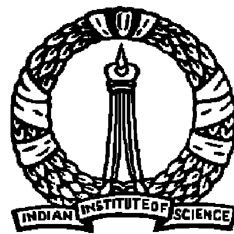
A Polymorphic Finite Field Multiplier

A Thesis
Submitted for the Degree of
Master of Science (Engineering)
in the **Faculty of Engineering**

by

Saptarsi Das



Center for Electronic Design and Technology
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

DECEMBER 2010

To

My parents and my sister

Acknowledgments

I owe my deepest gratitude to both my research advisors - Prof. S. K. Nandy and Prof. H. S. Jamadagni for their constant guidance and patience during the three-year course of my degree. I am heartily thankful to Prof. Nandy, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject. His perpetual energy and enthusiasm in research had motivated all his advisees, including me. In addition, he was always accessible and willing to help his students with their research. As a result, research life became smooth and rewarding for me. Prof. Jamadagni has also given his time and comments at certain important times during my degree. This thesis would not have assumed its current form had it not been for the sincere involvement and guidance provided by both my advisors.

I am indebted to Dr. Ranjani Narayan for her invaluable inputs. The technical as well as mental support that she has provided throughout my degree has made my task as a researcher much easier. I cannot thank her enough for the patience and sincerity she has shown to all my queries.

I was extremely fortunate and privileged to have a very dependable technical feedback group within my lab itself - comprising my lab-mates, Keshavan Varadarajan, Ganesh Garga and Mythri Alle. Their constant encouragement and technical feedbacks laid the path for my research to proceed smoothly. It was an honor for me to work in the same research-group that they belonged to. I offer my sincere gratitude to them for the enthusiasm they have shown towards my work and the patience to my questions (very trivial and weird at times). Besides, I would also like to acknowledge the many enjoyable discussions, both technical and trivial, that I have had with my other lab-mates - Prasenjit Biswas,

Rajdeep Mondal, Gaurav Kumar Singh, Vipin Gemini, Pramod Udupa, Sanjay Kumar, Alexander Fell and Adarsha Rao.

One can only try to acknowledge the blessings of one's parents. It is difficult to write what words cannot say. At this stage of life I can only say but one thing to my parents and my sister – I owe everything to you.

Abstract

Proliferation of various kinds of threats to the security of wireless communications has lead to an increased interest in cryptographic solutions for communication equipments. Traditionally various government agencies had kept cryptographic solutions under stringent regulatory policies in the interest of national security. However with the growth of threat levels of modern day hackers and attackers, strong cryptography has emerged as an inevitable part of different communication protocols for the consumers. As the world moves on to the 3G and 4G era, the desired data rate of wireless communications will escalate to 100Mbps for high mobility stations and upto 1 Gbps for stationary and low mobility stations such as Internet hot-spots. Moreover energy efficiency of a solution is also equally important as is performance or reliability. Most of the cryptography algorithms like the Advanced Encryption Standard (AES) and various elliptic curve cryptography algorithms (ECC) are designed using the algebraic properties of finite fields.

There is another class of applications that rely on finite field algebra, namely the various error detection and correction techniques. Most of the classical block codes used for detection of bit-error in communication over noisy communication channels apply the algebraic properties of finite fields. Cyclic redundancy check is one such algorithm used for detection of error in data in computer network. Reed-Solomon code is one of the most notable classical block codes because of its widespread use in storage devices like CD, DVD, HDD etc.

Performance of these class of applications are heavily dependent on the performance of the underlying field arithmetic kernels. The two operations defined over a finite field are addition and multiplication. Addition is relatively easy to realize in a real life system like

a processor or an ASIC. Multiplication is more expensive than addition in terms of both latency (time) and resource usage (space). In order to meet the demand of high throughput as a system, a cryptographic solution needs to accelerate multiplication operation. There is another dimension to this problem. Different applications in cryptography and error control codes require operations over finite fields of vastly varying order. In order to cater to these finite fields of different orders in an area efficient manner, it is necessary to design integrated solutions, keeping the performance requirements in mind. Due to their small area occupancy and high utilization, such circuits are less likely to stay idle and therefore are less prone to loss of energy due to leakage power dissipation.

In this thesis we present the architecture of a polymorphic multiplier for operations over various extensions of GF(2). We evolved the architecture of a textbook shift-and-add multiplier to arrive at the architecture of the polymorphic multiplier through a generalized mathematical formulation. The polymorphic multiplier is capable of morphing itself in runtime to create data-path for multiplication of various orders. In order to optimally exploit the resources, we also introduced the capability of sub-word parallel execution in the polymorphic multiplier. The synthesis results of an instance of such a polymorphic multiplier shows about 41% savings in area with 21% degradation in maximum operating frequency. We introduced the multiplier as an accelerator unit for field operations in the coarse grained runtime reconfigurable platform called REDEFINE. We observed significant improvement in performance (about 40-50%) of the AES algorithm with the multiplier used as against software realization of multiplication kernels.

Such results may make hardware solutions look attractive, but in certain applications, the sheer size of the finite field makes such a hardware solution infeasible. For example elliptic curve cryptography and hyper elliptic curve cryptography algorithms operate on fields of orders upto 2^{512}. For such big fields development of software algorithms for multiplication is inevitable. Karatsuba-Ofman multi-digit multiplication algorithm is one such algorithm that employs a divide-and-conquer policy for multiplication. However, the underlying operations in a Karatsuba-Ofman realization of multiplication are polynomial multiplications and additions. Unlike field multiplications, in polynomial multiplication

there is no reduction step. Although the polymorphic multiplier has interleaved reduction capability, we show that with a constraint on the input lengths, the polymorphic field multiplier is capable of polynomial multiplications as well. We have used the polymorphic multiplier to realize Karatsuba-Ofman multiplication on REDEFINE and observe a 23x improvement in performance compared to a purely software implementation.

Contents

Acknowledgments **i**

Abstract **iii**

1 Introduction **1**

1.1 Security in Mobile Communication 1

1.1.1 Authentication and Key Arrangement (AKA) in UMTS 3

1.1.2 Access Security in CDMA2000 3

1.1.3 The Blackberry Story 4

1.1.4 Strong Public Key Cryptography: Elliptic Curve Cryptography (ECC) 6

1.2 Error Detection and Correction 6

1.2.1 Error Detection Techniques 7

1.2.2 Error Correction Techniques 7

1.3 Need for Accelerating field operations 8

1.4 Organization of the thesis 9

2 Finite Field Arithmetic **10**

2.1 Finite Field Arithmetic Operations 10

2.1.1 Standard Basis Representation 11

2.2 Finite Field Multiplication in Literature 14

2.2.1 Reconfigurable Finite Field Multiplier 17

2.2.2 Sub-Word Parallel Multiplication 19

2.3 Summary of the chapter 21

3 Architecture of the Polymorphic Multiplier 22

3.1 Architecture of a Field Multiplier 23

3.2 Detailed Architecture of the Polymorphic Field Multiplier 28

3.2.1 Polymorphic, Sub-word Parallel Shifter 28

3.2.2 Polymorphic, sub-word Parallel Reduction Circuit 31

3.2.3 Polymorphic, sub-word Parallel Accumulation Circuit 34

3.2.4 Putting it all together 36

3.3 Multiplication Beyond $GF(2^m)$ 38

3.4 Supporting Shift and Rotate Operations Through Multiplication 41

3.4.1 Parity Check Through Rotation 42

3.5 Supporting Arbitrary Length Multiplication 44

3.6 Summary of the chapter 45

4 Results 46

4.1 Synthesis Results of the Polymorphic Multiplier 46

4.2 The Advanced Encryption Standard: A Case Study 48

4.2.1 REDEFINE: A Coarse Grained Reconfigurable Architecture 49

4.2.2 AES on REDEFINE 50

4.3 Big Multiplications 60

4.3.1 Different algorithms 61

4.4 Summary of the chapter 64

5 Conclusions and Future Work 66

5.1 Conclusions 66

5.2 Future Work 68

Acronyms 69

Bibliography 71

List of Figures

3.1 One bit-slice of the IGF multiplier 26

3.2 One stage of the IGF multiplier. 26

3.3 Schematic diagram of $m \times m$ IGF multiplier 27

3.4 Arrangement of Sub-Words in a Word 32

3.5 Figure showing one stage of the Polymorphic Field Multiplier. 36

3.6 Bit representations of $C(x)$, $C(x) \times x^j$, $C(x) \times x^{r_d}$ 40

List of Tables

2.1 Table showing addition operation defined over $GF(2)$. 11

2.2 Table showing multiplication operation defined over $GF(2)$. 11

2.3 Order and type of finite fields used in different applications. 14

4.1 Table showing the cell area and maximum frequency of operation for the dedicated multipliers considered in the thesis. 47

4.2 Table showing the cell area and maximum frequency of operation for the polymorphic multiplier. 47

4.3 Table showing memory requirements for inversion over different field sizes . 51

4.4 Performance of AES algorithms on *baseline* REDEFINE and a general purpose processor. 54

4.5 Performance improvement of various AES algorithms on REDEFINE by using the polymorphic multiplier. 58

4.6 Performance penalty of memory lookup free implementation of AES algorithms. 59

4.7 Performance Comparison of various multiplication algorithms on REDEFINE. 63

Chapter 1

Introduction

This chapter describes the importance of field arithmetic in the context of cryptography and error control coding, and the need for accelerating field arithmetic operations. Further it puts the work presented in this thesis in context through a brief investigation of the security scenario in wireless communication of the future. We also provide a brief overview of various error detection and correction techniques designed using algebraic properties of finite fields.

1.1 Security in Mobile Communication

In the pre-3GPP era Global System for Mobile Communications (GSM) and IS-95 (based on Code Division Multiple Access (CDMA)) were the two most popular wireless communication standards. GSM was designed to prevent cloning and to be no more vulnerable to eavesdropping than fixed phones. It addresses these goals by providing user-related security features for authentication, confidentiality and anonymity. The authentication feature is intended to allow a GSM network operator to verify the identity of a user such that it is practically impossible for someone to make fraudulent calls by masquerading as a genuine user. Confidentiality protects the user's traffic, both voice and data, and sensitive signaling data, such as dialed telephone numbers, against eavesdropping on the radio path. The anonymity feature was designed to protect the user against someone

who knows the user's International Mobile Subscriber Identity (IMSI) from using this information to track the location of the user, or to identify calls made to or from the user by eavesdropping on the radio path.

The GSM security features have addressed to a very large extent the needs of operators and the aspirations of users. Universal Mobile Telecommunications System (UMTS) security builds on the success of GSM by retaining the security features that have proved to be needed and that are robust. As in the case of GSM, UMTS uses a Universal Subscriber Identity Module, Universal Subscriber Identity Module (USIM) to store all the identification and security-related information that are needed to make or receive a call. Although GSM security has been very successful, an objective of the UMTS security design was to improve on the security of second generation systems like GSM by correcting real and perceived weaknesses. Some of the issues are discussed below.

- The encryption algorithms used for providing confidentiality in GSM services are unpublished and hence are unavailable for peer review. The reason for maintaining such *secrecy* is the considerably tighter regulatory situation in the 1990s. UMTS aims to adopt a more open approach to publish the algorithm specifications with the rest of the UMTS standards.

- The authentication algorithms used for GSM and UMTS need not be standardized. In GSM there was no example algorithm in the standards. This leads to the scenario where the service providers are free to choose different algorithms depending upon their interfacing requirements. It may be noted that COMP-128, one such algorithm was found to be vulnerable to attacks[1]. In order to overcome this shortcoming UMTS has included an example algorithm MILENAGE [2] in the standard specifications.

- The strength of the cipher algorithms used for providing confidentiality depends on the size of the cipher-keys used. In GSM 64 bit keys were in use. However 10 bits out of the 64 bits key were set to zero due to regulatory controls. As the regulatory controls are relaxed now it is possible to use the full 64 bits. Increasing the key

length from 64 bits in GSM is virtually impossible due to the enormous overhead involved in redesigning the algorithms and signaling protocols. UMTS requires a new ciphering mechanism and thus took the opportunity to increase the key length to 128.

- GSM was not explicitly designed to protect against *false base station attacks*.

1.1.1 Authentication and Key Arrangement (AKA) in UMTS

Before any communication can begin in a UMTS network, mutual authentication of the serving network and the terminal has to happen. There are three parties involved in the authentication procedure as listed below.

- Home Network and Authentication Center (AuC)
- Serving Network with Visitor Location Register (VLR)
- Terminal device with the USIM

The authentication is established using a challenge-response protocol. In order to prevent a *man-in-the-middle* from masquerading as a base-station, the challenge and response vectors need to be encrypted. 3GPP recommends the Advanced Encryption Standard (AES) as a kernel function to securely perform this mutual authentication. The Reijndael Encryption algorithm was chosen by the NIST as the successor of the DES. Due to its status as a standard algorithm , its secure implementation and protection against side channel attacks, AES has received a lot of attention from academics and industrial world alike.

1.1.2 Access Security in CDMA2000

CDMA2000 is another important 3G mobile communication standard that emerged from the IMT-2000. Specified by 3GPP2 (www.3gpp2.org), CDMA2000 is the direct successor

of the cdmaOne system. The CDMA2000 system has evolved from cdmaOne in a continuous fashion and therefore retained some of the older security mechanisms. The changes were motivated by the following considerations.

- Cryptanalysis of Cellular Authentication and Voice Encryption (CAVE)[3], Cellular Message Encryption Algorithm (CMEA)[4] and ORYX algorithm[5].
- Realization that 64-bit key is too short.
- Need for Mutual Authentication.

Many aspects of access security are shared between UMTS and CDMA2000. There are four entities participating in the security architecture of CDMA2000 as listed below.

- Home Network and Authentication Center (HLR/AuC)
- Serving Network with Visited Location Register (VLR)
- Mobile Station (MS)
- User Identity Module (UIM)

The AKA protocol used in CDMA2000 is the UMTS AKA mechanism with an exception [6]. CDMA2000 uses a two stage AKA mechanism. The first stage involves transfer of security credentials (authentication vectors) from the home network to the serving network. The second stage involves a challenge-response protocol. The core cryptographic algorithms used in the AKA mechanism are listed in [6]. CDMA2000 uses AES-128 as the encryption algorithm for confidentiality protection.

1.1.3 The Blackberry Story

Sometimes the security for the users conflicts with the requirements for access for national security. Traditionally this has resulted in reduced strength of encryption algorithms. However, there still remains a segment of users for whom security is of primary importance. For example, certain business corporations are unwilling to compromise the security of

their data communications and storages at any cost. In order to cater to such specific demands a completely different section of mobile equipments and services have emerged. The BlackBerry Enterprise Solution is such a solution.

Wireless Data Security

The BlackBerry Enterprise solution offers two different options for encryption over wireless channels. Advanced Encryption Standard and Triple Data Encryption Standard are used for all data communicated between BlackBerry Enterprise Server and BlackBerry smart-phones. Private encryption keys are generated in a secure, two-way authenticated environment and are assigned to each BlackBerry smart-phone user. Each secret key is stored only in two places: in the user's secure enterprise account and on their BlackBerry smart-phone. The secret key can be regenerated wirelessly by the user. Data sent to the BlackBerry smart-phone is encrypted by BlackBerry Enterprise Server using the private key retrieved from the user's mailbox. The encrypted data is decrypted in the smart-phone with the stored secret key. All information remains encrypted in transit and no data is decrypted outside the corporate firewall.

Stored Data Security

Along with security for wireless communications, the BlackBerry Enterprise Solution also offers security for stored data. To secure information stored on BlackBerry smartphones, password authentication can be made mandatory through the customizable IT policies of the BlackBerry Enterprise Server. Local encryption of all data can be enforced via IT policy. Advanced Encryption Standard (AES) encryption algorithm allows password entries to be stored securely on the smart-phone (e.g., banking passwords, PINs, etc.).

Future Smart-phones

As the cost of services and equipments continue to drop due to explosive increase in the volume of the wireless market and rapid advancement in technology, the user-base of smart-phones is rising at a consistent rate. The hand-held devices are becoming more

and more complex with unprecedented set of features and benefits for the users. All the smart-phones of future will have to have protection against security threats such as hackers, industrial espionage, government wire-tap abuses and spies. Therefore encryption capabilities have to be installed in the device itself on external storage cards and network links.

1.1.4 Strong Public Key Cryptography: Elliptic Curve Cryptography (ECC)

Public key cryptography algorithms are used for key arrangement before actual communication can begin between two or more parties. Elliptic Curve Diffie-Hellman Algorithm (ECDHA) is such an algorithm based on algebraic structures of elliptic curves over finite fields. Apart from key arrangement, another particularly useful application of elliptic curve cryptography is in Elliptic Curve Digital Signature Algorithm (ECDSA). The entire security of elliptic curve cryptography depends on the ability to compute a point multiplication and the inability to compute the multiplicand given the original and product points. The point multiplication and point addition operations eventually translates to operations over finite fields of large orders. Thus it suffices to state that the various elliptic curve cryptography algorithms are heavily dependent on algebraic properties of the underlying finite fields.

1.2 Error Detection and Correction

Various error detection and correction or error control techniques are used in communication for reliable delivery of digital data over unreliable communication channels. Error detection is the detection of errors caused by noise or other impairments during transmission from the transmitter to the receiver. Error correction is the detection of errors and reconstruction of the original, error-free data. In the subsequent sections various error control techniques are discussed.

1.2.1 Error Detection Techniques

Among the many schemes used for detection of errors in data transmitted over noisy channels, the most commonly used techniques are realized using suitable hash functions. There is a wide variety of hash function designs. Some of these hash functions are of particular importance because of either their simplicity or their suitability for detecting certain types of errors. Cyclic Redundancy Check (CRC) is a single-burst-error-detecting code and non-secure hash function designed to detect errors in data in computer network. It is realized as a long division over a finite field. CRCs are relatively easy to implement in hardware, and are commonly used in digital networks and storage devices such as compact discs (CD), digital versatile discs (DVD), hard disk drives etc. There are other techniques available for error detection such as parity check, checksums etc.

1.2.2 Error Correction Techniques

The various techniques used for correction of errors in digital data can be broadly divided into two categories namely, Automatic Repeat Request (ARQ) and Forward Error Correction (FEC). ARQ is not suitable under certain circumstances as listed below:

- Low latency transmission such as telephone conversations.
- Where transmitter immediately forgets the transmitted data.
- In absence of a return channel.

While using FEC, the sender encodes the data using some error correcting code before transmission begins. The additional information added along with the transmission facilitates reconstruction of a most probable copy of the original data. FEC employs two different types of error correcting codes, namely block codes and convolutional codes.

Block codes work on fixed-size block of bits. There are many types of block codes. Reed-Solomon[7] code is most notable among classical block codes because of its widespread use in storage devices like CD, DVD, HDD etc. Golay, BCH, Multidimensional parity, and Hamming codes are other examples of classical block codes. Nearly all classical block codes apply the algebraic properties of finite fields.

1.3 Need for Accelerating field operations

As discussed in the previous sections finite fields are widely used in constructing error correcting codes and cryptographic algorithms. In future generation communications the typical data-rate of a communication channel will escalate to 100 Mbps for high mobility stations and upto 1 Gbps for stationary and low mobility stations such as Internet hot-spots [8]. In order to cope with such demands on performance of certain applications, it is essential to accelerate arithmetic operations over finite fields. The most straightforward and perhaps the fastest solution is introduction of Application Specific Integrated Circuit (ASIC) or ASIC-like structures in the form of accelerators to speed-up the finite field operations. The various cryptographic and error correcting algorithms use finite fields of widely varying dimensions (from as small as $GF(2^8)$ to as large as $GF(2^{4096})$). The inherent *rigidity* of ASICs prevent them from being reused for multiple purposes. In order to cater to these finite fields of different orders in an area-efficient manner, it is necessary to design solutions in the form of hardware-consolidations, keeping the performance requirements in mind. Due to their small area occupancy and high utilization, such circuits are less likely to stay idle and therefore are less prone to lose energy due to leakage power dissipation.

In order to achieve a high degree of application scalability, it is necessary to build flexible hardware units. Low form factor devices built for hand-held equipments usually run on batteries. This necessitates development of embedded solutions. The complexity of present day embedded systems are so high that coarse grained reconfigurable architectures are emerging as *future-proof* solutions. REDEFINE is one such platform [9]. In the scope of this thesis a methodology for designing a class of finite field multipliers is discussed. The primary contributions of the work presented in the thesis are

- An exhaustive mathematical formulation of the methodology for designing a class of polymorphic finite field multipliers.
- Evaluation of performance improvement of the Advanced Encryption Standard using such a polymorphic multiplier on REDEFINE.

- Evaluation of performance improvement of Karatsuba-Ofman multi-digit multiplication algorithm using the polymorphic multiplier.

1.4 Organization of the thesis

Chapter 2: In chapter 2 we present a very brief overview of the arithmetic operations over binary fields using polynomial bases. We also present a brief survey of previous works related to multiplications over binary fields using polynomial bases including both software and hardware approaches.

Chapter 3: In chapter 3 we present a detailed mathematical derivation of the polymorphic multiplier architecture. We also prove that the polynomial multiplier is capable of performing some other operations as well.

Chapter 4: In chapter 4 we present a set of experimental results including the hardware synthesis results of an instance of the polymorphic multiplier. We also present a case study on the Advanced Encryption Standard on REDEFINE. Finally we evaluate performances of three well known multiplication algorithms on REDEFINE.

Chapter 5: In chapter 5 we present the conclusions of this research work and a set of possible future works.

Chapter 2

Finite Field Arithmetic

In algebra, a finite field or Galois Field (GF) (named in honor of *Évariste Galois*) is a field that contains only finite number of elements. Finite fields are important in number theory, algebraic geometry, Galois theory, cryptography, and coding theory. Finite field arithmetic differs from integer arithmetic in that, there are only a limited number of elements in the field. Therefore the operations are defined in such a way that the result always belongs to the original finite field. A brief introduction to the basic operations over finite fields is presented in this chapter.

2.1 Finite Field Arithmetic Operations

Finite fields used in designing cryptographic algorithms can be broadly classified into three types namely prime fields, binary fields and optimal extension fields. The scope of the work presented in this thesis is restricted to binary fields (fields of the form $GF(2^m)$). Binary fields are popular in computations due to their feasibility of representation. A Finite field can be viewed as a vector space of finite dimension and therefore representation of an element of a binary field changes depending upon the basis of representation. Different bases can be used to represent the elements of a binary field such as standard basis, normal basis and dual basis. The work presented in this thesis uses standard basis representation.

Table 2.1: Table showing addition operation defined over $GF(2)$.

+	0	1
0	0	1
1	1	0

Table 2.2: Table showing multiplication operation defined over $GF(2)$.

×	0	1
0	0	0
1	0	1

2.1.1 Standard Basis Representation

The standard basis representation of elements of binary fields is presented in this section. Binary fields of the form $GF(2^m)$ are viewed as extensions of the base field $GF(2)$. Two operations are defined over $GF(2)$. They are usually named addition (+) and multiplication (×). Since there are only two elements in $GF(2)$, the elements can be conveniently expressed as a one-bit number. The operations are defined as shown in tables 2.1 and 2.2. As can be seen from table 2.1, addition operation over $GF(2)$ can be realized as modulo-2 addition operation or in other words a logical XOR operation. Multiplication over $GF(2)$ can be realized as logical AND operation. Binary fields are defined as extensions of $GF(2)$, using a special class of polynomials over the base field called irreducible polynomials[10]. As the name suggests, irreducible polynomials are not *reducible* into factors. Let $P(x)$ be an irreducible polynomial of degree m, over $GF(2)$ (i.e. the coefficients of the polynomial belong to $GF(2)$) as defined in equation 2.1.

$$P(x) = x^m + \sum_{i=0}^{m-1} a_i x^i, \; where\; a_i \in GF(2)\; \forall i \tag{2.1}$$

A standard basis is specified by an irreducible polynomial. $P(x)$ can be used to specify a standard basis for $GF(2^m)$. Let us take a small example to demonstrate how an irreducible polynomial generates a finite field. $P(x) = x^4 + x + 1$ is an irreducible polynomial over $GF(2)$. Let us consider the finite field $GF(2^4)$. Let α be an element of $GF(2^4)$, such that α is a root of the equation $P(x) = x^4 + x + 1 = 0$. Then α is called a

primitive element. Therefore the non-zero elements of $GF(2^4)$ can be represented as the set $\{1, \alpha, \alpha^2, \alpha^3, \cdots, \alpha^{14}\}$[10]. Now we will show how the irreducible polynomial $P(x)$ can be used to define a polynomial basis for this finite field. We also note that $\alpha^4 \, mod P(\alpha) = P(\alpha) - \alpha^4 = \alpha + 1$ [10]. Equation 2.2 shows all the elements of $GF(2^4)$ in standard basis defined by the said irreducible polynomial.

$$\begin{aligned}
0 \; mod \, P(\alpha) &= 0 \\
1 \; mod \, P(\alpha) &= 1 \\
\alpha \; mod \, P(\alpha) &= \alpha \\
\alpha^2 \; mod \, P(\alpha) &= \alpha^2 \\
\alpha^3 \; mod \, P(\alpha) &= \alpha^3 \\
\alpha^4 \; mod \, P(\alpha) &= \alpha + 1 \\
\alpha^5 \; mod \, P(\alpha) &= \alpha^2 + \alpha \\
\alpha^6 \; mod \, P(\alpha) &= \alpha^3 + \alpha^2 \\
\alpha^7 \; mod \, P(\alpha) &= \alpha^3 + \alpha + 1 \\
\alpha^8 \; mod \, P(\alpha) &= \alpha^2 + 1 \\
\alpha^9 \; mod \, P(\alpha) &= \alpha^3 + \alpha \\
\alpha^{10} \; mod \, P(\alpha) &= \alpha^2 + \alpha + 1 \\
\alpha^{11} \; mod \, P(\alpha) &= \alpha^3 + \alpha^2 + \alpha \\
\alpha^{12} \; mod \, P(\alpha) &= \alpha^3 + \alpha^2 + \alpha + 1 \\
\alpha^{13} \; mod \, P(\alpha) &= \alpha^3 + \alpha^2 + 1 \\
\alpha^{14} \; mod \, P(\alpha) &= \alpha^3 + 1
\end{aligned} \tag{2.2}$$

Equation 2.2 demonstrates that the elements of $GF(2^4)$ can be represented as polynomials over $GF(2)$, i.e polynomials of the form $a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a_0$ where a_0, a_1, a_2 and a_3 belong to $GF(2)$. In other words each element can be considered as vectors defined with a basis given by $\{1, \alpha, \alpha^2, \alpha^3\}$. Similarly, the elements of $GF(2^m)$ can be represented as polynomials of degree upto $m - 1$ over the base field $GF(2)$ [10]. An example of such a

representation is given in equation 2.3.

$$a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0 \in GF(2^m),\ where\ a_i \in GF(2)\ \forall i \tag{2.3}$$

The string $(a_{m-1}, a_{m-2}, \cdots, a_1, a_0)$ can be used represent the polynomial $P(x)$. The multiplication operation defined over $GF(2^m)$ is performed modulo the irreducible polynomial $P(x)$[10]. Let $A(x)$ and $B(x)$ be two elements of $GF(2^m)$. The addition and multiplication operations are defined as follows:

$$\begin{aligned} Addition\ &:\ C(x) = A(x) + B(x) = (c_{m-1}, c_{m-2}, \cdots, c_1, c_0), \\ &\quad where\ c_i = a_i + b_i\ mod\ 2 \end{aligned} \tag{2.4}$$

$$\begin{aligned} Multiplication\ &:\ C(x) = A(x) \times B(x)\ mod\ P(x)\ where\ the\ operations\ over\ the\ base \\ &\quad field\ are\ given\ in\ tables\ 2.1\ and\ 2.2 \end{aligned} \tag{2.5}$$

Thus it is evident that addition is a trivial XOR operation over binary fields when represented using a standard basis. However, multiplication involves multiplication of polynomials modulo an irreducible polynomial. Therefore multiplication is an expensive operation in terms of latency and it is necessary to improve the efficiency of multiplication operation in an application in order to achieve high performance. Multiplication can be implemented both in software and hardware. A lot of research has been done for designing software algorithms and hardware multipliers to achieve high performance at low cost. Section 2.2 discusses various hardware and software techniques for multiplication over binary fields using standard basis.

Table 2.3: Order and type of finite fields used in different applications.

Application	Prime Field	Binary Field	Extension Field
ECC	160-512	160-512	160-512
HECC	40-256	40-256	40-256
DL (DH, DSA)	1024-4096	1024-4096	1024-4096
Block ciphers (AES))		8	
RS codes		8-16	
Signal processing		8-16	

2.2 Finite Field Multiplication in Literature

As mentioned in section 2.1.1 multiplication can be realized both in software and hardware. Guajardo et al. have discussed software implementations of various finite field arithmetic operations for cryptography in [11]. The authors also present the appropriate field sizes and type of fields for various applications. The table is reproduced here for the reader's convenience. From table 2.3 two important observations can be made. Firstly binary fields are most popular among the different types of finite fields. Secondly, the order of field varies widely among different applications. The order of binary fields used in applications like ECC and Hyper-Elliptic Curve Cryptography (HECC) algorithms are so huge that direct multiplication in hardware becomes infeasible. Though it is possible to implements ASICs for such *big* multiplications, the ASICs cannot be integrated with general purpose processors to act as accelerators for finite field operations due to the vast mismatch in granularity of operations. In other words, in order to design a flexible system for finite field operations that can operate on these fields of vastly different orders, development of software algorithms for multiplication is inevitable.

In [12], the authors have introduced several efficient algorithms for multiplication over $GF(2^m)$. The algorithms include *right-to-left comb method*, the *left-to-right comb method* and the *left-to-right comb method with window width w*. These algorithms outperform the traditional shift and add method when implemented in software. However the shift and add method is more suitable for implementation in hardware where shift operation is inexpensive. In [13] the authors A. Karatsuba and Y. Ofman have presented an algorithm for multi-digit multiplication. This algorithm is popularly known as the Karatsuba-Ofman

algorithm. The Karatsuba-Ofman employs a *divide-and-conquer* policy to break multplication in higher order fields into polynomial operations of lower degree. Hankerson et al. have discussed and compared the software implementations of the aforementioned algorithms in [14]. It is shown that traditional *shift-and-add* technique performs worse compared to the other techniques. Among the two comb techniques of multiplication, *left-to-right comb method* runs faster on a general purpose processor because it involves less number of shift operations. Karatsuba-Ofman algorithm is shown to perform better than both the comb techniques. The *left-to-right comb method with window width* w involves memory lookup of precomputed polynomial products. This technique is shown to be faster than all the other techniques. However, in multi-core systems where multiple kernels can be *active* at the same time, memory lookup based implementations for core kernels of a security application have certain vulnerability issues. For instance, if a global memory is used to feed all the cores of a system, a certain rogue kernel running on one of the cores can access stored data critical to a security kernel running on a different core. One such example of attack on an implementation of the AES algorithm is presented in [15]. In chapter 4 we evaluate the performances of this software algorithms for multiplication on REDEFINE. However, we decided to leave out the memory-lookup based algorithms from this evaluation due to their vulnerability to side-channel attacks.

Hardware solutions are usually faster than their software counterparts. With the advent of applications like cryptography and error correction codes coupled with demand for high throughput, hardware realization of multiplications over binary fields gained popularity. Hardware binary field multipliers can be broadly classified into two categories: bit-serial multipliers and bit-parallel multipliers. Gregory C. Ahlquist et al. have conducted an extensive survey of performances of various hardware multipliers for finite fields and presented the results in [16]. Brief descriptions of each of the multiplier architectures are reproduced here to familiarize the reader with the basics of hardware finite field multipliers. It is to be noted that, only binary multipliers using standard bases are discussed here.

- *Shift-and-Add Multiplier*: A shift and add multiplier employs the textbook shift

and technique for multiplication. The multiplier and the multiplicand are treated as polynomials over $GF(2)$. The multiplication is performed modulo an irreducible polynomial. The reduction of the result can be performed in an interleaved manner or after the result is produced. In [17] the two techniques are described in detail. The first technique is called Interleaved Galois Field Multiplier (IGF) and the second technique is called Modular Galois Field Multiplier (MGF). In the IGF multiplier the reduction process (modulo operation with the irreducible polynomial) is interleaved with the shift and add process. Whereas, in the MGF multiplier, the polynomial multiplication and the reduction processes are performed in a modular manner. The hardware multiplier presented in this thesis is based upon the IGF multiplier. In section 2.2.1 we present the motivation for choosing the IGF multiplier as the base architecture.

- *Linear Feedback Shift Register Multiplier*: The Linear Feedback Shift Register (LFSR) Multiplier is considered by many, the standard or canonical design for finite field multipliers [16]. The LFSR multiplier has parallel inputs and outputs, but processes one block of data at a time. A detailed description of LFSR multiplier can be found in [18]. Note that, the LFSR based architecture is suitable for reconfigurable multiplier design, but the latency of the architecture is strictly dictated by the size of the operands or in other words, the order of the finite field on which it operates.

- *Mastrovito Multiplier*: E. D. Mastrovito provided a completely combinatorial finite field multiplier design in his doctoral thesis [19],[20]. He showed that the hardware complexity of multiplication operation in standard basis can be reduced by choosing a special class of irreducible polynomials. In his work, he chose the trinomials of the form $x^m + x^n + 1$ as irreducible polynomials to reduce the result of multiplication. Later, Alper Halbutogullari and Çetin Kaya Koç presented a formulation based on the Mastrovito Multiplier for multplication over binary fields generated by arbitrary irreducible polynomials[21].

- *Hasan-Bhargava Multiplier*: Hasan and Bhargava designed a bit-serial finite field

multiplier implemented as a systolic array [22]. The multiplier streams bits sequentially through multiple stages of processing units with the product flowing bit-serially from the last processor. Each processing unit stage consists of storage and a small amount of combinatorial logic.

- *Paar-Rosner Multiplier*: Christof Paar and Martin Rosner presented an architecture of a parallel finite field multiplier using a composite basis in [23]. The composite basis facilitates representation of elements from a higher order finite field as polynomials over a lower order finite field [10]. For example the binary field $GF(2^{16})$ can be represented as $GF((2^8)^2)$. Therefore any element belonging to $GF(2^{16})$ can be represented as a polynomial over $GF(2^8)$. Thus multiplication and addition operation are performed over the lower order finite field. This allows an efficient implementation of multiplication. The Parr-Rosner multiplier has parallel inputs and outputs and single clock cycle latency.

2.2.1 Reconfigurable Finite Field Multiplier

As shown in table 2.3, the order of binary fields differ widely among different applications and application domains. In order to cater to these variable field order requirements which eventually translate to variable word-length requirements, traditionally there has been some interest in designing flexible multipliers that can work over binary fields of various orders. It is to be noted that unlike integer multipliers, field multipliers are not inherently reconfigurable. In other words an integer multiplier designed for $m-bit \times m-bit$ multiplication is capable of performing $m'-bit \times m''-bit$ multiplication, where $m', m'' \leq m$. On the contrary a finite field multiplier designed for a particular field order cannot perform multiplications over fields of lower orders. Thus, flexibility or reconfigurability across field orders is not *free* in finite field multipliers. It is widely recognized that bit-serial architectures are more suitable for reconfigurable design, whereas bit-parallel architectures perform better in terms of latency. Since bit-serial architectures process one bit of the inputs in a clock cycle to produce one bit of the output, the latency of multiplication over $GF(2^m)$ is m cycles. On the other hand, bit-parallel architectures produce the

result in one clock cycle. Moreover serial architectures occupy less area than parallel architectures. In general, bit-parallel $GF(2^m)$ multipliers have an area requirement of $\mathcal{O}(m^2)$, whereas bit-serial multipliers for binary fields of same order have area requirement of $\mathcal{O}(m)$. This is an example of the typical space-time trade-off involved in all architectural design methodologies. Two well known *flexible* architectures for binary field multipliers are presented here to further elaborate the trade-off analysis. Kitsos et al. have discussed a design of a bit-serial reconfigurable multiplier in [24]. Paar et al. have proposed a hybrid multiplier based on composite basis in [25]. The Kitsos-Theodoridis-Koufopavlou Multiplier (KTK) is a reconfigurable MSB-first architecture that can be used for variable field order m. The hardware implementation consists of a bit-sliced linear feedback shift register with some extra logic as part of control-path. It produces the result after m clock cycles. The KTK multiplier enjoys significant amount of flexibility both in terms of order of binary fields and irreducible polynomials used to generate those binary fields. The Paar-Fleischmann Multiplier (PF) uses algebraic techniques of composite field arithmetic to represent $GF(2^m)$ as $GF((2^n)^k)$, where $m = nk$. Therefore the elements of the bigger field $GF(2^m)$ are represented as polynomials over $GF(2^n)$. All component-wise multiplications and additions are performed in a bit-parallel manner over the sub-field $GF(2^n)$. A bit-serial structure is used over the extension field to compute the final product. Thus the latency is reduced from m to $\frac{m}{n}$. The value of k can be changed to provide a limited amount of flexibility. It is to be noted that, if m is a prime number and $n > 1$ the multiplier cannot operate. In summary, the serial KTK multiplier trades performance for flexibility. The PF multiplier offers good performance with limited flexibility. Therefore it is evident that, a *unified* multiplier for various field orders should be capable of delivering flexibility without paying in terms of latency.

In the work presented in this thesis, we have chosen the traditional shift-and-add multiplication technique as the base architecture. It is to be noted that the multiplier presented in this thesis does not conform to either of the two traditional approaches. It is also not a hybrid architecture like the PF multiplier. Since the proposed architecture is very regular and scalable, it is easy to introduce flexibility with minimal overhead in

the control logic. Moreover, unlike the bit-serial architecture the latency of operation is not dependent on the size of the operands. The regularity of the architecture makes the multiplier amenable for a pipelined design. We intentionally do not define the pipeline-depth of the multiplier. Rather, the cycle-count is left as a parameter for design space exploration by varying the pipeline depth, in the context of the specific computation platform. We refer to the multiplier as a polymorphic multiplier since it can *morph* itself to create the data-path for a set of binary fields of different orders on demand.

In the past, efforts have been made to increase performance of various finite field multipliers by fixing the irreducible polynomial to some specific class of polynomials like trinomials, pentanomials etc[19, 20]. However, certain domains of applications require the flexibility of supporting any arbitrary irreducible polynomial for any given order of the finite field. Therefore we do not to restrict the polymorphic multiplier to any specific irreducible polynomial or any class of irreducible polynomials.

2.2.2 Sub-Word Parallel Multiplication

Sub-word parallel processing, a form of Single Instruction Multiple Data (SIMD) processing, is one key feature among recent multimedia processors, general-purpose processors with multimedia extensions, and digital signal processors. Sub-word parallel processing partitions operands into multiple lower precision operands called sub-words and operates on these sub-words in parallel [26]. Krithivasan et al. have proposed a design of such a sub-word parallel multiplier in [26]. Wei-Ming Lim and M. Benaissa have proposed an architecture for sub-word parallel multiplications over binary fields in [27]. The multiplier can support two different modes of operation corresponding to two different field orders. For example, one instance of such a multiplier can operate over $GF(2^m)$ or $GF(2^q)$ where $m = p \times q$. While operating over the lower order field i.e. $GF(2^q)$, the multiplier can perform p parallel multiplications. Though the architecture is described in a generic manner, the parameters p and q cannot be varied at runtime. In other words, once designed, an instance of such a multiplier is capable of operating on only two different binary fields.

The architecture of the polymorphic multiplier described in this thesis is generic and scalable. One such instance of a polymorphic multiplier is capable of operating over $GF(2^m)$ and any lower order field. For example the multiplier designed for $GF(2^{32})$ can also operate on $GF(2^{16})$, $GF(2^8)$, $GF(2^4)$ and $GF(2^2)$. As expected from any sub-word parallel architecture, the polymorphic multiplier is capable of running multiple parallel threads of multiplication while operating over a lower order field. Continuing the example, the multiplier can support one multiplication over $GF(2^{32})$ or two multiplications over $GF(2^{16})$ or four multiplications over $GF(2^8)$ etc. Note that multiplication over $GF(2)$ is equivalent to logical AND operation. Thus we decided not to support multiplication over $GF(2)$ in the polymorphic multiplier. In the aforementioned example the lower order fields are of the form $GF(2^q)$ where q is an integral power of 2. Binary fields of this form is of particular interest in computation since any element of such a field can be conveniently expressed using words, bytes, nibbles etc. However for certain applications it is necessary to operate on fields of the form $GF(2^q)$ where q is not an integral power of 2. We show that the polymorphic multiplier is also capable of operating on such binary fields with some preprocessing of the operands and post processing of the result. This makes the polymorphic multiplier capable of supporting *any* binary field. S. Roy has introduced a multiply-accumulate (MAC) unit in [28]. The primary objective of the MAC unit is to act as an accelerator for finite field operation for a error detection and correction system. Roy has chosen Cyclic Redundancy check for error detection and Reed-Solomon code for error correction. One of the operands in the MAC unit is restricted to 8-bits. Therefore it can only represent elements from $GF(2^8)$. Thus the MAC unit is capable of multiplication over $GF(2^8)$. However, for higher order fields $GF(2^{16})$ or $GF(2^{32})$, all elements cannot be represented and therefore the architecture is not suitable for generic multiplication over $GF(2^{16})$ or $GF(2^{32})$. On the contrary, the polymorphic multiplier reported in this thesis is not restricted to any particular application or any class of applications. Thus it is capable of performing generic multiplication over any binary field. It is important to note that, we have arrived at the architecture of the polymorphic multiplier through a complete mathematical formulation and therefore the design is correct by construction. Also

the generality of the mathematical description of the architecture makes the architecture easily scalable. A detailed description of the architecture of the polymorphic multiplier is the subject of chapter 3.

2.3 Summary of the chapter

In this chapter we presented a very brief mathematical background of finite field arithmetic operations. We introduced the widely used polynomial basis (also known as standard or canonical basis) representation of elements in a binary fields, which is the most commonly used type of finite field in cryptography and other domains. We discussed the *difficulty* of multiplication operations over binary fields and presented a brief survey of various architectures and algorithms for multiplication. We also presented the need for developing reconfigurable and sub-word parallel multipliers and the various issues associated with it. In this context we compared the architecture presented in this thesis with previous reconfigurable and sub-word parallel architectures.

Chapter 3

Architecture of the Polymorphic Multiplier

In this chapter we build the foundation for designing the hardware logic of a polymorphic field multiplier through mathematical formalization. Later we show how the polymorphism of the proposed multiplier can be exploited to perform shift and rotate operations. As explained in section 2.1.1 in standard basis, the elements of binary fields of different orders are represented as polynomials of different degrees. For example, we represent elements of $GF(2^m)$ as polynomials of degree $m-1$, where the coefficients of the polynomial belong to $GF(2)$. Thus elements of $GF(2^m)$ are represented as a string of m bits. In the present scope of the work we have used standard basis to represent elements of finite fields.

Rest of the chapter is organized as follows. In section 3.1 we introduce the architecture of the well known Interleaved Galois Field multiplier[17], that forms the backbone of our proposed polymorphic multiplier. Sections 3.2.1, 3.2.2, and 3.2.3 describe the methodology of designing sub-word parallel, polymorphic shifter, reduction circuit and accumulation circuit, which are the components that compose a multiplier. In section 3.2.4 we put together the individual components of the multiplier to form different *stages* of the multiplier and then proceed to define the lower bound of *logic-density* at each

stage. In section 3.3 we prove that the polymorphic multiplier can also perform polynomial multiplication without addition of extra logic in the design. This behavior facilitates the usage of Karatsuba-Ofman algorithm[13] for performing multiplications over fields of order higher than the word length of the polymorphic multiplier. In section 3.4 we describe the mechanism to support shift and rotate operations through multiplication. This is extended in section 3.5 to support arbitrary field order multiplications.

3.1 Architecture of a Field Multiplier

Based on the Interleaved Galois Field multiplier reported in [17], below we reproduce some of the mathematical steps as preliminaries to evolve an architecture for polymorphic finite field multiplier. Consider two elements $A(x)$ and $B(x)$ belonging to $GF(2^m)$. The product of these two elements is given by

$$C(x) = (A(x) \times B(x)) \, mod \, P(x), \tag{3.1}$$

where $P(x)$ is an irreducible polynomial of order m. The computation of this multiplication proceeds according to the shift-and-add algorithm known for binary multiplication. However, a modular reduction is performed at each step of the multiplication to ensure that the result is always a valid element of the underlying finite field.The reduction technique is based on the following observation[10].

$$x^m \, mod \, P(x) = P(x) - x^m \tag{3.2}$$

Consider an element $A(x)$ that belongs to $GF(2^m)$ which has the form $A(x) =$

$\sum_0^{m-1} a_s x^s$. After multiplication of $A(x)$ with x we have

$$\begin{aligned} x \times A(x) &= \sum_{s=0}^{m-1} a_s x^{s+1} \, mod \, P(x) \\ &= a_{m-1} x^m \, mod \, P(x) + \sum_{s=0}^{m-2} a_s x^{s+1} \, mod \, P(x) \\ &= a_{m-1}(P(x) - x^m) + \sum_{s=0}^{m-2} a_s x^{s+1} \end{aligned} \quad (3.3)$$

Note that addition in $GF(2^m)$ is realized as bitwise XOR of the bit-string representations of the polynomials. From equation 3.3 it is evident that reduction process is translated to a conditional XOR operation. The conditional XOR is driven by the MSB of the input polynomial $A(x)$. We also observe that for reduction it is sufficient to represent $(P(x) - x^m)$ as a bit-string. Thus in the subsequent descriptions we will represent $(P(x) - x^m)$ as an m-bit wide string and we will refer to it as the irreducible polynomial. The equivalent logical expressions to emulate equation 3.3 are shown below:

$$A_{sh}[i] = \begin{cases} 0 & if \; i = 0 \\ A[i-1] & otherwise \end{cases} \quad (3.4)$$

$$A_{red}[i] = \begin{cases} A_{sh}[i] \oplus P[i] & if \; A[m-1] = 1 \\ A_{sh}[i] & otherwise \end{cases} \quad (3.5)$$

where $A[i]$ represents the ith bit of the bit string. $A_{sh}[i]$ represents the ith bit of the shifted polynomial. $A_{red}[i]$ represents the ith bit of the reduced result polynomial. $P[i]$ represents the ith bit of the irreducible polynomial (ignoring the term x^m). Multiplication by a higher power of x can be achieved by repeated application of the aforementioned technique.

In order to multiply the polynomial $A(x)$ with another polynomial $B(x)$ we express the multiplication operation as a repeated shift and accumulate process as described in

equation 6.

$$(A(x) \times B(x))\; mod\, P(x) = \sum_{s=0}^{m-1} b_s((A(x) \times x^s)\; mod\, P(x)) \tag{3.6}$$

where $B(x) = \sum_{s=0}^{m-1} b_s x^s$. To produce the final result we accumulate the terms $b_0 \times A_0(x), b_1 \times A_1(x), b_2 \times A_2(x)...b_{m-2} \times A_{m-2}(x), b_{m-1} \times A_{m-1}(x)$, where $A_s(x) = (A(x) \times x^s)\; mod\, P(x)$. We rename the accumulated result till the sth term as $R_s(x)$. Therefore the accumulation process can be expressed as follows:

$$\begin{aligned} R_s(x) &= R_{s-1}(x) + b_{s-1} \times A_{s-1}(x)\; for\; s > 0\; and \\ R_0(x) &= 0 \end{aligned} \tag{3.7}$$

The term $(A(x) \times x^s) modP(x)$ is generated in hardware by passing $A(x)$ through s one-bit left shifters and reduction circuits corresponding to their boolean descriptions in equation 3.4 and 3.5. The equivalent logic expression for equation 7 is given in equation 8.

$$R_s[i] = \begin{cases} R_{s-1}[i] \oplus A_{s-1}[i] & if\; B[s-1] = 1 \\ R_{s-1}[i] & otherwise \end{cases} \tag{3.8}$$

In figure 3.1 the bit-slice of the multiplier is shown that realizes equations 3.4, 3.5 and 3.8. For an $m \times m$ multiplier we have replicated the bit-slice m times to form one *stage* of the multiplier (refer to figure 3.2). Figure 3.3 shows that m such stages are employed to produce each of the accumulated results. The stages are numbered from 0 to $m - 1$. We denote the stage number by s. The final result is given by $C(x) = R_m(x)$. In other words the result is produced after stage $s = m - 1$.

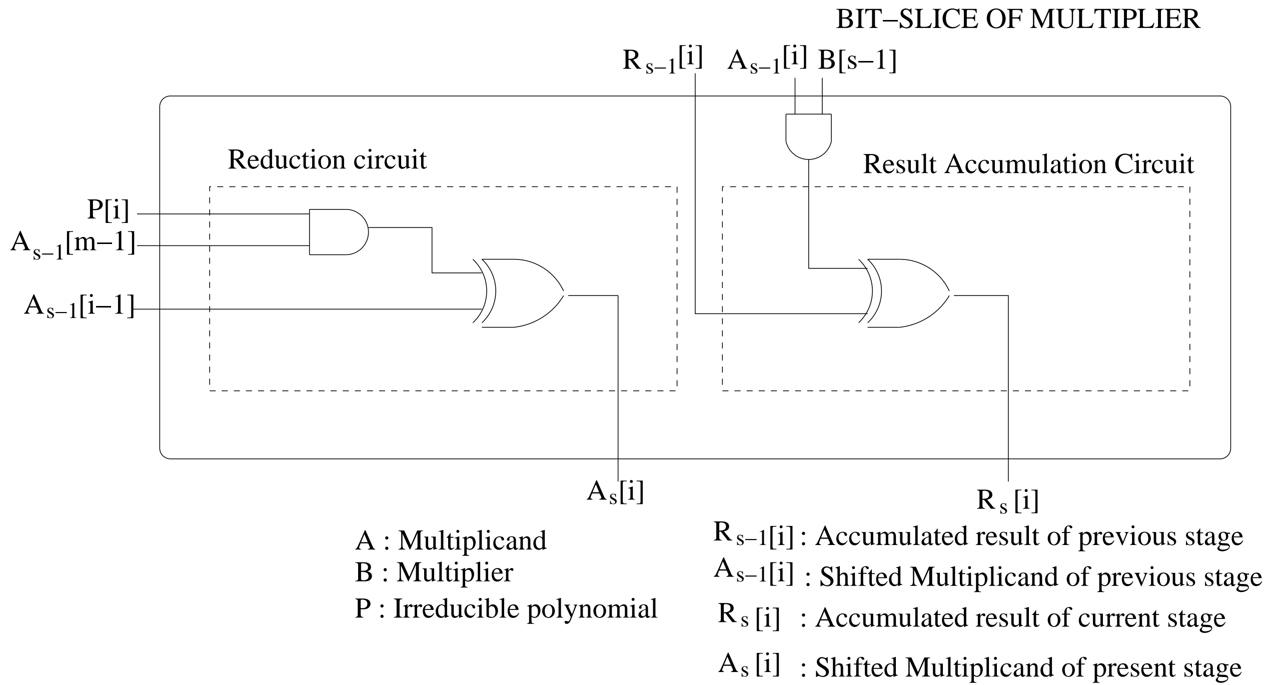


Figure 3.1: One bit-slice of the IGF multiplier

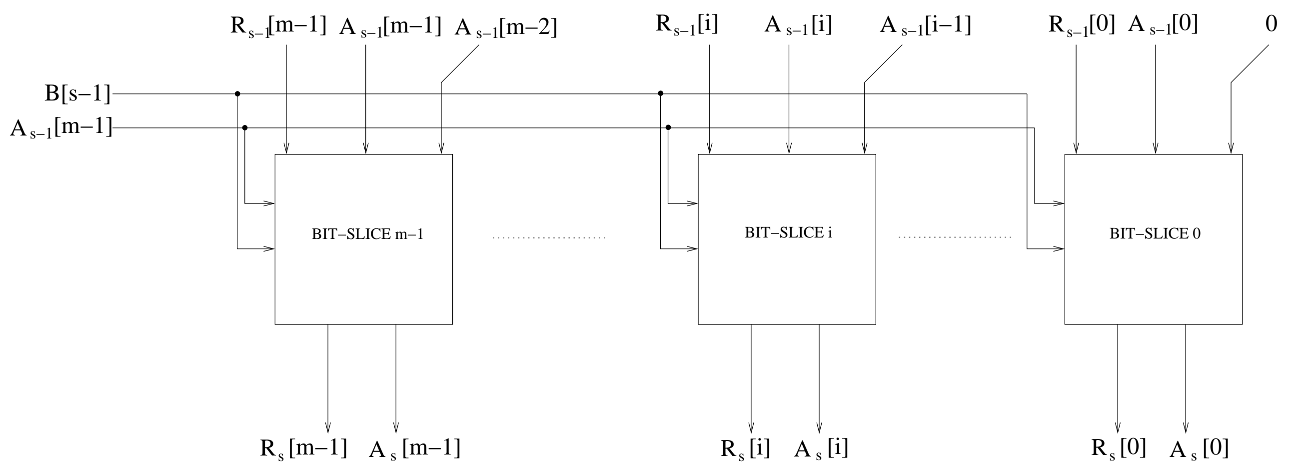


Figure 3.2: One stage of the IGF multiplier.

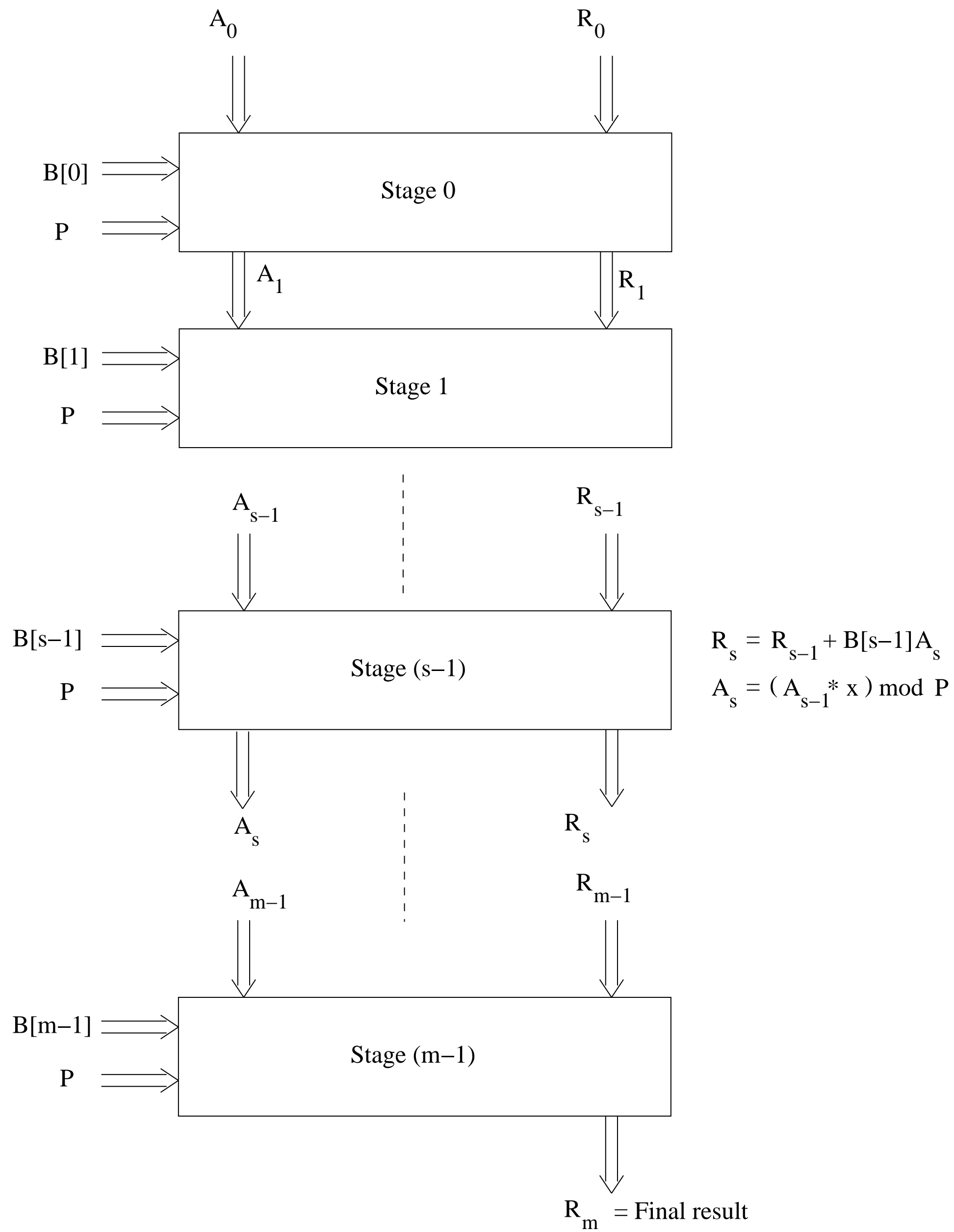


Figure 3.3: Schematic diagram of $m \times m$ IGF multiplier

3.2 Detailed Architecture of the Polymorphic Field Multiplier

In this section we develop a methodology for designing a polymorphic finite field multiplier based on the multiplier architecture described in section 3.1. This architecture can cater to various field order i.e. word-length requirements originating from different application domains. We perform lower order multiplications in a SIMD manner in order to maximally utilize resources pertaining to a higher order field multiplication. Therefore such an $m \times m$ polymorphic finite field multiplier will support lower order $n \times n$ field multiplications as well under the constraint

$$n \in \{2^l, 2^{l+1}, 2^{l+2}.., 2^{l+k}\}\ where\ l + k \leq \log_2 m \tag{3.9}$$

The polymorphic multiplier will also create data-paths for $\frac{m}{2^l}$ sub-word multiplications for sub-word length of 2^l. It is to be noted that we have restricted the sub-word lengths to powers of 2 because of their usefulness in different application domains. Later in section 3.4 we discuss a technique to overcome this constraint by some preprocessing on the input operands. In section 3.1 we have identified that the basic operations involved in finite field multiplication in standard basis are one bit logical left shift, reduction and accumulation of results. In the following sections 3.2.1, 3.2.2 and 3.2.3 we describe the design of each of these components to support polymorphism and sub-word parallel execution. For each of the components we derive the appropriate mathematical expressions that describe the behavior of the circuits, in a generalized fashion so as to maintain scalability.

3.2.1 Polymorphic, Sub-word Parallel Shifter

Consider an element of $GF(2^m)$, represented within an m-bit bit-string. For m-bit field multiplication, we need to left shift the entire bit-string at each stage, inserting a zero at the LSB position. Similarly, for a lower order n-bit mode of multiplication, we need to insert zeroes at the respective LSB positions, viz positions $0, n, 2 \times n$, etc. Thus, the

length of the sub-word influences the *locations* in the bit-string where zeroes need to be inserted. In other words we need to identify the boundaries of the sub-words packed in the m-bit bit-string. Assuming that the m-bit bit-string comprises smaller sub-words, each of length n-bits, we define a set $\mathcal{B}_n$ containing the indices of the LSBs of the smaller n-bit sub-words.

$$\mathcal{B}_n = \{0, n, 2 \times n, ..., p \times n\} \; where \, p = \left\lfloor \frac{m}{n} \right\rfloor - 1 \tag{3.10}$$

When m is a multiple of n, we get the following definition of $\mathcal{B}_n$.

$$\mathcal{B}_n = \{0, n, 2 \times n, ..., p \times n\} \; where \, p = \frac{m}{n} - 1 \tag{3.11}$$

Using equation 3.11 we can summarize the left shift operation as shown in equation 3.12, where A is the input bit-string of length m-bits and A_{sh} is the left shifted result.

$$A_{sh}[i] = \begin{cases} 0 & \forall i \in \mathcal{B}_n \\ A[i-1] & otherwise \end{cases} \tag{3.12}$$

Note that equation 3.12 describes one bit left shift operation upon an m-bit bit-string comprising smaller n-bit sub-words. Our aim is to design a one bit left shifter that can operate in $k+1$ different modes corresponding to the different sub-word sizes listed in equation 3.9. We incrementally arrive at the final design by starting with two different sub-word sizes 2^r and 2^q bits. For simplicity we will call these two modes of operation as mode 2^r and mode 2^q respectively. Note that all the sub-words for a specific mode of operation are of same length. Moreover the shifter can operate in only one mode at a time. In other words the two modes are mutually exclusive. In order to derive an optimized set of boolean expressions for each of the bit position we identify the positions of the bit-string where zeroes need to be inserted under different operating modes. We

observe that for positions with $i \in \mathcal{B}_{2^r} \cap \mathcal{B}_{2^q}$ a zero is to be inserted irrespective of the mode of operation. For bits with indices $i \in \mathcal{B}_{2^r} - \mathcal{B}_{2^q}$, a zero is to be inserted only when operating in mode 2^r. Otherwise $A_{sh}[i]$ gets the value of $A[i-1]$. Similarly for bits with indices $i \in \mathcal{B}_{2^q} - \mathcal{B}_{2^r}$, a zero is to be inserted only when operating in mode 2^q. Otherwise $A_{sh}[i]$ gets the value of $A[i-1]$. For positions whose indices do not belong to either of the two sets of LSBs, $A_{sh}[i]$ gets the value of $A[i-1]$ irrespective of the mode. The specific logic expressions for the different bit-positions are given in equation 3.13.

$$A_{sh}[i] = \begin{cases} 0 & \forall i \in \mathcal{B}_{2^r} \cap \mathcal{B}_{2^q} \\ 0 \; when\; mode\; 2^r & \forall i \in \mathcal{B}_{2^r} - \mathcal{B}_{2^q} \\ A[i-1] \; when\; mode\; 2^q & \forall i \in \mathcal{B}_{2^r} - \mathcal{B}_{2^q} \\ 0 \; when\; mode\; 2^q & \forall i \in \mathcal{B}_{2^q} - \mathcal{B}_{2^r} \\ A[i-1] \; when\; mode\; 2^r & \forall i \in \mathcal{B}_{2^q} - \mathcal{B}_{2^r} \\ A[i-1] & \forall i \in \overline{\mathcal{B}_{2^q}} \cap \overline{\mathcal{B}_{2^r}} \end{cases} \tag{3.13}$$

Equation 3.13 can be realized in hardware as a series two input multiplexers. For instance, for bits with indices $i \in \mathcal{B}_{2^r} - \mathcal{B}_{2^q}$, we introduce a signal called M_r which is high when operating in mode 2^r. Thus the multiplexer can be realized as given in equation 3.14, with M_r as the select signal.

$$A_{sh}[i] = (0 \wedge M_r) \vee (A[i-1] \wedge \overline{M_r}) = A[i-1] \wedge \overline{M_r} \tag{3.14}$$

Similarly the select signal M_q for mode 2^q is introduced. Therefore, equation 3.13 can be rewritten as follows.

$$A_{sh}[i] = \begin{cases} 0 & \forall i \in \mathcal{B}_{2^r} \cap \mathcal{B}_{2^q} \\ A[i-1] \wedge \overline{M_r} & \forall i \in \mathcal{B}_{2^r} - \mathcal{B}_{2^q} \\ A[i-1] \wedge \overline{M_q} & \forall i \in \mathcal{B}_{2^q} - \mathcal{B}_{2^r} \\ A[i-1] & i \in \overline{\mathcal{B}_{2^q}} \cap \overline{\mathcal{B}_{2^r}} \end{cases} \tag{3.15}$$

We also observe from equation 3.11 that $\mathcal{B}_{2^r} \subset \mathcal{B}_{2^{r-1}}$. With this observation we present the general expression that takes into account modes 2^r and 2^{r-1}.

$$A_{sh}[i] = \begin{cases} 0 & \forall i \in \mathcal{B}_{2^r} \\ A[i-1] \wedge \overline{M_{r-1}} & \forall i \in \mathcal{B}_{2^{r-1}} - \mathcal{B}_{2^r} \\ A[i-1] & otherwise \end{cases} \tag{3.16}$$

We generalize equation 3.16 further to incorporate modes corresponding to sub-word lengths $2^l, 2^{l+1}, ..., 2^{l+k}$ as given below.

$$A_{sh}[i] = \begin{cases} 0 & \forall i \in \mathcal{B}_{2^{l+k}} \\ A[i-1] \wedge \overline{(\vee_{r=l}^{l+k-1} M_r)} & \forall i \in \mathcal{B}_{2^{l+k-1}} - \mathcal{B}_{2^{l+k}} \\ A[i-1] \wedge \overline{(\vee_{r=l}^{l+k-2} M_r)} & \forall i \in \mathcal{B}_{2^{l+k-2}} - \mathcal{B}_{2^{l+k-1}} \\ ... & \\ ... & \\ ... & \\ A[i-1] \wedge \overline{M_l} & \forall i \in \mathcal{B}_{2^l} - \mathcal{B}_{2^{l+1}} \\ A[i-1] & otherwise \end{cases} \tag{3.17}$$

Using equation 3.17 we describe a logical left shifter that can operate in $k+1$ different modes in hardware.

3.2.2 Polymorphic, sub-word Parallel Reduction Circuit

The aforementioned shifter is used to multiply polynomials of various lengths with x. To *reduce* the result, in general, we perform conditional XOR of the shifted operand and the irreducible polynomial. In order to obtain a sub-word parallel reduction circuit, we need to be able to choose the *appropriate* MSB of the input polynomial for driving the conditional XOR. When operating in mode 2^r the bit-string A comprises sub-words of length 2^r. Consider $A_j(x)$, the j-th sub-word of length 2^r-bit from the input polynomial (refer to figure 3.4). After multiplying $A_j(x)$ with x, i.e shifting $A_j(x)$ towards left by

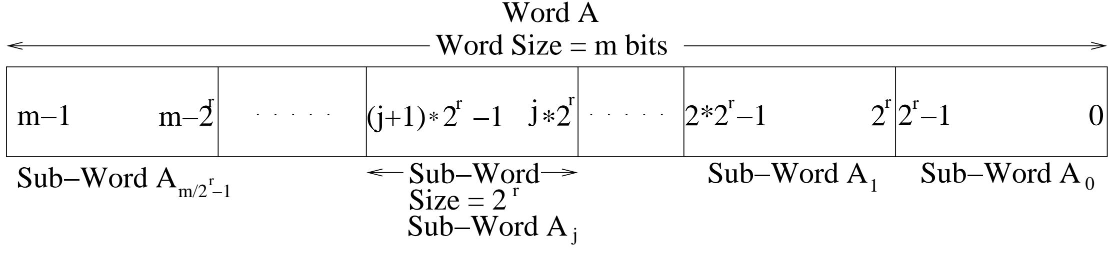


Figure 3.4: Arrangement of Sub-Words in a Word

one bit we represent the shifted polynomial as $A_{j,sh}(x)$. If the MSB of $A_j(x)$ is set then the shifted result needs to be reduced. We represent the j-th sub-word of the irreducible polynomial as $P_j(x)$. The output of the reduction operation is represented as $A_{j,red}(x)$. The reduction operation performed upon each of the bits of the polynomial is described in the equation 3.18.

$$A_{j,red}[i] = \begin{cases} A_{j,sh}[i] \oplus P_j[i] & if\ A_j[2^r - 1] = 1 \\ A_{j,sh}[i] & otherwise \end{cases} \tag{3.18}$$

The logical operations described in equation 3.18 can be realized in hardware as a multiplexer described as follows:

$$\begin{aligned} A_{j,red}[i] &= ((A_{j,sh}[i] \oplus P_j[i]) \wedge A_j[2^r - 1]) \vee \\ &\quad (A_{j,sh}[i] \wedge \overline{A_j[2^r - 1]}) \\ &= A_{j,sh}[i] \oplus (P_j[i] \wedge A_j[2^r - 1]) \end{aligned} \tag{3.19}$$

From equation 3.19, we observe that $A_j[2^r - 1]$ is used to mask all the bits from $i = 0$ to $2^r - 1$ of irreducible polynomial $P_j(x)$ before adding with $A_{j,sh}(x)$. In order to conveniently express this operation for all the sub-words of length 2^r we define a bit string $Mask_1$ given in equation 3.20.

$$Mask_1[i] = \begin{cases} A[2^r - 1] & \forall i \in [0, 2^r - 1] \\ A[2 \times 2^r - 1] & \forall i \in [2^r, 2 \times 2^r - 1] \\ A[3 \times 2^r - 1] & \forall i \in [2 \times 2^r, 3 \times 2^r - 1] \\ ... & \\ ... & \\ ... & \\ A[\frac{m}{2^r} \times 2^r - 1] & \forall i \in [m - 2^r, m - 1] \end{cases} \quad (3.20)$$

Equation 3.20 can be written as a general expression as given by equation 3.21.

$$Mask_1[i] = A[(\left\lfloor \frac{i + 2^r}{2^r} \right\rfloor \times 2^r) - 1] \quad (3.21)$$

Using equation 3.21 we generalize the reduction operation of the polynomial $A_{sh}(x)$ as described in equation 3.19 for all the bits from $i = 0$ to $m - 1$ as follows:

$$A_{red}[i] = A_{sh}[i] \oplus (Mask_1[i] \wedge P_j[i]) \quad (3.22)$$

where, $A_{sh}(x)$ is the shifted result and $A_{red}(x)$ is the reduced result. Note that, equation 3.21 describes the mask required for operation in mode 2^r. Thus we generalize equation to incorporate modes corresponding to sub-word lengths $2^l, 2^{l+1}, ..., 2^{l+k}$ as given in equation 3.23.

$$Mask_1[i] = \vee_{r=l}^{l+k} (A[(\left\lfloor \frac{i + 2^r}{2^r} \right\rfloor \times 2^r) - 1] \wedge M_r) \quad (3.23)$$

where M_r as introduced in equation 3.14, is a signal that is high only when operating in mode 2^r. This $Mask_1$ bit-string is bitwise ANDed with the irreducible polynomial

bit-string. The result of the AND is then bitwise XORed with A to obtain the reduced result (refer to equation 3.22).

3.2.3 Polymorphic, sub-word Parallel Accumulation Circuit

As explained in section 3.1, the accumulation logic is also a conditional XOR circuit (refer to equations 3.7 and 3.8). The modules described in sections 3.2.1 and 3.2.2 operate on one of the operands (which we have consistently referred to as $A(x)$). However the accumulation circuit that we will describe in this section operate on both the operands of multiplication namely $A(x)$ and $B(x)$. For convenience we will call $A(x)$ as multiplicand and $B(x)$ the multiplier. There is another important distinction between the modules of 3.2.1 and 3.2.2 and the accumulation circuit. The sub-word parallel shifter and the reduction circuit are stage-agnostic i.e the operations of these circuits are not dependent on any information about the stage of the multiplier where they belong. The accumulation circuit is however, strongly dependent upon the stage of the multiplier as can be seen from equation 3.8. In stage s, the selection between the two choices is based on the bit $B[s]$ (refer to figure 3.3), when sub-word length is m bits. When operating in the mode 2^r the polynomials $A(x)$ and $B(x)$ comprise elements from $GF(2^r)$. In stage s, the bits from B that are chosen to drive the conditional XOR circuit are $\{s, s+2^r, s+2\times 2^r, ..., s+(\frac{m}{2^r}-1)\times 2^r\}$. Let $A_s(x)$ denote the multiplicand multiplied with x^s. Therefore $A_s(x)$ denotes shifted and reduced multiplicand polynomial till s-th stage (refer to figure 3.3). Consider j-th sub-words $A_{j,s}(x)$ and $B_j(x)$ from the operands $A_s(x)$ and $B(x)$. Let us denote the accumulated result till the s-th stage as $R_{j,s}(x)$ and the accumulated result after s-th stage as $R_{j,s+1}(x)$. Therefore in stage s, the accumulation process can be described as follows:

$$R_{j,s+1}[i] = \begin{cases} R_{j,s}[i] \oplus A_{j,s}[i] & if\ B_j[s] = 1 \\ R_{j,s}[i] & otherwise \end{cases} \tag{3.24}$$

We observe that $B_j[s] = B[s + j \times 2^r]$. The conditional XOR circuit is realized in

hardware by a series of multiplexers as described in equation 3.25.

$$\begin{aligned} R_{j,s+1}[i] &= ((R_{j,s}[i] \oplus A_{j,s}[i]) \wedge B_j[s]) \vee (R_{j,s}[i] \wedge \overline{B_j[s]}) \\ &= R_{j,s}[i] \oplus (A_{j,s}[i] \wedge B_j[s]) \end{aligned} \tag{3.25}$$

From equation 3.25 we observe that $B_j[s]$ is used to mask all bits of $A_{j,s}(x)$ before adding with $R_{j,s}(x)$. To conveniently express the accumulation operation for all the sub-words of length 2^r we define a bit-string called $Mask_2$ given by equation 3.26.

$$Mask_2[i] = \begin{cases} B[s] & \forall i \in [0, 2^r - 1] \\ B[2^r + s] & \forall i \in [2^r, 2 \times 2^r - 1] \\ B[2 \times 2^r + s] & \forall i \in [2 \times 2^r, 3 \times 2^r - 1] \\ ... & \\ ... & \\ ... & \\ B[\frac{m}{2^r} \times 2^r + s] & \forall i \in [m - 2^r, m - 1] \end{cases} \tag{3.26}$$

Equation 3.26 is further generalized to a single expression as given by equation 3.27 .

$$Mask_2[i] = B[(\lfloor (i/2^r) \rfloor \times 2^r) + s] \tag{3.27}$$

Using equation 3.27 we can write the general expression for all the bits of $A_s(x)$ and $B(x)$ as given in equation 3.28.

$$R_{s+1}[i] = R_s[i] \oplus (Mask_2[i] \wedge A_s[i]) \tag{3.28}$$

Now we proceed to present the expression that describes the behavior of the circuit

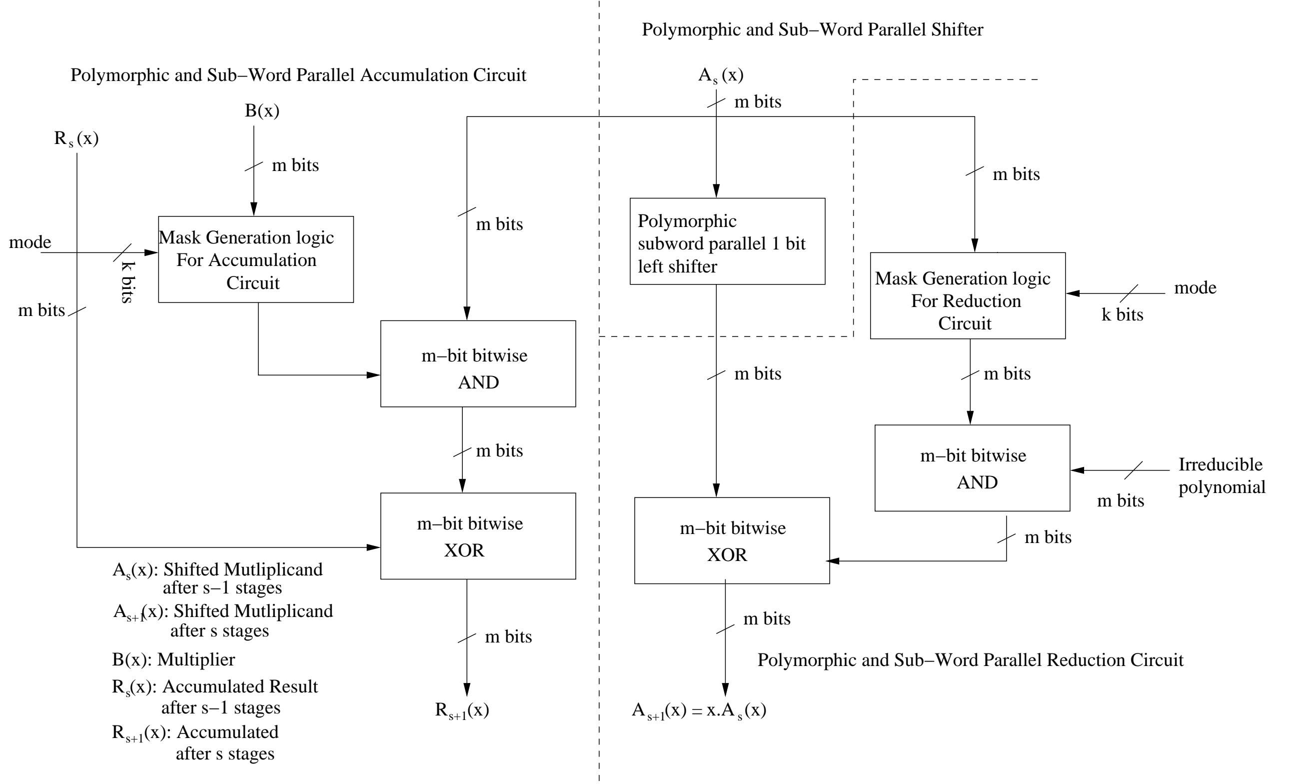


Figure 3.5: Figure showing one stage of the Polymorphic Field Multiplier.

incorporating modes corresponding to sub-word lengths $2^l, 2^{l+1}, ..., 2^{l+k}$.

$$Mask_2[i] = \vee_{r=l}^{2^{l+k}} (B[(\lfloor (i/2^r) \rfloor \times 2^r) + s] \wedge M_r) \tag{3.29}$$

where s is the stage number, ranging from 0 to $2^{l+k} - 1$.

$Mask_2$ is logically ANDed with the shifted operand and then added with accumulated result to generate the result (refer to equation 3.28).

3.2.4 Putting it all together

In this section we put together the components developed in sections 3.2.1, 3.2.2 and 3.2.3 and compose the polymorphic finite field multiplier. Figure 3.5 shows the interconnections among the components that constitute one stage of the multplier. The dotted line divides the stage into the three major components. In general multiplication of elements belonging to $GF(2^m)$ involves m such stages of shift, reduction and accumulation. In order to

perform multiplication in a sub-word parallel manner we introduced the capacity of supporting different modes of operation in the basic one bit left shifter and the conditional XOR circuits in section 3.2. Now we show how stage number affects the *logic-density* in each stage. Consider a sub-word parallel multiplier that can multiply elements that belong to two different finite fields of order m and n. Let us assume $m > n$. Therefore the multiplier must have $max(m, n) = m$ number of stages where m number of stages should be capable of handling elements from $GF(2^m)$ and n number of stages should be capable of handling elements from $GF(2^n)$. In order to reuse the hardware resources pertaining to higher order multiplication (in this example mode m) for lower order multiplications (in this example mode n), it is sufficient to support both the modes in the first n stages. Subsequent $m - n$ stages need not support mode n. Thus we note that the minimum number of modes that each stage needs to support varies from one stage to another. Therefore deploying only the minimum number number of modes that are required in each stage for functional correctness guarantees optimized hardware in terms of area.

We now proceed to present a general expression that will determine the number of modes each stage needs to support. Note that we have restricted the design to modes corresponding to sub-word sizes that are integral powers of two. Let us denote all the supported modes by the set $\mathcal{S}_m = \{2^l, 2^{l+1}, ..., 2^{l+k}\}$. The number of stages required in the multiplier is given by $max(2^l, 2^{l+1}, ..., 2^{l+k}) = 2^{l+k}$. Thus the range of the stage number s is from 0 to $2^{l+k} - 1$. When operating in mode 2^l the result is produced after stage $s = 2^l - 1$. Therefore it is evident that there is no need to support the mode 2^l in the subsequent stages. Similarly after stage $s = 2^{l+1} - 1$ there is no need to support mode 2^{l+1}. In order to minimize the hardware resource usage for supporting these different modes we derive the minimum number of modes that need to be supported as functions of stage number s . Let $\mathcal{R}_s$ denote the set of modes necessary for a particular stage s. The relationship between the stage number and the different modes of operation is shown below.

$$
\begin{aligned}
\forall s \in [0, 2^l - 1] \quad \mathcal{R}_s &= \mathcal{S}_m \\
\forall s \in [2^l, 2^{l+1} - 1] \quad \mathcal{R}_s &= \mathcal{S}_m - \{2^l\} \\
\forall s \in [2^{l+1}, 2^{l+2} - 1] \quad \mathcal{R}_s &= \mathcal{S}_m - \{2^l, 2^{l+1}\} \\
&\dots \\
&\dots \\
&\dots \\
\forall s \in [2^{l+k-1}, 2^{l+k} - 1] \quad \mathcal{R}_s &= \mathcal{S}_m - \{2^l, 2^{l+1}, ..., 2^{l+k-1}\}
\end{aligned}
\tag{3.30}
$$

We can rewrite equation 3.30 as a common expression as given below.

$$
\mathcal{R}_s = \mathcal{S}_m - \{2^j : l \leq j \leq \lfloor \log_2(s+1) \rfloor\} \tag{3.31}
$$

Equation 3.31 presents the minimum number of modes of operation that each stage of the polymorphic multiplier needs to support. Therefore, using equation 3.31 we can rewrite equations 3.17, 3.23, and 3.29 to restrict the number of modes for each stage to the minimum necessary limit. This guarantees functional correctness with an optimized hardware.

3.3 Multiplication Beyond $GF(2^m)$

In section 3.2 we have presented the design of a polymorphic multiplier that can multiply elements of $GF(2^n)$, where $1 < n \leq m$. In this section we provide a method for reusing this multiplier for multiplications of elements $GF(2^n)$, where $n > m$. In [13] the authors A. Karatsuba and Y. Ofman proposed an efficient multiplication algorithm for multi-digit numbers. The Karatsuba-Ofman algorithm can be applied for finite field multiplications as well. An example of such multiplication is given. Consider $A(x)$ and $B(x)$ to be

elements in $GF(2^n)$. We can break $A(x)$ and $B(x)$ into two-term polynomials as follows

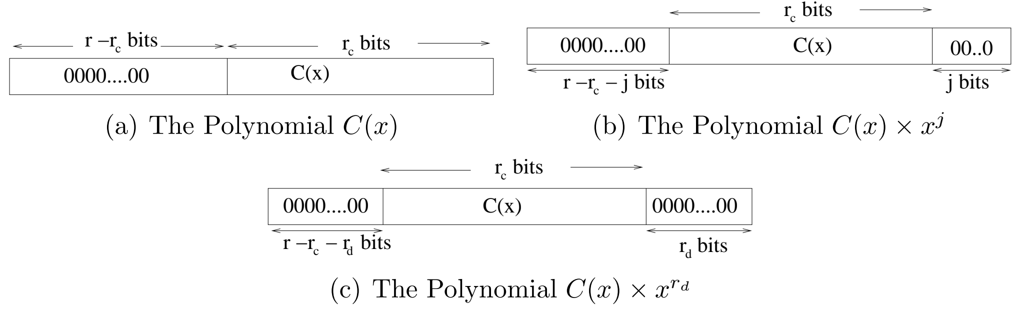
$$\begin{aligned} A(x) &= A_h(x)x^{n'} + A_l(x) \\ B(x) &= B_h(x)x^{n'} + B_l(x) \\ & \quad where\, n > n' \end{aligned} \tag{3.32}$$

Thus the multiplication operation of $A(x)$ and $B(x)$ can be written as given in equation 3.33.

$$\begin{aligned} A(x) \times B(x) &= (A_h(x)x^{n'} + A_l(x)) \times (B_h(x)x^{n'} + B_l(x)) \\ &= (A_h(x)B_h(x))x^{2n'} + (A_h(x)B_l(x) + \\ & \quad (A_l(x)B_h(x))x^{n'} + (A_l(x)B_l(x)) \end{aligned} \tag{3.33}$$

Equation 3.33 shows a method of *divide-and-conquer* to break polynomial multiplication into smaller polynomial multiplications. This procedure can be applied to the polynomials $A(x)$ and $B(x)$ iteratively to reach even lower order polynomials. After multiplication of the polynomials, the addition operations are performed to form the unreduced result. Then the result is reduced using an irreducible polynomial. Therefore in order to support Karatsuba-Ofman algorithm we need polynomial multipliers, adders and reduction circuit.

The polymorphic multiplier described in section 3.2 forms data path of multipliers for various orders of extension fields of $GF(2)$ in run time. We will now show that the field multiplier can also perform polynomial multiplication. Let us assume that the polymorphic multiplier is in a mode to multiply elements from $GF(2^r)$. Let us consider two polynomials $C(x)$ and $D(x)$ of degrees r_c and r_d respectively. Multiplication of these two polynomials is shown in equation 3.34.



(a) The Polynomial $C(x)$ (b) The Polynomial $C(x) \times x^j$

(c) The Polynomial $C(x) \times x^{r_d}$

Figure 3.6: Bit representations of $C(x)$, $C(x) \times x^j$, $C(x) \times x^{r_d}$

$$\begin{aligned} C(x) \times D(x) &= (\sum_{i=0}^{r_c} c_i x^i) \times (\sum_{j=0}^{r_d} d_j x^j) \\ &= \sum_{j=0}^{r_d} d_j (C(x) \times x^j) \end{aligned} \tag{3.34}$$

From equation 3.34 we notice that in order to compute the result of the multiplication operation, we need to accumulate the terms $d_0 C(x)$, $d_1 C(x) \times x$, $d_2 C(x) \times x^2 ... d_{r_d - 1} C(x) \times x^{r_d - 1}$, $d_{r_d} C(x) \times x^{r_d}$. In figure 3.6(a), $C(x)$ is represented in an r-bit string. The $r - r_c$ bits from the MSB are zeroes. Overflow will not occur even after shifting $C(x)$ to the left by r_d bits, if $r_c + r_d < r$. Therefore the shifters in the first $r_d + 1$ stages of the polymorphic multiplier can be employed to generate the terms without any loss of bits. We also note that, at stage j, reduction is performed if MSB of $d_j C(x) \times x^j$ is set to 1. From figure 3.6 it is evident that none of the polynomials $d_j C(x) \times x^j$ from $j = 0$ to r_d has a 1 in the MSB position. Therefore the reduction circuits of the first $r_d + 1$ stages of the polymorphic multiplier will have no effect on the result.

Therefore, if the sum of the degrees of the input polynomials ($r_c + r_d$ in the above example) is less than the order of the extension of $GF(2)$ in which the polymorphic multiplier is operating (r in the above example), then the behaviour of the polymorphic multiplier is equivalent to that of a polynomial multiplier. Thus we can apply the Karatsuba-Ofman algorithm to break bigger field multiplications and express them in

terms of smaller polynomial multiplications. However, separate hardware or software implementation is necessary to obtain the final result. Final result is obtained by selective additions of the results of the polynomial multiplications and reduction. It is important to note that as many parallel polynomial multiplications as permissible by the polymorphic multiplier can be run to enhance performance of the Karatsuba-Ofman algorithm.

3.4 Supporting Shift and Rotate Operations Through Multiplication

In this section we show that the polymorphic multiplier can behave as a polymorphic shift/rotate circuit as well. In order to prove our claim we make the following observations. Consider an m-bit number A. Let us assume that A is to be shifted by z bits towards the left. The result of the shift is equivalent to the product of the two polynomials $A(x)$ and $Z(x)$, where $Z(x) = x^z$. In section 3.3 we have shown that polymorphic multiplier behaves as a polynomial multiplier for certain range of inputs. The restriction on the inputs was imposed to eliminate loss of information due to truncation. In case of shift, loss of bits is not a concern. Therefore the polymorphic multiplier behaves as a shifter when the irreducible polynomial input is set to zero and the amount of of shift is passed to the multiplier in a one-hot-encoded format. This technique is applicable for left shifts only. For a right shift operation, through a preprocessing step the bits of the input are reversed and this is followed by the left shift operation. the output of the multiplier is reversed to produce the output in desired format.

Rotate operations can also be performed using the multiplier. One bit left rotate operation on the polynomial $A(x)$ is a one bit left shift followed by a conditional XOR and can be represented as shown in equation 3.35.

$$A_{rot}(x) = A(x) \times x + A_{MSB} \tag{3.35}$$

$A_{rot}(x)$ is the output of one bit left rotation and a A_{MSB} is the MSB of $A(x)$. This

operation is equivalent to a shift and reduce operation. In order to employ the one bit left shifter and a reduction circuit that are present in each stage of the polymorphic multiplier for this operation we rewrite equation 3.35 as follows.

$$A_{rot} = \begin{cases} A_{sh} \oplus 1 & if\ A[m-1] = 1 \\ A_{sh} & otherwise \end{cases} \tag{3.36}$$

A_{sh} is binary representation of $A(x) \times x$ and $A[m-1]$ is the MSB of A. It is evident from equation 3.36 that the irreducible polynomial input of the shift and reduce circuit of the polymorphic multiplier needs to be set to 1 in order to perform one bit rotation. Thus at each stage of the polymorphic multiplier the input number A rotates one bit towards the left. Therefore to perform a z-bit left rotate, the number A needs to be extracted after z rotations. We achieve this behavior by passing the amount of rotate (i.e. z) in a one-hot-encoded format (i.e. x^z) to multiplier operand. Thus only the rotated number till the z-th stage of the polymorphic multiplier contribute to the accumulation process. Therefore we conclude that the polymorphic multiplier behaves as a left rotator when the number to be rotated is passed to the multiplicand, amount of rotate is passed to the multiplier operand and the irreducible polynomial input is set to 1. Similar to shift operations, right rotations can be achieved by bit reversal of the numbers before and after the multiplication process. The polymorphic nature of the multiplier facilitates polymorphic shift and rotate operation.

3.4.1 Parity Check Through Rotation

In this section we show that the polymorphic multiplier is capable of even parity check through multiplication. Let us consider an m-bit number $A = (a_{m-1}, a_{m-2}, \cdots, a_1, a_0)$. Let p_A be the even parity of A as given by the equation 3.37.

$$EvenParity(A) = p_A = a_{m-1} \oplus a_{m-2} \oplus \cdots \oplus a_1 \oplus a_0 \tag{3.37}$$

Now we will present the method of computing even parity of a number through rotation. Consider the polynomial representation $A(x) = \sum_{i=0}^{m-1} a_i x^i$ of the number A. Let $A_j(x)$ represent the polynomial rotated towards left by j bits. Then the number rotated by one bit i.e. $A_1(x)$ is given by equation 3.38

$$A_1(x) = \sum_{i=1}^{m-1} a_{i-1} x^i + a_{m-1} \tag{3.38}$$

Generalizing equation 3.38 we get the following expression for the number rotated towards left by j bits.

$$A_j(x) = \sum_{i=j}^{m-1} a_{i-j} x^i + \sum_{i=0}^{j-1} a_{m+i-j} x^i \tag{3.39}$$

Note that when $A_m(x) = A(x)$, i.e. the original number is reproduced after m number of rotations. Let us now consider the sum of all the rotated numbers as given in the following equation.

$$\begin{aligned} \sum_{j=0}^{m-1} A_j(x) &= \sum_{j=0}^{m-1} (\sum_{i=j}^{m-1} a_{i-j} x^i + \sum_{i=0}^{j-1} a_{m+i-j} x^i) \\ &= \sum_{i=0}^{m-1} (\sum_{j=0}^{m-1} a_j).x^i \\ &= \sum_{i=0}^{m-1} p_A x^i \end{aligned} \tag{3.40}$$

From equation 3.40 we observe that the sum of all the rotated numbers has the even parity of the original number at each of the bit positions. As discussed in section 3.4, the polymorphic multiplier is capable of rotating the multiplicand input towards left at each stage. By setting all the bits of the multiplier operand to one, we make the accumulator circuit to perform the operation of equation 3.40.

3.5 Supporting Arbitrary Length Multiplication

Finite fields of order 2^n, where n is a power of 2 are very popular in different application domains such as cryptography and error correction coding. Thus, we have motivated the design of a polymorphic multiplier that can support direct multiplication over such extensions of $GF(2)$. However in certain applications, multiplications are performed over extensions of order 2^n, where n is not a power of 2. We show that the polymorphic multiplier described in this thesis can also perform such multiplications with some pre-processing on two of the input operands followed by post-processing of the result. Consider two polynomials $A(x)$ and $B(x)$ belonging to $GF(2^{n_1})$. Let us assume that $P(x)$ is an irreducible polynomial that generates $GF(2^{n_1})$. Let us also assume that n_1 is not a power of 2. Equation 3.41 shows how a multplication in a lower order field can be represented as a multiplication over a higher order field.

$$(A(x) \times x^n) \times B(x)\, mod\, (P(x) \times x^n) = (A(x) \times B(x)\, mod\, P(x)) \times x^n \qquad (3.41)$$

In order to multiply $A(x)$ and $B(x)$ we identify the next higher order extension $GF(2^{n_2})$ such that n_2 is a power of 2. We make the multiplicand and the irreducible polynomial input MSB-justified by left shifting $A(x)$ and $P(x)$ by $n_2 - n_1$ bits. Therefore the new multiplicand and irreducible polynomial becomes $A(x) \times x^{n_2-n_1}$ and $P(x) \times x^{n_2-n_1}$. Multiplication over $GF(2^{n_2})$ is directly supported by the polymorphic field multiplier. As dictated by equation 3.41 the result needs to be LSB-justified by right shifting n_2-n_1 bits. The polymorphic nature of the multiplier ensures that multiplication of any field order can be performed in a sub-word parallel manner, with support for runtime reconfiguration between various field orders.

3.6 Summary of the chapter

In this chapter we presented the architecture of the polymorphic multiplier. The polymorphic multiplier is based on the well known shift and add technique. Starting with a description of the shift and add technique we derived a general mathematical description of the polymorphic architecture. We also presented some *additional* features of the polymorphic multiplier namely, its ability to perform polynomial multiplications and shift/rotate operations. We also presented a method for computing even parity of a number through rotation. Finally we showed how the polymorphic multiplier can perform arbitrary length multiplications in its operating range in a sub-word parallel manner.

Chapter 4

Results

In this chapter we present a set of experimental results to conclude the discussion about the polymorphic multiplier. Section 4.1 describes area and performance results of a hardware instance of the polymorphic multiplier. In section 4.2 we present and analyze performances of AES encryption and decryption algorithms. In this context we bring out the improvement in throughput of AES algorithms using the polymorphic multiplier as hardware accelerator for field multiplication. REDEFINE, a coarse grain reconfigurable architecture is used as the target platform for implementing these algorithms. We present a brief description of REDEFINE in section 4.2.1. In section 4.3, we revisit various algorithms for multiplication over large finite fields (refer to section 2.2). We evaluate their performances on REDEFINE and present the improvement of performance of the Karatsuba-Ofman algorithm using the polymorphic multiplier for polynomial multiplications.

4.1 Synthesis Results of the Polymorphic Multiplier

The polymorphic multiplier described in chapter 3 is capable of multiplication over finite fields of the form $GF(2^{2^r})$. The architecture was described in a generalized manner. One such polymorphic multiplier designed for multiplications over $GF(2^m)$ is capable of multiplications over subfields given by $GF(2^r)$ where $r \in \{2^i : l \leq i \leq \log_2 m\}$. In the

Table 4.1: Table showing the cell area and maximum frequency of operation for the dedicated multipliers considered in the thesis.

	Cell Area (Sq. μ)	Max Operating Frequency
2-bit field multiplier	$A_1 = 288$	3.22 GHz
4-bit field multiplier	$A_2 = 792$	1.88 GHz
8-bit field multiplier	$A_3 = 2518$	1.01 GHz
16-bit field multiplier	$A_4 = 11645$	0.86 GHz
32-bit field multiplier	$A_5 = 41742$	458.71 MHz

Table 4.2: Table showing the cell area and maximum frequency of operation for the polymorphic multiplier.

	Cell Area (Sq. μ)	Max Operating Frequency
Polymorphic multiplier without optimized number of modes	$A_p = 81438$	273.97 MHz
Polymorphic multiplier with optimized number of modes	$A_p = 50511$	357.14 MHz
Combination of dedicated multipliers	$\sum_{i=1}^{5} \frac{32}{2^i} \times A_i = 86048$	$\leq$ 458.71 MHz

present exercise we set $m = 32$. Such a multiplier can perform sub-word multiplications where the sub-word lengths are given by the set $\mathcal{S}_{32} = \{2^1, 2^2, 2^3, 2^4, 2^5\}$.

The architecture of the interleaved field multiplier for standard basis described in section 3.1 was described in Verilog and then instantiated for five operand bit-widths 2, 4, 8, 16 and 32. These form the reference architectures for comparison. All the designs were functionally verified using ModelSim ISE and then synthesized using Synopsys Design Vision with 90nm UMC libraries. The architectures dedicated to specific field orders were compared with respect to their area consumption, maximum operating frequency and throughput. The results are tabulated in table 4.1.

The polymorphic multiplier is capable of performing sixteen 2×2 or eight 4×4 or four 8×8 or two 16×16 or one 32×32 multiplications at any time. An equivalent platform that can support all the modes of operation will need a combination of sixteen 2×2, eight 4×4, four 8×8, two 16×16 and one 32×32 multipliers. The area of such a platform, as listed in the third row of table 4.2 is given by $\sum_{i=1}^{5} \frac{32}{2^i} \times A_i$ (refer to table 4.1 for values of individual A_is). Table 4.2 also gives the area and maximum operating

frequency of the polymorphic multiplier. We have synthesized the polymorphic multiplier with and without the optimization (to support only the minimum necessary number of modes of operation in each stage) described in section 3.2.4. The area occupied and the maximum operating frequency for both the implementations are listed in table 4.2. The polymorphic multiplier with optimized number of modes in each stage occupies 38% less area compared to the unoptimized polymorphic multiplier. Reduction in number of modes in each stage translates to shorter combinational delay in the individual stages. Hence the optimized multiplier has a 30% higher maximum frequency of operation than the unoptimized multiplier. From table 4.2 we note that both the implementations of polymorphic multiplier occupies less area than the combination of dedicated multiplier. However the optimized polymorphic multiplier occupies 41% less area than the combination of dedicated multipliers. The combination of the dedicated multipliers is expected to operate at a frequency determined by its slowest component (in this example the 32-bit multiplier). From table 4.2 we note that the optimized polymorphic multiplier has a maximum operating frequency 21% less than the 32-bit field multiplier. However this slow down is compensated by the flexibility provided to support various field orders in a SIMD manner.

4.2 The Advanced Encryption Standard: A Case Study

The Advanced Encryption Standard (AES) is a symmetric key cryptography standard based on the Rijndael algorithm designed by Joan Daemen and Vincent Rijmen [29]. The standard comprises three different algorithms AES128, AES192 and AES256. The block size for each of these algorithms is 128 bits and the key sizes are 128 bits, 192 bits and 256 bits. AES was chosen as the successor of the Data Encryption Standard (DES) by the National Institute of Standards and Technology (NIST) in 2001. In this section we discuss the various kernels of AES and their realization on REDEFINE. In the section 4.2.1, we reproduce a brief overview of REDEFINE from [9, 30] for the readers' convenience.

4.2.1 REDEFINE: A Coarse Grained Reconfigurable Architecture

REDEFINE is a fabric of hardware resources comprising Compute Element (CE) with local storage and routers that communicate over RECONNECT, a network on chip (NoC) [31]. Its resources are controlled by the Support Logic. In REDEFINE, diverse data paths are composed in terms of computational structures at runtime. A computational structure is a physical aggregation of hardware resources that can perform a coarse grained operation, referred as a Hyper Operation (HyperOp). In REDEFINE, application specific data-paths are defined in terms of computational structures. In REDEFINE, applications are specified in a High Level Language (HLL) and are compiled into coarse grained operations containing metadata which capture the computation and communication requirements. This information is used to compose computational structures at runtime.

RETARGET is a compiler tool chain that is used to compile applications to an intermediate form and convert it into dataflow graphs [32]. These dataflow graphs are directed graphs of nodes where each node represents a HyperOp. Each HyperOp comprises multiple fine grained operations. In order to exploit parallelism that exists within a HyperOp (also due to storage limitation in a CE), each HyperOp is divided into several partitions (pHyperOp) and each pHyperOp is assigned a CE. RETARGET captures the computation to be performed by each pHyperOp in terms of compute metadata and the inter/intra HyperOp communication in terms of transport metadata.

Each of the CEs in the REDEFINE fabric comprises a local wait-match-unit and a general purpose arithmetic logic unit (ALU). The ALU is capable of performing basic arithmetic and logic operations in a static dataflow order. It is also possible to incorporate operations of higher granularity (such as field multiplications) in the ALU instruction set by introduction of custom function units (such as the polymorphic multiplier). The Custom Function Unit (CFU) can be invoked through software as hardware subroutine calls.

4.2.2 AES on REDEFINE

In this section we present an implementation of the AES algorithms on REDEFINE. The AES cipher comprises a number of transformation rounds that convert the input plaintext into the final output of cipher-text. Each round consists of several processing steps, including one that depends on the encryption key. A set of reverse rounds are applied to transform cipher-text back into the original plaintext using the same encryption key. The AES algorithms are block ciphers by nature. Each block comprises 16 bytes of data. The elements of the block i.e. each byte is treated as elements of $GF(2^8)$. Each round of AES processes one block of data at a time. Each round comprises the steps SubBytes, ShiftRows, MixColumns and AddRoundKey. The inputs and outputs of each of these steps are usually arranged as 4×4 matrices called the state matrix. The basic operations involved in each of the steps are discussed below.

The SubBytes step

SubBytes involves computation of inverse over $GF(2^8)$ followed by an affine transformation. There are various techniques for computing inverses over a finite field. The inversion techniques can be broadly divided into three categories:

- Memory Lookup based techniques: Since there are only finite number of elements in a finite field and each element has a unique inverse, it is possible to precompute the inverses of all the elements in a finite field and store them in a table. At runtime, inversion of an element can be performed through a table lookup. Table 4.3 lists the size of memory required for calculation of inversion over a set of finite fields. It is evident from table 4.3 that a purely memory based technique is not feasible beyond a certain field size due to the exponential increase in memory size. Moreover there are certain vulnerability issues associated with memory based inversion techniques as presented in [15].

- Computation of inverses: There are a number of algorithms for computing inverses over finite fields. The algorithms include the extended Euclidean algorithm [33],

Table 4.3: Table showing memory requirements for inversion over different field sizes

Field Order	Size of memory
2^2	8 bits
2^4	64 bits
2^8	256 bytes
2^{16}	128 KB
2^{32}	16 GB
2^n	$\frac{n \times 2^n}{8}$ bytes

Itoh-Tsujii inversion algorithm [34] and the Wang's algorithm [35]. Both Itoh-Tsujii and the Wang's algorithm use the algebraic property of finite fields: a finite field can be spanned by repeated multiplications of a primitive element. Therefore inverse of an element can be found out by a series of multiplications and squaring. Itoh-Tsujii algorithm is particularly popular for inversion in normal basis due to the fact that squaring translates to shift operation in normal basis. It is to be noted that computation of inverse is slower than memory lookup based techniques.

- Hybrid techniques: Some inversion techniques involve both computation and memory lookup. Morii and Kasahara proposed such a technique in [36]. The technique is based on an algorithm which makes use of a field extension of order 2. In [37] C. Paar has shown that the algorithm proposed by Morii-Kashara is actually a special case of Itoh-Tsujii algorithm for composite fields.

Apart from inversion, SubBytes involves an affine transformation of the bytes of the state matrix. The affine transformarion is described in the equation 4.1.

$$b'_i = b_i \oplus b_{(i+4)\ mod\, 8} \oplus b_{(i+5)\ mod\, 8} \oplus b_{(i+6)\ mod\, 8} \oplus b_{(i+7)\ mod\, 8} \oplus c_i \quad (4.1)$$

for $0 \leq i < 8$, where b_i is the ith bit of the byte, and c_i is the ith bit of a byte c with the value (63) or (01100011) (the value of c is specified by the standard). Here and elsewhere, a prime on a variable (e.g., b') indicates that the variable is to be updated with the value on the right. In matrix form, the affine transformation element of the SubBytes can be expressed as shown in equation 4.2.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{4.2}$$

Let R_i denote the ith row of the constant 8×8 matrix in equation 4.2. Let us denote the byte to be transformed as $b = (b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$. Then equation 4.2 can be written as shown in equation 4.3.

$$b'_i = EvenParity(b \wedge R_i) \oplus c_i \tag{4.3}$$

Note that equation 4.3 can be realized through both software and hardware. A software implementation will require one logical AND, one even parity check and one logical XOR for each byte of the state matrix. Even parity check involves logical XOR of all the bits in the byte. In order to perform even parity check it is necessary to align all the bits. In a high level language such as C/C++ it can be done through logical shifting followed by logical AND with a constant as shown in equation 4.4.

$$EvenParity(a) = \bigoplus_{i=0}^{i=7} ((a >> i) \wedge 1) \tag{4.4}$$

The ShiftRows step

In this step each row of the state matrix is rotated to the left. Let us denote the rows of the input state matrix as R_0, R_1, R_2 and R_3. The operations involved in the ShiftRows

step is given by the equation 4.5.

$$R'_i = LeftRotate(R_i,\ 8i) \tag{4.5}$$

The MixColumns step

In this step the state matrix is multiplied with a constant matrix. Equation 4.6 describes the matrix multiplication operation.

$$\begin{bmatrix} s'_{00} & s'_{01} & s'_{02} & s'_{03} \\ s'_{10} & s'_{11} & s'_{12} & s'_{13} \\ s'_{20} & s'_{21} & s'_{22} & s'_{23} \\ s'_{30} & s'_{31} & s'_{32} & s'_{33} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix} \tag{4.6}$$

The elements of the matrices belong to $GF(2^8)$. Multiplications between the elements are performed modulo the irreducible polynomial $x^8+x^4+x^3+x+1$. Note that, the irreducible polynomial is specified by the standard. The field multiplications can be realized through software or hardware. There are different software algorithms for multiplication over finite fields (refer to section 2.2).

The AddRoundKey step

In this step a round key is added with the state matrix. As mentioned in section 2.1.1 addition over binary fields using polynomial basis translates to logical XOR operation, the AddRoundKey step involves logical XOR of the state matrix with the round key matrix (see equation 4.7). Note that, addition is a bit-independent operation. Thus the granularity of addition can be set to any convenient value.

Table 4.4: Performance of AES algorithms on *baseline* REDEFINE and a general purpose processor.

AES Algorithm	Execution Time on REDEFINE (in cycle counts)	Execution Time on a GPP (in cycle counts)
AES128 Encryption	11070	327901
AES128 Decryption	12677	481351
AES192 Encryption	13140	411653
AES192 Decryption	15239	578856
AES256 Encryption	15210	536213
AES256 Decryption	17801	672410

$$\begin{bmatrix} s'_{00} & s'_{01} & s'_{02} & s'_{03} \\ s'_{10} & s'_{11} & s'_{12} & s'_{13} \\ s'_{20} & s'_{21} & s'_{22} & s'_{23} \\ s'_{30} & s'_{31} & s'_{32} & s'_{33} \end{bmatrix} = \begin{bmatrix} k_{00} & k_{01} & k_{02} & k_{03} \\ k_{10} & k_{11} & k_{12} & k_{13} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{30} & k_{31} & k_{32} & k_{33} \end{bmatrix} + \begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix} \tag{4.7}$$

The implementations of these steps will be discussed in more details in the next section.

Performance of AES without the Polymorphic Multiplier

We implemented the AES on a *baseline* version of REDEFINE where each CE houses a general purpose ALU, without any special assist for field operations. The throughput of this implementation was used as a benchmark to compare the performance of AES with the polymorphic multiplier as a CFU. The various algorithms of the standard were described in C and compiled by RETARGET. The executable was run on a system-C based cycle-accurate simulator of REDEFINE to evaluate the throughput. Table 4.4 lists the total execution time of each of the algorithms in terms of cycle counts. We also computed the cycle count of an equivalent implementation on an Intel Core 2 Duo processor using the GNU C Compiler. The performance of the general purpose processor is also provided in table 4.4.

Performance of AES with the Polymorphic Multiplier

The performance of AES algorithms is heavily influenced by performance of the underlying field arithmetic operations. Therefore, in order to improve the performance of AES, it is necessary to speedup the field arithmetic operations. SubBytes and MixColumns are the most strongly field operations dependent steps and therefore are natural candidates for acceleration.

Performance of SubBytes depends upon the performance of inversion and the affine transform. Therefore it necessary to realize these two functions efficiently in order to speed up SubBytes.In order to prevent cache based side channel attacks, we perform inversion, through computation. We employed the Itoh-Tsujii algorithm for computing inverse for two reasons. Firstly it is faster than the Wang's algorithm. Secondly, it involves field multiplications that can be efficiently performed using a hardware accelerator like the polymorphic multiplier. Unlike the Itoh-Tsujii algorithm, extended Euclidean algorithm uses only basic arithmetic and logical operation in a recursive way. Enhancing performance of extended Euclidean algorithm will require specialized coarse-grain hardware unit. The absence of such units in REDEFINE makes Itoh-Tsujii algorithm a better choice for inversion over smaller fields (i.e. for fields where multiplication can be performed atomically). Note that the state matrix contains sixteen data entities each of width one byte. Each of the bytes may be processed in parallel, since there is no dependency between the bytes. Also note that REDEFINE is a 32-bit machine. Thus any hardware assist for SubBytes in REDEFINE should be capable of handling four bytes at a time to utilize the existing resources optimally. A 32-bit incarnation of the polymorphic multiplier that can support four parallel threads of 8×8 multiplication is suitable for this purpose. Moreover the polymorphic multiplier is capable computing even parity of a number through rotation (refer to section 3.4.1). Thus we chose the polymorphic multiplier as a CFU for performance enhancement of SubBytes on REDEFINE. Since there is no data dependency between the byte-wise operations, any combination of four bytes can be chosen to process in parallel. In order to minimize the overhead of rearranging rows and columns of the state matrices at the inputs and outputs of the four major steps, we

decided to process one row of the state matrices at a time.

MixColumns involves multiplication of two matrices whose elements belong to $GF(2^8)$. As can be seen form equation 4.6, the MixColumns step operates on the columns of state matrix. In order to conform to our row-wise implementation of the AES cipher we rearranged the individual multiplications to convert the column operations into row operations and extract parallelism. We present the transformation of the column operations into row operations in the following equations. Let s'_{ij} be the element of the output state matrix corresponding to ith row and jth column. s'_{ij} is computed as the sum of the products of the elements of ith row of the constant matrix and jth column of the input state matrix (see equation 4.8).

$$s'_{ij} = \sum_{k=0}^{3}(c_{ik} \times s_{kj}) \tag{4.8}$$

One row of the output state matrix can be calculated using the following computations.

$$\begin{aligned}
s'_{i0} &= c_{i0}s_{00} + c_{i1}s_{10} + c_{i2}s_{20} + c_{i3}s_{30} \\
s'_{i1} &= c_{i0}s_{01} + c_{i1}s_{11} + c_{i2}s_{21} + c_{i3}s_{31} \\
s'_{i2} &= c_{i0}s_{02} + c_{i1}s_{12} + c_{i2}s_{22} + c_{i3}s_{32} \\
s'_{i3} &= c_{i0}s_{03} + c_{i1}s_{13} + c_{i2}s_{23} + c_{i3}s_{33}
\end{aligned} \tag{4.9}$$

From equation 4.9 it is observed that the individual multiplications are independent of each other. Let us introduce four *intermediate* variables $temp_0, temp_1, temp_2$ and $temp_3$. These variables are four-tuples as defined below in equation 4.10.

$$\begin{aligned}
temp_0 &= \{c_{i0}s_{00}, c_{i0}s_{01}, c_{i0}s_{02}, c_{i0}s_{03}\} \\
temp_1 &= \{c_{i1}s_{10}, c_{i1}s_{11}, c_{i1}s_{12}, c_{i1}s_{13}\} \\
temp_2 &= \{c_{i2}s_{20}, c_{i2}s_{21}, c_{i2}s_{22}, c_{i2}s_{23}\} \\
temp_3 &= \{c_{i3}s_{30}, c_{i3}s_{31}, c_{i3}s_{32}, c_{i3}s_{33}\}
\end{aligned} \tag{4.10}$$

Note that the intermediate variables are generated by one sub-word parallel (four $8 \times$

8) multiplication each. The corresponding row in the output state matrix can also be represented as the four-tuple $\{s'_{i0}, s'_{i1}, s'_{i2}, s'_{i3}\}$. The output row is given by the term-by-term addition of the intermediate variables as shown in equation 4.11.

$$\{s'_{i0}, s'_{i1}, s'_{i2}, s'_{i3}\} = temp_0 + temp_1 + temp_2 + temp_3 \tag{4.11}$$

The equivalent logical operations of equation 4.11 are three 32-bit logical XOR. Therefore the MixColumns operations translate to four sub-word parallel multiplications and three bit-wise XORs per row of the input state matrix.

ShiftRows involve logical rotations. Since there is no logical operator for rotation in C/C++ we used the polymorphic multiplier for rotation. In the ShiftRows step the rows of the input state matrix are treated as 32-bit numbers. A 32-bit instance of the polymorphic multiplier is capable of performing the required rotation operations (refer to section 3.4).

The performances of the various algorithms in the Advanced Encryption Standard are listed in the table 4.5. The fourth column in table 4.5 shows the reduction in cycle count of execution for the various algorithms. The improvement in performance is primarily attributed to two reasons. Firstly, the introduction of the polymorphic multiplier enabled *atomic* execution of field multiplications as opposed to a software approach where the multiplications are split into a number of basic arithmetic and logical operations. Secondly, the ability of the polymorphic multiplier to exploit parallelism existing in the applications, by executing in a sub-word parallel manner reduced the number of operations even further.

As mentioned before, inversion process in the SubBytes step of AES can be implemented as a memory lookup. Though, it is not advisable for platforms like REDEFINE, we implemented the memory lookup based inversion in presence of the polymorphic multiplier. Note that, in this implementation, the polymorphic multiplier acts as a CFU for the MixColumns and the ShiftRows steps only. The performance of the various AES algorithms in this implementation are shown in table 4.6. As expected the memory lookup

Table 4.5: Performance improvement of various AES algorithms on REDEFINE by using the polymorphic multiplier.

AES Algorithm	Cycle count in absence of the polymorphic multiplier	Cycle count with the polymorphic multiplier	% improvement
AES128 Encryption	11070	6043	45.4
AES128 Decryption	12677	6118	51.7
AES192 Encryption	13140	7239	44.9
AES192 Decryption	15239	7344	51.8
AES256 Encryption	15210	8435	44.5
AES256 Decryption	17801	8570	51.8

based implementation performs faster than the purely computation based implementation. The third and fourth column of table 4.6 shows the penalty in performance for taking a computation based approach over simple memory lookup. Since the memory based approach is not secure [15], computation of inverses is generally favored. From table 4.6 it is observed, that the penalty incurred due to computation of inverses is reduced by a great extent by the usage of the polymorphic multiplier.

However it should be noted that REDEFINE is an architecture that favors computation over communication. Memory transaction is not a *cheap* operation since it involves sending a memory request outside the fabric followed by the arrival of data (in case of load operations) and memory operation completion acknowledgement (in case of both load and store operations) [32]. This whole operation takes variable number of clock cycles depending on the origin of the request. Requests originating from the peripheral CEs are reach the load-store unit faster than requests originating from CEs deep within the fabric. Thus the performance penalty for computing inverse with respect to memory lookup based implementation will be more in traditional platforms where memory operations are significantly faster. Hence the need for a hardware assist for field operations will be more prominent in such traditional platforms.

In order to improve performance of memory-intensive kernels (such as the SubBytes step of AES algorithms) it may prove to be beneficial to introduce hierarchical memory architecture in REDEFINE. For example, analysis of the AES algorithms clearly demonstrates that the performance of SubBytes step can be immensely improved by making

Table 4.6: Performance penalty of memory lookup free implementation of AES algorithms.

AES Algorithm	Cycle count with memory	% penalty w.r.t. implementation without the polymorphic multiplier	% penalty w.r.t. implementation with the polymorphic multiplier
AES128 Encryption	4550	143.3	32.8
AES128 Decryption	4291	195.4	42.5
AES192 Encryption	5420	142.4	33.5
AES192 Decryption	5109	198.2	43.7
AES256 Encryption	6369	138.8	32.4
AES256 Decryption	5927	200.3	44.5

memory operations cheaper. However, to avoid the threats of memory based side channel attacks, it will be imperative to ensure a *secure* environment for memory transactions. One possible way of speeding up SubBytes step without losing security is to introduce local-memory subsystems inside each of the compute elements. The locality of such memory blocks will make operations such as table lookup inexpensive. Moreover, cache timing based attacks may be prevented by restricting access to local memories. In order to estimate the improvement in performance of AES128 encryption algorithm on REDEFINE, we simulated local memory transactions. The encryption of a 16 byte block of data took 2129 cycles.

The focus of this thesis was to introduce the polymorphic finite field multiplier as an accelerator for software solutions. We showed that the polymorphic multiplier can bring about 40-50% improvement in performance of the AES algorithms. We also discussed how the performance can be further improved by introducing local memory modules in compute elements. On an aside, several high performance implementations of AES algorithms appear in literature. They can be broadly divided into categories listed below.

- ASIC Solutions: Users with stringent performance-area-energy-security requirements usually prefer ASIC solutions. However, such precise requirements can only be met with a significant NRE cost associated with the lengthy and cumbersome hardware design procedure and the end result will offer almost no flexibility. Some notable works in this area can be found in [38], [39] and [40].

- FPGA Solutions: FPGA solutions offer a higher degree of flexibility when compared to ASICs, at the cost of performance. Though FPGA based designs also use RTL for design specification, the design cycle associated with FPGA solutions are usually shorter than ASIC solutions, . [41], [42] and [43] report some of the FPGA implementations of the AES algorithms.

  Kris Gaj and Pawel Chodowiec discussed the various aspects of designing hardware solutions for the AES algorithms in [44].

- Hybrid Solutions: Hybrid solutions such as the polymorphic AES implementation based on the MOLEN processor [45] try to combine the advantages of both the software and hardware implementations. Rivardo Chaves et al. have discussed the architecture of a polymorphic AES implementation in [46]. In this implementation, the main ciphering function, the more computational demanding component has been implemented in hardware in a Virtex II pro-20. The RTL description of this component enables ASIC like performance.

Clearly, the above approaches derive the implementation through RTL specification which is different from standard software solutions. Further, it is not possible to embed a hardware accelerator such as the polymorphic multiplier in a platform such as MOLEN.

4.3 Big Multiplications

Elliptic curve cryptographic algorithms and hyper-elliptic curve cryptography algorithms operate on finite fields of much higher order than the typical block ciphers like AES (refer to table 2.3). Constructing hardware multipliers for finite fields of such high order is infeasible, since they cannot be integrated with any existing processing platforms due to vast mismatch of granularity. Therefore software realization of such multiplications is inevitable. Software implementations of this multiplication involves multiplication of polynomials representing numbers in a finite field followed by reduction using an irreducible polynomial. In this section we present and analyze the performances of three such polynomial multiplication algorithms on REDEFINE.

4.3.1 Different algorithms

We chose three software algorithms for polynomial multiplication on REDEFINE. The algorithms include *right-to-left comb method*, the *left-to-right comb method* and the Karatsuba-Ofman algorithm. In the subsequent sections we briefly describe the algorithms and their implementation on REDEFINE.

Shift and Add technique based algorithms

The *right-to-left comb method* and the *left-to-right comb method* are two algorithms based on the shift and add technique of multiplication of polynomials. The primary difference of these algorithms from the textbook shift and add technique lies in their implementation of shift operations. As described in section 3.1, the shift and add technique involves successive logical left shifting of one of the two operands. In hardware, shift operations are inexpensive. Therefore the shift and add technique is suitable for hardware realization. However once the field order becomes high enough so that the size of the numbers exceeds the word-length of the processing platform, shift operations cannot be performed in a single clock cycle. Under such circumstances, the large number of vector shifts make the traditional shift and add technique less desirable for software implementations. The comb methods of multiplication reduce the number of shift operations involved in polynomial multiplication. We reproduce a brief description of the algorithms here from [12].

Let us consider the finite field $GF(2^m)$. Let $A(x)$ and $B(x)$ be two elements of this finite field. In software we can store $A(x)$ and $B(x)$ as t w-bit words where $t = \lceil \frac{m}{w} \rceil$: $A(x) = (A[t-1], A[t-2], \cdots, A[2], A[1], A[0])$ and $B(x) = (B[t-1], B[t-2], \cdots, B[2], B[1], B[0])$. The comb methods for multiplication are based on the observation that if $B(x).x^k$ has been computed for some $k \in [0, w-1]$ then $B(x).x^{w.j+k}$ can be computed by appending j zeros to the right of the vector representation of $B(x).x^k$.

Algorithm 1 considers the bits of the polynomial $A(x)$ from right to left while the algorithm 2 considers the bits of $A(x)$ from left to right.The following notation has been used: if $C(x) = (C[n], C[n-1], \cdots, C[1], C[0])$ is a vector then $C\{j\}$ is the truncated vector $(C[n], C[n-1], \cdots, C[j+1], C[j])$.

Input: Binary Polynomials A(x) and B(x)
Output: C(x) = A(x).B(x)
$C \leftarrow 0$
for $i \leftarrow 0$ **to** 31 **do**
    for $j \leftarrow 0$ **to** $t - 1$ **do**
        if *ith bit of A[j] is* 1 **then**
            add B to $C\{j\}$
        end
    end
    if $i \neq 31$ **then**
        $B \leftarrow B.x$
    end
end
Return(C)

Algorithm 1: Right to Left Comb method

Input: Binary Polynomials A(x) and B(x)
Output: C(x) = A(x).B(x)
$C \leftarrow 0$
for $i \leftarrow 31$ **to** 0 **do**
    for $j \leftarrow 0$ **to** $t - 1$ **do**
        if *ith bit of A[j] is* 1 **then**
            add B to $C\{j\}$
        end
    end
    if $i \neq 0$ **then**
        $C \leftarrow C.x$
    end
end
Return(C)

Algorithm 2: Left to Right Comb method

Table 4.7: Performance Comparison of various multiplication algorithms on REDEFINE.

Algorithm	Cycle count
Right to Left comb method	15160
Left to Right comb method	14399
Karatsuba-Ofman algorithm without the polymorphic multiplier	33200
Karatsuba-Ofman algorithm with the polymorphic multiplier	636

Karatsuba-Ofman algorithm

As discussed in section 3.3, the Karatsuba-Ofman algorithm for multiplication employs a divide and conquer technique for multiplication. In order to multiply to binary polynomials $A(x)$ and $B(x)$ of degree at most $m-1$, the polynomials are split into two term polynomials recursively (refer to section 3.3). Note that, the Karatsuba-Ofman algorithm breaks bigger multiplications into smaller multiplication thereby extracting a significant amount of parallelism from an apparently sequential process. The breaking is performed recursively till the word size of the particular processing platform is reached. Then the smaller polynomial multiplications are realized using the available set of instructions and computation resources. Therefore the efficiency of this method depends on two aspects of the processing platform: the efficiency of the realization of smaller multiplications and the ability of exploiting parallelism.

Performance of the algorithms on REDEFINE

We implemented the algorithms for multiplication of polynomials over the finite field $GF(2^{128})$. Since the word-length of data in REDEFINE is 32, any element of the field $GF(2^{128})$ can be represented using four words. The performance results of these three algorithms is listed in table 4.7.

In absence of a hardware assist for polynomial multiplication, the Karatsuba-Ofman algorithm takes more time to execute than both the comb methods because of the greater number of operations involved. This is attributed to the fact that in absence of any special assist for polynomial multiplication, the smaller multiplications are realized as

series of basic arithmetic and logical operations. Note that the time taken for execution is highly data dependent for these implementations due to the presence of a number of data-dependent branches. In order to estimate the average performance of these algorithms, we chose input numbers with equal number of zeroes and ones.

As shown in section 3.3 the polymorphic multiplier is capable of polynomial multiplication where reduction of the result is prohibited. Since, performance of the Karatsuba-Ofman algorithm is strongly dependent on performance of the underlying smaller multiplications, we introduced the polymorphic multiplier as a CFU for polynomial multiplications. The performance of this implementation is shown last row of table 4.7. The significant improvement in performance is attributed to the reduction in number of operations caused by *atomic* execution of polynomial multiplications. Moreover, the execution time is independent of data due to absence of any data-dependent branch in the application.

4.4 Summary of the chapter

In this chapter we discussed and analyzed some experimental results. We presented the synthesis results of the polymorphic multiplier. We showed that the polymorphic multiplier occupies 41% less area compared to a collection of canonical multiplier with equivalent functionality. Clearly, this will lead to better resource utilization and thus reduce energy loss due to leakage power dissipation.

We also presented a case study on the Advanced Encryption Standard. We used REDEFINE as the target architecture for this set of experiments. We presented the performances of three different implementations of the algorithms in the standard on REDEFINE. We showed that by introducing the polymorphic multiplier as a custom function unit for a set of operations the performance of AES is improved on REDEFINE by about 40-50%. We also implemented a memory based less secure version of AES on REDEFINE to evaluate the performance penalty of using a purely computation based implementation. We presented a brief analysis of the results by discussing a few artefacts of the REDEFINE architecture.

Finally we presented implementations of a set of software algorithms for multiplications over large finite fields. In this context we presented a significant improvement in performance of the Karatsuba-Ofman algorithm for multiplication on REDEFINE by using the polymorphic multiplier for the underlying polynomial multiplications.

Chapter 5

Conclusions and Future Work

In this chapter we draw the final conclusions from the work presented in this thesis and also discuss about some possible directions of future work.

5.1 Conclusions

In this thesis we motivated the design of a polymorphic multiplier for binary fields though a study of different algorithms in the cryptography and error correction domains. We derived a general description of the architecture of the polymorphic multiplier through mathematical formulation. Since we arrive at the design through a completely general mathematical formulation, the architecture is functionally correct by construction. During the course of the derivation we identified certain scopes for optimization. Instead of developing a behavioral description, which may not lead to the most optimal hardware manifestation, we performed the optimizations at the level of mathematical abstraction. Moreover the generality of the mathematical foundation of the design, makes the architecture scalable.

Our aim was to provide hardware support for multiplications over binary fields of various orders through a common solution, to save area. This objective led us to design a multiplier for fields of the form $GF(2^{2^r})$, since such fields easily comply with word

lengths of commonly used processing platforms. We also provided the support for sub-word parallel multiplications for sub-fields in order to maximize resource utilization and exploit parallelism wherever possible.

We proved that the polymorphic multiplier can operate on binary fields of arbitrary order within its operating range in a sub-word parallel manner. This, however requires some preprocessing of the input operands and some post processing of the output. We also showed that the polymorphic multiplier is capable of polynomial multiplications, where reduction of results is prohibited. The polymorphic multiplier is also capable of performing certain logical operations like shift , rotate and even parity check.

We synthesized a hardware instance of the polymorphic multiplier and compare with a collection of canonical multipliers. The polymorphic multiplier occupies about 41% less area compared to the collection of the dedicated multipliers. However the operating frequency of the polymorphic multiplier is about 21% less than the slowest multiplier in the collection of dedicated multipliers. This reduction in performance is compensated by the flexibility and area savings offered by the polymorphic multiplier. Note that, during synthesis we did not pipeline the architecture. Since the multiplier design is very regular and structural , it is amenable for a pipelined design, where the number of pipeline stages can be decided in the context of the specific computation platform.

We conducted some experiments to analyze performance of the Advanced Encryption Standard on REDEFINE. We observed that, performance of the algorithms in the standard are improved by a factor of about 2x by introducing the polymorphic multiplier as a hardware assist for field operations. We also evaluated the performance of three multiplication algorithms for multiplication over large finite fields. We observed a 23x improvement in performance of the well known Karatsuba-Ofman algorithm. These observations make a case for introducing hardware assists for field operations on different computing platforms. Moreover, the coexistence of all these algorithms requires integrated solutions like the polymorphic multiplier.

5.2 Future Work

In the past power and performance have defined the computing requirements for different solutions. However, as the world rushes headlong into the future, a third dimension is becoming increasingly important: protection. As the world of *connected-computing* grows, security is fast becoming one of the major focus areas for future processor designs. With the explosion in the number of hand-held mobile devices, the security of communication among the consumers has become a serious issue. In the near future, even consumer electronic devices will be connected to the Internet. Along with the more traditional variants of such devices (i.e. the personal computers and data-centers) the number of connected devices will grow exponentially. Providing real time security at low cost for all such devices is becoming a serious technological challenge. Software or hardware solutions alone cannot handle such high demand of computation at low power and high speed. Therefore it is necessary to develop security solutions with the needs of future in mind. The polymorphic multiplier presented in this thesis is an enabler of such future platforms.

In order to make *wiser* decisions about choice of algorithms and architectures for such future platforms it is necessary to perform extensive analysis of different aspects the security domain. This will require an extensive power-performance-area-security study of various algorithms and architectures for different kinds of applications. Algebraic properties of finite fields are very attractive to cryptographers. Therefore, they will continue to be the integral parts of core computations of various security kernels in the foreseeable future. Thus, the need for efficient realization of field arithmetic kernels will remain important. In this context it may be worthwhile to investigate certain mathematical aspects of the problems. For example usage of certain basis of representation for certain application may seem more beneficial. Such investigations may potentially lead to a set of solutions for different applications. Integrating them together into a system without compromising on power, performance, area and security will be another challenge.

Acronyms

GSM Global System for Mobile Communications 1

CDMA Code Division Multiple Access 1

IMSI International Mobile Subscriber Identity 2

UMTS Universal Mobile Telecommunications System 2

USIM Universal Subscriber Identity Module 2

AKA Authentication and Key Arrangement vi

AuC Authentication Center 3

VLR Visitor Location Register 3

CAVE Cellular Authentication and Voice Encryption 4

CMEA Cellular Message Encryption Algorithm 4

AES Advanced Encryption Standard 3

UIM User Identity Module 4

ECC Elliptic Curve Cryptography vi

ECDHA Elliptic Curve Diffie-Hellman Algorithm 6

ECDSA Elliptic Curve Digital Signature Algorithm 6

CRC Cyclic Redundancy Check 7

ARQ Automatic Repeat Request 7

FEC Forward Error Correction 7

ASIC Application Specific Integrated Circuit 8

GF Galois Field 10

HECC Hyper-Elliptic Curve Cryptography 14

IGF Interleaved Galois Field Multiplier . . . 16

MGF Modular Galois Field Multiplier . . . 16

KTK Kitsos-Theodoridis-Koufopavlou Multiplier . . . 18

PF Paar-Fleischmann Multiplier . . . 18

SIMD Single Instruction Multiple Data . . . 19

CE Compute Element . . . 49

CFU Custom Function Unit . . . 49

Bibliography

[1] J. R. Rao, P. Rohatgi, S. Tinguely, and H. Scherzer, "Partitioning attacks: Or how to rapidly clone some GSM cards," in *Proceedings of the IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society, Technical Committee on Security and Privacy. Oakland, CA: IEEE Computer Society Press, May 2002, pp. 31–41. [Online]. Available: http://researchweb.watson.ibm.com/intsec/gsm.ps

[2] "3G Security; Specification of the MILENAGE algorithm set: An example algorithm set for the 3GPP authentication and key generation functions ," http://ftp.3gpp.org/specs/html-info/35206.htm.

[3] W. Millan and P. Gauravaram, "Cryptanalysis of the cellular authentication and voice encryption algorithm," *IEICE Electronic Express*, vol. 1, no. 15, pp. 453–459, 2004. [Online]. Available: http://joi.jlc.jst.go.jp/JST.JSTAGE/elex/1.453

[4] Wagner, Schneier, and Kelsey, "Cryptanalysis of the cellular message encryption algorithm," in *CRYPTO: Proceedings of Crypto*, 1997.

[5] D. Wagner, L. Simpson, E. Dawson, J. Kelsey, W. Millan, and B. Schneier, "Cryptanalysis of ORYX," in *5th International Selected Areas in Cryptography Workshop*, 1998, pp. 296–305.

[6] G. Rose and G. Koien, "Access security in cdma2000, including a comparison with umts access security," *Wireless Communications, IEEE*, vol. 11, no. 1, pp. 19–25, feb 2004.

[7] R. Koetter and A. Vardy, “Algebraic soft-decision decoding of reed-solomon codes,” *IEEE Transactions on Information Theory*, vol. 49, no. 11, pp. 2809–2825, 2003.

[8] M. Woh, S. Seo, H. Lee, Y. Lin, S. A. Mahlke, T. N. Mudge, C. Chakrabarti, and K. Flautner, “The next generation challenge for software definedradio,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation, 7th International Workshop, SAMOS 2007, Samos, Greece, July 16-19, 2007, Proceedings*, ser. Lecture Notes in Computer Science, S. Hämäläinen, Ed., vol. 4599, 2007.

[9] M. Alle, K. Varadarajan, A. Fell, N. Joseph, C. R. Reddy, S. Das, P. Biswas, J. Chetia, S. K. Nandy, and R. Narayan, “REDEFINE: Runtime Reconfigurable Polymorphic ASIC,” *ACM Transactions on Embedded Systems, Special Issue on Configuring Algorithms, Processes and Architecture*, 2008.

[10] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes.* North-Holland Mathematical Library, June 1988. [Online]. Available: http://citeseer.ist.psu.edu/context/11456/0

[11] J. Guajardo, S. Kumar, C. Paar, and J. Pelzl, “Efficient software-implementation of finite fields with applications to cryptography,” *Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications*, vol. 93, no. 1, pp. 3–32, 2006. [Online]. Available: http://dx.doi.org/10.1007/s10440-006-9046-1

[12] J. López and R. Dahab, “High-speed software multiplication in f_{2m},” in *Progress in Cryptology INDOCRYPT 2000*, ser. Lecture Notes in Computer Science, B. Roy and E. Okamoto, Eds. Springer Berlin / Heidelberg, 2000, vol. 1977, pp. 93–102, 10.1007/3-540-44495-5_18. [Online]. Available: http://dx.doi.org/10.1007/3-540-44495-5_18

[13] A. Karatsuba and Y. Ofman, “Multiplication of multidigit numbers on automata,” in *Soviet Physics-Doklady*, vol. 7, 1963, pp. 595–596.

[14] Hankerson, Hernandez, and Menezes, "Software implementation of elliptic curve cryptography over binary fields," in *CHES: International Workshop on Cryptographic Hardware and Embedded Systems, CHES, LNCS*, 2000.

[15] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *J. Cryptol.*, vol. 23, no. 2, pp. 37–71, 2010.

[16] G. Ahlquist, B. Nelson, and M. Rice, "Optimal finite field multipliers for FPGAs," in *Field-Programmable Logic and Applications. Proceedings of the 9th International Workshop, FPL '99*, ser. Lecture Notes in Computer Science, P. Lysaght, J. Irvine, and R. Hartenstein, Eds., vol. 1673. Glasgow, UK: Springer-Verlag, Aug./Sep. 1999, pp. 51–60.

[17] H. Hinkelmann, P. Zipf, J. Li, G. Liu, and M. Glesner, "On the design of reconfigurable multipliers for integer and galois field multiplication," *Microprocess. Microsyst.*, vol. 33, no. 1, pp. 2–12, 2009.

[18] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1983.

[19] E. D. Mastrovito, "VLSI designs for multiplication over finite fields GF(2^m)," in *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 6th International Conference, (AAECC-6)*. Berlin - Heidelberg - New York: Springer, Jul. 1989, pp. 297–309.

[20] ——, "VLSI architectures for computations in Galois fields," Ph.D. thesis, Linköping University, Linköping, Sweden, 1991.

[21] A. Halbutogullari and Çetin Kaya Koç, "Mastrovito multiplier for general irreducible polynomials," *IEEE Trans. Computers*, vol. 49, no. 5, pp. 503–518, 2000. [Online]. Available: http://www.computer.org:80/tc/tc2000/t0503abs.htm

[22] M. A. Hasan and V. K. Bhargava, "Bit-serial systolic divider and multiplier for finite fields GF(2^m)," *IEEE Trans. Computers*, vol. 41, no. 8, pp. 972–980, 1992.

[23] C. Paar and M. Rosner, "Comparison of arithmetic architectures for reed-solomon decoders in reconfigurable hardware," in *FCCM*. IEEE Computer Society, 1997, pp. 219–225. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/FPGA.1997.624622

[24] P. Kitsos, G. Theodoridis, and O. Koufopavlou, "An efficient reconfigurable multiplier architecture for galois field $GF(2^m)$," *Microelectronics Journal*, vol. 34, no. 10, pp. 975–980, 2003.

[25] C. Paar, P. Fleischmann, and P. Soria-Rodriguez, "Fast arithmetic for public-key algorithms in galois fields with composite exponents," *IEEE Trans. Computers*, vol. 48, no. 10, pp. 1025–1034, 1999.

[26] S. Krithivasan and M. J. Schulte, "Multiplier architectures for media processing," in *Proceedings of the Thirty-Seventh Asilomar Conference on Signals, Systems & Computers: November 9–12, 2003, Pacific Grove, California*, M. B. Matthews, Ed. pub-IEEE:adr: IEEE Computer Society Press, 2003, pp. 2193–2197.

[27] W.-M. Lim and M. Benaissa, "Design space exploration of a hardware-software co-designed gf(2m) galois field processor for forward error correction and cryptography," in *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis.* New York, NY, USA: ACM, 2003, pp. 53–58.

[28] S. Roy, "A sub-word-parallel galois field multiply-accumulate unit for digital signal processors," in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, may 2005, pp. 1626–1629 Vol. 2.

[29] J. Daemen and V. Rijmen, "The design of rijndael," in *Springer*, 2001, pp. 221–227.

[30] A. Fell, M. Alle, K. Varadarajan, P. Biswas, S. Das, J. Chetia, S. K. Nandy, and R. Narayan, "Streaming fft on redefine-v2: an application-architecture design space exploration," in *CASES '09: Proceedings of the 2009 international conference on*

Compilers, architecture, and synthesis for embedded systems. New York, NY, USA: ACM, 2009, pp. 127–136.

[31] A. Fell, P. Biswas, J. Chetia, S. K. Nandy, and R. Narayan, "Generic Routing Rules and a Scalable Access Enhancement for the Network on Chip RECONNECT," in *22nd IEEE International SOC Conference*, Sep. 2009.

[32] M. Alle, K. Varadarajan, , A. Fell, S. K. Nandy, and R. Narayan, "Compiling Techniques for Coarse Grained Runtime Reconfigurable Architectures," in *ARC'09: Proceedings of the 5th IEEE International Workshop on Applied Reconfigurable Computing*, jul 2008.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, Sep. 2001.

[34] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases," *Information and Computation*, vol. 78, no. 3, pp. 171–177, Sep. 1988.

[35] Wang, Truong, Shao, Deutsch, Omura, and Reed, "VLSI architectures for computing multiplications and inverses in GF(2^m)," *IEEETC: IEEE Transactions on Computers*, vol. 34, 1985.

[36] M. Morii and M. Kasahara, "Efficient construction of gate circuit for computing multiplicative inverses over $GF(2^m)$," *Transactions of the Institute of electronics, information and communication engineers*, vol. 72, pp. 37–42, 1989.

[37] C. Paar, "Some remarks on efficient inversion in finite fields," Oct. 09 1995. [Online]. Available: http://citeseer.ist.psu.edu/47539.html;http://ece.wpi.edu/Research/crypt/publications/./documents/paper.ps

[38] Norbert Pramstaller and Elisabeth Oswald and Stefan Mangard and Frank K. Gürkaynak and Simon Häne, "A Masked AES ASIC Implementation."

[39] Wolkerstorfer, Oswald, and Lamberger, "An ASIC implementation of the AES SBoxes," in *CTRSA: CT-RSA, The Cryptographers' Track at RSA Conference, LNCS*, 2002.

[40] Panu Hämäläinen, Timo Alho and Marko Hännikäinen and Timo D. Hämäläinen, "Design and implementation of low-area and low-power AES encryption hardware core," in *DSD*. IEEE Computer Society, 2006, pp. 577–583. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/DSD.2006.40

[41] Rodriguez-Henriquez, F. and Saqib, N.A. and Diaz-Perez, A., "4.2 Gbit/s single-chip FPGA implementation of AES algorithm," *Electronics Letters*, vol. 39, no. 15, pp. 1115–1116, july 2003.

[42] S. Qu, G. Shou, Y. Hu, Z. Guo, and Z. Qian, "High throughput, pipelined implementation of aes on fpga," *Information Engineering and Electronic Commerce, International Symposium on*, pp. 542–545, 2009.

[43] S. Drimer, T. Güneysu, and C. Paar, "Dsps, brams, and a pinch of logic: Extended recipes for aes on fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, pp. 3:1–3:27, January 2010. [Online]. Available: http://doi.acm.org/10.1145/1661438.1661441

[44] K. Gaj and P. Chodowiec, "FPGA and ASIC implementations of AES," in *Cryptographic Engineering*, Çetin Kaya Koç, Ed. Springer US, 2009, pp. 235–294. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-71817-0_10

[45] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1363–1375, 2004. [Online]. Available: http://csdl.computer.org/comp/trans/tc/2004/11/t1363abs.htm

[46] R. Chaves and L. Sousa, "Polymorphic AES encryption implementation," 2008. [Online]. Available: http://www.scientificcommons.org/43435487