Efficient Unified Arithmetic for Hardware Cryptography

Erkay Savaş[1] and Çetin Kaya Koç[2]

[1] Sabanci University, erkays@sabanciuniv.edu
[2] Oregon State University, koc@cryptocode.net

The basic arithmetic operations (i.e. addition, multiplication, and inversion) in finite fields, $GF(q)$, where $q = p^k$ and p is a prime integer, have several applications in cryptography, such as RSA algorithm, Diffie-Hellman key exchange algorithm [1], the US federal Digital Signature Standard [2], elliptic curve cryptography [3, 4], and also recently identity based cryptography [5, 6]. Most popular finite fields that are heavily used in cryptographic applications due to elliptic curve based schemes are prime fields $GF(p)$ and binary extension fields $GF(2^n)$. Recently, identity based cryptography based on pairing operations defined over elliptic curve points has stimulated a significant level of interest in the arithmetic of ternary extension fields, $GF(3^n)$.

Even though the aforementioned three popular finite fields are dissimilar mathematical structures, their elements are represented using similar data structures inside the digital circuits and computers. Furthermore, similarity of algorithms for basic arithmetic operations in these fields allows a unified module design. For example, the steps of the original Montgomery multiplication algorithm [7], which is one of the most efficient methods for multiplication in finite fields, $GF(p)$ and rings slightly differ from those of the Montgomery multiplication algorithm for binary extension fields, $GF(2^n)$ given in [8]. In addition, it is almost straightforward to extend the Montgomery multiplication algorithm for ternary extension fields, $GF(3^n)$, by essentially keeping the steps of the algorithm intact. Similarly, addition or inversion operations can be performed using similar algorithms that can be realized together in the same digital circuit.

To summarize, an arithmetic module which is versatile in the sense that it can be adjusted to operate in more than one of the three fields is feasible, provided that this extra functionality does not lead to an excessive increase in area and dramatic decrease in speed. Quite contrarily, a *unified* module that is capable of performing arithmetic in more than one field in the same, unified datapath brings about many advantages, one of which is the improved $\{\text{area} \times \text{time}\}$ product.

1 Fundamentals of Extension Fields

The elements of the prime finite field $GF(p)$ are the integers $\{0, 1, 2, ..., p-1\}$ where p is an odd prime. The addition and multiplication operations in $GF(p)$ are modular operations performed in two steps:

1. regular integer addition or multiplication, and
2. reduction by the prime modulus p if the result of the first step is greater than or equal to the modulus.

The elements of the binary extension field $GF(2^n)$ can be represented as binary polynomials of degree less than n if polynomial basis representation is used. Analogous to the odd prime used in $GF(p)$, a binary irreducible polynomial of degree n is used to construct $GF(2^n)$. The addition in $GF(2^n)$ is simply performed by modulo-2 addition of corresponding coefficients of two polynomials. Since it is basically a polynomial addition there is no carry propagation and the degree of the resulting polynomial cannot exceed $n-1$. On the other hand, multiplication in $GF(2^n)$ is more complicated and sometimes it is beneficial to use other type of representation techniques than standard polynomial basis such as Gaussian normal basis [9]. Here, we always use polynomial basis for $GF(2^n)$ because of its suitability to the unified architecture.

Polynomial basis representation of $GF(2^n)$ is determined by an irreducible binary polynomial $p(x)$ of degree n. Given $p(x)$, all the binary polynomials of degree less than n, which has the form $A(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$, are elements of $GF(2^n)$. Multiplication in $GF(2^n)$, similar to multiplication in $GF(p)$, is performed in two steps:

1. polynomial multiplication followed by
2. a polynomial division of the result from Step 1 by the irreducible polynomial $p(x)$.

Similar to binary extension fields, the elements of ternary extension fields $GF(3^n)$ can be represented as (ternary) polynomials of degree at most $n-1$, whose coefficients are from the base field $GF(3)$. In order to utilize polynomial basis for ternary arithmetic, an irreducible *ternary* polynomial $p(x)$ of degree n is needed. The addition operation in $GF(3^n)$ is polynomial addition where the corresponding coefficients of two ternary polynomials are added modulo-3 and there is no carry propagation. The multiplication is also done in two steps: a polynomial multiplication followed by reduction by the irreducible ternary polynomial $p(x)$.

2 Addition and Subtraction

The most fundamental arithmetic operation in finite fields and rings, on which all other arithmetic operations are based, is the addition operation. The key point to an efficient finite field arithmetic is to design fast and light-weight

adder circuits. In many cryptographic applications in order to balance the speed and area efficiency, adders utilizing redundant representation are preferred. The most basic form of redundant representation is the carry-save form in which an integer is represented as the sum of two other integers, namely $x = x_C + x_S$ where x_C and x_S are known as carry and sum components of the integer, respectively. The addition operation for carry-save representation can then be performed using full-adders which have three binary inputs and two binary outputs. Full-adders connected to each other in cascaded fashion can perform addition where one of the operands are in redundant form while the other in non-redundant form.

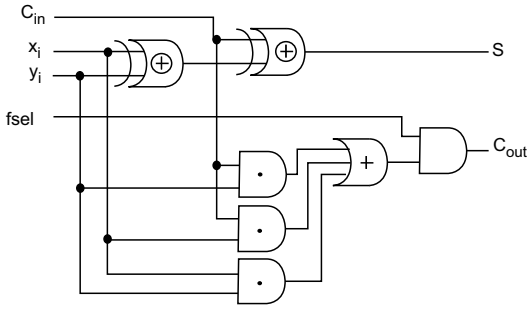

Fig. 1. The dual-field adder circuit

It is possible to perform both $GF(p)$ and $GF(2^n)$ addition operation using so-called dual-field adder (DFA) [12], which is illustrated in Figure 1. DFA shown in Figure 1 is basically a full-adder equipped with the capability of performing bit addition both with and without carry. It has an input denoted as **fsel** that provides this functionality. When **fsel** $= 1$, the dual-adder circuit performs bit-wise addition with carry which enables the circuit operating in $GF(p)$-mode. When **fsel** $= 0$, on the other hand, the output **Cout** is forced to 0 regardless of the values of the inputs. Consequently, the output S produces the result of modulo-2 addition of three binary input values. At most only two of the three binary input values of DFA can have nonzero values in $GF(2^n)$-mode.

An important aspect of designing a DFA is not to increase the critical path delay (CPD) of the circuit, which otherwise would have a negative effect in the maximum applicable clock frequency; a situation is against the design goal of the unified modules. However, a small amount of overhead in area can be accommodated. Gate level realization of DFA shown in Figure 1 clearly demonstrates that there is no increase in the CPD since the two XOR gates dominate the CPD as in the case of a regular full adder. Area differs slightly due to one extra input, i.e. **fsel** and additional gates that are used to suppress the carry out in $GF(2^n)$-mode. However, this increase in area is very small, therefore tolerable, compared to two separate adders for $GF(p)$ and $GF(2^n)$

which would incur much more overhead in area if a non-unified approach were preferred.

As described above 3×2 adder arrays in cascade are in many cases sufficient since addition operation is mostly needed in multiplications where one of the operands is always in non-redundant form as in [10]. In this case, the carry-save form is only used during the multiplication for partial product and the result of the multiplication has to be converted to non-redundant form using a carry-propagation adder after the multiplication is completed. However, when the two operands are both in carry-save redundant form, then 3×2 adder arrays in cascade cannot be used for unified addition. Instead, 4×2 adder arrays are needed to operate on both operands of redundant form. Using 4×2 adder arrays eliminate the need a conversion after multiplication, which is especially useful in elliptic curve cryptography where there are many addition and subtraction operations in between multiplication operations.

Classical carry-save redundant representation method has one major drawback due to the difficulty of performing subtraction operation. When two's complement representation is used to facilitate the representation of negative numbers as well as subtraction operation, the carry-save representation poses certain difficulties. For example, during the subtraction of two's complement operands, a carry overflow indicate whether the result is negative or positive. Since there can be a hidden carry overflow in carry-save representation, computationally intensive operations may be needed to determine the sign of the result, which in turn incurs significant increase in CPD and area.

Avizienis [11] proposed the redundant signed digit (RSD) representation to overcome this difficulty. Arithmetic in the RSD representation is almost identical to carry-save arithmetic. An integer is still represented by two positive integers; however, this time the integer is now represented as the difference (as opposed to the sum in carry-save representation) of two other integers. An integer X, therefore, is represented by x^+ and x^-, where $X = x^+ - x^-$. As can easily be deduced from the definition of RSD, there is no need for two's complement representation to handle negative numbers and subtraction operation. The RSD is, thus, a more natural representation when both addition and subtraction operations need to be supported. This is indeed the case in elliptic curve cryptography and Montgomery multiplication and inversion algorithms. An additional benefit of RSD representation is the fact that the comparison operation in $GF(p)$-mode is now possible and efficient. Integer comparison in $GF(p)$-mode can be performed utilizing a subtraction operation. After subtracting one integer from the other, a sign test can be performed directly by checking the first nonzero bit in significant positions of the result. This is in general an easy method that can be implemented by masking the most significant bits to determine which number is greater.

Realization of RSD arithmetic is very similar to carry-save arithmetic. RSD arithmetic needs generalized full adders which are shown in Figure 2. As observable from Figure 2, GFA-0 is a conventional full adder. From the realization perspective, GFA-1, GFA-2 and GFA-3 are equivalent to GFA-0

realization in ASIC and thus there is no associated overhead in either CPD or area.

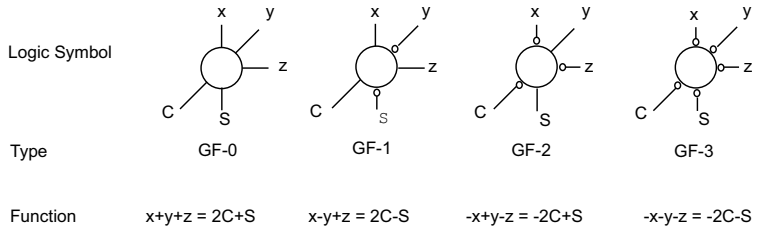

Fig. 2. Generalized full adders

The addition of two n-bit RSD integers, X and Y, $Z = X + Y$, can be done by cascading two layers of GFAs of types 1 and 2 as shown in Figure 3. An additional circuitry is needed to force the digit instances of (1, 1) to (0, 0) since $1 - 1 = 0$. Subtraction of two n-bit integers, $T = X - Y$ can be realized using the same addition circuit in Figure 3 by swapping y^+ and y^-. The adder (or subtractor) circuit which is originally designed for $GF(p)$ arithmetic can easily be converted into a dual-field adder (or subtractor) by forcing the carry output of each GFA into 0 in $GF(2^n)$-mode.

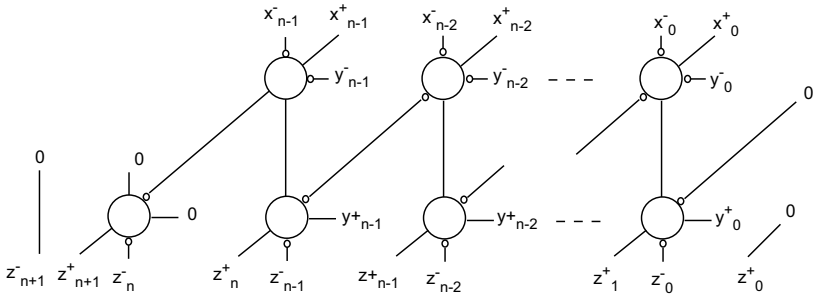

Fig. 3. Addition circuit with GFAs for two n-bit operands in RSD form

One of the side benefits of RSD representation and associated adder structures is their suitability to a full unified arithmetic that incorporates addition/subtraction in three major finite fields, namely $GF(p)$, $GF(2^n)$ and $GF(3^n)$. Below is the RSD representation of elements of these three fields:

- **Prime field** $GF(p)$: Elements of prime fields can be represented as integers in binary form. Assuming that the digits are signed, the values that digits have and their corresponding representations are $\{0, 1, -1\}$ and $\{(0, 0), (1, 0), (0, 1)\}$.

- **Binary extension field** $GF(2^n)$: A common practice is to consider elements of binary extension field as polynomials with coefficients from $GF(2)$. This allows to represent $GF(2^n)$ elements by simply arranging the coefficients of the polynomial into a binary string. A digit in $GF(2^n)$-mode can take the values of 1 and 0, that can be represented as $\{(0,0),(1,0)\}$.
- **Ternary extension field** $GF(3^n)$: Elements of ternary extension fields can be considered as polynomials whose coefficients are from $GF(3)$. Thus, each coefficient can take the values $-2,-1,-,1,2$. The digit values -2 and 2 are congruent to 1 and -1 modulo 3, respectively. Therefore, the RSD representations for possible coefficient values of 0, 1, and -1 are $\{(0,0),(1,0),(0,1)\}$.

A unified adder that operates in three fields can be derived from the addition circuit in Figure 3. When compared to $GF(p)$-only adder, the unified adder circuit has only marginally higher CPD while the overhead in area can be higher. However, when the area cost of three non-unified adders implemented in separate datapath far outweighs this overhead in the unified design as shown in [24].

3 Multiplication

In this section, we firstly provide the original unified Montgomery multiplication algorithm in [12], which operates only in $GF(p)$ and $GF(2^n)$. We then present a dual-radix unified multiplier in [13] where the multiplier calculates faster in $GF(2^n)$-mode than in $GF(p)$-mode. We finally discuss the support in the unified multiplier for multiplication in $GF(3^n)$.

3.1 Montgomery Multiplication Algorithm

In [7], Montgomery described a modular multiplication method which proved to be very efficient in both hardware and software implementations. An obvious advantage of the method is the fact that it replaces division operations with simple shift operations. The method adds multiples of the modulus rather than subtracting it from the partial result. And opposite to the subtraction of modulus in the regular modular multiplication which can be performed after all the digits of the multiplicand are processed, the addition operation can start immediately after the least significant digit of the multiplicand is processed. Especially the second feature accounts for the inherent concurrency in the algorithm. Refer to [7, 14, 15] for detailed explanation of the algorithm.

Given two integers a and b, and a prime modulus p, the Montgomery multiplication algorithm computes $\bar{c} = \textbf{MonMult}(a,b) = a \cdot b \cdot R^{-1} \pmod{p}$ where $R = 2^n$ and $a, b < p < R$ and p is an n-bit prime number. The Montgomery multiplication does not directly compute $c = a \cdot b \pmod{p}$, therefore certain transformation operations must be applied to the operands

a and b before the multiplication and to the intermediate result $\bar{c}$ in order to obtain the final result c. These transformations are applied as in the following example:

$$\bar{a} = \mathbf{MonMult}(a, R^2) = a \cdot R^2 \cdot R^{-1} \pmod{p} = a \cdot R \pmod{p},$$

$$\bar{b} = \mathbf{MonMult}(b, R^2) = b \cdot R^2 \cdot R^{-1} \pmod{p} = b \cdot R \pmod{p},$$

$$c = \mathbf{MonMult}(\bar{c}, 1) = c \cdot R \cdot R^{-1} \pmod{p} = c \pmod{p}.$$

Provided that $R^2 \pmod{p}$ is precomputed and saved, we need only a single **MonMult** operation to carry out each of these transformations. However, because of these transformation operations, performing a single modular multiplication using **MonMult** might not be advantageous even though there is an attempt to make it efficient for a few modular multiplications by eliminating the need for these transformations [16]. Its advantage, on the other hand, becomes obvious in applications requiring multiplication-intensive calculations such as modular exponentiation, elliptic curve point operations, and pairing calculations over elliptic curve points.

The Montgomery multiplication algorithm with radix-2^k for $GF(p)$ can be given as in the following:

Algorithm A

Input:	$a, b \in [1, p-1]$, p, and m
Output:	$c \in [1, p-1]$
Step 1:	$c := 0$
Step 2:	for $i = 0$ to $m-1$
Step 3:	$q := (c_0 + a_i \cdot b_0) \cdot (p'_0) \pmod{2^k}$
Step 4:	$c := (c + a_i \cdot b + q \cdot p)/2^k$

where $p'_0 = 2^k - p_0^{-1} \pmod{2^k}$. In the algorithm, the multiplier a is written with base (radix)-2^k as an array of digits a_i so that $a = \sum_{i=0}^{m-1} a_i \cdot 2^{k \cdot i}$, where m is the number of digits in a and $m = \lceil n/k \rceil$. In Step 4, the multiplicand b, the modulus p, and the partial result c enter the computations as full-precision integers. However, in the real implementations b, p, and c can be treated as multi-word integers in order to design a scalable multiplier and in each clock cycle one word of these values will be processed. One may also consider this representation as writing the multiplicand, the modulus and the partial result with digits $b^{(j)}$, $p^{(j)}$, and $c^{(j)}$ of w bits, so that $b = \sum_{j=0}^{e-1} b^{(j)} \cdot 2^{w \cdot j}$, $p = \sum_{j=0}^{e-1} p^{(j)} \cdot 2^{w \cdot j}$, and $c = \sum_{j=0}^{e-1} c^{(j)} \cdot 2^{w \cdot j}$ where $e = \lceil n/w \rceil$. Note that the base-2^w used to represent b, p, and c in Step 4 is different from the radix-2^k used to represent the multiplier a in Step 3. Note also that q, c_0, b_0, and p'_0 are all k-bit integers.

In order to avoid a possible confusion due to the usage of two different bases, we elect to refer the digits of b, p and c as words when implementing Step 4, and use the term *digit* exclusively for the multiplier a, and for b_0,

p'_0, and c_0 in Step 3 when they are in the same equation with the digits of a. Digits can be easily distinguished by the subscript notation (e.g. a_i or b_0) from superscript notation of word (e.g. $b^{(j)}$). We will also use the notation $x_{i,j}$ to denote the jth bit in the ith digit of x.

In addition, the radix of the multiplier architecture is determined by the base used to represent the multiplier a.

The Montgomery multiplication algorithm for $GF(2^n)$ is given below:

Algorithm B

Input:	$a(x), b(x)$, $p(x)$, and m
Output:	$c(x)$
Step 1:	$c(x) := 0$
Step 2:	for $i = 0$ to $m - 1$
Step 3:	$q(x) := (c_0(x) + a_i(x) \cdot b_0(x)) \cdot p'_0(x) \pmod{x^k}$
Step 4:	$c(x) := (c(x) + a_i(x) \cdot c(x) + q(x) \cdot p(x))/x^k$

where $p'_0(x) = p_0^{-1}(x) \pmod{x^k}$. As one easily observes, the two algorithms are almost identical except that the addition operation in $GF(p)$ becomes a bitwise modulo-2 addition in $GF(2^n)$. Although the operands are integers in the former algorithm and binary polynomials in the latter, the representations of both are identical in digital systems. In Algorithm A, there must be an extra reduction step at the end to reduce the result into the desired range if it is greater than the modulus. On the other hand, this step is not essential part of the algorithm and there are simple conditions that can be added to the algorithm in order to eliminate it [17, 18], hence we intentionally exclude it from the algorithm definitions.

One can also observe that the computations performed in Step 3 are of different nature in two algorithms and depending on the magnitude of the radix used, the part of the circuit in charge of implementing them might become very complicated. However, one can easily demonstrate that these computations can be performed in a unified circuitry for small radices.

From this point on, we will only use the notation introduced in Algorithm A for both $GF(p)$ and $GF(2^n)$ and leave polynomial notation completely out of our representation of field elements in $GF(2^n)$. Operations will be deduced from the mode ($GF(p)$ or $GF(2^n)$) in which the module is operated. The elements of both fields are represented identically in the digital systems.

Processing Unit

In this section, we explain the design details of the processing unit (PU) with radix-2, which is basically responsible for performing Step 3 and Step 4 of Algorithm A:

Step 3:	$q := (c_0 + a_i \cdot b_0) \cdot (p'_0) \pmod{2^k}$
Step 4:	$c := (c + a_i \cdot b + q \cdot p)/2^k$

Since we use radix-2 for our unified multiplier for sake of simplicity (noting that it is always possible to extend it to higher radices), the least-significant bits (LSB) of the operand digits, a_i, b_0, and c_0 will determine which one of the values in $\{0, b, p, b + p\}$ is added to the partial result c. In Figure 4, the architecture of the processing unit (PU) used in the unified multiplier with $w = 2$ is illustrated. The first layer of dual-field adder deals with the addition of b to the partial result c while the second layer does with the addition of p. The value q (binary for radix-2) calculated in Step 3 of Algorithm A determines whether the modulus p is added while the value a determines whether the multiplicand b is added to the partial result.

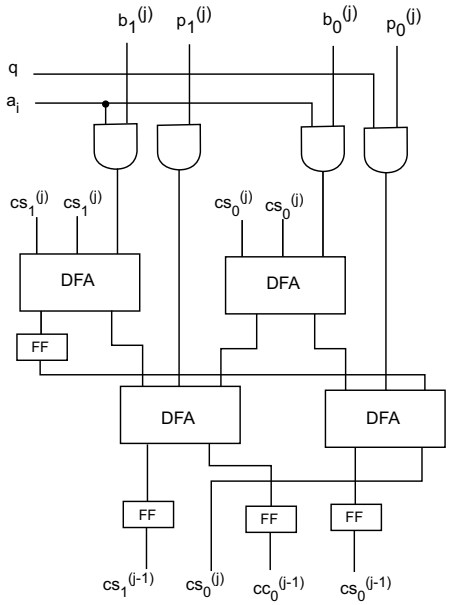

Fig. 4. Processing unit with radix-2 where word size $w = 2$

As can be observed from Figure 4, there are flip-flops (FF) to delay some of the bit values generated during the calculations. The FF right after the first dual-field adder layer delays the most significant bit of carry from the previous word to the current word. One can think of this bit as *carry-out* from the previous word since the carry part of c is one bit shifted to the left relative to the sum part in the carry-save form. The particular arrangement of FFs at the output of the second dual-field adder layer implements right-shift operation in Step 4 of Algorithm A.

The unified architecture consists of one or (generally) more processing units (PU), identical to the one shown in Figure 4, organized in a pipeline. Each PU takes a digit (k-bits) from the multiplier a, the size of which depends

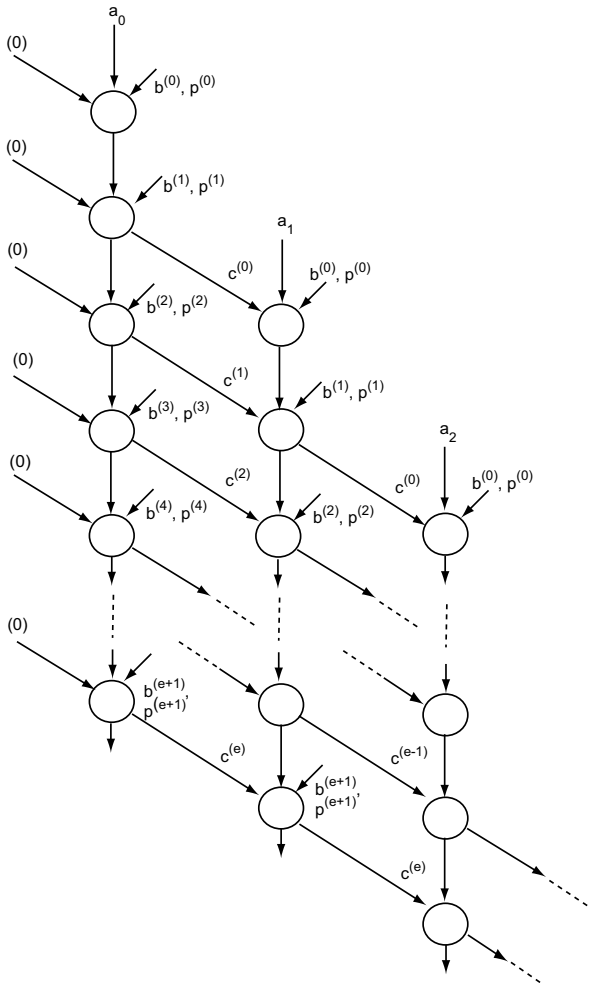

Fig. 5. Execution graph of Montgomery multiplication algorithm [10]

on the radix, and operates on the words of b, c and p successively starting from the least significant words. Starting from the second cycle it generates one word of partial result each cycle which is communicated to the next PU. After $e + 1$ clock cycles, where e is the number of words in the modulus (i.e. $e = \lceil n/w \rceil$), a PU finishes its portion of work and becomes free for further computation. When the last PU in the pipeline starts generating the partial results, the control circuitry checks if the first PU is available. If the first PU is still working on an earlier computation, the results from the last PU should be stored in a buffer until the first PU becomes available again. Refer to [10] for more information about the length of the buffer to store the partial results when there is no available PU in the pipeline. In Figure 5 the execution graph

of the Montgomery multiplication algorithm and dependencies between the processing units are illustrated.

Each column in the dependency graph represents the computation which is undertaken by a PU for one digit of the multiplicand a while each circle represents the operations for one word of p, b and c. The time advances from top to bottom where the operation represented by a circle takes exactly one clock cycle. An example of pipeline organization with t PUs is shown in Figure 6.

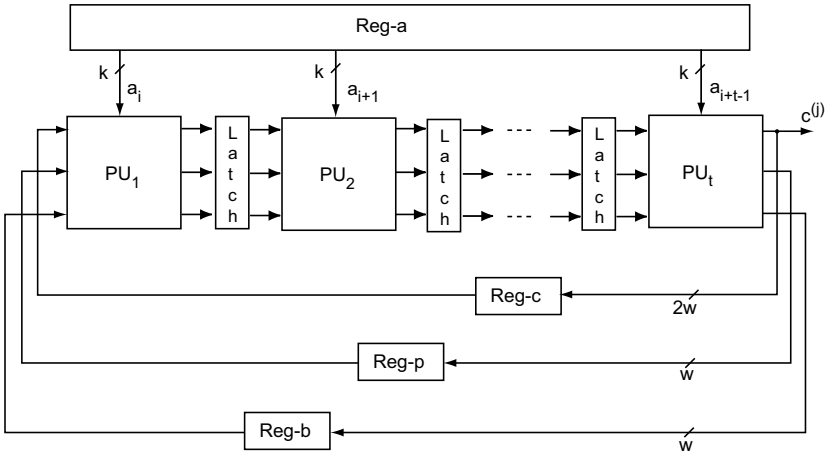

Fig. 6. Pipeline organization with two processing units

A redundant representation (carry-save) is used for the partial result in the architecture. Thus, for the partial result we can write $c = cc + cs$, where cc and cs stand for the carry and sum part of the partial result, respectively. In addition, one must note that the length of the register for partial result, c in Figure 6 is twice wider than the other registers.

Given that carry-save notation is used for the partial result and that each iteration is executed on word-by-word basis, the Algorithm A can be expressed as follows:

Algorithm A (modified)

Input:	$a, b \in [1, p-1]$, p, and m
Output:	$c \in [1, p-1]$, where $c = (cc, cs)$
Step 1:	$cc := 0, cs := 0$
Step 2:	for $i = 0$ to $m-1$
Step 3:	$q := (c_0 + a_i \cdot b_0) \cdot (p'_0) \pmod{2^k}$
Step 4:	for $j = 0$ to $e-1$
Step 5:	$(cc^{(j)}, cs^{(j)}) := (cc^{(j)} + cs^{(j)} + a_i \cdot b^{(j)} + q \cdot p^{(j)})/2^k$

The proposed architecture allows designs with different word lengths and pipeline organizations for different values of operand precision. In addition,

the area can be treated as a design constraint. Thus, one can adjust the design to the given area, and choose appropriate values for the word length and the number of pipeline stages, in accordance.

The propagation delay of PU is independent of word size w when w is relatively small (increases only slightly for larger values of w due to carry-free arithmetic), and thus we assume that the clock cycle is the same for all word sizes of practical interest. The area used by registers for partial sum, operands and modulus does not change with the word or digit sizes.

The proposed scheme yields the worst performance for the case $w = m$, since some extra cycles are introduced by PU in order to allow word-serial computation, when compared to other full-precision conventional designs. On the other hand, using many pipeline stages with small word size values brings about no advantage after certain point. Therefore, the performance evaluation reduces into finding an optimum organization for the circuit.

ASIC standard cell realizations of both unified and non-unified ($GF(p)$-only) designs demonstrate that area overhead of the unified multiplier is only 2.75% and that there is no overhead in critical path delay [12]. Therefore, the saving in the area is significant when the unified design is compared to a hypothetical architecture that has two separate datapath for $GF(p)$ and $GF(2^n)$ multipliers. Furthermore, this saving in area does not bring about a penalty in time performance, therefore improvement in area is identical to the improvement in metric of {area $\times$ time}.

3.2 Dual-Radix Multiplier

The original unified multiplier in [12] uses radix-2 design and offers an equal performance for both $GF(p)$ and $GF(2^n)$ of the same precision in terms of clock count. For this very reason, however, the original design is not optimized since it does not take the advantage of using $GF(2^n)$, which is, in general, more efficient than $GF(p)$ in hardware implementations. Our first observation is that this situation can be remedied by putting to use the part of the circuitry which is underutilized in $GF(2^n)$ mode. This allows us to run the multiplier module in higher radix values for $GF(2^n)$ than those for $GF(p)$ at the expense of using some amount of extra gates without significantly increasing the signal propagation time.

In this section, we present the radix-(2,4) multiplier architecture introduced in [13], where the multiplier uses radix-2 in $GF(p)$-mode while it uses radix-4 in $GF(2^n)$-mode. The radix-(2,4) multiplier is in fact the first member of the dual-radix multiplier family, which also includes radix-(4, 8) and radix-(8, 16) [13]. We only include the radix-(2,4) multiplier for the sake of simplicity in explaining.

Precomputation in Montgomery Multiplication Algorithm

The dual-radix unified multiplier architecture utilizes a precomputation technique in order to decrease the critical path delay of the original unified mul-

tiplier in [12]. Note that Step 4 of the Algorithm A computes

$$c := (c_0 + a_i \cdot b + q \cdot p)/2^k$$

where division by 2^k is simply a right shift by k bits and q is calculated in the previous step. Depending on the radix value chosen for the multiplier, the k-bit digit q can be determined by the least significant digits (LSD) of b, p and c, and the current digit of a. Similarly, the multiple of b that participates in the addition is determined solely by a_i. As a result, the LSDs of the operands, a_i, b_0, and c_0 will determine which one of the values in $\{0, b, p, b + p, 2p, 2b, 2b + 2p, \ldots\}$ is added to the partial result c. If one precomputes and stores the value of $b + p$, the calculations in Step 4 can be significantly simplified.

There are two implications of the precomputation technique. Firstly, the precomputed value must be stored, implying an increase in the register space. And secondly, there must be a so-called selection logic to select which multiples of b and p must participate in the addition in Step 4. The selection logic can be designed in such a way that it is parallel to PU and thus it results in no overhead in the critical path delay. On the other hand, the precomputation technique also simplifies the design since Step 4 can be performed with only one addition, once the selection logic generates its output.

Processing Unit

As pointed out earlier a processing unit (PU) is basically responsible for performing Step 3 and Step 4 of Algorithm A. Since the multiplier uses radix-2 for $GF(p)$, the least-significant bits (LSB) of the operand digits, a_i, b_0, and c_0 will determine which one of the values in $\{0, b, p, b+p\}$ is added to the partial result c. In the case of $GF(2^n)$, multiplication is performed in radix-4. Therefore, the LSDs (least significant digits) of b, p, and c and of the current digit of a are required in order to determine q. The LSB of p is always 1, then only $p_{0,1}$, the second least significant bit of the modulus, is included in the computations. Consequently, $a_{i,1}, a_{i,0}, b_{0,1}, b_{0,0}, c_{0,1}, c_{0,0}$ and $p_{0,1}$ determine one of the following values to be added to the partial result: $\{0, b, p, b + p, x \cdot b, x \cdot p, x \cdot (b + p)\}$ (Note that $a_{i,j}$ is the jth least significant bit of ith digit of a). Multiplication by x results in shifting one bit to the left, hence it is identical to multiplication by 2. Division by x^k and 2^k are identical operations and the latter is used to denote the right shift operation by k bits.

In Figure 7, the architecture of the processing unit (PU) used in the dual-radix multiplier is illustrated. The local control logic in Figure 7 contains the selection logic which generates the signals, to determine which multiples of b and p will be in the calculations. For example, The selection signal (1011) indicates that Step 4 will be $c := (c + 3b + 2p)/2^k$. The symbols cc_0 and cs_0 in Figure 7 represent the least significant digits of carry and sum part of the partial result c, respectively. Note that the cary part cc of the partial result is always 0 in $GF(2)$-mode. Similarly, in $GF(p)$-mode, the multiplexer on the right hand side always yields $cc^{(j)}$ since radix-2 is used in this mode.

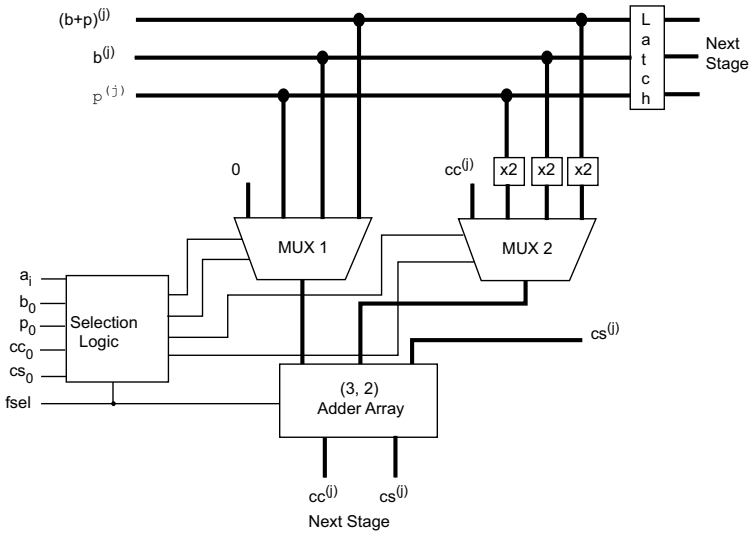

Fig. 7. Processing unit of dual-radix architecture with radix-2 for $GF(p)$ and radix-4 for $GF(2^n)$

3.3 Support for Ternary Extension Fields, $GF(3^n)$

The Montgomery multiplication algorithm for $GF(3^n)$, which is very similar to Algorithm B, is given below [24]:

Algorithm C

Input:	$a(x), b(x)$, $p(x)$, and m
Output:	$c(x)$
Step 1:	$c(x) := 0$
Step 2:	for $i = 0$ to $m - 1$
Step 3:	$q(x) := (c_0(x) + a_i(x) \cdot b_0(x)) \cdot p'_0(x) \pmod{x^k}$
Step 4:	$c(x) := (c(x) + a_i(x) \cdot c(x) + q(x) \cdot p(x))/x^k$

Only difference is due to the computation of $p'_0(x)$, which is $p'_0(x) = 2 \cdot p_0^{-1}(x) \pmod{x^k}$ (instead of $p'_0(x) = p_0^{-1}(x) \pmod{x^k}$ in Algorithm B).

Original unified multiplier architecture [12] utilizes two layers of (3×2) dual-field adder arrays to perform addition operations in Steps 3 and 4 of Montgomery multiplication algorithm. This is due to the fact that multiplicand (b or $b(x)$) and modulus (p or $p(x)$) are assumed to be always in non-redundant form. This assumption can hold for elliptic curve cryptography computations, where many multiplications are needed. If the result of a multiplication, which is produced in redundant form (e.g. carry-save representation), is needed for subsequent multiplications, it is immediately converted to non-redundant representation. In order to eliminate the need for conversion from redundant to non-redundant representation and associated circuitry, all

operands can be kept in redundant form throughout the entire elliptic curve computations (e.g. elliptic curve scalar point multiplication). This, however, requires using (4×2) adder arrays to perform addition(or subtraction) of two redundant form integers. Although it is laden with area and CPD overhead, one slice of (4×2) adder can easily be modified to perform one-digit addition in three fields $GF(p)$, $GF(2^n)$, and $GF(3^n)$ as explained in Section 2. A multiplier that can operate in three fields can be designed in the same way the original unified multiplier [12] is designed. Two important differences of the new unified multiplier from the original unified multiplier is that it has two control bits (as opposed to one in the original multiplier) to select the field mode ($GF(p)$, $GF(2^n)$, or $GF(3^n)$), and that the processing unit (PU) has now two layers of (4×2) modified-adder arrays. In addition, RSD arithmetic is employed instead of carry-save arithmetic.

In order to asses the merits of unified multiplier that performs multiplications of three fields in the same datapath, one needs to compare the unified multiplier against a hypothetical architecture which has three separate multipliers for these three fields. The {area $\times$ CPD} metric can be used in order to figure out the balance between the saving in area and overhead in the critical path delay that the unified multiplier will have when compared to hypothetical design. Implementations of both new unified multiplier and hypothetical design in ASIC standard cell library will demonstrate that the new unified multiplier considerably improves {area $\times$ time} metric when compared to hypothetical design [24].

4 Inversion

In this section, we give multiplicative inversion algorithms, which allow very fast and area-efficient unified hardware implementations. The presented algorithms are based on the Montgomery inversion algorithms given in [19]. While there are several unified inversion units reported in the literature [20, 21, 22] that compute in two fields $GF(p)$ and $GF(2^n)$ there has been no unified inversion unit proposed to operate in three fields. Therefore, we limit our discussion, which is based on the techniques and algorithms in [22], only to two basic fields, namely $GF(p)$ and $GF(2^n)$. It is, however, straightforward to extend the algorithm and its implementation to support the inversion in $GF(3^n)$.

4.1 The Montgomery Inversion Algorithms for $GF(p)$ and $GF(2^n)$

The Montgomery inversion algorithm as defined in [19] computes

$$b = a^{-1}2^n \pmod{p}\ , \tag{1}$$

given $a < p$, where p is a prime number and $n = \lceil log_2 p \rceil$. The algorithm consists of two phases: the output of Phase I is the integer r such that $r =$

$a^{-1}2^k \pmod p$, where $n \leq k \leq 2n$ and Phase II is a correction step and can be modified as shown in [23] in order to calculate a slightly different inverse that can more precisely be called *Montgomery inverse*:

$$b = MonInv(a2^n) = a^{-1}2^n \pmod p \ , \tag{2}$$

Algorithm D
Phase I
Input: $a2^n \in [1, p-1]$ and p
Output: $r \in [1, p-1]$ and k, where $r = a^{-1}2^{k-n} \pmod p$ and $n \leq k \leq 2n$

1: $u := p$, $v := a2^n$, $r := 0$, and $s := 1$
2: $k := 0$
3: while $(v > 0)$
4: if u is even then $u := u/2$, $s := 2s$
5: else if v is even then $v := v/2$, $r := 2r$
6: else if $u > v$ then $u := (u - v)/2$, $r := r + s$, $s := 2s$
7: else $v := (v - u)/2$, $s := s + r$, $r := 2r$
8: $k := k + 1$
9: if $r \geq p$ then $r := r - p$
10: return $r := p - r$ and k

The second phase of the Montgomery inversion algorithm simply performs $2n - k$ left (modular) shifts as a correction step to obtain $a^{-1}2^n \pmod p$ from $a^{-1}2^{k-n} \pmod p$. The left shift operations are modular in the sense that a modular reduction operation is performed whenever the shifted value exceeds the modulus.

In a similar fashion, the Montgomery inversion algorithm for $GF(2^n)$ can be given as follows:

Algorithm E
Phase I
Input: $a(x)x^n$ and $p(x)$, where $deg(a(x)x^n) < deg(p(x))$
Output: $s(x)$ and k, where $s(x) = a(x)^{-1}x^{k-n} \pmod{p(x)}$
and $deg(s(x)) < deg(p(x))$
and $deg(a(x)) + 1 \leq k \leq deg(p(x)) + deg(a(x)) + 1$

1: $u(x) := p(x)$, $v(x) := a(x)$, $r(x) := 0$, and $s(x) := 1$
2: $k := 0$
3: while $(u(x) \neq 0)$
4: if $u_0 = 0$ then $u(x) := u(x)/x$, $s(x) := xs(x)$
5: else if $v_0 = 0$ then $v(x) := v(x)/x$, $r(x) := xr(x)$
6: else if $deg(u(x)) \geq deg(v(x))$ then
$u(x) := (u(x) + v(x))/x$, $r(x) := r(x) + s(x)$, $s(x) := xs(x)$
7: else $v(x) := (v(x) + u(x))/x$, $s(x) := s(x) + r(x)$, $r(x) := xr(x)$
8: $k := k + 1$
9: if $s_{n+1} = 1$ then $s(x) := s(x) + xp(x)$

10: if $s_n = 1$ then $s(x) := s(x) + p(x)$
11: return $s(x)$ and k

Additions and subtractions in the original algorithm are replaced with additions without carry in $GF(2^n)$ version of the algorithm. Since it is possible to perform addition (and subtraction) with carry and addition without carry in a single arithmetic unit, this difference does not cause a change in the control unit of a possible unified hardware implementation. Step 6 of the proposed algorithm (where the degrees of $u(x)$ and $v(x)$ are compared) is different from that of the original algorithm. This necessitates a significant change to the control circuitry. In order to circumvent this problem we propose a slight modification in the original algorithm for $GF(p)$.

Before describing the new inversion algorithm, we first point out an important difference from the original Montgomery inversion algorithm. In Step 6 of the original Montgomery inversion algorithm two integers, u and v, are compared. Depending on the result of the comparison it is decided whether Step 6 or Step 7 is to be executed. We propose to modify Step 6 of the algorithm in a way that instead of comparing u and v, the number of bits needed to represent them are compared. As a result of this imperfect comparisons, u may become a negative integer. The fact that u might be a negative integer may lead to problems in comparisons in subsequent iterations, therefore u must be made positive again. To do that, it is sufficient to negate r. The proposed modifications can be seen in the modified algorithm given below. Note that Algorithm F is in fact a unified algorithm and it is reduced to Algorithm E provided that all addition and subtraction operations in $GF(p)$-mode are mapped to $GF(2^n)$ additions in $GF(2)$-mode. The variable $FSEL$ is used to switch between $GF(p)$ and $GF(2)$ modes.

Algorithm F
Phase I
Input: $a2^n \in [1, p-1]$ and p
Output: $s \in [1, p-1]$ and k, where $s = a^{-1}2^{k-n} \pmod p$ and $n \leq k \leq 2n$

1: $u := p$, $v := a2^n$, $r := 0$, and $s := 1$
2: $k := 0$ and $FSEL := 0$ // $FSEL := 1$ in $GF(2^n)$-mode
3: if u is positive then
4: if $(bitsize(u) = 0)$ then go to Step 15
5: if u is even then $u := u/2$, $s := 2s$
6: else if v is even then $v := v/2$, $r := 2r$
7: else if $bitsize(u) \geq bitsize(v)$ then $u := (u - v)/2$, $r := r + s$, $s := 2s$
8: else $v := (v - u)/2$, $s := s + r$, $r := 2r$
9: Update $bitsize(u)$, $bitsize(v)$ and sign of u
10: else (i.e. u is negative)
11: if u is even then $u := -u/2$, $s := 2s$ $r := -r$
12: else $v := (v + u)/2$, $u := -u$, $s := s - r$, $r := -2r$

13: $k := k + 1$
14: Go to Step 3
15: if $s_{n+2} = 1$ (i.e. s is negative)
16: $u := s + p$
17: $v := s + 2p$
18: if $u_{n+2} = 1$ then $s := v$
19: else $s := u$
20: $u := s - p$
21: $v := s - 2p$
22: if $v_{n+1} = 0$ then $s := v$
22-a: if $s_n = 1$ and $FSEL = 1$ then $s := s - p$
23: else if $u_n = 0$ then $s := u$
24: else $s := s$
25: return s and k

Changing the sign of both u and r simultaneously has the effect of multiplying both sides of the invariant $p = us + vr$ by -1. Therefore, new invariant when $r < 0$ is given as $\{-p = us + vr.\}$ While u and v remain to be positive integers, s and r might be positive or negative. Therefore, we need to alter the final reduction steps to bring s in the correct range, which is $[0, p)$. The range of s is $[-2p, 2p]$. As a result we need to use two more bits to represent s and r than the bitsize of the modulus.

The value u becomes negative as a result of $u = (u - v)/2$, when $bitsize(u) = bitsize(v)$ and $v > u$ before the operation. Since $u = (u - v)/2$ decreases the bitsize of absolute value of u at least by one independent of whether the result is negative or positive, u will become certainly less than v after the negation operation. Therefore, if a negative u is encountered during the operation only steps 11 and 12 are executed.

Note that the variable $FSEL$ is not needed for $GF(p)$-mode computations. Further, in $GF(p)$-mode $FSEL = 0$ and Step 22-a is never executed. This step becomes relevant in $GF(2)$-mode when $FSEL = 1$.

5 Conclusions

Unified arithmetic has gained a considerable amount of attention from the researchers and implementors working in applied cryptography. Basic premise of the unified arithmetic is that it is possible to use the same datapath for performing arithmetic operations in different fields. In this chapter, we provided the design principles of the unified arithmetic for three different fields, namely $GF(p)$, $GF(2^n)$ and $GF(3^n)$. We also pointed out the advantages of the unified arithmetic using different metrics such as area, critical path delay, operation timing, and time $\times$ area product. Although there is considerable amount of work for unified architectures for prime $GF(p)$, and binary extension $GF(2^n)$ fields, there arises a need for research on unified arithmetic units

that can operate in three fields $GF(p)$, $GF(2^n)$ and $GF(3^n)$ especially with the advent of pairing based cryptography.

6 Exercises and Projects

1. Obtain the truth tables for the four generalized full adders in Figure 2.
2. Provide a gate level implementation of generalized full adders in Figure 2. Realize your implementation using ASIC standard cell library and compare areas and critical path delays of generalized full adders.
3. Add an additional layer of logic gates to the output of the RSD adder in Figure 3 to force the output (1,1) to (0,0).
4. Modify the one bit of the RSD adder circuit in Figure 3 so that it computes addition in three fields, namely $GF(p)$, $GF(2^n)$, and $GF(3^n)$.
5. Design a unified multiplier of 8 bits that operates in three fields, $GF(p)$, $GF(2^n)$, and $GF(3^n)$.
6. Provide a gate level implementation of the selection logic in Figure 7.
7. Design the processing unit of a dual-radix $(4, 8)$ multiplier.
8. Obtain a Montgomery inversion algorithm by modifying the steps of Algorithm E.
9. Implement Algorithm E in software and provide some statistics such as average number of total iterations and average number of times Steps 4, 5, 6, and 7 are executed.
10. Modify the Step 6 of Algorithm E in such a way that $u(x)$ and $v(x)$ are compared as if they are integers. Implement the algorithm in software and check if it works. Obtain the same statistics you obtained in the previous exercise. Give a comparison.

References

1. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.
2. National Institute for Standards and Technology, "Digital Signature Standard (DSS)", *Federal Register* , 56:169, Aug 1991.
3. N. Koblitz, "Elliptic curve cryptosystems", Mathematics of Computation, 48(177):203–209, Jan 1987.
4. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, MA, 1993.
5. 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.
6. 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.

7. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
8. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). In *Proceedings of Third Annual Workshop on Selected Areas in Cryptography*, pages 95–106, Queen's University, Kingston, Ontario, Canada, August 15–16 1996.
9. IEEE. P1363: Standard specifications for public-key cryptography. 2000.
10. A. F. Tenca and C. K. Koc, A Scalable Architecture for Montgomery Multiplication, Lecture Notes in Computer Science, 1999, 1717, pp. 94-108.
11. A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Trans. Electron. Computers*, EC(10):389–400, September 1961.
12. E. Savaş, A. F. Tenca, and Ç. K. Koç, "A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$". In *Cryptographic Hardware and Embedded Systems,* Workshop on Cryptographic Hardware and Embedded Systems, pg. 277-292. Springer-Verlag, Berlin, 2000.
13. E. Savas, A. F. Tenca, M. E. Ciftcibasi, C. K. Koc, Multiplier architectures for $GF(p)$ and $GF(2^k)$, IEE Proceedings Computers and Digital Techniques, 151(2): 147-160, March 2004.
14. S. E. Eldridge. An faster modular multiplication algorithm. *International Journal of Comput. Math*, 40:63–68, 1991.
15. Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
16. J.-H.Oh and S.-J.Moon. Modular multiplication method. *IEE Proceedings*, 145(4):317–318, July 1998.
17. Colin D. Walter. Montgomery exponentitation needs no final subtractions. *Electronic Letters*, 35(21):1831–1832, October 1999.
18. G. Hachez and Jean-Jacques Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In *Cryptographic Hardware and Embedded Sytems*, Lecture Notes in Computer Science, No. 1965, pages 293–301. Springer-Verlag, Berlin, 2000.
19. B. S. Kaliski Jr., "The Montgomery inverse and its applications", IEEE Transactions on Computers, 44(8):1064–1065, Aug 1995.
20. A. A.-A. Gutub, A. F. Tenca, E. Savaş, and Ç. K.Koç, "Scalable and unified hardware to compute montgomery inverse in $GF(p)$ and $GF(2^n)$". In B.S. Kaliski Jr., Ç. K.Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, LNCS, pg. 485-500, Springer-Verlag Berlin, 2002.
21. E. Savaş and Ç. K. Koç, "Architecture for unified field inversion with applications in elliptic curve cryptography". In Proc. vol. 3, *The 9th IEEE International Conference on Electronics, Circuits and Systems - ICECS 2002*, pg. 1155-1158, Dubrovnik, Croatia, Sept 2002.
22. E. Savas, M. Naseer, A. A-A. Gutub, and C. K. Koc, Efficient Unified Montgomery Inversion with Multibit Shifting, IEE Proceedings Computers and Digital Techniques, 152(4): 489-498, July 2005.
23. E. Savaş and Ç. K. Koç, "The Montgomery modular inverse - revisited", IEEE Transactions on Computers, 49(7):763–766, Jul. 2000.
24. E. Öztürk and E. Savaş and B. Sunar, "A Versatile Montgomery Multiplier Architecture with Characteristic Three Support", Under review, 2008.