Invited Paper

CRYPTOGRAPHIC APPLICATIONS IN FPGA

J.-P. DESCHAMPS

Electronic Engineering Department, University Rovira i Virgili, Tarragona, Spain
jeanpierre.deschamps@urv.net

***Abstract*— This paper describes circuits for executing the most complex operations of public-key cryptography and gives estimations of their execution time within field programmable devices. The following operations are considered: mod *n* exponentiation, mod *p* division, mod *f*(*x*) multiplication of polynomials, mod *f*(*x*) division of polynomials and point multiplication over an elliptic curve.**

***Keywords*— public-key cryptography, finite fields, arithmetic, programmable devices, FPGA**

I. INTRODUCTION

The designer of systems including cryptographic algorithms – ciphering / deciphering, digital signature, authentication - is often faced with the following apparent contradiction: on the one hand, in many cases cryptographic algorithms are used within real time systems, so that their response time must be short; on the other hand, the security is related to the algorithm complexity. In order to make compatible those apparently contradictory characteristics, a possible solution is the use of specific hardware, that is, circuits specifically designed for executing those complex algorithms: they implement the particular computation primitives of the algorithms and take profit of their inherent parallelism. Among the technologies at hand for developing specific circuits are the field programmable devices, for example the Field Programmable Gate Arrays (FPGA). They constitute an attractive option for small production quantities as their non-recurrent engineering costs are much lower than those corresponding to Application Specific Integrated Circuits (ASIC). Furthermore, in order to reduce their size, and so the unit cost, an interesting possibility is to reconfigure them at run time so that the same programmable device can execute different predefined functions.

This paper describes circuits for executing the most complex operations of public-key cryptography and gives estimations of their execution time within field programmable devices. It is organized in the following way: section II briefly describes the main public-key cryptographic algorithms and deduces a list of complex computation primitives that should be implemented in hardware. Section III to VII propose generic algorithms[1] and circuits for executing the mod *n* exponentiation, the mod *p* division, the mod $f(x)$ multiplication of polynomials, the mod $f(x)$ division of polynomials and the point multiplication over an elliptic curve, respectively. The adjective "generic" alludes to the fact that particular characteristics of the underlying algebraic structure, for instance special values of n, p or $f(x)$, are not taken into account (except the case $p = 2$). Actually, a lot of improvements can be obtained if particular values of p and $f(x)$ are chosen, but their description falls beyond the scope of this paper.

II. MAIN ARITHMETIC OPERATIONS

The most time-consuming operations correspond to public-key cryptography, that is, encryption / decryption schemes using different keys for ciphering (public key) and deciphering (private key). Among the most used are the RSA and the Discrete Logarithm systems.

In the first case (RSA, Adleman *et al.*, 1978), two primes p and q are chosen. The public key is a pair (n,e) of naturals where $n = p.q$, e belongs to the interval $0 < e < (p-1)(q-1)$ and e is relatively prime with $(p-1)(q-1)$. The private key is $d = e^{-1} \bmod (p-1)(q-1)$. It can be shown that $x^{e.d} \equiv x \bmod n$, for any natural x. The encryption / decryption algorithm is the following: giving a message *mes* represented under the form of a natural belonging to the interval $0 < mes < n$, compute the ciphered text $c = mes^e \bmod n$. In order to decrypt c, compute $c^d \bmod n$. Observe that knowing the public key (n,e), the computation of the private key amounts to decompose n under the form $n = p.q$ and then calculate $d = e^{-1} \bmod (p-1)(q-1)$. Nowadays, the factorization problem is intractable for key sizes greater 1024 bits.

In the second case (Discrete Logarithm), a finite group $(G,*,1)$ is defined and some element g of G is chosen. Let n be the order of g. Thus, the set $\{1, g, g^2,..., g^{n-1}\}$ is a cyclic subgroup of G. The private key is a natural x belonging to the interval $0 < x < n$, and the public key is the element y of the cyclic subgroup defined by $y = g^x$. The message *mes* must be represented under the form of an element of G. The encryption algorithm is the following: randomly choose a natural k belonging to $0 < k < n$, compute $c_1 = g^k$ and $c_2 = mes * y^k$. The ciphered text is made up of c_1 and c_2. In order to decrypt the message, compute $c_2 * (c_1^x)^{-1}$. Observe that knowing the public key y, the computation of the private key x amounts to calculate $\log_g y$, presumably a very hard problem.

In the basic version of the Discrete Logarithm scheme (ElGamal, 1985), G is the set of natural $\{1,2,..., p-1\}$, where p is a prime, so that all operations are per-

[1] Most algorithms are described in Ada and complete source programs are available at http://www.ii.uam.es/~gsutter/arithmetic.

formed modulo p. Nevertheless, other groups can be used. Consider for example an elliptic curve E, over the binary extension field $GF(2^m)$, defined as being the set of elements (x,y) of $GF(2^m)$x$GF(2^m)$ such that $y^2 + xy = x^3 + ax + b$, where a and b are elements of $GF(2^m)$. It can be demonstrated that the set of points of E, plus the so-called point at infinity ∞, is a group $(E, +, \infty)$ whose basic operation (under additive notation, with neutral element ∞) is defined as follows:

$$(x , y) + \infty = \infty + (x , y) = (x , y); \quad (1)$$

$$(x_1 , y_1) + (x_1 , x_{1+}y_1) = \infty;$$
$$\text{in particular, } (0 , y_1) + (0 , y_1) = \infty; \quad (2)$$

$$\text{if } x_1 \neq 0, \text{ then } (x_1 , y_1) + (x_1 , y_1) = (x_3 , y_3) \text{ where}$$
$$x_3 = \lambda^2 + \lambda + a, \; y_3 = x_1^2 + \lambda x_3 + y_1,$$
$$\lambda = x_1 + y_1/x_1; \quad (3)$$

$$\text{if } (x_2 , y_2) \neq (x_1 , y_1) \text{ and } (x_2 , y_2) \neq (x_1 , x_{1+}y_1), \text{ then}$$
$$(x_1 , y_1) + (x_2 , y_2) = (x_3 , y_3) \text{ where}$$
$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a, \; y_3 = \lambda(x_1 + x_3) + x_3 + y_1,$$
$$\lambda = (y_1 + y_2)/(x_1 + x_2). \quad (4)$$

A Discrete Logarithm scheme can be defined by choosing a point P of E whose order is equal to n so that the set $\{\infty, P, 2P, \ldots , (n\text{-}1)P\}$ is a cyclic subgroup of E. The private key is a natural d belonging to the interval $0 < d < n$, and the public key is the element Q of the cyclic subgroup defined by $Q = dP$. A simple encryption / decryption algorithm would be the following: giving a message *mes* represented by a point M of E, randomly choose a natural number k belonging to $0 < k < n$, compute $C_1 = kP$ and $C_2 = M + kQ$. The ciphered text is made up of C_1 and C_2. In order to decrypt the message, compute $C_2 - dC_1$. Actually, other encryption / decryption schemes are used, avoiding among others the embedding of *mes* within E. Nevertheless, the operations to be performed are similar. Observe that knowing the public key Q, the computation of the private key amounts to looking for a natural d such that $dP = Q$, presumably a very hard problem. Nowadays, this problem is intractable for key sizes greater than 160 bits.

As a matter of fact, elliptic curves can be defined over any field. The group operation definition depends on the particular field, but they always amount to combinations of basic arithmetic operations (add, subtract, multiply, square and divide) over the chosen field.

Among all the mentioned operations, the most time-consuming are the following:

$z = y^x \bmod n$, where x and y are naturals included between 0 and n (RSA, ElGamal),

$z = x.y^{-1} \bmod p$, where p is prime, and x and y are naturals included between 0 and p (ElGamal, elliptic curve over $GF(p)$),

$z(x) = g(x).h(x) \bmod f(x)$, where f is a polynomial of degree m over $GF(p)$, and g and h are polynomials of degree less than m over $GF(p)$ (elliptic curve over $GF(p^m)$),

$z(x) = g(x).h^{-1}(x) \bmod f(x)$, where f is an irreducible polynomial of degree m over $GF(p)$, and g and h are polynomials of degree less than m over $GF(p)$ (elliptic curve over $GF(p^m)$).

III. EXPONENTIATION MOD *N*

Let x be represented in base 2, that is, $x = x_{m-1}.2^{m-1} + x_{m-2}.2^{m-2} + \ldots + x_1.2 + x_0$, with $m \geq log_2 n$. Then $z = y^x \bmod n$ can be computed according to the following computation scheme:

$$z = ((\ldots((1^2.y^{x_{m-1}})^2.y^{x_{m-2}})^2\ldots)^2.y^{x_1})^2.y^{x_0} \bmod n,$$

to which corresponds the following algorithm:

Algorithm 1 – base 2 mod *n* exponentiation (complete program available)

```
e := 1;
for i in 1 .. m loop
 e := (e*e) mod n ;
 if x(m-i) = 1 then e := (e*y) mod n;
end if;
end loop;
z := e;
```

If n is odd, so that 2 has an inverse mod n, a more effective algorithm uses the Montgomery reduction concept (Montgomery, 1985). Consider two naturals a and b belonging to $\{0, 1, \ldots , n\text{-}1\}$ and define the functions T and MP (Mongtgomery product) as follows: $T(a) = a.2^m \bmod n$, $MP(a, b) = a.b.2^{-m} \bmod n$. Obviously, $T(a.b) = MP(T(a), T(b))$ and $T^{-1}(a) = a.2^{-m} \bmod n = MP(1, a)$. Then, in algorithm 1 substitute 1 by $T(1) = 2^m \bmod n$, y by $T(y) = y.2^m \bmod n = MP(y, 2^{2.m} \bmod n)$, the product by the Montgomery product, and replace the last step by $z = T^{-1}(e) = MP(1, e)$.

Algorithm 2 – mod *m* exponentiation, Montgomery algorithm (complete program available)
--the constants $exp\_m = 2^m \bmod n$ and $exp\_2m = 2^{2.m} \bmod n$ are previously computed

```
e := exp_m;
ty := mp(y, exp_2m);
for i in 1 .. m loop
 e := mp(e, e);
 if x(m-i) = 1 then e := mp(e, ty);
end if;
end loop;
z := mp(e, 1);
```

The Montgomery product is computed as follows (Montgomery 1985; chapter 8 of Deschamps *et al*., 2006):

Algorithm 3 – base-2 Montgomery product (complete program available)

```
product := 0;
for i in 0 .. m-1 loop
 a := product + x(i)*y;
 product := (a + a(0)*n)/2;
```

```
end loop;
if product >= n then z := product-n;
else z := product; end if;
```

The efficiency of the algorithm comes from the fact that it does not include any mod n reduction. Observe that if *product* is smaller than $2.n$, then $a < 3.n$, and the new value of *product* is smaller than $4.n/2 = 2.n$. Thus, as initially *product* = 0, all along the algorithm execution *product* is an $(m+1)$-bit number and a an $(m+2)$-bit number. The part of the data-path corresponding to the execution of one step of the Montgomery product is shown in figure 1. It is made up of an $(m+1)$-bit conditional adder followed by an $(m+2)$-bit conditional adder, so that its computation time is equal to $m+3$ full-adder delays. The number of iteration steps of algorithm 3 is m, so that the computation time of one Montgomery product is equal to $m.(m+3)$ full-adder delays. The number of iteration steps of algorithm 2 is m and every step includes at most two Montgomery products. Thus the total computation time of y^x mod n is approximately equal to $2.m^3$ full-adder delays being m the size of the operands.

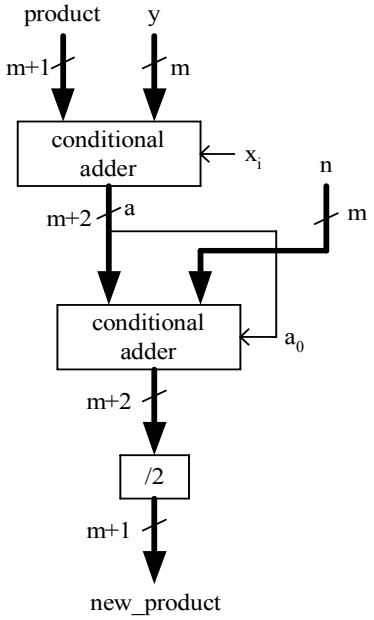

Figure 1 Montgomery product: basic step

With nowadays FPGAs including very fast carry-logic circuitry, delays smaller than $0.2ns$ per adder-bit are obtained (chapter 11 of Deschamps *et al*., 2006), so that the computation time is smaller than $0.4m^3$ *ns*. As an example, for $m = 1024$, the computation time is less than $0.5s$.

IV. DIVISION MOD *P*

There are two main types of division algorithms. The first type consists of extensions of algorithms for computing the greatest common divider (*gcd*) of two numbers, in particular the extended Euclidean algorithm (chapter 2 of Hankerson *et al*., 2004) and the binary algorithm (Brent and Kung, 1983). The second type includes those that are based on the Fermat's little theorem and use field multiplication as primitive operation. The latter ones are conceptually simple: given x in $GF(p)$, then $y.(x.y^{p-2}) \bmod p = x.y^{p-1} \bmod p = x$, so that $z = x.y^{p-2} \bmod p$. The former ones are based on the fact that the greatest common divider of y and p is equal to 1, so that any algorithm for computing the *gcd* of two naturals y and p based on the generation of sequences of naturals $a(0)$, $a(1)$, $a(2)$, ... , $b(0)$, $b(1)$, $b(2)$, ... , $c(0)$, $c(1)$, $c(2)$, ... , and $d(0)$, $d(1)$, $d(2)$, ... , such that $gcd(a(i), b(i)) = gcd(y, p)$, $a(i) < a(i\text{-}1)$, $a(i).x \equiv c(i).y \bmod p$ and $b(i).x \equiv d(i).y \bmod p$, eventually generates $a(n) = 0$ and thus $b(n) = 1$, $x \equiv d(n).y \bmod p$, that is, $z = d(n) \bmod p$. An example is the extended Euclidean algorithm. Another is the binary algorithm based on the following obvious properties:

if a is even and b is odd, then $gcd(a, b) = gcd(a/2, b)$,
if a is odd and b is even, then $gcd(a, b) = gcd(b/2, a)$,
if a is odd and b is odd and $a \geq b$, then $gcd(a, b) = gcd(a\text{-}b, b)$,
if a is odd and b is odd and $a < b$, then $gcd(a, b) = gcd(b\text{-}a, a)$;

furthermore, if $a.x \equiv c.y \bmod p$ and $b.x \equiv d.y \bmod p$, then

$(a/2).x \equiv (c.2^{-1} \bmod p).y \bmod p$,
$(b/2).x \equiv (d.2^{-1} \bmod p).y \bmod p$,
$(a\text{-}b).x \equiv (c\text{-}d).y \bmod p$,
$(b\text{-}a).x \equiv (d\text{-}c).y \bmod p$.

Initially define $a(0) = y$, $b(0) = p$, $c(0) = x$ and $d(0) = 0$, so that $a(0).x = c(0).y$ and $b(0).x \equiv d(0).y \bmod p$. In the next algorithm the function `divide_by_2(c, p)` returns $c.2^{-1} \bmod p$, that is, $c/2$ if c is even and $(c+p)/2$ if c is odd.

Algorithm 4 - mod *p* division, binary algorithm (complete program available)

```
a := y; b := p; c := x; d := 0;
while a > 0 loop
   while (a mod 2) = 0 loop
    a := a/2;
    c := divide_by_2(c, p);
   end loop;
   if a >= b then
    a := a-b; c := (c-d) mod p;
   else old_a := a; a := b-a;
    b := old_a; old_c := c;
    c := (d-c) mod p; d := old_c;
   end if;
end loop;
z := d;
```

A drawback of the preceding algorithm is the necessity of detecting whether a is smaller than b, or not. The plus-minus algorithm is a modified version of the binary algorithm avoiding the time-consuming comparison of a and b when large numbers are considered. Variables a

and b are allowed to be negative integers and, instead of comparing their actual values, logarithmic bounds are used, namely α and β, such that

$$-2^{\alpha} < a < 2^{\alpha}, -2^{\beta} < b < 2^{\beta}.$$

The algorithm is based on the following observation: if a and b are odds, then both $a+b$ and $a-b$ are even, and their sum $(a+b) + (a-b) = 2.a$ cannot be a multiple of 4 (a is odd), so that either $(a+b)$ mod 4 = 0 and $(a-b)$ mod 4 = 2, or $(a-b)$ mod 4 = 0 and $(a+b)$ mod 4 = 2. Thus, either $a+b$ or $a-b$ is divisible by 4. The following obvious properties are used:

if a is divisible by 4 and b is odd, then $gcd(a, b) = gcd(a/4, b)$,
if a is even and b is odd, then $gcd(a, b) = gcd(a/2, b)$,
if a is odd, b is odd and $a+b$ is divisible by 4, then $gcd(a, b) = gcd((a+b)/4, b) = gcd((a+b)/4, a)$,
if a is odd, b is odd and $a-b$ is divisible by 4, then $gcd(a, b) = gcd((a-b)/4, b) = gcd((a-b)/4, a)$.

When a is odd, the new value of b (could be a or b) is chosen in function of α and β: if $\alpha \geq \beta$ choose b, and if $\alpha < \beta$, choose a. As regards c and d, they are updated accordingly: the new value of c is $c.4^{-1}$ mod p, $c.2^{-1}$ mod p, $(c+d).4^{-1}$ mod p or $(c-d).4^{-1}$ mod p, and the new value of d is d or c. Actually, all decisions can be taken in function of the difference $dif = \alpha - \beta$. In order to update the values of α and β, the value of min = minimum(α, β) must be known. In the following algorithm the functions `divide_by_2(c, p)` and `divide_by_4(c, p)` return integers equivalent to $c.2^{-1}$ mod p and $c.4^{-1}$ mod p, respectively: $c/2$ or $(c+p)/2$ in the first case; $c/4$, $(c+p)/4$, $(c+2.p)/4$ or $(c-p)/4$, in the second case (in function of c mod 4 and p mod 4). During the execution of the algorithm the values of a, b, c and d remain included between $-p$ and p. Thus, if p is a k-bit number, a, b, c and d can be represented as $(k+1)$-bit 2's complement integers. More details can be found in Deschamps and Sutter (2006).

Algorithm 5 - mod *p* division, plus-minus algorithm (complete program available)

```
a := y; b := p; c := x; d := 0;
dif := 0; min := logp;
while min > 0 loop
   if a mod 4 = 0 then
      a := a/4;
      c := divide_by_4(c, p);
      if dif <= 0 then min := min-2;
      elsif dif = 1 then
       min := min-1;
      end if;
      dif := dif-2;
   elsif a mod 2 = 0 then
      a := a/2;
      c := divide_by_2(c, p);
      if dif <= 0 then
       min := min-1;
      end if;
      dif := dif-1;
   else
      old_a := a; old_c := c;
      if (a+b) mod 4 = 0 then
        a := (a+b)/4;
        c := divide_by_4(c+d, p);
        if dif  >= 0 then
          if dif = 0 then
           min := min-1;
          end if;
          dif := dif-1;
        else
          b:= old_a; d := old_c;
          dif := -dif-1;
        end if;
      else
        a := (a-b)/4;
        c := divide_by_4(c-d, p);
        if dif  >= 0 then
          if dif = 0 then
            min := min-1;
          end if;
          dif := dif-1;
        else
          b:= old_a; d := old_c;
          dif := -dif-1;
        end if;
      end if;
   end if;
end loop;
if b < 0 and d < 0 then z := -d;
elsif b < 0 and d >= 0 then
   z := p - d;
elsif b > 0 and d < 0 then
   z := p + d;
else z := d;
end if;
```

At each step of algorithm 5, a is substituted by either $a/4$, $a/2$, $(a+b)/4$ or $(a-b)/4$, and c by either $c.4^{-1}$ mod p,$c.2^{-1}$ mod p, $(c+d).4^{-1}$ mod p or $(c-d).4^{-1}$ mod p. The most time-consuming operations are those corresponding to c. They can be executed by the circuit of figure 2. The control unit defines the operation (add or subtract) in function of the two less significant bits of a and b (according to algorithm 5), and the correction term (0, p, $2.p$ or $-p$) in function of the two less significant bits of c and d (according to the definition of functions `divide_by_2(c, p)` and `divide_by_4(c, p)`).

The maximum delay corresponds to the computation time of a $(k+2)$-bit adder-subtractor followed by a $(k+3)$-bit adder, that is, $k+4$ full-adder delays. An upper bound of the number of steps is $2.k$, so that the total computation time is approximately equal to $2.k^2$ full-adder delays, being k the size of p. Thus, if a delay smaller than

0.2*ns* per adder-bit is assumed, the total computation time is smaller than $0.4k^2$ *ns*. As an example, for $p = 2^{192} - 2^{64} - 1$, so that $m = 192$, the computation times should be less than 14.8μs. Actually, a computation time of 11.5μs has been reported (Deschamps and Sutter, 2006).

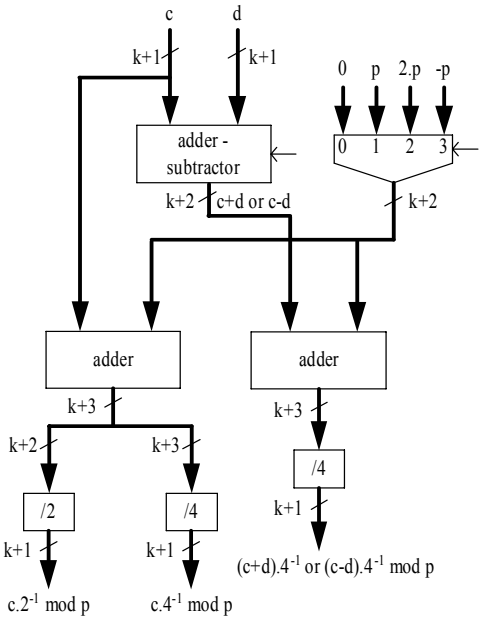

Figure 2 Plus-minus algorithm

V. MULTIPLICATION MODULO *F*(*x*)

Let $f(x)$ be a polynomial of degree m over $GF(p)$, and $a(x)$ and $b(x)$ polynomials of degree smaller than m over the same field. The multiplication of $a(x)$ by $b(x)$ can be performed according to the following computation scheme:

$a(x).b(x) \bmod f(x) = ((\ldots(((0.x + a(x).b_{m-1}).x + a(x).b_{m-2}).x) + \ldots + a(x).b_1).x + a(x).b_0) \bmod f(x)$.

In the following algorithm, the functions `multiply_by_x(a,f)`, `add(a,b)` and `product(a,coeff)` return $a(x).x \bmod f(x)$, $a(x) + b(x)$ and $a(x).coeff$, respectively.

Algorithm 6 – mod *f*(*x*) multiplication (complete program available)

```
z := 0;
for i in 1 .. m loop
   z := add(multiply_by_x(z,f),
   product(a,b(m-i)));
end loop;
```

Assuming that f is a monic polynomial ($f_m = 1$), the computation of $a(x).x \bmod f(x)$ can be performed as follows:

$a(x).x \bmod f(x) = a(x).x - a_{m-1}.f(x) = a_{m-2}.x^{m-1} + a_{m-3}.x^{m-2} + \ldots + a_0.x - a_{m-1}.(f_{m-1}.x^{m-1} + f_{m-2}.x^{m-2} + \ldots + f_0)$.

The corresponding circuit is shown in Fig. 3. The circuit corresponding to the iteration step of algorithm 6 is shown in figure 4.

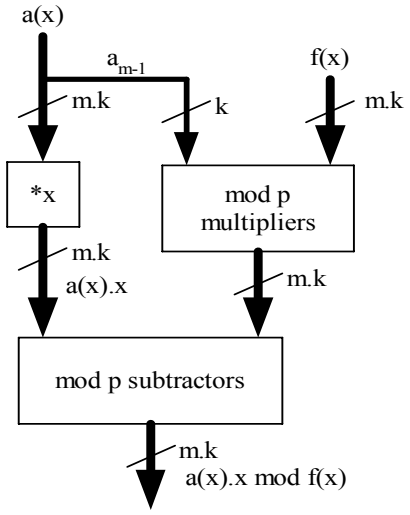

Figure 3 Computation of $a(x).x \bmod f(x)$

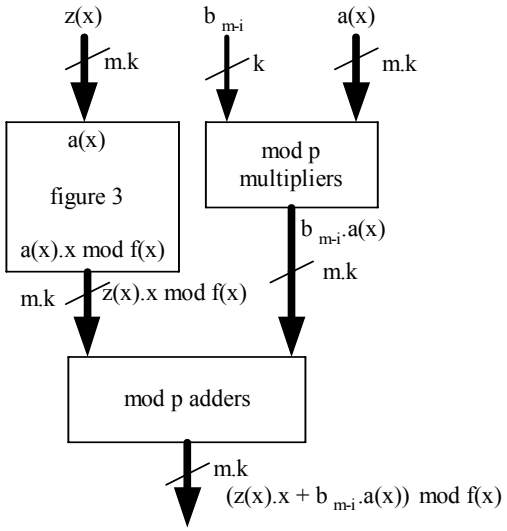

Figure 4 Multiplication mod $f(x)$

The delay of the circuit of figure 3 is the sum of the computation times of a mod p multiplier and a mod p subtractor, so that the delay of the circuit of figure 4 is the sum $t_{mult}(p)+t_{sub}(p)+t_{add}(p)$ of the computation times of a mod p multiplier, a mod p subtractor, and a mod p adder, respectively, and the total execution time of algorithm 6 is equal to $m.(t_{mult}(p)+t_{sub}(p)+t_{add}(p))$. In the binary case ($p = 2$), the multiplication is an AND function, and both the subtraction and the addition are XOR functions. Thus, the total delay of the circuit of figure 4 is equal to one AND-gate delay plus two XOR-gate delays. Assuming that every two-input Boolean function is implemented within an FPGA's look-up table (LUT), an approximation of the total computation time is 3.m LUT-delays. With nowadays FPGAs, LUT-delays smaller than 2*ns* are obtained, so that the computation time is smaller than 6.m *ns*. As an example, for $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$, so that $m = 163$, the computation times should be less than 1 μs.

VI. DIVISION MODULO $F(x)$

As in the case of the mod p division, there are two main types of algorithms: reduction to products of polynomials and inversion over $GF(p)$, and extensions of algorithms for computing the *gcd* of two polynomials. The algorithms of the first type are based on the fact that, given a polynomial $h(x)$, then $h^r(x)$, where $r = (p^m-1)/(p-1)$, is a polynomial of degree zero (Koblitz, 1994), that is, an element of $GF(p)$, so that the computation of $z(x) = g(x).h^{-1}(x) \bmod f(x) = g(x).(h^r(x))^{-1}.h^{r-1}(x) \bmod f(x)$ amounts to an inversion over $GF(p)$ and products of polynomials. The second type includes the extended Euclidean algorithm for polynomials (chapter 2 of Hankerson *et al*., 2004) and the binary algorithm for polynomials (Deschamps and Sutter, 2005). The latter is similar to the binary and plus-minus algorithms over $GF(p)$: the concept of an integer being divisible by 2 (i.e. least significant bit equal to 0) is now replaced by the concept of a polynomial being divisible by x (i.e. degree-zero coefficient equal to 0). Four sequences of polynomials are generated: $a(0)$, $a(1)$, $a(2)$, ... , $b(0)$, $b(1)$, $b(2)$, ... , $c(0)$, $c(1)$, $c(2)$, ... , and $d(0)$, $d(1)$, $d(2)$, ... , such that $gcd(a(i), b(i)) = gcd(h, f)$, $degree(a(i)) < degree(a(i\text{-}1))$, $a(i).g \equiv c(i).h \bmod f$ and $b(i).g \equiv d(i).h \bmod f$. After a finite number of steps, say n, $degree(a(n)) = 0$. If $a(n) = 0$, then $degree(b(n)) = 0$ and $z = d(n).b_0(n)^{-1} \bmod f$, and if $a(n) \neq 0$, then $z = c(n).a_0(n)^{-1} \bmod f$. The following properties are used:

if a is divisible by x, then $gcd(a, b) = gcd(a/x, b)$,
if a is not divisible by x, then $gcd(a, b) = gcd((a - b.a_0.b_0^{-1})/x, b) = gcd((a - b.a_0.b_0^{-1})/x, a)$;

furthermore, if $a.g \equiv c.h \bmod f$ and $b.g \equiv d.h \bmod f$, then

$$(a/x).g \equiv c.x^{-1}.h \bmod f,$$
$$((a - b.a_0.b_0^{-1})/x).g \equiv (c - d.a_0.b_0^{-1}).x^{-1}.h \bmod f.$$

At each step, upper bounds α and β of the degree of a and b are calculated:

$$degree(a) \leq \alpha \text{ and } degree(b) \leq \beta.$$

Initially define $a = h(x)$, $b = f(x)$, $c = g(x)$ and $d = 0$. In the next algorithm, the function `shift_one(a)` returns a/x, the function `divide_by_x(c, f)` returns $c.x^{-1} \bmod f = (c - f.c_0.f_0^{-1})/x$, the function `subtract(a,b)` returns a-b, the function `invert(coeff)` returns $coeff^{-1} \bmod p$, and the function `product(a,coeff)` returns $a.coeff$. As in the case of the binary algorithm, all decisions can be taken in function of the difference $dif = \alpha - \beta$, and the value of $min = \text{minimum}(\alpha, \beta)$ must be known.

Algorithm 7 – mod $f(x)$ division, binary algorithm (complete program available)

```
a := h; b := f; c := g;
dif := -1; min := m-1;
for i in 0 .. m loop d(i) := 0;
end loop;
while min > 0 loop
 if a(0) = 0 then
  a := shift_one(a);
  c := divide_by_x(c, f);
  if dif <= 0 then min := min - 1;
  end if;
  dif := dif - 1;
 else
  old_a := a; old_c := c;
  a := shift_one(subtract(a,
  product(b, ((a(0)*invert(b(0)))
  mod p))));
  c := divide_by_x(subtract(c,
  product(d, ((old_a(0)*invert(b(0)))
  mod p)),f);
  if dif >= 0 then
   if dif = 0 then min := min-1;
   end if;
   dif := dif - 1;
  else
   dif := -dif - 1;
   b := old_a; d := old_c;
  end if;
 end if;
end loop;
if a(0) = 0 then
z := product(d, invert(b(0)));
else z := product(c, invert(a(0)));
end if;
```

At each step of algorithm 7, $a(x)$ is substituted by either $a(x)/x$ or $(a(x)–a_0.b_0^{-1}.b(x))/x$, and $c(x)$ by either $c(x).x^{-1} \bmod f(x)$ or $(c(x)–a_0.b_0^{-1}.d(x)).x^{-1} \bmod f(x)$. The most time-consuming operations are those corresponding to $c(x)$. A circuit for multiplying a polynomial $a(x)$ by $x^{-1} \bmod f(x)$, that is, for computing $(a – f.a_0.f_0^{-1})/x$, is shown in Fig. 5, and the complete circuit for computing the new value of $c(x)$ in Fig. 6.

The computation times of all blocks, but the mod p inverter, are proportional to k, while the computation time of the mod p inverter is approximately equal to $2.k^2$ full-adder delays (section IV). The number of executions of the main loop (`while min > 0 loop`) is smaller than two times the degree m of f. Thus, for great values of k, the computation time is approximately equal to $4.m.k^2$ full-adder delays, being k the size of p and m the degree of f. For small values of k, the conclusion is different. In particular, in the binary case ($p = 2$), the algorithm can be simplified: the computation of $a(x).x^{-1} \bmod f(x)$ amounts to substituting every coefficient a_i of $a(x)$ by $(a_{i+1} + a_0.f_{i+1}) \bmod 2$, for $i = 0, 1, \ldots, m\text{-}2$, by $(a_{i+1} + a_0.f_{i+1}) \bmod 2$, for $i = 0, 1, \ldots, m\text{-}2$, and a_{m-1} by $a_0.f_m$; the corresponding delay is the sum of an AND-gate delay and an XOR-gate delay; the mod p inversion is trivial ($b_0^{-1} = b_0$); the mod p multiplication is an AND function and the mod p subtraction an XOR function. Thus, the total delay of the circuit of figure 6 is equal to three AND-gate delays plus two XOR-gate

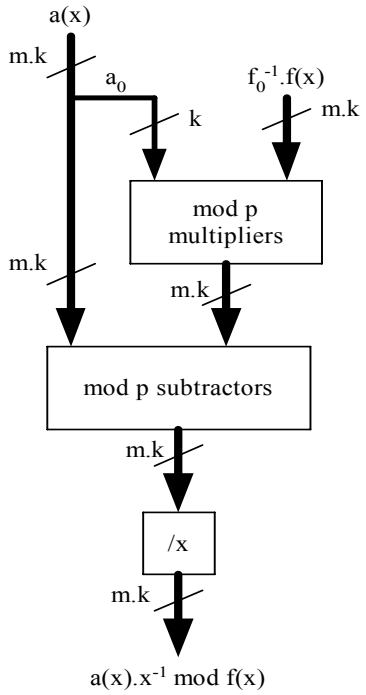

Figure 5 Computation of $a(x).x^{-1} \bmod f(x)$

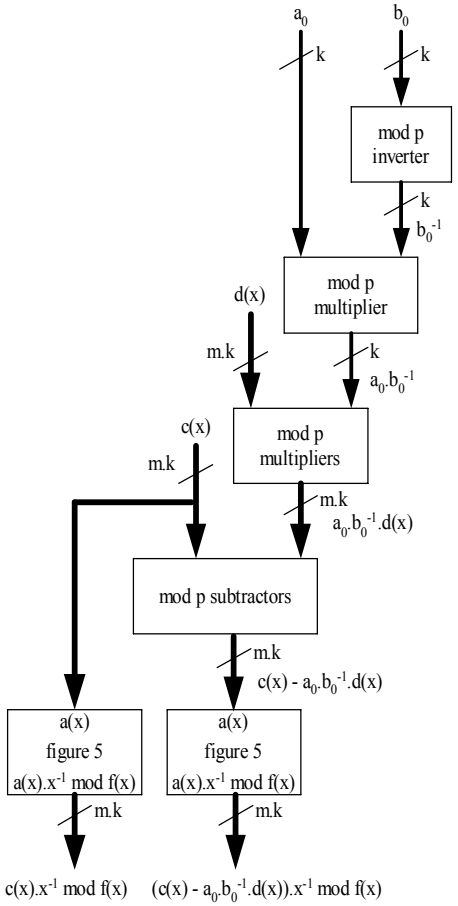

Figure 6 Binary algorithm

delays. Assuming that every two-input Boolean function is implemented within an FPGA's look-up table (LUT), an approximation of the total computation time is $10.m$ LUT-delays. If LUT-delays smaller than 2*ns* are assumed, the computation time is smaller than $20.m$ *ns*. As an example, for $m = 163$, the computation times should be less than 3260 *ns*. Actually, a computation time of 2445 *ns* has been reported (Deschamps and Sutter, 2006).

VII. POINT MULTIPLICATION

Point multiplication is the basic operation of Elliptic Curve Cryptography (section II): given a point R of an elliptic curve E and a natural s belonging to $0 < s < n$, it computes $sR = R + R + \ldots + R$. Generally n has the same order of magnitude as the number of field elements, that is p^m. In particular, if a curve E over $GF(2^m)$ is considered, then s is an m-bit number $s_0 + s_1.2 + \ldots + s_{m-1}.2^{m-1}$, and the point multiplication can be executed according to the following computation scheme:

$$sR = 2(\ldots (2(2\infty + s_{m-1}R) + s_{m-2}R) \ldots) + s_0R.$$

In the following algorithm the function `addition(U,V)` returns the sum of points U and V of E (relations (1) to (4)).

Algorithm 8 – Point multiplication

```
A := infinity_point;
for i in 1 .. m loop
  A := addition(A,A);
  if s(m-i) = 1 then A := addition(A,R);
end loop;
```

Every execution of the iteration step includes at most one point doubling ($A+A$, relation (3)) and one addition ($A+R$, relation (4)), that is, at most two field divisions and five field multiplications. Thus (sections V and VI) the iteration step execution time is less than $2\text{x}10m + 5\text{x}3m = 35m$ LUT-delays, and the total execution time less than $35m^2$ LUT-delays. If 2*ns* LUT-delays are assumed, the point multiplication execution time is less than $70m^2$ *ns*. As an example, for $m = 163$, the total computation time should be less than 2 *ms*.

VIII. CONCLUSIONS

Among all the considered operations, the most time-consuming is the exponentiation modulo n, with an order of magnitude of the delay proportional to m^3, being m the size of n, and a proportionality constant approximately equal to two per-bit-adder-delays. The second more time-consuming operation is the division modulo p, with an order of magnitude of the delay proportional to k^2, being k the size of p, and a proportionality constant approximately equal to two per-bit-adder-delays. As regards the operations modulo $f(x)$ over a binary field, their time complexities are proportional to m, being m the degree of f, that is the size of the field elements, and proportionality constants approximately equal to three LUT-delays (multiplication) and ten LUT-delays (division), respectively. As a matter of fact, the delays could be reduced if some of the FPGA's

dedicated AND and XOR gates were used, instead of general purpose LUTs.

Regarding the other (not considered here above) operations, notice that the addition and subtraction modulo n amount to two successive integer additions, so that their computation time is proportional to the size m of n. The multiplication modulo n can be realized with an m-by-m-bit parallel multiplier (a basic block of many FPGAs), whose computation time is proportional to m, followed by some modulo n reduction circuit, for example a circuit based on the Barrett algorithm (Blake, 2002; Hankerson, 2004) whose computation time is also proportional to m. The addition and subtraction modulo $f(x)$ are carry-free operations, so that their computation time is practically independent of the operand size.

To summarize, if specific circuits are used, all field operations have (at most) linear computation times, but the exponentiation modulo n (proportional to $(log\ n)^3$), the division modulo p (proportional to $(log\ p)^2$), and the point multiplication (proportional to $m^2 = (log\ q)^2$, where $q = 2^m$ is the number of field elements).

REFERENCES

Adleman, L.M., R.L.Rivest, and A.Shamir, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, **21**-**2**, 120-126 (1978).

Blake, I.V., G.Seroussi, and N.Smart, *Elliptic Curves in Cryptography*, Cambridge University Press (2002).

Brent, R.P., and H.T.Kung, "Systolic arrays for linear time GCD computation", *Proceedings of VLSI'83*, 145-154 (1983).

Deschamps, J-P., and G. Sutter, "Finite Field Division Implementation", *Proceedings of FPL 2005*, 670 - 675 (2005).

Deschamps, J-P., G.J.A. Bioul, and G. Sutter, *Synthesis of Arithmetic Circuits*, Wiley (2006).

Deschamps, J-P., and G. Sutter, "Hardware implementation of finite-field division", *Acta Applicandae Mathematicae, Special Issue on "Finite Fields: Applications and Implementations"*, Springer-Verlag (2006).

ElGamal, T.,"A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Transactions n Information Theory*, **IT-31**, 469-472 (1985).

Hankerson, D., A.J.Menezes, and S.Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag (2004).

Koblitz, N., *A Course in Number Theory and Cryptography*, Springer-Verlag (1994).

Montgomery, P., "Modular Multiplication without Trial Division", *Mathematics of Computation*, **44**, 519-521 (1985).

Received: April 14, 2006.
Accepted: September 8, 2006.
Recommended by Special Issue Editors Hilda Larrondo, Gustavo Sutter.