Finite Field Multiplier Architectures for Cryptographic Applications

by

Mohamed El-Gebaly

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Applied Science

in

Electrical Engineering

Waterloo, Ontario, Canada, 2000

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

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

Abstract

Security issues have started to play an important role in the wireless communication and computer networks due to the migration of commerce practices to the electronic medium. The deployment of security procedures requires the implementation of cryptographic algorithms. Performance has always been one of the most critical issues of a cryptographic function, which determines its effectiveness. Among those cryptographic algorithms are the elliptic curve cryptosystems which use the arithmetic of finite fields. Furthermore, fields of characteristic two are preferred since they provide carry-free arithmetic and at the same time a simple way to represent field elements on current processor architectures.

Multiplication is a very crucial operation in finite field computations. In this contribution, we compare most of the multiplier architectures found in the literature to clarify the issue of choosing a suitable architecture for a specific application. The importance of the measuring the energy consumption in addition to the conventional measures for energy-critical applications is also emphasized. A new parallel-in serial-out multiplier based on all-one polynomials (AOP) using the shifted polynomial basis of representation is presented. The proposed multiplier is area efficient for hardware realization. Low hardware complexity is advantageous for implementation in constrained environments such as smart cards.

Architecture of an elliptic curve coprocessor has been developed using the proposed multiplier. The instruction set architecture has been also designed. The coprocessor has been simulated using VHDL to very the functionality. The coprocessor is capable of performing the scalar multiplication operation over elliptic curves. Point doubling and addition procedures are hardwired inside the coprocessor to allow for faster operation.

Acknowledgements

All praise is due to Allah for guiding me throughout my life and giving me the ability to complete this work. I am at a loss of words to express my gratitude to my mother and my brother for their continuous love and support.

I am very fortunate to have had Prof. Hasan as my research advisor. This thesis would not have been possible without his support, encouragement, and patience of listening to my ideas.

Here in Waterloo I am grateful to all Waterloo faculty who have taught me, and my colleagues from whom I learned a lot. I want especially to mention Prof. Agnew and my colleague Amr Wassal for the useful discussions that helped me throughout this work.

To the memory of my father,

Maher Elgebaly.

Contents

List of Abbreviations

AOP	All-one polynomial
DB	Dual basis
DLP	Discrete logarithm problem
EC	Elliptic curve
ECC	Elliptic curve cryptosystem
ECDLP	Elliptic curve discrete logarithm problem
ESP	Equally-spaced polynomial
FSM	Finite state machine
GF	Galois field
GF(2^m)	Extension field of order m
KOA	Karatsuba-Ofman algorithm
LSB	Least-significant bit
MSB	Most-significant bit
NAF	Non-adjacent format
NB	Normal basis
PB	Polynomial basis
SPB	Shifted polynomial basis

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the tremendous growth of commerce transactions over wire and wireless media, the critical role that security plays is greatly emphasized. Electronic commerce practices are endangered by the possibility of unauthorized access, disclosure, alternation, substitution, or destruction of the information being transmitted. The necessity for security has fueled research in the area of cryptographic protocols and cryptographic algorithms.

Cryptographic computations are very intensive since the operand size is usually very large. This has led to the development of efficient hardware and software implementations to save system resources. In constrained environments such as mobile and portable devices, energy consumption is one of those resources that has to be optimized.

The use of elliptic curves (EC) in cryptography is promising for many reasons.

Elliptic curve cryptosystems (ECC) allow for shorter key lengths without compromising the security of the system. In comparison to more conventional methods of public key cryptographic protocols such as RSA and systems based on the discrete logarithm problem (DLP), key lengths are about 1024-bit, while EC systems , which are based on the elliptic curve discrete logarithm problem (ECDLP), use 160-bit operands. From a hardware point of view, this translates to increased performance, less area and lower bandwidth. From a security standpoint, ECC provide better long term security due to the lack of sub-exponential attacks which can be applied to DLP systems. ECC is currently being reviewed for standardization by the IEEE P1363 standards committee [20].

The elliptic curve cryptosystems which use the arithmetic of finite fields have been shown to have efficient implementations specially in constrained environments [25]. Furthermore, fields of characteristic two are preferred since they provide carry-free arithmetic and at the same time a simple way to represent field elements on current processor architectures. Addition in GF(2^m) can be as simple as bit-wise ex-or operations. However, finite field multiplication is much more difficult. Nevertheless, multiplication is an essential operation in finite field arithmetic since other operations such as inversion and exponentiation can be performed using repeated multiplication operations.

1.2 Hardware Cryptographic Architectures

Cryptographic computations are very demanding in terms of processing power and speed. This fact has led to the implementation of such systems on a hardware

chip rather than a software program. The chip is a piece of hardware dedicated to perform the computations in the underlying finite field. Many cryptographic chips have been implemented to speedup the cryptographic computations [21, 38] using parallel architectures to achieve the high speed required. In constrained environments such as smart cards, two very important design factors are to be considered: area and power consumption of the chip. Parallel architectures are not suitable for such environment since they consume much more area and power than what a device can support. As a result, bit or digit serial architectures are of more practical importance. To evaluate the hardware architectures suitable for a certain application, the following measures are to be considered:

- Hardware complexity (gate count).
- Time complexity (maximum delay).
- Regularity and Modularity.

In addition to the above measures, a very important metric in the evaluation process, especially for energy-critical applications, is the energy consumption. Chapter 3 shows the importance of the energy measure in evaluating multiplier architectures.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 provides a survey of most of the GF(2^m) multiplier architectures found in the literature. Based on the representations of the field elements, the architectures are grouped into four main categories,

namely, polynomial, normal, dual, and composite field multipliers. The comparison measures are the gate count and the critical path delay. Chapter 3 adds another performance measure to those discussed in Chapter 2. The conventional performance measures are found to be insufficient to select the most suitable architecture for a particular application. Energy consumption is shown to be a very critical performance measure for wireless and mobile applications. Extending the work done in [51], the dual basis multiplier is added to the comparison. Another group of multipliers based on AOPs are also added to the comparison.

A new serial multiplier architecture is proposed in Chapter 4. The proposed multiplier is based on all-one polynomials as the field defining polynomial and the field elements are represented in the shifted polynomial basis. The hardware complexity of the proposed architecture is efficient which is advantageous in constrained environments.

An elliptic curve coprocessor that is capable of performing the elliptic curve scalar multiplication operation is presented in Chapter 5. The main component inside the coprocessor architecture, the finite field multiplier, is the architecture proposed in Chapter 4. The datapath, the I/O unit, the control unit, as well as the instruction set architecture of the coprocessor are described.

Chapter 2

Architectural-Level Comparisons

2.1 Introduction

The importance of the multiplication operation amongst other finite field operations is greatly emphasized since finite field multiplication is a complex operation to perform. Many finite field multiplier architectures have been proposed in the literature based on different field bases or different application requirements. In most cases, gate count and critical path delays are used as complexity and performance measures to compare and evaluate different architectures. In this chapter, most of the GF(2^m) multiplier architectures found in the literature are clustered into four main groups, polynomial, normal, dual, and composite field multipliers. The multipliers within each group are compared in terms of the timing and area metrics.

With the emergence of wireless devices and the application of finite fields to securely and reliably transmitting data, the need for low-energy architectures and

implementations is much more emphasized. Conventional complexity and performance measures are no longer sufficient by themselves and have to be integrated with an energy metric. This metric is used to compare a group of multipliers in Chapter 3.

This Chapter is organized as follows. In Section 2.2, the mathematical background needed is quickly reviewed, presenting the different bases used in finite field arithmetic in general and multipliers in particular. Different architectural features that influence the choice of an architecture for a specific application are discussed in section 2.2.3. Performance and complexity metrics used to quantitatively differentiate multiplier architectures are also discussed introducing the energy-delay metric in that section. Section 2.3 provides a survey of most of the prominent multiplier architectures in the literature and compares them based on the conventional metrics.

2.2 Mathematical Background

To understand what finite field multipliers are about and how they work, a few mathematical concepts need to be reviewed [28, 45]. An *Abelian group* G is a set of elements together with a binary operation $*$ satisfying the following mathematical properties: closure, associativity, having an identity element, having inverses and commutativity. A *field* is a set F together with two operations, addition and multiplication such that F is an abelian group under addition with 0 as the identity element, the non-zero elements of F form an abelian group under multiplication with 1 as the identity element and the distributive law $a(b + c) = ab + ac$ holds

for all a, b, and c in the field. A field with a finite number of elements q is called a *finite field* of order q and is usually denoted by GF(q), i.e., Galois Field of order q. The order, q, must be a prime or power of prime to ensure that the field is a group under modulo-q multiplication. The binary field GF(2) and its extension GF(2^m) are of special interest because of their wide usage in computer hardware and communications equipment.

A polynomial $p(x)$ over GF(2) of degree m is said to be *irreducible* over GF(2) if $p(x)$ is not divisible by any polynomial over GF(2) of degree less than m but greater than zero. Also, an irreducible polynomial $p(x)$ of degree m is said to be *primitive* if the smallest positive integer n for which $p(x)$ divides $x^n + 1$ is $n = 2^m - 1$. An irreducible polynomial $p(x)$ of degree m is the generator of the extension field GF(2^m) if its nonzero elements are powers of α and α is a root of $p(x)$.

If $\{\beta_0, \beta_1, \ldots \beta_{m-1}\}$ is a basis of GF(2^m) over GF(2), each element $\alpha \in$ GF(2^m) can be uniquely represented in the form $\alpha = a_0\beta_0 + a_1\beta_1 + \ldots + a_{m-1}\beta_{m-1}$, where $a_i \in$ GF(2) for $0 \leq i \leq m-1$. Different multipliers use different bases and the choice of the underlying basis is heavily dependent on the application.

2.2.1 Bases of Representations

Polynomial Basis

This basis is also known as the *canonical* or *standard* basis. It is defined as the set $\{1, \alpha, \alpha^2, \ldots \alpha^{m-1}\}$, where α is a root of the irreducible polynomial $p(x)$ used to construct the field GF(2^m). In polynomial basis over GF(2^m), addition is simply bit-wise XORing. It is also worth noting that there is no carry to propagate in finite field

computations which means a smaller critical path compared to ordinary arithmetic operations. Polynomial basis multipliers are based on polynomial multiplication and modular reduction.

Normal Basis

This basis is given by the set $\{\alpha, \alpha^2, \ldots \alpha^{2^{m-1}}\}$ where $\alpha \in \mathrm{GF}(2^m)$. The concept of *Optimal Normal Basis* was introduced in [37] to reduce the complexity of multiplier architectures. Unfortunately, normal basis exists for approximately 23% of the fields $\mathrm{GF}(2^m)$, $2 \leqslant m < 1200$. Optimal normal basis has two types. Unlike type-II, type-I has very few irreducible polynomials.

Dual Basis

The concept of duality is defined as follows: Let $\{\alpha^j\}$ and $\{\beta_i\}$ to be two bases of representation for $\mathrm{GF}(2^m)$, Tr() is a linear function: $\mathrm{GF}(2^m) \rightarrow \mathrm{GF}(2)$. The bases $\{\alpha^j\}$ and $\{\beta_i\}$ are said to be *dual* with respect to Tr(), where $\{\alpha^j\}$ is a ploynomial basis, if [3]

$$\mathrm{Tr}(\alpha^j \beta_i) = \begin{cases} 1 & \text{if} \quad i = j, \\ 0 & \text{if} \quad i \neq j. \end{cases} \tag{2.1}$$

an extended definition of duality was given in [9]. Let $\gamma \in \mathrm{GF}(2^m)$, $\gamma \neq 0$, then

$$\mathrm{Tr}(\gamma\alpha_i\beta_i) = \begin{cases} 1 & \text{if} \quad i = j, \\ 0 & \text{if} \quad i \neq j. \end{cases} \tag{2.2}$$

The trace function over $\mathrm{GF}(2^m)$ is given by $\mathrm{Tr}(\alpha) = \alpha + \alpha^2 + \ldots + \alpha^{2^{m-1}}$. This basis was first used by Berlekamp [3] in the implementation of Reed-Solomon codecs. Another variation of the dual basis in the *Weakly Dual basis* introduced in [55].

Other Bases

Other bases that are less commonly used but are gaining more momentum include *Triangular basis* and *Redundant basis*. The triangular basis is similar to the dual basis in many aspects. This basis is the result of a pre-multiplication of *any* basis representation by a *triangular* transformation matrix, T, over GF(2) where

$$T = \begin{bmatrix} p_m & 0 & \ldots & 0 & 0 \\ p_{m-1} & p_m & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ p_2 & p_3 & \ldots & p_m & 0 \\ p_1 & p_2 & \ldots & p_{m-1} & p_m \end{bmatrix}$$

and the matrix entries p_i, $0 \leqslant i \leqslant m$ are the coefficients of the irreducible polynomial used to generate the field. For any irreducible polynomial $p(x)$, p_m is always 1 and T is guaranteed to be nonsingular. The advantage of this basis is that a

transform of coordinates to or from a polynomial basis can be done using shift registers with their connections determined by the irreducible polynomial used to construct the field [17]. It was used recently to build a variable dimension Galois Field coprocessor [13].

Another basis that has gained attention recently is the *Composite Field*, GF($(2^n)^m$) by extending the extension field GF(2^n) to degree m. Performing multiplication operations over large field polynomials by splitting those polynomials has been proposed in [41] based on the Karatsuba-Ofman algorithm (KOA) [26] for multiplication of large numbers. The architecture of the composite field multipliers is basically composed of multipliers and adders of the smaller order field represented in the polynomial [41] or the normal basis [43]. The complexity of a parallel multiplier implemented using KOA has reduced the complexity below the $O(m^2)$ bound [41].

2.2.2 Choices of Irreducible Polynomials

The choice of this polynomial greatly affects the complexity and regularity of the multiplier architecture, hence, special classes of irreducible polynomials are often used. However, using such special classes might limit the applications of the architecture, e.g., it might not be acceptable in some cryptographic applications. The availability or rarity of those polynomials for certain field dimensions also restricts their applications.

The *all-one polynomial*, AOP, $p(x) = 1 + x + x^2 + \ldots + x^m$ is irreducible if and only if $m + 1$ is a prime, $m + 1$ divides $2^m - 1$ and all the $(m + 1)th$ roots of unity are in GF(2^m) [35]. For $m \leq 100$, the AOP is irreducible for $m = 2, 4, 10,$

12, 18, 28, 36, 52, 58, 60, 66, 82 and 100. Polynomial basis multiplication based on the irreducible *trinomial* $x^m + x^k + 1$ with $1 \leq k \leq \lfloor m/2 \rfloor$, most commonly with $k = 1$ or $m/2$, are also attractive since they require fewer bit operations for modular reduction.

Another attractive case is the *Equally Spaced Polynomial* with a spacing s, s-ESP, is given by $1 + x^s + x^{2s} + \ldots + x^{ns}$. It has been shown that the ESP increases the regularity of the architecture to some extent [16, 23].

2.2.3 Performance and Complexity Metrics

Several performance and complexity metrics are used to compare and evaluate finite field multiplier architectures. These metrics and architectural features are reviewed below.

Gate Count

This is the main complexity metric which is usually given as the numbers of 2-input AND and XOR gates, flip-flops and switches or 2-to-1 multiplexers. It is sometimes tied to the silicon area used for implementation using the area and count of an equivalent 2-input NAND gate to represent the hardware complexity [51].

Throughput

Throughput is basically determined by the time required to complete a multiplication operation which is usually expressed in clock cycles. The clock period on the other hand is proportional to the critical path delay. The choice of an architecture

and an irreducible polynomial should try to minimize the critical path delay to decrease the clock period and increase the throughput. Also, pipelined architectures have the advantage of dividing the critical path delay over several stages, thus, increasing the clock frequency and the throughput. On the other hand, increasing the clock frequency has its negative effect in terms of power dissipation.

Latency

The delay between the first input and the first output of the multiplier expressed in clock cycles is defined here as the latency. This measure is of special importance in the semi-systolic and systolic architectures where the output experiences a delay of a number of clock cycles after the arrival of the input.

Power Dissipation and Energy

As described above, a quantitative approach is needed to select appropriate architectures for energy starved applications such as wireless and mobile applications. One approach was to seek the primitive polynomial that minimizes the power dissipation by reducing the switching activity [46]. However, this approach uses an exhaustive search to find the optimal polynomial which is not feasible for very large field dimensions. Another approach tries to minimize the power-delay product, and hence the energy, for an architecture through the logic style used in the implementation [51]. There is always a power versus delay tradeoff which governs practical VLSI architectures and minimizing their product achieves the best trade-off between power consumption and speed and conserves energy resources.

Regularity and Modularity

Although subjective, this is also a very important metric. Many applications use very large field dimensions which makes a regular multiplier that can be implemented in bit-slices a very attractive option. Polynomial basis multipliers are the best in terms of regularity while normal basis multipliers are the worst. Regularity usually affects performance positively too.

2.3 GF(2^m) Multiplier Architectures

This section compares most of the multiplier architectures found in the literature in terms of the architectural features previously mentioned. A previous comparison [19] has only covered three multiplier architectures, Berlekamp [3], Massey-Omura [29], and Scott-Tavares-Peppard [44]. The VLSI chip area was compared for the three multipliers for order $m = 8$. The dual basis multiplier by Berlekamp was the most efficient architecture.

2.3.1 Polynomial Basis Multipliers

Polynomial basis multipliers are the most common multipliers in the literature. Several architectures, serial or parallel, systolic or non-systolic, have been developed mainly because the polynomial basis is the simplest to represent. Throughout this section, the irreducible polynomial is referred to as $p(x) = 1 + p_1 x + p_2 x^2 + ... + x^m$ and for an AOP, $p(x) = 1 + x + x^2 + ... + x^m$. The multiplier operands will be referred to as A and B where $A = a_0 + a_1\alpha + a_2\alpha^2 + ... + a_{m-1}\alpha^{m-1}$ and

$B = b_0 + b_1\alpha + b_2\alpha^2 + ... + b_{m-1}\alpha^{m-1}$.

Non-Systolic Architectures

Many non-systolic polynomial basis multipliers have been developed mainly because their hardware complexity is smaller compared to the systolic architectures. Most of these architectures depend heavily on the choice of the irreducible polynomial used to generate the field. Selecting certain irreducible polynomials greatly simplifies the underlying architecture and increases its regularity and modularity. Choosing an AOP increases the regularity while using a trinomial has been shown to produce hardware efficient architectures [5, 14, 30, 48]. Using these special polynomials, on the other hand, puts restrictions on the order of the finite field used since those polynomials are irreducible only for certain orders.

Mastrovito [30] presented a parallel multiplier based on the reduction of the product polynomial from a degree of $2m-2$ to $m-1$. Consider that $C = AB$ mod $p(x)$ is the product of A and B reduced modulo the irreducible polynomial $p(x)$. The product polynomial, $d = (\sum_{i=0}^{m-1} a_i\alpha^i)(\sum_{j=0}^{m-1} b_j\alpha^j)$ which is of degree at most $2m-2$ is first computed. The product polynomial d is then reduced to an $m-1$ degree polynomial using the *multiplication matrix*, Z, $C = ZA$. The reduction process is performed through producing a *reduction matrix*, Q, to reduce the elements of orders m or higher to orders $\leq m-1$. This can be accomplished

by computing the coefficients of the reduction matrix Q as follows

$$\begin{bmatrix} \alpha^m \\ \alpha^{m+1} \\ \vdots \\ \alpha^{2m-2} \end{bmatrix} = Q \begin{bmatrix} \alpha^{m-1} \\ \alpha^{m-2} \\ \vdots \\ 1 \end{bmatrix} \tag{2.3}$$

The coefficients of the matrix Q depend on the choice of the irreducible polynomial used to generate the field. Selecting trinomials of the form x^m+x+1 or $x^m+x^{m/2}+1$ has been shown to reduce the number of terms in the Q matrix and therefore reduce the overall complexity of the multiplier [30].

Recently, the hardware complexity of the Mastrovito multiplier for the trinomial of the form $x^m + x^n + 1$ for $1 \leq n \leq m-1$ has been shown to be the same as that of the original Mastrovito multiplier [48]. It was also shown that the multiplication matrix Z can be constructed from three simpler matrices. For the special case of $k = m/2$, the number of the required XOR gates is reduced. The time complexity in that special case is also greatly reduced.

A generalized Mastrovito multiplier has been introduced in [47] for which the generating polynomial is of the form $p(x) = x^m + x^k + \sum_{i=K+1}^{k-1} p_i x^i + \sum_{i=0}^{K-1} x^i$. The multiplier has a complexity proportional to $(m - 1 - H(p))$, where $H(p)$ is the Hamming weight of the underlying irreducible polynomial. However, the original Mastrovito multiplier has a complexity proportional to $H(p)$. The multiplier proposed in [47] has a low hardware complexity if the Hamming weight of the generating polynomial is high while the original Mastrovito multiplier has a lower

complexity for low Hamming weights. For the AOP case, where $H(p) = m-1$, the generalized multiplier has a lower complexity that is exactly the same as the one proposed in [5].

Another category of irreducible polynomials commonly used in polynomial basis multipliers is the AOP. Using AOPs can considerably enhance the modularity of the architecture. For example, Hasan and Bharagava [15] presented a serial AOP multiplier which is based on a multiplication matrix. Assuming that $C = \sum_{i=0}^{m-1} c_i \alpha^i$ is the product of A and B where A, B, and C are all elements in GF(2^m). The product C can be computed using

$$\begin{bmatrix} \hat{c}_0 \\ \hat{c}_1 \\ \vdots \\ \hat{c}_{m-1} \end{bmatrix} = \begin{bmatrix} \hat{a}_0 & \hat{a}_2 & \dots & \hat{a}_{m-1} \\ \hat{a}_1 & \hat{a}_3 & \dots & \hat{a}_m \\ \vdots & \vdots & \ddots & \vdots \\ \hat{a}_{m-1} & \hat{a}_{m+1} & \dots & \hat{a}_{2m-2} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix} \tag{2.4}$$

where

$$\hat{a}_k = \sum_{i=0}^{m-1} a_i t_{m-1}^{[i+k]} \qquad 0 \leq k \leq 2m-2$$

and

$$\hat{c}_i = \begin{cases} c_{m-1} & i = 0 \\ c_{m-1-i} + \sum_{l=1}^{i} \hat{c}_{i-l} p_{m-l} & i = 1, 2, ..., m-1 \end{cases}$$

with $t_i^{[k]}$ representing the ith coordinate of the element α^k. The matrix in (2.4)

was used to construct a serial multiplier in [15]. The matrix multiplication can be realized using a *Linear-Feedback Shift Register* (LFSR) and m AND gates. Another unit is required to perform the partial products accumulation. An AOP was used in [16] as the irreducible polynomial to construct a parallel multiplier. It was shown that the use of an AOP greatly simplifies the construction of the multiplication matrix and increases the modularity of the design.

Itoh and Tsujii [23] also proposed a parallel multiplier based on AOPs. The operands and the product are expressed in a modified version of the polynomial basis representation. For example, for $A \in \mathrm{GF}(2^m)$, A is expressed as $A = A_0 + A_1\alpha + \ldots + A_{m-1}\alpha^{m-1} + A_m\alpha^m$, where $A_m = 0$. Hence, the product $C = AB$ can be written as

$$\begin{bmatrix} C_m \\ C_{m-1} \\ \vdots \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & \ldots & A_m \\ A_m & A_0 & \ldots & A_{m-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_2 & A_3 & \ldots & A_1 \\ A_1 & A_2 & \ldots & A_0 \end{bmatrix} \begin{bmatrix} B_m \\ B_{m-1} \\ \vdots \\ B_1 \\ B_0 \end{bmatrix} \tag{2.5}$$

A serial multiplier was derived from (2.5) in [11]. In that structure the operands A and B are represented in the polynomial basis form while the product C is

represented in the Itoh-Tsujii's form as follows

$$\begin{bmatrix} C_m \\ C_{m-1} \\ \vdots \\ C_0 \end{bmatrix} = \begin{bmatrix} A_1 & A_2 & \dots & 0 \\ A_0 & A_1 & \dots & A_{m-1} \\ 0 & A_0 & \dots & A_{m-2} \\ \vdots & \vdots & \ddots & \vdots \\ A_2 & A_3 & \dots & A_0 \end{bmatrix} \begin{bmatrix} B_{m-1} \\ B_{m-2} \\ \vdots \\ B_0 \end{bmatrix} \tag{2.6}$$

Equation (2.6) is obtained by the application of $A_m = B_m = 0$ to (2.5). Note that $A_i = a_i, B_i = b_i$ and $C_i = c_i$ for $i = 0, 1, \dots, m-1$. The product C is converted to the polynomial basis form using the fact that $c_i = C_i + C_m$ and the computation of $C_m = \sum_{i=1}^{m-1} A_i B_{m-i}$ [23]. This multiplier requires m clock cycles to produce the output sequence while the original architecture by Itoh and Tsujii requires $m + 1$ clock cycles.

Table 2.1: Non-systolic polynomial basis GF(2^m) multiplier architectures

Authors [Ref.]	S/P	Number of Gates				Latency	Comp. Time	Critical Path	Irreducible Polynomial
		AND	XOR	FF	MUX				
Mastrovito [30]	S	$2m$	$2m$	$2m+1$	m	m	$m+1$	$T_A + T_{X_2}$	Any
Hasan and Bhargava [15]	S	$3m$	$\frac{(m-1)}{2[H(p)-2]}+1$	$4m+1$	1	$2m$	$2m+1$	$T_A + log_2(m-1) + T_{X_2}$	Any
Itoh and Tsujii [23]	P	$m^2 + 2m+1$	m^2+2m	$2m$	0	1	T_{CP} *	$T_A + (\lfloor log_2(m) + log_2(m+2) \rfloor) T_{X_2}$	AOP
Mastrovito [30]	P	m^2	m^2-1	$2m$	0	1	T_{CP}	$T_A + (\lceil log_2(m) + 1 \rceil) T_{X_2}$	Any
Hasan and Bhargava [14]	P	m^2	m^2+m-2	$2m$	0	1	T_{CP}	$T_A + (m + \lceil log_2(m-1) \rceil) T_{X_2}$	AOP
Koç and Sunar [5]	P	m^2	m^2-1	$2m$	0	1	T_{CP}	$T_A + (2 + \lceil log_2(m-1) \rceil) T_{X_2}$	AOP

* T_{CP} = Critical path delay.

Koç and Sunar [5] utilized the *Shifted Polynomial Basis* representation and Mastrovito's multiplication algorithm to construct an AOP multiplier. It has been shown that the multiplication matrix Z in [30] can be divided into two simpler matrices $Z_1 + Z_2$.Those two matrices can be easily computed when an AOP is used as the irreducible polynomial. The gate and time complexities of this multiplier are better than that of Mastrovito's. The architecture has an extra advantage that it can perform multiplication in the normal basis as well as the polynomial basis. The normal basis multiplication is performed by adding a permutation circuit to the inputs and the inverse of that permutation to its output. The permutation circuit does not add any gate complexity to the multiplier since it can be done by rewiring. The normal basis version of the architecture proposed in [5] will be discussed in more details in Section 2.3.2. The non-systolic polynomial basis architectures covered in this chapter are compared quantitatively in Table 2.1.

Systolic Architectures

Systolic architectures are advantageous in some applications because they are easy to pipeline and to expand. The basic metric in measuring the hardware complexity of systolic architectures is the complexity of the basic cell. The critical path delay is also the critical delay of the basic unit. Due to the existence of pipelining registers in the data path, systolic architectures suffer from longer latency and longer initial delay.

Most of the systolic multipliers are based on array-type multiplication where one of the operands is processed in parallel and the other is processed one bit at a

time. Depending upon the order of processing of the second operand, the array-type algorithms are classified as *least-significant bit first* (LSB-first) and *most-significant bit first* (MSB-first) schemes. The LSB-first scheme processes the LSB of the second operand first while the MSB-first scheme processes the MSB first [24].

Assuming that A, B, and $C \in \mathrm{GF}(2^m)$ are represented in the polynomial basis and $p(x)$ is the irreducible polynomial, the product C of A and B can be written as

$$\begin{aligned} C &= AB \bmod p(x) \\ &= b_0 A + b_1(A\alpha \bmod p(x)) + b_2(A\alpha^2 \bmod p(x) + \ldots + b_{m-1}(A\alpha^{m-1} \bmod 7\ p(x)) \\ &= \sum_{k=0}^{m-1}(A\alpha^k)b_k = \sum_{i=0}^{m-1}(\sum_{k=0}^{m-1} a_i^{(k)}\alpha^i)b_k \\ &= \sum_{i=0}^{m-1}(\sum_{k=0}^{m-1} a_i^{(k)} b_k)\alpha^i \bmod p(x) \end{aligned} \tag{2.7}$$

where $A\alpha^k = \sum_{i=0}^{m-1} a_i^{(k)}\alpha^i$ $(0 \le k \le m-1)$.

The computation in (2.7) is called the LSB-first scheme and can be performed recursively on k for $0 \le k \le m-1$ as follows [57]

$$\begin{aligned} A^{(k)} &= [A^{(k-1)}]\alpha \bmod p(x), \\ C^{(k)} &= A^{(k-1)} b_{k-1} + C^{(k-1)}, \end{aligned}$$

where $C^{(k)} = \sum_{i=0}^{k-1} Ab_i\alpha^i$ and $C^{(0)} = 0$, $A^{(k)} = \alpha^k$ and $A^{(0)} = A$.

For $k = 0$, we have $A\alpha^0 = A$ while for $1 \leq k \leq m-1$, we have

$$\begin{aligned} A^{(k)} = A\alpha^k = (A\alpha^{k-1})\alpha = \sum_{i=0}^{m-1} a_i^{(k-1)}\alpha^{i+1} \\ = a_{m-1}^{(k-1)}\alpha^m + \sum_{i=1}^{m-1} a_{i-1}^{(k-1)}\alpha^i \end{aligned} \tag{2.8}$$

Since $p(\alpha) = 0$, $\alpha^m = p_{m-1}\alpha^{m-1} + p_{m-2}\alpha^{m-2} + ... + p_1\alpha + p_0$. Substituting α^m into (2.8) yields

$$A^{(k)} = \sum_{i=1}^{m-1}(a_{i-1}^{(k-1)} + a_{m-1}^{(k-1)}p_i)\alpha^i + a_{m-1}^{(k-1)}p_0. \tag{2.9}$$

From (2.9), we can write

$$\begin{aligned} a_i^{(k)} &= \begin{cases} a_{i-1}^{(k-1)} + a_{m-1}^{(k-1)}p_i & 1 \leq i \leq m-1 \\ a_{m-1}^{(k-1)}p_0 & i = 0, \end{cases} \\ c_i^{(k)} &= a_i^{(k-1)}b_{k-1} + c_i^{(k-1)}, \end{aligned} \tag{2.10}$$

where $a_i^{(k)}$ and $c_i^{(k)}$ denote the ith coefficients in $A^{(k)}$ and $C^{(k)}$ respectively and the final product C is $C^{(m)}$.

The MSB-first scheme uses the MSB as the first bit in the multiplication operation to produce the product as follows

$$C = (...(Ab_{m-1}\alpha \bmod p(x) + Ab_{m-2})\alpha \bmod p(x) + \ldots + Ab_1)\alpha \bmod p(X) + Ab_0. \tag{2.11}$$

The basic kth step in the MSB-bit algorithm performs the following computation:

$$C^{(k)} = C^{(k-1)}\alpha \bmod p(x) + Ab_{m-k}, \tag{2.12}$$

where $C^{(k)} = \sum_{i=1}^{k} Ab_{m-i}\alpha^{k-i}$, and $C^{(0)} = 0$. Using the fact that $\alpha^m = \sum_{i=0}^{m-1} p_i\alpha^i$, the coefficients of $C^{(k)}$ can be written as

$$c_i^{(k)} = \begin{cases} c_{i-1}^{(k-1)} + c_{m-1}^{(k-1)}p_i + a_i b_{m-k} & 1 \leq i \leq m-1 \\ c_{m-1}^{(k-1)}p_0 + a_0 b_{m-k} & i = 0. \end{cases} \tag{2.13}$$

The recursive equation (2.13) can be implemented using the architecture shown in Figure 2.1(a). The basic cell structure is shown in Figure 2.1(b).

The operations performed in both array-multiplication algorithms can be identified as *multiply-by-α*, *generate-current-partial-products* and *accumulate-to-previuos-result* [24]. The multiply-by-α operation is common in both schemes. In the LSB-first scheme, the three operations are performed in parallel while in the MSB-first scheme they are performed sequentially. Parallelism in the LSB-first scheme leads to efficient implementations with less area complexity than the MSB-first scheme. The LSB-first and the MSB-first schemes can be easily mapped into serial or parallel VLSI implementations. The choice of the implementation depends heavily on the nature of the application and the availability of the input operands at the beginning of computation.

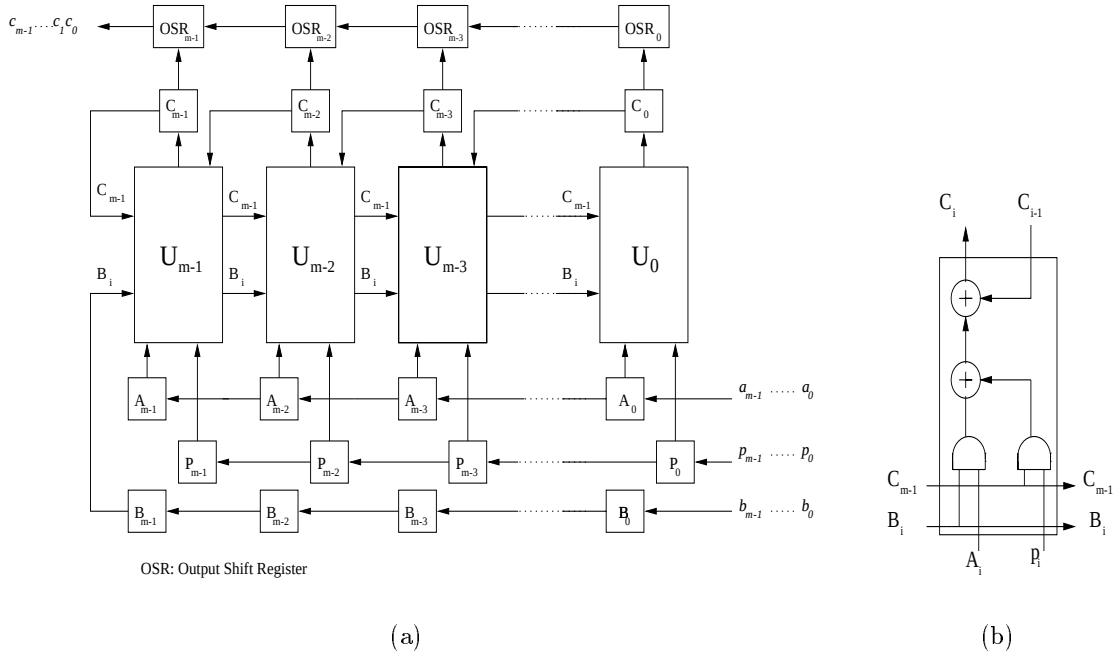


Figure 2.1: MSB-first multiplier architecture

In [57], the LSB-first scheme was used to implement a serial and a parallel systolic array multipliers. These multipliers can perform the product-sum operation, $P = AB + C$, in GF(2^m). Jain *et.al.* [24] proposed a semi-systolic multiplier architecture. Semi-systolic architectures have lower latency and smaller number of latches compared to those of the systolic architectures.

The MSB-first scheme was utilized in the implementation of several systolic polynomial basis multipliers. In [44], a bit-slice architecture for a serial-in serial-out multiplier was implemented. The architecture has two global control signals which introduce synchronization problems when the chip becomes large for large field dimensions. The first serial systolic architecture reported in the literature was due

to Zhuo [58]. It has only one control signal and a very efficient architecture. Other implementations using systolic array architectures were proposed in [49] and [54]. The two implementations has about the same area and timing complexity. Only one control signal is used in both designs which gives these architectures an advantage over the design presented in [44]. In [7], a systolic product-sum architecture was presented that is less complex than that proposed in [57]. However, they both have the same critical path delay.

In [31], an area efficient architecture was proposed utilizing the MSB-scheme to implement a serial systolic multiplier and another variation of it in the form of a serial/parallel systolic architecture. The design uses two bits of one operand as inputs to the basic cell. Hence, the required number of basic cells in the multiplier is reduced by half i.e. $m/2$ instead of m. The overall complexity of the multiplier is better than other designs but the critical path delay is longer. Also, two control signals are required to produce the output.

Table 2.2 shows a comparison between most of the systolic architectures proposed in the literature. The entries of the table represent the whole architecture rather than the basic cell.

Table 2.2: Systolic polynomial basis GF(2^m) multiplier architectures

Authors [Ref.]	S/P	Number of Gates				Latency	Comp. Time	Critical Path
		AND	XOR	FF	MUX			
Scott and Tavares [44]	S	$2m$	$(m-1)X_3$	$4m+2$	$2m$	m	$m+1$	$T_A + T_{X_3}$
Yeh and Reed [57]	S	$3m$	$2m$	$11m$	m	$2m$	$3m$	$T_A + 2T_{X_2} + T_{MUX}$
Zhou [58]	S	$3m$	$2m$	$7m$	m	$2m$	$3m-1$	$T_A + 2T_{X_2} + T_{MUX}$
Diab [7]	S	$3m$	$2m$	$5m$	m	$2m$	$3m$	$2T_A + 2T_{X_2} + T_{MUX}$
Wang and Lin [49]	S	$3m$	m of X_3	$10m$	$2m$	$2m$	$3m$	$T_A + T_{X_3} + T_{MUX}$
Mekhallalati *et.al.* [31]	S	$6\frac{m}{2}$	$2\frac{m}{2}$ of X_3	$16\frac{m}{2}$	$4\frac{m}{2}$	$3\frac{m}{2}$	$5\frac{m}{2}$	$2T_A + T_{X_3} + T_{MUX}$
Wu and Chang [54]	S	$3m-2$	$2m-1$	$10m-9$	$2m-1$	$2m-1$	$3m-2$	$2T_A + 2T_{X_2} + T_{MUX}$
Yeh and Reed [57]	P	$2m^2$	$2m^2$	$7m^2$	0	$2m$	$3m$	$T_A + T_{X_2}$
Wang and Lin [49]	P	$2m^2$	m^2 of X_3	$7m^2$	0	$2m$	$3m$	$2T_A + T_{X_3}$
Jain *et al.* [24]	P	$2m^2$	$2m^2$	$6m^2$	0	m	$m+1$	$T_A + T_{X_2}$
Wu and Chang [54]	P	$2m^2-m$	$2m^2-m$	$8m^2-7m$	0	$2m-1$	$3m-1$	$T_A + T_{X_2}$

2.3.2 Normal Basis Multipliers

An element $B \in \mathrm{GF}(2^m)$ is represented using the normal basis as

$$\begin{aligned} B &= b_0\alpha + b_1\alpha^2 + b_2\alpha^4 + \ldots + b_{m-1}\alpha^{2^{(m-1)}} \\ &= [b_0, b_1, b_2, \ldots, b_{m-1}] \cdot [\alpha, \alpha^2, \alpha^4, \ldots, \alpha^{2^{(m-1)}}]^t \\ &= \mathbf{b} \cdot \boldsymbol{\alpha}^t, \end{aligned} \tag{2.14}$$

or more simply by the vector of coordinates, $\mathbf{b}$, for the normal basis representation.

A powerful feature of the normal basis representation is that squaring of an element B is simply a cyclic shift of its coordinates which can be implemented using a binary shift register. Since $\alpha^{2^m} = \alpha$, B^2 can be represented as

$$\begin{aligned} B^2 &= b_0\alpha^2 + b_1\alpha^4 + \ldots + b_{m-2}\alpha^{2^{m-1}} + b_{m-1}\alpha^{2^m} \\ &= b_{m-1}\alpha + b_0\alpha^2 + b_1\alpha^4 + \ldots + b_{m-2}\alpha^{2^{m-1}} \\ &= \mathbf{b}^{(1)} \cdot \boldsymbol{\alpha}^t, \end{aligned} \tag{2.15}$$

where $\mathbf{b}^{(k)}$ represents the k-fold right cyclic shift of $\mathbf{b}$.

Massey and Omura [29] proposed a normal basis multiplier based on the following principle. If $\mathbf{a} = [a_0, a_1, a_2, \ldots, a_{m-1}]$ and $\mathbf{b} = [b_0, b_1, b_2, \ldots, b_{m-1}]$ are the vector representations of the coordinates of two elements $A, B \in \mathrm{GF}(2^m)$ in a nor-

mal basis form, then their product, C, can be written as

$$\begin{aligned} C &= \mathbf{a}\boldsymbol{\alpha}^t(\mathbf{b}\boldsymbol{\alpha}^t) \\ &= \mathbf{a}\boldsymbol{\alpha}^t\boldsymbol{\alpha}\mathbf{b}^t \\ &= \mathbf{a}\mathbf{M}\mathbf{b}^t, \end{aligned} \tag{2.16}$$

where the product matrix, $\mathbf{M}$, is defined by

$$\begin{aligned} \mathbf{M} = \boldsymbol{\alpha}^t\boldsymbol{\alpha} &= \begin{bmatrix} \alpha^{2^0+2^0} & \alpha^{2^0+2^1} & \dots & \alpha^{2^0+2^{m-1}} \\ \alpha^{2^1+2^0} & \alpha^{2^1+2^1} & \dots & \alpha^{2^1+2^{m-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha^{2^{m-1}+2^0} & \alpha^{2^{m-1}+2^1} & \dots & \alpha^{2^{m-1}+2^{m-1}} \end{bmatrix} \\ &= \mathbf{M}_0\alpha + \mathbf{M}_1\alpha^2 + \ldots + \mathbf{M}_{m-1}\alpha^{2^{m-1}}, \end{aligned} \tag{2.17}$$

where the element at row i and column j of $\mathbf{M}_k$ is in GF(2) and represents the coefficient α^{2^k} when $\alpha^{2^i+2^j}$ is represented using the normal basis and $k = 0, 1, \ldots, m-1$. From the above equations, the coordinates of the product can be obtained using

$$\begin{aligned} c_{m-1-k} &= \mathbf{a}\mathbf{M}_{m-1-k}\mathbf{b}^t, \quad k = 0, 1, \ldots, m-1, \\ &= \mathbf{a}^{(k)}\mathbf{M}_{m-1}\mathbf{b}^{(k)t}. \end{aligned} \tag{2.18}$$

Hence, the same logic function used to implement equation (2.18) can be used to compute all the coordinates of C in serial from the cyclically shifted coordinates of A and B. A parallel multiplier can also be achieved by using m identical replicas of

that same logic to simultaneously calculate the coordinates of C using shifted wiring for the inputs A and B. The main disadvantage of Massey-Omura multipliers is that the function implementing equation (2.18) depends heavily on the choice of the polynomial. Therefore, the structure is irregular and cannot be expanded easily to high order fields. A general architecture for the Massey-Omura serial multiplier is shown in Figure 2.2.

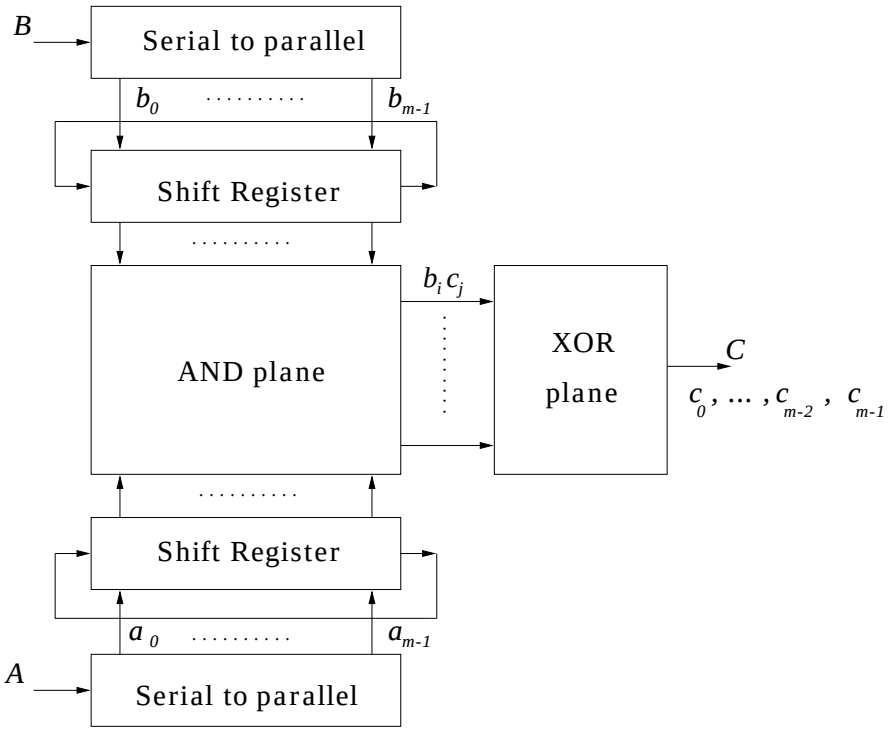


Figure 2.2: Massey-Omura serial multiplier

Wang *et al.* [50] proposed a VLSI pipelined implementation of both the serial and parallel versions using an AND-XOR implementation of equation (2.18) with pipelining registers between the different levels of the XOR tree. However, this

implementation increases the design area and power considerably for large fields. They have also pipelined the input into the shift registers in such a way that there is no time lost between operations except for an initial fixed time delay.

Hasan *et al.* [18] proposed a modified Massey-Omura multiplier based on choosing an irreducible AOP as the generating polynomial. This choice allows to write $\mathbf{M}_{m-1}$ as a sum of two matrices, $\mathbf{P} + \mathbf{Q}$, as follows

$$\mathbf{M}_{m-1} = \mathbf{P} + \mathbf{Q} \pmod 2, \tag{2.19}$$

where the elements of the matrix $\mathbf{P}$ are given by

$$p_{i,j} = \begin{cases} 1 & \text{if } i = \frac{m}{2} + j \pmod m, \\ 0 & \text{otherwise.} \end{cases} \tag{2.20}$$

Using equation (2.19), it was shown that equation (2.18) can be written as

$$c_{m-1-k} = \mathbf{a}\mathbf{P}\mathbf{b}^t + \mathbf{a}^{(k)}\mathbf{Q}\mathbf{b}^{(k)t} \pmod 2. \tag{2.21}$$

The proposed parallel architecture introduced a significant reduction in the hardware complexity compared to the original parallel Massey-Omura multiplier because the first term in the above equation is independent of k and needs to be computed only once for all of the product coordinates. However, the critical path delay is still the same as that in [50]. The only restriction on this architecture is in the use of AOPs as the generating polynomials.

Agnew *et al.* [1] presented a serial normal basis multiplier that has a regular

architecture and therefore suitable for VLSI implementations. The multiplication algorithm is based on that of Massey-Omura but the authors ended up with a regular architecture. Their algorithm starts by writing the coordinates of the product $C = AB$ in a *bilinear form* of the coordinates of A and B.

$$C = AB = \sum_{k=0}^{m-1} c_k \alpha^{2^k} = \sum_{0 \le i,j \le m-1} a_i b_j \alpha^{2^i} \alpha^{2^j}$$

Hence,

$$c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} t_{i,j}^{[k]} a_i b_j \quad \text{for } k = 0, 1, \ldots, m-1 \tag{2.22}$$

where $\alpha^{2^i} \alpha^{2^j} = \sum_{k=0}^{m-1} t_{i,j}^{[k]} \alpha^{2^k}, \quad i, j = 0, 1, \ldots, m-1$. Therefore $t_{i,j}^{[k]}$ is the element at row i and column j in $\mathbf{M}_k$ of equation (2.17). Similar to equation (2.18), c_k can be written as

$$c_k = \sum_{j=0}^{m-1} b_{j+k} \sum_{i=0}^{m-1} t_{i,j}^{[0]} a_{i+k} \tag{2.23}$$

where all the indices are reduced modulo m.

Equation (2.23) is another representation of Massey-Omura algorithm in (2.18). In [1], the function $F_j^{[k]}(t)$ is defined by

$$F_j^{[k]}(t) = b_{j+k+t} \sum_{i=0}^{m-1} t_{i,j}^{[0]} a_{i+k+t}. \tag{2.24}$$

Consequently,

$$c_k = \sum_{j=0}^{m-1} F_j^{[k]}(0). \tag{2.25}$$

The coordinates of A and B are stored in two registers **A** and **B** which are shifted cyclically. At cycle t, $\sum_{j=0}^{t-1} F_j^{(0)}(0), \ldots, \sum_{j=0}^{t-1} F_j^{(m-1)}(0)$ are formed in a special structure **C** that implements equation (2.24) from its previous contents and the current contents of register **A**. After m clock cycles, the result is stored in **C**. This multiplier architecture has a lower hardware complexity than that presented by Massey-Omura. The hardware complexity can be reduced further by using the optimal normal basis [37] as the basis of the multiplier.

As mentioned earlier, Koç and Sunar [5] proposed a normal basis multiplier based on their polynomial basis one. The generating polynomial must be an AOP. This allows them to write another set

$$\Omega = \{\alpha, \alpha^2, \alpha^3, \ldots, \alpha^m\}, \tag{2.26}$$

and use it as a basis to represent the elements of GF(2^m). This basis is a shifted version of the polynomial basis. The normal basis representation of an element A or $B \in$ GF(2^m) can be converted to that representation using the following relation

$$B = \sum_{i=0}^{m-1} b_i \alpha^{2^i} = \sum_{i=1}^{m} b'_i \alpha^i. \tag{2.27}$$

This conversion can be simply implemented using the permutation given by

$$b'_{2^i \bmod (m+1)} = b_i \quad \text{for} \quad i = 0, 1, \ldots, m-1. \tag{2.28}$$

This permutation is conducted at the input bits simply by rewiring them without any additional gates. The output of the polynomial basis multiplier is then $C = AB/\alpha^2$. Multiplying the output of the polynomial basis multiplier by α^2 and applying the inverse permutation results in the product in the normal basis representation. This results in the same hardware complexity as in the polynomial basis multiplier since the inverse permutation does not use any additional gates. Therefore, the proposed multiplier has the same architectural complexity and critical path delay as the polynomial basis multiplier. It can be shown that Hasan's multiplier [16] has exactly the same hardware complexity as the one proposed by Koç [5] however Hasan's multiplier has a better critical delay.

Table 2.3 shows a comparison between the normal basis multipliers covered in this chapter.

Table 2.3: Normal basis GF(2^m) multiplier architectures[1]

Authors [Ref.]	S/P	Number of Gates				Latency	Comp. Time	Critical Path	Irreducible Polynomial
		AND	XOR	FF	MUX				
Wang *et al.* [50]	S	$2m-1$	$2m-2$	2m	0	m	m	$T_A+(1+\lceil log_2(m-1)\rceil)T_{X_2}$	Any[2]
Agnew *et al.* [1]	S	m	$2m-1$	3m	0	m	m	$T_A+(1+\lceil log_2(m-1)\rceil)T_{X_2}$	Any
Wang *et al.* [50]	P	$2m^2-m$	$2m^2-2m$	$2m$	0	0	T_{CP}	$T_A+(1+\lceil log_2(m-1)\rceil)T_{X_2}$	Any[2]
Hasan *et al.* [18]	P	m^2	m^2-1	$2m$	0	0	T_{CP}	$T_A+(1+\lceil log_2(m-1)\rceil)T_{X_2}$	AOP
Koç and Sunar [5]	P	m^2	m^2-1	$2m$	0	0	T_{CP}	$T_A+(2+\lceil log_2(m-1)\rceil)T_{X_2}$	AOP

[1] All of the normal basis architectures that we were able to find in the literature were non-systolic.

[2] The figures prenested here correspond to using the optimal normal basis [30].

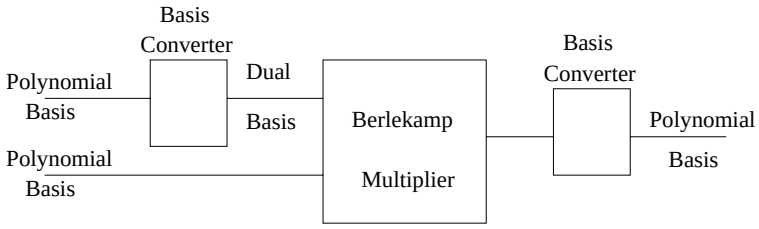


Figure 2.3: Berlekamp multiplier configured for polynomial basis multiplication

2.3.3 Dual Basis Multipliers

Dual basis multipliers, specially bit-serial ones, are known to have the lowest hardware complexity of all available GF(2^m) multipliers and to be particularly suited for constant multiplication [9]. The dual basis representation was first utilized in finite field multiplication by Berlekamp [3]. A setup for Berlekamp serial multiplier to perform polynomial basis multiplication is shown in Figure 2.3.

Consider the set $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ to be the dual basis of the polynomial basis $\{1, \alpha, \ldots, \alpha^{m-1}\}$, where α is a root of the polynomial $p(x)$. Using the definition of duality [9] in (2.2), the multiplication operation $C = AB$ where $A = \sum_{i=0}^{m-1} a_i \alpha^i$ is represented in the polynomial basis while $B = \sum_{j=0}^{m-1} b_j \beta_j$ and $C = \sum_{k=0}^{m-1} c_k \beta_k$ are in the dual basis, is performed through a matrix multiplication as follows:

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} b_0 & b_1 & \ldots & b_{m-1} \\ b_1 & b_2 & \ldots & b_m \\ \vdots & \vdots & \ddots & \vdots \\ b_{m-1} & b_m & \ldots & b_{2m-2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{bmatrix}, \tag{2.29}$$

where $b_k = \mathrm{Tr}(B\gamma\alpha^k)$ $(k = 0, 1, \ldots, 2m-2)$, $c_k = \mathrm{Tr}(C\gamma\alpha^k)$ $(k = 0, 1, \ldots, m-1)$, b_k and c_k are the dual basis coordinates of B and C respectively. The b_k $(k = m, m+1, \ldots, 2m-2)$ can be generated using an LFSR initialized with the coordinates b_k $(k = 0, 1, \ldots, m-1)$ and the feedback connections corresponding to the nonzero terms of $p(x)$. For example, $b_{m+k} = \mathrm{Tr}(B\gamma \sum_{j=0}^{m-1} p_j \alpha^{j+k}) = \sum_{j=0}^{m-1} p_j \mathrm{Tr}(B\gamma\alpha^{j+k}) = \sum_{j=0}^{m-1} p_j b_{j+k}$, where b_k $(k = 0, 1, \ldots, m-1)$ are the dual basis coordinates of B can be computed using such an LFSR. The coordinates of the product c_k $(k = 0, 1, \ldots, m-1)$ are

$$\begin{aligned} c_k &= \mathrm{Tr}(A\gamma\alpha^k(\sum_{j=0}^{m-1} b_j\beta_j)) \\ &= \sum_{j=0}^{m-1} b_j \mathrm{Tr}(A\alpha^k\gamma\beta_j) \\ &= \sum_{j=0}^{m-1} b_j [A\alpha^k]_j, \end{aligned} \tag{2.30}$$

where $[A\alpha^k]_j$ is the jth coordinate of $A\alpha^k$ in the polynomial basis.

The hardware complexity of the parallel multiplier presented in [9] depends upon the *Hamming* weight, $H(p)$, of the generating irreducible polynomial. The hardware complexity of this multiplier is at its minimum when $p(x)$ is a trinomial. Furthermore, the delay is minimal when $p(x)$ is a trinomial of the form $p(x) = x^m + x + 1$. Also, when $p(x)$ is a trinomial, only 2 additional XOR gates are required to convert from the dual to the polynomial basis. This adds more flexibility to the multiplier by allowing both operands to be used in the dual basis form. Fenn *et.al.* [10] proposed two systolic architectures to perform field multiplication in

serial and in parallel.

Table 2.4: Dual basis GF(2^m) multiplier architectures

Authors [Ref.]	S/P	(Non-) Systolic	Number of Gates				Latency	Comp. Time	Critical Path
			AND	XOR	FF	MUX			
Berlekamp [3]	S	N	$2m$	$2m-2$	0	0	0	0	$T_A+(\lceil(log_2 m)\rceil)T_{X_2}$
Fenn *et al.* [9]	P	N	m^2	$(m-1)(H(p)-2+m)$	$2m$	0	T_{CP}	T_{CP}	$T_A+(\lceil log_2(H(p)-1)\rceil+\lceil log_2 m\rceil)T_{X_2}$
Fenn *et al.* [10]	S	S	$2m$	$2m$	$10m$	$3m$	$2m$	$3m$	$T_A+T_{X_2}$
Wozniak [53]	S	S	$4m$	$3m$	$8m$	$2m$	$2m$	$3m$	$T_A+T_{X_2}$
Fenn *et al.* [10]	P	S	$2m^2$	$2m^2$	$7m^2$	0	$3m-1$	$4m$	$T_A+T_{X_2}$

A serial systolic architecture was presented in [53] that is based on the multiplication algorithm in (2.30). Using $p(x) = x^m + x + 1$ as the generating polynomial reduces the hardware complexity since there is no need to have an input for the coefficients of $p(x)$. These multipliers have smaller delays and require only one control signal while the one presented by Fenn et al. [10] has a longer delay and requires two control signals. Most of GF(2^m) dual basis multiplier architectures found in the literature are shown in Table 2.4.

2.3.4 Composite Field Multipliers

Using composite fields to implement parallel multipliers has been proposed in [41–43]. Performing the multiplication operation using a composite field has been shown to lower the area complexity of parallel multipliers below the $O(m^2)$ bound. A GF($(2^n)^m$) multiplier can be built using identical modules which provide GF(2^n) arithmetic. Consider the field GF(2^n) with $n > 1$. The elements of an extension field GF($(2^n)^m$) may be represented in the polynomial basis as polynomials with a maximum degree of $m-1$ over GF(2^n). Hence, the element $A \in$ GF($(2^n)^m$) can be represented by the vector $(a_0, a_1, \ldots, a_{m-1})$ where $a_i \in$ GF(2^n) $(0 \leq i \leq m-1)$. The field polynomial of the extension field is an irreducible polynomial $P(x)$ of degree m over GF(2^n).

The product of two elements A and $B \in$ GF($(2^n)^m$) $= AB$ mod $P(x)$ can be performed in two steps:

1. Ordinary polynomial multiplication and

2. Reduction modulo the field polynomial $P(x)$.

The first step is performed using the Karatsuba-Ofman algorithm KOA [26]. The multiplication in the KOA saves polynomial multiplications over GF(2^n) at the cost of polynomial addition which is free. For example, the first iteration in the KOA applied on A and B is to split each of them to a lower and an upper half as follows [40]:

$$A = x^{\frac{m}{2}}(x^{\frac{m}{2}-1}a_{m-1} + \ldots + a_{\frac{m}{2}}) + (x^{\frac{m}{2}-1}a_{\frac{m}{2}-1} + \ldots + a_0) = x^{\frac{m}{2}}A_h + A_l$$
$$B = x^{\frac{m}{2}-1}(x^{\frac{m}{2}}b_{m-1} + \ldots + b_{\frac{m}{2}}) + (x^{\frac{m}{2}-1}b_{\frac{m}{2}-1} + \ldots + b_0) = x^{\frac{m}{2}}B_h + B_l.$$

Three intermediate variables are now defined as:

$$\begin{aligned} d_0 &= A_l B_l, \\ d_1 &= (A_l + A_h)(B_l + B_h), \\ d_2 &= A_h B_h. \end{aligned}$$

The product polynomial $C' = AB$ is given by:

$$C' = d_0 + x^{\frac{m}{2}}(d_1 - d_0 - d_2) + x^m d_2.$$

The reduction modulo $P(x)$ operation can be viewed as a linear mapping of the coefficients resulting from the multiplication operation using the polynomial basis coefficients. The polynomial multiplication of the two polynomials A and B, results in the product polynomial C' over GF(2^n) with $deg(C') \leq 2m-2$. The modulo operation will result in a polynomial C with $deg(C) \leq m-1$ which represents the

final product.

The selection of the composite field order has a direct impact upon the hardware complexity of the multiplier. In [41], the polynomial $p(x) = x^2 + x + p_0$ where $p_0 \in \mathrm{GF}(2^n)$ has been used to develop an $\mathrm{GF}((2^n)^2)$ multiplier. The product $C = AB \bmod P$ is given by:

$$C = (a_0 b_0 + p_0 a_1 b_1) + x((a_0 + a_1)(b_0 + b_1) + a_0 b_0).$$

where a_i and $b_i \in \mathrm{GF}(2^n)$. The KOA was used in [42] to perform the multiplication operation over $\mathrm{GF}((2^n)^4)$. The first two iterations of KOA generate nine intermediate variables d_i, $i = 0, 1, \ldots, 8$ as follows:

$$\begin{aligned}
d_0 &= a_0 b_0 \\
d_1 &= (a_0 + a_1)(b_0 + b_1) \\
d_2 &= a_1 b_1 \\
d_3 &= (a_0 + a_2)(b_0 + b_2) \\
d_4 &= (a_0 + a_1 + a_2 + a_3)(b_0 + b_1 + b_2 + b_3) \\
d_5 &= (a_1 + a_3)(b_1 + b_3) \\
d_6 &= a_2 b_2 \\
d_7 &= (a_2 + a_3)(b_2 + b_3) \\
d_8 &= a_3 b_3.
\end{aligned}$$

The coefficients of the product polynomial C' can now be written as:

$$\begin{aligned}
\acute{c_0} &= d_0 \\
\acute{c_1} &= d_0 + d_1 + d_2 \\
\acute{c_2} &= d_0 + d_2 + d_3 + d_6 \\
\acute{c_3} &= d_0 + d_1 + d_2 + d_3 + d_4 + d_5 + d_6 + d_7 + d_8 \\
\acute{c_4} &= d_2 + d_5 + d_6 + d_8 \\
\acute{c_5} &= d_6 + d_7 + d_8 \\
\acute{c_6} &= d_8. \qquad (2.31)
\end{aligned}$$

The second step is to compute the final product polynomial, $C(x) = c_3x^3 + c_2x^2 + c_1x + c_0$ by performing a modulo reduction operation over the polynomial C'. The modulo reduction operation can be performed using the linear mapping between the coefficients c_i and $\acute{c}_i$ in (2.31) as follows:

$$c_i = \acute{c}_i + \sum_{j=0}^{2} r_{i,j}\acute{c}_{j+4} \quad i = 0, 1, 2, 3 \text{ and } r_{i,j} \in \mathrm{GF}(2^n). \qquad (2.32)$$

The reduction coefficients $r_{i,j}$ which are functions of the field polynomial $P(x) = x^4 + p_3x^3 + p_2x^2 + p_1x + p_0$ and $p_i \in \mathrm{GF}(2)$ $(i = 0, \ldots, 3)$ are given by:

$$r_{i,j} = \begin{cases} p_i & i = 0, \ldots, 3;\ j = 0 \\ r_{i-1,j-1} + r_{3,j-1}r_{i,0} & i = 0, \ldots, 3;\ j = 1, 2 \end{cases}$$

where $r_{i-1,j-1} = 0$ if $i = 0$. The complexity of the composite field multiplier

depends on the choice of the polynomial $P(x)$ over GF(2^n). For example, using the polynomial $P(x) = x^4 + x^3 + x^2 + x + 1$ in the GF($(2^n)^4$) multiplier proposed in [42] leads to the least hardware complexity amongst other choices.

2.4 Conclusions

The choice of GF(2^m) multiplier architecture depends heavily on the underlying basis representation as well as the hardware complexity and the critical path delay of the architecture. Polynomial basis representation has an advantage over the other bases as it can be performed using ordinary polynomial arithmetic. In the normal basis, squaring, which is crucial to other operations such as inversion, can be done for free. The dual basis representation yields the simplest architectures. Selecting serial or parallel architectures is solely dependent on the availability of the operands at the time of computation. Also, systolic architectures allow for pipelining while non-systolic structures are more hardware efficient. Taking all those factors into consideration, selecting the multiplier architecture is easier. For example, if the preference is for low hardware complexity, the non-systolic architectures come to mind. Semi-systolic architectures are also attractive because of their lower hardware complexity compared to fully-systolic architectures. On the other hand, common control signals used in the semi-systolic structures make it difficult to expand the multiplier to higher order fields. Using composite fields to construct multiplier architectures is also attractive. Multiplication over the composite field GF($(2^n)^m$) can be performed using GF(2^n) arithmetic modules which lower the area complexity and increase the modularity of the architecture. How-

ever, the selection of a particular multiplier within each category of multipliers is not quite clear. Combining the hardware complexity measure as well as the critical path delay into one metric is essential in energy-critical applications. The energy metric is the subject of the next chapter.

Chapter 3

Low-Energy GF Multipliers

3.1 Motivation

The tremendous demand for wireless communications devices has enforced the need to reassess designs from the power and energy dissipation perspective. This is due to the fact that wireless devices are mainly battery operated and the efficiency of such device is, then, directly proportional to its battery life. Cryptographic devices have been increasingly used since the rapid trend towards electronic financial transactions through automated banking machines or through the Internet. Cryptographic chips mounted on the surface of a smart card in a wireless phone or a portable device is a very good example for energy-critical designs. Such constrained environment needs a strong power and energy management strategy to cope with the highly demanding cryptographic computations and the limited energy available to the chip.

In this chapter, some of the Galois field (GF) multiplier architectures are compared in terms of the energy metric to assess their suitability for energy-critical

applications. GF multipliers are specifically chosen for comparison here since GF multiplication is a very critical operation in GF arithmetic. Many other operations such as inversion can be performed using multiple multiplication operations. This work has extended the work done in [51] to include the dual basis multipliers in the energy comparisons. Another group of multipliers, based on AOPs, are also to be included in the comparison.

This chapter is an attempt to bridge the gap between the many theoretical publications describing different approaches to VLSI suitable GF multipliers on one side, and the very few reports that compare the architectures from an implementation point of view on the other side.

3.2 Sources of power dissipation in CMOS circuits

In this section, the different components of power dissipation in CMOS circuits are defined. The power a CMOS circuit dissipates falls into two broad categories: *static* and *dynamic* [6]. Fig. 3.1 illustrates a simple CMOS inverter and the different components of power dissipation.

3.2.1 Static Power

Static power is defined as the power dissipated by a gate when it is not switching, i.e. when it is inactive or static. Static power is dissipated in a number of ways. The largest percentage of static power results from source-to-drain subthreshold

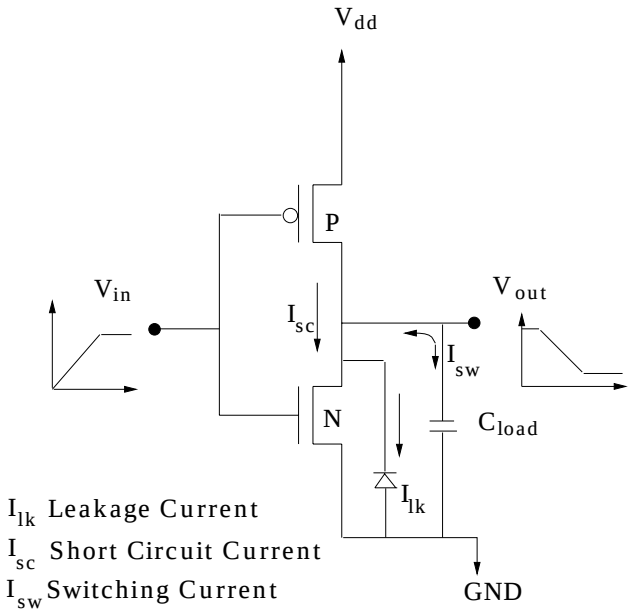


Figure 3.1: CMOS inverter and the different components of power dissipation

leakage. This leakage is caused by reduced threshold voltages that prevent the gate from completely turning off. Static power is also dissipated when current leaks between the diffusion layers and the substrate. For this reason, static power is often called *leakage* power. The total leakage power of a design is the sum of the leakage power of the design's constituent library cells as follows:

$$P_{\text{Leakage Power}} = \sum_{\forall\ \text{cells}(i)} P_{\text{CellLeakage}_i} \tag{3.1}$$

where

$P_{\text{Leakage Power}}$ is the total leakage power dissipation of the design, and

$P_{\text{CellLeakage}_i}$ is the leakage power dissipation for each cell i.

Leakage power is dominant when the circuit is idle but it becomes less than one percent of the total power when the circuit becomes active.

3.2.2 Dynamic Power

Dynamic power is the power dissipated when the circuit is active. A circuit is active anytime the voltage on a net changes due to some stimulus applied to the circuit. Because voltage on a net can change without necessarily resulting in a logic transition, dynamic power can be dissipated even when a net does not change its logic state. The dynamic power is composed of two main components: *Switching* power and *Internal* power.

Switching Power

The switching power of a driving cell is the power dissipated by the charging and discharging of the load capacitance at the output of the cell. The total load capacitance at the output of a deriving cell is the sum of the net and gate capacitances on the driving output. Because such charging and discharging is the result of the logic transitions at the output of the cell, switching power increases as logic transitions increase. Therefore, the switching power of a cell is a function of both the total load capacitance at the cell output and the rate of logic transitions. It is important to point out that switching power comprises 70-90 percent of the dynamic power dissipation in CMOS circuits.

The switching power (P_{sw}) can be calculated using the following formula:

$$P_{sw} = \frac{V_{dd}^2}{2} \sum_{\forall \text{nets}(i)} (C_{load_i} \times TR_i) \tag{3.2}$$

where

C_{load_i} Capacitive load on net i,

TR_i Toggle rate of net i, transitions per second, and

V_{dd} Supply voltage.

Internal power

The internal power is any power dissipated within the boundary of a cell. The definition of internal power includes power dissipated by a momentary short circuit between the P and N transistors of a gate, called *short circuit* power. This happens for a short period of time during a logic transition when both the N and P transistors are ON at the same time. During that time a *short circuit* current, I_{sc}, flows from V_{dd} to GND causing a *short circuit* power, P_{sc}, to be dissipated.

For circuits with fast transition times, short circuit power can be small. However, for circuits with slow transition times, short circuit power can account for 30 percent of the total power dissipated by the gate. Short circuit power is also affected by the dimensions of the transistors and the load capacitance at the gate's output. Internal power is mostly due to short circuit power and therefore the terms internal power and short circuit power are used interchangeably. The internal power

formula is:

$$P_{int} = E_{out} \times TR_{out} \tag{3.3}$$

where

P_{int}	Total internal power
E_{out}	Internal energy for the cell's output as a function of the input transitions and output load $= f[C_{load} \times WAvg_{(trans)}]$,
$WAvg_{(trans)}$	Weighted average transition time for the output $= \frac{\sum_{i=\text{inputs}} TR_i \times Trans_i}{\sum_{i=\text{inputs}} TR_i}$,
$Trans_i$	Transition time of input i.

3.3 Architectures Compared and Methodology

To show the importance of the energy performance measure in selecting energy-critical designs, two sets of architectures were selected based on the conventional measures. The gate-level architectures were built using Cadence^{TM} design tools suite. Then the architectures were simulated using Hspice^{TM} to measure the power consumption and the maximum delay.

3.3.1 Multiplier Architectures selection

The first set of multipliers includes three parallel non-systolic architectures featuring the least hardware complexity within each basis. The polynomial basis architecture

chosen was that of Mastrovito's [30]. Hasan's architecture [18] was chosen as the normal basis multiplier while the multiplier proposed in [9] by Fenn *et.al.* was selected to represent the dual basis multiplier architectures. The second set includes three parallel non-systolic polynomial basis architectures which use an AOP as the field defining polynomial. The architectures selected were those proposed in [16], [23] and [5]. The data of the two sets of multipliers are summarized in Table 3.1. From the data shown, the architectures within each group are close to each other in terms of the hardware complexity to guarantee a fair comparison between the multipliers selected.

Table 3.1: Multiplier architectures selected for comparison

Test Set	Basis	Authors [Ref.]	Number of Gates		Critical Path
			AND	XOR	
Inter-Basis	NB	Hasan *et.al.* [18]	m^2	m^2-1	$T_A+(1+\lceil log_2(m-1)\rceil)T_{X_2}$
	DB	Fenn *et.al.* [9] *	m^2	m^2-1	$T_A+T_{X_2}(4+\lfloor log_2 m \rfloor)$
	PB	Mastrovito [30] *	m^2	m^2-1	$T_A+(\lceil log_2(m)\rceil+1)T_{X_2}$
AOP-based	PB	Itoh *et.al.* [23]	m^2+2m+1	m^2+2m	$T_A+(\lceil log_2(m)\rceil+log_2(m+2)\rceil)T_{X_2}$
	PB	Hasan *et.al.* [16]	m^2	m^2+m-2	$T_A+(m+\lceil log_2(m-1)\rceil)T_{X_2}$
	PB	Koç *et.al.* [5]	m^2	m^2-1	$T_A+(2+\lceil log_2(m-1)\rceil)T_{X_2}$

* The irriducible polynomial used to generate the field is: x^4+x+1.

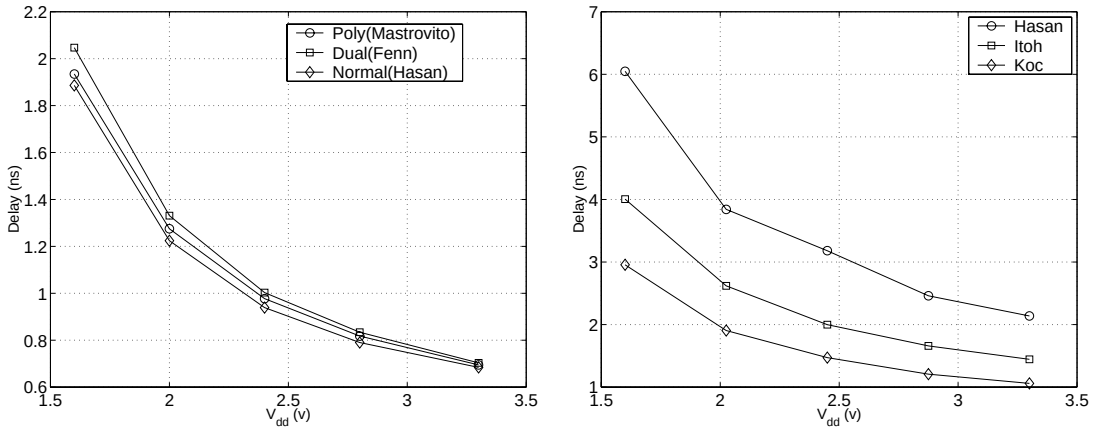
3.3.2 Methodology

The two sets of multipliers were implemented over the field GF(2^4) using a .35μm CMOS technology and simulated at the transistor-level using the Hspice simulator. The clock frequency used was 50 MHz. The order 4 was selected since the AOP is irreducible at that order and the simulation time is considerable. Going to higher orders, the AOP will force the next step to be of order 10 which is prohibitively time consuming to simulate. The polynomial used to construct the field for both the polynomial and the dual basis multipliers for that group was a trinomial of the form $x^4 + x + 1$, for $m = 4$. The normal basis architecture used an AOP. The test vectors were selected to include all the possible combinations as inputs to the multipliers. The comparison results of the first set will be referred to as the "Inter-Bases results" while the "AOP-based results" will refer to the results of the second set.

3.4 Comparison Results

3.4.1 Delay Comparison

This section shows the comparison results between the multipliers in terms of the critical path delay. Fig. 3.2 (a) shows the delay comparisons for the Inter-bases group of multipliers while Fig. 3.2 (b) shows that for the AOP group. It can be seen that the critical path delay measured is directly related to the critical path delay estimation given in Table 3.1.



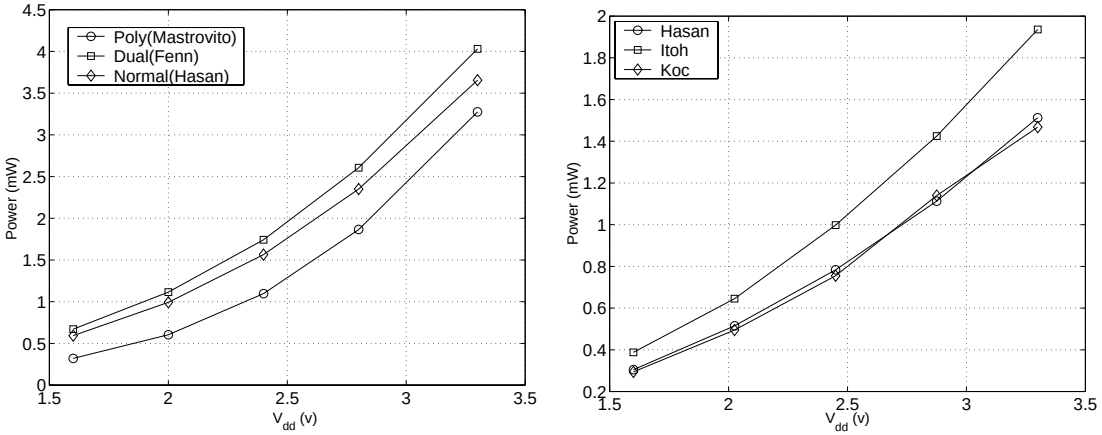
(a) Inter-bases results

(b) AOP-based results

Figure 3.2: Delay comparison

3.4.2 Power Comparison

The simulation results for the power consumption of the two sets of multipliers are shown in Fig. 3.3. Although the hardware complexity seems to be a good estimate for the power consumption, it fails to predict the power measure for the Inter-bases group. The hardware complexity of that set is almost the same for the three multipliers but the power consumption of Mastrovito's multiplier is relatively lower. This is because the switching activity is different for each architecture. Recall from (3.2) that the switching power is directly proportional to the switching activity. Therefore, not only the hardware complexity is a measure of the power consumption but also the interconnects inside the architecture affect the power dissipation.



(a) Inter-bases results

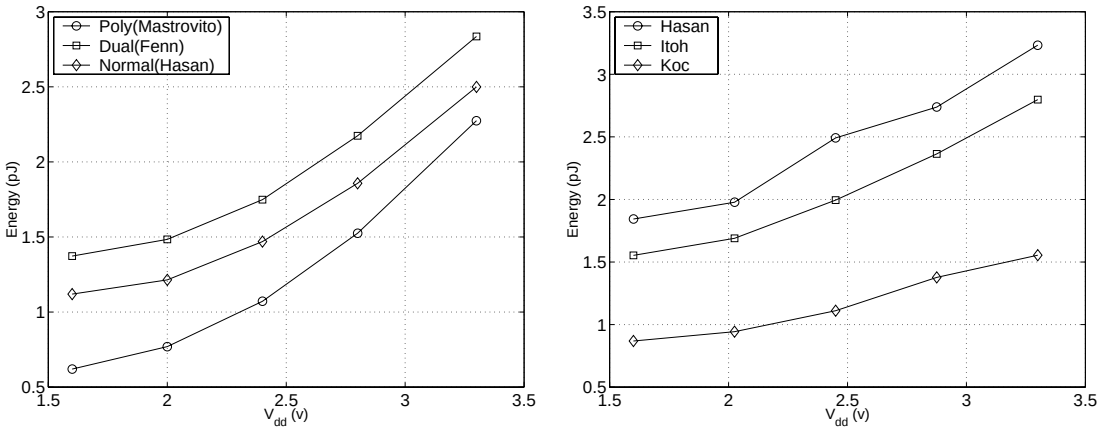
(b) AOP-based results

Figure 3.3: Power consumption comparison

3.4.3 Energy Comparison

Figure 3.4 illustrates the calculated energy results for the two groups of multipliers. In the inter-bases comparison, the results showed that the dual basis multiplier came at the third place for both the delay and power comparisons while the energy comparisons iterated that conclusion. However, if we were interested in the first and the second best architectures, the delay and power results would not have been sufficient. Ranking the multipliers is not quite obvious by just looking at the data given in Table 3.1. The energy metric makes the choice of the most efficient architecture very clear.

In the AOP polynomial basis comparison, the power results could not distinguish the best architecture since Hasan's and Koç's have approximately the same results. Moreover, both architectures are very close to each other in terms of power



(a) Inter-bases results

(b) AOP-based results

Figure 3.4: Energy comparison

consumption. When the delay results were taken into consideration in the energy comparison, selecting the most efficient architecture was fairly simple.

3.5 Conclusion

Energy critical designs would have to consider the energy consumed as the primary comparative measure rather than hardware complexity or critical path delay. Those traditional measures are no longer sufficient for evaluating the suitable architecture for wireless or portable devices. Selecting the most efficient device for energy-critical applications becomes unclear when the compared architectures have nearly the same hardware complexity and critical path delay. The energy comparison should be taken into consideration in order to select the best architecture for a particular application.

Chapter 4

Bit Serial Multiplication over a class of Finite Fields

4.1 Introduction

Many multiplier architectures have been previously introduced to efficiently perform finite field multiplication. Examples of serial multipliers are [8, 11, 14, 15, 56] while parallel ones can be found in [5, 16, 23, 30, 50]. A number of all-one polynomial (AOP) multipliers have been proposed since using an AOP as the field defining polynomial was shown to produce hardware efficient architectures [5, 11, 16, 23]. The first AOP-based multiplier was reported by Itoh and Tsujii [23]. In [16], Hasan *et.al.* extended the work of [23] and presented an efficient multiplier using AOPs. The extended polynomial basis representation, first introduced in [23], was used in [11] to develop a serial AOP multiplier. Koç and Sunar proposed another parallel polynomial basis AOP multiplier that can be also used for normal basis

multiplication through the introduction of the shifted polynomial basis [5]. Other basis representations have been recently proposed to develop hardware and software efficient multipliers such as the redundant basis [56], the polynomial ring [8] and palindromic representation [4].

In this chapter we present a parallel-in serial-out multiplier using all-one polynomials. The proposed multiplier uses a modified version of the polynomial basis, which is referred to as the shifted polynomial basis (SPB). This basis was first introduced in [5] and it can be formed when the field defining polynomial is an AOP. The SPB can be easily converted to the normal basis representation using a simple permutation circuit. This chapter is organized as follows. An introductory background and a review of the shifted polynomial basis representation are to follow in section 4.2. More background information can be found in [45] and [35]. The proposed multiplication algorithm is introduced in section 4.3. Section 4.4 describes the architecture of the proposed multiplier and presents a comparison between the proposed architecture and other GF(2^m) serial multipliers.

4.2 AOP Related Bases of Representations

Consider the AOP, $g(x) = 1 + x + \ldots + x^m$ of degree m. The polynomial $g(x)$ is irreducible if and only if $m + 1$ is prime and 2 is primitive mod $(m + 1)$ [35]. For $m \leq 100$, the AOP is irreducible when $m = 2, 4, 10, 12, 18, 28, 36, 52, 58, 60, 66, 82$ and 100 [23].

Consider the binary field GF(2) and its finite extension GF(2^m) which is of particular interest in many applications. Let $\alpha \in$ GF(2^m) be a root of $g(x)$ and

$\gamma_i = \alpha^i$. Then the basis $\{1, \alpha, \ldots, \alpha^{m-1}\}$ is a polynomial basis (PB). On the other hand, if we assume that $\gamma_i = \alpha^{2^i}$, then $\{\alpha, \alpha^2, \ldots, \alpha^{2^{m-1}}\}$ is a normal basis (NB).

Since α is a root of $g(x)$, $g(\alpha) = 1 + \alpha + \alpha^2 + \ldots + \alpha^m = 0$. Thus, $\alpha^m = 1 + \alpha + \ldots + \alpha^{m-1}$, and,

$$\alpha^{m+1} = 1 \tag{4.1}$$

If an AOP is used to construct the field GF(2^m) and from (4.1), the NB = $\{\alpha, \alpha^2, \ldots, \alpha^{2^{m-1}}\}$ can be rewritten in the form $\{\alpha, \alpha^2, \ldots, \alpha^m\}$ which is referred to as the shifted polynomial basis (SPB) [5]. For example, if $m = 4$, then $\alpha^5 = 1$ and $\alpha^8 = \alpha^3$. Therefore, the NB = $\{\alpha, \alpha^2, \alpha^4, \alpha^8\}$ can be rewritten as $\{\alpha, \alpha^2, \alpha^4, \alpha^3\}$ which is the SPB of GF(2^4) over GF(2).

Suppose that a field element A is given by $A = \sum_{i=0}^{m-1} \acute{a}_i \alpha^{2^i} = \sum_{i=1}^{m} a_i \alpha^i$ where $\acute{a}_i$, $i = 0, 1, \ldots, m-1$, is the ith coordinate of A represented in the NB and a_i, $i = 1, 2, \ldots, m$, is the ith coordinate of A represented in the SPB. The conversion from the NB to the SPB representation can be done using the permutation P [5] which is given by

$$a_{(2^i)} = \acute{a}_i\,, \quad i = 0, 1, \ldots, m-1.$$

where (.) denotes a mod $(m+1)$ operation. The result can be converted back to the NB form using the inverse permutation P^{-1}. Both the permutation and the inverse permutation are implemented by rewiring without using any extra gates. Converting SPB to PB can be done using $m-1$ XOR gates. The permutation

required for the conversion takes the form

$$\begin{aligned} \acute{a}_0 &= a_m, \\ \acute{a}_i &= a_i + a_m\,, \quad i = 1, 2, \ldots, m-1, \end{aligned}$$

where $\acute{a}_i\,,\ i = 0, 1, \ldots, m-1$ is the ith coordinate of A with respect to the PB. The inverse permutation is given by:

$$\begin{aligned} a_i &= \acute{a}_0 + \acute{a}_i, \quad i = 1, 2, \ldots, m-1 \\ a_m &= \acute{a}_0. \end{aligned}$$

Hence, converting the SPB to and from the NB can be performed for free while for the PB requires $m-1$ XOR gates.

4.3 Multiplication and Squaring over the Shifted Polynomial Basis

The algorithms to be described in this section use an approach first introduced in [15] and [16]. A serial AOP multiplier is to be developed here. In [15] a serial multiplier for any irreducible polynomial was developed while a parallel AOP multiplier was introduced in [16]. The basis of representation used here is the SPB while that used in [15] and [16] was the PB.

4.3.1 Multiplication

Let $C = AB \in \mathrm{GF}(2^m)$ and assume that A, B and C are all represented with respect to the SPB. Then,

$$C = AB = \sum_{j=1}^{m} a_j \alpha^j \sum_{i=1}^{m} b_i \alpha^i = \sum_{i=1}^{m} \sum_{j=1}^{m} a_j b_i \alpha^{i+j}. \tag{4.2}$$

Expressing α^{i+j} with respect to the SPB, we have

$$\alpha^{i+j} = \sum_{k=1}^{m} t_k^{[i+j]} \alpha^k \tag{4.3}$$

where $t_k^{[i+j]}$ is the kth coordinate of the element α^{i+j}, with respect to the SPB. Using (4.3), we obtain,

$$C = \sum_{k=1}^{m} c_k \alpha^k = \sum_{k=1}^{m} (\sum_{i=1}^{m} b_i \sum_{j=1}^{m} a_j t_k^{[i+j]} \alpha^k). \tag{4.4}$$

Hence,

$$c_k = \sum_{i=1}^{m} b_i \sum_{j=1}^{m} a_j t_k^{[i+j]} \quad k = 1, 2, \ldots, m. \tag{4.5}$$

Since $\alpha^{m+1} = 1$, for any integer l we have,

$$\alpha^l = \alpha^{(l)}$$

where $(l) = l \bmod(m+1)$. Using $\alpha + \alpha^2 + \ldots + \alpha^m = 1$, the term $t_k^{[i+j]}$ in (4.5) can be written as

$$t_k^{[i+j]} = \begin{cases} 1 & (i+j) = 0\,, \quad \text{and } k = 1, 2, \ldots, m \\ 1 & 1 \leq (i+j) \leq m\,, \quad \text{and } k = (i+j) \\ 0 & 1 \leq (i+j) \leq m\,, \quad \text{and } k \neq (i+j) \end{cases}$$

Then equation (4.5) can be written as

$$c_k = \sum_{i=1}^{m} b_i a_{(-i)} + \sum_{\substack{i=1 \\ i \neq k}}^{m} b_i a_{(k-i)} \qquad 1 \leq k \leq m \tag{4.6}$$

4.3.2 Squaring

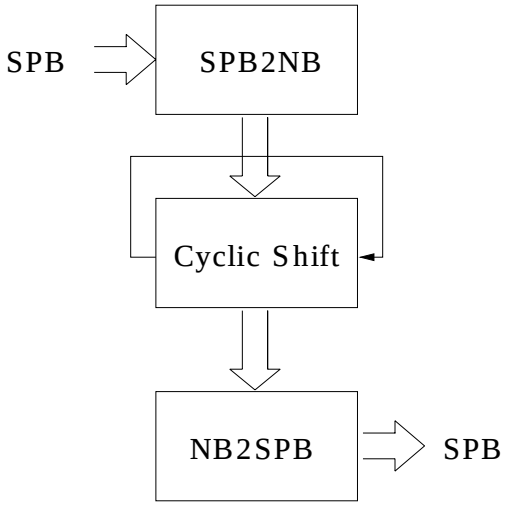


Figure 4.1: Squaring over the shifted polynomial basis

Squaring of a field element represented in the SPB form can be performed by

converting the SPB to the NB form. Converting SPB to/from NB is a simple permutation which can be implemented by rewiring as discussed earlier. Squaring in the NB is just cyclic shift. The squared NB element is then converted back to the SPB form. This operation can be performed in only 2 clock cycles. Fig. 4.1 shows the structure required to perform SPB squaring.

4.4 Multiplier Architecture And Comparison

Using the multiplication algorithm described in the previous section, a parallel-in serial-out multiplier is presented below. From (4.6), it can be noted that each coordinate, c_k, $1 \leq k \leq m$, has the terms corresponding to $a_{(-i)}$ and m elements from the sequence $\{0, a_m, a_{m-1}, \ldots, a_1\}$. The first coordinate, c_1, has the first m elements of the sequence and the second coordinate, c_2, contains the first m elements of the one-fold right cyclic shift of that sequence and so on. Therefore, the above architecture can be implemented using an $m+1$ cyclic register initialized by the elements of the sequence $\{0, a_m, a_{m-1}, \ldots, a_1\}$ and m XOR gates to add the terms corresponding to $a_{(-i)}$. In order to produce the partial products and to accumulate them, m AND gates and $m-1$ XOR gates are required. The architecture produces one bit of the product at a time starting with c_1. Clocking the register m times produces the output sequence $\{c_1, c_2, \ldots, c_m\}$. The structure of the proposed multiplier contains a total of $m + m - 1 = 2m - 1$ XOR gates and m AND gates in addition to $m + 1$ registers.

Example: For $m = 4$, the multiplication of any two elements A and $B \in \mathrm{GF}(2^4)$

represented in the SPB follows directly from (4.6) as:

$$\begin{bmatrix} a_4 & a_3+a_4 & a_2+a_3 & a_1+a_2 \\ a_4+a_1 & a_3 & a_2+a_4 & a_1+a_3 \\ a_4+a_2 & a_3+a_1 & a_2 & a_1+a_4 \\ a_4+a_3 & a_3+a_2 & a_2+a_1 & a_1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}$$

The matrix multiplication above can be performed using the multiplier architecture shown in Fig. 4.2. The register cells A_1 through A_4 are initialized by a_1 through a_4 respectively while A_5 is first set to 0. The matrix multiplication, the *inner-product* unit, is implemented by the AND and XOR gates at the upper part of the architecture. The multiplier needs 4 clock cycles to serially produce the result, $\{c_1, c_2, c_3, c_4\}$.

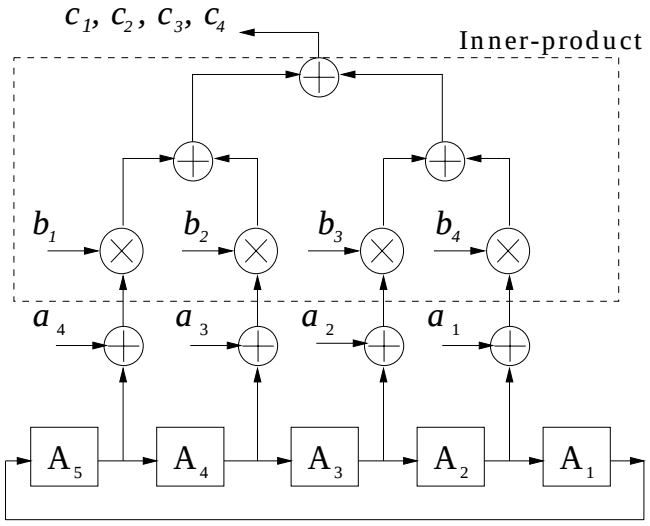


Figure 4.2: The proposed multiplier for multiplication over GF(2^4)

Table 4.1: Comparison between the proposed multiplier and other serial multipliers

Multiplier [Ref.]	Basis of Representation	Input/Output Format	XOR Gates	AND Gates	Registers	Clock cycles required	Critical path delay
Hasan *et.al* [15]	Polynomial	Serial/Serial	$3m-2$	m	$2m$	$2m$	$T_A+T_X(\lceil log_2(m-1)\rceil+1)$
AOPM, Fenn *et.al* [11]	Extended	Parallel/Serial	m	$m+1$	$m+1$	$m+1$	$T_A+T_X(\lceil log_2\, m\rceil)$
MAOPM, Fenn *et.al* [11]	Polynomial	Parallel/Serial	$2m-2$	$2m-1$	$m+1$	m	$T_A+T_X(\lceil log_2(m-1)\rceil+1)$
Drolet [8]	Polynomial Ring	Parallel/Serial	m	$m+1$	$m+1$	$m+1$	$T_A+T_X(\lceil log_2\, m\rceil)$
Wu *et.al.* [56]	Redundant	Parallel/Serial	m	$m+1$	$m+1$	$m+1$	$T_A+T_X(\lceil log_2\, m\rceil)$
Proposed here	Shifted	Parallel/Serial	$2m-1$	m	$m+1$	m	$T_A+T_X(\lceil log_2(m-1)\rceil+1)$

A comparison between the proposed multiplier and other related multipliers proposed in the literature is shown in Table 4.1. The proposed multiplier has the same number of AND gates but less number of registers as well as XOR gates than that proposed in [15] when its field defining polynomial is an AOP. Apparently, when the I/O format is serial-in/parallel-out, the proposed multiplier is the most efficient amongst the other architectures presented in Table 4.1. The all-one polynomial multiplier (AOPM) presented in [11] uses less XOR gates but it requires $m + 1$ clock cycles to complete the operation. When the multiplication operation is to be performed in m clock cycles in the modified all-one polynomial multiplier (MAOPM) architecture [11], the hardware complexity raises significantly. Comparing the proposed architecture to those presented in [8] and in [56], the proposed multiplier has a higher throughput but its hardware complexity is slightly larger. Higher throughput is attractive in many applications and this is going to favor the proposed multiplier over the other ones.

4.5 Conclusion

A parallel-in serial-out finite field multiplier based on using an irreducible AOP as the field defining polynomial is proposed. The multiplier uses the SPB representation. The proposed SPB multiplier can perform NB multiplication after adding conversion modules to the inputs and output. The conversion module can be implemented without any additional gate complexity to the multiplier structure. The proposed multiplier is also capable of performing polynomial basis multiplication by adding $m - 1$ XOR gates to convert to the SPB. Also, the multiplier can per-

form the multiplication operation more efficiently than other parallel-in/serial-out multipliers. The proposed multiplier has a very regular architecture and therefore well suited for VLSI implementation.

Chapter 5

Elliptic Curve Coprocessor

Elliptic Curve Cryptosystems (ECC) have been gaining attention recently as one the promising cryptographic techniques. ECCs offer the same level of security as other public key systems with much smaller key lengths. For example, an ECC with 160-bit key can provide the the same level of security as RSA with 1024-bit key length [25]. This allows for efficient hardware and software implementations over the other alternatives. Standardization of the ECC is currently underway. Some of the efforts in that regard are the *IEEE P1363* draft standard [20]. The *ANSI X9.62* and *X9.63* standards have been already approved by the US government [39]. ECC can be implemented over any group of elements. Of particular interest, the implementation over the group of integers less than a prime number p and over the finite field GF(2^m). The GF(2^m) implementation of the elliptic curve system has been shown to be practical in constrained environments such as smart cards [25].

In this chapter the Elliptic Curve Cryptosystem is described. The curves over the extension field of characteristic 2, GF(2^m) are to be considered. The multiplier

algorithm described in Chapter 4 is used to perform multiplication. The projective coordinates are used to avoid inversion over GF(2^m) [34]. The design of an Elliptic Curve coprocessor is also described. The coprocessor has been simulated using VHDL to verify the functionality.

5.1 Elliptic Curve Cryptosystem

Elliptic curve cryptosystem was independently proposed by Victor Miller [36] and Neil Koblitz [27] back in the mid-eighties. As a public key cryptosystem, it takes years to get a reasonable level of confidence. In the last few years the first commercial implementations have started to appear in many real-world applications, such as email security, web security, smart cards, etc. More detailed information about elliptic curve cryptosystems can be found in [32].

5.1.1 Elliptic curves governing equations over GF(2^m)

Elliptic curves over GF(2^m) can be divided into two sets. The set of all solutions for the equation

$$Y^2 + aY = X^3 + bX + c \tag{5.1}$$

where $a, b, c \in \mathrm{GF}(2^m)$, $a \neq 0$, together with the point O at infinity is a *supersingular* curve over GF(2^m). The second set includes all the solutions for the equation

$$Y^2 + XY = X^3 + aX^2 + b \tag{5.2}$$

where $a, b \in \mathrm{GF}(2^m)$, $b \neq 0$, together with the point O at infinity is a *non-supersingular* curve over $\mathrm{GF}(2^m)$. The pair (X, Y) represents a point on the curve E if both X and Y satisfy (5.1) or (5.2). The coordinate system represented by the pair (X, Y) is called the *affine* coordinate system. For cryptographic purposes, the non-supersingular family of curves is more attractive. That family of curves is more resistant against the *baby-step giant-step* attack, one of the most powerful attacking algorithms known today [2].

5.2 Elliptic Curve Operations over $\mathbf{GF}(2^m)$

The elliptic curve group operations for the non-supersingular family of curves are defined as the addition of two elliptic curve points $P_1 = (X_1, Y_1)$ and $P_2 = (X_2, Y_2)$ resulting in a third point $P_3 = (X_3, Y_3)$. The addition and doubling formulas are

$$\begin{aligned} X_3 &= \lambda^2 + \lambda + X_1 + X_2 + a, \\ Y_3 &= \lambda(X_1 + X_3) + X_3 + Y_1, \end{aligned} \tag{5.3}$$

where

$$\lambda = \begin{cases} \frac{Y_2+Y_1}{X_2+X_1} & \text{if} \quad P_1 \neq P_2, \\ X_1 + \frac{Y_1}{X_1} & \text{if} \quad P_1 = P_2. \end{cases}$$

The most common representations of the $\mathrm{GF}(2^m)$ elements in the elliptic curve computations are the *polynomial* and the *normal* basis representation, especially,

the *optimal normal basis* [2]. In this work, the *shifted polynomial basis* is used to represent the field elements. More information about the shifted polynomial basis can be found in chapter 4.

5.2.1 Group Operation Algorithms using Projective coordinates

The point addition formula in (5.3) requires inversion in the underlying finite field. That inversion operation is very slow and is considered the main bottleneck in the process. To avoid inversion the *projective coordinates* were proposed in [34].

The point $(X, Y) \in E$ in the affine coordinates is mapped to the point $(X : Y : Z) \in E$, $Z = 1$, in the projective coordinates. The inverse mapping between the projective and the affine coordinates is done through dividing by Z which results in $(X/Z : Y/Z : 1)$. The identity point O would be $(0 : 1 : 0)$. Using the projective coordinates, the addition of the two points $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : Z_2)$, if $P \neq Q$, would become

$$\begin{aligned} X_3 &= AD \\ Y_3 &= CD + A^2(BX_1 + AY_1) \\ Z_3 &= A^3 Z_1 Z_2 \end{aligned} \tag{5.4}$$

where $A = X_2 Z_1 + X_1 Z_2$, $B = Y_2 Z_1 + Y_1 Z_2$, $C = A + B$ and $D = A^2(A + aZ_1 Z_2) +$

$Z_1 Z_2 BC$. In case $P = Q$, then

$$\begin{aligned} X_3 &= AB \\ Y_3 &= X_1^4 A + B(X_1^2 + Y_1 Z_1 + A) \\ Z_3 &= A^3 \end{aligned} \tag{5.5}$$

where $A = X_1 Z_1$, $B = bZ_1^4 + X_1^4$.

The elliptic curve group operations are included in the IEEE P1363 draft standard [20]. Detailed information of those algorithms are shown below.

Projective elliptic point doubling algorithm:

The **Double** algorithm performs a point $P_1 = (X_1 : Y_1 : Z_1)$ doubling in terms of the projective coordinates.

Input: A point $P_1 = (X_1 : Y_1 : Z_1)$ on the elliptic curve defined by the parameters a and b.

Output: The point $P_2 = (X_2 : Y_2 : Z_2) = 2P_1$ on the curve.

Projective elliptic point addition algorithm

This algorithm, **Add-pnt**, adds two points $P_0 = (X_0 : Y_0 : Z_0)$ and $P_1 = (X_1 : Y_1 : Z_1)$ in terms of the projective coordinates.

Input:. The two points $P_0 = (X_0 : Y_0 : Z_0)$ and $P_1 = (X_1 : Y_1 : Z_1)$ on the elliptic curve defined by the parameters a and b.

Output: The point $P_2 = (X_2 : Y_2 : Z_2) = P_0 + P_1$ on the curve.

Add-pnt and **Double** are shown in Table 5.1.

Table 5.1: The Point Doubling (Double) and Point Addition (Add-pnt) algorithms

Double	Add-pnt	
1. $T_1 \leftarrow X_1$.	1. $T_1 \leftarrow X_0$.	23. $T_2 \leftarrow T_2 \times T_4$.
2. $T_2 \leftarrow Y_1$.	2. $T_2 \leftarrow Y_0$.	24. $T_7 \leftarrow T_3^2$.
3. $T_3 \leftarrow Z_1$.	3. $T_3 \leftarrow Z_0$.	25. $T_8 \leftarrow T_7 \times T_8$.
4. $T_4 \leftarrow \mathrm{c} = b^{2^{m-2}}$.	4. $T_4 \leftarrow X_1$.	26. $T_1 \leftarrow T_1 \times T_8$.
5. $T_2 \leftarrow T_2 \times T_3$.	5. $T_5 \leftarrow Y_1$.	27. $T_1 \leftarrow T_1 + T_2$.
6. $T_3 \leftarrow T_3^2$.	6. $T_8 \leftarrow a$.	28. $T_4 \leftarrow T_1 \times T_4$.
7. $T_4 \leftarrow T_3 \times T_4$.	7. $T_6 \leftarrow T_3^2$.	29. $T_2 \leftarrow T_4 + T_6$.
8. $T_3 \leftarrow T_1 \times T_3$.	8. $T_7 \leftarrow T_4 \times T_6$.	30. $X_2 \leftarrow T_1$.
9. $T_2 \leftarrow T_2 + T_3$.	9. $T_1 \leftarrow T_1 + T_7$.	31. $Y_2 \leftarrow T_2$.
10. $T_4 \leftarrow T_1 + T_4$.	10. $T_6 \leftarrow T_3 \times T_6$.	32. $Z_2 \leftarrow T_3$.
11. $T_4 \leftarrow T_4^2$.	11. $T_7 \leftarrow T_5 \times T_6$.	
12. $T_4 \leftarrow T_4^2$.	12. $T_2 \leftarrow T_2 + T_7$.	
13. $T_1 \leftarrow T_1^2$.	13. $T_4 \leftarrow T_2 \times T_4$.	
14. $T_2 \leftarrow T_1 + T_2$.	14. $T_3 \leftarrow T_1 \times T_3$.	
15. $T_2 \leftarrow T_2 \times T_4$.	15. $T_5 \leftarrow T_3 \times T_5$.	
16. $T_1 \leftarrow T_1^2$.	16. $T_4 \leftarrow T_4 + T_5$.	
17. $T_1 \leftarrow T_1 \times T_3$.	17. $T_5 \leftarrow T_3^2$.	
18. $T_2 \leftarrow T_1 + T_2$.	18. $T_6 \leftarrow T_4 \times T_5$.	
19. $T_1 \leftarrow T_4$.	19. $T_4 \leftarrow T_2 + T_3$.	
20. $X_2 \leftarrow T_1$.	20. $T_2 \leftarrow T_2 \times T_4$.	
21. $Y_2 \leftarrow T_2$.	21. $T_5 \leftarrow T_1^2$.	
22. $Z_2 \leftarrow T_3$.	22. $T_1 \leftarrow T_1 \times T_5$.	

5.2.2 Scalar Multiplication

The operation in which a point P on the elliptic curve E is to be added to itself k times is denoted by kP and is called *scalar multiplication*. The algorithm **Smultiply** performs the scalar multiplication of an elliptic curve point $P = (X : Y : Z)$ by an integer k. This is the main process in key establishment protocols such as the Diffie-Hellman key exchange.

Input: The point $P = (X : Y : Z)$ on the elliptic curve defined by the param-

eters a and b and a random integer number k.

Output: The point $Q = (X_2 : Y_2 : Z_2) = kP$ on the curve.

Table 5.2: The Scalar Multiplication (Smultiply) algorithm

Smultiply
1. If $k = 0$ or $Z = 0$ the output (1, 1, 0) and stop.
2. $X_2 \leftarrow$ X.
3. $Z_2 \leftarrow$ Z.
4. $Z_1 \leftarrow 1$.
5. $Y_2 \leftarrow$ Y.
6. If $Z_2 = 1$ then set $X_1 \leftarrow X_2$, $Y_1 \leftarrow Y_2$
7. Let $k_l k_{l-1} \ldots k_1 k_0$ be the binary representation of k.
8. For i from $l-1$ downto 1 do
8.1 Set $(X_2, Y_2, Z_2) \leftarrow$ **Double**$[(X_2, Y_2, Z_2)]$.
8.2 If $k_i = 1$ then Set $(X_2, Y_2, Z_2) \leftarrow$ **Add**$[(X_2, Y_2, Z_2), (X_1, Y_1, Z_1)]$.
9. Output (X_2, Y_2, Z_2).

Using the *non-adjacent form (NAF)* has been suggested to improve the performance of the **Smultiply** algorithm [12].

A NAF of an integer k is defined as a signed binary expansion with the property that no two consecutive coefficients are nonzero. Any integer k has a unique NAF(k) representation which has the fewest nonzero coefficients on any signed binary expansion of k. To derive NAF(k), k is repeatedly divided by 2 and the remainder of 0 or $\pm$ 1 is stored. If the remainder is to be $\pm$ 1, the stored remainder is chosen to make the quotient even.

The algorithm **NAF-Smultiply** uses NAF(k) instead of the pure binary representation of k is shown in Table 5.3.

Table 5.3: NAF-Scalar Multiplication (NAF-Smultiply) algorithm

NAF-Smultiply
1. If $k = 0$ or $Z = 0$ the output (1, 1, 0) and stop.
2. $X_2 \leftarrow X$.
3. $Z_2 \leftarrow Z$.
4. $Z_1 \leftarrow 1$.
5. $n \leftarrow k$.
5. $Y_2 \leftarrow Y$.
6. If $Z_2 = 1$ then set $X_1 \leftarrow X_2$, $Y_1 \leftarrow Y_2$
7. Let $k_l k_{l-1} \ldots k_1 k_0$ be the binary representation of k.
8. For i from $l-1$ downto 1 do
8.1 Set $(X_2, Y_2, Z_2) \leftarrow$ **Double**$[(X_2, Y_2, Z_2)]$.
8.2 If $k_i = 1$ then
8.2.1 Set $u \leftarrow 2$ - (k mod 4)
8.2.2 Set $k \leftarrow k$ - u
8.2.3 If $u = 1$ then set $(X_2, Y_2, Z_2) \leftarrow$ **Add-pnt**$[(X_2, Y_2, Z_2), (X_1, Y_1, Z_1)]$
8.2.4 If $u =$ -1 then set $(X_2, Y_2, Z_2) \leftarrow$ **Add-pnt**$[(X_2, Y_2, Z_2), (X_1, X_1 Z_1 + Y_1, Z_1)]$
9. Output (X_2, Y_2, Z_2).

5.2.3 Diffie Hellman Key Exchange

The discrete logarithm problem described by Diffie and Hellman is based on the problem of finding logarithms with respect to a primitive element in the multiplicative group of integers modulo a prime p. This idea can be extended to arbitrary groups with the difficulty of the problem depending upon the choice of the group. The EC-based Diffie-Hellman key exchange protocol is illustrated in Fig. 5.1. More details about the D-H key exchange can be found in [33, 52]. The basic steps are

1. Setup: Alice and Bob agree on a common elliptic curve, E, and a point on that curve, P, with the coordinates $(X_P : Y_P : Z_P) \in E$. Alice generates a random number a, which is her secret key, and Bob generates his secret key

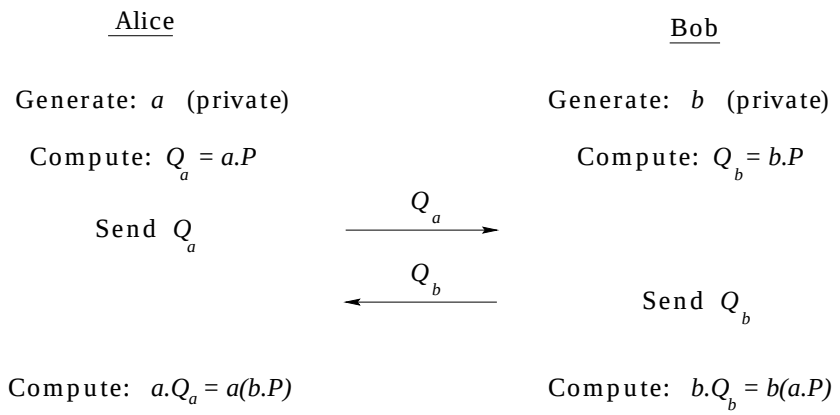


Figure 5.1: Diffie-Hellman Key Exchange Protocol

b.

2. Communication: Alice computes the new point $Q_A = a.P$ and sends it to Bob while Bob computes $Q_B = b.P$ and sends it to Alice. Now, Q_A and Q_B are the public keys. Although P is common in both Q_A and Q_B, the ECDLP insures that it is computationally infeasible to factor out Q_A to compute a or to factor Q_B to compute b.

3. Final Step: Alice computes $a.Q_B = a(b.P) = (X_A : Y_A : Z_A)$ Bob computes $b.Q_A = b(a.P) = (X_B : Y_B : Z_B)$

After the final stage, Alice and Bob can compute the shared session key K as $K = X_A/Z_A = X_B/Z_B$. An adversary cannot recover the session key, K, as he does not know the secret keys a and b. The difficulty of recovering the original message lies on the difficulty of recovering the secret keys from the public key. This recovery problem is called the Elliptic Curve Discrete Logarithm Problem (ECDLP). Once

the session key is established between Alice and Bob, both parties can communicate securely using private key algorithm such as DES for faster encryption speeds.

5.3 Elliptic Curve Coprocessor Architecture

In this section an elliptic curve coprocessor is presented. The coprocessor uses the projective coordinates to represent the curve points over GF(2^m). The multiplier and the squarer architectures selected in this implementation have been previously presented in Chapter 4 which use the SPB representation. Therefore, the field defining polynomial is restricted to be an AOP. Practically speaking, the field order should be 162, 172, 180, 196 or 226 to provide a reasonable security level with respect to today's standards [25].

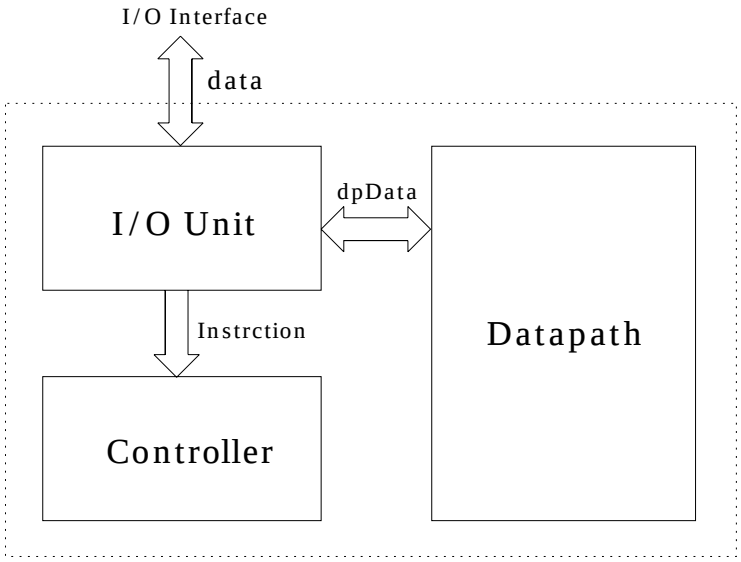


Figure 5.2: The elliptic curve coprocessor architecture

5.3.1 Overview

The elliptic curve coprocessor is designed to perform the cryptographic computations and to relief the main processor in the system from that task. The architecture can perform different finite field arithmetic operations. It is also capable of performing the elliptic curve group operations, add, double and scalar multiplication. The coprocessor architecture performs those operations through a hardwired control logic. The size of the field used is variable and can be reprogrammed if the design is to be implemented over a reconfigurable hardware, e.g. FPGAs.

5.3.2 Coprocessor Architecture

The coprocessor architecture is divided into three major blocks. The datapath unit has all the hardware required to perform the basic GF(2^m) operations such as multiplication, addition, and inversion, as well as EC operations and some registers to hold intermediate results. The control unit controls the operation of the whole coprocessor. The I/O unit buffers instructions and data coming from/written to the main processor. It is mainly used to buffer data to be read/written through the I/O interface which is usually smaller in size. The architecture of the coprocessor is shown in Figure 5.2.

Datapath

The datapath is the main computing core of the coprocessor. The datapath architecture is shown in Figure 5.3. The datapath can be divided into two main building blocks, the arithmetic block and the storage block. The arithmetic block

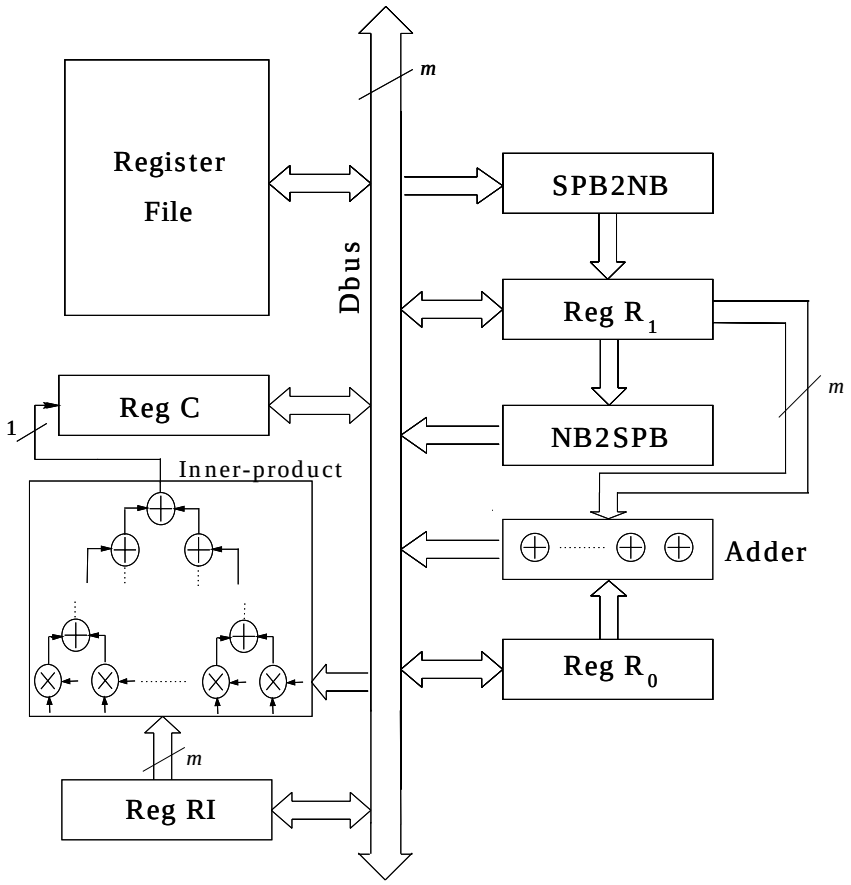


Figure 5.3: Datapath architecture

has four registers, R_0, R_1, RI, C, to perform the multiplication operation. R_1, RI and C are all m-bit registers while R_0 is an $m+1$-bit register. Both R_0 and R_1 are cyclic shift registers where R_0 can shifted to the right while R_1 can be shifted to the left. R_0 is acting as the A register in Fig. 4.2. A field adder is connected to registers R_0 and R_1. The inner-product unit is connected to the RI register and its result is stored in the C register in a bit serial fashion. Two conversion units are used to convert the SPB operands to/from NB. Those conversion modules are

used to implement the squaring operation. Squaring is performed using the relation between the shifted polynomial basis (SPB) and the normal basis (NB) mentioned in Section 4.3. Squaring in the SPB can be done in only two clock cycles, one to convert from the SPB to the NB and the other is to square in the NB and convert to the SPB at the same time. Squaring in the NB is done through cyclically shifting the register R_1 to the left. The storage space holds the temporary variables required for the elliptic curve points addition and doubling. There are eight temporary registers, T_1 through T_8, required for the point addition and point doubling algorithms. The registers T_7 and T_8 are special registers that can be used in the conversion of SPB to/from PB. The register T_7 is used to convert from SPB to PB while T_8 is used in the conversion from PB to SPB. Two registers are required to hold the original point coordinates, X and Y, and another two registers are used to hold the curve parameters a and b. The integer value, k used in the scalar multiplication operation is stored in the K register.

Controller

The control unit is a finite state machine, FSM, that includes all the control sequences for the different instructions. The FSM has the enumerated state {*Idle, Fetch, Decode, Exec1, Exec2, Exec3*}. The FSM remains in the *Idle* state while the *Receive* signal is inactive. The state of the FSM goes to *Fetch* when the *Receive* signal becomes active. The *Fetch* state remains until the instruction is available for the controller to be read. Once the instruction is read, the FSM goes to the *Decode* state. If the fetched instruction is a *Load*, the coprocessor goes to the *Exec* state

Table 5.4: Binary encoding of Datapath Registers

Register	Binary code
R0	0000
R1	0001
RI	0010
K	0011
T1	0100
T2	0101
T3	0110
T4	0111
T5	1000
T6	1001
T7	1010
T8	1011
X1	1100
Y1	1101
A	1110
B	1111

and remains there till the data is ready to be processed. However, if the instruction is not a *Load*, the FSM performs the required actions at the *Decode* state and goes to the *Exec* stage. Depending upon the type of instruction being executed, the FSM goes to *Exec2* or *Exec3* state and may remain in the *Exec3* state for some time till the execution ends. For example, when executing a *Multiply* instruction, the FSM remains in the *Exec3* state for m clock cycles before writing the result and going back to *Idle*. For the *Unload* instruction, the FSM remains in the *Exec* state until the DataOut buffer of the I/O unit is emptied then it goes back to *Idle*.

The elliptic curve point operations are hardwired inside the controller so that the *AddPoint* and *DoublePoint* instructions execute a sequence of other instruc-

tions according to Table 5.1. The scalar multiplication operation is also hardwired. Although hardwiring instructions increases the hardware required but it simplifies the programming of the coprocessor and reduces the program size. Two counters are used inside the control unit: the *k_counter* is the k-bit number being processed and *counter* is the current operation number in the sequence of operations inside a subroutine. This number is indicated before each operation in Table 5.1.

I/O Unit

The elliptic curve coprocessor is designed to handle operands in GF(2^m) of m-bit size. Other cryptosystems such as RSA require field sizes of more than a 1000-bit. This large size compared to ordinary arithmetic unit creates a challenge in the design of the I/O unit.

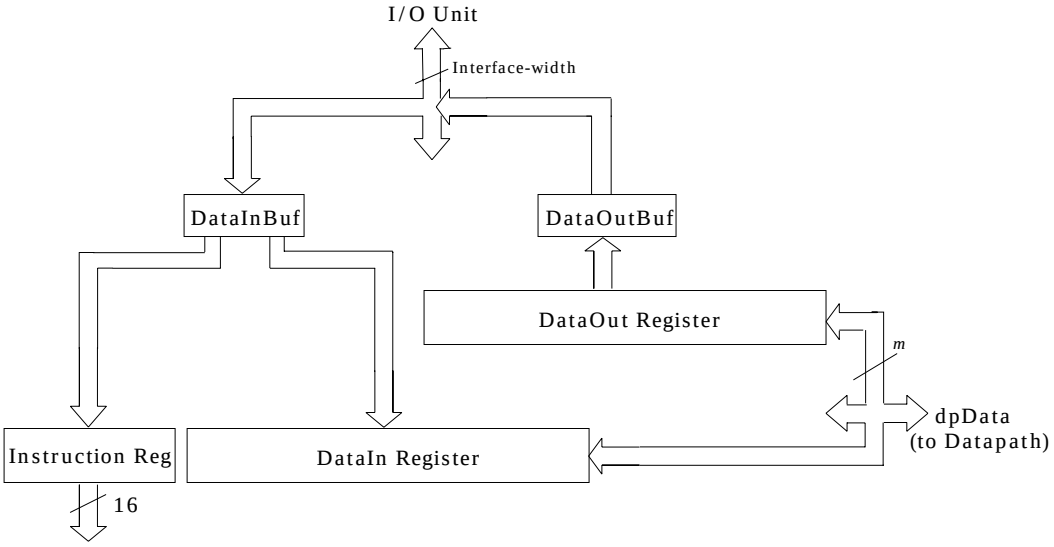
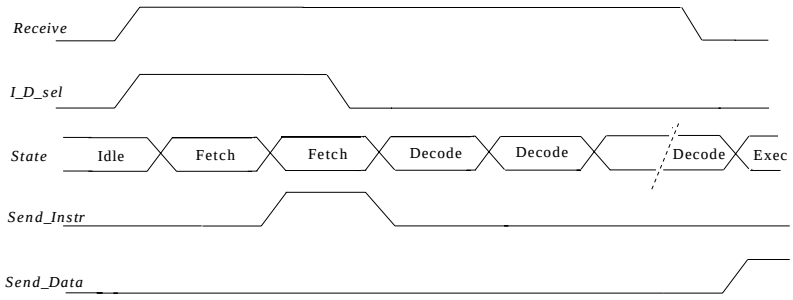
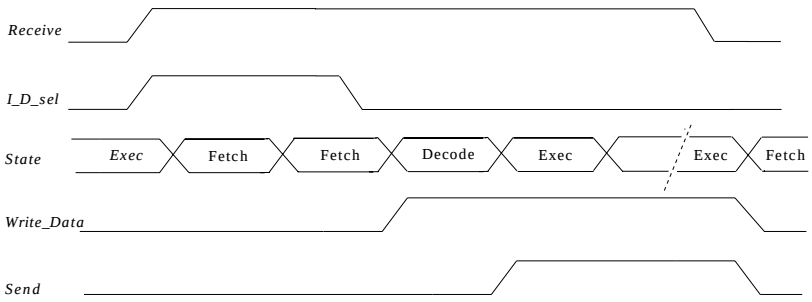


Figure 5.4: I/O unit structure

Bit-parallel I/O transfers for that large data size are prohibitively complex even with modern VLSI technologies. In contrast, bit-serial I/O operations are much more simple but exhibit a very long delay compared to their parallel counterparts. Splitting the large operand into smaller equally-sized chunks of bits seems to be the most efficient approach to accomplish I/O operations. The size of the bit-chunk can be set according to the interface width of the other device connected to the coprocessor.



(a) Read Data operation



(b) Write Data operation

Figure 5.5: Read/Write Operation

The I/O unit has two buffering registers to hold the data being transmitted.

The *DataIn* register holds the incoming data and the *DataOut* register buffers the outgoing data. The *InsructionReg* buffers the Instructions. The I/O unit structure is shown in Figure 5.4.

The I/O operation uses a handshaking protocol to start transmitting instructions/data. Two input signals, *Receive* and *I_D_sel*, and the *Ready* output signal are used to perform the input handshaking protocol. The *Receive* signal informs the coprocessor of incoming data/instruction while *I_D_sel* identifies its nature. The *Ready* signal becomes inactive whenever the instruction or data buffer is partially full. The *Send* signal is used to inform other devices connected to the coprocessor that data needs to be written out. *Send* becomes inactive when the outgoing data buffer is empty. The read and write handshaking operations are shown in Figs. 5.5 (a) and (b) respectively.

5.3.3 Instruction Set Architecture

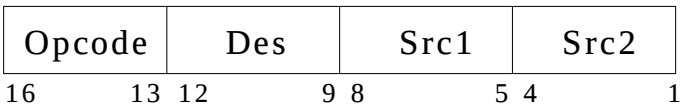


Figure 5.6: Instruction set architecture

The 16-bit instruction is divided into 4 parts, 4-bit each as shown in Fig. 5.6. The instruction set can be divided into three main groups. One group has the register load, copy, clear and unload operations. This group only provides the opcode and the destination, Des, register. The two source operands, Src1 and Src2, are not provided and are not used. The second group includes all the finite field

Table 5.5: Instruction Set

Opcode	Mnemonic	Operands	Description
0000	NOP	None	No Operation
0001	LoadReg	Des	Des $\leftarrow$ (dpData port in I/O unit)
0010	UnloadReg	Des	(Data port of I/O unit) $\leftarrow$ Des
0011	CopyReg	Des, Src1	Des $\leftarrow$ Src1
0100	ClrReg	Des	Des $\leftarrow$ 0
0101	Add	Des, Src1, Src2	Des $\leftarrow$ Src1 + Src2
0110	Multiply	Des, Src1, Src2	Des $\leftarrow$ Src1 * Src2
0111	Square	Des, Src1	Des $\leftarrow (Src1)^2$
1000	NB2SPB	Des, Src1	Des $\leftarrow$ NB-to-SPB(Src1)
1001	SPB2NB	Des, Src1	Des $\leftarrow$ SPB-to-NB(Src1)
1010	PB2SPB	Des, Src1 (Src1 = T_8)	Des $\leftarrow$ PB-to-SPB(Src1)
1011	SPB2PB	Des, Src1 (Src1 = T_7)	Des $\leftarrow$ SPB-to-PB(Src1)
1100	Invert	Des, Src1	Des $\leftarrow$ inv(Src1)
1101	DoublePoint	None	EC point $(T1, T2, T3)$ doubling
1110	AddPoint	None	EC point $(T1, T2, T3)$ addition
1111	Smultiply	None	EC point scalar multiplication

arithmetic instructions. Each instruction in this group has to provide two source operands and the destination operand in addition to the opcode. The arithmetic operations supported by the coprocessor architecture are: multiply, add, square, invert, convert SPB to/from NB and convert SPB to/from PB. The elliptic curve point operations form the third group. This group only provides the opcode and does not provide any information about the operands. The hardwired elliptic curve operations are: Add point, Double point and Scalar multiply. Although those hardware macros add a significant hardware complexity to the chip, they facilitate the programming task to a great extent. The coprocessor instructions are show in Table 5.5.

5.4 Comparison

Table 5.6 shows the number of field operations performed in each EC operation. Using the projective coordinates results in the large number of multiplications indicated to avoid field inversions.

Table 5.6: Operation count for Point Doubling and Addition

Operation	EC Doubling	EC Addition
Field Addition	4	6
Field Multiplication	5	13
Field Squaring	5	4

The performance analysis of the proposed design is shown in Table 5.7. The number of clock cycles required to perform the field operations is indicated. The clock cycle count of the EC point addition and doubling is also indicated. For $m = 196$ and a clock frequency of 80 MHz, the time required for each operation is given. Inversion is the slowest operation since the design is not optimized for inversion. The scalar multiplication operation was implemented over the *Galois Field Processor* (GFP) in [13] using a software program and using the affine coordinate system. The point doubling is two times faster than that implemented over the GFP, while EC addition is a bit slower. Point doubling is a crucial operation in the scalar multiplication operation since it is performed $m-1$ times in the scalar multiplication operation while point addition is performed $(m-1)/2$ times on the average, for the binary representation of the scalar.

Table 5.7: Performance of the proposed architecture

Operation	Clk Cycles required	Time in μsec	
		EC coprocessor [1]	GFP [2][13]
Field Addition	3	0.03	0.03
Field Multiplication	$m+2$	2.43	2.41
Field Squaring	3	0.03	2.41
Field Inversion	$3(m-1)+n(m+2)$. [3]	31.4	4.81
EC Doubling	$5m+37$	12.5	24.61
EC Addition	$13m+56$	31.9	27.05

[1] $m = 196$, assuming a clk frequency of 80 MHz.
[2] $m = 191$, assuming a clk frequency of 80 MHz.
[3] n = No. of multiplications required for one inversion (e.g. for $m = 196$, $n = 10$ [22]).

5.5 Conclusion

A GF(2^m) Elliptic Curve (EC) coprocessor is to speedup the Diffie-Hellman key exchange protocol. The coprocessor uses the parallel-in serial-out GF(2^m) finite field multiplier proposed in Chapter 4. The architecture of the coprocessor as well as the instruction set have been described. The design has been simulated using VHDL to verify the functionality. The proposed design uses the projective coordinate system. The coprocessor is to perform elliptic curve point addition, point doubling, and scalar multiplication. Those elliptic curve functions are hardwired inside the controller of the coprocessor to facilitate the programming task. The performance of the design in [13] when the EC point doubling on the affine coordinate system implemented using a software program has been found to be slower than the implementation of the same operation over the proposed architecture. Speeding up the EC point doubling operation has a positive impact on the performance of the elliptic curve scalar multiplication which is the main operation in the Diffie-Hellman

key exchange algorithm.

Chapter 6

Conclusion and Future Work

6.1 Summary and Conclusion

The choice of GF(2^m) multiplier architecture depends heavily on the underlying basis representation as well as the hardware complexity and the critical path delay of the architecture. Selecting serial or parallel architectures depends on the availability of the operands at the time of computation. Also systolic architectures allow for pipelining while non-systolic structures are more hardware efficient. Those are the conventional measures that can be used in the selection process. These traditional measures are no longer sufficient for choosing a certain architecture for wireless or portable devices. Selecting the most suitable device for energy-critical applications becomes unclear when the architectures under consideration are having nearly the same hardware complexity and critical path delay. The energy metric would have to be taken into consideration in order to select the most efficient architecture for a particular application.

A parallel-in serial-out finite field multiplier based on an irreducible AOP as the field defining polynomial has been proposed. The proposed multiplier can perform polynomial basis as well as normal basis multiplication after adding conversion modules to the inputs and output. Also, the multiplier can perform the multiplication operation more efficiently than other parallel-in/serial-out multipliers. The proposed multiplier has a very regular architecture and therefore well suited for VLSI implementation.

An elliptic curve coprocessor that uses the proposed multiplier is designed using the projective coordinates. The projective coordinates are advantageous in avoiding inversion in the underlying finite field. The coprocessor architecture as well as the instruction set have been developed. VHDL has been used in verifying the design. The coprocessor is able to perform elliptic curve point addition, point doubling, and scalar multiplication. Those elliptic curve functions are hardwired inside the controller of the coprocessor to facilitate the programming task. The use of the projective coordinates system has enabled us to reduce the computation time for point doubling. Speeding up the EC point doubling operation has a positive impact on the elliptic curve scalar multiplication which is the main operation in the Diffie-Hellman key exchange algorithm.

6.2 Recommendations for Future Work

This thesis has considered the hardware structure of an elliptic curve cryptosystem over the finite field GF(2^m). It has also examined the different multiplier architectures which can be used for that system. The implementation of such system

requires a through analysis of its building blocks. The work that could be still done in this regard is summarized below.

1. The energy measure could be extended to compare more finite field computing devices such as inverters and squarers.

2. The size of the multiplier architectures simulated in this work have been restricted by the available CAD tools. Simulating higher orders, even degree 10, would have taken a very long time. Using other tools such as PowerMill^{TM}, could have shorten the simulation time considerably.

3. Efficient AOP inversion would be very helpful is performing EC computations in the affine coordinates rather than the projective coordinates. Using the affine coordinates would save a considerable amount of storage on the chip.

Chapter 7

Appendix

7.1 Example

The following example was simulated on the coprocessor and verified using a MATLAB program. The field order is: $m = 4$.

Input

Point coordinates: $X = 0010$, $Y = 0001$.

Curve parameters are: $a = 0010$, $b = 0010$. Scalar factor: $k = 11$.

Output

$X_2 = 0101$.

$Y_2 = 1110$.

$Z_2 = 0100$.

The following simulation results validates the architecture.

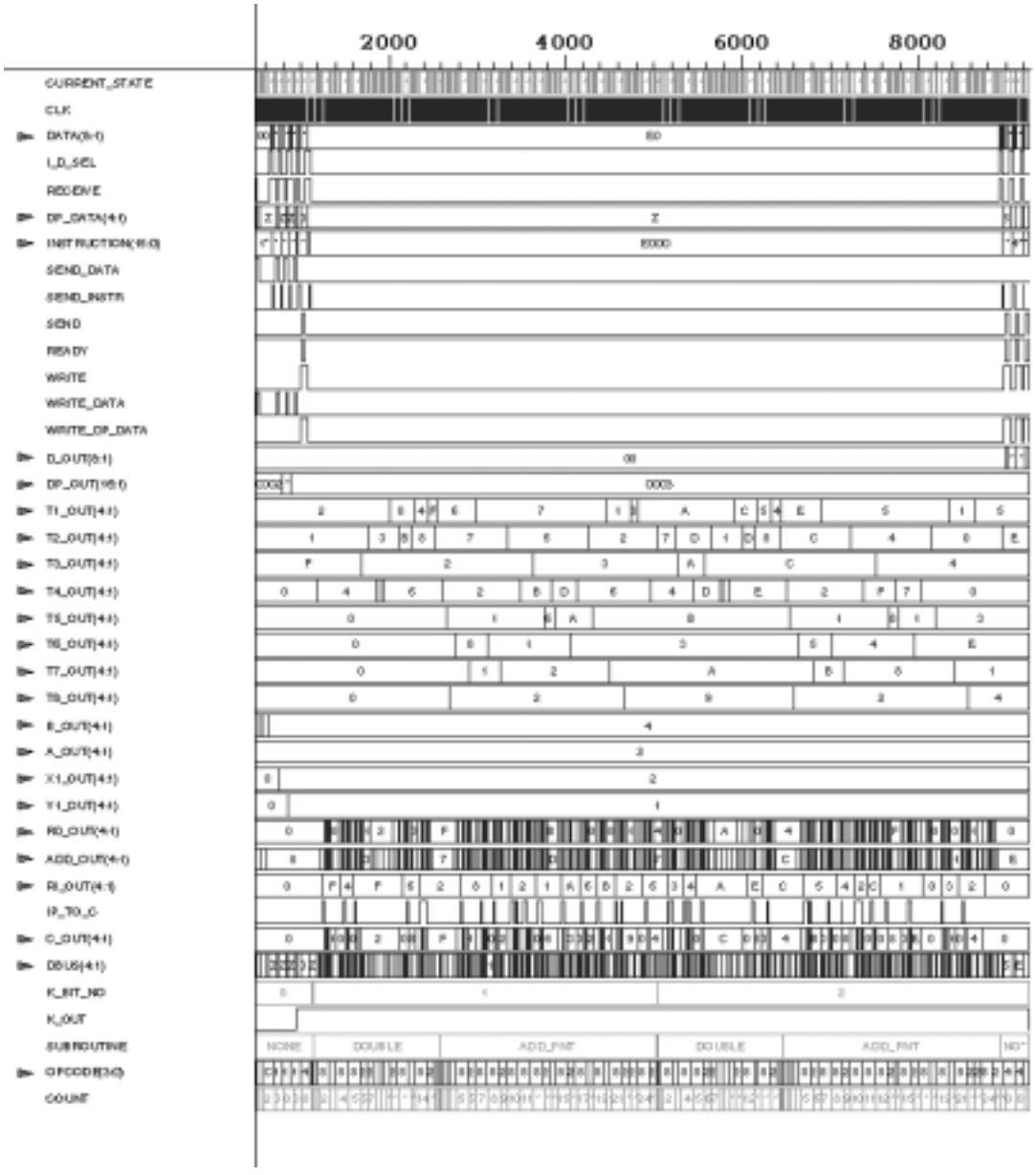


Figure 7.1: Simulation Waveforms

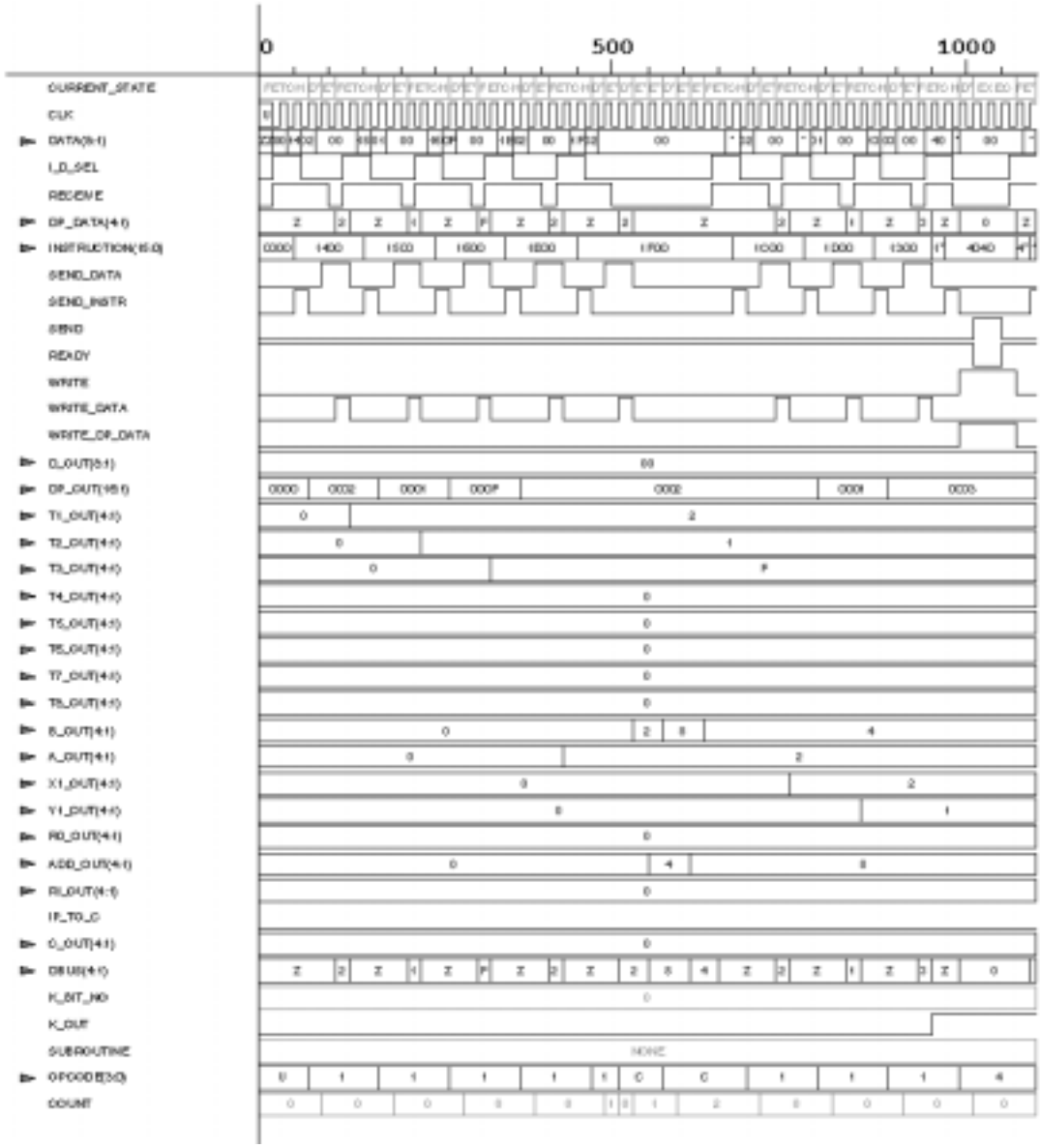


Figure 7.2: Simulation Waveforms (cont.)

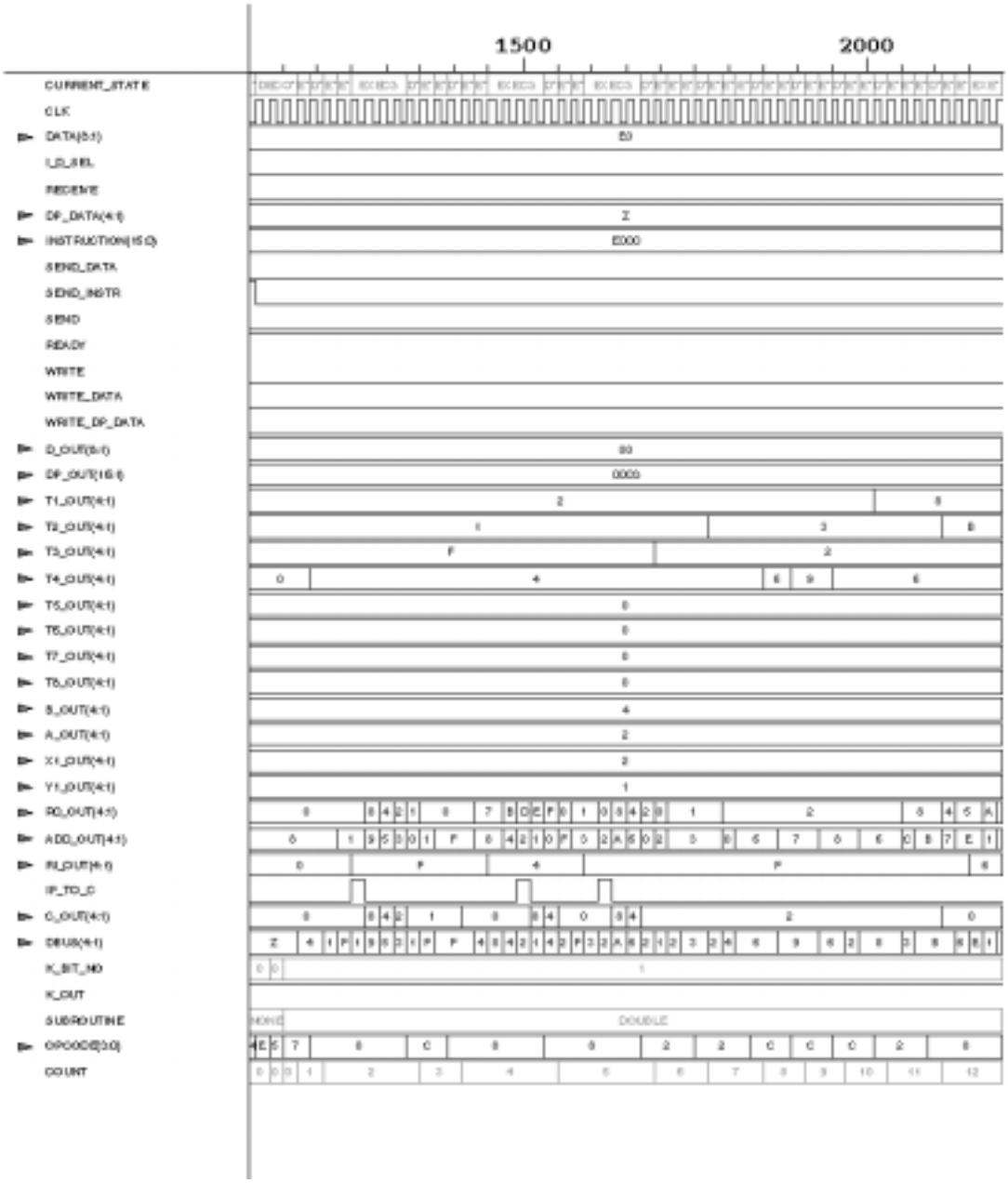


Figure 7.3: Simulation Waveforms (cont.)

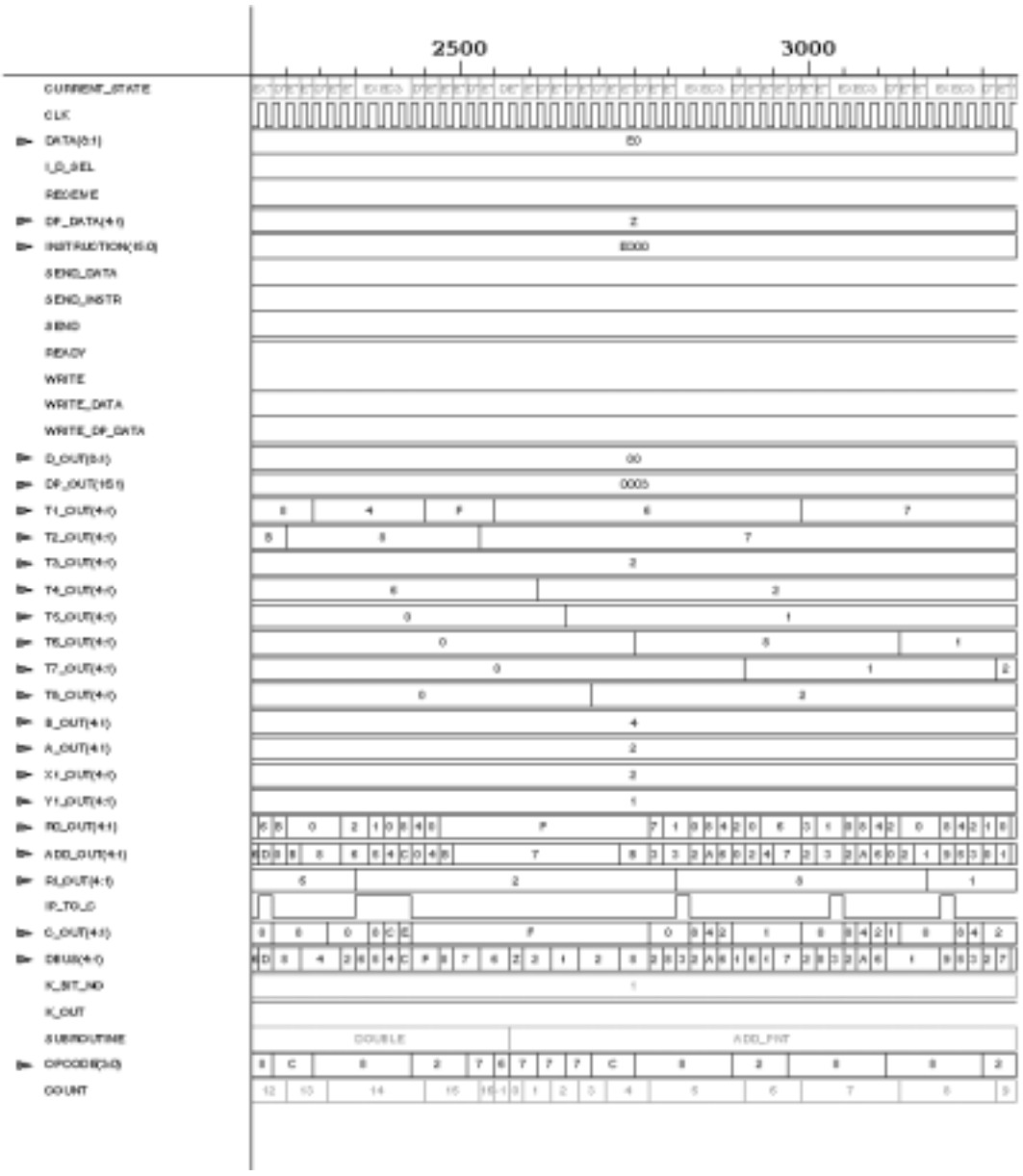


Figure 7.4: Simulation Waveforms (cont.)

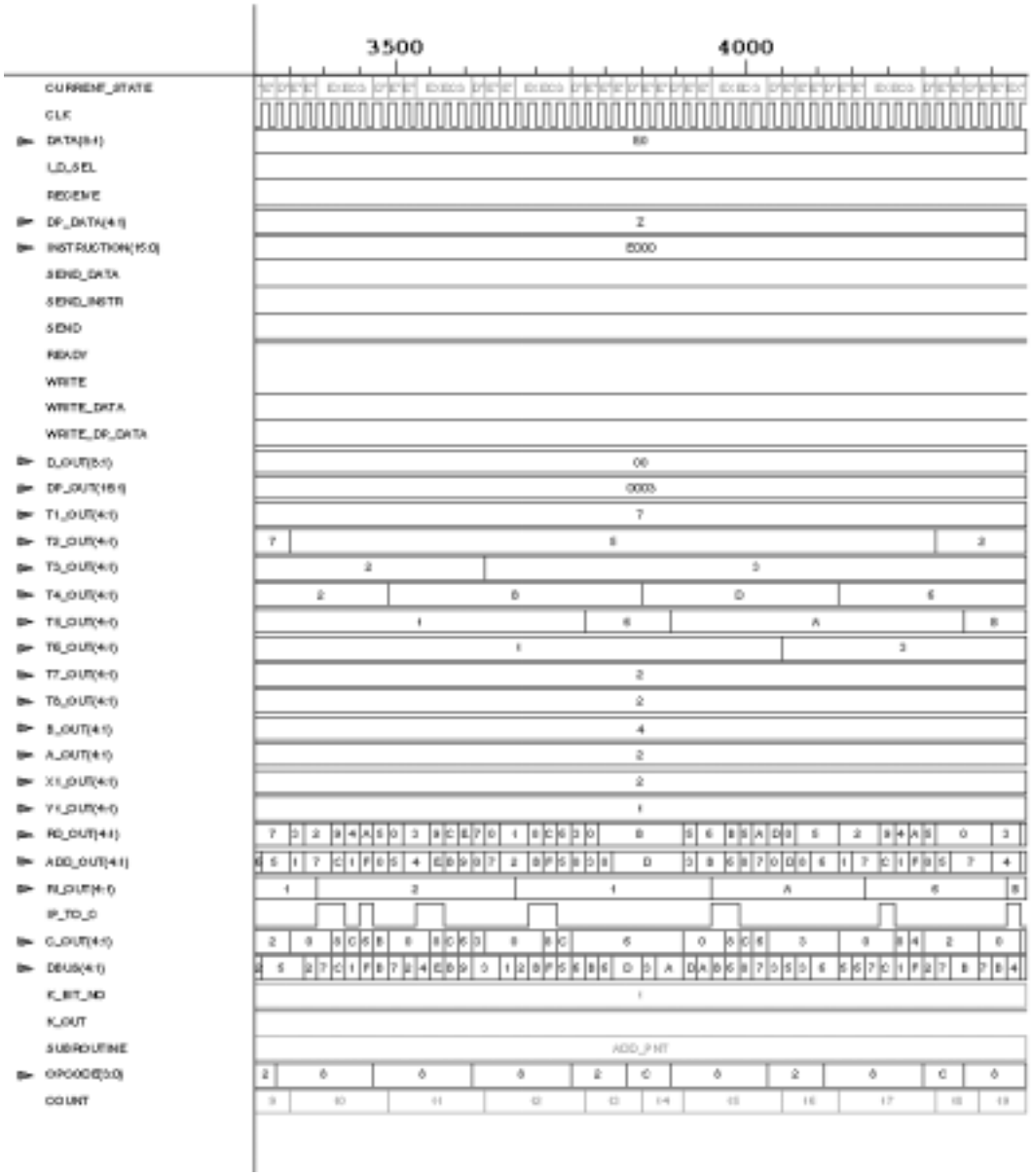


Figure 7.5: Simulation Waveforms (cont.)

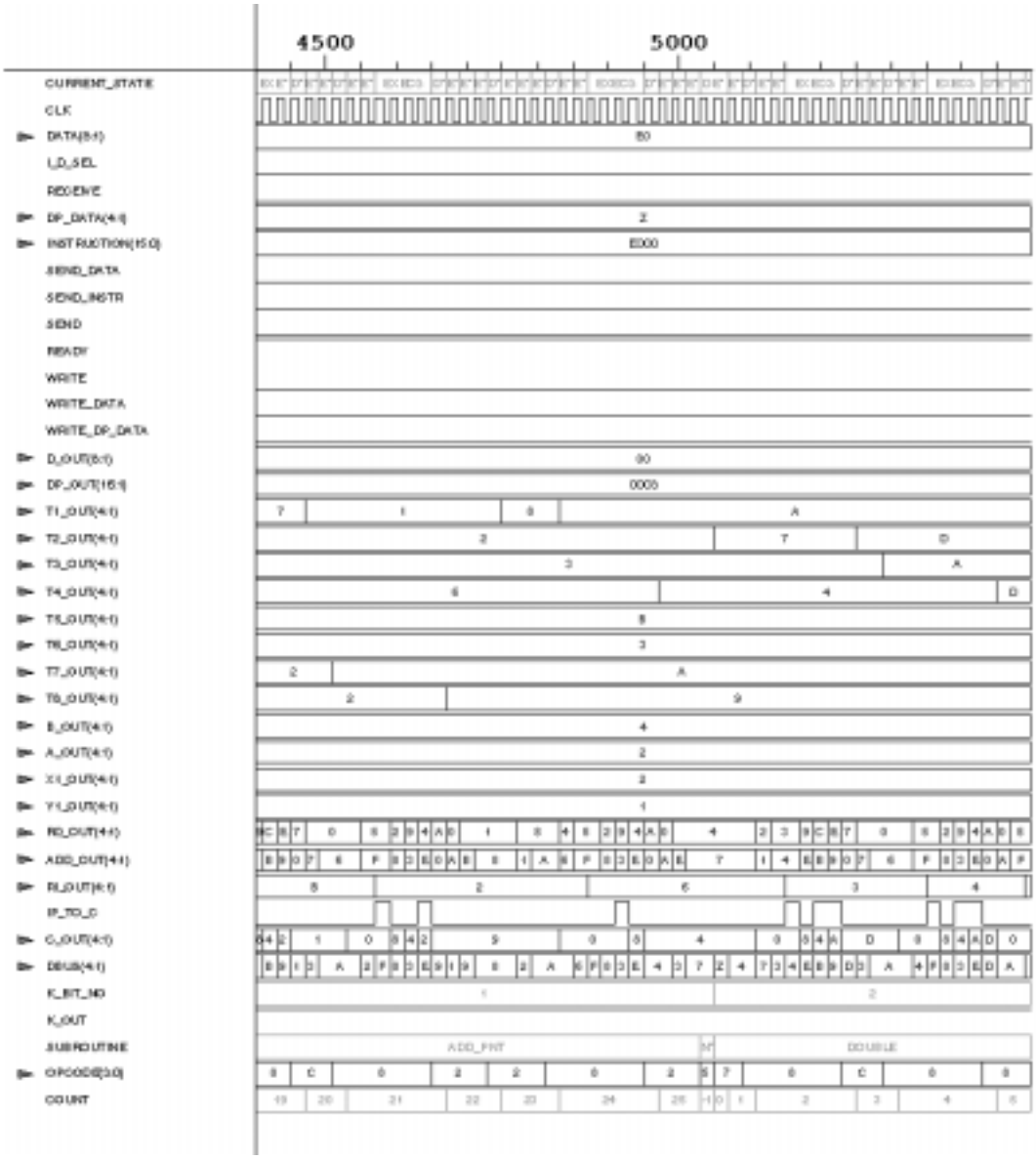


Figure 7.6: Simulation Waveforms (cont.)

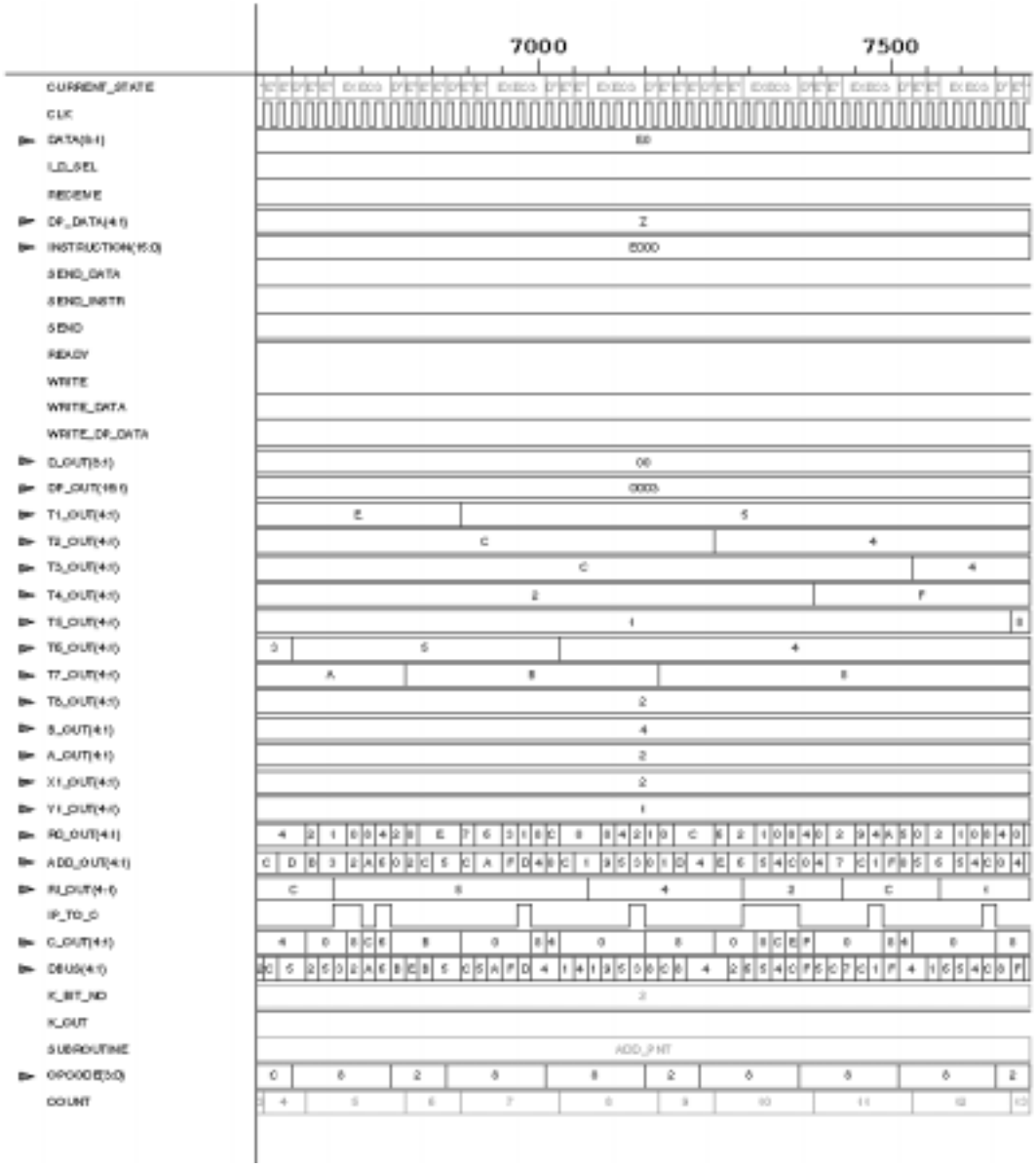


Figure 7.7: Simulation Waveforms (cont.)

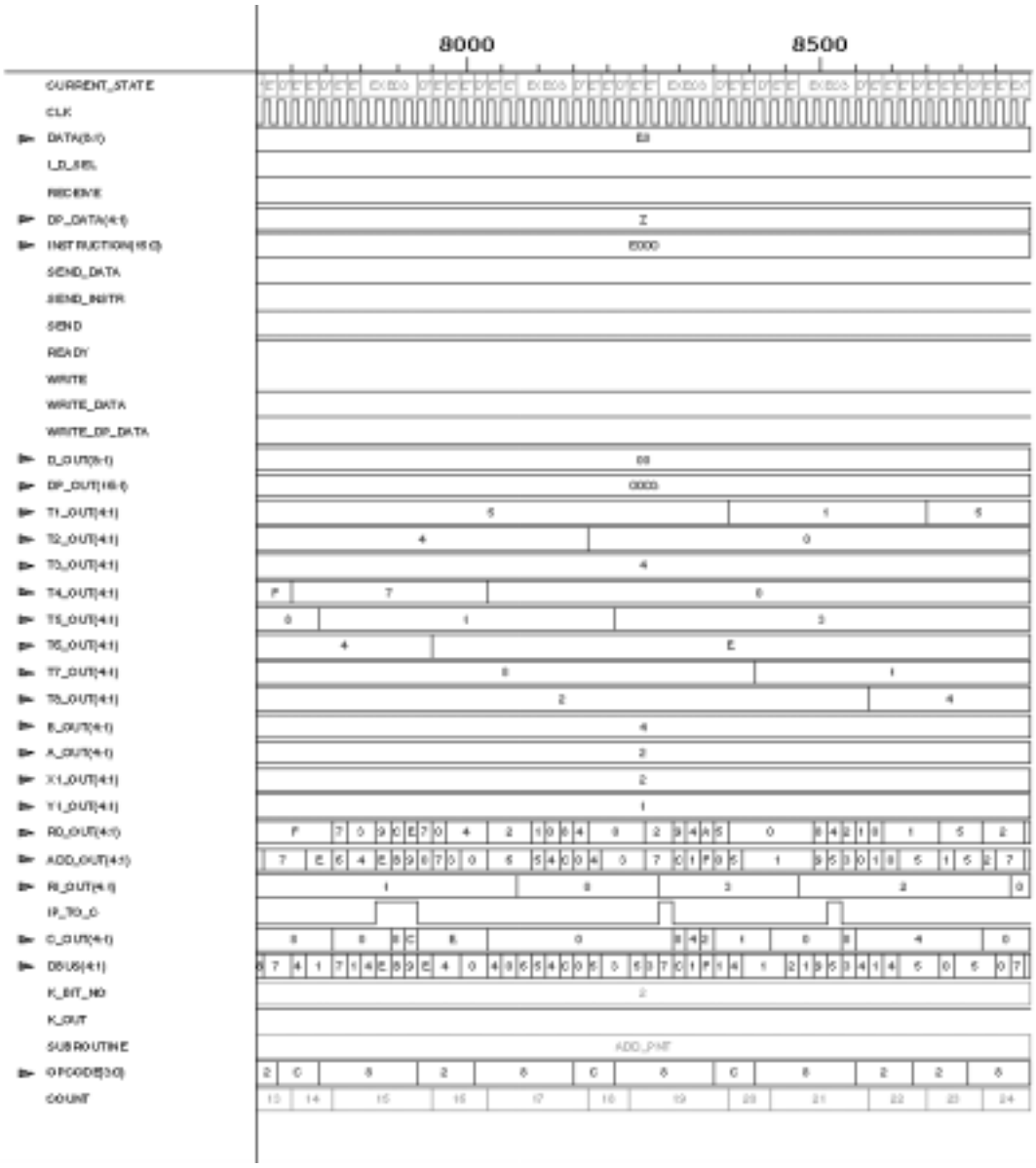


Figure 7.8: Simulation Waveforms (cont.)

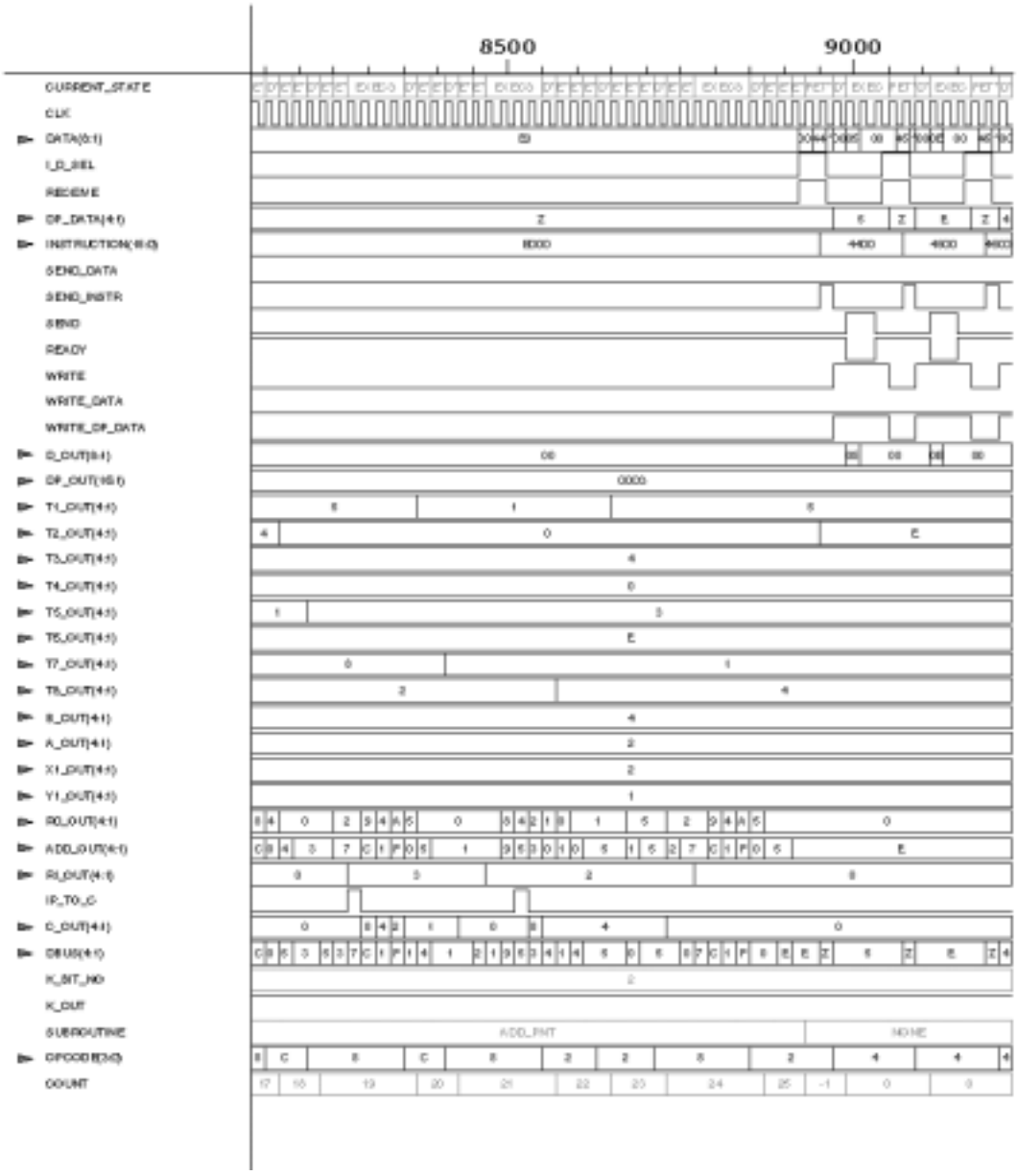


Figure 7.9: Simulation Waveforms (cont.)

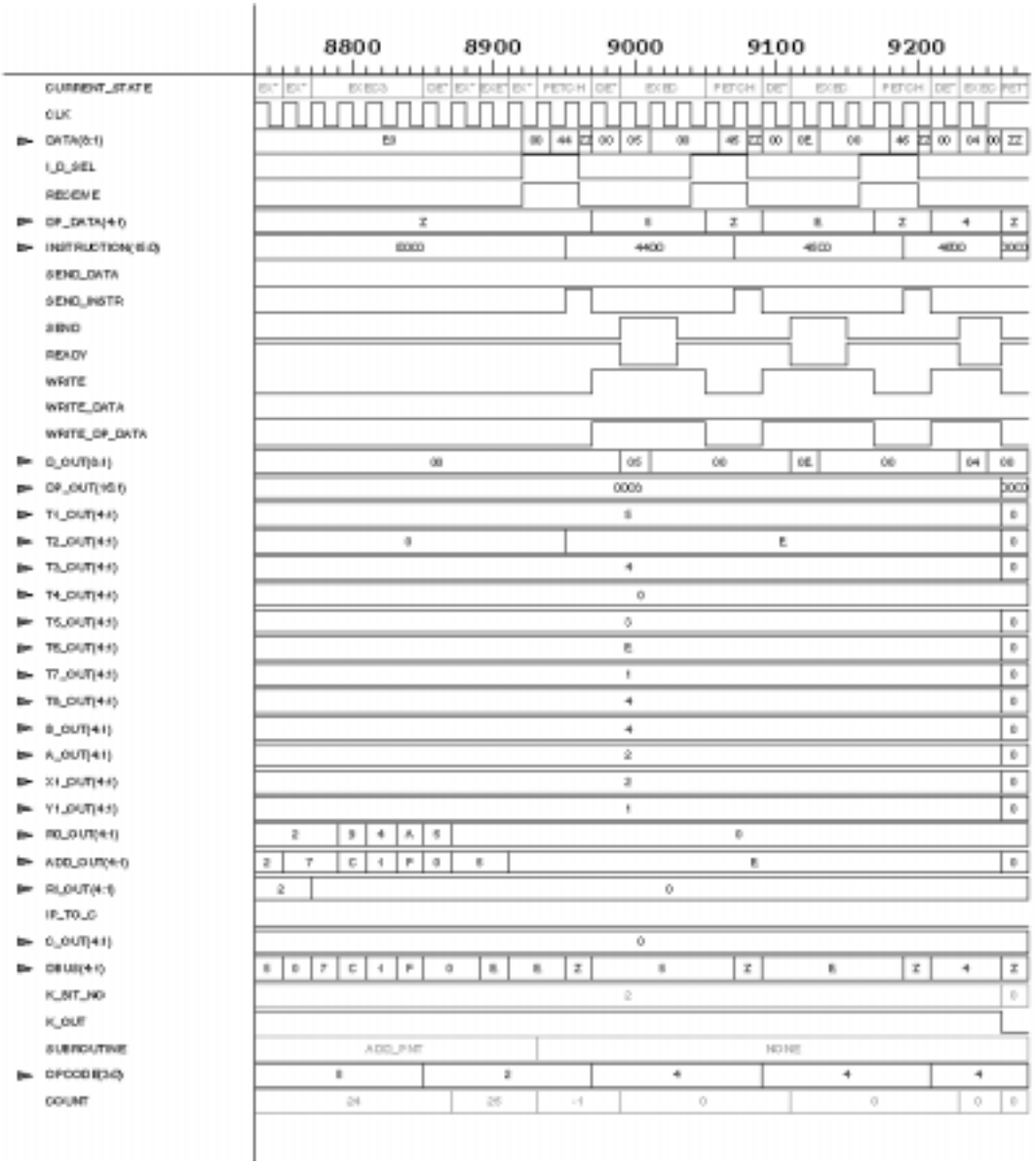


Figure 7.10: Simulation Waveforms (cont.)

Bibliography

[1] G. B. Agnew, R. C. Mullin, I.M. Onyszchuk, and S. A. Vanstone. An Implementation for a Fast Public-Key Cryptosystem. *Journal of Cryptology*, 3(2):63–79, 1991.

[2] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An Implementation of Elliptic Curve Cryptosystem Over F_2^{155}. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

[3] E.R. Berlekamp. Bit-Serial Reed-Solomon Encoders. *IEEE Transactions on Information Theory*, 28(6):869–874, Nov. 1982.

[4] I. Blake, R. Roth, and G. Seroussi. Efficient Arithmetic in GF(2^m) through palindromic representation. Visual computing dept., Hewlett Packard Laboratories, 1998. Available at: `http://www.hpl.hp.com/techreports/98/HPL-98-134.html`.

[5] Ç.K. Koç and B. Sunar. Low-Complexity Bit-Parallel Canonical and Normal Basis Multipliers for a Class of Finite Fields. *IEEE Transactions on Computers*, 47(3):353–356, March 1998.

[6] A. P. Chandraksan and R. W. Brodersen. *Low power digital CMOS design.* Kluwer Academic Publishers, 1995.

[7] M. Diab. Systolic architectures for Multiplication over Finite Field GF(2^m). In *Applied Algebra, Algebraic algorithms and Error-Correcting codes. 8th International Conference, AAECC-8*, pages 329–340, 1991.

[8] G. Drolet. A new representation of elements of Finite Fields GF(2^m) yielding small complexity arithmetic circuits. *IEEE Transactions on Computers*, 47(9):938–946, Sept. 1998.

[9] S.T.J. Fenn, M. Benaissa, and D. Taylor. GF(2^m) Multiplication and Division Over the Dual Basis. *IEEE Transactions on Computers*, 45(3):319–327, Mar. 1996.

[10] S.T.J. Fenn, M. Benaissa, and D. Taylor. Dual basis Systolic Multipliers for GF(2^m). *IEE Proceedings-E*, 144(1):43–46, Jan. 1997.

[11] S.T.J. Fenn, M.G. Parker, M. Benaissa, and D. Taylor. Bit-serial multiplication in GF(2^m) using Irreducible all-one polynomials. *IEE Proceedings-E*, 144(6):391–393, Nov. 1997.

[12] D. M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27:129–146, 1998.

[13] A. Hasan and A. Wassal. VLSI Algorithms, Architectures and Implementation of a Versatile GF(2^m) Processor. Submitted to the IEEE transactions on computers, 1999.

[14] M.A. Hasan and V.K. Bhargava. Bit-Serial Systolic Divider and Multiplier for Finite Fields GF(q^m). *IEEE Transactions on Computers*, 41(8):972–980, Aug. 1992.

[15] M.A. Hasan and V.K. Bhargava. Division and Bit-Serial Multiplication over GF(q^m). *IEE Proceedings-E*, 139(3):230–236, May 1992.

[16] M.A. Hasan and V.K. Bhargava. Modular Construction of low complexity Parallel Multipliers for a Class of Finite Fields GF(2^m). *IEEE Transactions on Computers*, 41(8):962–971, Aug. 1992.

[17] M.A. Hasan and V.K. Bhargava. Architecture for a low complexity rate-adaptive Reed-Solomon encoder. *IEEE Transactions on Computers*, 44(7):938–942, July 1995.

[18] M.A. Hasan, M.Z. Wang, and V.K. Bhargava. A Modified Massey-Omura Parallel Multiplier for a Class of Finite Fields. *IEEE Transactions on Computers*, 42(10):1278–1280, Oct. 1993.

[19] I. Hsu, T. Troung, L. Deutsch, and I. Reed. A Comparison of VLSI Architecture of Finite Field Multipliers using Dual, Normal, or Standard Bases. *IEEE Transactions on Computers*, 37(6):735–739, June 1988.

[20] IEEE. IEEE P1363: Editorial Contribution to Standard for Public-Key Cryptography, August 1999.

[21] S. Ishii, K. Oyama, and K. Yamanaka. A High-Speed Public Key Encryption Processor. *Systems and Computers in Japan*, 29(1):20–32, Jan. 1998.

[22] T. Itoh and S. Tsujii. Computing Multiplicative Inverses in GF(2^m) using Normal Bases. *Information and Computation*, 78(3):171–177, Sept. 1988.

[23] T. Itoh and S. Tsujii. Structure of Parallel Multipliers for a Class of Fields GF(2^m). *Information and Computation*, 83(1):21–40, Oct. 1989.

[24] S.K. Jain, L. Song, and K.K. Parhi. Efficient Semisystolic architectures for Finite Field Arithmetic. *IEEE Transactions on Computers*, 6(1):101–113, March 1998.

[25] D. B. Johnson and A. J. Menezes. Elliptic Curve DSA (ECDSA): An Enhanced DSA. Certicom White Papers, 1998. Available at: `http://www.certicom.com/ecc/wpaper.htm`.

[26] D. Knuth. *The art of computer programming. Vol. 2: Semi-numerical Algorithms.* Reading, Massachusetts: Addison-Wesley, 2nd ed., 1981.

[27] N. Koblitz. Elliptic Curve Cryptosystems. In *Mathematics of Computations*, volume 48, pages 203–209, 1987.

[28] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and their applications.* Cambridge University Press, 1994.

[29] J.L. Massey and J.K. Omura. Computational method and apparatus for Finite Field arithmetic. U.S. Patent 4587627. issued May 1986.

[30] E.D. Mastrovito. *VLSI architectures for computations in Galois field.* PhD thesis, Dept of Electrical Eng., Linköping University, S-581 83 Linköping, Sweden, 1991.

[31] M.C Mekhallalati and A.S. Ashur. Novel structures for Serial Multiplication over the Finite Field. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 15(3):223–245, March 1993.

[32] A. Menezes. *Elliptic Curve Public-Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[33] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[34] A. J. Menezes and S. A. Vanstone. Elliptic Curve Cryptosystems and their Implementations. *Journal of Cryptology*, 6:209–224, 1993.

[35] A.J. Menezes, editor. *Applications of Finite Fields*. Kluwer Academic Publishers, Boston, MA, 1993.

[36] V. Miller. Uses of Elliptic Curves in Cryptography. In *Lecture Notes in Computer Science, Advances in Cryptology: Proceedings of Crypto'85*, volume 218, pages 417–426. Springer-Verlag, Berlin, 1986.

[37] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson. Optimal Normal Bases in GF(p^n). *Discrete Applied Mathematics*, pages 149–161, 1988/1989.

[38] D. Naccache and D. M'Raihi. Cryptographic Smart Cards. *IEEE Micro*, 78(3):14–24, June 1996.

[39] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS), Feb 2000. Available at: `http://csrc.nist.gov/cryptval/dss/fr000215.html`.

[40] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994.

[41] C. Paar. A new architecture for a parallel Finite Field Multipliers with low complexity based on Composite Fields. *IEEE Transactions on Computers*, 45(7):856–861, July 1996.

[42] C. Paar, P. Fleischmann, and P. Roelse. Efficient Multiplier Architectures for Galois Fields GF($2^{\{4n\}}$). *IEEE Transactions on Computers*, 47(2):162–170, Feb 1998.

[43] A. Pincin. A new algorithm for Multiplication in Finite Fields. *IEEE Transactions on Computers*, 38(7):1045–1049, July 1989.

[44] P.A. Scott, S.E. Tavares, and L.E. Peppard. A Fast VLSI Multiplier for GF(2^m). *IEEE Journal on Selected Areas in Communications*, 4(1):62–66, Jan. 1986.

[45] S.Lin and D.J. Costello Jr. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, Inc., 1983.

[46] L. Song and K.K. Parhi. Optimum primitive polynomials for low-area low-power Finite Field Semi-Systolic Multipliers. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 375–384, New York, NY, 1997.

[47] L. Song and K.K. Parhi. Low-Complexity modified Mastrovito Multipliers over GF(2^m). In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCS'99*, pages I508–I512, San Diago, CA, May 1999.

[48] B. Sunar and Ç.K. Koç. Mastrovito Multiplier for all Trinomials. *IEEE Transactions on Computers*, 48(5):522–527, May 1999.

[49] C.-L. Wang and J.-L. Lin. Systolic Array Implementation of Multipliers for Finite Fields GF(2^m). *IEEE Transactions of Circuits and Systems*, 38(7):796–800, July 1991.

[50] C.C. Wang, T.K. Troung, H.M. Shao, L.J. Deutsch, J.K. Omura, and I.S. Reed. VLSI Architectures for Computing Multiplications and Inverses in GF(2^m). *IEEE Transactions on Computers*, 34(8):709–716, Aug. 1985.

[51] A.G. Wassal, M.A. Hasan, and M.I. Elmasry. Low-Power Design of Finite Field Multipliers for Wireless Applications. In *Proceedings of the IEEE 8th Great Lakes Symposium on VLSI*, pages 19–25, Lafayette, LA, Feb. 1998.

[52] E. D. Win and B. Preneel. Elliptic Curve Public-Key Cryptosystems: An Introduction. *State of the Art in Applied Cryptography. Course on Computer Security and Industrial Cryptography. Revised Lectures. Springer-Verlag, Berlin, Germany*, pages 131–141, 1998.

[53] J.J. Wozniak. Systolic dual basis serial multiplier. *IEE Proceedings-E*, 145(3):237–241, May 1998.

[54] C.-W. Wu and M.-K. Chang. Bit-Level Systolic Arrays for Finite-Field Multiplications. *Journal of VLSI Signal Processing*, 10(1):85–92, June 1995.

[55] H. Wu, A. Hasan, and I. Blake. New low-complexity bit-parallel Finite Field multipliers using Weakly Dual Bases. *IEEE Transactions on Computers*, 47(11):1223–1234, Nov. 1998.

[56] H. Wu, M. Hasan, and I. Blake. Finite Field Multipliers using Redundant Basis. To appear in Proceedings of CHES'99, Workshop on Cryptographic Hardware and Embedded Systems, 1999.

[57] C.-S. Yeh, I.S. Reed, and T.K. Truong. Systolic Multipliers for Finite Fields GF(2^m). *IEEE Transactions on Computers*, 33(4):357–360, April 1984.

[58] B.B. Zhou. A New Bit-Serial Systolic Multiplier Over GF(2^m). *IEEE Transactions on Computers*, 37(6):749–751, June 1988.