University of California
Los Angeles

Finite Field Arithmetic and its Application in Cryptography

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Electrical Engineering

by

Bijan Ansari

2012

Abstract of the Dissertation

Finite Field Arithmetic and its Application in Cryptography

by

Bijan Ansari

Doctor of Philosophy in Electrical Engineering

University of California, Los Angeles, 2012

Professor Frank Chang, Co-chair

Professor Ingrid Verbauwhede, Co-chair

The groundbreaking idea of public key cryptography and the rapid expansion of the internet in the 80s opened a new research area for finite field arithmetic. The large size of fields in cryptography demands new algorithms for efficient arithmetic and new metrics for estimating finite field operation performance. The area, power, and timing constraints on hand-held and embedded devices necessitate accurate models to achieve expected goals. Additionally, cryptosystems need to protect their secrets and hide their internal operation states against side-channel attacks. Fault-injection attacks or random errors reduce the security of a cryptosystem and can help a cryptanalyst to extract a system's secrets.

This dissertation covers various aspects of finite field arithmetic to provide predictable, efficient, and secure elements for cryptography. We provide architecture for an elliptic curve processor (ECP), which is essentially a finite field processor. We also provide finite field multipliers over polynomial and optimal normal bases for pipeline and parallel architectures.

To further analyze the behavior of finite field multipliers, we formalize timing, area, and energy consumption over binary extension fields. To ensure robustness of the multiplication operation, we provide concurrent error detection (CED) schemes for polynomial and normal base multipliers and provide the probability of error detection.

The dissertation of Bijan Ansari is approved.

Babak Daneshrad

Dejan Markovc

Milos Ercegovac

Ingrid Verbauwhede, Committee Co-chair

Frank Chang, Committee Co-chair

University of California, Los Angeles

2012

To my homeland heroes who defended their country during the 1980-1988 war, ...

and to the victims of the Iranian Green movement.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my committee members and my supervisor. I would also like to thank Deeona Columbia, Office of Student Affairs Manager, for her support.

Vita

2004	M.S. (Electrical Engineering), University of Windsor, Canada.
2004–2006	Research Assistance, Centre for Applied Cryptographic Research, University of Waterloo, Canada.
2008–present	Qualcomm Inc. San Diego, CA

Publications

Bijan Ansari and Ingrid Verbauwhede. , "FO4-based Models for Area, Delay and Energy of Polynomial Multiplication over Binary Fields." , In *IEEE Workshop on Signal Processing Systems: Design and Implementation*, pp. 420–425, 2010.

Bijan Ansari and Ingrid Verbauwhede. , "A Hybrid Scheme for Concurrent Error Detection of Multiplication over Finite Fields." , In *Proceedings of the 2010 IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems*, DFT '10, pp. 399–407, Washington, DC, USA, 2010. IEEE Computer Society.

Bijan Ansari and Ingrid Verbauwhede. , "A new Parallel Scheme for Type II ONB Multiplication with Concurrent Error Detection" ,under submission

Bijan Ansari and Ingrid Verbauwhede. , "Comments On "On Concurrent Detection of Errors in Polynomial Basis Multiplication"" ,under submission

Bijan Ansari and Ingrid Verbauwhede. , "Concurrent Error Detection for Polynomial Basis Multipliers over $GF(q^m)$" ,under submission

Agustin Dominguez-Oviedo, M. Anwar Hasan, and Bijan Ansari. , "Fault-Based Attack on Montgomery's Ladder Algorithm." , *J. Cryptology*, **24**:346–374, 2011.

Bijan Ansari and M. Anwar Hasan. , "High-Performance Architecture of Elliptic Curve Scalar Multiplication." , *IEEE Trans. Computers*, **57**(11):1443–1453, 2008.

Bijan Ansari and Huapeng Wu. , "Efficient finite field processor for GF(2^{163}) and its implementation." , *International Journal of High Performance Systems Architecture*, **1**:106–112, 2007.

Bijan Ansari and M. Anwar Hasan. , "Revisiting finite field multiplication using dickson bases." , Technical report, Scalar Multiplication, CACR Research Report, 2007.

Bijan Ansari and Huapeng Wu. , "Parallel scalar multiplication for elliptic curve cryptosystems." , In *Proceedings of the International Conference on Communications, Circuits and Systems*, volume 1, pp. 71– 73, 2005.

CHAPTER 1

Introduction

The introduction to this thesis briefly gives an overview of modern encryption techniques and the rule of finite field arithmetic in public key cryptography. Then, it summarizes the new contributions made to finite field arithmetic by this research.

Security is an integral part of modern communication networks and is achieved mainly through public key cryptography (PKC). PKC achieves data exchange security between two unfamiliar parties over an untrusted communications channel. The concept of PKC was introduced by Diffie and Hellman in 1976 [DH76]. Shortly thereafter, the first PKC system was contributed by Rivest, Shamir, and Adleman, called RSA. The modern encryption techniques can be divided into two main categories:

Symmetric key In symmetric key cryptography, two parties know each other a priori and have exchanged a shared secret key. The shared key is used to encrypt and decrypt the data exchanged between the two parties, using a publicly known algorithm.

Asymmetric key (public key) In asymmetric key cryptography, each party owns a public key for encryption and a private key for decryption. Public keys are freely distributed. Therefore, in principle, there is no need for two parties to know each other beforehand to encrypt messages and send them to the other party.

The RSA public key cryptosystem is based on the difficulty of integer factorization and is widely accepted for digital signature and key exchange over communication networks. The RSA algorithm uses modular arithmetic over integers of 1024 bits or longer and is computationally intensive. In 1985, Koblitz [Kob87] and Miller [Mil86] independently introduced

elliptic curve cryptography (ECC), which allows shorter operands to be used compared to the RSA. In ECC, an elliptic curve is defined over a finite field, and points on the curve construct a commutative additive group. The ECC is based on the difficulty of discrete logarithm problems on the points of an elliptic curve defined over a finite field.

Public key cryptosystems based on ECC require smaller key size, compared to the RSA. The reason is that the only known attack on ECC is the Pollard's rho algorithm, which runs approximately $O(\sqrt{n})$, where n is the dimension of the underlying finite field. On the other hand, the current fastest factoring algorithm used to break RSA is the General Field Sieve, which is much more efficient than $O(\sqrt{n})$.

Public key cryptosystems are computationally intensive and considerably slower than symmetric key cryptosystems. Efficient arithmetic plays a key role in the implementation of public key cryptosystems. For example, RSA computations are performed over the ring of integers of at least 1024 bits. ECC computations are performed over the finite fields of at least 160 bits.

Smaller size arguments in the ECC arithmetic require less computational power, which results in smaller hardware and less power consumption compared to the RSA. These features make ECC attractive for implementation on constrained devices.

Finite field arithmetic is used in ECC; block ciphers, such as the Advanced Encryption Standard (AES); coding theory, and test vector generation. The most prominent application of finite field arithmetic is in public key cryptography. The core arithmetic operation in a field is the multiplication operation. In ECC, where the dimension of fields is larger than 160 bits, the multiplier dominates the area in hardware and the computation time in software. This is one of the main reasons behind extensive research on finite field multipliers. In block ciphers that use finite fields and ECC, the correctness of finite field operation is vital, because various attacks use error injection techniques to break the cipher.

The ECC implementation challenges can be subdivided into multiple levels:

1. The top level contains elliptic curve scalar multiplication. Consider point P on an ellip-

tic curve E, and integer k. The scalar multiplication is defined as $kP = \underbrace{P \uplus P \uplus \cdots \uplus P}_{k \quad times}$, where the operation $\uplus$ is a commutative operation defined over point P. At this level, proper architecture should be used and computations should be properly streamlined to achieve high-speed operation and to minimize the area.

2. At the second level, where computations are performed over a finite field, efficient computation components are required. At this level, the focus is on implementing a proper architecture for finite field multiplication, because it is the most critical operation in terms of time and space complexity.

3. On the third level, precise models for area, timing, and power consumption are needed to model a finite field multiplier before actual implementation.

4. In the last level, the errorless operation of computational elements assures that cryptographic secrets do not leak as a result of malicious attacks or malfunctions on the cryptosystem.

This thesis follows a top-down approach to cover various aspects of finite field arithmetic in cryptography, covering the four levels explained earlier. The contributions resulting from this research are briefly described in the following paragraphs.

High-performance parallel and pipeline architecture The research led to the development of a high-performance parallel and pipeline architecture for elliptic curve scalar multiplication, along with a semi-pipeline finite field multiplier with a small critical path [AH08]. This architecture is one of the most efficient architectures for ECC reported in the literature [DQ07]. The critical path of a digit serial finite field multiplier is $T = T_{AND} + T_{AND} \log_2 w$, where w is the digit size. The critical path of our multiplier is $T = T_{AND} + T_{AND} \log_2 w/k$, where $k > 1$, due to its pipeline structure (Chapter *2*).

Concurrent error detection We have developed a concurrent error detection for finite field multipliers [AV10b], which improves the speed by a factor of $\times 5$ and area by a factor of $\times 10$ (for 20 parity bits), compared to published results in transactions on

very large-scale integration (TVLSI) systems [BH07a]. Random errors in very large-scale integration (VLSI) systems can happen as a result of long operation hours, harsh working environment, electromagnetic interference, and faulty logic. In cryptography, faulty outputs are particularly undesirable because they may be used to reveal cryptographic secrets, or compromise the security of the system [BDL01]. A concurrent error detection (CED) scheme can be used to ensure correct operation of a cryptosystem [KWM01]. It is always possible to protect a system against errors using N multiple redundancy (NMR). However, NMRs are costly for the obvious reason that they repeat the main system N times (Chapter 3).

New Parallel Algorithm for type II optimal normal basis We have also developed a new parallel algorithm for type II optimal normal basis (ONB) multipliers with concurrent error detection capability. This algorithm is agnostic to the multiplication architecture and can be implemented in digit-serial or sub-quadratic form. It is also suitable for implementation on single instruction, multiple data (SIMD) architectures and multiple CPUs. In addition, our algorithm does not impose any area or delay overhead compared to the Massey-Omura multiplier (Chapter 4).

New models for area, timing and power estimation Traditionally, simple gate counting has been used to estimate the area and delay of finite field multipliers. We have developed a new method based on the FO4 model for area and timing estimation. We have also proposed a new model for power consumption of multipliers over binary extension fields based on the FO4 model. Implementation results indicate that the conventional gate delay model has 40% error, while the presented model has less than 4% error. We also show that the ceiling function, conventionally used in delay equations, is not needed in the new model. While it is believed that the Karatsuba-Ofman multiplier is three times slower than of polynomial multiplier, we show that it is actually two times slower (Chapter 5) [AV10a].

Metric to compare power efficiency We have proposed a metric that can be used to

compare power efficiency of ECC processors, regardless of architecture and underlying technology. Using this metric, it can be shown that some published designs that have been specifically advertised as low power are not actually a low-power design. Low-power cryptography is essential for constrained devices, such as smart card and radio-frequency identification (RFID) tags. While some architectures have been proposed for low-power ECC, currently there are no unified metrics to compare various architectures implemented on different technologies (Chapter 6).

CHAPTER 2

High Performance Architecture of Elliptic Curve Scalar Multiplication

Abstract A high performance architecture of elliptic curve scalar multiplication based on the Montgomery ladder method over finite field $GF(2^m)$ is proposed. A pseudo-pipelined word serial finite field multiplier, with word size w, suitable for the scalar multiplication is also developed. Implemented in hardware, this system performs a scalar multiplication in approximately $6\lceil m/w \rceil (m-1)$ clock cycles and the gate delay in the critical path is equal to $T_{AND} + \lceil \log_2(w/k) \rceil T_{XOR}$, where T_{AND} and T_{XOR} are delays due to two-input AND and XOR gates respectively and $1 \leq k \ll w$ is used to shorten the critical path.

2.1 Introduction

Elliptic curve scalar multiplication kP, where k is an integer and P is a point on the curve, is a fundamental operation in elliptic curve crypto-systems. In the recent past, a number of hardware architectures have been proposed in the literature to speed up this operation, for example see [Mis06, DH04, GSE02]. Among them, parallel and pipeline structures have emerged as the most promising ones for high performance systems.

Elliptic curve scalar multiplication is normally performed by repeating point addition (ECADD) and doubling (ECDBL) operations over the curve in some special way. ECADD and ECDBL operations in turn rely on finite field (FF) operations such as addition/subtraction, multiplication and inversion. Elliptic curve scalar multiplication is quite different from field multiplication. One way to achieve parallel and pipelined scalar multiplication is to de-

compose ECADD and ECDBL operations into FF operations, which result in a sequence of FF addition, subtraction, squaring, multiplication and inversion operations. Proper grouping of these field operations reveals new possibilities for optimization. This idea is used in [IT02, Mis06, DH04, CTL05] to achieve parallelism and/or pipelining in the scalar multiplication operation. In [CTL05], a number of multipliers are used in parallel. In [IT02], finite field operations are optimized for single instruction multiple data (SIMD) architecture. In [DH04], such a grouping has been used for obtaining pipelining and systolic operations. In [Mis06], the sequence of operations is divided into a collection of uniform (similar) atomic blocks, where each block consists of a number of finite field operations. This leads to a pipelined algorithm, in which two blocks of operations run in parallel and consequently require a double sized hardware. The idea of atomic blocks has been used in [CCJ04] as a low cost solution to achieve immunity against simple power analysis attacks (SPA) on the scalar multiplication.

Elliptic curve processors may be seen as arithmetic processors, where instead of integer/floating point arithmetic, finite field arithmetic is performed. Therefore, all traditional architectural optimizations such as runtime parallelism detection, jump or branch prediction and pipelining are applicable. Unlike general purpose processors, elliptic curve processors (ECP) do not execute a random sequence of computations. Rather, they execute a specific sequence of FF computations, known as scalar multiplication, and once the sequence is selected and optimized for execution, there is no need for change. Runtime detection of parallelism has been employed in [GSE02, SBP06]. In general, this approach imposes complexity overhead. However, it is effective when the execution of multiple algorithms is required. Since scalar multiplication algorithms can be optimized off-line, VLIW architectures could be an alternative to the superscaling approach employed in [SBP06].

Grouping of finite field operations using uniform blocks with comparable computational complexity is a key factor in the implementation of parallel and/or pipelined algorithms. Among the finite field operations, the execution time of a squaring operation varies considerably depending on the type of fields– prime or extension. For field $GF(p)$, where p is prime,

the complexity of squaring is comparable to the complexity of multiplication. This approach has been used in [Mis06] to create the uniform grouping and atomic blocks. However, in binary extension field $GF(2^m)$, when the irreducible polynomial defining the field is known in advance, the complexity of squaring is significantly lower than that of multiplication and, under certain situations, becomes comparable to that of addition [Wu02, LH04, GSE02]. In practice, FF squaring, like FF addition, can be performed in one clock cycle. For example in the prime field a multiplication or a squaring operation may be grouped with an addition, while in the extension field a multiplication should be grouped with an addition or squaring to construct computation blocks with the same complexity. In fact, the pipelining scheme introduced in [Mis06] would not be possible/efficient over $GF(2^m)$ with known irreducible polynomial. Therefore, depending on the complexity of FF squaring, a different approach to the grouping is needed for elliptic curve computations.

Finite field multiplication is the bottleneck of scalar multiplication, especially using projective coordinates. Most high speed ECP's in $GF(2^m)$ use a word serial (WS) finite field multiplier, either in direct form [OP00, GSE02] or in the Karatsuba form [GBG03, EJM02, GSE02]. Assuming the field size of 2^m elements and the word size of w bits, a typical WS multiplication algorithm is normally performed in $\lceil m/w \rceil$ iterations. It is also common in the literature to ignore the execution time of FF addition (and sometimes FF squaring) compared to the execution time of FF multiplication. Simple analyses show that scalar multiplication is achievable in $N(m-1)\lceil m/w \rceil$ clock cycles, where N is the number of FF multiplications in one iteration of the scalar multiplication loop. However, this level of performance has not been reported in the literature yet and the main four reasons appear to be the followings:

1. In hardware implementation of WS finite field multipliers, a few extra clock cycles are spent on loading inputs and unloading outputs [OP00, GSE02, LH04]. In practice, this leads to a total of $\lceil m/w \rceil + c$ clock cycles for one multiplication using a word serial multiplier. Typically, the value of c is 3 for elliptic curves that are of practical interest [OP00, GSE02, LH04]. For other finite field arithmetic units, like adder and

Table 2.1: Typical Number of Clock Cycles of Basic Finite Field Operations

Design	m	Multiplication	Addition	Squaring
[OP00]	167	7	3	3
[GSE02]	163	7 to 12	3	3
[GBG03]	233	9	≥ 2	2 (est.)
[LH04]	163	7	3	2

squarer, extra clock cycles are spent to transfer data to and from memory/register file as well. Table 2.1 compares the execution times of these operations in terms of clock cycles as reported in various articles.

2. For high speed hardware implementation of operations of $GF(2^m)$ the computation times of addition and squaring in terms of number of clock cycles are comparable to that of multiplication (see Table 2.1) and may not be ignored.

3. In a typical processor architecture the computation units are connected to the memory/register file or to each other by a common bus. If two units require data at the same time, one of these wait or has to stay idle until the other unit releases the bus. This could lead to a large number of idle cycles for the computation units [LH04].

4. The FF multiplier, which occupies the bulk of hardware in a high performance design, is not used efficiently. In some cases, the inputs of one FF multiplication depend on the output of the previous FF operation. Therefore the FF multiplier, if implemented in pipelined form, stays idle while waiting for the next input. This is specifically true in two consecutive iterations of the scalar multiplication loop.

This work proposes an architecture/scheme for elliptic curve scalar multiplication over binary extension field $GF(2^m)$ that alleviates the above mentioned problems. In this scheme the output of one field multiplication operation is not used as an input to the next multiplication operation; rather the underlying finite field operations of the scalar multiplication are

divided into two streams (addition/squaring and multiplication) that are executed in parallel, and simultaneous loading of operands to the multiplier and adder/squarer is permitted. The proposed scalar multiplication scheme achieves better performance by preventing the finite field multiplier to become idle anytime during the entire $m-1$ iterations of the scalar multiplication loop.

We demonstrate the effectiveness of the scalar multiplication scheme by applying it to a classical processor architecture, which uses a pseudo-pipeline WS finite field multiplier. This multiplier computes one multiplication every $\lceil m/w \rceil$ clock cycles instead of $\lceil m/w \rceil + c$, where $\lceil m/w \rceil = 4$ and $c = 3$, as reported in the literature for practical applications [GSE02, LH04]. A decrease in the number of clock cycles from 7 to 4 is extremely useful and comes without any significant cost, since the hardware added for pipelining is negligible compared to the rest of the multiplier. This multiplier enables us access relevant variables in parallel with finite field computations.

The organization of the remainder of this article is as follows. Section 2.2 briefly reviews the Montgomery scalar multiplication algorithm, which has been used in a number of hardware implementations [GSE02, HLR03, LMW00, OP00, BDG02, CTL05]. There are data dependencies in the steps of the algorithm and hence the latter cannot be readily executed in pipelined fashion as desired. In Section 2.3, we develop a pipelined version of the scalar multiplication scheme. In Section 2.4 we adapt an architecture of a finite field multiplier suitable for the proposed scalar multiplication scheme. In this section, implementation issues are also considered and some results are presented. Finally, concluding remarks are given in Section 2.5.

2.2 Review of the Montgomery Scalar Multiplication

Points on an elliptic curve E, defined over a finite field $GF(2^m)$, along with a special point called infinity, and a group operation known as point addition, form a commutative finite

group. If P is a point on the curve E, and k is a positive integer computing

$$kP = \underbrace{P + P + P + \cdots + P}_{k \quad times}$$

is called scalar multiplication. The result of scalar multiplication is another point Q on the curve E. It is normally expressed as $Q = kP$. If E is an elliptic curve defined over $GF(2^m)$, the number of points in $E(GF(2^m))$ is called the order of E over $GF(2^m)$, denoted by $\#E(GF(2^m))$. For cryptographic applications $\#E(GF(2^m)) = rh$ where r is a large prime and h is a small integer and P and Q have order r. Scalars such as k are random integers where $1 \leq k \leq r-1$. Since $r \approx 2^m$, the binary representation of $k = \sum_{i=0}^{n-1} k_i 2^i$ has n bits where $k_i \in \{0,\ 1\}$ and $n \approx m$. Scalar multiplication is the most dominant computation part of elliptic curve cryptography. More on this can be found in [HMV03, BSS02].

Points on elliptic curves can be represented using affine or projective coordinates. Each point operation, namely point doubling or addition, requires $I + 2M + S$ cycles in affine coordinates [BSS02]. Extended Euclidean [GSE02] and Itoh-Tsujii [IT88] algorithms for finite field inversion over $GF(2^m)$ requires $2m$ and $(m-1)S + (\lfloor \log_2(m-1) \rfloor + h(m-1) - 1)M$ clock cycles respectively, where $h(j)$ is the number of non-zero bits in the binary representation of the integer j. Using projective coordinates, only one inversion at the end of scalar multiplication is required at the expense of extra multiplications in the point operation formulae. If the complexity of inversion over the underlying field is significantly higher than that of multiplication, then it may be advantageous to represent points using projective coordinates. This is the case for the processors in Table 2.1. The affine coordinate representation could be suitable for architectures which employ serial multipliers. For two interesting ECP implementations using affine coordinates the reader is referred to [DH04, HLR03].

Algorithm 1 shows the projective version of Montgomery scalar multiplication scheme for non-supersingular elliptic curves over binary fields as it was introduced in [LD99]. In this algorithm `Madd(`X_1`,`Z_1`,`X_2`,`X_2`)`, `Mdouble(`X_1`,`Z_1`)` and `Mxy(`X_1`,`Z_1`,`X_2`,`X_2`)` are functions for point addition, point doubling and conversion of projective coordinates to affine coor-

dinates. The computations involved in these functions can be found in the appendix. The reader is referred to [LD99, HMV03] for detailed explanation.

Algorithm 1 Montgomery scalar multiplication in projective coordinates

Input: A point $P = (x, y) \in E$, an integer $k > 0$, $k = 2^{n-1} + \sum_{i=0}^{n-2} k_i 2^i$, $k_i \in \{0, 1\}$

Output: $Q = kP = (x_k, y_k)$

1: $X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$ {compute P and $2P$}
2: **if** ($k = 0$ **or** $x = 0$) **then**
3: $x \leftarrow 0, y \leftarrow 0$ {This part is provided to have a complete presentation of algorithm. In a practice, the algorithm inputs must be checked for validity before entering the scalar multiplication unit.}
4: **stop**
5: **end if**
6: **for** $i = n - 2$ **to** 0 **do**
7: **if** $k_i = 1$ **then**
8: $(X_1, Z_1) \leftarrow \texttt{Madd}(X_1, Z_1, X_2, Z_2)$,
$(X_2, Z_2) \leftarrow \texttt{Mdouble}(X_2, Z_2)$
9: **else**
10: $(X_2, Z_2) \leftarrow \texttt{Madd}(X_2, Z_2, X_1, Z_1)$,
$(X_1, Z_1) \leftarrow \texttt{Mdouble}(X_1, Z_1)$
11: **end if**
12: **end for**
13: $Q \leftarrow \texttt{Mxy}(X_1, Z_1, X_2, Z_2)$
14: **return** Q

This algorithm has been used in several high speed ECC implementations [OP00, GSE02, ST03]. For a straight-forward implementation in hardware, it may take as many as $(m - 1)(6M + 3A + 5S) + (10M + 7A + 4S + I)$ clock cycles, where M, A, S and I are the number of clock cycles required for multiplication, addition, squaring and inversion respectively, in

the underlying finite field and m is the dimension of the binary extension field $GF(2^m)$.

2.3 Architecture for Scalar Multiplication

Since finite field multiplier is the bottle neck of scalar multiplication, it requires special consideration for realizing high performance architecture for scalar multiplication. One of our goals is to utilize the multiplier in such a way so that it effectively becomes the sole component that determines the time duration of each pass of the loop in the scalar multiplication algorithm. Our other goal is to keep the multiplier core working during the entire time of the loop of the algorithm including the transition from one iteration to the next iteration.

2.3.1 Merging of Two Execution Paths

In order to keep the algorithm uniform and suitable for hardware implementation we can merge the two k_i dependent execution paths of Algorithm 1. It is sufficient to swap X_1, with X_2 and Z_1 with Z_2 before computation and swap them back afterwards, if k_i equals to one, as it is shown in Algorithm 2.[1]

In hardware, when an indexing mechanism is utilized to access variables X_1, X_2, Z_1 and Z_2, swapping can be easily performed by exchanging the address lines to these registers or by an equivalent mechanism. Such swapping can be done on the fly using combinational logic only. A swap signal can be generated using the state of k_{i-1} and k_i. It can then be applied to the address logic of the register file.

We assume fixed irreducible polynomial which implies that the complexity of squaring is equal to that of addition [Wu02]. It is also assumed that the multiplication takes longer than addition and squaring which makes the critical path of the scalar multiplication operation dependent only on the finite field multiplication. The case of generic polynomials, i.e.

[1] The same result may be obtained by expressing the computation part in the following way: $(X_{\overline{k_i}+1}, Z_{\overline{k_i}+1}) \leftarrow \texttt{Madd}(X_1, Z_1, X_2, Z_2)$, $(X_1, Z_1) \leftarrow \texttt{Mdouble}(X_{k_i+1}, Z_{k_i+1})$, where $\overline{k_i}$ is the logical inversion of k_i.

Algorithm 2 Scalar multiplication algorithm with uniform addressing

Input: A point $P = (x, y) \in E$, an integer $k > 0$, $k = 2^{n-1} + \sum_{i=0}^{n-2} k_i 2^i$, $k_i \in \{0, 1\}$

Output: $Q = kP = (x_k, y_k)$

1: $X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$
2: **if** $(k = 0$ **or** $x = 0)$ **then**
3: $\quad Q \leftarrow \mathcal{O}$
4: $\quad$ **stop**
5: **end if**
6: **if** $k_{n-2} = 1$ **then**
7: $\quad$ `Swap`(X_1, X_2), `Swap`(Z_1, Z_2)
8: **end if**
9: **for** $i = n - 2$ **to** 0 **do**
10: $\quad (X_2, Z_2) \leftarrow$ `Madd`(X_1, Z_1, X_2, Z_2),
$\quad\quad (X_1, Z_1) \leftarrow$ `Mdouble`(X_1, Z_1)
11: $\quad$ **if** $(i \neq 0$ **and** $k_i \neq k_{i-1})$ **or** $(i = 0$ **and** $k_i = 1)$ **then**
12: $\quad\quad$ `Swap`(X_1, X_2), `Swap`(Z_1, Z_2)
13: $\quad$ **end if**
14: **end for**
15: $Q \leftarrow$`Mxy`(X_1, Z_1, X_2, Z_2)
16: **return** Q

unnamed curves or scalable processors and multiple fixed polynomials are discussed at the end of this section and the next section respectively.

2.3.2 Parallel Execution

If the finite field operations required for each `Madd(.)`and `Mdouble(.)` as defined in the appendix are performed in sequence, then each pass of the main loop of Algorithm 2 will require about $6M + 3A + 5S$ clock cycles.

Fig. 2.1 depicts the flowgraph of one pass of the loop of the scalar multiplication algorithm in which each multiplication is performed in parallel with an addition and/or a squaring. It is assumed that multiplication takes longer than addition and squaring, which makes the critical path of the scalar multiplication operation dependent only on the finite field multiplication. Using this technique, the execution time for one pass in the scalar multiplication loop is equal to $6M + A$.

Fig. 2.1 is based on the assumption that only one multiplier is available. Should two multipliers of the same specification be available, then two multiplications may be grouped along with addition and/or squaring and reduce the execution time to $3M + A$ clock cycles. It is also possible to utilize more multipliers in parallel [CTL05].

2.3.3 Data Dependency at Transitions of Iterations

Let $M = M_p + c$ be the number of clock cycles needed for a finite field multiplication, where $M_p (= \lceil m/w \rceil$ for a WS multiplier) is the number of clock cycles needed to compute the product and c is the total clock cycles needed to load the input and unload the product from the multiplier.

If not properly designed, during loading and unloading of operands that require c clock cycles, the multiplier core becomes idle. In the flowgraph shown in Fig. 2.1 performance can be improved if the idle period can be reduced. In order to prevent the multiplier core to become idle a new set of operands need to be fed to the multiplier at every M_p clock cycles.

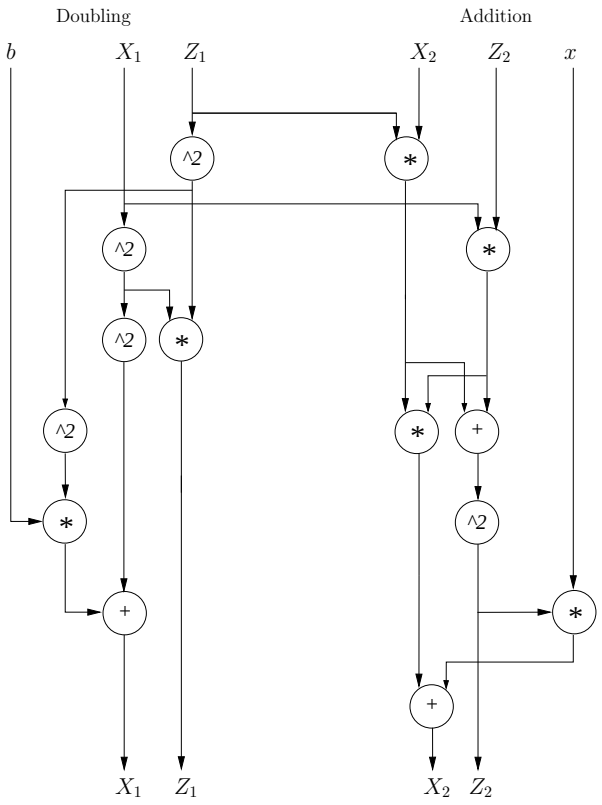


Figure 2.1: Parallel execution of multiplication and addition/squaring in one iteration

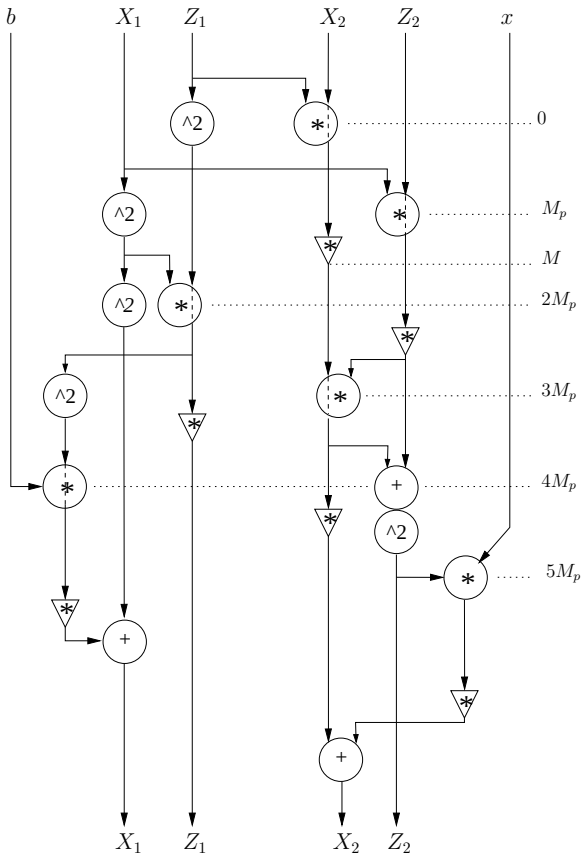


Figure 2.2: Parallel with no idle cycle in the middle of the iteration

At the same time, one needs to make certain that the next multiplication is not dependent on the output of current one because the results will be ready only after $M_p + c$ clock cycles.

The flowgraph of scalar multiplication in Fig. 2.2 assumes a multiplier with a computation time of M_p clock cycles and a total multiplication time of $M = M_p + c$ clock cycle. Each circle in the flowgraph corresponds to the start of a finite field multiplication. Vertically below each circle, there is a triangle to indicate the end of the multiplication that originated at the circle. The distance between a circle and the corresponding triangle is $M = M_p + c$ clock cycles. The minimum time difference between two consecutive circles (or two consecutive triangles) is the operation rate M_p of the multiplier. The result of the multiplication cannot be used before the triangle in the flowgraph, which is $M = M_p + c$ clock cycles after the corresponding circle. We assume that $M_p > A$ and $M_p > S$ to allow addition and squaring

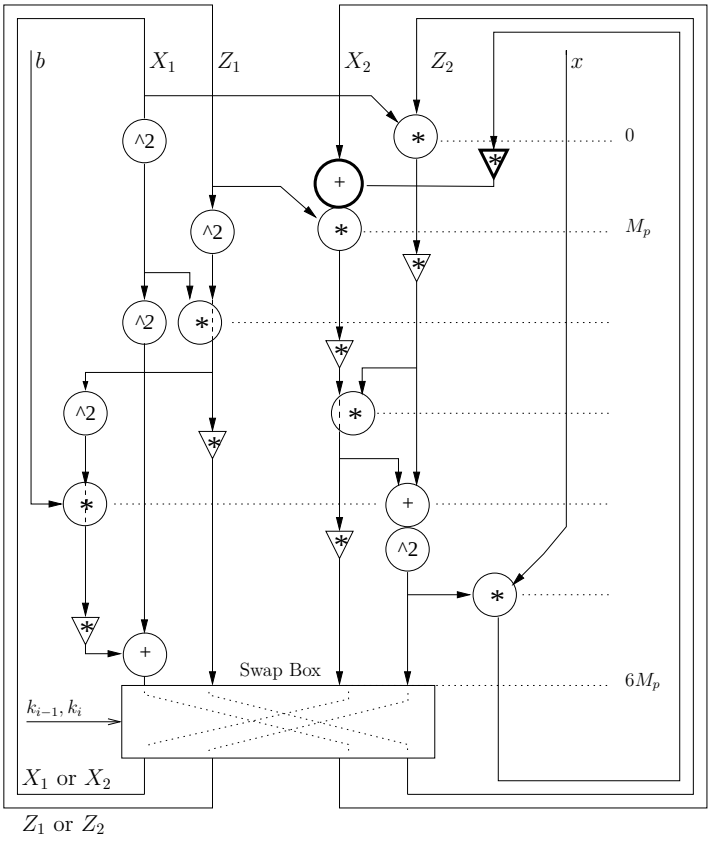


Figure 2.3: Parallel with no idle cycle in entire scalar multiplication loop

to be performed in parallel with multiplication. In the flowgraph the multiplier receives its operands regularly and at equal intervals, which is M_p clock cycles. Using this scheme the total execution time is reduced to $5M_p + M + A$ clock cycles compared to $6M + A$ clock cycles in the flowgraph of Fig 2.1.

Near the bottom of the flowgraph of Fig. 2.2, the adder needs to wait until the multiplier computes the field multiplication and the output of the adder would be the next input to the multiplier, which starts on the next iteration. This causes a delay of $(M - M_p + A)$ clock cycles per iteration, which in turn translates into an overall delay of $(m - 1)(M - M_p + A)$ clock cycles in the scalar multiplication operation. This delay can be eliminated as follows.

2.3.4 Resolving Data Dependency at Iteration Transitions

Based on `Madd(.)` in the appendix, the first multiplication in an iteration of the loop can be either $X_1 * Z_2$ or $X_2 * Z_1$. We observe that Z_1, Z_2 and X_1 are ready before X_2 becomes available in the last finite field multiplication in the flowgraph of Fig. 2.2. This suggests that the next iteration can be started by $X_1 * Z_2$ operation and before X_2 becomes available.

If $k_i = 0$, we may start the next iteration by the $X_1 * Z_2$ operation. However, if $k_i = 1$, the variables are swapped as in Algorithm 2; X_2 in the current cycle goes to X_1 in the next cycle. Therefore, we should start the next cycle with the $X_2 * Z_1$ operation which is actually a $X_1 * Z_2$ operation. In the new arrangement the first multiplication in the loop will depend on k_i. The complete loop is shown in Fig. 2.3.

A switch at the end (or start) of the flowgraph swaps registers properly. Swapping does not take extra clock cycles since the logic is rather simple and can be realized by combinational logic (Algorithm 2). The next iteration starts before the end of the last multiplication. One addition operation, from the end of the previous iteration and termination of one multiplication appears at the start of the next iteration. This is highlighted in Fig. 2.3 with a bold faced circle and triangle. As it is shown, each iteration takes $6M_p$ clock cycles and the multiplier does not become idle.

The scheme can be implemented by a multiplier that has a computational time long enough to allow an addition or squaring to be performed in parallel with the multiplication. A finite field multiplier suitable for this scheme is proposed in section 2.4.2.

Table 2.2 summarizes and compares the speed-ups of scalar multiplication operation as mapped on to the flowgraphs of Figs. 2.1, 2.2 and 2.3. For finite field operations indicated in the flowgraph, high speed architectures similar to [OP00,GSE02,GBG03,LH04] are assumed. In these architectures one typically has $A = S = 3$ clock cycles, $M_p = \lceil m/w \rceil$ clock cycles and $M = M_p + 3 = 7$ clock cycles. The first row in Table 2.2 serves as a basis of comparison and corresponds to a straight-forward hardware implementation of Algorithm 1. As an example of speed-ups, the bottom row of the table indicates that a scalar multiplication using Fig. 2.3 is almost 2.75 times faster than that using a straight-forward implementation of Algorithm 1.

As it is shown in Fig. 2.3 each multiplication is paired with one squaring. If unnamed curved are used another multiplier may be employed in parallel with original multiplier to perform the squaring operation. In the flowgraph some multiplications take their input from a squarer. The number of cycles in the flowgraph will be affected, unless squaring is performed in M_p or less cycles. The critical path of the hardware is not affected as all multiplications and squarings are performed in parallel.

2.4 Implementation

The number of clock cycles by itself is not an accurate measure of performance of the system, since the clock rates may vary considerably. Therefore, an implementation is carried out to verify the performance of the system.

2.4.1 Implemented Architecture

Traditional elliptic curve processors are based on an instruction set which allows them to execute different scalar multiplication schemes [OP00, GSE02, LH04]. The proposed scalar

Table 2.2: Performance with Various Enhancement Methods Assuming That $\lceil m/w \rceil = 4$, $A = S = 3$ Clock Cycles

Method	#Clks in one iteration	#Clks in $(m-1)$ iterations	Relative speed-ups
Straight-forward (Alg. 1)	$6M + 3A + 5S$	$66(m-1)$	1.00
Parallel addition/squaring (Fig. 2.1)	$6M + A$	$45(m-1)$	1.47
No idle cycle for the FF multiplier (Fig. 2.2)	$5M_p + M + A$	$30(m-1)$	2.13
No idle cycle in the entire operation (Fig. 2.3)	$6M_p$	$24(m-1)$	2.75

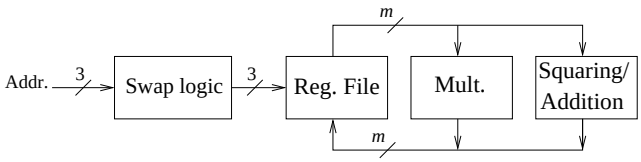


Figure 2.4: Implemented Architecture

multiplication scheme is highly optimized toward the execution of the Montgomery ladder in projective coordinates [Mon87, LD99, CCD05]. Therefore, it is implemented in the form of a programmable state machine. Fig. 2.4 shows the basic architecture of the execution unit, which consists of a squaring/addition unit, a finite field multiplier, a dual port $8 \times m$ bit register file, control unit (not shown), and an address swapping logic. An FF addition or squaring, an FF multiplication and a load/save operation from/to the register file can be performed in parallel. A squaring, for example, is performed in 3 clock cycles – one for each of the following operands: loading the accumulator with data from the register file, squaring and finally saving the result in the register file. The squarer can perform multiple squaring without storing the data in the register file. This property is used in the Itoh-Tsujii inversion algorithm at the end of the scalar multiplication where a^{2^i} operation is required. The multiplier and the squaring/addition unit can be loaded with the same data at the same clock cycle, to prevent redundant data transfer on the data bus. The register file is used to store the result at the end of each iteration and finally the scalar k is stored in a specific registered (not shown) and is provided to the control logic bit by bit at each iteration.

2.4.2 Pseudo-Pipelined WS Finite Field Multiplier with Short Critical Path

A word serial multiplication algorithm is applicable to any algebraic ring and essentially offers time-space trade-off. It has been employed in [GSE02, GBG03, LH04, OP00] over $GF(2^m)$. If the word size of w bits is used, the space complexity of the multiplier is $O(wm)$, the critical path is $T_{AND} + \lceil \log_2 w \rceil + T_{mod}$ and multiplication will take $\lceil m/w \rceil + 3$ clock cycles, where T_{mod} is modular reduction time, and loading operands and unloading the result take two and one clock cycles respectively.

In this section a pseudo-pipelined WS finite field multiplier is introduced which uses the pipeline to reduce the critical path and can be used in the proposed scalar multiplication scheme. A polynomial basis representation of the field and a fixed irreducible polynomial is assumed. For multiplying $A(x) \times B(x) \pmod{f(x)}$, input $A(x)$ is divided into $\lceil m/w \rceil$ words, i.e. , $A(x) = A_{\lceil m/w \rceil - 1} x^{(\lceil m/w \rceil - 1)w} + \ldots + A_2 x^{2w} + A_1 x^w + A_0$, where A_i's are each a polynomial of degree $\leq w - 1$ (degree of $A_{\lceil m/w \rceil - 1}$ might be less than the degree of other A_i's). Assume that a multiplication starts in clock cycle i. Then input $B(x)$ is loaded in register S in cycle i (see Fig. 2.5). In cycle $i+1$, $B(x)$ is copied onto register T, $A(x)$ is loaded into S and $A_{\lceil m/w \rceil - 1}$ of $A(x)$ is loaded into register t. Similar to the way $A(x)$ is divided, the content of t is divided into k words ${A_j}^{(k-1)}, {A_j}^{(k-2)}, \ldots {A_j}^{(0)}$ where $j = \lceil m/w \rceil - 1$ in cycle $i + 1$.

In cycle $i + 2$, the k bit parallel polynomial multipliers are used to compute $m \times w/k$-bit multiplications $A^{k-1}_{\lceil m/w \rceil - 1} B(x), \ldots, A^1_{\lceil m/w \rceil - 1} B(x), A^0_{\lceil m/w \rceil - 1} B(x)$, as shown in Fig 2.5. Each of these $m + w/k - 1$ bit product polynomials is buffered in $D_{k-1}, \ldots, D_1, D_0$ registers for one cycle and then forwarded to the next stage, where they are properly accumulated to create $A_{\lceil m/w \rceil - 1} B(x)$. The output of $D_{k-1}, \ldots, D_1, D_0$ and P are shifted properly, added together, reduced mod $f(x)$ and stored in P in the next cycle. The overall multiplication has a computation time of $3 + \lceil m/w \rceil$ clock cycles and a new pair of inputs can be fed at every $\lceil m/w \rceil$ cycles.

2.4.2.1 The Critical path

The critical path of the multiplier is $max(\text{path}_1, \text{path}_2)$, where path_1 is the delay of the combinational logic between T and D_i registers, and likewise path_2 is the delay between D_i and P registers.

Between T and D_i, a w/k-bit by m-bit polynomial multiplication over $GF(2)$ is performed, therefore $\text{path}_1 = T_{AND} + \lceil \log_2 w/k \rceil T_{XOR}$. In path_2, first $k + 1$ $(m + w - 1)$-bit polynomials are added together and then the result is reduced mod $f(x)$. Therefore,

Table 2.3: k versus Critical Paths

k	path_1	path_2
1	$T_{AND} + 6T_{XOR}$	$3T_{XOR}$
2	$T_{AND} + 5T_{XOR}$	$3T_{XOR}$
3	$T_{AND} + 4T_{XOR}$	$4T_{XOR}$

$\text{path}_2 = \lceil \log_2(k+1) \rceil T_{XOR} + T_{mod}$, where T_{mod} is the delay of modular reduction.

It has been shown that the reduction operation modulo a r-term irreducible polynomial $f(x)$ requires $(r-1)(m-1)$ bit operations, when two m-term polynomials are multiplied. The critical path of the reduction depends on the non-zero terms of $f(x)$. For trinomials $f(x) = x^m + x^k + 1$ where $1 < k < m/2$ the critical path is $T_{mod} = \lceil \log_2 r \rceil = 2T_{XOR}$ and $2m - 2$ gates are required [Wu02].

Delay calculation for pentanomial irreducible polynomials in general case is complicated. In our cases a w-term polynomial is multiplied by an m-term polynomial and NIST irreducible polynomials are used. For pentanomial $f(x) = x^m + x^{m_3} + x^{m_2} + x^{m_1} + 1$ where $m - m_3 > w$, the reduction requires the maximum of $4(w-1)$ additions over $GF(2)$ and the critical path is $T_{mod} = \lceil \log_2 r \rceil = 3T_{XOR}$. Similar results apply for trinomials.

The variable k is used to equalize path_1 and path_2 in order to reduce the overall critical path in the expense of increased number of D_i registers. It may be viewed as a parameter which transfers parts of computation from one stage of the pipeline to the next stage. For $k = 1$ the critical path is the same as the critical path of the multiplier used in [GSE02]. Consider implementation over $GF(2^{163})$ of NIST ECC standard where $f(x) = X^{163}+x^7+x^6+x^3+1$ and $\lceil m/w \rceil = 4$. Critical paths for different values of k are given in Table 2.3. Since the addition and reduction operation are mixed together by the synthesizer, $\text{path}_2 = \lceil \log_2(r+k+1) \rceil$. Note that either path_1 or path_2 can be set as the critical path. For $k = 3$ the total scalar multiplication time is reduced by almost $2(m-1)T_{XOR}$, compared to the conventional WS multiplier used in [GSE02]. In deep sub-micron CMOS technology interconnection and the layout congestion is a major cause of delay which is not taken into account here. Therefore,

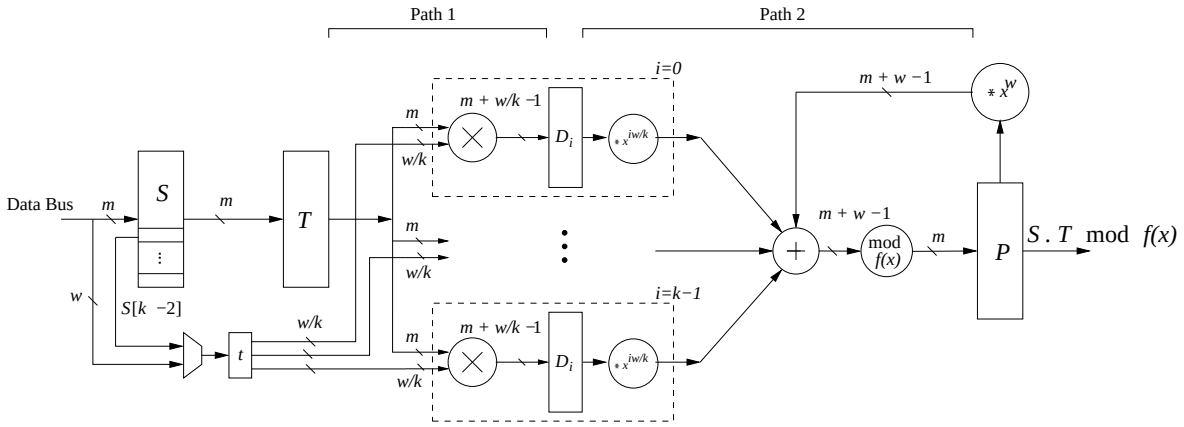


Figure 2.5: Pseudo-pipelined finite field multiplier

finding optimum k may require a trial and error approach.

2.4.2.2 Operation

The operation of the multiplier for the special case of $\lceil m/w \rceil = 4$ and $k = 2$ is presented in Table 2.4. It shows register contents for two consecutive multiplications, namely $A \times B$ and $U \times V$ assuming a "cold" start. In the table, operands A and U are split as $A = A_3x^{3w} + A_2x^{2w} + A_1x^w + A_0$ and similarly $U = U_3x^{3w} + U_2x^{2w} + U_1x^w + U_0$. One can see from the table that each multiplication takes 7 cycles. The multiplier has a pipeline rate of 4, i.e. after every 4 clock cycles a new set of input operands can start entering the multiplier.

The key to the fast execution of scalar multiplication is to perform loading of the operands, and unloading the results from the computation units in parallel with the finite field computations, namely addition, multiplication and squaring. As an example, consider the execution of $Z_1 = Z_1 * X_1$ in parallel with squaring $X_1 = X_1^2$ and $Z_2 = Z_1 + X_2$, and the start of another multiplication $X_2 \times Z_2$.

1: $S \leftarrow X_1, ACC \leftarrow X_1$ {*load register S and the accumulator simultaneously*}

2: $T \leftarrow S, S \leftarrow Z_1, ACC \leftarrow ACC^2$ {*The squaring is performed in one cycle.*}

3: $ACC \leftarrow Z_1, X_1 \leftarrow ACC$ {*Load ACC with the next operand (Z_1 is still intact). Save the result in ACC (changing X_1 will not affect the multiplication). Multiplier is still busy*}

Table 2.4: State Diagram of the Pseudo-pipelined Finite Field Multiplier

Cycle	S	T	t	D_0	D_1	P
1	B					
2	A	B	A_3			
3	A	B	A_2	$B \times {A_3}^{(0)}$	$B \times {A_3}^{(1)}$	
4	A	B	A_1	$B \times {A_2}^{(0)}$	$B \times {A_2}^{(1)}$	$B \times A_3 \pmod{f(x)}$
5	V	B	A_0	$B \times {A_1}^{(0)}$	$B \times {A_1}^{(1)}$	$(B \times A_3)x^w + B \times A_2 \pmod{f(x)}$
6	U	V	U_3	$B \times {A_0}^{(0)}$	$B \times {A_0}^{(1)}$	$((B \times A_3)x^w + B \times A_2)x^w + B \times A_1 \pmod{f(x)}$
7	U	V	U_2	$V \times {U_3}^{(0)}$	$V \times {U_3}^{(1)}$	$(((B \times A_3)x^w + B \times A_2)x^w + B \times A_1)x^w + B \times A_0 \pmod{f(x)}$ End of $A \times B \pmod{f(x)}$
8	U	V	U_1	$V \times {U_2}^{(0)}$	$V \times {U_2}^{(1)}$	$V \times U_3 \pmod{f(x)}$
9		V	U_0	$V \times {U_1}^{(0)}$	$V \times {U_1}^{(1)}$	$(V \times U_3)x^w + V \times U_2 \pmod{f(x)}$
10				$V \times {U_0}^{(0)}$	$V \times {U_0}^{(1)}$	$((V \times U_3)x^w + V \times U_2)x^w + V \times U_1 \pmod{f(x)}$
11						$(((V \times U_3)x^w + V \times U_2)x^w + V \times U_1)x^w + V \times U_0 \pmod{f(x)}$ End of $U \times V \pmod{f(x)}$

4: $ACC \leftarrow ACC + X_2$

5: $S \leftarrow X_2, Z_2 \leftarrow ACC$ {*Another multiplication can start here, however the result of* $Z_1 * X_1$ *is not ready yet*}

6: $S \leftarrow Z_2$

7: $P \leftarrow Z_1 * X_1$ {*The result of multiplication is ready.*}

2.4.2.3 Multiple Fixed Irreducible Polynomial

Expansion to multiple fixed irreducible polynomials is straight-forward. The modular reduction operation in Fig. 2.5 needs to be replaced with multiple mod modules and be selected using a multiplexer appropriately and the length of registers need to be expanded to fit the elements of the field with the largest dimension. The equation for critical path remains the same; it increases in logarithm scale as the underlying fields dimension increases. This arrangement is used in [GSE02].

2.4.3 Timing

The timing diagram for the flowgraph of Fig. 2.3 is represented in Fig. 2.6. It shows the application of the pseudo-pipeline WS multiplier in parallel with a squarer and an adder. The smallest $\lceil m/w \rceil$ which allows parallel execution with addition and squaring and fits in our FPGA turns out to be 4. Consequently, one iteration takes 24 clock cycles. The last multiplication and addition of the i^{th} iteration continues to the $i+1^{th}$ iteration and hence the execution of scalar multiplication is performed without an idle cycle in the multiplication.

2.4.4 Implementation Results

Results of the FPGA synthesis, place and route (P&R) for Xilinx XC4VLX200 using various fields are shown in Table 2.5. The number of flip flops in the table partially include the flip flops used to implement the $8 \times m$-bit register file. The synthesizer utilizes unused flip flops in some slices to implement the register file as well. Note that the finite field multiplier is a

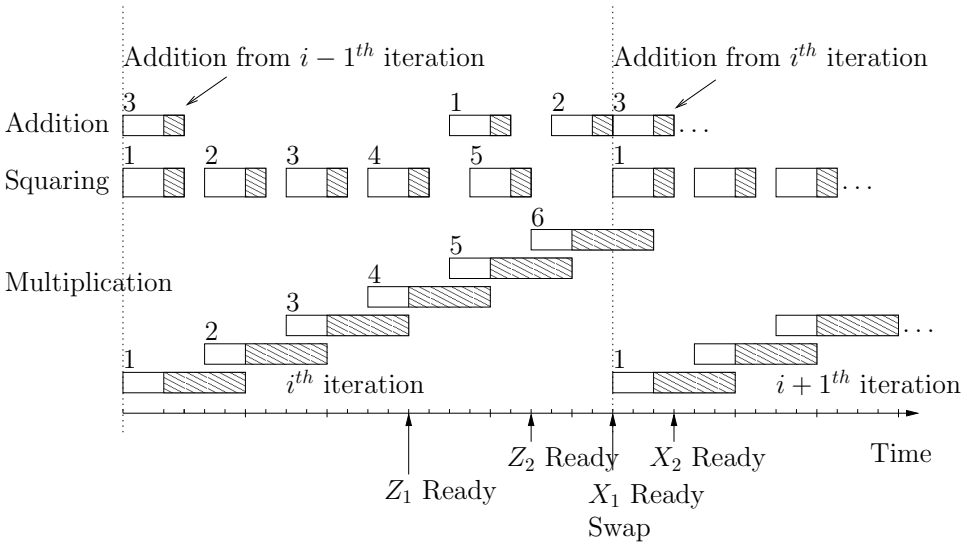


Figure 2.6: Timing of scalar multiplication with no idle cycle in entire loop

large combinational circuit and many flip flops in the allocated slices remain unused.

Table 2.7 shows the synthesis results for $\lceil m/w \rceil \in \{4, 8, 16, 32\}$ over $GF(2^{163})$. The proposed scheme is efficient when the running time of the multiplication is comparable to that of addition/squaring. If the multiplication operation is too slow, i.e. m/w is large, parallel operations and pseudo-pipelining might not be very effective. On the other hand if m/w is small, the space complexity of the multiplier becomes much higher than that of adders/squarers. Therefore, it might be beneficial to use an algorithm with multiple on-the-fly adders/squarers.

2.4.5 Comparison

The implementation results on XC2V2000 using Synplify Pro are reflected in Table 2.8, where $k = 1$ and for the bottom two rows the conventional quadratic multiplication and the Karatsuba algorithms have been used in the bit parallel multiplier of Fig. 2.5. For the Karatsuba version over $GF(2^{163})$, the 164×41 bit multiplication in Fig. 2.5 is first decomposed into 40×40 bit multiplication and then the Karatsuba algorithm is applied three times along with one 5-bit quadratic multiplication. More efficient schemes can be

Table 2.5: Synthesis, Place and Route Results for Virtex4:XC4VLX200 Using Two Different Synthesizers

m	Slices	Clk (MHz)	FF	LUT	Blk RAM
Synthesizer I					
79	1415	184	594	2589	4
113	2496	176	802	4675	5
163	4274	166	1142	8169	6
233	7778	136	1552	15043	8
283	10968	136	1862	21266	9
409	27541	124	2654	40930	13
571	51441	61	3810	76993	17
Synthesizer II (Synplify Pro)					
79	1472	224	701	2784	0
113	2391	201	1117	4430	0
163	4080	197	1502	7719	0
233	7130	190	2219	13396	0
283	10893	139	3603	19972	0
409	21956	137	4013	36237	0
571	34892	107	6445	66594	0

Table 2.6: FPGA Synthesis for Various Finite Fields $GF(2^m)$

Field m	15	79	113	163	233	283	409	571
Estimated Frequency	180	168	165	153	174	154	156	143
Flip Flops	321	734	1546	1308	3412	4657	6694	11430
Luts	861	2938	4450	7572	13570	19754	36859	68571
I/O Register bits	12	12	12	12	12	12	12	12
Block RAM	1	1	1	1	1	1	1	1
128x1 ROM	16	16	16	16	16	16	16	16

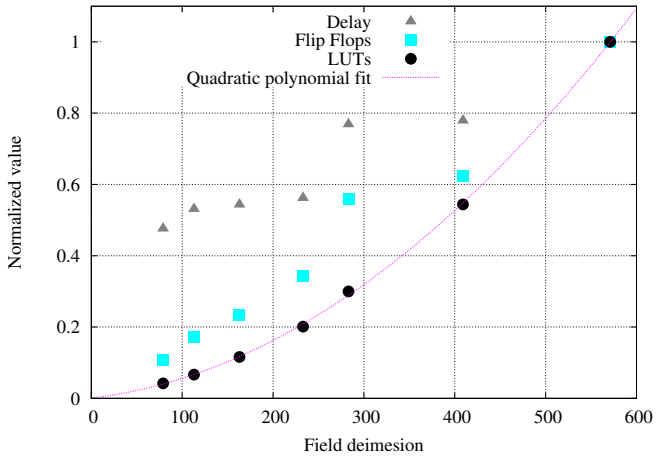


Figure 2.7: FPGA synthesis for various finite fields $GF(2^m)$

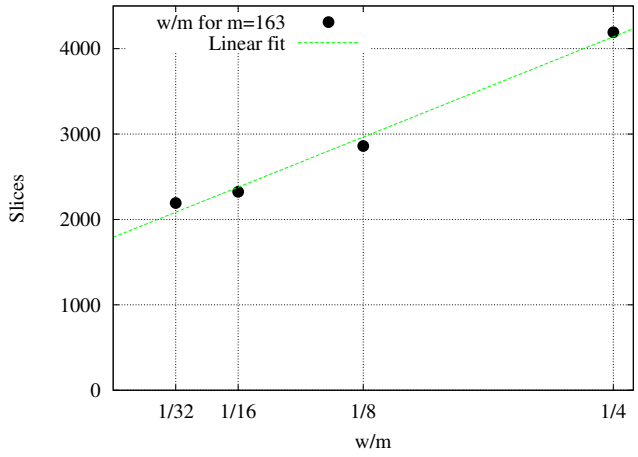


Figure 2.8: FPGA synthesis for various w/m for $GF(2^{163})$

Table 2.7: $\lceil \frac{m}{w} \rceil$ versus Slices over $GF(2^{163})$ for Virtex 2:XC2V2000

$\lceil \frac{m}{w} \rceil$	Slices
4	4192
8	2860
16	2323
32	2192

developed [Mon05, WP06]. Tables 2.8 and 2.5 indicate that using two different FPGAs of the same family can result in a large speed difference while difference in area is small.

In Table 2.8 a number of high speed elliptic curve processors (ECP) are compared with the proposed one. The hardware of [CTL05] uses normal bases. Four FF multipliers are utilized in parallel and addition and squarings are performed in series with multiplications. Each multiplication is performed in $5 + \lceil m/p \rceil$ clock cycles where p is the degree of parallelism and may be considered as equivalent to w. The maximum p, where the design has been implemented and can fit in an XILINX XC2V6000 FPGA, is 16. This design is using a composite field that is however avoided in some cryptographic systems for security reasons [HMV03]. On the other hand more efficient schemes are available for FF multiplication over composite fields. The scalar multiplier of [Mis06] is not included in the table above. This is because the scalar multiplier of [Mis06] uses a multiplier for squaring, which increases the total number of multiplications but it is necessary for the creation of atomic blocks. It also uses two multipliers, which means a larger hardware. Considering an implementation of [Mis06] for a 160−bit scalar (i.e. $m = 160$) and window based NAF with a window size of 4, an elliptic curve scalar multiplication over $GF(2^m)$ takes $1152(M+A+A+0) \approx 87(m-1)$ clock cycles (the cost of multiplication, addition and additive inversion is assumed to be six, three and zero clock cycles respectively).

In the scalar multiplication algorithm a conversion from projective to affine coordinates, which includes a finite field inversion, is required at the end of the loop (i.e. `Mxy(.)`). Using the Itoh-Tsujii algorithm it takes $(m-1) + M(\lfloor \log_2(m-1) \rfloor + h(m-1) - 1)$ clock cycles for an inversion. For parameters pertaining to ECC applications, the field multiplication related latency term, i.e., $M\lceil \log_2(m-1) \rceil + h(m-1) - 1)$ is smaller than $(m-1)$ and is almost negligible compared to the large number of clock cycles required for a complete scalar multiplication less the conversion. In order to give a simple expression, the field multiplication latency in the inversion is neglected in the total number of clock cycles for a scalar multiplication as shown in the fourth column of Table 2.8. If the extended Euclidean algorithm is used, an inversion takes $2m$ clock cycles [GSE02]. It can be concluded from the

table that the proposed scheme outperforms other systems in terms of the number of clock cycles as well as the critical path.

The last column in the table shows the algorithmic efficiency defined as throughput/area. It would be more accurate to use throughput/#slices, but slice counts were not reported by authors of other designs. Therefore, we have used throughput/#LUTs.

2.4.6 Comments

- Security Against Simple Power Analysis Attack (SPA): The Montgomery algorithm is considered to be inherently more resistant against SPA and timing attacks. This is because the computation cost does not depend on the specific bit of the scalar k. For each bit of the scalar, one point addition and one doubling are performed. The proposed scheme has two different execution paths depending on the current bit of the scalar k. Both execution paths have the same complexity and take the same number of clock cycles. If the attacker is not able to separate swapping operation in the whole process, it is expected that the new scheme have the same level of resistance against SPA attacks.

- Simple Swapping: In order to keep the swapping operation simple, parameters X_1, X_2, Z_1 and Z_2 are stored in a register file. Therefore this operation is performed by merely swapping the address information, this does not take any additional clock cycles. A simple scheme for swapping is shown in Fig. 2.9, which can be considered equivalent to the switching of maximum two gates. The swap signal is generated by the condition in Algorithm 2.

- Modular Construction: Fig. 2.3 can be used to derive a straight-forward architecture.

[1]In [OP00], the maximum value of w is 16. For our comparison, we have scaled it up to $w = 42$.
[2]The large size is due to its scalability.
[3]For [GBG03], we have assumed NAF representation for scalar k.
[4]See text.
[5]Using conventional quadratic complexity algorithm for 163×41 bit multiplication.
[6]Using the Karatsuba algorithm for 40×40 bit multiplication.

Table 2.8: Performance of the Scalar Multipliers

Design	m	w	#Clk for kP Cycle	Clk MHz	kP μS	Xilinx FPGA	HW resource LUT, FF	Efficiency $\frac{\text{Throughput}}{\#\text{LUTs}}$
[OP00][1] (2000)	167	42	$47(m-1)$ (est.)	76	210	XCV400E	3002, 1769 × 4	0.40
[GSE02][2] (2002)	163	41	$57(m-1)$ (est.)	66.5	143	XC2V2000	19508, 6442	0.36
[LH04] (2003)	163	41	$93(m-1)$ (est.)	66	233	XC2V2000	10017, 1930	0.43
[GBG03][3](2004)	233	-	$44(m-1)$ (est.)	100	132	XC2V6000	19440, 16970	0.39
[CTL05][4] (2005)	162	16	$35(m-1)$	54	110	XC2V6000	34000, 3400 (est.)	0.27
This work[5]	163	41	$25(m-1)$	100 (128 P&R)	41	XC2V2000	7559, 2059 (4192 slices)	3.23
This work[6]	163	41	$25(m-1)$	100 (122 P&R)	41	XC2V2000	6095, 2398 (3416 slices)	4.00

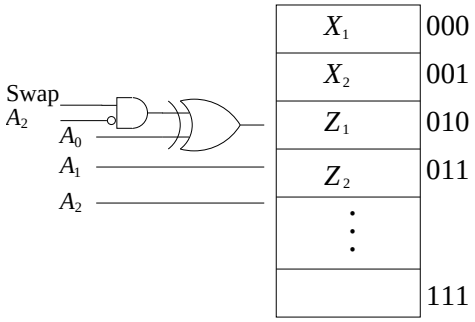


Figure 2.9: Swapping mechanism

The basic building block is composed of an adder, a squarer, a multiplier, and a set of registers which hold the output of these units and a data path control unit.The output of the registers goes to the data path control unit. It arranges the data for the next round of arithmetic operation. It takes M_p clock cycles for data to be processed in the basic building block. One can cascade m building blocks to construct a pipeline system which outputs the result of one scalar multiplication every M_p clock cycles with a latency of mM_p clock cycles.

- Designing for future FPGAs and multiple multipliers: A WS multiplier with $m/w = 1$ i.e., a fully bit-parallel multiplier, occupies almost the same area as four multipliers with $m/w = 4$ while it performs the multiplication in one clock cycle instead of four clock cycles. This suggests that one large multiplier may be used instead of multiple parallel multipliers. Using multiple multipliers, as used in [CTL05], has the disadvantage that we might not always be able to use all multipliers simultaneously or apply pseudo-pipelining technique, which could result in higher time-space complexity. If $m/w = 1$, based on the flowgraph of Fig. 2.3, a scalar multiplication may be performed in about $6(m-1)$ clock cycles. The problem is that for large m, such design does not fit in single FPGA chip that is presently available commercially.

2.5 Conclusion

A high performance scalar multiplication scheme based on the Montgomery scalar multiplication algorithm has been proposed. The proposed scheme has been optimized to be dependent only to the number of finite field multiplications. It hides the multiplier's overhead and prevents a pipeline multiplier from stall.

A pseudo-pipelined word serial multiplier suitable for the scheme has been introduced. The underlying finite field multiplier performs loading and unloading of operands for the next operation while it performs the multiplication and hence a field multiplication takes $\lceil m/w \rceil$ clock cycles where w is the word size. The gate delay in the critical path of the multiplier is $T_{AND} + \lceil \log_2(w/k) \rceil T_{XOR}$, which is smaller than that of conventional word serial multipliers.

The proposed scheme performs a scalar multiplication in $25(m-1)$ clock cycles, which is approximately 2.75 times faster than a straight-forward implementation and 1.6 times faster than best implementations reported in this category in the open literature.

Appendix:Functions in Algorithm 1

Assume that E is a non-supersingular elliptic curve over $GF(2^m)$ defined as $y^2 + xy = x^3 + ax^2 + b$ and $P = (x, y) \in E(GF(2^m))$. Functions `Madd(.)`, `Mdouble(.)` and `Mxy(.)` in Algorithm 1 are defined as follows [LD99]. In these functions, global variables x and y are the coordinates of the original point P which are fixed during the computation of kP and, x_k and y_k are the coordinates of $Q = kP$.

function `Mdouble` (**input** X_1, **input** Z_1)
{ See (8) in [LD99]
$X \leftarrow X_1^4 + b \cdot Z_1^4$
$Z \leftarrow Z_1^2 \cdot X_1^2$
return (X, Z)
}

function `Madd` (**input** X_1, **input** Z_1, **input** X_2, **input** Z_2)

{ See (9) in [LD99]

$Z \leftarrow (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2$

$X \leftarrow x \cdot Z + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1)$

return (X, Z)

}

function `Mxy` (**input** X_1, **input** Z_1, **input** X_2, **input** Z_2)

{

$x_k = X_1/Z_1$

$y_k = (x + x_k)[(y + x^2) + (X_2/Z_2 + x)(X_1/Z_1 + x)] \times (1/x) + y$

return (x_k, y_k)

}

CHAPTER 3

Concurrent Error Detection for Finite Field Multiplication

Abstract We present concurrent error detection (CED) schemes for finite filed multiplication over extension field $GF(q^m)$, using polynomial basis. CEDs for finite field multipliers over $GF(2^m)$, based on simple parity bits, have been proposed in the literature. We generalize the one-bit parity concept over $GF(2^m)$ to multiple parity and multiple residue over $GF(q^m)$, using parity generation polynomials. We also provide sequential and parallel architectures. For parallel multipliers, we combine this concept together with N-fold redundant reduction modules to create a hybrid architecture. Implemented over $GF(2^m)$, proposed schemes are faster and smaller than simple-parity-bit schemes and provide the same or better error detection capability. For example, implemented on FPGA for 20 parity bits over $GF(2^{163})$, the overhead and output delay of the hybrid scheme are 7% and 39 nS, while those of the parity protected scheme are 80% and 198 nS, respectively.

3.1 Introduction

Finite field arithmetic is largely used in crypto-systems that are based on elliptic curves, discrete logarithm problem, block ciphers such as AES and in communication coding theory. Multiplication is the base operation in finite fields (FF). It consists of a polynomial multiplication over the base field, followed by a modular reduction operation. Over $GF(2^m)$, field inversion can be efficiently represented in terms of multiplication [IT88]. Modular exponentiation is also based on field multiplication. Therefore, robustness and reliability of

such crypo-systems largely depends on the correct operation of field multiplication. Although modern CMOS technology is reliable, faults do occur in lengthly computations and in systems that work under harsh environment conditions. Side channel attacks on crypto-systems include injection of faults and analyzing the erroneous output to extract the secret. In cryptography, random hardware faults produce incorrect output, which may expose the cryptographic secrets such as keys [BDL01]. Concurrent error detection (CED) method is one way to protect such systems against random faults and fault injection attacks [KWM01].

A fault, by definition, is the representation of physical defect in the abstract functional level. Error is the wrong output caused by the fault. Among faults models such as delay faults and bridging faults, stuck at fault is the most popular fault model in digital systems at the register transfer level (RTL) [BA00]. Stuck at fault is modeled by assigning a fixed value 0 or 1 to a line in the circuit. In the single or multiple stuck at fault model one or multiple lines are assumed faulty at a time, respectively. Stuck at fault is a functional model not a physical defect model.

Single-bit parity is perhaps the simplest form of error detection code. The basic concept is straightforward. Single-bit parity codes add an extra bit to an operand such that the sum of the bits on resulting codeword is one for odd parity and zero for even parity. The simplicity of parity bit has encouraged researchers to apply the parity protection concept to FF multiplication.

In arithmetic circuit, the concept of parity prediction have been used since long time ago. For example, in [LT70], parity bit prediction has been used for concurrent error detection in adders and in [FGB98] it has been used to detect errors in bit serial FF multipliers based on all one polynomials and bit serial Massey-Omura multipliers. Recent works in this area have been started with [RH03], where simple parity bit has been used for protecting serial and parallel FF multipliers. Same methods have been elaborated in [RH04b]. The single-bit parity scheme for least-significant-bit (LSB) first and most-significant-bit (MSB) first bit-serial FF multipliers and two types of fully parallel FF multipliers have been discussed in [RH06]. In [Lee08, Lee10, LM10] parity bits have been used to protect sequential normal

bases multipliers. In all of the architectures cited above the single-bit parity technique has been used, where the parity bit has been generated by computing the sum of the bits of the operand. The single stuck at fault model has also been used as the fault model.

One of the major drawbacks of single-bit parity codes is that they do not detect common multi-bit errors. Many variations of basic parity code have been proposed and implemented to overcome this limitation. In the parity-bit-per-byte technique, or multiple simple parity bit scheme, parity bits are assigned to every bytes of data rather than to the entire data [KK07]. This approach has been used in [BBK03], [BH05] and [BH07a] to detect random errors. In [BH05], each operand of the multiplier has been divided into multiple segment and the simple parity bit technique has been applied to each segment. In [BH07a], the same results have been extended by providing the FPGA implementation.

The main obstacle in using error detection logic is the area and time overhead. The straight forward increase of the number of simple parity bits for better error detection, as proposed in [BH07a], results in significant time and area overhead. All the above mentioned work are based on the similar concept of predicting the single-bit parity or multiple single-bit parity of the output and comparing it with the computed actual parity bit of the output.

Modular redundancy is a common approach for increasing reliability of systems, and have been widely used in digital systems since long time ago. The NASA space shuttle is an example of modular redundancy application in a complex system [NAS]. Modular redundancy replicates functional units, and therefore is costly. An error detection scheme is expected to be more efficient than a modular redundancy system. Otherwise, a redundant system may simply be used.

The generalization consists of a parity generation polynomial $g(x)$. This is different from conventional CEDs which use simple or multiple simple parity bits, based on sum of the bits of the operand, over $GF(2^m)$. The original contributions described, can be summarized as follows:

1. Provides CED architectures on any extension field, not just binary extension fields

(Section 3.2, 3.3 and 3.4) .

2. Defines multiple parities by simply changing the degree of the parity generation polynomial $g(x)$ (Section 3.2).

3. Provides overlapped parity by defining independent multiple parity generation polynomial $g(x)$ and $h(x)$. (Section 3.4.5).

4. Reduces the time and area overhead (Table 3.2 and 3.3).

5. Separates the parity generation and error detection logic from the multiplier (Fig. 3.1 and Fig. 3.2).

6. Provides CED, without imposing extra delay on the multiplier output ((3.15) and (3.18)).

7. The techniques are applicable to the three most common multipliers: bit-serial, parallel and digit-serial multipliers.

In Section 3.2 parity is defined and Lemmas for parity prediction are provided. In Section 3.3 and 3.4 architectures for CED on sequential and parallel multipliers are explained, respectively. Error model and probability of error detection are discussed in Section 3.6.

3.1.1 Notations

The following notations have been uses here:
T_X, S_X, T_A, S_A : Delay and area of two input XOR and AND gates, respectively
S_F : Area of the flip flop
$T_S, S_S, T_{RS}, S_{RS}, T_{CS}, S_{CS}$: Time and space complexity of the serial multiplier, parity prediction and parity check logic in serial multiplication.
S_M, S_{mod}: Space complexity of polynomial multiplication and reduction operation in parallel multiplication
R_P, R_S : Area overhead of parallel and serial multipliers.

3.1.2 Implementation Considerations

Logic synthesizers have the tendency to purge redundant logics and also optimize logic circuits using resource sharing, pipelining and retiming techniques. Multiplication schemes studied here contain redundancy by nature that are meant to project properties of a particular architecture. It is crucial to setup synthesizers to maintain the architecture and redundancy by establishing a rigid hierarchy in the RTL. We have examined the synthesized gate level RTL and reports carefully to make sure redundant and parity protection modules are not purged by optimizer and multiplier architecture has not changed by the synthesizer. All implementations here have been carried out on Xilinx XC3S5000 FPGA, the same FPGA used in [BH07a]. Sinplify Premier and Xilinx ISE have been used for synthesis and place & route, respectively.

As CMOS feature size shrink in size, interconnections dominate the overall percentage of delay in circuit. This leads to less timing correlation between the gate delay of the architecture and the delay of the implemented architecture. Conventionally, gate delay has been used as a merit in analytical comparison of different finite field multipliers and we use the same approach in our analytical comparisons.

In modern CMOS technology gate delay is not the only factor in determining speed of an architecture. At 40 nm and below pre-route timing estimates and final post place-and-route results may not have tight correlation [Syn08]. This is also applicable to FPGA designs. Therefore, synthesized architectures have been placed and routed, to obtain reliable timing result.

3.2 Parity Definition and its properties

Concurrent error detection schemes for finite field multipliers over $GF(2^m)$ using single parity bits have been used in [RH03], [RH04b], [RH06], [BH05], [BH07a] and [BH07b]. In this section we develop a mathematical model for parity protection, which protects the

polynomial multiplication and separates the parity protection logic from the multiplication logic. Then, we use modular redundancy to protect the modular operation.

3.2.1 Parity Definition

Let q be a prime and $A = A(x) = \sum_{i=0}^{m-1} a_i x^i$, where $a_i \in GF(q)$, be a polynomial in $GF(q)[x]$ and $g(x) \in GF(q)[x]$ be a degree $k < m$ monic polynomial. The data check, or the parity of A is defined as

$$p_A = P(A) \triangleq A \pmod{g(x)} \tag{3.1}$$

and $g(x)$ is called the parity generation polynomial.

The following Lemmas specify properties of the parity defined in (3.1), which will be used to derive the proposed concurrent error detection scheme.

If $\mathcal{P} = \{p_A | A \in GF(q^m)\}$ is a subfield of the finite field $GF(2^m)$, then from $C = AB \pmod{f(x)}$ it is implied that $p_C = p_A p_B \pmod{g(x)}$. However, the extension fields considered here do not have a subfield other than the base field.

In the following, the $\pmod{p}$ operation for scalar multiplications and polynomial additions, is implied. The is similar to $GF(2^m)$ operations, where $\pmod{2}$ is implied.

Lemma 3.2.1 *Let $A, B \in GF(q^m)$, $b \in GF(q)$ and $g(x)$ be a degree k polynomial over $GF(q)$, where $k < m$. The* predicted parity *of $b \cdot A$ and $A + B$ are*

$$p_{bA} = b \cdot p_A \tag{3.2}$$

and

$$p_{A+B} = p_A + p_B, \tag{3.3}$$

respectively.

Proof 3.2.1 *Using the definition of parity in (3.1), we get*

$$\begin{aligned} p_{bA} &= P(bA) = b \cdot A \pmod{g(x)} \\ &= b \cdot (A \pmod{g(x)}) = b \cdot p_A. \end{aligned}$$

Since b is a degree zero polynomial, multiplication by b does not need extra reduction mod $g(x)$ *. Also,*

$$\begin{aligned} p_{A+B} &= P(A+B) && (3.4) \\ &= (A+B) \pmod{g(x)} \\ &= A \pmod{g(x)} + B \pmod{g(x)} && (3.5) \\ &= p_A + p_B. \end{aligned}$$

The core operation in polynomial basis multiplication is multiplication by the basis x and reducing the results modulo $f(x)$. The following Lemma gives the parity of such operation.

Lemma 3.2.2 *Let* $f(x)$ *be a monic, irreducible and degree* m *polynomial generating* $GF(q^m)$ *and* $A \in GF(q^m)$ *. The predicted parity of* $A^{(1)} = xA \pmod{f(x)}$ *is*

$$p_{A^{(1)}} = xp_A \pmod{g(x)} - a_{m-1} f_g(x), \tag{3.6}$$

where

$$f_g(x) \triangleq f(x) \pmod{g(x)}. \tag{3.7}$$

Proof 3.2.2 *By definition we have* $A^{(1)} = xA \pmod{f(x)}$*.* $f(x)$ *is a monic polynomial in the form of*

$$f(x) = x^m + \sum_{i=0}^{m-1} f_i x^i,$$

where $f_i \in GF(q)$*. Let*

$$\begin{aligned} r(x) &= xA(x) - a_{m-1} f(x) \\ &= -f_0 a_{m-1} + \sum_{i=1}^{m-1} (a_i - f_i a_{m-1}) x^i. \end{aligned}$$

By division algorithm, $r(x)$ is the remainder of dividing $xA(x)$ by $f(x)$, since $deg(r(x)) < deg(f(x))$. Therefore, $r(x) = xA \pmod{f(x)}$ and

$$xA \pmod{f(x)} = xA - a_{m-1}f(x).$$

Using the preceding equation, the parity of $A^{(1)}$ is computed as follows

$$\begin{aligned} p_{A^{(1)}} &= P(xA \pmod{f(x)}) \\ &= P(xA - a_{m-1}f(x)) \\ &= xA - a_{m-1}f(x) \pmod{g(x)} \\ &= x(A \pmod{g(x)}) \pmod{g(x)} \\ &\quad - a_{m-1}f(x) \pmod{g(x)}. \end{aligned}$$

Using (3.1) and (3.7), we get

$$p_{A^{(1)}} = xp_A \pmod{g(x)} - a_{m-1}f_g(x).$$

Lemma 3.2.3 *Assume $A, B, U \in GF_q[x]$ and*

$$C(x) = A(x) \times B(x) + U(x).$$

Then,

$$p_C = (p_A \times p_B) \pmod{g(x)} + p_U. \tag{3.8}$$

Proof 3.2.3 *Using our assumption, we have*

$$\begin{aligned} C(x) \pmod{g(x)} &= A(x) \times B(x) + U(x) \pmod{g(x)} \\ &= [A(x) \pmod{g(x)} \\ &\quad \times B(x) \pmod{g(x)} \\ &\quad + U(x) \pmod{g(x)}] \pmod{g(x)}. \end{aligned}$$

Using parity definition in (3.1), (3.8) follows.

3.3 Error Detection in Sequential multipliers

Based on the Lemmas in the previous section, we construct a sequential error detection method (SED) for sequential multipliers over extension fields. Let $GF(q^m)$ be an extension field constructed using the monic irreducible polynomial $f(x)$ over $GF(q)$. Let $\{1, \alpha, \alpha^2, \ldots, \alpha^{2^m-1}\}$ be the polynomial basis. Any element $A \in GF(q^m)$ can be represented as

$$A = A(\alpha) = (a_{m-1}, \ldots, a_1, a_0) = \sum_{i=0}^{m-1} a_i \alpha^i,$$

where $a_i \in GF(q)$ are the coordinates of A. Let C' be the product of two polynomials $A, B \in GF(q^m)$. Then, C' can be represented as

$$C' = A \times B = A \times \sum_{i=0}^{m-1} b^i \alpha^i = \sum_{i=0}^{m-1} b^i (\alpha^i A).$$

The product of A, B in $GF(q^m)$ is

$$\begin{aligned} C &= C' \pmod{f(\alpha)} \\ &= \sum_{i=0}^{m-1} b_i \left[(\alpha^i A) \pmod{f(\alpha)} \right] \\ &= \sum_{i=0}^{m-1} b_i A^{(i)}, \end{aligned} \tag{3.9}$$

where

$$A^{(i)} \triangleq (\alpha^i A) \pmod{f(\alpha)}. \tag{3.10}$$

It is possible to construct $A^{(i)}$ recursively as follows

$$A^{(i+1)} = \alpha A^{(i)} \pmod{f(\alpha)}. \tag{3.11}$$

Equations (3.9) and (3.11) describe a sequential multiplication algorithm, where the intermediate results are accumulated in m cycles. Let $C^{(k)}$ be the intermediate result at step k. That is,

$$C^{(k)} \triangleq \sum_{i=0}^{k} b_i A^{(i)},$$

and it follows that $C^{(m)} = A \times B \pmod{f(x)}$. Assuming the result at the $(i-1)$th iteration is known, the result in the ith iteration is

$$C^{(i)} = C^{(i-1)} + b_i A^{(i)}. \tag{3.12}$$

The preceding equation leads us to construct a state machine for parity prediction.

Theorem 3.3.1 *Let $C^{(i-1)}$ be the intermediate result of the sequential multiplication operation in (3.12) and lets assume $p_{C^{i-1}}$ is known. The predicted parity of $C^{(i)}$ is*

$$p_{C^{(i)}} = p_{C^{(i-1)}} + b_i \left[x p_{A^{(i-1)}} \pmod{g(x)} - a_{m-1}^{(i-1)} f_g(x) \right]. \tag{3.13}$$

Proof 3.3.1 *Reducing both sides of (3.12) modulo $g(x)$, we get*

$$\begin{aligned} p_{C^{(i)}} &= P(C^{(i-1)} + b_i A^{(i)}) \\ &= p_{C^{(i-1)}} + b_i P(A^{(i)}) \end{aligned}$$

Using Lemma 3.2.1 and 3.2.2, (3.13) follows.

3.3.1 Architecture

Fig. 3.1 depicts a sequential multiplier, its corresponding parity prediction architecture and the error detection logic. The multiplier on the right side of the figure is based on (3.12) and the parity detection on the top left is based on (3.13). It can be observed that the parity prediction architecture is very similar to that of sequential multiplier. In each iteration, it multiplies the parity of $A^{(i)}$ by the basis x, reduces the result modulo the parity generation polynomial and subtracts $a_{m-1} f_g(x)$ to produce the predicted parity $p_{A^{(i+1)}}$. In one iteration, the parity prediction logic works independent of the multiplication operation. Overall, the parity prediction architecture is dependent to the multiplier, since in the i-th cycle it requires the a_{m-1} coefficient of $A^{(i)}$. Error signal can be generated by comparing the parity of a node on the multiplier and compare it with its corresponding node on the parity predication architecture in Fig. 3.1.

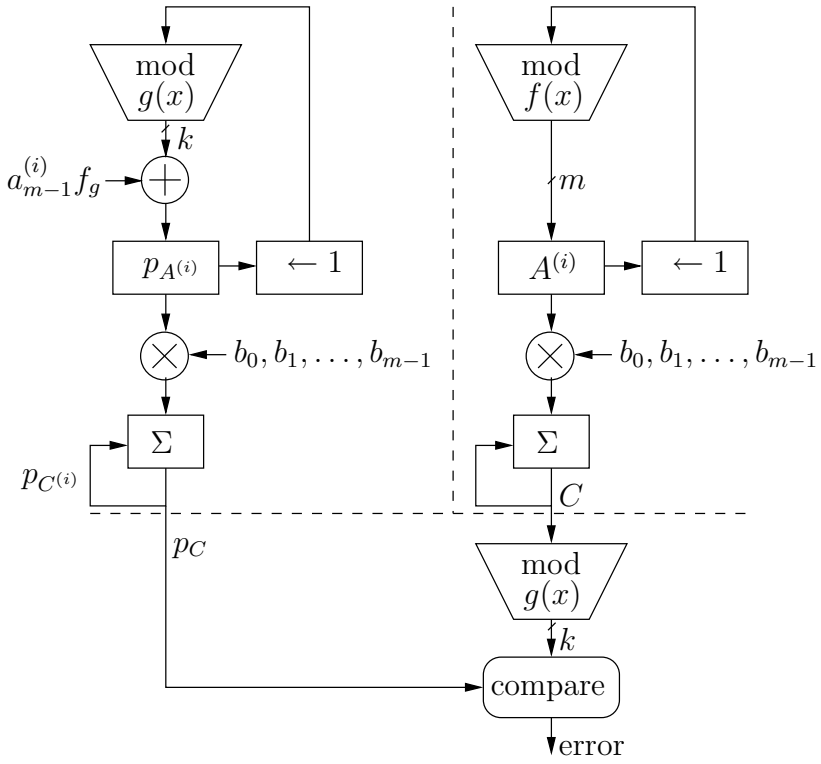


Figure 3.1: Error detection model for sequential multiplication over $GF(q^m)$

3.3.2 Complexities and Overhead Over $GF(2^m)$

We derive complexities for sequential multiplier over $GF(2^m)$. The delay of $xA \pmod{f(x)}$ operation is T_X and it requires $(w-1)$ XOR gates for the reduction operation, where w is the Hamming weight of the irreducible polynomial $f(x)$. The delay of the integration part $C^{(i)} = C^{(i-1)} + b_i A^{(i)}$, is $T_A + T_X$, and it requires m NAND gates. Thus, the time and space complexity of the sequential multiplier are

$$T_S = T_A + T_X$$

and

$$S_S(m, w) = (S_A + S_X + 2S_F)m + (w-1)S_X, \tag{3.14}$$

respectively.

The architecture of parity prediction is very similar to that of the multiplier. Except, one extra addition to the $a_{m-1}f_g(x)$ term is required. Assuming $f_g(x)$ is known, adding to $a_{m-1}f_g(x)$ takes T_X and requires k XOR gates. Thus, the delay and area of the parity prediction is

$$\begin{aligned} T_{RS} &= \max(T_X + T_X, T_X + T_A) \\ &= 2T_X \end{aligned} \tag{3.15}$$

and

$$S_{RS} = S_S(k, w_g) + kS_X,$$

where w_g is the hamming weight of the parity generation polynomial $g(x)$. The parity of the final output C needs to be generated and compared to the predicted parity p_C at the end of multiplication. In order to compute the parity of product output C, a mod $g(x)$ operation at the output is needed, which requires $(w_g - 1)(m-k)$ addition in the ground field [EYK06]. Then, p_C and the parity generated from C parity need to be compared using k XOR gates. The comparison results need to be ORed together using $k-1$ OR gates to produce final error signal. Here, for simplicity, we assume AND gates are used. In practice, logic synthesizers

use NAND gates and inverters, which are smaller and faster than OR gates and AND gates. The time and space complexity of the final parity check logic are

$$S_{CS} = (w_g - 1)(m - k)S_X + kS_X + (k - 1)S_A$$

and

$$T_{CS} = \lceil \log_2 \frac{m}{k} + 1 \rceil T_X + \lceil \log_2 k \rceil T_A, \tag{3.16}$$

respectively.

3.3.2.1 Area and Delay Overhead

The area overhead is

$$R_S = \frac{S_{RS} + S_{CS}}{S_S}.$$

In order to have a better picture, we approximate the area overhead. An XOR gate and a master slave flip-flop can be constructed using four and eight NAND gates, respectively. That is $S_X = 4SA$ and $S_F = 8S_A$. In practice, S_X and S_F are much smaller. Let have the minimum weight $w_g = 2$ for the parity generation polynomial, from preceding equation we get

$$\begin{aligned} R_S &= \frac{[2kS_F + (1 + 3k)S_A] + [(k - 1)S_A + mS_X]}{2mS_F + mS_A + (m + w - 1)S_X} \\ &\approx \frac{25k + 4}{21m + 4w - 4} + \frac{8m + k - 1}{21m + 4w - 4} \\ &\approx 1.2\frac{k}{m} + 0.4, \text{where we assume } m \gg k, m \gg w. \end{aligned}$$

The overhead equation indicates that overhead has a constant part, which is due to the parity checker and a linear part that belongs to the parity prediction logic and depends on the number of parity bits k. For eight parity bits over $GF(2^{163})$ the area overhead is about 56% and, as it will be discussed in (3.26) Section 3.6, the probability of undetected error is about 0.4%.

The critical path of the multiplier with error detection is T_{RS} (3.15). On FPGA, there will be no penalty, since $T_A = T_X$. On ASIC, the critical path might be slightly longer since

$T_X \geq T_A$. The error signal takes T_{CS} (3.16) longer than the final multiplication result C to appear on the output. Assuming $T_X \approx T_A$, T_{CS} can be approximated as

$$T_{CS} \approx T_X \log_2 m.$$

The computation time for the sequential multiplier is $T = 2mT_X$. Since $2mT_X \gg T_X \log_2 m$, the overall delay overhead is not significant.

3.3.3 Multiple Residue CED

If error vector is a multiple of the parity generation the polynomial $g(x)$, it becomes undetectable. To increase the probability of error detection, either the number of parities can be increased, or multiple parity generation polynomials $g(x)$ and $h(x)$ can be used. The multiple residue approach resembles the well known overlapping parity method, where each coefficient of the operand appeasers in more than one parity group. If one parity group does not detect the error, there is a high probability that the other parity group detects it. In the single parity polynomial approach, each coefficient of the operand A is contained only in one parity group. Since the proposed method isolates the error detection logic from the multiplier, construction of overlapping parity is straightforward. Simply two parity generation polynomials $g(x)$ and $h(x)$ can be defined, as indicated in (3.8) and (3.13), to construct the corresponding parity prediction logic. Then, OR the output error signals to generate the final error signal.

In this case, two parity generation logic on the left side of Fig. 3.1 are needed. One for predicting the parity modulo $g(x)$ and the other for predicting the parity modulo $h(x)$. If $g(x)$ is of degree k_1 and $h(x)$ is of degree k_2, we will have $k_1 + k_2$ parities, but the parities are overlapping. This is different from having a single parity generation polynomial of degree $k_1 + k_2$.

3.3.4 Single-bit and Multiple Single-bit Parity Schemes over $GF(2^m)$

Assume working on the base field $GF(2)$ and $g(x) = x + 1$, then $p_A = \sum_{i=0}^{m-1} a_i$, which is equivalent to the common definition of single-bit parity of A. In this case, the proposed scheme is equivalent to the parity protected bit-serial multiplier introduced in [RH03] and [RH06].

To have k simple parity bits for A, we can set $g(x) = x^k + 1$. Then,

$$\begin{aligned} p_A &= A(x) \pmod{x^k + 1} \\ &= \sum_{i=0}^{k-1} p_i x^i, \end{aligned}$$

where

$$p_i = \sum_{j=0}^{\lceil m/k \rceil} \hat{a}_{i+jk} \quad \text{and} \quad \hat{a}_j = \begin{cases} a_j & \text{if } 0 \leq j < m \\ 0 & \text{otherwise.} \end{cases}$$

The above definition divides the operand A into $\lceil \frac{m}{k} \rceil$ k-bit partitions. Then, defines one parity bit for a group of bits which consist of the j-th element in each of the $\lceil \frac{m}{k} \rceil$ partition. This case, which is a parity-bit-per-word technique, would be similar to the multiple simple parity bit in bit-serial multiplier proposed in [BH07a]. The error detection methods in [RH03, RH06] and [BH07a], which seems to be ad-hoc, are actually special cases of the proposed model, and can be explained mathematically.

3.3.4.1 Comparison

The proposed SED method has the following advantages over simple parity bit multipliers in [RH04b,BH07a]. First, SED can be used over $GF(q)$, where $q \neq 2$. Second, multiple parity bits can be easily defined using degree k polynomial $g(x)$. Third, for better error protection overlapped parity can be implemented using multiple parity generation polynomial $g(x)$ (Section 3.4.5). And, finally, architecture of the parity prediction logic is similar to that of the multiplier, and can be implemented separately from the multiplier itself. The area and time overhead over $GF(2^m)$ for single and multiple parity bits are similar to those

of [RH04b, BH07a].

3.4 Error Detection in Parallel Polynomial Basis Multiplication

A finite field multiplication can be subdivided into polynomial multiplication and modular reduction. In this section we develop a hybrid error detection (HED) method, which protects the polynomial multiplication and separates the parity protection logic from the multiplication logic. Then, it uses modular redundancy to protect the modular operation.

3.4.1 Protection of Modular Reduction Operation

The space complexity of modular reduction is smaller than that of polynomial multiplication, when irreducible polynomial is known. For example over $GF(2^m)$, constructed using irreducible polynomial $f(x)$, the area of polynomial multiplier and reduction operation area

$$S_M = m^2 S_A + (m-1)^2 S_X \tag{3.17}$$

and

$$S_{mod} = (m-1)(w-1)S_X,$$

respectively, where S_A and S_X are the area of AND and XOR gates and w is the hamming weight of $f(x)$ [EYK06]. The area ratio is $S_M/S_{mod} \approx 2m/w$. In practice, low-weight irreducible polynomials are used, where we have $m \gg w$ and therefore $S_M/S_{mod} \gg 1$. Thus, it is expected that redundant reduction modules does not cause large area overhead if N redundant modules are used to protect the modular reduction operation.

3.4.2 Architecture

The overall architecture is depicted in Fig. 3.2, where the critical path has been shown by a dash-dot line. The polynomial multiplier and its corresponding parity prediction logic are enclosed in the dotted polygon. Input operands A and B are multiplied using a polynomial multiplier over $GF(q)$. Then, the result is reduced modulo $f(x)$ using N redundant reduction

operation. Errors of modular reductions are detected by comparing outputs $D^{(j)}(x) = \sum_{i=0}^{2m-2} d_i^{(j)} x^i$ of each reduction module, where $1 \leq j \leq N$ and $d_i^{(j)} \in GF(2)$. Let e_i denote error for the i-th bit of the output $D^{(j)}$. Then, $e_i = (d_i^{(1)} \vee d_i^{(2)} \vee \ldots \vee d_i^{(N)}) \wedge (\bar{d}_i^{(1)} \vee \bar{d}_i^{(2)} \vee \ldots \vee \bar{d}_i^{(N)})$, where $\wedge$ and $\vee$ are logical AND and OR operations. If there is an error in any of the output bits, there is a fault in the system. Thus, the error signal is $e = e_0 \vee e_1 \vee \ldots \vee e_{m-1}$.

The parity prediction logic, is shown on the left side of the Fig. 3.2. First, it computes the parity of A and B using modulo $g(x)$ operations. Then, parities are multiplied together using a polynomial multiplier and reduced modulo $g(x)$ to produce the predicted parity of $A \times B$, as shown in (3.8). The result is compared to the actual parity of $A \times B$ using XOR gates and ORed with the error output of the reduction to produce the final error signal. Each line in Fig. 3.2 carries a polynomial, where the number of coefficients are indicated in the figure. In the following sections, we provide analysis and implementation results for the delay, area, overhead and error model of the architecture in Fig. 3.2 over $GF(2^m)$.

The architecture in Fig. 3.2 depicts the general form of error detection in modular multiplication. The concept can be utilized in different configurations. For our study, we have implemented fully parallel polynomial multipliers for both data and parity bits. Alternatively, digit serial or sub-quadratic polynomial multiplication algorithms may be used for the parity or the main polynomial multiplier to tradeoff speed and area [Mon05].

3.4.3 Complexities and Overhead Over $GF(2^m)$

Here, we derive area and time complexity of the HED, to determine the overhead of HED compared to the high speed parallel multiplication with no CED. Table 3.1 shows gate count and gate delay of each module in Fig. 3.2, implemented over $GF(2^m)$. The modules are numbered for ease of reference and T_A and T_X are gate delay of two input AND and XOR gates, respectively. Hereafter, we refer to the area and delay of the module i as S_i and T_i, respectively. For each module, the delay and area equations can be obtained using information provided in [EYK06].

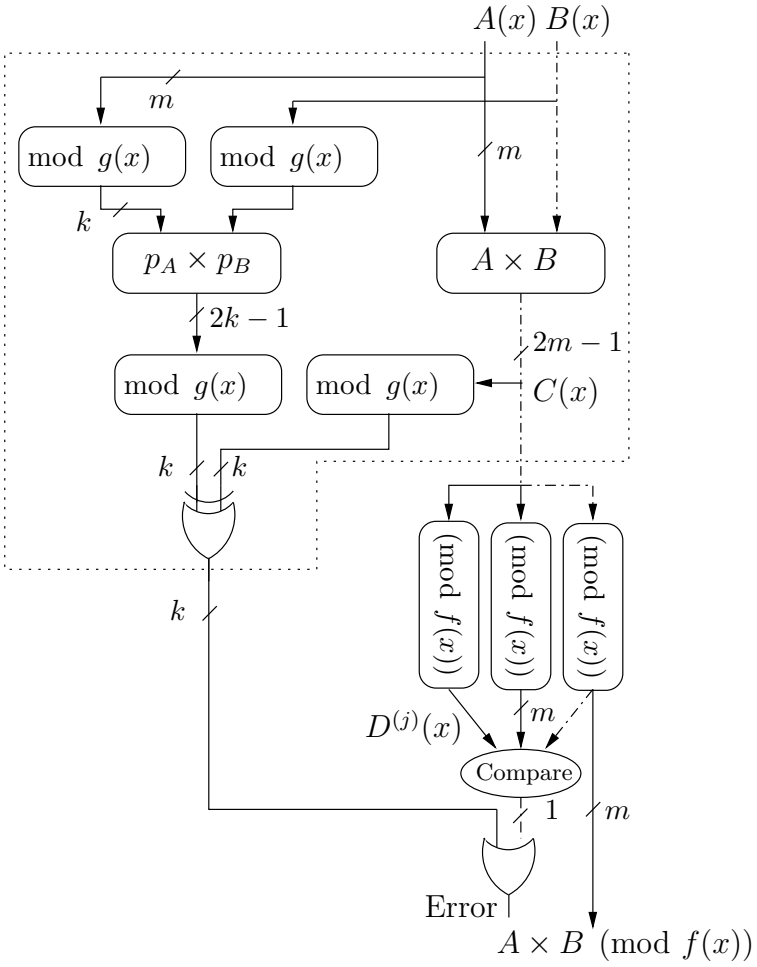


Figure 3.2: HED for finite field multiplication. The dash-dot line indicates the critical path for the error signal.

Table 3.1: Delay and area for the HED over $GF(2^m)$

Fig.	**Module No.**	**Functionality**	**Area** $k \geq 1$	**Delay**
3.2	1	$A \times B$	$m^2 S_A + (m-1)^2 S_X$	$T_A + \lceil \log_2 m \rceil T_X$
3.2	2	$N\times \pmod{f(x)}$ modules	$(m-1)(w-1)S_X$	δT_X, $\delta \in \{2,4\}$ for NIST polynomials
3.2	3	two $\pmod{x^k+1}$ modules, with input width m	$2(m-k)S_X$	$\lceil \log_2(m/k) \rceil T_X$
3.2	4	mod x^k+1, with input width $2k-1$	$(k-1)S_X$	T_X
3.2	5	mod x^k+1, with input width $2m-1$	$(2m-1-k)S_X$	$\lceil \log_2(2m-1/k) \rceil T_X$
3.2	6	$p_A \times p_B$	$k^2 S_A + (k-1)^2 S_X$	$T_A + \lceil \log_2 k \rceil T_X$
3.2	7	k XOR gates	kS_X	T_X
3.2	8	comparator, for the output of redundant modules	$(2mN-1)S_A$	$(1 + \lceil \log_2 N \rceil + \lceil \log_2 m \rceil)T_A$
3.2	9	error signal generator	kS_A	$\lceil \log_2(k+1) \rceil T_A$

3.4.3.1 Critical Path

The delay of the output result $C = AB \pmod{f(x)}$ can be expressed as

$$\begin{aligned} T_{out} &= T_1 + T_2 \\ &= T_A + (\delta + \lceil \log_2 m \rceil) T_X . \end{aligned} \tag{3.18}$$

The output delay is identical to the delay of a parallel finite field multiplier that performs modular reduction after polynomial multiplication without mixing these operations. The gate delay of the error signal T_{err} is

$$\begin{aligned} T_{err} &= T_{out} + T_8 + T_9 \\ &= (1 + \lceil \log_2(k+1) \rceil + \lceil \log_2 mN \rceil + \lceil \log_2 m \rceil) T_A . \end{aligned} \tag{3.19}$$

The delay of the error signal is larger than that of output. In contrary to the parity protected multiplier in [BH07a] (hereafter called LUM), the multiplier output is generated independent of the error signal. In finite field processors, such as elliptic curve crypto-processors, where many multiplications are performed, the multiplier output may be used in processing and the error signal can be taken into account after it is ready. This approach prevents excessive delay in processing but requires more attention in the implementation. For the synthesis, the error output should be excluded from the critical path or an extra pipeline delay could be added. The error signal will then be available with one cycle delay.

3.4.3.2 Area Overhead

The area overhead of the hybrid protection method equals the area of the parity prediction and error detection logic divided by the area of the original parallel multiplier. Thus, the area overhead is

$$R_P = \frac{(S_1 + NS_2 + \sum_{i=3}^{9} S_i) - (S_1 + S_2)}{(S_1 + S_2)} . \tag{3.20}$$

The above equation can be expressed in terms of design parameters m, w, N, k, S_A and S_X. On FPGAs, where $S_A = S_X$, (3.20) is reduced to

$$R_P = \frac{-2k + 2k^2 + m(8 + 2w) - 2w}{2 + 2m^2 + m(-3 + w) - w}. \tag{3.21}$$

Since the multiplication of parity bits is not in the critical path, slower multiplication scheme with less space complexity can be utilized without affecting the overall speed. Using sub-quadratic polynomial multiplication algorithms, the area overhead of the hybrid scheme can be reduced. In particular, for $k = 2^t$, where $t \in \mathbb{N}$, the recursive Karatsuba-Ofman algorithm (KOA) can be used. The space complexity of a k-bit KOA multiplier is

$$S_6 = k^{\log_2 3} S_A + (6k^{\log_2 3} - 8k + 2)S_X.$$

Using (3.20), the area overhead using KOA is

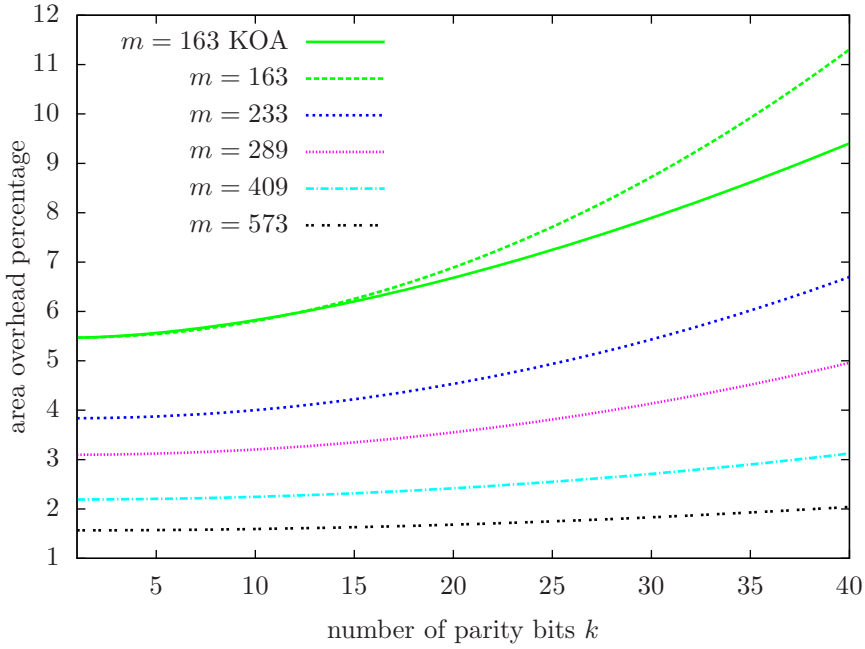
$$R_P = \frac{1 - 8k + 7k^{\log_2 3} + m(8 + 2w) - 2w}{2 + 2m^2 + m(-3 + w) - w}.$$

Fig. 3.3(a) shows the estimated area overhead for the NIST recommended fields and the KOA variant for $GF(2^{163})$. It is verified that the KOA variant provided less overhead for large number parities, where $k = 2^t$. If the polynomial multiplication for parities is too slow, the critical path for the error signal can be affected.

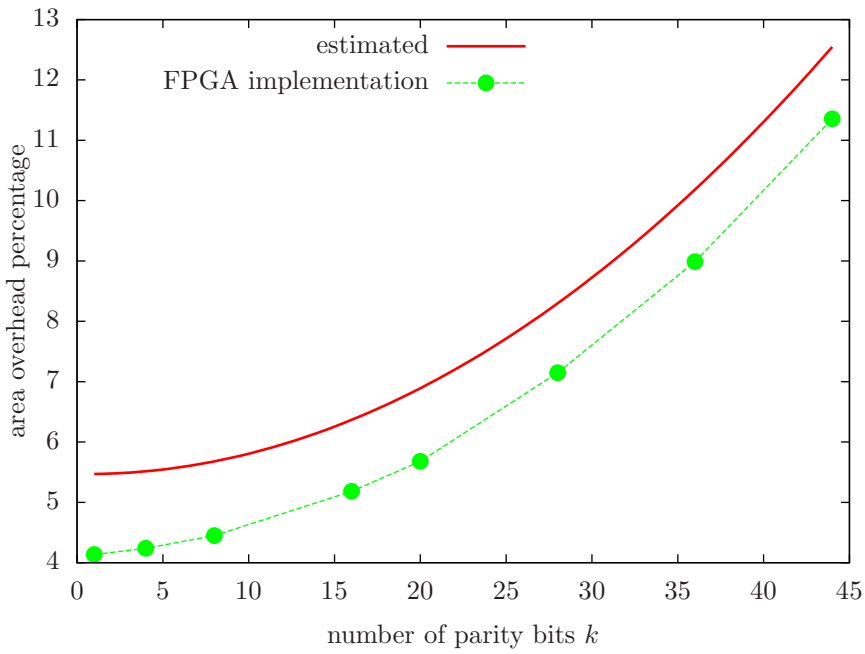
Fig. 3.3(a) indicates that for $k = 20$ and $m = 163$, the area overhead is less than 7%, while the overhead for the same number of parity bits for the LUM in [BH07a] is about 80% (Table 3.2). For large fields, it is possible to achieve good error protection with small overhead. For example, for $GF(2^{571})$ the overhead is less than 2% for 40 parity bits. As it will be discussed in (3.26) Section 3.6, the probability of undetected error is about 2^{-40} for large error probability.

3.4.4 Constructing Common Single-bit and Multiple Single-bit Parity Schemes over $GF(2^m)$

Assume working on the base field $GF(2)$ and $g(x) = x + 1$, then $p_A = \sum_{i=0}^{m-1} a_i$, which is equivalent to the common definition of single-bit parity of A. In this special case, the



(a) The estimated area overhead v.s. number of parity bits for $m \in \{163, 233, 289, 409, 573\}$



(b) Implemented and estimated (3.22) area overhead for the HED architecture over $GF(2^{163})$, v.s. number of parity bits

Figure 3.3: Implementation results

proposed scheme is equivalent to the parity protected bit-serial multiplier introduced in [RH03] and [RH06].

To have k simple parity bits for A, simply set $g(x) = x^k + 1$. Then,

$$p_A = A(x) \pmod{x^k + 1} = \sum_{i=0}^{k-1} p_i x^i, \text{where}$$

$$p_i = \sum_{j=0}^{\lceil m/k \rceil} \hat{a}_{i+jk} \quad \text{and} \quad \hat{a}_j = \begin{cases} a_j & \text{if } 0 \leq j < m \\ 0 & \text{otherwise.} \end{cases}$$

The above definition divides the operand A into $\lceil \frac{m}{k} \rceil$ k-bit partitions. Then, it defines one parity bit for a group of bits which consist of the jth element in each of the $\lceil \frac{m}{k} \rceil$ partition. This case, which is a parity-bit-per-word technique, would be similar to the multiple simple parity bit in bit-serial multiplier proposed in [BH07a]. The error detection methods in [RH03, RH06] and [BH07a], which seems to be ad-hoc, are special cases of $q = 2$ and $g(x) = x + 1$ of the proposed model, and can be explained mathematically.

3.4.4.1 Comparison

The proposed SED method has the following advantages over simple parity bit multipliers in [RH04b, BH07a]. First, SED can be used over any extension field $GF(q^m)$. Second, multiple parity bits can be easily defined using degree k polynomial $g(x)$. Third, for better error protection overlapped parity can be implemented using multiple parity generation polynomial $g(x)$ (Section 3.4.5). Finally, architecture of the parity prediction logic is similar to that of the multiplier and can be implemented separately from the multiplier itself. The area and time overhead over $GF(2^m)$ for single and multiple parity bits are similar to those of [RH04b, BH07a].

3.4.5 Multiple Parity Generation Polynomial and Error Correction

If error vector is a multiple of the parity generation polynomial polynomial $g(x)$, it becomes undetectable. To increase the probability of error detection, either the number of parity

bits can be increased, or multiple parity generation polynomials may be used. Definition of multiple parity generation polynomials is similar to the well known overlapping parity method, where each data bit appeasers in more than one parity group. If one parity group does not detect the error, there is a high chance that the other parity group detects it. In single parity polynomial approach, each data bit is contained in only one parity group.

Since the proposed method isolates the error detection logic from the multiplication, construction of overlapping parity in straightforward. Simply define two parity generation polynomials and construct the corresponding parity prediction logic, as explained earlier and shown in Fig. 3.2. Then, OR the output error signals to generate the final error signal.

Errors seldom happen on a single bit. Yet, under certain conditions, multiple parity generation polynomials can be used to located one error bit as well as correcting it by complementing the error bit. Choose another polynomial $h(x)$ as indicated in (3.8) and (3.13), and construct parity prediction logic for each of them. This approach can be applied to all multiplier architectures introduced here and shown in Fig. 3.1 and 3.2.

Let $g(x) = x^{k'} + 1$ and $h(x) = x^k + 1$ be two parity generation polynomials, where k' and k are coprime and $kk' \geq 2m - 1$ and $e(x) = x^i$ be the single bit error signal, where $0 \leq i < 2m$.

Let the erroneous output of the polynomial multiplier be $\widetilde{C}(x) = C(x) + e(x)$ and the parity of output $\widetilde{C}(x)$ with respect to $g(x)$ and $h(x)$ be $p_{\widetilde{C}} = \widetilde{C}(x) \pmod{g(x)}$ and $q_{\widetilde{C}} = \widetilde{C}(x) \pmod{h(x)}$, respectively. Application of multiple parity generation polynomial technique to the sequential multiplier is straightforward.

Let $\hat{p}_C$ be the predicted parity of the C using polynomial $g(x)$. Then, the output of the first parity checker is

$$\begin{aligned} \hat{p}_C + p_{\widetilde{C}} &= C \pmod{g(x)} + \widetilde{C} \pmod{g(x)} \\ &= P(e) = P(x^i) \\ &= x^i \pmod{g(x)} = x^{i \pmod{k'}} = x^{j_1}, \end{aligned}$$

where $j_1 \in \mathbb{Z}_\ell$. Similarly, for $h(x)$ we have

$$\hat{p}_C + p_{\widetilde{C}} = x^i \pmod{h(x)} = x^{i \pmod{k}} = x^{j_2},$$

where $j_2 \in \mathbb{Z}_k$. The parity check output $(j_1, j_2) \in \mathbb{Z}_{kk'}$ can be used as error syndrome. Since k and k' are coprime, (j_1, j_2) uniquely identifies the erroneous bit $i < kk'$ and can be computed using the Chinese remainder theorem.

Error correction for modular reduction modules is straight forward. Using a voter, a three modular redundancy for modular reductions can correct errors. Similar methods has been used to correct single-bit errors in integer arithmetic [RG71].

3.4.6 FPGA Implementation and Comparison

We have implemented the HED scheme on the Xilinx XC3S5000 FPGA, the same FPGA used in [BH07a]. The Sinplify Premier and Xilinx ISE have been used for the synthesis and place & route, respectively. In this implementation, triple modular redundancy for the reduction modules and the NIST recommended polynomial $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$ for ECDSA have been used. Therefore, we have $m = 163, w = 5, N = 3$ and $S_A = S_X$. Using these parameters, the area overhead (3.20) is simplified to

$$R_P = \frac{(2924 - 2k + 2k^2)}{53461}. \tag{3.22}$$

Fig. 3.3(b) shows the implementation results along with the area overhead estimation in (3.22). We have obtained the area overhead of the implemented architecture using (3.20), where the area of each module is represented by the number of LUTs. Fig. 3.3(b) indicates that implementation results follow the predicted overhead in (3.22) with few percent offset. The figure shows that for 8 parity bits the overhead is less than 6%, while the area overhead in parity protection scheme presented in [BH07a] is at least 50% (Table 3.2) for the same number of parity bits.

Table 3.2 shows the FPGA implementation results of the HED compared with the LUM, where the number of redundant reduction modules is $N = 3$. Using optimization techniques

Table 3.2: FPGA implementation results for HED v.s. LUM over $GF(2^{163})$

#Parity	**#Slices**		**Delay (nS)**	
bits	HED	LUM [BH07a]	HED	LUM see text
0	14620	13541	30	150
4	15050	19121	37	
8	15079	20049	38	
12	15102	21616	39	> 198
16	15122	22863	39	
20	15172 7% overhead	24390 80% overhead	39	
44	15724	>40000 estimate	39	

in Synplicity it is possible to achieve better timing for HED. Timing data for LUM has not been provided in [BH07a]. Therefore, we have implemented the LUM scheme as stated in [BH07a] and provided the timing information in Table 3.2. As it has been shown, for the same number of parity bits, the HED architecture provides smaller and faster design. The small area overhead of the HED allows a larger number of parity protection bits to be implemented. For example, our implementation results indicate that for $k = 40$ parity bits the LUT overhead is about 10% and delay is 39 nS and the probability of undetected error is almost 2^{-40} for large error probability (Section 3.6). The data provided in [BH07a] indicate that for $k = 40$ the area overhead LUM would more than 100%. Based on our implementation results, the delay of LUM is larger than 198 nS, which leads to more than 395% delay overhead considering a finite field multiplication can be performed in less than 39 nS. The probability of undetected error in both cases is almost the same.

The area and delay overhead equations for LUM and the HED over $GF(2^{163})$ as a function of number of parity bit k are summarized in Table 3.3. The overheads of HED is obtained using (3.18) and (3.21) and those of LUM are obtained using data provided in [BH07a]

and [RH06]. The Table indicates that for any meaningful number of parity bits, the HED is faster and has less overhead compared with LUM methods.

3.5 Error Detection in Digit Serial Multipliers over Extension Fields

Digit serial finite field multipliers are commonly used as a trade-off between area and speed. In a digit serial multiplier one of the operands is divided into K digits and the multiplication is performed in K cycles. Let $K = \lceil m/d \rceil$ and $B = \sum_{i=0}^{m-1} b_i x^i$, where $b \in \{0, 1\}$, be expressed using d-bit digits as follows

$$B = \sum_{i=0}^{K-1} B_i x^{di}, \text{ where } B_i = \sum_{j=0}^{d-1} \hat{b}_{(di+j)} x^j$$

$$\text{and} \quad \hat{b}_i = \begin{cases} b_i & \text{if } 0 \leq i \leq m-1 \\ 0 & \text{otherwise.} \end{cases}$$

The product of A and B over the field is

$$Z = \sum_{i=0}^{K-1} A \times B_i x^{di} \pmod{f(x)}. \tag{3.23}$$

Let C_i be defined recursively as

$$C^{(i-1)} \triangleq x^d C^{(i)} + A \times B_{i-1}, \tag{3.24}$$

and

$$Z^{(i)} = C^{(i)} \pmod{f(x)}, \tag{3.25}$$

where $C^{(K)} = 0$ and i is counting down from K down to 0. Then, $Z^{(0)}$ is the product of A and B in (3.23), and, (3.24) and (3.25) describe the digit serial multiplication.

Using (3.24) and (3.8), the parity prediction equation is

$$p_{C^{(i-1)}} = p_{x^d} \times p_{C^{(i)}} + p_A \times p_{B_{i-1}} \pmod{g(x)}.$$

Table 3.3: Delay and area overhead comparison over $GF(2^{163})$

	Delay overhead %		**Area overhead %**
HED (conventional multiplier)	output: 0	error signal: ≈ 100	$3.74 \times 10^{-3}(1462 - k + k^2)$
HED (KOA multiplier)	output: 0	error signal : ≈ 100	$1.3 \times 10^{-2}(417 - 1.14k + k^{\log_2 3})$
LUM [BH07a]	> 395		$23.41 + 2.87k$
LUM [RH06]	≈ 69 for $k = 1$		1.8 for $k = 1$

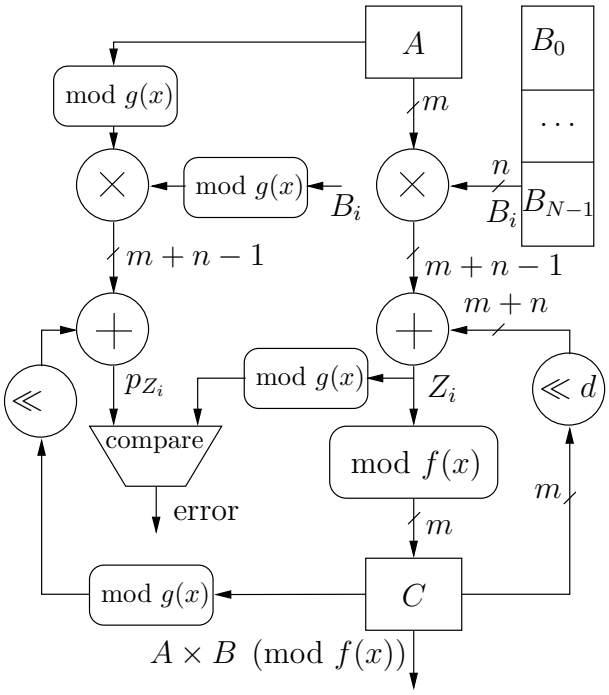


Figure 3.4: CED for digit serial finite field multiplication

Therefore, the parity of C in the next iteration can be predicted using the parity of C in the current iteration, parity of A and parity of the digit B_i. Since d is known, p_{x^d} can be precomputed. The modular reduction in (3.25) can be protected using modular redundancy, as explained earlier.

3.6 Error Model and Coverage

In this section, the error model is explained and equations for error detection probabilities are derived. A fault in the multiplier is assumed to flip one or more bits of the output or an intermediate result. Such fault can be modeled by a polynomial addition to the correct output value, over $GF(2)$. Let $C(x)$ be the output or an intermediate result polynomial of the degree $n-1$ and be protected by a parity generation polynomial $g(x)$. Let $e(x) = \sum_{i=0}^{n-1} e_i x^i$, where $e_i \in GF(2)$, represent the error polynomial have te same degree as $C(x)$. Then, the erroneous output can be represented as $\widetilde{C}(x) = C(x) + e(x)$. Each error bit at location i

of the erroneous output $\widetilde{C}(x)$ is represented by the corresponding coefficient $e_i = 1$ in $e(x)$. To have k protection bits, a degree k parity generation polynomial is required. We pick the degree k binomial $g(x) = x^k + 1$ as the parity generation polynomial, to simplify modular reduction operations.

3.6.1 Undetected Errors

If $e(x)$ is a multiple of the parity generation polynomial $g(x) = x^k+1$, then $C(x) \pmod{g(x)} = \widetilde{C}(x) \pmod{g(x)}$, which implies that the error detection module cannot detect the error. In order to derive the probability of undetected errors, we need to find probability $Pr[r(x) = 0]$, where $r(x) = e(x) \pmod{g(x)}$. In order to find the residue $r(x)$, we express $e(x)$ in terms of degree $(k-1)$ polynomials $\hat{e}(x)$. The number of term in output polynomial $C(x)$ is $2m-1$, which may not be a multiple of k. Let $\ell = \lceil (2m-1)/k \rceil$ and $\hat{m} = \ell k$. The error polynomial $e(x)$ be expressed as $e(x) = \sum_{i=0}^{\hat{m}-1} e_i x^i$, where $e_i \in GF(2)$ and $e_i = 0$ for $2m \leq i < \hat{m}$. The degree $\hat{m}-1$ polynomial $e(x)$ can be expressed as

$$\begin{aligned} e(x) = {} & x^0(e_0 + e_1 x + \ldots + e_{(k-1)}x^{k-1}) \\ & + x^k(e_k + e_{k+1}x + \ldots + e_{(2k-1)}x^{k-1}) \\ & + x^{2k}(e_{2k} + e_{2k+1}x + \ldots + e_{(3\ell-1)}x^{k-1}) \\ & \vdots \\ & + x^{(\ell-1)k}(e_{(\ell-1)k} + e_{(\ell-1)k+1}x + \ldots + e_{\ell k-1}x^{k-1}), \end{aligned}$$

or

$$e(x) = \sum_{i=0}^{\ell-1} x^{ik}\hat{e}^{(i)}(x), \text{ where } \quad \hat{e}^{(i)}(x) = \sum_{j=0}^{k-1} e_{(ik+j)}x^j.$$

Let $r(x) = \sum_{j=0}^{k-1} r_j x^j = e(x) \pmod{x^k+1}$. Then, using preceding we get

$$r(x) = \sum_{i=0}^{\ell-1} \hat{e}^{(i)}(x) = \sum_{j=0}^{k-1} \left(\sum_{i=0}^{\ell-1} e_{(ik+j)} \right) x^j = \sum_{j=0}^{k-1} r_j x^j,$$

since $x^k = 1 \pmod{x^k + 1}$. In order to have $r(x) = 0 \, \forall x$, we need to have

$$r_j = \sum_{i=0}^{\ell-1} e_{(ik+j)} = 0 \pmod 2, \;\; \forall \, 0 < j < k-1.$$

The coefficient r_j in the above equation is zero if the number of non-zero terms $e_{(ik+j)}$ is even. If there are no non-zero terms $e_{(ik+j)}$, there is no error at the output.

3.6.2 Probability of error detection

The error detection logic (EDL) outputs either "Error" or "No-Error". Let p and $q = 1 - p$ be the maximum probability of one bit error and no bit error at the output $C(x)$, respectively (not to be confused with q in $GF(q^m)$ of previous sections, which is a prime number). The probability of having exactly s number of ones in the r_jth bit of ℓ coefficients of $\hat{e}^{(i)}(x)$'s is,

$$\pi_\ell(s) = \binom{\ell}{s} p^s q^{\ell-s}.$$

The probability of a r_j's being zero is,

$$\begin{aligned} F_j &= Pr[r_j = 0] \\ &= \pi_\ell(0) + \pi_\ell(2) + \ldots + \pi_\ell(2\lfloor \ell/2 \rfloor) = \sum_{s=0}^{\lfloor \ell/2 \rfloor} \pi_\ell(2s). \end{aligned}$$

Using polynomial theorem we have,

$$(p+q)^\ell = \sum_{s=0}^{\ell-1} \binom{n}{s} p^s q^{\ell-s}$$

and

$$(p-q)^\ell = \sum_{v=0}^{\ell-1} \binom{n}{s} (-1)^s p^s q^{\ell-s}.$$

Adding above equations, the odd powers of p can be eliminated. Thus, we get

$$\begin{aligned} (p+q)^\ell + (q-p)^\ell &= 2 \sum_{s=0}^{\lfloor \ell/2 \rfloor} \binom{\ell}{s} p^{2s} q^{\ell-2s} \\ &= 2F_j = 1 + (q-p)^\ell. \end{aligned}$$

Let $q = 1 - p$ in the preceding equation, we get

$$F_j = [1 + (1 - 2p)^{\ell}]/2.$$

Since we have k coefficients in each polynomial $\hat{e}^{(i)}(x)$,

$$P[\text{No-Error}] = \Pi_{j=1}^{k-1} F_j = F_j^k$$

is the absolute probability of $r(x) = 0$, which is the absolute probability of the EDL returning No-Error. With probability of $(1 - p)^{(2m-1)}$, there is no error at the output. That is, coefficients of $e(x)$ are all zeros. Therefore, $\hat{F} = P[\text{No-Error}] - (1 - p)^{(2m-1)}$ is the absolute probability of EDL returning No-Error, while there are errors at the output. The $\hat{F}$ is the absolute probability of EDL not being able to detect an error. Probability of undetected error in EDL is the probability of not being able to detect and error, given there is error at the output. Thus, the probability of undetected error in EDL is

$$\begin{aligned} F_c &= \frac{F_j^k - (1 - p)^{(2m-1)}}{1 - (1 - p)^{(2m-1)}} \\ &= \frac{2^{-k}[1 + (1 - 2p)^{\ell}]^k - (1 - p)^{(2m-1)}}{1 - (1 - p)^{(2m-1)}} \end{aligned} \tag{3.26}$$

For large error probability p and $p < 0.5$, we have $F \approx 2^{-k}$. This is the key factor in keeping the probability of undetected errors small. Using a degree k polynomial for error detection, the upper bound for the probability of undetected error for large p is 2^{-k}. The probability of undetected error decreases exponentially with the increase of the number of protection bits.

3.6.3 Error Detection in N Redundant Reduction Modules

If all N redundant modules produce exactly the same erroneous output, the error is not detectable. Let $D^{(i)}(x)$ and $\widetilde{D}^{(i)}(x)$ denote the correct and erroneous output of the ith reduction module, respectively and the error signal at the ith redundant module be defined as $e^{(i)}(x) = \widetilde{D}^{(i)}(x) + D^{(i)}(x)$. The error is undetectable if we have $e^{(1)}(x) = e^{(2)}(x) = \ldots =$

$e^{(N)}(x)$ and $e^{(i)}(x) \neq 0, \forall\ 1 \leq i \leq N$. Let $e_j^{(i)}$ be the jth coefficient of $e_j^{(i)}(x)$, the probability of all error signals be equal in the jth bit is

$$\begin{aligned} F_j &= Pr\left[e_j^{(1)} = e_j^{(2)} = \ldots = e_j^{(N)}\right] \\ &= Pr\left[e_j^{(i)} = \ldots = e_j^{(N)}(x) = 1\right] \\ &\quad + Pr\left[e_j^{(i)} = \ldots = e_j^{(N)}(x) = 0\right] \\ &= p^N + q^N. \end{aligned}$$

All error signals are equal if their corresponding bits are equal. Thus, $Pr\left[e^{(1)}(x) = \ldots = e^{(N)}(x)\right] = \Pi_{j=1}^{k-1} P_j = (p^N + q^N)^m$. The probability of all zero should be excluded, since there would be no error if error signals are zero. Thus, the probability of undetected error is

$$F_{NMR} = (p^N + q^N)^m - q^{Nm}. \tag{3.27}$$

Assuming random error model, the m-bit error signals $e^{(i)}(x)$, for $0 \leq i < N$, can be viewed as N sequences of m random bits, where the probability of one and zero are p and $q = 1 - p$, respectively. An error is undetectable if all N random sequences are equal. For $m = 163, N = 3$ and $p = 1/2$, the problem is analogue to tossing coins 163 times by three people and having the same outcome for all of them. Using above equation, The probability of undetected error for such set up is about 7×10^{-99}.

Using(3.26) and (3.27), the probability of undetected error for polynomial bases multipliers and the NMR reduction modules are F_c and F_{NMR}, respectively. For the bit-serial multiplier, the probability of undetected error at the output is F_c.

3.7 Conclusion

We have generalized the concept of simple parity bit by using parity generation polynomials, which allows the definition of any number of parity bits as well as overlapped parity to protect field multiplication. The proposed parity scheme provides systematic analysis and design of field multipliers with CED. Architectures for CED in sequential and parallel multipliers has

been proposed. The probability of undetected error for k number of parity bits for over $GF(2^m)$ has been derived. The hybrid scheme for concurrent error detection on polynomial bases is smaller and faster than previous schemes using single and multiple simple parity bits.

CHAPTER 4

Concurrent Error Detection for Type II Optimal Normal Basis Multipliers

Abstract A new parallel architecture for type II optimal normal basis (ONB) with concurrent error detection capability is developed. The proposed ONB multiplier itself is based on polynomial multiplication, but is agnostic to the multiplication algorithm and can utilize digit-serial or sub-quadratic algorithms. It has the same area and approximately the same delay as the parallel Massey-Omura multiplier, if implemented with the conventional multiplication algorithm. In addition, it can utilize multiple parity generation polynomials to generate overlapped parity for error detection and keeps the error detection logic separate from the multiplier. Implemented on FPGA Over $GF(2^{173})$ with 17 parity bits, the area overhead is 7% and no delay is introduced in the multiplier output, with respect to the unprotected multiplier.

4.1 Introduction

Finite fields can be represented using different type of bases. Namely, polynomial basis, dual basis and normal basis (NB). Normal basis has the advantage that squaring can be accomplished by a simple shift operation, but the multiplication operation is more complicated compared to polynomial basis. A parallel-in serial-out NB multiplier architecture for multipliers was proposed by Massey and Omura in 1986 (MO) [OM86] and a variety of architectures have been proposed since then [WTS85, RH05, SK01] .

Single bit parity is a simple method to detect errors in data. Single-bit parity codes

add an extra bit to an operand such that the sum of the bits on the resulting codeword is one for odd parity and zero for even parity. The single parity bit has made its way to integer and finite field multipliers since long ago [LT70, FGB98]. Although single parity bit is simple to implement in data packets, error detection in multipliers using simple parity bits requires tracing the parity bits in the architecture. The resulting architecture often mixes the multiplier with the error detection logic and is hard to analyze and implement in a hardware description language (HDL). One of the advantages of the proposed multiplier is that it keeps the error detection logic separate from the multiplier.

In [CCL11, LMP10, CCL09, Lee10, Lee08], various architectures for CED of NB multiplication based on self-checking alternating logic, redundancy and single and multiple parity have been proposed. Single and multiple parity bit for concurrent error detection using polynomial basis over $GF(2^m)$ has been proposed in [RH03] and Chapter 3. In [FGB98], parity bits have been used for error detection of small field $GF(2^4)$, used in error control coding.

Here, we develop a multiplication scheme for type II optimal normal bases (ONB II), which is based on two regular polynomial multiplications over $GF(2)$ and a reduction step. Then, we use parity generation polynomials to detect errors in the polynomial multipliers and use NMR to detect errors in the reduction operation. We show that this scheme is efficient, has small area overhead and the resulting output is generated without extra delay.

In Section 4.2 a multiplication scheme for type II ONB is developed. In section 4.3, the error detection for the proposed multiplier is developed and its area and timing overhead is derived. The conclusion remarks follow.

4.2 Normal Basis Multiplication using Polynomials

In this section we use polynomial multiplication over $GF(2^m)$ to perform multiplication over normal basis. In later sections, we use parity polynomials to detect errors in the multiplier.

4.2.1 Preliminaries

If the elements of $\mathcal{M} = \{\alpha, \alpha^2, \alpha^{2^2}, \ldots, \alpha^{2^{m-1}}\}$ are linearly independent for some $\alpha \in GF(2^m)$, then $\mathcal{M}$ forms a normal basis in $GF(2^m)$. The element $\alpha \in GF(2^m)$ generates the normal basis and is called normal element. The concept and constructions of optimal normal basis (ONB) were introduced in [MOV89] to reduce the hardware complexity of field squaring. There are two special types of ONB, known as type I and type II, which minimize the complexity of the Massey-Omura multiplier. The following theorem from [Gao93] can be used to construct ONB-II.

Theorem 4.2.1 *Let $2m+1$ be a prime and assume that either*

(1) 2 is primitive in $\mathbb{Z}_{2m+1}$, or

(2) $2m+1 = 3 \pmod 4$ and 2 generate the quadratic residues in $\mathbb{Z}_{2m+1}$.

Then $\alpha = \gamma + \gamma^{-1}$ generates an optimal normal basis of $GF(2^m)$ over $GF(2)$, where $\gamma^{2m+1} = 1$.

It can be shown that if

$$\mathcal{M} = \{\gamma + \gamma^{-1}, \gamma^2 + \gamma^{-2}, \gamma^{2^2} + \gamma^{2^2}, \ldots, \gamma^{2^{(m-1)}} + \gamma^{2^{(m-1)}}\}$$

is a basis, then

$$\mathcal{N} = \{\gamma + \gamma^{-1}, \gamma^2 + \gamma^{-2}, \gamma^3 + \gamma^{-3}, \ldots \gamma^m + \gamma^{-m}\}$$

is also a basis and can be constructed from $\mathcal{M}$ by simple permutation [Gao93, SK01].

The symmetry in type II ONB has inspired many researchers to develop efficient multipliers [SK01, WHB02, RH05]. Permuted optimal normal basis is discussed and architectures for multiplication are proposed in [SK01, WHB02].

4.2.2 Architecture for ONB-II Multiplication

Let β_i be defined as

$$\beta_i \triangleq \gamma^i + \gamma^{-i}, \text{ where } -m \leq i \leq m. \tag{4.1}$$

It is easily verified that

$$\beta_i\beta_j = \beta_{i+j} + \beta_{i-j} \tag{4.2}$$

and

$$\beta_i = \beta_{-i}. \tag{4.3}$$

Over $GF(2^m)$ we have

$$\beta_0 = 2 \pmod 2 = 0. \tag{4.4}$$

Since $\gamma^{2m+1} = 1$, we get

$$\beta_m = \beta_{m+1}. \tag{4.5}$$

Lemma 4.2.1 *Considering above, we have*

$$\beta_{m+i} = \beta_{m+1-i}. \tag{4.6}$$

Proof 4.2.1 *We prove by induction. Using (4.5), it is verified that for $i = 0$, (4.6) holds. Assume (4.6) holds, we show $\beta_{m+i+1} = \beta_{m-i}$. Multiplying β_i on both side of (4.5), we get*

$$\beta_m\beta_i = \beta_{m+1}\beta_i.$$

Expanding preceding equation using (4.2) we get

$$\beta_{m+i} + \beta_{m-i} = \beta_{m+1+i} + \beta_{m+1-i}.$$

Subtracting both sides of (4.6) from preceding equation, we have $\beta_{m-i} = \beta_{m+i+1}$.

Let $\mathcal{N} = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be a Normal basis, where β_i is defined in (4.1). Thus, $\mathbf{a}$ and $\mathbf{b} \in GF(2^m)$ are represented using the normal basis as

$$\mathbf{a} = \sum_{i=1}^{m} a_i\beta_i,\ \mathbf{b} = \sum_{i=1}^{m} b_i\beta_i, \tag{4.7}$$

and by their coordinates as $\mathbf{a} = (a_1, a_2, \ldots, a_m)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_m)$, where $a_i, b_i \in GF(2)$. The *modular multiplication* of $\mathbf{ab}$ consists of *polynomial multiplication* $\mathbf{c} = \mathbf{a} \times \mathbf{b}$,

followed by modular reduction. We develop a scheme for the computation of $\mathbf{c}$ in bit parallel fashion. Using (4.7) we get

$$\begin{aligned}\mathbf{c} = \mathbf{a} \times \mathbf{b} &= (\sum_{i=1}^{m} a_i \beta_i)(\sum_{i=1}^{m} b_i \beta_i) \\ &= \sum_{i=1}^{m} \sum_{i=1}^{m} a_i b_j \beta_i \beta_j.\end{aligned}$$

Plugging (4.2), we get

$$\begin{aligned}\mathbf{c} &= \sum_{i=1}^{m} \sum_{i=1}^{m} a_i b_j (\beta_{i+j} + \beta_{i-j}) \\ &= \underbrace{\sum_{i=1}^{m} \sum_{j=1}^{m} a_i b_j \beta_{i+j}}_{\mathbf{c}^+} + \underbrace{\sum_{i=1}^{m} \sum_{j=1}^{m} a_i b_j \beta_{i-j}}_{\mathbf{c}^-}. \end{aligned} \tag{4.8}$$

Lemma 4.2.2 *Let the coordinates of* $\mathbf{a}$ *and* $\mathbf{b}$ *be the coefficients of two degree* $m-1$ *polynomials* $\mathbf{a}(x) = a_1 + a_2 x + \ldots + a_m x^{m-1}$ *and* $\mathbf{b}(x) = b_1 + b_2 x + \ldots + b_m x^{m-1}$. *The coefficients of the degree* $2m-2$ *polynomial* $\mathbf{a}(x) \times \mathbf{b}(x)$ *are the coordinates of* $\mathbf{c}^+$ *in (4.8).*

Proof 4.2.2 *It is well known that the k-the coefficient of* $\mathbf{a}(x) \times \mathbf{b}(x)$ *is* $\sum_{i+j=k} a_i b_j$, $1 \leq i, j \leq m$. *From (4.8), we have*

$$\mathbf{c}^+ = \sum_{i=1}^{m} \sum_{j=1}^{m} a_i b_j \beta_{i+j} = \sum_{k=2}^{2m} c_k^+ \beta_k, \tag{4.9}$$

$$\text{where } c_k^+ = \sum_{i+j=k} a_i b_j \text{ , and } 1 \leq i, j \leq m$$

Lemma 4.2.3 *Let the coordinates of* $\hat{\mathbf{b}} = (b_m, b_{m-1}, \ldots, b_1)$ *and* $\mathbf{a} = (a_1, a_2, \ldots, a_m)$ *be the coefficients of two degree* $m-1$ *polynomials* $\hat{\mathbf{b}}(x) = b_m + b_{m-1} x + \ldots + b_1 x^{m-1}$ *and* $\mathbf{a}(x) = a_1 + a_2 x + \ldots + a_m x^{m-1}$. *The coefficients of the degree* $2m-2$ *polynomial* $\mathbf{a}(x) \times \hat{\mathbf{b}}(x)$ *give the coordinates of* $\mathbf{c}^-$ *in (4.8).*

Proof 4.2.3 *We have*

$$\begin{aligned}\mathbf{a}(x) \times \hat{\mathbf{b}}(x) &= \sum_{i=1}^{m} a_i x^{i-1} \sum_{j=1}^{m} b_{m-j+1} x^{j-1} \\ &= \sum_{i=1}^{m} \sum_{j=1}^{m} a_i b_{m-j+1} x^{i+j-2} \\ &= \sum_{k=0}^{2m-2} \hat{c}_k x^k = \sum_{k=i-j+m+1} a_i b_j x^k \\ &= x^{m+1} \sum_{k=i-j} a_i b_j x^k, \ \textit{where} \ 1 \leq i, j \leq m. \end{aligned} \tag{4.10}$$

From (4.8) we get

$$\begin{aligned}\mathbf{c}^- &= \sum_{i=1}^{m} \sum_{j=1}^{m} a_i b_j \beta_{i-j} \\ &= \sum_{k=-(m-1)}^{m-1} c_k^- \beta_k \end{aligned} \tag{4.11}$$

$$= \sum_{k=i-j} a_i b_j \beta_k, \ \textit{where} \ 1 \leq i, j \leq m. \tag{4.12}$$

Comparing (4.10) and (4.12), it follows that the coefficients of $\hat{\mathbf{a}}(x) \times \mathbf{b}(x)$ *are the coordinates of* $\mathbf{c}^-$.

In order to compute the modular multiplication $\mathbf{z} = \mathbf{ab}$, the $\mathbf{c}^-$ and $\mathbf{c}^+$ terms in (4.8) must be reduced and added together.

Theorem 4.2.2 *Let* $\mathbf{a}$ *and* $\mathbf{b}$ *be represented with respect to the normal basis* $\mathcal{N}$. *Computation of* modular multiplication *of* $\mathbf{z} = \mathbf{ab}$ *over* $GF(2^m)$ *requires two m-term polynomial multiplications, a correction of* $\mathbf{c}^-$ *and a reduction of* $\mathbf{c}^+$.

Proof 4.2.4 *Using Lemmas 4.2.2 and 4.2.3 it can be shown that* $\mathbf{c}^+$ *and* $\mathbf{c}^-$ *can be computed with two m-term polynomial multiplication* $\mathbf{a}(x) \times \mathbf{b}(x)$ *and* $\mathbf{a}(x) \times \hat{\mathbf{b}}(x)$ *over* $GF(2)$, *respectively.*
Correction of $\mathbf{c}^-$*: Plugging (4.3) and (4.4) into* $\mathbf{c}^-$ *in (4.12), we get* $\mathbf{z}^- = \sum_{i=1}^{m-1} (c_i^- + c_{-i}^-) \beta_i$.

Reduction of $\mathbf{c}^+$*: Considering (4.2) and (4.5) , and by mathematical induction, it can be shown that*

$$\beta_{m+i} = \beta_{m+1-i}, \text{ where } 0 \leq i \leq m. \tag{4.13}$$

Plugging (4.13) into $\mathbf{c}^+$ *in (4.9), we get*

$$\mathbf{z}^+ = c^+_{2m}\beta_1 + \sum_{i=2}^{m}(c^+_i + c^+_{2m+1-i})\beta_i.$$

The final result is

$$\begin{aligned}\mathbf{z} &= \mathbf{z}^+ + \mathbf{z}^- \\ &= (c^+_m + c^+_{m+1})\beta_m \\ &+ \left(\sum_{i=2}^{m-1}(c^+_i + c^+_{2m+1-i} + c^-_i + c^-_{-i})\beta_i\right) \\ &+ (c^-_1 + c^+_1)\beta_1. \end{aligned} \tag{4.14}$$

The reduction and correction steps can be performed separately or combined and is referred to as reduction hereafter. The area and delay complexities of ONB-II reduction can be computed using (4.14), as shown in Table 4.1.

4.3 Concurrent Error Detection

A concurrent error detection scheme for multiplication over $GF(2^m)$ using polynomials has been proposed in Chapter 3. In this section we explain how to apply this scheme to the type II ONB and derive complexity equations.

4.3.1 Parity Definition

Let $A = A(x) = \sum_{i=0}^{m-1} a_i x^i$, where $a_i \in GF(2)$, be a polynomial in $GF(2)[x]$ and $g(x) \in GF(2)[x]$ be a degree $(k-1) < m$ polynomial. The data check, or the parity of A is defined as

$$p_A = P(A) \triangleq A \pmod{g(x)} \tag{4.15}$$

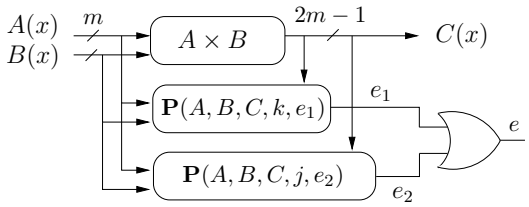


Figure 4.1: A polynomial multiplier with bi-residue parity protection and $j + k$ parity bits (PPM).

and $g(x)$ is called the parity generation polynomial. Assume $A, B \in GF(2)[x]$ and

$$C(x) = A(x) \times B(x). \tag{4.16}$$

Then the predicted parity of C is,

$$\begin{aligned} p_C &= [A(x) \times B(x)] \pmod{g(x)} \\ &= [A(x) \pmod{g(x)} \times B(x) \pmod{g(x)}] \\ &\quad \pmod{g(x)} \\ &= (p_A \times p_B) \pmod{g(x)}. \end{aligned} \tag{4.17}$$

The parity polynomial $g(x)$ is of degree $k - 1$, which generates k parity bits (Chapter 3). To reduce the complexity of parity generation logic, we use $g(x) = x^k + 1$ as the parity generation polynomial. We define the parity protection operation $\mathbf{P}(A, B, C, k, e)$ as follows:

1. Take A, B, C, k as inputs.
2. Compute the parity of C by using the parity definition (4.15).
3. Predict the parity of $A \times B$ using (4.17).
4. Compare outputs of step 2 and 3 and generate the error bit e.

4.3.2 Architecture

If the error vector is a multiple of the parity generation polynomial $g(x)$, it becomes undetectable. Multiple parity generation polynomials generate overlapping parity bits and thus

increase the probability of error detection. In overlapping parity, each data bit belongs to more than one parity group. If one parity group does not detect the error, there is a high probability that the error is detected by other parity group(s). The architecture in Fig. 4.1, hereafter called PPM, depicts a polynomial multiplier protected by two parity generation polynomials $g(x)$ and $h(x)$, of degree k and j, respectively. To have simpler reduction operation, we use binomials for parity generation, but in principle any polynomial can be used.

Construction of the ONB-II multiplier using polynomial generation multiplication over $GF(2)$ allows us to use parity polynomials for error detection. Two polynomial multipliers along with their parity protection logic are used to produce $\mathbf{c}^+$ (4.9) and $\mathbf{c}^-$ (4.12), and detect errors. They are shown with double-lined boxes in Fig. 4.2 and detailed in Fig. 4.1.

Table 4.1 shows gate count and gate delay of (4.15), (4.16) and (4.17) and modules in Fig. 4.1 and 4.2. Each module is referenced by its number. T_A and T_X are gate delay of two input AND and XOR gates, respectively. We have also assumed delay and area of OR gate is equal to those of the AND gate. Area and delay of the module i are referred to as S_i and T_i, respectively. The information provided in [EYK06] can be used to obtain complexities in Table 4.1.

Considering the space complexity of polynomial multiplication S_1 and the reduction logic S_7, we have

$$\frac{2S_1}{S_7} = \frac{2m^2S_A + 2 \times [(m-1)^2S_X]}{(3m-2)S_X} \approx \frac{4m}{3} \gg 1,$$

where we have assumed m is large and $S_A \approx S_X$. Since $m > 160$ in cryptography, the area of the polynomial multiplier is much larger than that of reduction logic. Therefore, we may be able to use modular redundancy to protect the reduction logic without having large area overhead. The idea has been depicted in Fig. 4.2, where N modular redundancy has been used to protect the reduction operation (4.14) against errors.

Table 4.1: Delay and gate count for modules in Fig. 4.1 and 4.2

Module No.	**Functionality**	**Area** ($k \geq 1$)	**Delay**
Fig. 4.1			
1	$A \times B$ KOA, $m = 2^t$	$m^2 S_A + (m-1)^2 S_X$ $m^{\log_2 3} S_A + (6m^{\log_2 3} - 8m + 2) S_X$	$T_A + \lceil \log_2 m \rceil T_X$ $T_A + 2T_X \log_2 m$
2	(mod $x^k + 1$) , width m	$(m-k)S_X$	$\lceil \log_2(m/k) \rceil T_X$
3	(mod $x^k + 1$), width $2k - 1$	$(k-1)S_X$	T_X
4	(mod $x^k + 1$), width $2m - 1$	$(2m - 1 - k)S_X$	$\lceil \log_2(2m-1)/k \rceil T_X$
5	$p_A \times p_B$	$k^2 S_A + (k-1)^2 S_X$	$T_A + \lceil \log_2 k \rceil T_X$
6	comparing actual and predicted parity and generating e	$kS_X + (k-1)S_A$	$T_X + \lceil \log_2 k \lceil T_A$
Fig. 4.2			
7	correction of $\mathbf{c}^-$ and reduction of $\mathbf{c}^+$	$(3m-2)S_X$	$2T_X$
8	comparing output of redundant modules	$(2mN-1)S_A$	$(1 + \lceil \log_2 N \rceil + \lceil \log_2 m \rceil)T_A$
9	final error signal generator	$3S_A$	$2T_A$

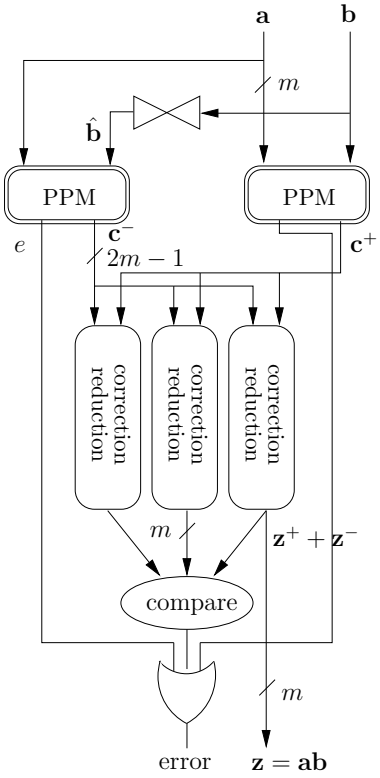


Figure 4.2: Parallel ONB-II multiplier with CED

4.3.3 Complexities and Overhead

For the polynomial multiplication of $\mathbf{a}(x) \times \mathbf{b}(x)$, first the products $a_i b_j$, where $0 \leq i, j \leq m-1$, needs to be generated using m^2 AND gates. Then, the partial products are summed up properly, using $(m-1)^2$ XOR gates, to produce the resulting polynomial. The two products $\mathbf{a}(x) \times \hat{\mathbf{b}}(x)$ and $\mathbf{a}(x) \times \mathbf{b}(x)$ can share the same AND gates, because the partial products are the same in both cases. Therefore, two separate polynomial multiplier in Fig. 4.2, can be partially merged to share the m^2 AND gates. For clarity in the figure, we have shown these multipliers separately.

The area overhead of the parity protected multiplier in Fig. 4.2 equals the area of the parity generation and error detection logic divided by the area of the original parallel multiplier. Thus, considering N modular redundancy for reduction and using Fig 4.1 and 4.2 and Table 4.1, the area of $\mathbf{P}(A, B, C, k, e)$ is

$$S_{\mathbf{P}}(k) = 2S_2(k) + \sum_{i=3}^{6} S_i(k),$$

where the parameter k is the number of parity bits. The two operation $\mathbf{P}(A, B, C, k, e_1)$ and $\mathbf{P}(A, \hat{B}, C, j, e_2)$ in Fig. 4.1 can share generation of p_A and save $S_2(k)$ gates.

Assuming two parity polynomial $g(x)$ and $h(x)$ of degree k and j, respectively, the area overhead is

$$\begin{aligned} R_O = & \frac{2\left(S_{\mathbf{P}}(k) + S_{\mathbf{P}}(j)\right) + \left((N-1)S_7 + S_8 + S_9\right)}{2S_1 + S_7 - m^2 S_A} \\ & - \frac{S_2(k) + S_2(j)}{2S_1 + S_7 - m^2 S_A} \end{aligned}$$

On FPGA, where $S_X = S_A$, and using triple modular redundancy, we have

$$\begin{aligned} R_O(m,k) &= \frac{4(k^2+j^2) - 3(k-j) - 11 + 26m}{3m^2 - m} \\ &\approx \frac{4k^2 + 4j^2}{3m^2} + \frac{26}{3m} \\ &< 6\% \text{ for } k + j \leq 0.1m \text{ and } m \geq 160 \end{aligned} \tag{4.18}$$

Extending preceding results to multiple parity generation polynomials is straightforward. Most of the area in the ONB-II multiplier is occupied by two polynomial multipliers, which consists of approximately m^2 AND gates and $2m^2$ XOR gates. Similarly, each of the four polynomial multipliers for error detection consist of m^2 AND gates and m^2 XOR gates. This explains the $4(k^2+j^2)/3m^2$ term in (4.18). The area and area overhead can be improved as it is discussed Section 4.3.4.

4.3.3.1 Timing Overhead

Adding the error detection logic does not add any extra delay to the output. The error signal, is generated by the N modular redundancy comparator module 8 and the final error generation logic module 9. It is generated after the output and has extra delay of $\mathrm{O}(\log_2 mNk)$. Considering Table 4.1, Fig. 4.1 and Fig. 4.2, the output delay is

$$T_{\text{out}} = T_1 + T_7 = T_A + (2 + \lceil \log_2 m \rceil) T_X.$$

The delay of the error signal is

$$T_{\text{err}} = T_{\text{out}} + T_8 + T_9.$$

The error signal is generated independent of the output but its delay is larger than that of the output. Since in practice many multiplications are performed, the multiplier output may be used in processing and the error signal can be taken into account after it is ready. It may also be more convenient to generate the error signal one clock after the output. This approach allows operating at full clock speed.

4.3.4 Optimizing the Architecture

The architecture in Fig. 4.2 uses polynomial multipliers for both $A \times B$ and $p_A \times p_B$ operations, without specifying a particular multiplication algorithm.

4.3.4.1 multiplication algorithm

Sub-quadratic, bit serial, or digit serial polynomial multipliers may be used to decrease the area and trade off area/time in both polynomial multipliers. For example, $p_A \times p_B$ is a small multiplier compared to the $A \times B$ and it is not in the critical path, therefore digit serial multiplier can be used to for $p_A \times p_B$ to further decrease the area overhead. Similarly, If the main multiplier $A \times B$ is too large for the target device, a digit serial multiplier can be used.

For $m = 2^t$, where $t \in \mathbb{N}$, the gate count of the recursive Karatsuba-Ofman algorithm (KOA) multiplier is

$$S_1 = m^{\log_2 3} S_A + (6m^{\log_2 3} - 8m + 2)S_X. \tag{4.19}$$

In cryptography, m is not a power of two and over $GF(2^m)$ it may not be efficient to perform KOA recursively to the last level. Therefore the preceding equation does not precisely model general application of KOA, but we use it for estimation purpose. Application of KOA and KOA-like algorithms to general degree m polynomials can be found in [Mon05] [WP06].

The the gate count and delay of parallel MO multiplier are

$$S_{MO} = 2m(m-1)T_X + m^2 T_A$$

and

$$T_{MO} = T_A + (1 + \lceil \log_2(m-1) \rceil)T_X,$$

respectively. The complexity of the proposed multiplier matches those of the MO multiplier. The multiplier in [SK01] *does not* provide CED and has the gate count

$$S = m^2 S_A + 1.5m(m-1)S_X. \tag{4.20}$$

Assuming $S_A = S_X$ in (4.19) and (4.20), it can be verified that for $m > 50$ we have $(2.5m^2 - 1.5m) > 2 \times (7m^{\log_2 3} - 8m)$. Therefore, proposed multiplier will be more area efficient than [SK01], if KOA is used for the $A \times B$ operation. Considering (4.20), S_1 and T_1, it is expected that for $m = 173$ the proposed multiplier be 60% more area efficient but 1.8 times slower than [SK01], although [SK01] does not provide CED. Therefore, the proposed multiplier without CED, is as efficient or better than [SK01] and MO [OM86] multipliers.

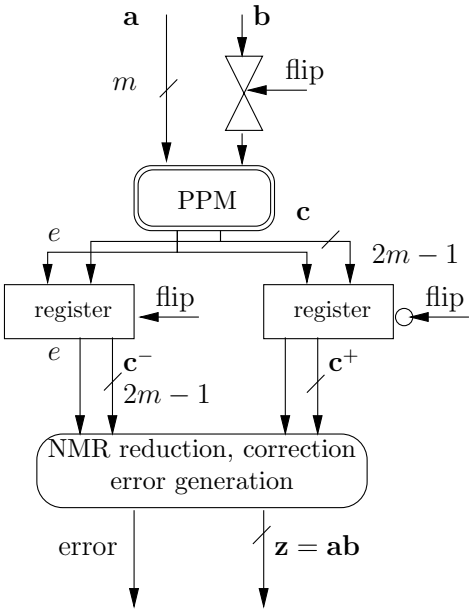


Figure 4.3: Parallel ONB-II with CED, using a single polynomial multiplier.

4.3.4.2 PPM multiplexing

Considering (4.9) and (4.12), a single PPM can be used to generate both $\mathbf{c}^+$ and $\mathbf{c}^-$. The idea has been shown in Fig 4.3, where the *flip* signal is used to produce $\hat{\mathbf{b}}(x)$ from $\mathbf{b}(x)$ and latch the output of PPM to the proper register, producing $\mathbf{c}^+$ and $\mathbf{c}^-$ in two cycles. The reduction and correction operations are protected by NMR as before. This arrangement reduces the area by almost a factor of two and doubles the output latency.

4.3.5 Error Model and Probability of Error Detection

The error model and probability of error detection for polynomial multipliers, using a single parity generation polynomial, and NMRs have been discussed in chapter 3.

Let the probability of error in a bit be p, polynomial $g(x) = x^k + 1$ be used for error detection , $\ell = \lceil (2m-1)/k \rceil$ and error bits be independent. It has been shown in Chapter 3 that the probability of undetected error in the output of a polynomial multiplication over

$GF(2^m)$ is

$$F(m,k) = \frac{2^{-k}[1+(1-2p)^{\ell}]^k - (1-p)^{(2m-1)}}{1-(1-p)^{(2m-1)}}. \tag{4.21}$$

Since we have two independent polynomial multipliers, the probability of undetected error is $F^2(m,k)$.

Consider $GF(2^{173})$, where there exists an ONB-II [Gao93]. Using (4.18) and (4.21), and for $k=8$ and $p=0.4$, the area overhead and the probability of undetected error are 4% and 0.008, respectively. Using 20 parity bits, the overhead and probability of undetected error are 5% and 10^{-6}, respectively. Since the overhead does not increase rapidly, large number of parity bits can be used for better error coverage.

The error in the N redundant modules are not detectable, if all modules produce exactly the same erroneous output. Assume $C^{(i)}(x)$ and $\widetilde{C}^{(i)}(x)$ are the correct and erroneous output of the ith reduction module, respectively. The error signal at the ith redundant module is defined as $e^{(i)}(x) = \widetilde{D}^{(i)}(x) + D^{(i)}(x)$. The error is undetectable if we have $e^{(1)}(x) = e^{(2)}(x) = \ldots = e^{(N)}(x)$ and $e^{(i)}(x) \neq 0, \forall\ 1 \leq i \leq N$. It can be shown in Chapter 3 that the probability of undetected error for the NMR is

$$F_{NMR} = (p^N + q^N)^m - q^{Nm},$$

where $q = 1-p$ in Chapter 3. For $m = 173, p = 0.4$, eight parity bits and triple modular redundancy, probability of undetected error is $F_{NMR} = 3 \times 10^{-146}$. Therefore, probability of undetected error is mostly dominated by the polynomial multiplication operation, rather than the reduction.

4.3.6 Comparison

Various ONB multipliers are listed in Table 4.2. The architecture in [CCL11] is a parallel type t NB multiplier that uses self checking alternating logic for CED. It uses three input and m input gates as well as NAND gates. The data reported in Table 4.2 is for $t=2$. Gate count and delays have been converted to the two input gate equivalent and NAND gates are

counted as AND gates. Using Table 4.2 and Table 6 in [CCL11], it can be concluded that for $t = 2$ [CCL11] has approximately $24m^2/3m^2 = 800\%$ area and $3\log_2 m/\log_2 m = 300\%$ timing overhead, with respect to the MO multiplier. Since proposed multiplier has almost the same area as MO multiplier, it can be used in an 8 NMR configuration and still operate faster than [CCL11], for $t = 2$.

Compared to the semi-systolic design [CCL09], the proposed architecture has smaller area and delay. Although [CCL09] has linear overhead, the area and latency of the original multiplier is large. In addition the delay of the error signal is larger than that of the propose architecture, since in cryptography it is very likely that $Nk < m$. For example, for triple modular redundance and 20 parity bits we have $Nk = 60$, which is smaller than m's used in cryptography.

As shown in (4.18), the area overhead of the multiplier with error detection is estimated to stay below 6% , if $k < 0.1m$ and $m > 160$, which is mostly the case for cryptographic purposes. Considering Table 4.1, the area of the proposed multiplier is almost equal to that of parallel MO multiplier in [OM86] and its gate delay is one XOR gate larger. The extra one XOR gate is the side effect of separating polynomial multiplication and reduction. The same effect can be observed in polynomial basis multipliers [Wu02]. As it has been discussed earlier if smaller area compared to MO is desired, digit serial or sub-quadratic polynomial multiplication algorithms can be used to produce $A \times B$.

4.3.7 Implementation

Table 4.3 shows the FPGA implementation of the architecture in Fig. 4.2 over $GF(2^{173})$, where polynomial multiplications $A \times B$ are performed by setting $m = 11 \times 2^4$ and using KOA algorithm. We have used Xilinx Virtex4-XC4VLX2000-10 FPGA. Synthesis and place-and-route are performed using Synplify Premier and Xilinx tools, respectively. It is verified from the results that number of parity bits does not slow down the multiplication operation. As it is shown, the overhead of the architecture stays low for large number of parity bits.

Table 4.2: Various ONB II multipliers with concurrent error detection

Multiplier	**[LMP10]**	**[CCL09]**	**[CCL11] for $t = 2$**	**here**
Multiplier				
AND	m	$2m^2 + 1$	$5(2m + 1)^2 + (m - 1)$	m^2
XOR	$2m$	$2m^2 + 5m + 1$	$2m(2m + 1)$	$2m^2 - 2m$
Latch	$2m$	$5m^2 + 2m + 2$	$3m + (2m + 1)$	$3m$
Latency	m	$2 \times (m + 3)$	1	1
Critical path	$T_A + 2T_X$	$T_A + T_X$	$(2\lceil \log_2(2m + 1) \rceil + 1)T_X$ $+(4 + \lceil \log_2 m \rceil)T_A$	$T_A + (2 + \log_2 m)T_X$
Architecture	bit serial	semi systolic	parallel	parallel
Error Detection Overhead				
AND	k	$\approx 13m + 6$	included in the multiplier	$2k^2 + 2k - 1 + 6m$
XOR	$m + 5k$			$2k^2 + 6k - 6 + 14m$
Error signal delay	$T_X \log_2 m$	$T_X +$ $T_A \log_2(2m + 1)$	no delay	$(1 + \lceil \log_2 N \rceil + \lceil \log_2 m \rceil$ $+\lceil \log_2(2k + 1) \rceil)T_A$
Error detection method	multiple parity	time and area redundancy	self checking alternating logic	parity generation polynomial and NMR
Area overhead	3 parity bits: 27%	57%	$\approx 800\%$	24 parity bits: $< 6\%$
Time overhead over $GF(2^{233})$	57%	1.9%	$\approx 300\%$ with respect to MO	0%

Table 4.3: FPGA implementation over $GF(2^{173})$ using KOA for $g(x) = x^k + 1$

Parity Bits k	**LUTs**	**Delay (ns)**	**Area Overhead %**
23	20,173	7.0	11
17	19,475	7.1	7
7	19,180	7.1	5
3	18,932	7.2	4
0	18,216	6.9	0

The overhead in Table 4.3 is slightly larger than the predicted overhead in (4.18) because the implementation uses KOA for $A \times B$, which has smaller area than the multiplier considered in (4.18). In this implementation, one parity generation polynomial is used.

4.4 Conclusion

We have developed a multi-residue concurrent error detection scheme for type II ONB, where the number of parity bits are determined by a parity generation polynomials. The delay and area of the proposed multiplier are approximately equal to those of the parallel MO multiplier and there is no timing overhead for the output, if the conventional multiplication algorithms is used. In addition, the error detection logic is not intermixed with the multiplier. The space complexity of the multiplier can be reduced by using sub-quadratic multiplication algorithms. For cryptographic purpose, where $m > 160$ and number of parity bits $k < m/10$, the area overhead is estimated to be less than 6%.

CHAPTER 5

FO4-based Models for Area, Delay and Energy of Polynomial Multiplication over Binary Fields

Abstract Accurate mathematical models are crucial for design and evaluation of systems and architectures. To provide models that can accurately describe polynomial multipliers in $GF(2)[x]$, we integrate the traditional gate-counting method and the fanout four (FO4) model for deriving strength of gates. We show that the resulting model provides much more accurate estimates than the traditional model and can be obtained by straight-forward modification to the traditional models. Implementation results indicate that the conventional gate delay model has 40% error, while the presented model has less than 4% error. We also show that the ceiling function, conventionally used in delay equations, are not needed in the new model. Moreover, we give a model for the energy consumption of polynomial multiplication, which can be used to estimate power consumption of finite field multiplication. While it is believed that the Karatsuba-Ofman multiplier is three times slower than of polynomial multiplier, we show that it is actually two times slower. To evaluate the accuracy of our models, we provide ASIC implementation results and compare them to results obtained from models.

5.1 Introduction

Many different bit-parallel architectures for finite field multiplication over $GF(2^m)$ have been proposed in the literature [Wu02], [SK99], [RH04a]. For different architectures, authors usually express area and time complexities in terms of area and delay two-input logic gates.

While speed benefit of using different architecture is in the range of a few XOR gates, the question is how accurate are those complexity expressions. For example, if a designer tries to implement a finite field multiplier, what speed and area can he or she expect? Here, we try to bridge the gap between the mathematical models of polynomial multipliers and their hardware implementations. We show that the accuracy of current models can be improved by providing models that are easy to use and closer to reality.

Parallel finite field multiplication over binary extension fields can be performed by polynomial multiplication followed by modular reduction operation [Wu02]. The polynomial multiplication can be performed using multiplication algorithms with quadratic or sub-quadratic complexity [Mon05], [Sun04]. Other approaches include the Mastrovito multiplier and its derivatives [SK99], [RH04a]. All of these approaches have common elements such as polynomial multiplication over $GF(2)[x]$, input lines that drive many gates and binary XOR trees. Here, we discuss polynomial multiplication over $GF(2)[x]$ and try to cover those common elements. Our purpose is to provide models of polynomial multiplication, which can help estimating the actual costs of a finite field multiplier.

Currently, gate-counting method is used to express delay and area estimates of finite field multipliers. The FO4 model is a linear delay model, which is used in CMOS technology to give estimation of gate size and speed of logic circuits [RCN03]. By considering both models and integrating them, we obtained more accurate estimates. Furthermore, this approach enables us to also provide an energy model, which can be used to estimate the power consumption of a multiplier.

In deep sub-micron technology, delay of interconnects affects the overall delay of logic circuits. In general, estimation of the post-layout delay may not be precise, as it depends on various parameters such as the CMOS feature size, manufacturing process, design congestion and the length of interconnects. The delay model presented here, is meant to give precise estimation for the post-synthesis stage, compared to the traditional models currently used in the literature. In layout,the area of a design is increased by a constant factor of 10%to 20% to accommodate interconnects. Therefore, presented area model can be used for post-layout

as well.

Let finite field $GF(2^m)$ be defined by the irreducible polynomial $f(x)$. Let $\{x, x^2, \ldots, x^{n-1}\}$ be the polynomial basis and $A, B \in GF(2^m)$ be defined as $A = \sum_{i=0}^{m-1} a_i x^i$ and $B = \sum_{i-0}^{m-1} b_i x^i$, where $a_i, b_i \in F(2)$. The polynomial multiplication $A \times B$ is defined as

$$D = A \times B = \sum_{i=0}^{2m-2} d_i x^i, \tag{5.1}$$

where $d_i = \sum_{j+k=i} a_j b_k$. The product of A and B in $GF(2^m)$ is $E = D \pmod{f(x)}$. Multiplication of the two degree m polynomials in $GF(2)[x]$ in (5.1), requires a matrix of m^2 two-input logical AND operations and $(m-1)^2$ two-input logical XOR operations [Wu02], [EYK06]. Low weight irreducible polynomials are commonly used to reduce the complexity of modular reduction. In particular, trinomial or pentanomials can be used to generate the field. The reduction operation requires $(w-1)(m-1)$ two-input XOR gates, where w is the hamming weight of the irreducible polynomial $f(x)$ [Wu02], [EYK06]. In cryptography, where m is large and low-hamming-weight irreducible polynomials are used, the delay and area complexity of modular reduction is much less than those of polynomial multiplication [EYK06].

In Section 5.2, we give models for delay, area and energy of polynomial multipliers over $GF(2)[x]$ and compare the modes with our ASIC implementation results. In Section 5.3, we investigate the delay of the Karatsuba-Ofman multiplication algorithm (KOA), and in the last section, we provide the conclusion.

5.2 Analysis of Polynomial Multiplication

Fig. 5.1 shows the logic diagram of (5.1) for $m = 5$, where the critical path is highlighted. In implementation, depending on the size of polynomials and the underlying hardware technology, buffers might be required to drive wires and other capacitive loads. Moreover, gates with different sizes might be used by logic synthesizers. As shown in Fig. 5.1, each AND gate and each XOR gate drives only one XOR gate. Therefore, in both cases, a minimum

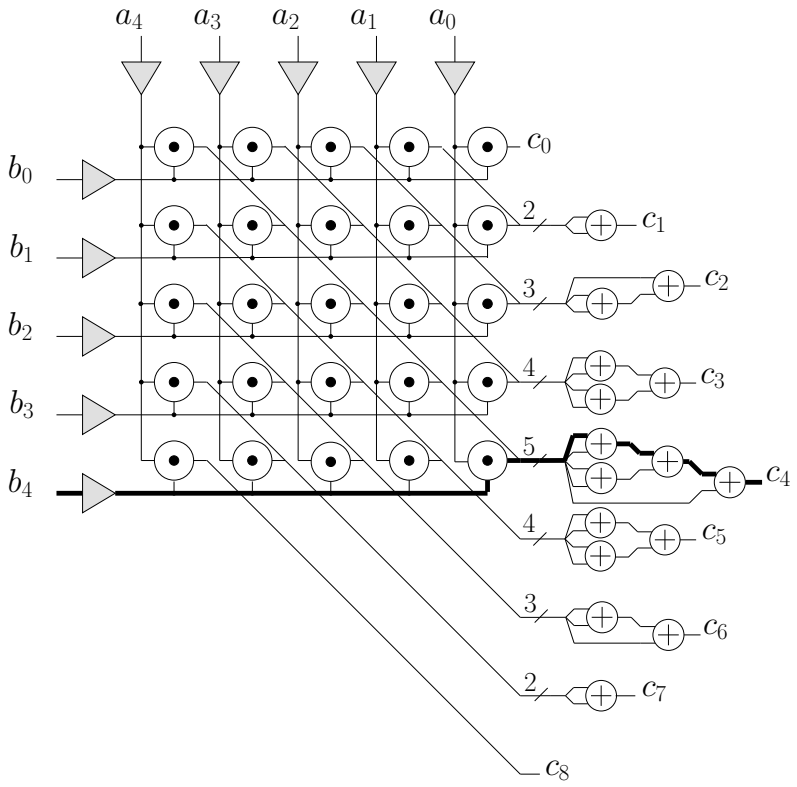


Figure 5.1: Functional diagram of the polynomial multiplication over $\mathbb{F}_2[x]$, where critical path is highlighted.

size gate would suffice. However, each input line a_i and b_i, where $0 \leq i < m$, drives m AND gates. In cryptography m can be very large; for example, in elliptic curve processors $m > 160$ and for crypto-systems based on the discrete logarithm problem $m > 1024$. The large capacitive load on input lines causes delay, and logic synthesizers insert sufficient number of buffers to drive the load and increase the speed. These buffers increase the overall speed, but they change the traditional delay and area models. In a traditional analysis of finite field multipliers over $GF(2^m)$, these buffers are ignored. The gate delay of the polynomial multiplier is considered to be:

$$T = T_{AND} + T_{XOR}\lceil \log_2 m \rceil. \tag{5.2}$$

Sometimes, for simplicity, the ceiling function is omitted and the equation is expressed as:

$$T = T_{AND} + T_{XOR} \log_2 m. \tag{5.3}$$

The area of the multiplier is expressed as:

$$S = m^2 S_{AND} + (m-1)^2 S_{XOR} \tag{5.4}$$

[Wu02], [SK99], [EYK06]. One reason for ignoring buffers could be that many multiplication architectures have been inherited from the coding theory, where m is small and, therefore, no buffers or a small number of buffers are required to drive the AND gate matrix. We show that to have a more precise model for the behavior of multiplication architectures over $GF(2)[x]$ and $GF(2^m)$, these buffers need to be taken into account, and we provide equations to model the effect of the buffers.

In CMOS technology, NAND gates are always smaller and faster than AND gates. An AND gate is made using a NAND gate cascaded by an inverter. Since inputs of an XOR gates can be inverted without affecting the output, in polynomial multiplication over $GF(2)[x]$, a NAND gates matrix should be used instead of an AND gate matrix. Therefore, it seems to be more appropriate to express the area as:

$$S = m^2 S_{NAND} + (m-1)^2 S_{XOR}. \tag{5.5}$$

We further discuss the accuracy of (5.4) and (5.5) in later sections. The effect of NAND gates on the delay is negligible, since there is only one NAND gate in the critical path but they do largely affect the area.

The effect of buffers is more prominent in the case of the m-term polynomial A multiplied to the n-term polynomial B, where $m > n$. This type of multiplication is common in digit-serial multipliers, where the digit size is $\lceil m/n \rceil$. In this case, the delay of the multiplier is considered to be:

$$T = T_{AND} + T_{XOR}\lceil \log_2 n \rceil. \tag{5.6}$$

In this type of multiplication, each b_i drives m NAND gates and, consequently, imposes a delay, while the delay expressed by (5.6) is only dependent to n and independent to m. The effect of ignoring buffers in this case, as it will be seen, would be large deviation of the expected delay from the actual delay. The extreme case is the bit-serial multiplier, where the delay is assumed to be $T_{AND} + T_{XOR}$. However, each input bit b_i drives m AND gates, which causes large delays.

The first step in our analysis is to verify that the above mentioned buffers really exist and NAND gates are used instead of AND gates in implementation. We have synthesized three multipliers using the Synposys Design Compiler and the CMOS 0.18μm library. Fig. 5.2 shows gates in the critical path of m-bit $\times$ n-bit multipliers, where $m \in \{16, 128, 256\}$ and $n = \frac{m}{4}$. The synthesizer uses NAND and NOR gates instead on AND or OR gates because the former are smaller and faster. The extra buffers (inverters) driving the NAND gates can be seen in Fig. 5.2 . The Dn suffix of gate names indicates the size of the gate.

5.2.1 The CMOS Model

In digital CMOS technology, a logic gate is characterized by its on-state resistance, gate input capacitance and intrinsics (diffusion) capacitance at the output. Interconnects are also modeled by their resistance and capacitance. Delay in a logic circuit is computed based on the above mentioned parameters. Let R_0 be the on-state resistance of a unit-

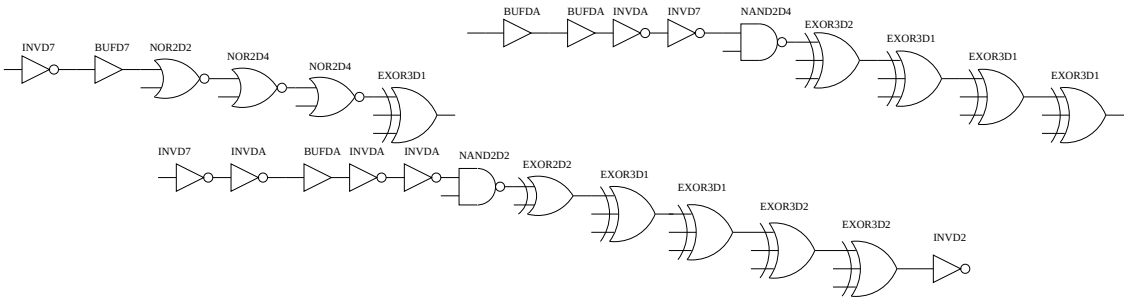


Figure 5.2: Gates in the critical path of 16×4, 128×32 and 256×64 bit multipliers, produced by the Synopsys Design Compiler synthesis tool.

Table 5.1: Estimates of Normalized Input and Intrinsics Capacitance of Logic Gates

Gate	**Input capacitance** $\overline{C}_{in}$	**Intrinsic capacitance** p
inverter	1	1
two-input NAND	4/3	2
two-input XOR	4	4

size inverter in a specific process and C_0 be the input capacitance of a unit-sized inverter in that technology. The delay of an ideal unit-size invertor, which does not have intrinsic capacitance at its output, driving an inverter is $\tau = R_0C_0$. To make analysis technology independent, all resistances, capacitances and delays are normalized with respect to R_0, C_0 and τ, respectively. The intrinsic capacitance of a gate consists of diffusion capacitance of transistors at the gate's output node. Table 5.1 shows the normalized input and intrinsic capacitance of various gates [RCN03] [SSH99].

The *fanout* of a gate is defined as the output load capacitance of the gate divided by the gate's input capacitance. It can be shown that the optimum speed for a logic circuit in path is achieved when each gate has a fanout almost equal to four [RCN03]. This is commonly known as the fanout four (FO4) in the literature. The delay of a FO4 inverter is a characteristic of the technology and is known by designers. In CMOS technology, buffers are made using inverters. Logic synthesizers insert the correct number of inverters and size

NAND gates so that the polarity of the signal does not change. In FO4 inverters (buffers), each inverter drives a load of $5C_0$, four other buffers and its own intrinsic load. Thus, the delay of a FO4 inverter is $R_0(4C_0 + C_0) = 5R_0C_0$ and, in normalized form, the delay is 5.

Consider the multiplier in Fig 5.1. Assume the AND gate matrix is replaced with a NAND gate matrix and input size m. The overall normalized capacitive load at each line a_i or b_i is $\overline{C}_{load} = \frac{4}{3} \times m$, which is the input capacitance of m NAND gates driven by an input line. Let us assume N stages of buffers are used to drive the NAND gate for each input line. Let the fanout of each buffer be four; that is, each buffer drives four identical buffers. Fig. 5.3 illustrates N stages of buffers driving NAND gates. The load of three NAND gates is $3 \times \frac{4}{3}$, which is four times the input capacitance of an inverter. Therefore, in the last stage, each buffer drives three NAND gates to have a fanout of 4.

5.2.2 Delay

Each input line a_i or b_i of the multiplier, needs to drive m NAND gates and each buffer in the last stage drives three NAND gates. Thus , we get:

$$m = 3 \times 4^{N-1}. \tag{5.7}$$

The normalized delay of each buffer stage is 5. Thus, the total delay of N stages is:

$$\overline{T}_B = 5N = 5 \log_4 \frac{4m}{3}.$$

The overall gate delay of the degree m polynomial multiplier is the delay of one NAND gate, buffers, and the longest XOR tree. Therefore:

$$T = T_{NAND} + T_{XOR} \log_2 m + 5T_{UB} \log_4 \frac{4m}{3}, \tag{5.8}$$

where T_{UB} is the delay of unit-size buffer and $m \geq 1$. Let us assume buffers are not used in Fig. 5.1 and each input line is driven by one buffer. The total normalized capacitive load at the input line would be: $(m \times \frac{4}{3} + 1)$. Therefore the overall delay would be:

$$T = T_{NAND} + T_{XOR} \log_2 m + T_{UB}(\frac{4}{3}m + 1).$$

Apparently, (5.8) results in a faster (and larger) multiplier.

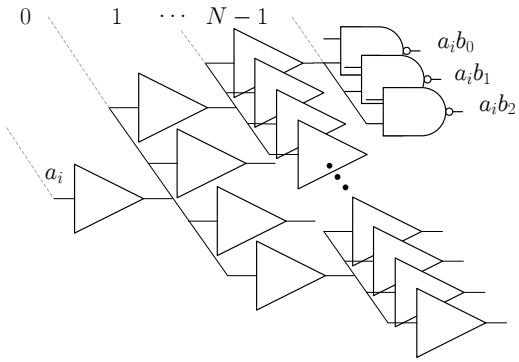


Figure 5.3: Buffers driving NAND gates in a polynomial multiplier. The last buffers can drive 3 NAND gates.

5.2.2.1 Simplifying the Delay Equation

In a particular technology the delay of an FO4 inverter is known. Therefore, in (5.8) we set $T_{INV} = 5T_{UB}$, where T_{INV} is the delay of the FO4 inverter. Thus, the gate delay of a degree m polynomial multiplier can be simplified as:

$$T_{poly} = T_{NAND} + T_{XOR} \log_2 m + 0.5 T_{INV} \log_2 m, \tag{5.9}$$

where it is assumed that $5T_{UB} \log_4(4/3) \approx T_{UB} \ll T$ and can be neglected. Note that we have also dropped the ceiling function from the log() functions. The reason is discussed in a the later section.

5.2.2.2 Implementation

We have implemented polynomial multipliers using Verilog hardware description language and have synthesized them using the Synopsys Design Compiler and a 0.18μm standard cell library. Fig. 5.4 shows (5.2), (5.3) and (5.8) and the synthesis result of 26 multipliers, where $m = 8i$, $1 \leq i \leq 26$. Fig. 5.4 indicates that the implementation delay is not a stair-case as it is presented in (5.2) and projected in the logic diagram of Fig. 5.1. The reason is that synthesizers construct each path in the logic circuit using various gates with small granularity in driving ability and size, rather than a single type of gate. In other words, in a logic circuit,

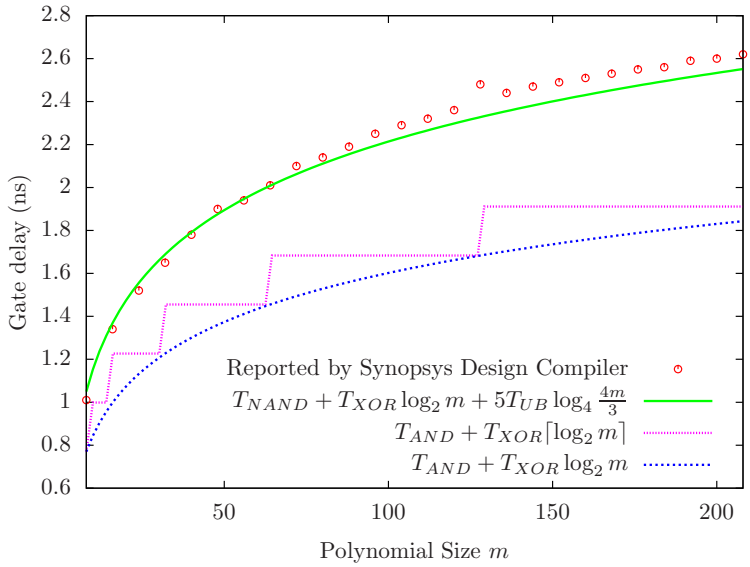


Figure 5.4: Gate delay of 26 synthesized multipliers compared with delay equations (5.8), (5.2) and (5.3)

not all NAND gates and XOR gates and buffers have the same size and delay. Therefore, the delay becomes more continuous as opposed to a stair-case.

For two reasons (5.2) can not be considered a precise gate delay model. First, the stair-case behavior is not observed in practice. Second, the delay of buffers are not taken into account and, therefore, the estimated delay is about 40% smaller than the synthesis results. As shown in Fig. 5.4, the delay predicted by (5.8) is more precise and is about 4% smaller than the delay obtained from the synthesis results. Implementation results and analysis projected in Fig. 5.4 indicates that for large multipliers, almost 40% of the delay is caused by the buffers. Since the depth of XOR trees in a multiplier are not equal, signals do not have to reach at the same time to all NAND gates. This property allows optimization of the drivers and has been explained briefly in the Appendix.

5.2.3 Area

The sum of the first N terms of the geometric series 4^i is: $\sum_{i=0}^{N-1} 4^i = \frac{4^N-1}{3}$. Therefore, using (5.7), the number of buffers for each line a_i or b_i is $(4m/3-1)/3$ and the total number of

buffers are $\overline{S}_B = \frac{2m(4m-3)}{9}$. Therefore, the total area is:

$$S_{total} = (m-1)^2 S_{XOR} + m^2 S_{NAND} + \frac{2m(4m-3)}{9} S_{UB}, \tag{5.10}$$

where S_{XOR}, S_{AND} and S_{UB} are the area minimum size gates.

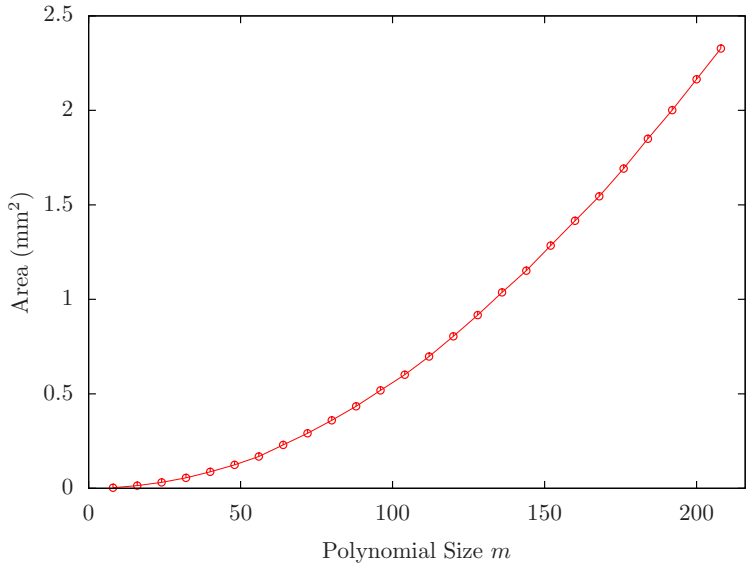
Fig. 5.5(a) shows the actual implementation results. As expected, the area is a quadratic function of the polynomial degree m. To get an idea of the size of a polynomial multiplier, the area of the popular ARM7TDMI RISC microprocessor, used in many cell phones and other portable devices, is about 0.6 mm^2 in 0.18 μm technology [ARM]. In the 0.18 μm standard cell library we used, the area of minimum-sized gates are $S_{XOR} = 32.53$ μm^2, $S_{NAND} = 12.2$ μm^2 and $S_{UB} = 8.13$ μm^2. These values have been used in (5.10), (5.4) and (5.5) to estimate the area of the multiplier as a function of m. Fig. 5.5(b) shows the deviation of (5.10), (5.4) and (5.5) from the actual implementation result in Fig. 5.5(a). As shown in the figure, the proposed equation in (5.10) has the lowest deviation from the actual implementation.

5.2.3.1 Two Wrongs Make a Right

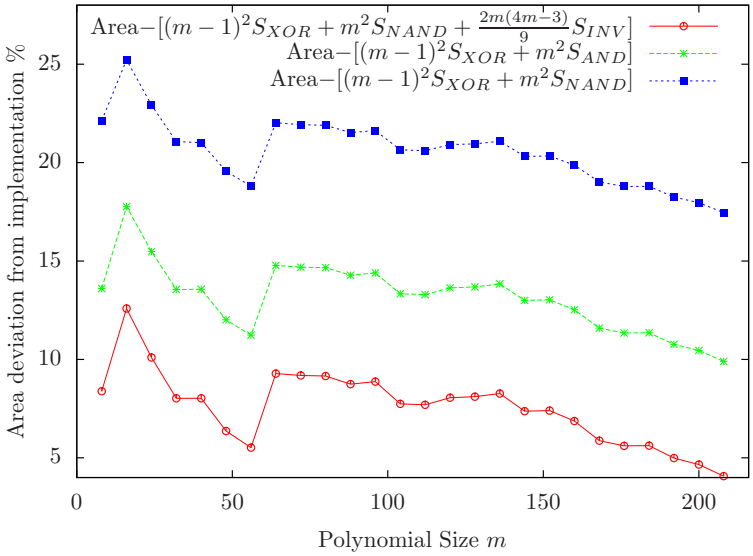
There are two problems with (5.4). First, buffers have not been considered. Second, NAND gates should be used, because they are faster and smaller than AND gates. The area of an AND gate is almost equal to the area of a NAND gate, plus the area of an inverter. Equation (5.4) implicitly assumes m^2 extra inverters for the multiplier, which is almost the same amount of inverters actually used as buffers in (5.10). In fact, unintentionally, (5.4) assumes NAND gates and buffers are used. Therefore, the area equation with AND gates in (5.4) is closer to the actual implementation result than the area equation with NAND gates in (5.5).

5.2.4 Energy

The finite field multiplier is the core computational block of the elliptic curve scalar multiplication operation. It occupies the bulk of hardware in a high performance elliptic curve processor (ECP) and, therefore, is the first target for reducing energy consumption. In this



(a) Area of 26 synthesized multipliers



(b) Deviation of the area equations (5.10), (5.4) and (5.5) from the actual area in (a).

Figure 5.5: Area evaluation

section, we investigate the energy consumption of the conventional polynomial multiplication algorithm.

To compute the energy consumption of the polynomial multiplier, we need the switching probability and capacitance at each node of Fig. 5.1, which illustrates the multiplication in

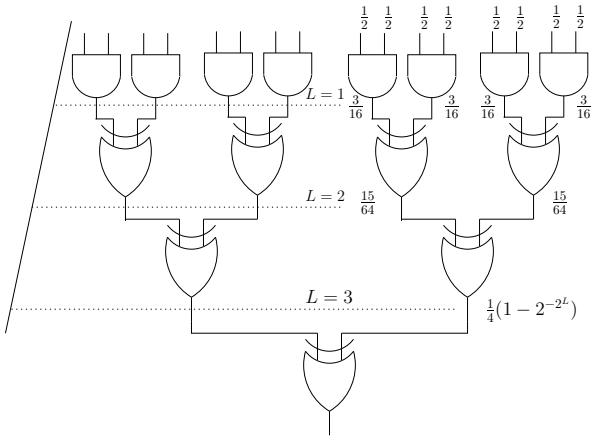


Figure 5.6: XOR tree and the switching probability at each node

(5.1). The switching probability of a node is $P(0 \to 1) = P(1)\left[1 - P(1)\right]$, where $P(1)$ is the static probability of the output of a logic level being high. Fig. 5.6 shows the toggle probability at nodes of an XOR tree, similar to those used in in the polynomial multiplier of Fig 5.1. If both inputs to an XOR gate have a static probability of $\frac{1}{2}$, the toggle probability at the output is $\frac{1}{4}$. In XOR trees used in multipliers, the input to XOR gates comes from NAND gates with an output switching probability of $\frac{3}{16}$. In this case, the toggle probability at a node i in level L of an XOR tree equals to $P_i(0 \to 1) = (1 - 2^{-2^L})/4$, where $L \in \mathbb{N}$ and $1 \leq L \leq \lceil \log_2 m \rceil$ and m is total number of inputs to the tree. The toggling probability approaches the maximum value of $\frac{1}{4}$ exponentially as the level number L increases. Thus, for simplicity, we assume the toggle probability to be $P_x = \frac{1}{4}$ for all nodes in the XOR tree.

To find the capacitance at each node, we normalize the load and intrinsic capacitance of each node [SSH99]. The total (normalized) capacitance at each node of the XOR tree equals the sum of the input capacitance of an XOR gate, plus the intrinsics capacitance of an XOR gate. Using Table 5.1, $\overline{C}_x = 4 + 4 = 8$. Thus, the normalized switched capacitance at the output of each XOR gate in the XOR tree equals to $\overline{C}_{XOR,SW} = \overline{C}_x P_x = 8 \times \frac{1}{4} = 2$. The switching probability at the output of a NAND gate is $P_a = 3/16$, and the normalized capacitance load is $\overline{C}_a = (2 + 4)$. Therefore, the normalized switch capacitance is $\overline{C}_{NAND,SW} = \overline{C}_a P_a = \frac{3}{16} \times (2 + 4) = \frac{9}{8}$. Each driver drives four other drivers and its own

intrinsic capacitance. Thus, the normalized capacitive load at the output of each buffer is $\overline{C}_i = 5$. The probability of switching at the output of each driver is $P_i = 1/2$, and the total switched capacitor is $\overline{C}_{UB,SW} = \overline{C}_i P_i = \frac{5}{2}$. To obtain the total normalized switched capacitance, we replace each S in (5.10) with the corresponding C from the preceding normalize capacitance equations. Thus, the total normalized capacitance is:

$$\overline{C}_{total,SW} = \frac{233}{36}m^2 - \frac{17}{3}n + 2 = 6.47m^2 - 5.67m + 2. \tag{5.11}$$

Using (5.11), the energy consumption of a polynomial multiplier for one multiplication in a specific technology is:

$$E_{total} = \overline{C}_{total,SW} C_{UB} V_{op}^2, \tag{5.12}$$

where V_{op} is the operating voltage of the circuit and C_{UB} is the input capacitance of the unit size inverter in that technology.

To evaluate the accuracy of our model, we expressed 26 polynomial multipliers in Verilog, and simulated them with 100,000 random inputs to capture the switching activity of each multiplier using the Cadence Verilog simulator (NCVerilog). Then, the switching activity is used in the Synopsys power analyzer to get the power consumption of each multiplier. Fig. 5.7 shows the estimated energy using (5.12) and the energy obtained using implementation versus polynomial size m. The estimated energy is larger than the implementation results. The reason is that the estimated value uses the $C_{UB} = 4.68$ fF, which also includes capacitances other than the gate capacitance of the unit-size inverter.

5.2.4.1 The Energy/Area Ratio

Using (5.10) and (5.12) the energy-over-area ratio for one multiplication is:

$$\frac{E_{total}}{S_{total}} = \frac{(\frac{233}{36}m^2 - \frac{17}{3}n + 2)C_{UB}V_{op}^2}{pm^2 - qm + r}, \tag{5.13}$$

where $p = S_{XOR} + S_{NAND} + \frac{8}{9}S_{INV}, q = (2S_{XOR} + \frac{2}{3}S_{INV})$ and $r = S_{XOR}$. Apparently, for a large m, the ratio approaches a constant value; but, the question is, how large should m be to get a relatively constant ratio? Fig. 5.8 shows the energy/area ratio versus polynomial

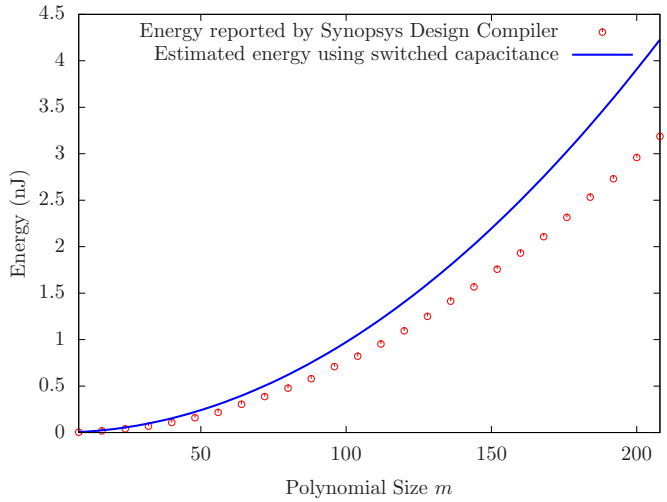


Figure 5.7: Energy consumption for one polynomial multiplication: Comparing Synopsys power report with the estimated energy in (5.12).

size m in (5.13). The figure indicates that for $m > 40$, the energy/area ratio is almost constant. Therefore, consumed energy for one polynomial multiplication using the conventional algorithm is proportional to its area. The implementation result of the energy/area ratio is shown in Fig. 5.8. Both the implemented results and (5.13) indicate that the energy/area is constant for sufficiently large values of m. Since the area of polynomial multiplication can be easily obtained, its energy consumption can be estimated using the constant energy/area coefficient.

5.3 The Karatsuba-Ofman Multiplication Algorithm

The influence of capacitive loads and buffers should also be considered in the design of other arithmetic structures. In this section, we show that such loads also exist in subquadratic multiplication algorithms, such as the KOA. We also show that the KOA on hardware is not as slow as it is commonly believed to be.

In the KOA algorithm, the multiplication of two two-term polynomials over ring $\mathbb{R}$ is

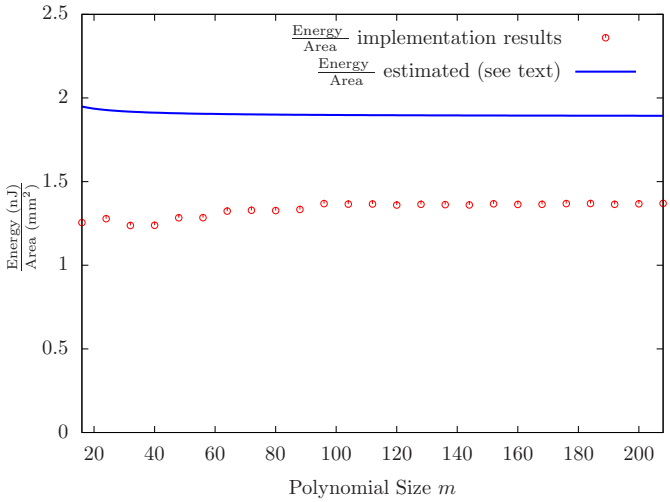


Figure 5.8: Predicted $\frac{\text{Energy}}{\text{Area}}$ for Polynomial Multiplication over $\mathbb{F}_2[x]$

performed as follows:

$$\begin{aligned}(a_1t + a_0)(b_1t + b_0) &= a_1b_1t^2 + a_0b_0 \\ &+ [(a_1 + b_1)(a_0 + b_0) - a_0b_0 - a_1b_1]\, t,\end{aligned} \tag{5.14}$$

where $a_0, a_1, b_0, b_1 \in \mathbb{R}$. Generalized forms of the KOA algorithm are discussed in [Mon05] and [Sun04].

5.3.1 Delay of KOA

The delay of the KOA algorithm is reported to be:

$$T = \mu + 3\alpha \log_2 m,$$

where μ and α are the delay of digit-multiplication and digit-addition (or subtraction) in the underlying ring [Sun04], [PFR98], [Paa96], [PL07], [GS05].

Considering (5.14), a_0b_0 and a_1b_1 can be performed in parallel and then added together to form $a_0b_0 + a_1b_1$, which would take $\mu + \alpha$. At the same time, $a_0 + b_0$ and $a_1 + b_1$ can be performed in parallel and then multiplied together to form $(a_0 + b_0)(a_1 + b_1)$. This operation would also take $\alpha + \mu$. Now, the two results can be added together to form

$(a_0 + b_0)(a_1 + b_1) - (a_0 b_0 + a_1 b_1)$, which would take another α. This procedure is visualized in Fig 5.9(a). Thus, the delay of performing (5.14) in hardware is $\alpha + 2\mu$. Fig. 5.9(b) shows the multiplication of two four-terms polynomials. The same argument applies, except, digit-multiplication is replaced with the two-term multiplier $M(2)$. Each recursion step adds 2α to the overall delay. Thus the delay of KOA is:

$$T = \mu + 2\alpha \log_2 m.$$

In $GF(2)[x]$, μ and α are translated to T_{AND} and T_{XOR}, respectively. The delay of KOA becomes:

$$T = T_{AND} + 2T_{XOR} \log_2 m.$$

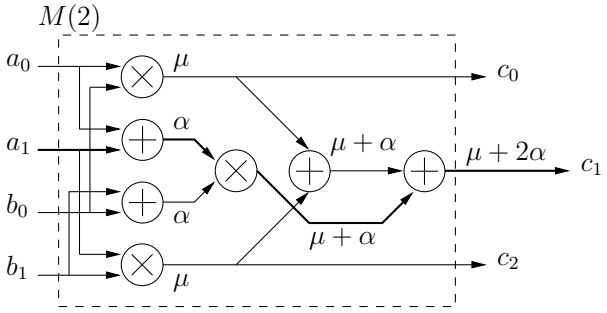
5.3.1.1 Limited fanout gates

Similar to the conventional multiplication algorithm, in KOA buffers are required to drive high capacitive loads. Let $M(i)$ denote the KOA module which multiplies two i-term polynomials in $GF(2)[x]$, where i is a power of two. The load on the inputs of $M(2)$ is $L(2) = \overline{C}_{AND} + \overline{C}_{XOR}$, where $\overline{C}_{AND}$ and $\overline{C}_{XOR}$ are the normalized input capacitance of AND and XOR gates respectively. Each recursion step adds the load of $\overline{C}_{XOR}$ to each input line. Thus the input load of $m = 2^k$ multiplier is:

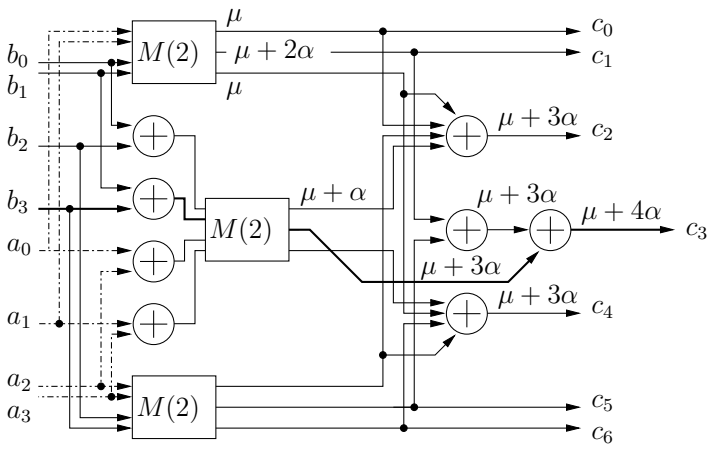
$$L(2^k) = \overline{C}_{AND} + k\overline{C}_{XOR}.$$

The input load in the KOA multiplier is smaller then that of conventional multiplier. However, a $M(2^i)$ multiplier, where $1 < i \leq k$, contains three $M(2^{k-1})$ multipliers, the load on inputs of each is $L(2^{k-1})$. Therefore, loads are distributed inside a $M(2^k)$ multiplier, rather than appearing only at input.

Derivation of the the delay equation for the KOA based on the FO4 model, is our future work. The implementation results for the delay of KOA versus the conventional algorithm is shown in Fig 5.10. The plot shows that after considering the effect of buffers, the delay the of KOA is still expected to be twice of that of the conventional algorithm.



(a) two-term polynomials



(b) four-term polynomials

Figure 5.9: Gate delay for multiplication using KOA

5.4 Future Work

The presented approach may be used to estimate power consumption of crypto-processors based on binary extension fields. Different architectures for optimal normal bases and Mastrovito multiplier can also be analyzed to include the effect of capacitive loads and derive energy consumptions. The architectures shown in appendix can also be considered to distribute buffers and provide more efficient multipliers.

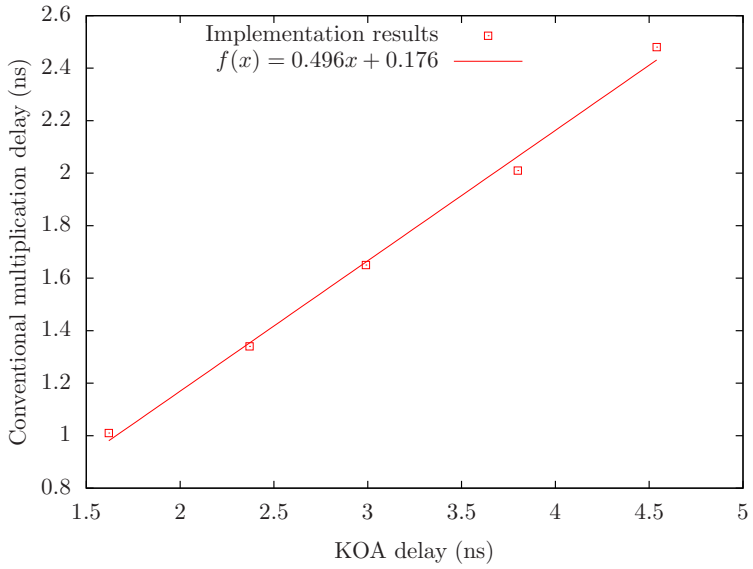


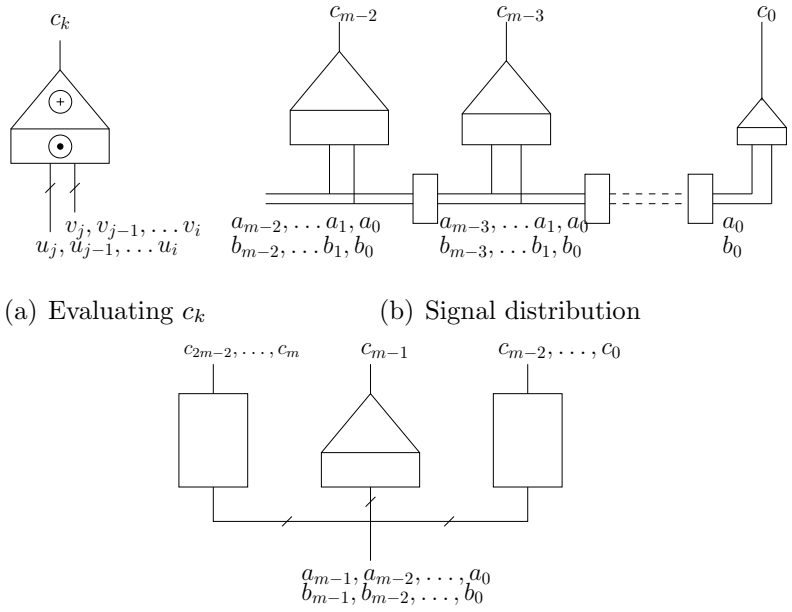
Figure 5.10: Conventional polynomial multiplication delay versus KOA delay for $m \in \{8, 16, 32, 64, 128\}$

5.5 Conclusion

We have derived models for area, delay and energy of polynomial multiplication in $GF(2)[x]$, and have compared them with the ASIC implementation results. We have showed that the accuracy of delay and area models can be improved by takeing into account drivers, using NAND gates instead of AND gates and removing the ceiling function from the $\log_2()$ functions. Our analysis and implementation results indicate that the energy consumption of one polynomial multiplication is proportional to the area of the multiplier. Therefore, the area of multipliers can be used to predict the energy or power consumption of multiplication. We also showed that the KOA algorithm is almost two times slower than the conventional polynomial multiplication algorithm.

Appendix: Gate Delay Optimization

There are a number of ways to reduce the critical path and the number of buffer stages in a multiplier.



(a) Evaluating c_k (b) Signal distribution

(c) Parallel multiplier

Figure 5.11: Equalizing delay path in in parallel polynomial multiplier

5.5.1 Partitioning and Buffer Distribution

In a multiplier over $GF(2)[x]$, the carry propagation chain does not exist. Considering (5.1), the multiplier may be considered as $2m-1$ separate and independent units with common inputs A and B, performing a dot product. Let $U = (u_j, u_{j-1}, \ldots, u_i)$ and $V = (v_j, v_{j-1}, \ldots, v_i)$, where $u_j, v_i \in \{0, 1\}$ and $j > i$, be inputs to such unit. The unit computes $\mathbf{w} = (u_j v_i, u_{j-1} v_{i+1}, \ldots, u_i v_j)$ and $c_k = \sum_{t=i}^{j} w_t$ using AND gates and a XOR tree, as shown by a rectangle and a triangle in Fig. 5.11(a). Delay of each unit depends on $|i-j|$ and maximum delay belongs to c_{m-1}.

In order to decrease the maximum delay, the signal distribution is made in such a way that slower units receive the signals earlier. Then, the signal is buffered and passed to the next unit. This is shown in Fig. 5.11(b). The structure in Fig. 5.11(b) produces c_i, where $0 \leq i \leq m-2$. A duplicate of such a structure can be used to produce c_i for $m \leq i \leq 2m-2$. As shown Fig. 5.11(c), a complete multiplier consists of two such structures and a unit which produces c_{m-1}.

5.5.2 Register Duplication

If the multiplier is part of a larger architecture such as an ECP, some bits of the input registers to the multiplier can be multiplicated to reduce the number of buffer stages. This adds more load, and consequently more delay, to the previous logic stage, but reduces the delay in the critical path of the multiplier. Basically, it equalizes the delay between the multiplier and its preceding logic state.

CHAPTER 6

Efficiency Metrics for Elliptic Curve Crypto-Processors

Abstract In this chapter, metrics for energy and area efficiency of elliptic curve crypto-processor (ECP). While many designs have been reported to be optimized towards low power consumption it is not really clear which ones are more efficient. Our goal is to provide a unified metric for comparison of different ECPs. It is desirable that power/energy be expressed in a process independent form so that the processors can be compared based on the architecture rather than the manufacturing process. A number of published ECPs have also been analyzed using the proposed metrics.

6.1 Introduction

Elliptic curve cryptography (ECC) is a public-key crypto-system based on the algebraic structure of elliptic curves over finite fields. ECC requires smaller key size compared to other public key crypto-systems such as RSA. This feature leads to hardware implementations which require less area, memory, CPU time and power. Consequently, ECC is more favorable in portable computing devices such as sensor networks, PDAs, cell phones and media players. The power and area constraints on such devices make it vital to explore power and area efficient algorithms, architectures, logic/circuit and layout techniques for ECCs.

ECC has a rich underlying mathematics and for the same level of security it may be implemented in many different ways. Generally, before designing the architecture, one needs to decide on the type of the underlying finite field (FF), the elliptic curve coefficients, representation of the points on the curve, scalar multiplication algorithm, FF representation and FF

computation algorithm. These options contribute to the speed, area and power efficiency of the systems. On the other hand, RSA public key crypto-systems is defined in one algebraic field and is defined by fixed equations. Similarly, in symmetric cryptography the encryption architecture is also fixed.

The fundamental operation in an ECC is elliptic curve scalar multiplication kP, where k is an integer and P is a point on the curve. Although an elliptic curve processor (ECP) may need to perform other cryptographical operations to realize a complete ECC, its main objective is to compute scalar multiplication efficiently. Scalar multiplication itself, is based on other primitive finite field operations namely addition, subtraction, multiplication and division. A finite field multiplier constructs the major component of an elliptic curve processor (ECP).

Low power ECC implementations have been reported in [GC01, HLR03, OSS04, SBG02, BMS06], where average power consumption has been reported as the measure of merit. Although average power consumption has been used traditionally to compare power efficiency of these processor, it is not a suitable measure since it depends on the operating clock frequency, operating voltage and the underlying CMOS technology. For example, porting a processor to the next generation technology decreases the power, but it does not make it a better architecture. In order to design a power efficient ECC architecture/algorithm a proper metrics is required. Lack of power and area metrics in this area of research causes confusion and makes it difficult to distinguish energy and area efficient ECC architectures and algorithms.

Here, we provide metrics to measure the power and area efficiency of elliptic curve processors for a certain level of security. These metrics helps us to compare ECPs and identify efficient architectures/algorithms. We focus on the architecture and do not consider low power electronics and circuit techniques such as gate sizing or supply and threshold voltage optimization.

In the next section scalar multiplication and difference between power consumption in a general purpose processor and an ECP is explained. In Section 6.3, equations for normalized

energy consumption are derived. In Section 6.4 a number of published ECPs are analyzed using the normalized energy formula and in the last section general methods for power/energy reduction in an ECP are discussed.

6.2 Energy Consumption in ECPs

In elliptic curve crypto-systems, points on an elliptic curve E, defined over a finite field $GF(2^m)$ or $GF(p)$, along with a special point called infinity, and a group operation known as point addition, form a commutative finite group. If P is a point on the curve E, and k is a positive integer computing

$$kP = \underbrace{P + P + P + \cdots + P}_{k \quad times}$$

is called scalar multiplication. Points P are represented by their coordinates (x, y) over a finite field and the point addition operation consists of a series of finite field operations such as addition, multiplication and inversion. The result of scalar multiplication is another point Q on the curve E. It is normally expressed as $Q = kP$. Scalars such as k are random integers. The binary representation of $k = \sum_{i=0}^{n-1} k_i 2^i$ has n bits where $k_i \in \{0,\ 1\}$ and $n \approx m$ or $n \approx \log_2 p$. Scalar multiplication is the most dominant computation part of elliptic curve cryptography. The reader is referred to [HMV03] for detailed explanation.

On a general-purpose processor, each instruction has different power/energy consumption. These processors perform variety of data processing and computations which are not known to the designers beforehand and hence are assumed to be random at design time. Therefore, it is more practical and convenient to measure the average energy consumption of a processor for a set of random instructions. Since the overall processing time of a processor is not known beforehand either, it is convenient to characterized them in terms of power consumption, ie., energy consumption per unit processing time. Later on, this figure maybe used to calculate the energy consumption of the chip for a specific computation.

In contrary to general-purpose processors, where computations are random, the main

task of an ECP is the computation of elliptic curve scalar multiplication operation, which is composed of series of finite field addition, multiplication, squaring and division over the underlying finite field. Therefore, it is more appropriate to characterized ECPs in terms of energy consumption for one scalar multiplication rather than average power consumption. The energy consumption of scalar multiplication depends on the number of parameters such as scalar multiplication algorithm, type of the finite field (binary or prime), field dimension, FF multiplication algorithm and the VLSI technology. It is desirable that power/energy be expressed in a process-independent form so that the processors can be compared based on their architecture rather than the manufacturing process.

6.3 Normalized Energy

The performance of CMOS technology is well predicted by the well-known scaling theory. Here, we use scaling theory to develop a metric to measure the energy efficiency in an ECP, which enables us to compare ECPs designed using different CMOS technologies and estimate the efficiency of algorithms used in an ECP.

6.3.1 Technology Scaling

CMOS scaling theory has been extensively used to predict and analyze the behavior of future CMOS technologies. However, the impact of device physics and non-fully scalable parameters like threshold voltage (V_T) and gate oxide thickness (t_{ox}) has limited the application of linear scaling in small feature size technologies. That is, the exponential dependence of transistor leakage current to V_T limits the allowable V_T and the tunneling gate current caused by small gate dielectric thickness limits the scaling [Bor99]. As a result, from $0.13\mu m$ technology forward, V_{DD}, V_T and t_{ox} has been scaled slowly. Here, we assume full scaling, where all technology parameters scale with the same factor. This assumption is reasonable here as we deal with the feature sizes equal or larger than $0.13\mu m$. Table 6.1 shows typical parameters for various CMOS technologies, where linear scaling factor equals to 0.7.

Table 6.1: Technology Scaling ($S = 0.7$)

Technology (μm)	0.5	0.35	0.25	0.18	0.13
Voltage (V)	5	3.3	2.5	1.8	1.2
V_t (V)		0.65	0.53	0.42	0.28
t_{ox} ($\dot{A}$)		77.8	55.6	40	28.9

6.3.2 Normalized Energy

The dynamic power consumption of a logic gate in a sequential circuit is characterized as $P = \alpha_{0\to1} C V_{op}^2 f$ where $\alpha_{0\to1}, C, V_{op}, f$ are the probability of transition from logic low to high (switching activity), output capacitance of the gate, operation voltage and frequency of the circuit. In a chip, the overall average dynamic power consumption equals to sum of power consumption of all gates:

$$P_{av} = \sum_{\text{for all gates } i} \alpha_i C_i V_{op}^2 f, \tag{6.1}$$

Let $T_{kp} = NT$ be the total time required for one scalar multiplication where $T = 1/f$ is the clock period and N is the number of clock cycles required for one scalar multiplication. Using (6.1), the energy consumption of an ECP for one scalar multiplication can be expresses as

$$E_{av} = N V_{op}^2 \sum_i \alpha_i C_i \tag{6.2}$$

where C_i is the total capacitance at the out of the gate i, which consists of the output parasitic and the load capacitance.

Let W_i be the total transistor width at the out of a gate, and W_{inv} and C_{inv} be the transistor width and input capacitance of unit size inverter respectively. Then

$$C_i = \frac{W_i}{W_{inv}} C_{inv}.$$

Equation (6.2) can be written as

$$E_{av} = N V_{op}^2 C_{inv} \sum_i \alpha_i \frac{W_i}{W_{inv}}.$$

Since $E_{inv} = V_{op}^2 C_{inv}$ is the energy consumption of unit size inverter in one clock cycle, we may write

$$\overline{E} = \frac{E_{av}}{E_{inv}} = N \sum_i \alpha_i \frac{W_i}{W_{inv}}. \tag{6.3}$$

$\overline{E}$ can be seen as technology independent normalized energy for on scalar multiplication. However, the problem with (6.3) is that E_{inv} is often not available in publications and therefore computation of $\overline{E}$ becomes infeasible. We develop another metric which is less intuitive but is more practical.

Equation (6.2) can be written as

$$E_{av} = N V_{op}^2 \sum_i \alpha_i \frac{\varepsilon_{ox} W_i L}{t_{ox}}, \tag{6.4}$$

where L and t_{ox} are the technology feature size and oxide thickness respectively. Considering ECPs under ideal scaling condition, (6.4) can be normalized by the channel width and operating voltage.

$$\overline{E} \ = \frac{E_{av}}{L V_{op}^2} = N \sum_i \alpha_i \frac{\epsilon_{ox} W_i}{t_{ox}}. \tag{6.5}$$

Assuming linear scaling, W_i/t_{ox} remains unchanged since both W_i and t_{ox} are scaled by the same factor. We define $\overline{E}$ as the normalized energy consumption of an ECP. It represents the energy consumption for one scalar multiplication for unit channel length and unit operating voltage for an ECP. It is independent of CMOS technology (as long as linear CMOS technology scaling holds) and depends on the architecture and the algorithm used for the scalar multiplication. It can be used as a bench mark to compare the energy consumption of ECPs regardless of the implemented technology may also be expresses as

$$\overline{E} = \frac{P_{av} T_{kP}}{L V_{op}^2}. \tag{6.6}$$

Equation (6.6) uses the parameters which are usually reported for hardware implementations. If a design is ported to another technology, and/or the operation voltage/frequency is changed, the normalized energy consumption does not improve.

6.3.3 Area Efficiency

The normalized energy does not reflect the area required to sustain a certain level of throughput. The first order technology scaling can be performed as

$$\overline{A} = \frac{Area}{A_{NAND}},$$

where A_{NAND} is the area of unit size inverter in a specific technology (A_{INV} may be used as well). If area is expressed in terms of gates, normalization is not required.

6.3.4 Effect of VLSI architectural improvements on the energy metric

The concept of parallelism, time multiplexing and pipelining are often used to improve the power and area efficiency of VLSI architectures [CSB92]. Parallelism, as defined [CSB92], is essentially a hardware duplication mechanism. It doubles the switched capacitance and throughout, and saves power by reducing the operating voltage. Hardware duplication doubles the number of transistors (W_i's) and decreases N by half in (6.5), which results in the same amount of normalized energy without taking into account leakage current. Since the amount of switched capacitors times the number of clock cycles for an operation does not change, the energy efficiency is not improved. It means that our metric does not merit hardware duplication. Similar argument holds for multiplexing and pipelining. Here, we would like to identify scalar multiplication algorithms which result is less switching activity. Our metric shows improvement if an algorithm can reduce the total switching activity or the total normalized capacitance while maintaining the same throughput. The traditional architectural improvements may be applied after an efficient ECP algorithm has be identified. The area metric should be taken into account since energy efficient architecture might not fulfill the area constraints.

Table 6.2: Power/Energy consumption of ECPs

Design	Finite Field	Tech. (L) μm	Voltage (V) V	Freq. (f) MHz	Area gates	Power mW	T_{kP} mS	$\overline{E}$
[GC01]	160 bits	0.25	2.0	50	880,000	75	5.2	390
[HLR03]	$GF(2^{163})$	0.35	3.5	100	56,000	10.6	2.4	6.7
[OSS04]	$GF((2^{167}+1)/3)$	0.13	1.2	20	30,333	0.99	31.9	168.7
[OSS04]	-	-	-	100	30,444	4.34	6.3	146.0
[OSS04]	-	-	-	200	34,390	9.98	3.1	165.3
[SBG02]	$GF(2^{2\times 89})$	0.50	5.0	20	112,000	150	4.4	52.8
[BMS06]	$GF(2^{163})$	0.13	1.2	0.5	9900	0.028	190	28.4

6.4 Comparison of ECPs

Table 6.2 indicates various parameters including the power consumption of ECPs designed over finite fields of approximately 160 bits, i.e. the same level of security. These designs are implemented using different underlying finite field, architecture and CMOS technology. Therefore, they have different clock frequencies, operation voltages, and scalar multiplication timings. These are the only published ECPs which we could find with power consumption data available. A brief explanation of each design follows:

[GC01] It is a dual field ECP which provides a mechanism for trade-off between security level and energy consumption. It is designed for energy-constrained devices and meant to be energy efficient.

[HLR03] It is designed over $GF(2^m)$ where $149 \leq n \leq 251$ and uses serial most-significant-bit-first finite field multiplier. In contrary to the most ECP's it uses affine coordinates for point representation. The architecture ensures the extra registers are not switching when it is working on a smaller field. The data on our table is extracted from the graphs in the paper.

[OSS04] It performs the 165-bit finite field operations over the field embedded in ring $\mathbb{Z}_{2^{167}+1}$, which provides low complexity modular reduction. In general redundant computation, where the complexity of an m-bit multiplication using d redundant bits is of $O((m+d)^2)$, required more gates. In this case, where $m = 165$ and $d = 2$, extra gates for the redundant computations are supposed to be compensated by the fewer number of gates in the reduction operation.

[SBG02] It uses point halving method for scalar multiplication and is optimized for power. It uses a composite field which is not recommended in elliptic curve cryptography but results in smaller computation complexity. The computation time is for EC Elgamal signature scheme, which is comprised of one scalar multiplication and a few more finite

filed multiplications. In our comparison, the time for extra multiplications are ignored compared to the number of multiplication required for the whole scalar multiplication.

[BMS06] It is designed for wireless sensor networks applications where small area and low energy are the main goals. It uses a digit serial finite field multiplier with small digit size.

In Table 6.2, energy consumption is normalized according to (6.6). The column labeled "$\overline{E}$" in the table indicates that at a certain level of security, [HLR03] would have the least amount of energy consumption for one scalar multiplication, had these designs been implemented with the same VLSI technology. The table also shows that low power consumption for [OSS04] at 20MHz is the reflection of working frequency and the underlying technology as its normalized energy consumption for the three different working frequencies are almost equal and are higher than that of [HLR03, OSS04, SBG02]. The higher energy consumption of [OSS04] could be the result of redundant computation and larger hardware due to computations in a prime field. The high power consumption of [GC01], which is being measured on the actual chip, could be the reflection of its high flexibility. If small area is required [BMS06] would be more appropriate while its energy consumption is not as low as [HLR03]. It can be concluded form the table that the low power consumption by itself may be misleading in the comparison of ECPs and the normalized energy model can be useful in the comparison of the energy efficiency of ECPs.

6.4.1 Decreasing the Energy Consumption in ECPs

Employments of the following methods can increase the energy efficiency of an ECP.

1. Synthesis constraints: Modern logic synthesizers can optimize the combination logic for power, area and speed. Synthesis for the highest speed is not energy efficient. A small percentage of excess delay can largely increase the power efficiency [MSN04].

2. Gate sizing, supply and threshold voltage optimization: Let $A(x) = \sum_{i=0}^{m-1} a_i x^i$ and

$B(x) = \sum_{i=0}^{w-1} b_i x^i$ where $a_i, b_i \in \{0, 1\}$ be two m and w terms polynomial over $GF(2)$. Polynomial multiplication in an ECP is computed as $D(x) = \sum_{i=0}^{m+w-2} c_i x^i$ where $c_k = \sum_{i+j=k} a_i b_j \pmod 2$. The output c_k is in critical if $w < k < m$. For other c_i's the path is shorter and hence the operating voltage may be decreased or the threshold voltage can be increased.

3. Reducing the total capacitance of the chip: Sub-quadratic multiplication algorithms [Mon05], reduce the overall complexity of the FF multiplier and result in less area on the chip which in turn reduces the total capacitance on the chip. However, it is not known whether they cause more switching activity or not. On the other hand such algorithms have a longer critical path which increases the total computation time.

6.4.2 Discussion

We realize that the proposed normalization technique involves approximations. For example: (a) The CMOS scaling theory is based on the first degree approximation. (b) The leakage power and the wiring capacitance are ignored in our analysis (We have applied the scaling on the feature size greater than 0.13 μm). (c) Power and timing estimations data are extracted from Synopsys and not the actual chip. However, the success of scaling theory on prediction of CMOS behavior may indicate that the proposed metric provides a more accurate/dependable results compared to the traditional ways of power comparisons in ECPs.

The energy metric determines the switching activity of an algorithm but it does not take in to account the impact of technology on the speed of an architecture. The delay and area requirements of an ECP architecture should also be taken in to account separately.

The security level of an ECP is determined by the dimension of the finite field and the size of prime p when the processor performs the operations of $GF(2^m)$ or $GF(p)$ respectively. An energy model which incorporates the security level is very useful in comparison of different implementations of ECPs.

6.5 Conclusion

Metrics for measuring the energy and area efficiency of ECPs, which are independent of the CMOS technology are introduced. It has been shown that relying on the power measurement can be misleading in identifying an energy efficient ECP. The proposed energy metric can be used to distinguish low energy arithmetic architectures for ECC.

CHAPTER 7

Conclusion and Future Work

We have covered various aspects of finite field arithmetic in cryptography. At the architectural level,an efficient parallel and pipeline elliptic curve processor, along with a semi-pipeline finite field multiplier with a short critical path, has been developed. A new parallel algorithm for finite field multiplication over type II ONB also has been proposed. Implemented in hardware, this algorithm runs twice as fast as conventional multipliers. In software, this algorithm can be used in SIMD or two core CPUs to increase multiplication speed. It has been shown that conventional power measurements are misleading if applied to ECC. New metrics for ECC power efficiency have been proposed, and models for accurate area, delay, and power consumption of multipliers over $GF(2^m)$ have been developed. The presented approach can be used to estimate power consumption of cryptoprocessors based on binary extension fields.

Error-free operation is critical in cryptography, because various fault-injection attacks can be used to extract cryptographic secrets or break a cipher. An efficient hybrid scheme for concurrent error detection of multiplication over $GF(p^m)$ using polynomial bases has been proposed, and this scheme has also been extended to type II ONB multipliers.

Many contributions in this thesis can be extended and generalized. The current area, delay, and power models have been developed for $GF(2^m)$ multipliers. These models can be extended to multipliers over $GF(p^m)$ and $GF(p)$. The power model only applies to a single multiplier. It would be desirable to extend this model to the whole elliptic curve processor (ECP) and compute the power consumption for one scalar multiplication. The delay model can also be extended to subquadratic multipliers such as KOA.

Although the proposed ECP was not targeted for low-power systems, initial estimation shows that its energy consumption is lower than many low-power ECPs. The low-energy consumption is due to the architecture where the multiplier never stalls and does not have any wasted cycles. This idea can be used to design low-energy ECPs.

Although the proposed CED scheme can be implemented on any $GF(p^m)$ multiplier, the probability of undetected errors only applies to the binary extension fields. It would be desirable to derive the probability of error on any extension field. The error model only covers random errors, whereas in cryptography, errors can be injected purposefully. A model for such injection errors would be desirable. The CED can be applied for the whole scalar multiplication and one ECP.

The proposed algorithm for ONB multiplication reduced the ONB multiplication to two polynomial multiplications. Since extensive research on polynomial multiplications exists, various schemes including subquadratic ones can be applied here. An ECP base on sub-quadratic multipliers can be very fast and small.

References

[AH08] Bijan Ansari and M. Anwar Hasan. "High-Performance Architecture of Elliptic Curve Scalar Multiplication." *IEEE Trans. Computers*, **57**(11):1443–1453, 2008.

[ARM] "http://www.arm.com/products/CPUs/ARM7TDMI.html.".

[AV10a] Bijan Ansari and Ingrid Verbauwhede. "FO4-based Models for Area, Delay and Energy of Polynomial Multiplication over Binary Fields." In *IEEE Workshop on Signal Processing Systems: Design and Implementation*, pp. 420–425, 2010.

[AV10b] Bijan Ansari and Ingrid Verbauwhede. "A Hybrid Scheme for Concurrent Error Detection of Multiplication over Finite Fields." In *Proceedings of the 2010 IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems*, DFT '10, pp. 399–407, Washington, DC, USA, 2010. IEEE Computer Society.

[BA00] Michael Lee Bushnell and Vishwani D. Agrawal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. Springer, 2000.

[BBK03] Guido Bertoni, Luca Breveglieri, Israel Koren, Paolo Maistri, and Vincenzo Piuri. "Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard." *IEEE Transactions on Computers*, **52**:2003, 2003.

[BDG02] Marcus Bednara, M. Daldrup, Joachim von zur Gathen, J. Shokrollahi, and Jürgen Teich. "Reconfigurable Implementation of Elliptic Curve Crypto Algorithms." In *IPDPS*. IEEE Computer Society, 2002.

[BDL01] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. "On the Importance of Eliminating Errors in Cryptographic Computations." *Journal of Cryptology*, **14**:101–119, 2001.

[BH05] S. Bayat-Sarmadi and M. A. Hasan. "Concurrent Error Detection of Polynomial Basis Multiplication over Extension Fields using a Multiple-bit Parity Scheme." In *DFT '05: Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 102–110, Washington, DC, USA, 2005. IEEE Computer Society.

[BH07a] Siavash Bayat-Sarmadi and M. Anwar Hasan. "On concurrent detection of errors in polynomial basis multiplication." *IEEE Trans. Very Large Scale Integr. Syst.*, **15**(4):413–426, 2007.

[BH07b] Siavash Bayat-Sarmadi and M. Anwar Hasan. "Run-Time Error Detection in Polynomial Basis Multiplication Using Linear Codes." In *ASAP*, pp. 204–209. IEEE Computer Society, 2007.

[BMS06] Lejla Batina, Nele Mentens, Kazuo Sakiyama, Bart Preneel, and Ingrid Verbauwhede. "Low-Cost Elliptic Curve Cryptography for Wireless Sensor Networks." In Levente Buttyán, Virgil D. Gligor, and Dirk Westhoff, editors, *ESAS*, volume 4357 of *Lecture Notes in Computer Science*, pp. 6–17. Springer, 2006.

[Bor99] Shekhar Borkar. "Design Challenges of Technology Scaling." *IEEE Micro*, **19**(4):23–29, 1999.

[BSS02] Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 2002.

[CCD05] Dario Catalano, Ronald Cramer, Ivan Damgard, Giovanni Di Crescenzo, David Pointcheval, and Tsuyoshi Takagi. *Contemporary Cryptology.* Birkhuser Basel, 2005.

[CCJ04] Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye. "Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity." *IEEE Trans. Computers*, **53**(6):760–768, 2004.

[CCL09] Che Wun Chiou, Chin-Chen Chang, Chiou-Yng Lee, Ting-Wei Hou, and Jim-Min Lin. "Concurrent Error Detection and Correction in Gaussian Normal Basis Multiplier over GF(2^m)." *IEEE Trans. Computers*, **58**(6):851–857, 2009.

[CCL11] Tai-Pao Chuang, Che-Wun Chiou, and Shun-Shii Lin. "Self-checking Alternating Logic Bit-Parallel Gaussian Normal Basis Multiplier with Type-t." In *IET Information Security*, pp. 44–42. IEEE, March 2011.

[CSB92] Ananth Chandrakasan, S. Sheng, and R. Brodersen. "Low-Power CMOS Digital Design." *IEEE Journal of Solid-State Circuits*, **27**(4):473–484, April 1992.

[CTL05] Ray C.C. Cheung, Nicolas Jean baptiste Telle, Wayne Luk, and Peter Y.K. Vjeung. "Customizable elliptic curve cryptosystems." *IEEE Trans. VLSI Syst.*, **13**(9):1048–1059, 2005.

[DBL04] *International Conference on Information Technology: Coding and Computing (ITCC'04), Volume 2, April 5-7, 2004, Las Vegas, Nevada, USA*. IEEE Computer Society, 2004.

[DH76] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography." *IEEE Transactions on Information Theory*, **IT-22**(6):644–654, 1976.

[DH04] Amir K. Daneshbeh and M. Anwarul Hasan. "Area Efficient High Speed Elliptic Curve Cryptoprocessor for Random Curves." In *ITCC (2)* [DBL04], pp. 588–593.

[DQ07] Guerric Meurice de Dormale and Jean-Jacques Quisquater. "High-speed hardware implementations of Elliptic Curve Cryptography: A survey." *J. Syst. Archit.*, **53**(2-3):72–84, February 2007.

[EJM02] M. Ernst, M. Jung, F. Madlener, S. Huss, and Rainer Blümel. "A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2^n)." In Kaliski et al. [KKP03], pp. 381–399.

[EYK06] Serdar S. Erdem, Tugrul Yanik, and Çetin K. Koç. "Polynomial basis multiplication over GF(2^m)." *Acta Applicandae Mathematicae*, **93**(1-3):35–55, 2006.

[FGB98] Sebastian Fenn, Michael Gossel, Mohammed Benaissa, and David Taylor. "On-Line Error Detection for Bit-Serial Multipliers in GF(2^m)." *J. Electron. Test.*, **13**(1):29–40, 1998.

[Gao93] Shuhong Gao. "Normal Bases over Finite Fields." Technical report, University of waterloo, 1993.

[GBG03] C. Grabbe, Marcus Bednara, Joachim von zur Gathen, J. Shokrollahi, and Jürgen Teich. "A High Performance VLIW Processor for Finite Field Arithmetic." In *IPDPS*, pp. 189–194. IEEE Computer Society, 2003.

[GC01] James Goodman and Anantha Chandrakasan. "An energy-efficient reconfigurable public-key cryptographyprocessor." *IEEE Journal of Solid-State Circuits*, **36**(11):1808–1820, November 2001.

[GS05] Joachim von zur Gathen and Jamshid Shokrollahi. "Efficient FPGA-Based Karatsuba Multipliers for Polynomials over F_2." In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pp. 359–369. Springer, 2005.

[GSE02] Nils Gura, Sheueling Chang Shantz, Hans Eberle, Sumit Gupta, Vipul Gupta, Daniel Finchelstein, Edouard Goupy, and Douglas Stebila. "An End-to-End Systems Approach to Elliptic Curve Cryptography." In Kaliski et al. [KKP03], pp. 349–365.

[HLR03] Chi Huang, Jinmei Lai, Junyan Ren, and Qianling Zhang. "Scalable Elliptic Curve Encryption Processor for Portable Application." In *5th International Conference on ASIC*, volume 2, pp. 1312 – 1316, October 2003.

[HMV03] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curves Cryptography.* Springer, 2003.

[IT88] Toshiya Itoh and Shigeo Tsujii. "A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases." *Inf. Comput.*, **78**(3):171–177, 1988.

[IT02] Tetsuya Izu and Tsuyoshi Takagi. "Fast Elliptic Curve Multiplications with SIMD Operations." In Robert H. Deng, Sihan Qing, Feng Bao, and Jianying Zhou, editors, *ICICS*, volume 2513 of *Lecture Notes in Computer Science*, pp. 217–230. Springer, 2002.

[KK07] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann, 2007.

[KKP03] Burton S. Kaliski, Çetin Kaya Koç, and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*. Springer, 2003.

[Kob87] Neal Koblitz. "Elliptic curve cryptosystems." *Mathematics of Computation*, **48**:203–209, 1987.

[KWM01] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. "Concurrent error detection of fault-based side-channel cryptanalysis of 128-bit symmetric block ciphers." In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pp. 579–584, New York, NY, USA, 2001. ACM.

[LD99] Julio López and Ricardo Dahab. "Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation." In Çetin Kaya Koç and Christof Paar, editors, *CHES*, volume 1717 of *Lecture Notes in Computer Science*, pp. 316–327. Springer, 1999.

[Lee08] Chiou-Yng Lee. "Concurrent Error Detection in Digit-Serial Normal Basis Multiplication over GF(2^m)." In *AINAW '08: Proceedings of the 22nd International Conference on Advanced Information Networking and Applications - Workshops*, pp. 1499–1504, Washington, DC, USA, 2008. IEEE Computer Society.

[Lee10] Chiou-Yng Lee. "Concurrent error detection architectures for Gaussian normal basis multiplication over GF(2^m)." *Integration*, **43**(1):113–123, 2010.

[LH04] Jonathan Lutz and M. Anwarul Hasan. "High Performance FPGA based Elliptic Curve Cryptographic Co-Processor." In *ITCC (2)* [DBL04], pp. 486–492.

[LM10] Chiou-Yng Lee, Pramod Kumar Meher, , and Jagdish Chandra Patra. "Concurrent Error Detection in Bit-Serial Normal Basis Multiplication Over GF(2^m) Using Multiple Parity Prediction Schemes." *IEEE Trans. VLSI*, **18**(8):1234–1238, 2010.

[LMP10] Chiou-Yng Lee, Pramod Kumar Meher, and Jagdish Chandra Patra. "Concurrent Error Detection in Bit-Serial Normal Basis Multiplication Over GF(2^m) Using Multiple Parity Prediction Schemes." *IEEE Trans. VLSI Syst.*, **18**(8):1234–1238, 2010.

[LMW00] K. H. Leung, K. W. Ma, W. K. Wong, and Philip Heng Wai Leong. "FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor." In *FCCM*, pp. 68–76. IEEE Computer Society, 2000.

[LT70] G. G. Langdon, Jr. and C. K. Tang. "Concurrent Error Detection for Group Look-Ahead Binary Adders." *IBM Journal of Research and Development*, **14**(5):563–573, September 1970.

[Mil86] Victor S Miller. "Use of elliptic curves in cryptography." In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pp. 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[Mis06] Pradeep Kumar Mishra. "Pipelined Computation of Scalar Multiplication in Elliptic Curve Cryptosystems." *IEEE Trans. Computers*, **55**(8):1000–1010, 2006.

[Mon87] Peter L. Montgomery. "Speeding the Pollard and elliptic curve methods of factorization." *Mathematics of Computation*, **48**:243–264, 1987.

[Mon05] Peter L. Montgomery. "Five, Six, and Seven-Term Karatsuba-Like Formulae." *IEEE Trans. Computers*, **54**(3):362–369, 2005.

[MOV89] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson. "Optimal normal bases in GF(p^m)." *Discrete Appl. Math.*, **22**:149–161, February 1989.

[MSN04] Dejan Markovic, Vladimir Stojanovic, Borivoje Nikolic, Mark A. Horowitz, and Robert W. Brodersen. "Methods for True Energy-Performance Optimization." *IEEE Journal of Solid-State Circuits*, **39**(8):1282–1293, August 2004.

[NAS] "http://history.nasa.gov/computers/Ch4-4.html.".

[OM86] J. Omura and J. Massey. "method and apparatus for finite field arithmetic.", May 1986.

[OP00] Gerardo Orlando and Christof Paar. "A High Performance Reconfigurable Elliptic Curve Processor for GF(2^m)." In Çetin Kaya Koç and Christof Paar, editors, *CHES*, volume 1965 of *Lecture Notes in Computer Science*, pp. 41–56. Springer, 2000.

[OSS04] E. Öztürk, Berk Sunar, and Erkay Savas. "Low-Power Elliptic Curve Cryptography Using Scaled Modular Arithmetic." In Marc Joye and Jean-Jacques Quisquater, editors, *CHES*, volume 3156 of *Lecture Notes in Computer Science*, pp. 92–106. Springer, 2004.

[Paa96] Christof Paar. "A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields." *IEEE Transactions on Computers*, **45**:856–861, 1996.

[PFR98] Christof Paar, Peter Fleischmann, and Peter Roelse. "Efficient Multiplier Architectures for Galois Fields GF(2^{4n})." *IEEE Trans. Comput.*, **47**(2):162–170, 1998.

[PL07] Steffen Peter and Peter Langendörfer. "An efficient polynomial multiplier in GF(2^m) and its application to ECC designs." In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pp. 1253–1258, San Jose, CA, USA, 2007. EDA Consortium.

[RCN03] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits, 2/E.* Prentice Hall, 2003.

[RG71] T. RAO and O. GARCIA. "Cyclic and multiresidue codes for arithmetic operations." *IEEE Transactions on Information Theory*, **17**(1):85 – 91, 1971.

[RH03] Arash Reyhani-Masoleh and M. Anwarul Hasan. "Error Detection in Polynomial Basis Multipliers over Binary Extension Fields." In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 515–528, London, UK, 2003. Springer-Verlag.

[RH04a] Arash Reyhani-Masoleh and M. Anwar Hasan. "Low Complexity Bit Parallel Architectures for Polynomial Basis Multiplication over GF(2^m)." *IEEE Trans. Computers*, **53**(8):945–959, 2004.

[RH04b] Arash Reyhani-Masoleh and M. Anwar Hasan. "Towards fault-tolerant cryptographic computations over finite fields." *ACM Trans. Embed. Comput. Syst.*, **3**(3):593–613, 2004.

[RH05] Arash Reyhani-Masoleh and M. Anwar Hasan. "Low Complexity Word-Level Sequential Normal Basis Multipliers." *IEEE Trans. Computers*, **54**(2):98–110, 2005.

[RH06] Arash Reyhani-Masoleh and M. Anwar Hasan. "Fault Detection Architectures for Field Multiplication Using Polynomial Bases." *IEEE Trans. Computers*, **55**(9):1089–1103, 2006.

[SBG02] Richard Schroeppel, Cheryl L. Beaver, Rita Gonzales, Russell Miller, and Timothy Draelos. "A Low-Power Design for an Elliptic Curve Digital Signature Chip." In Kaliski et al. [KKP03], pp. 366–380.

[SBP06] Kazuo Sakiyama, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. "Superscalar Coprocessor for High-Speed Curve-Based Cryptography." In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *Lecture Notes in Computer Science*, pp. 415–429. Springer, 2006.

[SK99] B. Sunar and Çetin. K. Koç. "Mastrovito Multiplier for All Trinomials." *IEEE Trans. Comput.*, **48**(5):522–527, 1999.

[SK01] Berk Sunar and Çetin Kaya Koç. "An Efficient Optimal Normal Basis Type II Multiplier." *IEEE Trans. Computers*, **50**(1):83–87, 2001.

[SSH99] Ivan Sutherland, Robert F. Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits.* Morgan Kaufmann Pub, 1999.

[ST03] Akashi Satoh and Kohji Takano. "A Scalable Dual-Field Elliptic Curve Cryptographic Processor." *IEEE Trans. Computers*, **52**(4):449–460, 2003.

[Sun04] Berk Sunar. "A Generalized Method for Constructing Subquadratic Complexity GF (2^k) Multipliers." *IEEE Trans. Comput.*, **53**(9):1097–1105, 2004.

[Syn08] Synplicity. *Synplicity FPGA Synthesis User Guide*, 2008.

[WHB02] Huapeng Wu, M. Anwar Hasan, Ian F. Blake, and Shuhong Gao. "Finite Field Multiplier Using Redundant Representation." *IEEE Trans. Computers*, **51**(11):1306–1316, 2002.

[WP06] André Weimerskirch and Christof Paar. "Generalizations of the Karatsuba Algorithm for Efficient Implementations." Cryptology ePrint Archive, Report 2006/224, 2006.

[WTS85] Charles C. Wang, Trieu-Kien Truong, Howard M. Shao, Leslie J. Deutsch, Jim K. Omura, and Irving S. Reed. "VLSI Architectures for Computing Multiplications and Inverses in GF(2^m)." *IEEE Trans. Computers*, **34**(8):709–717, 1985.

[Wu02] Huapeng Wu. "Bit-Parallel Finite Field Multiplier and Squarer Using Polynomial Basis." *IEEE Trans. Comput.*, **51**(7):750–758, 2002.