International Journal of Computer Network and Security(IJCNS)
Vol 7. No.1 2015 Pp. 14-24

available at : www.ijcns.com
ISSN: 0975-8283

ELLIPTIC CURVE PROCESSOR ARCHITECTURE TO DESIGN UNIFIED MONTGOMERY MULTIPLIER ARCHITECTURE FOR GF (p), GF (2^m) AND GF (3^m)

D. Prasadh[1] and M. Muthulakshmi[2]
[1,2]Assistant Professor, Dept of Computer Science, SSBSTAS College, Tamilnadu.
[1]prasadh2000@gmail.com, [2]mlaxmi0305@gmail.com

ABSTRACT

This work 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 infinite 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 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 separate hardware for GF3^m.

Keywords: ***network security, cryptography, decryption, encryption***

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 contact less 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 [1] 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. [2] We take a closer look at Elliptic Curve Cryptosystems (ECC) as the most promising candidate for low-power implementations. [3] We implemented the hardware design of low-power and novel ECC architecture.

After a short introduction into the motivation of the work done in this thesis and a brief introduction to the modular arithmetic in Section 1, Section 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, modulus scaling techniques can be used for the background research. Also in this Section, the inversion algorithm that was achieved by modulus scaling techniques will be described and analyzed for hardware implementation. Section 4 presents the reader the system architecture of the design and 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, Section 5 concludes the paper and presents avenues for future work.

1.2 **Modular Arithmetic**

Modular arithmetic has a variety of applications in cryptography. Many public-key algorithms heavily depend on modular arithmetic. [4]Among these, RSA encryption and digital signature schemes, discrete logarithm problem (DLP) based schemes such as the [5] Diffie-Helman key agreement and El-Gamal encryption and signature schemes, and elliptic curve cryptography 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.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, [6] 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. Low Hamming-weight irreducible polynomials such as trinomials and pentanomials became a popular choice for both hardware and software implementations of ECDSA over $GF(2^k)$. Particularly, trinomials of the form xk + x + 1 allow efficient reduction. For many bit-lengths such polynomials do not exist; therefore less efficient trinomials, i.e. xk + xu + 1 with u > 1, or polynomials, i.e. xk + xu + xv + xz + 1, are used instead. Hence, in many cases the performance suffers degradation due to extra additions and alignment adjustments.

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.b, and then to reduce the product using the modulus, c = t mod p. Traditionally, the reduction step is

implemented by a division operation, which is significantly more demanding than the initial multiplication.[7]To alleviate the reduction problem in integer modular multiplications, Crandall proposed 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 tl represent the low k-bits and t_h represent the high k-bits:

$$t = th2k + tl$$

$$\text{Hence } c = th2k + tl \pmod p$$

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

$$c = th.u + t\,l \pmod{2k - u}.$$

It should be noticed that $t_h.u$ 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 $c = t_h + tl \bmod p$. Similarly primes of the form $2^k + 1$ may simplify reduction into a modular subtraction $c = tl - t_h \bmod p$. 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 2521 - 1 and 2607 - 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, [8] Solinas 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 restricts the constant u in $p = 2^k - u$ to a small constant that fits into a single word.

3. MODULUS SCALING TECHNIQUES

3.1 General Method

The idea of modulus scaling was introduced by Walter. [9] 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.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 mod m) mod p ≡ a mod p : When a scaled modulus is used, residues will be in the range [m – 1, 0] = [s.p – 1, 0]. The number is not fully reduced and essentially we are using a redundant representation where an integer is represented using dlog2 se 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 section 2.

3.2 Special Primes

We present two heuristics that form a basis for efficient on-the-y 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

(DDDD...DDD)B. (B - 1) = (DDDD...DDD0) B - (DDDD...DDD) B = (D000 ... 000D)B.

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 = (5123456781234567812345678123456781234567 8123456807)16.

By inspection we identify (12345678)16 as a repeating pattern. By selecting the base B = 232, the repeating pattern becomes a digit. The scaling factor is the largest digit s = B - 1 = 232 - 1 = (FFFFFFFF) 16. The scaled modulus is computed as

m = s.p= (5123456730000000000000000000000000000000 0000000085DCBA97F9)16

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

Prime

p = (577777777777733333333338B) 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:

$m = p.s = (577777777777733333333338B)_{16} .(F)_{16}$

= (51FFFFFFFFFFFC0000000000525)16.

= (52000000000000 40000000000525)16.

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, 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 B-1/D.

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

$(DDD...DDD)_B .B\text{-}1/D= ((B - 1) (B -1) (B - 1) ... (B - 1))_B= (1000 ... 01)_B.$

The following example shows the power of this simple technique.

Example 2 Let the prime p be

p = (D79435E50D79435D79435E50D79435E50D794 35E50D79435E50D79435E50‖ D79435E50D79435E50D79435E50D7943 5E5)16

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 = 236. Since D| (B - 1) the scaling factor is computed as

$$s = 2^{36} – 1/ (0D79435E5)16= 19$$

The scaled modulus becomes

$$m = s.p = 2^{384} - 2^{320} – 1$$

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 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

proposed the Algorithm X for Mersenne primes of the form $2^q - 1$.

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 := $-(2^{q-eb})$ (mod 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) ≠ 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 $b = a^{-1}.s \pmod m$. In order to obtain the inverse of a when gcd(a,m) ≠ 1, an extra multiplication at the end is necessary:

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

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. s :

When the inverse computation is completed we have the following equality

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

and thus

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

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

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

Therefore, the algorithm automatically yields the inverse of a as $b = a^{-1}$ if the input is taken as s.a mod 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. [10] 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 ϵ [1,m - 1], d ϵ [1,m - 1], m, and q where $m = 2^q \pm 1$

Output:b ϵ [1;m - 1], where b = d/a (mod m)

1: a := a . s (mod m);
2: (b, c, u, v) := (d, 0, a,m);
3: Find e such that $2^e \| u$
4: u := $u/2^e$; // shift of trailing zeros
5: b := $\overline{+}(2^{q-e}b) \pmod m$; // 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 (mod m) ≡ d/a (mod 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 disappear 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

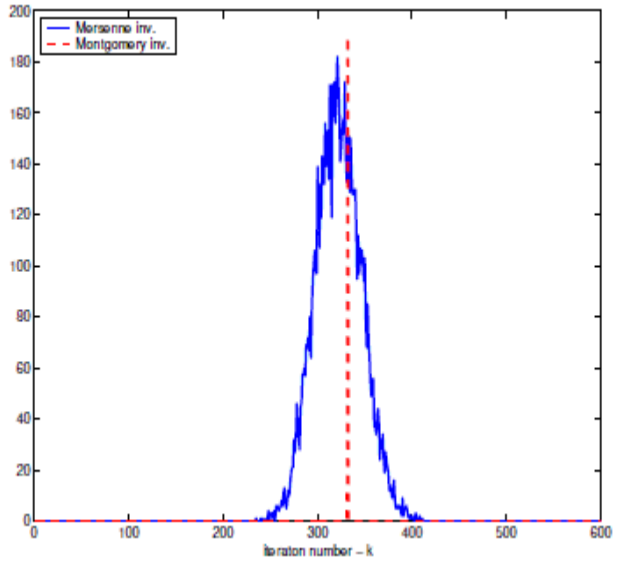

Figure: 1 Distribution of k

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 numbers of iterations are usually of random nature (but demonstrate 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 1).

4. THE ELLIPTIC CURVE ARCHITECTURE

We developed an elliptic curve architecture using the scaled modulus technique and our specialized inversion algorithm [11]. 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 $GF((2^{167}+1)/3)$. [12]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 section 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. For power efficiency we optimize our design to include minimal hardware. An effiective 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 Arithmetic Unit

The arithmetic unit is built around four main registers R0; R1; R2; R3, and two extra registers Rtemp0; Rtemp1 which are used for temporary storage [13]. 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 briey outline the implementation of basic arithmetic operations.

4.2.1 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

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

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

4.2.2 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

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

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 ks and kc denote the sum and carry parts of k, respectively. Since k = ks + kc then -k = -ks - kc = (k's + 3) + (k'c + 3) =ks'+kc'+6 mod m. Therefore the constant value 6 has to be added to the complements of the carry and sum registers in order to compute -k.

4.2.3 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 overdo. Therefore, the result needs to be reduced modulo m if the leftmost bit of the register R0 is 1.

4.2.4 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. **$b = (-2^{q-e} . b) \bmod m$**

   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 $b = 2^{-e} . b \pmod m$. 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 Figure3.

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.

4.3 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. [14] 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 1.5.168 = 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 $E(e) = \sum^{i=1to\infty} i(1/2)^i = 2$. 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)←(1,0,a,m)
2: Shift off all trailing zeros and rotate b
u ← u >>e b←b>>e (mod m)
3: Check terminate condition
If u = s return b
4: Update variables
(b,c,u,v) ← (b+c,b,u+v,u);
go back to Step 2

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.

4.4 Results and Comparison

The presented architecture was developed into Verilog modules and synthesized using the Synopsys tools Design Compiler and Power Compiler [15]. In the synthesis we used the TSMC 0:13 µm ASIC library, which is characterized for power. The resulting architecture was synthesized for three operating frequencies. The implementation

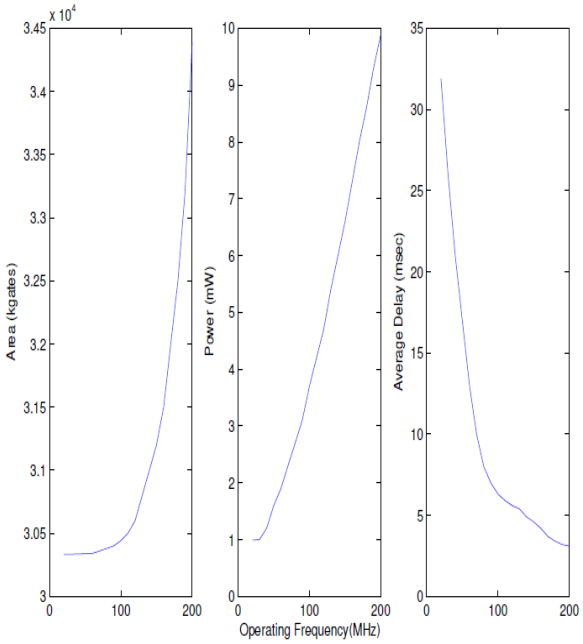

Figure 2: Implementation Results

results are shown in Figure 2. As seen in the figure the area varies around 30 K gates.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. [16] However, a point multiplication takes now only 3.1 msec. We performed a research to obtain the results 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. 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 [17] power optimization. The main contribution was the clever minimization of the gate logic through efficient tower field arithmetic. Note that their design includes 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. [18] 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.

5.CONCLUSION

In this paper 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 result 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. [18] 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.

References

[1] E. Savas, A. F. Tenca, and C.K. Koc. A Scalable and Unied Multiplier Architecture for Finite Fields gf(p) and gf(2m). 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.

[2] R. E. Crandall. Method and Apparatus for Public Key Exchange in a Cryptographic System. U.S. Patent Number 5,159,632, October 1992.
[3] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An Implementation of Elliptic Curve Cryptosystems over F2155 . IEEE Journal on Selected Areas in Communications, 11(5): 804{813, June 1993.
[4] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. IRE Trans. Electron. Computers, EC(10):389{400, September 1961.
[5] W. Di_e and M. E. Hellman. New Directions in Cryptography. IEEE Transactions on Information Theory, 22:644{654, November 1976.
[6] P. L. Montgomery. Modular multiplication without trial division. Mathe-matics of Computation, 44(170):519{521, April 1985.
[7] J. A. Solinas. Generalized Mersenne Numbers. CORR-99-39, CACR Technical Report, University of Waterloo, 1999.
[8] B. S. Kaliski Jr. The Montgomery Inverse and its Applications. IEEE Transactions on Computers, 44(8):1064{1065, 1995.
[9] N. Koblitz. Elliptic Curve Cryptosystems. Mathematics of Computation, 48(177): 203 209, January 1987.
[10] A. J. Menezes. Elliptic Curve Public Key Cryptosystems. Kluwer Academic Publishers, Boston, MA, 1993.
[11] P. L. Montgomery. Modular multiplication without trial division. Mathe-matics of Computation, 44(170):519{521, April 1985.
[12] 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.
[13] 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.
[14] 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.
[15] 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.
[16] J. J. Thomas, J. M. Keller, and G. N. Larsen. The Calculation of Multiplicative Inverses over GF(p) E_ciently where p is a Mersenne Prime. IEEE Transactions on Computers, 5(35):478,482, 1986.
[17] 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.
[18] E. Berlekamp. Algebraic Coding Theory. McGraw-Hill, New York, NY, 1968.
[19] 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.