New Hardware Algorithms and Designs for Montgomery Modular Inverse Computation in Galois Fields GF(p) and GF(2^n)

by

Adnan Abdul-Aziz Gutub

PhD Thesis

Electrical and Computer Engineering
Oregon State University
Corvallis, Oregon, USA

June 11, 2002

AN ABSTRACT OF THE THESIS OF

Adnan Abdul-Aziz Gutub for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on June 11, 2002.

Title: New Hardware Algorithms and Designs for Montgomery Modular Inverse Computation in Galois Fields GF(p) and GF(2^n).

Abstract approved: ______________________________

Alexandre Ferreira Tenca

The computation of the inverse of a number in finite fields, namely Galois Fields GF(p) or GF(2^n), is one of the most complex arithmetic operations in cryptographic applications. In this work, we investigate the GF(p) inversion and present several phases in the design of efficient hardware implementations to compute the Montgomery modular inverse. We suggest a new correction phase for a previously proposed almost Montgomery inverse algorithm to calculate the inversion in hardware. It is also presented how to obtain a fast hardware algorithm to compute the inverse by multi-bit shifting method. The proposed designs have the hardware scalability feature, which means that the design can fit on constrained areas and still handle operands of any size. In order to have long-precision calculations, the module works on small precision words. The word-size, on which the module operates, can be selected based on the area and performance requirements. The upper limit on the operand precision is dictated only by the available memory to store the operands and internal results. The scalable module is in principle capable of performing infinite-precision Montgomery inverse computation of an integer, modulo a prime number.

We also propose a scalable and unified architecture for a Montgomery inverse hardware that operates in both GF(p) and GF(2^n) fields. We adjust and modify a GF(2^n) Montgomery inverse algorithm to benefit from multi-bit shifting hardware features making it very similar to the proposed best design of GF(p) inversion hardware.

We compare all scalable designs with fully parallel ones based on the same basic inversion algorithm. All scalable designs consumed less area and in general showed better performance than the fully parallel ones, which makes the scalable design a very efficient solution for computing the long precision Montgomery inverse.

New Hardware Algorithms and Designs for Montgomery Modular Inverse Computation in Galois Fields GF(p) and GF(2^n)

by

Adnan Abdul-Aziz Gutub

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 11, 2002
Commencement June 2003

Doctor of Philosophy thesis of Adnan Abdul-Aziz Gutub presented on June 11, 2002

APPROVED:

Major Professor, representing Electrical and Computer Engineering

Chair of the Department of Electrical and Computer Engineering

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Adnan Abdul-Aziz Gutub, Author

ACKNOWLEDGMENTS

I am grateful to my advisor Alexandre Ferreira Tenca for his reviews, help, guidance and encouragement in conducting this research and producing the papers that formed a basis for this thesis. I also thank Çetin Kaya Koç for his assistance in proposing my research topic, and his valuable directions toward this accomplishment. I also thank the other members of my graduate committee for their valuable input.

A number of people have given me invaluable technical help, recommendations and comments at various stages in the writing of these papers. In this regard I am thankful to the Information Security Laboratory researchers of Oregon State University. In particular, I acknowledge Roger Traylor, Erkay Savas, Tugrul Yanik, Budiyoso Kurniawan, Serdar Erdem, Don Heer, and Colin Van Dyke, for providing valuable support and feedback during different research phases.

I acknowledge financial support for my PhD education from KFUPM-Saudi Arabia and NSF under the CAREER grant CCR-0093434-"*Computer Arithmetic Algorithms and Scalable Hardware Designs for Cryptographic Applications*".

Finally, I owe a debt of gratitude to the rosy part in my life my great parents (Amnah Sait and Abdul-Aziz Kutub), my wonderful wife (Manal Fattani), and my lovely children (Muna, Alaa, and Omar), without them being patient, this achievement could not have been achieved.

Adnan Abdul-Aziz Gutub
Corvallis, Oregon, USA
June 11, 2002

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

NEW HARDWARE ALGORITHMS AND DESIGNS FOR MONTGOMERY MODULAR INVERSE COMPUTATION IN GALOIS FIELDS GF(p) AND GF(2^n)

1 INTRODUCTION

Information security nowadays is a very important subject [32,40,41]. Governments, commercial businesses, and individuals are all demanding secure information in electronic documents, which is becoming preferred over traditional documents (paper and microfilm, for example). Documents in electronic form require less storage space, its transfer is almost instantaneous, and it is accessible via simplified databases. The ability to make use of information more efficiently has resulted in a rapid increase in the value of information. Businesses in a number of commercial arenas recognize information as their most valuable asset [34].

However, information in electronic form faces potentially more damaging security threats. Unlike information printed on paper, information in electronic form can virtually be stolen from a remote location. It is much easier to alter and intercept electronic communication than its paper-based predecessors.

Information security is described as the set of measures taken to prevent unauthorized use of electronic data, whether this unauthorized use takes the form of disclosure, alteration, substitution, or destruction of the data. The requirements to securely maintain electronic information are classified as the following three services:

- *Confidentiality* - hiding data from unauthorized parties.
- *Integrity* - assurance that data is genuine.
- *Availability* - the system still functions efficiently after security provisions are in place.

Several measures have been considered to provide these services but no single measure can ensure complete security [32]. Of the various proposed measures, the use of cryptographic systems offers the highest level of security, together with maximum flexibility [40,41]. A cryptographic system transforms electronic data into a modified form. The owner of the information in modified form is now assured of its security features. Depending on the security services required, the assurance may be that the data cannot be altered without detection, or it may be that the data is unintelligible to all but authorized parties.

In the past, cryptographic systems have provided only confidentiality. Preparing a message for a secure, private transference involves the process of *encryption*. *Encryption* transforms data from user readable form, called *plaintext*, to an illegible translation, called *ciphertext*. The conversion of plaintext to ciphertext is controlled by an electronic key E. The key is simply a binary number, which determines the effect of the encryption function. The reverse process of retrieving the plaintext back from the ciphertext is called *decryption*, and is controlled by a related key D.

Depending on the encryption/decryption key, cryptographic systems can be classified into two main categories: secret key cryptosystems and public key cryptosystems. The secret key cryptosystems use one key (E=D) for both encryption and decryption, as illustrated in Figure 1.1. Since the keys are the same, two users wishing to communicate in confidence must agree and maintain a common secret key. Each entity must trust the other to keep the key as a secret.

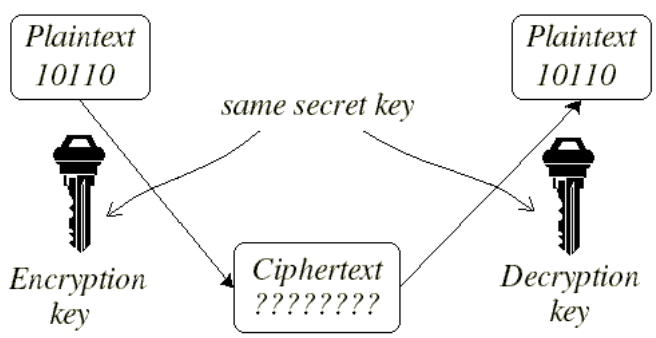

Figure 1.1 Secret key cryptosystem

Public key cryptosystems, however, use two different keys, one for encryption (E) and the other for decryption (D), where $D \neq E$. Public key cryptosystems were introduced in 1976 by Whitfield Diffie and Martin Hellman [5]. In a public-key cryptosystem, the abilities to perform encryption and decryption are separated. The encryption needs a public key (E) different but mathematically related to the decryption private key (D). Knowledge of the public key allows encryption of plaintext but does not allow decryption of the ciphertext. If somebody selects and publishes his public key, then everyone can use that public key to encrypt messages for that person. The private key is kept secret so that only the intended individual can decrypt the ciphertext. Figure 1.2 shows a public key cryptosystem methodology. One of the most promising public key cryptographic methods to be used is named the elliptic curve cryptography (ECC), which provides the best performance security of any public key cryptosystem known today [32,40,41]. ECC is based on the Discrete Logarithm problem over the points on an elliptic curve. In order to use ECC, an elliptic curve must be defined over a specific finite field. A finite field is a set of elements that have a finite order (number of elements). The order of a *Galois Field* (*GF*) is normally a prime number or a power of a prime number. The most popular finite fields used in ECC are *Galois Fields*, $GF(p)$ and $GF(2^n)$ [9-12,28,29].

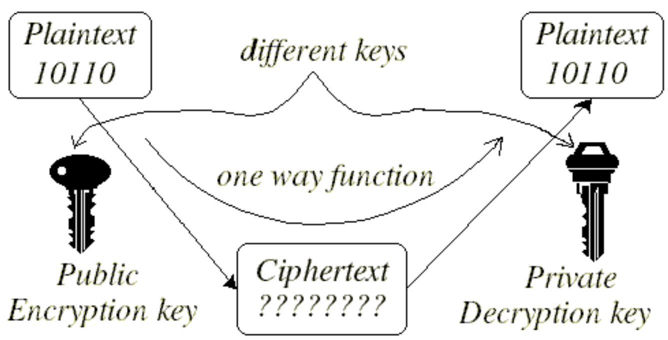

Figure 1.2 Public key cryptosystem

ECC is heavily based on modular multiplication, which involves division by the modulus in its computations. Division, however, is a very expensive operation [13,14]. This characteristic of modular operations made researchers seek out methods to reduce the division impact and make modulo multiplication less time consuming. In 1985, P. Montgomery proposed an algorithm to perform modular multiplication without trial division [15]. The algorithm replaces the complex division with simple divisions by two, which is effortlessly achieved in the binary representation of numbers (shifting the binary number one bit to the right). The penalty in using Montgomery's technique is paid with some extra computations to represent the multiplication operands into Montgomery domain and transform them back to integer domain [1,6,11,15]. The reader is referred to [15] for detailed information about Montgomery's method. A brief description of the Montgomery concept is provided in Chapter 3.

The ECC computation consists of different modular arithmetic operations where inversion is an essential one, with the slowest speed [1,2,9-12,20-25]. Since the use of Montgomery's method requires that the numbers are in Montgomery's domain, it is clear that having procedures dedicated to compute the modular inverse in the Montgomery domain [1,2] would be extremely beneficial. These dedicated modular inverse methods are named the Montgomery modular inverse algorithms.

Montgomery inverse (MonInv) computation can be performed in software or hardware for either GF(p) or GF(2^n). In this thesis, one of our goals is to design an efficient scalable MonInv hardware to operate in both finite fields GF(p) and GF(2^n). Scalability is the feature that allows the hardware to fit into restricted areas and operate with high clock frequency, which together are rarely possible for the fully parallel designs.

We start by considering the inverse computation in GF(p). It is well known that algorithms dedicated for GF(p) computation may be adjusted for GF(2^n). However, it is very difficult to modify a GF(2^n) algorithm for GF(p) [11,35]. The standard modular inverse over GF(p) can be defined by the following example. Assume a is an integer in the range [1,p-1]. Integer x is called the modular inverse, or modulo inverse, of integer a if-and-only-if: $ax \equiv 1 \pmod{p}$; where $x \in [1, p-1]$. It is normally represented as $x = a^{-1} \bmod p$ [1]. The MonInv algorithm suitable for our research is portrayed in [1]. The algorithm requires two main operations: an *almost Montgomery inverse* (AlmMonInv) and a correction phase (Montgomery product) operation. Our study modifies this algorithm for hardware and introduces a new faster correction phase. A similar GF(2^n) MonInv algorithm is also proposed, where both GF(p) and GF(2^n) MonInv algorithms are designed as a unified MonInv hardware for GF(p) and GF(2^n).

The motivation to focus our research on the design of inversion in hardware is explained in Section 1.1. Section 1.2 presents a brief review of the previous attempts to perform inversion in hardware. Section 1.3 details the thesis outline.

1.1 Motivation

Modular inverse arithmetic is a fundamental arithmetic operation in public-key cryptography. It is used in the Diffie-Hellman key exchange method [5], and to calculate private decryption key for RSA [4]. Modular inversion is considered an essential operation in the elliptic curve cryptography (ECC) [1,2,9-12,20-25]. This research is targeted mainly toward the ECC utilization because of its promise to replace several older cryptographic systems [9-12,20]. ECC arithmetic consists mainly in modular computations of addition, subtraction, multiplication, and inversion.

Inversion is well known to be the slowest operation among all other arithmetic operations in ECC [1,2,11,16-18]. Many researchers propose minimizing the use of modular inversion by adopting elliptic curves defined for projective coordinates, which substitute the inverse by several multiplication operations [9-12]. Projective coordinates are one of two coordinate systems used for the ECC arithmetic point operations; the other one is known as affine [11] coordinate system (detailed in Chapter 2). Inversion, in the projective coordinate systems, is required only once, to convert the projective coordinate points to affine coordinates at the end of the ECC point computation. However, if this one use of inversion takes too long, it will affect the performance of the whole ECC system.

To have a fast modular inverse calculation is one of the main reasons to do inversion in hardware instead of software [16-18]. If it is possible to compute the inverse faster than nine multiplication operations, then it is more efficient to use the affine coordinate system instead of going to the projective coordinate systems, as discussed in Chapter 2. Even if the speed to compute the inverse is not that good to justify the use of affine coordinates, the computation with hardware is still faster than software [6,16-18,20-25], which will provide better performance for the overall cryptographic system based on projective coordinates.

The other main reason to implement the modular inverse operation in hardware is security [32]. For cryptographic applications, it is more secure to have all the computations handled in hardware, inside an IC-chip for example, instead of mixing some computations performed in software with others processed in hardware. Software implementations are supported by operating systems, which can be interrupted and trespassed by intruders and this way compromise the application security. Such a security threat is not so easily attained in hardware implementations [32].

1.2 Previous Work

Modular inversion is often performed based on modifying, or directly using, the extended Euclidean algorithm [11]. Several inversion hardware attempts are described in the literature [16-18,20-25,35]. Most of the research [17,18,20-25] proposed hardware models specifically designed for inversion in GF(2^n). Among them, the large architectures in [23,24] suffer from the problem of signal broadcasting. The signal broadcasting problem should be avoided when implementing high-speed VLSI circuits [17,39].

In contrast, the designs in [17,18,21,22,25] are based on the concept of parallel systolic array structures. A systolic array [39] consists of a set of interconnected logic cells, each capable of performing the same simple operation. They work together and synchronously to perform a task. Within a systolic array or tree, information and data flow between the cells in a pipelined regular mode. Although systolic arrays are well suited for VLSI implementations due to its modular identical cells and simple and regular communications and control structures, it normally consumes a huge amount of hardware area in order to gain computation speed [17]. The area and time complexities of the designs in [17,18,21,22,25] are listed in Table 1.1.

Hardware Design	*Area Complexity*	*Time Complexity*
Guo & Wang [17]	$O(n^2)$	$O(1)$
Choudhury & Barua [18]	$O(n)$	$O(n^2)$
Guo & Wang [21]	$O(n^2)$	$O(1)$
Fenn, Benaissa & Tayler [22]	$O(n^2)$	$O(n)$
Kovac, Ranganathan & Varanasi [25]	$O(n.\ n^2)$	$O(1)$

Table 1.1 Area and time complexity of different inversion hardware designs

Hasan in [20] proposed a hardware design for the GF(2^n) inversion algorithm in a non-systolic structure consuming smaller hardware area and still operating with reasonable speed. The large operands are divided into words. The hardware performs the computation on a word-by-word serial manner, instead of computing all the words in parallel. GF(2^n) arithmetic requires simpler modular operations than GF(p) [11] because the carry propagation delays in addition or subtractions are completely eliminated. Since we focus first on GF(p) and then extend it to GF(2^n), the designs proposed for GF(2^n) in [17,18,21-25] were not beneficial to this work. As said before, extending a design done for GF(2^n) to GF(p) is not practical [11].

Naofumi Takagi in [16], proposed an inverse algorithm for hardware with a redundant binary representation. Redundant representation is used to reduce the carry propagation delay problem in additions. However, the redundant binary representation requires more area, because redundant digits require more bits to be coded and stored. Furthermore, redundant representation needs data transformation, which results in considerable extra cost.

Goodman and Chandrakasan in [35] presented a general cryptographic processor that computes modular algorithms coded in microcode, which can be modified with minimal effort. The processor can perform inversion in both GF(p) and GF(2^n) finite fields. Its datapath is reconfigurable and parameterized for numbers ranging in size from 8 to 1024 bits, controlled by a shut-down unit. This unit is responsible for disabling unused portions of the data path in order to minimize any unnecessary power consumption. The processor hardware is designed carefully to be energy efficient and faster than software-based implementations. The main disadvantage of this processor is its huge area, with a core containing 880,000 devices.

1.3 Thesis Outline

In the following chapter (Chapter 2), more detail is given to the ECC theory, which is the main scope of our research. The ECC arithmetic operations over the two finite fields GF(p) and GF(2^n) are compared. Then, some ECC based cryptographic applications are presented to give a practical flavor to the ECC theory.

A suitable GF(p) Montgomery inverse algorithm for hardware implementation was proposed in [1]. It requires two types of different routines, almost Montgomery inverse and Montgomery product. We present the design of the almost Montgomery inverse routine in Chapter 3. Two implementations are described there. The first one is a fixed precision (fully parallel) hardware, which has some inherent problems such as large (impractical) area and very low clock frequency. To solve this problem, it is proposed to use a scalable hardware design that performs the same function operating with higher clock frequency. The scalable hardware is also a module that can handle operands of any size. Based on the hardware area and performance requirements, the word-size, on which the module operates, is selected. The upper limit on the operand precision is dictated only by the available memory to store the operands and internal results. The scalable module is in principle capable of performing infinite-precision Montgomery inverse computation of an integer, modulo a prime number. This scalable hardware is compared with the fixed precision design showing very attractive results.

The longest path of the hardware designs passes through adders and subtractors. Chapter 4 contains the analysis of the impact of faster adders and subtractors in the hardware. Experimental performance results for the designs (fixed precision and scalable) using carry-look-ahead adders instead of carry-ripple adders are presented.

In Chapter 5, we propose a complete GF(p) Montgomery inversion (MonInv) procedure (almost Montgomery inverse plus correction phase). We modify the original procedure presented in [1] by replacing the Montgomery product used in its correction phase by a new straightforward correction phase. The advantage of the new correction phase is that it is implemented with roughly the same scalable hardware of the almost Montgomery inverse algorithm described in Chapter 3. The concept of multiple-bit shifting is also introduced in the proposed MonInv design.

Chapter 6 proposes a scalable and unified architecture for a Montgomery inverse hardware that operates in both GF(p) and $GF(2^n)$ fields. We present a $GF(2^n)$ Montgomery inverse algorithm that accommodates multi-bit shifting making it very similar to the GF(p) algorithm of Chapter 5.

The conclusion chapter (Chapter 7) summarizes the results of this thesis work and provides some future work in this area.

2 ELLIPTIC CURVE CRYPTOGRAPHY

2.1 Introduction

In 1985 Niel Koblitz and Victor Miller proposed the Elliptic Curve Cryptosystem (ECC) [9-11,28-33], a method based on the Discrete Logarithm problem over the points on an elliptic curve (EC). Since that time, ECC has received considerable attention from mathematicians around the world, and no significant breakthroughs have been made in determining weaknesses in the algorithm [32,40,41]. Although critics are still skeptical as to the reliability of this method, several ECC encryption techniques have been developed recently. The fact that the problem appears so difficult to crack means that key sizes can be reduced in size considerably, even exponentially [29,33], especially when compared to the key size used by other cryptosystems. ECC became an alternative to RSA, one of the most popular public key methods. ECC offers the same level of security as RSA but with much smaller key size [29].

In order to use ECC, an elliptic curve must be defined over a specific finite field. The EC arithmetic can be optimized depending on the type of finite field. The most popular finite fields used in ECC are *Galois Fields*, *GF(p)* and $GF(2^n)$ [9,12,28,29]. The following section will give some background on the EC theory followed by a comparison between ECC arithmetic performed in GF(p) and $GF(2^n)$. Then, some ECC applications will be introduced to give an idea of how ECC can be used.

2.2 Elliptic Curve Theory

Elliptic curves are described by cubic equations, similar to those used in ellipsis calculations. The general form for an elliptic curve equation is:

$$y^2+axy+by=x^3+cx^2+dx+e.$$

There is also a single element named the *point at infinity* or the *zero point* denoted 'φ'. The point at infinity is computed as the sum of any three points on an EC that lie on a straight line. If a point on the EC is added to another point on the curve or to itself, some special addition rules are applied depending on the finite field being used and also on the type of coordinate system (affine or projective) its applied to.

As mentioned earlier, a finite field is a set of elements that have a finite order (number of elements). There are many ways of representing the elements of the finite field. Some representations may lead to more efficient implementations of the field arithmetic in hardware or in software. The EC arithmetic is more or less complex depending on the finite field where the EC is applied and in which coordinate system the computation is performed. GF(p) and GF(2^n), in affine and projective coordinates are considered in this research because they are the most used in ECC [9,11,28,29].

2.2.1 Elliptic Curves over Finite Field GF(p)

GF(p) is comprised of the set of integers: $\{0, 1, 2, \ldots\ldots, p-2, p-1\}$. In this field, the basic arithmetic operations are:

- Addition: $a+b= r;$ where: $r,a,b \in$ GF(p), r is the remainder of $(a+b)$ divided by p. This is known as addition modulo p.
- Multiplication: $a \,.\, b=s;$ where $a,b,s \in$ GF(p), s is the remainder of ab divided by p. This is known as multiplication modulo p.
- Squaring: $a^2 = a \,.\, a = s;$ where $a,s \in$ GF(p), s is the remainder of a^2 divided by p. Squaring can be assumed as multiplication modulo p.
- Inversion: Assume a is a non-zero element in GF(p), the inverse of a modulo p, denoted a^{-1}, is the unique integer $c \in$ GF(p), for which $a \,.\, c = 1$.

The EC arithmetic over GF(p) is the usual mod p arithmetic. The EC equation in GF(p) is: $y^2 = x^3 + ax + b$; where $p > 3$, $4a^3 + 27b^2 \neq 0$, and $x, y, a, b \in$ GF(p).

The special addition rules in this field are the following:

$$\varphi = -\varphi$$
$$(x, y) + \varphi = (x, y)$$
$$(x, y) + (x, -y) = \varphi$$

2.2.1.1 Affine Coordinates

The addition of two different points on the EC in affine coordinates is computed as:

$(x_1 , y_1) + (x_2 , y_2) = (x_3 , y_3)$; where $x_1 \neq x_2$
$\lambda = (y_2 - y_1)/(x_2 - x_1)$
$x_3 = \lambda^2 - x_1 - x_2$
$y_3 = \lambda(x_1 - x_3) - y_1$

The addition of a point to itself (doubling a point) is computed as:

$(x_1 , y_1) + (x_1 , y_1) = (x_3 , y_3)$; where $x_1 \neq 0$
$\lambda = (3(x_1)^2 + a) /(2y_1)$
$x_3 = \lambda^2 - 2x_1$
$y_3 = \lambda(x_1 - x_3) - y_1$

We assume in this work that the squaring calculation is equivalent to a multiplication. Thus, to add two different points in GF(p) we need: six additions, one inversion, and three multiplication operations. To double a point we require: four additions, one inversion, and four multiplications.

2.2.1.2 Projective Coordinates

The projective coordinates are used to almost eliminate the need for performing inversion [11,28]. For elliptic curve defined over $GF(p)$, the normal elliptic point (x,y) is projected to (X,Y,Z), where $x=X/Z^2$,and $y=Y/Z^3$ [11]. This transformation between affine and projective coordinates is performed only twice: at the beginning and at the end.
The point addition of $P+Q$ in projective coordinates is computed as:

$P = (X_1,Y_1,Z_1);\ Q = (X_2,Y_2,Z_2);\ P+Q = (X_3,Y_3,Z_3);$ *where* $P \neq \pm Q$
$(x, y) = (X/Z^2,Y/Z^3) \rightarrow (X,Y,Z)$
$\lambda_1 = X_1Z_2^2$
$\lambda_2 = X_2Z_1^2$
$\lambda_3 = \lambda_1 - \lambda_2$
$\lambda_4 = Y_1Z_2^3$
$\lambda_5 = Y_2Z_1^3$
$\lambda_6 = \lambda_4 - \lambda_5$
$\lambda_7 = \lambda_1 + \lambda_2$
$\lambda_8 = \lambda_4 + \lambda_5$
$Z_3 = Z_1Z_2\lambda_3$
$X_3 = \lambda_6^2 - \lambda_7\lambda_3^2$
$\lambda_9 = \lambda_7\lambda_3^2 - 2X_3$
$Y_3 = (\lambda_9\lambda_6 - \lambda_8\lambda_3^3)/2$

The doubling of a point ($P+P$) in projective coordinates is computed as:

$$
\begin{aligned}
&P = (X_1,Y_1,Z_1);\ P+P = (X_3,Y_3,Z_3)\\
&(x, y) = (X/Z^2, Y/Z^3) \Rightarrow (X,Y,Z)\\
&\lambda_1 = 3X_1^2 + aZ_1^4\\
&Z_3 = 2Y_1Z_1\\
&\lambda_2 = 4X_1 Y_1^2\\
&X_3 = \lambda_1^2 - 2\lambda_2\\
&\lambda_3 = 8Y_1^4\\
&\lambda_4 = \lambda_2 - X_3\\
&Y_3 = \lambda_1 \lambda_4 - X_3
\end{aligned}
$$

The squaring calculation in $GF(p)$ is very similar to the multiplication computation. The number of multiplication processes for adding two points is found to be *sixteen*, while the number of operations for doubling a point is found to be only *ten*.

2.2.2 Elliptic Curves over Finite Field GF(2^n)

GF(2^n) is called a *characteristic two* field or a binary finite field. It can be viewed as a vector space of dimension n over the field $GF(2)$ that consists of the elements 0 and 1. That is, there exist n elements $x_0 , x_1 , x_2 \ldots , x_{n-1}$ in GF(2^n) such that each element $x \in$ GF(2^n) can be uniquely written in the form: $x = a_0 x_0 + a_1x_1 \ldots + a_{n-1}x_{n-1}$ **;** where $a_i \in$ GF(2). Such a set $\{x_0 , x_1 , x_2 \ldots , x_{n-1}\}$ is called the *basis* of GF(2^n) over GF(2). Given such a basis, a field element x can be represented as the bit string ($a_0\ a_1 \ldots a_{n-1}$). Addition of field elements is performed by bit-wise XOR-ing their vector representations. The complexity of multiplication depends on the selected basis. There are many different basis of GF(2^n) over GF(2). Some basis lead to more efficient software or hardware implementations of the arithmetic in GF(2^n) than others. The most popular basis are the polynomial (or standard) and the normal basis.

The EC equation over GF(2^n) is: $y^2+xy = x^3+ax^2+b$; where $x, y, a, b \in$ GF(2^n) *and* $b\neq 0$.

The addition rules in this field are as the following:

$$
\begin{aligned}
&\varphi + \varphi = \varphi\\
&(x, y) + \varphi = (x, y)\\
&(x, y) + (x, x+y) = \varphi
\end{aligned}
$$

2.2.2.1 Affine Coordinates

The affine coordinates addition of two different points on the EC is computed as:

$$(x_1 , y_1) + (x_2 , y_2) = (x_3 , y_3) \text{ ; where } x_1 \neq x_2$$
$$\lambda = (y_2 + y_1)/(x_2 + x_1)$$
$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$$
$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

The affine coordinates addition of a point to itself (doubling a point) is computed as:

$$(x_1 , y_1) + (x_1 , y_1) = (x_3 , y_3) \text{ ; where } x_1 \neq 0$$
$$\lambda = x_1 + (y_1)/(x_1)$$
$$x_3 = \lambda^2 + \lambda + a$$
$$y_3 = (x_1)^2 + (\lambda + 1)\, x_3$$

To add two different points in GF(2^n) we need: nine additions, one inversion, one squaring, and two multiplication operations. To double a point we require: five additions, one inversion, two squarings, and two multiplications.

2.2.2.2 Projective Coordinates

Calculating the inverse is the most expensive operation. Designs replace the inversion by several multiplication operations by representing the elliptic curve points as projective coordinate points [11,28,30,32]. To almost eliminate the need for performing inversion in $GF(2^k)$, its coordinates (x,y) are to be projected to (X,Y,Z), where $x=X/Z^2$, and $y=Y/Z^3$. The elliptic curve equation in this system is: $Y^2 + XYZ = X^3 + a X^2Z^2 + b Z^6$ [11].

The point doubling of an elliptic curve point (P+P) in projective coordinates is computed as:

$$P = (X_1,Y_1,Z_1);\ P+P = (X_3,Y_3,Z_3)$$
$$Z_3 = X_1 Z_1^2$$
$$X_3 = (X_1 + bZ_1^2)^4$$
$$\lambda = Z_3 + X_1^2 + Y_1 Z_1$$
$$Y_3 = X_1^4 Z_3 + \lambda X_3$$

The point addition of two elliptic curve points (*P+Q*) is computed as:

$$P = (X_1,Y_1,Z_1);\ Q = (X_2,Y_2,Z_2);\ P+Q = (X_3,Y_3,Z_3);\ \text{where } P \neq \pm Q$$
$$(x, y) = (X/Z^2, Y/Z^3) \Rightarrow (X,Y,Z)$$
$$\lambda_1 = X_1 Z_2^2$$
$$\lambda_2 = X_2 Z_1^2$$
$$\lambda_3 = \lambda_1 + \lambda_2$$
$$\lambda_4 = Y_1 Z_2^3$$
$$\lambda_5 = Y_2 Z_1^3$$
$$\lambda_6 = \lambda_4 + \lambda_5$$
$$\lambda_7 = Z_1 \lambda_3$$
$$\lambda_8 = \lambda_6 X_2 + \lambda_7 Y_2$$
$$Z_3 = \lambda_7 Z_2$$
$$\lambda_9 = \lambda_6 + Z_3$$
$$X_3 = a Z_3^2 + \lambda_6 \lambda_9 + \lambda_3^3$$
$$Y_3 = \lambda_9 X_3 + \lambda_8 \lambda_7^2$$

The number of multiplication processes for adding two points is found to be *twenty*, while it is found to be *ten* for doubling a point.

2.2.3 Comparing Arithmetic Complexity on GF(p) and GF(2^n)

The number of operations for affine coordinates addition of two different points is found to be basically the same (in GF(p) and GF(2^n)), as shown in Table 2.1. The computation of 'λ' requires one inversion and one multiplication. Computing 'x_3' needs only one squaring. The value of 'y_3' is obtained with one multiplication. The number of operations in both fields is identical: one inversion, one squaring, and two multiplications, neglecting the addition, subtraction, and multiplication by small numbers [11,28].

Point operations	**Operations in GF(p)**	**Operations in GF(2^n)**
Point addition	*1 Inversion* *3 Multiplications*	*1 Inversion* *3 Multiplications*
Point doubling	*1 Inversion* *4 Multiplications*	*1 Inversion* *4 Multiplications*

Table 2.1 Comparing GF(p) and GF(2^n) number of lengthy point operations in affine coordinates

Point doubling on affine coordinates requires the computation of 'λ', which in GF(p) requires an inversion, a multiplication, and a squaring of x_1, while it needs an inversion and a multiplication in GF(2^n). Calculating 'x_3' in both fields requires the same operation of squaring lambda. Computing 'y_3' in GF(p) requires only one multiplication, while it needs a multiplication and a squaring in GF(2^n). The number of operations is found to be the same in both fields: one inversion, two squarings, and two multiplications [11,28].

Considering projective coordinates (Table 2.2), the number of multiplication processes for adding two points in GF(p) is found to be sixteen, while it is found to be twenty in GF(2^n). The number of multiplication calculations for doubling a point is found to be ten in both GF(p) and GF(2^n). This shows that GF(p) projective coordinates consumes four less number of multiplications than GF(2^n), however, comparison of the number of operations is not accurate because operations in GF(p) require different time than GF(2^n). Computations in GF(p) require lengthy time due to the delay of propagating the carry, which GF(2^n) does not have.

Point operations	**Operations in GF(p)**	**Operations in GF(2^n)**
Point addition	*16 Multiplications*	*20 Multiplications*
Point doubling	*10 Multiplications*	*10 Multiplications*

Table 2.2 Comparing GF(p) and GF(2^n) number of lengthy point operations in projective coordinates

2.2.4 The Elliptic Curve Discrete Logarithm Problem

The elliptic curve discrete logarithm problem is the fundamental mathematical property that supports elliptic curves cryptography. The problem can be clarified by considering E as an elliptic curve and P and Q as points on E; the discrete logarithm problem consists in *finding an integer k such that kP=Q*, if such an integer exists. Figuring the integer k is considered a very hard problem especially if the numbers are large [11]. On the other hand, if integer k and the EC point P are known, computing the other EC point Q is possible. The ECC algorithm used for calculating kP (scalar multiplication of k by P, which is equivalent to add P to itself k times) from P is the binary method, since it is known to be efficient and practical to implement [11,12,29,32]. The binary method algorithm is:

Define n: number of bits in k;
k_i: the i^{th} bit of k
Input: P (a point on the elliptic curve).
Output: $Q = kP$ (another point on the elliptic curve).
1. if $k_{n-1} = 1$, then $Q:=P$ else $Q:=0$;
2. for $i = n-2$ down to 0;
3. $Q := Q+Q$;
4. if $k_i = 1$ then $Q:= Q+P$;
5. return Q;

The binary method algorithm scans the binary representation of k and doubles the point Q n-times. Whenever, a particular bit of k is found to be one, an extra operation is needed. This extra operation is $Q+P$.

2.2.5 Comparing Arithmetic Complexity of Affine and Projective Coordinates

The basic ECC operation consists in computing the point kP from P. Lets use the binary algorithm presented in Section 2.2.4. The number of binary bits of integer k is n, which indicates the exact number of point doublings but not point additions. Assume that the bits of k are half ones and half zeros (an average estimation for comparison reasons). The EC arithmetic operations required are n point doublings and $n/2$ point additions. The total number of multiplications and inversions for both GF(p) and GF(2^n) are listed in Table 2.3. If the time to compute $1.5n$ inversions and $5.5n$ multiplications is less than $18n$ GF(p) multiplications or $20n$ GF(2^n) multiplications, then the system based affine coordinates is faster than the one based on projective coordinates. In other words, if one inversion is calculated in less than approximately nine multiplications, then the affine coordinate arithmetic is more appropriate to use than the projective coordinates.

In any case, even with projective coordinates, the inverse computation is still needed at the end of the computation to convert back to affine coordinates and cannot be eliminated completely [1,2,11], which justifies the need to research the alternatives for the design of inverse operation in hardware.

Finite Field	***Affine Coordinates Operations***	***Projective Coordinates operations***
	n doublings & *n/2* point additions	
GF(p)	*1.5n* Inversions & *5.5n* Multiplications	*18n* Multiplications
GF(2^n)		*20n* Multiplications

Table 2.3 Comparing the affine and projective coordinates

2.3 Elliptic Curve Cryptography Applications

As described earlier (Section 2.2.4), it is easy to calculate the point kP from P. However, it is very hard to determine the value of k knowing the two points: kP and P. This property leads to several algorithms for cryptography [29,32]. Some of these techniques will be introduced in the following subsections.

2.3.1 Elliptic Curve Diffie-Hellman Key Exchange Method

Secret key cryptosystems are normally used for encryption/decryption purposes, because it is faster than public key cryptosystems. Secret key cryptosystems require a secret key to be agreed upon before the cryptographic process starts. This agreement can be performed by the elliptic curve Diffie-Hellman [29] key exchange method as described by the following example.

Suppose that users A and B want to agree upon a secret key, which will be used for secret key cryptography. Users A and B choose a finite field, GF(p) for example, and an elliptic curve 'E' defined over this field. They also take a randomly chosen point $P=(x,y)$ lying on the elliptic curve E; we refer to P as the *base point* of the cryptosystem. The finite field, the elliptic curve, and the base point are all publicly known.

User A then randomly chooses a large integer $a \in$ GF(p) and keeps a secret. User A now computes the point aP which will lie on E. User B does the same: B randomly chooses a large integer b and computes bP. Both A and B make aP and bP publicly known. These are their public keys. The secret key that A and B use to encrypt messages sent to each other is abP, which both A and B can compute. User A knows a and bP, and so can find abP. Whereas, B knows b and aP, and so can find abP. The security of this system lies in the fact that a third party C, for example, knows only aP and bP, and unless C can solve the elliptic curve discrete logarithm problem there is no efficient way to break the encryption.

2.3.2 Elliptic Curve Encryption/Decryption

There are many ways to apply elliptic curves for encryption/decryption purposes [29]. A simple method is presented here to give the flavor of elliptic curve encryption/ decryption techniques. Assume working with GF(p) finite field and an elliptic curve E. The users randomly chose a *base point* P_{base}, lying on the elliptic curve E. The plaintext (the original message to be encrypted) is coded into an elliptic curve point P_m. Each user selects a private key 'n' and compute his public key $P = nP_{base}$. For example, user A's private key is n_A and his public key is $P_A = n_A P_{base}$.

For any one to encrypt and send the message point P_m to user A, he/she needs to choose a random integer k and generate the ciphertext $C_m = \{kP_{base}\ ,\ P_m + kP_A\ \}$. The ciphertext pair of points uses A's public key, where only user A can decrypt the plaintext using his private key.

To decrypt the ciphertext C_m, the first point in the pair of C_m, kP_{base}, is multiplied by A's private key to get the point: $n_A(kP_{base})$. Then this point is subtracted from the second point of C_m, the result will be the plaintext point P_m. The decryption operations are:

$$(P_m + kP_A) - n_A(kP_{base}) = P_m + k(n_A P_{base}) - n_A(kP_{base}) = P_m$$

2.3.3 Elliptic Curve Digital Signature Algorithm (ECDSA)

The process of ECDSA [29] is composed of three main steps: key generation, signature generation and signature verification. Each step is described as follows:

2.3.3.1 ECDSA key generation

Each user of the scheme does the following:

1. Select an elliptic curve E over a finite field, say GF(p). The number of points on E should be divisible by a large prime n.
2. Select a point $P = (x,y) \in$ GF(p) of order n.
3. Select a random integer d in the range $[1, n\text{-}1]$.
4. Compute $dP = Q$.
5. The user's public key is (Q, n, P, E); the user's private key is d.

2.3.3.2 ECDSA signature generation

To sign a message, m, the user does the following:

1. Select a random integer k in the range $[1, n\text{-}1]$.
2. Compute $kP=(x_1, y_1)$, and set $x_1 \bmod n = r$. If r is zero then go back to step 1. In other words, if $r=0$ then the signing equation will not involve the private key d.
3. Compute $k^{-1} \bmod n$.
4. Compute $s = k^{-1}(h(m) + dr) \bmod n$, where h is the hash value obtained from a suitable hash function.
5. If $s=0$ go to step 1. This because if s is zero then $s^{-1} \bmod n$ does not exist and we would not be able to verify the signature.
6. The signature to be included in the message m is the pair of integers (s, r).

2.3.3.3 ECDSA signature verification

To verify the signature (r, s) on the message m, the following should be done:

1. Obtain an authentic copy of the public key (Q, n, P, E).
2. Verify that r and s are integers in the range $[1, n\text{-}1]$.
3. Compute $w = s^{-1} \bmod n$ and $h(m)$.
4. Compute $u_1= h(m).w \bmod n$ and $u_2 = r.w \bmod n$.
5. Compute $u_2Q + u_1P = (x_0 , y_0)$ and $v = x_0 \bmod n$.
6. Accept the signature if and only if $r= v$.

In the above ECDSA algorithms, each user generates their own elliptic curve E, along with a base point P. However, this means that the public key sizes become quite large. If, instead, the users agree upon a fixed curve E and base point P, as system parameters, then each user needs only to define the point Q, which is then all that is included in the public key.

3 SCALABLE HARDWARE ARCHITECTURE FOR GF(p) ALMOST MONTGOMERY MODULAR INVERSE COMPUTATION

3.1 Introduction

Modular inversion is often performed by algorithms based on the Extended Euclidean algorithm [11]. Several inversion hardware attempts are described in the literature [16-18,20-25]. Most of the research [17,18,20-25] proposed hardware designs for inversion in Galois Fields GF(2^n). Several [17,18,21-25] are based upon parallelism of data flow in an array structure. The inversion in GF(2^n) is fast due to the elimination of the carry propagation delay in GF(2^n) calculations. However, the area used in parallel organizations are very large, of order O(n^2). Hasan in [20] proposed a design of the inversion algorithm that is smaller in area and still keeps a fair speed. He performed a word-by-word computation of the numbers instead of computing the whole numbers in parallel. Since we focus on GF(p), the designs proposed in [17,18, 21-25] have no direct link to this work. Takagi in [16], proposed an inverse algorithm for hardware with a redundant binary representation. Each number is represented by digits in the set {0,1,-1}. Redundant representation is used to reduce the carry propagation delay problem. However, it requires more area. It also needs data transformation that is usually expensive.

The Montgomery modular inverse algorithm suitable for our research is presented in [1]. The algorithm requires two main operations: a Montgomery product and an *almost Montgomery inverse* (AlmMonInv) operation. This Chapter is directed towards the implementation of the AlmMonInv. The Montgomery product is beyond the scope of this work and scalable Montgomery multipliers, such as the ones proposed in [6-8] can generate it.

Two AlmMonInv designs are presented in this Chapter, namely a fixed precision design and a scalable hardware design. The fixed precision design is fully parallel and processes full precision numbers at every clock cycle. The scalable hardware, however, divides the numbers in words where each word is processed in a clock cycle. We show that the scalable hardware is more appropriate for cryptographic applications.

3.2 Montgomery Inverse Algorithms

Two Montgomery modular inverse studies are found in the literature [1,2]. Both modify a technique proposed by Kaliski in 1995 [3], to make it more suitable and faster for cryptography using Montgomery's idea. Kaliski method, derived from the extended Euclidean algorithm [3], basically takes an integer a, and produces x, where $x{=}a^{-1}2^{m} \bmod p$. If a is an integer, the algorithm will calculate the inverse of a, but represented in Montgomery domain, as shown in Figure 3.1. When the number a is already in Montgomery domain, the application of Kaliski's routine will not give the needed Montgomery inverse result. Thus, some extra arithmetic operations are required to get it. Kaliski method is summarized next. It is followed by a brief explanation of two modifications to Kaliski's work to make it compute the Montgomery inverse and to make it faster.

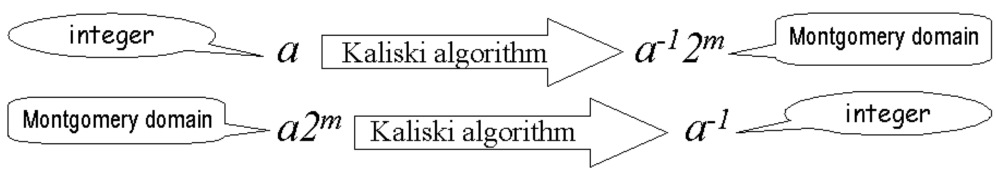

Figure 3.1 Types of input/output numbers for Kaliski algorithm

3.2.1 Kaliski Algorithm

Kaliski algorithm [1,3] is shown below and it is divided in two phases. Phase one, also called almost Montgomery inverse (AlmMonInv) in this work, takes the inputs a and p and give outputs r and k; where $r = a^{-1}2^{k} \bmod p$, and $n < k < 2n$ (n is the actual number of bits of the modulus p). Phase two takes the outputs of phase one as its inputs, and gives the final result of Kaliski algorithm: $x = a^{-1}2^{m} \bmod p$. Note that in both phases the integers: a and x are within the range [1,p-1]. Kaliski's two phases are outlined as follows:

Phase One: Almost Montgomery Inverse, AlmMonInv(a)

Input: a and p; where a is in the range [1,p-1].
Output: r and k; where $r = a^{-1}2^k \; mod \; p$, $n < k < 2n$, n = number of bits of p

1. $u = p;\ v = a;\ r = 0;$ and $s = 1$
2. $k = 0$
3. while ($v > 0$)
4. if u is even then $u = u/2;\ s = 2s$
5. else if v is even then $v = v/2;\ r = 2r$
6. else if $u > v$ then $u = (u - v)/2;\ r = r+s;\ s = 2s$
7. else $v = (v - u)/2;\ s = s+r;\ r = 2r$
8. $k = k + 1$
9. if $r \geq p$ then $r = r - p$
10. return $r = p - r$

Phase Two

Input: r,p,k & m; where r & k from phase one, & $m \geq n$ (m=Montgomery constant)
Output: x; where $x = a^{-1}2^m \; mod \; p$

11. for $i = 1$ to $k - m$ do
12. if r is even then $r = r/2$
13. else $r = (r + p)/2$
14. return $x = r$

3.2.2 Modifications to Kaliski Algorithm

T. Kobayashi and H. Morita in 1999 [2], proposed techniques for modular inversion to make it more than five times faster than the original Kaliski routine. They gained speed from the comparison of the values of u and v (step 6), they compare the most significant word only. Their way to achieve more speed consisted in combining the multiplication and the shifting operations. Long numbers were divided into words. They modified the AlmMonInv algorithm by performing several matrix multiplications, instead of the simple multiplications by two. The modification was targeted toward software implementations.

In July 2000, Savas and Koç [1] proposed to replace phase two of Kaliski's algorithm with a Montgomery multiplication, which resulted in a faster process. They also presented a complete Montgomery modular inverse algorithm by adding extra Montgomery multiplication operations. These extra multiplications are performed after the AlmMonInv. The Montgomery inverse computation algorithm in [1] is outlined below:

Montgomery Inverse Algorithm

Input: $a2^m$ *(mod p), p, n, m,* and $R^2 mod\ p$
Output: $x = a^{-1}2^m\ mod\ p$, where x is in the range [1,p-1].
1. (r,k) = AlmMonInv($a2^m$); where $r = a^{-1}2^{-m}2^k\ mod\ p$, and $n \leq k \leq m+n$
2. If $n \leq k \leq m$ then
2.1 r = MonPro(r,R^2) $= (a^{-1}2^{-m}2^k)(a^{2m})(2^{-m}) = a^{-1}\ 2^k\ \ mod\ p$
2.2 $k = k+m > m$
3. r = MonPro(r,R^2) $= (a^{-1}2^{-m}2^k)(2^{2m})(2^{-m}) = a^{-1}2^k\ \ mod\ p$
4. r = MonPro($r,\ 2^{2m-k}$) $= (a^{-1}2^k)(a^{2m-k})(2^{-m}) = a^{-1}2^m\ mod\ p$
5. Return $x = r$; where $x = a^{-1}2^m\ mod\ p$

The input parameters are the integers $aR\ mod\ p$ (residue representation of a)*, n, m* and p (the modulus, a prime number of size n-bits, $m \geq n$), and $R^2 mod\ p$ (a pre-computed integer based on the Montgomery radix, $R = 2^m$ [1]). The two main procedures used in the Montgomery inverse algorithm are the *Montgomery product* (MonPro) and the *almost Montgomery inverse* (AlmMonInv) [1], modeled in hardware as shown in Figure 3.2. Our contribution consists in the implementation of the almost Montgomery inverse procedure in hardware. The MonPro is beyond the scope of this work.

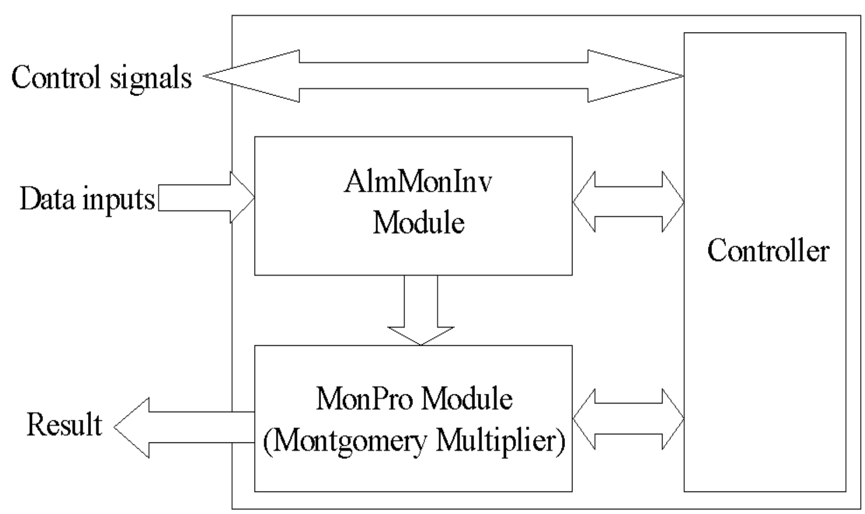

Figure 3.2 The complete Montgomery modular inverse hardware

3.3 The Fixed Precision Design

This section discuses a fixed precision hardware design. We present some hardware issues applied to the algorithm to compute the almost Montgomery inverse (AlmMonInv) subroutine, which is basically phase-one of Kaliski's algorithm.

3.3.1 Hardware Issues Applied to the Algorithm

When observed from hardware point-of-view, the AlmMonInv algorithm contains operations that are easily mapped to hardware features. For example, one-bit shifting the binary representation of number u to the right (ShiftR(u,1)) is equivalent to perform division by two, or one-bit shifting s to the left (ShiftL(s,1)) is equal to do multiplication by two. Checking for a number to be even or odd requires a test of the least significant bit (LSB). If it is found to be zero the number is even, otherwise the number is odd. The comparison of two numbers to see which one is bigger is performed after subtracting one from the other. If the subtraction result is positive (the borrow-bit is zero) the first number is bigger, or vice-versa. Such hardware mapping is shown in the hardware algorithm below:

The Fixed Precision Hardware AlmMonInv Algorithm (FHW-Alg)

Registers: u, v, r, s, and p (all five registers are to hold n-bits).
Input: $a \in [1, p-1]$, p = modulus.
Output: result $\in [1, p-1]$ and k; where result = $a^{-1}2^k \pmod p$ and $n<k<2n$

1. u = p; v = a; r = 0; s = 1; $k = 0$
2. if ($u_0 = 0$) then { u = ShiftR(u,1) ; s = ShiftL(s,1)}; go to step 7
3. if ($v_0 = 0$) then: { v = ShiftR(v,1) ; r = ShiftL(r,1)}; go to step 7
4. S1 = Subtract (u, v); S2 = Subtract (v, u); A1 = Add (r, s)
5. if ($S1_{borrow} = 0$) then {u = ShiftR(S1,1)); r = A1; s = ShiftL(s,1)}; go to step 7
6. s = A1; v = ShiftR(S2,1); r = ShiftL(r,1)
7. $k = k + 1$
8. if ($v \neq 0$) go to step 2
9. S1 = Subtract (p, r), S2 = Subtract (2p, r)
10. if ($S1_{borrow} = 0$) then {return result = S1}; else {return result = S2}

Consider step 6 of AlmMonInv, if $u>v$ then the subtraction (u-v) takes place, otherwise, the subtraction (v-u) is calculated. In the worst case, two subtraction operations are performed, because the comparison of u and v may be accomplished through subtraction of u and v anyway. These two subtractions can be done in parallel (two subtraction modules) as shown in step 4 of FHW-Alg. The same case applies to step 9 and step 10 of AlmMonInv, both subtractions may be performed in parallel.

All actual integers are represented by n-bit vectors, such as $u = (u_{n-1},u_{n-2},\ldots\ldots,u_2,u_1,u_0)$. The modulus is loaded into register u at step 1, then, register u is modified along with the algorithm. The modulus is essential at steps 9 and 10 (FHW-Alg) and for this reason, it is stored in a special register named p. The value of r cannot equal p except when a equals infinity. Thus, the result of AlmMonInv equals either $2p$-r if r is greater than p, or p-r when r is less than p, as described in step 10 of FHW-Alg.

3.3.2 The Fixed Precision Hardware Design

The fixed precision design is made up of a memory unit, a controller, a k-counter, and a data path (arithmetic unit). The block diagram for this hardware design is shown in Figure 3.3. All data buses are n_{max} bits wide (n_{max} is the maximum number of bits the hardware can handle). The memory unit is made of five registers u, v, r, s and p to hold n_{max} bits each. The memory unit sends out all its content and loads new ones at every clock cycle, except for register p that does not change during the computation. The data path (DP) takes the memory unit outputs and gives back the computed data to be stored through buses: *u_out*, *v_out*, *r_out*, and *s_out*. For example, in step 3 of FHW-Alg, only v and r are modified. However, the DP provides the data to all four buses. Buses *v_out* and *r_out* are found to be modifications of v and r, while *u_out* and *s_out* are just the same u and s fed back.

The DP performs the required computation depending on the LSBs of u and v, as clarified by FHW-Alg. It contains several multiplexers to route and shift the data buses to perform steps 2, 3, 5, 6 and 10, as shown in Figure 3.4. It consists of an adder and two subtractors to perform steps 4 and 9. The counter unit performs step 7 of FHW-Alg. All the components in the design are directed and synchronized by the controller.

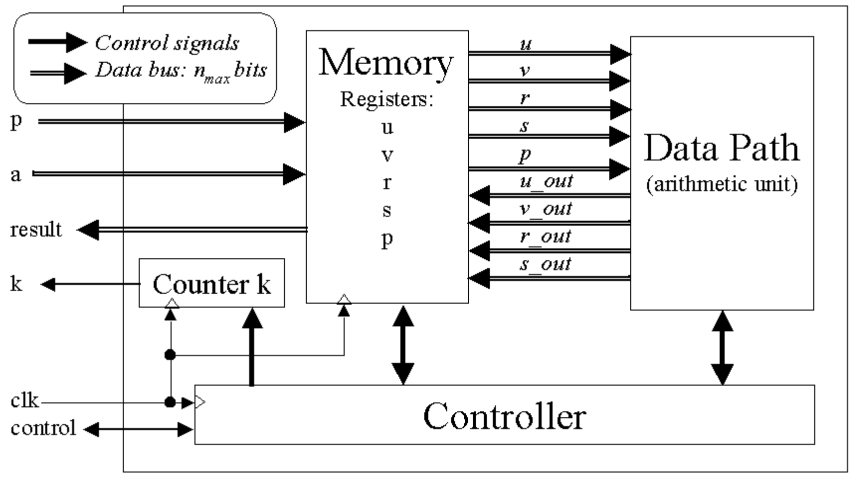

Figure 3.3 The block diagram of implementing the FHW-Alg

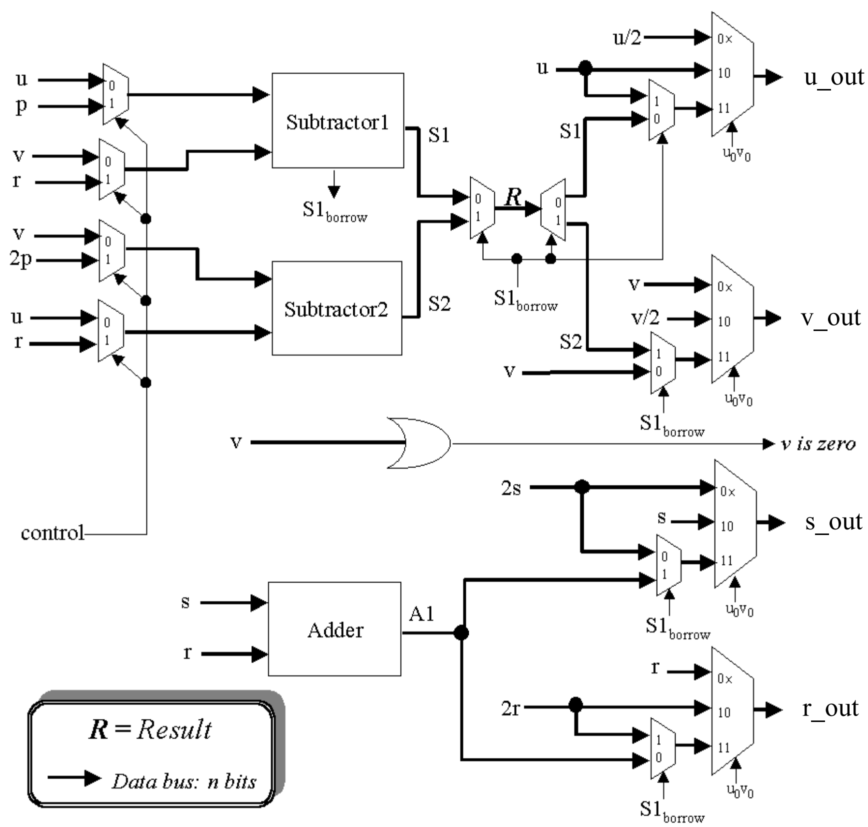

Figure 3.4 The fixed precision hardware data path

3.4 The Scalable Design

3.4.1 Why scalable design?

Application specific hardware architectures are usually designed to deal with a specific maximum number of bits, 512-bits for example. If this number of bits is to be increased, even by one, the complete hardware needs to be replaced. In addition to that, if the design is implemented for a large number of bits, the hardware is huge and its' longest path usually is impractical. It will cause the hardware to run at a very low clock frequency unless architectural changes are applied. These issues motivated the search for a scalable hardware similar to what is proposed by Tenca and Koç in their *Scalable Architecture for Montgomery Multiplication* [6].

The scalable architecture solves the previous problems with the following three hardware features. First, the design's longest path should be short and independent of the operands' length. Second, it is designed in such a way that it fits in restricted hardware regions (flexible area). Finally, it can handle the computation of numbers in a repetitive way up to a certain limit usually imposed by the size of the memory in the design. If the number of bits in the data exceeds the memory size, the memory unit is replaced while the scalable computing unit is not changed.

3.4.2 Scalable Hardware Issues Applied to the Algorithm

Differently from what happens in the fixed precision hardware design, the scalable hardware has multi-precision operators for addition, subtraction and comparison. The subtraction used for comparison ($u > v$), is performed on a word-by-word basis until all the actual data words are processed, then, the subtractor borrow out bit is used to decide on the result. Also, depending on the subtraction completion, variable r or s has to be shifted. All variables, u, v, r and s, need to remain as is until the subtraction processes complete, and the borrow out bit appears. Such a constraint forced the use of three more variables: x, y

and z; where $x= u\text{-}v$, $y= v\text{-}u$ and $z= r+s$. These variables are stored in extra registers as outlined in the scalable algorithm.

The Scalable Hardware Algorithm (SHW-Alg)

Registers: u, v, r, s, x, y, z & p (all eight registers are to hold n_{max} bits).
Input: $a \in [1, p\text{-}1]$, p = modulus.
Output: result $\in [1,p\text{-}1]$ & k; where result = $a^{-1}2^{k}$(mod p) & $n<k<2n$, ($n < n_{max}$)

1. u = p; v = a; r = 0; s = 1; x = 0; y = 0; z = 0; k = 0
2. if (u_0 = 0) then { u = ShiftR(u,1) ; s = ShiftL(s,1) }; go to step 7
3. if (v_0 = 0) then { v = ShiftR(v,1) ; r = ShiftL(r,1) }; go to step 7
4. x = Subtract (u, v); y = Subtract (v, u); z =Add (r, s)
5. if (x_{borrow}=0)then{u= ShiftR(x,1); r = z; s = ShiftL(s,1)}; goto step7
6. s = z; v = ShiftR(y,1); r = ShiftL(r,1)
7. k := k + 1
8. if (v $\neq$ 0) go to step 2
9. x = Subtract (p, r); y = Subtract (2p, r)
10. if (x_{borrow} = 0) then {return result = x}; else {return result = y}

All operations (addition, subtraction, and shifting) of the scalable hardware algorithm are multi-precision computations. In other words, the numbers are utilized in each operation on a word-by-word basis until the entire number is processed.

3.4.3 The Scalable Hardware Design

The scalable hardware design is built of two main parts, a memory unit and a computing unit. The memory unit is not scalable because it has a limited storage defined by the value n_{max}. The data values of a and p are first loaded in the memory unit. Then, the computing unit read/write (modify) the data using a word size of w bits. The computing unit is completely scalable. It is designed to handle w bits every clock cycle. The computing unit does not know the total number of bits, n_{max}, the memory is holding. It computes until the controller indicates that all operands words were processed. Note that the actual numbers used may have much less than n_{max} bits.

The block diagram for the scalable hardware is shown in Figure 3.5. The memory unit is connected to the computing unit components. The computing unit is made of four hardware blocks, add/subtract block, shifter block, data router block, and the controller. All these computing unit blocks are briefly clarified after describing the non-scalable memory unit. The memory unit contains a counter to compute k (step 7 of SHW-Alg) and eight first-in-first-out (FIFO) registers used to store the algorithm's variables. All registers, *u*, *v*, *r*, *s*, *x*, *y*, *z* and *p*, are limited to hold at most n_{max} bits. Each FIFO register has its own reset signal generated by the controller. They have counters ($n_{counter}$ bits each) to keep track of *n* (the number of bits actually used by the application).

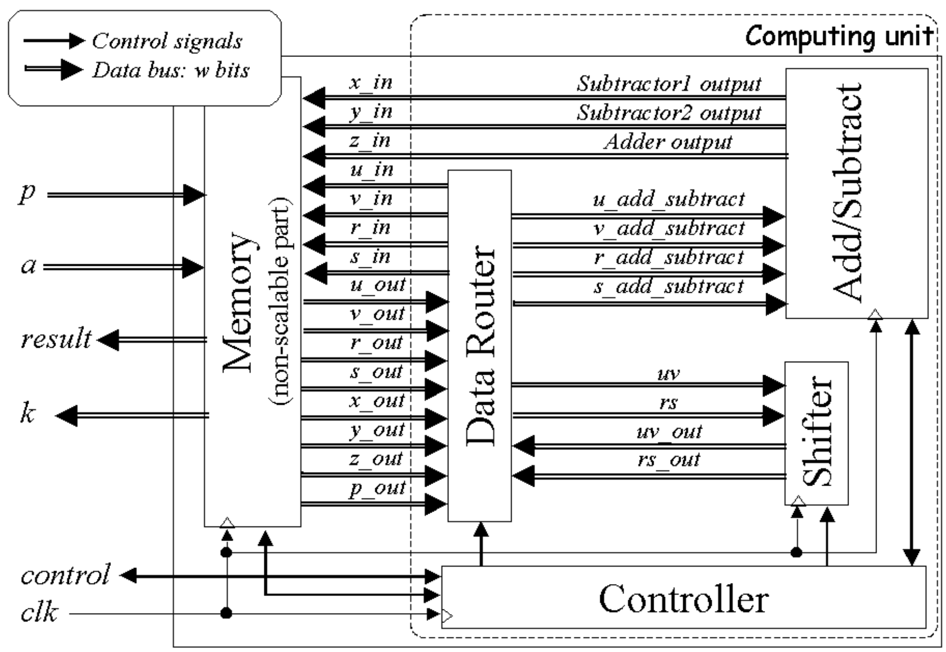

Figure 3.5 The scalable hardware overall block diagram

The add/subtract unit is built of an adder, two subtractors, four flip-flops, three multiplexers, a comparator, and logic gates, connected as shown in Figure 3.6. This unit performs one of two operations, either to calculate step 4 of SHW-Alg: $x = u\text{-}v$, $y = v\text{-}u$, and $z = r\text{+}s$, or to calculate step 9: $x = p\text{-}r$ and $y = 2p\text{-}r$. Three flip-flops are used to hold the intermediate carry-bit of the adder and borrow-bits of the two subtractors to implement the multi-precision operations. The fourth flip-flop is used to store a flag that keeps track of the comparison between *u* and *v*. This flag is used to perform step 8 of SHW-Alg. The first subtractor borrow out bit is connected to the controller through a signal that is useful only at the end of the each multi-precision addition/subtraction operation. It will affect the flow of the operation to choose either step 5 or step 6 of SHW-Alg. It is also essential in choosing the final result observed in step 10 of SHW-Alg.

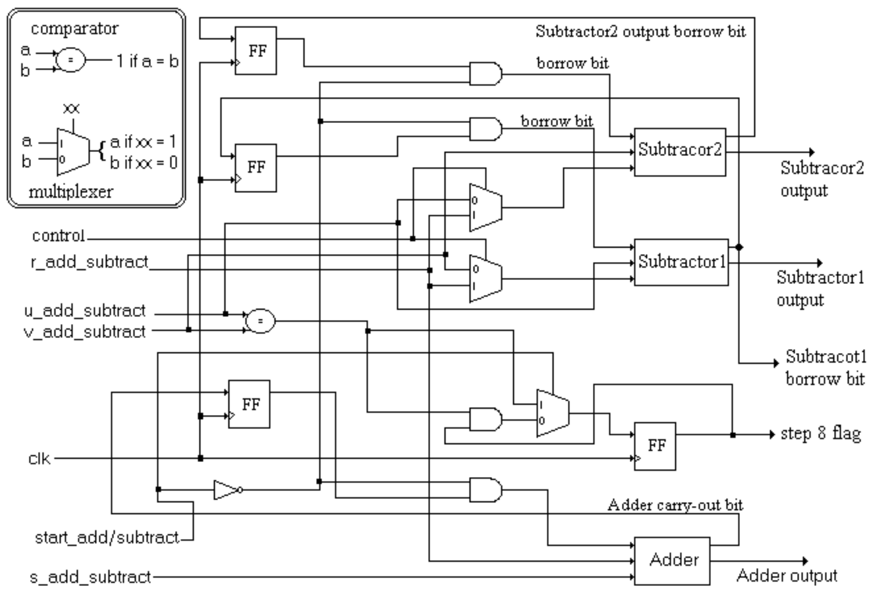

Figure 3.6 The scalable add/subtract unit

The shifter is made of two registers with special mapping of some data bits, as shown in Figure 3.7. Two types of shifting are needed in the hardware algorithm, shifting an operand (*u* or *v*) through the *uv* bus one bit to the right, and shifting another operand (*r* or *s*) through the *rs* bus one bit to the left. The input buses *uv* and *rs* are *w* bits noted in figure 3.7 as vectors $uv_{[w-1:0]}$ and $rs_{[w-1:0]}$, respectively. Shifting *u* or *v* is performed through Register1, which is of size *w-1* bits. For each word, all the bits of *uv* are stored in Register1 except the least significant bit ($uv_{[0]}$), it is read out immediately as the most significant bit (MSB) of the output bus *uv_out* ($uv\_out_{[w-1]}$). The MSB of the output of Register2 (vector $rs\_out_{[w:0]}$, bit $rs\_out_{[w]}$) is mapped as least significant bit (LSB) of the input of Regester2, to perform the shifting to the left.

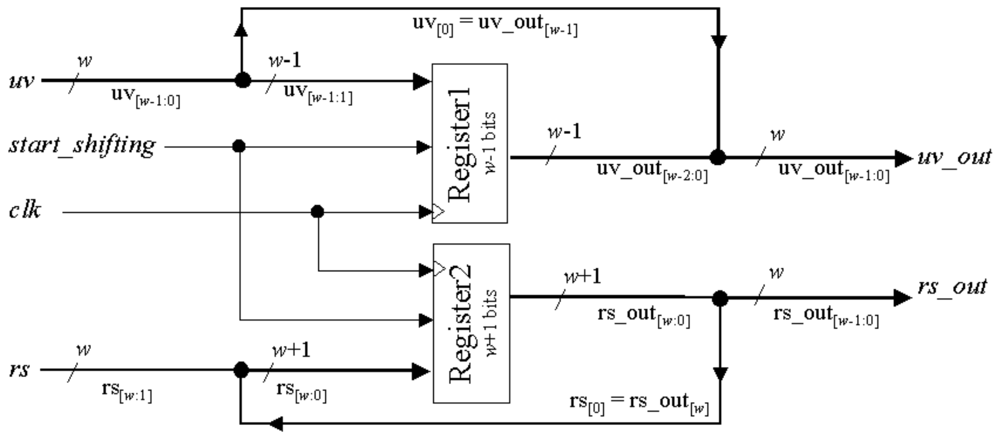

Figure 3.7 The scalable hardware shifter

The data router is made of ten multiplexers to connect the data going out of the memory unit to the inputs of the add/subtract unit or shifter. It also directs the shifted data values to go to their required locations in the memory unit. The possible configurations of the data router are shown in Figure 3.8.

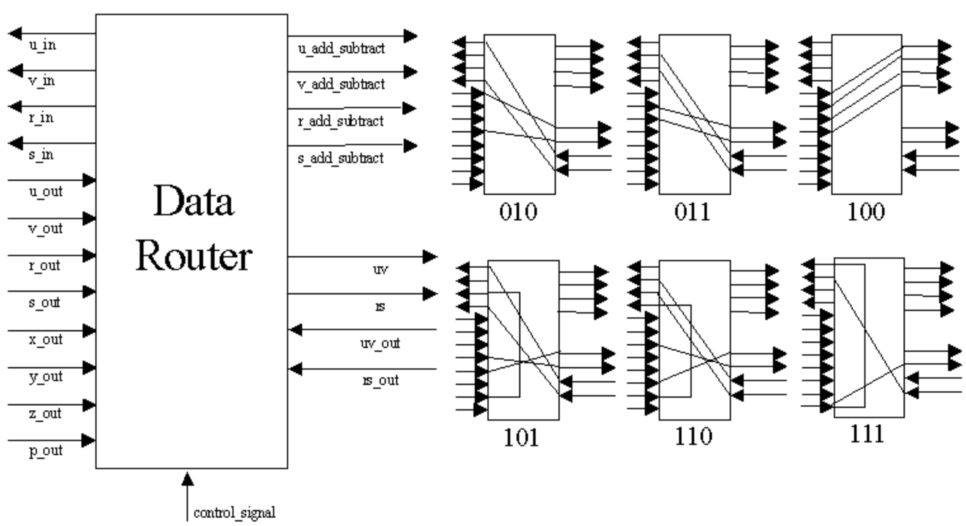

Figure 3.8 The data router configurations

The controller is the unit that coordinates the flow of data to guide the hardware computation. Its made of a state machine easily derived from SHW-Alg. The controller does not include counters to avoid any dependency on the number of bits that the system can handle. Such modules are left into the memory block.

3.5 Modeling and Analysis

Both designs were modeled and simulated in VHDL. The developed VHDL implementation of the scalable hardware has two main parameters, namely n_{max} and w. The fixed precision hardware, however, is parameterized by n_{max} only. Their area and speed are presented in this section. We didn't define a specific architecture for the adders and subtractors used in the design. Thus, the synthesis tool chooses the best option in terms of area from its library of standard cells. The impact of the use of different adders is described in Chapter 4.

3.5.1 Area Comparison

The exact area of any design depends on the technology and minimum feature size. For technology independence, we use the number of NOT-gates as an area measure [14]. A CAD tool from Mentor Graphics (Leonardo) was used. Leonardo takes the VHDL design code and provides a synthesized model with its area and longest path delay. The target technology is a $0.5\mu m$ CMOS defined by the 'AMI0.5 fast' library provided in the ASIC Design Kit (ADK) from the same Mentor Graphics Company [19]. It has to be mentioned here that the ADK is developed for educational purposes and cannot be thoroughly compared to technologies adopted for marketable ASICs. It however, provides a framework to contrast the scalable hardware with the fixed precision one.

The only problem we faced with our VHDL code is that Leonardo cannot synthesize the scalable design memory unit because of its behavioral parametrizable feature. So we present an area function to calculate the scalable design memory. In Table 3.1, the number of NOT gates comparable to several standard logic gates is listed [14]. Other logic modules, needed in the memory unit of the scalable design, are constructed from these basic gates. These modules and their area are listed in Table 3.2 [13,14]. The modules in Table 3.2 are related to n_{max} and $n_{counter}$ bits, which are related to each other by the formula:

$$n_{counter} = log_2(2n_{max}+1).$$

Thus, n_{max} is the only parameter controlling the area of the memory unit of the scalable design. The memory unit is made of eight registers that hold n_{max} bits and seventeen $n_{counter}$ bit counters, which totally corresponds to [$170\ log_2(2n_{max}+1) + 48n_{max}$] gates.

Standard gate type	Fan-in	Number of equivalent NOT gates
NOT	1	1
NAND, NOR	2	1
NAND, NOR	3	2
NAND	4	2
NOR	4,5	4
NAND	5	4
NAND, NOR	6	5
NAND, NOR	8	6
XOR, XNOR	2	3
XOR, XNOR	3	6
AND, OR	2,3	2
AND, OR	4	3

Table 3.1 Area of the standard logic gates

Memory Components	Building Logic	Area (number of NOT gates)
n_{max} bit register	n_{max} DFF	$6n_{max}$
$n_{counter}$ bit counter	$n_{counter}$ DFF + $n_{counter}$ AND + $n_{counter}$ OR	$6n_{counter}+2n_{counter}+2n_{counter}$ $=10\ n_{counter}$

Table 3.2 Area of the modules building the memory unit of the scalable design

Using the estimate of the memory block and Leonardo's results, it is possible to compare the sizes of the two designs in Figure 3.9. Observe that the fixed design has a better area if the maximum number of bits used (n_{max}) is less than 32. However, this is not used in cryptography, as small numbers are useless [11]. In fact, the advantage of the scalable hardware is found to make the size of the design as small as possible. For example, if n_{max}=*512* bits, the scalable hardware can be designed in less than half the area necessary for the fixed precision hardware.

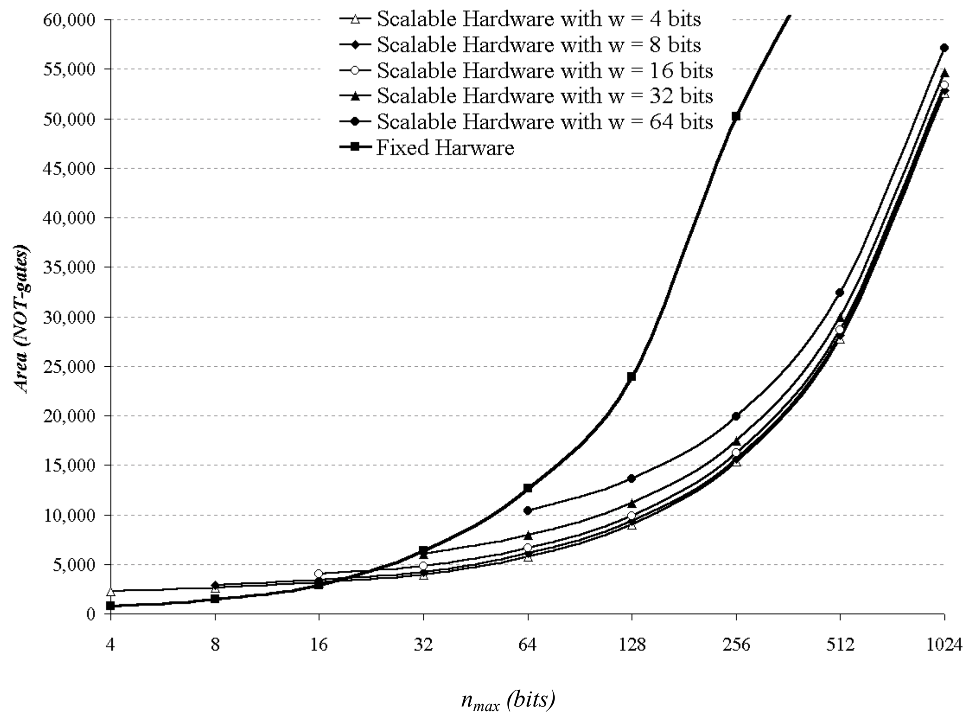

Figure 3.9 The area comparison

3.5.2 Speed Comparison

The total computation time is a product of the number of clock cycles the algorithm takes and the clock period of the VLSI implementation. This number of clock cycles depends completely on the data and its computation. For the fixed precision design, the number of clock cycles is *k+4*, where *k* is the number of iterations counted through the loop, step 2 to step 7 of FHW-Alg. The value of k (FHW-Alg) is within the range [$n,2n$] [1], which justify the use of its average of $3n/2$, for comparison purposes. This assumption makes the number of clock cycles required for the fixed precision design to complete a computation equal to

$$C_f = (3n/2) + 4.$$

The number of clock cycles in the scalable design is a function of three parameters: k, w and n. The number of cycles to compute any scalable addition and/or subtraction is calculated as $\lceil n/w \rceil$, which makes the actual number of clock cycles depend on the real data used and its size. However, after several experiments, we concluded that approximately half the time step 2 or 3 of SHW-Alg is needed and the other half step 4 is required. But the loop iteration time to execute step 2 or 3 is different than step 4. Step 4 needs extra cycles for the shifting operation after it. The number of cycles to perform each loop iteration (step 2 to step 7 of SHW-Alg) is calculated as

$$\text{CPLI} = [(\lceil n/w \rceil + 1)/2] + \lceil n/w \rceil + 3,$$

(CPLI stands for the clock cycles per loop iteration). The number of loop iterations of the algorithm is exactly equal to k. The overall number of cycles equals the CPLI $\times$ k (the number of loop iterations), plus the final operation of steps 9 and 10 (SHW-Alg). The total number of clock cycles of the scalable hardware equals to

$$C_s = 7+(7/2)k+[(4+(3/2)k)(\lceil n/w \rceil)],$$

which was verified by VHDL simulation. If k is approximated to its average of $3n/2$ (similar to the fixed precision design), the function of the clock cycles would be

$$C_s = 7+[(21/4)\ n] + [(4+(9/4)\ n)(\lceil n/w \rceil)].$$

The clock period of the hardware designs changes with the value of w in the scalable hardware, and changes with the value of n_{max} in the fixed precision hardware. This is because $w = n_{max}$ in the fixed precision hardware. Two speed comparison studies are carried out, one using the synthesize tool clock period for each design (technology dependent) and the other uses a technology independent estimation.

3.5.2.1 Technology dependent speed comparison

The real time clock period depends on the technology and the efficiency of the CAD tool used [13]. Table 3.3 lists the real time clock period for each design generated by Leonardo. We excluded the memory unit from all designs when synthesizing for the longest path delay assuming its effect will be the same for both scalable and fixed precision design, because the scalable design memory unit couldn't be synthesized (Section 3.5.1).

n_{max}	Scalable Hardware where w =					Fixed Precision Hardware
	4	8	16	32	64	
4	9.62	12.39	19.48	30.66	54.93	11.41
8	9.62	12.39	19.48	30.66	54.93	15.96
16	9.62	12.39	19.48	30.66	54.93	26.5
32	9.62	12.39	19.48	30.66	54.93	48
64	9.62	12.39	19.48	30.66	54.93	92
128	9.62	12.39	19.48	30.66	54.93	178
256	9.62	12.39	19.48	30.66	54.93	350
512	9.62	12.39	19.48	30.66	54.93	694
1024	9.62	12.39	19.48	30.66	54.93	1382

Table 3.3 Clock cycle period for all designs (nsec)

The scalable hardware can have several designs for each n_{max} depending on w. For example, Figure 3.10 shows the delay of five designs of the scalable hardware compared to the fixed precision hardware, all modeled for $n_{max} = 256$ bits. Observe how the actual data size (n) plays a big role on the speed of the designs. In other words, as n reduces for small w, the number of clock cycles decrease significantly, which considerably reduces the overall computing time of the scalable design. This is a major advantage of the scalable hardware over the fixed precision one.

The number of clock cycles of the fixed precision model depends on the actual size of the data used. However, its period always operate on n_{max} bits. For example, if we are using $n = 64$ bits, and the design is made for $n_{max} = 256$ bits, as of Figure 3.10, the fixed precision design will assume the operands are using all *256* bits by placing zeros for the unused bits. All n_{max} bits are processed into the computation causing the fixed precision design to have more delay than all different scalable ones. This fact is shown again in Figure 3.11 which presents the delay of the designs made for $n_{max} = 512$ bits.

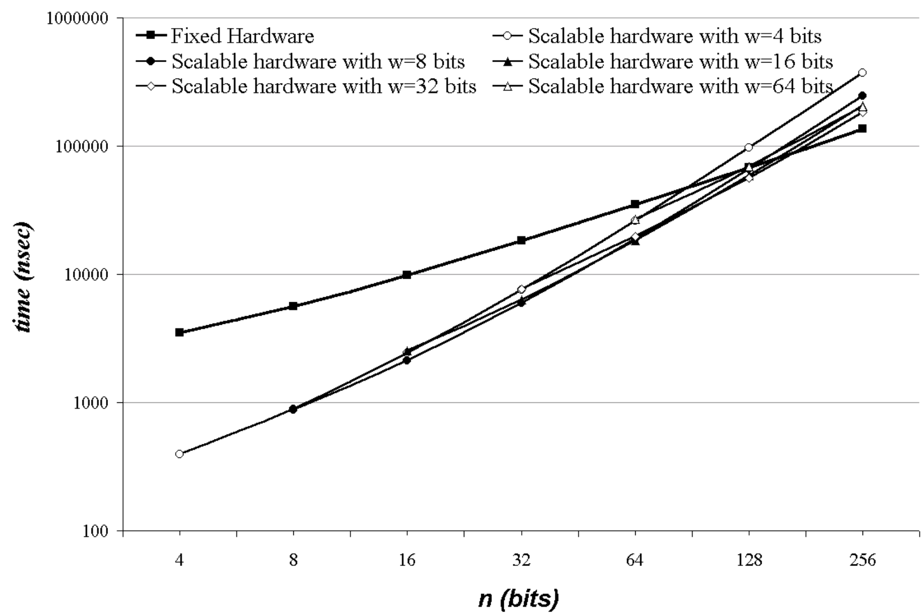

Figure 3.10 Delay comparison of designs with n_{max} = 256 bits

Another observation from Figures 3.10 and 3.11 is that the delay of all the scalable designs are better than the fixed precision one when $n \leq n_{max}/2$, except for w = 4 bits that is better when $n \leq 3n_{max}/8$. Suppose our design is target to handle 512 bits maximum, as the case of Figure 3.11, which is a practical number for future ECC applications [11]. The scalable designs with w = 8, 16, 32, and 64 bits are faster than the fixed precision one as long as $n \leq 256$ bits ($n \leq n_{max}/2$). However, for the scalable design with w = 4, it is faster than the fixed precision one while $n \leq 192$ bits ($n \leq 3n_{max}/8$). In fact, as w gets bigger the delay decreases, which is a normal speed area trade-off.

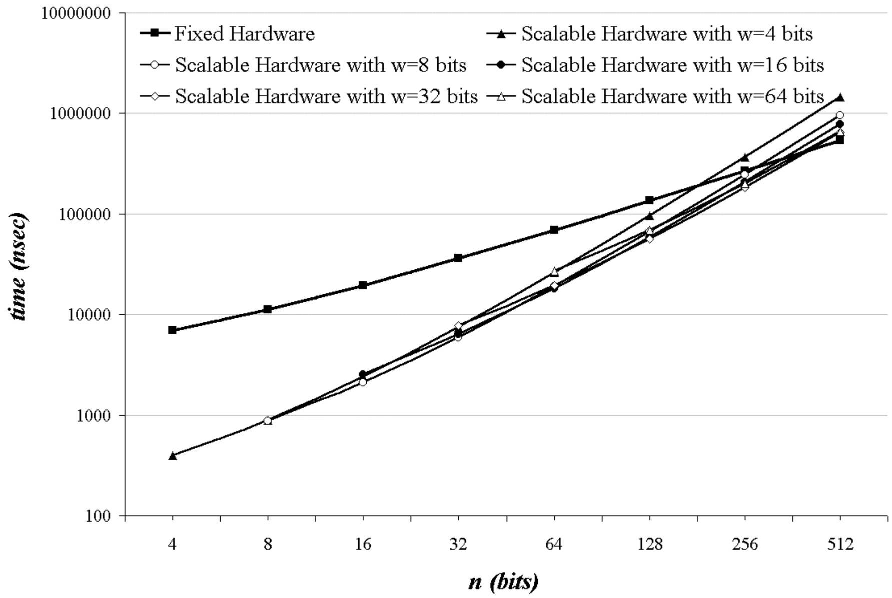

Figure 3.11 Delay comparison of designs with n_{max} = 512 bits

3.5.2.2 Technology independent speed comparison

To have a technology independent speed comparison we use the fact that the designs longest path passes through the adders (carry-ripple) and we are going to evaluate the clock period as a function of δ (the delay of each full adder). The delays of carry-ripple adders depend linearly on the number of bits they are built for, as listed in Table 3.4.

Adder number of bits	*4*	*8*	*16*	*32*	*64*	*128*	*256*	*512*
Estimated delay in δ units	*4δ*	*8δ*	*16δ*	*32δ*	*64δ*	*128δ*	*256δ*	*512δ*

Table 3.4 Adders δ delay estimation depending on the number of bits

The total time for each design is computed in δ units. For simplicity, consider $\delta=1$ which results in an estimated total time as a figure of merit. The technology independent speed comparison of all designs for $n_{max}=512$ bits is shown in Figure 3.12. Observe how the graph shows roughly similar behavior to the technology dependent speed comparison (Figures 3.10 and 3.11). Another observation from Figure 3.12 is that the scalable designs are faster than the fixed precision one as long as:

$$n < n_{max} / 2.5 \text{ and } w < n_{max}/4.$$

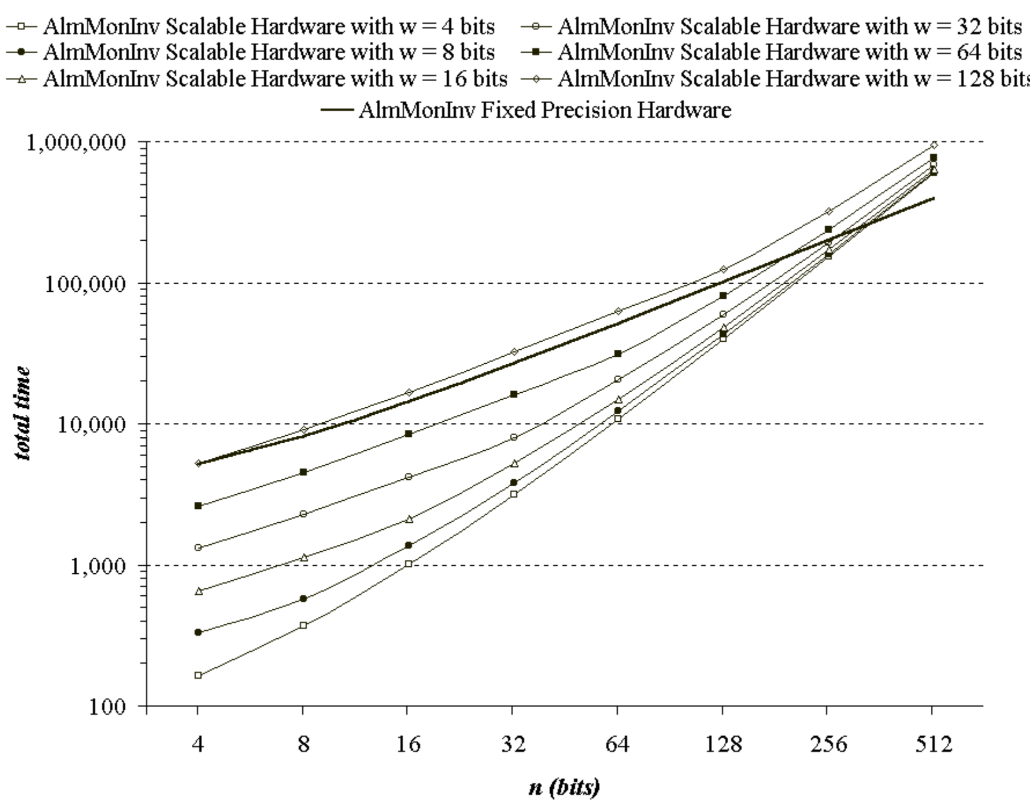

Figure 3.12 Technology independent speed comparison for designs with $n_{max}=512$ bits

3.6 Summary

This Chapter presents two hardware designs of an algorithm used in the computation of Montgomery modular inverse arithmetic. The two designs are the fixed precision hardware and the scalable hardware. The scalable architecture makes the design's longest-path shorter, compared to the fixed precision hardware, with a corresponding higher clock frequency. The scalable hardware is also designed to fit in a small area with the computation of numbers performed in a repetitive way. The maximum number of bits (n_{max}) the scalable hardware can handle depends only on the memory. If the number of bits exceeds the memory size, the memory unit is the only part that needs to be modified, while the scalable computing unit does not change. On the other hand, all the fixed precision hardware components need to be changed completely if any extra bit is to be added beyond the memory limit.

The scalable design shows area flexibility, depending on the number of bits used at each clock cycle (w). For example, if w = 4 bits and the design can handle up to 512 bits, the area of the scalable design is 60% less and faster in general than the fixed precision hardware. The comparisons show that this scalable structure is very attractive for cryptographic systems, particularly for ECC because of its need for modular inversion of large numbers, which differ in size repetitively depending on the application usage.

4 REDUCING THE CLOCK PERIOD OF THE ALMOST MONTGOMERY INVERSE HARDWARE DESIGNS

4.1 Introduction

The total computation time of the almost Montgomery inverse (AlmMonInv) hardware is a product of the number of clock cycles it takes and the clock period. The number of clock cycles depends on the input data. The clock cycle period depends on the design's critical path, which is dominated by the adders used in the design. In the previous chapter, the longest path delay of the proposed designs was put to its maximum due to the area optimization option selected for the synthesis phase. The synthesizer, Leonardo, optimized the design for the smallest area, and used the slow but small carry-ripple (CR) adders. In this chapter, a delay optimization option is applied that forces the synthesis tool to use one-level carry-look-ahead (CLA) adders. This is done to verify the impact of faster adders on the system performance and provide a clear idea of the area/time tradeoffs.

4.2 Shortening the Critical Path

As mentioned earlier, the critical path of the hardware is through the adders. The CR adders are the smallest and slowest adders [13,14]. A four bit CR adder, for $w=4$, is shown in Figure 4.1. This adder is made of 8 XOR gates and 12 NAND gates, which is equivalent to 36 NOT gates (or equivalent gates [14]). Observe the longest path involves the carry chain through all the four full adders. The longest path passes by 2 XOR gates and 6 NAND gates [14].

In order to reduce the critical path in the hardware, a faster adder should be used. When delay optimization is requested to the synthesis tool, it uses a CLA adder. This adder is faster than the CR adder but uses more area. A four bit CLA adder is shown in Figure 4.2. It is constructed of: 4 NAND, 4 XOR, 5 NOT, 7 NOR and 14 AND gates, which is equivalent to 56 NOT gates [14]. The longest path of this adder, however, passes through the following gates: NAND, AND, NOR and XOR, as shown in Figure 4.2.

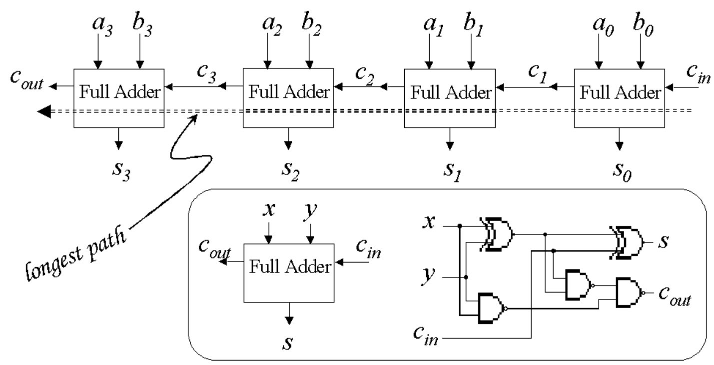

Figure 4.1 The carry-ripple adder

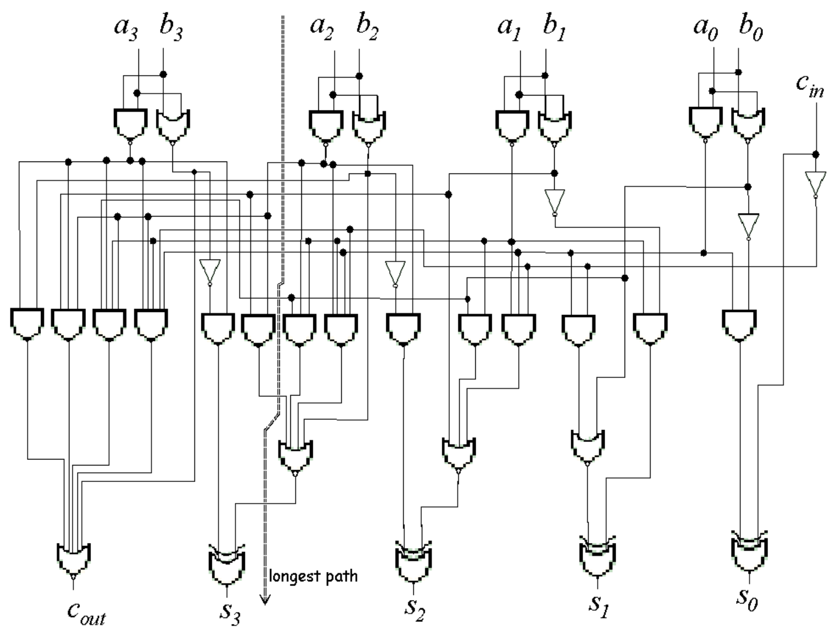

Figure 4.2 A four bit carry-look-ahead adder

4.3 Area & Delay Comparison

The four bits CR adder's area is equivalent to 36 gates while the CLA adder is equivalent to 56 gates, which corresponds to 55.5% area increase. On the other hand, the delay of the CR adder is through 2 XOR and 6 NAND gates while it's through a NAND, AND, NOR, and XOR gates for the CLA adder, which is shorter (the delay of an XOR gate is much more than AND, OR, and NAND) [14]. When the adders are used in the design, the speed and area impact on the complete hardware differs, because of the speed and area contributions of the other system components. A study of the impact of different optimizations (changing the adders) on the synthesis of the AlmMonInv scalable design is listed in Table 4.1. The results of the same experimentation performed on the AlmMonInv fixed precision hardware is shown in Table 4.2.

		Area Optimization		*Delay Optimization*		*Area*	*Delay*
n_{max} (bits)	*w* (bits)	*Period* (nsec)	*Area* (gates)	*Period* (nsec)	*Area* (gates)	*loss Percentage*	*Improvement Percentage*
128	4	9.62	9032	9.5	10364	14.75 %	1.20 %
128	8	12.39	9313	10.74	10568	13.47 %	13.3 %
128	16	19.48	9887	15.4	11357	14.87 %	21.0 %
128	32	30.66	11177	25.72	13148	17.63 %	16.1 %
128	64	54.93	13602	43.91	17028	25.19 %	20.0 %
128	128	102. 2	24453	79.53	31112	27.23 %	22.1 %
256	4	9.62	15346	9.5	17610	14.75 %	1.20 %
256	8	12.39	15627	10.74	17814	14.02 %	13.3 %
256	16	19.48	16201	15.4	18603	14.83 %	21.0 %
256	32	30.66	17491	25.72	20394	16.61 %	16.1 %
256	64	54.93	19916	43.91	24274	21.89 %	20.0 %
256	128	102. 2	30767	79.53	38358	24.67 %	22.1 %
512	4	9.62	27804	9.5	31906	14.75 %	1.20 %
512	8	12.39	28085	10.74	32110	14.33 %	13.3 %
512	16	19.48	28659	15.4	32899	14.8 %	21.0 %
512	32	30.66	29949	25.72	34690	15.83 %	16.1 %
512	64	54.93	32374	43.91	38570	19.14 %	20.0 %
512	128	102. 2	43225	79.53	52654	21.81 %	22.1 %

Table 4.1 Area and delay optimizations of the AlmMonInv scalable design

Consider Table 4.1 of the scalable design, the average area loss percentage is calculated to be 17.81%, which gains in the delay an average of 15.62%. Whereas, from Table 4.2, the fixed precision design's average area loss is calculated to be 21.23% to raise the average speed by 5.5%. This study clearly shows that changing the adders to fast ones benefits the scalable hardware in a much better way than the fixed precision one. The extra area needed to reduce the clock cycle period is much less for the scalable hardware than it is for the fixed precision design.

n_{max} (bits)	*Area Optimization*		*Delay Optimization*		*Area loss Percentage*	*Delay improvement Percentage*
	Period (nsec)	*Area* (gates)	*Period* (nsec)	*Area* (gates)		
4	11.41	796	11	925	16.20 %	3.59 %
8	15.96	1501	15	1817	21.05 %	6.01 %
16	26.5	2911	26	3576	22.84 %	1.88 %
32	48	6395	47	7496	17.21 %	2.08 %
64	92	12672	89	14944	17.92 %	3.26 %
128	178	23952	165	29001	21.07 %	7.30 %
256	350	46512	317	57010	22.57 %	9.42 %
512	694	69327	621	90907	31.12 %	10.5 %

Table 4.2 Area and delay optimizations of the AlmMonInv fixed precision design

5 A SCALABLE HARDWARE ARCHITECTURE FOR MONTGOMERY INVERSION IN GF(p)

5.1 Introduction

The starting point for the research of a complete Montgomery modular inverse hardware implementation is presented in [1]. The algorithm in [1] requires two main operations and in this Chapter we suggest replacing one of them by a simpler operation. A further modification to the inversion algorithm to use multi-bit shifting instead of single-bit shifting is also proposed. These two improvements reduce the number of clock cycles without significantly increasing the clock period, which results in an overall speedup of the inverse computation.

The improved algorithm is mapped to hardware when the scalability feature presented in Chapter 3 is also incorporated. In this hardware design, the long-precision numbers are divided into words and each word is processed in a clock cycle. It is shown that this hardware is appropriate for cryptographic applications. This work shows the area and speed of several scalable hardware configurations compared with a fixed precision design presented in [27].

Section 5.2 presents the Montgomery inverse algorithm including the new correction phase proposed in this work. Section 5.3 explains the multi-bit shifting strategy and corresponding modifications to the hardware algorithm. In Section 5.4 the scalable hardware design is described in some detail. The comparison between different hardware configurations is given in Section 5.5.

5.2 Montgomery Inverse Algorithm and Proposed Modifications

5.2.1 New Approaches for Montgomery Inverse

Let's consider the main Montgomery inverse problem again (introduced before in Section 3.2). An approach to calculate $x=a^{-1}2^{n} mod\ p$ from $a2^{n}$ can be to compute a first and then calculate the *AlmMonInv* (*Kaliski Phase one* (Section 3.2.1)) followed by *Kaliski Phase two* to get the desired inverse result. The first computation of a from $a2^n$ is performed by a modular division by 2^n named *Preparation Phase* as shown below.

Preparation Phase (Divide by 2^n)

Input: $r = a2^n$, n & p; where p=modulus & $2^{n-1} \leq p < 2^n$
Output: x; where $x = a \bmod p$
1. for $i = 1$ to n do
2. if r is even then $r = r/2$
3. else $r = (r + p)/2$
4. return $x = r$

Note that calculating a from $a2^n$ may be also obtained by a Montgomery multiplication [1] as follows:

$$\text{MonPro}(a2^n, 1) = a2^n\,(2^{-n}) \bmod p = a \bmod p.$$

However, the preparation phase is preferred in our case instead of MonPro, since it clearly can be implemented using the same hardware components of the AlmMonInv already proposed in Section 3.4.

Another new way to calculate the Montgomery inverse is by applying the *AlmMonInv* on the input $a2^n$ to produce r and k according to the formula:

$$(r,k) = AlmMonInv\,(a2^n)$$

where

$$r = (a2^n)^{-1} 2^k \bmod p = a^{-1} 2^{k-n} \bmod p$$

Recall that Montgomery inverse of $a2^n$ is $a^{-1}2^n \bmod p$, which implies that the *AlmMonInv* result ($a^{-1}2^{k-n} \bmod p$) must be corrected. It is possible to find a constant C such that:

$$C \times (a^{-1}2^{k-n} \bmod p) = a^{-1}2^n \bmod p.$$

Applying some algebra we get:

$$C=(a^{-1}2^n \bmod p)/(a^{-1}2^{k-n} \bmod p)=(a^{-1}2^n)/(a^{-1}2^{k-n})=(2^n)/(2^{k-n})=2^{n-(k-n)}=2^{2n-k}$$

The modular multiplication of ($a^{-1}2^{k-n} \bmod p$) by (2^{2n-k}) can be performed as follows:

$$([((((a^{-1}2^{k-n})\underbrace{.2).2).2)\ldots\ldots\ldots\ldots\ldots..2).2}_{2n\text{-}k\ times})] \bmod p) = a^{-1}2^n \bmod p$$

This arrangement of applying the modular operation after completing the multiplication is very expensive because the result of the multiplication by 2^{2n-k} may be much greater than the modulus and a large amount of hardware will be required to handle it [11]. However, the operation can be simplified by introducing the modular reduction after each multiplication by 2 as the following:

$$[((((((a^{-1}2^{k-n}).2) \bmod p).2) \ldots 2) \bmod p).2) \bmod p)]=a^{-1}2^{n} \bmod p$$

The modular reduction operation is performed by a subtraction of p whenever the number exceeds p. The proposed correction phase consists then in performing a multiplication of $a^{-1}2^{k-n}$ by $C = 2^{2n-k}$ as outlined below:

Correction Phase (Multiply by 2^{2n-k})

Input: r, p, n & k; where r & k are *AlmMonInv* outputs
Output: x; where $x = a^{-1}2^{n} \bmod p$
1. for $i = 2n\text{-}k$ to 0 do
2. $\quad r = 2r$
3. $\quad$ if $r > p$ then $r= (r - p)$
4. return $x = r$

5.2.2 Evaluation of Alternatives

Several methods considered for hardware computation of the Montgomery inverse are shown in Figure 5.1; including the procedures proposed by Savas and Koç in [1] using MonPro. Each path in the graph has its own set of routines and its total computation time. Figure 5.1 presents the approximate number of iterations for each routine. Note that the number of iterations for multiplication is estimated considering serial-parallel multipliers, because fully parallel multipliers are extremely large [6].

All approaches of Figure 5.1 lead to the same final result. However, the number of iterations in each path proves that our two-phase method, the *AlmMonInv* followed by the *correction phase* (path: 1-4-6), is the fastest. It requires only $2n$ iterations to complete the inversion as shown in Table 5.1, the *AlmMonInv* needs $1.5n$ iterations, and the correction phase (*CorPh*) needs $0.5n$ iterations, assuming an average value of $k=1.5n$ [1].

Observe that the other approach proposed in Section 5.2.1 (path: 1-2-3-6) would require $3n$ iterations in average to complete the inversion (Table 5.1); it is a slow alternative and for this reason will not receive further attention. For the previously proposed methods using *MonPro* multipliers (path: 1-4-3-6 or 1-4-5-6) [1], even if the multipliers are completely parallel (one iteration instead of n), they need more than $2n$ iterations, which is still slower than using the path 1-4-6. The proposed method is the only two-phase method in the graph (Figure 5.1).

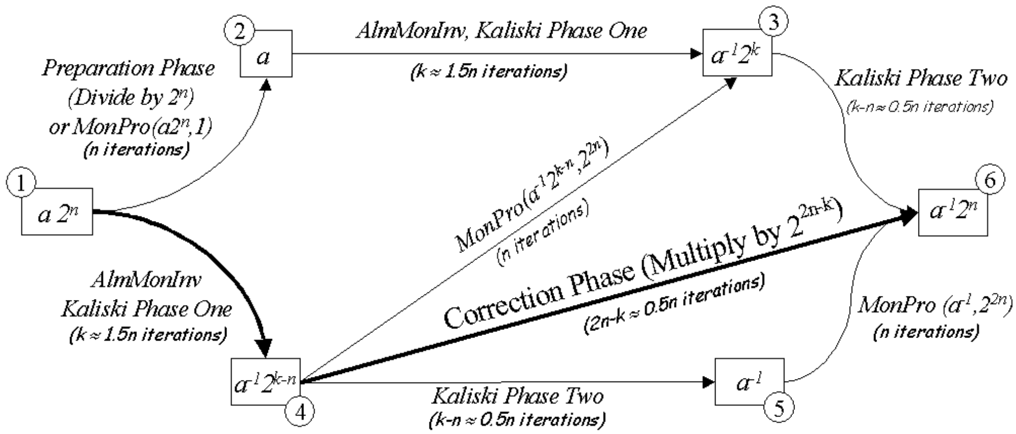

Figure 5.1 Different ways to compute the Montgomery inversion

	MonInv computation path	*Delay (number of iterations)*
	1-4-3-6	*1.5n+n+0.5n = 3n*
New	*1-2-3-6*	*n+1.5n+0.5n = 3n*
	1-4-5-6	*1.5n+0.5n+n = 3n*
New	*1-4-6*	*1.5n +0.5n = **2n***

Table 5.1 Delay of different ways to compute the Montgomery inverse

5.3 Multi-Bit Shifting

The AlmMonInv algorithm needs to finish its computation completely before the CorPh algorithm begins processing. This data dependency allows the use of the same hardware to execute both algorithms, i.e., both the AlmMonInv and CorPh. The following sections present an improvement of the AlmMonInv and CorPh algorithms based on a multi-bit shifting method.

5.3.1 AlmMonInv Hardware Algorithm

The AlmMonInv algorithm, when observed from hardware point-of-view, contains operations that are easily mapped to hardware as described in Section 3.3.1, which also provides the fixed precision hardware AlmMonInv algorithm (FHW-Alg) used in this section. Observe step 10 of the AlmMonInv algorithm (Section 3.2.1), the result of $r=p$ occurs if-and-only-if $a=\infty$, which cannot happen since $a\in[1,p-1]$. Thus, the result of AlmMonInv algorithm equals either $2p-r$ when $r>p$, or $p-r$ when $r<p$ (as described in step 10 of the FHW-Alg).

5.3.2 Best Maximum Distance for Multi-bit Shifter

Consider the FHW-Alg (Section 3.3.1). The operation to shift numbers u and s (step 2), or v and r (step 3), are performed depending on u_0 and v_0. In fact, when u_0 or v_0 is zero, only shift operation happens. Suppose that the four least significant (LS) bits of u are zeros. The shifting process on u will consume four iterations to be completed.

The multi-bit shifting method can be applied to shift two, three, four, five or more bits depending on the number of continuous zeros found at the LS bit positions of u and v. However, this number of zeros depends on data that are modified during the process. Thus a probabilistic analysis of the bit vectors u and v will give us an idea about maximum number of bits to be shifted.

Let p be the probability of a bit to be zero and $q=(1-p)$ be the probability of being one. The probability function PF used to calculate the probability of having x consecutive LS bits of u or v as zeros is: $PF(x) = qp^x$; where x is the number of LS zeros [26]. Note that as x gets larger $PF(x)$ reduces tremendously. The $PF(x)$ values show that multi-bit shifting should be investigated only for $x<6$ bits.

In the FHW-Alg presented in Section 3.3.1, the loop (steps 2 through 8) is executed for *k* iterations. Based on experimental statistics collected with a software implementation of the AlmMonInv algorithm, nearly half of the *k* algorithm iterations are used executing step 4 (addition and subtraction) and the other half executing only steps 2 or 3 (shifting process) [1]. Applying the multi-bit shifting approach will reduce the number of iterations for the shifting process only. Reusing $p=0.5$ as the probability of performing a shift operation, we estimate the average number of iterations based on a probabilistic model. Table 5.2 shows probabilistic equations to compute the number of iterations when a multi-bit shifter of up to *x* bits is available. The first polynomial term (as clarified in Figure 5.2 for the case of $x=3$) stands for the number of iterations used for addition and subtraction (step 4 of FHW-Alg). This term is not affected at all by *x* (the maximum number of bits to be shifted). The following terms consider the use of multi-bit shifting. The total number of iterations (*k*) will be affected according to the number of bits shifted. Given the value *p* that was defined before, the average number of iterations (*i*) is computed as listed in the last column of Table 5.2.

x	*Probabilistic Equations*	*i*
1	$(1-p)k + pk$	*1.00 k*
2	$(1-p)k + p[(1-p)k + p\,k/2]$	*0.88 k*
3	$(1-p)k + p[(1-p)k + p((1-p)\,k/2 + p\,k/3)]$	*0.85 k*
4	$(1-p)k + p[(1-p)k + p((1-p)\,k/2 + p\,[(1-p)\,k/3 + p\,k/4])]$	*0.849 k*
5	$(1-p)k + p[(1-p)k +p((1-p)\,k/2+p[(1-p)k/3+p((1-p)k/4+pk/5)])]$	*0.847 k*

Table 5.2 Average number of iterations (*i*)

After comparing the different *i* values, the notable improvement is found for the case with $x=3$ (shifting up to three bits), which gives the average of 15% reduction in the number of iterations (*k*). Note that there is not a significant improvement when $x>3$.

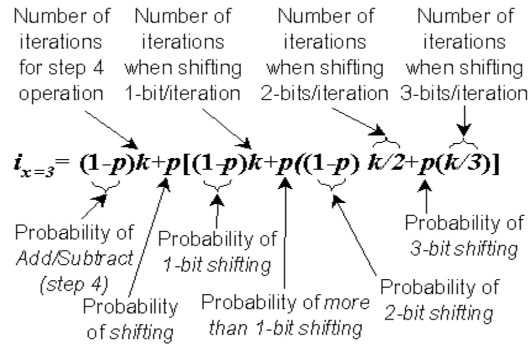

Figure 5.2 Description of *i* for the case of $x = 3$

5.3.3 Adjustments to FHW-Alg

The new capability to shift up to three bits requires a modification in the FHW-Alg, which is reflected in some units of the AlmMonInv hardware. The modified algorithm is shown below as the *multi-bit shifting AlmMonInv hardware algorithm*.

Multi-Bit Shifting HW-Alg (MHW-Alg)

Registers: u, v, r, s, & p (all five registers hold n bits)
Input: $a \in [1, p-1]$, p = modulus.
Output: result$\in [1, p-1]$ & k; where result$=a^{-1}2^k mod\ p$ & $n \leq k \leq 2n$
1. u = p, v = a, r = 0, s = 1, k = 0
2. if($u_2u_1u_0$=000)then{u=ShiftR(u,3);s=ShiftL(s,3);k=k+3};goto 8
2.1. if($u_2u_1u_0$=100)then{u=ShiftR(u,2);s=ShiftL(s,2);k=k+2};goto 8
2.2. if($u_2u_1u_0$=110)then{u=ShiftR(u,1);s=ShiftL(s,1)};goto 7
3. if($v_2v_1v_0$=000)then{v=ShiftR(v,3);r=ShiftL(r,3);k=k+3};goto 8
3.1. if($v_2v_1v_0$=100)then{v=ShiftR(v,2);r=ShiftL(r,2);k=k+2};goto 8
3.2. if($v_2v_1v_0$=110)then{v=ShiftR(v,1);r=ShiftL(r,1)};goto 8
4. S1 = Subtract (u, v); S2 = Subtract (v, u); A1 = Add (r, s)
5. if($S1_{borrow}$=0)then{u=ShiftR(S1,1);r=A1;s=ShiftL(s,1)};goto 7
6. s = A1; v = ShiftR(S2,1); r = ShiftL(r,1)
7. $k = k + 1$
8. if (v $\neq$ 0) go to step 2
9. S1 = Subtract (p, r); S2 = Subtract ($2p$, r)
10. if($S1_{borrow}$=0)then{return result=S1}; else{return result = S2}

The MHW-Alg when implemented in hardware requires: two subtractors (used in steps 4 and 9), an adder (step 4), a k-counter (that variably increments up to three), two multi-bit shifters (to shift u and s or v and r up to three bits, steps 2 to 3.2), and five n-bit registers (to store all the variables: u, v, r, s and p).

5.3.4 Suitable Multi-Bit Shifting the CorPh

The *CorPh* algorithm contains operations that are easily mapped to hardware components as shown in the *CorPh* hardware algorithm (HW-Alg2) below:

CorPh Hardware Algorithm (HW-Alg2)

Registers: r & p (two registers to hold n bits).
Input: r,p,n,k; where r $(r = a^{-1}2^{k-n} \bmod p)$ & k from *AlmMonInv*
Output: result; where result = $a^{-1}2^{n} \pmod p$.
11. $j = 2n-k-1$
12. While $j>0$
13. r = ShiftL(r,1); $j = j$-1
14. S1 = Subtract(r, p)
15. if ($S1_{borrow} = 0$) then {r = S1}
16. return result = r

To implement the HW-Alg2 in fixed precision hardware we need: two n-bit registers (to store r and p), a subtractor (step 14), a shifter, and a counter (step 13). The one-bit shifter (step 13) can be easily modified to perform multi-bit shifting and clearly reduce the number of iterations. The ideal situation is to implement HW-Alg2 utilizing the same MHW-Alg (Section 5.3.3) hardware components. Since the shift operation in the HW-Alg2 is followed by a subtraction, applying the multi-bit shifting technique to the algorithm demands extra subtractors to perform these operations in parallel and fully speedup the process. The total number of iterations and the corresponding number of subtractors for some shifting distances are listed in Table 5.3.

The practical choice of the maximum shifting distance in the *CorPh* implementation is two. This decision is due to the need of three subtractors when shifting two bits, which are already found in the *AlmMonInv* hardware (assuming two's complement subtraction). If the maximum distance is three, seven subtractors are required, which is far beyond the *AlmMonInv* hardware capability. To clarify this issue and how Table 5.3 is generated, let us start by assuming single-bit shifting. Observe that $r<p$, and $2r$ cannot reach $2p$, at most one subtraction will be needed when $2r>p$. When the distance is two, we need to shift r two bits to obtain $4r$, where $4r<4p$. This time, $4r$ must be reduced by subtractions of $3p$, $2p$, or p if necessary. The CorPh algorithm is modified to accommodate the two-bit shifting as shown in the multi-bit shifting CorPh hardware algorithm below (MHW-Alg2).

Multi-Bit Shifting HW-Alg2 (MHW-Alg2)

Registers: r, u, v & p (all four registers are to hold n bits).
Input: r,p,n,k; where r $(r=a^{-1}2^{k-n} mod\ p)$&k from *AlmMonInv*
Output: result; where result = $a^{-1}2^{n}$ $(mod\ p)$.
11. $j = 2n\text{-}k\text{-}1$
12. $v = 2p$; $u = 3p$
13. While $j > 0$
14. if $j = 1$ then {r = ShiftL(r,1); $j=j-1$}
15. else {r = ShiftL(r,2); $j=j-2$}
16. S1=Subtract(r,p);S2=Subtract(r,v);S3=Subtract(r,u)
17. if ($S3_{borrow} = 0$) then {r = S3}
18. else if ($S2_{borrow} = 0$) then {r = S2}
19. else if ($S1_{borrow} = 0$) then {r = S1}
20. return result = r

Number of bits to be shifted per iteration	*CorPh hardware number of subtractors*	*CorPh execution number of iterations*
1	1	*(2n-k)*
2	3	*(2n-k)/2*
3	7	*(2n-k)/3*
4	15	*(2n-k)/4*

Table 5.3 Speed and hardware changes due to multi-bit shifting the CorPh algorithm

The three subtraction operations are performed in parallel, as step 16 of MHW-Alg2. Four registers are needed to hold the variables r, u, v and p. The value of p is already available in register p, however, the values of $2p$ and $3p$ have to be computed once at the beginning of the *CorPh* and stored in registers v and u respectively (step 12). The counter, j, is set to $2n-k-1$ at step 11 (using the value k from *AlmMonInv*); it is used to keep track of the number of iterations in the algorithm.

5.4 The Scalable Design

5.4.1 Scalable Hardware Issues Applied to the Algorithms

Differently from what normally happens in the full-precision hardware design, the scalable hardware, as in [27], has multi-precision operators for shifting, addition, subtraction and comparison. Consider the MHW-Alg shown in Section 5.3.3, for example, the subtraction used for comparison ($u>v$) is performed on a word-by-word (w-bit slices) basis until all the data words (all n bits) are processed, as outlined below:

for i = 1 to n/w
$(x_{borrow}, x_{iw-1:iw-w})$ = Subtract $(u_{iw-1:iw-w}, v_{iw-1:iw-w}, x_{borrow})$
$(y_{borrow}, y_{iw-1:iw-w})$ = Subtract $(v_{iw-1:iw-w}, u_{iw-1:iw-w}, y_{borrow})$
$(z_{carry}, z_{iw-1:iw-w})$ = Add $(r_{iw-1:iw-w}, s_{iw-1:iw-w}, z_{carry})$

Then, the final word borrow out bit is used to decide on the result. Also, depending on the subtraction completion, variable r or s has to be shifted. All variables, u, v, r and s, cannot change until the subtraction processes complete, and the borrow-out bit appears. This forces the use of three more variables: x, y and z; where $x=u-v$, $y=v-u$ and $z=r+s$. These variables are stored in extra registers increasing the number of hardware registers to eight. All the registers hold n_{max} bits even though the actual number of bits in the numbers are $n \le n_{max}$ bits. This n_{max} limit defines the memory capability and does not degrade the total computation time of the inversion process; i.e., the total delay of the computation depends on the actual number of bits (n) and not on n_{max}.

5.4.2 Scalable Hardware Design

The scalable hardware design is built of two main parts, a memory unit and a computing unit, as shown in Figure 5.3. It is very similar, in principle, to the scalable hardware presented in [27]. The memory unit is not scalable because it has a limited storage defined by the value of n_{max}. The data values of a and p are first loaded in the memory unit. Then, the computing unit read/write (modify) the data using a word size of w bits. The computing unit is completely scalable. It is designed to handle w bits every clock cycle. The computing unit does not know the total number of bits, n_{max}, the memory is holding. It computes until the controller indicates that all operands' words were processed. Note that the actual numbers used may be way smaller than n_{max} bits.

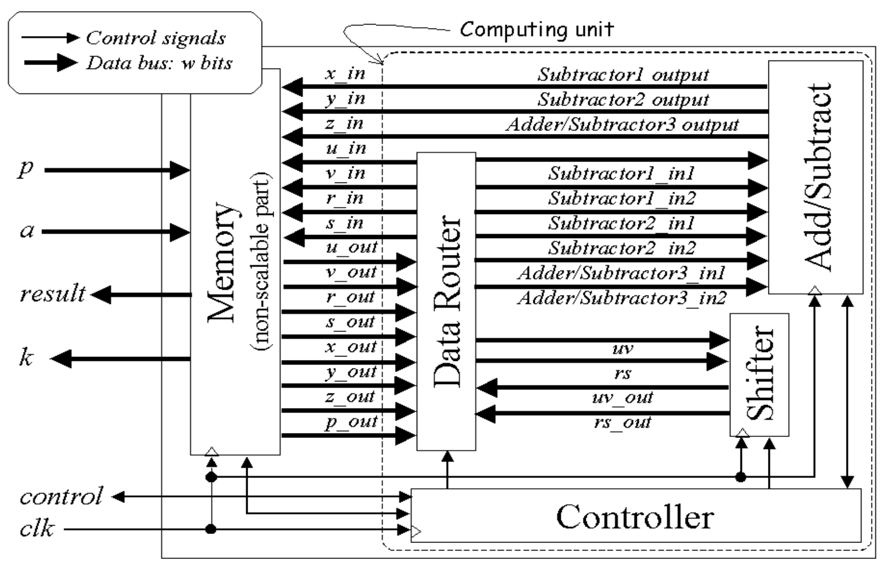

Figure 5.3 Montgomery inverse scalable hardware block diagram

The memory unit contains a counter to compute variable k and eight first-in-first-out (FIFO) registers used to store the inversion algorithm's variables. All registers, u, v, r, s, x, y, z and p, are limited to hold at most n_{max} bits. Each FIFO register has its own reset signal generated by the controller. They have counters to keep track of n (the number of bits actually used by the application).

The computing unit is made of four hardware blocks, the add/subtract, shifter, data router, and controller block. The add/subtract unit is built of two subtractors, an adder/subtractor, four flip-flops, one multiplexer, a comparator, and logic gates, connected as shown in Figure 5.4. This unit performs one of two operations, either two subtractions and one addition for the MHW-Alg (Section 5.3.3), or three subtractions for the MHW-Alg2 (Section 5.3.4). To execute MHW-Alg the Adder/Subtractor3 is controlled to work as an adder (step 4 of MHW-Alg). The same Adder/Subtractor3 is used as subtractor to execute step 16 of the MHW-Alg2. Three flip-flops are used to hold the intermediate borrow-bits of the subtractors and the carry-bit of the adder to implement the multi-precision operations. The fourth flip-flop is used to store a flag that keeps track of the comparison between u and v, which is used to perform step 8 of MHW-Alg. The borrow-out bits from the subtractors are connected to the controller used only at the end of the each multi-precision addition/subtraction operation. Subtractor 1 borrow-out bit is used to test the condition in step 5 of MHW-Alg. It is also essential in electing the result observed in step 10 of MHW-Alg. The three subtractors borrow-out bits ($S1_{borrow}$, $S2_{borrow}$, $S3_{borrow}$) are likewise necessary to select the correct 'if' condition to be used in steps 17, 18, or 19, of the MHW-Alg2 algorithm.

The multi-bit shifter is made of two multiplexers and two registers with special mapping of some data bits, as shown in Figure 5.5. The two multiplexers are used to select the correct set to be used in the multi-bit shifter. Depending on the controller signal *Distance*, the shifter acts as a one, two, or three-bit shifter. Two types of shifting are needed in the MHW-Alg algorithm, right shifting an operand (u or v) through the uv bus (one, two, or three bits) and left shifting another operand (r or s) through the rs bus (by similar number of bits). Right shifting u or v is performed through Register1, which is of size w-1 bits. For each word, w-1 bits of uv are stored in Register1. The LS bit(s) of each word is (are) read out immediately as the most significant bit(s) of the output bus *uv_out*. Left shifting r or s is performed via Register2, which is of size w+3 bits, in a similar fashion. When executing the MHW-Alg2, the left shifting is performed to a distance of either one or two bits using the rs path only.

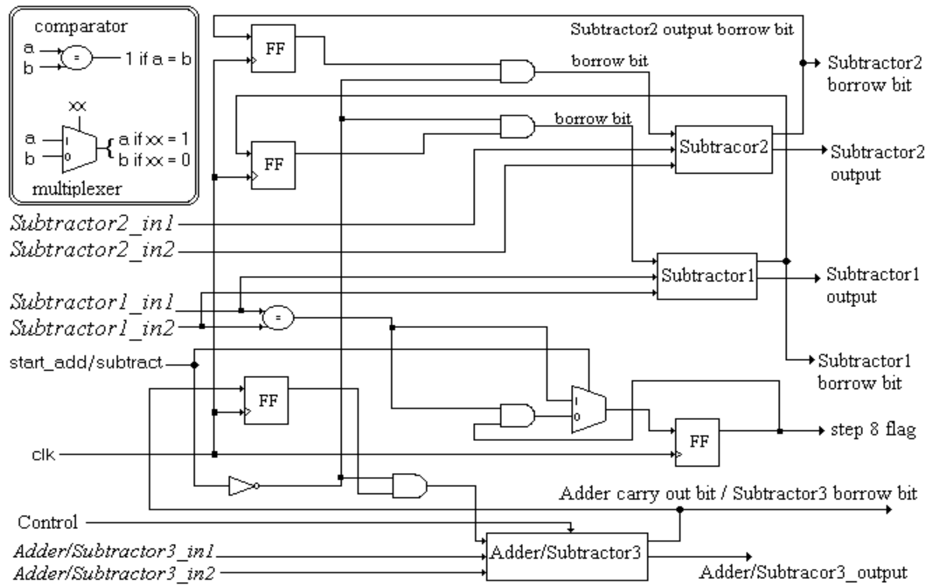

Figure 5.4 Add/subtract unit

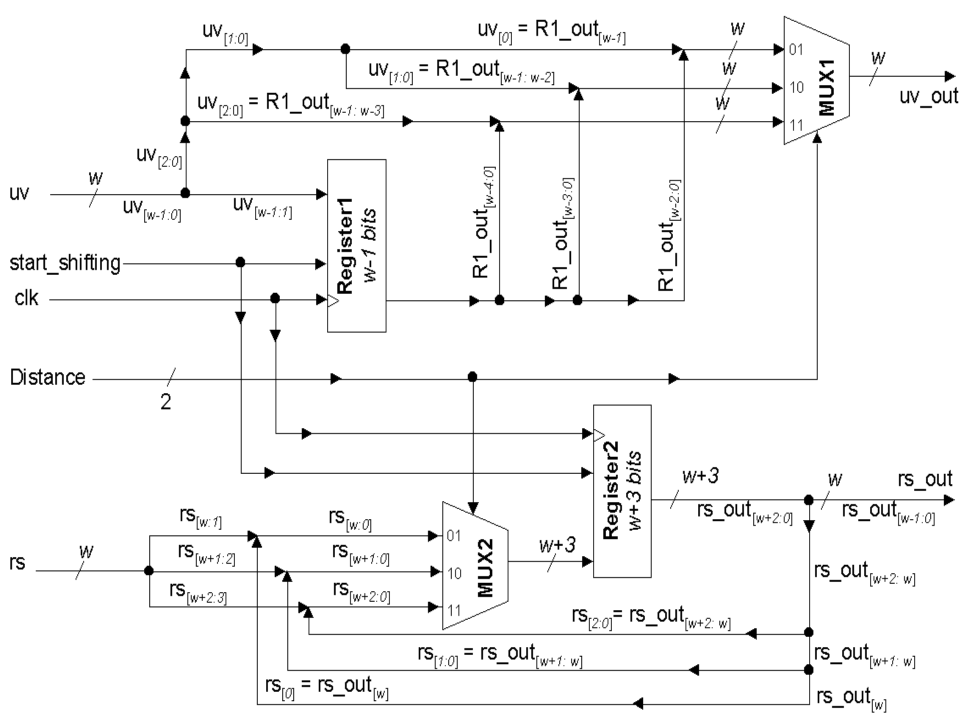

Figure 5.5 Multi-bit shifter (max distance = 3)

The data router shown in the complete hardware (Figure 5.3) is made of twelve multiplexers to connect the data going out of the memory unit to the inputs of the add/subtract unit or shifter and also transfers the shifted data values to their destination locations in the memory unit. The possible configurations of the data router are shown in Figure 5.6.

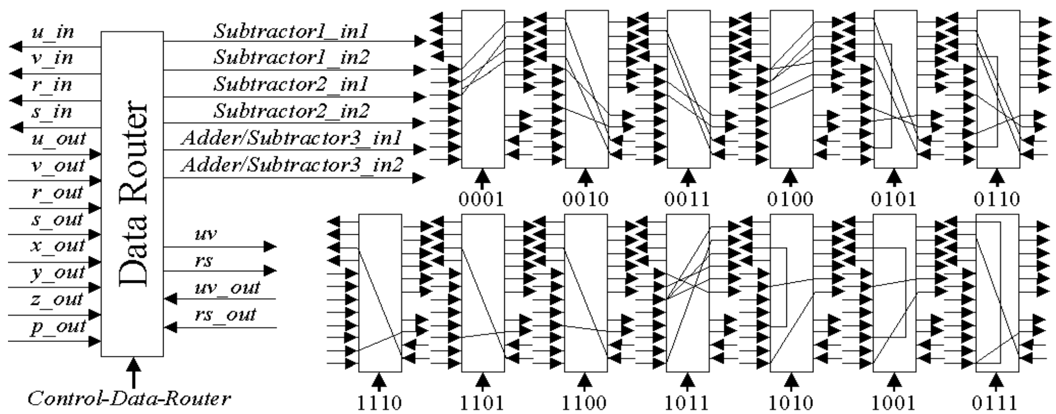

Figure 5.6 Data router configurations

The controller is the unit that coordinates the flow of data. It consists in a state machine easily derived from both MHW-Alg and MHW-Alg2. The controller does not include counters to avoid any dependency on the number of bits (n_{max}) that the system can handle. Such counters are located in the memory block, which is the non-scalable component in the system.

5.5 Modeling and Analysis

The proposed Montgomery inverse scalable design was modeled and simulated in VHDL similar to Section 3.5. It has two main parameters, namely n_{max} and w, which define several hardware configurations. These design configurations are compared in this work with other *fixed precision designs* previously described in [27] only parameterized by n_{max} because $w=n_{max}$ in their case.

For both area and speed comparisons, we show the *fixed precision design* in [27] modified to execute both MHW-Alg and MHW-Alg2, to be realistic and functionally similar to the scalable hardware of this work. Note that the area presented in [27] is the same given here because modifying the AlmMonInv hardware to process both AlmMonInv and CorPh will increase the area with a negligible amount due to modification in the controller. However, the time of [27] is different than what is here since it considers the execution of the complete Montgomery inverse computation. We didn't define a specific architecture for the adders and subtractors used in the designs. Thus, the synthesis tool chooses the best option from its library of standard cells.

5.5.1 Area Comparison

The area of the scalable designs and the fixed precision one are compared in Figure 5.7. As n_{max} increases the difference between the fixed precision hardware and scalable ones increases, which is expected because of the increasing burden of the computing unit of the fixed precision design. Observe that the fixed precision design has larger area than all scalable ones except for the configuration with w=128 and n_{max}<160 bits. As w approaches n_{max}, the scalable design's benefit reduces and the extra hardware used for multi-precision computation shows up. In other words, the scalable design with $w=n_{max}$ has the same size of adder and subtractors as the fixed one with extra hardware for scalability features, making it more expensive.

5.5.2 Speed Comparison

The total computation time is the product of the number of clock cycles the algorithm takes and the clock period of the final implementation. This clock period changes with the value of w in the scalable hardware (Table 5.4), and changes with the value of n_{max} in the fixed precision hardware (Table 5.5). Tables 5.4 and 5.5 lists the clock period for each design obtained from synthesis of the VHDL models.

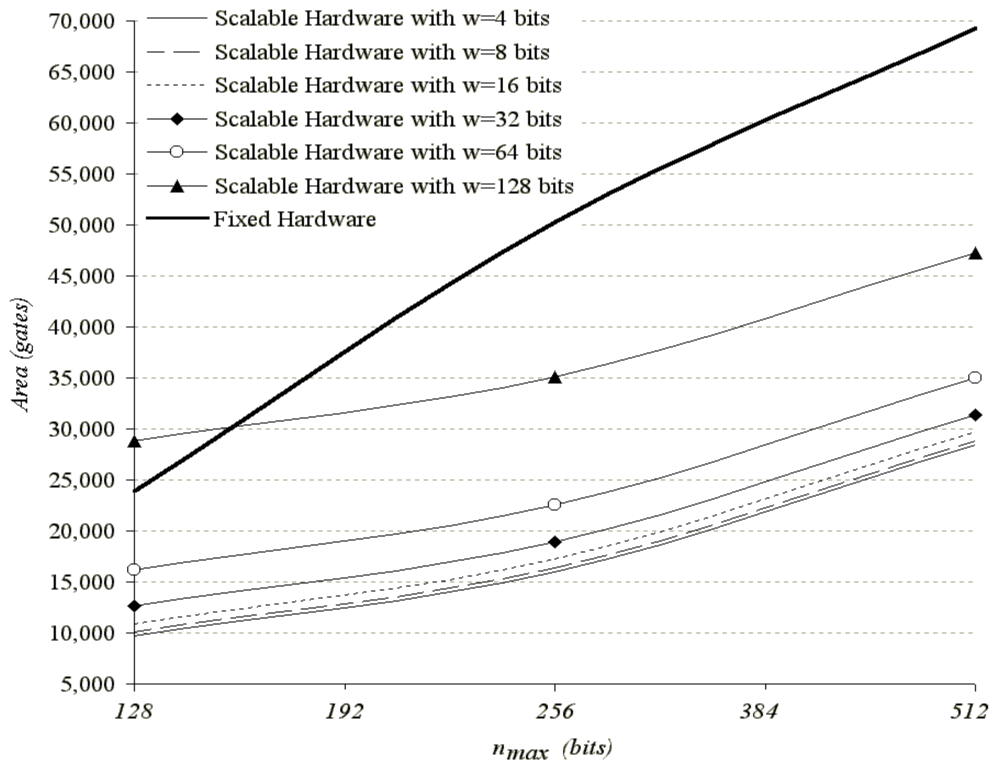

Figure 5.7 Area comparison

w	4	8	16	32	64	128
Period	12	14	19	28	47	82

Table 5.4 Clock cycle period for scalable designs (nsec)

n_{max}	32	64	128	256	512	1024
Period	50	93	178	351	694	1382

Table 5.5 Clock cycle period for fixed designs (nsec)

The number of clock cycles for all designs depends completely on the data and its computation. For the scalable design, the number of cycles is a function of three parameters: k, w and n. To compute any shifting, addition and/or subtraction, the number of cycles is calculated as $\lceil n/w \rceil$. The total number of clock cycles to execute step 2 or 3 is different than step 4. Step 4 needs extra $\lceil n/w \rceil$ cycles for the shifting operation after it (steps 5 or 6). The average number of clock cycles to perform each iteration of MHW-Alg (step 2 through step 8) is calculated as

$$\mathrm{CPI}_1=(0.5\lceil n/w \rceil)+(0.5(2\times\lceil n/w \rceil)),$$

(CPI stands for the clock cycles per iteration within the loop: step 2 to 8). The number of iterations of FHW-Alg is originally equal to k, but applying the multi-bit shifting of section 5.3.2, the average number of iterations reduces to $0.85k$. An extra $\lceil n/w \rceil$ cycles are needed once after ending the loop of MHW-Alg (Section 5.3.3) to perform steps 9 and 10. The overall average number of cycles to execute MHW-Alg equals

$$(\mathrm{CPI}_1\times 0.85k)+\lceil n/w \rceil.$$

Similarly, the average number of clock cycles of the scalable hardware to execute MHW-Alg2 (Section 5.3.4) equals to

$$\mathrm{CPI}_2\times(2n-k)/2;$$

where $\mathrm{CPI}_2=2\times\lceil n/w \rceil$ and $(2n-k)/2$ is the average number of iterations when shifting two bits per iteration, as explained in section 5.3.4. The value of k (MHW-Alg and MHW-Alg2) is within the range $[n,2n]$ [1], which justify the use of its average of $3n/2$, for comparison purposes. The total number of clock cycles required by the scalable design to complete Montgomery inverse computation is then calculated as

$$C_s=(2.4125n+1)\lceil n/w \rceil,$$

which was verified by several VHDL simulations.

For the fixed precision design to perform the CorPh after the AlmMonInv both using multi-bit shifting algorithms as MHW-Alg and MHW-Alg2, the total average number of clock cycles is $n+0.35k$; where $0.85*k$ cycles are used to execute MHW-Alg, and $(2n-k)/2$ cycles are allocated for MHW-Alg2. If k is approximated to its average of $3n/2$ (similar to the scalable design), the number of the clock cycles will be given by the function

$$C_f=1.525n.$$

Several scalable hardware configurations are designed depending on different n_{max} and w parameters. Each configuration can have different computation time depending on the actual number of bits, n, used. For example, Figure 5.8 shows the delay of six scalable hardware designs compared to the fixed precision hardware, all modeled for $n_{max}=512$ bits, which is a practical number for future ECC applications [11]. Observe how the actual data size (n) plays a big role on the speed of the designs. In other words, as n reduces and w is small, the number of clock cycles decrease significantly, which considerably reduces the overall computing time of the scalable design compared to the fixed precision one. This is a major advantage of the scalable hardware.

Recall that the number of clock cycles of all designs depends on the actual size of the data used and the actual data value. However, the fixed precision hardware clock period is always assumed to have n_{max} bits to process. i.e., if the application needs only $n=128$ bits, and all designs are made for $n_{max}=512$ bits, as the example of Figure 5.8, the fixed precision design clock frequency is not affected by n and all n_{max} bits are treated in the computation causing the fixed precision design to have a total time greater than all configurations of the scalable designs. This observation was found valid for other n_{max} values (designs for these cases we actually tested and synthesized). It was observed that all scalable designs are faster than the fixed precision one while

$$n < \begin{cases} (\log_2 w - 1)n_{max}/4 & \text{when } w < n_{max}/16 \\ n_{max} & \text{when } w \geqslant n_{max}/16 \end{cases}$$

In Figure 5.8, for example, as $n<n_{max}/2$ $(n=256)$ the fixed precision hardware is faster than the scalable one with $w=4$ bits and very similar to the design with $w=8$ bits. As $n>3n_{max}/4$ $(n=384)$ the scalable design with $w=16$ has a speed that falls below the fixed precision one. When $n=n_{max}=512$ the scalable design with $w=32$ bits has almost the same speed as the fixed precision one, but the ones with $w>n_{max}/16$ bits remain faster. In fact, as w gets bigger, the total time decreases, which is also true when comparing among the different scalable designs as long as $n \geq w$ (Figure 5.8). Whenever $n<w$, considering the scalable designs only, the advantage of the scalable designs reduces indicating that the number of words to be processed reached its lower limit, but still the scalable designs are faster than the fixed precision one.

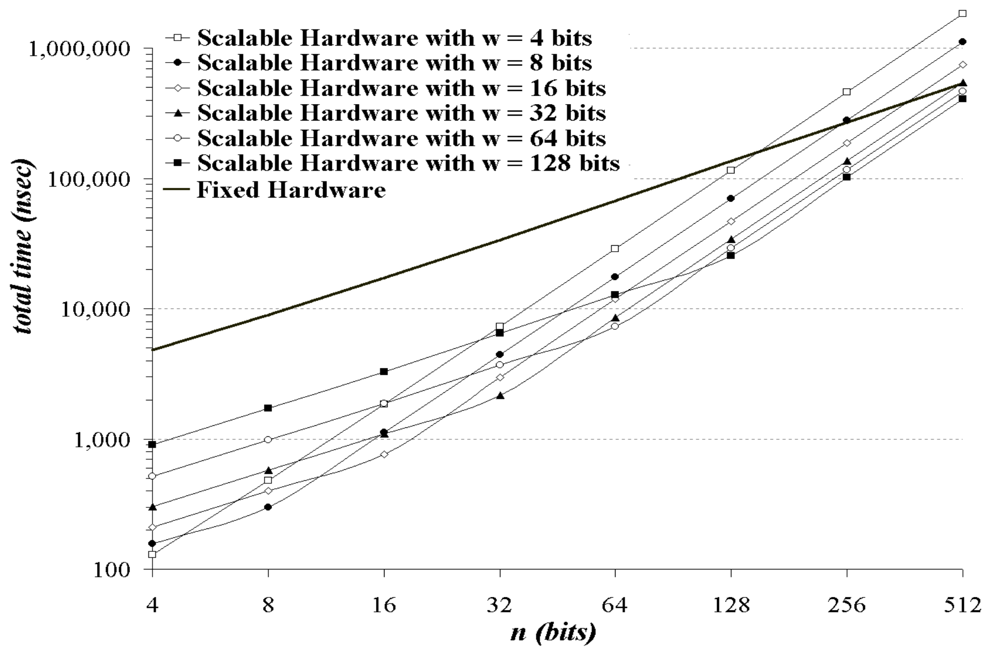

Figure 5.8 Delay comparison of designs with $n_{max} = 512$ bits

The previous speed comparison results depend on Leonardo's clock periods (technology dependent). If we use the technology independent method discussed in Section 3.5.2.2, the speed comparison is as shown in Figure 5.9. Note that all scalable hardware designs are faster than the fixed precision designs while:

$$n < n_{max} / 1.5$$

Another observation from Figure 5.9 is that the scalable design configurations speeds converge (tending to be very similar) when:

$$n > w.$$

This gives the general impression of disagreement with Leonardo's speed comparison results (Figure 5.8), which is due to the technology independent assumption (Section 3.5.2.2) of considering the longest path of the designs only by the adders. Although the adders dominate the longest paths of the designs, other components (controller and data router) affects too.

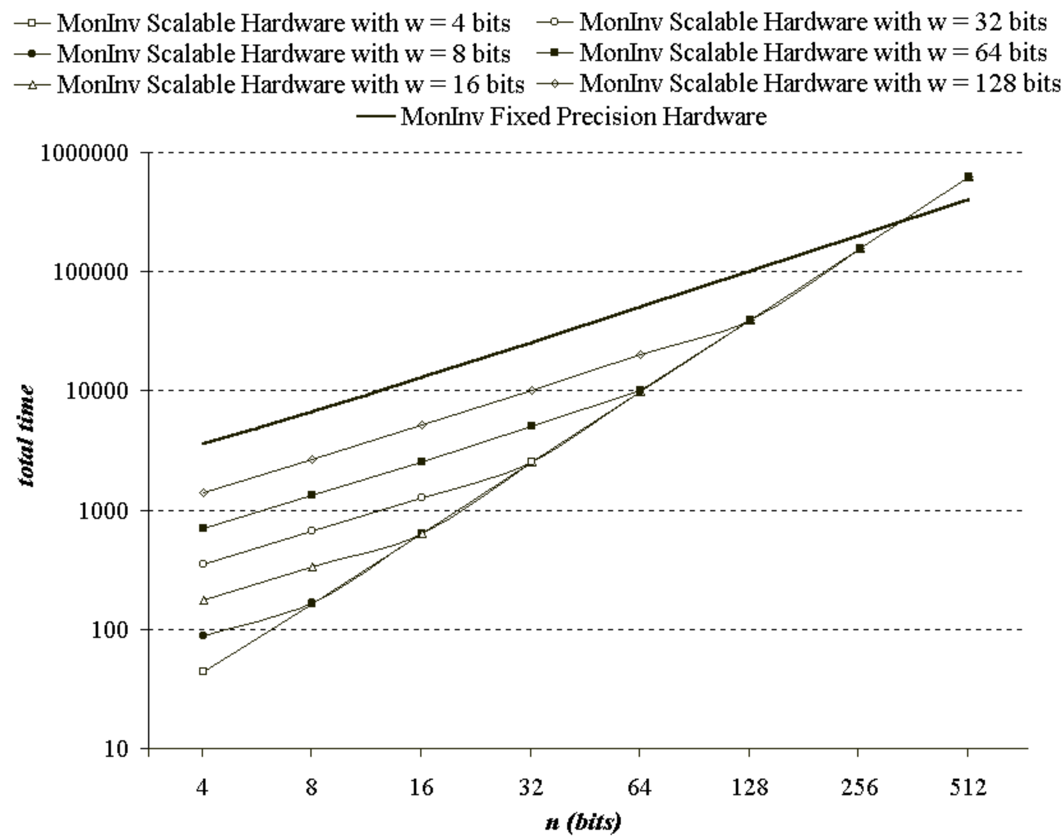

Figure 5.9 Technology independent speed comparison for all designs with n_{max}=512 bits

6 SCALABLE AND UNIFIED HARDWARE TO COMPUTE MONTGOMERY INVERSE IN GF(p) AND GF(2^n)

6.1 Introduction

Cryptographic inverse calculations are normally defined over either prime or binary extension fields [11], more specifically *Galois Fields GF(p)* or *GF(2^n)*. All available application-specific integrated circuit (ASIC) implementations for inversion computation [16-18,20-25] are created strictly for one finite field, either GF(p) or GF(2^n). If the hardware at hand is for GF(2^n) calculations, such as [17,18,20-25], and the application needs GF(p) computation, a completely different hardware is required [11]. It is inefficient to have two hardware designs (one for GF(p) and another for GF(2^n)) when only one is needed each time. This issue motivated the search for a single unified hardware architecture used to compute inversion in either finite field GF(p) or GF(2^n), similar, in principle, to the multiplier idea proposed in [7].

The GF(p) Montgomery inverse (MonInv) algorithm (presented in Chapter 5) is an efficient method for doing inversion with an odd modulus. The algorithm is particularly suitable for implementation on application specific integrated circuits (ASICs). For GF(2^n) inversion, the original inverse procedure (presented in [37]) has been extended to the finite field GF(2^n) in [35]. It replaces the modulus (*p*) by an irreducible polynomial (*p(x)*), and adjusts the algorithm according to the properties of polynomials. We implemented the inversion algorithms in hardware based on the observation that the Montgomery inverse algorithm for both fields GF(p) and GF(2^n) can be very similar. We show that a unified architecture computing the Montgomery inversion in the fields GF(p) and GF(2^n) is designed at a price only slightly higher than the one for only the filed GF(p), providing major savings when having both types of inverters is desirable or required.

A scalable Montgomery inverter design methodology for GF(p) was introduced in Chapters 3 and 5. This methodology allows the use of a fixed-area Montgomery inverter ASIC design to perform the inversion of unlimited precision operands. The design tradeoffs for best performance in a limited chip area were also analyzed in Section 5.5. We use the design approach as in [27] to obtain a scalable hardware module. Furthermore, the scalable inverter described in this Chapter is capable of performing inversion in both finite fields GF(p) and GF(2^n) and is for this reason called a scalable and unified Montgomery inverter.

There are two main contributions of this Chapter. First, we show that a unified architecture for inversion can be easily designed without compromising scalability and without significantly affecting delay and area. Second, we investigate the effect of word length (*w*) and the actual number of bits (*n*) on the hardware area, based on actual implementation results obtained by synthesis tools. In Section 6.2, we propose the GF(2^n) extended Montgomery inverse procedure that has several features suitable for an efficient hardware implementation. The unified architecture and its operation in both types of finite fields, GF(p) and GF(2^n), are described in Section 6.3. Section 6.4 presents the area/time tradeoffs and appropriate choices for the word lengths of the scalable module. Finally, a summary is discussed in Section 6.5.

6.2 Montgomery Inverse Hardware Procedures For *GF(p)* and *GF(2^n)*

In order to design a unified Montgomery inverse architecture, the GF(p) and GF(2^n) algorithms need to be very similar and this way consume the least amount of extra hardware. Extending the GF(p) Montgomery inverse algorithm to GF(2^n) is practical due to the removal of carry propagation required in the addition of GF(p) element and simple adjustments of test conditions. In other words, the GF(2^n) algorithm is like a simplification of the GF(p) one. The converse (modifying GF(2^n) algorithms for GF(p)), on the other hand, is very difficult [7,11,35,36].

As explained before (Section 5.2), the scalable GF(p) Montgomery inverse (*MonInv*) procedure proposed in this work consists in two phases: the almost Montgomery inverse (*AlmMonInv)* and the correction phase (*CorPh*). Both GF(p) AlmMonInv and CorPh algorithms were mapped to hardware features and further modified for multi-bit shifting, a concept discussed in Section 5.3, which resulted in an efficient implementation of the GF(p) Montgomery inverse. The GF(p) multi-bit shifting for both AlmMonInv and CorPh hardware algorithms (MHW-Alg and MHW-Alg2, respectively), are outlined in Figure 6.1.

MHW-Alg: GF(p) Multi-Bit Shifting AlmMonInv Hardware Algorithm

Registers: u, v, r, s, x, y, z, and p (all registers hold n_{max} bits)

Input: $a2^m \in [1, p\text{-}1]$; Where p = modulus, and $m \geq n$ ($2^{n-1} \leq p \leq 2^n$)

Output: result$\in [1, p\text{-}1]$ & k;
Where result=$a^{-1}2^{k-m} mod\ p$ & $n<k<2n$

1. u = p; v = $a2^m$; r = 0; s = 1; x = 0; y = 0; z = 0; k = 0
2. if($u_2u_1u_0$=000)then{u=ShiftR(u,3);s=ShiftL(s,3);k=k+3};goto 8

2.1. if($u_2u_1u_0$=100)then{u=ShiftR(u,2);s=ShiftL(s,2);k=k+2};goto 8

2.2. if($u_2u_1u_0$=110)then{u=ShiftR(u,1);s=ShiftL(s,1)};goto 7

3. if($v_2v_1v_0$=000)then{v=ShiftR(v,3);r=ShiftL(r,3);k=k+3};goto 8

3.1. if($v_2v_1v_0$=100)then{v=ShiftR(v,2);r=ShiftL(r,2);k=k+2};goto 8

3.2. if($v_2v_1v_0$=110)then{v=ShiftR(v,1);r=ShiftL(r,1)};goto 8

4. x = Subtract (u, v); y = Subtract (v, u); z = Add (r, s)
5. if(x_{borrow}=0)then{u=ShiftR(x,1); r=z; s=ShiftL(s,1)};goto 7
6. s = z; v = ShiftR(y,1); r = ShiftL(r,1)
7. $k = k + 1$
8. if (v ≠ 0) go to step 2
9. x = Subtract (p, r); y = Subtract ($2p$, r)
10. if(x_{borrow} = 0)then{result=x}; else{ result = y}

MHW-Alg2:GF(p) Multi-Bit Shifting CorPh Hardware Algorithm

Registers: r, u, v, x, y, z, and p (all registers hold n_{max} bits)

Input: r, p, n, k;
Where r (r=$a^{-1}2^{k-m} mod\ p$)&k from *MHW-Alg*

Output: result; Where result = $a^{-1}2^m$ *(mod p)*.

11. $j = 2m\text{-}k$; x = 0; y = 0; z = 0
12. $v = 2p$; $u = 3p$
13. While $j > 0$
14.   if $j = 1$ then {r = ShiftL(r,1); j=j-1}
15.   else {r = ShiftL(r,2); j=j-2}
16.   x=Subtract(r,p);y=Subtract(r,v);z=Subtract(r,u)
17.   if (z_{borrow} = 0) then {r = z}
18.   else if (y_{borrow} = 0) then {r = y}
19.   else if (x_{borrow} = 0) then {r = x}
20. result = r

Figure 6.1 Montgomery inverse hardware algorithm for GF(p)

Differently from what normally happens in a full-precision hardware design, the scalable hardware, as in [6-8,27], has multi-precision operators for shifting, addition, subtraction and comparison. Observe the AlmMonInv algorithm in Figure 6.1, for example, the scalable subtraction (step 4) is also used for comparison ($u>v$), which is performed on a word-by-word (w-bit words) basis until all the actual data words (all n bits) are processed. Then, the final word borrow out bit is used to decide on the result. Also, depending on the subtraction completion, variable r or s has to be shifted. All variables, u, v, r and s, need to remain as is until the subtraction process is complete, and the borrow out bit appears. For this reason, eight registers are required, as shown in Figure 6.1.

6.2.1 Representation and Manipulation of Elements in GF(2^n)

The inversion algorithm for GF(2^n) considered in this work was presented in [35]. Although prime and binary extension fields, GF(p) and GF(2^n), have different properties, the elements of either field are represented using similar data structures. The elements of the field GF(2^n) can be represented in several different ways [11]. The polynomial representation, however, is a useful and appropriate form to the unified implementation, as used for the unified multiplier in [7]. According to the GF(2^n) polynomial representation, an element $a(x)$ in GF(2^n) is a polynomial of length n, i.e., of degree less than or equal to $n-1$, written as

$$a(x)=a_{n-1}x^{n-1}+a_{n-2}x^{n-2}+ \dots +a_2x^2+a_1x+a_0,$$

where a_i is an element in GF(2). These coefficients a_i are represented as bits in the computer and the element $a(x)$ is represented as a bit vector $a=(a_{n-1}\ a_{n-2} \dots a_2\ a_1\ a_0)$.

The addition/subtraction of two elements $a(x)$ and $b(x)$ in GF(2^n) is performed by adding/subtracting the polynomials $a(x)$ and $b(x)$, where the coefficients are added/subtracted in the field GF(2). As a consequence, both addition and subtraction operations are exactly the same and equivalent to bit-wise XOR operations on the bit vectors a and b ($a_i \oplus b_i$). In order to compute the inverse of element $a(x)$ in GF(2^n), we need an irreducible polynomial of degree n. Let the irreducible polynomial be

$$p(x)=x^n+p_{n-1}x^{n-1}+p_{n-2}x^{n-2}+ \dots +p_2x^2+p_1x+p_0 \text{ [11].}$$

where p_i is an element in GF(2). Whenever a polynomial degree, in the intermediate inversion calculations, equals n, the polynomial should be reduced (XORed) by $p(x)$. Lets use the notation $\|p(x)\|$ to represent the degree of a polynomial $p(x)$. If, for example, $\|r(x)\| = \|p(x)\|$ then r is replaced by $p \oplus r$. Note that in some cases $\|r(x)\| = \|p(x)\|$ while $r < p$ (recall that r is the binary representation of $r(x)$ as it is p for $p(x)$). These cases restrict the comparison of r to 2^n to indicate if $r(x)$ needs to be reduced by $p(x)$ ($r=p \oplus r$), which means that $r(x)$ is compared with x^n (represented by 2^n) and not $p(x)$.

6.2.2 Montgomery Inverse in GF(2^n)

The GF(2^n) Montgomery inverse of $a(x)x^m mod\ p(x)$ is $a(x)^{-1}x^m mod\ p(x)$ [11]. The Montgomery factor 2^m of GF(p) is replaced by x^m in GF(2^n), which is exactly equal to 2^m in a binary representation [7,11,35]. The restriction on m is the same as GF(p), it should not be less than the number of bits (n), i.e., $m \geq n$ [1]. The elements of GF(p) and GF(2^n) are represented using similar binary data structure. Element a for both GF(p) and GF(2^n) is resembled by ($a_{n-1}\ a_{n-2}\ \dots\ a_2\ a_1\ a_0$) while $p=(p_{n-1}\ p_{n-2}\ \dots\ p_2\ p_1\ p_0)$ for GF(p) and $p=(1\ p_{n-1}\ p_{n-2} \dots p_2\ p_1\ p_0)$, polynomial $p(x)$ for GF(2^n) [11]. Our adjusted binary GF(2^n)

Montgomery inverse (MonInv) procedure consists in a GF(2^n) AlmMonInv and a GF(2^n) CorPh routines as outlined in Figure 6.2.

GF(2^n) AlmMonInv Algorithm

Input: $a2^m \in$ GF(2^n) & p; (p=irreducible polynomial & $m \geq n$)
Output: result$\in$[1, p-1] & k (result=$a^{-1}2^{k-m} mod\ p$ & $n<k<2n$)
1. u = p; v = $a2^m$; r = 0; s = 1; k = 0
2. While (v > 0)
3. if u_0 = 0 then {u = u/2; s = 2s}
4. else if v_0 = 0 then {v = v/2; r = 2r}
5. else if u>v then {u = (u$\oplus$v)/2; r = r$\oplus$s; s = 2s}
6. else {v = (u$\oplus$v)/2; s = r$\oplus$s; r = 2r}
7. k=k+1
8. if r $\geq 2^{n+1}$ (||r|| > ||p||) then {result = 2$p\oplus$r}
9. else if r $\geq 2^n$ (||r|| = ||p||) then {result = $p\oplus r$}
10. else result = r

GF(2^n) CorPh Algorithm

Input: $r, p, m,$ & cowherd r & k from *AlmMonInv*
Output: result; Where result = $a^{-1}2^m$ (mod p)
11. $j = 2m-k$
12. While $j > 0$
13. $r = 2r$
14. if r $\geq 2^n$ (||r|| = ||p||) then {r = $p\oplus r$}
15. $j = j-1$
16. result = r

Figure 6.2 GF(2^n) Montgomery inverse algorithm in its binary representation

For more clarification of the GF(2^n) MonInv computation, see the numerical example in Figure 6.3. It takes as inputs the polynomial $a(x)=x^3+1$, represented into Montgomery domain as $a(x)x^9 mod\ p(x) = x^4+x^2$ ($m=9 \geq n=5$), and $p(x)=x^5+x^2+1$ as the irreducible polynomial. All the data are shown in its binary representation ($a=1001$, $a2^m=10100$, and $p=100101$). The example (Figure 3) follows the convention:

condition met ➔ *affected registers with their updated values*.

The AlmMonInv routine generates the results $a^{-1}2^{k-m}=1000$, and $k=(10)_{10}$ (k is a normal decimal counter), which are used by the CorPh to provide the Montgomery inverse result of *111* (x^2+x+1 in the polynomial form). The reader is referred to Appendix B for checking the result of this example.

Observe on Figure 6.2 the several hardware operations applied to compute the MonInv in finite field GF(2^n). For example, the division and multiplication by two are equivalent to one bit shifting the binary representation of polynomials to the right and to the left, respectively. Checking the condition of step 5, if $u>v$, is performed through normal (borrow propagate) subtraction and test of the borrow-out bit. The subtraction result is completely discarded, only the borrow bit is observed. If the borrow bit is zero, then $u(x)$ is greater than $v(x)$. Similarly, the conditions of steps 8, 9, and 14 demands normal subtraction. However, the subtraction this time is used to check $\|r(x)\|$, which requires the availability of x^n (2^n in binary) saved in a register.

GF(2ⁿ) AlmMonInv Numerical Example	***GF(2ⁿ) CorPh Numerical Example***
$a = 1001 \in$ GF(2^5), $p=100101$, $m=9$, $n=5$	$p=100101$, $m=9$, $n=5$
$a2^m \bmod p = 10100 \in$ GF(2^6) (a in Montgomery domain)	$r = 1000 \in$ GF(2^6), $k=10$ (from AlmMonInv)
$u = p = 100101$, $v = a2^m = 10100$, $s = 1$, $r = k = 0$	$j = 8$
$v_0 = 0 \Rightarrow v = 1010$, $r = 0$, $k=1$	$r = 10000$, $j = 7$
$v_0 = 0 \Rightarrow v = 101$, $r = 0$, $k=2$	$r = 100000$, $\|r\| = \|p\| \Rightarrow r = 101$, $j = 6$
$u > v \Rightarrow u = 10000$, $r = 1$, $s = 10$, $k=3$	$r = 1010$, $j = 5$
$u_0 = 0 \Rightarrow u = 1000$, $s = 100$, $k=4$	$r = 10100$, $j = 4$
$u_0 = 0 \Rightarrow u = 100$, $s = 1000$, $k=5$	$r = 101000$, $\|r\| = \|p\| \Rightarrow r = 1101$, $j = 3$
$u_0 = 0 \Rightarrow u = 10$, $s = 10000$, $k=6$	$r = 11010$, $j = 2$
$u_0 = 0 \Rightarrow u = 1$, $s = 100000$, $k=7$	$r = 110100$, $\|r\| = \|p\| \Rightarrow r = 10001$, $j = 1$
$v > u \Rightarrow v = 10$, $s = 100001$, $r = 10$, $k=8$	$r = 100010$, $\|r\| = \|p\| \Rightarrow r = 111$, $j = 0$
$v_0 = 0 \Rightarrow v = 1$, $r = 100$, $k=9$	
$u = v \Rightarrow v = 0$, $r = 1000$, $s = 100101$, $k=10$	$\therefore$GF(2^n) MonInv of $10100 = 111$ ($a^{-1}2^m$);
$\|r\|<\|p\| \Rightarrow$ result = r	Where $m=9$ & $n = 5$

Figure 6.3 GF(2^n) MonInv computation numerical example

6.2.3 Multi-Bit Shifting

A further improvement on the GF(2^n) MonInv algorithm is performed based on a multi-bit shifting method making it similar to the GF(p) algorithm in Figure 6.1. After comparing different multi-bit shifting distances applied to reduce the number of iterations of the GF(p) MonInv algorithm, the best maximum distance for multi-bit shifting was found to be three, as clarified in Section 5.3. The GF(2^n) inverse algorithm (Figure 6.2) is mapped to hardware involving multi-bit shifting and making it very similar to the GF(p) algorithm (Figure 6.1) as shown in Figure 6.4. Note that x^n is required in the GF(2^n) algorithm as an extra variable that is needless in the GF(p) MonInv algorithm; x^n (2^n) is saved in register y in MHW-Alg3 (used in step 9), and in register s in MHW-Alg4 (used in step 16.1). These registers (y in MHW-Alg3 and s in MHW-Alg4) are not changed during the algorithms execution.

For both GF(p) and GF(2^n) MonInv hardware algorithms (Figure 6.1 and Figure 6.4, respectively), the AlmMonInv algorithm needs to finish its computation completely before the CorPh begins processing. This data dependency allows the use of the same hardware to execute both algorithms, i.e., both the AlmMonInv and CorPh. The algorithms are implemented in the unified and scalable hardware architecture as described in the following section.

MHW-Alg3:GF(2^n) Multi-Bit Shifting AlmMonInv HW Algorithm

Registers: u, v, r, s, x, y, z, & p (all registers hold n_{max} bits)
Input: $a2^m$, $2^n \in [1, p-1]$ (p=irreducible polynomial & $m \geq n$)
Output: result $\in [1, p-1]$ & k (result=$a^{-1}2^{k-m} \bmod p$ & $n<k<2n$)
1. u = p; v = $a2^m$; r = 0; s = 1; x = 0; y = 2^n; z = 0; k = 0
2. if($u_2u_1u_0$=000)then{u=ShiftR(u,3);s=ShiftL(s,3);k=k+3};goto 8
2.1. if($u_2u_1u_0$=100)then{u=ShiftR(u,2);s=ShiftL(s,2);k=k+2};goto 8
2.2. if($u_2u_1u_0$=110)then{u=ShiftR(u,1);s=ShiftL(s,1)};goto 7
3. if($v_2v_1v_0$=000)then{v=ShiftR(v,3);r=ShiftL(r,3);k=k+3};goto 8
3.1. if($v_2v_1v_0$=100)then{v=ShiftR(v,2);r=ShiftL(r,2);k=k+2};goto 8
3.2. if($v_2v_1v_0$=110)then{v=ShiftR(v,1);r=ShiftL(r,1)};goto 8
4. S1 = Subtract (u, v); x = v⊕u; z = r⊕s
5. if($S1_{borrow}$=0)then{u=ShiftR(x,1); r=z; s=ShiftL(s,1)};goto 7
6. s = z; v = ShiftR(x,1); r = ShiftL(r,1)
7. $k = k + 1$
8. if (v $\neq$ 0) go to step 2
9. x = p⊕r ; z = $2p$⊕r ; S1 = Subtract (y,x); S2 = Subtract (y,z)
10. if($S1_{borrow}$=0)then{result=x}
10.1 else if($S2_{borrow}$=0)then{result=z}
10.2 else {result = r}

MHW-Alg4:GF(2^n) Multi-Bit Shifting CorPh HW Algorithm

Registers: r, u, v, s, x, y, z, & p (all registers hold n_{max} bits)
Input: r, p, m, 2^n & k;
Where r ($r=a^{-1}2^{k-m} \bmod p$)& k from *HW-Alg3*
Output: result; Where result = $a^{-1}2^m \pmod p$.
11. $j = 2m-k-1$; x = 0; y = 0; z = 0
12. $v = 2p$; $u = 3p$; s = 2^n
13. While $j > 0$
14. if $j = 1$ then {r = ShiftL(r,1); j=j-1}
15. else {r = ShiftL(r,2); j=j-2}
16. x = p⊕r ; y = u⊕r ; z = u⊕r
16.1 S1=Subtract(s,x);S2=Subtract(s,y);S3=Subtract(s,z)
17. if ($S3_{borrow}$ = 0) then {r = z}
18. else if ($S2_{borrow}$ = 0) then {r = y}
19. else if ($S1_{borrow}$ = 0) then {r = x}
20. result = r

Figure 6.4 Montgomery inverse hardware algorithm for GF(2^n)

6.3 Unified and Scalable Inverter Architecture

Taking into account the amount of effort, time, and money that must be invested in designing an inverter, a scalable and unified architecture which can perform arithmetic in two commonly used algebraic finite fields, GF(p) and GF(2^n) [11,35], is clearly advantageous. In this section, we present the hardware design of a Montgomery inverse architecture that can be used for both types of fields following the design methodology presented in [27]. The proposed unified architecture is obtained from the scalable architecture given in [27] but with some modifications, which slightly increased the longest path propagation delay and area. The scalable GF(p) Montgomery inverse architecture presented in [27] consisted in two main units, a non-scalable memory unit and a scalable computing unit. The memory unit is not scalable because it has a limited storage defined by the value of n_{max}. The data values of a and p are first loaded in the memory unit. Then, the computing unit read/write (modify) the data using a word size of w bits. The computing unit is completely scalable. It is designed to handle *w* bits every clock cycle. The computing unit does not know the total number of bits, n_{max}, the memory is holding. It computes until the controller indicates that all operands' words were processed. Note that the actual numbers used may be way smaller than n_{max} bits. The user needs to identify the type of finite field his application needs at the beginning of the computation. An input signal *FSEL* (field select) is dedicated to tell the architecture weather GF(p) or GF(2^n) is the arithmetic domain for this particular inversion calculation.

The block diagram for the Montgomery inverter hardware is shown in Figure 6.5. The memory unit is connected to the computing unit components. The memory unit is not changed from what is presented in [27]. It contains a counter to compute variable *k* and eight first-in-first-out (FIFO) registers used to store the inversion algorithm's variables. All registers, *u*, *v*, *r*, *s*, *x*, *y*, *z* and *p*, are limited to hold at most n_{max} bits. Each FIFO register has its own reset signal generated by the controller. They have counters to keep track of *n* (the number of bits actually used by the application).

The computing unit is made of four hardware blocks, the add/subtract, shifter, data router, and controller block. The GF(p) add/subtract unit and the data router are the only components that need to be adjusted to make the inverter hardware unified for GF(p) and GF(2^n) finite fields.

The GF(p) add/subtract unit is originally built of two *w*-bit subtractors, a *w*-bit adder/subtractor, four flip-flops, one multiplexer, a *w*-bit comparator, and logic gates, as detailed in [27]. This unit is adjusted to operate for GF(2^n) by adding a set of *3w* parallel XOR gates used for steps 4 and 9 of MHW-Alg3 and step 16 of MHW-Alg4. The new add/subtract unit is shown in Figure 6.6. The signal *Control* lets the unit perform either two subtractions and one addition (step 4 of MHW-Alg), or three subtractions (step 16 of MHW-Alg2 and step 16.1 of MHW-Alg4). Three flip-flops are used to hold the intermediate borrow-bits of the subtractors and the carry-bit of the adder to implement the multi-precision operations. The fourth flip-flop is used to store a flag that keeps track of the comparison between *u* and *v*, which is used to perform step 8 of MHW-Alg and MHW-Alg3. The subtractors borrow-out bits are connected to the controller through signals that are useful only at the end of each multi-precision addition/subtraction operation. Subtractor1 borrow-out bit will affect the flow of the operation to choose either step 5 or step 6 of both MHW-Alg and MHW-Alg3. It is also essential in electing the result observed in step 10 of MHW-Alg and of MHW-Alg2. The three subtractors borrow-out bits ($S1_{borrow}$, $S2_{borrow}$, $S3_{borrow}$) are likewise necessary for selecting the correct solution of the 'if' condition to be one of the steps 17, 18, or 19, from the MHW-Alg2 and from the MHW-Alg4 algorithms.

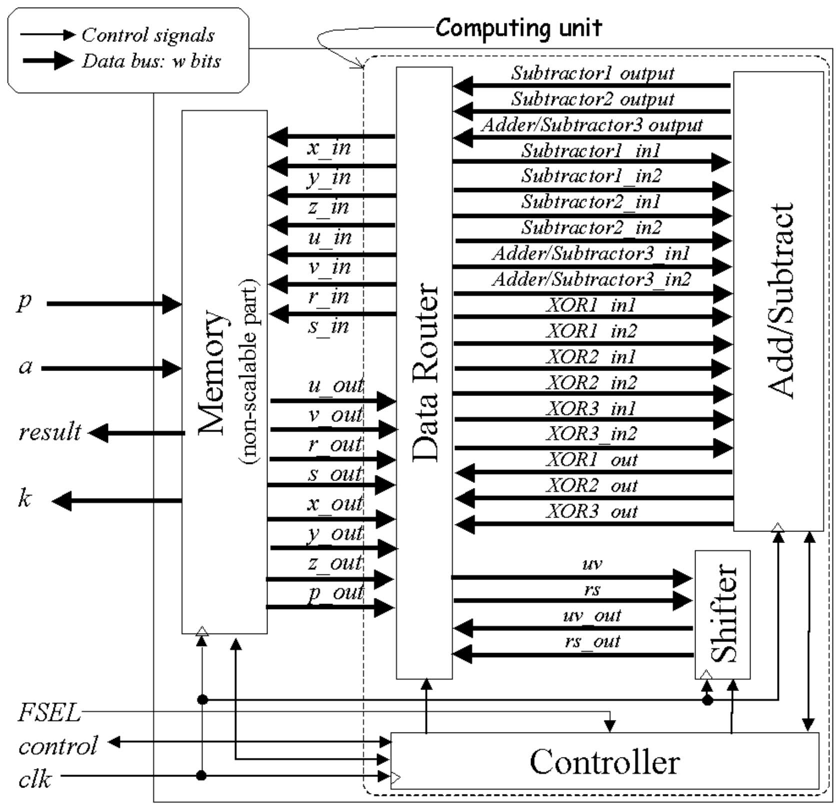

Figure 6.5 Scalable and unified inverter hardware

The shifter is made of two multiplexers and two registers with special mapping of some data bits, as shown in Figure 5.5. Depending on the controller signal *Distance*, the shifter acts as a one, two, or three-bit shifter. Two types of shifting operations are needed in the MHW-Alg and the MHW-Alg3 algorithms, shifting an operand (*u* or *v*) through the *uv* bus one, two, or three bits to the right, and shifting another operand (*r* or *s*) through the *rs* bus by a similar number of bits to the left. Shifting *u* or *v* is performed through

Register1, which is of size w-1 bits. For each word, all the bits of uv are stored in Register1 except for the least significant bit(s) to be shifted, it is (or they are) read out immediately as the most significant bit(s) of the output bus *uv_out*. Shifting r or s to the left is performed via Register2, which is of size w+3 bits similar to shifting uv but to the other direction. When executing the MHW-Alg2 or MHW-Alg4, the shifting is performed either to one or two bits to the left only, which is via MUX2 and Register2 ignoring MUX1 and Register1.

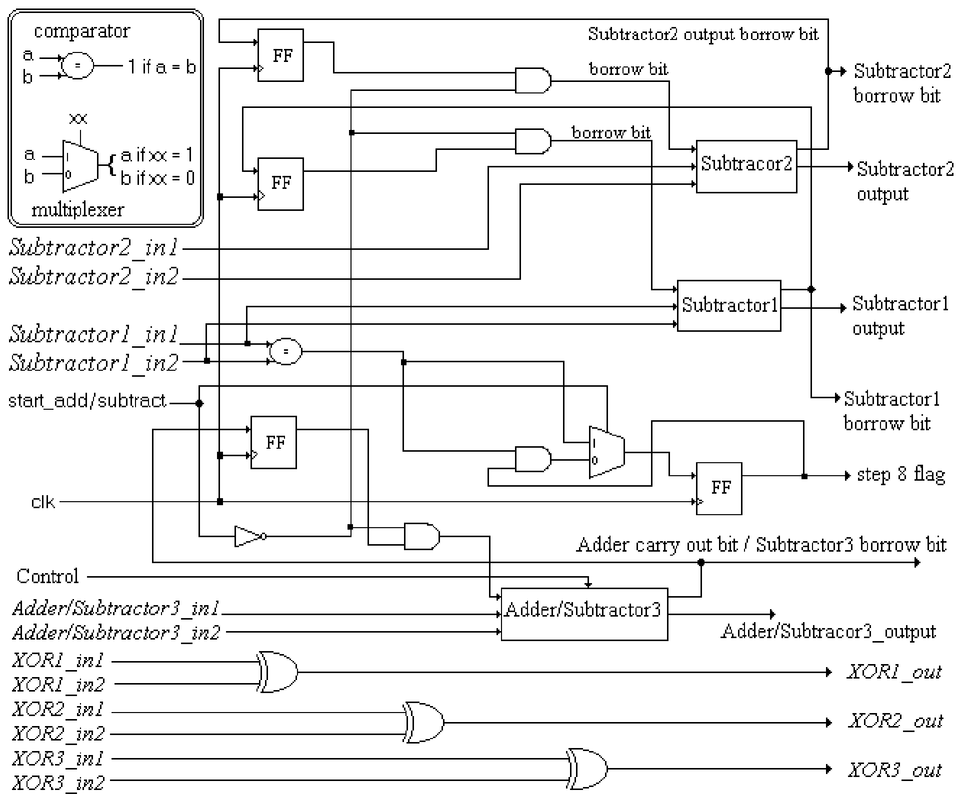

Figure 6.6 Add/Subtract unit of the scalable and unified hardware

The data router capabilities are extended to satisfy the unified architecture requirements. It interconnects the memory, add/subtract, and shifter units. The possible configurations of the data router are shown in Figure 6.7.

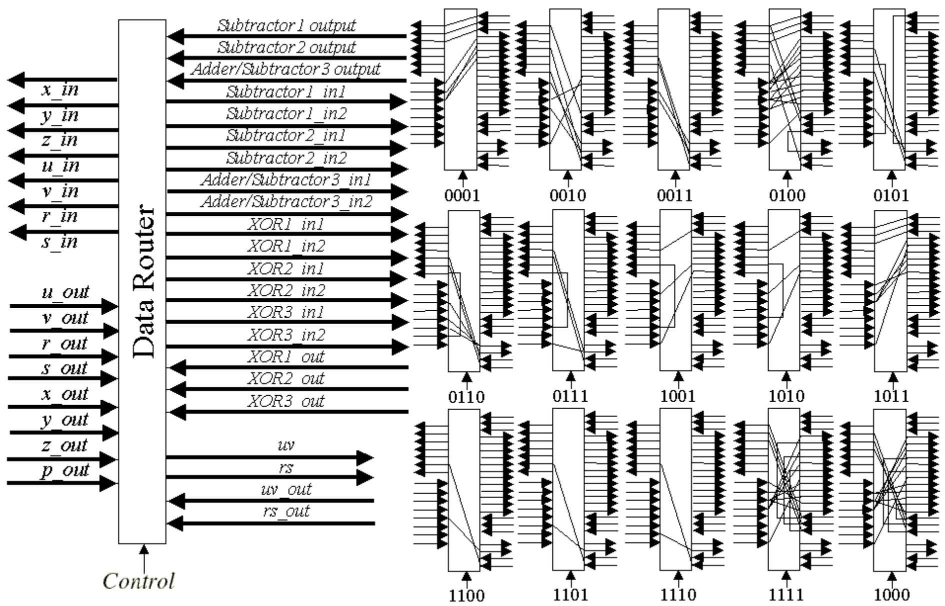

Figure 6.7 Data router configurations

6.4 Modeling and Analysis

The unified and scalable inverter was modeled and simulated in VHDL. Previously, a GF(p) fixed precision and other scalable inverter designs were also implemented in VHDL. All developed VHDL implementations of the scalable designs, including the new unified ones, have two main parameters, namely n_{max} and w. The fixed precision hardware, however, is parameterized by n_{max} only. Their area and speed are presented in this section. Also a reconfigurable hardware [35] that can perform the inversion in both GF(p) and GF(2^n) is considered in the comparison. As in the previous Chapters, we didn't define a specific architecture for the adders and subtractors to be used in our VHDL implementations. Thus, the synthesis tool chooses the best option in terms of area from its library of standard cells.

6.4.1 Area Comparison

The exact area of any design depends on the technology and minimum feature size. For technology independence, we use the equivalent number of NOT-gates as an area measure [14]. A CAD tool from Mentor Graphics (Leonardo) was used like Section 3.5. In general, Leonardo takes the VHDL design code and provides a synthesized model with its area and longest path delay. The target technology is a $0.5\mu m$ CMOS defined by the 'AMI0.5 fast' library provided in the ASIC Design Kit (ADK) from the same Mentor Graphics Company [19]. It has to be mentioned here that the ADK is developed for educational purposes and cannot be thoroughly compared to technologies adopted for marketable ASICs. It however, provides a framework to contrast all scalable hardware designs together and with the fixed precision one. The sizes of the designs are compared in Figure 6.8. Observe that the fixed precision design has a better area if the maximum number of bits used (n_{max}) is small which is useless in cryptographic applications [11]. The unified designs are larger than the GF(p) ones with a calculated average of 8.4% more hardware area.

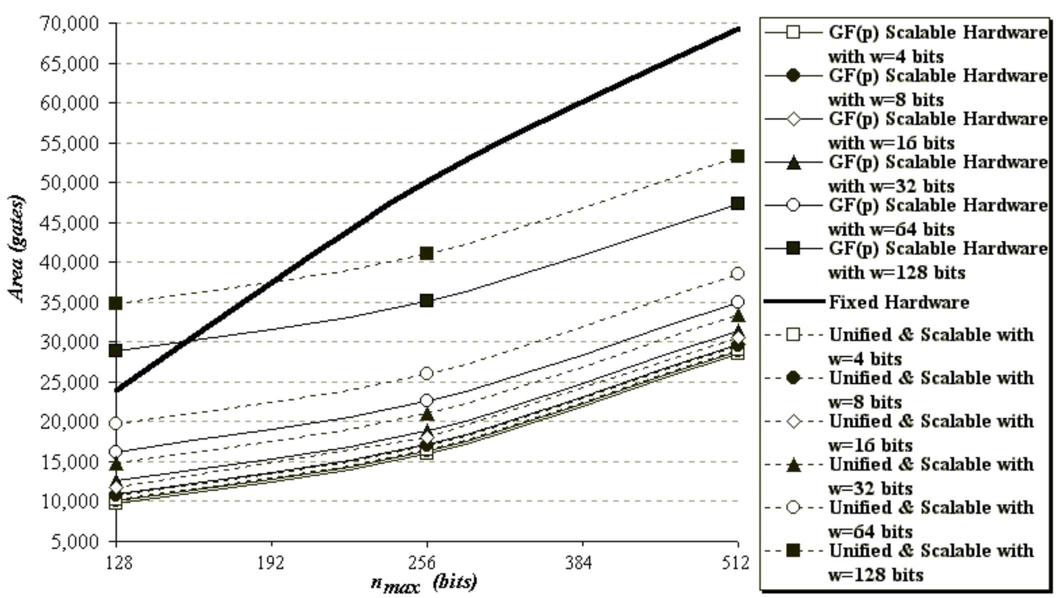

Figure 6.8 Area comparison

The areas of the unified designs were also compared with the reconfigurable hardware [35], but not shown in Figure 6.8. The reconfigurable design core is built of 880,000 devices [35]. Assume a device is corresponding to a transistor and our NOT-gate is equivalent to two transistors [14], so the reconfigurable hardware core is equivalent to 440,000 gates, which is greater eight times than the largest unified hardware.

6.4.2. Speed Comparison

The total computation time is a product of the number of clock cycles the algorithm takes and the clock period of the final VLSI implementation. This clock period changes with the value of w in the unified and scalable hardware, and changes with the value of n_{max} in the fixed precision hardware. This is because $w = n_{max}$ in the fixed precision hardware. All VHDL coded designs clock cycle periods are generated automatically by Leonardo, which determines the longest path delay of the hardware circuits. The clock period of the reconfigurable design is set to 20 nanoseconds/cycle (it operates at 50MHz clock frequency) [35].

The number of clock cycles depends completely on the data and the algorithm. A probabilistic study described in Chapter 5 is used to estimate the average number of clock cycles. For the fixed precision design, the average number of clock cycles equal to

$$C_f = 1.525n.$$

For all scalable designs, the function of the average number of clock cycles would be

$$C_s=(2.4125n+1)\lceil n/w \rceil,$$

which is exactly the same for the unified designs presented in this paper. Hence, adjusting the scalable designs to be unified did not change the number of clock cycles of the inverse computation. However, the clock cycle period of the unified designs increased slightly, making the total computation time of the unified hardware worse than what was given in Chapter 6. The number of clock cycles for the reconfigurable hardware [35] to complete the inversion process is reported as

$$C_r=14.5n.$$

Similar to the GF(p) scalable hardware of Chapter 6, the unified and scalable hardware can have several designs for each n_{max} depending on w. For example, Figure 6.9

shows the delay of several designs of the unified and scalable hardware compared to the reconfigurable, GF(p) scalable, and fixed precision hardware designs, all modeled for $n_{max}=512$ bits. Observe how the actual data size (n) plays a big role on the speed of the designs. In other words, as n reduces and w is small, the number of clock cycles decrease significantly, which considerably reduces the overall computing time of all scalable designs (including the unified ones) compared to the fixed precision and reconfigurable ones. This is a major advantage of the scalable hardware over the fixed precision [27] and reconfigurable ones.

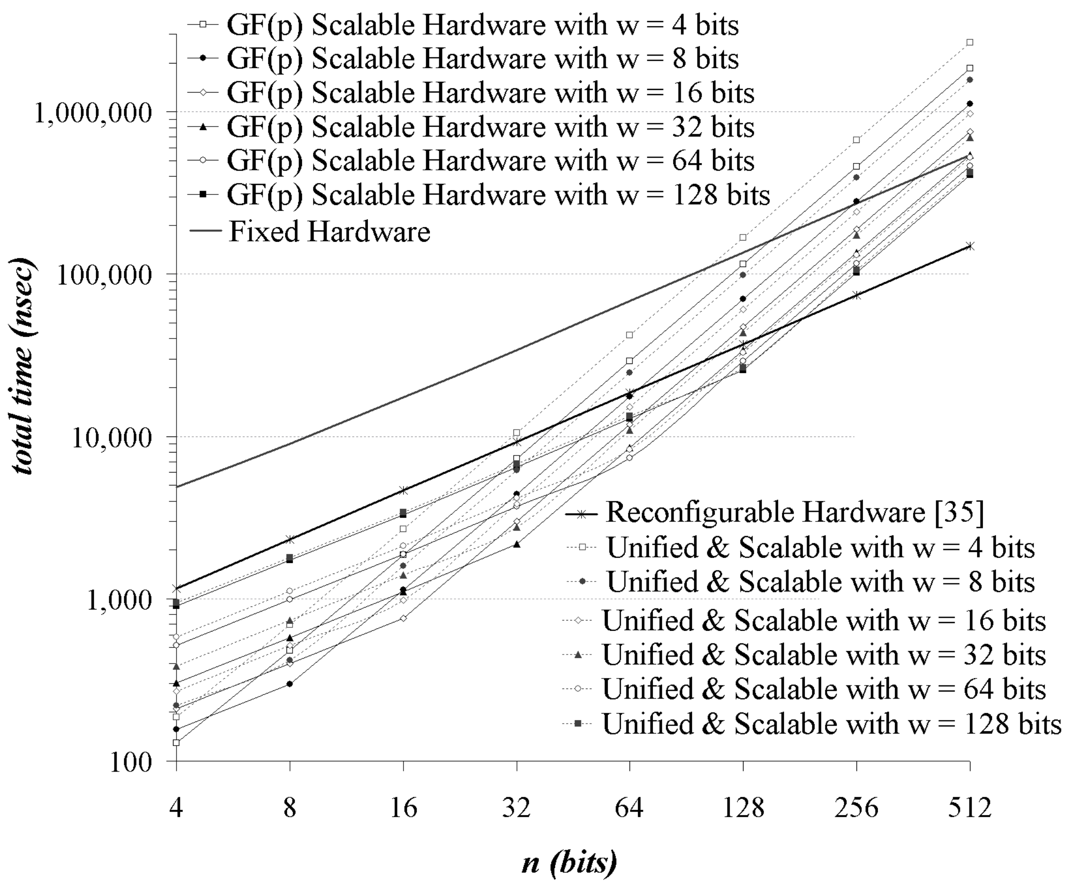

Figure 6.9 Delay comparison of designs with n_{max} = 512 bits

The new unified designs when compared to the GF(p) scalable ones have very similar characteristics. Overall, it computes the inverse in an average of 19.8% more time than the GF(p) desings (Chapter 5). Another observation from Figure 6.9 is that the unified designs are faster than the fixed precision one as long as:

$$n < \begin{cases} (\log_2 w) n_{max} / 8 & \text{when } w < n_{max} / 8 \\ n_{max} & \text{when } w \geq n_{max} / 8 \end{cases}$$

which is generalized for all different n_{max} designs after performing several experimental tests, namely for n_{max}= *32, 64, 128, 512* and *1024* bits. Figure 6.9 also shows that the unified designs are comparable to the reconfigurable one giving better performance while:

$$n < \begin{cases} (\log_2 w)\, n_{max}/32 & \text{when } w < n_{max}/8 \\ (\log_2 w)\, n_{max}/25 & \text{when } w \geq n_{max}/8 \end{cases}$$

Consider the case when $n=n_{max}=512$ bits in Figure 6.9, the unified design with $w=64$ bits has almost the same speed as the fixed precision one, but the ones with $w=128$ bits remain faster. In fact, as w gets bigger the total time decreases, which is also true when comparing among the different unified designs while $n \geq w$, as also proven before in Chapter 5 for the GF(p) scalable designs. Whenever $n<w$ considering the unified and scalable designs, the scalability advantage of these designs is reduced since the number of words to be processed reached its lower limit, but still the unified and scalable designs are faster than the fixed precision one.

6.5 Summary

This Chapter presents a scalable inverter for both finite fields GF(p) and GF(2^n) in a unified hardware module that applies the design approach proposed in [27]. The primary contribution of this research is to show that it is possible to design a unified hardware without compromising scalability and area efficiency. The unified inverter hardware is built of two main units, a memory unit and a computing unit. The memory unit defines the upper bound of the number of bits that the hardware can handle. The computing unit is the real scalable hardware, it is designed to fit in constrained areas and perform the computation of numbers in a repetitive way. Our analysis shows that as the word size of the scalable computing unit reduces, the hardware area decreases and the possible clock frequency increases.

The comparisons with other designs show that this unified and scalable structure is very attractive for cryptographic systems, particularly for ECC because of its need for modular inversion of large numbers in both finite fields GF(p) and GF(2^n). The experimental work shows that the scalable and unified design can be faster or competitve with other alternatives using significantly less area.

7 CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this thesis, several contributions to the Montgomery modular inverse computation in hardware have been achieved as summarized below:

- We considered the GF(p) Montgomery inverse algorithms and proposed modifications that are applicable for hardware implementations.
- We proposed new scalable designs to compute the Montgomery inverse, which consists in a hardware module that fits on constrained areas and still handle operands of any size. In order to have long-precision calculations, the module works on small precision words. The word-size, which the module operates, can be selected based on the area and performance requirements. The upper limit on the operand precision is dictated only by the available memory to store the operands and internal results. The scalable module is in principle capable of performing infinite-precision Montgomery inverse computation of an integer, modulo a prime number.
- We adopted multi-bit shifting technique to the Montgomery inverse algorithms, which reduced the number of iterations significantly and speeded up the entire inversion process with small amount of extra hardware.
- We proposed a fast Montgomery inverse method by introducing a new correction phase for a previously proposed almost Montgomery inverse algorithm. This approach eliminated the need for a multiplier in the inversion process, using nearly the same hardware designed for the almost Montgomery inverse algorithm.
- We proposed a scalable and unified architecture for a Montgomery inverse hardware that operates in both GF(p) and GF(2^n) fields. We adjust a GF(2^n) Montgomery inverse algorithm to accommodate the hardware features and benefit from the multi-bit shifting method making it very similar to the proposed best design of GF(p) inversion hardware. A comparison of our scalable and unified design with a reconfigurable hardware [35] shows that the scalable design saves a lot of area and operates in comparable speed. We also compared all scalable

designs with fully parallel ones based on the same basic inversion algorithm. All scalable designs consumed less area and in general showed better performance than the fully parallel ones, which concluded that the scalable design a very efficient solution for the long precision numbers Montgomery modular inverse computation.

7.2 Future Work

Several future research works may be considered as a continuation on this study.

- The registers of the non-scalable part could be modified to incorporate the bit-shifting operation. This way, the registers would have the capability to shift all their bits at once inside the memory. This feature will reduce the shifting operation delay from ($\lceil n/w \rceil + 1$) clock cycles to one.
- The longest path of the inversion design is through the adders. Other adders, besides carry-look-ahead, could be used in the designs and give a more definite picture of its impact on the overall performance.
- The proposed Montgomery inverse algorithms are performing two main operations (shifting and adding). These operations are performed separately in different clock cycles. It would be interesting to investigate if the shifting operation can be merged with addition and further speedup the inversion process.
- The non-scalable part (memory unit) is not synthesized, which needs its components (registers and counters) to be modeled specifically for each w value. In other words, the non-scalable part is built in a parametrizable manner to let it be flexible for any w value. This flexibility prevented it from being synthesized. This requests that this non-scalable part is to be redesigned in different modules structures. Each module is built specifically for every w value such as $w= 4, 8, 16, 32, 64,$ and 128 bits. Every specific non-scalable module will be connected to the scalable part and synthesized together, which is promising to give more realistic area and frequency values.
- The non-scalable part is the limiting part, which will limit the hardware capability. If this limit is exceeded even by one bit the non-scalable part is to be replaced.

Instead of replacing it, the non-scalable part could be implemented separately on a programmable hardware, such as an FPGA, which is reprogrammed whenever any change is to take place, while the scalable part remains the same.

BIBLIOGRAPHY

[1] Savas, and Koç, "The Montgomery Modular Inverse – Revisited", *IEEE Trans. on Computers*, 49(7):763-766, July 2000.

[2] Kobayashi, and Morita, "Fast Modular Inversion Algorithm to Match Any Operation Unit", *IEICE Trans. Fundamentals*, E82-A(5):733-740, May 1999.

[3] Kaliski, "The Montgomery Inverse and its Applications", *IEEE Trans. on Computers*, 44(8):1064-1065, August 1995.

[4] Rivest, Shamir, and Adleman, "A Method for Obtaining Digital Signature and Public-Key Cryptosystems", *Comm. ACM*, 21(2):120-126, February 1978.

[5] Diffie, and Hellman, "New Directions on Cryptography", *IEEE Trans. on Information Theory*, 22:644-654, November 1976.

[6] Tenca, and Koç, "A Scalable Architecture for Montgomery Multiplication", *In Cryptographic Hardware and Embedded Systems*, no. 1717 in Lecture notes in Computer Science, Springer, Berlin, Germany, 1999.

[7] Savas, Tenca, and Koç, "A Scalable and Unified Multiplier Architecture for Finite Fields GF(p) and GF(2^k)", *In Cryptographic Hardware and Embedded Systems*, Lecture notes in Computer Science. Springer, Berlin, Germany, 2000.

[8] Tenca, Todorov, and Koç, "High-Radix Design of a Scalable Modular Multiplier", *Workshop on Cryptographic Hardware and Embedded Systems, CHES 2001*, Paris, France, May 14-16 2001.

[9] Chung, Sim, and Lee, "Fast Implementation of Elliptic Curve Defined over GF(p^m) on CalmRISC with MAC2424 Coprocessor", *Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000*, Massachusetts, August 2000.

[10] Atsuko Miyaji, "Elliptic Curves over F_P Suitable for Cryptosystems", *Advances in cryptology- AUSCRUPT'92*, Australia, December 1992.

[11] Blake, Seroussi, and Smart, *Elliptic Curves in Cryptography*, Cambridge University Press: New York, 1999.

[12] Hankerson, Hernandez, and Menezes, "Software Implementation of Elliptic Curve Cryptography Over Binary Fields", *Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000*, Massachusetts, August 2000.

[13] Tocci, R. J. and Widmer, N. S., "*Digital Systems: Principles and Applications*", Eighth Edition, Prentice-Hall Inc., New Jersey, 2001.

[14] Ercegovac, M. D., Lang, T., and Moreno, J. H., *Introduction to Digital System*, John Wiley & Sons, Inc., New York, 1999.

[15] Montgomery, P.L., "Modular Multiplication Without Trail Division", *Mathematics of Computation*, 44(170): 519-521, April 1985.

[16] Naofumi Takagi, "Modular Inversion Hardware with a Redundant Binary Representation", *IEICE Transactions on Information and Systems*, E76-D(8): 863-869, August 1993.

[17] Guo, J.-H., and Wang, C.-L., "Hardware-Efficient Systolic Architecture for Inversion and Division in GF(2^m)", *IEE Proceedings: Computers and Digital Techniques*, 145(4): 272-278, July 1998.

[18] Choudhury, P. Pal., and Barua, R., "Cellular Automata Based VLSI Architecture for Computing Multiplication and Inverses in GF(2^m)", *Proceedings of the 7th IEEE International Conference on VLSI Design*, Calcutta, India, January 5-8 1994.

[19] Mentor Graphics Co., *ASIC Design Kit*, http://www.mentor.com/partners/hep/AsicDesignKit/dsheet/ami05databook.html

[20] Hasan, M. A., "Efficient Computation of Multiplicative Inverse for Cryptographic Applications", *Proceeding of the 15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, June 11-13 2001.

[21] Guo, J.-H., and Wang, C.-L., "Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in GF(2^m)", *IEEE Trans. on Computers*, 47(10):1161-1167, October 1998.

[22] Fenn, S. T. J., Benaissa, M., and Taylor, D., "GF(2^m) Multiplication and Division Over the Dual Basis", *IEEE Trans. on Computers*, 45(3):319-327, March 1996.

[23] Wang, C. C., Truong, T. K., Shao, H. M., Deutsch, L. J., Omura, J. K., and Reed, I. S., "VLSI Architectures for Computing Multiplications and Inverses in GF(2^m)", *IEEE Trans. on Computers*, C-34(8):709-717, August 1985.

[24] Feng, G.-L., "A VLSI Architecture for Fast Inversion in GF(2^m)", *IEEE Trans. on Computers*, 38(10):1383-1386, October 1989.

[25] Kovac, M., Ranganathan, N. and Varanasi M., "SIGMA: A VLSI Systolic Array Implementation of Galois Field GF(2^m) Based Multiplication and Division Algorithm", *IEEE Trans. on VLSI*, 1(1):22-30, March 1993.

[26] Charles J. Stone, *A course in probability and statistics*, Duxbury Press, Belmont, 1996.

[27] A. A. Gutub, A. F. Tenca, and C. K. Koç, "Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation", *ISVLSI 2002 - IEEE Computer Society Annual Symposium On VLSI*, Pittsburgh, Pennsylvania, April 25-26 2002.

[28] Miyaji A., "Elliptic Curves over F_P Suitable for Cryptosystems", *Advances in cryptology- AUSCRUPT'92*, Australia, December 1992.

[29] Stallings, W. *Cryptography and Network Security: Principles and Practice*, Second Edition, Prentice Hall Inc., New Jersey, 1999.

[30] Okada, Torii, Itoh, and Takenaka, "Implementation of Elliptic Curve Cryptographic Coprocessor over GF(2^m) on an FPGA", *Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000*, Massachusetts, August 2000.

[31] Orlando, and Paar, "A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m)", *Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000*, Massachusetts, August 2000.

[32] Stinson, D. R., *Cryptography: Theory and Practice*, CRC Press, Boca Raton, Florida, 1995.

[33] Paar, Fleischmann, and Soria-Rodriguez, "Fast Arithmetic for Public-Key Algorithms in Galois Fields with Composite Exponents", *IEEE Transactions on Computers*, 48(10), October 1999.

[34] Michener, J. R., and Mohan, S. D., "Internet Watch: Clothing the E-Emperor", *Computer – Innovative Technology for Computer Professionals, IEEE Computer Society*, 34(9):116-118, September 2001.

[35] Goodman, J. and Chandrakasan, A. P., "An Energy-Efficient Reconfigurable Public-Key Cryptogrphy Processor", *IEEE Journal of solid-state circuits*, 36(11):1808-1820, November 2001.

[36] Koc and Acar, "Montgomery multiplication in GF(2^k)", *Designs, Codes and Cryptography*, 14(1):57-69, April 1998.

[37] D.E. Knuth, *The Art of Computer Programming – Seminumerical Algorithms*, 2nd ed. Vol. 2, Reading, MA : Addison-Wesley, 1981.

[38] Tudor Jebelean, "Systolic Algorithms for Long Integer GCD Computation", *CONPAR 94 - VAPP VI, Third Joint International Conference on Vector and Parallel Processing*, Linz, Austria, September 6-8, 1994, Proceedings. Lecture Notes in Computer Science 854, pages 241-252, Springer, 1994.

[39] Kung, H. T., "Why Systolic Architectures?", *Computer*, 15:37-46, 1982.

[40] Menezes, A.J., P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, Florida, 1996.

[41] Schneier, B., *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, New York, 2nd edition, 1996.

APPENDICES

A THE EXTENDED EUCLIDEAN ALGORITHM

When we divide one integer by another (nonzero) integer we get an integer *quotient* (the "answer") plus a *remainder* (generally a rational number). For instance,

13/5 = 2 ("the quotient") + 3/5 ("the remainder").

We can rephrase this division, totally in terms of integers, without reference to the division operation:

$$13 = 2 \times 5 + 3$$

Note that this expression is obtained from the one above it by multiplying both sides of the equation by the divisor *5*.

If *a* and *b* are positive integers, there exist unique non-negative integers *q* and *r* such that :

$$a = q \times b + r \text{ , where } 0 \le r < b.$$

q is called the *quotient* and *r* the *remainder*.

The *greatest common divisor* of integers *a* and *b*, denoted by *gcd(a,b)*, is the largest integer that divides (*remainder = 0*) both *a* and *b*. So, for example:

gcd(15, 5) = 5, gcd(7, 9) = 1, gcd(12, 9) = 3, gcd(81, 57) = 3.

The *gcd* of two integers can be found by repeated application of the division algorithm, this is known as the *Euclidean Algorithm* [11]. In this algorithm, the divisor is repeatedly divided by the remainder until the remainder of this operation is 0. The *gcd* is the last non-zero remainder in this algorithm.

The Euclidean Algorithm:

Inputs: integers *a*, *b*.
Output: *gcd(a, b)*
1. while $b \neq 0$:
2. $r = a \bmod b$
3. $a = b$
4. $b = r$
5. return *gcd(a, b) = a*

The following example shows the algorithm. Finding *gcd(81,57)* by the Euclidean Algorithm:

$$81 = 1 \times 57 + 24$$
$$57 = 2 \times 24 + 9$$
$$24 = 2 \times 9 + 6$$
$$9 = 1 \times 6 + 3$$
$$6 = 2 \times 3 + 0.$$

It is well known [11] that if the $gcd(a, b) = r$ then there exist integers u and s such that:

$$u \times a + s \times b = r$$

By reversing the steps in the Euclidean Algorithm, it is possible to find these integers u and s. We shall do this with the above example:

Starting with the next to last line, we have:

$$3 = 9 - 1 \times 6$$

From the line before that, we see that $6 = 24 - 2 \times 9$, so:

$$3 = 9 - 1 \times (24 - 2 \times 9) = 3 \times 9 - 1 \times 24$$

From the line before that, we have $9 = 57 - 2 \times 24$, so:

$$3 = 3 \times (57 - 2 \times 24) - 1 \times 24 = 3 \times 57 - 7 \times 24$$

And, from the line before that $24 = 81 - 1 \times 57$, giving us:

$$3 = 3 \times 57 - 7 \times (81 - 1 \times 57) = 10 \times 57 - 7 \times 81$$

So we have found $u = -7$ and $s = 10$.

The procedure we have followed above is a bit messy because of all the back substitutions we have to make. It is possible to reduce the amount of computation involved in finding u and s by doing some auxiliary computations as we go forward in the Euclidean algorithm (and no back substitutions will be necessary). This is known as the *Extended Euclidean Algorithm*.

The Extended Euclidean Algorithm:

Inputs: two non-negative integers a, b with $a \geq b$
Outputs: $d = gcd(a,b)$ and integers x,y such that $ax+by=d$
1. If $b = 0$ then set $d \leftarrow a, x \leftarrow 1, y \leftarrow 0$ and return (d,x,y)
2. Set $x_2 \leftarrow 1, x_1 \leftarrow 0, y_2 \leftarrow 0, y_1 \leftarrow 1$
3. While $b > 0$ do :
   3.1 $q \leftarrow \lfloor a/b \rfloor, r \leftarrow a - qb, x \leftarrow x_2 - qx_1, y \leftarrow y_2 - qy_1$
   3.2 $a \leftarrow b, b \leftarrow r, x_2 \leftarrow x_1, x_1 \leftarrow x, y_2 \leftarrow y_1, y_1 \leftarrow y$
4. Set $d \leftarrow a, x \leftarrow x_2, y \leftarrow y_2$ and return (d,x,y)

A.1 The Extended Euclidean Algorithm to obtain the inverse of a number *mod p*

Suppose we had to find the inverse of a number *mod p*. This turned out to be a difficult task (and not always possible) [11]. A number x has an inverse *mod p* (i.e., a number y so that $x.y = 1 \bmod p$) if and only if $gcd(x,p) = 1$, which implies that there exist integers u and s such that

$$u.x + s.p = 1.$$

But this says that $u.x = 1 + (-s)p$, or in other words, $u.x \equiv 1 \pmod p$. So, u (reduced *mod p* if need be) is the inverse of x *mod p*. The Extended Euclidean algorithm will give us a method for calculating u efficiently (note that in this application we do not care about the value of s, so we will simply ignore it.)

Let's take a numerical example to find the inverse of *15 mod 26*. We will number the steps of the extended Euclidean algorithm computation starting with step 0. The quotient obtained at step i will be denoted by q_i. As we carry out each step of the extended Euclidean algorithm, we will also calculate an auxiliary number, u_i. For the first two steps, the value of this number is given: $u_0=0$ and $u_1=1$. For the remaining steps, we recursively calculate $u_i = u_{i-2} - u_{i-1}\, q_{i-2} \pmod p$. Continue this calculation for one step beyond the last step of the algorithm to find the inverse. The algorithm starts by "dividing" p by x. If the last non-zero remainder occurs at step k, then if this remainder is 1, x has an inverse and it is u_{k+2}. (If the remainder is not 1, then x does not have an inverse.) Here are the steps of the numerical example to find the inverse of *15 mod 26*.

step 0:	$26 = 1 \times 15 + 11$	$u_0 = 0$
step 1:	$15 = 1 \times 11 + 4$	$u_1 = 1$
step 2:	$11 = 2 \times 4 + 3$	$u_2 = 0 - 1 \times 1 \bmod 26 = 25$
step 3:	$4 = 1 \times 3 + 1$	$u_3 = 1 - 25 \times 1 \bmod 26 = -24 \bmod 26 = 2$
step 4:	$3 = 3 \times 1 + 0$	$u_4 = 25 - 2 \times 2 \bmod 26 = 21$
		$u_5 = 2 - 21 \times 1 \bmod 26 = -19 \bmod 26 = 7$

Notice that $15 \times 7 = 105 = 1 + 4 \times 26 \equiv 1 \pmod{26}$.

A.2 The Binary Euclidean Algorithm

The Euclidean algorithm can be rephrased to a division-free approach by applying the following three observations:

1. If u and v are both even, gcd(u,v) = 2 gcd(u/2, v/2).
2. If u is even and v is odd, gcd(u,v) = gcd(u/2, v).
3. Otherwise both are odd, and gcd(u,v) = gcd(|u-v|/2, v). (Euclid's algorithm with a division by 2 since the difference of two odd numbers is even).

Here is the algorithm. It is especially efficient for operations on binary representations.

The Binary Euclidean Algorithm

```
Inputs:  integers u, v.
Output:  gcd(u, v)
1.    g = 1
2.    while u is even and v is even
 2.1      u = u/2 (right shift)
 2.2      v = v/2
 2.3      g = 2*g (left shift)
          now u or v (or both) are odd
3.    while u > 0
 3.1      if u is even, u = u/2
 3.2      else if v is even, v = v/2
 3.3      else
 3.4      t = |u-v|/2
 3.5      if u < v, then v = t else u = t
4.    return gcd(u, v)= g*v
```

This algorithm was extended as the *binary extended Euclidean algorithm* as presented in [37], which was further studied by Kaliski [3] who proposed the Montgomery inverse algorithm. Kaliski's Montgomery inverse algorithm worked as the basic algorithm of our research.

B GF(2^n) NUMERICAL EXAMPLE VERIFICATION

This Appendix details the computations and verifies the results used in the GF(2^n) MonInv numerical example shown in Figure 7.3. The example defines $m=9$ and $n=5$; where n is the degree of the irreducible polynomial and m (of the Montgomery constant 2^m) is any number as long as $m \geq n$. To simplify the arithmetic lets only use the binary representation of polynomials. The MonInv takes the inputs $a=1001$ and $p=100101$. However, a is represented into Montgomery domain as $a2^m$, which is calculated as follows:

$$a=1001$$

$$a2^m=a2^9=1001000000000$$

but since 1001000000000 needs to be reduced by p or a multiple of p until the number of significant bits of $a2^9$ is less or equal to n (the degree of polynomial $a(x)x^m mod\ p(x)$ should be less than the degree of the irreducible polynomial ($p(x)$)), so

$$a2^9 \oplus 2^7p=100100\ 0000000 \oplus 100101\ 0000000=10000000$$

and 10000000 also needs reduction

$$10000000 \oplus 2^2p = 10000000 \oplus 10010100 = 10100$$

So $a2^m mod\ p = a2^9 mod\ p = 1001000000000\ mod\ p \equiv 10100$

The GF(2^n) MonInv of 10100 = 111 = $a^{-1}2^m$, which can be verified similarly:

The MonInv numerical example (Figure 3) calculated that

$$a^{-1}2^9=111 \Rightarrow a^{-1} = 111/2^9.$$

Any congruent polynomial can be XORed with the irreducible polynomial, such as:

$$a^{-1}2^9=111 \equiv 111 \oplus 100101 = 100010 \Rightarrow a^{-1}2^8=10001$$

$$a^{-1}2^8=10001 \equiv 10001 \oplus 100101=110100 \Rightarrow a^{-1}2^6=1101$$

$$a^{-1}2^6=1101 \equiv 1101 \oplus 100101=101000 \Rightarrow a^{-1}2^3=101$$

$$a^{-1}2^3=101 \equiv 101 \oplus 100101=100000 \Rightarrow a^{-1}=100$$

To confirm this result:

$$a\ .\ a^{-1} mod\ p \text{ must equal to } 1$$

$$a\ .\ a^{-1}= 1001\ .\ 100 = 100100$$

$$100100\ mod\ p = 100100 \oplus 100101=1$$

which confirms that the GF(2^n) MonInv of 10100 is 111; where $m=9$ and $n=5$.