SPA resistant elliptic curve cryptosystem using addition chains

Andrew Byrne*, Francis Crowe and William Peter Marnane

Department of Electrical and Electronic Engineering,
University College Cork,
Cork, Ireland
E-mail: andrewb@rennes.ucc.ie
E-mail: francisc@rennes.ucc.ie
E-mail: liam@rennes.ucc.ie
*Corresponding author

Nicolas Meloni and Arnaud Tisserand

LIRMM, CNRS,
University Montpellier 2, France
E-mail: nicolas.meloni@lirmm.fr
E-mail: arnaud.tisserand@lirmm.fr

Emanuel M. Popovici

Department of Microelectronic Engineering,
University College Cork,
Cork, Ireland
E-mail: e.popovici@ucc.ie

Abstract: There has been a lot of interest in recent years in the problems faced by cryptosystems due to side channel attacks. Algorithms for elliptic curve point scalar multiplication such as the *double-and-add* method are prone to such attacks. By making use of special addition chains, it is possible to implement a Simple Power Analysis (SPA) resistant cryptosystem. In this paper, a reconfigurable architecture for a cryptographic processor is presented. A SPA resistant algorithm for point multiplication is implemented and is shown to be faster than the *double-and-add* method. Post place and route results for the processor are given.

Keywords: cryptography; elliptic curves; addition chains; side channel attacks.

Reference to this paper should be made as follows: Byrne, A., Crowe, F., Marnane, W.P., Meloni, N., Tisserand, A. and Popovici, E.M. (2007) ‘SPA resistant elliptic curve cryptosystem using addition chains’, *Int. J. High Performance Systems Architecture*, Vol. 1, No. 2, pp.133–142.

Biographical notes: Andrew Byrne received a BE in Electronic and Computer Engineering from NUI, Galway in 2004 and the MEngSc in Microelectronics from the NUI, Cork in 2005. He is currently working towards PhD.

Francis Crowe received a BE in Electrical Engineering from NUI, Cork in 2002. He is currently working towards PhD.

William Peter Marnane received his BE in Electrical Engineering from NUI, Cork in 1984 and the DPhil from the University on Oxford in 1989. He is a Senior Lecturer in the Department of Electrical and Electronic Engineering at the National University of Ireland, Cork since 1999.

Nicolas Meloni received an MS in Mathematics from the University of Montpellier in 2004. Since October 2004, he has been a PhD student in Computer Science in the LIRMM laboratory.

Arnaud Tisserand received his PhD (1997) and MSc (1994) in Computer Science from the Ecole Normale Suprieure de Lyon, France and BSc (1991) in Mechanical Engineering from the University of Lyon. He is a Senior Researcher CNRS, in LIRMM.

Emanuel M. Popovici is a Lecturer with the Department of Microelectronic Engineering, NUI, Cork since 2002. He received the Dipl. Ing. in Computer Engineering from the Politehnica University Timisoara, Romania (1997) and a PhD in Microelectronic Engineering from the National University of Ireland (2002), respectively.

Copyright © 2007 Inderscience Enterprises Ltd.

1 Introduction

Elliptic curve cryptography was proposed by Miller (1986) and Koblitz (1987). It provides a means for two hosts to generate a secret key for communication across an insecure channel. The strength of cryptography lies in the difficulty of an encryption schemes inverse operation. Elliptic curve cryptography provides relatively better security per bit than other cryptographic standards such as RSA (Vanstone, 2004).

Therefore, Elliptic Curve Cryptosystems (ECC) consume less memory and hardware resources to implement. The main operation of ECC is point scalar multiplication, given an elliptic curve E and a point P on E, the point $[k]P = P + P + \cdots + P$ for some given integer k. The basis for the strength of the ECC is the Elliptic Curve Discrete Logarithm Problem (ECDLP). Given two points Q and P on an elliptic curve E, find the integer k such that $Q = kP$. For a large enough key size, a brute force attack would require too much computing power and time to be feasible (Koblitz et al., 2000).

Recently, more effort has been carried out to secure EC point multiplication against side channel attacks (Kocher et al., 1999). By monitoring side channel information such as the power consumption of a device, it is possible to recover the secret information. Simple Power Analysis (SPA) attacks function on a single execution of the cryptographic algorithm under attack. By looking at the power trace of the execution, it is possible to identify the different functions of the algorithm. Algorithms such as the *double-and-add* method are prone to these types of attacks. Euclid's addition chains can provide both a secure and efficient scheme of exponentiation when combined with elliptic curves (Meloni, 2006).

Modular multiplication is one of the basic arithmetic operations used for cryptographic applications such as RSA and ECC. Several algorithms for modular arithmetic have been proposed and implemented. Montgomery (1985) proposed an efficient algorithm for fast multiplication using a series of additions and right shifts. Daly and Marnane (2002) implements a number of designs for multipliers based on the Montgomery multiplication.

The rest of this paper is structured as follows: Section 2 provides some background information on elliptic curves. Section 3 briefly introduces side channel attacks. In Sections 4.1 and 4.2, new algorithms for Point Addition (PA) and point scalar multiplication using addition chains are discussed. Section 5 describes the versatile processor and all its components. Implementation results are given.

2 Elliptic curves

An elliptic curve $E(\mathrm{GF}(p))$ over $\mathrm{GF}(p)$ is the set of points $P = (x, y), x, y \in \mathrm{GF(p)}$ such that

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,\ a_i \in \mathrm{GF}(p) \quad (1)$$

along with a special point at infinity ∂.

Elliptic curves over large prime fields are described using the Weierstrass equation

$$y^2 = x^3 + a_4x + a_6 \quad (2)$$

where x, y, a_4 and $a_6 \in \mathrm{GF}(p)$ and $4a_4^3 + 27a_6^2 \neq 0$.

Points on the elliptic curve can be represented in Jacobian coordinates which avoids the need for an expensive inversion operation (Daly et al., 2003). Converting from affine to projective coordinates is a simple operation, $(x, y, 1) \rightarrow (X, Y, Z)$. Conversion back, however, requires a number of modular multiplications and inversions, $(x, y) \leftarrow (X/Z^2, Y/Z^3)$. In Jacobian coordinates, the curve in Equation (2) is given by

$$Y^2 = X^3 + a_4XZ^4 + a_6Z^6 \quad (3)$$

2.1 EC multiplication

Given an elliptic curve E and a point P on the curve, the point Q is calculated by point scalar multiplication where the point P is added to itself k times to get the point $[k]P$.

Algorithms such as the *double-and-add* algorithm, given in Algorithm 1 are used to calculate point multiplication efficiently. The *double-and-add* algorithm requires n_k Point Doubling (PD) operations (n_k is the bit length of the key) and $w(k)$ PAs ($w(k)$ is the binary weight of the key).

Algorithm 1 Double and add point scalar multiplication

input : $P \in E(\mathrm{GF}(q)), k = \sum_{i=0}^{n_k-1} k_i2^i$
output: $Q = [k]P \in E(\mathrm{GF}(q))$
Initialise: $Q = P$;
for $i \leftarrow n_k - 2$ **to** 0 **do**
  $Q = 2Q$ //Point Doubling (PD);
  if $k_i = 1$ **then**
    $Q = Q + P$ //Point Addition (PA);
  end
end

2.2 EC point addition and doubling

Consider two separate points on an elliptic curve, $P = (x_p, y_p)$ and $Q = (q_t, q_t)$. A line l is drawn through the points P and Q. The line l intersects the curve at a third point. $Q' = (x_{q'}, y_{q'})$ is the inverse of that point, where $Q' = P + Q$. The PA formulae for the curve defined in Equation (2) using Jacobian coordinates are given in Algorithm 2. The computational cost of a PA is 15 multiplications and 7 add/subs.

Algorithm 2 PA in jacobian coordinates

input : $P(X_1, Y_1, Z_1), Q(X_2, Y_2, Z_2) \in \mathrm{GF}(q)$
output: $P + Q(X_3, Y_3, Z_3) \in E(\mathrm{GF}(q))$
$A = X_1Z_2^2,\ B = X_2Z_1^2,\ C = Y_1Z_2^3,\ D = Y_2Z_1^3$;
$E = B - A,\ F = D - C$;
$X_3 = -E^3 - 2AE^2 + F$;
$Y_3 = -CE^3 + F(AE^2 - X_3),\ Z_3 = Z_1Z_2E$

If $T = P$, then this is PD and a tangent to the point is used. The tangent intersects with the curve at a second point, $T' = 2(T)$ is the inverse of this point. Algorithm 3 gives the formulae for PD for the curve in Equation (2). The computational cost for PD is 10 multiplications and 8 add/subs.

Algorithm 3 Point doubling in jacobian coordinates

input : $P(X_1, Y_1, Z_1) \in \mathrm{GF}(q)$
output: $[2]P(X_3, Y_3, Z_3) \in E(\mathrm{GF}(q))$
$A = 4X_1Y_1^2,\ B = 3X_1^2 + a_4Z_1^4;$
$X_3 = -2A + B^2;$
$Y_3 = -8Y_1^4 + B(A - X_3),\ Z_3 = 2Y_1Z_1$

3 Side channel attacks

In recent years, cryptosystems have come under attack from various forms of side channel attack. Kocher et al. (1999) discovered that cryptosystem implementations leak information which can help an attacker recover secret data. Two techniques for retrieving secret information in this manner are SPA and Differential Power Analysis (DPA). DPA executes the cryptographic algorithm under attack a number of times and uses statistical analysis to determine the secret information (Coron, 1999). Countermeasures for ECC such as randomisation of the inputs (Joye and Tymen, 2001) and blinding techniques (Messerges et al., 1999) can be implemented to protect against such attacks.

SPA involves monitoring the power consumption of a single execution of a cryptographic algorithm. Every instruction has a different power consumption, therefore it is possible to retrieve the sequence of instructions during the algorithm execution. For example, the *double-and-add* algorithm has two primary operations, PA and PD. Each of these operations produce a different power trace when executed because of the different number of multiplications and additions in each algorithm. Since, the execution of a PA in the *double-and-add* is directly related to the secret key, it is possible to retrieve the secret key by monitoring the power consumption of a single execution of a scalar multiplication. The first successful power analysis attack against an FPGA was done by Ors et al. (2003) in which they attacked an elliptic curve processor and retrieved the secret key.

SPA attacks work well on algorithms where the the power consumption can be directly related to the instruction being executed. In order to resist SPA attacks, the power consumption of the instructions executed in a cryptographic algorithm must not be directly related to the secret data. In the double and add method, the branch instruction based on k_i leaks information about the secret key.

Joye and Yen (2003) describe the *Montgomery Power Ladder* for exponentiation. Algorithm 4 illustrates the standard algorithm for the montgomery ladder. It can be seen that the operations for each branch of the algorithm are the same, a multiplication followed by a squaring. Therefore, the Montgomery ladder is protected against simple side channel attacks, including SPA. The cost of implementing this, however, is $2n_k$ point multiplications and squarings, n_k is the bit length of the key.

In this paper, we make use of special *Addition Chains* to perform a point multiplication using only point additions. This technique removes the conditional branching that allows an SPA attack to identify the bits of the secret key.

Algorithm 4 Montgomery ladder

input : g ; $k = \sum_{i=0}^{n_k-1} k_i 2^i$
output: $y = g^k$
Initialise: $R_0 \leftarrow 1$; $R_1 \leftarrow g$;
for $j \leftarrow n_k - 1$ **to** *0* **do**
  if $k_j = 0$ **then**
    $R_1 \leftarrow R_0R_1$; $R_0 \leftarrow (R_0)^2$;
  else
    $R_0 \leftarrow R_0R_1$; $R_1 \leftarrow (R_1)^2$;
  end
end
RETURN : R_0

4 Euclidean addition chains

In this section, we present the Euclidean addition chains and how they can be adapted for elliptic curve point scalar multiplication. Using a new algorithm for the point addition of two points on an elliptic curve that share the same z-coordinate, we can improve the calculation time of the new point scalar multiplication algorithm.

Definition 1: *An addition chain computing an integer k is given by a sequence* $v = (v_1, \ldots, v_s)$ *where* $v_1 = 1,\ v_s = k$ *and* $\forall 1 \le i \le s,\ v_i = v_{i_1} + v_{i_2}$ *for some* i_1 *and* i_2 *lower than* i.

Definition 2: *An Euclidean Addition Chain (EAC) computing an integer k is an addition chain which satisfies* $v_1 = 1$, $v_2 = 2,\ v_3 = v_2 + v_1$ *and* $\forall 3 \le i \le s - 1$, *if* $v_i = v_{i-1} + v_j$ *for some* $j < i - 1$, *then* $v_{i+1} = v_i + v_{i-1}$*(case 1) or* $v_{i+1} = v_i + v_j$ *(case 2).*

Case 1 will be called big step (we add the biggest of the two possible numbers to v_i*), and case 2, small step (we add the smallest one).*

As an example (1, 2, 3, 4, 7, 11, 15, 19, 34) is an Euclidean addition chain computing 34. For instance, in step 4, we have computed $4 = 3 + 1$, thus in step 5, we must add 3 or 1 to 4, in other words, from step 4 we can only compute $5 = 4 + 1$ or $7 = 4 + 3$. In this example, we have chosen to compute $7 = 4 + 3$ so, at step 6, we can compute $10 = 7 + 3$ or $11 = 7 + 4$, etc. Another classical example of EAC is the Fibonacci sequence (1, 2, 3, 5, 8, 13, 21, 34) (which is only made of big steps).

Finding such chains is quite simple, it suffices to choose an integer g coprime with k and apply the subtractive form of Euclid's algorithm.

Example 1: *Let* $k = 34$ *and* $g = 19$ *and apply them to the subtractive form of Euclid's algorithm:*

$34 - 19 = 15$ (big step)
$19 - 15 = 4$ (small step)
$15 - 4 = 11$ (small step)
$11 - 4 = 7$ (big step)
$7 - 4 = 3$ (big step)

$$4-3=1 \quad \text{(small step)}$$
$$3-1=2$$
$$2-1=1$$
$$1-1=0$$

Reading the first number of each line gives the EAC $(1, 2, 3, 4, 7, 11, 15, 19, 34)$.

Finally, in order to simplify the writing of the algorithm, we will use the following notation : if $v = (1, 2, 3, v_4, \ldots, v_s)$ is an EAC, then we only consider the chain from v_4 and we replace all the v_i's by 0 if it has been computed using a big step and by 1 for a small step.

For instance, the sequence:	$(1,\ 2,\ 3,\ 4,\ 7,\ 11,\ 15,\ 19,\ 34)$
will be written:	$(1,\ 0,\ 0,\ 1,\ 1,\ 0)$

4.1 New point addition formulae

Given the formulae in Algorithm 2 and the points $P_1 = (X_1, Y_1, Z)$ and $P_2 = (X_2, Y_2, Z)$ on the curve E over $GF(p)$, $p > 3$. Then for $P_1 + P_2 = P_3 = (X_3, Y_3, Z_3)$

$$\begin{aligned} X_3 &= (Y_2Z^3 - Y_1Z^3)^2 - (X_2Z^2 - X_1Z^2)^3 \\ &\quad - 2X_1Z^2(X_2Z^2 - X_1Z^2)^2 \\ &= ((Y_2 - Y_1)^2 - (X_2 - X_1)^3 - 2X_1(X_2 - X_1)^2)Z^6 \\ &= ((Y_2 - Y_1)^2 - (X_1 + X_2)(X_2 - X_1)^2)Z^6 \\ &= X_3'Z^6 \end{aligned}$$

$$\begin{aligned} Y_3 &= -Y_1Z^3(X_2Z^2 - X_1Z^2)^3 \\ &\quad + (Y_2Z^3 - Y_1Z^3)(X_1Z^2(X_2Z^2 - X_1Z^2)^2 - X_3) \\ &= (-Y_1(X_2 - X_1)^3 \\ &\quad + (Y_2 - Y_1)(X_1(X_2 - X_1)^2 - X_3'))Z^9 \\ &= Y_3'Z^9 \end{aligned}$$

$$\begin{aligned} Z_3 &= Z^2(X_2Z^2 - X_1Z^2) \\ &= Z(X_2 - X_1)Z^3 \\ &= Z_3'Z^3 \end{aligned}$$

Thus, we have $(X_3, Y_3, Z_3) = (X_3'Z^6, Y_3'Z^9, Z_3'Z^3) \sim (X_3', Y_3', Z_3')$.

So, when P_1 and P_2 have the same z-coordinate, $P_1 + P_2$ can be obtained using the following formulae:

Algorithm 5 Point addition, P and Q sharing same Z-coordinate

input : $P(X_1, Y_1, Z), Q(X_2, Y_2, Z) \in GF(q)$
output: $P + Q(X_3', Y_3', Z_3') \in E(GF(q))$
$A = (X_2 - X_1)^2,\ B = X_1A,$
$C = X_2A,\ D = (Y_2 - Y_1)^2$,;
$X_3' = D - B - C$,;
$Y_3' = (Y_2 - Y_1)(B - X_3) - Y_1(C - B)$,;
$Z_3' = Z(X_2 - X_1)$

This addition involves 5 multiplications, 2 squarings and 7 additions/subtractions.

As they require special conditions, these formulae are logically more efficient than any general or mixed addition formulae. Compared to the 15 multiplications and 7 additions/subtractions for point addition in Algorithm 2, this is a great saving.

4.2 Point scalar multiplication

From the previous section, we have seen that our formulae are quite efficient in terms of computational cost (more than a doubling) but cannot be used with classical double-and-add algorithms and require specific exponentiation schemes.

We note the chain $c = (c_4, \ldots, c_s)$ instead of v in order to prevent confusion between both representations.

We can now propose an algorithm performing a point scalar multiplication using a new function, `NewADD`. The `NewADD` function works in the following way: let P_1 and P_2 be two points sharing the same z-coordinate then `NewADD`(P_1, P_2) returns two points, $P_1 + P_2$ and P_1, sharing the same z-coordinate. $P_1 + P_2$ is calculated as in Algorithm 5.

Algorithm 6 Euclid-Exp(c, P)

Data: P, $[2]P$ with $Z_P = Z_{[2]P}$ and an EAC $c = (c_4, \ldots, c_s)$ computing k
Result: $[k]P \in E$
begin
  $(U_1, U_2) \leftarrow ([2]P, P)$
  for $i = 4 \ldots s$ **do**
    if $c_i = 0$ **then**
      $(U_1, U_2) \leftarrow$ `NewADD`(U_1, U_2) ;
    else
      $(U_1, U_2) \leftarrow$ `NewADD`(U_2, U_1) ;
    end
  end
  $(U_1, U_2) \leftarrow$ `NewADD`(U_1, U_2) ;
  return U_1
end

Example 2: *Let us see what happens with the chain $c = (1, 0, 0, 1, 1, 0)$ computing 34:*

begin	$([2]P, P)$
$c_4 = 1$	*NewADD*$(P, [2]P) = ([3]P, P)$
$c_5 = 0$	*NewADD*$([3]P, P) = ([4]P, [3]P)$
$c_6 = 0$	*NewADD*$([4]P, [3]P) = ([7]P, [4]P)$
$c_7 = 1$	*NewADD*$([4]P, [7]P) = ([11]P, [4]P)$
$c_8 = 1$	*NewADD*$([4]P, [11]P) = ([15]P, [4]P)$
$c_9 = 0$	*NewADD*$([15]P, [4]P) = ([19]P, [15]P)$
	NewADD$([19]P, [15]P) = [34]P$

If we consider that the point P is given in affine coordinate (i.e. $Z = 1$), then the doubling step can be performed using $3M$ and $3S$, and so, the total computational cost of our algorithm is $(5s - 7)M$ and $(2s - 1)S$.

Some cryptographic protocols only require the x-coordinate of the point $[k]P$. In this case, it is possible to save one multiplication by step of Algorithm 6 by noticing that Z does not appear during the computation of X'_3 and Y'_3, thus it is not necessary to compute Z'_3 during the process. The x-coordinate can be recovered at the end.

4.3 About Euclid's addition chains length

At this point, we know that Euclidean addition chains are easy to compute, however, finding small chains is a lot more complicated.

We begin by a theorem proved by Knuth and Yao (1975).

Theorem 1: *Let $S(k)$ denote the average number of steps to compute $gcd(k, g)$ using the subtractive Euclid's algorithm when g is uniformly distributed in the range $1 \le g \le k$. Then*

$$S(k) = 6\pi^{-2}(\ln k)^2 + O(\log k(\log k)^2)$$

This theorem shows that if, in order to find an EAC for an integer k, we choose an integer g at random, it will return a chain of length about $(\ln k)^2$, which is too long to be used with ECC. Indeed, for a 160-bit exponent to be efficient, Algorithm 6 requires chains of length, at the most, 320. The previous theorem tells us that, theoretically, random chains for a 160-bit exponent have a length of 7000 on average (it is rather 2500 in practice). Therefore, we need to find ways to reduce the length of these chains.

A classic way to limit the length of EAC is to choose g close to k/ϕ, where $\phi = (1 + \sqrt{5}/2)$ is the golden section. This guarantees that the last steps of the EAC will be big steps. In practice, this method allows EAC of an average length of 1100 to be found.

Considering 160-bit integers, finding EAC of length 320 can be done by checking (on average) about 30 $g's$. Finding shorter chains is a lot more difficult, as an example, finding chains of length 270 requires testing more than 45,000 $g's$. Such a computation cannot be integrated into any exponentiation algorithm, so, if some offline computations cannot be performed, one should not expect to use EAC whose length is shorter than 320.

5 Elliptic curve processor

A generic architecture (Figure 1) was designed for cryptographic operations which incorporates RAM, a ROM controller and a number of arithmetic units for a given field. Software was developed using C++ to generate the VHDL for a customised processor for any characteristic p and extension field m. Everything from the size of the RAM block to configuring the arithmetic units and generating the ROM instruction set for a given algorithm is controlled by the program.

For prime characteristic fields, there is a choice of arithmetic units to choose from for the architecture. Through manipulation of the ROM instructions alone, the processor can be configured for various algorithms including the *double-and-add* algorithm or exponentiation using addition chains. In this way, we can quickly compare these and other cryptographic algorithms. In the next section, we will look at the arithmetic units used in the processor.

5.1 Arithmetic units

The point addition and doubling algorithms described in Sections 2.2 and 4.1 require modular additions, subtractions and multiplications. While addition and subtraction are relatively easy to implement, modular multiplication is much more complex. An in depth review of modular arithmetic and architectures can be found in Daly et al. (2005).

The processor architecture in Figure 1 is capable of controlling a number of arithmetic units. There are two architecture types available for the GF(p) processor. Dedicated units for each of addition, subtraction and multiplication can be implemented. The number of multipliers implemented in the processor can be configured based on the speed/area constraints of the target technology and the application of the design. Since addition and subtraction only take 4 clock cycles to complete, two of which are RAM read/writes, these operations are best performed in series and do not gain from an increased number of arithmetic units. Alternatively, we can use configurable Arithmetic Logic Units (ALU) that can be set to perform modular addition, subtraction or multiplication. The increased functionality of the units reduces the area consumption compared to the three dedicated units combined. As with the dedicated multipliers, the number of ALUs can be changed to give optimum results based on the target device constraints.

5.1.1 Multiplication

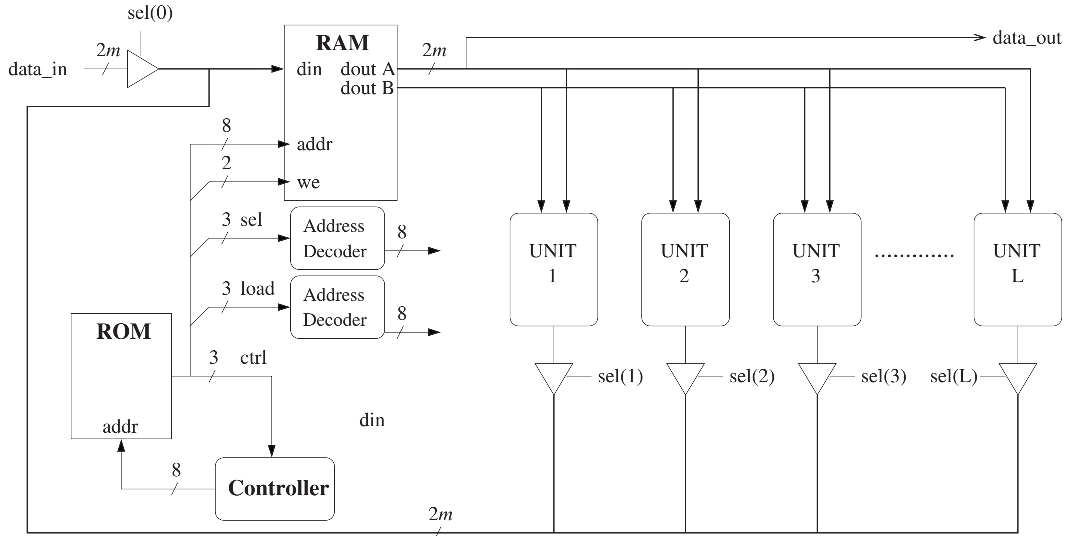
In 1985, Montgomery (1985) proposed an efficient method for performing modular multiplication using a series of additions and right shifts. This method avoids the need for costly trial division of the modulus. The Montgomery modular product is defined in Equation (4).

$$\text{Res} = \text{Mont}(A, B, p) = AB2^{-p_b+2} (\text{mod } p) \tag{4}$$

The output of a Montgomery multiplication is a factor 2^{-p_b+2} times smaller than the desired result, p_b is the field size in bits. In order to correct the result, the output must be Montgomery multiplied by $(2^{2p_b+2} \text{ mod } p)$. When a large number of multiplications are required, it becomes inefficient to correct every result. A better solution is to initially convert the numbers to the Montgomery domain. To do this, the number is Montgomery multiplied by $(2^{2p_b+2} \text{ mod } p)$. To convert a number back, it is Montgomery multiplied by 1.

The algorithm for the Montgomery multiplication is given in Algorithm 7. The number of iterations performed is $p_b + 2$ in order to bound the output in the range $[0, 2p - 1]$ for multiplicands up to twice the modulus. This allows it to be used as an input for further multiplications without the need for conditional subtraction.

Figure 1 General Elliptic curve processor

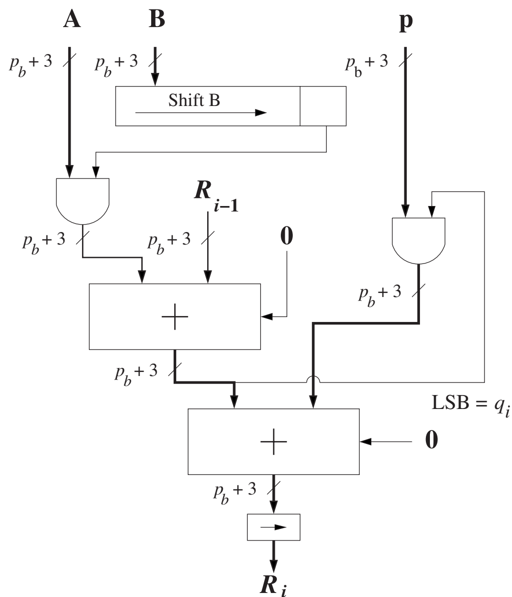


Algorithm 7 Montgomery multiplication

input : $A = \sum_{i=0}^{p_b} a_i 2^i$, $B = \sum_{i=0}^{p_b} b_i 2^i$,;
$M = \sum_{i=0}^{p_b} p_i 2^i$
output: $R = AB2^{-p_b+2}\ (mod\ p)$
Initialise: $R \leftarrow 0$; $b_{p_b+1} \leftarrow 0$;
for $i \leftarrow 0$ **to** $p_b + 1$ **do**
  $q_i = R_{i-1} + b_i A\ (mod 2)$;
  $R_i = (R_{i-1} + q_i M + b_i A)/2$;
end

A hardware implementation of the Montgomery multiplier can be seen in Figure 2. Multiplication is performed according to Algorithm 7. The inputs to the first adder are $b_i A$ and the previous result R_{i-1}. $q_i p$ is added to the sum of the first adder if the LSB of the sum (q_i) is equal to 1. A shift register scans each bit of B for $b_i A$ and the final result is right shift divided by 2.

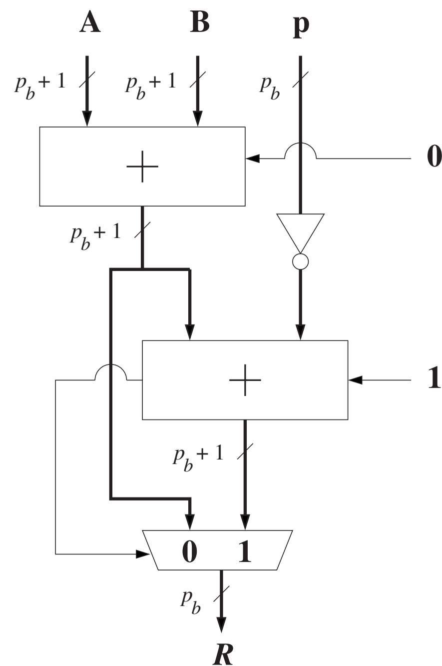
Figure 2 Modular multiplier



5.1.2 Modular addition and subtraction

The modular addition operation adds A and B in the first adder and subtracts the modulus p from the sum. To subtract the modulus from the intermediate result, the modulus is bitwise inverted and added to $(A + B)$ with the carry-in set to 1, thus performing a two's complement subtraction. The carry-out of the second adder controls which intermediate result is the correct result. If $(A + B)$ is in the correct range, the result of the first adder is the correct result. Otherwise, the result from the second adder is correct. The architecture for the adder/subtractor in Figure 3 is configured for modular addition.

Figure 3 Modular adder



Modular subtraction is performed similarly. In this case, however, B is bitwise inverted and added to A with the carry-in set to 1. If the carry-out of this adder is low, the modulus must be added to give an output in the correct range.

5.1.3 Configurable ALU

Figure 4 presents the architecture for the configurable ALU. By combining the different arithmetic functions in one unit, the same hardware can be configured to perform each function. Therefore, we reduce the time during the execution of some cryptographic algorithm where the arithmetic units are idle. The modular addition, subtraction and multiplication operations are controlled with a 2-bit mode signal set to 00.01 and 10 for these operations respectively. A 2-bit load signal is used also to load the operands.

Figure 4 Configurable ALU

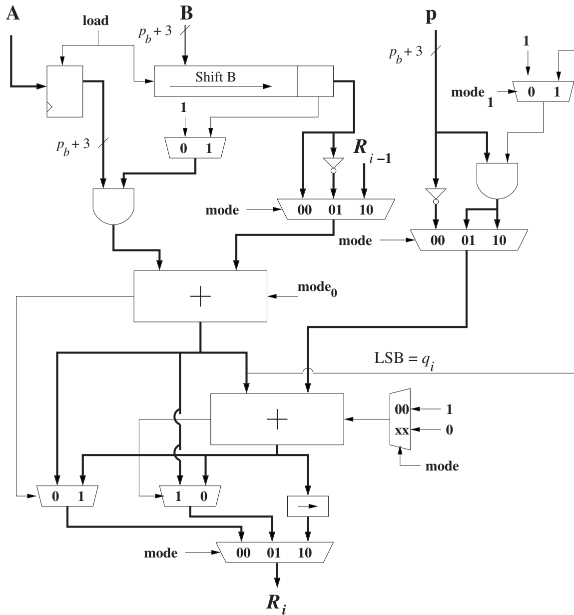


Table 1 gives the speed and area performance of the arithmetic units used for the processor. All results are based on 192-bit prime modulus and are implemented on the xc2v6000-4.

Table 1 Post place and route results for Xilinx xc2v6000-4

	ALU	*Multiplier*	*Adder/subtractor*
Slices	966	397	391
Min. period	25.707 ns	21.333 ns	26.398 ns
Clock Freq.	38.9 MHz	46.876 MHz	37.882 MHz

The dedicated multiplier, adder and subtractor give better area results than the configurable ALU. This is due to the extra control logic surrounding the ALU for selecting the mode thus increasing the minimum area. The dedicated multiplier also performs faster than the configurable ALU and the adder/subtractor. This is due to the large multiplexers needed for the adder and subtractor to select the correct result from the carry adders.

5.2 ROM instruction set

When deciding the control for the processor, there are two options. The first is a finite state machine which can be set up to perform specific operations such as elliptic curve point scalar multiplication. This does not allow for much flexibility in the design, however. Instead, the use of microcode stored in Xilinx BlockROM was implemented. A similar approach was taken by Leong and Leung (2002) which helped reduce the development time of the processor and increased the flexibility of the design. A major advantage of this is that the instruction set can be updated to perform any number of operations without the need to recompile the entire processor.

When generating the ROM instruction set, several considerations need to be taken into account such as, are the points on the elliptic curve being represented by projective or affine coordinates; what algorithm is being implemented; how many arithmetic units are available and what are their timing constraints.

For the field p, p large prime, configurable and dedicated ALUs for modular addition, subtraction and multiplication based on Crowe et al. (2005) were implemented. In this section, we will look only at the architecture implementing configurable ALUs. For the architecture to accommodate this type of unit, *mode* bits were needed to set the operation of the ALUs. The instruction set for a reconfigurable ALU based processor over GF(p) using projective coordinates is shown in Table 2. After initially loading the elliptic curve parameters and Montgomery constants into RAM, the controller performs operations for the selected cryptographic algorithm.

Table 2 Instruction set using reconfigurable ALUs

Instruction set					
Ctrl	*Mode*	*Load*	*Sel*	*We*	*Addr A and B*
00	00000000	000	000	00	00001 00010
01	00100000	011	000	00	00000 00000
00	00000000	000	011	01	00000 00011

Here, we are using projective coordinates to represent the points on the elliptic curve to remove the need for inversions which are time consuming. The 12 LSBs control read and write access to the dual port RAM. Bits 12 → 14 control the tri-states connected to the outputs of the ALUs. Only one of these is set high when writing data to RAM. To reduce the impact of a large number of arithmetic units in the design on the size of ROM, a 3-to-8 address decoder is used to make full use of all combinations of the 3 select bits. Bits 15 → 17 are the load bits which are used to load new vectors to a specific ALU. Bits 18 → 25 control the mode signal for each ALU. In this example, there are 4 ALUs in the design, 2 mode bits per ALU. The two MSBs are extra control bits for the state machine controlling the processor.

Some operations such as addition and subtraction execute in a single clock and have no extra timing requirements associated with them. Multiplication, however, takes $p_b + 3$ clocks. The controller for the ROM is a simple counter that goes through each address sequentially on each rising clock edge. To account for the operational time between loading the operands and getting the result, a state machine is needed to handle these exceptions. By monitoring bit 26 of the instruction, the state machine can halt normal execution of the instructions for a set number of clock cycles while a series of multiplications are performed.

5.3 Hardware generator

The versatile processor is presented in Figure 1. VHDL for the processor is generated by software developed in C++. This allows for the system to be completely reconfigurable for any characteristic and extension field. The generation of the ROM is explained in Section 5.2. The architecture presented makes use of dual-port RAM (though it can be configured for single-port RAM) so both operands of an arithmetic unit can be loaded in parallel.

The processor in Figure 1 can be configured for a number of different algorithms over GF(p) using the arithmetic units described in Section 5.1. It can also be configured for GF(2^m) and GF(3^m) where dedicated units are used for multiplication, addition, subtraction, inversion and division (Byrne and Marnane, 2006). For GF(p), the number of configurable ALUs implemented can be modified using the software developed and is restricted by the FPGA resources and the point where additional units no longer give a substantial improvement in the design. Likewise, using dedicated units, the number of multipliers, adders and subtractors can be configured within the constraints of the target device and performance gain. All changes to the processor are handled by the software.

5.3.1 ALU configurations and results

There is a tradeoff between speed and area that is affected by the number of ALUs implemented in the processor. More ALUs will reduce execution time, but will increase the area consumption of the device. To be truly reconfigurable, an automated scheduling tool must be used to ensure an efficient implementation. The efficient transfer and processing of data through the processor can greatly improve speed/area tradeoffs.

Figure 5 Schedule for point addition using three ALUs

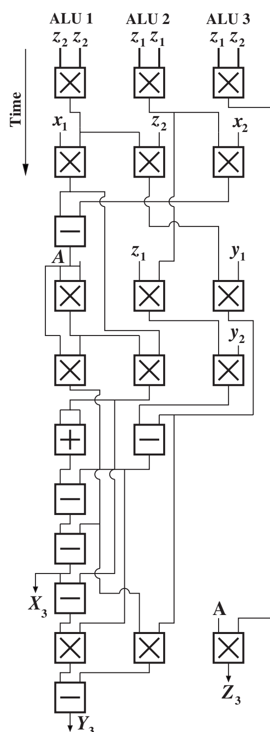
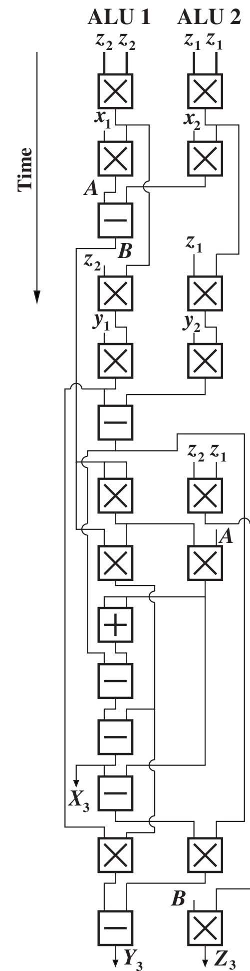


Figure 6 Schedule for point addition using two ALUs



Two simple methodologies, As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) (Parhi, 1999) assume limitless resources and schedule operations either at the earliest or latest time step possible. Given the lack of constraints, the scheduling results are poor and can lead to an inefficient use of resources. More advanced scheduling algorithms such as List-Based-Scheduling (LBS) (Bertoni et al., 2006) can efficiently schedule a set of instructions, given a set of resource constraints. With LBS, we can limit the resources available to the processor and schedule operations for a given algorithm efficiently based on these constraints.

Table 3 Execution time in terms of clock cycles

ALUs	*PA*	*PD*	*New PA*
1	$15p_b + 103$	$10p_b + 102$	$7p_b + 63$
2	$8p_b + 82$	$6p_b + 90$	$4p_b + 54$
3	$5p_b + 73$	$5p_b + 87$	$3p_b + 51$

Schedules for each algorithm were generated for a varying number of ALUs and it was found that for the regular point addition algorithm (Algorithm 2), the best speed/area tradeoff was three ALUs. Though it is possible to have four multiplications in parallel, the algorithm can be scheduled more efficiently using three ALUs instead of four. Figures 5 and 6 illustrate the schedules for the point addition in Algorithm 2 using three and two ALUs, respectively. Implementing three ALUs, multiplication results in five

Table 4 Post place and route results for Xilinx xc2v6000-4

	Double-and-add			*Addition chains*		
ALUs	*1*	*2*	*3*	*1*	*2*	*3*
Slices	1644 (4%)	2412 (7%)	3735 (11%)	1644 (4%)	2412 (7%)	3735 (11%)
min. period (ns)	24.96	25.981	25.732	24.96	25.981	25.732
Clock Freq.	40.064 MHz	38.49 MHz	38.86 MHz	40.064 MHz	38.49 MHz	38.86 MHz
Time (ms)	13.8	8.52	6.43	11.2	6.86	5.14

multiplication stages as seen in Figure 5. Using two ALUs in Figure 6 increases the number of multiplication stages to eight.

Table 3 lists the optimum execution times in terms of clock cycles for the Point Addition (PA) and Point Doubling (PD) algorithms in Sections 2.2 and 4.1. It is clear from the table that the reduced point addition algorithm where the two points share the same z-coordinate can be calculated in far less time. Even with only one ALU implemented, it outperforms the regular point addition using two ALUs. The best speed results are achieved with three ALUs for all the algorithms. Increasing the number of ALUs further does not improve the execution time.

As described in Section 5.1.1, multiplication is executed in $p_b + 2$ clock cycles. Additions and subtractions take 2 clock cycles. For a field size of 192 and performing all multiplications and add/subs in series, point addition described by Algorithm 2 is executed in 2983 clock cycles while a point doubling is executed in 1983 cycles. Using the improved formulae from Algorithm 5, a point addition can be executed in 1407 clock cycles, half that of the previous algorithm for point addition.

The architecture presented in Section 5 was evaluated on Xilinx xc2v6000-4. The post place and route results for point multiplication using the *double-and-add* and addition chain methods are listed in Table 4. Each design consumes approximately 5% of the Block RAMs available. The largest designs using 3 ALUs only need 23 bits for the ROM instruction set leaving room to extend the instruction for future applications. The results are based on a 160-bit key size. For the addition chains, a chain of length 320 was used.

From the results in Table 4, it can be seen that the new point scalar multiplication algorithm using the modified point addition algorithm gives a significant improvement in execution time over the *double-and-add* method. Since the addition chain method does not depend on point doubling, the effect of the new point addition algorithm is much greater. Implementing three ALUs with the improved point addition using addition chains gives the best time results with a slight increase in area consumption.

Using addition chains, the dedicated modular adder, subtractor and Montgomery multiplier were also implemented in the processor. With a configuration of 1 adder, 1 subtractor and 2 Montgomery multipliers, the processor consumes 3001 slices (8% of the target device) and operates at 45.92 MHz. Although there is a slight increase in area consumption, the faster clock frequency means the execution time for the new point scalar multiplication algorithm for a 192-bit field size executes in 5.75 ms.

6 Conclusions

In this paper, a reconfigurable cryptographic processor has been used to efficiently compare two algorithms for elliptic curve point multiplication. Using addition chains, we have found that not only can the cryptosystem resist SPA attacks, but can also outperform a double-and-add approach. By varying the number of arithmetic units implemented in the architecture, the processor can be configured for devices with different levels of resources. An improvement in execution time was also found when implementing dedicated units instead of the configurable ALU. This is at the expensive of a slight increase in overall area.

Acknowledgements

The support of the Informatics Commercialisation initiative of Enterprise Ireland and Ulysses 'The France-Ireland Exchange Scheme' are gratefully acknowledged.

References

Bertoni, G., Breveglieri, L., Fragneto, P. and Pelosi, G. (2006) 'Parallel hardware architectures for the cryptographic tate pairing', *Third Internation Conference on Information Technology: New Generations*, pp.186–191.

Byrne, A. and Marnane, W.P. 2006 'Versatile processor for GF(p^m) arithmetic for use in cryptographic applications', *24th IEEE Norchip Conference*, pp.281–284.

Coron, J.S. (1999) 'Resistance against differential power analysis for elliptic curve cryptosystems', *Cryptographic Hardware and Embedded Systems – CHES*, Vol. 1717 of *Lecture Notes in Computer Science*, pp.292–302.

Crowe, F., Daly, A. and Marnane, W. (2005) 'A scalable dual mode arithmetic unit for public key cryptosystems', *IEEE International Conference on Information Technology: Coding and Computing (ITCC)*, Vol. 1, pp.568–573.

Daly, A. and Marnane, W. (2002) 'Efficient architectures for implementing montgomery modular multiplication and rsa modular exponentiation on reconfigurable logic', *International Symposium on Field Programmable Gate Arrays*, pp.40–49.

Daly, A., Marnane, W., Kerins, T. and Popovici, E. (2005) 'An FPGA implementation of a GF(p) ALU for encryption processors', *Elsevier Journal on Microprocessors and Microsystems (Special Issue on FPGAs: Applications and Designs)*, Vol. 28, No. 5–6, pp.253–260.

Daly, A., Marnane, W. and Popovici, E. (2003) 'Fast modular inversion in the Montgomery domain on reconfigurable logic', *Irish Signals and Systems Conference*, Vol. 19, pp.363–367.

Joye, M. and Tymen, C. (2001) 'Protections against differential analysis for elliptic curve cryptography: an algebraic approach', *Cryptographic Hardware and Embedded Systems – CHES*, Vol. 2162 of *Lecture Notes in Computer Science*, pp.337–390.

Joye, M. and Yen, S. (2003) 'The Montgomery powering ladder', *Cryptographic Hardware and Embedded Systems – CHES*, Vol. 2523 of *Lecture Notes in Computer Science*, pp.291–302.

Knuth, D. and Yao, A. (1975) 'Analysis of the subtractive algorithm for greatest common divisors', *Proceedings of National Academic Science*, Vol. 72, No. 12, pp.4720–4722.

Koblitz, N. (1987) 'Elliptic curve cryptosystems', *Mathematical Computation*, Vol. 48, pp.203–209.

Koblitz, N., Menezes, A. and Vanstone, S. (2000) 'The state of elliptic curve cryptography', *Design Codes and Cryptography*, Vol. 19, pp.173–193.

Kocher, P., Jaffe, J. and Jun, B. (1999) 'Differential power analysis', *Advances in Cryptology – CRYPTO'99*, Vol. 1666 of *Lecture Notes in Computer Science*, pp.388–397.

Leong, P. and Leung, I. (2002) 'A microcoded elliptic curve processor using FPGA technology', *IEEE Transactions on VLSI Systems*, Vol. 10, No. 5, pp.550–559.

Meloni, N. (2006) 'Fast and secure elliptic curve scalar multiplication over prime fields using special addition chains', *Cryptology ePrint Archive, Report 2006/216*, Available at: http://eprint.iacr.org/.

Messerges, T., Dabbish, E. and Sloan, R. (1999) 'Power analysis attacks of modular exponentiation in smartcards', *Cryptographic Hardware and Embedded Systems – CHES*, Vol. 1717 of *Lecture Notes in Computer Science*, pp.144–157.

Miller, V. (1986) 'Use of elliptic curves in cryptography', *CRYPTO '85*, Vol. 218 of *Lecture Notes in Computer Science*, pp.417–426.

Montgomery, P.L. (1985) 'Modular multiplication without trial division', *Mathematics of Computations*, Vol. 44, pp.519–521.

Ors, S.B., Oswald, E. and Preneel, B. (2003) 'Power-analysis attacks on an FPGA – First experimental results', *Cryptographic Hardware and Embedded Systems – CHES*, Vol. 2279 of *Lecture Notes in Computer Science*, pp.35–50.

Parhi, K. (1999) *VLSI Digital Signal Processing Systems: Design and Implementation*, Wiley-Interscience.

Vanstone, S. (2004) 'ECC holds key to next-gen cryptography', *Technical Report, Certicom Corporation*.