Efficient Modular Reduction Algorithm in $\mathbb{F}_q[x]$ and Its Application to "Left to Right" Modular Multiplication in $\mathbb{F}_2[x]$

Jean-François Dhem

Gemplus Corporate Product R&D
Security Technologies Department
Card Security Group (STD/CSG/CHA)
Avenue du Jujubier – ZI Athelia IV
F-13705 La Ciotat Cedex
France
jf.dhem@ieee.org

Abstract. This paper describes a new efficient method of modular reduction in $\mathbb{F}_q[x]$ suited for both software and hardware implementations. This method is particularly well adapted to smart card implementations of elliptic curve cryptography over $\mathsf{GF}(2^p)$ using a polynomial representation. Many publications use the equivalent in $\mathbb{F}_2[x]$ of Montgomery's modular multiplication over integers. We show here an equivalent in $\mathbb{F}_q[x]$ to the generalized Barrett's modular reduction over integers. The attractive properties of the last method in $\mathbb{F}_2[x]$ allow nearly ideal implementations in hardware as well as in software with minimum additional resources as compared to what is available on usual processor architecture.
An implementation minimizing the memory accesses is described for both Montgomery's implementation and ours. This shows identical computing and memory access resources for both methods. The new method also avoids the need for the bulky normalization (denormalization) which is required by Montgomery's method to obtain a correct result.

Keywords: Smart card, cryptography, modular multiplication, quotient evaluation, elliptic curves, ECDSA, Montgomery, Barrett, multiply and add without carries, multiplications in $\mathbb{F}_2[x]$.

1 Introduction

Montgomery's multiplication in $\mathbb{F}_2[x]$ is a well known "right to left" modular reduction (see for example [1]) and directly derives from Montgomery's modular reduction over integers [2,3]. It is often used for efficient software implementations of elliptic curves crypto-systems like ECDSA [4], using polynomial representations. The main disadvantage of Montgomery's method is the bulky normalization - denormalization phase required to obtain a correct modular reduction. In the set of integers, the corresponding "left to right" methods [5,6,

C.D. Walter et al. (Eds.): CHES 2003, LNCS 2779, pp. 203–213, 2003.

7] are less efficient in software because they require additional non standard (on general purpose microprocessors) resources (special multiplier) to obtain similar performances. We will show that the generalized Barrett's method can be made more efficient in $\mathbb{F}_2[x]$ so that it can compete directly with the Montgomery's method. Even more, because of the absence of normalization, the memory requirements for data and ROM code of our method are smaller. This could be of main importance in constrained implementations (e.g. smart cards).

The easiest "left to right" method to compute a quotient and a remainder is the one taught at school: the quotient is calculated by first zeroing the upper (most significant) part of the numerator by adding / subtracting a multiple of the denominator. Montgomery's method, however, zeroes the least significant bits of the number to reduce.

In section 2, the way how the quotient is computed in order to perform an improved "left to right" modular reduction in $\mathbb{F}_q[x]$ is described.

In section 3, the particular case of the modular multiplication in $\mathbb{F}_2[x]$ is studied.

2 Quotient Evaluation in $\mathbb{F}_q[x]$

In order to compute $S(x) = U(x) \bmod N(x)$ we could first evaluate, in a scholastic way, the quotient $Q(x)$ defined by the equation $U(x) = Q(x)N(x) + S(x)$ where $S(x)$ and $Q(x)$ are respectively the remainder and the quotient of the Euclidean division [8] of $U(x)$ by $N(x)$. By similarity to computations over integers we define $Q(x) = \left\lfloor \frac{U(x)}{N(x)} \right\rfloor$. The degree p of the polynomial $N(x)$ is noted $\deg(N)$. We also define $\alpha = \deg(U) - \deg(N)$.

In most applications, like in elliptic curve crypto-systems, $N(x)$ is fixed (when working on a given elliptic curve). To speed up the computations, we can precompute $\left\lfloor \frac{x^{p+\beta}}{N(x)} \right\rfloor$ $(= R(x))$, for some value of β defined hereafter. The quotient's evaluation may then be reduced to a multiplication and an appropriate shift (division by a power of x) as shown in equation 1 (see [5] and [7] when working over integers).

$$\hat{Q}(x) = \left\lfloor \frac{\left\lfloor \frac{U(x)}{x^p} \right\rfloor \left\lfloor \frac{x^{p+\beta}}{N(x)} \right\rfloor}{x^\beta} \right\rfloor = \left\lfloor \frac{T(x)R(x)}{x^\beta} \right\rfloor \tag{1}$$

For the comprehension of equation 1, $\lfloor A(x)/B(x) \rfloor$ represents the quotient of the polynomial division of $A(x)$ by $B(x)$, discarding the remainder ($A(x)$ and $B(x)$ are some polynomials). In the next sections we will demonstrate the equivalence of equation 1 with the real quotient $Q(x) = \lfloor U(x)/N(x) \rfloor$.

2.1 Equivalence between $\hat{Q}(x)$ and $Q(x)$

In the present section, $A_k(x)$ represents the polynomial $A(x)$ of degree k (where $A(x)$ is some polynomial).

We can write:

$$\frac{U(x)}{x^p} = \Phi_\alpha(x) + \frac{\varphi_{p-1}(x)}{x^p}$$

where $\Phi_\alpha(x)$ is the quotient and $\varphi_{p-1}(x)$ the remainder of the division. Similarly, we can write:

$$\frac{x^{p+\beta}}{N(x)} = \Lambda_\beta(x) + \frac{\lambda_{p-1}(x)}{N_p(x)}$$

Where $\Lambda_\beta(x)$ is the quotient and $\lambda_{p-1}(x)$ the remainder of the division. We now can write:

$$Q(x) = \left\lfloor \frac{\left(\frac{U(x)}{x^p}\right)\left(\frac{x^{p+\beta}}{N(x)}\right)}{x^\beta} \right\rfloor \tag{2}$$

$$= \left\lfloor \frac{\left(\Phi_\alpha(x) + \frac{\varphi_{p-1}(x)}{x^p}\right)\left(\Lambda_\beta(x) + \frac{\lambda_{p-1}(x)}{N_p(x)}\right)}{x^\beta} \right\rfloor \tag{3}$$

$$= \left\lfloor \frac{\Phi_\alpha(x)\Lambda_\beta(x) + \Phi_\alpha(x)\frac{\lambda_{p-1}(x)}{N_p(x)} + \cancel{\Lambda_\beta(x)\frac{\varphi_{p-1}(x)}{x^p}} + \cancel{\frac{\varphi_{p-1}(x)}{x^p}\frac{\lambda_{p-1}(x)}{N_p(x)}}}{x^\beta} \right\rfloor \tag{4}$$

The second term of equation 4 is nullified only if $\beta \geq \alpha$. In that case only we can write:

$$Q(x) = \left\lfloor \frac{\Phi_\alpha(x)\Lambda_\beta(x)}{x^\beta} \right\rfloor = \left\lfloor \frac{\lfloor\Phi_\alpha(x)\rfloor \lfloor\Lambda_\beta(x)\rfloor}{x^\beta} \right\rfloor \tag{5}$$

$$= \left\lfloor \frac{\left\lfloor\Phi_\alpha(x) + \frac{\varphi_{p-1}(x)}{x^p}\right\rfloor \left\lfloor\Lambda_\beta(x) + \frac{\lambda_{p-1}(x)}{N_p(x)}\right\rfloor}{x^\beta} \right\rfloor \tag{6}$$

$$= \left\lfloor \frac{\left\lfloor\frac{U(x)}{x^p}\right\rfloor \left\lfloor\frac{x^{p+\beta}}{N(x)}\right\rfloor}{x^\beta} \right\rfloor = \hat{Q}(x) \tag{7}$$

□

There is no need to chose $\beta > \alpha$ because this would require more computations than when choosing $\beta = \alpha$ as $R(x)$ will then be larger.

The previous quotient computation is thus valid for $\mathbb{F}_q[x]$.

In the next section, we will use our method in $\mathbb{F}_2[x]$ which will show improvements of its software (and hardware) implementations. $Q(x)$ is then a binary polynomial of degree α requiring $\alpha + 1$ bits for its binary representation.

3 Modular Multiplication in $\mathbb{F}_2[x]$

In $\mathbb{F}_2[x]$, the coefficients n_i of the polynomial $N(x) = n_p x^p + n_{p-1}x^{p-1} + \ldots + n_1 x + n_0$ are either '0' or '1'. This gives a *binary representation* of the polynomials

in $\mathbb{F}_2[x]$, the upper bits of the representation being the upper coefficients of the polynomial. For example, the polynomial $x^5 + x^3 + 1$ can be represented as the binary number '101001'.

The modular multiplication in $\mathbb{F}_2[x]$ is one of the most important operation in elliptic curves cryptography over $\mathsf{GF}(2^p)$. We will show that one can obtain, with the method described in the previous section, similar performances to the one using Montgomery's modular multiplication in $\mathsf{GF}(2^p)$ [1].

From now on, we will suppose that the polynomial modular multiplication is implemented on a t-bit architecture (e.g. 32-bit). The modular multiplication $A(x)B(x) \bmod N(x)$ can be written as a sum of modular products of words of the first operand by the second operand:

$$\sum_{i=0}^{p_A-1} A_i(x)B(x)x^{it} \bmod N(x) = U(x) \bmod N(x) \tag{8}$$

where, $A_i(x)$ is a polynomial of degree $t-1$ such that $A(x) = \sum_{i=0}^{p_A-1} A_i(x)x^{it}$ and where $p_A = \left\lceil \frac{p_a}{t} \right\rceil$ with p_a, degree of $A(x)$.

Let's first recall some characteristics on the polynomial computations in $\mathbb{F}_2[x]$:

- The product of a polynomial of degree $t-1$ (which can be represented as a t-bit binary vector) by a polynomial of degree $n-1$ is a polynomial of degree $n+t-2$ represented as a $(n+t-1)$-bit vector. Over integers however, the result of a product of a t-bit by an n-bit integer is a $(n+t)$-bit number.
- The result of a polynomial addition of two polynomials of degree p is a polynomial of degree p (same number of bits in its binary representation). Over integers, the result may have one more bit in its binary representation because of carry propagations.
- The "modulo" operation (remainder of the division between two polynomials) gives a polynomial which is one degree smaller than the modulus. This means that the binary vector representing the remainder has always one bit less than the one of the modulus. Over integers, the remainder is smaller than the modulus but can still have the same number of bits in its binary representation.

Given equation 8, we can now evaluate how the size of quotient $Q(x)$ (equation 1) changes. To reduce the required memory and minimize accesses to it, equation 8 can be computed by interleaving the multiplication (from the highest index of A_i to the smallest one) with the reduction (by $N(x)$) as shown in figure 1.

Using the above characteristics of computations in $\mathbb{F}_2[x]$, the temporary polynomial $U(x)$ is always, at every stage as shown in figure 1, at a maximum degree of $t + p_N - 1$. This means that when computing the quotient $Q(x)$ as in equation 1, α has to be replaced by $t-1$. The corresponding $T(x)$ in equation 1 will then be of degree $t-1$ and $R(x)$ in equation 1 of degree $t-1$. A binary

```
1 :  U(x) = 0
2 :  for i = p_A − 1  down to 0
3 :      U(x) = U(x)x^t ⊕ A_i(x)B(x)
4 :      Q(x) = ⌊U(x)/N(x)⌋
5 :      U(x) = U(x) ⊕ Q(x)N(x)        [U(x) = U(x) mod N(x)]
6 :  end for i
```

Fig. 1. Modular multiplication in $\mathbb{F}_2[x]$.

vector representation in t-bit will thus perfectly match the computations on a t-bit architecture (CPU).

This also means that our method would require "standard" computations as in Montgomery's method described in [1]. In Montgomery's multiplication, a t-bit polynomial multiplication (with polynomials of degree $t-1$) and a division by x^t is required. In the present case, a t-bit polynomial multiplication and a division by x^{t-1} is needed. The remaining part of the computations can be the same in both methods but the way the computations are made is different (we start here from the most significant word A_{p_A-1} instead of A_0 in the Montgomery's method).

```
1 :  U(x) = A_{p_A−1}(x)B(x)
2 :  for i = p_A − 2  down to 0
3 :      Q(x) = ⌊(T(x)R(x))/x^{t−1}⌋
4 :      for j = 0  to  p_N − 1
5 :          U(x) = (U(x) ⊕ Q(x)N_j(x))x^{t(j+1)} ⊕ A_i(x)B_j(x)x^{tj}
6 :      end for j
7 :  end for i
8 :  Q = ⌊U(x)/N(x)⌋
9 :  U(x) = U(x) ⊕ Q(x)N(x)
```

Fig. 2. Interleaved modular multiplication in $\mathbb{F}_2[x]$.

It is possible to further reduce the memory accesses on $U(x)$. This is very important since memory accesses are often an important limiting factor in terms of execution time, namely in smart cards. To do so, we take the first computation $A_{p_A-1}(x)B(x)$ out of the i loop as shown in figure 2 allowing the i loop to start with the quotient computation ($Q(x)$) and the two computations of $U(x)$ in line (3) and (5) as shown in figure 1 to be merged into line (5) as shown in figure 2. Such a computation requires a final reduction outside the i loop (lines (8) and (9) in figure 2). The only disadvantage of interleaving the multiplication and the reduction phase using a unique j loop is that the numbers of $N_{j(x)}$ and $B_j(x)$ are identical ($p_B = p_N$), meaning that if, for example, the degree of $B(x)$ is smaller than the one of $N(x)$ it should be padded left with zeroes when storing it in $B[j]$'s. This does nevertheless not influence the speed of practical implementations since $B(x)$ is normally considered with an identical size to $N(x)$.

3.1 Software Implementations on a t-Bit Processor Architecture

For the sake of clearness, when comparing with existing implementations, we will suppose that $p = p_A = p_N$. The detailed implementation of the modular multiplication is shown in figure 4.

In this figure, a "$(\mathsf{Hi}, \mathsf{Lo})$" is a $2t$-bit register (such a register is common on RISC architectures providing a $t \times t$-bit multiplication with a $2t$-bit result) which is the concatenation of registers Hi and Lo (both t-bit registers), where Hi is the upper part (most significant bits) of that register and Lo, the lower part. The expression "$(\mathsf{Hi}, \mathsf{Lo}) \gg t$" means that this virtual register is shifted left of t-bit. In other words, the result is $\mathsf{Hi} = 0$ and $\mathsf{Lo} = \mathsf{Hi}$, the old value of Lo being discarded.

In figure 4, a '$\oplus$' represents a bitwise `XOR` operation and a '$\otimes$' means a polynomial multiplication in $\mathbb{F}_2[x]$ where the polynomials have at most a degree of $t-1$. In other words, a computation like $(\mathsf{Hi}, \mathsf{Lo}) \oplus = A \otimes B$ is simply a multiply and accumulate calculation, just like the one present on most RISC processors and DSP's, but with the internal carries in multiplications and additions being disabled. An algorithmic representation of this computation is shown in figure 3. Such a calculation is already implemented as an instruction in some high-end smart card microprocessors to improve elliptic curve computations in $\mathsf{GF}(2^p)$.

```
1 :  for i = 0  to  t − 1
2 :      (Hi, Lo) = (Hi, Lo) ⊕ ((A · ((B ≫ i) AND 1)) ≪ i)
                                      └── ith bit of B ──┘
3 :  end for i
```

Fig. 3. Simple program simulating the computation of $(\mathsf{Hi}, \mathsf{Lo}) \oplus = A \otimes B$.

Ideally, U_{sup} in lines (0) and (0) of figure 4, can be replaced by Lo, only if the Most Significant Bit (MSB) of $N[p-1]$ corresponds to the upper most significant coefficient of $N(x)$. Otherwise $U_{sup} = (Lo \ll k) \oplus (Rs \gg (t-k))$, where k is the shift value that would be needed to align the most significant coefficient of $N(x)$ stored in $N[p-1]$ with the MSB of $N[p-1]$.

Figure 4 also shows the number of multiply and accumulate instructions without carries (column '$\#\oplus$') and the number of memory accesses. In comparison with the paper of Koç and Acar [1] we have exactly the same number of multiplications without carries but without the additional `XOR` operations. To be correct, most of the `XOR` are included in our multiply and accumulate operation.

Improved implementation. The pseudo-code in figure 4 can still be compacted by computing $A_i B$ interleaved with QN in the reverse order as shown in figure 5 (computing first $A_i B[p-1]$ and $QN[p-1]$). This was made possible since there is no carry propagation when working in $\mathbb{F}_2[x]$ (compared with the computations over integers). Aligning the upper coefficient of $N(x)$ with the upper

		$\#\otimes$	#LOAD	#STORE
1 :	$\mathsf{Hi} = 0$			
2 :	$\mathsf{Lo} = 0$			
3 :	$A_{p-1} = A[p-1]$		1	
4 :	**for** $j = 0$ **to** $p-1$			
5 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = A_{p-1} \otimes B[j]$	p	p	
6 :	$Rs = \mathsf{Lo}; U[j] = Rs$			p
7 :	$(\mathsf{Hi}, \mathsf{Lo}) \gg t$			
8 :	**end for** j			
9 :	**for** $i = p-2$ **down to** 0			
10 :	$Q = (U_{sup} \otimes R) \gg (t-1)$	$p-1$		
11 :	$A_i = A[i]$		$p-1$	
12 :	$\mathsf{Hi} = U[0]$		$p-1$	
13 :	$\mathsf{Lo} = 0$			
14 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = A_i \otimes B[0]$	$p-1$	$p-1$	
15 :	$U[0] = \mathsf{Lo}$			$p-1$
16 :	$(\mathsf{Hi}, \mathsf{Lo}) \gg t$			
17 :	**for** $j = 1$ **to** $p-1$			
18 :	$\mathsf{Hi} = U[j]$		$(p-1)^2$	
19 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = A_i \otimes B[j]$	$(p-1)^2$	$(p-1)^2$	
20 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = Q \otimes N[j-1]$	$(p-1)^2$	$(p-1)^2$	
21 :	$Rs = \mathsf{Lo}; U[j] = Rs$			$(p-1)^2$
22 :	$(\mathsf{Hi}, \mathsf{Lo}) \gg t$			
23 :	**end for** j			
24 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = Q \otimes N[p-1]$	$p-1$	$p-1$	
25 :	**end for** i			
26 :	$Q = (U_{sup} \otimes R) \gg (t-1)$	1		
27 :	$\mathsf{Lo} = U[0]$		1	
28 :	**for** $j = 0$ **to** $p-2$			
29 :	$\mathsf{Hi} = U[j+1]$		$p-1$	
30 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = Q \otimes N[j]$	$p-1$	$p-1$	
31 :	$U[j] = \mathsf{Lo}$			$p-1$
32 :	$(\mathsf{Hi}, \mathsf{Lo}) \gg t$			
33 :	**end for** j			
34 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = Q \otimes N[p-1]$	1	1	
35 :	$U[p-1] = \mathsf{Lo}$			1
	TOTAL	$2p^2 + p$	$3p^2 + p$	$p^2 + p$

Fig. 4. Interleaved modular multiplication in $\mathbb{F}_2[x]$.

bit of $N[p-1]$, when storing $N(x)$ in the $N[j]$'s (before staring the computations), also simplifies the computations as shown in figure 5. This only requires one additional adaptation (right final shift) to the final result if the modulus is

constant during a whole set of modular multiplications. This is exactly the case with most cryptographic algorithms (e.g. ECDSA over $\mathsf{GF}(2^p)$ [4]).

		$\#\otimes$	#LOAD	#STORE
1 :	**for** $j = 0$ **to** $p - 1$			
2 :	$U[j] = 0$			p
3 :	**for** $i = p - 1$ **down to** 0			
4 :	$\mathsf{Hi} = U[p - 1]$		p	
5 :	$\mathsf{Lo} = U[p - 2]$		p	
6 :	$A_i = A[i]$		p	
7 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = A_i \otimes B[p - 1]$	p	p	
8 :	$Q = ((\mathsf{Hi}, \mathsf{Lo})_{sup} \otimes R) \gg (t - 1)$	p		
9 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = Q \otimes N[p - 1]$	p	p	
10 :	**for** $j = p - 2$ **down to** 1			
11 :	$(\mathsf{Hi}, \mathsf{Lo}) \ll t$			
12 :	$\mathsf{Lo} = U[j - 1]$		$p(p - 2)$	
13 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = A_i \otimes B[j]$	$p(p - 2)$	$p(p - 2)$	
14 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = Q \otimes N[j]$	$p(p - 2)$	$p(p - 2)$	
15 :	$U[j + 1] = \mathsf{Hi}$			$p(p - 2)$
16 :	**end for** j			
17 :	$(\mathsf{Hi}, \mathsf{Lo}) \ll t$			
18 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = A_i \otimes B[0]$	p	p	
19 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = Q \otimes N[0]$	p	p	
20 :	$U[1] = \mathsf{Hi}$			p
21 :	$U[0] = \mathsf{Lo}$			p
22 :	**end for** i			
	TOTAL	$2p^2 + p$	$3p^2 + p$	$p^2 + p$

Fig. 5. Interleaved modular multiplication with internal loop starting in the reverse order.

In this case (figure 5), U_{sup} is simply $((U[p-1], U[p-2]) \oplus A[i] \otimes B[p-1]) \gg (t-1)$. Indeed, there is no influence of $A[i] \otimes B[p-2]$ on the required upper part of $U(x)$ as this only influences the $(t-1)$ first bits of $(U[p-1], U[p-2]) \oplus A[i] \otimes B[p-1])$ and there is no carry propagation. Another consequence and advantage of such a computation is that there is no more need to compute $A_{p-1}(x)B(x)$ in advance. No additional final reduction by $Q(x)N(x)$ is necessary such that the lines (1) to (0) and (0) to (0) in figure 4 are no more necessary in figure 5.

As shown in figure 5, the global number of operations is identical to the one in figure 4, but the code size is smaller. Except for the last reason, the choice between the two implementations will only depend on the (processor's) architecture.

3.2 Comparison with Montgomery's Modular Multiplication

In figure 6, Montgomery's modular multiplication is implemented in the same way as our method. The main difference in our implementation compared to Koç and Acar's one [1] is the merging between the multiplication and the reduction phase of the algorithm to reduce the memory accesses. The number of memory accesses is smaller in our case as compared to the one required by the Montgomery's method described by Koç and Acar. Their method needs $(6s^2 - s)$ load and $(3s^2 + 2s + 1)$ store operations as given in table 2 of their paper.

As shown in figures 4, 5 and 6, our method is similar to Montgomery's one in terms of the number of multiply and accumulate without carries and the number of memory accesses.

		#⊗	#LOAD	#STORE
1 :	**for** $j = 0$ **to** $p - 1$			
2 :	$U[j] = 0$			p
3 :	**end for** j			
4 :	**for** $i = 0$ **to** $p - 1$			
5 :	$A_i = A[i]$		p	
6 :	$\mathsf{Lo} = U[0]$		p	
7 :	$\mathsf{Hi} = U[1]$		p	
8 :	$(\mathsf{Rt}, \mathsf{Lo}) \oplus = A_i \otimes B[0]$	p	p	
9 :	$(\mathsf{Hi}, \mathsf{Q}) = \mathsf{Lo} \otimes N_0^{'}$	p		
10 :	$(\mathsf{Rt}, \mathsf{Lo}) \oplus = Q \otimes N[0]$	p	p	
11 :	$(\mathsf{Hi}, \mathsf{Lo}) \gg t$			
12 :	**for** $j = 1$ **to** $p - 2$			
13 :	$\mathsf{Hi} = U[j + 1]$		$p(p-2)$	
14 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = A_i \otimes B[j]$	$p(p-2)$	$p(p-2)$	
15 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = Q \otimes N[j]$	$p(p-2)$	$p(p-2)$	
16 :	$U[j - 1] = \mathsf{Lo}$			$p(p-2)$
17 :	$(\mathsf{Hi}, \mathsf{Lo}) \gg t$			
18 :	**end for** j			
19 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = A_i \otimes B[p - 1]$	p	p	
20 :	$(\mathsf{Hi}, \mathsf{Lo}) \oplus = Q \otimes N[p - 1]$	p	p	
21 :	$U[p - 2] = \mathsf{Lo}$			p
22 :	$U[p - 1] = \mathsf{Hi}$			p
23 :	**end for** i			
	TOTAL	$2p^2 + p$	$3p^2 + p$	$p^2 + p$

Fig. 6. Interleaved Montgomery's modular multiplication in $\mathbb{F}_2[x]$.

However, a possible inconvenience of our method (for a software implementation on a general purpose processor), as compared to Montgomery's one, would

be the slower "extraction" of U_{sup} from the intermediate values of $U(x)$ as well as the right shift by $(t-1)$ when computing Q. This disadvantage can be simply taken into account, with a very minimal cost, in a hardware implementation.

The main disadvantage of Montgomery's method is not visible in figure 6. Indeed, our method exactly computes $A(x)B(x) \bmod N(x)$ which is not the case for Montgomery's one which computes $A(x)B(x)x^{-p} \bmod N(x)$ [1]. This last computation is mostly not desired. This is why Montgomery's method often requires substantial additional code to deal with that computation (e.g replace $A(x)$ by $A(x)x^p \bmod N(x)$ and $B(x)$ by $B(x)x^p \bmod N(x)$ before the computations and then finish the computations by multiplying the final result by '1' using Montgomery's method). These computations penalize Montgomery's method in terms of code size (it can be critical in smart card's context) and may complicate the use of the Montgomery's method outside the scope of modular exponentiations computations (e.g. for a single modular multiplication).

3.3 Speed Comparisons on a Real Implementation

Table 1 shows the results, in terms of clock cycles, obtained by both Montgomery's multiplication (figure 6) and the two versions of our method described in figures 4 and 5.

	256-bit multiplication	512-bit multiplication
Algorithm in fig. 4	910	3230
Compact Algorithm in fig. 5	812	3028
Montgomery's method (fig. 6)	756	2916

Table 1. Algorithms' speed in clock cycles on a Montgomery's optimized processor.

Measurements were done on a 32-bit processor's (usable in smart cards) simulator using the modified multiply accumulate instruction (without internal carries) as described in section 3.1. This processor was designed to speed up Montgomery's multiplications. This explains why, in table 1, the Montgomery's method has still an advantage. As explained before, this is only due to the small additional computations required for computing $Q(x)$ in our implementation. Indeed, 7 additional clock cycles for each $Q(x)$ computation (equivalent to line (8) in figure 5) are required as compared to what is done in the Montgomery's implementation (line (9) in figure 6). However, similar results for both implementations can be obtained by very slightly modifying a few processor's instructions.

The comparison made here only involves the "core's" modular multiplication itself. As explained in section 3.2, the fact that Montgomery's method computes $A(x)B(x)x^{-p} \bmod N(x)$ in place of the exact value $(A(x)B(x) \bmod N(x))$ for our method, can also deeply influence the choice between one algorithm and the other.

4 Conclusions

We have first extended the generalized Barrett's modular reduction to $\mathbb{F}_q[x]$. We then described an efficient way to implement fast "left to right" modular multiplication in $\mathbb{F}_2[x]$ which is at least as efficient as the best known methods, namely Montgomery's multiplication [1]. Furthermore, our method has the advantage of computing the modular multiplication without the inconvenience of a normalization as needed in the Montgomery's one. This makes this method particularly attractive for smart cards and hardware implementations. A way of reducing the memory accesses in both methods has been described. Both methods can be efficiently implemented by having a multiply and accumulate instruction without internal carries to perform fast competitive software implementations of elliptic curve crypto-systems in $\mathsf{GF}(2^p)$.

Acknowledgments. The author would like to thank Jacques Fournier, Marc Joye and the anonymous referees for their useful comments.

References

1. Koç, C., Acar, T.: Montgomery multiplication in GF(2^k). In Publishers, K.A., ed.: Designs, Codes and Cryptography. Volume 14., Boston (1998) 57–69
2. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press (1997)
3. Montgomery, P.: Modular multiplication without trial division. Mathematics of Computation **44** (1985) 519–521
4. IEEE: Std 1363-2000. IEEE standard specifications for public-key cryptography, New York, USA (2000) Informations available at `http://grouper.ieee.org/groups/1363/`.
5. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Odlyzko, A., ed.: Advances in Cryptology - CRYPTO '86, Santa Barbara, California. Volume 263 of Lecture Notes in Computer Science., Springer-Verlag (1987) 311–323
6. Quisquater, J.J.: Encoding system according to the so-called RSA method, by means of a microcontroller and arrangement implementing this system. U.S. Patent # 5,166,978 (1992)
7. Dhem, J.F., Quisquater, J.J.: Recent results on modular multiplications for smart cards. In Quisquater, J.J., Schneier, B., eds.: Proc. CARDIS'98, Smart Card Research and Applications, Louvain-la-Neuve, Belgium. Volume 1820 of Lecture Notes in Computer Science., Springer-Verlag (2000) 336–352
8. Cohen, H.: A Course in Computational Algebraic Number Theory. 2nd edn. Graduate Texts in Mathematics. Springer (1995)
9. De Win, E., Bosselaers, A., Vandenberghe, S., De Gersem, P., Vandewalle, J.: A fast software implementation for arithmetic operations in GF(2^n). In Kim, K., Matsumoto, T., eds.: Advances in Cryptology - ASIACRYPT '96, Kyongju, Korea. Volume 1163 of Lecture Notes in Computer Science., Springer (1996) 65–76