Concurrent Error Detection in Finite Field Arithmetic Operations

by

Siavash Bayat-Sarmadi

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

With significant advances in wired and wireless technologies and also increased shrinking in the size of VLSI circuits, many devices have become very large because they need to contain several large units. This large number of gates and in turn large number of transistors causes the devices to be more prone to faults. These faults specially in sensitive and critical applications may cause serious failures and hence should be avoided.

On the other hand, some critical applications such as cryptosystems may also be prone to deliberately injected faults by malicious attackers. Some of these faults can produce erroneous results that can reveal some important secret information of the cryptosystems. Furthermore, yield factor improvement is always an important issue in VLSI design and fabrication processes. Digital systems such as cryptosystems and digital signal processors usually contain finite field operations. Therefore, error detection and correction of such operations have become an important issue recently.

In most of the work reported so far, error detection and correction are applied using redundancies in space (hardware), time, and/or information (coding theory). In this work, schemes based on these redundancies are presented to detect errors in important *finite field* arithmetic operations resulting from hardware faults. Finite fields are used in a number of practical cryptosystems and channel encoders/decoders. The schemes presented here can detect errors in arithmetic operations of finite fields represented in different bases, including polynomial, dual and/or normal basis, and implemented in various architectures, including bit-serial, bit-parallel and/or systolic arrays.

Acknowledgements

I would like to express sincere gratitude to Professor M. A. Hasan for his supervision and guidance throughout the course of my graduate studies.

The insightful review of this work by the thesis examiners, Professor Israel Korn, Professor Alfred Menezes, Professor Gord Agnew and Professor Guang Gong is also greatly appreciated.

I also would like to thank Dr. Miguel F. Anjos for letting me run part of the simulations of this work on his computer. I was also grateful for being able to use some of the computing facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET:www.sharcnet.ca).

I would like to acknowledge my family, mainly my parents, for their endless moral support and encouragement.

My special thanks goes to my beloved wife, Zohreh Mesbah-Moosavi, for her help and patience in the process of this work, without which I would not have succeeded.

Dedication

To my beloved wife, Zohreh.

To my dear parents, Azam and Hormoz.

Contents

List of Tables

List of Figures

List of Abbreviations

AOP	All-Ones Polynomial.
CED	Concurrent Error Detection.
CEC	Concurrent Error Correction.
DB	Dual Basis.
DIE	Double Input Encoding.
DIMP	Double Input Multiple Parity.
DMR	Dual Modular Redundancy.
DPC	Double Parity Checker.
ECDSA	Elliptic Curve Digital Signature Algorithm.
FIS	Fault Injection Select.
FPGA	Field Programmable Gate Arrays.
FV	Fault Value.
GF	Galois Field.
NB	Normal Basis.
NIST	National Institute of Standards and Technology.
NMOS	N-type Metal Oxide Semiconductor.
ONB1	Type I Optimal Normal Basis.
ONB2	Type II Optimal Normal Basis.
OPC	Ordinary Parity Checker.
OPV	Original Pin Value.
PB	Polynomial Basis.
PMOS	P-type Metal Oxide Semiconductor.
PPC	Parity Prediction Circuit.

P_{r_D}	Probability of Error Detection.
P_{r_U}	Probability of an Undetected Error.
RESO	REcomputing with Shifted Operands.
SIE	Single Input Encoding.
SIMP	Single Input Multiple Parity.
SM	Scalar Multiplication.
SR	Shift-and-Reduce.
VA	Vector Addition.
VHDL	VHSIC hardware description language.
VHSIC	Very-High-Speed Integrated Circuits.
VLSI	Very Large Scale Integration.

Chapter 1

Introduction

1.1 Motivation

"A fault-tolerant system is one that can continue the correct performance of its specified tasks in the presence of hardware and/or software faults. Fault detection is the process of recognizing that a fault has occurred. Fault detection is often required before any recovery procedure can be implemented" [51].

Recently a number of schemes have been developed for the detection and/or correction of errors in hardware implementation of some arithmetic operations [24, 55–57], which have applications in cryptography [9, 10, 14–16, 20, 32–36, 68, 69], deep space channel coding [66], VLSI testing [52]. The main reasons for increased interest in such schemes are as follows:

- Having correct functionality in the presence of faults: Digital systems that require large number of circuits for their implementation can be more prone to produce erroneous results simply because of the increase in the probability that one of the circuits may become faulty while in use. As a result, for

sensitive or critical applications large digital systems are generally designed with some kind of mechanism to provide correct functionality or to detect errors.

- Avoiding fault-based attacks [13, 29, 63]: Fault attacks are based on injecting some faults into a cryptosystem and observing any leak of secret information. Boneh et al. presented the first fault-based attack [15, 16]. Their attack has been applied to some public key cryptosystems such as RSA and the Rabin signature scheme. Since RSA is usually implemented using the Chinese Remainder theorem (CRT), having one correct signature and one faulty signature of the same message can lead to the modulus factorization. In [11], Biham and Shamir presented a fault-based side channel cryptanalysis of DES. They recovered the last round key by less than 200 cipher texts and then they found the round key of the second-last round and so on. They extended their work to show that it could uncover the structure of an unknown cryptosystem in a smart card. Anderson and Kuhn [3] introduced some other fault based attacks using fault injection into instruction memory of a smart card and by overwriting specific memory locations of a smart card.

One technique to detect errors in hardware implementation is on-line testing or concurrent error detection (CED). CED is used to concurrently test a system while the system is operating normally [26, 53, 60]. CED can test the circuit at full operating speed without stopping the system or switching it to test mode. Accordingly, CED can detect transient faults, which may not be detected in off-line testing, since they may not occur in test mode. Furthermore, concurrent error correction (CEC) can offer some advantages such as higher yield factors and increased availability in addition to the above mentioned advantages for CED.

To detect or correct errors, some kinds of redundancy are usually required: hardware, time, and/or information [53]. This thesis focuses mainly on the detection

and/or correction of random errors in extension field arithmetic operations. The cause of such errors can be natural faults. Moreover, some faults deliberately injected by attackers may cause random errors, e.g., some faults resulting from electro magnetic interferences (EMIs). It is worth mentioning that some random errors in a number of hardware based cryptosystems or their arithmetic accelerators can be detected at an upper level operation, e.g., in elliptic curve cryptography (ECC), if a point leaves the curve it can be easily detected by point verification [10, 20]. This is, however, not always possible. In the case of ECC, a fault may move a point to another point without leaving the curve and this has been exploited in the so-called sign change fault attack[1] [14]. As a result, some kind of mechanisms for error detection in finite field operations can be quite important in cryptography as well as other critical applications where finite field operations of various sizes are used.

Although the schemes proposed in this thesis does not provide a complete solution to the problem of deliberately injected faults, they may reduce the success probability of an attack. This is because the number of faults that can be injected by an attacker is reduced to the number of faults that cannot be detected by the scheme. Furthermore, these schemes are more suitable mainly for large finite field arithmetic and may not be very suitable for those systems that use small finite field operations such as $GF(2^8)$ operations of AES [21].

The majority of the work of this thesis is for extended binary field multipliers mainly because the complexity of multiplication is higher than the basic operations such as addition and subtraction. Also, other complex finite field arithmetic operations such as inversion and exponentiation over binary extension fields can be preformed by repeated multiplications [2, 67].

[1]Note that a random fault may move one point on the curve to another point on the curve with a very low probability [22].

1.2 Scope of this Work

In order to detect and/or correct random errors in finite field operations, a number of approaches can be considered, including:

- Using parity bits: In this approach, basically, the parity of the output is predicted and compared against its actual party. In [24], Fenn et al. presented a concurrent error detection scheme for bit-serial multipliers, using a number of bases for representation of fields, defined by an irreducible all-ones polynomial. In [55–57], parity based error detection schemes for both bit-serial and bit-parallel polynomial basis multipliers are presented. However, these schemes are not generic and can detect mostly odd number of erroneous bits and one of them can correct one bit.

- Scaling techniques: The second approach, which is also used in [27], is for example to scale the inputs of a multiplier by a factor and at the end of the multiplication the correctness of the result is checked by one or two divisions.

- Nonlinear techniques: One example for this approach [28] is to compute a nonlinear residue for each input of the operation and then predict the residue of the output using these residues. To assure the correctness of the operation, one can compare the predicted residue against the actual output residue. This approach is expensive in terms of area and time and in turn may not be very efficient for detecting random errors.

- Time redundancy based techniques: In this approach some methods such as recomputing with shifted operands are used to detect errors in operations. In [19, 41], this method is used for detecting errors in polynomial basis multipliers.

Additionally, a number of schemes for detecting errors in arithmetic operations of the symmetric block ciphers are presented in [17]. These schemes are mostly based on parity and/or residue codes.

This thesis presents a number of schemes for concurrent error detection of the arithmetic operations over binary extension fields based on some of the above-mentioned approaches. These schemes are more generic than previous schemes in the sense that they can be applied to different implementations, e.g., bit-serial, bit-parallel and/or digit-serial, and also they can be applied to different bases for the field representation such as polynomial, dual and optimal normal bases. Additionally, the schemes presented in this thesis have high error detection capability, e.g., based on our simulations, the majority of the schemes have a percentage of error detection higher than 99% with a moderate amount of redundancies.

1.3 Thesis Outline

The organization of the remainder of the proposal is as follows. A brief overview of required background is presented in Chapter 2.

Chapter 3 presents a number of schemes for detecting errors concurrently in polynomial, dual and normal bases arithmetic operations. These schemes are based on recomputing with shifted operands technique and are efficient for pipelined architectures such as systolic arrays. To investigate more on this scheme, one finite field semi-systolic multiplier is presented for each of the polynomial, dual, type I and type II optimal normal bases. Then the CED scheme is applied to them. Additionally, the space and time complexity of these multipliers are compared against a number of systolic and/or semi-systolic multipliers previously published in the literature. Furthermore, the capability of error detection of each multiplier is evaluated by simulation-based fault injection. The results show that having better or

similar space and time overheads compared to a number of related previous work, the multipliers have generally a high error detection capability, e.g., the percentage of error detection of the scheme for the single and multiple stuck-at faults in a polynomial basis multiplier is 100%. Finally, we also comment on how RESO can be used for concurrent error correction to deal with transient faults.

In Chapter 4, one parity based scheme to detect multiple-bit errors in polynomial basis multipliers is presented. In this scheme one input of the multiplier is divided into a number of parts and a parity bit is considered for each part. To obtain a realistic area and time overheads of the scheme, it is implemented in bit-serial and bit-parallel fashions on FPGAs. Also, the capability of the scheme is investigated using a theoretical analysis as well as a simulation-based fault injection. Having a high error detection capability, this scheme produces area and time overheads lower than dual modular redundant systems.

In Chapter 5, the parity based scheme presented in the previous chapter is extended to both inputs of the polynomial basis multiplier. This is because the errors on the second input cannot be detected, although, the error detection capability of the scheme, presented in previous chapter, is high, e.g., the percentage of error detection is 99.61% for eight partitions on the first input. This scheme is also implemented on FPGAs to determine the area and time overheads of the scheme. Moreover, the error detection capability of the scheme is evaluated by simulation-based fault injection. This scheme has slightly better percentage of error detection and slightly more area overhead compared to the previous scheme. In the second part of this chapter, both parity based schemes are extended to other bases, including dual, type I and type II normal bases.

In Chapter 6, a number of schemes based on linear codes are presented to detect errors in polynomial basis multipliers. In the first scheme one input of the multiplier is encoded by a generator polynomial. The correctness of the result is checked by

decoding of the final and/or intermediate results of the multiplication. To resolve the problem of error detection on the second input of the multiplier, a second scheme is presented. In this scheme, both inputs of the multiplier are encoded by two different generator polynomials. Another scheme that combines the parity based scheme for the first input and the linear code based scheme for the second input is presented in this chapter. This scheme basically has a lower area overhead compared to the second scheme presented in this chapter, however, their error detection capabilities are almost the same. These scheme are also implemented on FPGAs and their error detection capability are evaluated by simulation-based fault injection.

Finally, summary and conclusions of the thesis as well as directions for further research are given in Chapter 7.

1.4 Research Contributions

Achieving an acceptable security level for hardware implementations of large digital systems specially with critical applications has received significant attention recently. Some of these digital systems use finite field arithmetic operations such as extension field multipliers. The major contributions of this thesis are the development of some schemes to concurrently detect and/or correct errors in such operations. Some specific contributions of the thesis are as follows:

- Concurrent error detection in binary extension field operations suitable for pipelined architectures and systolic arrays. These schemes use RESO (RE-computing with Shifted Operands) method for finite fields represented in polynomial, dual and optimal normal bases.

- Concurrent error detection in binary extension field multipliers for general bit-serial, digit-serial and/or bit-parallel implementations.

- Two of the schemes, SIMP and DIMP, are based on the parity codes and are for the finite fields represented using polynomial, dual and optimal normal bases.
- Three other schemes, SIE, DIE and hybrid, are based on the linear codes (scaling technique) and are for the finite fields represented using polynomial basis.

Chapter 2

Background

In this chapter, first finite fields are reviewed, secondly a number of arithmetic operations particularly multiplication algorithms are explained, and finally a brief introduction to fault tolerant systems is given.

2.1 Finite Fields

In this section, proofs of theorems and lemmas are omitted for brevity. For interested readers there are many good texts on algebra and finite fields such as [12, 30, 31, 42, 43, 46, 58].

Basic Definitions and Properties

Definition 2.1 A group is a set $\mathbb{G}$ with a binary operation '*' on $\mathbb{G}$ if it satisfies the following conditions:

- The binary operation '*' is associative, i.e.,

$$\forall a, b, c \in \mathbb{G} \quad a * (b * c) = (a * b) * c$$

- There is an identity element e in $\mathbb{G}$ such that,

$$\forall a \in \mathbb{G} \quad a * e = e * a = a$$

- There is an inverse element for each element such that,

$$\forall a \in \mathbb{G}, \exists a^{-1} \in \mathbb{G} \quad a * a^{-1} = a^{-1} * a = e$$

The identity element of a group and the inverse of each element of a group are unique. Furthermore, a group is an Abelian or communicative group if '$*$' also satisfies the following condition:

$$\forall a, b \in \mathbb{G} \quad a * b = b * a$$

Definition 2.2 A set $\mathbb{F}$ with two operations denoted by '+' and '.' is a field if it satisfies the following conditions:

- $(\mathbb{F}, +)$ is an Abelian group and 0 is its identity element.
- $(\mathbb{F}^*, .)$ is an Abelian group and 1 is its identity element where $\mathbb{F}^*$ is the set of nonzero elements in $\mathbb{F}$.
- '.' is distributive over '+', i.e., $\forall a, b, c \in \mathbb{F}$

$$a.(b + c) = a.b + a.c$$

and

$$(b + c).a = b.a + c.a$$

Definition 2.3 A set $\mathbb{R}$ with two operations denoted by '+' and '.' is a ring if it satisfies all conditions of a field except the condition that each element should have

a multiplicative inverse. In other words, the elements of a ring may not have a multiplicative inverse.

Definition 2.4 A field that contains a finite number of elements is called a finite field (also known as Galois field).

Definition 2.5 The number of elements in a Galois field is called the order of the field and a Galois field with the order of q is denoted by $GF(q)$.

Definition 2.6 Let a be an element in $GF(q)$. Then the smallest positive integer m is called characteristic of the field such that $ma = 0$.

Theorem 2.1 *The characteristic of any finite field is prime.*

Theorem 2.2 *In a Galois field, the order of the field is a prime or a power of a prime.*

Lemma 2.1 *Characteristic of* $GF(q) = \begin{cases} q & \text{if } q \text{ is prime;} \\ p & \text{if } q \text{ is a power of a prime } p. \end{cases}$

Definition 2.7 The order of a nonzero element $a \in GF(q)$ is the smallest positive integer n such that,

$$a^n \equiv 1$$

Theorem 2.3 *Let a be a nonzero element of $GF(q)$. Then $a^{q-1} \equiv 1$.*

Using Theorem 2, the inverse of any nonzero element a could be computed by $a^{-1} \equiv a^{q-2}$.

Definition 2.8 In a finite field $GF(q)$, a nonzero element a is primitive if its order is $q - 1$.

Definition 2.9 V is a vector space over F with multiplication operation '.': $F \times V \rightarrow V$ if for all $a, b \in F$ and $v, w \in V$:

1. $(V, +)$ is a commutative group
2. $a(v + w) = av + aw$
3. $(a + b)v = av + bv$
4. $(ab)v = a(bv)$
5. $1v = v$

Elements of F and V are called scalars and vectors, respectively. '+' is called vector addition and '.' is called scalar multiplication.

Polynomials

A polynomial over $GF(p)$ is an expression of the following form:

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

where $a_i \in GF(p)$ and n, the degree of $F(x)$, is a nonnegative integer. Polynomial $F(x)$ is called monic if $a_n = 1$.

Definition 2.10 A polynomial $F(x)$ over $GF(p)$ is said to be irreducible if it cannot be written as the product of some lower degree polynomials over $GF(p)$.

Definition 2.11 Suppose a polynomial $F(x)$ over $GF(p)$ which $F(0) \neq 0$. The order (or period) of $F(x)$ is the least positive integer t such that $F(x)|x^t - 1$.

Definition 2.12 Let $F(x)$ be a polynomial of degree m over $GF(p)$. Polynomial $F(x)$ is said to be a primitive polynomial if its order is $p^m - 1$.

Definition 2.13 A polynomial $F(x)$ is said to be an all-ones polynomial (AOP) if all of its coefficients are one.

Construction and Representation of $GF(p^m)$

To construct $GF(p^m)$ the following theorem can be applied.

Theorem 2.4 *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 referred to as the field defining polynomial.

Definition 2.14 Let F and G be two fields and $F \subset G$. Then F is called a subfield of G and G is called an extension field of F.

For example, $GF(p^m)$ is an extension field of $GF(p)$.

$GF(p^m)$ is a vector space of dimension m over $GF(p)$. Thus, the basis of vector space $GF(p^m)$ could be any set of m linearly independent elements, e.g., $\{x_0, x_1, \ldots x_{m-1}\}$. A linear combination of these elements is as follows:

$$\mathcal{A} = a_0x_0 + a_1x_1 + \ldots a_{m-1}x_{m-1}$$

where $a_i \in GF(p)$.

There are some well-known bases for representation of extension fields such as canonical (polynomial) basis, normal basis, dual basis and triangular basis. Below we introduce the first three bases.

Definition 2.15 Let x be the root of an irreducible polynomial $F(x)$ over $GF(p)$ of degree m, i.e., $F(x) = 0$. Canonical (polynomial) basis is defined as the following set:

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

Hereafter, this basis is called polynomial basis (PB).

Definition 2.16 Let $a \in GF(p^m)$. The trace of a is:

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

The dual basis (DB) of the polynomial basis $\{1, x, x^2, \ldots x^{m-1}\}$ is a basis $\{y_0, y_1, \cdots, y_{m-1}\}$, where $y_i \in GF(2^m)$, such that

$$\mathrm{Tr}(y_i x^j) = \begin{cases} 0, & i \neq j; \\ 1, & \text{otherwise.} \end{cases} \tag{2.1}$$

Definition 2.17 A normal basis (NB) of $GF(p^m)$ has the following form:

$$\{z, z^p, z^{p^2}, \ldots z^{p^{m-1}}\}$$

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

2.2 Field Arithmetic Operations

In this section, arithmetic operations except addition and subtraction which are considered to be trivial are briefly explained. Two arithmetic operations, i.e., multiplication and inversion over extension fields are discussed in the two following sections. It will be shown that the arithmetic operations such as inversion, division, exponentiation can be computed using repeated multiplications. We will use polynomial basis for representation of the elements of the binary extension field $GF(2^m)$.

2.2.1 Multiplication over $GF(2^m)$

General Bit-level Multiplications

Suppose that $A, B \in GF(2^m)$ and $F(x)$ is the modulus (defining polynomial of the field). Thus, given bits $a_i, b_i \in GF(2)$, we have:

$$\begin{aligned} A &= \sum_{i=0}^{m-1} a_i x^i, \quad B = \sum_{i=0}^{m-1} b_i x^i \\ C &= AB \mod F(x) = A \sum_{i=0}^{m-1} b_i x^i \mod F(x) \\ &= \sum_{i=0}^{m-1} A b_i x^i \mod F(x) \\ &= (b_0 A + b_1 x A + \cdots + b_{m-1} x^{m-1} A) \mod F(x) \end{aligned} \tag{2.2}$$

According to (2.2), bit-level algorithm of multiplication from low bit to high bit is given in Algorithm 2.1.

Algorithm 2.1 Low-to-high bit-level multiplication in $GF(2^m)$ [31]

Input: A, B, $F(x)$
Output: $C = AB \mod F(x)$

```
D := A
C := 0
For i = 0 to m − 1 do {
    C := C + D.b_i
    D := xD mod F(x)
}
```

Equation (2.2) can also be written as follows:

$$\begin{aligned} C &= AB \mod F(x) \\ &= A(b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_0) \mod F(x) \\ &= (\ldots(Ab_{m-1}x + b_{m-2})x + \ldots)x + Ab_0 \mod F(x) \\ &= (\ldots(Ab_{m-1}x \mod F(x) + b_{m-2})x \mod F(x) + \ldots)x \mod F(x) + Ab_0 \end{aligned} \tag{2.3}$$

Based on (2.3), an algorithm for bit-level multiplication from high bit to low bit of element B is given in Algorithm 2.2.

Algorithm 2.2 High-to-low bit-level multiplication in $GF(2^m)$ [31]

Input: A, B, $F(x)$
Output: $C = AB \mod F(x)$

```
C := A.b_{m-1}
For i = m − 2 to 0 do {
    C := xC mod F(x)
    C := C + A.b_i
}
```

An advantage of the second algorithm is that D, which is a register in hardware implementation, is not needed.

Bit-Serial Multiplications

Let $A \in GF(2^m)$ and $F(x)$ be the defining polynomial. $X = xA$ can be computed as:

$$\begin{aligned} X = xA &= x\sum_{i=0}^{m-1} a_i x^i \mod F(x) \\ &= \sum_{i=0}^{m-1} a_i x^{i+1} \mod F(x) \\ &= \sum_{i=0}^{m-2} a_i x^{i+1} + a_{m-1}x^m \mod F(x) \\ &= \sum_{i=0}^{m-2} a_i x^{i+1} + a_{m-1}\sum_{i=0}^{m-1} f_i x^i \end{aligned} \tag{2.4}$$

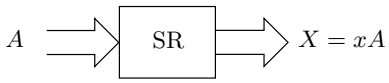


Figure 2.1: SR module

Figure 2.1 shows a module whose input and output are A and X, respectively. This module basically performs a shift operation and a reduction operation. Hence, it is hereafter referred to as Shift-and-Reduce or SR module.

Multiplication of $A, B \in GF(2^m)$ can be written as:

$$\begin{aligned} C = AB &= A(b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_0) \\ &= (b_{m-1}x^{m-1}A + b_{m-2}x^{m-2}A + \cdots + b_0 A) \\ &= (b_{m-1}A^{(m-1)} + b_{m-2}A^{(m-2)} + \cdots + b_0 A^{(0)}) \end{aligned} \tag{2.5}$$

where $A^{(0)} = A$ and $A^{(i)} = xA^{(i-1)}$.

Since $GF(2^m)$ is a vector space, in (2.5), '.' is a scalar multiplication and '+' is a vector addition [see Definition 2.9]. Additionally, bit b_i is the i^{th} coordinate of B. Figure 2.2 shows a low-to-high bit-serial multiplier where D is initialized with A. Scalar multiplication and vector addition of a bit-serial multiplier are bitwise-AND

and bitwise-XOR, respectively. Modules that perform scalar multiplication and vector addition are hereafter referred to as SM module and VA module, respectively. These two modules and the SR module discussed earlier are the main components of a PB multiplier. In accordance with (2.5) and using these three main components, a bit-serial PB multiplier can be constructed as shown in Figure 2.2 (see [61] for a similar multiplier architecture).

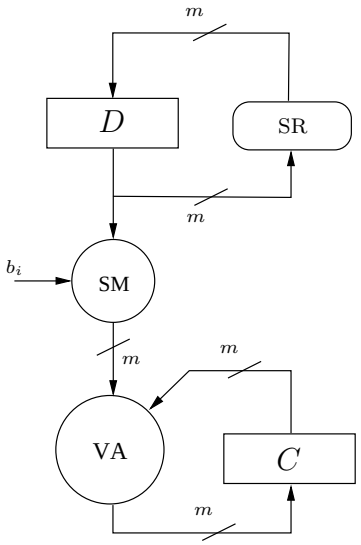


Figure 2.2: Low-to-high bit-serial multiplication

Bit-parallel Multiplications

According to (2.5), we can implement multiplication in a bit-parallel fashion. Figure 2.3 shows a bit-parallel multiplication. A bit-parallel multiplication needs $(m-1)$ SR modules, m scalar multiplications and $(m-1)$ vector additions.

Digit-Serial Multiplications

Another method to implement a PB multiplier is to combine the above-mentioned implementations. In other words, instead of computing the multiplication of input

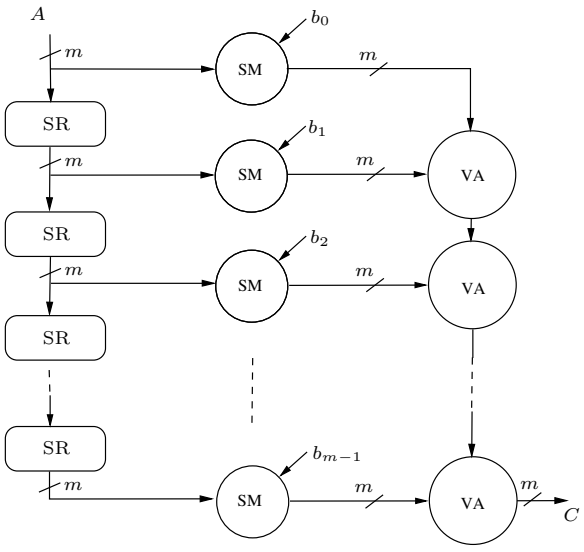


Figure 2.3: Bit-parallel multiplication

A by one bit of input B per clock cycle (like bit-serial implementations), one can multiply A by a number of bits of B (i.e., a digit of B). This implementation is referred to as digit-serial implementation.

2.2.2 Inversion over $GF(2^m)$

There are several algorithms for inversion [31] which are based on one of the following methods.

1. Repeated squaring-and-multiplications in extension field $GF(2^m)$
2. Use of extended Euclidean algorithm over some subfield of $GF(2)$
3. Solution of a system of linear equations over some subfield of $GF(2)$

We consider the first two methods. The former is based on Theorem 2.3, such that $\forall a \in GF(q = 2^m)$:

$$a^{q-1} = 1$$

$$a^{-1} = a^{q-2}$$

The latter is discussed below.

Extended Euclidean Algorithm (EEA) Based Inversion

Let $A(x) \in GF(2^m)$ and $F(x)$ be the defining polynomial of the field. We want to find $B(x)$ such that,

$$A(x)B(x) = 1 \mod F(x).$$

Since $\gcd(F(x), A(x)) = 1$, $B(x)$ can be obtained using EEA (Algorithm 2.3):

$$A(x)B(x) + F(x)C(x) = 1$$

Algorithm 2.3 Extended Euclidean algorithm [31]

Input: $F(x)$ and $A(x) \neq 0$
Output: B(x)
Step1:
$R^{(-1)}(x) = F(x), R^{(0)}(x) = A(x)$
$U^{(-1)}(x) = 0, U^{(0)}(x) = 1$
$i = 0$
Step2:
do{
$i = i + 1$
$Q^{(i)}(x) = \lfloor R^{(i-2)}(x)/R^{(i-1)}(x) \rfloor$
$R^{(i)}(x) = R^{(i-2)}(x) - Q^{(i)}(x)R^{(i-1)}(x)$
$U^{(i)}(x) = U^{(i-2)}(x) - Q^{(i)}(x)U^{(i-1)}(x)$
}while $(R^{(i)}(x) \neq 0)$
Step3:
$B(x) = U^{(i-1)}(x)$

It is worth mentioning that in addition to inversion, two other arithmetic operations, i.e., division and exponentiation, can be performed using repeated multiplications because:

$$\frac{A}{B} = A \times B^{-1},$$
$$A^n = \underbrace{A \times A \cdots \times A}_{n},$$

where n is a non-negative integer.

2.3 Fault Tolerant Systems

2.3.1 Faults

Faults can be investigated at different levels, including gate-level and architecture-level. Below we give brief descriptions of gate and architecture level faults.

Gate-level faults can be categorized as open faults, short (bridging) faults, and stuck-at faults. An example for an open fault is a disconnected wire. The disconnected wire may keep its previous value. Figure 2.4 shows a circuit with an open fault which the gate output is disconnected. Assuming that the value of the circuit before disconnection was 1, the correct output (c) and the faulty output (c') for different values of the gate inputs are presented in Table 2.1. Clearly, the error can be modelled by a bit-flip ($e_c = c \oplus c'$) in a higher level of abstraction such as architecture-level .

Figure 2.4: Open fault example

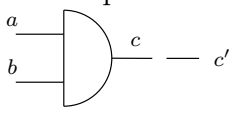


Table 2.1: Open fault table

a	b	c	c'	e_c
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Similarly, Figure 2.5 shows two circuits, which one of them is fault free and the other one has two stuck-at faults. Table 2.2 presents the expected correct output values (d and f), the faulty output values (d' and f') and the multiple-bit flip errors that model the stuck-at faults (e_d and e_f).

Figure 2.5: Stuck-at fault example

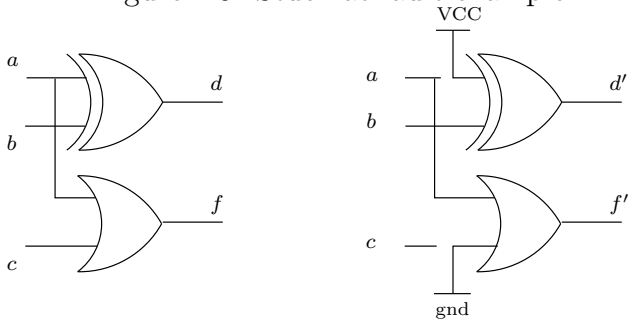


Table 2.2: Stuck-at fault table

a	b	c	d	f	d'	f'	e_d	e_f
0	0	0	0	0	1	0	1	0
0	0	1	0	1	1	0	1	1
0	1	0	1	0	0	0	1	0
0	1	1	1	1	0	0	1	1
1	0	0	1	1	1	1	0	0
1	0	1	1	1	1	1	0	0
1	1	0	0	1	0	1	0	0
1	1	1	0	1	0	1	0	0

2.3.2 Fault Tolerant Techniques

For having fault tolerance or detection capability, one can use some forms of redundancies. There are three common types of redundancies [51] used in practice: 1) Hardware redundancy, 2) Time redundancy and 3) Information redundancy.

An example for hardware redundancy is a dual modular redundant (DMR) system. In this system, two identical modules are functioning simultaneously and their results are compared. Any difference between the results indicates an error. A DMR has (more than) 100% hardware redundancy. On the other hand, one system with one module can perform the computations twice and then compare the results. This system has 100% time redundancy and clearly some hardware redundancies for storing the intermediate result and comparing the results. In the following section, some forms of information redundancies are briefly explained. For further information, references [1, 37, 42, 51] are recommended.

2.3.3 Information Redundancy

Definition 2.18 A code is a set of rules by which some information or data is represented.

For example, a code can be 4-bit representation of each digit of a number which is called binary coded decimal (BCD).

Definition 2.19 A collection of bits which is representing some information or data using a code is said to be a codeword. The bits are called digits if they are numbers. A binary code contains only 0 and 1 digits which are called bits.

Definition 2.20 A codeword is valid if it satisfies all of the rules of the code.

For example, '1001' is a valid BCD , but '1100' is invalid. In the following a number of special codes are reviewed.

Parity Codes

The simplest form of a code is the parity code.

Definition 2.21 Let $A = (a_{k-1}, a_{k-2}, \cdots, a_1, a_0)$ be a k-bit message. The single-bit parity code of A is $A_C = (a_c, a_{k-1}, a_{k-2}, \cdots, a_1, a_0)$ where a_c can be obtained as follows:

$$\begin{cases} a_c = \sum_{i=0}^{n-1} a_i \mod 2; & \text{even parity} \\ a_c = \left(1 + \sum_{i=0}^{n-1} a_i\right) \mod 2; & \text{odd parity} \end{cases} \tag{2.6}$$

For a hardware implementation of the above even (or odd) parity generator, k (or $k+1$) XOR gates are needed[1].

The single-bit parity code can detect any single-bit error but it cannot correct it. Moreover, such codes can detect any odd number of erroneous bits, since they change an odd parity codeword to an even parity one and vice versa. Hereafter, parity is used instead of even parity for brevity.

Arithmetic Codes

One useful code especially for arithmetic operations are arithmetic codes [51]. The data are encoded before the operations are performed. The result should be a valid codeword, otherwise, an error is signalled.

Let b and c be the data and '*' be the operation and $A()$ be the encoding function. Then the arithmetic code must be invariant to the operation, i.e.,

$$A(b * c) = A(b) * A(c)$$

.

- The simplest arithmetic code is called AN code. It is constructed by multiplying a constant A by a data word N. AN codes are invariant to addition

[1]Precisely, for the odd parity generator, k XOR gates and one NOT gate are needed.

and subtraction, but, they are variant to multiplication. In case of binary codes, A should not be a power of 2. Let $N = (a_{n-1}, a_{n-2}, \cdots, a_1, a_0)$ be multiplied by $A = 2^i$. AN is:

$$(a_{n-1}, a_{n-2}, \cdots, a_1, a_0, 0, \cdots, 0)$$

where i zeroes appended to original N. Let any bit before the appended zeroes, i.e., a_i's, be flipped. Then the code cannot detect the fault, since it is still divisible by 2^a.

One of the well-known AN codes is $3N$ code which can be efficiently constructed by addition of $2N$ and N. It increases the size of original message by 2 bits.

- Residue code is another type of arithmetic codes. For constructing a residue codeword, the original information is divided by a constant which is called modulus. Then the remainder, which is called the residue, is appended to the original information. This code is invariant to modular addition.

  Let the size of modulus and original information be m bits and n bits, respectively. The size of the residue codeword is $m + n$ bits.

- Inverse residue code is a modification of the residue code. Inverse residue is calculated as $M - R$, where M is the modulus and R is the residue of original information. The inverse residue codeword is constructed by appending inverse residue to the original information.

  The advantage of such codes is their better tolerance in presence of repeated-use faults. A repeated-use fault is one that is encountered multiple times before the code is checked, since the hardware is used multiple times before checking [51].

Duplication Codes

In duplication codes, a codeword consists of completely duplicating of original information. The detection capability of such codes are the best. Since the information are assumed to be fault free if both parts are the same. The main disadvantage of such codes is its 100% informational overhead. But in many applications such as data transmission or memory, in addition to 100% informational overhead, there is a 100% redundancy in hardware and/or time. A variation of a basic duplication code is to complement the duplicated portion of the codeword. This is a major advantage when the original information and its duplicate must pass through the same way or must be processed by the same hardware.

Chapter 3

Concurrent Error Detection Using RESO

This chapter presents a number of schemes for detecting errors concurrently in polynomial, dual and normal bases arithmetic operations. The schemes presented in this chapter are based on recomputing with shifted operands (RESO) technique and are efficient for pipelined architectures such as systolic arrays. To investigate more on this scheme, one finite field semi-systolic multiplier is presented for each of the polynomial, dual, type I and type II optimal normal bases. Then the CED scheme is applied to them. Additionally, the space and time complexities of these multipliers are compared against a number of systolic and/or semi-systolic multipliers previously published in the literature. Furthermore, the capability of error detection of each multiplier is evaluated by simulation-based fault injection. The results show that having better or similar space and time overheads compared to a number of related previous work, the multipliers have generally a high error detection capability, e.g., the percentage of error detection of the scheme for the single and multiple stuck-at faults in a polynomial basis multiplier is 100%. Finally, we also comment on how RESO can be used for concurrent error correction to deal

with transient faults.

The organization of the remainder of this chapter is as follows. In Section 3.1, the RESO method is reviewed. The concurrent error detection strategy is presented in Section 3.2. General pipelined architectures, which are suitable for these schemes, along with an overhead analysis are given in Section 3.3. In Section 3.4, the CED scheme is investigated with more details for polynomial, dual and normal bases multipliers. The error detection capability of the scheme is then evaluated in Section 3.5. In Section 3.6, some comments on the concurrent error correction strategy are given. Finally, Section 3.7 gives a summary of the chapter.

This work also appeared in [5].

3.1 RESO Method

REcomputing with Shifted Operands (RESO) is a technique for concurrent error detection (CED) in arithmetic and logic units introduced by Patel and Fung in [48, 49]. This technique is based on time redundancy. Suppose x and $f(x)$ are the input and output of a computation unit f, respectively. Also, suppose E and D are two functions such that $D(f(E(x))) = f(x)$. Now, we store the result of the computation of $f(x)$ (first step) in a register and compare it with the result of the computation of $D(f(E(x)))$ (second step). Any difference between results of these two steps indicates an error. The functions E and D are referred to as encoding and decoding functions, respectively, and they can be usually chosen such that $D = E^{-1}$. It is worth mentioning that for conventional binary operands, E and D are simple shifts of operand bits and this is why it is referred to as RESO.

3.2 Concurrent Error Detection Strategy

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

In this section, we use RESO method to concurrently detect errors in arithmetic operations over the field $GF(2^m)$. For the polynomial, normal and dual bases, the encoding and decoding functions are chosen in a way that the overhead costs (in terms of area and time) are fairly low. Additionally, in this chapter, the arithmetic operations addition/subtraction, multiplication, inversion, division, and exponentiation are considered. Figure 3.1 shows a general architecture of an operation with concurrent error detection. In the figure, two encoding functions of the inputs are E_1 and E_2 and the decoding function of the output is D. Clearly, for inversion the second input should not be considered. Also, for exponentiation the exponent is a non-negative integer number and is not an extension field element. Therefore, this input of exponentiation is not considered as well.

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

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

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

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

As a result,

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

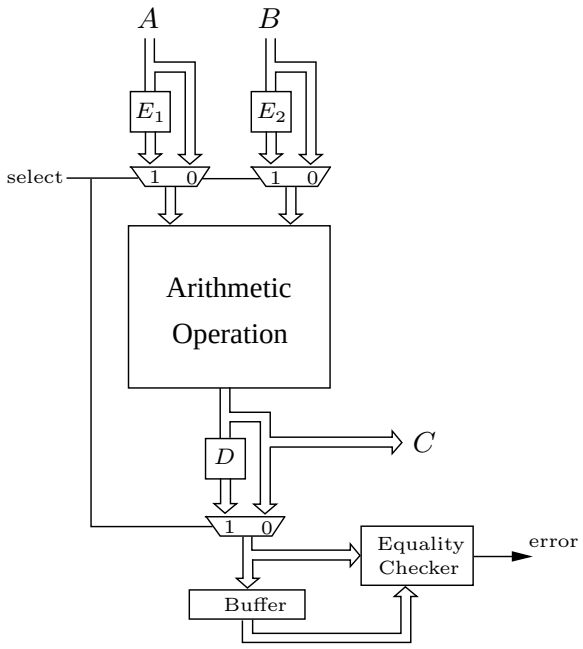


Figure 3.1: General architecture for the arithmetic operations with CED

In the following, the above-mentioned concurrent error detection strategy for each basis is investigated.

3.2.1 CED for Polynomial Basis (PB) Arithmetic Operations

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

follows:

$$\begin{aligned} x^m &= 1 + \sum_{i=1}^{m-1} f_i x^i \mod F(x), \\ x^{-1} &= x^{m-1} + \sum_{i=0}^{m-2} f_{i+1} x^i \mod F(x). \end{aligned} \tag{3.1}$$

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

$$\begin{aligned} xA &= \sum_{i=0}^{m-1} a_i x^{i+1} \mod F(x) = \sum_{i=0}^{m-2} a_i x^{i+1} + a_{m-1} x^m \mod F(x) \\ &= (0, a_0, \cdots, a_{m-3}, a_{m-2}) + a_{m-1}(1, f_1, \cdots, f_{m-2}, f_{m-1}). \end{aligned} \tag{3.2}$$

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

$$\begin{aligned} x^{-1}A &= \sum_{i=0}^{m-1} a_i x^{i-1} \mod F(x) = a_0 x^{-1} + \sum_{i=1}^{m-1} a_i x^{i-1} \mod F(x) \\ &= (a_1, a_2, \cdots, a_{m-1}, 0) + a_0(f_1, f_2, \cdots, f_{m-1}, 1). \end{aligned} \tag{3.3}$$

Hereafter, the former and the latter are referred to as (forward) scaling and inverse scaling, respectively. Additionally, both scalings are very inexpensive in hardware implementations. An overhead analysis will be given in Section 3.3.1.

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

1. Addition/Subtraction: $E_1 = x$, $E_2 = x$, $D = x^{-1}$.

   (a) Compute $A + B = C$; Store in a register;

(b) Compute $Ax + Bx = (A + B)x = Cx$; Inverse scaling; Compare this result with that of (a).

2. Multiplication: $E_1 = x$, $E_2 = x$, $D = x^{-2}$.

   (a) Compute $A \times B = C$; Store in a register;

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

3. Inversion: $E_1 = x$, $D = x$.

   (a) Compute $\frac{1}{A} = C$; Store in a register;

   (b) Compute $\frac{1}{Ax} = (\frac{1}{A})x^{-1} = Cx^{-1}$; Forward scaling; Compare this result with that of (a).

4. Division: $E_1 = x$, $E_2 = x^{-1}$, $D = x^{-2}$.

   (a) Compute $\frac{A}{B} = C$; Store in a register;

   (b) Compute $\frac{Ax}{Bx^{-1}} = \frac{A}{B}x^2 = Cx^2$; Two inverse scalings; Compare this result with that of (a).

5. Exponentiation: $E_1 = x$, $D = x^{-n}$, where n is a non-negative integer.

   (a) Compute $A^n = C$; Store in a register;

   (b) Compute $(Ax)^n = A^n x^n = Cx^n$; n inverse scaling; Compare this result with that of (a).

For large n, to speed up the exponentiation, one can pre-compute and store x^{-n} for $1 \leq n \leq 2^m - 1$ in some fault-tolerant manner.

Alternative encodings and decodings for multiplication and division are as follows:

- Multiplication: $E_1 = x$, $E_2 = x^{-1}$, No decoding.

  1. Compute $A \times B = C$; Store in a register;
  2. Compute $Ax \times Bx^{-1} = (A \times B) = C$; Compare this result with that of (a).

- Division: $E_1 = x$, $E_2 = x$, No decoding.

  1. Compute $\frac{A}{B} = C$; Store in a register;
  2. Compute $\frac{Ax}{Bx} = \frac{A}{B} = C$; Compare this result with that of (a).

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

3.2.2 CED for Dual Basis (DB) Arithmetic Operations

Similar to PB arithmetic operations, a suitable candidate for encoding and decoding functions in DB representation of the elements of the field is multiplication of an element by x or x^{-1}. This multiplication is considered in Lemma 3.1.

Lemma 3.1 *Let $A = (a'_0, a'_1, \cdots, a'_{m-1}) \in GF(2^m)$ be represented in dual basis. Let $F(x) = \sum_{i=0}^{m} f_i x^i$ be the field defining polynomial. Then the (forward) scaling and inverse scaling can be performed as follows:*

$$\begin{aligned} xA &= (a'_1, a'_2, \cdots, a'_{m-1}, \sum_{i=0}^{m-1} f_i a'_i) \\ x^{-1}A &= (\sum_{i=0}^{m-1} f_{i+1} a'_i, a'_0, a'_1, \cdots, a'_{m-2}) \end{aligned} \tag{3.4}$$

Proof The proof for forward scaling can be found in [65]. Similarly, the inverse scaling can be proved as follows. Assume that the PB representation of A is $(a_0, a_1, \cdots, a_{m-1})$. Then according to Section 2.1, we have:

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

Moreover, according to [65], we have:

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

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

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

Also,

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

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

Note that for low weight $F(x)$, the hardware implementation of DB scalings requires only a few gates (see Section 3.3.2). Moreover, functions E_1, E_2 (if applicable), and D can be chosen same as those chosen for PB representation in Section 3.2.1 and similar CED procedure can be performed.

3.2.3 CED for Normal Basis (NB) Arithmetic Operations

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

$$\begin{aligned} A^2 &= \sum_{i=0}^{m-1} \left(\hat{a}_i \alpha^{2^i}\right)^2 = \sum_{i=0}^{m-1} \hat{a}_i \alpha^{2^{i+1}} \\ &= (\hat{a}_{m-1}, \hat{a}_0, \hat{a}_1, \cdots, \hat{a}_{m-2}). \\ A^{\frac{1}{2}} &= \sum_{i=0}^{m-1} \left(\hat{a}_i \alpha^{2^i}\right)^{\frac{1}{2}} = \sum_{i=0}^{m-1} \hat{a}_i \alpha^{2^{i-1}} \\ &= (\hat{a}_1, \cdots, \hat{a}_{m-2}, \hat{a}_{m-1}, \hat{a}_0). \end{aligned} \tag{3.5}$$

The hardware implementations of NB squaring and taking the square root have no cost (see Section 3.3.3). Therefore, in NB arithmetic operations, proper choices for encoding and decoding functions are squaring and taking the square root. Moreover, the procedures of CED in NB arithmetic operations are more uniform since the encoding function(s) and the decoding function are always squaring and taking the square root, respectively. The CED procedures follow, assuming that $A, B, C \in GF(2^m)$ and n is a non-negative integer.

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

3. Inversion:

   (a) Compute $\frac{1}{A} = C$; Store in a register;

   (b) Compute $\frac{1}{A^2} = (\frac{1}{A})^2 = C^2$; Take square root; Compare this result with that of (a).

4. Division:

   (a) Compute $\frac{A}{B} = C$; Store in a register;

   (b) Compute $\frac{A^2}{B^2} = \left(\frac{A}{B}\right)^2 = C^2$; Take square root; Compare this result with that of (a).

5. Exponentiation:

   (a) Compute $A^n = C$; Store in a register;

   (b) Compute $(A^2)^n = (A^n)^2 = C^2$; Take square root; Compare this result with that of (a).

3.3 Pipeline Architecture and Overhead Analysis

The proposed CED scheme is based on time redundancy. A straightforward implementation causes more than 100% time redundancy which may not be desirable. An efficient architecture that can reduce the time overhead significantly is a pipeline architecture. Additionally, this architecture has a moderate area overhead. An example for the pipelined architecture is the systolic array, which is used for high performance arithmetic operations. As shown in Figure 3.2(a), one buffer is added to the end of the pipeline architecture to store the result of the (first) computation of the arithmetic operation. Then the result of the second computation after decoding will be compared against the content of the last buffer of the pipeline. Another possibility is to start performing the operation with encoded inputs first

and then perform the normal computation. In this case, a decoder should be placed after the added buffer as shown in Figure 3.2(b).

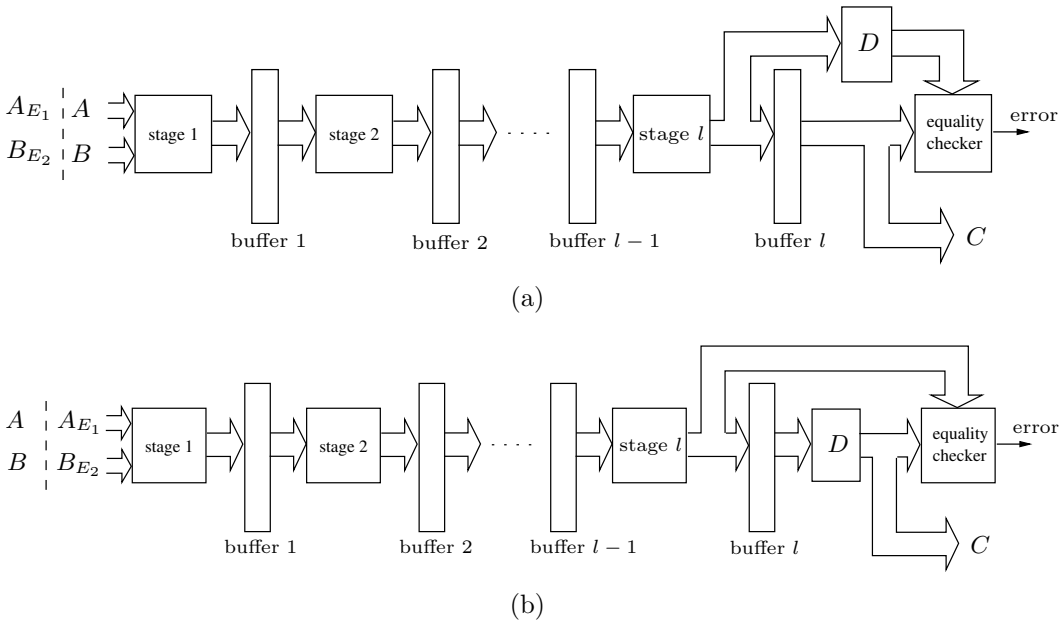


Figure 3.2: General pipelined architecture of an arithmetic operation with the proposed CED

Suppose that the number of pipeline stages is l. Let the propagation delays of the encoding function, the decoding function, the i^{th} stage of the pipeline (including a buffer), the buffer, the equality checker, and one XOR gate be t_e, t_d, t_i, t_b, t_c, and t_X, respectively. Let t'_{clk} and t_{clk} be the clock period of the pipelined arithmetic operation with and without CED, respectively. Clearly, $t_{clk} \geq \text{Max}\{t_i\}$ for $1 \leq i \leq l$. Also, in practice, $t_{clk} \geq t_X$. For each pipeline architecture of Figure 3.2, t'_{clk}, the clock period and latency overheads are given in Table 3.1. One can choose one of the above-mentioned architectures which has a smaller latency overhead.

It is worth mentioning that in some pipeline architectures such as systolic arrays, the delay of the equality checker (t_c) can be larger than other delays mentioned in

Architecture	t'_{clk}	Clock period overhead (Δ_t)	Latency overhead (Γ_t)
(a)	$\geq$ Max $\{t_e+t_1, \text{Max}\{t_i\}, t_l-t_b+t_d+t_c\}$	$t'_{clk} - t_{clk}$	$l \times \Delta_{t_{(a)}} + t'_{clk}$
(b)	$\geq$ Max $\{t_e+t_1, \text{Max}\{t_i\}, t_l-t_b+t_c, t_d+t_c\}$	$t'_{clk} - t_{clk}$	$l \times \Delta_{t_{(b)}} + t'_{clk}$

Table 3.1: The time overheads for the different pipelined architectures

the second column of Table 3.1 under t'_{clk}. In this case, one may be able to reduce t_c using a suitable method such as pipelining the checker. This will be addressed with more details in Section 3.4.4.

3.3.1 Overheads in PB Operations

The hardware implementations of the PB scaling or inverse scaling are very inexpensive since they need a cyclic shift to the right or left, which is free of cost in hardware, and $\omega - 2$ XOR gates, where ω is the Hamming weight of $F(x)$. Figure 3.3 shows the implementations of both scalings. As shown in the figure, the propagation delay for one PB scaling or inverse scaling is t_X, since there is one level of XOR gates in the implementation.

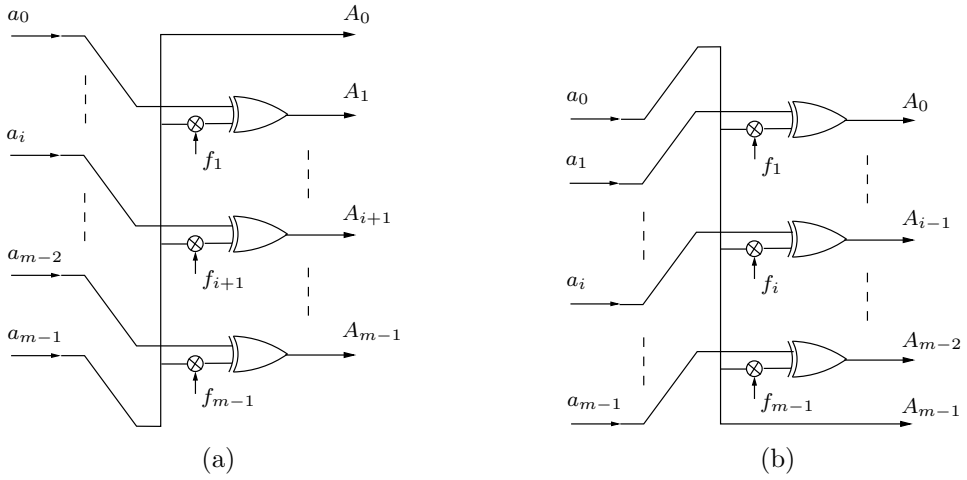


Figure 3.3: (a) Scaling and (b) inverse scaling in PB operations

As mentioned before, the multiplication of a finite field element with x^i or x^{-i}

can be implemented by i scalings or inverse scalings, respectively. Therefore, the maximum number of XOR gates required for the implementation is $i(\omega - 2)$.

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

$$t_e \approx t_X,$$
$$t_d \leq 2t_X.$$

3.3.2 Overheads in DB Operations

The hardware implementations of the DB scalings need a cyclic shift to right or left and $\omega - 2$ XOR gates, where ω is the Hamming weight of $F(x)$ (see Figure 3.4). As shown in the figure, the propagation delay for one DB scaling or inverse scaling is $(\omega - 2)t_X$ due to the propagation delay of the least significant bit of a scaling or the most significant bit of an inverse scaling.

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

$$t_e \approx (\omega - 2)t_X,$$
$$t_d \leq 2(\omega - 2)t_X.$$

3.3.3 Overheads in NB Operations

Squaring and taking the square root of an element represented in NB needs just a cyclic shift to right or left (see Figure 3.5).

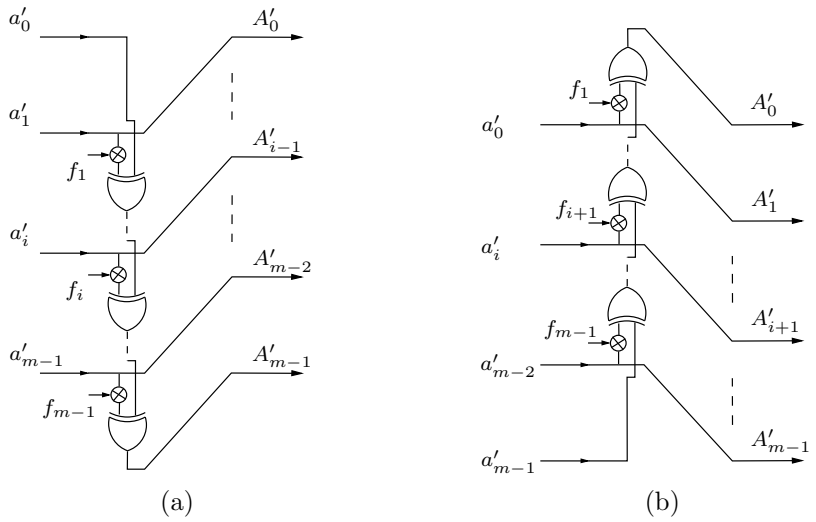


Figure 3.4: (a) Scaling and (b) inverse scaling in DB operations

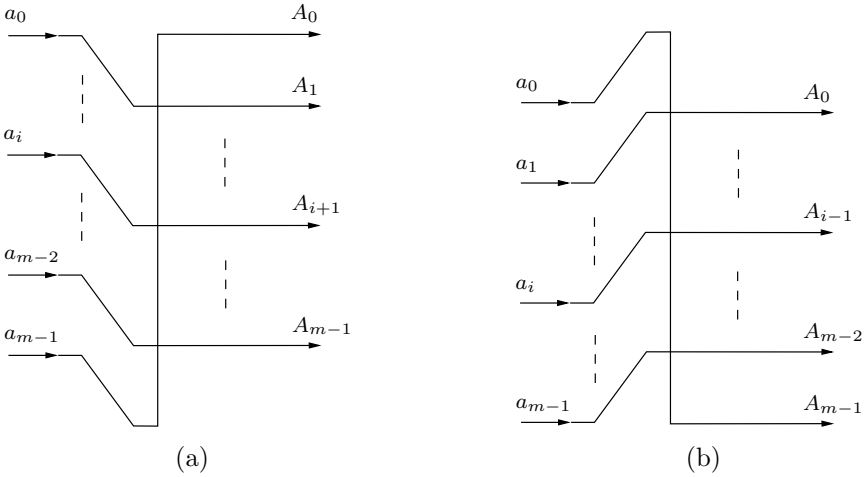


Figure 3.5: (a) Squaring and (b) taking the square root in NB operations

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

$$t_e = t_d = 0.$$

The area overheads for pipeline implementations of PB, DB and NB $GF(2^m)$

arithmetic operations with the proposed CED are summarized in Table 3.2. It is worth mentioning that the added buffer is m bits long. Also, m XOR gates and one m-input OR gate are needed in the equality checker unit. Note that for all practical values of m, either a trinomial ($\omega = 3$) or a pentanomial ($\omega = 5$) can be found as the field defining polynomial [59].

		PB[1]	DB[1]	NB
Maximum area overhead	m-bit Buffer	1	1	1
	2-input XOR	$(m + 4\omega - 8)$	$(m + 4\omega - 8)$	m
	m-input OR	1	1	1

Table 3.2: The area overheads of PB, DB and NB arithmetic operations with the proposed CED

[1]The exponentiation operation is not considered.

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

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

3.4.1 A Systolic PB Multiplier with CED

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

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

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

$$C = b_0A^{(0)} + b_1A^{(1)} + \cdots + b_{m-1}A^{(m-1)} \mod F(x).$$

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

$$C_j = \sum_{i=0}^{m-1} b_i A_j^{(i)}. \tag{3.6}$$

Expression (3.6) can be written in a recursive manner as follows:

$$C_j^{(i)} = C_j^{(i-1)} + b_i A_j^{(i)}, \tag{3.7}$$

where $0 \leq i \leq m-1$ and $c_j^{(-1)} = 0$. Also according to (3.2) for $1 \leq i \leq m-1$, we have:

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

Therefore,

$$A_j^{(i)} = \begin{cases} a_j; & i = 0, \\ a_{j-1}^{(i-1)} + a_{m-1}^{(i-1)} f_j; & 1 \leq i \leq m-1. \end{cases} \tag{3.8}$$

where $a_{-1}^{(i)} = 0$. Substituting (3.8) in (3.7), we have:

$$C_j^{(i)} = \begin{cases} b_0 a_j; & i = 0, \\ C_j^{(i-1)} + b_i \left(a_{j-1}^{(i-1)} + a_{m-1}^{(i-1)} f_j \right); & 1 \leq i \leq m-1. \end{cases} \tag{3.9}$$

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

Figure 3.6 shows a general view of an arbitrary cell of a semi-systolic PB mul-

tiplier based on expression (3.9). The reason that we call it semi-systolic is that each cell may receive a number of input signals from non-adjacent cells or output some signals to them.

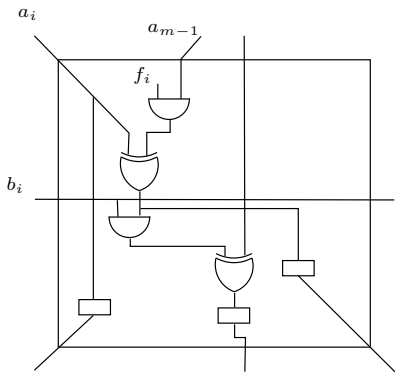


Figure 3.6: General cell architecture for a semi-systolic PB mutliplier

It is worth mentioning that except for f_0 and f_m, the number of non-zero f_i's for $1 \leq i \leq m-1$ is $\omega - 2$, where ω is the Hamming weight of $F(x)$. Therefore, we can have two different cells, one for those that have $f_i = 0$ (type 1) and another for those that have $f_i = 1$ (type 2) as shown in Figure 3.7.

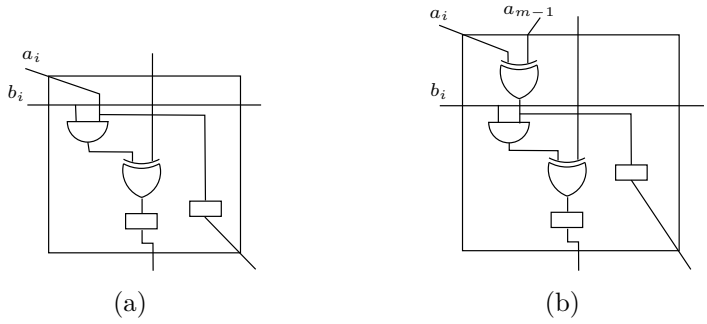


Figure 3.7: (a) Type 1 cell and (b) type 2 cell of a semi-systolic PB multiplier

Let us refer to type 1 and type 2 cells as PBT1 and PBT2. Figure 3.8 shows the semi-systolic PB multiplier.

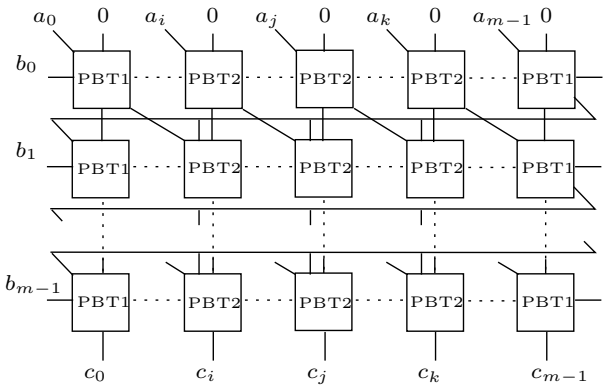


Figure 3.8: A semi-systolic PB mutliplier

In the figure, it is assumed that f_i, f_j and f_k are not zero. Consequently, the cells of the columns i, j and k are type 2 (PBT2). Generally, we have $(m - w + 2)$ PBT1 and $(\omega - 2)$ PBT2 in each row. Furthermore, PBT1 and PBT2 contain (1 AND, 1 XOR, 2 Latches) and (1 AND, 2 XORs, 2 Latches), respectively. Table 3.3(a) presents the total number of required gates or latches for a semi-systolic PB multiplier. It is worth mentioning that for all practical values of m, one can find irreducible low-weight polynomials, either a trinomial or a pentanomial, where a trinomial does not exist [59]. Also, the computation time for each type of cells is presented in Table 3.3(b).

(a)

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

(b)

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

Table 3.3: Space and time complexities of the semi-systolic PB multiplier

For the purpose of error detection, we applied the method discussed in Section 3.2.1. In other words, each encoding function is one forward scaling and the decoding function is two inverse scalings. Table 3.4 shows the area and time complexities of this work along with a number of previously published systolic or

semi-systolic PB multipliers without CED capability and with CED capability if applicable.

In [41] and [19], two checkers (comparators) are used. One is to detect errors at the output of the circuit and another is to detect errors on global lines. These lines are horizontal lines that connect a bit of one of the inputs (say input B) to all cells in a row of the multiplier. However, in our proposed scheme since both inputs are encoded, errors in the global lines can also be detected.

According to Table 3.4, the space complexities and latencies of the multipliers with or without CED presented in this work seem to be better as compared to the other multipliers mentioned in the table. Note that $\omega = 3$ or $\omega = 5$ and the latency of the multiplier without CED in [41] is the same as our work but that multiplier is not general. Apparently, the cell time complexity of our work is not among the best. However, this may not be considered as a drawback in multipliers with CED because the bottleneck for determining the minimum clock period is usually the propagation delay of equality checkers, not the cell time complexity. This will be further investigated in Section 3.4.4.

3.4.2 A Systolic DB Multiplier with CED

Suppose that $A, B, C \in GF(2^m)$ and $C = A.B \mod F(x)$. The formulation for DB multiplication is similar to the PB one except for the following. Let $A^{(0)} = A$. Then according to Lemma 3.1 for $1 \leq i \leq m-1$, we have:

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

Table 3.4: Space and time complexities of a number of systolic or semi-systolic PB multipliers

Multipliers	[41]		[19]		[64]	[70]	[39]	This work	
Generating polynomial	AOP		General		General	General	Trinomial	General	
CED	No	Yes	No	Yes	No	No	No	No	Yes
Cell no.	m^2	m^2+3m	m^2+m	m^2+2m	m^2	m^2	m^2	m^2	m^2
Space complexity	[((1))]								
XOR_2	$2m^2$	$2m^2+7m$	—	$2m$	—	$2m^2$	m^2+m	$m(m+\omega-2)$	$m^2+(\omega-1)m+4(\omega-2)$
XOR_3	—	—	m^2+m	m^2+2m	m^2	—	—	—	—
AND_2	$2m^2$	$2m^2+4m$	$2m^2+2m$	$2m^2+4m$	$2m^2$	$2m^2$	m^2	m^2	m^2
AND_3	—	m	—	—	—	—	—	—	—
$Latch_1$	$2m^2$	$2m^2+6m$	$3.5m^2+3.5m$	$3.5m^2+7.5m+1$	$7m^2$	$7m^2$	$3.5m^2+m$ [((2))]	$2m^2$	$2m^2+m$
OR_m	—	2	—	2	—	—	—	—	1
2-1 Switch	—	—	—	—	—	—	m	—	—
Min. CLK period	$2T_A+2T_X+T_L$	[((3))]	$T_A+T_{3X}+T_L$	[((3))]	$T_A+T_X+2T_L$	$T_A+T_X+2T_L$	$T_A+T_X+T_L$	$T_A+2T_X+T_L$	[((3))]
Latency	m	$m+2$	$m+1$	$m+5$	$3m$	$3m$	$m+k$	m	$m+1$

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

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

[((3))] Should be similarly computed according to Table 3.1 and $\text{Max}\{t_i\}$ is same as that of the multiplier without CED.

Therefore,

$$A_j^{(i)} = \begin{cases} a'_j; & i = 0, 0 \leq j \leq m-1, \\ a_{j+1}^{'(i-1)}; & 1 \leq i \leq m-1, 0 \leq j \leq m-2, \\ \sum_{k=0}^{m-1} f_k a_k^{'(i-1)}; & j = m-1, 1 \leq i \leq m-1. \end{cases} \tag{3.10}$$

Substituting (3.10) in (3.7), we have:

$$C_j^{(i)} = \begin{cases} b_0 a'_j; & i = 0, 0 \leq j \leq m-1, \\ C_j^{(i-1)} + b_i a_{j+1}^{'(i-1)}; & 1 \leq i \leq m-1, 0 \leq j \leq m-2, \\ C_j^{(i-1)} + b_i \sum_{k=0}^{m-1} f_k a_k^{'(i-1)}; & j = m-1, 1 \leq i \leq m-1. \end{cases} \tag{3.11}$$

Considering Expression (3.11), we can consider two types of cells for semi-systolic DB multipliers as shown in Figure 3.9. Except for the last column of the multiplier ($j \neq m-1$), type 1 cells are used (see Figure 3.9(a)). These cells requires one 2-input AND gate, one 2-input XOR gate and two 1-bit latches.

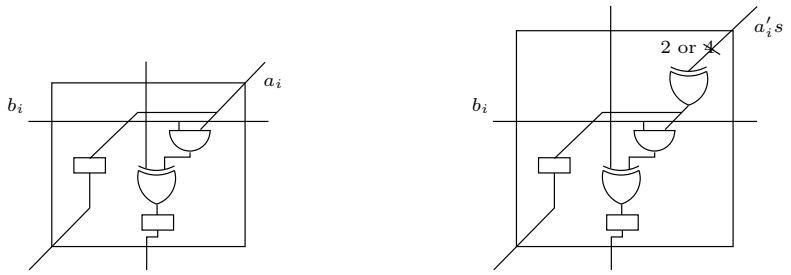


Figure 3.9: (a) Type 1 cell and (b) type 2 cell of a semi-systolic DB multiplier

Type 2 cells (Figure 3.9(b)) are used in the last column. In addition to the gates and latches needed for type 1 cells, type2 cells require one extra 2-input or 4-input XOR gate depending on whether the defining polynomial of the underlying field is a trinomial ($\omega = 3$) or pentanomial ($\omega = 5$), respectively. Figure 3.10 shows a semi-systolic DB multiplier.

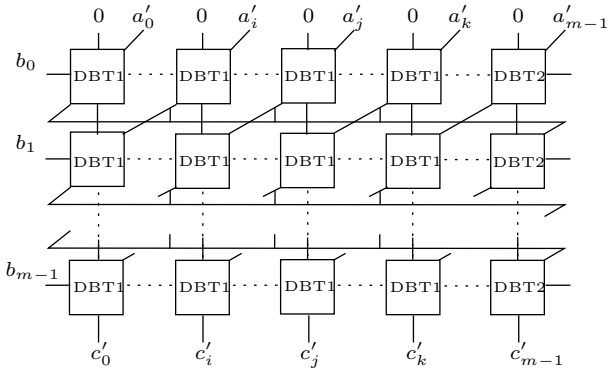


Figure 3.10: A semi-systolic DB multiplier

The error detection scheme presented in Section 3.2.2 is applied to this multiplier. Hence, the encoding and decoding functions are one forward scaling and two inverse scalings, respectively. Table 3.5 summarizes the time and space complexities of this work and a number of previously published multipliers with and without CED capability as appropriate.

According to Table 3.5 the multipliers (with and without CED) presented in this work can be considered as the best ones in terms of space complexity and latency. It is worth mentioning that latches are relatively more area consuming components and hence the multiplier in [39] requires more space than our work. The cell time complexity of our work is not lower than the other multipliers, however, this does not imply that the minimum clock period (MCP) of our work with CED is larger than another multiplier with CED. This case mostly happens when the other delay parameters for determining MCP (according to Table 3.1), such as propagation delay of the equality checker, is larger than the cell time complexity.

3.4.3 A Systolic NB Multiplier with CED

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

Table 3.5: Space and time complexities of a number of systolic or semi-systolic DB multipliers

Multipliers	[25]	[40]		[39]	This work			
CED	No	No	Yes	No	No		Yes	
Cell no.	m^2	m^2	m^2+m	m^2	m^2		m^2	
Space complexity					$\omega=3$	$\omega=5$	$\omega=3$	$\omega=5$
XOR_2	$2m^2$	$2m^2$	$3m^2+3m-2$	m^2+m	m^2+m	m^2-m	m^2+2m+4	m^2+12
XOR_4	—	—	—	—	—	m	—	m
AND_2	$2m^2$	$2m^2$	$3m^2-m$	m^2	m^2		m^2	
$Latch_1$	$7m^2$	$5m^2$	$10m^2-2m-4$	$3.5m^2+m$	$2m^2$		$2m^2+m$	
OR_m	—	—	1	—	—		1	
2-1 Switch	—	—	—	m	—		—	
Min. CLK period	$T_A+T_X+T_L$	$T_A+T_X+T_L$	((1))	$T_A+T_X+T_L$	$T_A+2T_X+T_L$	$T_A+T_X+T_{4X}+T_L$	((2))	
Latency	$3m$	$3m$	$3m+1$	$m+k$	m		$m+1$	

((1))Should be computed according to Table 3.1, where $\text{Max}\{t_i\} = T_A + 2T_X + T_L$.

((2))Should be similarly computed according to Table 3.1 and $\text{Max}\{t_i\}$ is same as that of the multiplier without CED.

Type I Optimal Normal Basis (ONB1)

Suppose that $m+1$ is a prime number and 2 is primitive in $GF(m+1)$. Then the field defining polynomial, can be chosen to be $F(x) = \sum_{i=0}^{m} x^i$, which is an all-ones irreducible polynomial over $GF(2^m)$. Let x be the root of $F(x)$. Since $F(x)|(x^{m+1}-1)$, we have $x^{m+1} \equiv 1$. Therefore, the set of normal basis presented in Section 2.1, can be reduced accordingly. The resulting set has m linearly independent elements [45] as follows and is referred to as type I optimal normal basis:

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

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

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

Suppose that the NB and ONB1 representations of $A \in GF(2^m)$ are $A = \hat{a}_0 x + \hat{a}_1 x^2 + \hat{a}_2 x^{2^2} + ... + \hat{a}_{m-1} x^{2^{m-1}}$ and $A = a_1 x + a_2 x^2 + a_3 x^3 + ... + a_m x^m$, respectively. Also, let us assume that after permutation we have $a_j = \hat{a}_i$, Then:

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

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

notations we have:

$$\begin{aligned} C &= A.B \mod (x^{m+1}-1) \\ &= \hat{b}_0 A^{(0)} + \hat{b}_1 A^{(1)} + \cdots + \hat{b}_{m-1} A^{(m-1)} + \hat{b}_m A^{(m)} \mod (x^{m+1}-1). \end{aligned} \tag{3.12}$$

Expression (3.12) is very similar to PB multiplication except that this multiplication is $(m+1)$ bits long. Additionally, for $1 \leq i \leq m$ we have:

$$\begin{aligned} A^{(i)} &= \sum_{j=0}^{m} \hat{a}_j^{(i-1)} x^{j+1} \mod (x^{m+1}-1) \\ &= \hat{a}_m^{(i-1)} x^{m+1} + \sum_{j=0}^{m-1} \hat{a}_j^{(i-1)} x^{j+1} \mod (x^{m+1}-1) \\ &= \sum_{j=0}^{m} \hat{a}_{<j-1>}^{(i-1)} x^j. \end{aligned} \tag{3.13}$$

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

Now, similar to expression (3.7) for $1 \leq i, j \leq m$, we have:

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

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

$$C_j^{(i)} = \begin{cases} \hat{b}_0 \hat{a}_j; & i = 0, \\ C_j^{(i-1)} + \hat{b}_i \hat{a}_{<j-1>}^{(i-1)}; & 1 \leq i \leq m-1. \end{cases} \tag{3.14}$$

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

Note that $C^{(m)}$ is not necessarily in ONB1 representation, since $C_0^{(m)}$ may not

be zero. To resolve this issue, one can zero out $C_0^{(m)}$ as follows:

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

Figure 3.11 shows a cell of a semi-systolic ONB1 multiplier.

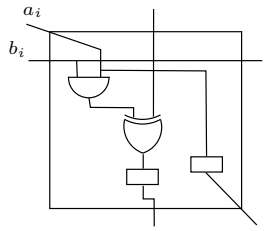


Figure 3.11: The cell of a semi-systolic ONB1 multiplier

It is worth mentioning that since $F(x)$ is an all-ones polynomial, one can simply add the LSB of $C^{(m)}$ with all other bits of $C^{(m)}$ in the hardware implementation. Figure 3.12 shows a semi-systolic ONB1 multiplier.

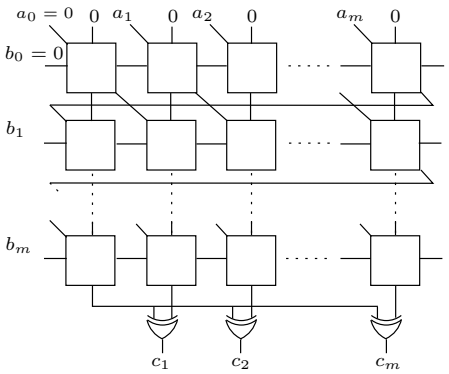


Figure 3.12: A semi-systolic ONB1 multiplier

Furthermore, as shown in Figure 3.12, $b_0 = 0$ is one input of all AND gates of the cells in the first row. Therefore, the first row can be omitted and then a_i's should be fed into the first row after one rotation (see Figure 3.13). Clearly, in this way the space and latency of the multiplier are slightly reduced.

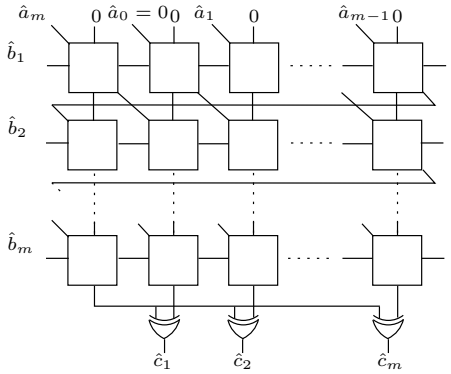


Figure 3.13: A semi-systolic ONB1 multiplier with m rows

The error detection scheme presented in Section 3.2.3 is applied to this multiplier. The encoding and decoding functions are one or two shifts to the left or right. Apparently, the encodings should be performed before the permutation Γ_1 and the decoding should be performed after the inverse permutation Γ_1^{-1}. The time and space complexities of this work with and without CED capability are presented in Table 3.6.

According to Table 3.6, the ONB1 multiplier presented here is better than that in [39] in terms of space complexity and latency and they are the same in terms of cell time complexity. Furthermore, to the best of our knowledge this is the first work that addressed the CED in NB multipliers.

Type II Optimal Normal Basis (ONB2)

Suppose that $2m+1$ is a prime number and either of the following conditions holds:

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

Then the field $GF(2^m)$ can be constructed using the normal element $\gamma + \gamma^{-1}$ [45] and the basis for field representation is referred to as type II optimal normal basis

as follows:

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

where γ is a primitive $(2m+1)^{th}$ root of unity. Hence, for $1 \leq i \leq 2m-1$, $\gamma^i = 1$ only when $i = 2m+1$. It is worth mentioning that the above set can be rewritten as follows [62]:

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

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

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

Suppose that the NB and ONB2 representations of $A \in GF(2^m)$ are $A = \hat{a}_0 x + \hat{a}_1 x^2 + \hat{a}_2 x^{2^2} + ... + \hat{a}_{m-1} x^{2^{m-1}}$ and $A = a_1(\gamma+\gamma^{-1}) + a_2(\gamma^2+\gamma^{-2}) + a_3(\gamma^3+\gamma^{-3}) + \cdots + a_m(\gamma^m+\gamma^{-m})$, respectively. Also, let us assume that after permutation we have $a_j = \hat{a}_i$. Then:

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

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

Now, suppose that $A, B, C \in GF(2^m)$ are represented in ONB2, e.g., $A = \sum_{i=1}^{m} \hat{a}_i(\gamma^i + \gamma^{-i})$. Then following [62], we have:

$$\begin{aligned}
C = A.B &= \sum_{i=1}^{m}\sum_{j=1}^{m} \hat{a}_i\hat{b}_j(\gamma^i + \gamma^{-i})(\gamma^j + \gamma^{-j}) \\
&= \sum_{i=1}^{m}\sum_{j=1}^{m} \hat{a}_i\hat{b}_j \left[(\gamma^{(i-j)} + \gamma^{-(i-j)}) + (\gamma^{(i+j)} + \gamma^{-(i+j)})\right] \\
&= \sum_{i=1}^{m}\sum_{j=1}^{m} \hat{a}_i\hat{b}_j \left[\gamma^{(i-j)} + \gamma^{-(i-j)}\right] + \sum_{i=1}^{m}\sum_{j=1}^{m} \hat{a}_i\hat{b}_j \left[\gamma^{(i+j)} + \gamma^{-(i+j)}\right] \\
&= C_1 + C_2. \\
C_1 &= \sum_{i=1}^{m}\sum_{j=1}^{i} \hat{a}_i\hat{b}_j \left[\gamma^{(i-j)} + \gamma^{-(i-j)}\right] + \sum_{i=1}^{m}\sum_{j=i+1}^{m} \hat{a}_i\hat{b}_j \left[\gamma^{(i-j)} + \gamma^{-(i-j)}\right] \\
&= C_{11} + C_{12}. \\
C_2 &= \sum_{i=1}^{m}\sum_{j=1}^{m-i} \hat{a}_i\hat{b}_j \left[\gamma^{(i+j)} + \gamma^{-(i+j)}\right] + \sum_{i=1}^{m}\sum_{j=m-i+1}^{m} \hat{a}_i\hat{b}_j \left[\gamma^{(i+j)} + \gamma^{-(i+j)}\right] \\
&= C_{21} + C_{22}.
\end{aligned}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Finally, we have:

$$C = C_I + C_{II} = \sum_{i=1}^{m} \sum_{k=1}^{m} \hat{a}_i \left(\hat{b}_{|i-k|} + \hat{b}_{||i+k||} \right) \left(\gamma^k + \gamma^{-k} \right), \tag{3.15}$$

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

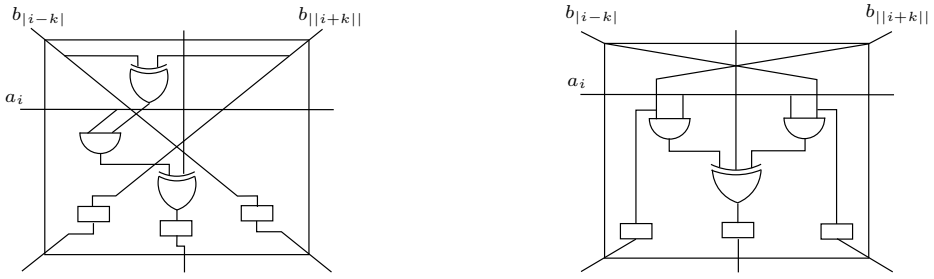
$$\hat{c}_k = \sum_{i=1}^{m} \hat{a}_i \left(\hat{b}_{|i-k|} + \hat{b}_{||i+k||} \right).$$

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

$$\begin{aligned} C_k^{(i)} &= C_k^{(i-1)} + \hat{a}_i \left(\hat{b}_{|i-k|} + \hat{b}_{||i+k||} \right) \\ &= C_k^{(i-1)} + \hat{a}_i \hat{b}_{|i-k|} + \hat{a}_i \hat{b}_{||i+k||}, \end{aligned} \tag{3.16}$$

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

Figure 3.14 shows two possible implementations for a cell of the ONB2 semi-systolic multiplier according to Expression 3.16. One can choose one of the above-mentioned implementations based on space and/or time complexities.



(a) (b)

Figure 3.14: Two cells of a semi-systolic ONB2 multiplier with same functionality

In this work, we have chosen the cell shown in Figure 3.14(b). A complete semi-systolic ONB2 multiplier is shown in Figure 3.15.

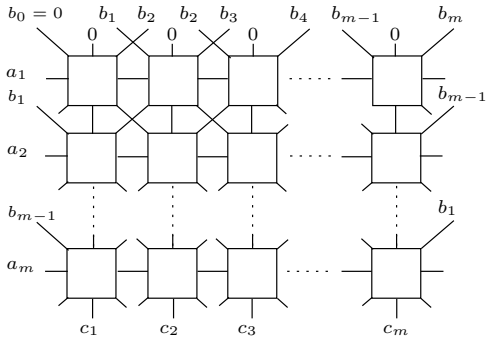


Figure 3.15: A semi-systolic ONB2 multiplier

Similar to the ONB1 multiplier, the error detection scheme presented in Section 3.2.3 is applied to this multiplier. The encoding and decoding functions are one or two shifts to the left or right. It is worth mentioning that the encodings and the decoding should be performed before the permutation function Γ_2 and after the inverse permutation function Γ_2^{-1}, respectively. The time and space complexities of this work for ONB1 and ONB2 along with a number of other related previous work with and/or without CED capability are presented in Table 3.6.

According to Table 3.6, the ONB2 multiplier presented here can be considered among the best in terms of space complexity and is the best in terms of latency. Additionally as mentioned earlier, to the best of our knowledge this is the first work addressing the CED in NB multipliers.

3.4.4 Some Notes About Delays of Cells and Equality Checkers

The clock rate of a pipeline architecture can be determined according to a number of parameters presented in Table 3.1 (second column). The propagation delays of the stages of some pipeline architectures such as systolic or semi-systolic arrays are

Table 3.6: Space and time complexities of a number of systolic or semi-systolic ONB multipliers

Multipliers	[39]		[38]	This work			
ONB Type	I	II	II	I		II	
CED	No	No	No	No	Yes	No	Yes
Cell no.	$(m+1)^2$	m^2	m^2	$m(m+1)$	$m(m+1)$	m^2	
Space complexity							
XOR_2	$(m+1)^2+m$	m^2+m	m	$m(m+1)$	m^2+2m	—	m
XOR_3	—	—	m^2	—	—	m^2	m^2
AND_2	$(m+1)^2$	m^2	$2m^2+m$	$m(m+1)$	$m(m+1)$	$2m^2$	$2m^2$
$Latch_1$	$3.5(m+1)^2$	$3.5m^2+m$	$5m^2$	$2m(m+1)$	$2m^2+3m$	$3m^2$	$3m^2+m$
OR_m	—	—	—	—	1	—	1
Min. CLK period	$T_A+T_X+T_L$	$T_A+T_X+T_L$	$T_A+T_{3X}+T_L$	$T_A+T_X+T_L$	((1))	$T_A+T_{3X}+T_L$	((1))
Latency	$m+1$	$m+1$	$m+1$	m	$m+1$	m	$m+1$

((1))Should be computed according to Table 3.1 and $\text{Max}\{t_i\}$ is same as that multiplier without CED.

small. Particularly for these architectures, one important parameter to determine the clock rate is the delay of the equality checker. In the following this issue is investigated.

The equality checker is basically one level of XOR gates to check the equality of the bits of two inputs and one OR unit to determine the final error signal as shown in Figure 3.16.

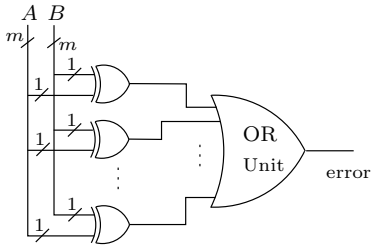


Figure 3.16: An m-bit equality checker

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

- 1-level 163-input OR
- 2-level 13-input OR
- 3-level 6-input OR
- 4-level 4-input OR
- 8-level 2-input OR

For the purpose of simulation, gates were modeled using *ratioed* logic that uses only one PMOS transistor in the pull-up network. We used $0.18\mu m$ technology.

Also, we initialized all inputs of the gates with zero and after a while we changed the value of only one of them to one. The result of the transient response simulation is shown in Figure 3.17.

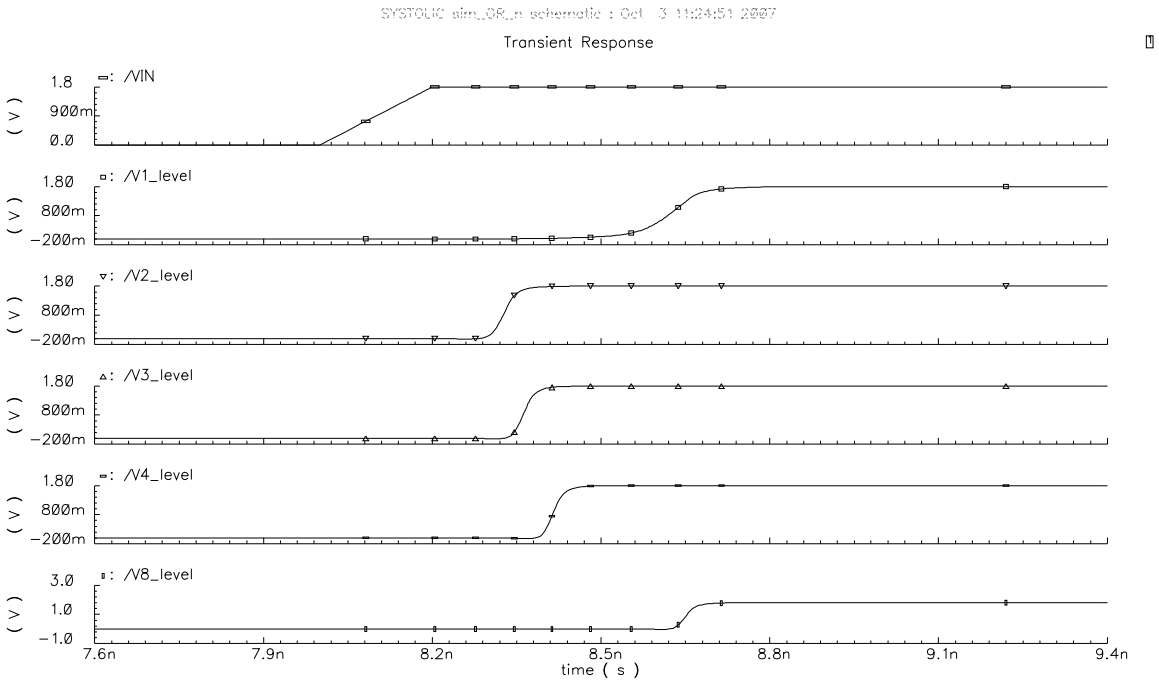


Figure 3.17: The propagation delays for different ways of implementing an m-input OR unit

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

It is worth mentioning that if one needs to use the standard cells of a library, the best choice is 4-level 4-input OR design because the 13-input OR gate are often unavailable in the standard cells and it has the smallest propagation delay after

2-level 13-input design.

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

3.5 Error Detection Capability

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

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

The percentage of error detection for the transient faults is 100%, because either these faults make the output erroneous or they are masked. In the first case, the result of the first run and the second run are different. Hence, the fault causing errors is detected.

For permanent (or intermittent) faults, we performed a number of simulation-based fault injections on the PB, DB, ONB1 and ONB2 multipliers presented in Section 3.4. Fault injections were performed in a C model of the multiplier. We injected stuck-at faults (both stuck-at 1 and stuck-at 0) at the input and output pins of the gates of the multiplier. In the proposed scheme, same faults are injected

in the same locations of the circuits in both runs. Fault injection in a complete multiplier with a field degree of approximately 163 is extremely time consuming. Therefore, faults were injected in only one randomly chosen row of cells of the semi-systolic multipliers. We performed the fault injections in two phases as discussed below.

Single-Bit Stuck-at Faults

In this experiment, only one-bit stuck-at fault was injected during the multiplication. As mentioned earlier, the location of a fault can be at the input and/or output pins of gates. Hence, to perform fault injection, a multiplexer can be placed at the fault location, where the control signal of the multiplexer selects between the original value of that point and the fault. Moreover, the fault value can be chosen to be 1 or 0. This is shown in Figure 3.18.

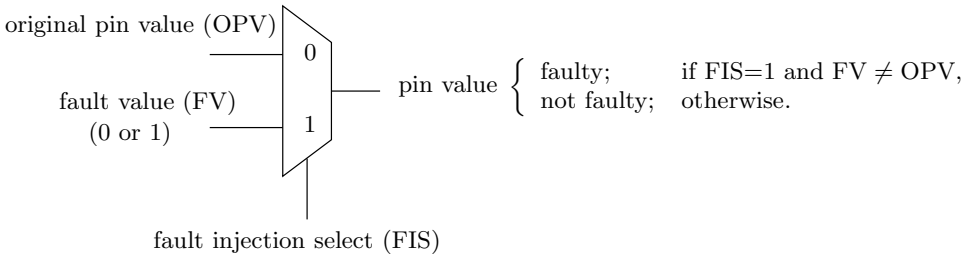


Figure 3.18: Conventional fault injection at a gate pin

The number of faults that can be injected to a multiplier for each set of inputs depends on the number of AND gates and XOR gates of that multiplier (see Table 3.4, Table 3.5 and Table 3.6). It is worth mentioning that the output pins of AND gates in each cell, which are direct inputs of the XOR gates, were not injected. In this experiment, we simulated the fault injection for PB, DB, ONB1 and ONB2 multipliers. Each multiplier was simulated for one million random input pairs and for every pair, all the above mentioned single-bit stuck-at faults were injected. Ta-

ble 3.7 shows the number of injected single-bit stuck-at faults for each set of inputs as well as the percentage of error detection for each multiplier. Note that the field $GF(2^{163})$ cannot be represented using ONB1 or ONB2. Therefore, we chose other fields of close degrees instead. The number of detected, masked and undetected faults are 1074930775, 575069225 and 0 for the PB multiplier, 1078354457, 563645543 and 0 for the DB multiplier, 994414981, 634206218 and 3378801 for the ONB1 multiplier, and 1716316667, 1053683343 and 0 for the ONB2 multiplier, respectively.

Error detection scheme	No. of stuck-at faults	No. of random inputs	Percentage of error detection
$GF(2^{163})$ PB	1650	1000000	100%
$GF(2^{163})$ DB	1642	1000000	100%
$GF(2^{162})$ ONB1	1632	1000000	99.66%
$GF(2^{173})$ ONB2	2770	1000000	100%

Table 3.7: Percentage of error detection of the RESO based scheme for finite field multipliers against single stuck-at faults

In the following, we give an example for a single stuck-at fault injection at a $GF(2^4)$ ONB1 multiplier. Let $A = 6$ and $B = 3$ be the inputs of the multiplier. The fault free result of the multiplication is 2 or 0100[1]. We inject a stuck-at one fault at the right hand side input of the XOR gate (see Figure 3.11) in the first cell of the second row of the multiplier. In the first computation according to Figure 3.12, we have:

1. Converting from NB to ONB1: $A_{ONB1} = 00101$ and $B_{ONB1} = 01100$

2. Computation in presence of the fault:

[1]Binary representations in this example are least significant bit (LSB) first.

row	$b_{i=\text{row}-1}$	vertical output of the cells	diagonal output of the cells
1	0	00000	00101
2	1	00010	10010
3	1	01011	01001
4	0	01011	10100
5	0	01011	01010

3. Converting the result of multiplication, i.e., 01011, from ONB1 to NB: 1011

In the second computation, we have:

1. Encoding (squaring): $enA = 0011$ and $enB = 0110$

2. Converting from NB to ONB1: $enA_{ONB1} = 00011$ and $enB_{ONB1} = 00101$

3. Computation in presence of the fault:

row	$b_{i=\text{row}-1}$	vertical output of the cells	diagonal output of the cells
1	0	00000	00011
2	0	10000	10001
3	1	01000	11000
4	0	01000	01100
5	1	01110	00110

4. Converting the result of multiplication, i.e., 01110, from ONB1 to NB: 1101

5. Decoding (taking the square root): 1011

The final results of the first and the second computations are same and both are incorrect. Therefore, the fault cannot be detected. It is worth mentioning that as presented in Table 3.7 we could not find any undetected error for the PB, DB and ONB2 multipliers based on our simulations.

Multiple-Bit Stuck-at Faults

To inject multiple-bit stuck-at faults, the locations for the injections were randomly selected from the above mentioned single-bit fault locations. Then one stuck-at 0 or stuck-at 1 was randomly injected there. We injected 500 multiple-bit stuck-at faults for each set of inputs. Furthermore, one million random sets of inputs were simulated in each experiment. For example for a $GF(2^{163})$ PB multiplier, there are 825 single-bit stuck-at fault locations. All simulations for PB, DB, ONB1 and ONB2 multipliers result in the detection of all errors.

For more information about our procedures of the fault injections, see Appendix A.

3.6 Concurrent Error Correction

The RESO method can also be used for correcting errors resulting from transient faults. As stated earlier, we assume that locations of these faults, occurred naturally or injected by an attacker, are random. This scheme, however, is not suitable to correct errors due to permanent faults.

Figure 3.19 shows a general architecture for correcting errors confined in one of the three runs. This architecture uses a well known majority voter and it can be easily extended to correct $M \leq \lfloor \frac{N-1}{2} \rfloor$ errors using N runs (see [37]).

Bellow we give the encoding and decoding functions for CEC corresponding to Figure 3.19 of each basis:

- Encoding and decoding functions for PB and DB

    1. Addition/Subtraction:
    $E_{1,1} = x$, $E_{1,2} = x$, $D_1 = x^{-1}$
    $E_{2,1} = x^{-1}$, $E_{2,2} = x^{-1}$, $D_2 = x$

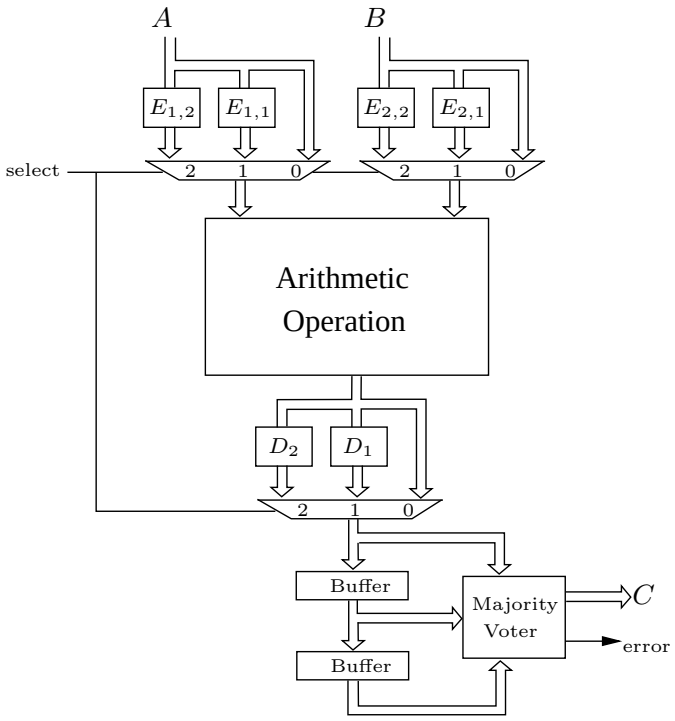


Figure 3.19: General architecture for the arithmetic operations with CEC (using a 2-of-3 system)

2. Multiplication:

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

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

3. Inversion:

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

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

4. Division:

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

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

5. Exponentiation:

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

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

- Encoding and decoding functions for NB

  1. Addition/Subtraction/Multiplication/Division:

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

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

  2. Inversion/Exponentiation:

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

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

3.7 Summary

This chapter has presented a number of schemes, which are efficient for pipelined architectures and are based on recomputing with shifted operands (RESO) method. These schemes have been developed to concurrently detect errors in polynomial, dual, type I and type II optimal normal bases arithmetic operations. We have also presented one semi-systolic multiplier for each of above-mentioned bases and applied the CED scheme to them. We have compared these multipliers with a number of previously published systolic and/or semi-systolic ones. The results show that this scheme can be considered among the best. Also, a simulation-based fault injection has been performed for each of the multipliers. Results of the simulations for single stuck-at faults show 100%, 100%, 99.66% and 100% error detection for polynomial, dual, type I and type II bases multipliers, respectively. The simulations also show that the percentage of error detection of this scheme for the above-mentioned multipliers against multiple stuck-at faults is 100%. Finally, we also commented on how RESO can be used for concurrent error correction to deal with transient faults.

Chapter 4

Single Input Multiple Parity (SIMP) Error Detection Scheme for Polynomial Basis Multipliers

This chapter focuses on the detection of errors in polynomial basis multipliers. In [24], Fenn et al. presented a concurrent error detection scheme for finite field multipliers over binary extension fields. They used a parity bit for detecting errors in bit-serial multipliers, using a number of bases for representation of fields, defined by an irreducible all-ones polynomial. Thus, the scheme is not generic in the sense that it cannot be used for other field defining polynomials. In [18], Chiou presented a concurrent error detection for two bit-parallel systolic multipliers for extension fields in which the field defining polynomials are irreducible all-ones polynomials or irreducible equally spaced ones. In [55, 57], Reyhani-Masoleh and Hasan developed a generic parity based error detection scheme for both bit-serial and bit-parallel polynomial basis multipliers. The scheme can detect any odd number of erroneous bits. In this scheme, input parity is propagated through the multiplier, and predicted output parity is compared to actual output parity. In case of inequality of

the parities, an error signal is given.

This work extends the work of [55, 57] by applying multiple parity bits to polynomial basis multipliers. Like [57], this work can be applied to any finite field GF(2^m). However, unlike [57], our work can detect all odd parity errors as well as most of the even parity errors. Additionally, our work can detect at least m multiple-bit errors in the multiplier.

The main contributions of this chapter are summarized as follows:

- A multiple parity scheme that can detect multiple-bit errors in both bit-serial and bit-parallel polynomial basis multipliers over binary extension fields are presented. The error detection capability of the scheme in the presence of multiple-bit random errors is also investigated. With our proposed frequency of check points, a maximum of one multiple-bit error in each round of the bit-serial operation (or each slice of the bit-parallel operation) can be detected. This implies that in a $GF(2^m)$ polynomial basis multiplier, at least m multiple-bit errors can be detected.

- A number of experimental analyses are presented, including the simulation-based fault-injection evaluation of the scheme and the analyses of the area and time overheads. Our experimental results show that the area overhead tends to increase linearly as the number of parity bits increases but the probability of undetected errors decreases quite quickly. Furthermore, the area overhead for the bit-serial implementation is quite low, e.g., for 8 parity bits the area overhead is 10.29% and the error detection probability is 0.996. The area overhead for a bit-parallel implementation of the multiplier is greater than the corresponding bit-serial one, but it is still lower than the conventional *dual modular redundant* systems. The average time overhead due to the use of the scheme in bit-parallel implementations is 25%. For bit-serial implementations, time overheads have been observed to be small to negligible.

The organization of the remainder of this chapter is as follows. A concurrent error detection strategy is presented in Section 4.1. In Section 4.2, the error detection capability of the scheme is investigated. Our experimental results for this scheme are reported in Section 4.3. An alternative partitioning is presented in Section 4.4. Finally, Section 4.5 gives a summary of the chapter.

This work appeared in [4, 7].

4.1 Concurrent Error Detection Strategy

In this section, an error detection scheme for PB multipliers is presented. Errors may be caused by different types of faults such as open faults, short (bridging) faults, and/or stuck-at faults. Furthermore, the faults can be transient or permanent. The goal of this scheme is to detect as many random errors as possible including single and multiple errors. Towards this goal, we use a parity based method. One-bit parity is able to detect the presence of any odd number of erroneous bits [42]. Here, we use additional parity bits in order to increase the error detection capability. In particular, an m-bit input is divided into k parts and for each part one parity bit is used. Thus, the m-bit PB representation of $A \in GF(2^m)$ is divided as follows:

$$A = (A_0, A_1, A_2, \cdots, A_{k-1}).$$

The length of A_j, $0 \leq j \leq k-1$, is

$$l_j = \begin{cases} \lfloor \frac{m}{k} \rfloor + 1 & \text{if } j < m \bmod k; \\ \lfloor \frac{m}{k} \rfloor & \text{otherwise.} \end{cases}$$

For the sake of simplicity, we assume that $k|m$ and the length of each part is

$l = \frac{m}{k}$, i.e.,

$$A_j = x^{jk} \sum_{i=0}^{l-1} a_{jk+i} x^i = (a_{jk}, a_{jk+1}, a_{jk+2}, \cdots, a_{jk+l-1}).$$

The parity of A_j is denoted as $P(A_j)$. Using parity bits of A_j's, a k-bit parity of A is formed as follows:

$$P(A) = (P(A_0), P(A_1), P(A_2), \cdots, P(A_{k-1})).$$

Then using the parity $P(A)$, we construct the encoded A as follows:

$$E(A) = (A_0, A_1, A_2, \cdots, A_{k-1}, P(A)).$$

Unlike A which is represented with m bits, the field defining irreducible polynomial $F(x)$ requires $m+1$ bits. In order to have the same length for partitioning, we exclude the leading coefficient of $F(x)$ and divide $F(x) - x^m$ into k parts as follows:

$$F(x) - x^m = (F_0, F_1, \cdots, F_{k-1}).$$

The parity bit of F_j, $0 \leq j \leq k-1$, is denoted as $P(F_j)$.

One of the important issues in detecting errors in the output of a finite field multiplier (or an arbitrary circuit, in general) is parity prediction. The latter refers to the task of determining the parity of the expected outputs by using the corresponding inputs as well as the functionality of the circuit. As mentioned in Section 2.2.1, a polynomial basis multiplier consists of three modules: 1) SR module 2) SM module, and 3) VA module. In the following, the parity prediction method for each of these modules will be discussed.

4.1.1 Multiple Parity Prediction in SR Module

In the following, the output parity of an SR module is predicted.

Let $A' = xA$, i.e.,

$$\begin{aligned} A' &= \sum_{i=0}^{l-1} a_i x^{i+1} + x^l \sum_{i=0}^{l-1} a_{l+i} x^{i+1} + \\ &\quad \cdots + x^{(k-1)l} \sum_{i=0}^{l-1} a_{(k-1)l+i} x^{i+1} \\ &= \left(0 + \sum_{i=1}^{l-1} a_{i-1} x^i\right) + x^l \left(a_{l-1} + \sum_{i=1}^{l-1} a_{l+i-1} x^i\right) + \\ &\quad \cdots + x^{(k-1)l} \left(a_{(k-1)l-1} + \sum_{i=1}^{l-1} a_{(k-1)l+i-1} x^i\right) + a_{kl-1} x^{kl}. \end{aligned}$$

A' must be reduced by $F(x) = x^m + \sum_{j=0}^{k-1} F_j(x)$ as follows:

$$\begin{aligned} A' \mod F(x) = & \\ & \left(0 + \sum_{i=1}^{l-1} a_{i-1} x^i\right) + x^l \left(a_{l-1} + \sum_{i=1}^{l-1} a_{l+i-1} x^i\right) \\ & + \cdots + x^{(k-1)l} \left(a_{(k-1)l-1} + \sum_{i=1}^{l-1} a_{(k-1)l+i-1} x^i\right) \\ & + a_{m-1} \left(\sum_{j=0}^{k-1} F_j(x)\right). \end{aligned}$$

Now, we group the expression and obtain

$$\begin{aligned} A' \mod F(x) = & \\ & \left(0 + \sum_{i=1}^{l-1} a_{i-1}x^i + a_{m-1}\sum_{i=0}^{l-1} f_i x^i\right) \\ & + x^l\left(a_{l-1} + \sum_{i=1}^{l-1} a_{l+i-1}x^i + a_{m-1}\sum_{i=0}^{l-1} f_{l+i}x^i\right) \\ & + \cdots + x^{(k-1)l}\left(a_{(k-1)l-1} + \sum_{i=1}^{l-1} a_{(k-1)l+i-1}x^i\right. \\ & \left. + a_{m-1}\sum_{i=0}^{l-1} f_{(k-1)l+i}x^i\right). \end{aligned}$$

Thus, the j^{th} part of A' for $0 \leq j \leq k-1$ can be derived as:

$$A'_j = x^{jl}\left(a_{jl-1} + \sum_{i=1}^{l-1} a_{jl+i-1}x^i + a_{m-1}\sum_{i=0}^{l-1} f_{jl+i}x^i\right) \tag{4.1}$$

where $a_{-1} = 0$. Figure 4.1 shows a circuit diagram implementing A'_j. In practice, many coefficients of $F(x)$ are zero and hence the corresponding XOR gates in Figure 4.1 are not needed. By cascading k copies of the circuit shown in Figure 4.1, an SR module can be constructed as illustrated in Figure 4.2.

Let ω be the Hamming weight of $F(x)$. The total number of two-input XOR gates required in an SR module is $\omega - 2$, since no XOR gate is needed for the first and the last coefficients of $F(x)$.

For parity prediction of the j^{th} part of the SR module, we have the following lemma where $A' = xA$ and $P_{F_j} = \sum_{i=0}^{l-1} f_{jl+i}$.

Lemma 4.1 *Let $P(A_j)$ and $P(A'_j)$ be the parities of the input and the expected*

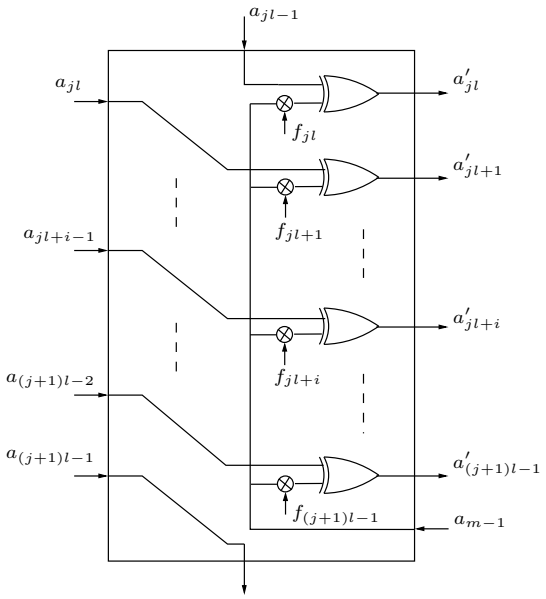


Figure 4.1: The j^{th} part of the SR module

output of the j^{th} part of the SR module, respectively. Then,

$$P(A'_j) = a_{jl-1} + P(A_j) + a_{(j+1)l-1} + a_{m-1}P_{F_j}.$$

Proof Using (4.1) the proof is immediate. ∎

Figure 4.3 shows the parity prediction circuit of the j^{th} part of the SR module, where $P(x)$ is predicted parity of x. The parity of the j^{th} part of $F(x)$ is P_{F_j} and is assumed to be known, since it can be pre-computed. Thus, the corresponding AND gate is not really required. On the other hand, $F(x)$ can be a trinomial or a pentanomial and usually it can be chosen so that the parities of all parts become zero, i.e., $P_{F_j} = 0$ for $0 \leq j \leq k-1$. In this case, the value of $a_{k-1,l-1}$ is not important and one XOR gate is removed. In the worst case the circuit of Figure 4.3 can be implemented with 3 two-input XOR gates. The total number of two-input XOR gates for the whole parity prediction circuit is $3k$.

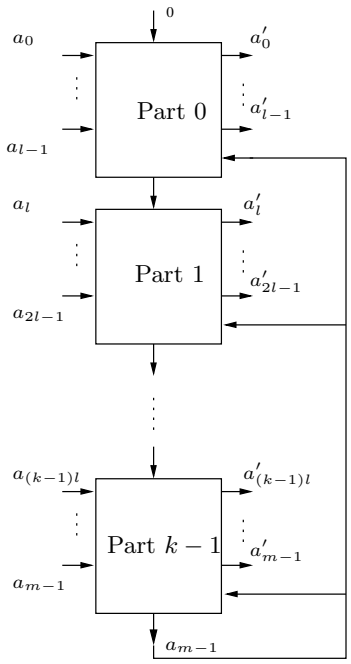


Figure 4.2: SR module

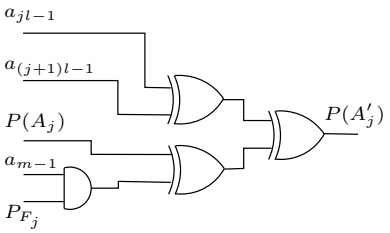


Figure 4.3: Parity prediction circuit of the j^{th} part of the SR module

Hereafter, an SR module together with its parity prediction circuit (PPC) is referred to as SR-P module. It should be mentioned that different partitioning of A and F can change the parity prediction circuit of the SR module. Section 4.4 presents a partitioning of A and F that reduces the number of XOR gates of each parity prediction circuit by two, i.e., parity prediction circuit can be constructed by only one XOR gate.

4.1.2 Parity Prediction in Scalar Multiplication and Vector Addition Modules

In this work, scalar multiplication refers to multiplication of an element of $GF(2)$ by an element of $GF(2^m)$ and vector addition refers to addition of two elements of $GF(2^m)$. For $b_i \in GF(2)$ and $A \in GF(2^m) = (a_0, a_1, \cdots, a_{m-1})$, scalar multiplication of b_i and A is $b_i.A = (b_i a_0, b_i a_1, \cdots, b_i a_{m-1})$. Thus,

$$\begin{aligned} P(b_i.A) &= b_i a_0 + b_i a_1 + \cdots + b_i a_{m-1} \\ &= b_i(a_0 + a_1 + \cdots + a_{m-1}) = b_i P(A). \end{aligned} \tag{4.2}$$

For $A, B \in GF(2^m)$, vector addition of A and B is:

$$A + B = \sum_{i=0}^{m-1} a_i x^i + \sum_{i=0}^{m-1} b_i x^i = \sum_{i=0}^{m-1} (a_i + b_i) x^i.$$

Thus,

$$\begin{aligned} P(A + B) &= \sum_{i=0}^{m-1} (a_i + b_i) = \sum_{i=0}^{m-1} a_i + \sum_{i=0}^{m-1} b_i \\ &= P(A) + P(B). \end{aligned} \tag{4.3}$$

The circuit of the parity prediction, as defined in (4.2) and (4.3), are shown in Figure 4.4 where they need k two-input AND gates and k two-input XOR gates, respectively. These circuits for parity bits are now included with the SM and the VA modules appropriately and the resulting new modules are hereafter referred to as SM-P and VA-P.

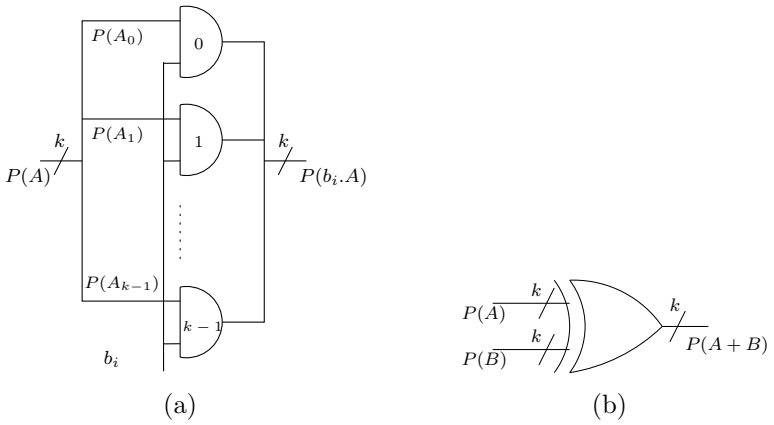


Figure 4.4: PPC for a) SM module and b) VA module

4.1.3 Parity Checking Circuit

In order to detect errors in the multiple parity scheme, the predicted parity bits should be compared with the corresponding actual parity bits. Actual parity bits are generated by parity generating circuit. Figure 4.5 shows the parity generator and the parity checker.

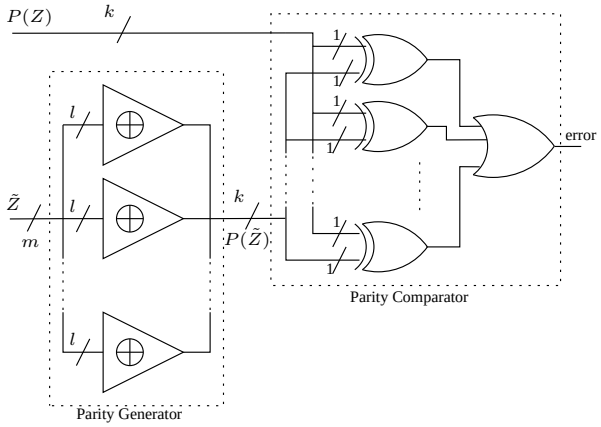


Figure 4.5: Multiple-bit parity checker

In Figure 4.5, Z and $\tilde{Z}$ can be considered as the expected and the actual outputs of one of the three modules discussed earlier. $P(Z)$ and $P(\tilde{Z})$ are k-bit parities of

Z and $\tilde{Z}$, respectively. The result of bit by bit comparison of $P(Z)$ and $P(\tilde{Z})$ are ORed to signal any difference which indicates an error. The parity generator is constructed by XOR trees which contain $l-1$ two-input XOR gates. Furthermore, k two-input XOR gates are required for comparison. Total numbers of two-input XOR and OR gates required for a parity checker are m $(= k(l-1)+k)$ and $k-1$, respectively.

4.1.4 Polynomial Basis Multiplier with CED

To construct a bit-serial and a bit-parallel multiplier with concurrent error detection capability, we will use PPC embedded modules SR-P, SM-P, and VA-P. Figure 4.6 shows a bit-serial multiplier with PPC. A and B are the inputs of the multiplier. Register D is initialized with A and its k-bit parity $P(A)$. A parity checker can be at each of the three locations: L1, L2 and L3. In the next section, the frequency of check points will be discussed.

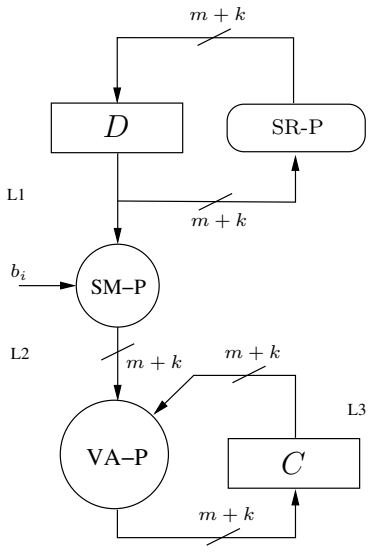


Figure 4.6: Bit-serial polynomial basis multipliers with parity prediction circuit

Figure 4.7 shows a bit-parallel multiplier with PPC. In the bit-parallel multiplier

a parity checker can be placed after each modules. Thus, there can be as many as $3m - 2$ error checkers for a bit-parallel multiplier.

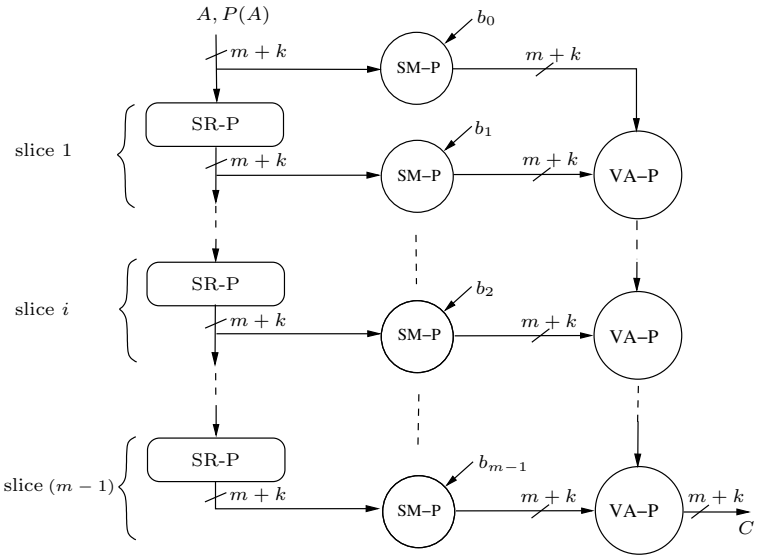


Figure 4.7: Bit-parallel polynomial basis multipliers with parity prediction circuit

4.2 Error Detection Capability

In this section, first the error model is explained. Then the probability of error detection at the output of the circuit using the multiple parity method is determined. Finally, the frequency of the check points is discussed.

4.2.1 Error Modelling

The effect of a fault, such as a transient fault, in one location of the multiplier circuit is modelled by XORing an error vector with the expected correct "value" of that location. The i^{th} bit of the error vector of a location being one implies that the i^{th} bit of the value of the location has changed from 0 to 1 or vice versa due to a fault.

If the location is one of the main components (SR-P, SM-P or VA-P), without loss of generality we can assume that the error vector should be XORed with the output of the component. It is worth mentioning that the parity prediction circuits, parity generators and parity checkers should be fault free or at least self-checking [53]. Since in practice the number of parity bits, k, is much less than the size of the input operands of the multiplier, m, the self-checking technique is feasible. In this work, these circuits are assumed to be fault free or self-checking. It will be shown in Section 4.3 that with a moderate number of parity bits the probability of error detection becomes quite close to unity. As an example, for $m = 163$, with 8 parity bits, the error detection probability is approximately 0.996.

Let $e = (e_0, e_1, \cdots, e_{m+k-1})$ be the representation of an error of a location in the multiplier. The first m bits of e correspond to errors in an element, say $A \in GF(2^m)$ that is part of the value of that location. The remaining k bits of e correspond to errors in the k-bit parity vector $P(A)$. Note that although we assume the parity prediction and the parity checking circuits to be fault free or self-checking, an error may occur in the parity bits any where in the remainder of the multiplier circuit such as the registers in the bit-serial implementation of the multiplier or the wires through which the parity signals propagate. If one assumes otherwise, i.e., the parity bits/signals are error free, then all registers and wires through which these signals travel have to be fault free, even though some of these registers and wires are *not* part of the parity prediction and checking circuits.

Since e is an $(m+k)$-tuple vector and the all-zero $e = (0, 0, \cdots, 0)$ corresponds to no error, the number of possible errors is $2^{m+k} - 1$. We logically divide e into k parts each of length $l + 1 = \frac{m}{k} + 1$ bits where the j^{th} part is

$$(e_{jl}, e_{jl+1}, \cdots, e_{jl+l-1}, e_{m+j}).$$

In the following, we investigate which kind of errors cannot be detected by the k-bit

parity scheme.

4.2.2 Probability of Error Detection

Let e_O be an odd parity error, i.e., the number of 1's in e_O is odd. Then the parity of at least one of the k partitions is odd. Therefore, e_O can be detected by the proposed CED method and the probability of undetected error is $Pr_U(e_O) = 0$.

Let e_E be a nonzero even parity error. Since $k < m$, there is at least one error, e_E, such that all of its partitions have even parity. Then the error cannot be detected. Accordingly, $Pr_U(e_E) \geq 0$.

Theorem 4.1 *Let k be the number of parity bits of the scheme. Suppose p is the probability that $e_i = 1$ for $0 \leq i \leq m + k - 1$. The probability of error detection is given as follows:*

$$Pr_D(e) = 1 - \left[\left(\frac{(1-2p)^{\frac{m}{k}+1} + 1}{2} \right)^k - (1-p)^{m+k} \right]. \tag{4.4}$$

Proof Our proof for Theorem 4.1 follows. $Pr_D = 1 - Pr_U$ where Pr_U is the probability of undetected errors. As it is mentioned, all nonzero errors with even parity in their partitions are undetectable. Thus, considering error vectors are $(m + k)$-bit long and each of them has k partitions, first we need to compute the probability of an $(\frac{m}{k} + 1)$-bit number with even parity.

Let E_i and O_i be the probabilities that an i-bit number has even parity and odd parity, respectively. Thus, $E_i = 1 - O_i$. Moreover, let q be the probability that a bit of the error vector is zero, i.e., $q = 1 - p$. We proceed in a recursive manner.

$$\begin{aligned} E_{i+1} &= qE_i + pO_i \\ &= (1-p)E_i + p(1-E_i) \\ &= (1-2p)E_i + p. \end{aligned}$$

Let $1-2p=A$ and $p=B$. We determine E_i for some i to find a closed formula:

$$\begin{aligned}
E_0 &= 1 \\
E_1 &= q \\
E_2 &= Aq + B \\
E_3 &= A^2q + AB + B \\
E_4 &= A^3q + A^2B + AB + B \\
&\vdots \\
E_i &= A^{i-1}q + A^{i-2}B + \cdots + AB + B \\
&= A^{i-1}q + B\left(\frac{A^{i-1}-1}{A-1}\right).
\end{aligned}$$

Now, we write the expression only in terms of p:

$$\begin{aligned}
E_i &= (1-2p)^{i-1}(1-p) + p\left(\frac{(1-2p)^{i-1}-1}{(1-2p)-1}\right) \\
&= (1-2p)^{i-1}(1-p) - \frac{(1-2p)^{i-1}-1}{2} \\
&= (1-2p)^{i-1}(1-p-1/2) + 1/2 \\
&= \frac{(1-2p)^i+1}{2}.
\end{aligned}$$

The probability that an $\left(\frac{m}{k}+1\right)$-bit partition of the error vector has even parity is $E_{i=\frac{m}{k}+1}$. Moreover, the partitions are independent. Thus, the probability of having a vector with even parity in each of its partitions is $\left(E_{i=\frac{m}{k}+1}\right)^k$ or

$$\left(\frac{(1-2p)^{\frac{m}{k}+1}+1}{2}\right)^k.$$

However, the zero vector should be excluded and hence,

$$Pr_U = \left(\frac{(1-2p)^{\frac{m}{k}+1}+1}{2}\right)^k - (1-p)^{m+k}.$$

As a result,

$$Pr_D = 1 - \left[\left(\frac{(1-2p)^{\frac{m}{k}+1}+1}{2}\right)^k - (1-p)^{m+k}\right]. \quad \blacksquare$$

As mentioned, p is the probability of an error vector bit being one. A reduction of p increases the probability of having an all-zero error vector. This reduction means a reduction in the probability of (nonzero) errors, which in turn means a reduction in the probability of undetectable errors. Thus, with a reduction in p, the probability of error detection increases.

As it can be determined from Equation (4.4), as the number of parity bits increases, the probability of error detection quickly approaches unity so that it reaches 0.996 for 8 parity bits.

4.2.3 Frequency of the Check Points

Suppose that there are several multiple-bit errors in a location of the circuit of a PB multiplier. For having an error detection capability Pr_D as given in Theorem 4.1, each of the above mentioned locations in Section 4.1.4 should have a parity checker. This causes a very high area overhead especially for bit-parallel multipliers. The following lemma helps us reduce the number of checkers considerably.

Lemma 4.2 *Suppose only a maximum of one multiple-bit error occurs per round of a bit-serial multiplier or per slice of a bit-parallel multiplier (see Figure 4.6 and Figure 4.7). Then any such error can be detected with the probability Pr_D, given in Section 4.2.2, using a parity checker at L3 of the bit-serial multiplier or a parity*

checker before the vertical input of every VA-P and one parity checker after the final VA-P in the bit-parallel multiplier.

Proof It should be verified if a detectable error vector can be changed to an undetectable one after passing through a main component and before reaching one of the check points.

If a detectable error vector passes through an SR-P module, it can be changed to an undetectable one. However, the check points are located so that any error vector can reach one of the check points without passing through any SR-P module. Therefore, one of the following cases should be considered: 1) a detectable error vector passes through an SM-P module or 2) a detectable error vector passes through a VA-P module or 3) both.

In the first case, if $b_i = 0$ then regardless of the other input value, the value of the output vector and parity are zero. This is a correct result and there is no error anymore. If $b_i = 1$ then the input and the output of the SM-P module are equal. Hence, the error vector passes SM-P without any change.

In the second case, if only one of the two inputs of VA-P module has erroneous bits, the error vector can pass the VA-P module without any change. Since a maximum of one multiple-bit error is allowed in a round of a bit-serial multiplier or in a slice of a bit-parallel multiplier, only one of the inputs of VA-P can be erroneous.

In the third case, the error must occur before an SM-P module but after the SR-P module (in the corresponding slice of a bit-parallel multiplier). Therefore, according to case 1 and case 2, it passes SM-P and VA-P modules and reaches the parity checker. ∎

4.3 Results

Important performance measures for an error detection scheme include error detection capability, area and time overheads. In this section the results of our studies on these measures are presented. The results can guide the choice of a proper number of parity bits for design requirements.

4.3.1 Simulation-Based Fault Injection

We have injected stuck-at faults to a $GF(2^{163})$ PB multiplier with $k = 8$ to evaluate the error detection capability of the proposed scheme. The fault injection was performed in a C model of the multiplier. Furthermore, the fault injection was at the gate-level, i.e., stuck-at faults (both stuck-at 1 and stuck-at 0) were injected at the input and output pins of the gates of the multiplier. In the proposed scheme, a checker is placed at the end of a round of a bit-serial multiplier (or at the end of the slice of a bit-parallel one). Moreover, the scheme can detect an error if the error can be detected in one round of a bit-serial multiplier (or a slice of a bit-parallel one). Fault injection in a complete multiplier of $GF(2^{163})$ is extremely time consuming. In order to reduce the time for completing experiments, faults were injected in only one round of a bit-serial multiplier (and a slice of a bit-parallel one). In Appendix A, more information about our procedures of the fault injections is given. In the following, two phases of our fault injections are presented.

Single-Bit Stuck-at Faults

In this experiment, single-bit stuck-at faults were injected at the input or output pins of gates. To inject a fault at a point, the conventional fault injection method at a gate pin described in Section 3.5 is also used here.

In a $GF(2^m)$ PB multiplier, there are $\omega - 2$ two-input XOR gates, m two-input

AND gates, and m two-input XOR gates in the SR, SM and VA modules, respectively, where ω is the Hamming weight of the field defining polynomial. Single-bit stuck-at faults are injected at all input and output pins except the output pins of AND gates of SM module because they are direct inputs of the VA module's XOR gates. Therefore, the number of locations for single-bit stuck-at fault injections at a round of a bit-serial (or a slice of a bit-parallel) multiplier is $3(\omega - 2) + 5m$. Additionally, for each input or output gate pin, two single-bit faults can be injected. Hence, the number of single-bit stuck-at faults that should be injected at a round of a bit-serial (or a slice of a bit-parallel) multiplier is $6(\omega - 2) + 10m$.

In this experiment, we simulated the multiplier for one million random inputs and for every input, all the above mentioned single-bit stuck-at faults were injected. The number of detected and masked faults are 656481969 and 991518031, respectively. Results are shown in Table 4.1.

Type of stuck-at faults	No. of stuck-at faults[1]	No. of random inputs	Percentage of error detection
Single-bit	1648	1000000	100%
Multiple-bit	500	1000000	99.61%

[1] in one round of a bit-serial (or one slice of a bit-parallel) multiplier

Table 4.1: Percentage of error detection of the SIMP scheme for a $GF(2^{163})$ PB multiplier against stuck-at faults

Multiple-Bit Stuck-at Faults

For multiple-bit stuck-at fault injection, the location of the above mentioned single-bit faults were randomly selected and a stuck-at 0 or stuck-at 1 was randomly injected there. Furthermore, simulations were performed for one million random inputs and for every input, 500 random multiple-bit stuck-at faults were injected. It is worth mentioning that for a $GF(2^{163})$ multiplier experiment, there are 824 single-bit stuck-at fault locations. The number of detected and undetected faults

are 498047234 and 1952766, respectively. As shown in Table 4.1, the percentage of error detection for multiple-bit stuck-at fault injections is 99.61%.

4.3.2 Time and Area Overheads

We have described the multiple-bit parity scheme by VHDL to obtain a realistic approximation of area and time overheads. In order to reduce the number of XOR gates in the multiplier, field defining polynomial $F(x)$ can be chosen to be a trinomial or a pentanomial such that the parity of $F(x)$ in each partition is zero, i.e., $P_{F_j} = 0$. In Section 4.4.2, the complexity of the parity prediction circuit for NIST recommended irreducible polynomials for ECDSA is discussed.

We used Modelsim$^{(TM)}$ to simulate the design for checking its correct functionality. We implemented the multiple parity scheme on a Xilinx Spartan 3 (XC3S5000) FPGA using Xilinx ISE 7.1i.

Bit-Serial PB Multiplication

The circuit of a complete bit-serial multiplier with CED is shown in Figure 4.8. The circuit consists of two major blocks: 1) PB multiplier with PPC and 2) checker. The parity generator of the checker is used at the initialization phase to generate the parity of input A. Note that no extra clock cycle is needed for the circuit shown in Figure 4.8 when compared to a bit-serial PB multiplier without CED.

From the first experiment, we obtained the area overhead percentage of the scheme for multipliers of different field sizes. The number of parity bits for this experiment was chosen to be 8 bits since the probability of error detection was within acceptable range for our experiment (≈ 0.996). Furthermore, the defining polynomial of the fields used in the experiment included the NIST recommended irreducible polynomials for ECDSA. Figure 4.9 shows the result of the experiment.

As shown in the figure, the area overhead for a fixed number of parity bits tends

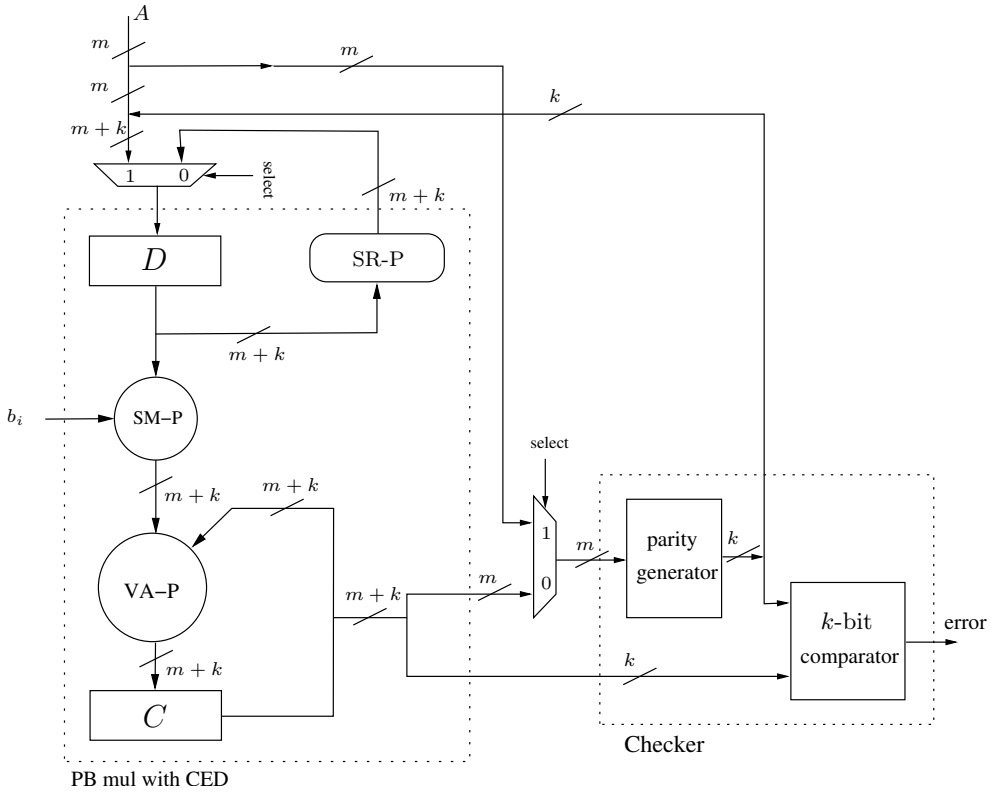


Figure 4.8: A complete bit-serial multiplier with CED

to decrease as the size of the field increases. The area overhead does not decrease in a strictly monotonic way because the FPGA compiler used in the experiment optimizes the multiplier for different field sizes differently. The worst area overhead percentage among the fields implemented is for $GF(2^{201})$ and is still reasonably low, i.e., $< 12\%$.

In the second experiment, we implemented the scheme for $m = 163$ and $m = 283$ using the NIST recommended field defining polynomials for ECDSA $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$ and $F(x) = x^{283} + x^{12} + x^7 + x^5 + 1$, respectively. Both of these polynomials are quite suitable for implementation because the parity prediction circuits of the scheme would be in the simplest form since, in a k-bit parity scheme,

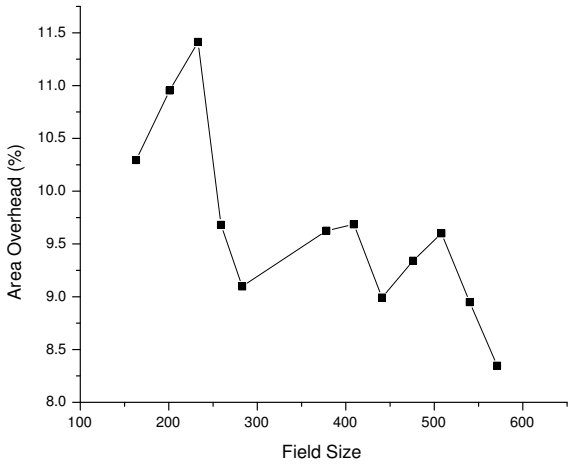


Figure 4.9: Area (i.e., slice) overhead for bit-serial PB multipliers for different size of fields

we have:

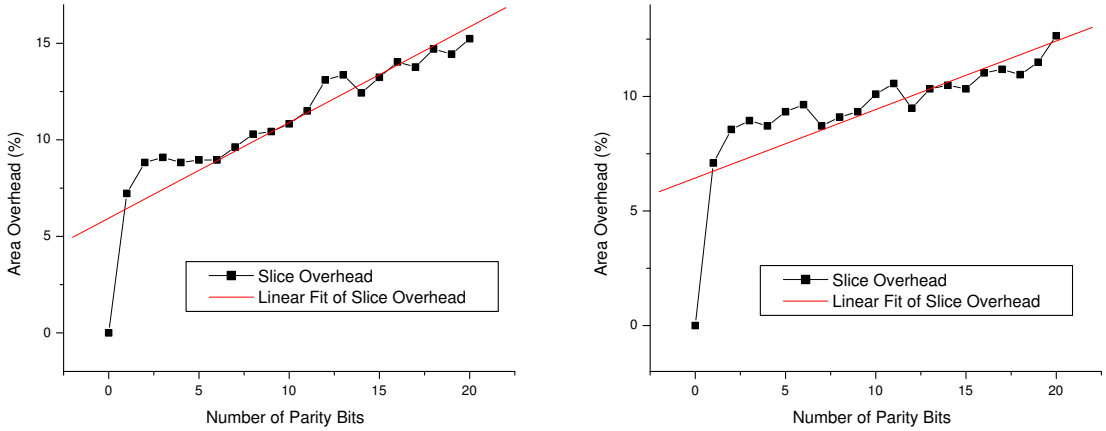
$$\{P(F_i) = 0 \mid 0 \leq i \leq k-1 \text{ and } 2 \leq k \leq 20\}.$$

As shown in Figure 4.10, area overhead cost increases as the number of parity bits increases. For all points in each graph depicted in the figure, a line is fitted as follows:

$$\begin{aligned} &\text{for } GF(2^{163}) : \text{overhead } (\%) = 0.50 \times (\#\text{ of parity bits}) + 5.94, \\ &\text{for } GF(2^{283}) : \text{overhead } (\%) = 0.30 \times (\#\text{ of parity bits}) + 6.44. \end{aligned} \tag{4.5}$$

As expected according to the first experiment, the slope of the fitted line for $GF(2^{163})$ is more than that for $GF(2^{283})$, i.e., the area overhead increase rate vs. parity-bit numbers in $GF(2^{283})$ is lower. Furthermore, based on the experimental results, area overhead tends to increase linearly except for very small numbers of parity bits.

Note that Equation (4.5) implies that even if one parity is used for each infor-



(a) $GF(2^{163})$ (b) $GF(2^{283})$

Figure 4.10: Area overhead vs. parity-bit number

mation bit, circuit overhead is not expected to be more than 100%, which is the overhead for the conventional dual modular redundant (DMR) scheme.

In the second experiment, we also investigated the time overhead of the $GF(2^{163})$ and $GF(2^{283})$ PB multipliers for different numbers of parity bits. Since there is no extra clock cycle, the time overhead is equal to the clock period overhead. We obtain the clock periods from the post place and route static timing report of Xilinx ISE. Except for four cases, there was no clock period overhead and in turn no time overhead for the bit-serial implementation of the multipliers. These four cases belong to the $GF(2^{163})$ PB multiplier shown in Table 4.2. According to the table, the time overheads even for these cases are small.

No. of parity bits	1	4	11	13
Time overhead (%)	12.27	4.39	15.26	4.79

Table 4.2: Nonzero time overheads for bit-serial implementation which belong to the $GF(2^{163})$ PB multiplier

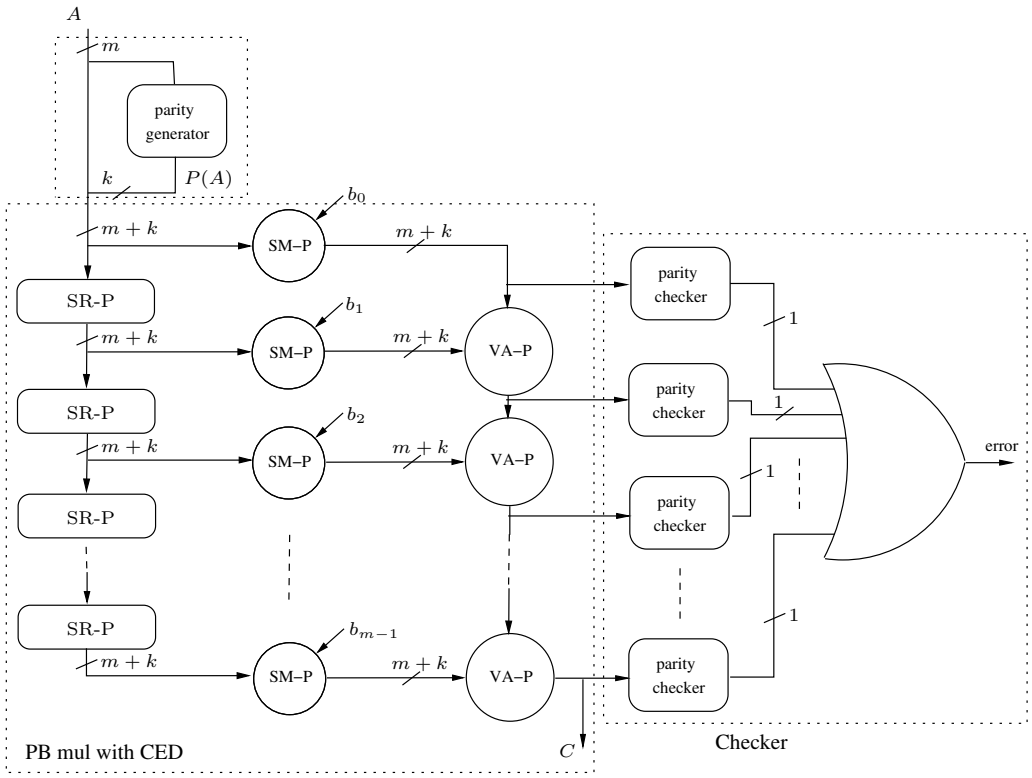


Figure 4.11: A complete bit-parallel multiplier with CED

Bit-Parallel PB Multiplication

A circuit diagram of a complete bit-parallel polynomial basis multiplier with CED is depicted in Figure 4.11. The parity checker is very similar to that presented in Figure 4.8. As shown in Figure 4.11, once the inputs A and B are updated, the results of the multiplication and error detection are ready after certain amount of delay due to the propagation of various signals through the circuit where no clocking is used.

For bit-parallel multiplier, the first experiment was to measure the area overhead percentage of the eight parity-bit scheme for multipliers of different field sizes. The results show that the area overhead decreases as the field size increases (Figure 4.12).

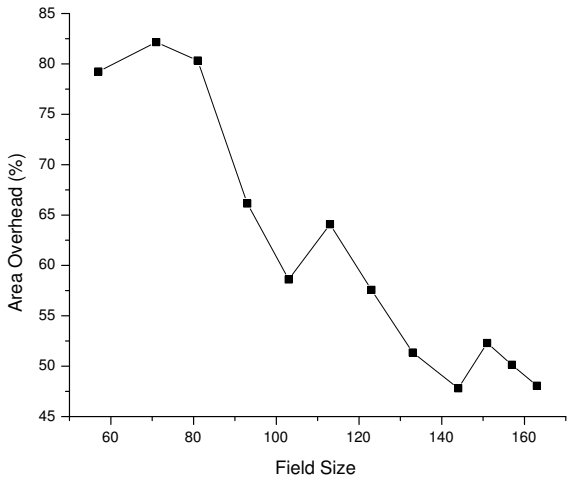


Figure 4.12: Area (i.e., slice) overhead for bit-parallel PB multipliers for different size of fields

There is a major difference between the structure of bit-serial and bit-parallel PB multipliers and this affects the area overhead considerably. A bit-serial PB multiplier contains simple and shift registers, but a bit-parallel multiplier does not. Basically, registers are relatively area consuming components in FPGAs. Therefore, assuming that one wants to implement a PB multiplier for a field of size m, the area (in terms of slices) needed for a bit-parallel PB multiplier without CED is significantly smaller than m times the area needed for a bit-serial multiplier. Accordingly, CED overhead on a bit-parallel PB multiplier is much higher than that on a bit-serial one. This fact can be observed easily in the experiments reported in this section.

The second experiment was to investigate the area and time overheads' increase rates vs. the number of parity bits for the field $GF(2^{163})$ (see Figure 4.13). The field defining polynomial is $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$. According to Table 4.3, the bit-parallel implementation is very area consuming; therefore, similar experiments

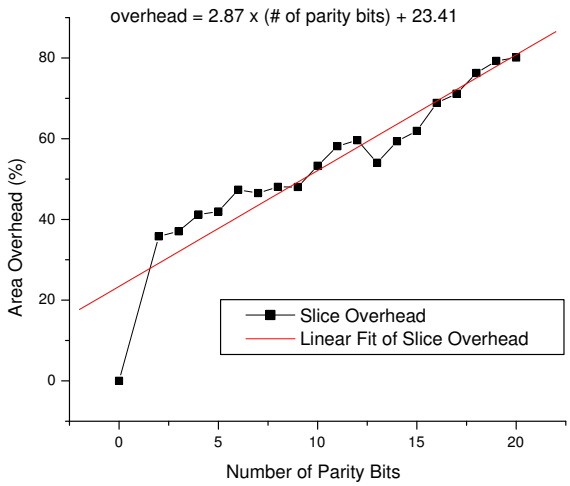


Figure 4.13: Area overhead vs. parity-bit number for the field $GF(2^{163})$

for the field $GF(2^{283})$ are extremely time consuming and clearly that design does not fit into our current FPGA. However, the area overhead results for higher values of m are expected to be better than the result of this experiment as one can infer from Figure 4.12, where the number of parity bits is fixed to eight.

No. of parity bits	Without CED	4	8	12	16	20
Number of required Slices	13541	19121	20049	21616	22864	24390
FPGA area consumption (%)[1]	40.69	57.45	60.24	64.95	68.70	73.29

[1]The total number of slices in a Xilinx Spartan 3 (XC3S5000) FPGA is 33280.

Table 4.3: FPGA area consumption for a bit-parallel $GF(2^{163})$ PB multiplier

Figure 4.13 illustrates that as the number of parity bits increases, the area overhead for a bit-parallel implementation increases at a greater rate compared to the bit-serial implementation. However, the area overhead may be still acceptable for some applications. This is because for obtaining a sufficiently high probability

of error detection (say ≈ 0.996), one needs only about 8 parity bits in the proposed scheme and it results in about 50% area overhead, which is better than 100% overhead of the DMR scheme.

In the bit-parallel implementation, the time overhead is the delay of the critical path, i.e., the maximum propagation delay from one of the input pins to one of the output pins. We obtain the delay of all input pins to output pins from the post place and route static timing report of Xilinx ISE. The time overhead for the bit-parallel implementation of a $GF(2^{163})$ PB multiplier vs. number of parity bits is given in Fig 4.14, which shows that the time overhead is generally less than 25% when more than a couple of parity bits are used.

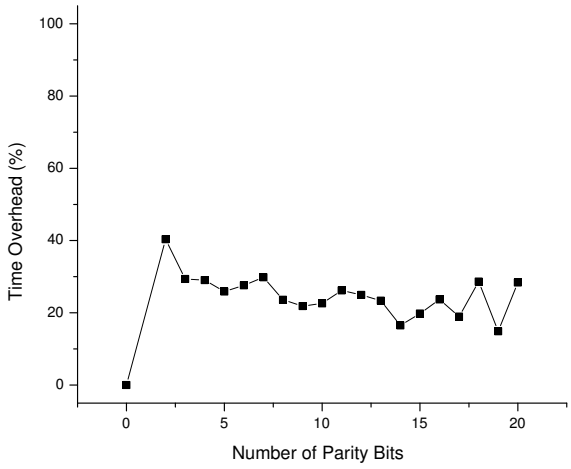


Figure 4.14: Time overhead vs. parity-bit number for the field $GF(2^{163})$

4.4 Alternative Partitioning

In this section another partitioning of A and F is presented. The new partitioning reduces the overhead of the parity prediction circuit of the SR module.

As mentioned $A = \sum_{i=0}^{m-1} a_i x^i$ is partitioned into k parts. As before, we assume that m is divisible by k and $l = m/k$. The alternative (vertical) partitioning is illustrated below:

$$\begin{array}{ccccccccc}
a_0 & , & a_1 & , & a_2 & , & \cdots & , & a_{k-1} & , \\
a_k & , & a_{k+1} & , & a_{k+2} & , & \cdots & , & a_{2k-1} & , \\
\vdots & , & & , & \ddots & , & & , & \vdots & , \\
a_{(l-1)k} & , & a_{(l-1)k+1} & , & a_{(l-1)k+2} & , & \cdots & , & a_{lk-1} & \\
\underbrace{\qquad} & & \underbrace{\qquad} & & \underbrace{\qquad} & & & & \underbrace{\qquad} & \\
A_0 & & A_1 & & A_2 & & \cdots & & A_{k-1} &
\end{array}$$

For $0 \leq j \leq k-1$, the j^{th} partition is:

$$A_j = \sum_{i=0}^{l-1} a_{ik+j} x^{ik+j} = (a_j, a_{k+j}, a_{2k+j}, \cdots, a_{(l-1)k+j}).$$

4.4.1 Structure of SR Module

$$
\begin{aligned}
A' =& xA \mod F(x) \\
=& \sum_{j=0}^{k-1} \sum_{i=0}^{l-1} a_{ik+j} x^{ik+j+1} \mod F(x) \\
=& \sum_{j=1}^{k} \sum_{i=0}^{l-1} a_{ik+j-1} x^{ik+j} \mod F(x) \\
=& \sum_{j=1}^{k-1} \sum_{i=0}^{l-1} a_{ik+j-1} x^{ik+j} + \sum_{i=0}^{l-2} a_{k(i+1)-1} x^{k(i+1)} \\
& + (a_{m-1} x^m \mod F(x)) \\
=& \sum_{j=1}^{k-1} \sum_{i=0}^{l-1} a_{ik+j-1} x^{ik+j} + \sum_{i=1}^{l-1} a_{ki-1} x^{ki} + a_{m-1} \sum_{i=0}^{m-1} f_i x^i \\
=& \sum_{j=1}^{k-1} \sum_{i=0}^{l-1} \left(a_{ik+j-1} + a_{m-1} f_{ik+j}\right) x^{ik+j} \\
& + \sum_{i=0}^{l-1} \left(a_{ki-1} + a_{m-1} f_{ki}\right) x^{ki} \\
=& \sum_{j=0}^{k-1} \sum_{i=0}^{l-1} \left(a_{ik+j-1} + a_{m-1} f_{ik+j}\right) x^{ik+j}
\end{aligned}
\tag{4.6}
$$

where $a_{-1} = 0$.

Figure 4.15 shows the j^{th} part of the SR module. The complete SR module is shown in Figure 4.16. The number of gates is exactly the same as for the previous SR module mentioned in Section 4.1.1, as only the position of the coordinates is changed.

The following lemma discusses parity prediction in the j^{th} part of the SR module.

Lemma 4.3 *Let $P(A_j)$ and $P(A'_j)$ be the input and the expected output parities of the j^{th} part of the SR module, respectively and $P_{F_j} = \sum_{i=0}^{l-1} f_{ik+j}$. Then,*

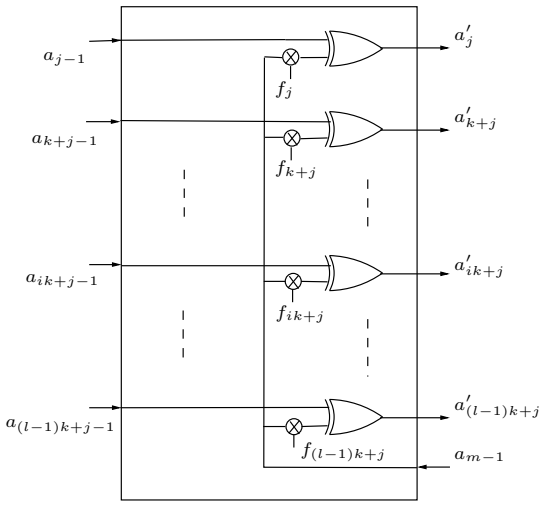


Figure 4.15: The j^{th} part of the SR module using the vertical partitioning

$$P(A'_j) = \begin{cases} P(A_{j-1}) + a_{m-1}P_{F_j} & if \;\; 1 \leq j \leq k-1, \\ P(A_{k-1}) + a_{m-1}(P_{F_0}+1) & if \;\; j = 0. \end{cases}$$

Proof According to (4.6), we have:

$$A'_j = \sum_{i=0}^{l-1} \left(a_{ik+j-1} + a_{m-1}f_{ik+j}\right) x^{ik+j}.$$

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

$$P(A'_j) = P\left(\sum_{i=0}^{l-1} a_{ik+j-1}x^{ik+j}\right) + P\left(\sum_{i=0}^{l-1} a_{m-1}f_{ik+j}x^{ik+j}\right) = P(A_{j-1}) + a_{m-1}P_{F_j}.$$

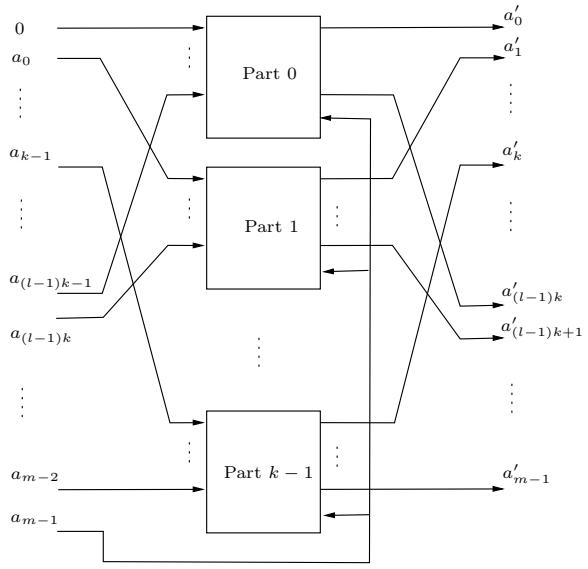


Figure 4.16: SR module

For $j = 0$, we have:

$$\begin{aligned} P(A'_0) &= P\left(\sum_{i=0}^{l-1} a_{ik-1}x^{ik}\right) + P\left(\sum_{i=0}^{l-1} a_{m-1}f_{ik}x^{ik}\right) \\ &= (P(A_{k-1}) + a_{m-1}) + a_{m-1}P_{F_0} \\ &= P(A_{k-1}) + a_{m-1}(P_{F_0} + 1). \quad \blacksquare \end{aligned}$$

P_{F_j}'s can be pre-computed. Therefore, the maximum number of gates required for the parity prediction circuit of each part of the SR module is one XOR gate. No XOR gate is needed for the parity prediction circuit of a part of the SR module when $P_{F_0} = 1$ or $P_{F_j} = 0$ for $0 < j < k$. Furthermore, the probability of error detection can be computed by Theorem 4.1, since the conditions are the same.

Irreducible polynomials $F(x)$	No. of nonzero-parity partitions		No. of 2-input XOR gates for PPC of SR-P	
	Horizontal partitioning	Vertical partitioning	Horizontal partitioning	Vertical partitioning
$x^{163}+x^7+x^6+x^3+1$	0	4	15	3
$x^{233}+x^{74}+1$	2	2	17	1
$x^{283}+x^{12}+x^7+x^5+1$	0	4	15	3
$x^{409}+x^{87}+1$	2	2	17	1
$x^{571}+x^{10}+x^5+x^2+1$	0	2	15	1

Table 4.4: XOR counts for PPC of an SR module for NIST recommended irreducible polynomials for ECDSA application

4.4.2 Comparison of SR-P Modules

According to Section 4.3.1, the scheme with eight partitions results in a fairly high probability of error detection for values of m that are of interest for elliptic curve cryptosystems. Therefore, we have divided each of corresponding NIST recommended irreducible polynomials into eight partitions using our horizontal and vertical partitioning methods. Table 4.4 gives the number of partitions with nonzero parity and the number of required two-input XOR gates for PPC of the SR module along with the NIST recommended irreducible polynomials.

As it can be seen in Table 4.4, the SR-P module is relatively area efficient in the vertical paritioning than the horizontal partitioning. However, the SR-P module is much less resource consuming than any of the SM-P and VA-P modules. Therefore, the overheads resulting from the vertical partitioning are expected to be very similar to those presented in Section 4.3 for horizontal partitioning.

4.5 Summary

In this work, a multiple parity error detection scheme is presented for concurrent detection of errors in polynomial basis multipliers. In this scheme, the probability of error detection for random errors is more than 75% and it quickly approaches unity for approximately 8 parity bits. The overhead of our implementation tends to increase linearly as the number of parity bits increases. Results show that the area overhead cost of the bit-serial implementation is lower than that for the bit-parallel one. Both implementations have lower area overheads than the traditional dual modular redundant scheme for a sufficient number of parity bits. Additionally, the average time overhead due to the use of the scheme in bit-parallel implementations is around 25%, while for bit-serial implementations time overheads have been observed to be small to negligible. It is hoped that using the results presented in this chapter, one will be able to choose an appropriate number of parity bits for specific applications.

Chapter 5

Extending SIMP

In the previous chapter, a parity based error detection scheme, referred to as SIMP, for PB multipliers is presented. This chapter extends the SIMP scheme by partitioning both inputs of the multiplier and considering a parity bit for each partition. This scheme is referred to as double input multiple parity (DIMP) scheme. This work can be applied to any finite field GF(2^m). This scheme has a better error detection capability than SIMP because the latter cannot detect errors in one of the inputs of the multiplier. This improvement in error detection capability is achieved with a slightly more area overhead and quite similar time overhead compared to SIMP. This scheme can detect multiple-bit errors in both digit-serial and bit-parallel polynomial basis multipliers over binary extension fields.

In this chapter, the capability of the proposed error detection schemes is evaluated by simulation-based fault injection. Additionally, experimental analyses of the area and the time of the scheme are presented. Our results show that the area overhead has a linear increasing trend as the number of parity bits increases but the probability of undetected errors decreases very quickly. Additionally, the area overhead for the bit-parallel implementation is considered to be in an acceptable range, e.g., having eight and three parity bits for the first and the second inputs,

respectively, the scheme obtains an area overhead of 55.59% which is lower than the conventional *dual modular redundant* systems. The average time overhead due to the use of the scheme in bit-parallel implementations is approximately 25%.

Towards the end of this chapter, both SIMP and DIMP schemes are also applied to field arithmetic using dual, type I and type II optimal normal bases.

The organization of the remainder of this chapter is as follows. In Section 5.1, the double input multiple parity scheme is investigated. The experimental results are given in Section 5.2. Section 5.3 extends SIMP and and the scheme presented in this chapter to dual and optimal normal Bases. Finally, Section 5.4 gives a summary of the chapter.

Part of this work appeared in [6].

5.1 Double Input Multiple Parity (DIMP) Scheme

In the SIMP scheme, the parity bits are preserved for only one input of the PB multiplier. Although this can detect errors on the first input (input A) and/or inside of the multiplier with a certain probability, the errors on the second input cannot be detected. One way to improve this situation is to consider other parity bits for the second input operand as well. Therefore, the second input can be similarly divided, say, into s partitions and for each partition one parity bit can be considered. In general, the number of partitions of the first and the second inputs can be different. Additionally, due to the structure of the scheme, this scheme can be applied to digit-serial and bit-parallel implementations and is not suitable and efficient for bit-serial implementations. In this work, we investigate the double input multiple parity (DIMP) scheme using a bit-parallel implementation.

5.1.1 Parity Prediction in DIMP

This section investigates how the parity bits of the second input, $B \in GF(2^m)$, can propagate through SR, SM and VA modules. The first input, $A \in GF(2^m)$, was already divided into k parts. We assume that $k|m$ and the length of each part is $l = \frac{m}{k}$ (see Section 4.1 for the case that m is not divisible by k). Also, the parity propagation through the SR module for the SIMP scheme, which has been investigated in Section 4.1.1, is summarized in Lemma 5.1.

Lemma 5.1 *Suppose that $A^{(n-1)}$ and $A^{(n)}$ are the input and output of the n^{th} SR module in the multiplier for $1 \leq n \leq m-1$. Then we have:*

$$A^{(n)} = \sum_{j=0}^{k-1} x^{jl} \left(a_{jl-1}^{(n-1)} + \sum_{i=1}^{l-1} a_{jl+i-1}^{(n-1)} x^i + a_{m-1}^{(n-1)} \sum_{i=0}^{l-1} f_{jl+i} x^i \right),$$

$$P(A_j^{(n)}) = a_{jl-1}^{(n-1)} + P(A_j^{(n-1)}) + a_{(j+1)l-1}^{(n-1)} + a_{m-1}^{(n-1)} P(F_j),$$

where $0 \leq j \leq k-1$, $A_j^{(0)} = A_j$ and $a_{-1}^{(n-1)} = 0$.

Now for DIMP, let us assume that m is divisible by s and the length of each partition is t; for the case that m is not divisible by s, the lengths of the partitions are different but a similar argument given in the following theorem holds. Theorem 5.1 helps us to predict the parity of every t^{th} slice of the multiplier using the parities of both inputs A and B.

Theorem 5.1 *Let $P(C_j^{(q)})$ be the parity of the j^{th} partition of the output of the VA module in slice $(q+1)t$ for $0 \leq q \leq s-1$. Let $P(B^{(q)}) = \sum_{i=0}^{q} P(B_i)$, where $P(B_i)$ is the parity of the i^{th} partition of B. Then we have:*

$$P(C_j^{(q)}) = P(A_j)P(B^{(q)}) + \sum_{i=1}^{(q+1)t-1} b_i h_j^{(i)},$$

where $0 \leq j \leq k-1$, $A_j^{(0)} = A_j$ *and* $h_j^{(i)} = \sum_{k=0}^{i-1} a_{jl-1}^{(k)} + a_{(j+1)l-1}^{(k)} + a_{m-1}^{(k)} P(F_j)$.

Proof According to Lemma 5.1, for $1 \leq i \leq m-1$, we have:

$$P(A_j^{(i)}) = a_{jl-1}^{(i-1)} + P(A_j^{(i-1)}) + a_{(j+1)l-1}^{(i-1)} + a_{m-1}^{(i-1)} P(F_j).$$

One can unroll the above recursive formula as follows given $A_j^{(0)} = A_j$:

$$P(A_j^{(i)}) = P(A_j) + \sum_{k=0}^{i-1} a_{jl-1}^{(k)} + a_{(j+1)l-1}^{(k)} + a_{m-1}^{(k)} P(F_j). \tag{5.1}$$

Now, we define:

$$h_j^{(i)} = \begin{cases} 0 & ; \quad i = 0, \\ \sum_{k=0}^{i-1} a_{jl-1}^{(k)} + a_{(j+1)l-1}^{(k)} + a_{m-1}^{(k)} P(F_j) & ; \quad 1 \leq i \leq m-1. \end{cases}$$

Therefore, expression (5.1) can be rewritten as:

$$P(A_j^{(i)}) = P(A_j) + h_j^{(i)}. \tag{5.2}$$

Since parity function $P()$ is linear and considering expression (2.5), we have:

$$P(C) = \sum_{i=0}^{m-1} b_i P(A^{(i)}).$$

Therefore, the parity of the j^{th} partition in the SIMP scheme can be computed as:

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

Additionally, $P(C_j)$ is the parity of the j^{th} partition after the VA module of the last slice, i.e., the slice $(m-1)$. Accordingly, such parity of an arbitrary slice, say r, can be computed simply by substituting m by r. In fact, we are interested in

computing such parity for every t^{th} slice, i.e., $r = (q+1)t$ where $0 \leq q \leq s-1$, as follows:

$$\begin{aligned} P(C_j{}^{(q)}) &= \sum_{i=0}^{(q+1)t-1} b_i P(A_j^{(i)}) \\ &= b_0 P(A_j) + \sum_{i=1}^{(q+1)t-1} b_i P(A_j^{(i)}). \end{aligned} \tag{5.3}$$

Substituting (5.2) to (5.3), we have:

$$\begin{aligned} P(C_j{}^{(q)}) &= b_0 P(A_j) + \sum_{i=1}^{(q+1)t-1} b_i \cdot \left(P(A_j) + h^{(i)}\right) \\ &= b_0 P(A_j) + \sum_{i=1}^{(q+1)t-1} b_i P(A_j) + b_i h^{(i)} \\ &= \sum_{i=0}^{(q+1)t-1} b_i P(A_j) + \sum_{i=1}^{(q+1)t-1} b_i h^{(i)} \\ &= P(A_j) \sum_{i=0}^{(q+1)t-1} b_i + \sum_{i=1}^{(q+1)t-1} b_i h^{(i)} \\ &= P(A_j) P(B^{(q)}) + \sum_{i=1}^{(q+1)t-1} b_i h^{(i)}. \quad \blacksquare \end{aligned} \tag{5.4}$$

In the following, based on the theorem, the parity prediction circuits (PPCs) for every module are presented. Figure 5.1 and Figure 5.2 show the PPCs of the j^{th} parts of the SR and SM modules, respectively. In Figure 5.1, $P(F_j)$ is fixed. Therefore, if $P(F_j) = 0$ then the circuit needs only three 2-input XOR gates.

Additionally, in Figure 5.2, the last inputs of the PPCs (i.e., $\sum_{k=1}^{i-1} b_k h^{(k)}$) for the SM modules of the first two slices of the PB multiplier (i.e., for $i = 0, 1$) are zero. As a result, the second output of the first SM module (i.e., $\sum_{k=1}^{i} b_k h^{(k)}$ for $i = 0$) is zero as well. For each SR and each SM modules, k copies of the PPCs are

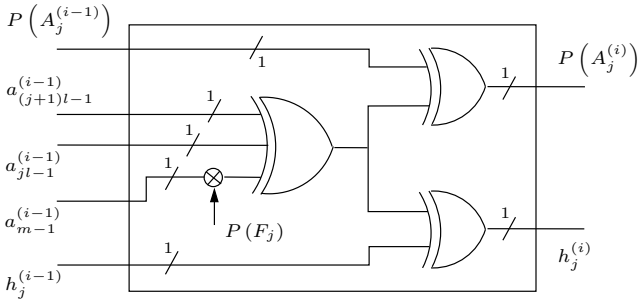


Figure 5.1: PPC of the j^{th} part of an SR module

required. These modules along with their PPCs are hereafter referred to as SR-P2 and SM-P2 modules, respectively.

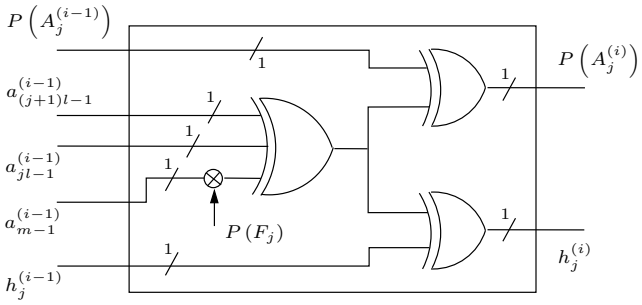


Figure 5.2: PPC of the j^{th} part of an SM module

Unlike SR and SM modules, two types of PPCs are needed for VA modules. Figure 5.3(a) shows the circuit of the first type of PPCs for VA modules, which predicts the parities related to both inputs A and B. This circuit should be incorporated in the VA modules of every t^{th} slice.

The second type of PPCs of VA modules only predicts the parity related to the first input, A. The circuit of this type is shown in Figure 5.3(b) and should be incorporated in all VA modules but every t^{th} ones. The VA modules along with these PPCs are the same as VA-P modules in the SIMP scheme. It is worth mentioning that since the first slice of a PB multiplier does not contain VA module (see Figure 1.b), the circuit shown in Fig 5.3(b) does not exist for the first slice, i.e., for $i = 0$. Additionally, $P_{\mathrm{VA}}(A_j^{(0)}) = P_{\mathrm{SM}}(A_j^{(0)})$.

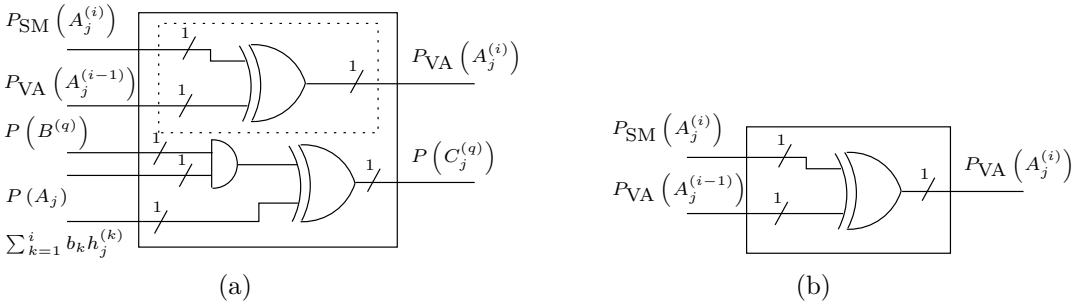


Figure 5.3: PPC of the j^{th} part of VA for (a) every t^{th} slice, (b) all other slices

5.1.2 Polynomial Basis Multipliers with CED Using DIMP

To construct a PB multiplier with CED using the DIMP scheme, one should use an SR-P2, an SM-P2 and a VA-P2 modules for every t^{th} slice as shown in Figure 5.4

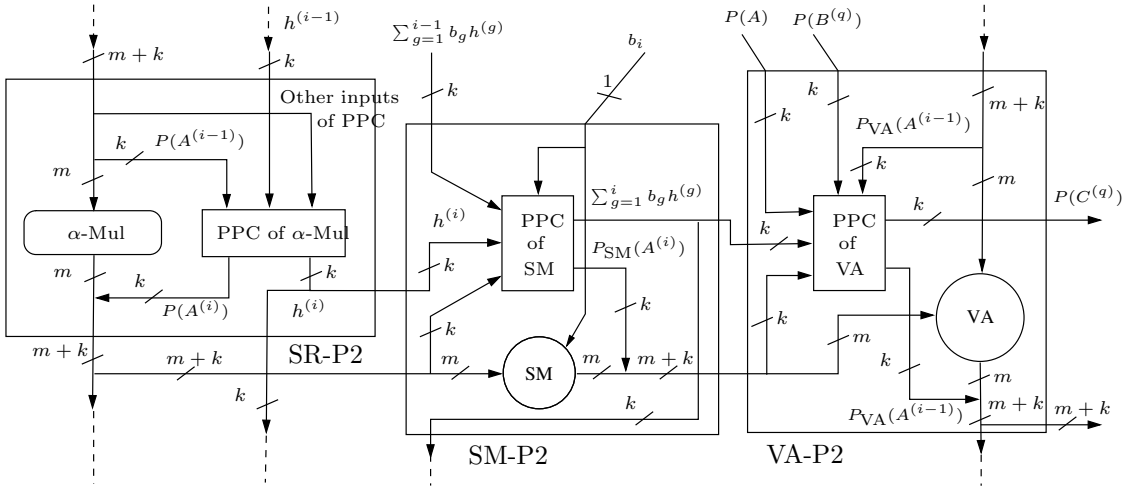


Figure 5.4: One of every t^{th} slices of the PB multiplier with CED

Furthermore, in any slice of the PB multiplier except every t^{th} slice, VA-P modules should be used instead of VA-P2 ones. For the purpose of error detection, a number of parity equality checkers are also required to be placed at the end of each slice of the PB multiplier before the first VA module and after other ones (see Section 4.2.3 for more information about the frequency of the check points).

Two types of checkers are needed in DIMP as shown in Figure 5.5. In the figure, $P(Z)$ is the k-bit predicted parity related to the first input and is available at the end of every slice. However, $P'(Z)$ is the k-bit predicted parity related to the second input and is available at every t^{th} slice. Moreover, $P(\tilde{Z})$ is the k-bit actual parity. As shown in the figure, an ordinary parity checker (OPC) only checks the equality of the predicted parities related to input A against the actual ones. However, a double parity checker (DPC) checks the equality of the predicted parities of both inputs against the actual ones. DPCs are placed at the end of every t^{th} slice after the VA modules while OPCs are placed at the end of all other slices.

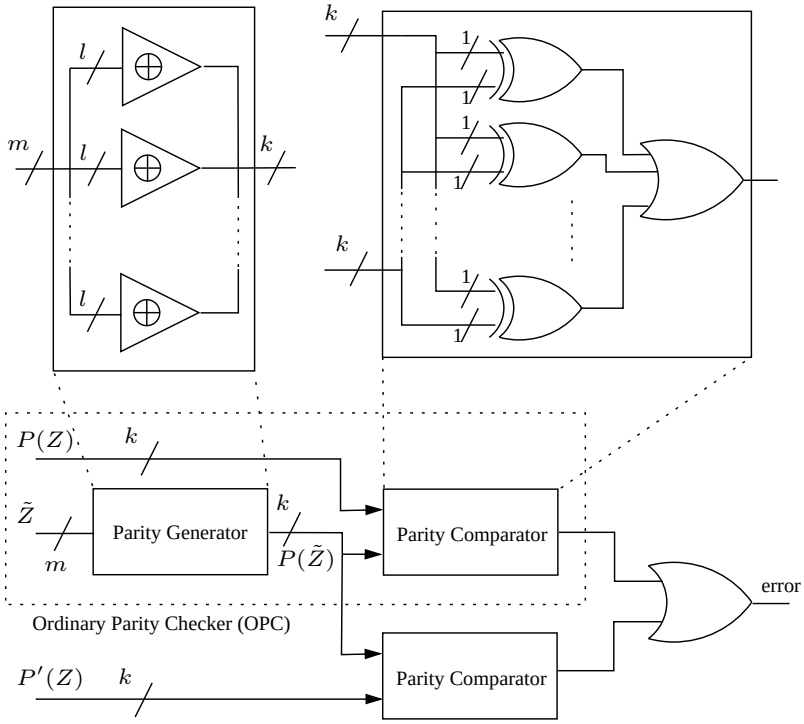


Figure 5.5: Double parity checker (DPC)

It is worth mentioning that one of the inputs of the second type PPC for VA module as shown in Figure 5.3(a) is $P(B^{(q)})$, where $0 \leq q \leq s-1$. This value is the parity of the first $q+1$ parts of input B and is referred to as cumulative parity of

the first $q+1$ partitions. Figure 5.6 shows a circuit that generates the cumulative parities.

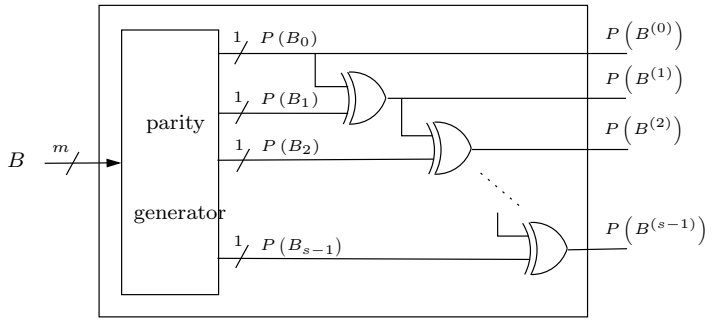


Figure 5.6: Cumulative parity generator

5.1.3 CED Capability of DIMP

As mentioned in Section 4.2, suppose that the expected value of one location of the multiplier circuit is changed due to one fault or more. Then this change can be modelled by XORing an error vector with the expected correct value of that location; i.e. the error model of this work is bit-flip. If the location is one of the SR, SM or VA modules, without loss of generality we can assume that the error vector should be XORed with the output of that module. Furthermore, the parity prediction circuits, parity generators and parity checkers should be fault free or at least self-checking [53]. In this work, these circuits are assumed to be fault free although the self-checking technique is feasible because the number of parity bits is practically much less than the size of the input operands of the multiplier. Now, considering this error model, a multiple-bit error on input B can be detected if at least one of the s partitions of B is not zero. Furthermore, if the parities of all partitions of input A are zero, then errors on input B cannot be detected. This can be easily inferred from Theorem 5.1.

Lemma 5.2 *Suppose e' is a random error of length m that occurs on input B. Assuming that input A of the multiplier is random, the probability of detecting the error e' on B is:*

$$Pr_D(e') = \left[1 - \left(\frac{(1-2p)^{\frac{m}{k}} + 1}{2}\right)^k\right] \cdot \left(1 - \left[\left(\frac{(1-2p)^{\frac{m}{s}} + 1}{2}\right)^s - (1-p)^m\right]\right),$$

where p is the probability that a bit of an m-bit vector, either e' or A, becomes one.

Proof Let event1 be that the parity of all k partitions of input A are zero. Also, let event2 be that an m-bit error with s partitions has at least one partition with nonzero parity. According to Theorem 4.1, the probability of having an m-bit vector with j partitions whose parities are all zero is:

$$\left(\frac{(1-2p)^{\frac{m}{j}} + 1}{2}\right)^j.$$

Therefore,

$$Pr(event1) = \left(\frac{(1-2p)^{\frac{m}{k}} + 1}{2}\right)^k.$$

Furthermore, the probability of the complement of event2 is:

$$Pr(\overline{event2}) = \left(\frac{(1-2p)^{\frac{m}{s}} + 1}{2}\right)^s - (1-p)^m,$$

where $(1-p)^m$ is the probability of having a zero error vector. Hence,

$$Pr(event2) = 1 - \left[\left(\frac{(1-2p)^{\frac{m}{s}} + 1}{2}\right)^s - (1-p)^m\right]. \tag{5.5}$$

On the other hand, the probability of detecting the error e' using DIMP is:

$$Pr_D(e') = Pr(event1) \cdot Pr(event2|event1) + Pr(\overline{event1}) \cdot Pr(event2|\overline{event1}).$$

Since $Pr(event2|event1) = 0$ and $Pr(event2|\overline{event1}) = Pr(event2)$, we have:

$$Pr_D(e') = Pr(\overline{event1}) \cdot Pr(event2),$$

where $Pr(\overline{event1}) = 1 - \left(\frac{(1-2p)^{\frac{m}{k}}+1}{2}\right)^k$ and $Pr(event2)$ is given in (5.5). ∎

Now, Let us similarly assume that for a PB multiplier using the DIMP scheme only a maximum of one multiple-bit error occurs per slice of a bit-parallel implementation, then we have:

- if errors occur on input A and/or inside the PB multiplier, they can be detected with a probability given in (4.4).
- if an error occurs on input B, it can be detected with the probability given in Lemma 5.2.

Note that the number of partitions of B, i.e., s, can be preferably chosen smaller than that of A, i.e., k; since the parity checking mechanism in the second input is used for detecting errors in itself, i.e., input B. Furthermore, this choice slightly decreases the area overhead of the scheme.

5.2 Experimental Results

5.2.1 Simulation-Based Fault Injection

We injected a large number of stuck-at faults into a C model of a $GF(2^{163})$ bit-parallel PB multiplier to evaluate the error detection capability of the DIMP scheme and compare it against SIMP. Since fault injection in a complete PB multiplier is extremely time consuming, we performed fault injection in a slice of the PB multiplier. We injected the faults at the inputs and the outputs of the gates of a

slice of the PB multiplier. Additionally, to show the strength of DIMP rather than SIMP, we performed fault injection at the b_i input of the SM module as well.

In a slice of a $GF(2^m)$ PB multiplier, the number of two-input gates for SR, SM and VA modules are $(\omega - 2)$ XOR gates, m AND gates and m XOR gates, respectively, where ω is the Hamming weight of the field defining polynomial. Additionally, except the outputs of AND gates of the SM module, where are the direct inputs of XOR gates of VA modules, all other inputs and outputs can be locations for fault injection. Therefore, considering the b_i input of the SM module, the number of locations for fault injection in a slice of the PB multiplier is $3(\omega-2)+5m+1$. For more information about our procedures of the fault injections, see Appendix A.

Single Stuck-at Fault Injection

For single stuck-at fault injection, every pin location was injected by two fault values zero and one. Hence, the total number of injected faults in a slice of the $GF(2^m)$ PB multiplier was $6(\omega - 2) + 10m + 2$. In this experiment, we injected all above-mentioned faults for one million random inputs. The number of detected, masked and undetected faults are 656517171, 992482830 and 999999 for the SIMP scheme and 657514067, 992481934 and 3999 for the DIMP scheme, respectively. Table 5.1 shows the result of the simulation for both SIMP and DIMP.

Error detection scheme	No. of stuck-at faults	No. of random inputs	Percentage of error detection
SIMP	1650	1000000	99.8479%
DIMP	1650	1000000	99.9994%

Table 5.1: Single stuck-at fault injection in a slice of a bit-parallel $GF(2^{163})$ PB multiplier

In fact, SIMP cannot detect errors on input B; however, it can detect the errors on b_i's inside of the SM module with the probability given in (4.4). In Figure 5.7, a

single-bit error on the main input of the SM module denoted by cross sign cannot be detected by SIMP but errors on other locations denoted by circle sign can be detected with a certain probability.

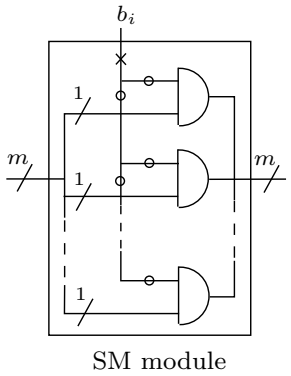


Figure 5.7: Possible fault locations in b_i input of the SM module

According to Table 5.1, the percentage of error detection of DIMP is higher than SIMP, although DIMP cannot detect single-bit errors on input B when the parities of all partitions of input A are zero (see Section 5.1.3).

Multiple Stuck-at Fault Injection

For each multiple stuck-at fault injection, we randomly chose a number of above-mentioned locations, i.e., $3(\omega-2)+5m+1$ locations, in a slice of the PB multiplier and then we randomly injected either stuck-at 0 fault or stuck-at 1 fault at each chosen location. In this experiment, 500 random multiple stuck-at faults were injected for each of 1000000 million random inputs. This experiment was very time consuming since for each multiple stuck-at fault injection all above-mentioned locations should be accessed. The number of detected and undetected faults are 498046872 and 1953128 for SIMP and 498472548 and 1527452 for DIMP, respectively. According to the results of the experiments, which are presented in Table 5.2, DIMP has a higher percentage of error detection compared to SIMP.

Error detection scheme	No. of stuck-at faults	No. of random inputs	Percentage of error detection
SIMP	500	1000000	99.6094%
DIMP	500	1000000	99.6945%

Table 5.2: Multiple stuck-at fault injection in a slice of a bit-parallel $GF(2^{163})$ PB multiplier

Furthermore, we performed the multiple fault injections in an alternative method. In this experiment, we randomly chose 2 to 5 locations from the above-mentioned locations in a slice of the PB multiplier and then we randomly injected either stuck-at 0 fault or stuck-at 1 fault at each chosen location, i.e., the Hamming wight of our injected error vectors were more than 1 and less than 6. Similarly, in this experiment, we injected 500 random multiple stuck-at faults for each of 1000000 million random input pairs. The reason for performing this experiment is that low weight faults may be more probable in real circumstances. The number of detected, masked and undetected faults are 382881592, 97749191 and 19369217 for SIMP, and 383212154, 97748343 and 19039503 for DIMP, respectively. The results of the experiment are presented in Table 5.3.

Error detection scheme	No. of stuck-at faults	No. of random inputs	Percentage of error detection
SIMP	500	1000000	95.18%
DIMP	500	1000000	95.27%

Table 5.3: Injection of low weight multiple stuck-at faults in a slice of a bit-parallel $GF(2^{163})$ PB multiplier

5.2.2 Analysis of Time and Area Overheads

A circuit diagram of a complete bit-parallel PB multiplier with CED is depicted in Figure 5.8. The input value of the very first slice is input A along with its parity

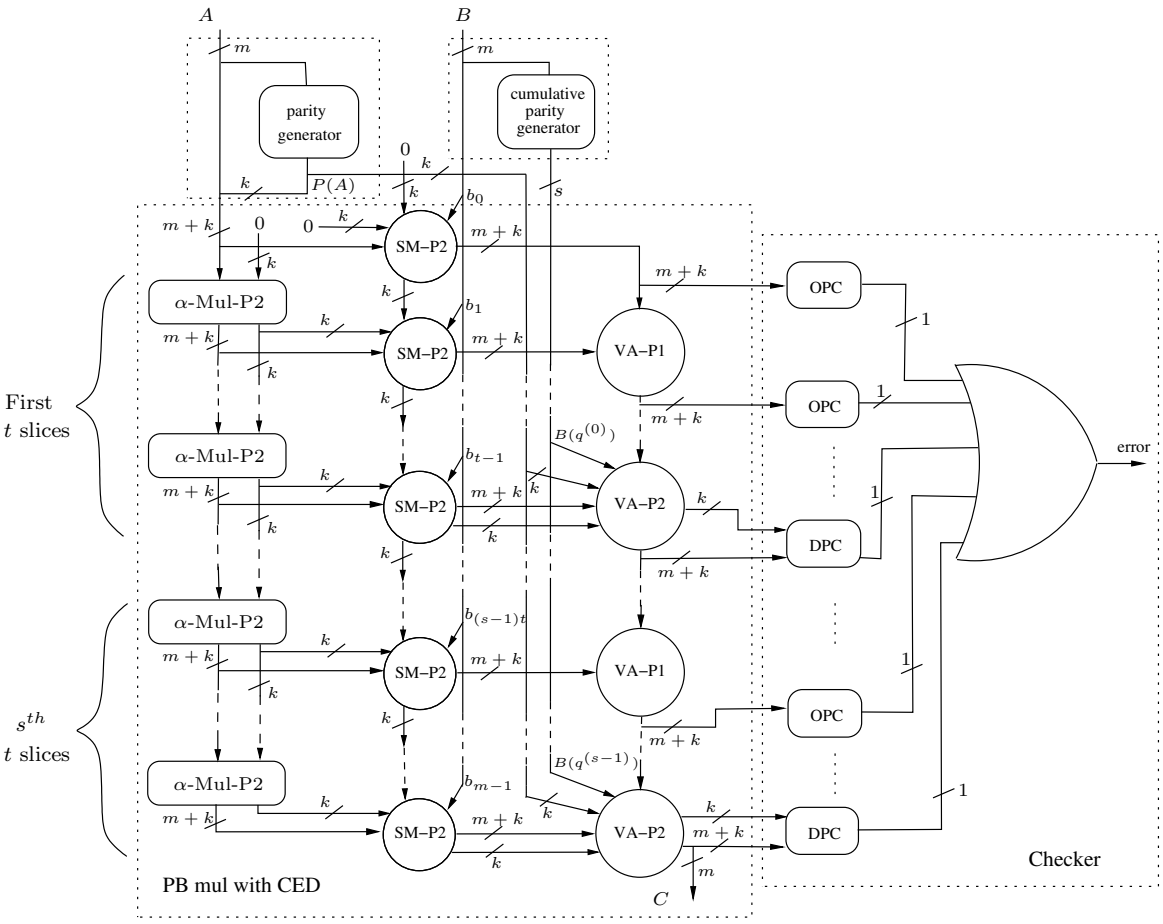


Figure 5.8: A complete bit-parallel multiplier with CED

$P(A)$ generated by a parity generator. In the first SR-P2 and SM-P2 modules, the inputs related to parity prediction of input B are initialized with zero vectors as discussed in Section 5.1.1. The parity checkers OPC and DPC are same as those presented in Figure 5.5. As shown in Figure 5.8, an OPC is located at the end of every slice except every t^{th} slice, where a DPC is located. Additionally, once the inputs A and B are updated, the results of the multiplication and error detection are ready after certain amount of delay due to the propagation of various signals through the circuit.

We have described the DIMP scheme by VHDL to obtain a realistic approximation of area and time overheads. As it can be inferred from Lemma 5.1 and Theorem 5.1, choosing the field defining polynomial $F(x)$ in such a way that the parity of $F(x)$ in each partition is zero, i.e., $P_{F_j} = 0$, slightly reduces the number of XOR gates in parity prediction circuits. Additionally, choosing $F(x)$ to be a trinomial or a pentanomial reduces the number of XOR gates in the design and make the entire design smaller. After RTL coding, to check the correct functionality of the design, Modelsim$^{(TM)}$ was used. Furthermore, we implemented the scheme on a Xilinx Spartan 3 (XC3S5000) FPGA using Xilinx ISE 7.1i.

The first experiment was to measure the area and time overheads of the DIMP scheme for a $GF(2^{163})$ multiplier for different numbers of parity bits of the first input while the number of parity bits of the second input was fixed to three. We implemented the scheme using one of the NIST recommended field defining polynomials for ECDSA $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$. This polynomial is very suitable for implementation because the parities of all of its partitions are zero when the number of the partitions ranges from 2 to 20.

To have a realistic impression about area consumption of the DIMP schemes, we present Table 5.4 which investigates the number of required FPGA slices and the area consumption percentage of the scheme.

No. of parity bits	NoCED	4	8	12	16	20
Number of required Slices	13541	19679	21069	22497	24518	26018
FPGA area consumption (%)[1]	40.69	59.13	63.30	67.60	73.67	78.18

[1]The total number of slices in a Xilinx Spartan 3 (XC3S5000) FPGA is 33280.

Table 5.4: FPGA area consumption for a bit-parallel $GF(2^{163})$ PB multiplier

As shown in Figure 5.9, the area overhead increases as the number of parity bits increases. For all points in the graph depicted in the figure, a line is fitted as

follows:

$$\text{for } GF(2^{163}) : \text{overhead } (\%) = 3.47 \times (\#\text{ of parity bits}) + 25.09. \tag{5.6}$$

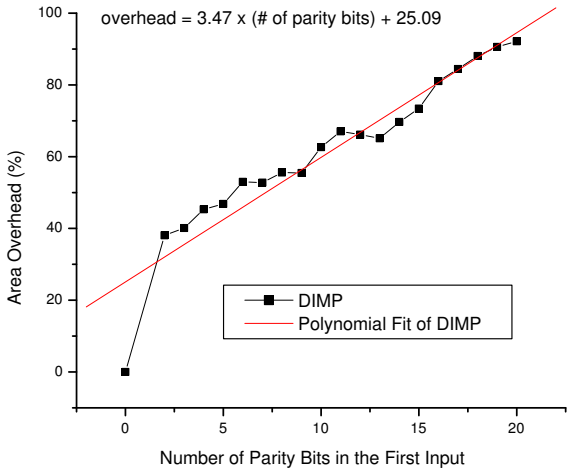


Figure 5.9: Area overhead of DIMP vs. parity-bit number for the field $GF(2^{163})$

Based on the experimental results and as it can be inferred from (5.6), area overhead tends to increase linearly except for very small numbers of parity bits.

Furthermore, the area overhead of the SIMP and the DIMP schemes are compared in Figure 5.10. The results for SIMP are obtained from [7]. As expected, the area overhead of DIMP is more than SIMP in similar implementations, however, the difference is not significant. Additionally, the area overhead is in an acceptable range because for obtaining a sufficiently high probability of error detection (say ≈ 0.997), one needs only eight parity bits for the first input and three parity bits for the second one in the proposed scheme. This results in about 55% area overhead, which is lower than 100% overhead of the DMR scheme.

We also investigated the time overheads of the $GF(2^{163})$ PB multipliers for

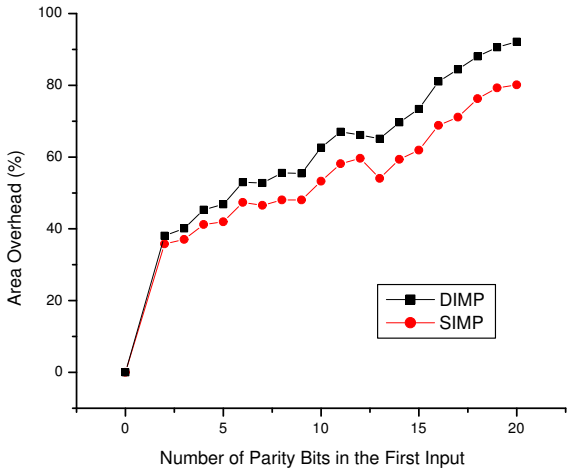
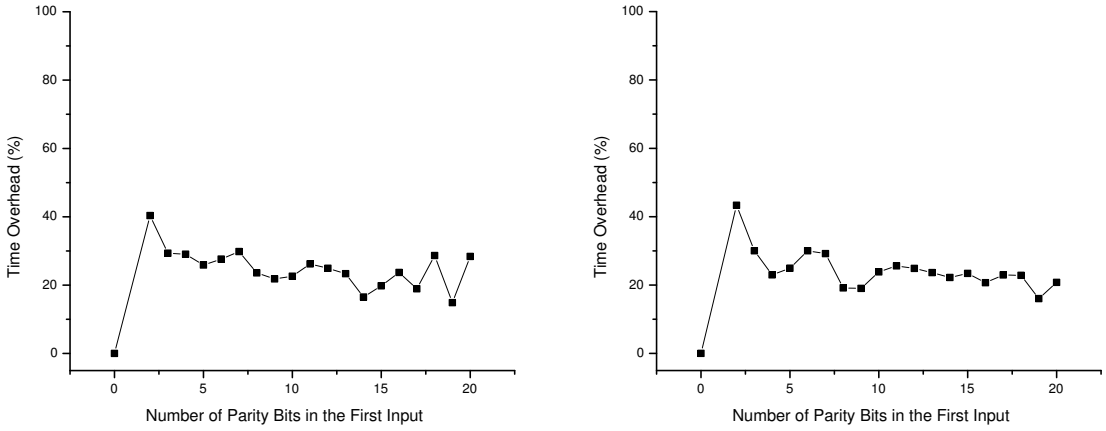


Figure 5.10: Area overhead DIMP vs. SIMP for the field $GF(2^{163})$

different numbers of parity bits as shown in Figure 5.11. The time overhead of the bit-parallel implementation is the delay of the critical path, i.e., the maximum propagation delay from one of the input pins to one of the output pins. We obtain the delays of all input pins to output pins from the post place and route static timing report of Xilinx ISE. Results show that the time overheads are generally less than 30% when more than two parity bits are used.

In the second experiment, we fixed the number of parity bits of the first input to eight bits and measured the area and time overheads of DIMP when the number of parity bits for the second input ranges from 3 to 20. As shown in Figure 5.12(a), the area overhead increases slightly in this range, i.e., less than 2%. Additionally, as shown in Figure 5.12(b), the range of changes in the time overhead is not very significant, i.e., less than 15%, and is in a similar range as the time overhead of the first experiment.



(a) SIMP (b) DIMP

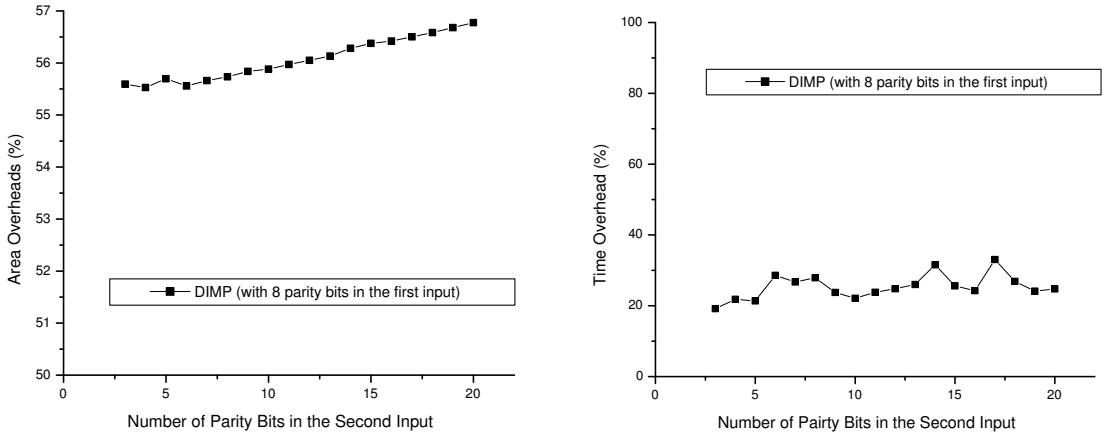
Figure 5.11: Time overhead vs. parity-bit number for the field $GF(2^{163})$

5.3 Extending SIMP and DIMP to Dual and Normal Bases

In this section, the SIMP and DIMP schemes are extended to dual and normal bases (type I and type I optimal normal bases). In the following sections, we divide the first input of the multiplier, i.e., input A, into k parts. The length of each part is $l = \frac{m}{k}$ for DB and ONB2 and is $l = \frac{m+1}{k}$ for ONB1. Moreover, the second input B is divided into s parts in DIMP. Similarly, the length of each part is $t = \frac{m}{s}$ for DB and ONB2 and is $t = \frac{m+1}{s}$ for ONB1(see Section 4.1 for the cases that m or $m+1$ are not divisible by k or s).

5.3.1 SIMP and DIMP in Dual Basis

As mentioned in Section 2.1, suppose that the set $\{y_0, y_1, \cdots, y_{m-1}\}$ is the dual of the polynomial basis $\{x^0, x^1, \cdots, x^{m-1}\}$. Hence, $A \in GF(2^m)$ can be represented



(a) Area overhead (b) Time overhead

Figure 5.12: Area and time overheads of DIMP vs. parity-bit number of the second input for the field $GF(2^{163})$

in DB as follows:

$$A = \sum_{i=0}^{m-1} a_i' y_i.$$

Moreover, for $A, B, C \in GF(2^m)$ we have:

$$\begin{aligned} C_{DB} &= B_{PB} A_{DB} \\ &= b_0 A_{DB} + b_1 x A_{DB} + \cdots + b_{m-1} x^{m-1} A_{DB}. \end{aligned}$$

Hence, we can implement DB multiplications using a similar architecture as PB ones. Given that $A^{(n-1)}$ and $A^{(n)}$ are input and output of an SR module, respectively, we have: $A^{(n)} = xA^{(n-1)}$. Also, VA and SM modules are same as those in a PB multiplier. As an example, Figure 5.13 shows the n^{th} slice of a bit-parallel DB multiplier, where $1 \leq n \leq m-1$ and $A^{(0)} = A$ and $C^{(0)} = b_0 A$.

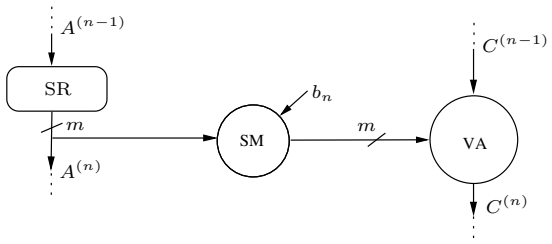


Figure 5.13: A slice of a bit-parallel DB multiplier

SIMP in Dual Basis

Recall that in SIMP, we divide the input A into k parts. Considering that $A_j^{(n-1)}$ and $A_j^{(n)}$ are the j^{th} parts of $A^{(n-1)}$ and $A^{(n)}$, respectively, we have:

$$\begin{aligned} A^{(n-1)} &= \sum_{i=0}^{l-1} a_i^{'(n-1)} y_i + \sum_{i=l}^{2l-1} a_i^{'(n-1)} y_i + \cdots + \sum_{i=(k-1)l}^{kl-1} a_i^{'(n-1)} y_i \\ &= \sum_{j=0}^{k-1} \sum_{i=jl}^{(j+1)l-1} a_i^{'(n-1)} y_i \\ &= \sum_{j=0}^{k-1} A_j^{(n-1)}. \end{aligned}$$

Also, according to Lemma 3.4 we have:

$$\begin{aligned} A^{(n)} &= \sum_{i=0}^{m-2} a_{i+1}^{'(n-1)} y_i + \sum_{i=0}^{m-1} f_i a_i^{'(n-1)} y_{m-1} \\ &= \sum_{j=0}^{k-2} \left(\sum_{i=jl}^{(j+1)l-2} a_{i+1}^{'(n-1)} y_i + a_{(j+1)l}^{'(n-1)} y_{(j+1)l-1} \right) \\ &\quad + \left(\sum_{i=(k-1)l}^{m-2} a_{i+1}^{'(n-1)} y_i + \left(\sum_{i=0}^{m-1} f_i a_i^{'(n-1)} \right) y_{m-1} \right). \end{aligned}$$

Therefore,

$$A_j^{(n)} = \begin{cases} \sum_{i=jl}^{(j+1)l-1} a_{i+1}^{'(n-1)} y_i, & 0 \le j \le k-2, \\ \sum_{i=(k-1)l}^{m-2} a_{i+1}^{'(n-1)} y_i + \left(\sum_{i=0}^{m-1} f_i a_i^{'(n-1)} \right) y_{m-1}, & j = k-1. \end{cases}$$

Using parity function $P(.)$, we have:

$$\begin{aligned} P(A_j^{(n)}) &= \begin{cases} P\left(\sum_{i=jl}^{(j+1)l-1} a_{i+1}^{'(n-1)} y_i \right), & 0 \le j \le k-2, \\ P\left(\sum_{i=(k-1)l}^{m-2} a_{i+1}^{'(n-1)} y_i \right) + P\left(\left(\sum_{i=0}^{m-1} f_i a_i^{'(n-1)} \right) y_{m-1} \right), & j = k-1, \end{cases} \\ &= \begin{cases} a_{jl}^{'(n-1)} + P(A_j^{(n-1)}) + a_{(j+1)l}^{'(n-1)}, & 0 \le j \le k-2, \\ a_{(k-1)l}^{'(n-1)} + P(A_{k-1}^{(n-1)}) + P_{FA}^{(n-1)}, & j = k-1, \end{cases} \end{aligned} \tag{5.7}$$

where $P_{FA}^{(n-1)} = a_0^{'(n-1)} + \sum_{i=0}^{\omega-2} a_{\theta_i}^{'(n-1)}$ and $1 \le n \le m-1$. Moreover, ω is the Hamming wight of the field defining polynomial $F(x)$ and θ_i's contain the indices of the $\omega - 2$ nonzero coefficients of $F(x)$ other than f_0 and f_m.

According to Figure 5.13, we can write:

$$C^{(n)} = C^{(n-1)} + b_n A^{(n)}.$$

Similarly for each partition, we have:

$$C_j^{(n)} = C_j^{(n-1)} + b_n A_j^{(n)}. \tag{5.8}$$

Therefore,

$$P(C_j^{(n)}) = P(C_j^{(n-1)}) + b_n P(A_j^{(n)})$$
$$= \begin{cases} b_0 P(A_j), & n = 0, 0 \le j \le k-1, \\ P(C_j^{(n-1)}) + b_n a_{jl}^{'(n-1)} + b_n P(A_j^{(n-1)}) + b_n a_{(j+1)l}^{'(n-1)}, & 1 \le n \le m-1, 0 \le j \le k-2, \\ P(C_j^{(n-1)}) + b_n a_{(k-1)l}^{'(n-1)} + b_n P(A_{k-1}^{(n-1)}) + b_n P_{FA}^{(n-1)}, & 1 \le n \le m-1, j = k-1. \end{cases}$$

Now, the parity of each partition of the output of a VA module in each slice can be computed using the information of the previous slice. In SIMP, this should be compared against the actual parity of that part.

DIMP in Dual Basis

Let us write (5.7) in an iterative manner for $1 \le n \le m-1$ as follows:

$$P(A_j^{(n)}) = P(A_j) + \begin{cases} \sum_{g=0}^{n-1} a_{jl}^{'(g)} + a_{(j+1)l}^{'(g)}, & 0 \le j \le k-2, \\ \sum_{g=0}^{n-1} a_{(k-1)l}^{'(g)} + P_{FA}^{(g)}, & j = k-1. \end{cases}$$

Furthermore, let us define $h_j^{'(n)}$ as follows:

$$h_j^{'(n)} = \begin{cases} 0, & n = 0, 0 \le j \le k-1, \\ \sum_{g=0}^{n-1} a_{jl}^{'(g)} + a_{(j+1)l}^{'(g)}, & 1 \le n \le m-1, 0 \le j \le k-2, \\ \sum_{g=0}^{n-1} a_{(k-1)l}^{'(g)} + P_{FA}^{(g)}, & 1 \le n \le m-1, j = k-1. \end{cases}$$

Therefore, we have:

$$P(A_j^{(n)}) = P(A_j) + h_j^{'(n)}. \tag{5.9}$$

Now, we consider s partitions for input B. In DIMP, we would like to compute the predicted parity of each partition of the output of every t^{th} VA module based on the parities of both inputs A and B. Therefore, after unrolling (5.8) we can have an expression similar to (5.3) as follows:

$$\begin{aligned} P({C_j}^{(q)}) &= \sum_{n=0}^{(q+1)t-1} b_n P(A_j^{(n)}) \\ &= b_0 P(A_j) + \sum_{n=1}^{(q+1)t-1} b_n P(A_j^{(n)}), \end{aligned} \tag{5.10}$$

where $0 \leq q \leq s-1$.

Substituting (5.9) in (5.10) and similar to (5.4), for $0 \leq j \leq k-1$ and $0 \leq n \leq m-1$ we have:

$$P({C_j}^{(q)}) = P(A_j)P(B^{(q)}) + \sum_{n=1}^{(q+1)t-1} b_n {h'}^{(n)},$$

where $P(B^{(q)}) = \sum_{i=0}^{q} P(B_i)$ and $P(B_i)$ is the parity of the i^{th} partition of B.

5.3.2 SIMP and DIMP in Type I Normal Basis

As mentioned in Section 3.4.3, an element $A \in GF(2^m)$, for certain values of m, can be represented using type I optimal normal basis (ONB1) as $A = \sum_{i=1}^{m} \hat{a}_i x^i$. Alternatively, we can have $A = \sum_{i=0}^{m} \hat{a}_i x^i$, where $\hat{a}_0 = 0$. Let $B \in GF(2^m)$ also be represented in ONB1. Then multiplying A and B, we have:

$$\begin{aligned} C &= A.B \mod x^{m+1} + 1 \\ &= \sum_{i=0}^{m} \hat{b}_i A^{(i)}, \end{aligned} \tag{5.11}$$

where $A^{(0)} = A$, $A^{(i)} = \sum_{j=0}^{m} \hat{a}_{<j-1>}^{(i-1)} x^j$ and $< j-1 >= j-1 \mod m+1$ (see Section 3.4.3). Hence, an ONB1 multiplier can be constructed with an architecture similar to that of PB. However, the SR module performs a rotation or cyclic shift such that the MSB rotates back to the LSB position. Figure 5.14 shows a slice of

a bit-parallel ONB1 multiplier.

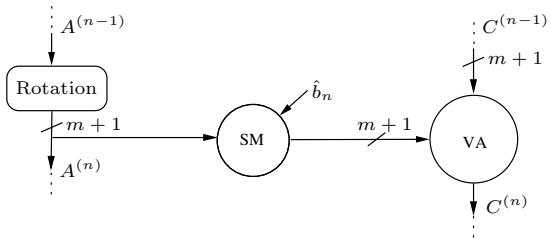


Figure 5.14: A slice of a bit-parallel ONB1 multiplier

It is worth mentioning that the intermediate and final results of the multiplication are $(m+1)$ bits long. Therefore, as mentioned in Section 3.4.3, we should XOR the LSB of C with its other bits to have the final result of the multiplication in ONB1 representation.

SIMP in ONB1

As mentioned earlier, we divide input A into k parts. The length of each part is $l = \frac{m+1}{k}$. Suppose that $A_j^{(n-1)}$ and $A_j^{(n)}$ are the j^{th} parts of $A^{(n-1)}$ and $A^{(n)}$, respectively. Then we have:

$$\begin{aligned} A^{(n-1)} &= \sum_{j=0}^{k-1} x^{jl} \left(\sum_{n=0}^{l-1} \hat{a}_{jl+n}^{(n-1)} x^n \right) \\ &= \sum_{j=0}^{k-1} A_j^{(n-1)}. \end{aligned}$$

Moreover, considering (5.11), we have:

$$\begin{aligned} A^{(n)} &= \sum_{j=0}^{k-1} x^{jl} \left(\sum_{i=0}^{l-1} \hat{a}_{<jl+n-1>}^{(n-1)} x^n \right) \\ &= \sum_{j=0}^{k-1} A_j^{(n)}. \end{aligned}$$

where $< jl + n - 1 >= jl + n - 1 \mod m + 1$.

Now applying parity function $P(.)$, we have:

$$\begin{aligned} P(A_j^{(n)}) &= P\left(x^{jl} \sum_{n=0}^{l-1} \hat{a}_{<jl+n-1>}^{(n-1)} x^n \right) \\ &= \sum_{n=0}^{l-1} \hat{a}_{<jl+n-1>}^{(n-1)} \\ &= \hat{a}_{<jl-1>}^{(n-1)} + \left(\sum_{i=0}^{l-1} \hat{a}_{jl+n}^{(n-1)} \right) + \hat{a}_{(j+1)l-1}^{(n-1)} \\ &= \hat{a}_{<jl-1>}^{(n-1)} + P(A_j^{(n-1)}) + \hat{a}_{(j+1)l-1}^{(n-1)}, \end{aligned} \tag{5.12}$$

where $0 \leq j \leq k-1$.

Using (5.8) and following a similar procedure as in Section 5.3.1, we have:

$$P(C_j^{(n)}) = P(C_j^{(n-1)}) + \hat{b}_n \hat{a}_{<jl-1>}^{(n-1)} + \hat{b}_n P(A_j^{(n-1)}) + \hat{b}_n \hat{a}_{(j+1)l-1}^{(n-1)},$$

where $0 \leq j \leq k-1$, $1 \leq n \leq m$ and $C_j^{(0)} = 0$.

In SIMP, we compare the predicted parity of the j^{th} part with its actual parity after every VA module and before the first one (in bit-parallel implementation).

DIMP in ONB1

In DIMP, the first input is partitioned same as SIMP and the second input is divided into s where the length of each partition is $t = \frac{m+1}{s}$. Now, using (5.12), for $0 \leq j \leq k-1$ and $1 \leq n \leq m$ we have:

$$\begin{aligned} P(A_j^{(n)}) &= P(A_j) + \sum_{g=0}^{n-1} \hat{a}_{<jl-1>}^{(g)} + P(A_j^{(g)}) + \hat{a}_{(j+1)l-1}^{(g)} \\ &= P(A_j) + \hat{h}_j^{(n)}, \end{aligned}$$

where $\hat{h}_j^{(n)} = \sum_{g=0}^{n-1} \hat{a}_{<jl-1>}^{(g)} + P(A_j^{(g)}) + \hat{a}_{(j+1)l-1}^{(g)}$. Following a similar procedure as in Section 5.3.1, for $0 \leq j \leq k-1$ an $0 \leq q \leq s-1$ we have:

$$P(C_j{}^{(q)}) = P(A_j)P(B^{(q)}) + \sum_{n=1}^{(q+1)t-1} \hat{b}_n \hat{h}_j^{(n)}, \tag{5.13}$$

where $P(B^{(q)}) = \sum_{i=0}^{q} P(B_i)$ and $P(B_i)$ is the parity of the i^{th} partition of B. Using (5.13), one can compare the parity of the j^{th} part of the output of a VA module at every t^{th} slice of the multiplier with its actual parity.

5.3.3 SIMP and DIMP in Type II Normal Basis

As mentioned in Section 3.4.3, to represent an element $A \in GF(2^m)$ in type II normal basis (ONB2), the following set can be used:

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

where γ is a normal element in the field. Suppose that $A, B \in GF(2^m)$. Then according to (3.15) the result of their multiplication is as follows:

$$C = \sum_{n=1}^{m} \sum_{i=1}^{m} \hat{b}_n \left(\hat{a}_{|n-i|} + \hat{a}_{||n+i||}\right) \left(\gamma^i + \gamma^{-i}\right),$$

where $\hat{a}_0 = 0$.

The above expression can be expressed as:

$$C = \sum_{n=1}^{m} \hat{b}_n A^{(n)}, \tag{5.14}$$

where $A^{(n)} = \sum_{i=1}^{m} \left(\hat{a}_{|n-i|} + \hat{a}_{||n+i||}\right) (\gamma^i + \gamma^{-i})$.

One way to implement the multiplication mentioned in expression (5.14) is in a

bit-parallel fashion as shown in Figure 5.15.

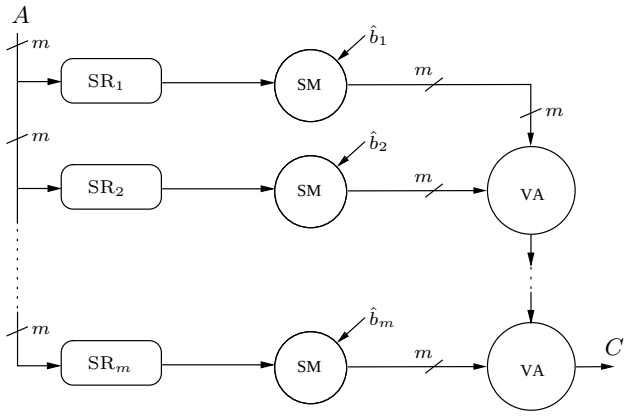


Figure 5.15: A bit-parallel ONB2 multiplier

In the above figure, module SR_n receives input A and computes $A^{(n)}$ according to (5.14). Considering that $\hat{a}_{|n-i|} = 0$ when $n = i$, the number of XOR gates in each SR_n module is $m + 1$. Table 5.3.3 shows the number of 2-input gates needed for the multiplier shown in Figure 5.15.

Module	Number of gates	
	2-input AND	2-input XOR
SR_n	—	$m(m-1)$
SM	m^2	—
VA	—	$m(m-1)$
Total	m^2	$2m(m-1)$

Table 5.5: The number of 2-input gates needed for the bit-parallel ONB2 multiplier

Note that the first input A is divided into k parts in the SIMP and DIMP schemes and the second input B is divided into s parts in DIMP. The length of each part is $l = \frac{m}{k}$ for the first input and $t = \frac{m}{s}$ for the second input, respectively.

SIMP in ONB2

Dividing input A into k parts for $1 \leq n \leq m$, we have:

$$\begin{aligned} A^{(n)} &= \sum_{j=0}^{k-1} \sum_{i=jl+1}^{(j+1)l} (\hat{a}_{|n-i|} + \hat{a}_{||n+i||})(\gamma^i + \gamma^{-i}) \\ &= \sum_{j=0}^{k-1} A_j^{(n)}. \end{aligned}$$

Let us compute the parity of each part of the first SR module, i.e., SR_1, directly from A for $0 \leq j \leq k-1$ as follows:

$$\begin{aligned} P\left(A_j^{(1)}\right) &= \sum_{i=jl+1}^{(j+1)l} (\hat{a}_{|1-i|} + \hat{a}_{||1+i||}) \\ &= \hat{a}_{|1-jl-1|} + \hat{a}_{|1-jl-2|} + \left(\sum_{i=jl+3}^{(j+1)l} \hat{a}_{|1-i|} \right) \\ &\quad + \left(\sum_{i=jl+1}^{(j+1)l-2} \hat{a}_{||1+i||} \right) + \hat{a}_{||(j+1)l||} + \hat{a}_{||(j+1)l+1||} \\ &= \hat{a}_{jl} + \hat{a}_{|1+jl|} + \hat{a}_{||(j+1)l||} + \hat{a}_{||(j+1)l+1||}. \end{aligned}$$

Considering that $\hat{a}_0 = 0$ for $j = 0$ and $\hat{a}_{||m+1||} = \hat{a}_{||m||}$ for $j = k-1$, we have:

$$P\left(A_j^{(1)}\right) = \begin{cases} \hat{a}_1 + \hat{a}_l + \hat{a}_{l+1}, & j = 0, \\ \hat{a}_{jl} + \hat{a}_{jl+1} + \hat{a}_{(j+1)l} + \hat{a}_{(j+1)l+1}, & 1 \leq j \leq k-2, \\ \hat{a}_{(k-1)l} + \hat{a}_{(k-1)l+1}, & j = k-1. \end{cases} \tag{5.15}$$

Therefore, the PPC of each partition can be constructed using three 2-input XOR gates at maximum. Now, we compute the parities of the other SR modules as follows:

$$P\left(A_j^{(n)}\right) - P\left(A_j^{(n-1)}\right) = \sum_{i=jl+1}^{(j+1)l} (\hat{a}_{|n-i|} + \hat{a}_{||n+i||}) - \sum_{i=jl+1}^{(j+1)l} (\hat{a}_{|n-1-i|} + \hat{a}_{||n-1+i||})$$
$$= \hat{a}_{|n-1-jl-1|} + \hat{a}_{|n-(j+1)l|} + \hat{a}_{||n-1+jl+1||} + \hat{a}_{||n+(j+1)l||}.$$

Therefore, for $2 \leq n \leq m$, we have:

$$P\left(A_j^{(n)}\right) = P\left(A_j^{(n-1)}\right) + \hat{a}_{|n-jl-2|} + \hat{a}_{|n-(j+1)l|} + \hat{a}_{||n+jl||} + \hat{a}_{||n+(j+1)l||}. \quad (5.16)$$

Similar to PB, DB and ONB1, we have:

$$P(C_j^{(n)}) = P(C_j^{(n-1)}) + \hat{b}_n P(A_j^{(n)}),$$

where $0 \leq j \leq k-1$, $1 \leq n \leq m$ and $C_j^{(0)} = 0$.

Figure 5.16 shows the parity prediction strategy for SIMP in ONB2.

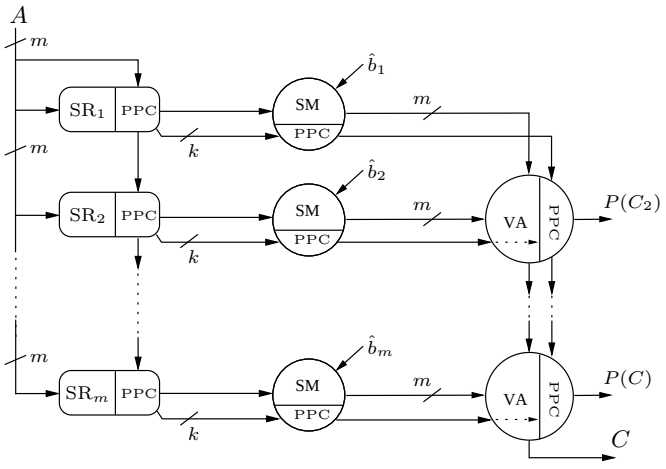


Figure 5.16: Parity prediction strategy for SIMP in ONB2

DIMP in ONB2

Let us write (5.16) in an iterative form as follows:

$$P\left(A_j^{(n)}\right) = \begin{cases} (5.15), & n = 1, \\ P\left(A_j^{(1)}\right) + \sum_{g=2}^{n} \hat{a}_{|g-jl-2|} + \hat{a}_{|g-(j+1)l|} + \hat{a}_{||g+jl||} + \hat{a}_{||g+(j+1)l||}, & 2 \leq n \leq m. \end{cases}$$

Now, we define $\hat{h}_j^{(n)}$ as follows:

$$\hat{h}_j^{(n)} = \begin{cases} 0, & n = 1, \\ \sum_{g=2}^{n} \hat{a}_{|g-jl-2|} + \hat{a}_{|g-(j+1)l|} + \hat{a}_{||g+jl||} + \hat{a}_{||g+(j+1)l||}, & 2 \leq n \leq m. \end{cases}$$

Hence, for $1 \leq n \leq m$, we have:

$$P\left(A_j^{(n)}\right) = P\left(A_j^{(1)}\right) + \hat{h}_j^{(n)}.$$

Now, similar to PB, DB and NB, for $0 \leq j \leq k-1$ an $0 \leq q \leq s-1$ we have:

$$P(C_j{}^{(q)}) = P(A_j^{(1)})P(B^{(q)}) + \sum_{n=1}^{(q+1)t-1} \hat{b}_n \hat{h}_j^{(n)}, \tag{5.17}$$

where $P(B^{(q)}) = \sum_{i=0}^{q} P(B_i)$ and $P(B_i)$ is the parity of the i^{th} partition of B.

5.4 Summary

In this chapter, a concurrent error detection scheme, referred to as DIMP, is presented for polynomial basis multipliers. In this scheme, multiple parity bits are used for both inputs of the multiplier. The scheme can detect errors in both inputs

and inside of the multiplier with a probability of more than 75% using two party bits for each input. This probability approaches unity as the number of parity bits of the inputs increases. Results of our bit-parallel implementation show that the area overhead has an increasing trend. However, this overhead is lower than that of the traditional dual modular redundant scheme for a sufficient number of parity bits. Furthermore, the average time overhead due to the use of the scheme in the bit-parallel implementations is around 25%. In this chapter, the SIMP and DIMP schemes are also extended to finite field multipliers that use a dual or an optimal normal basis.

Chapter 6

Linear Code Based Error Detection Schemes

In the previous chapters, RESO based and parity based schemes are used for detecting errors. This chapter presents three schemes for the detection of errors in both bit-serial and bit-parallel polynomial basis multipliers over binary extension fields based on the scaling technique (see Chapter 1). The proposed schemes can be applied to any finite field $GF(2^m)$. In these schemes, we use linear codes. Such codes have also been used in [27]. Important differences between this work and [27] are as follows. First, the error model of this work is more generic and the error can occur in any location of the circuit. Secondly, this work gives much more flexibility to choose the field defining and the code generator polynomials. This leads to a reduction in the number of redundant bits and in turn a reduction in the area overhead.

In this chapter, the error detection probability of the code presented in this work is investigated. Also, the error detection capabilities of the schemes are evaluated by a number of simulation-based fault injections. Among three schemes presented in this chapter, one has a similar percentage of error detection as SIMP and the

other two schemes have slightly better percentage of error detection compared to DIMP. Furthermore, the area and time overheads of the schemes for both bit-serial and bit-parallel implementations are presented. Results show that, in our bit-serial implementations for eight redundant bits, the area overheads are in a reasonable rang, i.e., lower than *dual modular redundant* systems, and the time overheads are quite small, i.e., less than 15%. In bit-parallel implementations, however, the time and area overheads of only one of the schemes are in the acceptable range.

The organization of this chapter is as follows. In Section 6.1, a class of linear codes, denoted as $\mathcal{L}$, is presented. Three concurrent error detection schemes are presented in Section 6.2. Using one of the schemes, namely the single-input encoding, we develop error detectable bit-serial and bit-parallel multiplier structures in Section 6.3. The error detection capability of the single-input encoding scheme is then investigated in Section 6.4. Our second scheme is explained in Section 6.5. In Section 6.6, the third scheme is discussed. The time and area overheads of the schemes are presented in Section 6.8. Finally, Section 6.9 gives a summary of the chapter.

Part of this work has been presented in [8].

6.1 A Class of Linear Codes: $\mathcal{L}$ Code

In an (n, m) block code, the input information sequence is divided into m-bit blocks and each block is encoded to an n-bit codeword ($n > m$). One important class of block codes is linear codes. These are extensively used in communication applications for correcting/detecting errors in transmission channels. Here, the binary linear codes are considered for detecting errors in the polynomial basis multipliers. In the simplest form, an (n, m) block code is linear if and only if the modulo-2 addition of two codewords is also a codeword.

Let $V = (v_0, v_1, \cdots, v_{n-1})$ be a codeword. A polynomial whose coefficients are the components of V, is said to be a code polynomial. A code polynomial of degree up to $n-1$ is generated with a polynomial of degree $n-m$ of the following form:

$$g(x) = g_0 + g_1 x + g_2 x^2 + \cdots + g_{n-m-1} x^{n-m-1} + x^{n-m},$$

where $g_i \in GF(2)$. Polynomial $g(x)$ is called a generator polynomial. Every code polynomial in the code is a multiple of $g(x)$. In fact, our (n, m) linear code $\mathcal{L}$ maps an element of a finite field $GF(2^m)$ to an element of a commutative ring with modulus $F(x) = f(x)g(x)$, where $f(x)$ is the irreducible polynomial used for representing the elements of $GF(2^m)$. Note that, in this chapter, $f(x)$ and $F(x)$ are the field defining polynomial and the modulus of the ring, respectively.

It is worth mentioning that the well-known cyclic code has the corresponding modulus as $x^n - 1$. If one wants to use cyclic codes to encode the elements of the field, then $f(x)g(x) = x^n - 1$. Therefore, for a given $f(x)$, this limits the number of choices of $g(x)$ and n.

6.2 Concurrent Error Detection Schemes

Errors may be caused by different types of faults such as open faults, short (bridging) faults, and/or stuck-at faults. Furthermore, the faults can be transient or permanent. In this chapter, we investigate three schemes for detecting random errors.

In the first scheme, which lays foundation of discussions for the other schemes, only one of the inputs of the PB multiplier is encoded, i.e., it is multiplied by generator $g(x)$. The second input is not encoded. In the second scheme, both inputs are encoded. In general, they can be encoded with two different generators, $g_1(x)$ and $g_2(x)$. The first and the second schemes are referred to as single-input encoding

(SIE) and double-input encoding (DIE), respectively. In the third scheme, referred to as hybrid scheme, the SIMP scheme presented in Chapter 4 is used along with the encoding scheme. In other words, one of the inputs is divided into a number of partitions and a parity bit for each partition is considered and preserved throughout the multiplication. Additionally, the second input is encoded by a generator $g'(x)$.

As expected, DIE has better error detection capability than SIE but its area overhead is higher. Nevertheless, the probability of error detection of SIE can be within an acceptable range because for some applications, for example in an elliptic curve cryptographic processor, the second input either comes from other operations such as adders and multipliers or comes as the direct input to the multiplier. In the first case, if the previous operation has an error detection circuitry, its output, which is the second input of the current multiplier, is expected to be error free. In the second case, one can use a concurrent error detection technique for the input of the multiplier once to avoid faulty inputs. On the other hand, it turns out that the hybrid scheme has similar error detection capability as DIE and it also has significantly lower area overhead. Depending on the further use of the multiplier's output, the PB multiplier with one of these schemes can produce either an encoded output, i.e., multiplied by only one generator, or an unencoded output.

6.3 SIE Based Error Detectable Multipliers

As mentioned in Section 6.1, a PB multiplier can be constructed with three types of modules: 1) SR, 2) SM, and 3) VA. In the following, (n, m) $\mathcal{L}$ codes are applied to the inputs of these modules to obtain error detectable multipliers. For bit-serial implementation, clearly, the size of registers should increase from m bits to n bits.

6.3.1 SM and VA Modules

Suppose that an (n, m) $\mathcal{L}$ code is used and $g(x)$ is the generator polynomial. Let A, B, S and $P \in GF(2^m)$ and $b \in GF(2)$, where scalar multiplication $b.A = P$ and vector addition $A + B = S$. Suppose A', B', S' and $P' \in GF(2^n)$ are the results of encoding A, B, S and P, respectively. Thus, for scalar multiplication we have:

$$b.A' = b.Ag = Pg = P',$$

and for vector addition we have:

$$A' + B' = Ag + Bg = (A + B)g = Sg = S'.$$

Accordingly, for using $\mathcal{L}$ codes, the sizes of SM and VA modules increase from m bits to n bits each.

6.3.2 SR Module

As shown in Figure 6.1(a), the unencoded input and the output of the SR module are $U(x) = \sum_{i=0}^{m-1} u_i x^i$ and $U_s(x) = \sum_{i=0}^{m-1} u_{s_i} x^i$, respectively. The code generator polynomial, $g(x)$, over $GF(2)$ of degree $n - m$ is used for encoding. The encoded input and the output of the SR module (see Figure 6.1(b)) are $V(x) = \sum_{i=0}^{n-1} v_i x^i$ and $V_s(x) = \sum_{i=0}^{n-1} v_{s_i} x^i$, respectively.

In an SR module with unencoded input, we have:

$$U_s(x) = xU(x) \mod f(x).$$

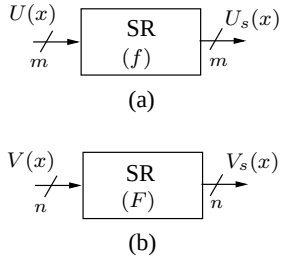


Figure 6.1: SR module: (a) with unencoded input, SR depends on $f(x)$, (b) with encoded input, SR depends on $F(x)$

According to (2.4):

$$\begin{aligned} U_s(x) &= \sum_{i=0}^{m-2} u_i x^{i+1} + u_{m-1} \sum_{i=0}^{m-1} f_i x^i = x \sum_{i=0}^{m-1} u_i x^i \\ &+u_{m-1} \left(x^m + \sum_{i=0}^{m-1} f_i x^i \right) = xU(x) + u_{m-1} f(x). \end{aligned} \tag{6.1}$$

On the other hand, for encoded inputs to SR module we have:

$$V(x) = U(x)g(x). \tag{6.2}$$

Thus, using (6.1) and (6.2), for input $V(x)$ the output of the SR module is:

$$\begin{aligned} V_s(x) &= U_s(x)g(x) = xU(x)g(x) + u_{m-1}f(x)g(x) \\ &= xV(x) + u_{m-1}F(x). \end{aligned} \tag{6.3}$$

Since $F(x)$ can be considered to be fixed, it can be pre-computed. On the other hand, $v_{n-1} = u_{m-1}.g_{n-m}$ and $g_{n-m} = 1$, thus:

$$v_{n-1} = u_{m-1}. \tag{6.4}$$

Therefore, using (6.3) and (6.4) we have:

$$V_s(x) = xV(x) + v_{n-1}F(x). \tag{6.5}$$

In (6.5), $F(x)$ is a reducible polynomial but similar to the case of irreducible polynomial the following remark holds.

Remark 6.1 *Let $\omega(F)$ be the Hamming weight of $F(x)$. The number of XOR gates required for constructing the SR module with encoded input, shown in Figure 6.2, is $\omega(F) - 2$.*

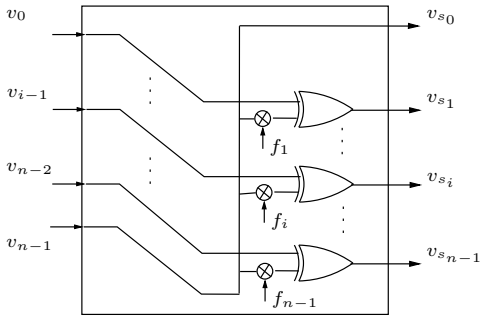


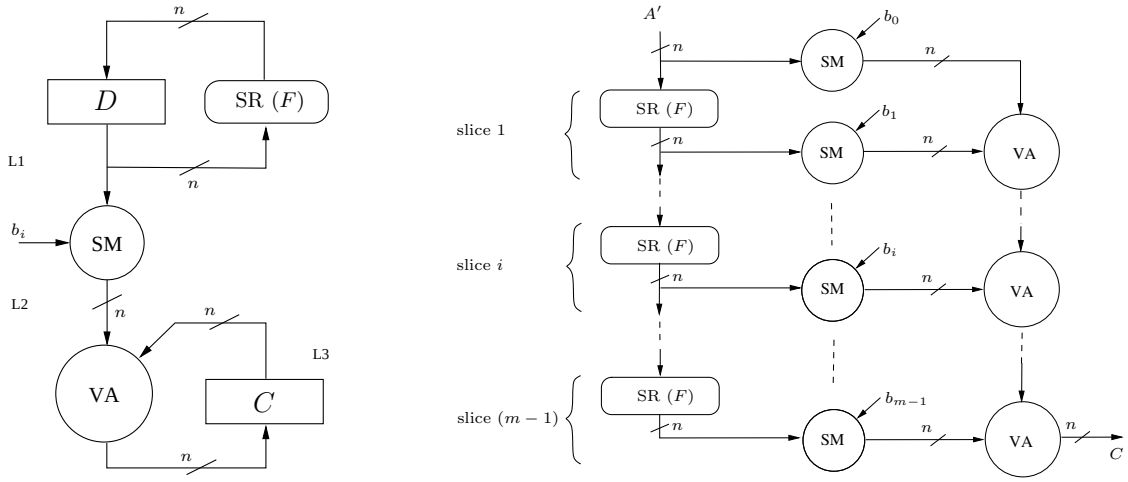
Figure 6.2: SR module with encoded input

6.3.3 Bit-serial and Bit-parallel Polynomial Basis Multipliers

To construct a bit-serial and a bit-parallel multiplier with concurrent error detection capability, we will use updated versions of SR, SM, and VA modules with encoded input. Figure 6.3(a) shows a bit-serial multiplier with concurrent error detection (CED) capability. For multiplying A and B with CED capability, register D is initialized with encoded A, i.e., A'. An error checker can be placed at each of the

three locations: L1, L2 and L3. In the next section, the frequency of check points will be discussed.

Figure 6.3(b) shows a bit-parallel multiplier with CED capability. In the bit-parallel multiplier an error checker can be placed after each modules. Thus, there can be as many as $3m - 2$ error checkers for a bit-parallel multiplier.



(a) Bit-serial (b) Bit-parallel

Figure 6.3: Polynomial-basis multiplication

6.3.4 $\mathcal{L}$ Code Encoders and Checkers

Encoders, decoders and/or checkers of linear codes are well studied in the literature, e.g., see [50] for shift register based architecture.

For encoding, data (i.e., an element of $GF(2^m)$) is multiplied by generator polynomial, $g(x)$. The encoder can be implemented in serial fashion using shift registers and combinational gates or in parallel fashion using only combinational circuits. In this work, we only consider the parallel one, since it is much faster. For parallel implementation of an encoder, a parallel multiplier that multiplies the data by a generator $g(x)$ should be used.

To check whether an n-tuple at a certain location in the circuit is a codeword, a checker is placed at that point. A checker, basically, divides the polynomial corresponding to the n-tuple by the generator polynomial, $g(x)$, of the $\mathcal{L}$ code and if the division has a nonzero remainder, an error signal is given. Again, checkers can be implemented in serial fashion using linear feedback shift registers or in parallel fashion using only combinational logic. For parallel implementation, a parallel divider can be used.

6.4 Error Detection Capability

In this section, our error model and the probability of an undetected error of the SIE scheme are given. The frequency of the check points is also discussed.

6.4.1 Error Modelling

Similar to Section 4.2, the error model in this work is a bit-flip model. To illustrate the model, suppose that the error free value of a location, say L, of a polynomial basis multiplier is an n-tuple, say $v = (v_0, v_1, \cdots, v_{n-1})$. An error vector is also an n-tuple, say $e = (e_0, e_1, \cdots, e_{n-1})$. The number of possible errors is $2^n - 1$. The erroneous value of the location L is $v_e = v + e$, where '+' is bitwise XOR. In other words, an error is a modulo-2 additive term at a certain location of a PB multiplier and the i^{th} bit of the error vector e being one implies that the i^{th} bit of the value of the location L has changed from 0 to 1 or vice versa. If the location is one of the modules (SR, SM or VA), without loss of generality we can assume that the error vector should be XORed with the output of the component.

Note that the encoders and checkers should be fault free or at least self checking [53]. Since in practice the number of redundant bits, $n-m$, is expected to be much less than the size of the input operands of the multiplier, m, the self checking

technique is feasible. In this work, for simplicity we assume that these encoders and decoders are error free. It will be shown in Section 6.8 that with a moderate number of redundant bits the probability of error detection becomes quite close to unity.

6.4.2 Probability of an Undetected Error

For the purpose of error detection, a received n-tuple should be checked if it is still a codeword or not. Therefore, based on our error model, any nonzero error that is a multiple of the generator polynomial $g(x)$ cannot be detected. In other words, any nonzero error vector from the set of all codewords is an undetectable error.

Let the probability of error detection and the probability of an undetected error be referred to as Pr_D and Pr_U, respectively. Clearly, $Pr_D = 1 - Pr_U$. Suppose W_i is the number of codewords of weight i in an (n, m) $\mathcal{L}$ code, i.e., W_i is the number of codewords that contain i ones. The probability of an undetected error can be computed using such weight distribution of the code. As mentioned, an undetected error occurs when the error vector is among one of the nonzero codewords. Thus,

$$Pr_U = \sum_{i=1}^{n} W_i p^i (1-p)^{n-i},$$

where p is the probability of a bit of error vector being one.

The weight distribution is known for some special codes such as Hamming codes and Reed-Solomon codes, however, the distribution is not known for the one we use in this work. Hence, a closed form for Pr_U cannot be obtained and the probability of an undetected error is investigated by simulation. In this simulation, we generated a large number (one million) of error vectors. These vectors were generated based on the error model discussed in Section 6.4.1. In other words, the probability of having one bit of the vector being one is p. Then we investigated how many of these

error vectors cannot be detected. As mentioned in the previous section, assuming that v is an error free value of a location and e is the error vector in that location, we have $v_e = v + e$. Since v is a valid codeword, it is divisible by $g(x)$. Therefore, v_e becomes a valid codeword if and only if e is a valid one, i.e., e is divisible by $g(x)$. As a result, to investigate the probability of an undetected error for the $\mathcal{L}$ code, we need to determine the probability of having an error vector as a valid codeword of the code. Figure 6.4 shows the result of our simulation for $(167, 163)$, $(169, 163)$ and $(171, 163)$ $\mathcal{L}$ codes with generator polynomials $g(x) = x^4 + x + 1$, $g(x) = x^6 + x + 1$, and $g(x) = x^8 + x^4 + x^3 + x^2 + 1$, respectively.

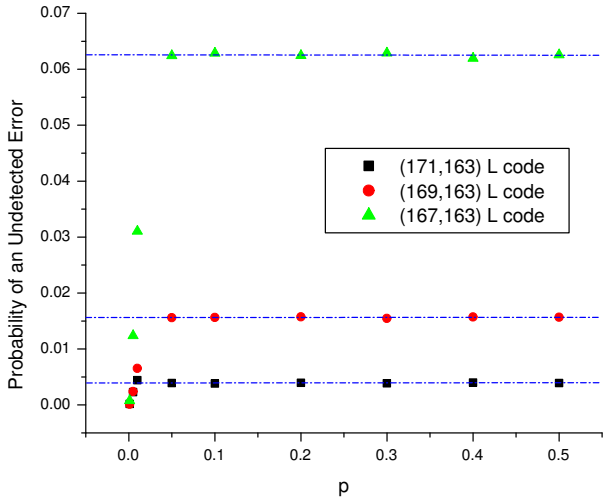


Figure 6.4: Probability of an undetected error vs. p

A well-known upper bound for the probability of an undetected error for some (n, m) codes such as the Hamming code is $2^{-(n-m)}$. Here, the numbers of redundant bits are 4, 6 and 8, and the dashed and dotted lines in Figure 6.4 show the values 2^{-4}, 2^{-6} and 2^{-8}, respectively. As it can be seen in the figure, the values of Pr_U are either smaller than or quite close to the bounds for all three cases.

6.4.3 Frequency of Check Points

Similar to Section 4.2.3, suppose that there are several multiple-bit errors in a location of the circuit of a PB multiplier. For having an error detection capability Pr_D as discussed in previous section, each of the above mentioned locations in Section 6.3.3 should have a checker. This requires a very high area overhead especially for bit-parallel multipliers. The following lemma enables us reduce the number of checkers considerably.

Lemma 6.1 *Suppose only a maximum of one multiple-bit error occurs per round of a bit-serial multiplier or per slice of a bit-parallel multiplier (see Figure 6.3). Then any such error can be detected with probability Pr_D, discussed in Section 6.4.2, using a parity checker at L3 of the bit-serial multiplier or a parity checker before the vertical input of every VA and one parity checker after the final VA in the bit-parallel multiplier.*

The proof can be found in Section 4.2.3.

6.5 Double-Input Encoding (DIE)

Having only one input of the PB multiplier encoded can be of concern. If the second input of the multiplier becomes erroneous, it cannot be detected. One way to improve this situation is to encode both input operands. In general, the generators for encoding inputs can be different. However, there are some issues with regard to choosing the generators that need to be dealt with and they are briefly discussed in Section 6.5.2.

6.5.1 Polynomial Basis Multipliers with CED Capability

In the double-input encoding, input A is encoded by the generator $g_1(x)$ and B by $g_2(x)$. Let $C = A \cdot B \mod f(x)$, where $f(x)$ is the field defining polynomial. Multiplying each side by $g_1(x)g_2(x)$, we obtain:

$$Cg_1g_2 = ABg_1g_2 \mod fg_1g_2.$$

Hence,

$$E_{g_1g_2}(C) = E_{g_1}(A)E_{g_2}(B) \mod \mathcal{F}(x),$$

where $\mathcal{F}(x) = f(x)g_1(x)g_2(x)$ and $E_g(Z)$ implies that Z is encoded by generator g. Let the degrees of $g_1(x)$ and $g_2(x)$ be r_1 and r_2, respectively. Clearly, the degree of $\mathcal{F}(x)$ is $\mathcal{N} = m + r_1 + r_2$.

An SR module can be constructed using (6.5) and by replacements of $F(x)$ and n with $\mathcal{F}(x)$ and $\mathcal{N}$, respectively. To construct a bit-serial multiplier and/or a bit-parallel multiplier with concurrent error detection capability, we use updated versions of SR, SM, and VA modules in a very similar manner as shown in Figure 6.3. Here, the number of rounds of the bit-serial multiplier and the number of slices of the bit-parallel multiplier are $m + r_2$ each. Figure 6.5 shows a complete bit-parallel multiplier with CED using the DIE scheme.

6.5.2 Error Detection Using DIE

Like Section 6.4.1, here, the bit-flip error model is assumed. As shown in Figure 6.5, for the purpose of error detection, checkers that use the generator g_1 are placed in the same locations as discussed in Section 6.3.3. If there is no error in the circuit, then the output value of the last checker that uses the generator g_1 is $Cg_2 = ABg_2$. Therefore, one more checker that uses the generator g_2 should be placed at the output of the last checker. Then the final result of the multiplication is the output

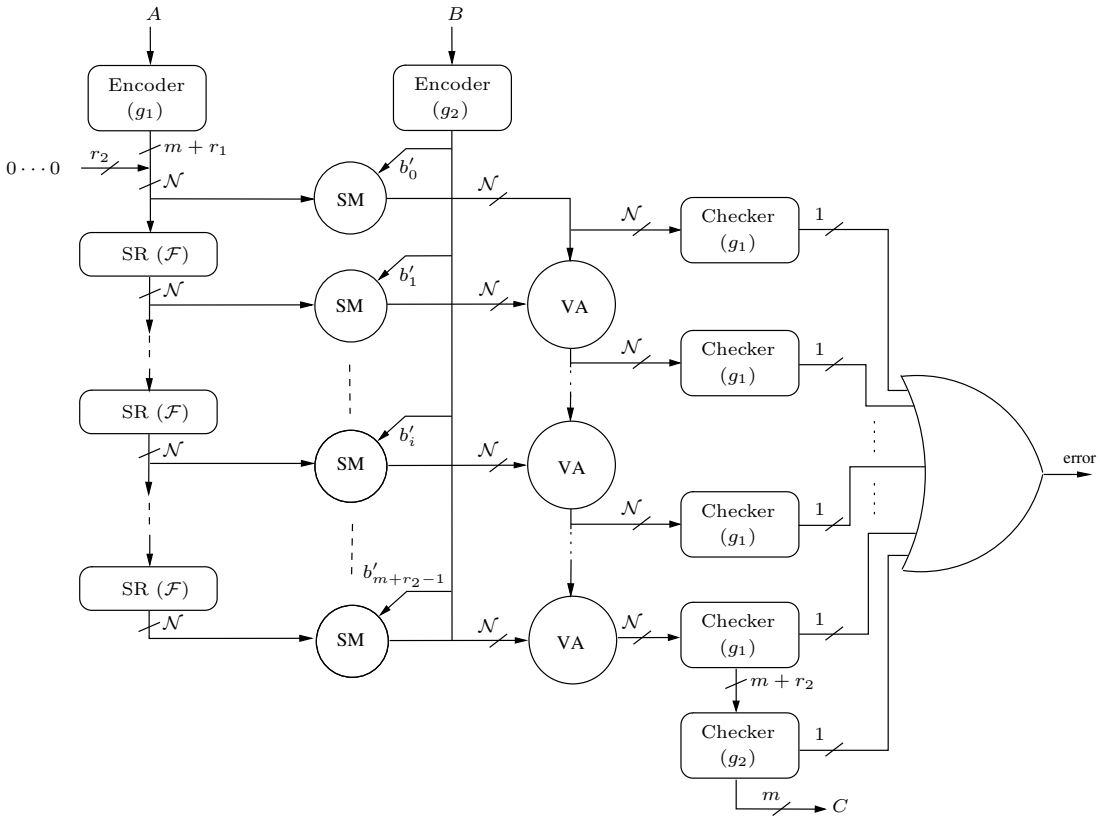


Figure 6.5: A complete bit-parallel multiplier with CED using the DIE scheme

of the checker that used the generator g_2. Assuming that only a maximum of one multiple-bit error occurs per round of a bit-serial multiplier or per slice of a bit-parallel multiplier, we have:

- if an error occurs on input B and the error is a multiple of g_2, it cannot be detected.
- if errors occur on input A and/or inside the PB multiplier and they are not multiples of g_1, they are detected. If they are multiples of g_1 but the output of the last checker that used generator g_1 is not a multiple of g_2, the errors are detected as well. Otherwise, they are not detected.

Note that g_2 can be preferably chosen such that its degree is smaller than that of g_1. Polynomial g_2 is mainly used for detecting errors in input B although it affects the error detection of the entire multiplier circuit. Furthermore, this choice decreases the area overhead of the scheme.

6.6 Hybrid Scheme

As presented in Section 6.8, the parallel implementations of the SIE and the DIE schemes have high area overheads. In this section, we present a hybrid scheme whose error detection capability is similar to the DIE scheme but the area overhead of the parallel implementation is much lower and the time overhead appears to be reasonable for some practical applications.

6.6.1 Polynomial Basis Multipliers with CED Capability

This scheme combines the SIMP scheme and input encoding scheme discussed above. In this hybrid scheme, input B is encoded by the generator $g'(x)$, where the degree of $g'(x)$ is r'. Then $A' =$'$0 \cdots 0A$' is divided into k parts and a parity bit is assigned to each partition, where the number of appended zeros is r'. The reason of extending A before partitioning follows. Considering $C = A \cdot B \mod f(x)$, where $f(x)$ is the field defining polynomial with degree m, we multiply each side by $g'(x)$ as follows:

$$Cg' = ABg' \mod fg'.$$

Let $\mathcal{F}' = fg'$. Since the modulus ($\mathcal{F}'$) is of degree $m + r'$ and we use a bit-level architecture for multiplication (see Figure 6.3), all the intermediate and the final results are of degree $m + r' - 1$. Therefore, we extend the size of input A before

partitioning. Now, the hybrid scheme can be described by the following equation:

$$\begin{aligned}\{P_k(E_{g'}(C)), E_{g'}(C)\} = \\ \{P_k(A'), A'\} \cdot E_{g'}(B) \mod \{P_k(\mathcal{F}'), \mathcal{F}'\},\end{aligned} \tag{6.6}$$

where $E_{g'}(Z)$ implies that Z is encoded by generator g' and $\{P_k(Z), Z\}$ is a vector and implies that the SIMP scheme with k partitions is used. In this scheme the size of SM and VA modules should be extended to $m + r' + k$.

Lemma 6.2 *Let* $V_s(x) = \sum_{i=0}^{m+r'-1} v_{s_i} x^i$ *and* $V(x) = \sum_{i=0}^{m+r'-1} v_i x^i$ *be the output and input of the SR module, respectively. Additionally, suppose the SR module depends on the polynomial* $\mathcal{F}' = E_{g'}(F) = \sum_{i=0}^{m+r'} f'_i x^i$. *Then in the hybrid scheme, the SR module is updated as follows:*

$$\begin{aligned}V_s(x) &= \sum_{j=0}^{k-1} x^{jl} \left(v_{jl-1} + \sum_{i=1}^{l-1} v_{jl+i-1} x^i + v_{m+r'-1} \sum_{i=0}^{l-1} f'_{jl+i} x^i \right), \\ [P_k(V_s)]_j &= v_{jl-1} + [P(V)]_j + v_{(j+1)l-1} + v_{m+r'-1}[P(\mathcal{F}')]_j,\end{aligned} \tag{6.7}$$

where $0 \le j \le k-1$, $v_{-1} = 0$, $l = \lceil \frac{m+r'}{k} \rceil$ [1], $[P_k(V_s)]_j$ *is the* j^{th} *bit of the* k*-bit parity,* $[P(V)]_j$ *and* $[P(\mathcal{F}')]_j$ *are the parity of the* j^{th} *part of* V *and* $\mathcal{F}'$, *respectively.*

The proof of the lemma can be found in Section 4.1.1.

To construct a bit-serial multiplier and/or a bit-parallel multiplier with concurrent error detection capability, we use updated versions of SR, SM, and VA modules in a very similar manner as shown in Figure 6.3. Since the length of $E_{g'}(B)$ is $m+r'$, the number of rounds of the bit-serial multiplier and the number of slices of the bit-parallel multiplier are $m + r'$ each. Assuming that $\mathcal{N}' = m + r' + k$, Figure 6.6 shows a bit-parallel multiplier with CED using the hybrid scheme.

[1] If $m + r'$ is not divisible by k, one can append necessary zeros as most significant bits. See Section 4.1 for other methods of partitioning.

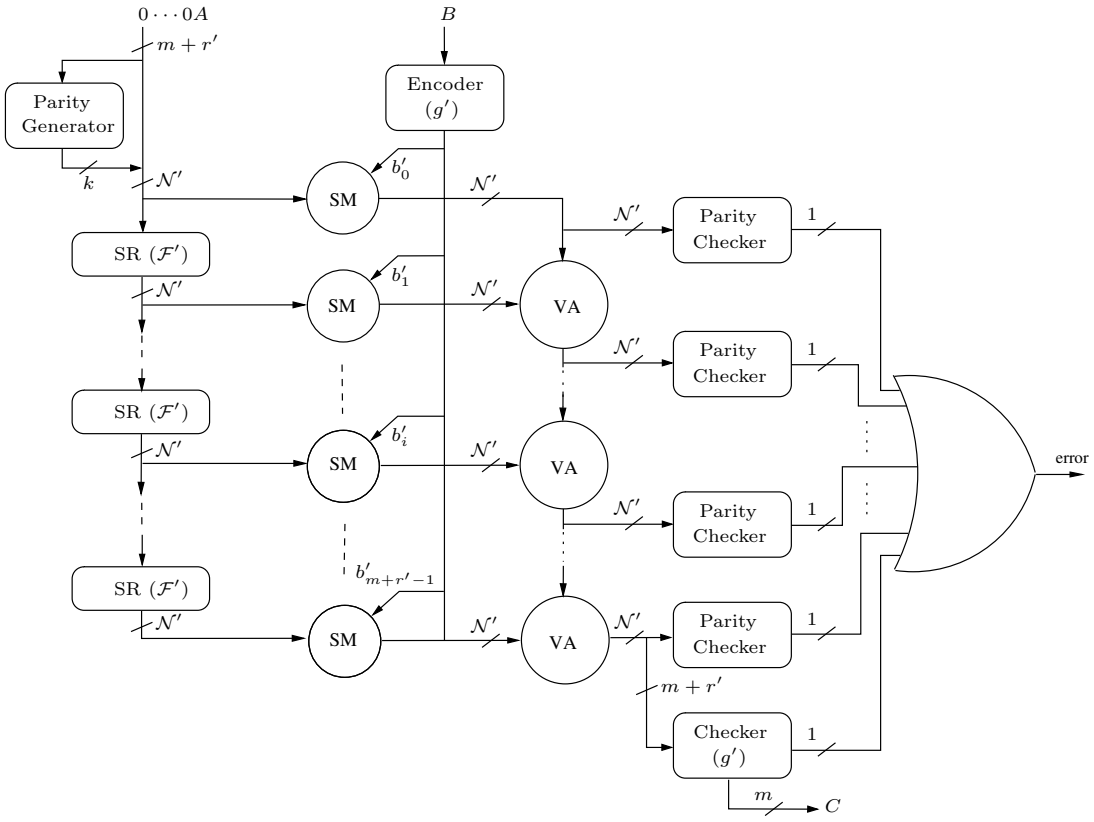


Figure 6.6: A complete bit-parallel multiplier with CED using the hybrid scheme

6.6.2 Error Detection Using Hybrid Scheme

Having the same error model as the SIE and the DIE schemes, we develop the error detection strategy as follows (see Figure 6.6). A k-bit parity equality checker is placed after the VA module of each round of the bit-serial multiplier (or each slice of the bit-parallel one). The k-bit parity equality checker computes the parities of the k partitions of the actual value and compares them against the k predicted parity bits. The checker gives an error signal for any inequality. If there is no error in the circuit, the output after the last VA module is $Cg' = ABg'$. If only a maximum of one multiple-bit error occurs per round of a bit-serial multiplier or per slice of a bit-parallel multiplier, we have:

- if an error occurs on input B and the error is a multiple of g', it cannot be detected.

- if errors occur on input A and/or inside the PB multiplier and the parity of each error in at least one partition is not zero, they are detected. If the parity of the error in each partition is zero but the output of the last VA module is not a multiple of g', the errors are detected as well. Otherwise, they are not detected.

Similar to the DIE scheme, here g' can be preferably chosen such that its degree is smaller than k, since this choice can decrease the area overhead of the scheme.

6.7 Simulation-Based Fault Injection

To evaluate the capability of error detection of the SIE, DIE and hybrid schemes, we injected a large number of stuck-at faults into a C model of a $GF(2^{163})$ bit-parallel PB multiplier. The field defining polynomial was $x^{163} + x^7 + x^6 + x^3 + 1$. Also, the generators for SIE and DIE were $g_1(x) = x^8 + x^4 + x^3 + x^2 + 1$ and $g_2(x) = x^3 + x + 1$. For the hybrid scheme, the first input is divided into 8 parts, i.e., an SIMP with $k = 8$ is used, and the second input is scaled by $g'(x) = g_2(x)$. Since fault injection in a complete PB multiplier is extremely time consuming, we performed fault injection in a slice of the PB multipliers. The faults were injected at the inputs and the outputs of the gates of a slice of PB multipliers. Additionally, a fault was injected at the b_i or b'_i input of each SM module as well.

As mentioned earlier, in a slice of a $GF(2^m)$ PB multiplier, the number of two-input gates for SR, SM and VA modules are $(\omega - 2)$ XOR gates, m AND gates and m XOR gates, respectively, where ω is the Hamming weight of the field defining polynomial. Additionally, except for the outputs of AND gates of the SM module, where are the direct inputs of XOR gates of VA modules, all other inputs and

outputs can be locations for fault injection. Therefore, considering the b_i or b'_i input of the SM module, the number of locations for fault injection in a slice of the PB multiplier is $3(\omega - 2) + 5m + 1$. Appendix A presents more information about our procedures of the fault injections.

6.7.1 Single Stuck-at Fault Injection

For single stuck-at fault injection, we injected two faults (zero and one) at every pin location. Hence, the total number of injected faults in a slice of the $GF(2^m)$ PB multiplier was $6(\omega - 2) + 10m + 2$. Also, we injected all above-mentioned faults for one million and one random inputs. Table 6.1 shows the number of injected faults for each random input along with the result of the simulation for the SIE, DIE and hybrid schemes. The number of detected, masked and undetected faults are 656518037, 992483822 and 999791 for SIE, 657517827, 992483823 and 0 for DIE and 657507841, 992493809 and 0 for Hybrid, respectively.

Error detection scheme	No. of stuck-at faults	No. of random inputs	Percentage of error detection
SIE	1766	1000001	99.85%
DIE	1856	1000001	100%
Hybrid	1784	1000001	100%

Table 6.1: Single stuck-at fault injection in a slice of a bit-parallel $GF(2^{163})$ PB multiplier

The DIE and hybrid schemes can detect 100% of single stuck-at faults. However, percentage of error detection of SIE is less than 100%. In fact, SIE cannot detect errors on input B. This issue was addressed in Section 5.2.

6.7.2 Multiple Stuck-at Fault Injection

To perform multiple stuck-at fault injection, we randomly chose a number of the locations mentioned earlier, i.e., $3(\omega - 2) + 5m + 1$ locations, in a slice of the PB multiplier and then we randomly injected either stuck-at 0 fault or stuck-at 1 fault at each chosen location. In this experiment, 500 random multiple stuck-at faults were injected for each of 1000000 million random inputs. Since the experiment is very time consuming (see Section 5.2), we performed this simulation only for one slice. The number of detected and undetected faults are 498046733 and 1953267 for SIE, 498474218 and 1525782 for DIE, and 498474199 and 1525801 for Hybrid, respectively. The result of the simulations are presented in Table 6.2.

Error detection scheme	No. of stuck-at faults	No. of random inputs	Percentage of error detection
SIE	500	1000000	99.60935%
DIE	500	1000000	99.69485%
Hybrid	500	1000000	99.69484%

Table 6.2: Multiple stuck-at fault injection in a slice of a bit-parallel $GF(2^{163})$ PB multiplier

Comparing the Pr_U values and the percentage of error detection for the SIMP and SIE schemes, one can conclude that these schemes have almost same error detection capabilities. Accordingly, the DIE and hybrid schemes have almost same capabilities.

6.8 Analysis of Time and Area Overheads

In this section, area and time overheads of the SIE, the DIE and the hybrid error detection schemes are investigated.

We used the NIST recommended field defining polynomials for ECDSA $f(x) =$

$x^{163}+x^7+x^6+x^3+1$ for our bit-serial implementations. Due to a resource limitation, we could implement the bit-parallel schemes for $m = 144$ using the field defining polynomial $f(x) = x^{144} + x^7 + x^4 + x^2 + 1$. Furthermore, the code polynomial for the SIE scheme was of degree 8 and two code polynomials required for the DIE scheme were of degrees 8 and 3. For the hybrid scheme, eight parity bits and a code polynomial of degree 3 were used. We described the scheme by VHDL to obtain a realistic approximation of the area overhead. We used Modelsim^{TM} to simulate the design for checking its correct functionality and we implemented the scheme on a Xilinx Spartan 3 (XC3S5000) FPGA using Xilinx ISE 7.1i.

The area overhead and the time overhead (clock period overhead or latency overhead) of the bit-serial implementations of the SIE, the DIE, and the hybrid schemes for a polynomial basis multiplier are given in Table 6.3. The DIE scheme has the largest area overhead and very small time overhead. The hybrid scheme has the largest time overhead, but still is in a reasonable range, and the smallest area overhead. On the other hand, as mentioned earlier, the DIE and the hybrid schemes have similar capabilities of error detection which are higher than the SIE scheme. Therefore, for bit-serial implementations, one can choose any of the schemes based on the area overhead, time overhead and/or error detection capability.

	Bit-serial implementations		
Overhead	SIE	DIE	Hybrid
area (%)	39.71	52.94	24.33
clock cycle	0	$r_2 = 3$	$r_2 = 3$
clock period (%)[1]	0	0	12.60
latency (%)	0	1.84	14.67

[1]can be considered as throughput overhead.

Table 6.3: The time and the area overheads for the bit-serial implementations of the SIE, the DIE and the hybrid schemes

The time and area overheads of bit-parallel implementations of the schemes are

also investigated. As shown in Table 6.4, both the time and the area overheads of the hybrid scheme are significantly lower than the SIE and the DIE schemes. Therefore, the best choice in bit-parallel implementations is the hybrid scheme. Note that one can reduce the number of intermediate checkers in bit-parallel implementations in order to achieve lower area and time overheads but this causes a reduction in the capability of error detection.

	Bit-parallel implementations		
Overhead	SIE	DIE	Hybrid
area (%)	164.96	169.95	55.14
Time (%)[1]	165.71	191.19	69.98

[1]can be considered as maximum delay overhead.

Table 6.4: The time and the area overheads for the bit-parallel implementations of the SIE, the DIE and the hybrid schemes

6.9 Summary

This chapter has investigated three schemes for detection of multiple-bit random errors in binary polynomial basis multipliers using linear codes. Based on our simulation, the probability of an undetected error for the $\mathcal{L}$ code is approximately 0.004 with eight redundant bits in the codewords. Also, the error detection capabilities of the schemes are evaluated by a number of simulation-based fault injections. Among three schemes presented in this chapter, one has a similar percentage of error detection as SIMP and the other two schemes have slightly better percentage of error detection compared to DIMP. Furthermore, the overheads of the error detection schemes for bit-serial implementations are lower than the overhead of the dual modular redundant scheme for a sufficient number of redundant bits. Additionally, the time overheads of the schemes have been observed to be small, i.e., less than 15%. In bit-parallel implementations, among all three linear code based schemes,

the hybrid scheme has acceptable area and time overheads.

Chapter 7

Conclusions and Future Work

7.1 Summary and Conclusions

In this thesis, a number of schemes for concurrent error detection of the binary extension field operations have been presented. For each scheme, the error detection capability has been evaluated by simulation-based fault injections. Moreover, the time and space complexities of each scheme have been investigated.

A number of schemes have been presented in Chapter 3, which are efficient for pipelined architectures and are based on recomputing with shifted operands (RESO) method. These schemes have been developed to concurrently detect errors in polynomial, dual, type I and type II optimal normal bases arithmetic operations. We have also presented one semi-systolic multiplier for each of above-mentioned bases and applied the CED scheme to them. We have compared these multipliers with a number of previously published systolic and/or semi-systolic ones. The results show that this scheme can be considered among the best. Also, a simulation-based fault injection has been performed for each of the multipliers. Results of the simulations for single stuck-at faults show 100%, 100%, 99.66% and 100% error detection for polynomial, dual, type I and type II bases multipliers, respectively.

The simulations also show that the percentage of error detection of this scheme for the above-mentioned multipliers against multiple stuck-at faults is 100%. Finally, we have also commented on how RESO can be used for concurrent error correction to deal with transient faults.

The third scheme, single-input multiple-parity (SIMP) scheme, is to detect errors in bit-serial and/or bit-parallel polynomial basis multipliers. In this scheme one input of the multiplier has been divided into a number of partitions. In this scheme, the probability of error detection for random errors is more than 75% and it quickly approaches unity for approximately 8 parity bits. The overhead of our implementation tends to increase linearly as the number of parity bits increases. Results show that the area overhead cost of the bit-serial implementation is lower than that for the bit-parallel one. Both implementations have lower area overheads than the traditional dual modular redundant scheme for a sufficient number of parity bits. Additionally, the average time overhead due to the use of the scheme in bit-parallel implementations is around 25%, while for bit-serial implementations time overheads have been observed to be small to negligible. This scheme is also extended to dual, type I and type II normal bases multipliers.

The fourth scheme, double-input multiple-parity (DIMP) scheme, is a concurrent error detection scheme for polynomial basis multipliers and can be considered as an extension to SIMP. In this scheme, multiple parity bits are used for both inputs of the multiplier and hence the scheme can detect errors on both inputs and/or inside of the multiplier. This scheme can be applied to digit-serial and bit-parallel multipliers. Based on the simulations, the percentage of error detection of DIMP is slightly more than SIMP. Additionally, the area overhead of DIMP is slightly higher than SIMP. This scheme is also extended to dual, type I and type II normal bases multipliers.

This thesis has also investigated three schemes for detection of multiple-bit ran-

dom errors in binary polynomial basis multipliers using linear codes. Based on our simulation, the probability of an undetected error for the $\mathcal{L}$ Code presented in Chapter 6 is approximately 0.004 with eight redundant bits in the codewords. Furthermore, the overheads of the error detection schemes for bit-serial implementations are lower than the overhead of the dual modular redundant scheme for a sufficient number of redundant bits. Additionally, the time overheads of the schemes have been observed to be small, i.e., less than 15%. In bit-parallel implementations, among all three linear code based schemes, the hybrid scheme has acceptable area and time overheads.

As mentioned earlier, the RESO based schemes are efficient for pipelined architectures such as systolic arrays. However, the last five schemes can be applied to both bit-serial and/or bit-parallel multipliers and in this sense they are more general. Table 7.1 compares the last five schemes in terms of their overheads and error detection capabilities.

	From the lowest to highest				
Overhead	SIMP	DIMP	Hybrid	SIE	DIE
	From the highest to lowest				
Error detection capability	Hybrid DIE		DIMP	SIMP SIE	

Table 7.1: Comparison of the SIMP, DIMP, SIE, DIE and hybrid schemes

7.2 Future Work

In this thesis, for the RESO based scheme, the number of required gates and their delays have been given as estimations for time and area complexities. These estimations are quite realistic, however, a more accurate and actual one can be evaluated by actual VLSI implementation of those operations with CED. Moreover, as

mentioned earlier, applying RESO based scheme to polynomial and dual bases exponentiation are not inexpensive. Therefore, in addition to suggested look-up table based method in the thesis, other methods to decrease the overheads are desirable.

The SIMP, DIMP, SIE, DIE and hybrid schemes have been implemented in bit-serial and/or bit-parallel fashion[1]. It is expected to have the area overheads of the digit-serial implementations smaller than those of bit-parallel implementations and larger than those of bit-serial ones. An FPGA or ASIC implementation can give the accurate results. Furthermore, these five schemes have been applied to conventional multipliers. Although some CED schemes were already applied to special multipliers such as Montgomery multipliers [19], the applications of the above-mentioned schemes to such multipliers are also desirable

The SIE, DIE and hybrid schemes may be extended to the multipliers in the bases other than the polynomial basis. Also, one can investigate finding a non-linear technique to be applied to the multipliers or other operations with a reduced overhead.

It is worth mentioning that although applying the proposed schemes and in turn adding some error checking circuitry can help to detect random errors in finite field multipliers, it may be possible that a cryptographic systems that uses such extra circuit-equipped multipliers becomes more vulnerable to some types of side-channel attacks, e.g., power analysis attacks [54]. Effects of such side-channel attacks on the error-detecting finite field multipliers proposed in thesis may be investigated in future.

[1]Note that DIMP cannot be implemented in bit-serial fashion.

Appendix A

Simulation-Based Fault Injection

In this appendix, the simulation-based fault injection technique is explained. For each proposed scheme, we performed both single-bit fault and multiple-bit fault injections. To inject single-bit faults, we chose a slice of a bit-parallel multiplier and considered a number of locations in that slice. Usually these locations were all the input and output pins of the gates in the slice. As mentioned earlier, if an input of one gate was directly connected to the output of another gate, usually one of them was injected. Furthermore, for a set of inputs A and B, both single stuck-at 0 and single stuck-at 1 were injected at all locations. The last step was performed for one million pairs of random inputs.

To inject one multiple-bit fault, a number of above-mentioned locations, e.g., 2 to 5 locations for some experiments, in the chosen slice were randomly selected and a random value (zero or one) was injected at each of the selected locations. We injected 500 multiple-bit faults for each set of inputs A and B. The previous step was also repeated for one million pairs of random inputs.

To explain the fault injection procedure with more details, a number of pseudo codes are given. Procedure A.1 presents the main function of either a single or a multiple stuck-at fault injection.

Procedure A.1 Main procedure of a fault injection
for 1000000 times **do**

1. Generate random input A
2. Generate random input B
3. Perform fault injection at simulated circuits with inputs A and B

end for

In the following, step 3 of Procedure A.1 is explained for different schemes.

A.1 Fault Injection in Information Redundancy Based Schemes

The information redundancy based schemes are SIMP, DIMP, SIE, DIE and Hybrid. For these schemes, both transient and permanent faults can be considered and they are treated similarly. Procedure A.2 presents the single stuck-at fault injection for these schemes.

Procedure A.2 Single stuck-at fault injection for information redundancy based schemes

1. FFC ⟵ Fault free computation
2. **Inject** one single stuck-at fault at one arbitrary slice {
   (a) RCF ⟵ Result of the computation in presence of fault
   (b) PP ⟵ Predicted parity of the output of the VA module of the slice
   (c) AP ⟵ Actual parity of the output of the VA module of the slice
   (d) **if** (PP $\neq$ AP) **then**
   an error/fault was detected
   else if (RCF = FFC) **then**
   a fault was masked
   else
   an error/fault was not detected
3. }**until**{all locations (mentioned earlier) are injected with both stack-at 0 and stuck-at 1 faults}
4. Percentage of error detection$(PED) = \frac{\text{No. of detected errors/faults}}{\text{No of all faults– No. of masked faults}}$

Furthermore, Procedure A.3 presents the multiple stuck-at fault injection for these schemes.

Procedure A.3 Multiple stuck-at fault injection for information redundancy based schemes

1. FFC ⟵ Fault free computation
2. **for** 500 times **repeat**{
   (a) Randomly choose a number of locations
   (b) Inject randomly either zero or one at selected locations
   (c) RCF ⟵ Result of the computation in presence of fault
   (d) PP ⟵ Predicted parity of the output of the VA module of the slice
   (e) AP ⟵ Actual parity of the output of the VA module of the slice
   (f) **if** (PP ≠ AP) **then**
   an error/fault was detected
   else if (RCF = FFC) **then**
   a fault was masked
   else
   an error/fault was not detected}
3. $PED = \frac{\text{No. of detected errors/faults}}{\text{No of all faults– No. of masked faults}}$

A.2 Fault Injection in RESO Based Schemes

As mentioned in Section 3.5, this scheme can detect all transient faults. Hence, we perform the simulations for permanent faults. Procedure A.4 presents the single stuck-at fault injection for these schemes.

Procedure A.4 Single stuck-at fault injection for RESO based schemes

1. FFC ⟵ Fault free computation
2. Compute and store the encoded inputs
3. **Inject** one single stuck-at fault at one arbitrary row{
   (a) CUI ⟵ Computation with unencoded inputs
   (b) CEI ⟵ Computation with encoded inputs
   (c) **if** (CUI ≠ CEI) **then**
   an error/fault was detected
   else if (CUI = FFC) **then**
   a fault was masked
   else
   an error/fault was not detected
4. }**until**{all locations (mentioned earlier) are injected with both stack-at 0 and stuck-at 1 faults}
5. $PED = \frac{\text{No. of detected errors/faults}}{\text{No of all faults– No. of masked faults}}$

Additionally, Procedure A.5 presents the multiple stuck-at fault injection for these schemes.

Procedure A.5 Multiple stuck-at fault injection for RESO based schemes

1. FFC ⟵ Fault free computation
2. Compute and store the encoded inputs
3. **for** 500 times **repeat**{
   (a) Randomly choose a number of locations;
   (b) Inject randomly either zero or one at selected locations
   (c) CUI ⟵ Computation with unencoded inputs
   (d) CEI ⟵ Computation with encoded inputs
   (e) **if** (CUI ≠ CEI) **then**
   an error/fault was detected
   else if (CUI = FFC) **then**
   a fault was masked
   else
   an error/fault was not detected}
4. $PED = \frac{\text{No. of detected errors/faults}}{\text{No of all faults– No. of masked faults}}$

It is worth mentioning that for optimal normal bases, a permutation should be performed on the inputs of each computation to convert them from NB to ONB. Additionally, an inverse permutation should be performed on the output of each computation to convert it from ONB to NB (see Section 3.4.3).

Bibliography

[1] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design.* Wiley-IEEE Press, NewYork, 1994.

[2] G. B. Agnew, T. Beth, R.C. Mullin, and S.A. Vanstone. Arithmetic operations in $GF(2^m)$. *Journal of Cryptography*, 6(1):3–13, 1993.

[3] R. J. Anderson and M. G. Kuhn. Low cost attacks on tamper resistant devices. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 125–136, London, UK, 1998.

[4] S. Bayat-Sarmadi and M. A. Hasan. Concurrent error detection of polynomial basis multiplication over extension fields using a multiple-bit parity scheme. In *Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 102–110, Monterey, CA, 2005.

[5] S. Bayat-Sarmadi and M. A. Hasan. Concurrent error detection in finite field arithmetic operations using pipelined and systolic architectures. CACR Technical Report archive, Report 2007-30, 2007. http://www.cacr.math.uwaterloo.ca/techreports/2007/cacr2007-30.pdf.

[6] S. Bayat-Sarmadi and M. A. Hasan. Detecting errors in a polynomial basis multiplier using multiple parity bits for both inputs. In *Proceedings of the 25th*

IEEE International Conference on Computer Design (ICCD), pages 368–375, Lake Tahoe, CA, 2007.

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

[8] S. Bayat-Sarmadi and M. A. Hasan. Run-time error detection of polynomial basis multiplication using linear codes. In *Proceedings of the 18th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 204–209, Montreal,Canada, 2007.

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

[10] I. Biehl, B. Meyer, and V. Muller. Differential fault attacks on elliptic curve cryptosystems. In *Proceedings of the 20th Int'l Conf. CRYPTO*, pages 131–146, Santa Barbara, CA, 2000. Springer-Verlag.

[11] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *CRYPTO '97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, pages 513–525, Santa Barbara, CA, 1997.

[12] R. E. Blahut. *Fast Algorithms for Digital Signal Processing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[13] J. Blomer and M. Otto. Wagners attack on a secure CRT-RSA algorithm reconsidered. In *Proceedings of the 3rd Workshop on Fault Tolerance and Diagnosis in Cryptography (FTDC)*, pages 13–23, Yokohama, Japan, 2006.

[14] J. Blomer, M. Otto, and J. P. Seifert. Sign change fault attacks on elliptic curve cryptosystems. In *Proceedings of the 3rd Workshop on Fault Tolerance and Diagnosis in Cryptography (FTDC)*, pages 36–52, Yokohama, Japan, 2006.

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

[16] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, December 2001.

[17] L. Breveglieri, I. Koren, and P. Maistri. An operation-centered approach to fault detection in symmetric cryptography ciphers. *IEEE Trans. on Computers*, 56(5):635–649, 2007.

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

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

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

[21] J. Daemen and V. Rijmen. AES Proposal: Rijndael. http://www.esat.kuleuven.ac.be/ rijmen/rijndael/rijndaeldocV2.zip, 2001.

[22] A. Dominguez-Oviedo and M. A. Hasan. Improved error-detection and fault-tolerance in ECSM using input randomization. CACR Technical Report archive, Report 2006-41, 2006. http://www.cacr.math.uwaterloo.ca/techreports/2006/cacr2006-41.pdf.

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

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

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

[26] E. Fujiwara, N. Mutoh, and K. Matsuoka. A self-testing group-parity prediction checker and its use for built-in testing. *IEEE Transactions on Computers*, 33(6):578– 583, June 1984.

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

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

[29] C. Giraud. DFA on AES. Cryptology ePrint Archive, Report 2003/008, 2003. http://eprint.iacr.org/2003/008.

[30] S. W. Golomb and G. Gong. *Signal Design for Good Correlation, for Wireless Communication, Cryptography and Radar.* Cambridge University Press, 2005.

[31] M. A. Hasan. Selected topics in cryptographic computation, 2003. ECE 720-T2 lecture notes, University of Waterloo.

[32] N. Joshi, K. Wu, and R. Karri. Concurrent error detection schemes for involution ciphers. In *Proceedings of the 6th International Workshop on Crypto-*

graphic Hardware and Embedded Systems (CHES), pages 400–412, Cambridge, MA, July 2004. Springer-Verlag.

[33] N. Joshi, K. Wu, and R. Karri. Concurrent error detection for involutional functions with applications in fault-tolerant cryptographic hardware design. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1163–1169, June 2006.

[34] R. Karri, G. Kuznetsov, and M. Goessel. Parity-based concurrent error detection in symmetric block ciphers. In *Proceedings of the IEEE International Test Conference (ITC)*, pages 919–926, Charlotte , NC, October 2003.

[35] R. Karri, G. Kuznetsov, and M. Goessel. Parity-based concurrent error detection of substitution-permutation network block ciphers. In *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 113–124, Cologne, Germany, August 2003. Springer-Verlag.

[36] R. Karri, K. Wu, P. Mishra, and Y. Kim. Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1509–1517, December 2002.

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

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

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

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

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

[42] S. Lin and D. J. Costello, Jr. *Error Control Coding: Fundamentals and Applications.* Prentice Hall, Inc., 1983.

[43] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 1989.

[44] D. A. McGrew and J. Viega. The Galois/Counter mode of operation (GCM). NIST Modes of Operation for Symmetric Key Block Ciphers, 2005. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-revised-spec.pdf.

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

[46] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[47] J. K. Omura and J. L. Massey. Computational method and apparatus for finite field arithmetic, May 1986. US Patent 4587627.

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

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

[50] W. W. Peterson and E. J. Weldon. *Error Correcting Codes.* MIT Press, Cambridge, MA, 2nd edition, 1972.

[51] D. K. Pradhan. *Fault-Tolerant Computer System Design.* Prentice Hall, Inc., 1996.

[52] D. K. Pradhan and M. Chatterjee. GLFSR-a new test pattern generator for built-in-self-test. In *Proceedings of the International Test Conference*, pages 481–490, Washington, DC, 1994.

[53] T. R. N. Rao and E. Fujiwara. *Error-Control Coding for Computer Systems.* Prentice Hall, 1989.

[54] F. Regazzoni, T. Eisenbarth, J. Grobschadl, L. Breveglieri, P. Ienne, I. Koren, and C. Paar. Power attacks resistance of cryptographic s-boxes with added error detection circuits. In *Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, pages 508–516, Washington, DC, 2007. IEEE Computer Society.

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

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

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

[58] B. Schneier. *Applied Cryptography.* John Wiley & Sons, Inc., second edition, 1996.

[59] G. Seroussi. Table of low-weight binary irreducible polynomials, August 1998. HP Labs Tech. Report HPL-98-135.

[60] E. S. Sogomonian. Design of built-in self-checking monitoring circuits for combinational devices. *Automation and Remote Control*, 35(2):280–289, July 1974.

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

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

[63] D. Wagner. Cryptanalysis of a provably secure CRT-RSA algorithm. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 92–97, Washington DC, USA, 2004.

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

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

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

[67] H. Wu and M.A. Hasan. Efficient exponentiation of a primitive root in $GF(2^m)$. *IEEE Transactions on Computers*, 46(2):162–172, February 1997.

[68] K. Wu, R. Karri, G. Kuznetsov, and M. Goessel. Low cost concurrent error detection for the Advanced Encryption Standard. In *Proceedings of the IEEE International Test Conference (ITC)*, pages 1242–1248, Charlotte , NC, October 2004.

[69] K. Wu, P. Mishra, and R. Karri. Concurrent error detection of fault-based side-channel cryptanalysis of 128-bit RC6 block cipher. *Microelectronics Journal*, 34(1):31–39, January 2003. Special Issue on Defect and Fault Tolerance in VLSI Systems.

[70] C. S. Yeh, I. S. Reed, and T. K. Truong. Systolic multipliers for finite fields $GF(2^m)$. *IEEE Trans. Computers*, 33(4):357–360, April 1984.