RIELAC, Vol. 41 3/2020 p. 1-20 Septiembre-Diciembre ISSN: 1815-5928

RIELAC
Revista de Ingeniería Electrónica, Automática y Comunicaciones

Speeding up elliptic curve arithmetic on ARM processors using NEON instructions

Raudel Cuiman Márquez, Alejandro J. Cabrera Sarmiento, Santiago Sánchez-Solano

ABSTRACT / *RESUMEN*

This paper studies the use of NEON instructions for the implementation of elliptic curve cryptographic primitives on ARM Cortex-A processors. Starting from the analysis of point arithmetic formulas in different coordinate systems it was possible to identify several operations with no data dependency. Then, these operations were conveniently grouped in pairs to perform them in parallel using the NEON engine. Following this approach, dual NEON-based multiplications and squarings in the finite field $\mathbb{F}_p$ are proposed. Furthermore, these dual $\mathbb{F}_p$ operations are also used to speed up multiplications and squarings over the field extension $\mathbb{F}_{p^2}$. Finally, after integrating them into the point addition and point doubling formulas, we measure their impact on the execution time of scalar multiplications on elliptic curves defined over both finite fields. By using a mixed C/NEON implementation approach our solution is easily scalable at run time to support different curve sizes. Experiments conducted on the ARM Cortex-A9 processing system embedded in the Xilinx XC7Z020 device reported performance improvements of the NEON-based scalar multiplication between 32% and 38% and between 9% and 34% compared to a conventional implementation of the same operation on 254-bit, 384-bit and 510-bit curves over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ respectively.

Keywords: elliptic curve cryptography, scalar point multiplication, ARM Cortex-A processors, NEON instruction set.

Este trabajo estudia el empleo del repertorio de instrucciones NEON para la implementación de primitivas criptográficas de curvas elípticas sobre procesadores ARM Cortex-A. Realizando un análisis de las ecuaciones para la aritmética de puntos en diferentes sistemas de coordenadas fue posible identificar varias operaciones sin dependencia de datos entre ellas. De esta manera, dichas operaciones fueron agrupadas en pares para ser ejecutadas simultáneamente utilizando el coprocesador NEON. Siguiendo este enfoque se implementan operaciones de doble multiplicación y doble cuadrado en el campo finito $\mathbb{F}_p$. Adicionalmente, estas operaciones dobles en $\mathbb{F}_p$ son empleadas para acelerar las operaciones de multiplicación y cuadrado sobre la extensión de campo $\mathbb{F}_{p^2}$. Finalmente, al integrar todas estas operaciones dentro de los procedimientos para suma y doblado de puntos, se mide el impacto de las mismas en el rendimiento de la multiplicación escalar en curvas elípticas definidas sobre ambos campos finitos. Gracias a una implementación mixta empleando C y NEON nuestra solución es fácilmente escalable en tiempo de ejecución para brindar soporte a varios tamaños de curva. Los experimentos realizados en el sistema de procesamiento ARM Cortex-A9 empotrado en el dispositivo XC7Z020 de Xilinx reportaron mejoras de rendimiento entre un 32% y un 38% y entre un 9% y un 34% para una multiplicación escalar basada en NEON con respecto a una implementación convencional de dicha operación en curvas de 254 , 384 y 510 bits sobre $\mathbb{F}_p$ y $\mathbb{F}_{p^2}$ respectivamente.

Palabras claves: criptografía de curvas elípticas, multiplicación escalar, ARM Cortex-A, NEON.

Aceleración de la aritmética de curvas elípticas en procesadores ARM utilizando instrucciones NEON

1. -INTRODUCTION

The use of elliptic curves in cryptography was proposed independently by Miller [1] and Koblitz [2] when they discovered that the set of points satisfying the curve equation together with point addition as group law form a suitable group to build

Recibido: 17/2/2020 Aceptado: 4/8/2020

discrete logarithm systems. Since then, many protocols based on elliptic curves have been developed and included into several standards [3-6]. From an implementation perspective, the main advantage of elliptic curve cryptography relays on its relatively small key-length requirement compared to that of systems based on integer factorization or discrete logarithms in the multiplicative group of finite fields. Shorter keys translate into lower storage requirements and smaller computation times, two great features in general, and specially for embedded platforms where memory space and processing capabilities are usually constrained.

Scalar multiplication is the most important operation in elliptic curve protocols. For this reason, numerous mechanisms aimed to improve the performance of this operation have been proposed. Some of them intend to reduce the number of point addition and point doubling operations required to compute a scalar multiplication. Other mechanisms explore different elliptic curve point representations leading to efficient addition and doubling formulas; while a third group is focused on minimizing the computational cost of the underlying finite field arithmetic [7]. These alternatives are not mutually exclusive, in fact, in most practical cases they are used combined with each other in order to obtain better results. Whatever the case, a common way to boost up performance is to complement those algorithmic improvements with the proper use of any specific feature of the selected implementation platform providing processing acceleration. In this sense, the ARM Cortex-A family of processors comes equipped with NEON, a Single Instruction Multiple Data (SIMD) extension that can be exploited to speed up elliptic curve arithmetic on ARM-powered devices. In particular, this work focuses on using NEON instructions to accelerate operations in the underlying finite fields on top of which elliptic curves are built.

Different approaches can be followed to implement finite field arithmetic using NEON. A first option is to parallelize computations within a single field operation. Inconveniences arise with this alternative when using operands represented in the conventional non-redundant (full-radix) form since most SIMD architectures, including NEON, do not support carry propagation across data items that are processed in parallel. Accordingly, several implementations adopt the reduced-radix (redundant) representation suggested in [8] to ease the handling of carry propagation. However, as stated in [9], such approach leads to more intermediate partial products being computed. For that reason and although there are clever proposals [9,10] achieving parallelization within a single field operation involving full-radix operands, this work explores a second way of using the NEON engine. It consists of performing two field operations in parallel as described for the attribute-based encryption scheme implemented in [11]. Following such approach, the non-redundant representation does not suffer from carry propagation issues. That is, a dual multi-precision finite field operation can be split into a sequence of consecutive dual single-precision computations intended to be executed in separate iterations. Thus, carry values can be passed from one iteration to another until the entire multi-precision computation finishes.

In this paper we construct NEON-based $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ finite field multiplication and squaring operations with the final objective of speeding up elliptic curve arithmetic. We begin with the identification of those field multiplications and squarings than can be parallelized within the point addition and doubling formulas in different coordinate systems. Next, we placed our NEON-based variants of these operations into the elliptic curve point arithmetic. As result, we observed performance improvements between 32% and 38% (over $\mathbb{F}_p$) and between 9% and 34% (over $\mathbb{F}_{p^2}$) for the scalar multiplication primitive on 254-bit, 384-bit and 510-bit curves.

The rest of this paper is organized as follows: Section 2 briefly discusses some previous results related to the subject presented in this work. In Section 3 an overview on elliptic curve arithmetic is given. Section 4 presents our implementation of NEON-based field operations as well as their application into the elliptic curve arithmetic. Section 5 shows the timing results obtained from the experiments conducted on the ARM Cortex-A9 processing system embedded in the Xilinx XC7Z020 device. Finally, concluding remarks are provided in Section 6.

2.- RELATED WORK

Several researches targeting SIMD-based implementations of cryptographic primitives have been reported in the literature. In particular, the use of NEON vectorization has been proposed by [12-15] to speed up elliptic curve arithmetic. In [11] the authors also proposed the use of NEON in the context of an attribute-based encryption scheme which exploits the computation of bilinear pairings on elliptic curves. Other works like [9,10] used NEON instructions to implement modular multiplication and modular squaring primitives which are common to several cryptographic schemes including elliptic curves. The particular scenarios targeted by these works are quite diverse. For example, in [12] a reduced-radix representation is used to perform NEON-based multiplications to accelerate Curve25519 and Ed25519 curve arithmetic. NEON vectorization was applied across two independent multiplications inside point arithmetic formulas as well as within a single multiplication for those that could not be paired. The authors of [13] implemented a GLV-based scalar multiplication for the Ted127-glv4 curve in which interleaved ARM-NEON instructions were used to perform independent 128-bit multiplications in parallel. The application of NEON vectorization to boost up the computational performance of elliptic curves defined over binary fields has been

studied in [14]. Specifically, NEON was used to accelerate the polynomial multiplication of two vectors of eight 8-bit polynomials producing 128-bit products. This primitive was then used in point arithmetic on random and Koblitz binary curves. In [15], interleaved ARM-NEON instructions were employed to speed up multiplications over $\mathbb{F}_{p^2}$ with the final goal of accelerating a 4-dimensional scalar multiplication on the Four$\mathbb{Q}$ twisted Edwards curve. In the case of [11], the authors showed a way to perform two simultaneous modular multiplications using NEON vectorization to accelerate the computation of the optimal Ate pairing over a 254-bit Barreto-Naehrig curve used in the context of attribute-based encryption.

Researches summarized above exemplify the successful application of SIMD techniques on ARM processors to speed up cryptography. Along with NEON most of them employed algorithmic optimizations applicable to their particular scenarios and, from an implementation perspective, all of them targeted specific-length implementations optimizing code for a particular bit-length. That is the main difference compared to our proposal. Our intention is to exploit NEON vectorization while keeping the implementation flexible enough to allow scalability. Thus, the library implemented in this work is able to switch at run time between curves of the same family but of different sizes while keeping the same NEON-based processing core.

3.- ELLIPTIC CURVE ARITHMETIC

Let $\mathbb{F}_{p^k}$ be a finite field with a prime characteristic p different from 2 and 3, and integer $k > 0$. An elliptic curve E over $\mathbb{F}_{p^k}$ can be defined by the simplified Weierstrass equation $E(\mathbb{F}_{p^k})\colon y^2 = x^3 + ax + b$ where curve coefficients $a, b \in \mathbb{F}_{p^k}$ must satisfy the inequality $4a^3 + 27b^2 \neq 0$ (see [7] for further details).

The most important operation used in elliptic curve protocols is the scalar point multiplication. It is denoted by $[s]P$, where s is a positive integer and P is a point in $E(\mathbb{F}_{p^k})$. A scalar multiplication can be interpreted as the addition of the point P by itself s times which leads to a point Q also in $E(\mathbb{F}_{p^k})$. Computing scalar multiplications from this naive approach is extremely inefficient. In practice, algorithms exploit some special representation of s in order to reduce the required number of point additions. The most common approaches are derived from the binary representation of s. Algorithm 1 [7], for example, outlines the left-to-right strategy in which s (with $s_{n-1} = 1$) is scanned bit-by-bit from the left while Q, firstly initialized to P, is doubled at each iteration. If the corresponding bit of s is set, the point Q is additionally updated by adding P. Once all bits of s are exhausted Q will hold the result $[s]P$. Point doubling (i.e., $P + P = [2]P$) is distinguished from point addition since doubling formulas are usually more efficient in terms of storage requirements, computing time or both.

Algorithm 1
Left-to-right scalar multiplication

INPUT: $s = (s_{n-1}, \cdots, s_1, s_0)_2$, $P \in E(\mathbb{F}_{p^k})$.

OUTPUT: $Q = [s]P$.

1. $Q = P$;
2. **for** $i = n - 2$ **to** 0 **do**
3. $\quad Q = [2]Q$;
4. $\quad$ **if** $s_i == 1$ **then**
5. $\quad\quad Q = Q + P$;
6. $\quad$ **end**
7. **end**
8. Return Q;

Addition and doubling formulas allowing to compute $P_3 = P_1 + P_2$ and $P_3 = [2]P_1$ are given by equations (1) and (2) respectively, where $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3)$ are points in $E(\mathbb{F}_{p^k})$ [7].

$$\begin{aligned} x_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \\ y_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \end{aligned} \tag{1}$$

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1$$
$$y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \quad (2)$$

Operations involved in the above equations are defined in the underlying finite field $\mathbb{F}_{p^k}$. Denoting a field inversion, a field multiplication and a field squaring by **I**, **M** and **S** respectively, it is easy to see in equation (1) that a point addition costs 1**I**+2**M**+1**S**, and in equation (2) that a point doubling requires 1**I**+2**M**+2**S**. Throughout this work we do not consider the contribution of field additions, subtractions and multiplications by small constants because their impact is negligible when compared to that of **I**, **M** and **S**. Then, the performance of the elliptic curve arithmetic will heavily depend on how efficiently we are able to perform field inversions, multiplications and squarings. Improvements made down into these field operations should be reflected up into the high-level elliptic curve arithmetic.

3.1.- PROJECTIVE COORDINATES

Equations (1) and (2) are said to be the affine equations for point addition and point doubling respectively since they act on points represented in affine coordinates. In most cases field inversion is the most expensive operation. For example, in the base field $\mathbb{F}_p$ (i.e., taking $k = 1$) inversions can be computed using the Itoh-Tsujii method [16] or the binary Extended Euclidean Algorithm [17] while best choices for multiplications are derived from Montgomery's method [18]. In this setup, the inversion to multiplication (**I/M**) ratio is considered, at best, not less than 8 [19]. Some practical implementations report **I/M**-ratios between 13 and 35 [20,21].

Field inversions can be avoided at the cost of extra field multiplications by employing projective coordinates. As long as the total multiplications count does not exceed the inversion to multiplication ratio, projective equations will perform better than affine formulas. A point in projective coordinates is represented by a triple (X, Y, Z) where $(X, Y, Z) \in \mathbb{F}_{p^k}$ and $Z \neq 0$. The map $(X, Y, Z) \mapsto (XZ^{-c}, YZ^{-d})$ allows to move a projective point to its affine representation. Different values of c and d correspond to different projective coordinate systems. Accordingly, when $c = d = 1$ we are in presence of *standard projective coordinates*, while values of $c = 2$ and $d = 3$ define *Jacobian coordinates* [7,22].

Inversion-free point addition and doubling formulas are derived from affine equations by simple substitutions of $x_i = X_i Z_i^{-1}$ and $y_i = Y_i Z_i^{-1}$ for standard coordinates and $x_i = X_i Z_i^{-2}$ and $y_i = Y_i Z_i^{-3}$ for Jacobian coordinates. In the case of point addition not always both points must be in their projective form. Note that the point P remains unchanged during the entire run of Algorithm 1. This suggests that P can be treated as an affine point (i.e., $Z_P = 1$) which slightly reduces the complexity of point additions. In the case of standard coordinates we can use the equation (3) to compute the point addition $P_3 = P_1 + P_2$ at a cost of 9**M**+2**S**, where $P_3 = (X_3, Y_3, Z_3)$, $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$ with $Z_2 = 1$. Similar equations can be obtained either for point doubling in standard or point doubling and point addition in Jacobian coordinates [20].

$$X_3 = (X_1 - X_2 Z_1)[Z_1(Y_1 - Y_2 Z_1)^2 - (X_1 - X_2 Z_1)^2(X_1 + X_2 Z_1)]$$
$$Y_3 = (Y_1 - Y_2 Z_1)\left[X_2 Z_1 (X_1 - X_2 Z_1)^2 - [Z_1(Y_1 - Y_2 Z_1)^2 - (X_1 - X_2 Z_1)^2(X_1 + X_2 Z_1)]\right] - Y_2 Z_1 (X_1 - X_2 Z_1)^3 \quad (3)$$
$$Z_3 = Z_1(X_1 - X_2 Z_1)^3$$

Note that expressions for X_3, Y_3 and Z_3 in equation (3) share several terms. Fast implementations of elliptic curve point arithmetic take advantage from this fact to save computation time at the expense of few extra memory locations to store reusable intermediate values. We refer the reader to the *Explicit-Formulas Database* web site [23] for a vast compendium on fast point arithmetic formulas.

Table 1 summarizes multiplication and squaring counts for the standard and Jacobian coordinate equations used in this work. Recall from Algorithm 1 that a point doubling always occurs at each iteration of the scalar multiplication loop while point additions are only performed when the corresponding bit of the scalar s is set. Since s is selected uniformly at random in most elliptic curve cryptographic protocols it is expected to have similar proportions of '1' and '0' in its binary representation. Hence, for n-bit parameters a scalar multiplication will cost $n - 1$ point doublings and about $(n - 1)/2$ point additions. Taking $n = 256$, for example, this implies an average cost of 2678**M**+1275**S** in standard and 1785**M**+1403**S** in Jacobian

coordinates. Note that multiplications prevail in both cases. Therefore, accelerating this operation would be a good starting point.

Table 1
Operation counts of point arithmetic

Coordinates	Point operation	
	Addition	Doubling
Standard	9**M**+2**S**	6**M**+4**S**
Jacobian	8**M**+3**S**	3**M**+4**S**

4.- NEON-BASED ELLIPTIC CURVE ARITHMETIC

One alternative to boost up the performance of field arithmetic is to take advantage of any specific feature available in the selected implementation platform. In this sense, our work focuses on devices populated with ARM processors provided with the NEON extended instruction set.

Typical sizes for the operands used in elliptic curve schemes are currently in the order of 256, 384 and 512 bits [24]. However, most ARM processors exhibit a 32-bit architecture [25]. Then, a 256x256 multiplication, for example, have to be split into several 32x32 multiplications and 32x32 additions that can be handled by conventional ARM instructions. However, the ARM Cortex-A series processors come equipped with NEON, a 128-bit Single Instruction Multiple Data extension. Thus, the use of NEON instructions could be very helpful to speed up the field operations underlying elliptic curve arithmetic. NEON engine is built around 16 registers of 128 bits (Q0 ~ Q15) that can be accessed as 32 registers of 64 bits (D0 ~ D31). Every register can hold a vector of n lanes with m bits each. Refer to [26] for allowed combinations of n and m.

Most NEON instructions act over n lanes in parallel. They perform the same operation between equivalent lanes on the input vectors and store the results in the corresponding lanes on the output vector. Nevertheless, not all instructions support all possible combinations of n and m. For example, the `vmull` instruction illustrated in Figure 1 does not support 64-bit lanes. Even when we can specify a 128-bit register as destination operand and two 64-bit registers as source operands, `vmull` is only allowed to process lanes of up to 32-bit. We must also point out that there is no support for carry propagation between lanes which is a great inconvenient unless a redundant numeric representation is used. Notwithstanding the above NEON provides a useful degree of flexibility by means of the so-called instruction *shapes*. Most NEON arithmetic instructions come in at least two of four different shapes: *normal*, *long*, *wide* and *narrow*. We only emphasize the *long* case due to its relevance for this work (see [26] for further details). Long-shape arithmetic instructions act on source vectors of n lanes of m bits each and produce an output vector of n lanes of $2m$ bits each. This fact is clear for the `vmull` example. Observe in Figure 1 that while input vectors have 2 lanes of 32 bits the result is a vector also with 2 lanes but of 64 bits each. This shows a way in which one can perform simultaneously two 32x32 multiplications obtaining two 64-bit products as result.

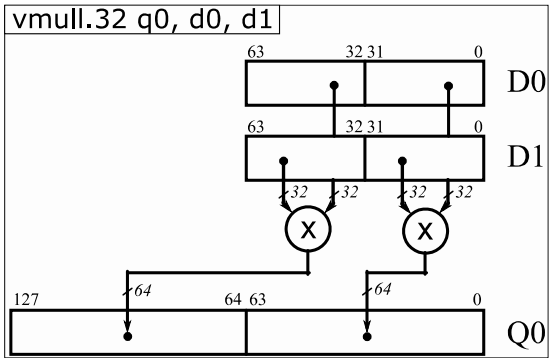

Figure 1

NEON Long-shape instruction example.

It would be logical to think that the use of the `vmull` instruction would double the speed of a 256x256 multiplication since we would be able to perform two 32x32 multiplications at once. However, partial products must be accumulated to obtain the final result which would require carry values propagating across lanes, feature not supported by NEON. Alternatively, we can compute only partial products in parallel and later perform carry propagation sequentially but this would incur in extra time penalties mitigating any benefit provided by the parallel multiplication step. A different approach is not to parallelize a single 256x256 multiplication but to perform two independents 256x256 multiplications simultaneously. The rest of this section discusses in details how to use such approach in the context of elliptic curves. One of our design goals is to build an implementation flexible enough such that it enables us to switch at run time between different bit-lengths. For this purpose, rather than constructing a fixed-length implementation fully coded with NEON instructions, we propose the implementation of some NEON-based kernels that are properly inserted into C code to achieve a balance between speed and flexibility.

4.1.- POINT ARITHMETIC OVER $\mathbb{F}_p$ USING NEON

As suggested above, anywhere there exist two field multiplications acting on independent data, it is possible to parallelize them by means of NEON instructions. The same principle applies for field squarings. Fortunately, point arithmetic equations, especially in projective coordinates, can be conveniently arranged to extract many independent field operations. In this sense, we propose Algorithm 2 in which operations have been carefully scheduled to break data dependency and group into one pair the two squarings and into others four pairs eight of the nine field multiplications required to perform point addition in standard coordinates. Multiplications and squarings inside each pair can now be computed in parallel assuming we are provided with the proper functions to do it.

Algorithm 2
Point addition in standard coordinates with independent field multiplications and squarings grouped in pairs

INPUT: $P_1 = (X_1, Y_1, Z_1)$, $P_2 = (X_2, Y_2, Z_2)$ in standard coordinates with $Z_2 = 1$.
OUTPUT: $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$.

1. $W = Y_2 \cdot Z_1$; $A = Y_2 \cdot Z_1$;	**8.** $T_1 = T_1 - T_2$;
2. $V_1 = X_1 - W$;	**9.** $T_1 = T_1 - T_3$
3. $U_1 = Y_1 - A$;	**10.** $W = W - T_1$
4. $T_2 = V_1^2$;$T_1 = U_1^2$;	**11.** $Y_3 = U_1 \cdot W$; $A = A \cdot T_2$;
5. $T_1 = T_1 \cdot Z_1$; $W = W \cdot T_2$;	**12.** $X_3 = V_1 \cdot T_1$; $Z_3 = T_2 \cdot Z_1$;
6. $T_3 = 2W$;	**13.** $Y_3 = Y_3 - A$;
7. $T_2 = V_1 \cdot T_2$;	**14.** Return (X_3, Y_3, Z_3);

Hereafter we generically refer to the parallel computation of two field multiplications as a *dual field multiplication*. When a concrete finite field needs to be specified, then the word *field* will be substituted by the notation used to identify that particular finite field. That is, a dual $\mathbb{F}_p$ multiplication refers to a dual field multiplication in the base field $\mathbb{F}_p$. The same reasoning applies for squarings. Let us now denote a dual field multiplication by **Md** and a dual field squaring by **Sd**. The running time of Algorithm 2 will be dominated by the cost of 4**Md**+1**Sd**+1**M** which should be better than 9**M**+2**S** (see Table 1) as long as we get **Md** < 2**M** and **Sd** < 2**S**.

Although we only exemplify the parallelization opportunities of point addition in standard coordinates, it is worthwhile to mention that similar analyses were conducted for the remaining cases. As a result, we built procedures for point doubling in standard coordinates as well as for point addition and point doubling in Jacobian coordinates with computational costs of 3**Md**+1**Sd**+2**M**, 4**Md**+1**Sd**+1**S** and 1**Md**+2**Sd**+1**M** respectively. This emphasizes the need to build functions to compute dual field multiplications and squarings. We now continue with the implementation of such functions by using NEON instructions.

4.1.1.- DUAL FIELD MULTIPLICATION IN $\mathbb{F}_p$

Best choices to perform multiplications in $\mathbb{F}_p$ are derived from Montgomery's method. In this work we use the Separated Operand Scanning (SOS) algorithm proposed in [27]. This algorithm proceeds by performing a multi-precision multiplication followed by a Montgomery reduction.

Multi-precision integer multiplications are commonly computed through the well-known schoolbook method [28]. The inputs of this method consist of two l-length arrays `a[]` and `b[]` holding the coefficients of the representation in base 2^w of the n-bit

integers a and b, with $l = \lceil n/w \rceil$. The value of w is usually selected to match the word size of the target processor. The product $t = a \cdot b$ is stored in the $2l$-length array t[] which should have all positions firstly initialized to zero. Two nested loops, running from i=0 to l-1 the outer and from j=0 to l-1 the inner, gradually fill t[] with the result. To achieve this, the multiply-and-accumulate operation shown in equation (4) is executed at each iteration of the inner loop.

$$(C, t[i+j]) = t[i+j] + a[j] \cdot b[i] + C \tag{4}$$

The first step towards a dual $\mathbb{F}_p$ multiplication is to perform two multi-precision multiplications in parallel. This is where NEON technology comes into action. Let have arrays x[], y[] and u[] holding the additional multiplicands and product respectively. Using the `vpaddl` and `vmlal` instructions we coded an assembly language subroutine called *`neon_dual_mac2`* which is able to compute two simultaneous multiply-and-accumulate operations. This routine takes parameters pt_{ij}, pu_{ij}, a_j, x_j, b_i and y_i. The first two are pointers to the location i+j of both t[] and u[] respectively. Thus, we can access the values at these locations, process them and write back the corresponding results. The remaining parameters are just the values a[j], x[j], b[i] and y[i] to be multiplied at the current iteration.

Figure 2 graphically illustrates the arithmetic kernel of *`neon_dual_mac2`*. Register Q0 holds a 4x32-bit vector with the values t_{ij} and u_{ij} pointed by pt_{ij} and pu_{ij} respectively, and carry values C_0 and C_1 that are initially loaded with zero. Then, the pairwise long-shape addition `vpaddl.u32 q0, q0` computes in parallel values $C_0 + t_{ij}$ and $C_1 + u_{ij}$ storing the results also in Q0 but now as a 2x64-bit vector. Register Q1 is accessed through its two separated 64-bit D-registers D2 and D3. Register D2 allocates a 2x32-bit vector holding the multiplicand words a_j and x_j while register D3 is loaded with b_i and y_i. Finally, the long-shape multiply-and-accumulate instruction `vmlal.u32 q0, d2, d3` computes at the same time the value $(C_0', t_{ij}') = t_{ij} + a_j \cdot b_i + C_0$ and the value $(C_1', u_{ij}') = u_{ij} + x_j \cdot y_i + C_1$. Both of these results are also stored in the Q0 register viewed as a 2x64-bit vector. The computed values t_{ij}' and u_{ij}' are sent back from the lower half of both 64-bit lanes to the memory locations pointed by pt_{ij} and pu_{ij} respectively. Carry values C_0' and C_1' are left inside Q0 ready for a subsequent call to *`neon_dual_mac2`* at the next iteration of the dual multi-precision multiplication procedure.

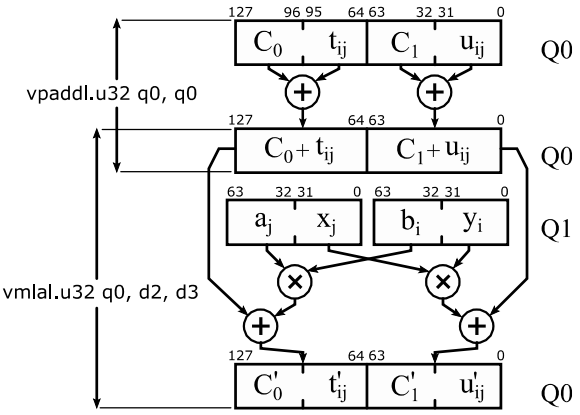

Figure 2

Kernel of *`neon_dual_mac2`*.

Load and store instructions allowing to move data between ARM memory and NEON vectors were omitted from Figure 2 to keep the diagram as simple as possible. However, it is worthwhile to mention that the time required for these data transfers to take place obviously contributes to the total execution time of the dual multi-precision multiplication.

The second part of the SOS modular multiplication algorithm consists in the Montgomery reduction phase. A single Montgomery reduction takes as inputs an l-length array p[] containing the n-bit prime modulus p together with an $2l$-length array t[] holding the product $t = a \cdot b$ from a previous multi-precision multiplication step. Once the computation finishes the result $c = t \cdot R^{-1} \bmod p$ is stored in the l-length output array c[]. Here $l = \lceil n/w \rceil$ as defined previously. In addition, Montgomery reduction requires precomputed parameters $R = 2^N \bmod p$, where $N = l \cdot w$ and $n_0' = -p_0^{-1} \bmod R$. From the implementation point of view, a multi-precision Montgomery reduction is very similar to a multi-precision multiplication

since it is also based on a nested loops structure. Furthermore, at the heart of the inner loop we also find a multiply-and-accumulate operation similar to that of a multi-precision multiplication. Therefore, when turning a Montgomery reduction into its dual variant, it is possible to reutilize the *neon_dual_mac2* routine discussed above. However, in this case there are two additional operations that have to be performed using the NEON engine: a dual single-precision multiplication and a dual carry propagation. For this purpose, we coded two new subroutines called *neon_dual_mul* and *neon_dual_carry* respectively. The subroutine *neon_dual_mul* takes a pointer pq_i and words t_i, u_i and n'_0 as input parameters. As shown in Figure 3, only the registers D0 and D1 and a normal-shape `vmul` instruction are involved. Register D0 is used as a 2x32-bit vector whose lanes are loaded with the values t_i and u_i. Then, both lanes are multiplied by the parameter n'_0 contained into the lower half of D1. The resulting product $q_i=t_i \cdot n'_0$ is stored into the memory location pointed by pq_i while $r_i=u_i \cdot n'_0$ is sent back as the return value of the subroutine. This different treatment in the way of returning the results allows us to save a data transfer between ARM system memory and NEON registers.

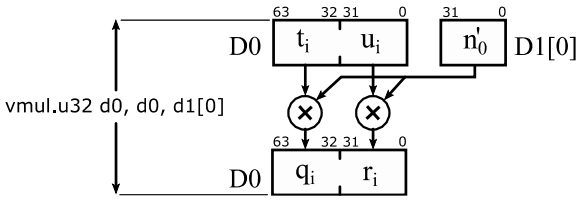

Figure 3

Kernel of *neon_dual_mul*.

Figure 4 shows the *neon_dual_carry* functionality. The input parameters are the pointers pt and pu pointing to the position of arrays t[] and u[] respectively from which it is desired to start the propagation of the corresponding carry values. The length l indicating the end of carry propagation is also given as input. Observe that the procedure only involves the Q0 register and the long-shape `vpaddl` instruction used both in the same way they were used at the beginning of *neon_dual_mac2*. At each iteration the current coefficients t_i and u_i are accessed through pt and pu and loaded into Q0 at the positions shown in Figure 4. Then, they are both added with the corresponding carry values and the results are stored into Q0 again. The updated coefficients t'_i and u'_i are sent back through pt and pu to the corresponding locations on arrays t[] and u[] respectively. The new carry values C'_0 and C'_1 are left in Q0 ready for the next iteration. This process ends when the iteration count matches length l.

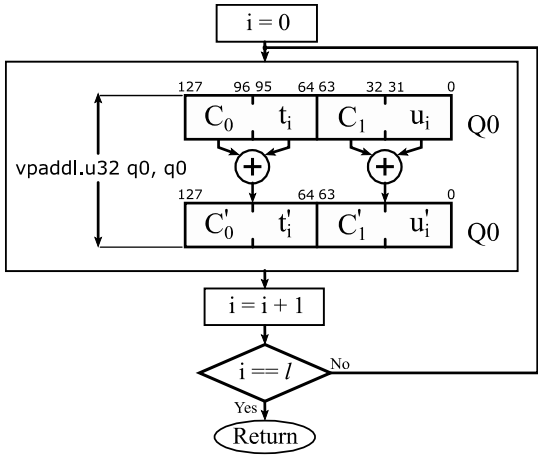

Figure 4

Kernel of *neon_dual_carry*.

Algorithm 3 shows how the above pieces are tied together to conform a mixed C/NEON dual SOS Montgomery modular multiplication. This mixed approach allowed us to build a flexible and scalable solution that supports different field sizes at run time. Step 7 performs the dual multi-precision multiplication phase while step 12 computes the dual Montgomery

reduction. Note that a correction is applied to the values resulting from Montgomery reduction if they are greater or equal to the modulus. Also note that this final correction step is not parallelized since it turns out that not always both values need to be corrected at the same time. Even though comparisons t[]≥p[] and u[]≥p[] always take place and so at least them could be performed in parallel, we experimentally found that there were no speed improvements in doing so due to the time penalties caused by the required extra data transfers between the ARM system memory and NEON registers.

Algorithm 3
Dual SOS modular multiplication procedure

INPUT: l-length arrays a[], b[], x[], y[] and p[]; parameter n'_0.

OUTPUT: l-length arrays c[] and z[].

/* Input arrays a[], b[], x[], y[] and p[] should hold the coefficients of the radix-2^w representation of operands a, b, x, y and modulus p respectively. Once computation finishes, output arrays c[] and z[] will hold the coefficients of products $c = abR^{-1} \bmod p$ and $x = xyR^{-1} \bmod p$ in radix-2^w representation.*/

```
1.  Define t[], u[]; /* Arrays t[] and u[] must be of length 2l to hold the radix-2^w coefficients of the intermediate
    multi-precision products t = a · b and u = x · y respectively. */
2.  Define q, r;
3.  for i = 0 to 2l − 1 do // Required before multi-precision multiplication
4.      t[i] = 0;
5.      u[i] = 0;
6.  end
7.  for i = 0 to l − 1 do // Dual multi-precision multiplication
8.      for j = 0 to l − 1 do
9.          neon_dual_mac2(&t[i+j], &u[i+j], a[j], x[j], b[i], y[i]);
10.     end
11. end
12. for i = 0 to l − 1 do // Dual multi-precision Montgomery reduction
13.     r =neon_dual_mul(n'0, &q, t[i], u[i]);
14.     for j = 0 to l − 1 do
15.         neon_dual_mac2(&t[i+j], &u[i+j], q, r, p[j], p[j]);
16.     end
17.     neon_dual_carry(&t[i+j], &u[i+j], l);
18. end
19. for j = 0 to l − 1 do
20.     c[j]=t[j+l];
21.     z[j]=u[j+l];
22. end
/* Verifying if final corrections are required. */
23. if t[]≥p[] then // Multi-precision comparison between the higher half of array t[] and the modulus p[]
24.     for j = 0 to l − 1 do // Final correction required
25.         c[j]=c[j]-p[j];
26.     end
27. end
28. if u[]≥p[] then // Multi-precision comparison between the higher half of array u[] and the modulus p[]
29.     for j = 0 to l − 1 do // Final correction required
30.         z[j]=z[j]-p[j];
31.     end
32. end
```

4.1.2.- DUAL FIELD SQUARING IN $\mathbb{F}_p$

Multi-precision squarings are usually more efficient than generic multi-precision multiplications of an integer a by itself because they take advantage from the fact that intermediate products $a_j \cdot a_i$ and $a_i \cdot a_j$ are the same. Hence, almost half of the single-precision multiplications can be skipped [29]. According to this, the **S/M**-ratio is expected to be less than 1. Presumably, dual multi-precision squarings should exhibit a similar behavior. Then, it should be better to perform dual multi-precision squarings instead of dual multi-precision multiplications whenever possible. To accomplish this, Algorithm 3 can be turned into dual field squaring by simply substituting the multi-precision multiplication at step 7 by a multi-precision squaring. The reduction phase keeps the same.

The dual multi-precision squaring implementation is also based on a nested loops structure in which two simultaneous multiply-and-accumulate operations need to be performed. However, as shown in equation (5), an extra multiplication by 2, that was not present in the case of multi-precision multiplication, is now required. This apparently simple change actually forces us to use a few extra operations to handle arising overflows which prevent us from reusing the `neon_dual_mac2` routine. In addition, when $i = j$ (i.e., $a_j = a_i$), which happens at every new iteration of the outer loop, a different and simpler treatment is required. For those reasons, we defined two new subroutines: `neon_dual_sqr_mac2` to handle intermediate products of the type $2 \cdot a_j \cdot a_i$ and `neon_dual_mac` which takes care of the $a_i \cdot a_i$ cases.

$$(C, t[i+j]) = t[i+j] + 2 \cdot a[j] \cdot a[i] + C \tag{5}$$

The subroutine `neon_dual_sqr_mac2` takes the same input parameters as those of `neon_dual_mac2`. However, the operands to be pairwise multiplied are not located in different arrays. They come from different locations of the same array a[] in the case of operands a_j and a_i, as well as x_j and x_i which correspond to different locations of the array x[]. Figure 5 shows the functional diagram of `neon_dual_sqr_mac2`. Instructions 1, 2 and 5 perform the actual dual multiply-and-accumulate operation. Note the similarity between them and the kernel of `neon_dual_mac2` depicted in Figure 2. The left shifting at instruction 4 corresponds to the multiplication by 2 while instructions 3, 6 and 7 are those that handle the overflows. In particular, instruction 3 saves the most significant bit of both intermediate products $a_j \cdot a_i$ and $x_j \cdot x_i$ before they are shifted out. Later, instructions 6 and 7 combine these bits with the updated carry values C'_0 and C'_1 putting the results in the register Q0. These combined values are the input carry values of the next iteration. Finally, the outputs t'_{ij} and u'_{ij} resulting from instruction 5 are sent back to the correct memory locations of arrays t[] and u[] respectively.

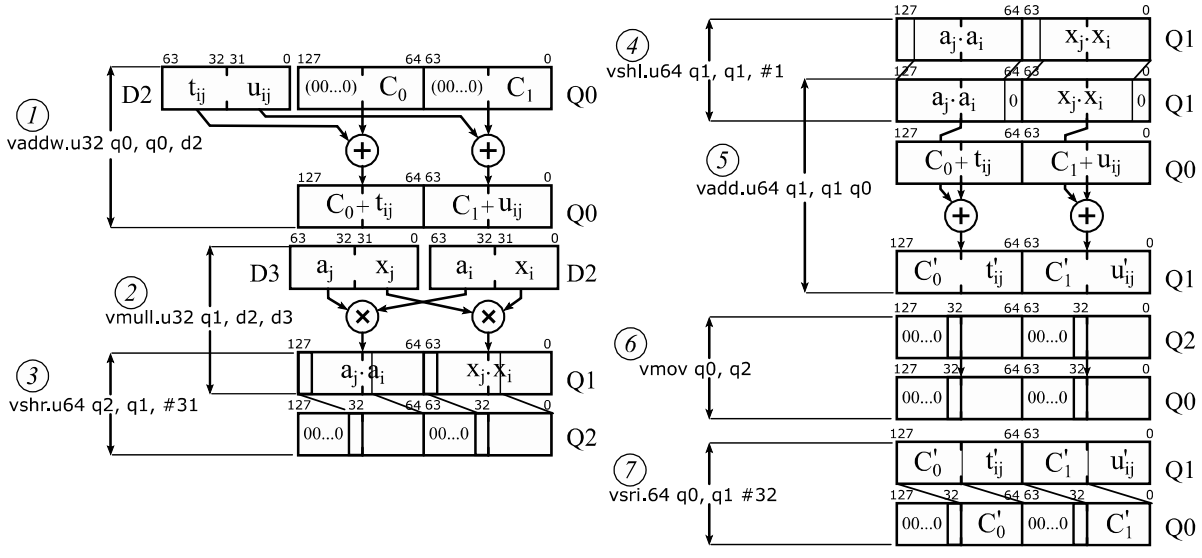

Figure 5

Kernel of `neon_dual_sqr_mac2`.

The subroutine `neon_dual_mac` is very much simpler. It is actually a reduced version of `neon_dual_mac2` since it does not handle any input carry values. This can be easily appreciated in Figure 6. Note that `neon_dual_mac` only uses a single `vmlal` instruction to compute the multiply-and-accumulate operations $(C_0',t_{ii}')=t_{ii}+a_i \cdot a_i$ and the value $(C_1',u_{ii}')=u_{ii}+x_i \cdot x_i$. The right shift instruction used next is only intended to get the register Q0 ready for a following and immediate call to the subroutine `neon_dual_sqr_mac2`. Obviously, before the values t_{ii}' and u_{ii}' get lost because of the shifting instruction they are sent back to their corresponding locations inside arrays t[] and u[].

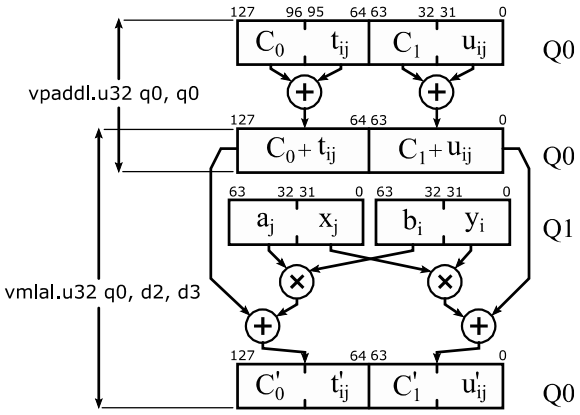

Figure 6

Kernel of *neon_dual_mac*.

Replacing the multi-precision multiplication loop at step 7 of Algorithm 3 by the squaring procedure shown in Algorithm 4 turns the dual SOS modular multiplication into a dual SOS modular squaring operation. Although it is evident that `neon_dual_sqr_mac2` is much more complex than `neon_dual_mac2`, the shorter inner loops involved in the squaring algorithm ensure the expected improvement in terms of speed.

Algorithm 4
Dual multi-precision squaring loop

INPUT: l-length arrays a[] and x[].

OUTPUT: $2l$-length arrays t[] and u[].

/* Input arrays a[] and x[] should hold the coefficients of the radix-2^w representation of operands a and x respectively. Once computation finishes, output arrays t[] and u[] will hold the coefficients of $t = a^2$ and $u = x^2$ in radix-2^w representation. */

1. **for** $i = 0$ **to** $l - 1$ **do**
2. `neon_dual_mac`(&t[i+i], &u[i+i], a[i], x[i]);
3. **for** $j = i + 1$ **to** $l - 1$ **do**
4. `neon_dual_sqr_mac2`(&t[i+j], &u[i+j], a[j], x[j], a[i], x[i]);
5. **end**
6. **end**

Before closing this section, it is worthwhile to point out a situation that sometimes arises in the point arithmetic formulas. Until now we have only considered to pair multiplications and squarings separately. This required us to group in pairs either multiplications or squarings to later perform them simultaneously. Such a thing is sometimes just impossible. However, it could be the case that one unpaired multiplication can be performed in parallel with an unpaired squaring. In such situation it is advantageous to consider a squaring as a simple multiplication and then execute it as part of a dual NEON-based multiplication.

4.2.- APPLYING DUAL OPERATIONS IN POINT ARITHMETIC OVER $\mathbb{F}_{p^2}$

We found interesting to devote this section to evaluate the impact of dual NEON-based operations on the performance of elliptic curve arithmetic over $\mathbb{F}_{p^2}$. Elliptic curves over $\mathbb{F}_{p^2}$ are widely used in pairing-based cryptography. For example, when computing the optimal Ate pairing over Barreto-Naehrig (BN) pairing-friendly elliptic curves [30], most of the pairing computation is performed on the degree-2 field extension $\mathbb{F}_{p^2}$, since, even when BN-curves come equipped with an embedding degree $k = 12$ they admit a sextic twist allowing to move computation from $\mathbb{F}_{p^{12}}$ to $\mathbb{F}_{p^2}$ [31,32].

Point addition and doubling formulas in $\mathbb{F}_{p^2}$ are the same as those for $\mathbb{F}_p$. Indeed, recall from Section 3 that elliptic curves were defined over a generic finite field $\mathbb{F}_{p^k}$. Precisely, it is the underlying field what changes now. When $k = 1$ field elements were integers modulo p and field operations were those defined in modular arithmetic. In particular, we paid attention to modular multiplication and modular squaring. Field elements in $\mathbb{F}_{p^2}$ are no longer integers, they are degree-1 polynomials (binomials) with coefficients in $\mathbb{F}_p$. Consequently, field operations requiring our attention now are polynomial multiplication and squaring modulo an irreducible degree-2 polynomial [33].

4.2.1.- NEON-BASED $\mathbb{F}_{p^2}$ MULTIPLICATION

Dual NEON operations cannot be directly applied to point arithmetic in $\mathbb{F}_{p^2}$ to perform, for example, two simultaneous polynomial multiplications. Instead, NEON can be used to parallelize those $\mathbb{F}_p$ operations involved inside $\mathbb{F}_{p^2}$ polynomial arithmetic. In this work $\mathbb{F}_{p^2}$ is built on top $\mathbb{F}_p$ such that $\mathbb{F}_{p^2} = \mathbb{F}_p[\mu]/(\mu^2 + 1)$. The choice of $\mu^2 + 1$ as irreducible polynomial is suggested in several pairing-related researches since it makes it possible to obtain field multiplication and squaring procedures that are more efficient than a generic polynomial multiplication or squaring followed by a polynomial reduction. See [34] for a detailed description on the topic of field extensions construction.

Let $a = (a_1\mu + a_0)$ and $b = (b_1\mu + b_0)$ be two $\mathbb{F}_{p^2}$ elements with coefficients a_0, a_1, b_0 and b_1 in $\mathbb{F}_p$. Equation (6) shows the Karatsuba-Ofman multiplication for binomials [35] already combined with reduction modulo $\mu^2 + 1$ to compute the product $c = (c_1\mu + c_0)$. Note that $\mathbb{F}_p$ multiplications $(a_1 \cdot b_1)$ and $(a_0 \cdot b_0)$ act on independent data so they can be performed in parallel.

$$\begin{aligned} c_1 &= (a_0 + a_1) \cdot (b_0 + b_1) - [(a_1 \cdot b_1) + (a_0 \cdot b_0)] \\ c_0 &= (a_0 \cdot b_0) - (a_1 \cdot b_1) \end{aligned} \tag{6}$$

Algorithm 5 shows the resulting $\mathbb{F}_{p^2}$ multiplication procedure. The paired $\mathbb{F}_p$ multiplications at step 1 are meant to be executed in parallel by means of the dual NEON-based modular multiplication algorithm discussed in Section 4.1.1. Therefore, a multiplication in $\mathbb{F}_{p^2}$ will have a cost of 1**Md**+1**M**.

Algorithm 5
Karatsuba-Ofman field multiplication in $\mathbb{F}_{p^2} = \mathbb{F}_p[\mu]/(\mu^2 + 1)$

INPUT: $\mathbb{F}_{p^2}$ elements $a = (a_1\mu + a_0)$, $b = (b_1\mu + b_0)$.

OUTPUT: $c = (c_1\mu + c_0) = a \cdot b$

1. $\alpha_1 = a_1 \cdot b_1$; $\alpha_2 = a_0 \cdot b_0$; // Dual $\mathbb{F}_p$ multiplication
2. $\beta_1 = a_0 + a_1$;
3. $\beta_2 = b_0 + b_1$;
4. $\beta_1 = \beta_1 \cdot \beta_2$; // Single $\mathbb{F}_p$ multiplication
5. $\beta_1 = \beta_1 - a_1$;
6. $c_1 = \beta_1 - a_2$;
7. $c_0 = \alpha_2 - a_1$;

Further refinements can be achieved by using the lazy reduction technique [36]. Allowing intermediate values to grow above the modulus, the number of Montgomery reductions can be decreased to only one at the end of the multiplication procedure. We followed two approaches to implement lazy reduction. When possible we selected the prime characteristic p with a bit-

length n a few bits lower than N, where $N = l \cdot w$ as in Section 4.1.1. Thus, intermediate values can grow above the modulus but residing within the boundary of the l-length or $2l$-length arrays used store them. Alternatively, when we were forced to use a prime characteristic p, such that $n = N$, we added an extra w-bit location to store the extra bits resulting from multi-precision multiplications and additions without interleaved reduction steps. In this case the Montgomery reduction itself needed to be modified by including an additional iteration responsible for handling this additional word. Consequently, the parameter R has to be redefined as $R = 2^{(N+w)} \bmod p$. The penalty introduced by this extra iteration is widely mitigated by removing a full Montgomery reduction.

Algorithm 6 shows the procedure for multiplications in $\mathbb{F}_{p^2}$ using lazy reduction. Observe that intermediate multiplications are conventional integer multiplications instead of field multiplications in $\mathbb{F}_p$. Subtractions at steps 5, 6 and 7 are also ordinary multi-precision subtractions between integer numbers. Finally, observe that, as mentioned before, now it is only required a unique dual multi-precision Montgomery reduction. It is not difficult to see that this represents a time saving equivalent to a single multi-precision Montgomery reduction respect to Algorithm 5. Remember that multiplications in $\mathbb{F}_p$ involve a reduction step, then $\mathbb{F}_p$ multiplications at steps 1 and 4 of Algorithm 5 implicitly perform one dual and one single Montgomery reduction respectively. Precisely, the latter is the one that is saved.

Algorithm 6
Karatsuba-Ofman field multiplication in $\mathbb{F}_{p^2} = \mathbb{F}_p[\mu]/(\mu^2 + 1)$ using lazy reduction

INPUT: $\mathbb{F}_{p^2}$ elements $a = (a_1\mu + a_0)$, $b = (b_1\mu + b_0)$.

OUTPUT: $c = (c_1\mu + c_0) = a \cdot b$

1. $\alpha = a_0 + a_1$;
2. $\beta = b_0 + b_1$;
3. $\omega = \alpha \cdot \beta$; // Multi-precision integer multiplication.
4. $\alpha = a_1 \cdot b_1$; $\beta = a_0 \cdot b_0$; // Dual multi-precision integer multiplication.
5. $\omega = \omega - \alpha$;
6. $\omega = \omega - \beta$;
7. $\beta = \beta - \alpha$;
8. $c_1 = \omega R^{-1} \bmod p$; $c_0 = \beta R^{-1} \bmod p$; // Dual multi-precision Montgomery reduction.

4.2.2.- NEON-BASED $\mathbb{F}_{p^2}$ SQUARING

The Karatsuba-Ofman multiplication combined with reduction by $\mu^2 + 1$ can also be adapted to perform squarings in $\mathbb{F}_{p^2}$. In this case the resulting formulas are quite simple compared to that of multiplication. As can be appreciated in equation (7) there is no opportunity nor is it necessary to take advantage from the lazy reduction technique. However, as Algorithm 7 shows, multiplications $a_0 \cdot a_1$ and $(a_0 + a_1) \cdot (a_0 - a_1)$ can be executed in parallel with a dual NEON-based $\mathbb{F}_p$ multiplication.

$$\begin{aligned} c_1 &= 2 \cdot a_0 \cdot a_1 \\ c_0 &= (a_0 + a_1) \cdot (a_0 - a_1) \end{aligned} \tag{7}$$

Algorithm 7
Karatsuba-Ofman field squaring in $\mathbb{F}_{p^2} = \mathbb{F}_p[\mu]/(\mu^2 + 1)$

INPUT: $\mathbb{F}_{p^2}$ element $a = (a_1\mu + a_0)$.

OUTPUT: $c = (c_1\mu + c_0) = a^2$

1. $\alpha = a_0 + a_1$;
2. $\beta = a_0 - a_1$;
3. $c_1 = a_1 \cdot a_0$; $c_0 = \alpha \cdot \beta$; // Dual $\mathbb{F}_p$ multiplication.
4. $c_1 = 2 \cdot c_1$;

5.- IMPLEMENTATION RESULTS

This section evaluates the impact of the dual NEON-based finite field arithmetic on the performance of a software implementation of elliptic curve primitives. Experiments were carried out on the Xilinx XC7Z020 Zynq-7000 device. The XC7Z020 chip integrates a dual-core ARM Cortex-A9 processing system running at 667 MHz. Timing measurements were performed for the underlying finite field and the point arithmetic levels for both C and mixed C/NEON implementations. Average timing results for each operation were computed on the basis of 10000 independent runs of the operation under test. Timing samples were acquired by means of the Cortex-A9 64-bit global timer which is clocked at half of the processor's frequency. Our choices of elliptic curves for experimentation were the BN-curves shown in Table 2. However, we point out that the NEON-based field arithmetic can be directly applied to any other elliptic curve.

Table 2
Elliptic curves used for experiments

Curve	Bit-length		
	254	384	510
$E(\mathbb{F}_p)$	$y^2 = x^3 + 22$	$y^2 = x^3 + 12$	$y^2 = x^3 + 23$
$E(\mathbb{F}_{p^2})$	$y^2 = x^3 + 22(\mu + 1)^{-1}$	$y^2 = x^3 + 12(\mu + 1)^{-5}$	$y^2 = x^3 + 23(\mu + 1)^{-1}$

Table 3 presents the average execution times of the pure C and the mixed C/NEON variants of field multiplications (mul) and squarings (sqr) in both $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ finite fields. The column "Factor" shows the speedup ratios achieved by the NEON-based implementations of the field arithmetic. In the particular case of $\mathbb{F}_p$, this metric refers to the ratio between the running time of one dual NEON-based field multiplication or squaring and that of two consecutive calls to their corresponding single counterparts implemented in C (i.e., 1**Md**/2**M**).

Table 3
Timing performance of $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ arithmetic

Field	Bit-length	Operation	Timing (in 10^3 clock cycles)		Factor
			C	C/NEON	
$\mathbb{F}_p$	254	mul	3.2	3.8	0.59
		sqr	3	3.4	0.57
	384	mul	7.4	8.5	0.57
		sqr	6.8	7.5	0.55
	510	mul	12	13.8	0.58
		sqr	11.2	11.9	0.53
$\mathbb{F}_{p^2}$	254	mul	10.4	7.8	0.75
		mul-lr (*)	8.4	6.1	0.73
		sqr	6.8	4.2	0.62
	384	mul	22.9	16.9	0.74
		mul-lr (*)	19.4	13.8	0.71
		sqr	15.3	9	0.59
	510	mul	37	26.7	0.72
		mul-lr (*)	30.5	21.1	0.69
		sqr	24.6	14.5	0.59

(*) In this context lr stands for lazy reduction.

As shown in Table 3, for the three bit-lengths considered in this work, performing two single $\mathbb{F}_p$ multiplications is in average between 41% and 43% worse than performing one dual $\mathbb{F}_p$ multiplication. The same occurs when comparing two singles against one dual $\mathbb{F}_p$ squaring, where improvements between 43% and 47% are achieved. Also note that computing one dual $\mathbb{F}_p$ multiplication is between 39% and 41% faster than performing a single multiplication plus a single squaring in $\mathbb{F}_p$, which confirms the premise postulated at the end of Section 4.1.2. Mixed C/NEON field arithmetic in $\mathbb{F}_{p^2}$ also exhibits better performance than its pure C counterpart, with improvements between 25% and 41% for both multiplications and squarings. In addition, Table 3 shows that NEON instructions together with lazy reduction produce the best results for $\mathbb{F}_{p^2}$ multiplications with speedups between 18% and 22% compared to not using lazy reduction.

Let us now inspect the timing results of elliptic curve point arithmetic. Table 4 shows the values corresponding to point addition (Add) and point doubling (Dbl) on the elliptic curves over $\mathbb{F}_p$. As expected, point arithmetic in affine coordinates exhibits the lowest performance because of the need for costly field inversions. In addition, $E(\mathbb{F}_p)$ affine formulas do not have paired multiplications or squarings that can be performed simultaneously using the NEON engine. On the other hand, operations in standard and Jacobian coordinates take a great advantage from NEON-based $\mathbb{F}_p$ arithmetic. Performance improvements achieved by the NEON-based elliptic curve operations with respect to those not using NEON are between 31% and 36% for a point doubling and between 36% and 39% for point additions in both Jacobian and standard coordinates.

Table 4
Timing performance of $E(\mathbb{F}_p)$ point arithmetic

Bit-length	Coordinates	Point operation	Timing (in 10^3 clock cycles)		Factor
			C	C/NEON	
254	Affine	Add	123.9	–	–
		Dbl	126.9	–	–
	Standard	Add	35.9	23	0.64
		Dbl	32.6	22.6	0.69
	Jacobian	Add	35.5	22.7	0.64
		Dbl	22.8	15	0.66
384	Affine	Add	266.1	–	–
		Dbl	270.1	–	–
	Standard	Add	80.8	50.6	0.63
		Dbl	73.4	49.8	0.68
	Jacobian	Add	80.9	50.1	0.62
		Dbl	50.8	32.7	0.64
510	Affine	Add	446.4	–	–
		Dbl	455	–	–
	Standard	Add	131.4	80.7	0.61
		Dbl	118.6	79.4	0.67
	Jacobian	Add	130.8	80	0.61
		Dbl	82.5	51.5	0.62

Timing results for point arithmetic on elliptic curves over $\mathbb{F}_{p^2}$ are collected in Table 5. These results correspond to the use of lazy reduction for the underlying $\mathbb{F}_{p^2}$ multiplications. The first lazy reduction approach mentioned in Section 4.2.1 was used in the 254-bit and 510-bit curves while for the 384-bit instance we used the second approach. In the case of point arithmetic over $\mathbb{F}_{p^2}$, even affine coordinates operations are benefited from using the NEON extension. Although affine formulas in $\mathbb{F}_{p^2}$ do not present paired multiplications or squarings, the optimized NEON-based underlying $\mathbb{F}_{p^2}$ finite field operations

extrapolate their better performance up to the point arithmetic. Indeed, this argument is also valid for standard and Jacobian coordinates. Even when standard projective and Jacobian point arithmetic formulas have parallelizable multiplications and squarings, these operations in $\mathbb{F}_{p^2}$ could not be performed simultaneously using NEON. Doing that would require NEON instructions operating on 256-bit registers but the NEON engine works over 128-bit registers at most.

Although affine operations experiment some speedup when using NEON-based $\mathbb{F}_{p^2}$ arithmetic, note that improvements are between 27% and 35% in both standard and Jacobian in contrast to a maximum of 13% achieved in affine coordinates. This is because affine formulas in $\mathbb{F}_{p^2}$ still require a costly field inversion which is replaced in standard and Jacobian coordinates by field multiplications and squarings that take advantages from the use of the NEON technology.

Table 5
Timing performance of $E(\mathbb{F}_{p^2})$ point arithmetic

Bit-length	Coordinates	Point operation	Timing (in 10^3 clock cycles)		Factor
			C	C/NEON	
254	Affine	Add	153	139.9	0.91
		Dbl	159	143.8	0.9
	Standard	Add	91.4	66.4	0.73
		Dbl	80.4	56.9	0.71
	Jacobian	Add	89.7	64.5	0.72
		Dbl	54.8	38	0.69
384	Affine	Add	330.8	297.5	0.9
		Dbl	344.8	305.5	0.87
	Standard	Add	208.5	145.3	0.7
		Dbl	181.6	122.4	0.67
	Jacobian	Add	203.7	140.3	0.69
		Dbl	122.6	80.9	0.66
510	Affine	Add	543.5	489.9	0.9
		Dbl	568.1	505.5	0.89
	Standard	Add	327.3	222.9	0.68
		Dbl	285.3	189	0.66
	Jacobian	Add	321.4	216.2	0.67
		Dbl	193.3	125	0.65

Finally, Table 6 and Table 7 show the timings obtained for scalar multiplication in $E(\mathbb{F}_p)$ and $E(\mathbb{F}_{p^2})$ respectively. As expected, in all cases the best results come from the use of NEON instructions. It is also clear that the best options in terms of speed correspond to the mixed C/NEON scalar multiplications in Jacobian coordinates which exhibit improvements over the pure C implementations between 35% and 38% in the case of $E(\mathbb{F}_p)$ and between 29% and 34% in the case of $E(\mathbb{F}_{p^2})$.

Table 6
Timing performance of scalar multiplication in $E(\mathbb{F}_p)$

Bit-length	Coordinates	Timing (in 10^6 clock cycles)		Factor
		C	C/NEON	
254	Affine	47.3	–	–
	Standard	12.6	8.6	0.68
	Jacobian	10.1	6.6	0.65
384	Affine	147.4	–	–
	Standard	41.4	27.2	0.66
	Jacobian	32.7	20.7	0.63
510	Affine	334.4	–	–
	Standard	91.3	59.4	0.65
	Jacobian	72.7	45	0.62

Table 7
Timing performance of scalar multiplication in $E(\mathbb{F}_{p^2})$

Bit-length	Coordinates	Timing (in 10^6 clock cycles)		Factor
		C	C/NEON	
254	Affine	58.9	53.6	0.91
	Standard	31.5	22.6	0.72
	Jacobian	24.9	17.6	0.71
384	Affine	185.2	165.5	0.89
	Standard	103.5	71	0.69
	Jacobian	80.8	54.5	0.67
510	Affine	415	373.8	0.9
	Standard	222.1	148.5	0.67
	Jacobian	174	114.5	0.66

5.1.- COMPARISON WITH RELATED WORKS

Even when the goal of most works summarized in Section 2 was to accelerate elliptic curve arithmetic by using NEON vectorization, the choices for curves, underlying fields, point arithmetic formulas and scalar representations were quite diverse. Such different choices have a direct impact in the resulting performance. Therefore, it is very difficult to establish a fear comparison to evaluate only the influence of the NEON-based approach employed by each of those researches. The parameters setup closer to our settings is that of [11]. Consequently, our implementation timings of NEON-based finite field and point arithmetic are compared against their results in Table 8. Comparison only involves curve arithmetic in Jacobian coordinates since this was the point representation considered in that work.

Table 8
Results comparison for finite field and elliptic curve arithmetic

Work	Implementation	Bit-length	Timing (in 10^3 clock cycles)									
			$\mathbb{F}_p$		$E(\mathbb{F}_p)$			$\mathbb{F}_{p^2}$		$E(\mathbb{F}_{p^2})$		
			mul	sqr	Add	Dbl	$[s]P$	mul	sqr	Add	Dbl	$[s]P$
[11][a]	NEON (fixed length)	254	–	–	–	–	1698	2.29	2.00	–	–	2933
This work[b]	Mixed C/NEON (scalable)	254	3.8*	3.4*	22.7	15	6592	6.1	4.2	64.5	38	17551
		384	8.5*	7.5*	50.1	32.7	20710	13.8	9	140.3	80.9	54489
		510	13.8*	11.9*	80	51.5	44939	21.1	14.5	216.2	125	114537

a. Galaxy Note (ARM v7) Exynos 4 Cortex-A9 at 1.4 GHz.
b. Zynq ZC7Z020 (ARM v7) Cortex-A9 at 667 MHz.
* Refers to dual $\mathbb{F}_p$ multiplications and squarings timings.

As shown in Table 8 our mixed C/NEON scalar multiplication in $E(\mathbb{F}_p)$ is 74% slower and in $E(\mathbb{F}_{p^2})$ a 83% slower. There are two factors contributing to these results. Firstly, the 2-dimensional and 4-dimensional GLV [37] scalar multiplications used in [11] for $E(\mathbb{F}_p)$ and for $E(\mathbb{F}_{p^2})$ respectively, which are more efficient than the simpler binary left-to-right strategy used in this work. In second place is the fact that the flexibility provided by our library in terms of scalability comes at the cost of a significant performance loss respect to implementations optimized for a specific bit-length. Note that our mixed C/NEON implementation of $\mathbb{F}_{p^2}$ arithmetic is a 62% slower in the case of multiplications and a 52% slower in the case of squarings. To stress this argument, we ran a quick experiment in which we built multiplication, squaring and modular reduction primitives optimized for 254-bit parameters by unrolling the nested loops of our mixed C/NEON implementations and replicating as needed the NEON arithmetic kernels described in sections 4.1.1 and 4.1.2. As a result we obtained average running times of 1.58×10^3 and 1.5×10^3 clock cycles for $\mathbb{F}_{p^2}$ multiplications and squarings respectively. These metrics are about 31% and 25% faster than their homologous from [11].

6.- CONCLUSIONS

Achieving implementations of cryptographic primitives with a good performance is crucial to ensure a smooth end-user experience. This is especially relevant in the context of embedded solutions where the processing resources are somewhat limited when compared to that of modern general-purpose computers. Then, taking advantages of any platform specific feature providing support for acceleration could be determinant. Accordingly, this work evaluated the use of the NEON instruction set of ARM Cortex-A processors to speed up elliptic curve arithmetic. We followed the approach of parallelizing the underlying finite field operations using NEON instructions since we observed that elliptic curve point arithmetic formulas, especially in projective coordinates, involve several field multiplications and squarings that can be executed simultaneously. After implementing such NEON-based field operations, it was shown that using them in the context of elliptic curves boosted up the performance of scalar multiplication between 35% and 38% in $E(\mathbb{F}_p)$ and between 29% and 34% in $E(\mathbb{F}_{p^2})$ compared to their equivalent baseline implementations coded in C. Such improvements were obtained in standard and Jacobian coordinates. However, this was not the case for the implementations in affine coordinates in which there were at most a 13% of speedup in scalar point multiplications in $E(\mathbb{F}_{p^2})$ and no speedup at all in $E(\mathbb{F}_p)$. A costly field inversion together with the lack of parallelizable multiplications and squarings prevented affine equations from behave better. Finally, we stress that even when our library is flexible enough in terms of scalability thanks to the mixed C/NEON approach followed in this work, it is not comparable in terms of performance with implementations optimized for a specific bit-length. That is, a mixed C/NEON approach similar to our proposal could be a successful alternative if the main goal is to build an easily scalable software implementation while taking, at the same time, some advantage from NEON vectorization to improve performance. However, if the priority is to further minimize running time, then a full NEON-based implementation optimized for a specific bit-length is a better choice.

REFERENCES

1. Miller VS. Use of Elliptic Curves in Cryptography. In Williams HC, editor. Advances in Cryptology - CRYPTO '85, LNCS 218; 1986. p. 417-426.
2. Koblitz N. Elliptic Curve Cryptosystems. Mathematics of Computation. 1987; 48(177): 203-209.
3. National Institute of Standards and Technology. Digital Signature Standard (DSS) Gaithersburg; 2013.
4. Certicom Research. Elliptic Curve Cryptography; 2009.
5. Certicom Research. Recommended Elliptic Curve Domain Parameters; 2010.
6. National Institute of Standards and Technology. Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography Gaithersburg; 2018.
7. Hankerson D, Menezes A, Vanstone S. Guide to Elliptic Curve Cryptography Berlin, Heidelberg: Springer-Verlag; 2003.
8. Intel Corporation. Using Streaming SIMD Extensions (SSE2) to Perform Big Multiplications; 2000.
9. Seo H, Liu Z, Grobschadl J, Choi J, Kim H. Montgomery Modular Multiplication on ARM-NEON Revisited. In Lee J, Kim J, editors. Information Security and Cryptology - ICISC 2014, LNCS 8949; 2015. p. 328-342.
10. Bos JW, Montgomery PL, Shumow D, Zaverucha GM. Montgomery Multiplication Using Vector Instructions. In Lange , et al. , editors. Selected Areas in Cryptography - SAC 2013, LNCS; 2013. p. 471-489.
11. Sánchez AH, Rodríguez-Henríquez F. NEON Implementation of an Attribute-Based Encryption Scheme. In Jacobson , et al. , editors. Applied Cryptography and Network Security - ACNS 2013, LNCS 7954; 2013. p. 322-338.
12. Bernstein DJ, Schwabe P. NEON Crypto. In Prouff E, Schaumont P, editors. Cryptographic Hardware and Embedded Systems - CHES 2012, LNCS 7428; 2012: Springer, Berlin, Heidelberg. p. 320-339.
13. Faz-Hernández A, Longa P, Sánchez AH. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. IACR ePrint archive. 2013; 2013(158).
14. Camara DF, Gouvea CPL, López J, Dahab R. Fast software polynomial multiplication on ARM processors using the NEON engine. In Cuzzocrea , et al. , editors. Security Engineering and Intelligence Informatics - CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, LNCS 8128; 2013. p. 137-154.
15. Longa P. FourQNEON: Faster Elliptic Curve Scalar Multiplications on ARM processors. In Avanzi R, Heys H, editors. Selected Areas in Cryptography - SAC 2016, LNCS 10532; 2016: Springer.
16. Itoh T, Tsujii S. A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases. Information and Computation. 1988; 78(3): 171-177.
17. Shoup V. A Computational Introduction to Number Theory and Algebra. 2nd ed. New York: Cambridge University Press; 2009.
18. Montgomery PL. Modular Multiplication without Trial Division. Mathematics of Computation. 1985; 44(170): 519-521.
19. Bernstein DJ, Lange T. Analysis and optimization of elliptic-curve single-scalar multiplication. In Mullen GLea, editor. Finite fields and applications: Eighth international conference on finite fields and applications; 2007; Melbourne.
20. Cohen H, Miyaji A, Ono T. Efficient Elliptic Curve Exponentiation Using Mixed Coordinates. In Ohta K, Pei D, editors. Advances in Cryptology - ASIACRYPT '98, LNCS 6487; 1998: Springer-Verlag. p. 1-20.
21. Lauter K, Montgomery PL, Naehrig M. An Analysis of Affine Coordinates for Pairing Computation. In Joye M, et al. , editors. Pairing-Based Cryptography - Pairing 2010, LNCS 6487; 2010: Springer-Verlag. p. 1-20.
22. Chudnovsky DV, Chudnovsky GV. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. Advances in Applied Mathematics. 1986; 7(4): 385-434.
23. Explicit-Formulas Database. [Online]. [cited 2018 October 24. Available from: https://hyperelliptic.org/EFD/.
24. National Institute of Standards and Technology. Recommendation for Key Management Part 1: General Gaithersburg; 2016.
25. ARM. ARM Cortex-A Series Programmer's Guide Cambridge; 2014.
26. ARM. NEON Programmer's Guide Cambridge; 2013.

27. Koc CK, Acar T, Kaliski BS. Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro. 1996; 16(3): 26-33.
28. Menezes AJ, Vanstone SA, van Oorschot PC. Handbook of Applied Cryptography. 1st ed. Boca Raton: CRC Press, Inc.; 1996.
29. Knuth DE. The Art of Computer Programming, Volume 2: Seminumerical Algorithms. 3rd ed. Boston: Addison-Wesley Longman Publishing Co.; 1997.
30. Barreto PSLM, Naehrig M. Pairing-Friendly Elliptic Curves of Prime Order. In Preneel B, Tavares S, editors. Selected Areas in Cryptography - SAC 2005, LNCS 3897; 2006: Springer-Verlag. p. 319-331.
31. Hess F, Smart NP, Vercauteren F. The Eta Pairing Revisited. IEEE Transactions on Information Theory. 2006; 52(10): 4595-4602.
32. Vercauteren F. Optimal Pairings. IEEE Transactions on Information Theory. 2010; 56(1): 455-461.
33. Scott M. Implementing Cryptographic Pairings. In Takagi T, et al. , editors. Pairing-Based Cryptography - Pairing 2007, LNCS 4575: Springer-Verlag. p. 177-196.
34. Devegili AJ, O hEigeartaigh C, Scott M, Dahab R. Multiplication and Squaring on Pairing-Friendly Fields. IACR ePrint archive. 2006.
35. Weimerskirch A, Paar C. Generalizations of the Karatsuba Algorithm for Efficient Implementations. IACR Eprint archive. 2006.
36. Lim CH, Hwang HS. Fast Implementation of Elliptic Curve Arithmetic in GF(p^n). In Imai H, Zheng Y, editors. Public Key Cryptography - PKC 2000, LNCS 1751; 2000: Springer-Verlag. p. 405-421.
37. Gallant RP, Lambert RJ, Vanstone SA. Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In Kilian J, editor. Advances in Cryptology - CRYPTO 2001, LNCS 2139; 2001: Springer, Berlin, Heidelberg. p. 190-200.
38. Koblitz AH, Koblitz N, Menezes A. Elliptic curve cryptography: The serpentine course of a paradigm shift. Journal of Number Theory. 2011; 131: 781-814.

AUTORES

Raudel Cuiman Márquez, Ingeniero en Automática en 2009, Máster en Sistemas Digitales en 2014 por la Universidad Tecnológica de La Habana "José Antonio Echeverría" (CUJAE), Profesor Auxiliar del Departamento de Automática y Computación de la CUJAE, La Habana, Cuba. ORCID: 0000-0002-5145-7612. Email: raudel@automatica.cujae.edu.cu. Sus intereses de investigación están relacionados con la implementación eficiente de algoritmos criptográficos sobre sistemas empotrados.

Alejandro J. Cabrera Sarmiento, Ingeniero Electricista, Doctor en Ciencias Técnicas por la Universidad Tecnológica de La Habana "José Antonio Echeverría" (CUJAE), Profesor Titular del Departamento de Automática y Computación de la CUJAE, La Habana, Cuba. ORCID: 0000-0003-4129-911X. Email: alex@automatica.cujae.edu.cu. Sus líneas de investigación principales están relacionadas con el desarrollo de sistemas empotrados basados en FPGA y la aceleración de algoritmos sobre hardware reconfigurable, con aplicaciones en procesado de imágenes, control inteligente y criptografía.

Santiago Sánchez-Solano, Doctor en Ciencias Físicas por la Universidad de Sevilla en 1990, Investigador Científico del CSIC adscrito al Instituto de Microelectrónica de Sevilla, IMSE-CNM, (CSIC/Universidad de Sevilla), Sevilla, España. ORCID: 0000-0002-0700-0447. Email: santiago@imse-cnm.csic.es. Sus líneas de investigación se centran en el desarrollo de sistemas empotrados con componentes hardware-software sobre FPGA y la realización microelectrónica de sistemas neurodifusos, así como sus aplicaciones en robótica, procesamiento de imágenes, seguridad y redes de sensores inteligentes.