2005-05-04

Low Power Elliptic Curve Cryptography

Erdinc Ozturk
Worcester Polytechnic Institute

Low Power Elliptic Curve Cryptography

by

Erdinç Öztürk

A Thesis
Submitted to the Faculty
of the

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the
Degree of Master of Science
in

Electrical Engineering

by

April, 2004

Approved:

Dr. Berk Sunar Thesis Advisor ECE Department	Dr. Brian King Thesis Committee ECE Department
Dr. David Cyganski Thesis Committee ECE Department	Dr. Fred J. Looft Department Head ECE Department

Abstract

This M.S. thesis introduces new modulus scaling techniques for transforming a class of primes into special forms which enable efficient arithmetic. The scaling technique may be used to improve multiplication and inversion in finite fields. We present an efficient inversion algorithm that utilizes the structure of a scaled modulus. Our inversion algorithm exhibits superior performance to the Euclidean algorithm and lends itself to efficient hardware implementation due to its simplicity. Using the scaled modulus technique and our specialized inversion algorithm we develop an elliptic curve processor architecture. The resulting architecture successfully utilizes redundant representation of elements in $GF(p)$ and provides a low-power, high speed, and small footprint specialized elliptic curve implementation.

We also introduce a unified Montgomery multiplier architecture working on the extension fields $GF(p)$, $GF(2^m)$ and $GF(3^m)$. With the increasing research activity for identity based encryption schemes, there has been an increasing need for arithmetic operations in field $GF(3^m)$. Since we based our research on low-power and small footprint applications, we designed a unified architecture rather than having a seperate hardware for $GF3^m$. To the best of our knowledge, this is the first time a unified architecture was built working on three different extension fields.

Preface

In this thesis I describe research work I performed in the Cryptography and Information Security Lab during my graduate studies at WPI Electronics and Computer Engineering Department. I would like to use this place to thank many people who contributed to this work. First, I would like to thank our sponsors, the Intel Corporation and the National Science Foundation (ANI-0133297, ANI-0112889), for partially funding my research. I would like to thank my advisor Prof. Berk Sunar for his advice and support throughout the past years. His guidance and ancouragement made it possible for me to finish this work. I look forward to continue my research with him being my advisor for my Ph.D.

I also would like to thank Prof. Erkay Savas who introduced me into the world of cryptography. I am grateful to him for having me involved with cryptography. With his support and guidance during my last years of undergraduate studies, I was able to understand the notion of cryptography and attend the CRIS lab in WPI later on. I am very grateful to the members of my committee, Prof. Brian King and Prof. David Cyganski, for their support, advice and time they provided. I also want to thank my colleagues in the CRIS lab, Gunnar Gaubatz and Jens Peter Kaps, for all of their helps during my work, and sharing their knowledge.

Erdinc Ozturk

April 2005

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The incredible improvements in ubiquitous computing, and its indispensable implications gives rise to its being an effective domain of interest. As the notion of ubiquitous computing is becoming more and more part of our lives, various applications consisting of this new technology can be encountered. RFIDs are currently being introduced into the supply chain. Wireless sensor networks are widely used for many applications. In some cities most of the people carry at least one contactless smart card in their pockets.

These applications are becoming widespread, with an ultimate need of security. Currently, RFID applications have no security at all. Moreover, these applications

have limited power resources, which make them ultra-low power devices. Power-efficient implementations need to be used. Security applications are a part of the implementations, so they also have to be power-efficient.

So far, public key cryptography has not even been considered for these devices due to its perceived complexity. The common perception of public key cryptography is that it is complex, slow and power hungry, and as such not at all suitable for use in these environments.

It is therefore common practice to emulate the asymmetry of traditional public key based cryptographic services through a set of protocols using symmetric key based message authentication codes (MACs). Although the low computational complexity of MACs is advantageous, the protocol layer requires time synchronization between devices on the network and a significant amount of overhead for communication and temporary storage. The requirement for a general purpose CPU to implement these protocols as well as their complexity makes them prone to vulnerabilities and practically eliminates all the advantages of using symmetric key techniques in the first place.

Our aim is to challenge the basic assumptions about public key cryptography which are based on a traditional software based approach. We propose a custom hardware assisted approach for which we claim that it makes public key cryptography feasible for low-power applications, provided we use the right selection of algorithms

and associated parameters, careful optimization, and low-power design techniques.

Several public key schemes can be used to provide the security services described above. We take a closer look at Elliptic Curve Cryptosystems (ECC) as the most promising candidate for low-power implementations. We implemented the hardware design of a low-power and novel ECC architecture.

1.2 Modular Arithmetic

Modular arithmetic has a variety of applications in cryptography. Many public-key algorithms heavily depend on modular arithmetic. Among these, RSA encryption and digital signature schemes, discrete logarithm problem (DLP) based schemes such as the Diffie-Helman key agreement [DH76] and El-Gamal encryption and signature schemes [Nat91], and elliptic curve cryptography [Kob87, Men93] play an important role in authentication and encryption protocols. The implementation of RSA based schemes requires the arithmetic of integers modulo a large integer, that is in the form of a product of two large primes $n = p \cdot q$. On the other hand, implementations of Diffie-Helman and El-Gamal schemes are based on the arithmetic of integers modulo a large prime p. While ECDSA is built on complex algebraic structures, the underlying arithmetic operations are either modular operations with respect to a large prime modulus ($GF(p)$ case) or polynomial arithmetic modulo a high degree irreducible polynomial defined over the finite field $GF(2)$ ($GF(2^k)$ case). Special moduli

for $GF(2^k)$ arithmetic were also proposed [Ber68, SOOS95]. Low Hamming-weight irreducible polynomials such as trinomials and pentanomials became a popular choice [SOOS95, AMV93] for both hardware and software implementations of ECDSA over $GF(2^k)$. Particularly, trinomials of the form $x^k + x + 1$ allow efficient reduction. For many bit-lengths such polynomials do not exist; therefore less efficient trinomials, i.e. $x^k + x^u + 1$ with $u > 1$, or pentanomials, i.e. $x^k + x^u + x^v + x^z + 1$, are used instead. Hence, in many cases the performance suffers degradation due to extra additions and alignment adjustments.

In this thesis we utilize integer moduli of special form, which is reminiscent of low-Hamming weight polynomials. Although the idea of using a low-Hamming weight integer modulus is not new [Cra92], its application to Elliptic Curve Cryptography was limited to only elliptic curves defined over Optimal Extension Fields (i.e. $GF(p^k)$ with mid-size p of special form), or non-optimal primes such as those utilized by the NIST curves [Nat91]. In this work we achieve moduli of Mersenne form by introducing a modulus scaling technique. This allows us to develop a fast inversion algorithm that lends itself to efficient inversion hardware. For proof of concept we implemented a specialized elliptic curve processor. Besides using scaled arithmetic and the special inversion algorithm, we introduced several innovations at the hardware level such as a fast comparator for redundant arithmetic and shared arithmetic core for power optimization. The resulting architecture requires extremely low power at very small

footprint and provides reasonable execution speed.

1.3 Thesis Outline

After a short introduction into the motivation of the work done in this thesis and a brief introduction to the modular arithmetic in Chapter 1, Chapter 2 will present some of the earlier works in the field. The concepts that we used in our research will be analyzed for useful ideas.

Following that, in Chapter 3, modulus scaling techniques that were used for the background research of this thesis will be presented. Also in this chapter, the inversion algorithm that was achieved by modulus scaling techniques will be described and analyzed for hardware implementation. Chapter 4 presents the reader the system architecture of the design that has been developed as part of this thesis.

Chapter 5 presents a unified multiplier architecture that can work for three extension fields. First the background research is presented, than the structure of the presented hardware is described. The algorithms used for Montgomery multiplication are examined in this chapter.

Finally, Chapter 6 summarizes the research done for this thesis. It briefly explains the results of the research done.

Chapter 2

Previous Work

A straightforward method to implement integer and polynomial modular multiplications is to first compute the product of the two operands, $t = a \cdot b$, and then to reduce the product using the modulus, $c = t \bmod p$. Traditionally, the reduction step is implemented by a division operation, which is significantly more demanding than the initial multiplication. To alleviate the reduction problem in integer modular multiplications, Crandall proposed [Cra92] using *special primes*, primes of the form $p = 2^k - u$, where u is a small integer constant. By using special primes, modular reduction turns into a multiplication operation by the small constant u, that, in many cases, may be performed by a series of less expensive shift and add operations.

Let the number t represent the $2k$-bit result of a multiplication operation of two k-bit numbers. Let t_l represent the low k-bits and t_h represent the high k-bits:

$$t = t_h 2^k + t_l$$

$$\text{hence } c = t_h 2^k + t_l \pmod{p}$$

which can be reduced for $p = 2^k - u$ to

$$c = t_h \cdot u + t_l \pmod{2^k - u} .$$

It should be noticed that $t_h \cdot u$ is not fully reduced. Depending on the length of u, a few more reductions are needed. The best possible choice for a special prime is a Mersenne prime, $p = 2^k - 1$, with k fixed to a word-boundary, i.e. $k = 16, 32, 64$. In this case, the reduction operation becomes a simple modular addition $c = t_h + t_l \bmod p$. Similarly primes of the form $2^k + 1$ may simplify reduction into a modular subtraction $c = t_l - t_h \bmod p$. Unfortunately, Mersenne primes and primes of the form $2^k + 1$ are scarce. For degrees up to 1000 no primes of form $2^k + 1$ exist and only the two Mersenne primes $2^{521} - 1$ and $2^{607} - 1$ exist. Moreover, these primes are too large for ECDSA which utilizes bit-lengths in the range $160 - 350$. Hence, a more practical choice is to use primes of the form $2^k - 3$. For a constant larger than $u = 3$, and a degree k that is not aligned to a word boundary, some extra shifts and additions may be needed. To relax the restrictions, Solinas [Sol99] introduced a generalization for special primes. His technique is based on signed bit recoding. While increasing the number of possible special primes, additional low-level operations are needed. The special modulus reduction technique introduced by Crandall [Cra92] restricts the constant u in $p = 2^k - u$ to a small constant that fits into a single word.

Chapter 3

Modulus Scaling Techniques

3.1 General Method

The idea of modulus scaling was introduced by Walter [Wal92]. In this work, the modulus was scaled to obtain a certain representation in the higher order bits, which helped the estimation of the quotient in Barrett's reduction technique. The method works by scaling to the prime modulus to obtain a new modulus, $m = p \cdot s$. Reducing an integer a using the new modulus m will produce a result that is congruent to the residue obtained by reducing a modulo p. This follows from the fact that reduction is a repetitive subtraction of the modulus. Subtracting m is equivalent to s times subtracting p and thus $(a \bmod m) \bmod p \equiv a \bmod p$. When a scaled modulus is used, residues will be in the range $[m-1, 0] = [s \cdot p - 1, 0]$. The number is not fully

reduced and essentially we are using a redundant representation where an integer is represented using $\lceil \log_2 s \rceil$ more bits than necessary. Consequently, it will be necessary that the final result is reduced by p to obtain a fully reduced representation. Here we wish to use scaling to produce moduli of special form. If a random pattern appears in a modulus, it will not be possible to use the low-weight optimizations discussed in Chapter 2. However, by finding a suitable small constant s, it may be possible to scale the prime p to obtain a new modulus of special form, that is either of low-weight or in a form that allows efficient recoding. To keep the redundancy minimal, the scaling factor must be small compared to the original modulus. Assuming a random modulus, such a small factor might be hard or even impossible to find. We concentrate again on primes of special forms.

3.2 Special Primes

We present two heuristics that form a basis for efficient on-the-fly scaling using primes of special forms:

3.2.1 Heuristic 1

Heuristic 1 *If the base B representation of an integer contains a series of repeating digits, scaling the integer with the largest possible digit, produces a string of repeating zero digits in the scaled and recoded integer.*

The justification of the heuristic is quite simple. Assume the representation of the modulus in base B contains a repeating digit of arbitrary value D. We use the constant scaling factor $s = B - 1$ to compute m. When a string of repeating D-digits is multiplied with the scaling factor, and written in base B we obtain the following

$$\begin{aligned}(DDDD \ldots DDD)_B \cdot (B-1) &= (DDDD \ldots DDD0)_B - (DDDD \ldots DDD)_B \\ &= (D000 \ldots 000\bar{D})_B.\end{aligned}$$

The bar over the least significant digit denotes a negative valued digit.

We provide the following example:

Example 1 *We select the following prime*

$$p = (\texttt{51234567812345678123456781234567812345678123456807})_{16}.$$

By inspection we identify $(\texttt{12345678})_{16}$ *as a repeating pattern. By selecting the base* $B = 2^{32}$*, the repeating pattern becomes a digit. The scaling factor is the largest digit* $s = B - 1 = 2^{32} - 1 = (\texttt{FFFFFFFF})_{16}$*. The scaled modulus is computed as*

$$\begin{aligned}m &= s \cdot p \\ &= (\texttt{5123456730000000000000000000000000000000000000085DCBA97F9})_{16}\end{aligned}$$

The representation may contain more than one repeating digit. For instance, the prime

$$p = (\texttt{577777777777773333333333338B})_{16}$$

has two repeating digits 7 *and* 3. *Since both fit into a digit in base* $B = 16$, *scaling with* $B - 1 = 15$ *will work on both strings:*

$$\begin{aligned} m &= p \cdot s \\ &= (\texttt{57777777777777333333333338B})_{16} \cdot (F)_{16} \\ &= (\texttt{51FFFFFFFFFFFFC0000000000525})_{16}. \\ &= (\texttt{52000000000000}\overline{\texttt{4}}\texttt{0000000000525})_{16}. \end{aligned}$$

The presented scaling technique is simple, efficient, and only requires the modulus to have repeating digits. Since the scaling factor is fixed and only depends on the length of the repeating pattern – not its value –, a modulus with multiple repeating digits can be scaled properly at the cost of increasing the length of the modulus by a single digit. We present another heuristics for scaling, this technique is more efficient but more restrictive on the modulus.

3.2.2 Heuristic 2

Heuristic 2 *Given a modulus containing repeating* D*-digits in base* B *representation, if* $B - 1$ *is divisible by the repeating digit, then the modulus can be efficiently scaled by the factor* $\frac{B-1}{D}$.

As earlier the heuristic is verified by multiplying a string of repeating digits with the scaling factor and then by recoding.

$$\begin{aligned}(DDD\ldots DDD)_B \cdot \frac{B-1}{D} &= ((B-1)(B-1)(B-1)\ldots(B-1))_B \\ &= (1000\ldots0\bar{1})_B.\end{aligned}$$

The following example shows the power of this simple technique.

Example 2 *Let the prime p be*

$$\begin{aligned}p = (&\texttt{D79435E50D79435D79435E50D79435E50D79435E50D79435E50D79435E50}\| \\ &\texttt{D79435E50D79435E50D79435E50D79435E5})_{16}\end{aligned}$$

By inspection the repeating pattern is detected as $D = (0D79435E5)_{16}$. The digit D fits into 36-bits, thus the base is selected as $B = 2^{36}$. Since $D|(B-1)$ the scaling factor is computed as

$$s = \frac{2^{36}-1}{(0D79435E5)_{16}} = 19.$$

The scaled modulus becomes

$$m = s \cdot p = 2^{384} - 2^{320} - 1.$$

We have compiled a list of primes that when scaled with a small factor produce moduli of the form $2^k \pm 1$ in Table 4 (see Appendix A). These primes provide a wide range of perfect choices for the implementation of cryptographic schemes.

3.3 Scaled Modular Inversion

In this section we consider the application of scaled arithmetic to implement more efficient inversion operations. An efficient way of calculating multiplicative inverses is to use binary extended Euclidean based algorithms. The Montgomery inversion algorithm proposed by Kaliski [Kal95] is one of the most efficient inversion algorithms for random primes. Montgomery inversion, however, is not suitable when used with scaled primes since it does not exploit our special moduli. Furthermore, it can be used only when Montgomery arithmetic is employed. Therefore, what we need is an algorithm that takes advantage of the proposed special moduli. Thomas et al. [TKL86] proposed the Algorithm X for Mersenne primes of the form $2^q - 1$ (see Appendix B).

Due to its simplicity Algorithm X is likely to yield an efficient hardware implementation. Another advantage of Algorithm X is the fact that the carry-free arithmetic can be employed. The main problem with other binary extended Euclidean algorithms is that they usually have a step involving comparison of two integers. The comparison in Algorithm X is much simpler and may be implemented easily using carry-free arithmetic.

The algorithm can be modified to support the other types of special moduli as well. For instance, changing Step 4 of the algorithm to $b := -(2^{q-e}b) \pmod{p}$ will make the algorithm work for special moduli of the form $2^q + 1$ with almost no

penalty in the implementation. The only problem with a special modulus, m is the fact that it is not prime (but multiple of a prime, $m = sp$) and therefore inverse of an integer $a < m$ does not exist when $\gcd(a, m) \neq 1$. With a small modification to the algorithm this problem may be solved as well. Without loss of generalization the solution is easier when s is a small prime number. Algorithm X normally terminates when $y = 1$ for integers that are relatively prime to the modulus, m. When the integer a is not relatively prime to the modulus, then Algorithm X must terminate when $y = \gcd(a, m) = s$ resulting $b = a^{-1} \cdot s \pmod{m}$. In order to obtain the inverse of a when $\gcd(a, m) \neq 1$, an extra multiplication at the end is necessary:

$$b = b \cdot (s^{-1} \pmod{p}) \pmod{m}$$

where $s^{-1} \pmod{p}$ needs to be precomputed. This precomputation and the task of checking $y = s$ as well as $y = 1$, on the other hand, may be avoided utilizing the following technique. The integer a, whose inverse is to be computed, is first multiplied by the scale s before the inverse computation: $a' = a \cdot s$. When the inverse computation is completed we have the following equality

$$a' \cdot b + m \cdot k = s$$

and thus

$$a \cdot s \cdot b + p \cdot s \cdot k = s \ .$$

When both sides of the equation is divided by s we obtain

$$a \cdot b + p \cdot k = 1.$$

Therefore, the algorithm automatically yields the inverse of a as $b = a^{-1}$ if the input is taken as $s \cdot a \bmod m$ instead of a. Although this technique necessitates an extra multiplication before the inversion operation independent of whether a is relatively prime to modulus m or not, eliminating the precomputation and a comparison is a significant improvement in a possible hardware implementation. Furthermore, this multiplication will reduce to several additions when the scale is a small integer such as the $s = 3$ in $p = (2^{167} + 1)/3$. Another useful modification to Algorithm X is to transform it into a division algorithm to compute operations of the form d/a. The only change required is to initialize b with d instead of 1 in Step 1 of the algorithm. This simple modification saves one multiplication in elliptic curve operations. The Algorithm X modified for division with scaled modulus is shown below:

Algorithm X - modified for division with scaled modulus

Input: $a \in [1, m-1]$, $d \in [1, m-1]$, m, and q where $m = 2^q \pm 1$

Output:$b \in [1, m-1]$, where $b = d/a \pmod{m}$

1: $a := a \cdot s \pmod{m}$;

2: $(b, c, u, v) := (d, 0, a, m)$;

3: Find e such that $2^e || u$

4: $u := u/2^e$; // shift off trailing zeros

5: $b := \mp(2^{q-e}b) \pmod{m}$; // circular left shift

6: if $u = s$ return b;

7: $(b, c, u, v) := (b + c, b, u + v, u)$;

8: go to Step 3

It should be noted that the notation $2^e||u$ stands for the largest integer value of e such that 2^e exactly divides u.

One can easily observe that the Algorithm X has the loop invariant $b/u \pmod{m} \equiv d/a \pmod{m}$. Note that the Step 3 of Algorithm X can be performed using simple circular left shift operations. The advantage of performing the Step 3 with simple circular shifts may dissappear for moduli of the form $2^q - c$ with even a small c. Many inversion algorithms consist of a big loop and the efficiency of an inversion algorithm

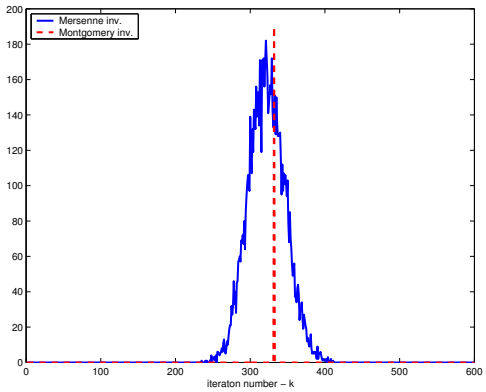


Figure 3.1: Distribution of k

depends on the number of iterations in this loop, k , which, in turn, determines the total number of additions, shift operations to be performed. The number of iterations are usually of random nature (but demonstrates a regular and familiar distribution) and only statistical analysis can be given. In order to show that Algorithm X is also efficient in terms of iteration number, we compared its distribution for k against that of Montgomery inversion algorithm. We computed the inverses of 10000 randomly chosen integers modulo $m = 2^{167} + 1$ using Algorithm X. Since $p = m/3$ is a 166-bit prime we repeated the same experiment with the Montgomery inversion algorithm using p. Besides having much easier operations in each iteration we observed that the average number of iterations of Algorithm X is slightly lower than the total number of iterations of the Montgomery inversion algorithm (Figure 3.1).

Chapter 4

The Elliptic Curve Architecture

We developed an elliptic curve architecture using the scaled modulus technique and our specialized inversion algorithm. Our aim in implementing this hardware was to actually see the outcomes of our techniques.

4.1 Design Methodology

We built our elliptic curve scheme over the prime field $GF((2^{167}+1)/3)$. This particular prime allows us to use a scaled modulus $m = 2^{167} + 1$ with a very small scaling factor $s = 3$. To implement the field operations we use Algorithm X as outlined in Chapter 3.3. Our simulation for this particular choice of prime showed that our inversion technique is only by about three times slower than a multiplication operation. Furthermore, the inversion is implemented as a division saving one multiplication

operation. Thus the actual ratio is closer to two. Since inversion is relatively fast, we prefer to use affine coordinates. Besides faster implementation, affine coordinates provides a significant amount of reduction in power and circuit area since projective coordinates requires a large amount of extra storage. For an elliptic curve of form $y^2 = x^3 + ax + b$ defined over $GF(2^{167} + 1)/3)$ we use the standard point addition operation defined in [Men93].

For power efficiency we optimize our design to include minimal hardware. An effective strategy in reducing the power consumption is to spread the computation to a longer time interval via serialization which we employ extensively. On the other hand, a reasonable time performance is also desired. Since the elliptic curve is defined over a large integer field $GF(p)$ (168-bits) carry propagations are critical in the performance of the overall architecture. To this end, we built the entire arithmetic architecture using the carry-save methodology. This design choice regulates all carry propagations and delivers a very short critical path delay, and thus a very high limit for the operating frequency.

The redundant representation doubles all registers in the arithmetic unit, i.e. we need two separate registers to hold both the carry part and the sum part of a number. Furthermore, the inherent difficulty in comparing numbers represented in carry-save notation is another challenge. In addition, shifts and rotate operations become more cumbersome. Nevertheless, as evident from our design it is possible to overcome these

difficulties.

In developing the arithmetic architecture we primarily focused on finding the minimal circuit to implement Algorithm X efficiently. Since the architecture is built around the idea of maximizing hardware sharing among various operations, the multiplication, squaring and addition operations are all achieved by the same arithmetic core. The control is hierarchically organized to implement the basic arithmetic operations, point addition, point doubling, and the scalar point multiplication operation in layers of simple state machines. The simplicity of Algorithm X and scaled arithmetic allows us to accomplish all operations using only a few small state machines.

4.2 Implementation of the Control Block

Since the arithmetic core is a general purpose hardware, we needed a control block that can handle the desired arithmetic operations by switching the select inputs of the multiplexers accordingly. We have a 15-state state machine implementing the inversion algorithm. Each state corresponds to a step in the algorithm. Mainly there are three states in the state machine: Initialize, shift right, and add.

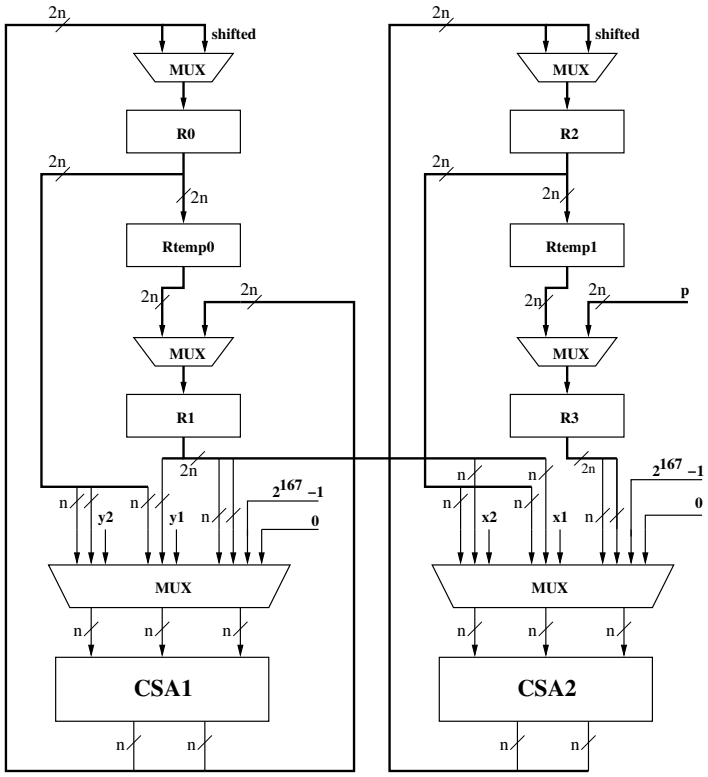


Figure 4.1: Block diagram of the arithmetic unit

4.3 Implementation of the Arithmetic Unit

The arithmetic unit shown in Figure 4.1 is built around four main registers R0, R1, R2, R3, and two extra registers Rtemp0, Rtemp1 which are used for temporary storage. Note that these registers store both the sum and carry parts due to the carry-save representation. For the same purpose the architecture is built around two (almost) parallel data paths.

We briefly outline the implementation of basic arithmetic operations.

4.3.1 Comparison

Comparing a two numbers in carry-save architecture is difficult since the redundant representation hides the actual values. On the positive side, the comparison in Algorithm X is only with respect to a constant value of $s = 3$. Such a comparator may be built using a massive OR tree with $2n$ inputs. Unfortunately, such an OR tree would cause serious latency ($O(\log_2 n)$ gate delays) and significantly increase the critical path delay. We instead prefer a novel comparator design that works only for comparing a number with zero. In order to compare a number with 3, extra logic is needed for the first two bits, which is nothing more than a pair of xor gates. The rest of the bits are connected directly to the comparator. The comparator is built by connecting three-state buffers together as shown in Figure 4.2. The input lines are connected together and set to logic 1. Similarly the output lines are connected together and taken as the output of the comparator. We feed the bits of the data input in parallel to the enable inputs of the three-state buffers. Hence, if one or more of the bits of the data input is logic 1, which means the number is not equal to 0, we see logic 1 at the output of the comparator. If the number is 0, none of the three-state buffers is enabled and therefore we see a Hi-Z (high impedance) output. Note that our comparator design works in constant time (O(1) gate delays) regardless of the length of the operands.

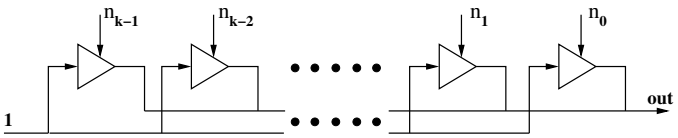


Figure 4.2: Comparator unit built using tri-state buffers

4.3.2 Modulo Reduction

Since the hardware works for $m = 2^{167} + 1$, 168-bit registers would be sufficient. However, we use an extra bit to detect when the number becomes greater than m. If one of the left-most bits of the number (carry or sum) is one, the number is reduced modulo m. Note that

$$2^{168} = 2 \cdot (2^{167} + 1) - 2 = 2m - 2 = m - 2 \pmod{m}.$$

Hence, the reduction is achieved by subtracting 2^{168} (or simply deleting this bit) and adding $m - 2 = (11 \ldots 11111)_2$ (167 bits) to the number. If both of the leftmost bits are 1 then: $2 \cdot (2^{168}) = 4 \cdot (2^{167} + 1) - 4 = 4m - 4 = m - 4 \pmod{m}$. Therefore $m - 4 = (111 \ldots 11101)_2$ (167 bits) has to be added to the number and both of the leftmost bits are deleted.

4.3.3 Subtraction

Suppose k is a 168 bit number which we want to subtract from another number modulo m. The bitwise complement of k is found as

$$k' = (2^{168} - 1) - k = 2 \cdot (2^{167} + 1) - 3 - k = -3 - k \pmod{m} .$$

Thus $-k = k' + 3 \bmod m$. This means that to subtract k from a number we simply add the bitwise complement of k and 3 to the number. There is a caveat though. Remember that our numbers are kept in carry save representation, so, there are two 168-bit numbers representing k. Let k_s and k_c denote the sum and carry parts of k, respectively. Since $k = k_s + k_c$ then $-k = -k_s - k_c = (k_s' + 3) + (k_c' + 3) = k_s{}' + k_c{}' + 6 \bmod m$. Therefore the constant value 6 has to be added to the complements of the carry and sum registers in order to compute $-k$.

4.3.4 Multiplication

We serialize our multiplication algorithm by processing one bit of one operand and all bits of the second operand in each iteration. The standard multiplication algorithm had to be modified to make it compatible with the carry save representation. Due to the redundant representation, the value of the leftmost bit of the multiplier is not known. Hence, the left to right multiplication algorithm may not be used directly. We prefer to use the right to left multiplication algorithm. With this change, instead of shifting the product we multiply the multiplicand by two (or shift left) in each iteration step.

There are 3 registers used for the multiplication: R0 (multiplicand), R1 (product) and R2 (multiplier). The multiplication algorithm has 3 steps :

1. Initialization: This is done by the control circuit. The multiplicand is loaded

to R0, the multiplier is loaded to R2 and R1 is reset.

2. Addition: This step is only done when the rightmost bit of register R2 is 1. The content of register R0 is added to R1.

3. Shifting: The multiplier has to be processed bit by bit starting from the right. We do this by shifting register R2 to the right in each iteration of the multiplication. Since the register R2 is connected to the comparator, the algorithm terminates after this step if the number becomes 0 else the algorithm continues with Step 2. Note that no counters are used in the design. This eliminates potential increases in the critical path delay. The multiplicand needs to be doubled in each iteration as well. This is achieved by shifting register R0 to the left. This operation is performed in parallel with shifting R2, so no extra clock cycles are needed. However, shifting to the left can cause overflow. Therefore, the result needs to be reduced modulo m if the leftmost bit of the register R0 is 1.

4.3.5 Inversion

To realize the inversion operation there are four registers used to hold b, c, u and v, two temporary registers are used for the addition of two numbers in carry-save architecture. Two carry-save adders, multiplexers and comparator architecture are also utilized.

The inversion algorithm shown in Algorithm X has 5 steps:

1. Initialization: This is done by the control circuit. Load registers with $b = 1, c = 0, u = x$ (the data input) and $v = m = (2^{167} + 1)$.

2. $u = u/2^e$: This operation is done by shifting u to the right until a 1-bit is encountered. However, due to the carry-save architecture this operation requires special care. The rightmost bit of the carry register is always zero since there is no carry input. Thus just checking the rightmost bit of the sum register is sufficient. Also, the carry has to be propagated to the left in each iteration. This is done by adding 0 to the number. If a 1-bit is encountered, the operation proceeds to the next step.

3. $b = (-2^{q-e} \cdot b) \bmod m$: Assume u holds a random pattern, e will be very small (not more than 3 for most of the cases). Thus, $q - e$ is most likely a large number. Therefore, multiplication by 2^{q-e} would require many shifts to left. To compute this operation more efficiently, this step is rewritten using the identity $2^q = -1 \bmod m$ as $b = 2^{-e} \cdot b \pmod m$. Therefore, b needs to be halved e-times. If b is even we may shift it to the right and thereby divide it by two. Otherwise, we add m to it to make it even and then shift. Since this step takes e iterations, it can be performed concurrently with the 2nd step of the algorithm. Hence no extra clock cycles are needed for this step.

4. Compare u with $s = 3$:The comparator architecture explained above is used to implement this step. There are two cases when $u = 3$: $u_s = (11)_2, u_c = (00)_2$ and $u_s = (01)_2, u_c = (10)_2$. Therefore, the rightmost two bits need a special logic for the comparison, and the rest of the bits are connected directly to the three-state comparator shown in Figure 4.2.

5. Additions in $(b, c, u, v) := (b+c, b, u+v, u)$. Two clock cycles are needed to add two numbers in carry-save architecture, since a carry-save adder has 3 inputs and there are 4 numbers to add. During the addition operation to preserve the values of b and u the two temporary registers are used.

4.4 Performance Analysis

In this section we analyze the speed performance of the overall architecture and determine the number of cycles required to perform the elliptic curve operations. The main contributors to the delay are field multiplications and inversion operations. Field additions are performed in 1 cycle (or 2 cycles if both operands are in the carry-save representation). Therefore field additions which take place outside of the multiplication or inversion operations are neglected.

The multiplication operation iterates over the bits of one operand. On average half of the bits will be ones and will require a 2 cycle addition. Hence, 168 clock cycles

will be needed. The multiplicand will be shifted in each cycle and modulo reduced in about half of the iterations. Hence another $1.5 \cdot 168 = 252$ cycles are spent. The multiplication operation takes on average a total of 420 cycles.

The steps of the inversion algorithm are reorganized in Table 1 according to the order and concurrency of the operations. Note the two concurrent operations shown in Step 2. In fact this is the only step in the algorithm which requires multiple clock cycles, hence the concurrency saves many cycles. In Step 2, u is shifted until all zero bits in the LSB are removed. Each shift operation takes place within one cycle. For a randomly picked value of u the probability of the last e bits all being zeroes is $(1/2)^e$, hence the expected value of e is $E(e) = \sum_{i=1}^{\infty} i(1/2)^i = 2$. In each iteration of the algorithm we expect on average of 2 cycles to be spent. Step 3 does not spend any cycles since the comparator architecture is combinational. The additions in Step 4 require 2 clock cycles. Hence a total of 4 cycles is spent in each iteration of the inversion algorithm. Our simulation results showed that (see Section 3.3) the inversion algorithm would iterate on average about 320 times. The total time spent in inversion is found as $1,280$ cycles. This is very close to our hardware simulation results which gave an average of $1,288$ cycles.

1:	Initialize all registers
	$(b, c, u, v) \leftarrow (1, 0, a, m)$
2:	Shift off all trailing zeros and rotate b
	$u \leftarrow u >> e \qquad b \leftarrow b >> e \pmod{m}$
3:	Check terminate condition
	if $u = s$ return b
4:	Update variables
	$(b, c, u, v) \leftarrow (b + c, b, u + v, u);$
	go back to Step 2

Table 1: Hardware algorithm for inversion.

The total number of clock cycles for point addition and doubling is found as $2,120$ and $2,540$, respectively. The total time required for computing a point multiplication is found as $545,440$ cycles.

4.5 Results and Comparison

The presented architecture was developed into Verilog modules and synthesized using the Synopsys tools Design Compiler and Power Compiler. In the synthesis we used the TSMC $0.13\,\mu$m ASIC library, which is characterized for power. The resulting architecture was synthesized for three operating frequencies. The implementation

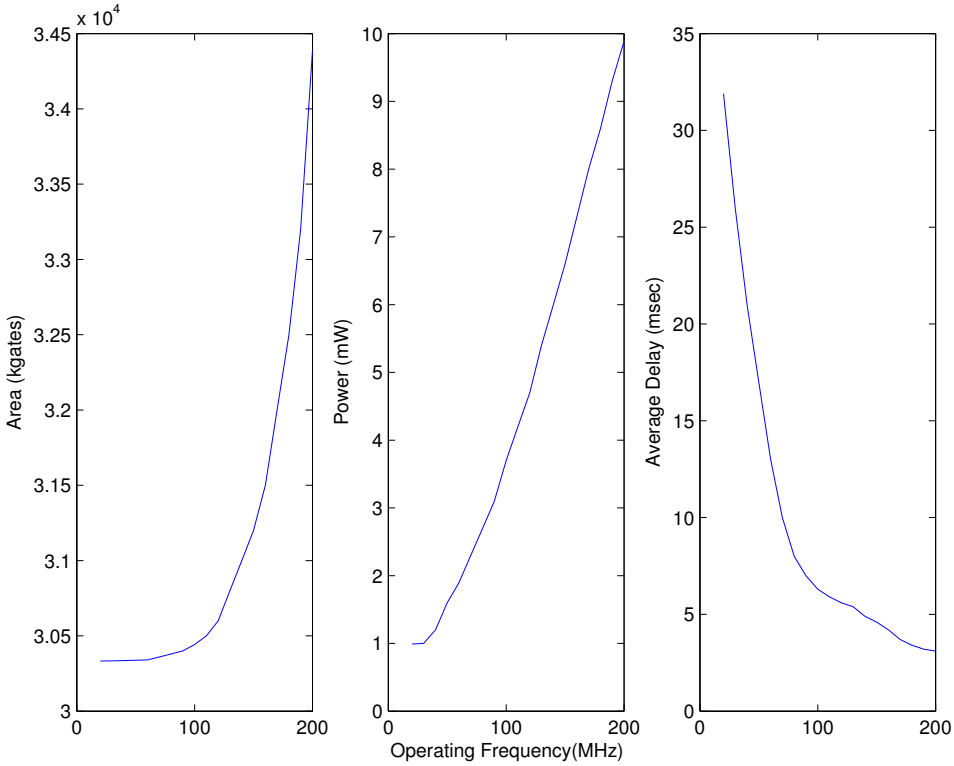


Figure 4.3: Implementation Results

results are shown in Figure 4.3. As seen in the figure the area varies around 30 Kgates. The circuit achieves its intended purpose by consuming only 0.99 mW at 20 Mhz. In this mode the point multiplication operation takes about 31.9 msec. Although this is not very fast, this operating mode might be useful for interactive applications with extremely stringent power limitations. On the other hand, when the circuit is synthesized for 200 Mhz operation, the area is slightly increased to 34 Kgates, and the power consumption increased to 9.89 mW. However, a point multiplication takes now only 3.1 msec.

We performed a research to obtain the resutls from the previously built ECC architectures. However, we concluded with a result that there has not been any work done for low-power ECC architecture design. We compare our design with another customized low-power elliptic curve implementation presented by Schroeppel et al. in CHES 2002 [SBM+02]. Their design is the closest to a low-power ECC design. Their design employed an elliptic curve defined over a field tower $GF(2^{178})$ and used specialized field arithmetic to minimize the design. A point halving algorithm was used in place of the traditional point doubling algorithm. The design was power optimized through clock gating and other standard methods of power optimization. The main contribution was the clever minimization of the gate logic through efficient tower field arithmetic. Note that their design includes a fully functional signature generation architecture whereas our design is limited to point multiplication. Although a side by side comparison is not possible, we find it useful to state their results: The design was synthesized for 20 Mhz operation using 0.5 μm ASIC technology. The synthesized design occupied an area of 112 Kgates and consumed 150 mW. The elliptic curve signature was computed in 4.4 msec. Unfortunately, since we did not have access to the 0.5 μm technology, which would have made the comparison precise.

An architectural comparison of the two designs shows that our design operates bit serially in one operand whereas their design employs a more parallel implementation strategy. This leads to lower critical paths and much smaller area in our design. The

much shorter critical path allows much higher operating frequencies requiring more clock cycles to compute the same operation. However, due to the smaller area, when operated at similar frequencies our design consumes much less power.

Chapter 5

Unified Multiplier Architecture

5.1 Introduction

There has been an increase in research activity for the cryptosystems on pairing based operations. Schemes utilizing these pairing schemes, such as identity based encryption [BF01] and signature algorithms have been developed. Identity based cryptography was first proposed by Shamir [Sha85] in 1985. Rather than deriving a public key from a private information, which would be the case in traditional schemes, in identity based schemes a user identity, an arbitrary string, plays the role of a public key. This reduces the computations for authentication and has a number of security characteristics. These identity based schemes are implemented the most efficient on the field $GF(3^m)$.

Previously we designed an ECC hardware which works on the extension field $GF(p)$. Since the identity based encryption schemes are getting more important and there is more research being conducted on these schemes, we realized the importance of adding the field $GF(3^m)$ to our design for identity based cryptography. Since we based our design on small footprint and low-power application, we aimed on building a unified architecture that supports arithmetic in three fields, $GF(p)$, $GF(2^m)$ and $GF(3^m)$ rather than having separate hardware for all of them. The results of our research on previous work showed that previously a unified architecture for $GF(p)$ and $GF(2^m)$ has been built ([STK00]). Also, hardware architectures for arithmetic in characteristic three have appeared in [PS02], [TKM04]and [BGK$^+$03]. However, to the best of our knowledge, a unified architecture working for all the three fields has not been appeared.

The basic arithmetic operations (i.e., addition, multiplication and inversion) in the arithmetic extension fields $GF(p)$, $GF(2^m)$ and $GF(3^m)$ have several applications in elliptic curve cryptography. The most important of these three arithmetic operations is the field multiplication operation since it is the core operation in many cryptographic functions. The Montgomery multiplication algorithm [Mon85] is an efficient method for performing modular multiplication. With this motivation and, we designed a Unified Montgomery Multiplier Architecture for the arithmetic extension fields $GF(P)$, $GF(2^m)$ and $GF(3^m)$ using a different number representation.

5.2 Redundant Signed Digit (RSD) Arithmetic

Although carry-save arithmetic decreases the propagation delay in addition operations, the use of carry-save arithmetic for modular subtraction operations, which is required for arithmetic in algebraic fields introduces significant problems. When two's complement representation is used for subtraction, the carry overflow must be ignored. If there is no carry overflow, the result is negative. Since there can be hidden carry overflow with carry-save representation, it is hard to be sure that the result is positive or negative. It requires additional operations and additional hardware, which increases both latency and area.

RSD arithmetic was introduced by Avizienis [Avi61] and is quite similar to carry-save arithmetic. An integer is still represented by two positive integers, however the non-redundant form of the representation is the difference between these two positive integers, not the sum. If the number X is represented by x^+ and x^- then $X = x^+ - x^-$.

One advantage of using RSD is the fact that it eliminates two's complement form to handle negative numbers. It is thus much easier to do both addition and subtraction operations without worrying about the carry and borrow chain. Furthermore, the subtraction operation does not require taking two's complement of the subtrahend. It is thus a more natural representation if both addition and subtraction operations have to be performed, which is the case in the Montgomery inverse algorithm. Also, comparison of two integers is much easier with RSD representation. After subtracting

one integer from the other one, sign test can be performed directly by testing the first nonzero bit, which is an easy way of telling which number is bigger than the other one.

5.2.1 Number Representations

As mentioned before, the integer X is represented by two integers, x^+ and x^-, and $X = x^+ - x^-$. For RSD representations, we can use the notation (x^+, x^-) to represent the number X.

The RSD number representation for the given extension fields are described as follows:

1. $GF(p)$ In the extension field $GF(p)$, the integers are in binary form and a digit can have three different values: 1, 0 and -1. In RSD form, these three digits are represented as:

$$\begin{aligned} 1 &\rightarrow (1,0) \\ 0 &\rightarrow (0,0) \\ -1 &\rightarrow (0,1) \end{aligned}$$

2. $GF(2^m)$ In field $GF(2^m)$, the integers are also in binary form. However, since there is no carry chain in $GF(2)$ arithmetic, a digit can have the values 1 or 0.

These are represented as:

$$1 \rightarrow (1,0)$$

$$0 \rightarrow (0,0)$$

3. $GF(3^m)$ In field $GF(3^m)$, the integers are in base 3. Digits can have the values -2 -1 0 1 and 2. However, since there is no carry chain in $GF(3^m)$ arithmetic, the digit values -2 and 2 are congruent to 1 and -1, respectively. The RSD representations are:

$$2 \rightarrow (0,1)$$

$$1 \rightarrow (1,0)$$

$$0 \rightarrow (0,0)$$

$$-1 \rightarrow (0,1)$$

$$-2 \rightarrow (1,0)$$

5.3 Unified Arithmetic Core

We first build a unified arithmetic core for the basic arithmetic operations (i.e., addition, subtraction and comparison). The core is unified so that it can do the arith-

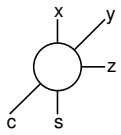
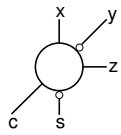
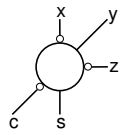
Logic symbol			
Type	GFA–0	GFA–1	GFA–2
Function	x+y+z = 2c+s	x–y+z=2c–s	–x+y–z=–2c+s

Figure 5.1: Generalized full adders

metic operations in three extension fields, $GF(p)$, $GF(2^m)$ and $GF(2^n)$. Since the elements of these fields are represented using almost the same data structure inside the computer, and the algorithms for basic arithmetic operations in all three fields have structural similarities, we were able to build a unified arithmetic core.

5.3.1 Architecture

The conventional 1-bit full adder assumes positive weights for all of its three binary inputs and two outputs. However, full adders can be generalized to have both positive and negative weight inputs and outputs. This allows us to construct an adder design with both inputs and outputs in RSD form, since we can have negative weight numbers as inputs. In our core design, we used two forms of the generalized full adders (Figure 5.1), one negative weight input (GFA-1) and two negative-weight inputs (GFA-2).

The logic behaviors of these generalized full adders are shown in Figure 5.2). As can be seen from the truth table, they have same logical characteristics. The only

			GFA–0		GFA–1		GFA–2	
x	y	z	s	c	s	c	s	c
0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	1	1
0	1	0	1	0	0	1	0	1
0	1	1	0	1	0	0	0	0
1	0	0	1	0	1	1	1	1
1	0	1	0	1	1	0	1	0
1	1	0	0	1	0	0	0	0
1	1	1	1	1	1	1	1	1

Figure 5.2: Logic tables of the three generalized full adders

difference is the order of the inputs and outputs. Same hardware is used for both of the generalized full adders. However, it should be noted that the decoding of the outputs are different. For GFA-1, the result is decoded as $2c - s$. For GFA-2, the result is decoded as $-2c + s$.

A 1-digit adder unit is constructed using two of the generalized full adders (Figure 5.3). This adder unit has two integers in RSD representation as inputs and one integer in RSD representation as output. This 1-digit unit also has carry inputs and outputs, which are only used for arithmetic in $GF(p)$. In total, the unit has 5 bits of input and 3 bits of output.

We started designing the hardware for the extension field $GF(p)$ first. Two generalized full adders connected as shown in figure 5.3 without any other logic in between handles the arithmetic operations in $GF(p)$.

Since the carry chain in $GF(2^m)$ arithmetic is neglected, all we had to do to make

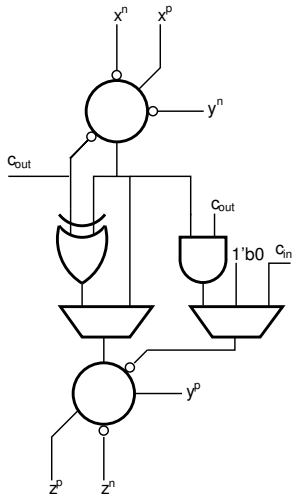


Figure 5.3: RSD adder unit with both inputs and outputs in RSD form

this architecture also work for $GF(2^m)$ was suppressing the carry chain. Also, since the digits can have the values $(0, 0)$ and $(1, 0)$, the negative weight inputs of the adder are initialized as logic 0.

Modifying this hardware design so that it will also work for $GF(3^m)$ is the most complex part of the design. Since the numbers are in base 3 form and we have a hardware that works for base 2, the design became complicated. The carry-free structure of the $GF(3^m)$ arithmetic operations allowed us to build an adder which works for both base 3 and binary forms.

When doing arithmetic operations in $GF(3^m)$, the outputs of the adders has to be decoded. Since the generalized full adder works in binary form, the output is also binary. We needed to convert this output to base 3 before entering the data into the second generalized full adder. An XOR gate and an AND gate is sufficient enough

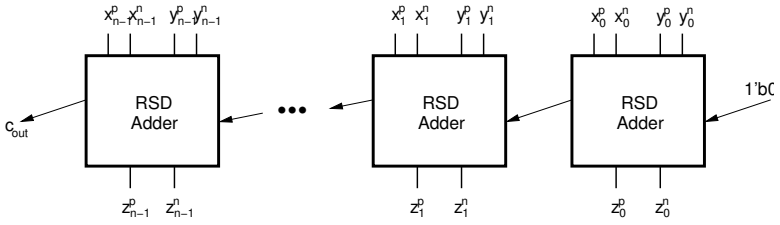


Figure 5.4: RSD adder

for this conversion (figure 5.3). There is also need for multiplexers, and the select inputs of the multiplexers determine in which field the adder is working. The carry bits seen are only used for $GF(p)$.

Now all we need to do is to connect the 1-digit RSD units back to back in order to build an n-bit RSD adder. Figure 5.4 shows the backbone of the structure. There are n 1-digit RSD adders and just one GFA-1 adder to handle the last carry bit, which is omitted in $GF(2^m)$ and $GF(3^m)$.

5.3.2 Addition

The addition operation is done as shown in Figure 5.4. The negative and positive parts of the numbers are entered accordingly and the select inputs of the multiplexers are set for desired field operations. There are two control inputs for selecting the field, sel_2 and sel_3. These inputs are decoded accordingly and they determine the select inputs of the multiplexers.

5.3.3 Subtraction

Subtraction operation is identical to the addition operation. The only difference is that the positive and the negative parts of the numbers in RSD form are swapped before the operation. Swapping the positive and negative parts negates the number:

$$\begin{aligned} X &= (x^+, x^-) = x^+ - x^- \\ Y &= (y^+, y^-) = y^+ - y^- \\ X - Y &= (x^+, x^-) - (y^+, y^-) = (x^+, x^-) + (y^-, y^+) \end{aligned}$$

5.3.4 Comparison

In order to perform comparison between two numbers, one must be subtracted from the other one. After subtraction, a sign test is applied to the result. Doing the sign test is a simple process as it can be performed directly by testing the first nonzero bit pair. If the positive part of first nonzero bit pair is logic 1, the subtracted number is smaller than the other one. If the negative part of the first nonzero bit pair is logic 1, the subtracted number is greater than the other one. If all the bit pairs are zero, the numbers are equal. Since our proposed hardware does not allow both of the bit pairs be logic 1 at the same time, there is no need to check if both of the bit pairs are logic 1.

5.4 Montgomery Multiplication

The proposed adder design is used to build a Montgomery multiplier architecture. Since there are three different extension fields upon which we want our hardware to work on, the algorithm had three different versions. We tried to find the similarities of these versions and integrate them together into a single hardware implementation.

5.4.1 The Multiple-Word Radix-2 Montgomery Multiplication Algorithm for $GF(p)$

The use of a fixed precision word alleviates the broadcast problem in the circuit implementation. Furthermore, a word-oriented algorithm allows design of a scalable unit.

For a modulus of m-bit precision, and a word size of w-bits, $e = \lceil m + 1/w \rceil$ words are required. Note that an extra bit is used for the variables holding the partial sum in the Montgomery algorithm for $GF(p)$, since the partial sums can reach $m + 1$-bit precision. The algorithm [TK99] we used scans the multiplicand operand (B) word-by-word, and the multiplier operand (A) bit-by-bit.

The vectors used in multiplication operations are expressed as

$$B = (B^{(e-1)}, ..., B^{(1)}, B^{(0)}),$$

$$A = (A_{(m-1)}, ..., A_{(1)}, A_{(0)}),$$

$$p = (p^{(e-1)}, ..., p^{(1)}, p^{(0)}),$$

where the words are marked with superscripts and the bits are marked with subscripts. For example, the ith bit of the kth word of B is representeed as $B_i^{(k)}$. A particular range of bits in a vector B from position i to j where $j > i$ is represented as $B_{j..i}$. $(x|y)$ represents the concatenation of two bit sequence. Finally, 0^m stands for an all-zero vector of m bits. The algorithm is given below:

Input: $A, B \in GF(p)$ and p

Output:$C \in GF(p)$

Step 1: $T := 0^m$

Step 2: for $i = 0$ to $m - 1$

Step 3: $(Carry|T^{(0)}) := a_i \cdot B^{(0)} + T^{(0)}$

Step 4: $Parity := T_0^{(0)}$

Step 5: $(Carry|T^{(0)}) := Parity \cdot p^{(0)} + (Carry|T^{(0)})$

Step 6: for $j = 1$ to $e - 1$

Step 7: $(Carry|T^{(j)}) := a_i \cdot B^{(j)} + T^{(j)} + Parity \cdot p^{(j)} + Carry$

Step 8: $T^{(j-1)} := (T_0^{(j)}|T_{w-1..1}^{(j-1)})$

Step 9: $T^{e-1} := (Carry|T_{w-1..1}^{(e-1)})$

Step 10: $C := T$

Step 11: if $C >$ then $C := C - p$

Step 12: return C

We use the RSD form for every vector used in the multiplication algorithm, so every bit expressed in this algorithm is represented by actually two bits, positive and negative parts of the numbers. As an example: $T_0^0 = (T_{0,p}^0, T_{0,n}^0)$.

5.4.2 Multiple-Word Radix-2 Montgomery Multiplication Algorithm for $GF(2^m)$

The Montgomery multiplication algorithm for $GF(2^m)$ is given below. Since there is no carry computation in $GF(2^m)$ arithmetic, the intermediate addition operations are replaced by bitwise XOR operations, which are represented below using the symbol $\oplus$.

Input: $A, B \in GF(2^m)$ and p

Output:$C \in GF(2^m)$

Step 1: $T := 0^m$

Step 2: for $i = 0$ to $m - 1$

Step 3: $T^{(0)} := a_i B^{(0)} \oplus T^{(0)}$

Step 4: $Parity := T_0^{(0)}$

Step 5: $T^{(0)} := Parity \cdot p^{(0)} \oplus T^{(0)}$

Step 6: for $j = 1$ to $e - 1$

Step 7: $T^{(j)} := a_i B^{(j)} \oplus T^{(j)} \oplus Parity \cdot p^{(j)}$

Step 8: $T^{(j-1)} := (T_0^{(j)} | T_{w-1..1}^{(j-1)})$

Step 9: $T^{(e-1)} := (0 | T_{w-1..1}^{(e-1)})$

Step 10: $C := T$

Step 12: return C

Notice that this algorithm differs from the previous one only with the carry chain.

5.4.3 Multiple-Word Radix-3 Montgomery Multiplication Algorithm for $GF(3^m)$

Montgomery multiplication algorithms for $GF(p)$ and $GF(2^m)$ are similar to each other because they are both implemented in r'adix-2 and they only differ in the carry chain. Since we implement Montgomery multiplication algorithm for $GF(3^m)$ in radix-3, there have to be some changes in the implementation. We already explained the differences for the addition part in RSD representation and we showed that radix-2 and radix-3 representations can be both implemented on a single hardware.

For a modulus of m-bit precision and a word size of w-bits, $e = \lceil m + 1/w \rceil$ words are required. Since there is no carry computation in $GF(3^m)$ arithmetic, there won't be any extra digits used other than the variable vectors. Every digit is represented

by two bits in the hardware, one for positive and one for negative, since the numbers are in RSD representation. The algorithm scans the operand B word-by-word, and the operand A digit-by-digit. In radix-3 representation, the digits are marked with subscripts. For example, the ith digit of the kth word of B is represented as $B_i^{(k)}$. (x, y) represents a digit in RSD representation. The algorithm is given below:

Input: $A, B \in GF(3^m)$ and p

Output:$C \in GF(3^m)$

Step 1: $T := 0^m$

Step 2: for $i = 0$ to $m - 1$

Step 3: $T^{(0)} := a_i \cdot B^{(0)} + T^{(0)}$

Step 4: if $T_0^{(0)} = p_0^{(0)}$ then

Step 5: $T^{(0)} := T^0 - p^{(0)}$

Step 6: for $j - 1$ to $e - 1$

Step 7: $T^{(j)} := a_i \cdot B^{(j)} + T^{(j)} - p^{(j)}$

Step 8: $T^{(j-1)} := (T_0^{(j)} | T_{w-1..1}^{(j-1)})$

Step 9: else

Step 10: $T^{(0)} := T^0 + p^{(0)}$

Step 11: for $j - 1$ to $e - 1$

Step 12: $T^{(j)} := a_i \cdot B^{(j)} + T^{(j)} + p^{(j)}$

Step 13: $T^{(j-1)} := (T_0^{(j)} | T_{w-1..1}^{(j-1)})$

Step 14: $T^{(e-1)} := ((0,0)|T^{(e-1)}_{w-1..1})$

Step 15: $C := T$

Step 16: if $C < T$ then $C := C + p$

Step 17: return C

5.5 Multiplier Architecture

5.5.1 Pipeline Organization

All of the three different algorithms are concurrent in the loops, outer and inner loops with the variables i and j. Each processor unit is responsible for one step of the outer loop with the variable i. Each processor unit gets the a_i digit as input. Also, every processor unit gets $B^{(j)}$, $p^{(j)}$ and $T^{(j)}$ as inputs, according to the inner loop variable j. The pipelined structure is shown in Figure 5.5.

An important aspect of this pipeline organization is the register file design. The digits a_i of the multiplier A are given serially to the PUs, and are used only for one iteration of the outer loop. So they can be discarded immediately after use. Therefore, a simple shift register with a load input would be sufficient. Also, rather than registering the multiplier A in the hardware, we can have a serial input for every digit and we register the necessary a_i digits inside the hardware, whenever

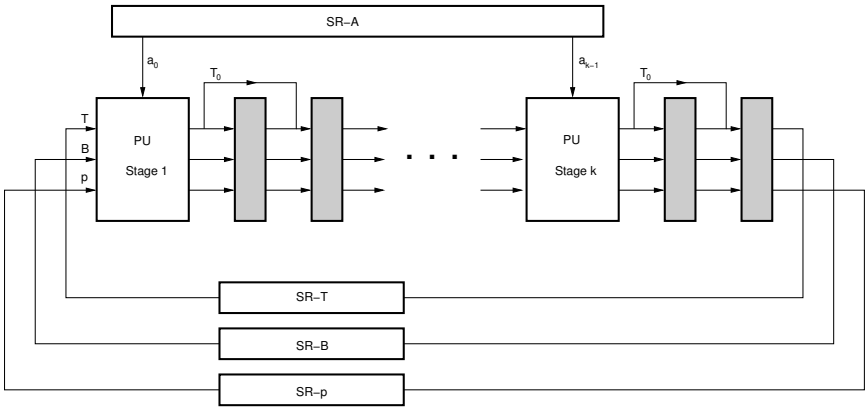


Figure 5.5: Pipeline organization

needed. This will reduce the area and power use of the hardware. The registers for the modulus p and multiplicand B can also be shift registers.

The multiplication starts with the first PU unit. It starts by processing the first iteration of the outer loop of the algorithms. As can be seen from the algorithm, enough data will be ready for the second iteration to start in 2 clock cycles. Therefore, the second PU has to be delayed from the first PU by 2 clock cycles. This is done by using two stages of registers in between. Also, these registers are handling the shift operations for the partial sum (Step 8 of the algorithms) as shown in Figure 5.5.

When the first PU is done with the operations of an iteration step of the outer loop, it starts working on the next available iteration loop, and the second PU will be done in 2 clock cycles and start working on the next available iteration, and this goes on for the entire pipeline organization. If there is no pipeline stall, which means if the first PU is done with an iteration when the last PU operated on an iteration

for clock cycles, there is sufficient enough PUs and no need for intermediate shift registers to hold the data. The pipeline can go on working without stalling. However, if the number of PUs is not sufficient enough, which means if pipeline stalls occur, the modulus and multiplicand words generated at the end of the pipeline have to be registered. $SR-T$, $SR-p$ and $SR-B$ are shift registers to hold these data when there is pipeline stall. The length of these shift registers are of crucial importance and are determined by the number of pipeline stages (k) and the number of words (e) in the modulus. The width of the shift registers is equal to w, the word size. The length of these registers can be given as

$$L = \begin{cases} e - 2 \cdot (k-1) & : \quad \text{if}(e+1) > 2k \\ 0 & : \quad \text{otherwise.} \end{cases}$$

The global control block was not mentioned since its function can be inferred from the algorithms.

5.5.2 Processing Unit

The processing unit consists of two layers of adder blocks, which we call *Unified Arithmetic Core*. The function of a Unified Arithmetic Core was described in section 5.3. It is capable of doing addition and subtraction operation in fields $GF(p)$, $GF(2^m)$ and $GF(3^m)$. The block diagram of a processing unit with the word size w=3 is shown in figure 5.6.

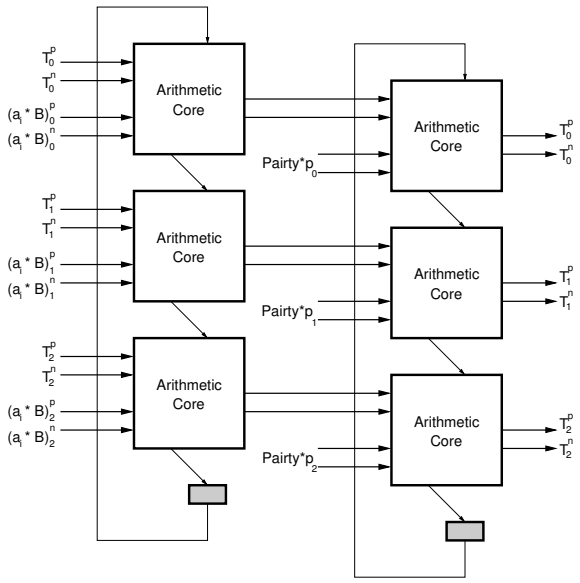


Figure 5.6: Processing Unit (PU) with $w = 3$.

As can be seen in the figure, a PU is responsible for performing the operation:

$$a_i \cdot B^{(j)} + T^{(j)} \pm p^{(j)}$$

This step is common for all the three fields, so this part of the PU is a very simple combination of the unified arithmetic cores. The inputs to these adders come from decoders designed to handle arithmetic in three different fields.

We need a simple logic for multiplying one digit a_i of the multiplier A with one word $B^{(j)}$ of the multiplicand B (for the part $a_i \cdot B^{(j)}$). Since a_i can only have the values $(0, 0)$, $(1, 0)$ or $(0, 1)$, the result of $a_i \cdot B^{(j)}$ can be 0, $B^{(j)}$ or $-1 \cdot B^{(j)}$ respectively. Negating an integer is simply swapping positive and negative bits of digits, so a simple and small special encoder would be sufficient for this.

We need another logic to find the parity for every iteration of the outer loop. We check the right-most digit of the modulus, (p_0^0) and the right-most digit of the operation $T^{(0)} = a_0 \cdot B^{(0)} + T^{(0)}$, T_0^0 and find the parity:

$$Parity = \begin{cases} (0,0) & : \quad \text{if} T_0^0 = (0,0) \\ (0,1) & : \quad \text{if} p_0^0 = T_0^0 \\ (1,0) & : \quad \text{otherwise} \end{cases}$$

So, this is very similar to the previous encoder logic we used. One difference is that since the parity is calculated only once for every iteration step, it needs to be registered after being calculated by the PU.

Chapter 6

Conclusion

In this thesis we demonstrated that scaled arithmetic, which is based on the idea of transforming a class of primes into special forms that enable efficient arithmetic, can be profitably used in elliptic curve cryptography. To this end, we implemented an elliptic curve cryptography processor using scaled arithmetic. Implementation results show that the use of scaled moduli in elliptic curve cryptography offers a superior performance in terms of area, power, and speed. We proposed a novel inversion algorithm for scaled moduli that results in an efficient hardware implementation. It has been observed that the inversion algorithm eliminates the need for projective coordinates that require prohibitively a large amount of extra storage. The successful use of redundant representation (i.e. carry-save notation) in all arithmetic operations including the inversion with the introduction of an innovative comparator design leads

to a significant reduction in critical path delay resulting in a very high operating clock frequency. The fact that the same data path (i.e. arithmetic core) is used for all the field operations leads to a very small chip area. Comparison with another implementation demonstrated that our implementation features desirable properties for resource-constrained computing environments.

We also implemented a Unified Multiplier Architecture for the extension fields $GF(p)$, $GF(2^m)$ and $GF(3^m)$. Considering the results we obtained from the previous architecture, we used a different number representation, Redundant Signed Digit representation. As a result we achieved the construction of a novel and low-power architecture for Montgomery multiplication algorithm.

Bibliography

[AMV93] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An Implementation of Elliptic Curve Cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

[Avi61] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Trans. Electron. Computers*, EC(10):389–400, September 1961.

[Ber68] E. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill, New York, NY, 1968.

[BF01] D. Boneh and M. Franklin. Identity-based Encryption from the Weil Pairing. In *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag, 2001.

[BGK+03] G. Bertoni, J. Guajardo, S. Kumar, G. Orlando, C. Paar, and T. Wollinger. Efficient GF(p^m) Arithmetic Architectures for Crypto-

graphic Applications. In *Topics in Cryptology - CT RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 158–175. Springer-Verlag, 2003.

[Cra92] R. E. Crandall. Method and Apparatus for Public Key Exchange in a Cryptographic System. U.S. Patent Number 5,159,632, October 1992.

[DH76] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[Kal95] B. S. Kaliski Jr. The Montgomery Inverse and its Applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.

[Kob87] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[Men93] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, MA, 1993.

[Mon85] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[Nat91] National Institute for Standards and Technology. Digital Signature Standard (DSS). *Federal Register*, 56:169, August 1991.

[PS02] D. Page and N. P. Smart. Hardware Implementation of Finite Fields of Characteristic Three. In B. S. Kaliski Jr., C. K. Koc, and C. Paar, editors, *Cryptographic Hardware and Embedded Sytems — CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 529–539. Springer-Verlag Berlin, 2002.

[SBM+02] R. Schroeppel, C. Beaver, R. Miller, R. Gonzales, and T. Draelos. A Low-Power Design for an Elliptic Curve Digital Signature Chip. In B. S. Kaliski Jr., C. K. Koc, and C. Paar, editors, *Cryptographic Hardware and Embedded Sytems — CHES 2002*, Lecture Notes in Computer Science, pages 366–380. Springer-Verlag Berlin, 2002.

[Sha85] A. Shamir. Identity-Based Cryptosystems and Signature Schemes. In *Advances in Cryptology - CRYPTO 1985*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer-Verlag, 1985.

[Sol99] J. A. Solinas. Generalized Mersenne Numbers. CORR-99-39, CACR Technical Report, University of Waterloo, 1999.

[SOOS95] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast Key Exchange with Elliptic Curve Systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO 95*, Lecture Notes in Computer Science, No. 973, pages 43–56. Springer-Verlag, 1995.

[STK00] E. Savas, A. F. Tenca, and C.K. Koc. A Scalable and Unified Multiplier Architecture for Finite Fields $gf(p)$ and $gf(2^m)$. In C. K. Koc and C. Paar, editors, *Cryptographic Hardware and Embedded Sytems — CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 277–292. Springer-Verlag, 2000.

[TK99] A. F. Tenca and C. K. Koc. A scalable architecture for Montgomery multiplication. In C. K. Koc and C. Paar, editors, *Cryptographic Hardware and Embedded Sytems*, Lecture Notes in Computer Science, No. 1717, pages 94–108. Springer, Berlin, Germany, 1999.

[TKL86] J. J. Thomas, J. M. Keller, and G. N. Larsen. The Calculation of Multiplicative Inverses over $GF(p)$ Efficiently where p is a Mersenne Prime. *IEEE Transactions on Computers*, 5(35):478–482, 1986.

[TKM04] E. Popovici T. Kerins and W. P. Marnane. Algorithms and Architectures for Use in FPGA Implementations of Identity Based Encryption Schemes. In *Field Programmable Logic and Applications*, volume 3203 of *Lecture Notes in Computer Science*, pages 74–83. Springer-Verlag, 2004.

[Wal92] C. D. Walter. Faster Modular Multiplication by Operand Scaling. In J. Feigenbaum, editor, *Advances in Cryptology — CRYPTO'91*, Lecture

Notes in Computer Science, No. 576, pages 313–323. Springer-Verlag, 1992.

Appendix A

Modulus Scaling

A table of special primes is given below. Each row lists all degrees up to $i = 1024$ for which a prime exists in the form specified at the beginning of the row.

PRIME	$0 < i < 1024$
$2^i + 1$	1, 2, 4, 8, 16
$2^i + 3$	1, 2, 3, 4, 6, 7, 8, 16, 12, 15, 16, 18, 28, 30, 55, 67, 84, 228, 390, 784
$2^i + 5$	1, 3, 5, 11, 47, 53, 141, 143, 191, 273, 341
$3 \cdot 2^i + 1$	1, 2, 5, 6, 8, 12, 18, 30, 36, 41, 66, 189, 201, 209, 276, 353, 408, 438, 534
$5 \cdot 2^i + 1$	1, 3, 7, 13, 15, 25, 39, 55, 75, 85, 127
$3 \cdot 2^i + 5$	1, 2, 3, 4, 5, 6, 7, 8, 14, 16, 19, 22, 24, 27, 29, 32, 38, 54, 57, 60, 76, 94, 132, 139, 175, 187, 208, 230, 379, 384, 632
$5 \cdot 2^i + 3$	1, 2, 3, 4, 5, 7, 8, 11, 12, 18, 20, 26, 28, 32, 34, 43, 44, 50, 52, 58, 65, 66, 107, 140, 197 274, 280, 380, 393, 506, 664, 738, 875, 944, 1016
$2^i - 1$	2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607
$2^i - 3$	3, 4, 5, 6, 9, 10, 12, 14, 20, 22, 24, 29, 94, 116, 122, 150, 174, 213, 221, 233, 266, 336, 452, 545, 689, 694, 850
$2^i - 5$	3, 4, 6, 8, 10, 12, 18, 20, 26, 32, 36, 56, 66, 118, 130, 150, 166, 206, 226, 550, 706, 810
$3 \cdot 2^i - 1$	1, 2, 3, 4, 6, 7, 11, 18, 34, 38, 43, 55, 64, 76, 94, 103, 143, 206, 216, 306, 324, 391, 458, 470, 827
$5 \cdot 2^i - 1$	2, 4, 8, 10, 12, 14, 18, 32, 48, 54, 72, 148, 184, 248, 270, 274, 420
$3 \cdot 2^i - 5$	2, 3, 4, 7, 9, 10, 13, 15, 25, 31, 34, 48, 52, 64, 109, 145, 162, 204, 207, 231, 271, 348, 444, 553, 559
$5 \cdot 2^i - 3$	1, 2, 3, 5, 6, 8, 9, 12, 17, 20, 27, 29, 30, 36, 62, 72, 83, 117, 119, 137, 149, 152, 176, 201, 243, 470, 540, 590, 611, 887, 996

Table 2: List of special primes up to degree 1024.

In the following table a list of scaled moduli of the form $2^k \pm 1$ is shown.The scaling factor and the prime modulus is provided in the same row.

Modulus	Scale	Prime Modulus (hexadecimal)
$2^{83}-1$	167	C4372F855D824CA58E9
$2^{92}+1$	17	F0F0F0F0F0F0F0F0F0F0F1
$2^{97}-1$	11447	B73493DECFD9B68318EF9
$2^{101}+1$	3	AAAAAAAAAAAAAAAAAAAAAAB
$2^{104}+1$	257	FF00FF00FF00FF00FF00FF01
$2^{107}+1$	1929	10FCAEA5E3998C02A77B49EB9
$2^{116}+1$	1009681	109DC950DA32FC88E84D688F1
$2^{127}+1$	3	2AAAAAAAAAAAAAAAAAAAAAAAAAAAAB
$2^{131}-1$	263	7C97D9108C2AD4329DB02EB8F166349
$2^{148}+1$	17	F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F1
$2^{167}+1$	3	2AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
$2^{179}-1$	514447	104E5A80A157457ABC6482776A0E7EE78C616DA91
$2^{191}+1$	3	2AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
$2^{197}-1$	7487	1181B149E3E4C85E5F1FB2507D481CB8C6DD39E358BAD41
$2^{199}+1$	3	2AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
$2^{233}+1$	39173361	DB47AE1104FD220D294905CAD4166DB817CE5936FBFBCAC5B411
$2^{281}-1$	80929	19E9D9CE852ACD5A5A35C4EAA034F0BFF8EA0E7187964BD94B554C27D831862B81F
$2^{313}+1$	3	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
$2^{356}+1$	17	F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F1

Table 3: Scaled moduli of the form $2^k \pm 1$.

Appendix B

Inversion Algorithm for Mersenne Primes of the Form $2^q - 1$

Algorithm X

Input: $a \in [1, p-1]$, p, and q where p is prime and $p = 2^q - 1$

Output: $b \in [1, p-1]$, where $b = a^{-1} \pmod{p}$

1: $(b, c, u, v) := (1, 0, a, p)$;

2: Find e such that $2^e || u$

3: $u := u/2^e$; // shift off trailing zeros

4: $b := (2^{q-e} b) \pmod{p}$; // circular left shift

5: if $u = 1$ return b;

6: $(b, c, u, v) := (b + c, b, u + v, u)$;

7: go to Step 2