Time Efficient Dual-Field Unit for Cryptography-Related Processing

Alessandro Cilardo and Nicola Mazzocca

Università degli Studi di Napoli Federico II
Dipartimento di Informatica e Sistemistica
via Claudio 21, 80125 Naples, Italy
acilardo@unina.it

Abstract. Computational demanding public key cryptographic algorithms, such as Rivest-Shamir-Adleman (RSA) and Elliptic Curve (EC) cryptosystems, are critically dependent on modular multiplication for their performance. Modular multiplication used in cryptography may be performed in two different algebraic structures, namely $GF(N)$ and $GF(2^n)$, which normally require distinct hardware solutions for speeding up performance. For both fields, Montgomery multiplication is the most widely adopted solution, as it enables efficient hardware implementations, provided that a slightly modified definition of modular multiplication is adopted. In this paper we propose a novel unified architecture for parallel Montgomery multiplication supporting both $GF(N)$ and $GF(2^n)$ finite field operations, which are critical for RSA ad ECC public key cryptosystems. The hardware scheme interleaves multiplication and modulo reduction. Furthermore, it relies on a modified Booth recoding scheme for the multiplicand and a radix-4 scheme for the modulus, enabling reduced time delays even for moderately large operand widths. In addition, we present a pipelined architecture based on the parallel blocks previously introduced, enabling very low clock counts and high throughput levels for long operands used in cryptographic applications. Experimental results, based on $0.18\mu m$ CMOS technology, prove the effectiveness of the proposed techniques, and outperform the best results previously presented in the technical literature.

1 Introduction

The increasing centrality of networking and Internet applications are stimulating an ever-growing demand for high-performance implementations of cryptographic algorithms and protocols. Two widely adopted public-key cryptosystems, in particular, are the Rivest-Shamir-Adleman (RSA) [11] and the Elliptic Curve (EC) [1] cryptosystems. While various standardization bodies recommend prime fields $GF(N)$ or binary extension fields $GF(2^n)$ for elliptic curve cryptosystems, RSA cryptography is essentially based on integer modular arithmetic, similar in its implementation to $GF(N)$ operations. Both types of finite fields have in common that the multiplication of elements implies a reduction operation, either modulo a prime N or modulo an irreducible binary polynomial $N(x)$ of degree n.

C. Piguet, R. Reis, and D. Soudris (Eds.): VLSI-SoC 2008, IFIP AICT 313, pp. 191–210, 2010.

The so-called Montgomery algorithm [9] has proved to be the most effective implementation technique for modular multiplication [2, 17]. It is in fact based on a slightly different definition of the modular product, which enables particularly efficient implementations.

Originally introduced for integer numbers (and thus for $GF(N)$ arithmetic), Montgomery multiplication has been effectively extended to binary fields $GF(2^n)$ [8]. As a consequence, during the last years several works have addressed the problem of implementing *unified* arithmetic blocks, suitable for computing operations in both fields using the same underlying hardware [4, 6, 12, 13, 14, 18].

In this paper, we propose a novel unified architecture for parallel Montgomery multiplication supporting both $GF(N)$ and $GF(2^n)$ operations. The hardware unit interleaves multiplication and modulo reduction in a parallel scheme. Furthermore, it relies on a modified Booth recoding technique for the multiplicand and a radix-4 scheme for the modulus, enabling reduced time delays for moderately large operand widths. We also present a pipelined architecture based on the parallel component previously introduced, enabling very low clock counts and high throughput levels for long operands used in cryptographic applications. Experimental results, based on $0.18\mu m$ CMOS technology, prove the effectiveness of the proposed techniques, and outperform the best results previously presented in the technical literature.

The paper is structured as follows. Section 2 provides a brief introduction to the properties of Montgomery multiplication algorithm. Section 3 presents the state-of-the-art of architectures suitable for unified integer/$GF(N)$ and $GF(2^n)$ arithmetic. Section 4 describes the proposed parallel arithmetic unit supporting unified Montgomery multiplication. Section 5 presents a high-throughput pipelined core based on the previously introduced parallel multiplier. Section 6 presents our results and compares them to the state-of-the-art. Section 7 concludes the paper with some final remarks.

2 Modular Multiplication Algorithm

A slight variant of standard modular multiplication, Montgomery multiplication performs the following operation:

$$A \cdot B \cdot R^{-1} \bmod N$$

where $R = 2^n$ is a power of two and n is equal to, or slightly larger than the number of bits in the modulus N, ensuring $R > N$. The value R^{-1} is the inverse of R modulo N, i.e. a number such that $R^{-1}R \bmod N = 1$. In order for such a number to exist, it suffices that $\gcd(N, R) = 1$. Since in both Elliptic Curve cryptography based on prime fields and in RSA cryptography N is always an odd number, this condition is always satisfied when R is a power of two. Montgomery multiplication can be performed with the following algorithm.

Algorithm 1. *Montgomery Modular Multiplication*

```
Input:
    $N$, $R$ and $\tilde{N}$ such that $R \cdot R^{-1} - N \cdot \tilde{N} = 1$,
    $A, B < N$
Output:
    $P \equiv A \cdot B \cdot R^{-1} \bmod N, \quad P < N$
Algorithm:
    1. $Q = AB \cdot \tilde{N} \bmod R$
    2. $P = \frac{AB + Q \cdot N}{R}$
    3. if $P > N$ then $P = P - N$
```

The above algorithm returns a quantity P which is congruent with $AB \cdot R^{-1}$ modulo N (step 2), and is less than N (at step 2, $P = \frac{AB+Q \cdot N}{R} < \frac{N \cdot N + Q \cdot N}{R} < \left[\frac{N}{R} + \frac{Q}{R}\right] \cdot N < 2N$). The multiple $Q \cdot N$ of the modulus is defined at step 1 in such a way as to make the quantity $AB + Q \cdot N$ divisible by R [9].

An interesting property enabled by Montgomery multiplication is the possibility to work on *N-residues* of numbers, defined as $\overline{A} = A \cdot R \bmod N$. It can be easily seen that the Montgomery product of two numbers in N-residue form is still in N-residue form: $\overline{A} \cdot \overline{B} \cdot R^{-1} \bmod N = AR \cdot BR \cdot R^{-1} \bmod N = (AB) \cdot R \bmod N = \overline{AB}$. This also holds true for modular addition: $(\overline{A} + \overline{B}) \bmod N = \overline{A + B}$. All operations used in RSA and EC cryptography can be reduced to a composition of modular multiplications and additions, and can thus always handle operands in Montgomery form.

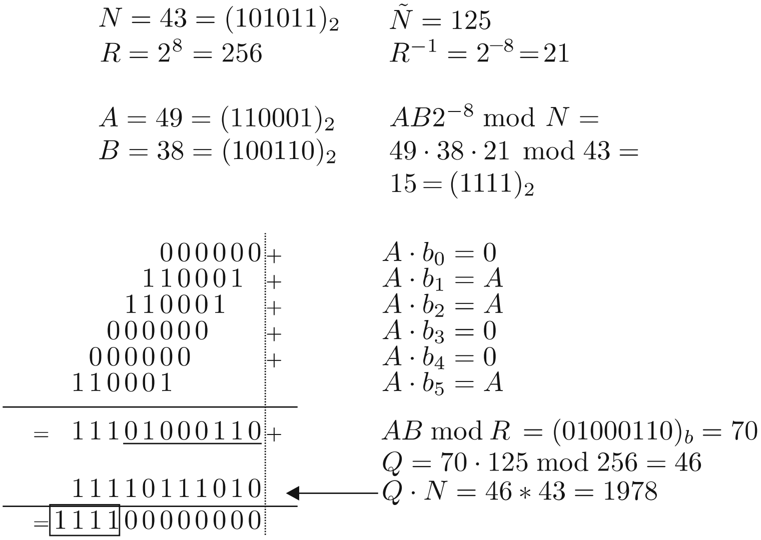

Fig. 1. An example of Montgomery multiplication execution

Notice that Algorithm 1 requires a magnitude comparison (Step 3) in order to ensure the result is actually less than the modulus N. However, when many consecutive multiplications are to be performed, we can allow intermediate results to be in the range $[0, 2N[$ with a proper choice for R. In fact, if we choose $R > 4N$, it can be easily seen that the reduction algorithm accepts multiplicands $A, B < 2N$, i.e. not necessarily less than N: $P = \frac{AB+Q \cdot N}{R} < \frac{2N \cdot 2N + Q \cdot N}{R} < \left[\frac{4N}{R} + \frac{Q}{R}\right] \cdot N < 2N$, so the algorithm preserves the invariant that inputs and output are less than $2N$. By avoiding magnitude comparison, the above version of Montgomery algorithm greatly improves performance, so we will refer to this version of the algorithm in the following. Figure 1 provides an example of execution of the Montgomery algorithm variant exploiting the above property.

The central operation of Montgomery algorithm, i.e. the computation of the product $A \cdot B$ and the multiple of the modulus $Q \cdot N$, can be implemented in a very efficient way, as it is suitable for deeply pipelined and systolic implementations [2, 10, 16, 17]. For scalable implementations, a natural choice is to partition operands into words, and process them separately. Precisely, we will refer in this paper to the so-called *finely integrated operand scanning* (FIOS) method [7], reported below.

Algorithm 2. *FIOS method for w-bit words*

Input:

$A = \sum_{i=0}^{m-1} A_i (2^w)^i$, $B = \sum_{i=0}^{m-1} B_i (2^w)^i$,
$N = \sum_{i=0}^{m-1} N_i (2^w)^i$, $\tilde{N} = \sum_{i=0}^{m-1} \tilde{N}_i (2^w)^i$,
with $A_i, B_i, N_i, \tilde{N}_i < 2^w$ and
$0 \leq A, B < 2N, \quad m \cdot w \geq 2 + \lceil \log_2 N \rceil$ (i.e. $2^{m \cdot w} > 4N$)

Output:

$P \equiv A \cdot B \cdot 2^{-n} \bmod N$, with $\ n = m \cdot w$

Algorithm:

```
1. $P = 0$
2. for $j = 0$ to $m - 1$
3.     $C = 0$
4.     $Q_j = (P_0 + B_j A_0)\tilde{N}_0 \bmod 2^w$
5.     for $i = 0$ to $m - 1$
6.         $S := P_i + B_j A_i + Q_j N_i + C$
7.         if $(i \neq 0)$ then $P_{i-1} := S \bmod 2^w$
8.         $C := S / 2^w$
9.     $P_{m-1} := C$
```

The w-bit words of operands A, B, and N are processed in two nested loops. During the execution of the algorithm, temporary variables S and C can be stored in a $2w + 1$ bit and $w + 1$ bit register, respectively, while variable P

needs a full precision register since it is shared among consecutive “rows” (i.e., m iterations of the inner loop with constant j).

Authors in [8] extended Montgomery multiplication to binary fields $GF(2^n)$, by adopting polynomial representation and replacing the factor $R^{-1} = 2^{-n}$ with x^{-n}. With polynomial representation, $GF(2^n)$ field elements can be handled as binary polynomials and multiplication can be performed modulo an irreducible polynomial $N(x)$. Addition of $GF(2^n)$ elements is performed as a bitwise XOR of their components, while multiplication/division by powers of x are performed by left/right-shifting an element’s components. As a result, the structure and the basic operations of Montgomery algorithm in $GF(2^n)$ turn out to be very similar to the integer/$GF(N)$ case. Essentially, the control-flow of the algorithm (including the above FIOS variant) remains unchanged, shift operations are also identical, while integer addition is replaced by a bitwise XOR. The $GF(2^n)$ counterpart of Algorithm 2 is presented, for example, in [13].

3 State-of-the-Art in Unified Field Arithmetic

Since the structure of Montgomery variants for $GF(N)$ and $GF(2^n)$ are similar, several authors have proposed *unified* hardware solutions for computing both operations with the same processing unit. To enable this approach, Savaş et al. proposed in [14] a basic building block able to perform a one-digit addition in both $GF(N)$ and $GF(2^n)$ fields. The basic component is the *Dual Field Adder*, i.e. an ordinary full adder whose carry input can be disabled, so that the sum output is simply the XOR of the two input bits (i.e., their $GF(2)$ sum). Figure 2 shows a possible implementation of such a component. Based on a similar idea, Großschädl [4] proposed a bit-serial unified multiplier processing the multiplicand in full precision. Montgomery modular reduction is computed by interleaving the addition of partial products and the modulus. A hardware

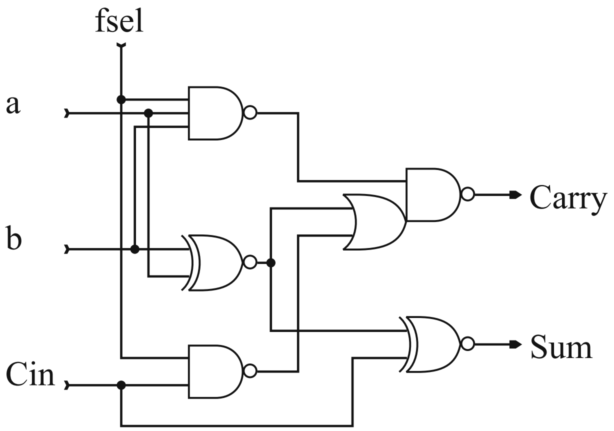

Fig. 2. An implementation of the Dual Field Adder [5, 6]

solution for dual-field arithmetic is also presented by Wolkerstorfer in [18]. The author introduces a low power design enabling short critical paths and high clock frequencies by using carry save adders. In [6], the authors present the design of a low-power multiply/accumulate (MAC) unit for efficient arithmetic in finite fields. The unit combines integer and polynomial arithmetic into a single functional unit supporting both $GF(N)$ and $GF(2^n)$ fields. The emphasis is mostly put on power consumption, as the authors show that a properly designed unified multiplier may consume significantly less power if used in polynomial mode compared to integer mode.

The fastest solution for unified field multiplication was proposed by Satoh and Takano [13]. They present a scalable elliptic curve cryptographic processor supporting both $GF(N)$ and $GF(2^n)$ finite fields. The core of the processor is a parallel dual-field multiplier, based on a Wallace tree scheme. The delay for a multiplication is logarithmic in the input-size, although it is different for the two types of fields. In fact, a sub-portion of the Wallace tree is used for obtaining a $GF(2^n)$ product, while the whole structure, including a fast carry propagation adder, is required for $GF(N)$ operations. The authors evaluate different parallelisms, developing the multiplier for word sizes of 8, 16, 32, or 64 bits, depending on the desired trade-off between area requirements and performance. One advantage of their approach is that it does not require any special full adder, such as the dual-field adder, unlike works in [4, 6, 14] and others. This makes it possible to optimize the partial product addition network. Furthermore, at a higher level, the performance of point multiplication over an elliptic curve is improved by converting on-the-fly the integer multiplicand in a redundant form.

Finally, a recent solution proposes a fast modular arithmetic-logic unit [12] that is scalable in the digit size and the field size. The datapath is based on chains of carry save adders to speed up arithmetic operations over large integers in $GF(N)$. This enables efficient execution of modular multiplication and addition/subtraction. The unit is prototyped in FPGA technology achieving interesting throughput levels, although inferior to the ASIC-based work presented in [13].

4 Parallel Montgomery Multiplier

In this section, we propose a novel unified architecture for parallel Montgomery multiplication supporting both $GF(N)$ and $GF(2^n)$ operations. Unlike previously proposed parallel multipliers, such as the solution in [6, 13], the hardware unit merges multiplication and Montgomery reduction, allowing a word-level modular multiplication to be performed is a single cycle. The proposed multiplier relies on a modified Booth recoding scheme for integer multiplication, and a radix-4 scheme for $GF(2^n)$ multiplication and Montgomery reduction. As a result, the number of partial products to be added in the parallel unit can be approximately halved, resulting in both reduced area and improved speed.

The basic full-precision algorithm for a radix-4 digit-serial interleaved Montgomery multiplication is given below (see for example [15]). For the sake of clarity, we refer to the integer/$GF(N)$ version of the algorithm. As explained in Section 2, the extension to binary fields $GF(2^n)$ is straightforward, provided that a dual-field data path is available.

Algorithm 3. *Radix-4 Montgomery Modular Multiplication*

Input:

$2 < N < 4^k$,
$\tilde{N}$ such that $4^{k+1} \cdot 4^{-(k+1)} - N \cdot \tilde{N} = 1$,
$A = \sum_{i=0}^{k} A_i 4^i < 2N, \quad B = \sum_{i=0}^{k} B_i 4^i < 2N, \quad$ with $A_i,\, B_i < 4$

Output:

$P \equiv A \cdot B \cdot 4^{-(k+1)} \bmod N, \quad P < 2N$

Algorithm:

```
$P = 0$
for $i = 0$ to $k$
    $Q_i = (P_0 + B_i \cdot A_0) \cdot \tilde{N}_0 \bmod 4$
    $P = (P + B_i \cdot A + Q_i \cdot N)/4$
```

It can be easily proved that, by using $k+1$ iterations (i.e., by computing $A \cdot B \cdot 4^{-(k+1)} \bmod N$, $A, B < 2N$) the final value of P is still less than $2N$. In fact, we have $P = \frac{A \cdot B + Q \cdot N}{4^{k+1}} < \left[\frac{4N}{4^{k+1}} + \frac{Q}{4^{k+1}}\right] \cdot N < 2N$, where $Q = \sum_{i=0}^{k} Q_i 4^i$. Notice that Q_i only depends on the two least significant bits of $(P_0 + B_i \cdot A_0)$ and N, so it can be computed by a simple circuit or a look-up table. Its value is defined in such a way as to make the least significant digit of $(P + B_i A + Q_i N)_4$ zero at each iteration. Figure 3 gives an example of radix-4 Montgomery multiplication execution.

In the following, we will call $AA^{(i)}$ and $NN^{(i)}$ a partial product $B_i \cdot A$ and a multiple of the modulus $Q_i \cdot N$, respectively. In the case of radix-4, B_i and Q_i are 2-bit numbers. Thus, the value sets of $AA^{(i)}$ and $NN^{(i)}$ are as follows:

$$AA^{(i)} \in \{0, A, 2A, 3A\}, \quad NN^{(i)} \in \{0, N, 2N, 3N\}$$

requiring two extra adders to compute $3A$ and $3N$ on the fly. In the case of $GF(2^n)$ operations, using polynomial representation, $B_i(x)$ and $Q_i(x)$ are polynomial of degree less than 2, so the value sets of $AA^{(i)}(x)$ and $NN^{(i)}(x)$ are as follows:

$$AA^{(i)}(x) \in \{0,\ A(x),\ xA(x),\ xA(x) + A(x)\}$$
$$NN^{(i)}(x) \in \{0,\ N(x),\ xN(x),\ xN(x) + N(x)\}$$

In standard multipliers, Booth recoding scheme is normally used in order to avoid the expensive calculation of the multiple $3A$ in the $AA^{(i)}$ value set. The recoding scheme takes the bits of the multiplier $(b_{2i+1}, b_{2i}, b_{2i-1})$ as input and generates a recoded $AA^{(i)}$ according to Table 1, where b_{-1} is defined to be 0. As a consequence, Booth recoding scheme transforms the value set of $AA^{(i)}$ into $\{-2A, -A, 0, +A, +2A\}$. All elements in the set are calculated with simple operations such as bit inversion and/or bit shift. For $GF(2^n)$ operations, elements are

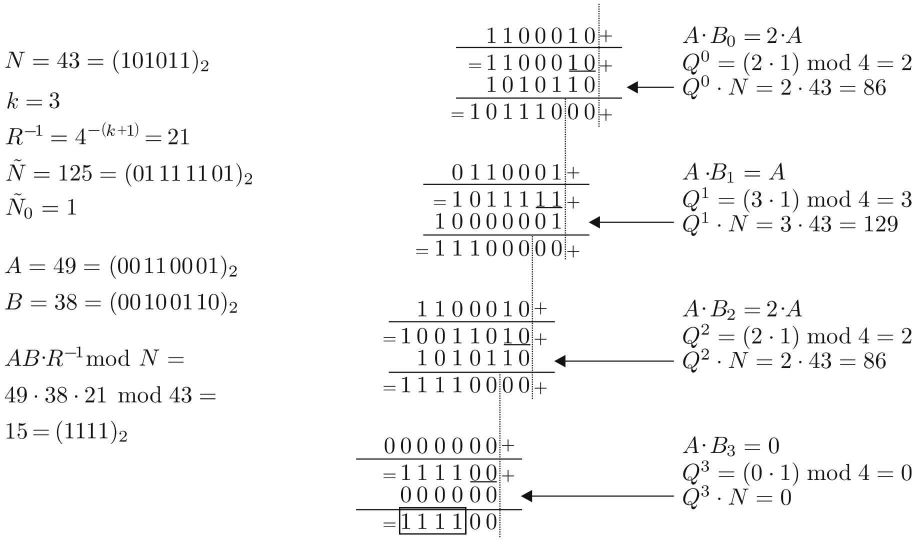
$N = 43 = (101011)_2$
$k = 3$
$R^{-1} = 4^{-(k+1)} = 21$
$\tilde{N} = 125 = (01111101)_2$
$\tilde{N}_0 = 1$
$A = 49 = (00110001)_2$
$B = 38 = (00100110)_2$
$AB{\cdot}R^{-1} \bmod N =$
$49 \cdot 38 \cdot 21 \bmod 43 =$
$15 = (1111)_2$
1100010+
=1100010+
1010110
=10111000+
$A{\cdot}B_0 = 2{\cdot}A$
$Q^0 = (2 \cdot 1) \bmod 4 = 2$
$Q^0 \cdot N = 2 \cdot 43 = 86$
0110001+
=1011111+
10000001
=11100000+
$A{\cdot}B_1 = A$
$Q^1 = (3 \cdot 1) \bmod 4 = 3$
$Q^1 \cdot N = 3 \cdot 43 = 129$
1100010+
=100110010+
1010110
=111100000+
$A{\cdot}B_2 = 2{\cdot}A$
$Q^2 = (2 \cdot 1) \bmod 4 = 2$
$Q^2 \cdot N = 2 \cdot 43 = 86$
0000000+
=111100+
000000
=111100
$A{\cdot}B_3 = 0$
$Q^3 = (0 \cdot 1) \bmod 4 = 0$
$Q^3 \cdot N = 0$

Fig. 3. An example of radix-4 Montgomery multiplication execution

Table 1. Partial product generation for integers and binary polynomials

Field Select	**Three input bits**			**Recoded digit**	**Recoded partial product**	**Control signals**		
$fsel$	b_{2i+1}	b_{2i}	b_{2i-1}	B_i	$AA^{(i)}$	inv	trp	shl
1	0	0	0	0	0	–	0	0
1	0	0	1	1	$+A$	0	1	0
1	0	1	0	1	$+A$	0	1	0
1	0	1	1	+2	$+2A$	0	0	1
1	1	0	0	−2	$-2A$	1	0	1
1	1	0	1	−1	$-A$	1	1	0
1	1	1	0	−1	$-A$	1	1	0
1	1	1	1	0	0	–	0	0
0	0	0	0	0	0	–	0	0
0	0	0	1	0	0	–	0	0
0	0	1	0	1	$A(x)$	0	1	0
0	0	1	1	1	$A(x)$	0	1	0
0	1	0	0	x	$xA(x)$	0	0	1
0	1	0	1	x	$xA(x)$	0	0	1
0	1	1	0	$x+1$	$xA(x)+A(x)$	0	1	1
0	1	1	1	$x+1$	$xA(x)+A(x)$	0	1	1

handled as binary polynomials. In this case, a pure radix-4 polynomial multiplication is adopted. In other words, multiples $AA^{(i)}(x)$, calculated as in Table 1, only depend on radix-4 digits (b_{2i+1}, b_{2i}).

For the proposed parallel Montgomery multiplier, in addition to summing partial products $AA^{(i)}$, we also need to sum modulus multiples $NN^{(i)}$ (or $NN^{(i)}(x)$ for $GF(2^n)$ multiplication). In [15] authors adopt a method named *Montgomery recoding scheme* to change the possible values of $NN^{(i)}$ so that they can all be obtained by simple shifts and inversions, similar to Booth recoding. Let (sp_1, sp_0) be the 2 bits in the least significant digit (LSD) of the partial product to be reduced $SP = P + AA$ and (n_1, n_0) be the 2 bits in the LSD of the modulus N. According to the input condition that N has to be odd, n_0 is always 1. Then, Montgomery recoding scheme takes (sp_1, sp_0, n_1) as input and generates a recoded $NN^{(i)}$ value according to Table 2, where Q_i represents the recoded quotient digit for an $NN^{(i)}$ multiple at the i-th iteration. Montgomery recoding scheme transforms the value set of NN into $\{-N, 0, +N, +2N\}$.

In polynomial mode the addition becomes a bitwise XOR. For this reason, we need to sum a different value of $NN^{(i)}(x)$ in order to reduce the least significant digits $(sp_1, sp_0)_2$ of $SP(x) = P(x) + AA(x)$. Notice that, in order to perform modular multiplication in $GF(2^n)$ with the same recoding scheme, we use an additional control signal, $fsel$ (field select), which allows us to switch between integer-mode ($fsel = 1$) and polynomial mode ($fsel = 0$). In Table 2 we show the unified Montgomery recoding scheme, including polynomial mode for $GF(2^n)$.

Due to the two recoding schemes, it is easy to calculate all the elements in the value sets of $AA^{(i)}$ and $NN^{(i)}$. Notice that, for integer multiplication, this technique changes the range of the Montgomery algorithm output, which may now be negative.

Table 2. Montgomery moduli generation for integers and binary polynomials

Field Select	**Three input bits**			**Recoded quotient**	**Recoded modulus**	**Control signals**		
$fsel$	sp_1	sp_0	n_1	Q_i	$NN^{(i)}$	inv	trp	shl
1	0	0	0	0	0	–	0	0
1	0	0	1	0	0	–	0	0
1	0	1	0	-1	$-N$	1	1	0
1	0	1	1	$+1$	$+N$	0	1	0
1	1	0	0	$+2$	$+2N$	0	0	1
1	1	0	1	$+2$	$+2N$	0	0	1
1	1	1	0	$+1$	$+N$	0	1	0
1	1	1	1	-1	$-N$	1	1	0
0	0	0	0	0	0	–	0	0
0	0	0	1	0	0	–	0	0
0	0	1	0	1	$N(x)$	0	1	0
0	0	1	1	$x+1$	$xN(x)+N(x)$	0	1	1
0	1	0	0	x	$xN(x)$	0	0	1
0	1	0	1	x	$xN(x)$	0	0	1
0	1	1	0	$x+1$	$xN(x)+N(x)$	0	1	1
0	1	1	1	1	$N(x)$	0	1	0

The core of the proposed parallel Montgomery multiplier is made of a sequence of *Partial Product Generators* (PPGs) and *Montgomery Modulues Generators* (MMGs), wired as in Figure 4. Their outputs are summed together, making up an unrolled implementation of the loop in Algorithm 3.

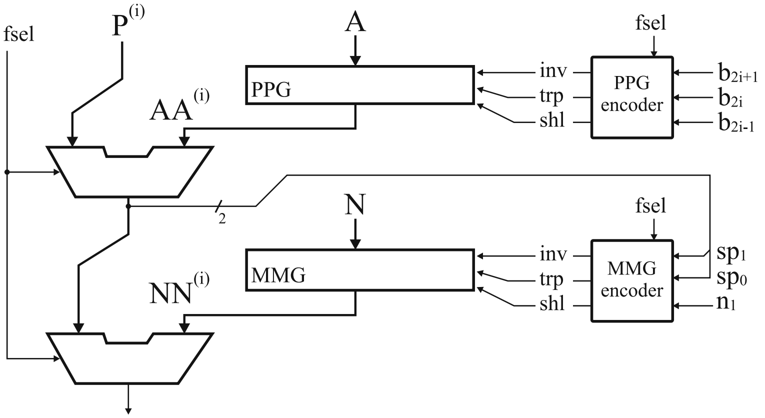

Fig. 4. The basic row in the proposed radix-4 parallel Montgomery multiplier

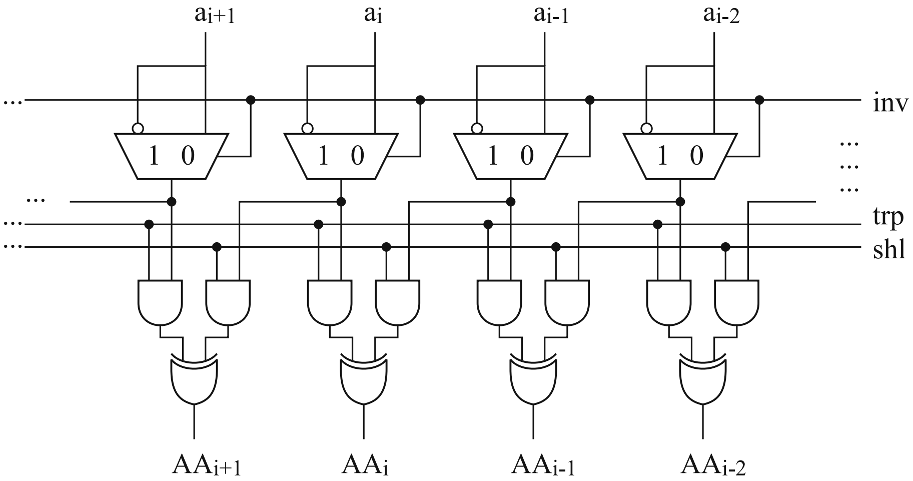

Fig. 5. The internal structure of a Partial Product Generator (PPG) [6]. A similar circuit is used for the Montgomery Modulus Generator (MMG).

The structures of PPGs and MMGs are identical, and are similar to that described in [6]. The corresponding circuit is depicted in Figure 5. PPGs and MMGs are controlled by an encoder via the three signals *inv* (invert), *trp* (transport), and *shl* (shift left), which represent the recoded digit B_i and the recoded quotients Q_i, respectively. Precisely, when $inv = 1$, the corresponding modulus is negative, i.e. $NN^{(i)} = -N$. Control signal $trp = 1$ means $NN^{(i)} = N$ (no left-shift). On the other hand, when $shl = 1$, a 1-bit left-shift has to be performed, i.e. $NN^{(i)} = 2N$. Finally, $NN^{(i)} = 0$ is generated by $trp = shl = 0$. Notice that in $GF(2^n)$ mode, i.e. when $fsel = 0$, the input value $inv = 0$, $trp = 1$, $shl = 1$ generates the multiple $xN(x) + N(x)$ needed for radix-4 Montgomery reduction. Similar considerations hold true for the Partial Product Generator used to calculate the values of $AA^{(i)}$.

Selection signals inv, trp, and shl depend on the multiplier digit bits $b_{2i+1}, b_{2i}, b_{2i-1}$, in the case of PPG, and the two least significant bits (sp_1, sp_0) of SP and n_1, in the case of MMG, according to the equations below, derived from Table 2. For PPGs, selection signals can be written as follows:

$$\begin{aligned}
inv &= fsel \cdot b_{2i+1} \\
trp &= \overline{fsel} \cdot b_{2i} + b_{2i} \cdot \overline{b_{2i-1}} + fsel \cdot \overline{b_{2i}} \cdot b_{2i-1} \\
shl &= \overline{fsel} \cdot b_{2i+1} + b_{2i+1}\overline{b_{2i}} \cdot \overline{b_{2i-1}} + fsel \cdot \overline{b_{2i+1}} \cdot b_{2i} \cdot b_{2i-1}
\end{aligned} \qquad (1)$$

For MMGs, selection signals can be written as follows:

$$\begin{aligned}
inv &= fsel \cdot \overline{sp_1} \cdot \overline{n_1} + \overline{fsel} \cdot sp_1 \cdot sp_0 \cdot n_1 \\
trp &= sp_0 \\
shl &= sp_1 \cdot \overline{sp_0} + \overline{fsel} \cdot sp_1 \cdot \overline{n_1} + \overline{fsel} \cdot \overline{sp_1} \cdot sp_0 \cdot n_1
\end{aligned} \qquad (2)$$

A parallel $(w \times w)$-bit multiplier for signed/unsigned modular multiplication contains $\lfloor w/2 \rfloor + 1$ PPGs and $\lfloor w/2 \rfloor + 1$ MMGs and the same number of PPG/MMG encoder circuits generating selection signals inv, trp, and shl.

Partial products $AA^{(i)}$ and moduli $NN^{(i)}$ are $w + 2$ bits long as they are represented in two's complement form. Besides a bitwise complement of their binary representation, negative multiples need a 1 to be added at the least significant position of the partial product. Let $ca^{(i)}$, $cn^{(i)}$ denote such bits. We will thus have $ca^{(i)} = 1$ and $cn^{(i)} = 1$ when the partial products $AA^{(i)}$ and the Montgomery moduli $NN^{(i)}$ are negative, respectively.

Notice that the parallel multiplier handles internal operands in carry-save form to reduce the architectural critical path. Special care must be put, in this case, for summing negative numbers. In principle, we would need to sign extend possibly negative partial products $AA^{(i)}$ and moduli $NN^{(i)}$ to full $2w$-bit length, causing a large waste of full-adders in each row of the multiplier. By recoding the addends, however, we can have only positive-weight bits to be added in the multiplier, provided that a suitable constant K is added along with them as the last row in the multiplier array [3]. Let $P = (-2^n)p_n + \sum_{i=0}^{n-1} 2^i p_i$ be a two's complement number. Recoding works as follows:

$$P = (-2^n)p_n + \sum_{i=0}^{n-1} 2^i p_i = -2^n + \left[2^n \overline{p_n} + \sum_{i=0}^{n-1} 2^i p_i \right]$$

where all number's components have a positive weight, while the only negative term is constant. If we have many partial products P to be summed together, we can thus recode them as shown above, sum their positive components p_i (including $\overline{p_n}$) by adopting a usual array multiplier, separate their constant terms -2^n and accumulate them in a single full-length constant K to be added as the last row.

Some further optimizations can be applied to reduce the architectural critical path of the design. Let (S, C) denote a carry-save pair. In a non-optimized Montgomery multiplier with modified Booth recoding, the sum of the partial products and the Montgomery moduli in the carry-save stages (CSAs) proceeds as follows:

$$\begin{array}{l} \ldots \\ \left(Stmp^{(i)}, Ctmp^{(i)}\right) = AA^{(i)} + S^{(i)} + C^{(i)} \\ \left(S^{(i+1)}, C^{(i+1)}\right) = N^{(i)} + Stmp^{(i)} + Ctmp^{(i)} \\ \left(Stmp^{(i+1)}, Ctmp^{(i+1)}\right) = AA^{(i+1)} + S^{(i+1)} + C^{(i+1)} \\ \ldots \end{array}$$

$(Stmp^{(i)}, Ctmp^{(i)})$ is given by the sum of the i-th recoded partial product $AA^{(i)}$ and the previous $AA^{(j)}$, $0 \leq j < i$ with the recoded moduli $NN^{(j)}$, $0 \leq j < i$. Recoding of partial products and moduli, however, also implies the sum of the sign bits ca and cn. In principle, this would require the use of two additional CSA stages. Indeed, since ca and cn are in the right-most positions of partial products and moduli, we can juxtapose them with other partial products and moduli down in the multiplier array, since these are left-shifted and so leave free slots on the right. For the sake of clarity, Figure 6 gives a practical example of this organization, for the case $w = 6$. The generic stage within the proposed multiplier scheme performs the following operation:

$$\begin{array}{l} \ldots \\ \left(Stmp^{(i)}, Ctmp^{(i)}\right) = S^{(i-1)} + C^{(i-1)} + AA^{(i+1)} + ca^{(i)} \\ \left(S^{(i)}, C^{(i)}\right) = Stmp^{(i)} + Ctmp^{(i)} + NN^{(i)} + cn^{(i)} \\ \ldots \end{array}$$

Overall, we need:

- $\lfloor w/2 \rfloor + 1$ CSA stages to compute $Stmp^{(i)}$, $Ctmp^{(i)}$
- $\lfloor w/2 \rfloor + 1$ CSA stages to compute $S^{(i)}$, $C^{(i)}$

The main optimizations adopted consist in (see Figure 6):

- reorganizing the sum of the LSB $ca^{(i)}$ and $cn^{(i)}$ of the output carry vector in order to avoid additional CSA stages. Notice that, although interchangeable for the accumulation of partial products and moduli, bits $ca^{(i)}$ are needed for the determination of the next modulus $NN^{(i+1)}$ to be summed. The MMG selection circuit must take this into account, and read also the bit $ca^{(i)}$ to anticipate the evaluation of $NN^{(i+1)}$
- postponing the sum of the least significant bits $\{s_1^{(i)}, s_0^{(i)}, c_0^{(i)}\}$ of $S^{(i)}$ and $C^{(i)}$ respectively, to save area and CSA stages. Similar to the previous optimization, these operations imply a complication of the MMG selection network,

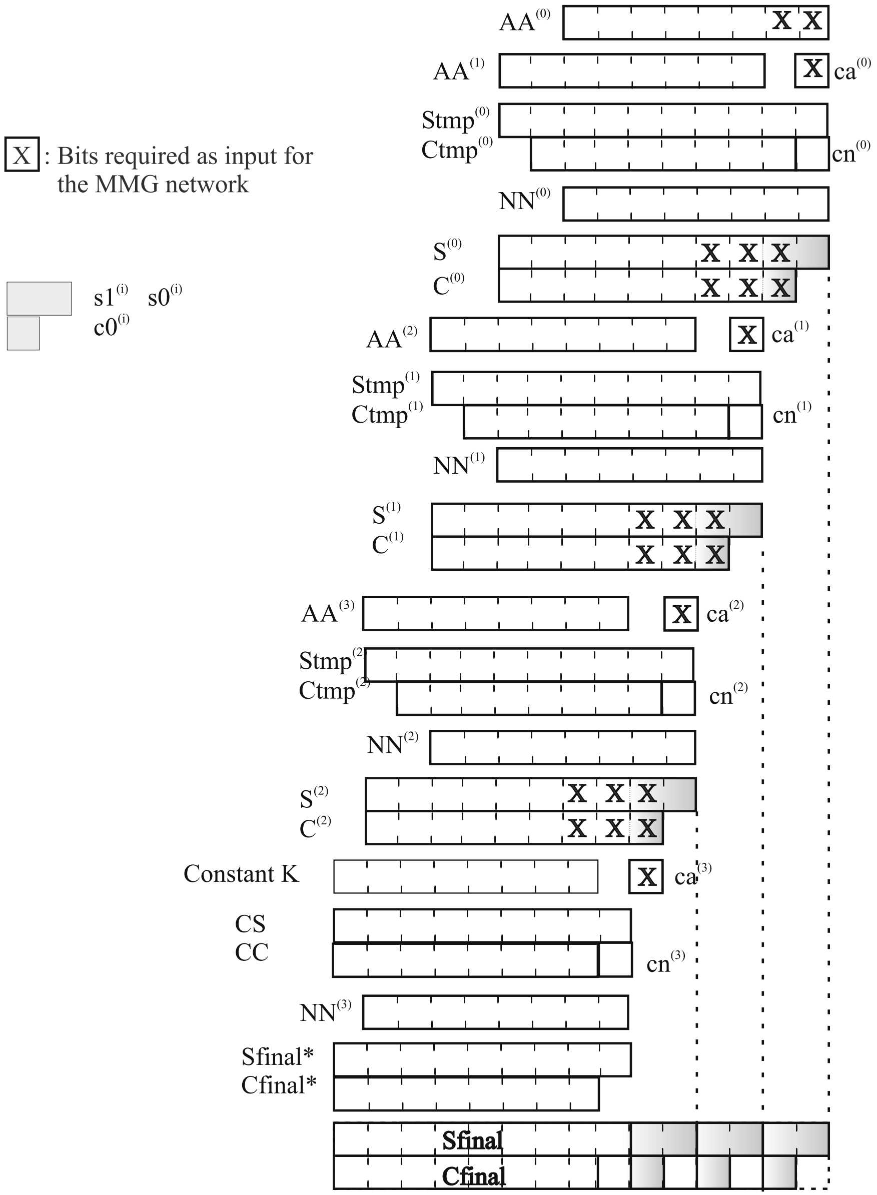

Fig. 6. Addition of Partial Products and Montgomery Moduli with Booth Recoding in an optimized scheme for $w = 6$

which needs more inputs to infer the values of bits sp_1, sp_0, handled here in redundant, carry-save form

- reversing the order of the sum of $AA^{(i)}$, $NN^{(i)}$, in order to improve the critical path. This operation does not alter the computation of $NN^{(i)}$, due to the encoding network previously described, which tests the bits needed for the computation of the modulus before the addition of the $AA^{(i+1)}$ vector.

After the final stage, we need a *Dual-Field Carry-Look-Ahead* adder (not shown in Figure 6) that converts the Carry/Sum pair back to non-redundant form. The structure of the Dual-Field Carry Look-Ahead is depicted in Figure 7. The essential idea is to disable carry generation throughout the adder structure in $GF(2^n)$ mode, i.e. when $fsel = 0$. In this case, all internal carry signals C_i are zero, independent of propagate conditions P_i. As a result, output bits S_i coincide with propagate signals $P_i = a_i \oplus b_i$, i.e. a $GF(2)$ sum. The fundamental advantage of this solution is that it enables the reuse of highly-optimized fast carry look-ahead circuits which are normally available for a given target technology.

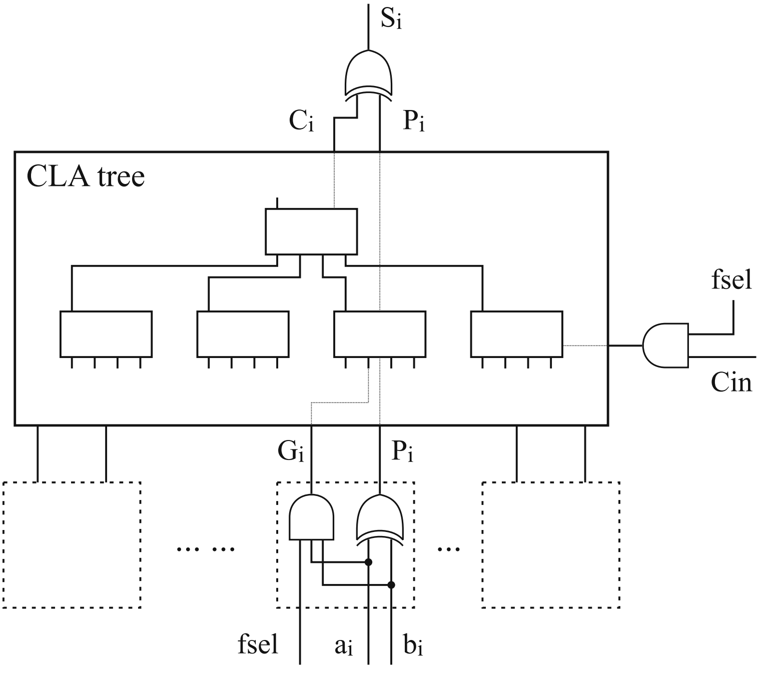

Fig. 7. Dual-Field Carry Look-Ahead adder

5 Pipelined Montgomery Multiplier

Previous works (e.g. Satoh and Takano's 64-bit multiplier [13]) suggest that it is normally convenient to adopt a large parallelism for achieving higher throughput levels. Our parallel architecture has a relatively complex selection network and a linear critical path, which results in large time delays as the word size increases. In order to achieve high throughput levels and propose a scalable scheme, we present in this section a pipelined architecture, using the parallel unit as the basic building block. The architecture can process single words of w bits. By partitioning long operands into w-bit words, a full-length Montgomery multiplication can be carried out based on the FIOS variant of the Montgomery algorithm (see Algorithm 2).

We implemented the unit for a bit length w of 64 bits. Figure 8 shows the internal structure of a 64x64-bit unit composed of eight pipelined modules. The "smaller" multipliers on the right are in fact four instances of the parallel unit presented in the previous section: in other words, they can generate the recoded multiples (i.e. Q_i recoded as the signals *shl*, *trp*, *inv*) of the modulus N and the multiplicand A for the whole row, in addition to adding them. The four "larger" multipliers on the left side of Figure 8, on the other hand, only need to sum the multiples of N and A, as determined by right-multipliers. Since left-multipliers are much simpler in their structure and have consequently a shorter delay, they are designed so that they process longer data. Furthemore, right-multipliers also need an additional input signal, called $first\_word$, which can enable/disable the generation of multiples of the modulus Q_i. This is necessary to process intermediate words during a row scanning of the FIOS algorithm (steps 5-8 in Algorithm 2), where we need to process new w-bit words in the pipelined unit reusing a previously generated value of Q_i.

As we use two's complement representation in the carry-save form, it is desirable to keep intermediate sums in carry-save form and convert the final result back to binary form only at the end of the pipelined structure. We thus need to transfer carry-save numbers between subsequent multiplier modules having different output/input sizes. This required the use of a suitable technique [19] to sign-extend the carry-save pair and properly propagate sign information.

Figure 9 describes how the pipelined unit is used to process multi-word operands, showing how the portions of the operands are scheduled in the pipeline. Numbers in parentheses indicate which of the eight blocks in the unit works on which portion of operands A, N, and B at which clock cycle (starting from cycle 1 for the top right-most multiplier). The unit has a latency of eight cycles, introducing a stall at the end of each row only if the number of words m is less than 8. This makes the unit particularly suitable for high-performance multiplication on large multi-word operands, when many words on the same row are to be processed consecutively. The throughput of the architecture is one multiplication word per clock cycle in this case.

Right modules in Figure 8 have a 16×16 bit size, while left modules have a 48×16 bit size. The architecture is designed so that the single blocks, especially the smaller right-multipliers, can be optimized to minimize the clock period. Notice that, with a slight modification to the scheme of Figure 8, the first and the last row (possibly connected to an external bus) may be designed with a smaller height than the multipliers in the second and third row, so as to balance the delay of each stage in the pipeline. The carry-save stages are followed by a Dual-Field Carry-Look-Ahead adder, not shown in Figure 8, converting results back to the non-redundant form.

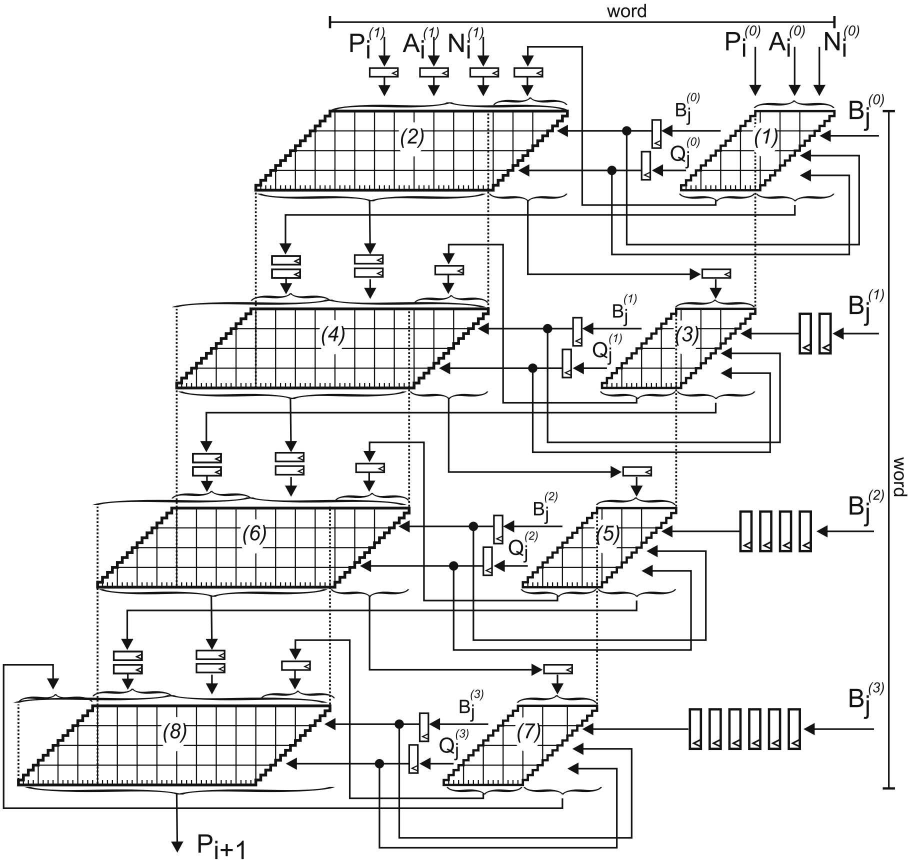

Fig. 8. Pipelined architecture of the arithmetic core. Superscript numbers in parentheses indicate the different portions into which a single w-bit word is partitioned inside the pipelined unit.

The overall architecture of the dual-field multiplication unit is shown in Figure 10. From the scheme in Figure 9 it is clear that at the beginning of each row we need to drive in the unit three different words, namely A_0, N_0, and B_j, while the words of the intermediate result P are stored internally in a dedicated memory. This is the only case when we need three concurrent accesses to the external memory. To overcome this problem and limit the number of external buses, we observe that it is convenient to store the first word of the modulus N, N_0, in an internal register. This trick only requires w additional flip-flops and some selection logic, independent of the full size of the operands and the modulus. N_0 is stored before starting a multiplication (or a sequence of multiplications sharing the same modulus). As a consequence, at the beginning of each row in the multiplication pipeline we only need A_0 and B_j, while for the subsequent words we need A_i and N_i (B_j is constant through the row), which are driven into the multiplication unit through the same pair of buses.

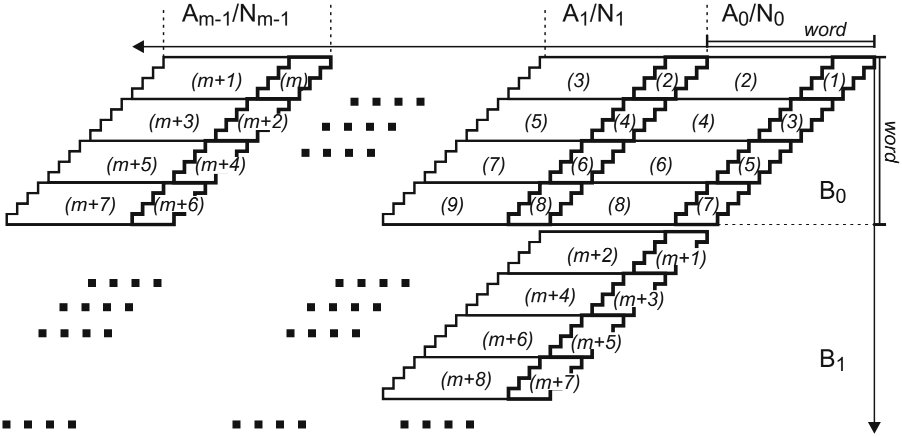

Fig. 9. Scheduling for a multi-word Montgomery multiplication. A_i, N_i, and B_j are w-bit words. Word sub-portions enter the pipelined w-bit unit according to the schedule indicated in parentheses.

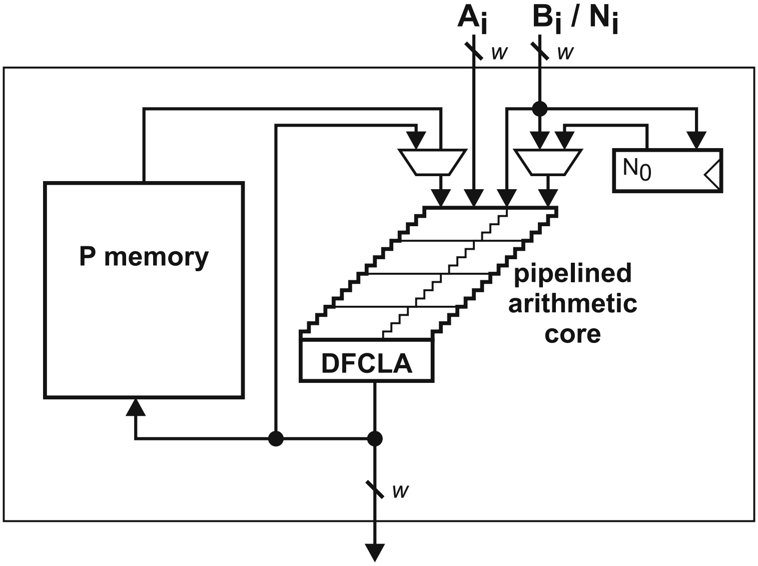

Fig. 10. Overall architecture of the dual-field multiplication unit

6 Experimental Results and Comparisons

The pipelined multiplier core of Figure 8 was described in VHDL and then synthesized for a CM0S $0.18\mu m$ standard cell library technology by using Cadence Build Gates synthesis tool. Post-synthesis area requirements are estimated to be $1316k\mu m^2$, while the minimum clock period is $12.2ns$.

Although there are different related works presenting unified Montgomery multiplication (see Section 3), we only compare our results with the multiplier introduced in [13], since it achieves the highest throughput among the various works available in the literature. Both their work and ours are synthesized as a CMOS ASIC, but the design in [13] relies on a $0.13\mu m$ technology, more advanced than the $0.18\mu m$ target used in our design. When implemented in the same technology, our solution is thus likely to enable even better improvements than emphasized in the following discussion. The table below reports some results referred to integer (i.e. $GF(N)$) modular multiplication for different operand lengths, choosing the field sizes indicated by NIST standards for elliptic curve cryptography. Performance improvements are especially evident in terms of clock counts.

	Satoh and Takano [13]		**This work**	
	ASIC $0.13\mu m$ clock period: 7.26 ns		ASIC $0.18\mu m$ clock period: 12.2 ns	
$GF(N)$ field size	clock count	throughput [Mbit/s]	clock count	throughput [Mbit/s]
192	45	587.7	27	582.9
224	66	467.5	36	510.0
256	66	534.3	36	582.9
284	91	429.9	45	517.3
521	231	310.7	90	474.5

Authors in [13] emphasize that a higher frequency could be used if the unified multiplier were used only in $GF(2^n)$ mode, since the output of their unit is connected, in this case, to a subportion of the Wallace tree in the multiplier. If a dual clock frequency were allowed, $GF(2^n)$ operations would be worse in our case, while remaining superior for the more critical integer/$GF(N)$ arithmetic. In the case a dual frequency implementation is not possible, on the other hand, our multiplier has better performance also for $GF(2^m)$, and comparisons with the multiplier in [13] appear similar to those given in the above table for the integer/$GF(N)$ case.

7 Conclusions

The approach presented in this paper, based on dual-field parallel Montgomery multiplication, proves to be a promising choice, especially for the reduction in clock count. As a future work, we plan to study new techniques to further reduce the delay of the parallel Montgomery unit, described in Section 4, thereby improving the clock period and the throughput achievable by the pipelined unit.

Acknowledgements

This work was partially supported by Regione Campania and Ditron S.R.L. in the framework of “Progetto Metadistretto del settore ICT - Misura 3.17:

Sistema di comunicazione per l'integrazione delle informazioni nella distribuzione commerciale dei punti vendita".

References

1. Blake, I.F., Seroussi, G., Smart, N.P.: Elliptic Curves in Cryptography. Cambridge University Press, Cambridge (1999)
2. Blum, T., Paar, C.: High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware. IEEE Transactions on Computers 50, 759–764 (2001)
3. Burgess, N.: Removal Of Sign-Extension Circuitry From Booth's Algorithms Multiplier-Accumulators. Electronics Letters 26, 1413–1415 (1990)
4. Großschädl, J.: A bit-serial unified multiplier architecture for finite fields $GF(p)$ and $GF(2^n)$. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, p. 202. Springer, Heidelberg (2001)
5. Großschädl, J., Kamendje, G.A.: Instruction set extension for fast elliptic curve cryptography over binary finite fields $GF(2^m)$. In: Proceedings of the 14th IEEE Int. Conference on Application-specific Systems, Architectures and Processors (ASAP 2003), pp. 455–468. IEEE Computer Society Press, Los Alamitos (2003)
6. Großschädl, J., Kamendje, G.A.: Low Power Design of a Functional Unit for Arithmetic in Finite Fields $GF(p)$ and $GF(2^m)$. In: Chae, K.-J., Yung, M. (eds.) WISA 2003. LNCS, vol. 2908, pp. 227–243. Springer, Heidelberg (2004)
7. Koç, Ç.K., Acar, T., Kaliski, B.S.: Analyzing and Comparing Montgomery Multiplication Algorithms. IEEE Micro 16, 26–33 (1996)
8. Koç, Ç.K., Acar, T.: Montgomery Multiplication $GF(2^n)$. Designs, Codes and Cryptography 14, 57–69 (1998)
9. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44, 519–521 (1985)
10. Örs, S.B., Batina, L., Preneel, B., Vandewalle, J.: Hardware Implementation of a Montgomery Modular Multiplier in a Systolic Array. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003), p. 184b (2003)
11. Rivest, R.L., Shamir, A., Adleman, L.: A Method for obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM 21, 120–126 (1978)
12. Sakiyama, K., Preneel, B., Verbauwhede, I.: A Fast Dual-Field Modular Arithmetic Logic Unit and its Hardware Implementation. In: Proc. IEEE International Symposium on Circuits and Systems (ISCAS 2006), pp. 787–790 (2006)
13. Satoh, A., Takano, K.: A Scalable Dual-Field Elliptic Curve Cryptographic Processor. IEEE Transanctions on Computers 52, 449–460 (2003)
14. Savaş, E., Tenca, A.F., Koç, Ç.K.: A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, pp. 281–296. Springer, Heidelberg (2000)
15. Son, H.K., Oh, S.G.: Design and Implementation of Scalable Low-Power Montgomery Multiplier. In: Proceedings of the IEEE International Conference on Computer Design (ICCD 2004), pp. 524–531 (2004)
16. Tsai, W.C., Shung, C.B., Wang, S.J.: Two systolic architectures for modular multiplication. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 8, 103–107 (2000)

17. Walter, C.D.: Systolic Modular Multiplication. IEEE Transactions on Computers 42, 376–378 (1993)
18. Wolkerstorfer, J.: Dual-field arithmetic unit for $GF(p)$ and $GF(2^m)$. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 500–514. Springer, Heidelberg (2003)
19. Tenca, A.F., Tawalbeh, L.A.: Carry-Save Representation is Shift-Unsafe: The Problem and Its Solution. IEEE Transanctions on Computers 55, 630–635 (2006)