end{vmatrix} = \begin{vmatrix} a_0 & 0 & 0 & . & . & . & ... & 0 & 0 \\ a_1 & a_0 & 0 & . & . & . & ... & 0 & 0 \\ a_2 & a_1 & a_0 & . & . & . & ... & 0 & 0 \\ a_3 & a_2 & a_1 & a_0 & . & . & ... & 0 & 0 \\ . & . & . & . & . & . & ... & 0 & 0 \\ . & . & . & . & . & . & ... & 0 & 0 \\ . & . & . & . & . & . & ... & 0 & 0 \\ a_{m-2} & a_{m-3} & . & . & . & . & ... & a_0 & 0 \\ a_{m-1} & a_{m-2} & . & . & . & . & ... & a_1 & a_0 \\ 0 & a_{m-1} & . & . & . & . & ... & 0 & a_1 \\ 0 & 0 & a_{m-1} & . & . & . & ... & 0 & a_2 \\ . & . & . & . & . & . & ... & 0 & 0 \\ . & . & . & . & . & . & ... & 0 & 0 \\ . & . & . & . & . & . & ... & 0 & 0 \\ 0 & 0 & 0 & . & . & . & ... & 0 & a_{m-1} \end{vmatrix} . \begin{vmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ . \\ . \\ . \\ b_{m-2} \\ b_{m-1} \end{vmatrix}$$

This can be represented as follows, [16]

$$d_k = \begin{cases} \sum_{i=0}^{k} a_i b_{k-i} & \text{k = 0, 1, ... , m-1} \\ \sum_{i=k}^{2(m-1)} a_{(k-i)+(m-1)} b_{i-(m-1)} & \text{k = m, ... , 2(m-1)} \end{cases}$$

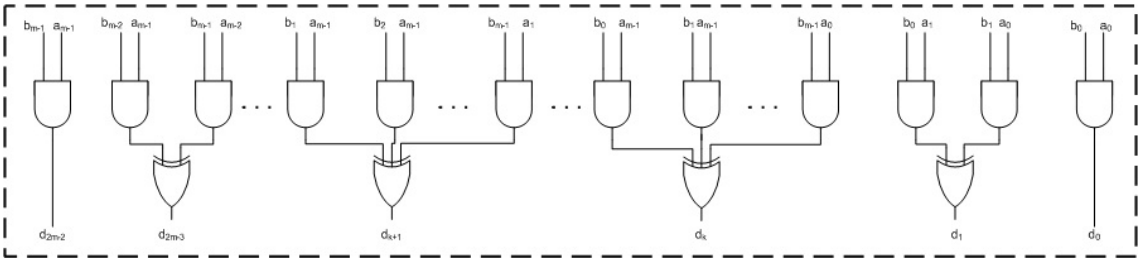


FIGURE 1. Datapath of the Multiplier [16]

Remark 1. *Delay Calculations*
Let the delay of a single AND gate is Δ_{AND} and a single XOR gate is Δ_{XOR}. XOR delay is through the longest binary tree, which is of depth m due to the d_{m-1}^{th} coordinate. Hence the total delay in the polynomial multiplication is

$$Total\ \ Delay = \Delta_{AND} + \lceil log_2 m \rceil \Delta_{XOR} \tag{8}$$

STEP 2: Reduction Modulo F(x):
The aim of this step is to reduce the product polynomial of degree $2(m-1)$ using the monic polynomial of degree m, such that the final product will be of degree less than or equal to m. Reduction modulo can be viewed as a linear mapping of all the 2(m-1) coordinates in D into m coordinates of C. This mapping can be represented in the matrix form as follows, [16] [13],

$$\begin{vmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ . \\ . \\ . \\ c_{m-2} \\ c_{m-1} \end{vmatrix} = \begin{vmatrix} 1 & 0 & \cdots & 0 & r_{0,0} & r_{0,1} & \cdots & r_{0,m-2} \\ 0 & 1 & \cdots & 0 & r_{1,0} & r_{1,1} & \cdots & r_{1,m-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & r_{m-1,0} & r_{m-1,1} & \cdots & r_{m-1,m-2} \end{vmatrix} \begin{vmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ . \\ . \\ . \\ . \\ d_{2m-3} \\ d_{2m-2} \end{vmatrix}$$

The linear map is directly depended on the m degree monic polynomial, $F(x)$ given by,

$$r_{j,i} = \begin{cases} f_j & \text{j = 0, 1, ... , m-1 : i = 0} \\ r_{j-1,i-1} + r_{m-1,i-1} r_{j,0} & \text{j = 0, 1, ... , m-1 : i = 1, ... , m-2} \end{cases}$$

and $r_{j-1,i-1} = 0$, if $j = 0$ [16]. The $r_{j,i}$ terms form the Reduction matrix. R.

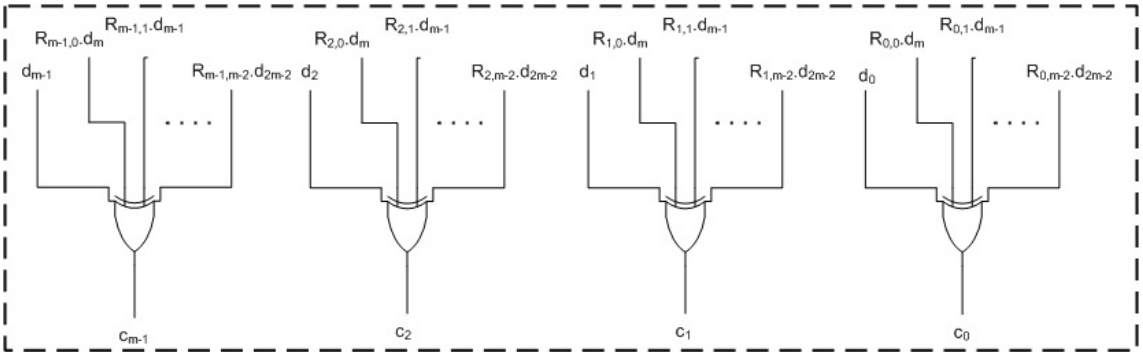


FIGURE 2. Circuit that performs reduction modulo $F(x)$ [16]

7.2.1. *Classic Multiplication.* Let *multiplication*(a, b) is a function that performs the multiplication of the coordinates of the two elements A and B and returns the $2m-2$ coordinates of D and let *reduction*(f) is the function that returns the reduction matrix.

Algorithm 2. *Classic Multiplication:*

(1) *d = multiplication(a,b)*
(2) *R = reduction(f)*
(3) *for j = 0 to m-1 do*
    $c(j) = d(j)$
(4) *end for*
(5) *for j = 0 to m-1 do*
    for i = 0 to m-2 do
    $c(j) = c(j) \oplus (R(j,i) \bullet d(m+i))$
    end for
(6) *end for*

7.2.2. *Bit-Parallel Multiplier.* If C is the product of two elements $A, B \in GF(2^m)$, then using the classic two-step polynomial multiplication, [12]

$$D = A.B = \sum_{i=0}^{m-1} b_i \alpha^i = \sum_{i=0}^{m-1} b_i.(A\alpha^i) \tag{9}$$

(α has the same meaning as stated before).

$$\begin{aligned} C = S \ mod \ F(\alpha) &= \sum_{i=0}^{m-1} b_i.((A\alpha^i) \ mod \ F(\alpha)) \\ &= \sum_{i=0}^{m-1} b_i \ . \ X^{(i)}, \end{aligned} \tag{10}$$

where,

$$X^{(i)} = \alpha.X^{(i-1)} mod \ F(\alpha), \ 1 \leq i \leq m-1, \tag{11}$$

and

$$X^{(0)} = A. \tag{12}$$

A bit-parallel multiplier will perform the the multiplication using three modules,

(1) sum module, which add two elements of $GF(2^m)$ using m XOR gates
(2) pass-through module, which multiply a $GF(2^m)$ element by a $GF(2)$ element, $X^{(i)}$ element with a b_i element.

$$b_i.X^{(i)} = \begin{cases} X^{(i)} & \text{if } b_i = 1 \\ 0 & \text{if } b_i = 0 \end{cases}$$

This is realized by using m AND gates.
(3) α module, which reduces the product of an element from $GF(2^m)$ and α, by modulo $F(\alpha)$ i.e. Equation(11).
We know that α is a root of the polynomial and $F(\alpha) = 0$. The term $A.\alpha$ can be written as,

$$A.\alpha = \left(\sum_{i=0}^{m-1} a_i \alpha^i \right) \ . \ \alpha = \sum_{i=0}^{m=0} a_i.\alpha^{i+1} = \sum_{i=1}^{m-1} a_{i-1}\alpha^i + a_{m-1}\alpha^m \tag{13}$$

With the Equation(13), we can write $A.\alpha \bmod F(\alpha)$ as,

$$X = a_{m-1}.f_0 + \sum_{i=1}^{m-1}(a_{i-1} + a_{m-1}.f_i)\alpha^i \tag{14}$$

where the coordinates of X, x_i are elements of 0,1. Since any irreducible polynomial defined over this field will have $f_0 = 1$, Equation(14) can be used to derive each coordinate of X with the polynomial basis.

$$x_i = \begin{cases} a_{i-1} + a_{m-1}.f_i & i \leq i \leq m-1 \\ a_{m-1} & \text{i} = 0 \end{cases}$$

The space requirements of this module depends on the polynomial used in the multiplier.

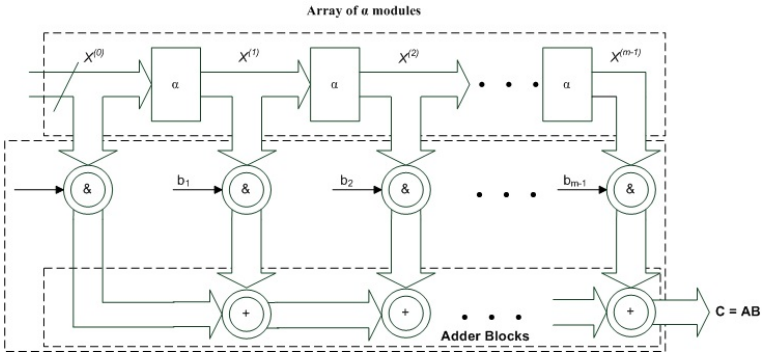


FIGURE 3. Bit-Parallel Multiplier for $GF(2^m)$ [12]

The big circles with the '&' symbols indicate the array of AND gates.

7.2.3. *Mastrovito Multiplication.* Complexity of the Hardware implementation of multiplication evidently depends on the basis representations of the field elements. The most widely used are polynomial bases. Hardware complexity is thus heavily depended on the irreducible polynomial selected. Many architectures and approaches have been proposed. In the research, Mastrovito Approach is considered as a starting point.
Continuing the description of multiplication in the form of matrix-vector operations, we can consider two approaches for computation of a field product. The previously described, Section(7.2.1), two step multiplication is one of them. The polynomial multiplication is done by any method and then the resultant higher degree polynomial is reduced by using a reduction matrix. The other approach is to combine the two steps, polynomial multiplication and modular reduction into one single step by using a *Mastrovito product matrix*. [16], [21], [22].
In the matrix-vector representation, the multiplication can be written as,

$$c_{m-1}x^{m-1} + ... + c_1x + c_0 = (a_{m-1}x^{m-1} + ... + a_1x + a_0)(b_{m-1}x^{m-1} + ... + b_1x + b_0) modf(x) \tag{15}$$

This can be written as,

$$C = ZB \tag{16}$$

where Z = h(a(x), f(x)), and, $C = \begin{vmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ . \\ . \\ . \\ c_{m-2} \\ c_{m-1} \end{vmatrix} = ZB = \left| \begin{array}{ccc|c} z_{0,0} & \cdots & z_{0,m-1} & b_0 \\ z_{1,0} & \cdots & z_{1,m-1} & b_1 \\ z_{2,0} & \cdots & z_{2,m-1} & b_2 \\ z_{3,0} & \cdots & z_{3,m-1} & b_3 \\ . & \cdots & . & . \\ . & \cdots & . & . \\ . & \cdots & . & . \\ z_{m-2,0} & \cdots & z_{m-2,m-1} & b_{m-2} \\ z_{m-1,0} & \cdots & z_{m-1,m-1} & b_{m-1} \end{array} \right|$

and the matrices can be represented as vectors,

$$C = [c_0, c_1, ..., c_{m-1}]^T \tag{17}$$

$$B = [b_0, b_1, ..., b_{m-1}]^T \tag{18}$$

The m×m matrix is the *Mastrovito Product Matrix*. The coefficients $z_{i,j}$ are dependent on a_is and $p_{i,j}$s

$$z_{i,j} = \begin{cases} a_i & \text{j = 0, i = 0, 1, ... , m-1} \\ \mu(i-j)a_{i-j} + \sum_{t=0}^{j-1} p_{j-1-t,i} a_{m-1-t} & \text{j ,i = 0, 1, ... , m-1 ; j} \neq 0 \end{cases}$$

where step-function, $\mu(u)$ is

$$\mu(u) = \begin{cases} 1 & \text{u} \geq 0 \\ 0 & \text{u} < 0 \end{cases}$$

The higher degree coordinates resulting from the polynomial multiplication can be represented as follows:

$$\begin{bmatrix} x^m \\ x^{m+1} \\ \vdots \\ x^{2m-2} \end{bmatrix} = \text{P} \begin{bmatrix} 1 \\ x \\ \vdots \\ x^{m-1} \end{bmatrix} \text{ mod f(x)} = \begin{bmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,m-1} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m-2,0} & p_{m-2,1} & \cdots & p_{m-2,m-1} \end{bmatrix} \begin{bmatrix} 1 \\ x \\ \vdots \\ x^{m-1} \end{bmatrix} \text{ mod f(x)}$$

The entries into the P matrix are binary. They depend on the coefficients of the irreducible polynomial $F(x) = x^m + f_{m-1}x^{m-1} + ... + f_1 x + f_0$ as follows,

$$p_{i,j} = \begin{cases} p_{i-1,m-1} & \text{i = 1, ... , m-2 : j = 0} \\ p_{i-1,j-1} + p_{i-1,m-1} p_0 & \text{i = 1, ... , m-2 ; j = 1, ... , m-1} \end{cases}$$

and $p_{0,j} = f_j$ and Reduction Matrix R = P^T
Thus from (16) a Mastrovito multiplier consists of and Z network which generates the Z matrix and an IP network which generates the vector - matrix multiplication, produces c_i's.

Let P_Matrix(f) be a function that returns the m-1× m, P matrix, and mastrovito_multmatrix(a,P) returns the m×m Z matrix.

Algorithm 3. *Mastrovito Multiplication*

(1) *C ← 0*
(2) *P = P_matrix(f)*
(3) *Z = mastrovito_multmatrix(a,P)*

(4) *for i = 0 to m-1 do*
for j = 0 to m-1 do
c(i) = c(i) $\oplus$ *(Z(i,j)* $\bullet$ *b(j))*
end for
(5) *end for*

Mastrovito proposed a specific procedure in selection of these polynomials. There can be a lot of possibilities in selecting the polynomials, if we were to mathematically rule out each polynomial considering all possibilities will be extremely tedious. According to Mastrovito [22], the procedure is as follows,

- Select the polynomials of minimum width and if the number of these polynomials are small, then use them to find the product matrix.
- If the number of these polynomials are big, then consider the widths for a set of polynomials and use the rule that MAX[W_1,$W_1 + 1$], where W_1, W_2 are the widths of the two polynomials under consideration and select the subset of these polynomials with minimum width. If this subset is small enough then perform the step 1.
- If an irreducible polynomial of weight k exists then, if $m = 2.3^l$, $l \in 0, 1, 2, ...$ then choose $x^m + x^{m/2} + 1$ otherwise choose $x^m + x^k + 1$ for lowest k, and such a polynomial is found to exist for all values of m in $2 \leq m \leq 1000$. Further, if $x^m + x + 1$ is an irreducible polynomial of *GF(2)* then the Mastrovito Product Matrix and the reduction matrix R, is of the form,

$$Z = \begin{bmatrix} a_0 + a_{m-1} & a_1 & a_2 & a_3 & . & . & a_{m-2} & a_{m-1} \\ a_{m-1} + a_{m-2} & a_0 + a_{m-1} & a_1 & a_1 & . & . & . & a_{m-2} \\ a_{m-2} + a_{m-3} & a_{m-1} + a_{m-2} & a_0 + a_{m-1} & a_1 & . & . & . & a_{m-3} \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ a_2 + a_1 & a_3 + a_2 & . & . & . & . & . & a_1 \\ a_1 & a_2 & . & . & . & . & . & a_0 \end{bmatrix}$$

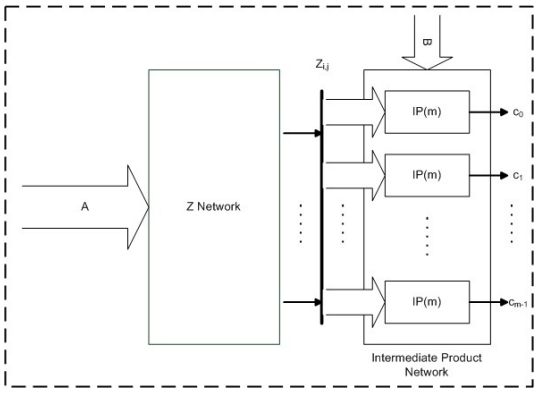


FIGURE 4. Z and intermediate product network in a mastrovito multiplier [13]

$$\mathrm{R} = \begin{bmatrix} 0 & 0 & 0 & 0 & . & . & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & . & . & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & . & . & 1 & 1 & 0 & 0 \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ 0 & 0 & 1 & 1 & . & . & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & . & . & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & . & . & 0 & 0 & 0 & 0 \end{bmatrix}$$

This suggests a maximum width for the polynomials of 2, which is ideal. it has m-1 distinct functions f_j^i of width > 1. This also requires only m-1 XOR gates to generate m^2 entries of the product matrix. These functions are the left most column and on the next-to-last row of the M matrix. In [21], they also have given the value of m which gives irreducible trinomials between 2 and 127. They are m = 2,3,4,6,7,9,15,22,28,30,46,60,63,127. They also propose that if $x^m + x^{m-1} + x^{m-2} + ... + x + 1$ is an irreducible polynomial then the product matrix has a width of 2 and the number of distinct functions f_j^i of width > 1 is $(m^2 - m)/2$. Hence $(m^2 - m)/2$ gates are required. It should be noted that the lowest maximum width is to be considered for the polynomials of the speed is the design constraint. Various new state of the art techniques are introduced like [13], [32]

Remark 2. *Delay Calculations:*
From the equation $C = ZB$, and its matrix representation, its clear that there are m^2 modulo 2 multiplications, therefore a bit-parallel multiplier, described in the Section(7.2.2), using Mastrovito's scheme requires m^2 AND gates and more than $(m^2 - 1)$ XOR gates. The number of XOR gates will depend on the polynomial for $f(x) = x^m + x + 1$, $(m^2 - 1)$ XOR gates are required [16], [28]. Similar to (8), the delay of a bit-parallel multiplier is,

$$Total\ \ Delay = \Delta_{AND} + 2\lceil log_2 m \rceil \Delta_{XOR} \tag{19}$$

7.2.4. *Low Complexity Bit Parallel Multiplier.* The scheme introduced by Mastrovito is extended by decomposing the Z matrix [16], [13]. From (16), we know that C = Z.B. In [13] Z matrix can be expressed as,

$$Z = L + P^T U \tag{20}$$

Also,

$$R = P^T \tag{21}$$

So we can express Z as,

$$Z = L + RU \tag{22}$$

The matrices, L and U are defined as follows, [13],

$$L_{m \times m} = \begin{bmatrix} a_0 & 0 & \cdots & 0 \\ a_1 & a_0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1} & a_{m-2} & \cdots & a_0 \end{bmatrix} \text{ and}$$

$$U_{m\times m} = \begin{bmatrix} 0 & a_{m-1} & a_{m-2} & \cdots & a_1 \\ 0 & 0 & a_{m-1} & \cdots & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m-2} \\ 0 & 0 & 0 & \cdots & a_{m-1} \end{bmatrix}$$

$$D = LB \tag{23}$$

$$E = UB \tag{24}$$

$$\begin{aligned} C &= D + P^T E \\ &= D + RE \end{aligned} \tag{25}$$

From the definition of L and U, the coefficients of D and E can be calculated as follows,

$$\begin{aligned} d_i &= \sum_{k=0}^{i} a_k b_{i-k}; i = 0, 1, ..., m-1 \\ e_i &= \sum_{k=i}^{m-2} a_{m-1-(k-i)} b_{k+1}; i = 1, ..., m-2 \end{aligned} \tag{26}$$

Algorithm 4. *Mastrovito Multiplication with decomposed Z matrix:*
Let D_calc(a,b) and E_calc(a,b) are two functions which take A and B as input and return D and E and reduction_matrix(f) be a function that return the Reduction Matrix R.
Let IR be the intermediate result.

(1) *C ← 0*
(2) *IR ← 0*
(3) *R = reduction_matrix(f)*
(4) *for i = 0 to m-1 do*
    for j = 0 to m-2 do
    IR(i) = IR(i) ⊕ (R(i,j) • E(j))
    end for
(5) *end for*
(6) *for i = 0 to m-1 do*
    C(i) = D(i) ⊕ IR(i)
(7) *end for*

The Structure of the circuits that implement these modules, i.e. Vector E Generator, Vector D Generator and Reduction Circuit in the table of figures(1).

7.3. **Inversion.** Multiplicative Inverse of $a(x)$ in the Galois Field $GF(2^m)$ is denoted by $a^{-1}(x)$ and is defined as an element which satisfies the following equation.

$$a^{-1}(x).a(x) = 1 \tag{27}$$

and the operation performed is multiplication in $GF(2^m)$. Most popular method to find multiplicative inverse is by using *Fermat's Theorem*.

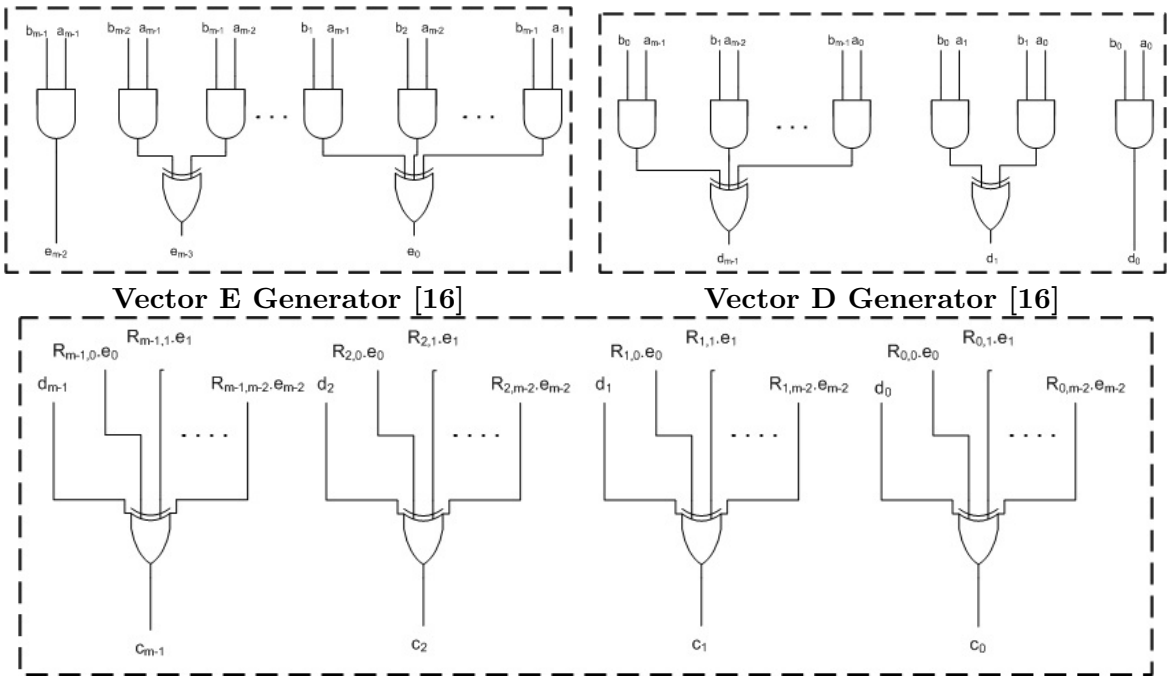


TABLE 1. Structure of Vector E and D Generator and the Reduction Circuit

7.3.1. *Calculating inverse using Fermat's theorem.* Using Definition: 13 and Definition: 3, we know that,

$$\begin{aligned} a^{q-1} &= 1 \\ a^{q-1} &= a.a^{-1} \\ therefore \quad a^{-1} &= a^{q-2} \end{aligned} \tag{28}$$

Hence inversion can be viewed as simply an extension of multiplication and squaring operations. Normal basis is a better choice for squaring of the elements, [27] since squaring is performed in normal basis by a cyclic - shift, and in polynomial bases squaring is performed by bit-extension through insertion of 0 between the consecutive bits followed by modular reductions to reduce the extended polynomial of degree 2(m -2) to degree m-1. Hence various algorithms exist by using Normal Basis. [15], [7], [1].

7.4. **Squaring.** Squaring operation is usually implemented by using multiplication algorithms. Squaring takes one operand $a(x)$ and performs,

$$c(x) = a(x).a(x) \ \ mod \ \ f(x) \tag{29}$$

Classic two-step squaring derived directly from Algorithm(2) is given below,

Algorithm 5. *Let multiplication(a, b) is a function that performs the multiplication of the coordinates of the two elements A and B and returns the $2m-2$ coordinates of D and let reduction(f) is the function that returns the reduction matrix.*

(1) *d = multiplication(a,a)*

(2) *R = reduction(f)*
(3) *for j = 0 to m-1 do*
    $c(j) = d(j)$
(4) *end for*
(5) *for j = 0 to m-1 do*
    for i = 0 to m-2 do
    $c(j) = c(j) \oplus (R(j,i) \bullet d(m+i))$
    end for
(6) *end for*

The idea is that the multiplier module can be used to perform the squaring as well, irrespective of how the multiplier module is implemented.

8. Fault Tolerant Systems

Reliability of a system is a major concern as system complexity increases. Data integrity in the case of massively taxed client-server model of computing is crucial for its functioning. Robustness of the hardware is another factor which cries for attention, as there is a saturation reached in achieving it through just design techniques. All this has sort of obviated the need for fault tolerance. [29, p. xiii] As technology permeates into more and more crucial applications the reliability, availability and robustness of these systems is crucial in more ways than those easily imagined.

. Lack of fault tolerance does not just compromise the reliability of the system, but also its computational abilities. Operational time lost due to faults could result in regrettable computational loss in complex systems. At an age when the systems are becoming increasingly distributed and parallel, dependability is also compromised in the absence of some sort of fault tolerant mechanism.

High performance systems with high circuit density and high operating frequencies with low voltage levels and very small voltage margins, are highly at risk of faults. Feature sizes have come down to incredible levels make the logic as well as memory error prone, [25].

. A *fault tolerant system* is one that can perform its correct functionality in the presence of hardware and, or software faults. *Fault tolerance* is the quality of a system to be able to function in the presence of faults.*Fault detection* is the process of sighting them and *Fault correction* is the process of rectifying them [29]. A number of schemes have been developed for correcting and detecting these faults right from the design stage to the actual implementation of many systems. The reasons for investing so much on achieving fault tolerance is reiterated below.

- Correct performance in the presence of faults.
- Fault based attacks can be avoided: In order to reveal information in an encrypted message certain faults are injected by attackers. In [2],it is explained that the private key of an encrypted message passed from the source can be obtained within approximately 100 hours. The fault attack didn't even require access to the system but just proximity to it. [6] Hence,the importance of avoiding such attacks in case of strategic systems is pivotal.

The GF Arithmetic Processors are used in various fields like cryptography, VLSI Testing and Digital Signal Processing, and their hardware implementation takes millions of logic gates and a fault in any one of them can compromise the use of these systems. Making the basic structure fault tolerant will increase the efficiency of the system.

8.1. **Types of Faults.** Faults can be broadly divided as gate-level faults and architectural-level faults. Open, short and stuck at faults are the basic gate level faults. The output will be erroneous depending upon whether the fault is manifested on to the output. The faults can be modeled at an architectural level.

8.2. **Methods to detect faults.** Concurrent Error Detection is an efficient way of detecting errors. The system is under test constantly. Theres no special test mode.

. Redundancy techniques are used to detect faults. There are three types of redundancies.,

- Hardware Redundancy: This is a technique in which hardware is replicated. The input for the system is given to both the hardware units and the computation is done by both the systems and the results are compared. The output is valid only if both the units gave the same output. This technique has an area over head but cost of hardware is decreasing. So in case of small systems we can efficiently use hardware redundancy techniques. There are three kinds of hardware redundancy techniques,
  - Passive Redundancy (Triple Modular Redundancy Systems)
  - Active Redundancy
  - Hybrid Redundancy
- Time Redundancy: This is a technique in which the same module performs the same operation twice and compares the results. Such a system has 100% time redundancy.
- Information Redundancy: This is a technique in which a redundant information is added to the data in order to allow fault detection [29, p. 21]. Error Correction Codes is an example for information redundancy. There are various error correction/detection codes. One of the most significant and commonly used, and very thoroughly invested in this proposed project is *Parity Codes*

8.3. **Triple Modular Redundancy.** Developing systems with hardware redundancy for fault tolerance is not a new technique. The trend dates back to the days of STAR (Self Testing And Repairing) computers in Jet Propulsion Laboratories. These systems consists of the combination of a multiplexed system with a majority voting system and standby spare units. A simple Triple Modular Redundancy system is shown below.
The reliability of such a system can be approximated as,

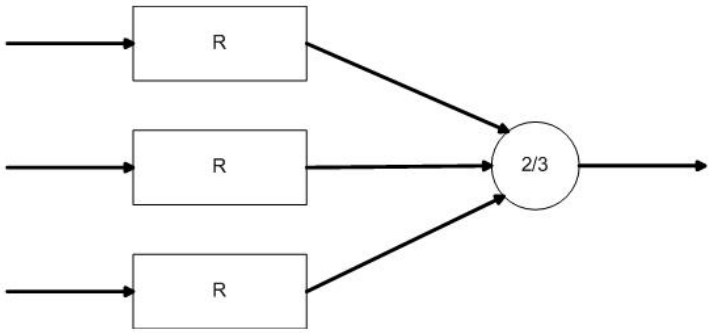


FIGURE 5. TMR System with individual modules of Reliability R [29]

$$R_{TMR} = R^3 + 3R^2(1 - R)$$

where R is the reliability of a single module.
The following figure shows the Reliability of an NMR system vs. normalized time. The

family of curve follows the failure law $e^{-\lambda T}$, where λ is the failure rate of the non-redundant system when it is active [29] [23].

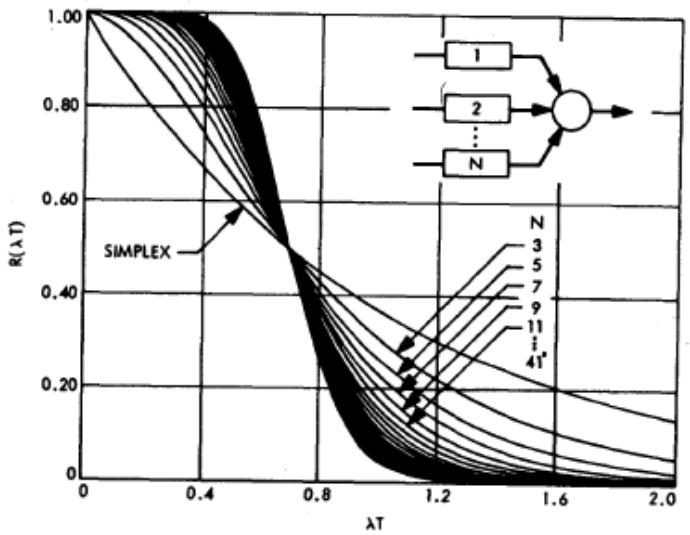


FIGURE 6. Reliability of NMR type systems vs normalized times [23]

8.4. **Parity Codes.** This is a method employing information redundancy.

Theorem 17. *Let* $A = (a_{m-1},a_{m-2},....,a_1,a_0)$ *be an* m bit *message. The Single bit parity code A is* $A_C = (a_c,a_{m-1},a_{m-2},....,a_1,a_0)$*, where* a_c *can be obtained as follows, [31]*

$$a_c = \sum_{i=0}^{n-1} a_i mod 2 \tag{30}$$

$$a_c = (1 + (\sum_{i=1}^{n-1} a_i))mod 2 \tag{31}$$

The equation (30) calculates Even parity: The value of a_c will be 0 if the sum of all the digits in the m bit message is even and 1 otherwise. The equation (31) finds Odd Parity: The value of a_c will be 0 id the sum of all digits in the m bit message is odd and 1 other wise.

This type of single parity code can detect single bit errors and they can detect only odd number of erroneous bits.

There are several implementations for finding out parity and even better ways of predicting them from the original data before the calculations are done. Thus making testing the results easier. This is an idea which will be heavily researched during the project. The application of various parity prediction algorithms and the comparisons with respect to power, area and time of the GF processors. A proper understanding of this idea is vital.

Compared to other fault tolerant implementations of ALUs, the parity prediction scheme has the advantage that it requires the minimum hardware overhead for the adder/ALU and the minimum hardware overhead for the other data-path blocks. [24] This is a huge advantage because fault tolerant is achieved by introducing some more design into the original block, which will increase the area requirement, power requirement/usage, and might also effect the longest delay path to the correct output. Thus, finding an optimum scheme is very important.

8.5. **Error Correction.** Error Correcting Codes find their application in achieving fault tolerance in Memories, Disks and Communication systems. Traditional techniques like Triple Modular Redundancy (8.3), and duplex (duplicating hardware) are not useful at all for correcting errors in memories and data communication systems. The bandwidth required will be more than three times the usual and it will slow down the data transfer by two times.

8.5.1. *Methods for Error Correction.* In-order to ensure that the data obtained as an output of a system is correct, certain number of bits are inserted to the input data and they are verified at the end and a mismatch will indicate an error. i.e, to a data of $'k'$ bits, $'r'$ bits are added and the data along with the extra bits added forms the *codeword*. This codeword is verified in the end after separating the data and the added bits for their match, and a mismatch will indicate an error.
A pair of words in a set of codewords should be separated by a minimum *hamming distance*. Hamming distance is the difference between the corresponding bits between a pair of words. A distance of 1 is enough to detect a single error. For example, with a single error, the data, say 101 can go to either 001,111 or 100. These bits vary by one bit and hence the hamming distance is 1 and the minimum required for single error detection is evidently, *one*. A simple parity code can successfully detect odd number of single bit errors in data. The table(2)is the codewords for 4-bit data with a single bit parity.

The single parity-check code $(5, 4)$, an extra parity bit is added in the end, this make sure that the number of one's in the codeword is always even. This parity check code is a single error detecting code with $k = 4$ and $r = 1$ and minimum hamming distance $d_{min} = 2$. As stated before, this code can detect odd number of single bit errors.
Error Correction requires the vectors with single errors to not overlap. To correct a single bit error in a k bit data word, we require at-least $2 * 1 + 1$ extra bits. [18]. A general model for this kind of error correction is quite established. In the case of k data bits and r extra bits, there can be $k + r$ erroneous states and 1 state with no error at all. So the possible combination of r bits should be able to accommodate all these cases. i.e,

$$2^r \geq k + r + 1 \tag{32}$$

The table(3) shows how to assign parity in a 4 bit data-word with 3 parity bits, as $2^3 = 8 \geq 4 + 3 + 1$, in bit positions (a_3 a_2 a_1 a_0 p_2 p_1 p_0).

Data-word	Codeword
0000	00000
0001	00011
0010	00101
0011	00110
0100	01001
0101	01010
0110	01100
0111	01111
1000	10001
1001	10010
1010	10100
1011	10111
1100	11000
1101	11011
1110	11101
1111	11110

TABLE 2. Data-words and the corresponding Error Correction Code words

State	Erroneous parity check(s)	Syndrome
No errors	None	000
Bit 0 p_0 error	p_0	001
Bit 1 p_1 error	p_1	010
Bit 2 p_2 error	p_2	100
Bit 3 a_0 error	p_0, p_1	011
Bit 4 a_1 error	p_0, p_2	101
Bit 5 a_2 error	p_1, p_2	110
Bit 6 a_3 error	p_0, p_1, p_2	111

TABLE 3. Assignment of parity bits

This code is called $(7, 4)$ code. If p_0 fails when bit 3, 4 or 6 fails. i.e.
p_0 indicates error in bits 0,3,4,6 $\rightarrow p_0 = a_0 \oplus a_1 \oplus a_3$,
p_1 indicates error in bits 1,3,5,6 $\rightarrow p_1 = a_0 \oplus a_2 \oplus a_3$,
p_2 indicates error in bits 2,4,5,6 $\rightarrow p_2 = a_1 \oplus a_2 \oplus a_3$.
This gives rise to the Hamming matrix

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

For example, if $a_3a_2a_1a_0 = 1010$ and $p_2p_1p_0 = 010$ and if there is an error in the a_2 bit, i.e., $a_3a_2a_1a_0p_2p_1p_0 = 1110010$, we first recalculate the parity bits, $p_2p_1p_0 = 100$, the difference between the expected and the original parity bits is, 110, this is called the *syndrome* and it indicates the error in the data-word is at the a_2 bit, and the corrected data-word is 1010. This method will correct *single* bit errors only. If two bits are

wrong, say a_2 and a_1, then $a_3a_2a_1a_0p_2p_1p_0 = 1100010$ then the calculated syndrome will be $p_2p_1p_0 = 100$ and the error in the bit a_1 will go undetected. In-order to detect double errors in data bit, usual method is to add another parity bit. Thus for $k = 4$, for Double Error Detection, $r = 4$. Thus a $(8, 4)$ code is used for double error detection/single error correction.

9. Summary of Previous Work done in Fault Tolerant Architectures for Galois Field Multipliers

9.1. **Fault Detection.** Many architectures exist for detecting errors in the output generated by the Field Multipliers. Parity Prediction is an efficient method for detecting single and certain multiple stuck-at faults. Schemes which can be reconfigured with respect to the polynomial is preferred. [12]

9.2. **Parity Prediction.** Self Checking Arithmetic Logic Units(ALUs) used arithmetic codes as a concurrent error detection mechanism. This type of information redundancy was preferred given the fact that these codes can be found using arithmetic operations and they can be thought of as a type of arithmetic operation. Also they are not different for different operations. These codes will be the same for the result of an operation if the same code was applied to the operands as well. For example of the encoding function is *F()* and if a and b are the operands and if '*' is the operation, the arithmetic code will be invariant

$$F(a * b) = F(a) * F(b)$$

These kind of codes were used in JPL STAR Computers [3].

These codes were completely useless in case of single faults that occur in carry look-ahead schemes. They were also not efficient in case of Logical Operations because extra hardware is required for code prediction. [24]

This is were Parity Prediction Self - Checking Adders find their effectiveness. They have very less hardware overhead. They are compatible with various memory schemes including those based on Hamming SEC/DED (Single Error Correction/Double Error Detection). This scheme efficiently detects error in the output of the Unit. But there can be single faults which can produce error in the carry bit which will be propagated to other units in the block, thus is not completely *fault secure*. Total Fault Secure systems have circuits which detect all errors produced by any modeled fault.

9.3. **Strategy for Fault Detection using Parity Prediction.** Parity prediction is used in most of the systems in order to detect single and certain multiple stuck-at faults. The basic structure of the parity prediction scheme is given below.

The parity generation block predicts the actual parity of the output. Then the actual parity of the output generated by the Unit Under Test is compared with the predicted parity. Then the compared result is monitored to check whether the output is correct or not. The parity prediction method predicts the parity of the outputs using the inputs alone. It is assumed that the parity prediction block is fault free or can be easily made fault free.

Considering the particular case of Field Multipliers, the diagram can be analyzed and the variables can be described as follows:

$A, B \in GF(2^m)$ are the two inputs to the multiplier and $Y \in GF(2^m)$ is the output. The

PP, parity prediction block predicts the parity of the inputs and returns $\hat{PY}$and the PG, parity generation block generates the parity of the output obtained from the multiplier and returns PY. In the Binary tree XOR final comparison block, the predicted parity and the generated parity is compared and returns $\hat{e}_{CUT}$. If $\hat{e}_{CUT} = 1$ then there is an error in the obtained result, and if $\hat{e}_{CUT} = 0$ then there is no error in the output.
The parity prediction function is denoted as $\hat{PY} = \Gamma_{CUT}(A, B)$, indicating its completely depended on the primary inputs to the multiplier. Since the analysis is simpler, a case with a single fault case is considered [17], [12].
As described in the Section(7.2.2), the bit parallel multiplier has three modules and the Parity Prediction can be done on individual modules.

9.3.1. *Parity Prediction on individual modules.*

(1) Parity Prediction in α module In [12], the parity prediction function for the α module is written as,

$$\hat{p}_X = \Gamma_\alpha = p_A + a_{m-1} \tag{33}$$

and $p_A \in GF(2)$ where a_{m-1} is the $(m-1)^{th}$ coordinate of the input A. This is proved by the authors of [12], [5]. In the same way, we can see that, $p_X + p_A = a_{m-1}$.

(2) Parity Prediction in Sum and Pass-through Modules The sum module is just a binary tree of XOR gates which performs, the addition of two field elements. The parity prediction function for this module is simply,

$$\hat{p}_S = \Gamma_{sum} = p_A + p_B \tag{34}$$

where p_A and p_B are the parity of the inputs A and B respectively. Also, $p_A, p_B \in 0, 1$. This will add one extra XOR gate in the module, if the parity bits are assumed to be provided as primary inputs.
Since the output of the pass-through module is depended on the $b_i \in GF(2) = 0, 1$, the output of this module is either 0 when $b_i = 0$, or A(the other input to

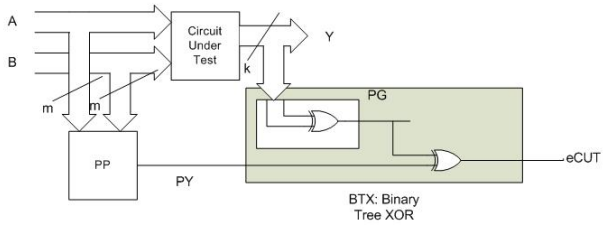


FIGURE 7. Error Detection in the circuit under test using parity prediction [12]

the pass-through module), when $b_i = 1$. The parity of the output of this module $b.A$, can be thus predicted by using the function,

$$\hat{p}_P = \Gamma_{pass} = b.p_A \tag{35}$$

This requires an extra AND gate to implement.

9.3.2. *Architecture.* In a bit-parallel multiplier, the modules are cascaded in-order to obtain parallelization. Hence, the parity bits out of each individual α, sum and pass-through modules will modify the structure of the multiplier and the following structure is obtained. p_X is the parity of the output obtained, by XOR-ing all the $m-1$ bits.

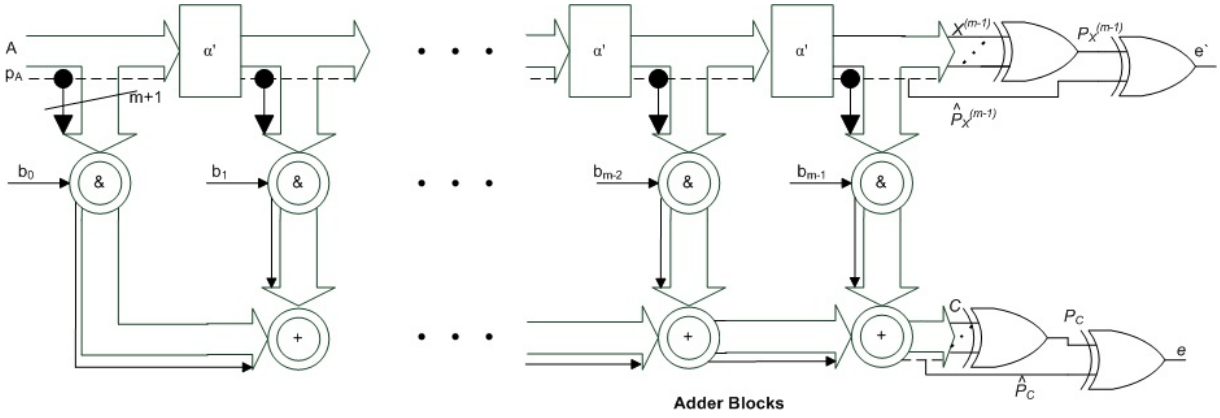


FIGURE 8. Bit Parallel Multiplier with Fault Detection [12]

(Note that the circles with '&' are the array of AND Gates/the bit product blocks) Here, the parity prediction function of all the cascaded α modules will take the form, [12]

$$\hat{p}_{X^{(j)}} = p_A + \sum_{k=0}^{j-1} x_{m-1}^{(k)}, \tag{36}$$

where j = 1, 2, ... , m-1 and the $x_{m-1}^{(k)}$ is the $(m-1)^{th}$ coordinate of $X^{(j)} = A\alpha^j \ mod\ F(\alpha)$ with respect to the basis. In [12], the parity prediction function for the final product C is obtained as,

$$\hat{p_C} = \sum_{j=0}^{m-1} b_j \hat{p}_{x^j} \tag{37}$$

This predicted parity of the final output C is compared with the generated parity $p_C = \sum_{i=0}^{m-1} c_i$ can generate an error flag, e which indicates the presence or absence of a fault in the output in C and the error flag e' indicates error in the α' modules. This scheme can also detect few multiple faults as well. The only disadvantage of this scheme is that if there is an error in the right-most of the α' blocks, multiple bits in the output will get affected and they may go unnoticed, as a single error caused even number of faults in the output bits. Comparison of the number of AND, XOR gates and the delay and area overheads are given in the table(4)

Fault detection Multiplier	Complexities = original structure + [overhead]		Overhead
Traditional Bit - Parallel	#AND	$m^2 + m$	0.6%
	#XOR	$(m + \omega - 2)(m - 1) + m - 2$	2.4%
	Delay	$\Delta_{AND} + m\Delta_{XOR} + \lceil log_2(m + 1) \rceil \Delta_{XOR}$	4.9%
Low Complexity Bit - Parallel	#AND	$m^2 + 1$	0.004%
	#XOR	$(m + \omega - 2)(m - 1) + 3m$	1.8%
	Delay	$\Delta_{LCBP} + \lceil log_2(m) \rceil \Delta_{XOR}$	69.2%

TABLE 4. Overhead Costs of Fault Detection Structures for Multiplication Using Polynomial Bases [12]

Variables have the same meaning as before. ω is the Hamming weight of the irreducible polynomial, and Δ is the Delay. Both the schemes get a coverage of 100 percent.

9.4. **Fault Correction Schemes.** Many techniques have been proposed for Error Correction in Galois Field Multipliers, like [8]. Parity Prediction techniques are used for detecting errors [12]. Another way is to multiply/scale the inputs to the multiplier by a factor and then the result is verified by divisions [10]. One of particular interest in this research is, Error Correction using Low Density Parity Check Codes. This method is proven to be very efficient in terms of area overhead, only slightly over 100% [19].
Large degree of parallelism and high performance makes LDPC a very efficient method of error correction. [11]. It is preferred over traditional Hamming code based techniques for error correction because of its decreased decoding complexity. In this method [19], a multiple parity prediction circuit is used to predict the parity of the output with respect to the inputs. The complexity of this circuit is dependent only on the number of bits in the input and the number of parity bits to be predicted by the rule, Equation(32) in Section(8.5). An example of a multiplier over the field $GF(2^4)$, is considered and the method is applied.

Let $\overrightarrow{z}' = [z'_0, z'_1, ..., z'_{m-1}]^T$ is the output of the multiplier and $\overrightarrow{z} = [z_0, z_1, ..., z_{m-1}]^T$ be the output after error correction. Let r be the number of parity bits following the Equation(32), and the predicted parity bits are $\overrightarrow{p} = [p_0, p_1, ..., p_{r-1}]^T$ and the parity generated from the output of the multiplier, the original parity is $\overrightarrow{p}' = [p'_0, p'_1, ..., p'_{r-1}]^T$. H matrix is constructed as given in the Section(8.5). It should be noted that the number of 1's in the rows are as small as possible in-order to reduce the logic complexity. Here, there will be m non-zero r-bit column vectors with two 1's and r column vectors with one 1. The dimension of the matrix is $r \times (m + r)$.

$$H = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & p_0 & p_1 & p_2 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Thus,
$p_0 = c_0 \oplus c_2 \oplus c_3$
$p_1 = c_0 \oplus c_1 \oplus c_3$
$p_2 = c_1 \oplus c_2 \oplus c_3$.

No. of Outputs	'r' LDPC	Delay
2-3	3	$4\Delta_{XOR} + \Delta_{AND}$
4	4	$4\Delta_{XOR} + \Delta_{AND}$
5-6	4	$4\Delta_{XOR} + \Delta_{AND}$
7-10	5	$5\Delta_{XOR} + \Delta_{AND}$
11	6	$5\Delta_{XOR} + \Delta_{AND}$
12-15	6	$5\Delta_{XOR} + \Delta_{AND}$
16-21	7	$5\Delta_{XOR} + \Delta_{AND}$
22-26	8	$5\Delta_{XOR} + \Delta_{AND}$

TABLE 5. Delay Overhead in Error Correction For Various Multiplier sizes [19]

These predicted parity bits are compared with the original parity bits obtained from the output of the multiplier and then decoded according to the syndrome. The implementation details of the output bits depends on the multiplier under consideration. After the output from the multiplier is obtained, the parity is calculated and compared with that of the predicted parity. Then the result is decoded and the correction for the bits of the output, if any, is given out and the correct output is obtained after XORing the obtained correction with the generated output. Delay Overhead in Error Correction For Various Multiplier sizes is given in the table(5).

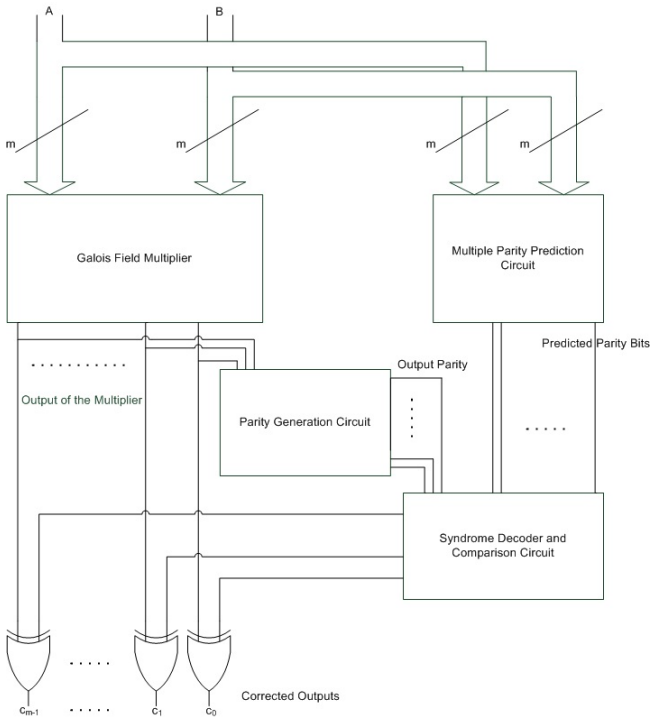


FIGURE 9. Fault Correction Architecture using LDPC Codes [19]

In the further sections, the abbreviations have the following meanings:

- DSM: Digit Serial Multiplier
- MPPC: Multiple Parity Prediction Circuit
- PG: Parity Generation
- SDOG: Syndrome Decoder and Output Generator

10. Description of the Work Carried out

10.1. **Galois Field Multiplier: Optimum Design. Digit Serial Multipliers** As seen in the previous sections, there are countless number of implementations of Finite Field Multipliers. Trade off between speed and area, and better efficiency, performance, all these factors count into the design. The designs discussed so far are either bit-serial or bit-parallel algorithms. These algorithms are the define the limits to which the envelope of the multipliers can be pushed. The fine tuning was limited to finding better types of polynomials and the complex mathematics involved in the interdependencies. Digit Serial multipliers offer an optimum alternative. [34] Most ECC implementations over characteristics 2 fields use digit multipliers with digit sizes of powers of 2, [26].
A Single Accumulator Multiplier is designed. The number of accumulators in the design can be increased achieving a higher operating frequencies. Consider Field Multiplication of two elements in the field $GF(2^m)$, $A = \sum_{i=0}^{m-1} a_i \alpha^i$ and B gives $C = A.B \bmod f(\alpha)$. F(α) is the irreducible polynomial which defines the Field. In a bit-serial multiplier, a single coefficient of B is multiplied with every coefficient of A. This is followed by reducing A with respect to the polynomial. Then the next coefficient of B is multiplied with every coefficient of A and so on. In bit-parallel multiplication, every coefficient of B is multiplied with every coefficient of A, followed by a reduction step. The former is slower but takes up less hardware real-estate where as the latter is faster but bigger. In digit serial multiplication, a certain number of coefficients of B is multiplied with A and then A is realigned with respect to the polynomial and then the next set of coefficients of B arrives to be multiplied with A. This process continues till all the coefficients of B are multiplied with those of A. The number of co-efficients of B processed at a time is called digit-size D. The number of digits in the polynomial of degree $m-1$ is given by, $d = \lceil m/D \rceil$. Then,

$$B = \sum_{i=0}^{d-1} B_i \alpha^{Di}, \tag{38}$$

where,

$$B_i = \sum_{j=0}^{D-1} b_{Di+j} \alpha^j; 0 \leq i \leq d-1 \tag{39}$$

It should be made sure that B is padded with zero coefficients for the Most Significant digits after being rotated out. The Algorithm for digit serial multiplication is given below. [34], [36]

Algorithm 6. *Least Significant Digit, Digit-Serial Multiplier*

(1) $C \leftarrow 0$
(2) *for* $i = 0$ *to* $\lceil m/D \rceil - 1$ *do*
    $C \leftarrow B_i A + C$
    $A \leftarrow A\alpha^D mod\ f(\alpha)$
(3) *end for*
(4) *Return* $(C) mod\ f(\alpha)$

10.1.1. *Choosing a Polynomial.* In the step $A \leftarrow A\alpha^D \bmod f(\alpha)$, the terms of the form, $A\alpha^D$ are to be reduced by using the irreducible polynomial. Methods are defined for doing that by general polynomials. Although better results can be obtained by using certain polynomials that minimize the complexity of the reduction operation. Certain theorems exist which helps in finding them [36]. In [34], their application is shown and its proven that for a given polynomial of the form, $f(\alpha) = \alpha^m + f_k\alpha^k + \sum_{j=0}^{k-1} p_j\alpha^j$, the value of D is chosen with respect to the value of k, i.e. the degree of the second highest coefficient in the irreducible polynomial.

10.1.2. *Details of the Architecture.* The single accumulator multiplier described in the algorithm(6), can be modeled as given below.

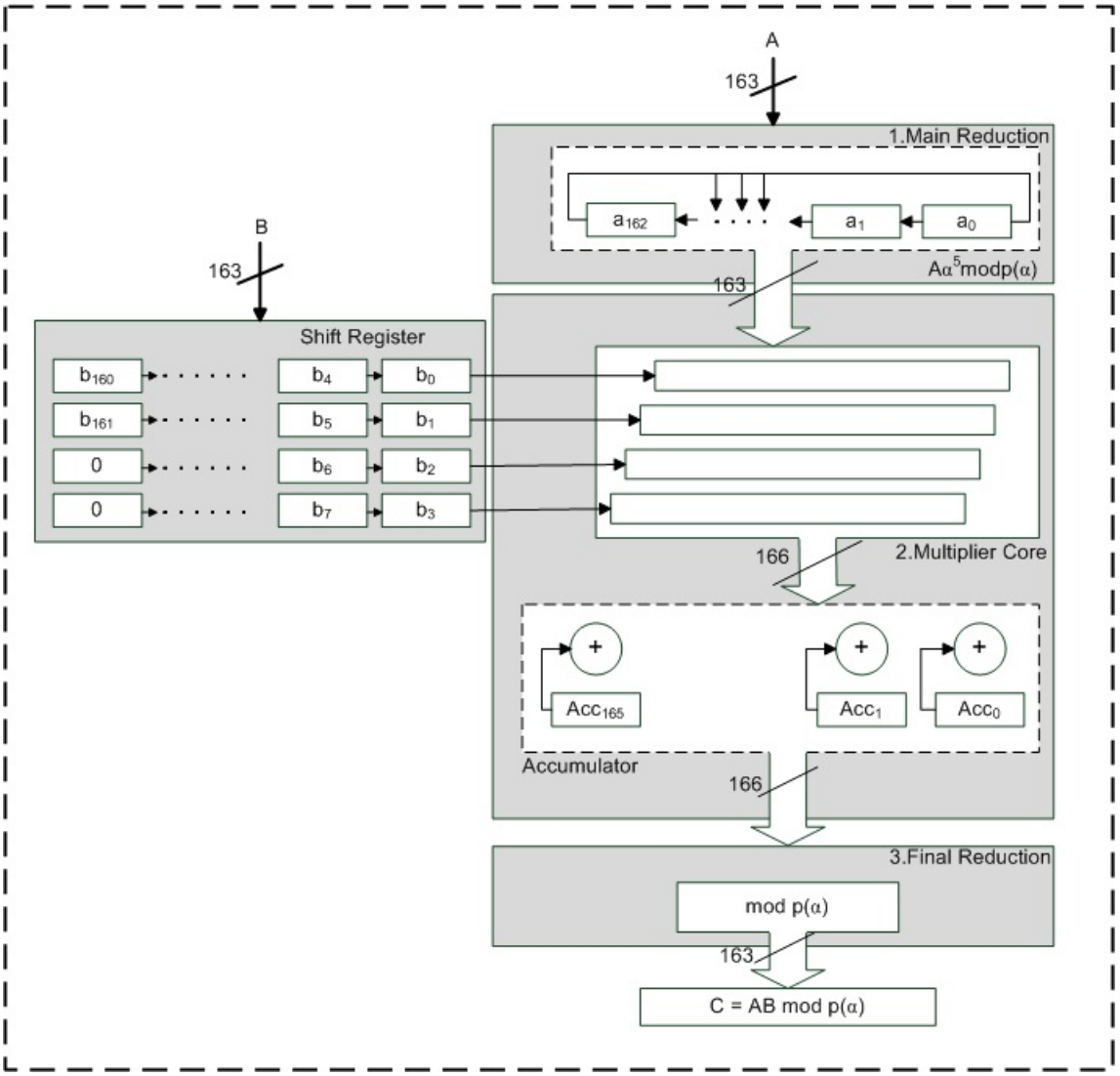


FIGURE 10. Digit Serial Multiplier for $D = 4$ over $GF(2^m)$ [34]

This contains three parts:

- Reduction Circuit
  This circuit performs the operation, $A \leftarrow A\alpha^D \text{mod } f(\alpha)$. This is done by shifting the operand A by D places and then reducing it by mod $p(\alpha)$. The polynomial is selected according to the Section(10.1.1). The critical path delay of the circuit will be more equal or less than equal to the multiplier core. The main reduction circuit requires $k+1$ AND gates and k XOR gates for one reduction, where k is the degree of the second highest coefficient in the polynomial. A total of $(k+1)D$ AND and kD XOR gates for the entire operation. A m number of Flip flops are also required for storing the multiplicand. The critical path is given by $\Delta_{AND} + \lceil log_2(D+1) \rceil \Delta_{XOR}$.
- Multiplication Core
  This circuit computes the intermediate C, i.e. $C \leftarrow B_i A + C$ and stores it in an accumulator. Depending upon the implementation, there can be a single or more accumulators. All the components run in parallel and they require one clock cycle each to perform. Multiplier critical path depends upon the multiplier core. An example for the multiplier core with $m = 11, D = 5$ is given below [34].

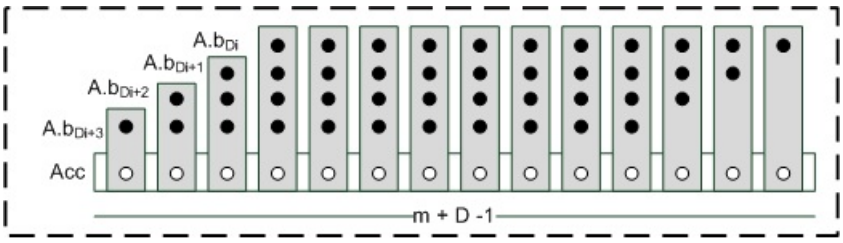


FIGURE 11. Single Accumulator Digit Serial Multiplier Core [34]

  It performs AND-ing of the multiplicand with each element of the digit of the multiplier and storing the result in an *Acc*, the number of elements in *Acc* is $m+D=1$ and requires the same number of Flip Flops to implement them. The multiplier core has a critical delay of one AND gate, Δ_{AND}, because all the AND operations are done in parallel, and delay of XOR-ing $D+1$ elements. Therefore the total critical path delay is given by $\Delta_{AND} + \lceil log_2(D+1) \rceil \Delta_{XOR}$.
- Final Reduction Circuit
  The contents of the accumulator is reduced by mod $p(\alpha)$ and the final result is obtained. i.e. $(Acc) modf(\alpha)$, This is done the same way as the main reduction without the shifting of terms. $(k+1)(D-1)$ AND gates and $(k+1)(D-1)$ XOR gates are required for the operation. The delay is $\Delta_{AND} + \lceil log_2(D) \rceil \Delta_{XOR}$.

The Polynomial can be hard coded, rather than being stored in a set of flip-flops. The table(6), shows the number of AND and XOR gates required for a Digit Serial Multiplier with a polynomial of degree m and digit-size D.

Multiplier	Number of XOR gates	Number of AND gates
SAM (D $\geq$ 2)	$(m+k)D+(k+1)(D-1)$	$(m+k+1)D+(k+1)(D-1)$

TABLE 6. Comparison of number of XOR and AND gates in the Multiplier

10.1.3. *Implementation Procedure.*

- select the value of m
- select the polynomial
- design the multiplier according to the Algorithm(6)

Case 1: $m = 4$**,** $D = 2$**,** $F(x) = x^4 + x + 1$ The field defining polynomial,

$$F(x) = x^4 + x + 1 \tag{40}$$

As opposed to the design in the Section(10.1.2), few changes are made in the design and the changed architecture w.r.t the given case is given below.
In the design of Digit-Serial Multiplier, as opposed to the array of shift registers used to obtain digits of B in [34], a single shift register is used and it is rotated by the number of digits to the right. It is also made sure that the MSBs are filled with zeros after the bits are shifted.

- Main Reduction Circuit
  The multiplicand A is shifted D times to the left here. i.e.

...	...	a_3x^3	a_2x^2	a_1x^1	a_0x^0	$\Longrightarrow$	a_3x^5	a_2x^4	a_1x^3	a_0x^2	...	...

  We know that, if $F(x) = x^m + \sum_{i=0}^{m-1} f_i x^i$ is the polynomial and if x is a root of this polynomial, then $F(x) = 0$, implies,

$$x^m = \sum_{i=0}^{m-1} f_i x^i = 0 \tag{41}$$

  From Equation(41), for the Field Defining irreducible monic polynomial given in the Equation(40), we can derive,

$$x^4 = x + 1$$
$$x^5 = x^2 + x$$

  This means,

a_3x^5	a_2x^4	a_1x^3	a_0x^2	$\Longrightarrow$	$a_3(x^2+x)$	$a_2(x+1)$	a_1x^3	a_0x^2

  Finally,

a_1x^3	$(a_3+a_0)x^2$	$(a_2+a_3)x^1$	a_2x^0

  The same process is continued for $\lceil m/D \rceil$ iterations. i.e. 2 in this case.
- Multiplier Core
  At first the all the digits in the multiplier is multiplied with that of the multiplicand. i.e. b_0 and b_1 with a_3, a_2, a_1, a_0. And its accumulated in $Acc[m + D - 1]$.Multiplier core produces the following result after the first iteration.

...	$a_3.b_0$	$a_2.b_0$	$a_1.b_0$	$a_0.b_0$
$a_3.b_1$	$a_2.b_1$	$a_1.b_1$	$a_0.b_1$	...

The Accumulator will be filled as follows,

Acc[4] $\Longrightarrow$					
	$a_3.b_1$	$a_3.b_0 + a_2.b_1$	$a_2.b_0 + a_1.b_1$	$a_1.b_0 + a_0.b_1$	$a_0.b_0$

After the first iteration, the main reduction block will shift and reduce the multiplicand according to the previous section. After the second iteration, the contents of the Accumulator are,

$$Acc[4] = a_2.b_2 + a_0.b_0$$
$$Acc[3] = a_2.b_2 + a_3.b_2 + a_3.b_3 + a_1.b_0 + a_0.b_1$$
$$Acc[2] = a_3.b_2 + a_0.b_2 + a_2.b_3 + a_2.b_0 + a_1 b_1$$
$$Acc[1] = a_1.b_2 + a_3.b_3 + a_0.b_3 + a_3.b_0 + a_2.b_1$$
$$Acc[0] = a_1.b_3 + a_3.b_1$$

- Final Reduction The final reduction circuit performs the reduction of the Accumulator by mod $f(\alpha)$. This is done the same way as the main reduction but without the shift.

$$c[3] = Acc[0] \oplus Acc[4]$$
$$c[2] = Acc[1] \oplus Acc[4]$$
$$c[1] = Acc[2]$$
$$c[0] = Acc[3]$$

These results are cross verified with that obtained from the multiplier implemented using the Algorithm(4).
The area and delay calculations are presented in the Results.
The design can be found in the next page.
Case 2: $m = 163$, $D = 2$, $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$ The same procedure followed in the previous case is continued to extend the multiplier for a Field $GF(2^{163})$. The value of $m = 163$ and $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$ are selected because they are NIST(National Institute of Standards and Technology) recommended. These values are selected for optimum security and implementation efficiency. [9].

- The operand A is shifted 2 spaces to the left resulting,

$a_{162}x^{162}$	$a_{161}x^{161}$	...	...	a_1x^1	a_0x^0
		$\Downarrow$			
$a_{162}x^{164}$	$a_{161}x^{163}$	...	...	a_1x^3	a_0x^2

With the Equation(41), the higher degree terms can be reduced by mod $p(\alpha)$.

$$x^{163} = x^7 + x^6 + x^3 + 1$$
$$x^{164} = x^8 + x^7 + x^4 + x$$

The reduction scheme with respect to these equations will be, (the new coefficients of a to be used in the next iteration is named as aa for convenience)

$aa_0 \Longleftarrow a_{161}$
$aa_1 \Longleftarrow a_{162}$
$aa_2 \Longleftarrow a_0$
$aa_3 \Longleftarrow a_{161} \oplus a_1$
$aa_4 \Longleftarrow a_{162} \oplus a_2$
$aa_5 \Longleftarrow a_{161} \oplus a_3$
$aa_6 \Longleftarrow a_{162} \oplus a_{161} \oplus a_4$
$aa_7 \Longleftarrow a_{162} \oplus a_{161} \oplus a_5$
$aa_8 \Longleftarrow a_{162} \oplus a_6$
The rest of the terms are shifted two places to the left.
$aa_{i+2} \Longleftarrow a_i$ for all values from $i = 7$ to $i = 160$ This process is repeated for $\lceil m/D \rceil$ times. In this case: 82 times.

- The multiplier core functions the same way, except the Accumulator is $164(m + D - 1)$ bits long. The number of iterations is 82. This is the critical path of the multiplier as well.

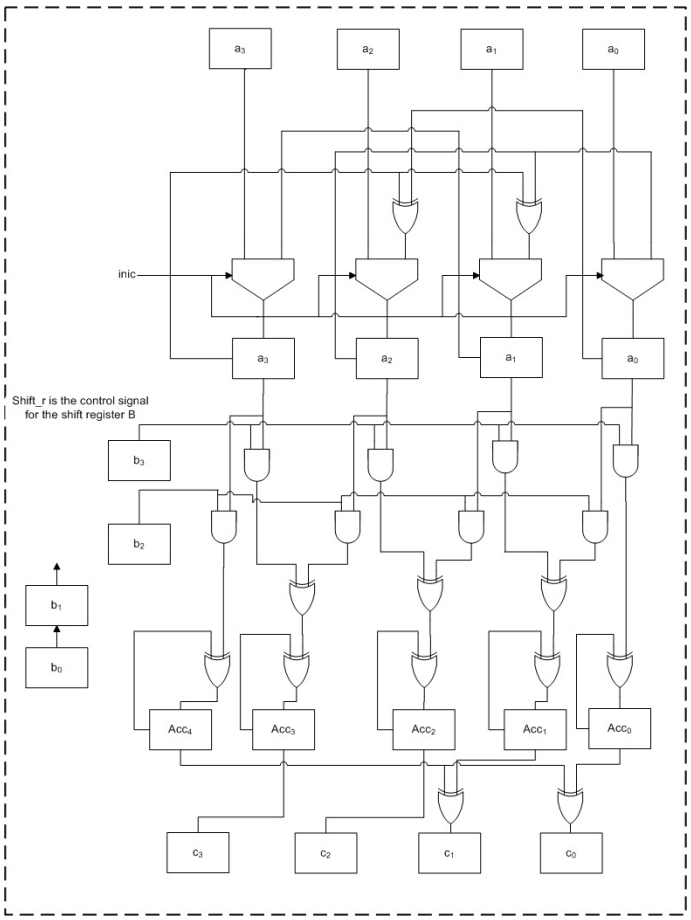


FIGURE 12. SAM DSM with $m = 4$, $D = 2$, $F(x) = x^4 + x + 1$

- In the final reduction, the term with the higher degree is rotated back and the Accumulator is reduced by mod $f(\alpha)$.
  $c_7 \Longleftarrow Acc_7 \oplus Acc_{163}$
  $c_6 \Longleftarrow Acc_6 \oplus Acc_{163}$
  $c_5 \Longleftarrow Acc_5$
  $c_4 \Longleftarrow Acc_4$
  $c_3 \Longleftarrow Acc_3 \oplus Acc_{163}$
  $c_2 \Longleftarrow Acc_2$
  $c_1 \Longleftarrow Acc_1$
  $c_0 \Longleftarrow Acc_0 \oplus Acc_{163}$
  The rest of the terms continue to be the same $c_i \Longleftarrow Acc_i$ from $i = 8$ to $i = 162$.

The same procedure is continued to analyze the multiplier with different digit-sizes. The observations are presented in the Results.

10.2. **Fault Tolerant Design.** Fault detection techniques for Digit Serial Multipliers using polynomial basis doesn't exist. A lot of research has been carried out in the field of Normal basis digit serial multipliers. The digit-serial multipliers using polynomial basis are really compact and tightly packed with their internal logic that, any intrusion, in-order to detect faults is not possible without breaking down the logic and the structure. The scheme in [12] cannot be extended to the scheme, as inherently the reduction schemes are different. The method of finding the parity of the reduction module is not possible here because the result of the multiplication is not direct in a digit serial multiplier as there is an intermediate Accumulated result, hence the Equations(33) and (37) in the Section(9.3.1), cannot be applied here.
The method that can be applied is duplex method, but that doesn't give any advantage w.rt area or performance. Since the area overhead is almost double for just fault detection, fault correction schemes like [19] which considerably less area and delay overhead is considered. The Fault Tolerant technique discussed in [19] is extended to the 163-bit Digit Serial Multiplier and design optimizations are done. The design is explained below and the observations are presented in the results.

10.2.1. *Design Procedure.*

- Divide 163 bits of data into manageable groups and find out the number of parity bits for the given LDPC code.
- Since 163 is a prime number, it cannot be divided into equal groups, hence separate **H** matrices have to be derived.
- Generate predicted parity expressions in terms of the Accumulator Acc_i terms.

The number of parity bits required for $m = k = 163$ is,

$$2^8 \geq 163 + 8 + 1$$

Therefore, it requires only 8 bits to correct single bit errors in a 163 bit data. But encoding 163 data bits with 8 parity bits will be tedious and the module will be able to correct only single bit flip errors. In-order to increase the error correction capabilities and to reduce the complexity of the design, the 163 bit data is divided into fourteen 11 bit groups and one 9 bit group. $(14 * 11) + 9 = 163$. If the number of data bits is 11,

State	Erroneous parity check(s)	Syndrome
No errors	None	0000
Bit 0 p_0 error	p_0	0001
Bit 1 p_1 error	p_1	0010
Bit 2 p_2 error	p_2	0100
Bit 3 p_3 error	p_3	1000
Bit 4 c_0 error	p_0, p_1	0011
Bit 5 c_1 error	p_0, p_2	0101
Bit 6 c_2 error	p_1, p_2	0110
Bit 7 c_3 error	p_0, p_3	1001
Bit 8 c_4 error	p_1, p_3	1010
Bit 9 c_5 error	p_2, p_3	1100
Bit 10 c_6 error	p_0, p_1, p_2	0111
Bit 11 c_7 error	p_0, p_1, p_3	1011
Bit 12 c_8 error	p_0, p_2, p_3	1101
Bit 13 c_9 error	p_1, p_2, p_3	1110
Bit 14 c_{10} error	p_0, p_1, p_2, p_3	1111

TABLE 7. Assignment of parity bits in case of 11 data bits

then the number of parity bits required is 4, as

$$2^4 \geq 11 + 4 + 1$$

This is true in the case of 9 bits of data as well.

$$2^9 \geq 9 + 4 + 1$$

Thus a total of 60 parity bits are used. The *hamming matrices* can be constructed as follows,
For 11 bits, the H matrix is:

c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_10	:	p_0	p_1	p_2	p_3
0	1	0	1	1	0	0	1	1	1	1	:	1	0	0	0
1	0	0	0	1	1	1	0	1	1	1	:	0	1	0	0
0	1	1	0	0	1	1	1	0	1	1	:	0	0	1	0
1	0	1	1	0	0	1	1	1	0	1	:	0	0	0	1

This matrix is used for all data bits till c_{153}. For 9 bits, the H matrix is:

c_{154}	c_{155}	c_{156}	c_{157}	c_{158}	c_{159}	c_{160}	c_{161}	c_{162}	:	p_{56}	p_{57}	p_{58}	p_{59}
0	1	1	0	0	1	1	1	1	:	1	0	0	0
0	0	1	1	1	0	1	1	1	:	0	1	0	0
1	0	0	1	1	1	0	1	1	:	0	0	1	0
1	1	0	0	1	1	1	0	1	:	0	0	0	1

The encoding and the syndrome for 11 data bits is given in the table(7).

State	Erroneous parity check(s)	Syndrome
No errors	None	0000
Bit 0 p_{56} error	p_{56}	0001
Bit 1 p_{57} error	p_{57}	0010
Bit 2 p_{58} error	p_{58}	0100
Bit 3 p_{59} error	p_{59}	1000
Bit 4 c_{154} error	p_{56}, p_{57}	0011
Bit 5 c_{155} error	p_{56}, p_{59}	1001
Bit 6 c_{156} error	p_{58}, p_{59}	1100
Bit 7 c_{157} error	p_{58}, p_{59}	0110
Bit 8 c_{158} error	p_{56}, p_{57}, p_{58}	0111
Bit 9 c_{159} error	p_{56}, p_{57}, p_{59}	1011
Bit 10 c_{160} error	p_{56}, p_{58}, p_{59}	1101
Bit 11 c_{161} error	p_{57}, p_{58}, p_{59}	1110
Bit 12 c_{162} error	$p_{56}, p_{57}, p_{58}, p_{59}$	1111

TABLE 8. Assignment of parity bits in case of 9 bits

This predicted parity bit outputs based on the check matrix is obtained as follows:

$$p_0 = c_1 + c_3 + c_4 + c_7 + c_8 + c_9 + c_{10}$$
$$p_1 = c_0 + c_4 + c_5 + c_6 + c_8 + c_9 + c_{10}$$
$$p_2 = c_1 + c_2 + c_5 + c_6 + c_7 + c_9 + c_{10}$$
$$p_3 = c_0 + c_2 + c_3 + c_6 + c_7 + c_8 + c_{10}$$

The encoding and the syndrome for 9 data bits is given in the table(8).

This gives us the parity bits encoding/parity prediction as stated in [19]

$$p_{56} = c_{155} + c_{156} + c_{159} + c_{160} + c_{161} + c_{162}$$
$$p_{57} = c_{156} + c_{157} + c_{158} + c_{160} + c_{161} + c_{162}$$
$$p_{58} = c_{154} + c_{157} + c_{158} + c_{159} + c_{161} + c_{162}$$
$$p_{59} = c_{154} + c_{155} + c_{158} + c_{159} + c_{160} + c_{162}$$

Parity Generation:
In the same way as the parity bits are generated in the multiple parity prediction, the parity bits from the generated output is calculated. The relationship between the data bits and the parity bits are discussed in the previous section. The parity generation takes place after the output is obtained from the multiplier.

Comparison and Encoding:
These operations are done after the parity of the generated output and the predicted parity bits are available. The syndromes are compared and then the output bits are decoded out of the parity bits. This is also based on the tables given above. The decoding of the first 11 bits, out of the 4 parity bits is shown below. The same pattern

is continued till c_{153} and the decoding after that is done according to the second table, and the check matrix for 9-bit data.

$$
\begin{aligned}
c_0 &= p_0 + p_1 \\
c_1 &= p_0 + p_2 \\
c_2 &= p_1 + p_2 \\
c_3 &= p_0 + p_3 \\
c_4 &= p_1 + p_3 \\
c_5 &= p_2 + p_3 \\
c_6 &= p_0 + p_1 + p_2 \\
c_7 &= p_0 + p_1 + p_3 \\
c_8 &= p_0 + p_2 + p_3 \\
c_9 &= p_1 + p_2 + p_3 \\
c_{10} &= p_0 + p_1 + p_2 + p_3
\end{aligned}
$$

These are the corrected bits. These are XOR-ed with the obtained output and error, if any, is corrected. [19]. The delay overhead is the due to the comparison and syndrome decoding module and the output generation module. Two more clock cycles are required for these operations.

10.3. **Fault Tolerant Digit-Serial Multiplier.** All the modules explained so far are put together. The final design which contains the multiplier module and with error correction using LDPC codes. The Error Correction Modules, i.e,

- Multiple Parity Prediction Circuit
- Syndrome Decoder and Comparison Circuit
- Output Generation Circuit

takes the following form for a multiplier of $DigitSize = 2$ and over the field $GF(2^4)$. The same is extended over a the field $GF(2^m)$. Here, from Section(9.4), the Hamming Matrix for 4 bits is,

$$
H = \begin{bmatrix}
c_0 & c_1 & c_2 & c_3 & p_0 & p_1 & p_2 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1
\end{bmatrix}
$$

Thus, the parity bits are,
$p_0 = c_0 \oplus c_2 \oplus c_3$
$p_1 = c_0 \oplus c_1 \oplus c_3$
$p_2 = c_1 \oplus c_2 \oplus c_3$.
The design is given below,

10.4. **Field Addition.** As explained in the Section(7.1), field addition is done by XORing the corresponding bits of the two inputs. The same architecture is implemented in hardware and an adder is designed over $GF(2^{163})$ The area calculations are presented in the results section.

10.5. **Fault Tolerant Adder.** Adder is also made fault tolerant by using LDPC codes. The same method used for implementing error correction in multiplier is extended to the adder.

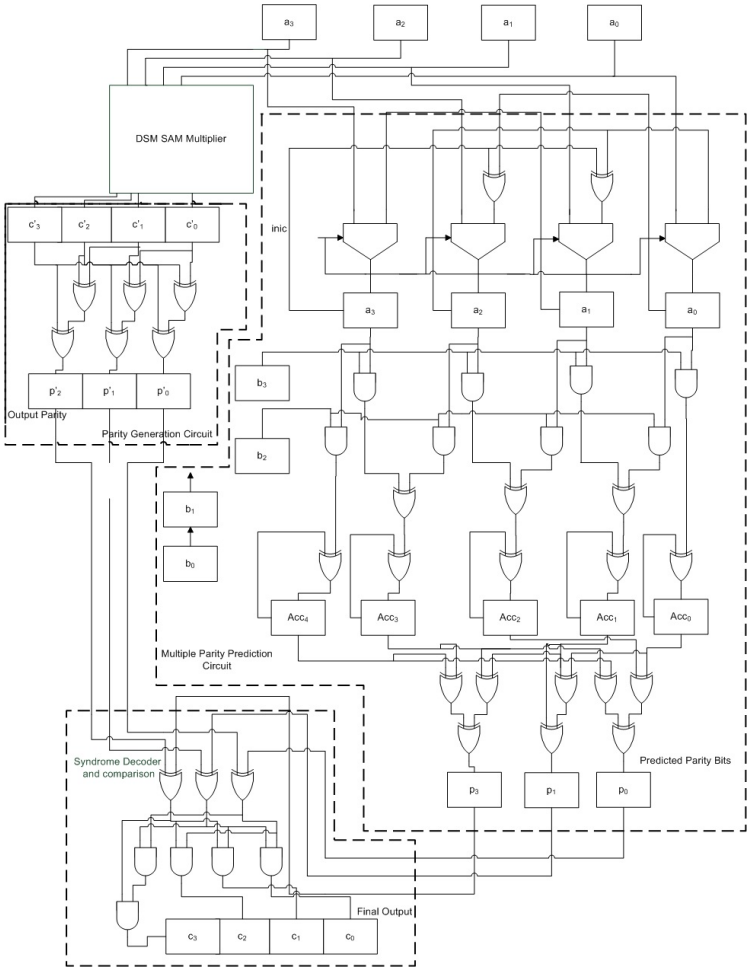


FIGURE 13. Fault Tolerance in Digit Serial Multiplier using LDPC codes

11. Experimental Results

11.1. **Area and Delay: Multiplier.** Digit-serial Multipliers of different field and digit sizes where designed and coded in VHDL. The first analysis made is over the number of AND gates and XOR gates and number of flip flops required in the design. Confirming the results stated in [34], for a Galois Field of $m = 15, k = 1$ the results tabulated in the table(9)were obtained.

Digit-sizes	No. of XORs	No. of ANDs	No. of Flip-Flops
2	34	36	33
4	70	74	35
6	106	112	37
8	142	150	39

Table 9. Number of XOR and AND gates in the multipliers for different digit sizes

The results are presented in the form of a graph in the figure(14)for better understanding.

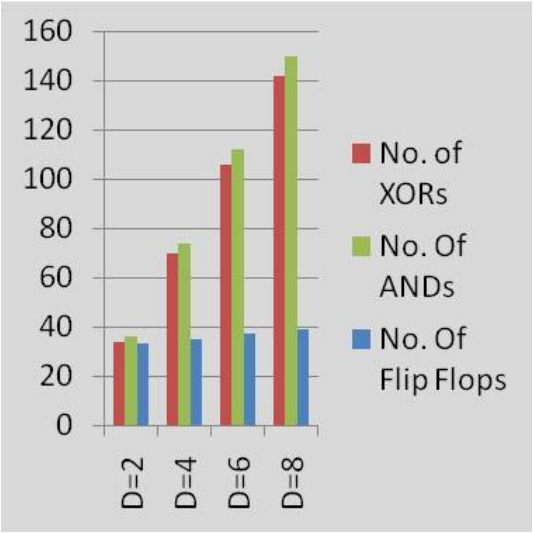
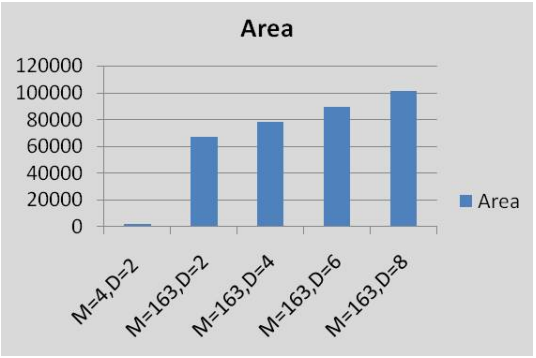


Figure 14. Number of Gates and Flip-Flops

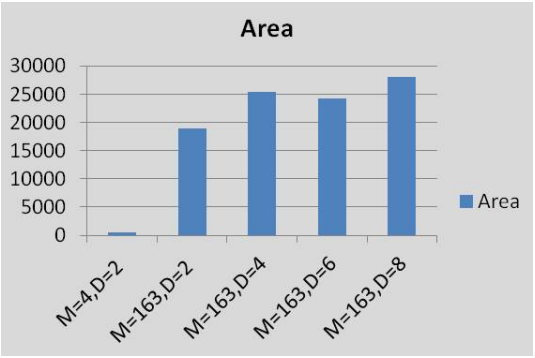
The multiplier was then extended over the field $GF(2^m)$ and various digit sizes. The designed were simulated using $ModelSim^{TM}$. The designs were then synthesized using $Synopsys^{TM}$ tools in the UMC technology, with 180nm and 90nm technology nodes. The results obtained are presented in the tables (10) and (11).
The comparison of areas can be better understood from the graphs in the figure (15a) and (15b). The worst case slack comparisons are given in the figure(16a) and (16b).

Multiplier	Area	Worst Case Slack
$m = 4, D = 2$	$471.34\mu m^2$	49.56nS
$m = 163, D = 2$	$18979.96\mu m^2$	49.13nS
$m = 163, D = 4$	$25374.79\mu m^2$	48.74nS
$m = 163, D = 6$	$24257.11\mu m^2$	48.97nS
$m = 163, D = 8$	$28097.69\mu m^2$	48.77nS

TABLE 11. Area and Worst Case Slack delay values across multipliers of various digit sizes synthesized using 90 nm technology node



(A) Comparison of Areas of the Multipliers of various digit sizes (180nm technology node)

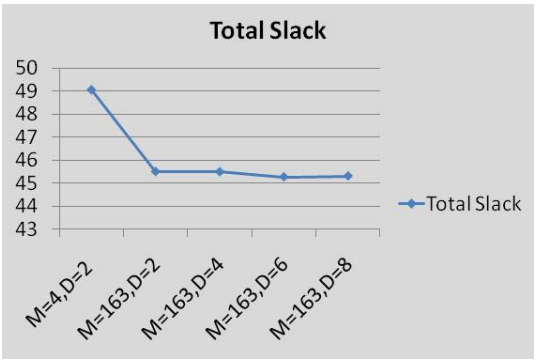


(B) Comparison of Areas of the Multipliers of various digit sizes (90nm technology node)

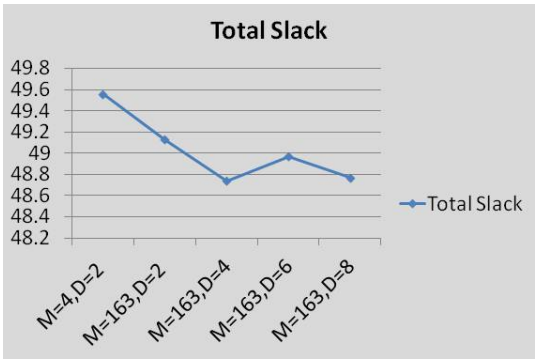
FIGURE 15. Comparison of Area of the Multipliers across digit sizes synthesized in 180nm and 90nm technology nodes

Multiplier	Area	Worst Case Slack
$m = 4, D = 2$	$1738.598022\mu m^2$	49.06nS
$m = 163, D = 2$	$66789.218750\mu m^2$	45.50nS
$m = 163, D = 4$	$78059.468750\mu m^2$	45.49nS
$m = 163, D = 6$	$89265.218750\mu m^2$	45.27nS
$m = 163, D = 8$	$101361.328125\mu m^2$	45.31nS

TABLE 10. Area and Worst Case Slack delay values across multipliers of various digit sizes synthesized using 180 nm technology node



(A) Comparison of Worst case slacks of the Multipliers of various digit sizes (180nm technology node)



(B) Comparison of Worst case slacks of the Multipliers of various digit sizes (90nm technology node)

FIGURE 16. Comparison of Worst case slacks of the Multipliers across digit sizes synthesized in 180nm and 90nm technology nodes

11.1.1. *Area and Delay Overhead due to Error Correction Modules.* From the diagram in Section(10.3), the modules added in-order to achieve error correction are:

- Multiple Parity Prediction Unit
- Parity Generation Unit, and
- Syndrome Decoder and Output Generator Unit

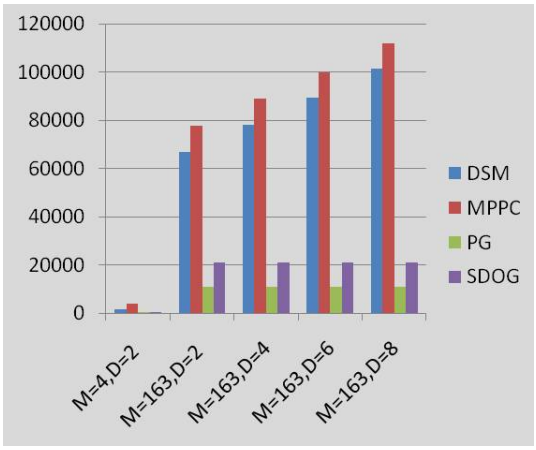
Comparison of Area of different modules synthesized with 180nm and 90nm technology nodes: These modules are also different w.rt to the digit sizes and field size. Each of the modules are designed and coded by using VHDL, and synthesized using $Synopsys^{TM}$ tools. The Area of the individual modules for different digit sizes and field sizes are obtained from the Area Reports in *Synopsys, DesignVision*^{TM}. The comparison of the area of the different modules synthesized using 180nm technology node is given in the table(12) and the results of synthesizing using 90nm technology node is given in the table(13). The acronyms are DSM: Digit Serial Multiplier, MPPC: Multiple Parity Prediction Circuit, PG: Parity Generation, SDOG: Syndrome Decoder and Output Generator. The comparison of area of the modules with both the technologies can be better understood from the Figure(17a) and (17b)

Multiplier	DSM	MPPC	PG	SDOG
$m = 4, D = 2$	$1738.598022\mu m^2$	$4038.425\mu m^2$	$358.042\mu m^2$	$532.22\mu m^2$
$m = 163, D = 2$	$66789.218750\mu m^2$	$77578.85\mu m^2$	$10786.4\mu m^2$	$21063.25\mu m^2$
$m = 163, D = 4$	$78059.468750\mu m^2$	$88849.1\mu m^2$	$10786.4\mu m^2$	$21063.25\mu m^2$
$m = 163, D = 6$	$89265.218750\mu m^2$	$100054.8\mu m^2$	$10786.4\mu m^2$	$21063.25\mu m^2$
$m = 163, D = 8$	$101361.328125\mu m^2$	$111925.2\mu m^2$	$10786.4\mu m^2$	$21063.25\mu m^2$

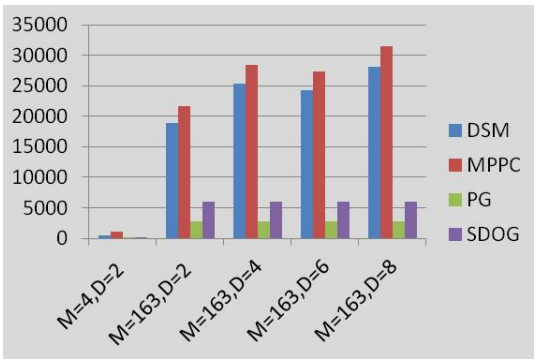
TABLE 12. Comparison of Area of different modules synthesized using 180nm technology node

Multiplier	DSM	MPPC	PG	SDOG
$m = 4, D = 2$	$471.34\mu m^2$	$1049.93\mu m^2$	$95.96\mu m^2$	$151.70\mu m^2$
$m = 163, D = 2$	$18979.94\mu m^2$	$21709.90\mu m^2$	$2790.64\mu m^2$	$5989.84\mu m^2$
$m = 163, D = 4$	$25374.79\mu m^2$	$28369{,}35\mu m^2$	$2790.64\mu m^2$	$5989.84\mu m^2$
$m = 163, D = 6$	$24257.12\mu m^2$	$27399.86\mu m^2$	$2790.64\mu m^2$	$5989.84\mu m^2$
$m = 163, D = 8$	$28097.68\mu m^2$	$31545.25\mu m^2$	$2790.64\mu m^2$	$5989.84\mu m^2$

TABLE 13. Comparison of Area of different modules synthesized using 90nm technology node



(A) Comparison of Areas of different modules in the design, across the digit sizes with 180nm technology node



(B) Comparison of Areas of different modules in the design, across the digit sizes with 90nm technology node

FIGURE 17. Comparison of Area of different modules in the design across digit sizes synthesized in 180nm and 90nm technology nodes

The Area overhead by percentage is given in the tables (14) and (15).

Multiplier	Only MPPC block	Including all the blocks
$m = 163, D = 2$	116.15%	163.85%
$m = 163, D = 4$	113.82%	154.62%
$m = 163, D = 6$	112.08%	147.77%
$m = 163, D = 8$	110.42%	141.84%

TABLE 14. Area Overhead: 180nm Technology Node

Multiplier	Only MPPC block	Including all the blocks
$m = 163, D = 2$	114.38%	160.65%
$m = 163, D = 4$	111.80%	146.40%
$m = 163, D = 6$	112.95%	149.15%
$m = 163, D = 8$	112.26%	143.52%

TABLE 15. Area Overhead: 90nm Technology Node

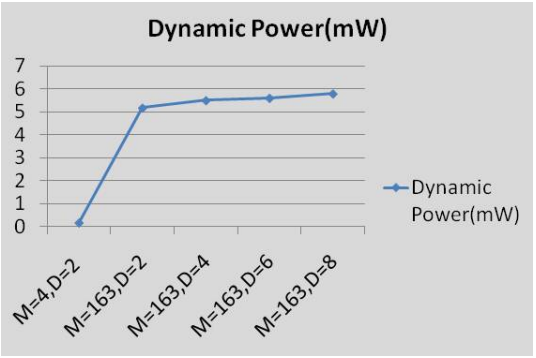
11.1.2. *ASIC implementations: Dynamic and Cell Leakage Power comparisons: Multiplier.* The codes are synthesized using Synopsys, Design Vision. *Synopsys Power Compiler*^{TM} is used for power analysis. The table(16) shows the Dynamic Power and Cell Leakage Power for the multipliers, implemented with a 180nm technology node and the table(17) have the values generated using 90nm technology node. The comparison of the dynamic power values across the designs and technology nodes can be understood better from the graphs illustrated in the Figures (18a), (18b) and (19a), (19b). The Multiple Parity Prediction Block, which is the overhead for error correction, is analyzed by using *Synopsys Power Compiler*^{TM} and the results obtained are presented in the Figure(20a) and (20b) with 180nm and 90nm nodes respectively which gives a comparison between the power consumption of the DSM(Digit Serial Multiplier:Data Path) and the MPPC(Multiple Parity Prediction Circuit) across both 180nm and 90nm and it shows that the power requirements/overhead due to the extra blocks is very less. The analysis of these data is done in the Section(12)

Multiplier	Dynamic Power	Cell Leakage Power
$m = 4, D = 2$	148.430μW	7.916nW
$m = 163, D = 2$	5.1852mW	290.2nW
$m = 163, D = 4$	5.5119mW	347.63nW
$m = 163, D = 6$	5.6141mW	414.338nW
$m = 163, D = 8$	5.8052mW	490.63nW

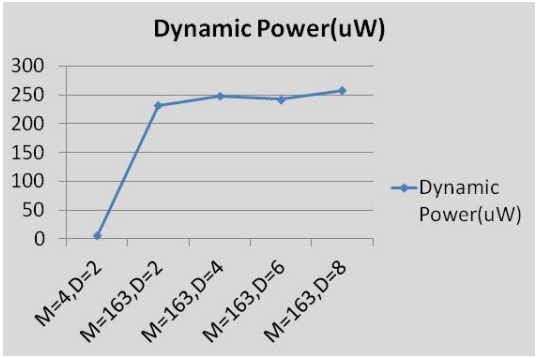
TABLE 16. Comparison of Dynamic and Cell Leakage Power across digit sizes synthesized using 180nm technology node

Multiplier	Dynamic Power	Cell Leakage Power
$m = 4, D = 2$	5.5μW	0.529μW
$m = 163, D = 2$	231.83μW	19.40μW
$m = 163, D = 4$	248.64μW	27.82μW
$m = 163, D = 6$	242.86μW	28.34μW
$m = 163, D = 8$	258.29μW	33.36μW

TABLE 17. Comparison of Dynamic and Cell Leakage Power across digit sizes synthesized using 90nm technology node

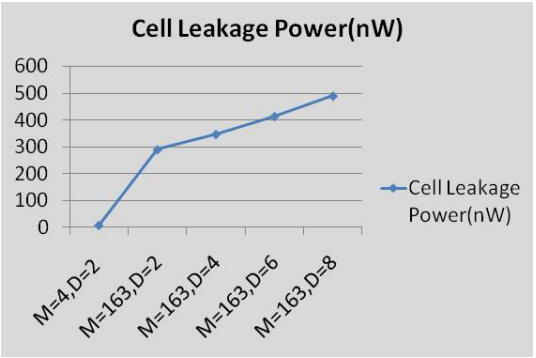


(A) Comparison of Dynamic Power Values(180nm node)

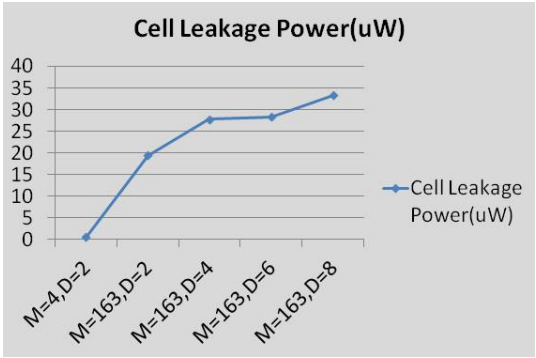


(B) Comparison of Dynamic Power Values(90nm node)

FIGURE 18. Comparison of Dynamic Power values across designs synthesized with 180nm and 90nm technology nodes

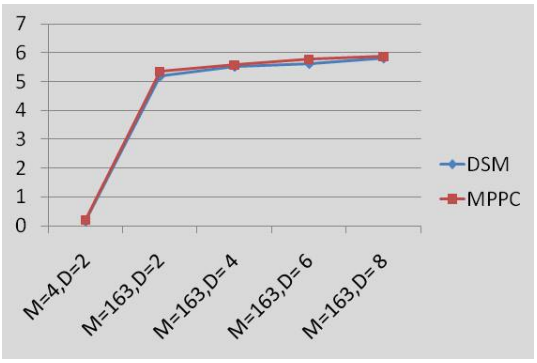


(A) Comparison of Cell Leakage Power Values(180nm node)

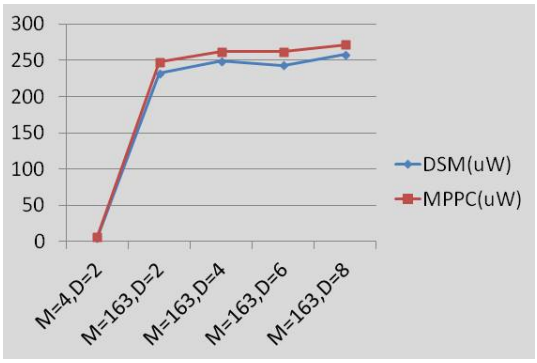


(B) Comparison of Cell Leakage Power Values(90nm node)

FIGURE 19. Comparison of Cell Leakage Power values across designs synthesized with 180nm and 90nm technology nodes



(A) Comparison of the power consumption of the Digit Serial Multiplier(DSM) and the Multiple Parity Prediction Block(MPPC)(180nm node)



(B) Comparison of the power consumption of the Digit Serial Multiplier(DSM) and the Multiple Parity Prediction Block(MPPC)(90nm node)

FIGURE 20. Comparison of the power consumption of the Digit Serial Multiplier(DSM) and the Multiple Parity Prediction Block(MPPC) synthesized with 180nm and 90nm technology nodes

11.1.3. *Layouts.* Entire design, which is implemented in VHDL, is synthesized using *Synopsys*. An optimized Verilog description of the design can be created using Synopsys and it is used with *SOC Encounter* to create a layout of the design. The layout of the Digit-Serial Multiplier is given in the figure(21). This can be easily compared and with the layout of the entire design given in the figure(22). The density of the used up real-estate shows the difference in the design.

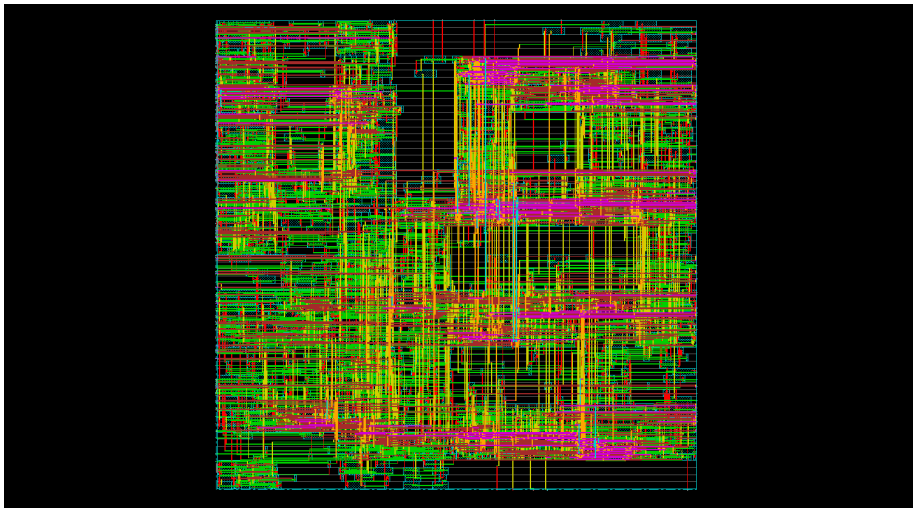


FIGURE 21. Layout of the Digit Serial Multiplier without Fault Tolerance, synthesized using 180nm technology node

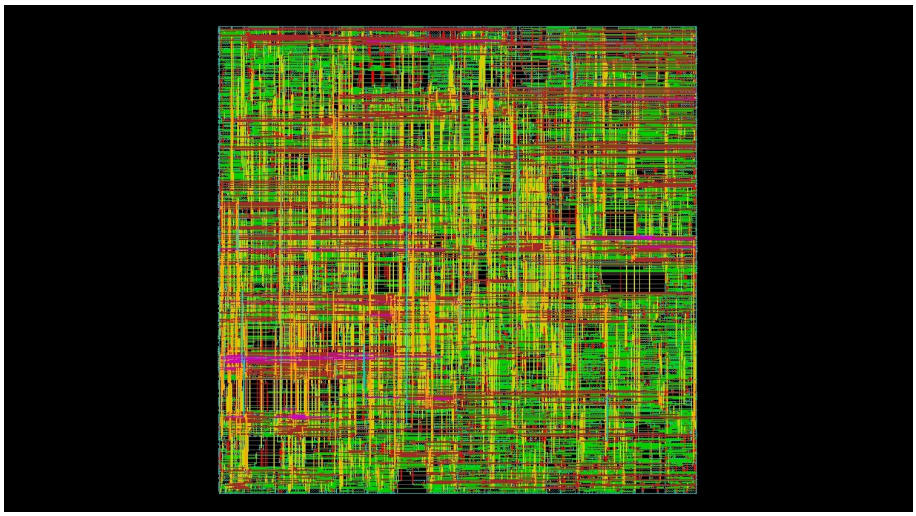


FIGURE 22. Layout of the Fault Tolerant Digit Serial Multiplier synthesized using 180nm technology node

11.1.4. *FPGA implementations.* Using Xilinx Design Suite 13.2, the VHDL codes were implemented on two different Target Devices, XC7v1500T and XC7v200T. These two devices comes under the Virtex®-7 FPGA family. Devices in this family are optimized for advanced systems which require high performance and high bandwidth connectivity. These devices are built on a common 28nm architecture. They have remarkable power efficiency and twice as much better system performance with half as much power consumption than the previous generation of FPGAs. They have upto 2 million logic cells, 67Mb of internal memory, 4.5 Tera-MACS DSP throughput, 2.8 Tb/s serial bandwidth and fully integrated Agile Mixed signal Capabilities, [38].
Multipliers over the field, $GF(2^{163})$, with various digit sizes were implemented and placed on these FPGAs. The Figure(23) and Table(18) contain the number of registers and number of LUTs consumed by the designs. The table(19) shows the worst case slack on both the devices for all the multipliers.

XC7v1500T, XC7v2000T: No. of Registers	
$m = 163, D = 2$	890
$m = 163, D = 4$	892
$m = 163, D = 6$	894
$m = 163, D = 8$	896

XC7v1500T: No. of LUTs	
$m = 163, D = 2$	764
$m = 163, D = 4$	952
$m = 163, D = 6$	1105
$m = 163, D = 8$	1268

XC7v1500T: No. of LUTs	
$m = 163, D = 2$	780
$m = 163, D = 4$	968
$m = 163, D = 6$	1117
$m = 163, D = 8$	1313

TABLE 18. Comparison of the number of registers and LUTs consumed by the design

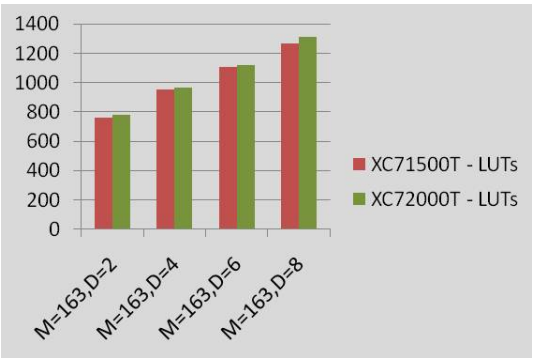


FIGURE 23. Comparison of the number of Look Up Tables occupied by the multipliers

Multiplier	XC7v1500T	XC7v2000T
$m = 163, D = 2$	2.377nS	2.627nS
$m = 163, D = 4$	2.434nS	2.312nS
$m = 163, D = 6$	2.475nS	2.334nS
$m = 163, D = 8$	2.340nS	2.478nS

TABLE 19. Comparison of Worst Case Delays for the Multipliers of various digit sizes implemented on XC7v1500T and XC7v2000T- Virtex-7 devices

11.2. **Area and Delay: Addition.** A simple adder circuit is created over the Field $GF(2^{163})$as per the Algorithm explained in the Section(7.1), using VHDL. The designs are implemented using *Synopsys* using 180nm and 90 technology nodes. The number of gates required are the same as the field size. For a field of 163 it requires 326 XOR gates for Bit-Parallel Addition. Area and the worst case slack obtained for when the designs were implemented are given in the table (20) (180nm and 90nm respectively).

Technology Node	Area	Worst Case Slack
$180nm$	$19865.26\mu m^2$	49.23nS
$90nm$	$4463.63\mu m^2$	49.58nS

TABLE 20. Area and Worst Case Slack delay values of adders synthesized using 180 nm and 90 nm technology nodes

11.2.1. *Area and Delay Overhead due to Error Correction Modules.* A fault Tolerant Adder is designed using LDPC(Low Density Parity Check) codes. Each module are synthesized in Synopsis over both 180nm and 90nm technology nodes. Area of the individual modules across both the technology nodes are given in the table (21). The acronyms have the following meaning, MPPC(Multiple Parity Prediction Circuit), PG(Parity Generation) and SDOG (Syndrome Decoder and Output Generator). The Area overhead percentages are given in the table (22)

Technology Node	Adder	MPPC	PG	SDOG
$180nm$	$19865.26\mu m^2$	$32592.06\mu m^2$	$12700.19\mu m^2$	$23188.33\mu m^2$
$90nm$	$4463.63\mu m^2$	$7247.22\mu m^2$	$2790.65\mu m^2$	$5283.53\mu m^2$

TABLE 21. Comparison of Area of different modules in the Field Adder synthesized using 180nm technology node

Technology Node	Area Overhead
$180nm$	164.06%
$90nm$	162.36%

TABLE 22. Area Overhead: over 180nm and 90nm Technology Nodes

11.2.2. *ASIC Implementations: Dynamic and Cell Leakage Power Comparisons: Adder.* Similar to the power analysis performed on the multiplier, the adder module is also compiled using *Synopsys Power Compiler*^{TM}. The table (23) shows the dynamic power and cell leakage power is shown in the table (24).

Technology Node	Dynamic PowerμW
$180nm$	$527.89\mu W$
$90nm$	$68.35\mu W$

TABLE 23. Dynamic Power Consumption of adders synthesized over 180nm and 90nm technology nodes

Technology Node	Dynamic PowerμW
$180nm$	$0.112\mu W$
$90nm$	$5.99\mu W$

TABLE 24. Cell Leakage Power values of the Adders synthesized over 180nm and 90nm Technology Nodes

A comparison of the power consumption of the Error Correction Module and the Adder module is given in the table (25)to show the efficiency of the added module w.r.t power overhead.

Technology Node	Adder Module	Error Correction Module
$180nm$	$526.89\mu W$	$710.07\mu W$
$90nm$	$68.35\mu W$	$89.40\mu W$

TABLE 25. Cell Leakage Power values of the Adders synthesized over 180nm and 90nm Technology Nodes

11.2.3. *FPGA implementations: Addition.* The VHDL code can be implemented on a FPGA using *Xilinx* tools. The number of registers used and the LUTs occupied by the design is an important measure showing the efficiency of the design. These figures are given in the table (26).

	XC7v1500T	XC7v2000T
Registers	332	332
LUTs	191	167

TABLE 26. Number of Registers used and LUTs occupied by the Adder on XC7v1500T and XC7v2000T

12. Analysis of Results

12.1. **Observations from Area Calculations.** The main observations can be summarized as given below:

- Four versions of the Digit Serial Multipliers are designed in VHDL and they are compared with each other and with the multipliers described in the Section(7). The results proves that the Digit-Serial Multipliers are the smallest and the most optimized Filed Multipliers. The tables (4) and the table(9) can be compared and it can be inferred that this point stands proven.
- All the previous architectures described evidently prove that the complexity of the multiplier depends on the primitive polynomial and the polynomial $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$, is found to be the best for the field size of $m = 163$ as recommended in [34].
- As the digit-size increases there are two important advantages
  - the area overhead goes down(refer Table(14) and (15)).
  - the multiplier is faster as the number of cycles required is $\lceil m/D \rceil$ for the multiplication + 1 cycle for the final reduction + 2 for the error correction, where $m = fieldsize$ and $D = DigitSize$.

  Going along the same idea, it is assumed that the multiplier can be the most optimum for the $DigitSize \leq 152$, [34].
- The area of the multiplier and the Parity Prediction Unit is about the same size. The area overhead along with the logic to derive the 15 parity bits, for all the versions of the multipliers is less than 116% for implementations in 180nm and 90nm technology nodes (refer Table(14) and (15)).

12.2. **Performance Analysis: Power.** From [19], the power consumption overhead for Hamming Code based methods of error correction is 106.2%. The power analysis of the implemented designs shows the power consumption overhead to be 101.2% for a multiplier with $digitSize = 8$ and $m = 163$, and this value is almost the same for all the implemented multipliers suggesting that the implemented Fault Tolerant Multiplier is superior.

12.3. **Performance Analysis: Time.** The implemented Fault tolerant Processor (both the modules: Adder and Multiplier), gives error corrected output with a delay overhead of only two cycles. If the Digit Serial Multiplier is implemented with maximum optimization with $digitSize = 152$, the overhead would be 166.6%, but given the design complexity with such a large digitsize, we can afford to have two extra cycles for simpler multipliers. For example, for the implemented multipliers, the one with maximum optimization, (i.e. digitSize = 8) the delay overhead is 109.1%.
In case of the adder there is no other way to implement a bit-parallel addition without a 300% delay overhead. Since the focus of this research is multipliers other fast, optimal implementations of adders like [27] are not considered. Adders in [27] consider a digit serial approach for addition, hence the two cycle overhead can be afforded, in comparison. The worst case delay values for the multipliers for both FPGA and ASIC implementation evidently proves that ASIC gives better performance in terms of time. (refer tables(10) and (11) and (19))

13. Evaluation of Results

The aim of the project is to investigate design methodologies for reliable low power Galois Field Arithmetic Processor, and to introduce Fault Tolerance to the design. This has been successfully completed. The Objectives stated in the Section(5), are evaluated and their completion is justified in the following section.

- **Design Methodologies for reliable low power Galois Field Arithmetic Processor**
  Thorough research was carried out from the beginning of the project to understand the complexities of the mathematics involved in the project and the previously proposed architectures were completely understood in order to select the correct approach for the design. Before selecting the final design, a Digit Serial Multiplier with a Single Accumulator, different other approaches like those stated in [16], [13], [12] were studied and compared for their efficiency and performance. The idea was to select the most compact design in terms of area and speed.
  Consideration was given to the adaptability of the design, so that Fault tolerance could be added easily without a break down of the architecture.
  The final design which is a spin out from the Digit Serial Multipliers stated in [34] and [36]. Certain changes were made in the implementation of the design, which are explained in the Section(10). The architecture improvements helped in the reduction of complexity of the design. The entire design works with a finite state machine logic, which can be easily made power efficient by switching on and off the modules which are/are not being used. The same can be said about the adder block. Thus we can conclude that the objective of designing *a reliable, low power Finite Field Processor* has been achieved.
- **Efficiency of the design for GF arithmetic Processor**
  A design can be argued to be efficient, but there can always be an improvement. The efficiency of a design should be stated with respect to the context in which its efficiency is considered. In our case, a trade off is achieved between speed and area. It can be concluded that the designs implemented in the project, both the multiplier and the adder, which are the two operations of prime importance in Field Arithmetic, are *optimum*. The Digit Serial Multipliers provide an alternate between the fast and big bit-parallel and the slow and small bit serial multipliers.
- **Investigate and implement state-of-the-art methods of fault tolerance**
  This is the second development stage of the project. Several Fault detection and correction architectures were considered, [12], [14], but these methods were found to be not efficient for the digit serial multiplier we implemented. The processor blocks were optimized conceptually and design-wise to the maximum possible extend that any additions to the architecture disturbed it. Concurrent Fault Detection techniques by using parity bits proposed in [13] for Bit-Serial and Bit-parallel architectures were combined conceptually for fault detection in digit serial multipliers but the idea, as novel as it sounds was not easy to investigate or implement. The intricacies of the digit multiplication and the final reduction stages of the digit serial multipliers posed a tough wall to break through. Had it been a 100% mathematical research project, dwelling deep in the concepts of

field multiplication, this would have been successful. The trials made in this direction proved to be futile as the area overhead was around a TMR system.
The adaptable Fault Correction architectures proposed in [19], proved to be an efficient alternative. LDPC codes provided and easy and efficient method of fault correction. Parity Prediction and the error correction using this method proved that a change in the design with a huge area overhead for just fault detection is unnecessary. Parity Prediction using LDPC codes suggest the use of $'r'$ parity bits for $'d'$ data bits, such that $2^r \geq d + r + 1$ [17]. This would suggest the use of $'8'$ parity bits for the $'163'$ data bits. Since the design of the multiple parity prediction unit, the logic that derives the parity bits is optimum enough, it was decided to increase the efficiency of fault tolerance. Instead of considering the parity of the entire block of data, the 163 bit data is broken down to small units which can be dealt with 4 parity bits each. Thus the 15 blocks of data is separately considered with parity bits for each and the implemented design can *correct* up to 40 single bit errors. The area and delay overhead introduced due to these added blocks are analyzed in the Section(12) Thus we can conclude that the objective of *designing an efficient fault tolerant architecture for the GF Arithmetic processor* is achieved.

- **Reduce potential delay overhead that may be introduced by redundant logic**
  The threat of the Fault tolerant modules slowing the processor down is given importance right from the conception of the design. The core modules, the multiplier in the processor and the multiplier parity prediction module have almost redundant logic. The total parity prediction logic comes close to about 100%. The parity prediction circuit completes it operations and will be ready with its parity by the time the multiplier produces the output. There is no delay overhead due to the redundant logic at all, as they are disjoint.
- **Overhead due to the added blocks should compromise with the reliability they provide** This is the trade-off. The method by which a careful middle ground achieved in terms of reliability and efficiency but without a huge overhead was contemplated through out the project. At times when soft errors in digital circuits are causing a lot of malfunctions and the threat of fault based attack on Cryptographic systems is on a higher side, the necessity of introducing fault tolerance is important.
  In [19], the delay overhead is due to,
  - parity generation block, which calculates the parity of the generated output,
  - syndrome decoder and comparison circuit, which compares the predicted parity bits with that of the output parity,
  - output generator, which generates the final output.

  In the implemented design there is only a two stages of delay as the one XOR delay due to the output generator block is removed, as the output generator and syndrome decoder/comparison circuits are combined. This reduces the delay from three clock cycles to two.
  As compared to the TMR systems which requires more than three times the hardware, the implemented design takes very less area overhead as well.

- **Design should be efficient in terms of performance** The analysis of the design in the Section(12), undoubtedly proves that the design is efficient in both ASIC implementation and on FPGAs. The slack/area results obtained are presented in the same section.

The efficiency of the design in being

- Fast - high performance
- Low power
- Fault Tolerant

is thus enunciated. The points stated above undeniably conclude that the aims and objectives proposed at the beginning of the research project have been successfully achieved.

14. Suggestions and Future Work

- **Multipliers:** In [34], the digit-serial multiplier is further optimized increasing the number of accumulators. These architecture options are better in terms of area and delay. These multipliers can further push the limits of optimization when coupled with the research into the special polynomials selected by methods suggested in [36]. The potential of research into the mathematics that binds these architectures are immense.
- **Fault Tolerant Techniques specifically for DSM:** Since there are no Fault Detection Techniques tailor made for DSMs using polynomial bases, it will be an interesting adventure to find one. The mathematics is so intriguing and fascinating that a lot of time was utilized only for fault detection in DSMs during the research. A team of coding theory experts and mathematicians can come up with a novel idea.
- **ECC, point operations:** This processor can be further extended for point based operations for Elliptical Curve Cryptography. Some research has already been done along this lines, [35], these ideas can be incorporated along with the fault tolerant approach in this paper and a new design can be developed.
- **Improvement in Fault Tolerant methods:** Systematic approach for error correction is presented only in [20] and [19]. These are completely disjoint Fault Tolerant techniques for Field Multipliers. These architectures can be further extended in such a way that the area overhead can be further reduced by incorporating the fault tolerance into the multiplier core. The ideas can be derived from [12].
- **Fault Based Attack Resilient Processors:** Extensive study is being conducted in this field, and the possibility of the implemented architecture to be fault based attack resilient can be investigated.
- **Inversion, Exponentiation and Squaring modules can be added**
- **Special instruction set can be provided for the processor**

References

[1] G. B. Agnew, T. Beth, R. C. Mullin, S. A. Vanstone *Arithmetic operations in* $GF(2^m)$, Journal of Cryptology, Vol. 6, No. 1. pp.3-13, 1993.

[2] Andrea Pellegrini, Valeria Bertacco and Todd Austin *Fault-Base Attack of RSA Authentication* ,University of Michigan, Published in: DATE '10 Proceedings of the *Conference on Design, Automation and Test in Europe European Design and Automation Association 3001* Leuven, Belgium, Belgium. 2010

[3] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin *"The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design,"* IEEE Trans. Computer., vol. C-2O, pp. 1312–1321, November, 1971.

[4] I. Blake, G. Seroussi, N. P. Smart *Elliptic curves in cryptography*, in London Mathematical Society Lecture Note Series, Cambridge, U.K. Cambridge University Press, 1999.

[5] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, V. Piuri, *Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard*, IEEE Transactions on Computers, Special issue on Cryptographic Hardware and Embedded Systems, Vol. 52, No. 4, pp. 492-505, April, 2003.

[6] D. Boneh, R. A. DeMillo, R. J. Lipton, *On the Importance of Eliminating Errors in Cryptographic Computations*, Journal of Cryptology, 14, pp. 101–119, 2001.

[7] G.L. Feng *A VLSI Architecture for Fast Inversion in* $GF(2^m)$ IEEE TRANSACTIONS ON COMPUTERS, VOL. 38, No. 10 October, 1989.

[8] S. Fenn, M Gossel, M Benaissa, D. Taylor, *Online Error Detection for Bit-serial Multipliers in* $GF(2^m)$, J. Electronic Testing: Theory and Applications, Vol. 13, pp.29-40, 1998.

[9] Federal Information Processing Standards Publication, 186-3, Digital Signature Standard(DSS), Category: Computer Security, Sub-category: Cryptography.

[10] G. Gaubatz, B. Sunar, *Robust finite field arithmetic for fault tolerant public-key cryptography*, Second Workshop on Fault Tolerance and Diagnosis in Cryptography (FTDC), 2005.

[11] R. Gallager, *Low-Density Parity-Check Codes*, Massachusetts Institute of Technology Press, Cambridge, 1963.

[12] Arash Reyhani-Masoleh and M. Anwar Hasan, *Fault Detection Architectures for Field Multiplication Using Polynomial Bases* IEEE TRANSACTIONS ON COMPUTERS, VOL. 55, NO. 9 September, 2006.

[13] Arash Reyhani-Masoleh and M. Anwar Hasan, *Low Complexity Bit Parallel Architectures for Polynomial Basis Multiplication over* $GF(2^m)$ IEEE TRANSACTIONS ON COMPUTERS, VOL. 53, NO. 8, August, 2004.

[14] Siavash Bayat-Sarmadi, M.A. Hasan, *Concurrent Error Detection in Finite Field Arithmetic Operations using Pipelined and Systolic Architectures* University of Waterloo, Ontario, Canada November, 2007

[15] T. Itoh and S. Tsujii *A fast Algorithm for computing multiplicative inverses in* $GF(2^m)$ *using normal basis* "Information and Computation", Vol. 78, No.3, pp.21-40, September, 1988.

[16] Jean-Pierre Deschamps, Jose Luis Imana and Gustavo D. Sutter, *Hardware implementation of Finite Field Arithmetic* The McGraw-Hill Companies Inc. 2009.

[17] B. W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.

[18] Jimson Mathew, *Fault Tolerant Computing and VLSI Testing*, Lecture Notes. FAULT TOLERANT LECTURE 6, Dept. of Computer Science, University of Bristol, 2010-2011.

[19] Jimson Mathew, J. Singh, A. M. Jahir, M. Hosseinabady, D. K. Pradhan, *Fault Tolerant Bit Parallel Finite Field Multipliers using LDPC Codes*, Dept. Of Computer Science, University of Bristol, and Oxford Brookes University, 2008.

[20] Jimson Mathew, A. Costas, A. M. Jabir, H. Rahaman, D.K. Pradhan, *Single Error Correcting Finite Field Multipliers Over GF(2m)*. 21^{st} International Conference on VLSI Design, January, 2008.

[21] E.D. Mastrovito, *VLSI Architectures for Computation in Galois Fields* PhD thesis, Linkoping Univ., Linkoping, Sweden, 1991.

[22] E.D. Mastrovito, *VLSI Designs for Multiplication over Finite Fields* $GF(2^m)$ Proceedings at the Sixth Symposium, Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC-6) July, 1988.

[23] Francis P. Mathur, Algirdas Avizienis, *Reliability analysis and architecture of a hybrid-redundant digital system: Generalized triple modular redundancy with self repair* Spring Joint Computer Conference, 1970

[24] Michael Nicolaidis *Carry Checking/Parity Prediction Adders and ALUs* IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 11, NO. 1, February 2003

[25] Mitra S., Seifert N., Zhang M., Shi Q, Kim K, *Robust System Design with Built-In Soft Error Resilience*, IEEE Computer, Vol. 38, No. 2, pp. 43-52, February, 2005.

[26] G. Orlando, C.Paar, *A high Performance Reconfigurable Elliptic Curve Processor for* $GF(2^m)$, Cryptographic Hardware and Embedded Systems - CHES 2000, Vol. LNCS 1965, Sprinfer-Verlag, 2000.

[27] Pramod Kumar Meher *On Efficient Implementation of Accumulation in Finite Field Over* $GF(2^m)$ *and its applications* IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 17, NO. 4, April, 2009.

[28] C. Paar *Efficient VLSI Architectures for Bit Parallel Computation in Galois Field* PhD Thesis, University of GH Essen, 1994.

[29] Dhiraj K. Pradhan, *Fault-Tolerant Computer System Design.* Prentice Hall PTR, New Jersey, 1st Edition, 1996.

[30] Robert J. McEliece, *Finite Fields for Computer Scientists and Engineers*, Kluwer Academic Publishers, Norwell, Massachusetts, USA, 1986

[31] Siavash Bayat-Sarmadi *Concurrent Error Detection in Finite Field Arithmetic Operations* ,presented to University of Waterloo, Dept. of Electical and Computer Engineering Waterloo, Ontario, Canada, 2007.

[32] Ç.K. Koç, B. Sunar, *Low-Complexity Bit-Parallel Canonical and Normal Basis Multipliers for a Class of Finite Fields* IEEE TRANSACTIONS ON COMPUTERS, VOL. 47, NO. 3, March 1998

[33] Robert R. Stoll, *Set Theory and Logic* W. H. Freeman And Company, San Fransisco and London 1961

[34] Sandeep Kumar, Thomas Wollinger, Christof Paar, *Optimum Digit Serial* $GF(2^m)$ *Multipliers for Curve Based Cryptography*, Communication Security Group, (COSY), Ruhr-Universitaet Bochum, Germany, March, 2006.

[35] Sandeep Kumar, *Elliptic Curve Cryptography for Constrained Devices*, Ph.D Thesis, Ruhr-Universitaet Bochum, Germany, June, 2006.

[36] L. Song and K.K. Parhi, *Low energy digit-serial/parallel finite field multipliers*, Journal of VLSI Signal Processing, 19(2):149-166, June, 1998.

[37] K. Wu, R. Karri, G. Kuznetsov, M. Goessel *Low Cost Concurrent Error Detection for the Advanced Encryption Standard* Proceedings of IEEE International Test Conference. (ITC 2004), 2004.

[38] Xilinx, 7 Series FPGA Overview, Advanced Product Specification, DS180 (v1.8), September, 2011.