A versatile Montgomery multiplier architecture with characteristic three support

E. Öztürk [a], B. Sunar [a], E. Savaş [b],*

[a] *Department of Electrical and Computer Engineering, Worcester Polytechnic Institute, Worcester, MA 01609, USA*
[b] *Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul TR-34956, Turkey*

A R T I C L E   I N F O

A B S T R A C T

We present a novel unified core design which is extended to realize Montgomery multiplication in the fields $GF(2^n)$, $GF(3^m)$, and $GF(p)$. Our unified design supports RSA and elliptic curve schemes, as well as the identity-based encryption which requires a pairing computation on an elliptic curve. The architecture is pipelined and is highly scalable. The unified core utilizes the redundant signed digit representation to reduce the critical path delay. While the carry-save representation used in classical unified architectures is only good for addition and multiplication operations, the redundant signed digit representation also facilitates efficient computation of comparison and subtraction operations besides addition and multiplication. Thus, there is no need for a transformation between the redundant and the non-redundant representations of field elements, which would be required in the classical unified architectures to realize the subtraction and comparison operations. We also quantify the benefits of the unified architectures in terms of area and critical path delay. We provide detailed implementation results. The metric shows that the new unified architecture provides an improvement over a hypothetical non-unified architecture of at least 24.88%, while the improvement over a classical unified architecture is at least 32.07%.

1. Introduction

In the recent years there has been an increase in the research activity on pairing-based cryptography such as the identity-based cryptosystems [5]. Identity-based cryptography was first proposed by Shamir [18] in 1985. Rather than deriving a public key from a private information, in the identity-based schemes identity of a user plays the role of the public key. This reduces the computations required for authentication, and simplifies key management.

Elliptic curve and RSA (or Diffie–Hellman) schemes are typically implemented over $GF(p)$ or $GF(2^n)$ and over Z_n (or $GF(p)$). Numerous architectures were proposed to support arithmetic for elliptic curve cryptography and RSA-like schemes [16,3]. Unified architectures for the fields $GF(p)$ and $GF(2^n)$ were also proposed [16,8,25,15,21,17,1]. However, the emergence of pairing-based cryptography has attracted a significant level of interest in arithmetic in $GF(3^m)$. Hardware architectures for arithmetic in the characteristic three have appeared in [13,19,4].

Pairing-based cryptography may utilize all the three kinds of mathematical structures. Moreover, ECC and RSA schemes are typically implemented over prime or binary fields and integer rings, respectively. Thus, it would be highly desirable to have a single piece of unified hardware that supports arithmetic in all the three kinds of domains simultaneously. To the best of our knowledge, such an architecture is still lacking.

---

* Corresponding author. Tel.: +90 216 483 9606; fax: +90 216 483 9550.
  E-mail addresses: erdinc@wpi.edu (E. Öztürk), sunar@wpi.edu (B. Sunar), erkays@sabanciuniv.edu (E. Savaş).

While a unified architecture is highly desirable, the scalability and efficiency of the hardware is important. Here, we use the notion of scalability as introduced in [20]. The design should scale without the redesign of the architecture, by simply increasing the number of processing units. The scalability feature along with the unified approach would allow the architecture to support a wide spectrum of operating points ranging from low-end and low-power devices to high-end server platforms. For efficiency reasons, we design our architecture around a carry-free architecture. Furthermore, the scalable nature of the design allows the pipelining techniques to be used to further improve efficiency. Our architecture supports the basic arithmetic operations (i.e. addition, multiplication and inversion) in the arithmetic extension fields $GF(p)$,[1] $GF(2^n)$ and $GF(3^m)$. All operations are carried out in the residue space defined by the Montgomery multiplication algorithm [11].

Contributions of this work are outlined as follows:

- We propose a new and more efficient unified multiplier that operates in three fields, namely $GF(p)$, $GF(2^n)$, and $GF(3^m)$. To the best of our knowledge, this is the first attempt to combine the arithmetic of these three, cryptographically important, finite fields in a single datapath.
- We present a metric to quantitatively demonstrate the advantages of the proposed unified multiplier over the classical unified multiplier that supports arithmetic only in $GF(p)$ and $GF(2^n)$. The unified architectures proposed so far [16,8,17,25] lacked the quantitive analysis of the advantage of using a unified approach. It has only been reported that unified architecture results in negligible overhead in area and in critical path delay (CPD). In this work, we quantified the gain in terms of the Area × CPD metric, which showed that the benefits of the new unified architecture far exceed that of the classical unified architecture.
- We utilize a different carry-free arithmetic that allows efficient comparison and subtraction operations in $GF(p)$-mode. The classical unified architectures [16,8,17,25] utilize the carry-save representation in order to eliminate the carry propagation in $GF(p)$ mode. It is not easy to perform subtraction and comparison operations in the carry-save representation, where field elements are expressed as the sum of two integers. For instance [25] transforms the elements of $GF(p)$ that are in the carry-save form to the non-redundant form by adding the number to itself repeatedly in order to perform comparison and subtraction operations necessary to realize other field operations such as multiplicative inversion. For our carry-free arithmetic, the field elements are represented as the difference of two field elements, instead of sum. This representation facilitates efficient subtraction and comparison operations. Consequently, all arithmetic operations in cryptographic computations can be performed without the need of transformations between the redundant and the non-redundant forms.
- We computed the execution times of basic operations for three prominent public key cryptography algorithms: ECC scalar point multiplication, RSA exponentiation, and Tate pairing computations. The results show that the Tate pairing computations used in the identity-based cryptosystems can be performed by the proposed unified architecture in a comparably efficient manner.

In addition, a contribution of lesser importance is the introduction of scalable Montgomery algorithm for ternary extension field, $GF(3^m)$. Although it is a straightforward adaptation of the algorithm presented in [20] to ternary extension fields, it is the first attempt to formulate such an algorithm.

In Section 2, we introduce the traditional RSD representation and our notational conventions. Then, the unified core design is explained in Section 3. Section 4 presents the Montgomery multiplication algorithms for the three fields. In Section 5, we introduce the Montgomery multiplier design, and describe the relevant system-level architectural details such as pipelining and architectural scaling. We then present the complexity analysis and implementation results in Section 6. We provide the timing estimates for the particular configurations with varying number of processing units and give a comparative analysis in Section 7. Section 8 provides a discussion on the side-channel attacks, that is followed by the conclusion.

2. Redundant signed digit (RSD) arithmetic

Although carry-free arithmetic decreases the propagation delay in addition operations, the use of carry-free arithmetic for the modular subtraction operations introduces significant problems. When two's complement representation is used for subtraction, the carry overflow must be ignored. If there is no carry overflow, the result is negative. Since there can be hidden carry overflow with carry-free representation, it is hard to be sure that the result is positive or negative. It requires additional operations and additional hardware, which increases both latency and area. The RSD representation was introduced by Avizienis [2] in an effort to overcome this difficulty.

Arithmetic in the RSD representation is quite similar to carry-free arithmetic. An integer is still represented by two positive numbers; however, the non-redundant form of the representation is the difference between these two numbers, not the sum. If the number X is represented by x^p and x^n, then $X = x^p - x^n$.

One advantage of using the RSD representation is that it eliminates the need for two's complement representation to handle negative numbers. It is thus much easier to do both addition and subtraction operations without worrying about the carry and borrow chain. Furthermore, the subtraction operation does not require taking two's complement of the subtrahend. It is a more natural representation if both addition and subtraction operations need to be supported. This is indeed the case in

---

[1] Since we do not make use of the field properties in our design, the architecture supports also arithmetic in integer rings Z_n and hence supports RSA.

the Montgomery multiplication and inversion algorithms. Also, comparison of two integers is much easier with the RSD representation. After subtracting one integer from the other one, which is a simple addition operation, a conventional comparator can be utilized.

2.1. Number representations

As mentioned earlier, the integer X is represented by two integers, x^p and x^n, and $X = x^p - x^n$. For the RSD representation, we reserve the notation (x^p, x^n) to represent the number X. The RSD representation for the extension fields is described as follows:

1. *Prime field GF(p):* Elements of the prime field $GF(p)$ may be represented as integers in the binary form. In the binary RSD representation, its digits can have three different values: 1, 0 and -1. These three digit values are represented as

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

2. *Binary extension field GF(2^n):* Elements of the field $GF(2^n)$ may be considered as polynomials with coefficients from $GF(2)$. This allows one to represent $GF(2^n)$ elements by simply ordering its coefficients into a binary string. Since there is no carry chain in $GF(2)$ arithmetic, a digit can have the values 1 or 0. These values are represented as

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

3. *Ternary extension field GF(3^m):* Elements of the extension field $GF(3^m)$ may be considered as polynomials over $GF(3)$. The coefficients can take the values -2, -1, 0, 1, and 2. However, since there is no carry propagation in $GF(3^m)$ polynomial arithmetic, the digit values -2 and 2 are congruent to 1 and -1, respectively. The RSD representations for possible coefficient values are given as

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

3. Unified arithmetic core

We first build a unified arithmetic core for the basic arithmetic operations (i.e. addition, subtraction and comparison). The core is unified so that it can perform the arithmetic operations of three extension fields: $GF(p)$, $GF(2^n)$ and $GF(3^m)$. Since the elements of the three different fields are represented using a very similar data structure, the algorithms for the basic arithmetic operations in these fields are structurally identical. We use this fact to our advantage to realize a unified arithmetic core.

3.1. The architecture

The conventional 1-bit full adder assumes positive weights for all its three binary inputs and two outputs. However, full adders can be generalized to have both positive- and negative-weight inputs and outputs. This allows us to construct an adder design with both inputs and outputs in the RSD form, since we can have negative-weight numbers as inputs. In our core design, we used two forms of the generalized full adders as shown in Fig. 1: one negative-weight input (GFA-1) and two negative-weight inputs (GFA-2). Note that GFA-0 is identical to a common full adder design.

The logic behaviors of a common full adder and two generalized full adders are shown in Fig. 2. As visible from the truth table, GFA-1 and GFA-2 have the same logical characteristics. The only difference is the order of the inputs and outputs. The same hardware is used for both types of generalized full adders. However, it should be noted that the decoding of the outputs is different. For GFA-1, the result is decoded as $2c - s$. For GFA-2, the result is decoded as $-2c + s$.

A single digit unified adder unit is constructed using two of the generalized full adders as shown in Fig. 3b. The unified adder unit has two digits in the RSD representation as inputs and one digit in the RSD representation as output. The unified digit adder unit also has carry input and output, which are only used for arithmetic in $GF(p)$. In total, the unit has 5 bits input and 3 bits output.

We start by designing the hardware for the prime fields ($GF(p)$) first. Two generalized full adders connected in the configuration shown in Fig. 3a are sufficient to handle the digit arithmetic of $GF(p)$ elements. To make the adder architecture
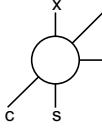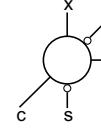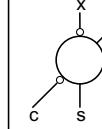
	GFA–0	GFA–1	GFA–2
Logic symbol	(x, y, z, c, s)	(x, y, z, c, s)	(x, y, z, c, s)
Type	GFA–0	GFA–1	GFA–2
Function	$x+y+z = 2c+s$	$x-y+z=2c-s$	$x-+yz=2c+s$

Fig. 1. Generalized full adders.

			GFA–0		GFA–1		GFA–2	
x	y	z	s	c	s	c	s	c
0	0	0	0	0	0	0	00	
0	0	1	1	0	1	1	1	1
0	1	0	1	0	1	0	1	0
0	1	1	0	1	0	0	0	0
1	0	0	1	0	1	1	1	1
1	0	1	0	1	0	1	0	1
1	1	0	0	1	0	0	0	0
1	1	1	1	1	1	1	1	1

Fig. 2. Logic tables of the three generalized full adders.

(a) Single RSD adder unit.   (b) Unied RSD adder unit.
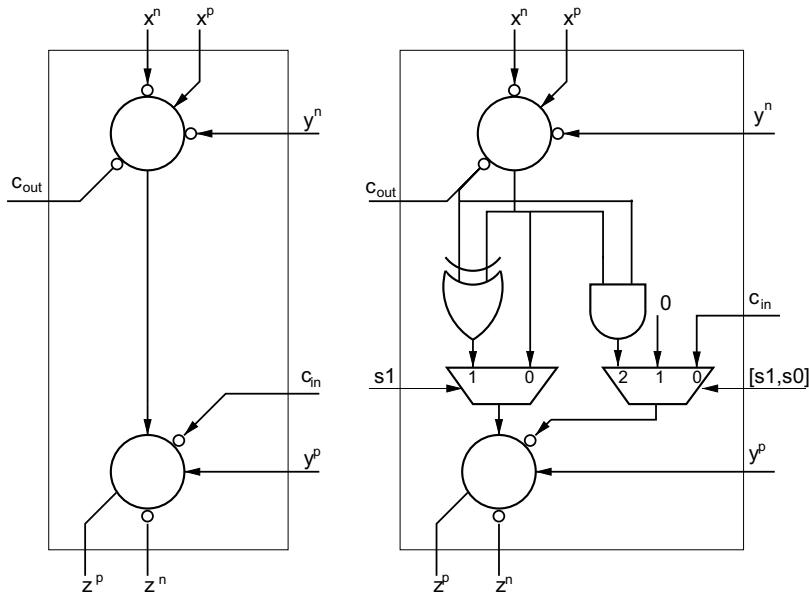
Fig. 3. RSD adder unit with both inputs and outputs in RSD form.
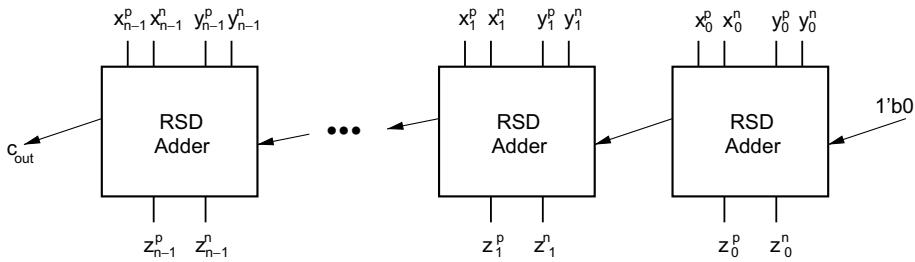
Fig. 4. RSD adder.

work for $GF(2^n)$ arithmetic, we inhibit the carry chain. Also, since the digits can only have the values $(0,0)$ and $(1,0)$, the negative-weight inputs of the adder are set to logic 0.

Modifying the adder design to make it also work for $GF(3^m)$ is more difficult, since the hardware works for base two and we need to support base three. The carry-free structure of the $GF(3^m)$ arithmetic operations makes our task easier. When carrying out arithmetic operations in $GF(3^m)$, the outputs of the adders have to be decoded. Since the generalized full adder works in binary form, the output is also in binary. We need to convert this output to base 3 before entering the data into the second generalized full adder. An XOR gate and an AND gate are sufficient for this conversion as shown in Fig. 3b. There is also need for multiplexers, where the select inputs of the multiplexers determine the field in which the adder is operating. The carry bits are only used when the circuit functions in $GF(p)$ mode. In Fig. 3b, $s1$ and $s0$ are the select inputs of the multiplexers. The modes of the hardware are

$$[s1, s0] = 0, 0 \rightarrow GF(p),$$

$$[s1, s0] = 0, 1 \rightarrow GF(2^n),$$

$$[s1, s0] = 1, 0 \rightarrow GF(3^m).$$

Now, we need to cascade n single digit RSD units in order to build an n-digit RSD adder. Fig. 4 shows the backbone of the structure. There are n 1-digit RSD adders and one GFA-1 adder to handle the last carry bit, which is omitted in $GF(2^n)$ and $GF(3^m)$.

3.2. Addition

The addition operation is implemented as shown in Fig. 4. The negative and positive parts of the numbers are entered accordingly, and the select inputs of the multiplexers are set for the desired field operations. There are also two control inputs to the adder for selecting the field, sel_2 and sel_3, which are not shown in Fig. 4. These inputs are decoded accordingly and they determine the select inputs of the multiplexers. It should be noted that, carry propagation occurs only between neighbouring cells as shown in Fig. 4.

3.3. Subtraction

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

$$X = (x^p, x^n) = x^p - x^n,$$

$$Y = (y^p, y^n) = y^p - y^n,$$

$$X - Y = (x^p, x^n) - (y^p, y^n) = (x^p, x^n) + (y^n, y^p).$$

3.4. Comparison

To compare two numbers given in the RSD representation, first one must be subtracted from the second one. After subtraction, the positive and negative components of the result are compared. This can be realized using a conventional comparator design. If the positive part is larger, the first number is greater than the second one. If the negative part is larger, the second number is greater than the first one. If both parts are equal, then the numbers being compared are equal.

The comparison operation has two components: RSD adder and comparator. There are already RSD adders in the design and one of them could be utilized for comparison. Also, a single RSD adder can be instantiated for comparison reasons only, without a significant area overhead.

Table 1
Implementation results of comparator design with different word sizes

Word length	500 MHz		Max. freq.	
	Area	CPD (ns)	Area	CPD '(ns)
8	47	0.72	70	0.39
16	95	0.80	161	0.42
32	191	1.24	391	0.49
64	451	1.35	756	0.55

Furthermore, a conventional comparator is used for comparing the positive and negative parts of the resultant of the subtraction operation. We designed this comparator using Verilog and synthesized with Synopsys Design Compiler with 0.13 μm ASIC library. The results are given in Table 1.

We also implemented a single RSD adder to utilize for comparison. Synthesis results showed that the minimum CPD of a single RSD adder is 0.66 ns. This shows that the critical path of an adder and a comparator connected back to back will not be more than the overall circuit, even for the 64-bit case. Thus, the word comparison operation can be performed in a single clock cycle.

It should be noted that most of the field arithmetic operations require the equality comparison of two numbers. Hence, a much simpler comparator could be utilized for comparison operations.

4. Montgomery multiplication

The Montgomery multiplication algorithm [11] is an efficient method for performing modular multiplication with an odd modulus. The algorithm replaces costly division operations with simple shifts, which are particularly suitable for the implementations on general-purpose computers.

Given two integers A and B, and the odd modulus M, the Montgomery multiplication algorithm computes $Z = \text{MonMul}(A, B) = A \cdot B \cdot R^{-1} \bmod M$, given $A, B < M$ and R such that $\gcd(R, M) = 1$. Even though the algorithm works for any R which is relatively prime to M, it is more useful when $R = 2^n$, where $n = \lceil \log_2(M) \rceil$. Since R is chosen to be a power of 2, the Montgomery algorithm performs divisions by a power of 2, which is basically shift operations in digital computers. The Montgomery multiplication algorithm for binary extension fields $GF(2^n)$ is first introduced in [10]. We describe the Montgomery multiplication algorithm for ternary extension fields $GF(3^m)$ in the subsequent sections.

The proposed adder design is used to build a Montgomery multiplier architecture. Since we want our hardware to support arithmetic in three different fields, we identify similarities between the arithmetic algorithms and integrate them together into a single hardware implementation.

4.1. Radix-2 Montgomery multiplication algorithm for GF(p) and GF(2^n)

The use of a fixed precision word alleviates the broadcast problem in the circuit implementation. Furthermore, a word-oriented algorithm allows the design of a scalable unit. For a modulus of n-bit precision, and a word size of w bits, $e = \lceil (n+1)/w \rceil$ words are required for storing field elements. Note that an extra bit is used for the variables holding the partial sum in the Montgomery algorithm for $GF(p)$, since the partial sums can reach $(n+1)$-bit precision. The algorithm we used [20] scans the multiplicand operand B word-by-word, and the multiplier operand A bit-by-bit. The vectors used in the multiplication operations are expressed as

$$B = (B^{(e-1)}, \ldots, B^{(1)}, B^{(0)}),$$
$$A = (a_{n-1}, \ldots, a_1, a_0),$$
$$p = (p^{(e-1)}, \ldots, p^{(1)}, p^{(0)}),$$

where the words are marked with superscripts and the bits are marked with subscripts. For example, the ith bit of the kth word of B is represented as $B_i^{(k)}$. A particular range of bits in a vector B from position i to j where $j > i$ is represented as $B_{j \ldots i}$. $(x|y)$ represents the concatenation of two bit sequences. Finally, 0^n stands for an all-zero vector of n bits. The algorithm is shown in Algorithm 1.

Algorithm 1: Montgomery multiplication algorithm for $GF(p)$

    Require: $A, B \in GF(p)$ and p
    Ensure: $C = A \cdot B \cdot 2^{-n} \in GF(p)$, where $n = \lceil \log_2 p \rceil$
    1: $T := 0^n$
    2: **for** i from 0 to $n - 1$ **do**
    3:    $(Carry|T^{(0)}) := a_i \cdot B^{(0)} + T^{(0)}$
    4:    $Parity := T_0^{(0)}$
    5:    $(Carry|T^{(0)}) := Parity \cdot p^{(0)} + (Carry|T^{(0)})$

```
6:   for j from 1 to e − 1 do
7:      (Carry|T^(j)) := a_i · B^(j) + T^(j) + Parity · p^(j) + Carry
8:      T^(j−1) := (T_0^(j) | T_{w−1...1}^(j−1))
9:   end for
10:   T^{e−1} := (Carry | T_{w−1...1}^{(e−1)})
11: end for
12: C := T
13: if C > p then C := C − p
14: return C
```

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

The $GF(2^n)$ version of the algorithm is structurally identical with only a few minor differences. First of all, the operands and temporary variable T are represented as polynomials in the algorithm. The modulus is also a polynomial, $P(x)$. As a result of the polynomial arithmetic, the addition symbols, i.e. '+' represent carry-free addition or bit-wise XOR operation. Since polynomial addition is a carry-free operation, $Carry$ is ignored in Steps 3, 5, 7 and 9. Also, Step 13 is not operated.

4.2. Radix-3 Montgomery multiplication algorithm for $GF(3^m)$

Montgomery multiplication algorithms for $GF(p)$ and $GF(2^n)$ are similar to each other because they are both implemented in radix-2. Since the Montgomery multiplication algorithm for $GF(3^m)$ is implemented in radix-3, the algorithm needs to be modified. We already explained the differences for the addition part in RSD representation and we showed that both radix-2 and radix-3 representations can be implemented on a single hardware.

We will use the polynomial basis representation for $GF(3^m)$. For a modulus size of m and a word size of w, $e = \lceil (m + 1)/w \rceil$ words are required. Since there is no carry computation in $GF(3^m)$ arithmetic, there will be no need for any extra digits used other than those used for the variable polynomials. Every coefficient of the operands and the modulus is represented by 2 bits in the hardware, one for the positive part and one for the negative part, since the coefficients are in RSD representation. The algorithm scans the words of operand $B(x)$, and the coefficients of operand $A(x)$. In the radix-3 representation, the polynomials used in the multiplication operation are expressed as

$$B(x) = \left(b^{(e-1)} \cdot x^{(e-1)\cdot w} + \cdots + b^{(1)} \cdot x^w + b^{(0)}\right),$$

$$A(x) = (a_{n-1} \cdot x^{n-1} + \cdots + a_1 \cdot x + a_0),$$

$$p(x) = \left(p^{(e-1)} \cdot x^{(e-1)\cdot w} + \cdots + p^{(1)} \cdot x^w + p^{(0)}\right),$$

where the words are marked with superscripts and the coefficients are marked with subscripts. For example, the ith coefficient of the kth word of $B(x)$ is represented as $B_i^{(k)}$. The algorithm is shown in Algorithm 2.

Algorithm 2: Montgomery multiplication algorithm for $GF(3^m)$

```
Require: A(x), B(x) ∈ GF(3^m) and p(x)
Ensure: C(x) = A(x) · B(x) · 3^{−m} ∈ GF(3^m), where m is the degree of p(x)
1: T(x) := 0
2: for i from 0 to m − 1 do
3:    T^(0) := a_i · B^(0) + T^(0)
4:    if T_0^(0) = p_0^(0)
5:       T^(0) := T^0 − p^(0)
6:       for j from 1 to e − 1 do
7:          T^(j) := a_i · B^(j) + T^(j) − p^(j)
8:          T^(j−1) := (T_0^(j) | T_{w−1...1}^(j−1))
9:       end for
10:   else
11:       T^(0) := T^0 + p^(0)
12:       for j from 1 to e − 1 do
13:          T^(j) := a_i · B^(j) + T^(j) + p^(j)
14:          T^(j−1) := (T_0^(j) | T_{w−1...1}^(j−1))
15:       end for
16:   end if
17:   T^{e−1} := ((0,0) | T_{w−1...1}^{(e−1)})
18: end for
19: return T(x)
```
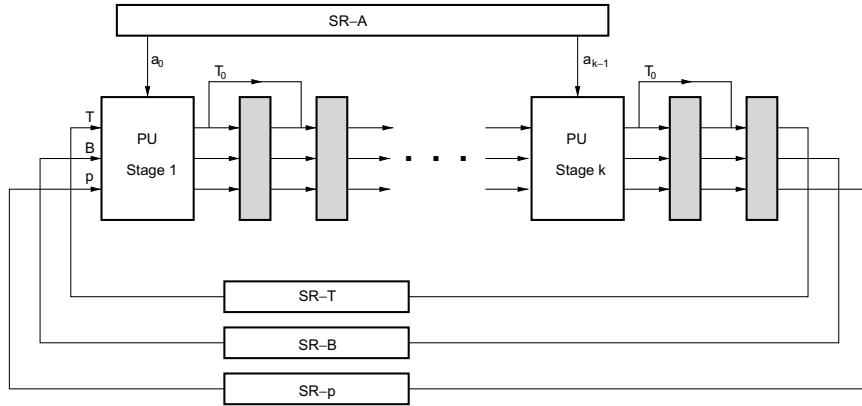
Fig. 5. Pipeline organization for the Montgomery multiplier.

5. Multiplier architecture

In this section, we explain the multiplier design which implements Algorithms 1 and 2 in a single architecture. We do not go into the detail of the global control logic path since its function can be inferred easily from the algorithms.

5.1. Pipeline organization

The presented Montgomery multiplication algorithms have the same loop structure: outer and inner loops with the variables i and j, respectively. Each processor unit (PU)[2] is responsible for one step of the outer loop with the variable i. Each PU receives the a_i digit as input. Also, every PU receives $B^{(j)}$, $p^{(j)}$ and $T^{(j)}$ as inputs, according to the inner loop variable j. The pipeline organization is shown in Fig. 5.

An important aspect of the pipeline is the organization of the registers. The digits a_i of the multiplier A are given serially to the PUs, and are used only for one iteration of the outer loop. So they can be discarded immediately after use. Therefore, a simple shift register with a load input will be sufficient. Also, rather than storing the multiplier A in a register, we can have a serial input for every digit and we store only the necessary a_i digit inside a register, only when it is needed. This will reduce the area and power consumption of the architecture. The registers for the modulus p and multiplicand B can also be shift registers.

The multiplication starts with the first PU by processing the first iteration of the outer loop of the algorithm. As can be seen from Algorithm 1, the data required for the second iteration will be ready after 2 clock cycles. Therefore, the second PU has to be delayed from the first PU by 2 clock cycles. This is realized by using two stages of registers in between. Also, these registers are handling the shift operations for the partial sum (Step 8 of Algorithm 1) as shown in Fig. 5.

When the first PU finishes the operations of an iteration step of the outer loop, it starts working on the next available iteration loop, and the second PU will be done in 2 clock cycles and will start working on the next available iteration. The same computation pattern is repeated for the entire pipeline organization.

If there are sufficiently many PUs, the first PU will be done with the first iteration of the loop when the last PU operates on the last iteration of the same loop. There will be no pipeline stall and no need for intermediate shift registers hold the data. The pipeline can continue working without stalling. This condition is satisfied if the number of PUs is at least half of the number of words of the operand. However, if there are not sufficiently many PUs, which means that a pipeline stall occurs, the modulus and multiplicand words generated at the last stage of the pipeline have to be stored in registers.

The shift registers SR-T, SR-p and SR-B hold these values when there is a pipeline stall. The length of these shift registers is of crucial importance and is determined by the number of pipeline stages k and the number of words e in the modulus. The width of the shift registers is equal to w, the word size. The length of these registers can be given as

$$L = \begin{cases} e - 2 \cdot (k - 1) & \text{if } e \geqslant 2k, \\ 0 & \text{otherwise}. \end{cases}$$

5.2. Processing unit

The processing unit consists of two layers of adder blocks or unified arithmetic cores (cf. Section 3). The arithmetic core is capable of performing addition and subtraction operations in the fields $GF(p)$, $GF(2^n)$ and $GF(3^m)$. The block diagram of a processing unit with word size $w = 3$ is shown in Fig. 6.

---

[2] We will define the internal structure of the PU in the following section.
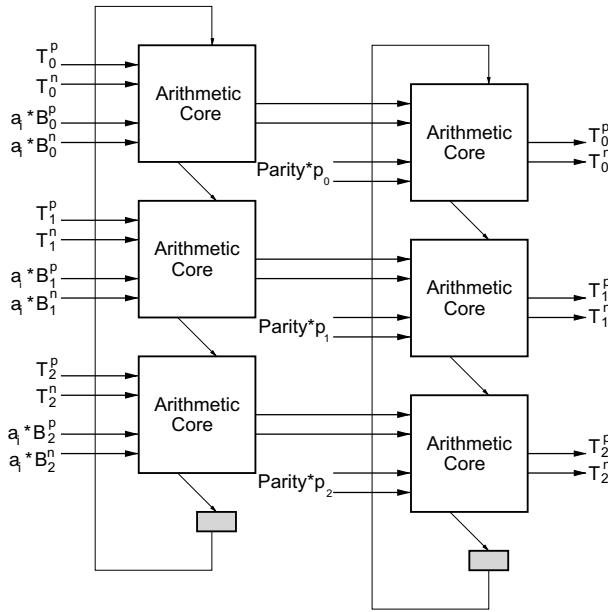
Fig. 6. Processing unit (PU) with $w = 3$.

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

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

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

We need a simple logic for multiplying a single digit a_i of the multiplier A with a word $B^{(j)}$ of the multiplicand B to realize the first part $a_i \cdot B^{(j)}$ of the operation. Since a_i can only have the values $(0,0)$, $(1,0)$ or $(0,1)$, the result of $a_i \cdot B^{(j)}$ can be 0, $B^{(j)}$ or $-1 \cdot B^{(j)}$, respectively. Negating an integer is realized by simply swapping the positive and negative bits of its digits. A simple special encoder would be sufficient for this. We need another logic circuit to determine the parity in each iteration of the outer loop. We check the right-most digit of the modulus, i.e. $p_0^{(0)}$ and the right-most digit of the operation $T^{(0)} = a_0 \cdot B^{(0)} + T^{(0)}$, $T_0^{(0)}$ and determine the parity:

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

This is very similar to the encoder logic we used earlier. One difference is that since the parity is computed only once for every iteration step, it needs to be stored in a register after being computed by the PU.

6. Complexity analysis

As mentioned earlier, if the number of PUs is at least half of the number of words in the operand, the pipeline will not stall and every PU will continuously operate. For multiplication, the total computation time, latency (clock cycles), is given as

$$Latency = \begin{cases} 2(m-1) + e & \text{if } e \geqslant 2k, \\ (\lceil \frac{m}{k} \rceil)e + 2((m-1) \bmod k) & \text{otherwise.} \end{cases} \tag{1}$$

The graphs given in Fig. 7 illustrate how the latency of Montgomery multiplication changes for various operand lengths and for a variable number of PUs.

Table 2 shows the estimates for the number of clock cycles required for realizing ECC scalar point multiplication, RSA exponentiation, and Tate pairing computations with the modified Duursma–Lee algorithm. We pick a word size of 8-digits. For the implementation of ECC with 160 bits, we assume mixed coordinates and the NAF representation are used to realize the scalar point multiplication operation. For point doubling we use Jacobian coordinates and for point addition we use affine + Jacobian coordinates. For RSA, we assume a full 1024-bit exponent and use the square multiply algorithm. The Tate pairing computation is realized using the modified Duursma–Lee algorithm [9] over the field $GF(3^{6 \times 97})$. (The original Duursma–Lee algorithm was proposed in [7].) Note that the chosen lengths provide similar levels of security. We are not getting
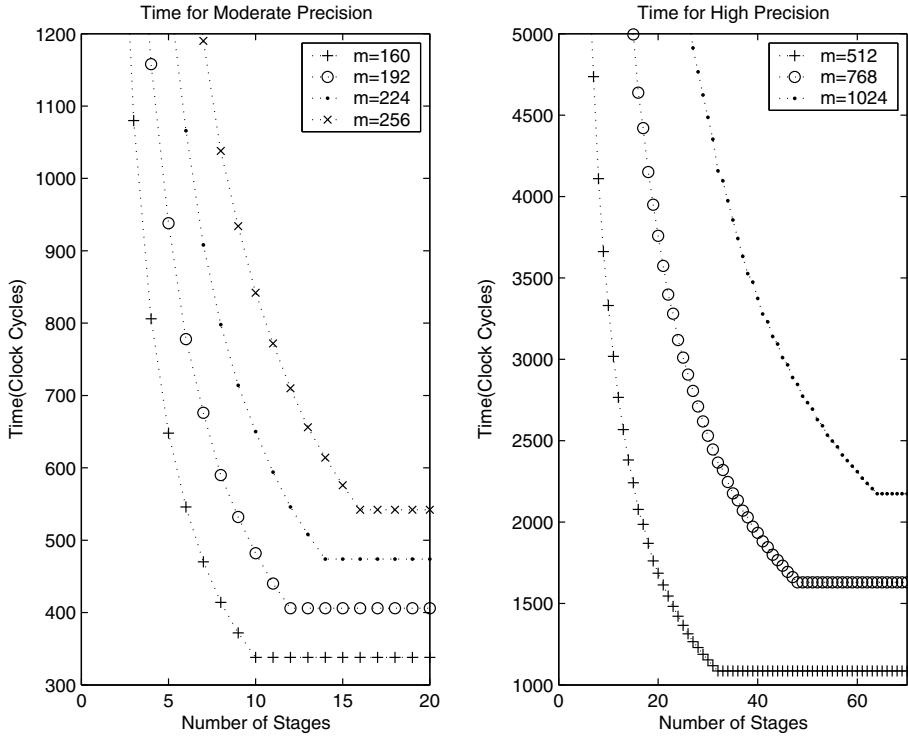
Fig. 7. Computation time of Montgomery multiplication for various number of PUs and operand lengths.

Table 2
The execution times for the ECC scalar multiplication, the RSA exponentiation and the modified Duursma–Lee algorithms

Number of PUs	160-bit ECC (clock cycles)	1024-bit RSA (clock cycles)	Tate pairing $GF(3^{97})$ (clock cycles)
4	1,507,728	50,340,864	1,211,808
8	772,524	25,187,328	796,220
16	630,708	12,628,992	796,220
32	630,708	6,386,688	796,220

into the details of the clock cycle computations for the ECC and the RSA cases, since the computations are trivial. For the Tate pairing case we note that the modified Duursma–Lee algorithm [9] iterates 97 times and works by performing the operations in the field $GF(3^{97})$. In each iteration, 20 multiplications and 10 cubing operations are carried out in the field $GF(3^{97})$. Each cube computation may be realized via two multiplications bringing the total number of multiplications to 40 per iteration of the main loop of the modified Duursma–Lee algorithm. Including the additional four multiplications performed in the initialization of the algorithm, the total number of multiplications are found as $40 \cdot 97 + 4 = 3884$. In the 4 PU case, the latency of one multiplication is found using Eq. (1) as 312 clock cycles. Hence, the total paring computation requires $3884 \cdot 312 = 1,211,808$ cycles. For the 8 PU case, the latency of one multiplication operation is found as 205 clock cycles leading to a total number of 796,220 clock cycles.

7. Results and comparison

In this section, we provide implementation results of the proposed unified architecture to demonstrate its advantage over classical architectures. We also include the implementation results of unified Montgomery multiplier circuit that operates in three finite fields. In addition, we present a qualitative comparison of the proposed architecture with the previously defined architectures.

7.1. PU architecture

The presented architecture was developed into Verilog modules and synthesized using the Synopsys Design Compiler tool. In the synthesis, we used the TSMC 0.13 μm ASIC library and assumed a word size of 8 bits. The maximum operating frequency of the design was found as 800 MHz. However, the synthesis tool will try to optimize the circuit for timing if we

Table 3
Execution times at frequency f = 500 MHz (Section 7)

Number of PUs	160-bit ECC (ms)	1024-bit RSA (ms)	Tate pairing $GF(3^{97})$ (ms)
4	3.015	100.681	2.424
8	1.545	50.374	1.592
16	1.261	25.258	1.592
32	1.261	12.773	1.592

set the target frequency at 800 MHz. Thus, for the rest of this section, we assume a target frequency of 500 MHz for synthesis results. The timing results at 500 MHz for three prominent public key operations are given in Table 3. We note that if the pipeline does not stall, as the number of PUs increases the register space will increase. Otherwise, the register space will stay constant with the increasing number of PUs.

For proof of concept, we built and synthesized different PUs working on different fields. First category of implementations are those working on a single field only. The implementations, denoted as A_1, A_2, and A_3, are those working in fields $GF(p)$-only, $GF(2^n)$-only, and $GF(3^m)$-only, respectively. In the second category, there are two unified architectures. The implementation, denoted as A_4, is a unified architecture working in both fields $GF(p)$ and $GF(2^n)$. And finally, the implementation A_5 is the unified architecture working in all three fields, namely $GF(p)$, $GF(2^n)$, and $GF(3^m)$. All five architectures are implemented for three different word sizes: 8, 16, and 32, and the implementation results of these architectures are summarized in Table 4.

From Table 4, the cost of unified architectures compared to $GF(p)$-only implementation can be captured as overhead both in the area and in the critical path delay (CPD). However, the figures in Table 4 hardly give an idea about the advantage of the unified architectures. Apparently, the advantage of the unified architectures is saving in the area without too much adverse effect on the critical path delay. In order to measure the advantage of the unified architecture, we used (Area × CPD) as the metric. We first investigated the first unified architecture A_4 that has a single datapath for $GF(p)$ and $GF(2^n)$ and compared it against the implementation results of a hypothetical architecture, denoted as $A_1 + A_2$, that has two separate datapaths for $GF(p)$ and $GF(2^n)$. For the hypothetical architecture $A_1 + A_2$, the area is the sum of areas of A_1 and A_2 architectures, while the critical path delay is the maximum CPD of these two architectures. The implementation results are summarized in Table 5. The improvement of the architecture is found to be about 7–8.5% in terms of the Area × CPD metric.

Similarly, we also investigated the advantage of the unified architecture, A_5 over a hypothetical architecture, $A_1 + A_2 + A_3$, that has three separate datapaths for the fields $GF(p)$, $GF(2^n)$, and $GF(3^m)$. The results summarized in Table 6 shows that the advantage of using the unified architecture A_5 is at least 34.83% in terms of the metric (Area × CPD). The improvement figures in Table 6 clearly demonstrate that the unified architecture A_5 provides far superior performance compared to the classical unified architectures working for only the fields $GF(p)$ and $GF(2^n)$.

Table 4
Implementation results of a PU with different word sizes

Word length	A_1		A_2		A_3		A_4		A_5	
	Area	CPD (ns)	Area	CPD (ns)	Area	CPD (ns)	Area	CPD (ns)	Area	CPD (ns)
8	516	1.91	91	0.77	656	1.92	576	1.87	795	1.91
16	963	1.90	168	0.79	1257	1.92	1034	1.90	1556	1.92
32	1980	1.89	329	0.84	2534	1.92	2132	1.90	3013	1.92

Table 5
The advantage of the unified architecture A_4, for $GF(p)$ and $GF(2^n)$

Word length	Area				CPD				Area × CPD		Improvement (%)
	A_1	A_2	$A_1 + A_2$	A_4	A_1	A_2	$A_1 + A_2$	A_4	$A_1 + A_2$	A_4	
8	516	91	607	576	1.91	0.77	1.91	1.87	1159	1077	7.07
16	963	168	1131	1034	1.90	0.79	1.90	1.90	2149	1965	8.56
32	1980	329	2309	2132	1.89	0.84	1.89	1.90	4364	4051	7.17

Table 6
The advantage of unified architecture A_5, for $GF(p)$, $GF(2^n)$, and $GF(3^m)$

Word length	Area			CPD			Area × CPD		Improvement (%)
	A_3	$A_1 + A_2 + A_3$	A_5	A_3	$A_1 + A_2 + A_3$	A_5	$A_1 + A_2 + A_3$	A_5	
8	656	1263	795	1.92	1.92	1.91	2425	1518	37.40
16	1257	2388	1556	1.92	1.92	1.92	4585	2988	34.83
32	2534	4843	3013	1.92	1.92	1.92	9299	5785	37.78

Table 7
The advantage of the new unified architecture A_5 over the classical unified architecture A_4

Word length	Area		CPD		Area $\times$ CPD		Improvement (%)
	$A_4 + A_3$	A_5	$A_4 + A_3$	A_5	$A_4 + A_3$	A_5	
8	1232	795	1.92	1.91	2365	1518	35.81
16	2291	1556	1.92	1.92	4399	2988	32.07
32	4666	3013	1.92	1.92	8959	5785	35.43

Table 8
Synthesis results for Montgomery multiplier architectures, with unified and separate datapaths

# of PUs	Area		CPD		Area $\times$ CPD		Improvement (%)
	Separate paths	Unified	Separate paths	Unified	Separate paths	Unified	
4	10,644	8372	2	1.91	21,288	15,991	24.88
8	15,672	12,128	2	1.91	31,344	23,164	26.10

In order to see more clearly what one can gain with the new unified architecture A_5 over the classical one, A_4, we also compared the two unified architectures in terms of the Area $\times$ CPD metric. The results summarized in Table 7 highlight the advantage of the new unified architecture over the classical one, which is at least 32%.

7.2. Montgomery multiplier architecture

The Montgomery multiplier architecture presented in Section 5 was developed into Verilog modules and synthesized using the Synopsys Design Compiler. In the synthesis, we used the TSMC 0.13 µm ASIC library and assumed a word size of 8 bits. The maximum operating frequency of the multiplier architecture was found as 800 MHz. This shows that the PU constitutes the critical path of the entire design. The synthesis results showed that the area of the multiplier for 4 PUs and 8 PUs was 11,512 and 15,361 two-input NAND equivalent gates, respectively. We note that as the number of PUs increases, the register space will increase if the pipeline does not stall. Otherwise, the register space will stay constant with the increasing number of PUs.

Similarly, we also investigated the advantage of the unified Montgomery multiplier architecture over a hypothetical architecture that has three separate datapaths for the fields $GF(p)$, $GF(2^n)$, and $GF(3^m)$. The results, summarized in Table 8, show that the advantage of using the unified architecture is at least about 25% in terms of the metric (Area $\times$ CPD). The improvement figures in Table 8 clearly demonstrate that the unified multiplier architecture provides far superior performance compared to the classical unified architectures working for only the fields $GF(p)$ and $GF(2^n)$.

For our architecture, the final results are in the RSD form. After the field operations are completed, the results need to be converted back to the more conventional form before being sent to the adversary. For example, if we are using our multiplier in a Diffie–Hellman protocol, we need to perform an exponentiation operation first. During the exponentiation operation, the intermediate results will stay in the RSD form. After completing the exponentiation operation, the final result has to be converted back to the desired form, depending on the protocol. This conversion can be performed serially utilizing an 8-bit ripple carry adder. Since this is done only once, the latency overhead it produces is negligible, we could even use a bit-serial adder. However, we built an 8-bit ripple carry adder using Verilog and synthesized it with Synopsys Design Compiler, with 0.13 µm library with a target frequency of 500 MHz. Synthesis results showed that the critical path of this adder is 1.34 ns, which is in the range of our multiplier circuit. The area of this adder is 66 gates equivalent. Thus, a word-serial addition operation can be performed without a significant area or a latency overhead.

7.3. Comparison with the previous unified architectures

In this section, we compare the new architecture against the previously proposed unified architectures in [1,8,15–17,21,25] to put it in a perspective in relation to other unified architectures. The architecture in [16] is the first and perhaps the most basic unified architecture, whose simplified processing unit (PU) for three bits is shown in Fig. 8. It basically consists of two layers of dual-field adders (that add with or without carry) and assumes that all inputs are in the non-redundant form. It keeps a temporary result in the redundant form, and therefore the final result is produced in the redundant form as well. Consequently, the result must be converted back to non-redundant form if further computation is needed, which is the case with all public key cryptography algorithms. For instance, a scalar point multiplication in ECC with moderate security level (e.g. 160 bit) requires hundreds of multiplications,[3] which results in as many conversion operations.

The redundant representation used in the previous unified architectures is the carry-save form, where an integer is represented as the sum of two other integers. The disadvantages of carry-save form are that (i) two integers in carry-save form

---

[3] More than a thousand multiplications are required for the same security level if the projective coordinates are used.
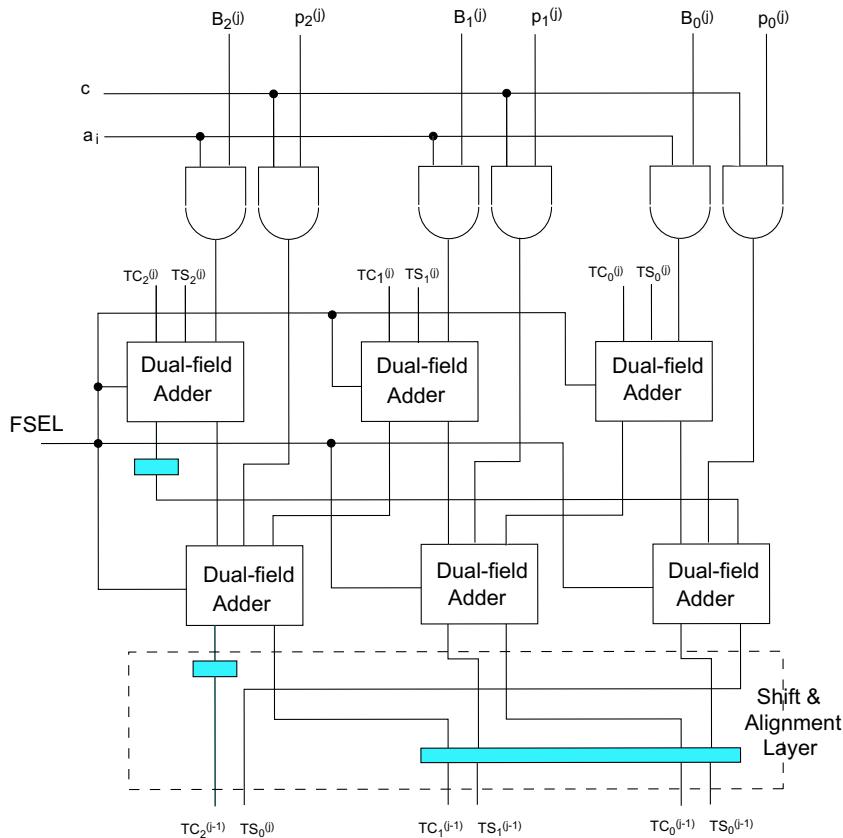
Fig. 8. Processing unit (PU) of the original unified architecture with $w = 3$.

cannot be compared and (ii) subtraction is costly. Therefore, the partial results during the computations of cryptographic operations (i.e. elliptic curve scalar point multiplication RSA exponentiation, etc.) must be converted back to the non-redundant form after every multiplication operation. The cost of the back transformation is twofold: (i) area for converter circuit and (ii) time overhead (clock cycles) for reverse transformation. At the expense of extra overhead in time, the need for an extra inverter circuit can be eliminated as suggested in [25], where conversion is achieved by repeated carry-save addition.

In summary, all the previously proposed unified architectures are designed to efficiently perform a single field multiplication operation. They offer different properties to be appealing from various perspectives. The original unified architecture [16] utilizes single-radix, where the multiplier is scanned one bit at a time. Au and Burgess [1] and Tenca et al. [21] proposes unified multipliers that scan the multiplier two or three bits at a time in order to reduce the cycle count without too much adverse effect on the critical path delay. The multiplier in [17] scans higher number of multiplier bits in $GF(2^n)$ mode than in $GF(p)$ mode in order to speedup the $GF(2^n)$ multiplication. The multipliers in [8,25] are not scalable (i.e. work for a fixed precision) while the architecture in [25] is suitable for performing other field operations with the aid of conversion between the redundant and the non-redundant representations. Finally, Satoh and Takano [15] introduces a word-level (i.e. r-bit $\times$ r-bit) unified multiplier to be used in a ECC processor. An extensive comparison of all the unified architectures and the proposed one is summarized in Table 9.

Table 9
Comparison of unified architectures

Architecture	$GF(3)$ support	Scalable	Conversion necessary?	High-radix possible?	Dual-radix possible?	Support for comparison and subtraction
[1]	No	Yes	Yes	High-radix	No	No
[8]	No	No	Yes	No	No	No
[15]	No	No	Yes	No	No	No
[16]	No	Yes	Yes	Extensible	Extensible	No
[17]	No	Yes	Yes	High-radix	Dual-radix	No
[21]	No	Yes	Yes	High-radix	No	No
[25]	No	No	Yes	No	No	Yes
Proposed	Yes	Yes	No	Extensible	Extensible	Yes

The proposed unified architecture is currently a single-radix implementation. However, it can easily be modified to work in higher radix or dual radices by applying the design techniques in [1,21,17]. There is support for other arithmetic operations such as comparison and subtraction in $GF(p)$-mode due to the new redundant signed representation. This support also exist in [25] at the expense of conversion operations from the redundant representation to the non-redundant representation.

8. A note on side-channel attacks

In this section, we would like to briefly comment on the side-channel characteristics of the proposed RSD multiplier as it is crucial to prevent information leakage through, so-called side-channels (i.e. execution time, power consumption, EM and temperature profiles, etc.) in cryptographic applications. We would like to note that most of the side-channel countermeasures are typically applied at either the algorithm or the circuit levels. For instance, an effective DPA counter-measure implemented at the algorithm layer is the randomized exponentiation [6]. On the other hand, at the circuit level masking techniques may be applied [12]. At even lower levels, the so-called power balanced cell libraries [22,23,14] which provide IC primitives that (ideally) have power consumption which is independent of the input bits, may be utilized. Any one of these techniques can be used alongside with the proposed multiplier. For instance, the presented architecture may be re-synthesized using a power balanced library at the cost of growing the area by roughly 2–3 times. On the other hand, a similar increase in area would be expected if the (non-unified) multiplier units are separately re-synthesized with the same cell library.

As far as the side-channel performance of the individual components at the arithmetic level are concerned we could identify very little work in the literature. In [24], Walter and Samyde demonstrated a direct correlation between the Hamming weights of the operands, and the power traces obtained during their multiplication. The authors conclude that it would be possible to gain useful side-channel information from a parallel multiplier built using Wallace trees. The processing element used in the multiplier proposed in this paper utilizes a redundant representation which will significantly reduce (if not eliminate) the correlation between the power traces from the Hamming weight of the operands. We can clearly claim that the proposed multiplier will be more resilient from this perspective than the more traditional multipliers to side-channel attacks. Furthermore, the same Ref. [24] considers pipelining to be an effective countermeasure to power attacks as multiple words of the operands are processed together. This will make the task of discerning operand bits from the power traces more difficult. The proposed architecture, therefore, has an additional level of protection against side-channel attacks due to its highly pipelined design.

9. Conclusion

We presented a scalable and unified architecture to support arithmetic in $GF(2^n)$, $GF(3^m)$, and $GF(p)$. Our design makes use of the redundant signed digit representation (RSD), which reduces the critical path delay and simplifies the support for the characteristic three arithmetic. Previous unified architectures are exclusively designed to implement field multiplication operations and thus carry-save representation they utilized makes it very difficult to perform other operations such as comparison and subtraction. Consequently, classical unified architectures have to transform the redundant representation to the non-redundant representation to perform these operations. However, these operations benefit from the proposed architecture. For instance, a subtraction operation results in no overhead compared to addition since it can be done by wiring in hardware.

Although there has been a consensus on the benefits of the unified architectures, no attempt has been reported in the literature to this date to quantify this benefit. We, for the first time, characterized and compared our unified architecture in terms of the {Area × CPD} metric and provided extensive implementation results to concretely establish the value of the proposed architecture. We have found out that the proposed unified architecture provides at least 24.88% and 32.07% improvement over non-unified architectures and classical unified architectures, respectively.

Our design is pipelined for improved efficiency and is scalable. Hence, different precisions can be easily supported without the redesign of the core. The number of processing units can be adjusted to given silicon area and/or the desired performance. We believe that this highly versatile architecture will fulfill a critical need in supporting elliptic curve cryptography, RSA/DH schemes, and identity-based cryptography using a single architecture in an efficient manner.

Acknowledgements

References

[1] Au Lai-Sze, Burgess Neil. Unified radix-4 multiplier for $GF(p)$ and $GF(2^n)$. In: ASAP; 2003. p. 226–36.
[2] Avizienis A. Signed-digit number representations for fast parallel arithmetic. IRE Trans Electron Comput, EC 1961(10):389–400.

[3] Bajard Jean-Claude, Imbert Laurent, Nègre Christophe, Plantard Thomas. Efficient multiplication in $GF(p^k)$ for elliptic curve cryptography. In: IEEE symposium on computer arithmetic; 2003. p. 181–7.

[4] Bertoni G, Guajardo J, Kumar SS, Orlando G, Paar C, Wollinger TJ. Efficient $GF(p^m)$ arithmetic architectures for cryptographic applications. In: Joye M, editor. Topics in Cryptology – CT RSA 2003. Lecture notes in computer science, vol. 2612. Springer-Verlag; 2003. p. 158–75.

[5] Boneh D, Franklin MK. Identity-based encryption from the Weil pairing. In: Kilian J, editor. Advances in Cryptology – CRYPTO 2001. Lecture notes in computer science, vol. 2139. Springer-Verlag; 2001. p. 213–29.

[6] Coron J-S. Resistance against differential power analysis for elliptic curve cryptosystems. In: Koç ÇK, Paar C, editors. CHES 1999. Lecture notes in computer science, vol. 1717. Springer-Verlag; 1999. p. 292–302.

[7] Duursma IM, Lee H-S. Tate pairing implementation for hyperelliptic curves $y^2 = x^p - x + d$. In: Laih C-S, editor. Advances in Cryptology – Asiacrypt 2003. Lecture notes in computer science, vol. 2894. Springer-Verlag; 2003. p. 111–23.

[8] Großschädl J. A bit-serial unified multiplier architecture for finite fields GF(p) and GF(2^m). In: Koç ÇK, Naccache D, Paar C, editors. CHES 2001. Lecture notes in computer science, vol. 2162. Springer-Verlag; 2001. p. 202–19.

[9] Kerins T, Marnane WP, Popovici EM, Barreto PSLM. Efficient hardware for the tate pairing calculation in characteristic three. In: Rao JR, Sunar B, editors. CHES 2005. Lecture notes in computer science, vol. 3659. Springer-Verlag; 2005. p. 412–26.

[10] Koç ÇK, Acar T. Montgomery multiplication in $GF(2^k)$. In: Proceedings of third annual workshop on selected areas in cryptography. Kingston, Ontario, Canada: Queen's University; 1996. p. 95–106. August 15–16.

[11] Montgomery PL. Modular multiplication without trial division. Math Comput 1985;44(170):519–21.

[12] Oswald E, Mangard S, Pramstaller N. Secure and efficient masking of AESA mission impossible. Technical report, Technical Report IAIK-TR 2003/11/1. <http://eprint.iacr.org/>; 2004.

[13] Page D, Smart NP. Hardware implementation of finite fields of characteristic three. In: Kaliski Jr BS, Koç ÇK, Paar C, editors. Cryptographic hardware and embedded systems — CHES 2002. Lecture notes in computer science, vol. 2523. Berlin: Springer-Verlag; 2002. p. 529–39.

[14] Regazzoni F, Badel S, Eisenbarth T, Grobschadl J, Poschmann A, Toprak Z, et al. A simulation-based methodology for evaluating the DPA-resistance of cryptographic functional units with application to CMOS and MCML technologies. In: International conference on embedded computer systems: architectures, modeling and simulation 2007 – IC-SAMOS 2007; 2007. p. 209–14.

[15] Satoh A, Takano K. A scalable dual-field elliptic curve cryptographic processor. IEEE Trans Comput 2003;52(4):449–60.

[16] Savaş E, Tenca AF, Koç ÇK. A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$. In: Koç ÇK, Paar C, editors. Cryptographic hardware and embedded systems – CHES 2000. Lecture notes in computer science, vol. 1965. Springer-Verlag; 2000. p. 277–92.

[17] Savaş E, Tenca AF, Çifçibaşi ME, Koç ÇK. Multiplier architectures for $GF(p)$ and $GF(2^n)$. IEE Proc Comput Digital Tech 2004;151(2):147–60.

[18] Shamir A. Identity-based cryptosystems and signature schemes. In: Advances in cryptology – CRYPTO 1985. Lecture notes in computer science, vol. 196. Springer-Verlag; 1985. p. 47–53.

[19] Kerins T, Popovici E, Marnane WP. Algorithms and architectures for use in FPGA implementations of identity based encryption schemes. In: Field Programmable logic and applications. Lecture notes in computer science, vol. 3203. Springer-Verlag; 2004. p. 74–83.

[20] Tenca AF, Koç ÇK. A scalable architecture for Montgomery multiplication. In: Koç ÇK, Paar C, editors. Cryptographic hardware and embedded systems. Lecture notes in computer science, vol. 1717. Berlin, Germany: Springer; 1999. p. 94–108.

[21] Tenca AF, Savaş E, Koç ÇK. A design framework for scalable and unified multipliers in $GF(p)$ and $GF(2^m)$. Int J Comput Res 2004;13(1):68–83.

[22] Tiri K, Akmal M, Verbauwhede I. A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards. In: Proceedings of the 28th European solid-state circuits conference 2002 – ESSCIRC 2002; 2002. p. 403–6.

[23] Toprak Z, Leblebici Y. Low-power current mode logic for improved DPA-resistance in embedded systems. In: IEEE international symposium on circuits and systems 2005 – ISCAS 2005; 2005. p. 1059–62.

[24] Walter Colin D, Samyde David. Data dependent power use in multipliers. In: ARITH'05: Proceedings of the 17th IEEE symposium on computer arithmetic. Washington (DC), USA: IEEE Computer Society; 2005. p. 4–12.

[25] Wolkerstorfer Johannes. Dual-field arithmetic unit for $GF(p)$ and $GF(2^m)$. In: Kaliski Jr BS, Koç ÇK, Paar C, editors. Cryptographic hardware and embedded systems. Lecture notes in computer science, vol. 2523. Berlin, Germany: Springer; 2002. p. 500–14.