Low-Power Elliptic Curve Cryptography Using Scaled Modular Arithmetic

E. Öztürk[1], B. Sunar[1], and E. Savaş[2]

[1] Department of Electrical & Computer Engineering, Worcester Polytechnic Institute, Worcester MA, 01609, USA,
erdinc, sunar@wpi.edu
[2] Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey TR-34956
erkays@sabanciuniv.edu

Abstract. We introduce new modulus scaling techniques for transforming a class of primes into special forms which enables 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 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.

1 Introduction

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 [4] and El-Gamal encryption and signature schemes [8], and elliptic curve cryptography [6, 7] 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 [2, 10]. Low Hamming-weight irreducible polynomials such as trinomials and pentanomials became a popular choice [10, 1] 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 paper 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 [3], 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. 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.

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 [3] 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:

$$\begin{aligned} t &= t_h 2^k + t_l \\ c &= t_h 2^k + t_l \pmod{p} \\ c &= t_h \cdot u + t_l \pmod{2^k - u} \ . \end{aligned}$$

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. 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 $2^k + 1$ 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 [11] 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 [3] restricts the constant u in $p=2^k-u$ to a small constant that fits into a single word.

3 Modulus Scaling Techniques

The idea of modulus scaling was introduced by Walter [13]. 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 Section 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. We present two heuristics that form a basis for efficient on-the-fly scaling:

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.

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.

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.

$$\begin{aligned}(DDD\ldots DDD)_B \cdot \frac{B-1}{D} &= ((B-1)(B-1)(B-1)\ldots(B-1))_B \\ &= (1000\ldots 0\bar{1})_B.\end{aligned}$$

We provide two examples for the heuristics in Appendix A. 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.

4 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 [5] 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. [12] 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 $u = 1$ 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 $u = \gcd(a, m) = s$ 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: $\bar{a} = a \cdot s$. When the inverse computation is completed we have the following equality

$$\bar{a} \cdot b + m \cdot k = s$$

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

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 5 of Algorithm X can be performed using simple circular left shift operations. The advantage of performing the Step 5 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 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.

5 The Elliptic Curve Architecture

We build our elliptic curve scheme over the prime field $GF((2^{167} + 1)/3)$. This particular prime allows us to utilize a very small scaling factor $s = 3$. To implement the field operations we use Algorithm X as outlined in Section 4. 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 provide a significant amount of reduction in power and circuit area since projective coordinates require a large number 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 [7].

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 build 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 focus 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. Since we lack the space we do not discuss the control circuit any further but focus on the basic functionality and describe the innovations in the arithmetic core.

The arithmetic unit shown in Figure 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.

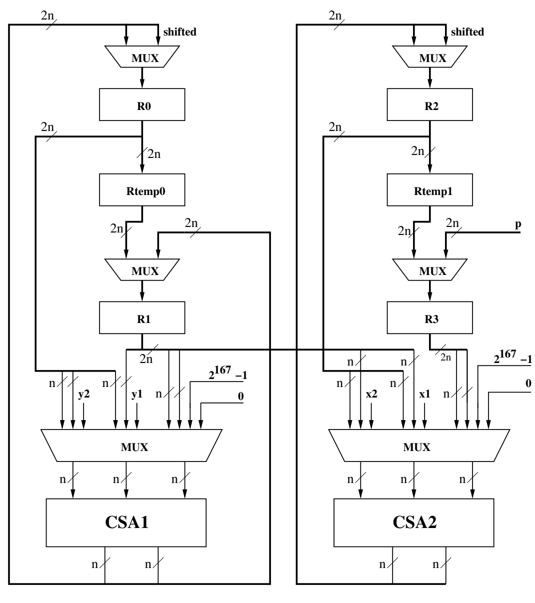

Figure 1: Block diagram of the arithmetic unit.

We briefly outline the implementation of basic arithmetic operations as follows:
Comparison Comparing 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 a constant value of $s = 3$. Such a comparator may be built using a massive OR tree with $2k$ inputs. Unfortunately, such an OR tree would cause serious latency ($O(\log_2 k)$ 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 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.

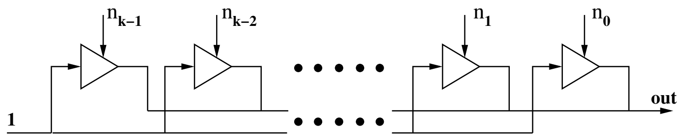

Figure 2: Comparator unit built using tri-state buffers.

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 $m - 2 = (11\ldots11111)_2$ (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 $m - 4 = (111\ldots11101)_2$ (167 bits) has to be added to the number and both of the leftmost bits are deleted.
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, 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 $-k$.

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.

Division To realize the division 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 division algorithm shown in Algorithm X has 5 steps:

1. Initialization: This is done by the control circuit. Load registers with $b = d, c = 0, u = a$ (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 5.
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.

6 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 division 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 division 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 division algorithm are reorganized in Figure 3 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 division algorithm. Our simulation results showed that the division algorithm would iterate on average about 320 times. The total time spent in division 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 $(b, c, u, v) \leftarrow (d, 0, a, m)$
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

Figure 3: Hardware algorithm for division.

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.

7 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 global operating voltage is 1 V. The resulting architecture was synthesized for three operating frequencies. The implementation results are shown in Table 1. As seen in the table 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.

Op. Freq. (MHz)	**Area (gates)**	**Power (mW)**	**Avg. Delay (msec)**
20	30,333	0.99	31.9
100	30,443	4.34	6.3
200	34,390	9.89	3.1

Table 1: Implementation Results.

We compare our design with another customized low-power elliptic curve implementation presented by Schroeppel et al. in CHES 2002 [9]. 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.

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.

8 Conclusions

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 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.

References

1. 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.

2. E. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill, New York, NY, 1968.
3. R. E. Crandall. Method and Apparatus for Public Key Exchange in a Cryptographic System. U.S. Patent Number 5,159,632, October 1992.
4. W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.
5. B. S. Kaliski Jr. The Montgomery Inverse and its Applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.
6. N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.
7. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic Publishers, Boston, MA, 1993.
8. National Institute for Standards and Technology. Digital Signature Standard (DSS). *Federal Register*, 56:169, Auguts 1991.
9. 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.
10. 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.
11. J. A. Solinas. Generalized Mersenne Numbers. CORR-99-39, CACR Technical Report, University of Waterloo, 1999.
12. 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.
13. 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

Example 1. We select the following prime

$$p = (\texttt{5123456781234567812345678123456781234567812345678}07)_{16}.$$

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

$$\begin{aligned} m &= s \cdot p \\ &= (\texttt{51234567300000000000000000000000000000000000000085DCBA97F9})_{16} \end{aligned}$$

The representation may contain more than one repeating digit. For instance, the prime $p = (\mathtt{57777777777777333333333338B})_{16}$ 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:

$$\begin{aligned} m &= p \cdot s \\ &= (\mathtt{52000000000000\bar{4}000000000525})_{16}. \end{aligned}$$

Example 2. Let the prime p be

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

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

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	AAAAAAAAAAAAAAAAAAAAAAAAB
$2^{104}+1$	257	FF00FF00FF00FF00FF00FF01
$2^{107}+1$	1929	10FCAEA5E3998C02A77B49EB9
$2^{116}+1$	1009681	109DC950DA32FC88E84D688F1
$2^{127}+1$	3	2AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
$2^{131}-1$	263	7C97D9108C2AD4329DB02EB8F166349
$2^{148}+1$	17	F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F1
$2^{167}+1$	3	2AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
$2^{179}-1$	514447	104E5A80A157457ABC6482776A0E7EE78C616DA91
$2^{191}+1$	3	2AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
$2^{197}-1$	7487	1181B149E3E4C85E5F1FB2507D481CB8C6DD39E358BAD41
$2^{199}+1$	3	2AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
$2^{233}+1$	39173361	DB47AE1104FD220D294905CAD4166DB817CE5936FBFBCAC5B411
$2^{281}-1$	80929	19E9D9CE852ACD5A5A35C4EAA034F0BFF8EA0E7187964BD94B554C27D831862B81F
$2^{313}+1$	3	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
$2^{356}+1$	17	F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F1

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

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