Hardware Elliptic Curve Cryptographic Processor Over $\mathrm{GF}(p)$

Ciaran J. McIvor, Máire McLoone, *Member, IEEE*, and John V. McCanny, *Fellow, IEEE*

***Abstract*—A novel hardware architecture for elliptic curve cryptography (ECC) over $\mathrm{GF}(p)$ is introduced. This can perform the main prime field arithmetic functions needed in these cryptosystems including modular inversion and multiplication. This is based on a new unified modular inversion algorithm that offers considerable improvement over previous ECC techniques that use Fermat's Little Theorem for this operation. The processor described uses a full-word multiplier which requires much fewer clock cycles than previous methods, while still maintaining a competitive critical path delay. The benefits of the approach have been demonstrated by utilizing these techniques to create a field-programmable gate array (FPGA) design. This can perform a 256-bit prime field scalar point multiplication in 3.86 ms, the fastest FPGA time reported to date. The ECC architecture described can also perform four different types of modular inversion, making it suitable for use in many different ECC applications.**

***Index Terms*—Elliptic curve cryptography (ECC), modular inversion, Montgomery modular multiplication.**

I. Introduction

Elliptic curve cryptosystems (ECCs) [1], [2] are increasingly becoming a more attractive alternative to traditional RSA systems [3], as these offer similar security but for a much smaller key size. For instance, a 256-bit ECC key size provides the same level of security as an equivalent 3072-bit RSA key [4]. This key size gap is set to further increase in future systems when security requirements become more demanding, as shown in Table I [5]. Therefore, smaller finite field arithmetic is required to encrypt data by using ECCs, allowing potentially higher data rates at a much lower implementation cost.

Over the past decade or more many efficient software implementations of ECCs have been reported, including those in [6]–[9] [over $\mathrm{GF}(2^m)$] and in [10] (over $\mathrm{GF}(p)$(. However, there is also an increasing need to develop efficient hardware implementations of finite field arithmetic processors. Firstly, hardware processors potentially provide significantly higher cryptographic processing rates when compared with software-only implementations, particularly for public-key computations [11]. This is increasingly a very important demand in Internet applications, with the rapid growth in data rates that are taking place. Therefore, as bandwidth requirements increase then so too does the speed with which encryption must be performed. Secondly, hardware-based cryptographic solutions intrinsically can provide significant security improvements over software solutions. For example, hardware can protect secret keys and other parameters, as this information only leaves the tamper-proof integrated circuit in an encrypted form; it is much more difficult to protect such information in a software environment [12].

TABLE I
ECC Versus RSA Key Sizes

ECC Key Size (Bits)	RSA Key Size (Bits)	Key Size Ratio	AES Key Size (Bits)
163	1024	1:6	–
256	3072	1:12	128
384	7680	1:20	192
512	15360	1:30	256

A relatively large amount of research has been performed on hardware implementations of ECCs over $\mathrm{GF}(2^m)$, including those given in [13]–[16]. This is because $\mathrm{GF}(2^m)$ implementations are generally faster and more compact than their $\mathrm{GF}(p)$ equivalents. However, conventional hardware implementations of ECCs over $\mathrm{GF}(2^m)$ are not very flexible, as the field parameters used are often fixed. In practice, ECC has many different operational modes and primitives [17] and therefore architectures need to be flexible. Thus, $\mathrm{GF}(2^m)$ may not be the ideal choice if flexibility is an important requirement. Moreover, RSA, the most widely used public-key cryptosystem uses arithmetic over $\mathrm{GF}(p)$. It is therefore attractive to also create hardware architectures suitable for ECCs over $\mathrm{GF}(p)$, which have the capability of supporting RSA as well as ECC cryptography.

To date, relatively little research has been reported on ECC hardware architectures over $\mathrm{GF}(p)$. However, important contributions have been described in [18]–[20]. Satoh and Takano [18], for example, proposed an elliptic curve cryptographic processor able to operate in $\mathrm{GF}(p)$ and $\mathrm{GF}(2^m)$. The Montgomery multiplier architecture used comprises a Wallace tree and carry propagation adders. In addition, throughput rate is boosted by the use of a new on-the-fly redundant binary converter, which reduces the number of point additions required when using the double-and-add algorithm (see Section II). However, their work does not fully deal with the problem of modular inversion, which is the most costly operation to be performed in an ECC, even when projective coordinates are used. Instead, they compute this using Fermat's Little Theorem, which requires modular exponentiation, a very computationally intensive operation. Orlando and Paar [19] proposed an architecture that uses a high-radix Montgomery multiplier, which pre-computes frequently used values. Here, an arithmetic unit—comprising a multiplier, adders, and a register—is used to compute the

Manuscript received May 12, 2005; revised December 10, 2005. This paper was recommended by Associate Editor Z. Wang.

The authors are with the Institute of Electronics, Communications, and Information Technology (ECIT), Queen's University Belfast, Belfast BT3 9DT, Northern Ireland, U.K. (e-mail: c.mcivor@ecit.qub.ac.uk; m.mcloone@ecit.qub.ac.uk; j.mccanny@ecit.qub.ac.uk).

Digital Object Identifier 10.1109/TCSI.2006.880184

field additions/subtractions and multiplications, and to perform comparisons. However, the register, which stores pre-computed values, is quite large. A limitation of this architecture is that it is primarily targeted towards memory-rich hardware such as field-programmable gate arrays (FPGAs). Again, Fermat's Little Theorem is used to perform modular inversion, with its ensuing implications in terms of computational requirements.

In this paper we introduce a new approach to modular inversion. This is based on a unified inversion algorithm that we have derived [21]. This can compute the classical modular inverse of an integer, the Montgomery modular inverse of an integer in the Montgomery domain and the Montgomery inverse of an ordinary integer or the classical inverse of an integer in the Montgomery domain using a computational processor. This builds on, but extends the approach of Savas and Koc [22] by observing that the two algorithms they present can be combined into a single more efficient version. This leads to savings of around one third in the number of Montgomery multiplications required for modular inversion and, in turn, reduces silicon hardware requirements. This leads to important efficiency improvements over systems based on the use of Fermat's Little Theorem, requiring considerably fewer clock cycles to perform modular inversion.

The modular multiplier architecture presented is also based on the original full-word Montgomery algorithm [23]. In this approach computations are performed using full-word operands rather than using bit-wise computations as has been the case in previous research [20], [24], [25]. These typically require many more clock cycles to compute a modular multiplication. The proposed approach requires both conventional word-length multiplications, as well as additions, and was previously thought to be unsuited to hardware designs, due to the performance limitations incurred by using large word-length multipliers. However, it is shown that by using these efficiently, designs can be created that require much fewer clock cycles than previous methods, while still maintaining a competitive critical path delay. This allows very high throughput rates to be achieved.

The methods adopted have been used to create a new ECC processor over GF(p). A 256-bit FPGA demonstrator based on these ideas is presented. This can perform a 256-bit prime field scalar point multiplication in 3.86 ms, the fastest FPGA time reported to date. The ECC architecture described can also perform four different types of modular inversion, making it suitable for use in many different ECC applications.

The structure of this paper is as follows. A description of ECCs over GF(p) and the arithmetic functions required in these is provided in Section II. The new approaches for modular inversion and multiplication and the ECC architecture developed based on these are presented in Section III, along with demonstrator performance results. A summary of the work and general conclusions are provided in Section IV.

II. ECC

This section provides a description of ECCs over GF(p) and the finite field arithmetic involved. More information on elliptic curve cryptographic primitives and detailed mathematical background can be found in [17].

A. *ECCs Over* GF(p) *Using Affine Coordinates*

The *Weierstrass* equations defining an elliptic curve over GF(p) for $p > 3$ are as follows:

$$E : y^2 = x^3 + ax + b \tag{1}$$

where x and y are elements of GF(p) and a and b are integers *modulo p* satisfying

$$4a^3 + 27b^2 \neq 0 (\bmod p). \tag{2}$$

An elliptic curve E over GF(p) consists of the solutions (x, y) as defined by (1) and (2) along with an additional element called the point at infinity, denoted by O. The set of points (x, y) are in the so-called *affine* coordinate point representation.

Elliptic curve cryptographic primitives [17] require scalar point multiplication. That is, given a point P on an elliptic curve, one needs to compute *eP*, where e is a positive integer. This is achieved by a series of additions and doublings of P, dependent on the value of the integer e. This can be computed using an algorithm similar to the binary modular exponentiation algorithm (used in the RSA cryptosystem), where the square and multiply stages are simply replaced by point doublings and point additions. This is known as the double-and-add algorithm [17].

There are distinct formulae to calculate elliptic curve point addition and point doubling. When an affine coordinate representation is used, these operations require the relatively expensive operation of modular inversion.

B. *Use of Projective Coordinates*

The need for modular inversion for each point addition or point doubling is eliminated, by using *projective* coordinates, as defined in [17]. In this case, only one modular inversion is required at the end of a full scalar point multiplication when converting back from projective to affine coordinates. Conversion formulae and point addition and point doubling formulae using projective coordinates are given below, where (x, y) is a point on the curve E. Essentially, one converts from affine to projective coordinates, performs the arithmetic functions, and then converts back to affine coordinates.

Conversion from *affine* coordinates to *projective* coordinates (trivial)

$$X \leftarrow x, Y \leftarrow y, Z \leftarrow 1. \tag{3}$$

Conversion from *projective* coordinates to *affine* coordinates

$$x = X/Z^2, y = Y/Z^3. \tag{4}$$

Elliptic curve point addition using projective coordinates requires the computation of

$$(X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2) \tag{5}$$

where

$$\begin{aligned} U_0 &= X_0 Z_1^2 \\ S_0 &= Y_0 Z_1^3 \\ U_1 &= X_1 Z_0^2 \\ S_1 &= Y_1 Z_0^3 \\ W &= U_0 - U_1 \\ R &= S_0 - S_1 \\ T &= U_0 + U_1 \\ M &= S_0 + S_1 \\ Z_2 &= Z_0 Z_1 W \\ X_2 &= R^2 - TW^2 \\ V &= TW^2 - 2X_2 \\ 2Y_2 &= VR - MW^3. \end{aligned} \tag{6}$$

Elliptic curve point doubling using projective coordinates computes

$$2(X_1, Y_1, Z_1) = (X_2, Y_2, Z_2) \tag{7}$$

where

$$\begin{aligned} M &= 3X_1^2 + aZ_1^4 \\ Z_2 &= 2Y_1 Z_1 \\ S &= 4X_1 Y_1^2 \\ X_2 &= M^2 - 2S \\ T &= 8Y_1^4 \\ Y_2 &= M(S - X_2) - T. \end{aligned} \tag{8}$$

As can be seen from (6) and (8), point addition and point doubling require sixteen and ten prime field multiplications, respectively. There are also two and five and one and three field additions and subtractions to be performed for point addition and doubling, respectively. The modular inversion, modular multiplication, and modular addition and subtraction operations can be performed using the elliptic curve cryptographic processor architecture, described in Section III.

III. Elliptic Curve Cryptographic Processor

The proposed processor can perform classical or Montgomery modular inversion, Montgomery modular multiplication, and modular addition and subtraction. This is based on the unified inversion and the full-word Montgomery multiplication architectures, introduced in Sections III-A and III-C, respectively. Combining these allows an ECC processor to be created, which exhibits a low scalar point multiplication time when implemented in hardware. Full details are presented in Sections III-E and III-F, respectively.

A. Modular Inversion Algorithm

Montgomery modular inversion is a common operation required in many public-key cryptosystems, including ECCs [1], [2] and RSA [3]. In 2000, Savas and Koc [22] proposed two modular inversion algorithms for computing the classical and Montgomery modular inverses, respectively. These are based on work originally undertaken by Kaliski [26]. These algorithms were originally derived for software implementations on general-purpose microprocessors. In this section, we propose a new, single and unified Montgomery modular inverse algorithm [21], which performs both classical and Montgomery modular inversion and is suitable for both hardware and software implementations. This extends the work of Savas and Koc [22]. The new algorithm reduces the number of Montgomery multiplication operations required by 33% and also reduces the resources required in an implementation by almost half. The previous algorithms and the new unified algorithm are presented below.

The classical modular inverse of an integer $a < p$, where p is a k -bit prime number, is given by

$$\mathrm{ModInv}(a) = a^{-1}(\mathrm{mod}\, p). \tag{9}$$

The algorithm for computing this is broken into two phases. Phase 1 computes the *Almost Montgomery Inverse* given by

$$\mathrm{Phase1}(a) = a^{-1}2^z(\mathrm{mod}\, p) \tag{10}$$

where z is an integer and $k \leqslant z \leqslant 2k$. The algorithm for computing Phase 1 is given as Algorithm 1 [22], [26].

Phase 2 then completes the operation by taking *Phase1(a)* and z as inputs and returning *ModInv(a)*, as shown in Algorithm 2. However, when using public-key cryptography, it is also desirable to compute the Montgomery modular inverse of an integer $a2^k$ *(mod p)* (already in the Montgomery domain). This is given by

$$\mathrm{MonInv}(a2^k(\mathrm{mod}\, p)) = a^{-1}2^k(\mathrm{mod}\, p). \tag{11}$$

These two different types of modular inverse can be achieved using the iterative Algorithms 2 and 3 [26], respectively.

The loop statements at step 2 of Algorithms 2 and 3 divide r_1 by two, z and $z-k$ times, in order to remove the 2^z and $2^{-k}2^z$ factors, respectively. The *MontMult* function at step 3 of Algorithm 3 refers to the Montgomery modular multiplication of two k -bit integers, as discussed in Section III-C, this converts $r_1 = a^{-1}(\mathrm{mod}\ p)$ to $r_1 = a^{-1}2^k(\mathrm{mod}\ p)$.

Algorithm 1: Phase 1

```
Input: $a, p$ where $a < p$;
Output: $\mathrm{Phase1}\ (a) = a^{-1}2^z(\mathrm{mod}\, p), z$ where $k \le z \le 2k$;
1. $u = p; v = a; s1 = 0; s2 = 1; z = 0;$
2. while $v > 0$ loop
     if $u$ is even then
       u = u/2; s2 = 2s2;
     elsif $v$ is even then
       $v = v/2; s1 = 2s_1;$
     elsif $u > v$ then
       $u = (u - v)/2; s_1 = s_1 + s_2; s_2 = 2s_2;$
     elsif $v \geq u$ then
       $v = (v - u)/2; s_2 = s_2 + s_1; s_1 = 2s_1;$
```

```
        end if;
        $z = z + 1$;
    end loop;
3. if $s_1 \geq p$ then $s_1 = s_1 - p$;
4. return $\mathrm{Phase1}(a) = p - s_1$;
    return z.
```

Algorithm 2: Kaliski's ModInv

```
Input: $a, p$ where $a < p$;
Output: $r_1 = a^{-1} (\mathrm{mod}\, p)$;
1. $r_1 = \mathrm{Phase1}(a) = a^{-1}2^z \ (\mathrm{mod}\, p)$;
2. for $i = 1$ to z loop
        if $r_1$ is even then
            $r_1 = r_1/2$;
        else
            $r_1 = (r_1 + p)/2$;
        end if;
    end loop;
3. return $r_1 = a^{-1} (\mathrm{mod}\, p)$.
```

Algorithm 3: Kaliski's MonInv

```
Input: $a2^k \ (\mathrm{mod}\ p), p, k, R^2$ where $a < p$ and
$R^2 = 2^{2k} \ (\mathrm{mod}\, p)$;
Output: $r_1 = a^{-1}2^k \ (\mathrm{mod}\, p)$;
1. $r_1 = \mathrm{Phase1}(a2^k) = a^{-1}2^{-k}2^z \ (\mathrm{mod}\, p)$;
2. for $i = 1$ to $z - k$ loop
        if $r_1$ is even then
            $r_1 = r_1/2$;
        else
            $r_1 = (r_1 + p)/2$;
        end if;
    end loop;
3. $r_1 = \mathrm{MontMult}(r_1, R^2) = a^{-1} \times 2^{2k} \times 2^{-k} \ (\mathrm{mod}\, p) =$
   $a^{-1}2^k \ (\mathrm{mod}\, p)$;
4. return $r_1 = a^{-1}2^k \ (\mathrm{mod}\, p)$.
```

Savas and Koc [22] modified these approaches by suggesting that Montgomery multiplication can be used to replace the iterative loops in Algorithms 2 and 3 and proposed the two individual algorithms—given as Algorithms 4 and 5. Here m is an integer multiple of the word size, w, of the computer system used and $m \geqslant k$. In this case, the output z from Algorithm 1 is an integer satisfying $k \leqslant z \leqslant k+m$. Also, $R^2 = 2^{2m} (\mathrm{mod}\ p)$ and the inputs to the MontMult function are assumed to be m-bit integers.

Algorithm 4: Savas/Koc ModInv

```
Input: $a, p, m$ where $a < p$;
Output: $r_1 = a^{-1} \ (\mathrm{mod}\, p)$;
1. $r_1 = \mathrm{Phase\,1}(a) = a^{-1}2^z \ (\mathrm{mod}\, p)$;
2. if $z > m$ then
        $r_1 = \mathrm{MontMult}(r_1, 1) = a^{-1} \times 2^z \times 1 \times 2^{-m} \ (\mathrm{mod}\, p)$
        $= a^{-1}2^{z-m} \ (\mathrm{mod}\, p)$;
        $z = z - m < m$;
    end if;
3. $r_1 = \mathrm{MontMult}(r_1, 2^{m-z}) = a^{-1} \times 2^z \times$
        $2^{m-z} \times 2^{-m} \ (\mathrm{mod}\ p) = a^{-1} \ (\mathrm{mod}\, p)$;
4. return $r_1 = a^{-1} \ (\mathrm{mod}\, p)$.
```

Algorithm 5: Savas/Koc MonInv

```
Input: $a2^m \ (\mathrm{mod}\ p), p, m, R^2$ where $a < p$ and
$R^2 = 2^{2m} \ (\mathrm{mod}\, p)$;
Output: $r_1 = a^{-1}2^m \ (\mathrm{mod}\, p)$;
1. $r_1 = \mathrm{Phase\,1}(a2^m) = a^{-1}2^{-m}2^z \ (\mathrm{mod}\, p)$;
2. if $k \leq z \leq m$ then
        $r_1 = \mathrm{MontMult}(r_1, R^2) = a^{-1} \times 2^{-m} \times 2^z \times 2^{2m} \times$
        $2^{-m} \ (\mathrm{mod}\ p) = a^{-1}2^z \ (\mathrm{mod}\, p)$;

        $z = z + m > m$;
    end if;
3. $r_1 = \mathrm{MontMult}(r_1, R^2) = a^{-1} \times 2^z \times 2^{-m} \times 2^{2m} \times$
        $2^{-m} \ (\mathrm{mod}\, p) = a^{-1}2^z \ (\mathrm{mod}\, p)$;
4. $r_1 = \mathrm{MontMult}(r_1, 2^{2m-z}) = a^{-1} \times 2^z \times 2^{2m} \times 2^{-z} \times$
        $2^{-m} \ (\mathrm{mod}\ p) = a^{-1}2^m \ (\mathrm{mod}\, p)$;
5. return $r_1 = a^{-1}2^m \ (\mathrm{mod}\, p)$.
```

Savas and Koc [22] reported software implementations and comparisons of their new algorithms (Algorithms 4 and 5), which showed significant speed improvements when compared with similar implementations based on Algorithms 2 and 3. In particular, speed-ups of factors of 1.51 and 1.56 were reported by using Algorithm 4 instead of Algorithm 2 for 160- and 192-bit operands, respectively. They also reported speed-ups of 1.32 and 1.36 using Algorithm 5 instead of Algorithm 3 for 160- and 192-bit operands, respectively.

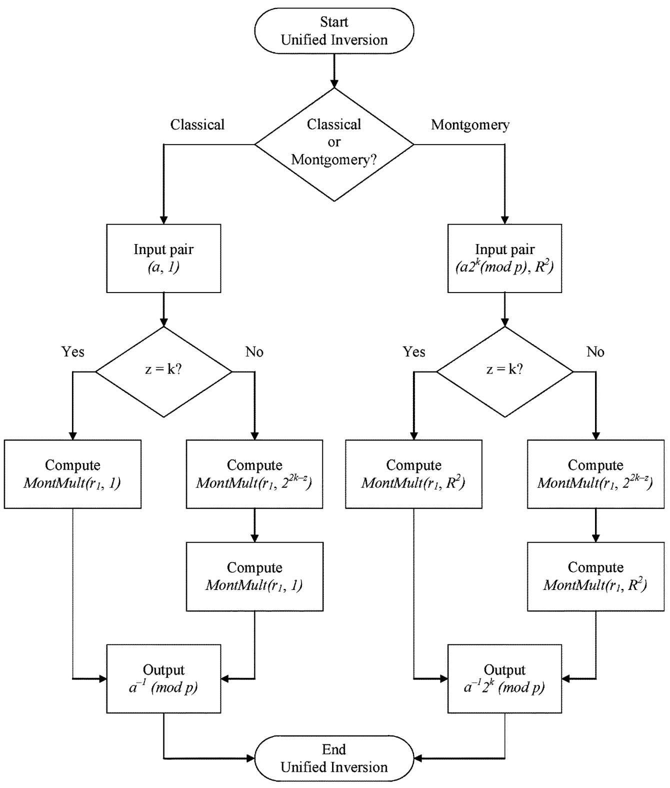

Fig. 1. Flowchart of functionality of unified algorithm.

It is noted that Algorithm 4 requires one or two Montgomery multiplications and Algorithm 5 requires 2 or 3 multiplications. Also, if these are used as the basis of a hardware inverter implementation, then two distinct circuit architectures would be required.

However, through careful observation and by manipulating these, it can be seen that they can be rewritten in a unified form to create a new single algorithm. This unified algorithm is given as Algorithm 6 (6a and 6b), below. This can be implemented as a single inverter architecture and can compute both the classical modular inverse of an integer and the Montgomery modular inverse of an integer in the Montgomery domain. Here, it is assumed that $R^2 = 2^{2k} \pmod p$ and the inputs to the MontMult function are k-bit integers. The algorithm is written twice in order to illustrate its functionality for two different sets of inputs. These are highlighted in bold type. If one wishes to calculate the classical modular inverse then the pair $(a, 1)$ is input to the algorithm, whereas the Montgomery modular inverse of an integer already in the Montgomery domain is calculated by inputting the pair $(a2^k (\bmod p), R^2)$ to the algorithm. This process is explained further using the flowchart given in Fig. 1.

Algorithm 6a: The Unified Inverse Algorithm for ModInv

```
Input: $a, p, k, 1$ where $a < p$;
Output: $r_1 = a^{-1} \pmod p$;
1. $r_1 = \text{Phase}\,1(a) = a^{-1}2^z \pmod p$;
2. if $z = k$ then
      $r_1 = \text{MontMult}(r_1, 1) = a^{-1} \times 2^z \times 1 \times 2^{-k} \pmod p = a^{-1} \pmod p$;
   else
      $r_1 = \text{MontMult}(r_1, 2^{2k-z}) = a^{-1} \times 2^z \times 2^{2k} \times 2^{-z} \times 2^{-k} \pmod p = a^{-1}2^k \pmod p$;
      $r_1 = \text{MontMult}(r_1, 1) = a^{-1} \times 2^k \times 1 \times 2^{-k} \pmod p = a^{-1} \pmod p$;
   end if;
3. return $r_1 = a^{-1} \pmod p$.
```

Algorithm 6b: The Unified Inverse Algorithm for MonInv

```
Input: $a2^k \pmod p, p, k, R^2$ where $a < p$ and $R^2 = 2^{2k} \pmod p$;
Output: $r_1 = a^{-1}2^k \pmod p$;
1. $r_1 = \text{Phase}\,1(a2^k) = a^{-1}2^{-k}2^z \pmod p$;
2. if $z = k$ then
      $r_1 = \text{MontMult}(r_1, R^2) = a^{-1} \times 2^{-k} \times 2^z \times 2^{2k} \times 2^{-k} \pmod p = a^{-1}2^k \pmod p$;
   else
      $r_1 = \text{MontMult}(r_1, 2^{2k-z}) = a^{-1} \times 2^{-k} \times 2^z \times 2^{2k} \times 2^{-z} \times 2^{-k} \pmod p = a^{-1} \pmod p$;
      $r_1 = \text{MontMult}(r_1, R^2) = a^{-1} \times 2^{2k} \times 2^{-k} \pmod p = a^{-1}2^k \pmod p$;
   end if;
3. return $r_1 = a^{-1}2^k \pmod p$.
```

The unified algorithm can also compute the Montgomery inverse of an ordinary integer or the classical inverse of an integer in the Montgomery domain. This is achieved by extracting the output after step 1 if $z = k$ or after the first Montgomery multiplication if $z \neq k$, respectively. The algorithm can therefore be used to convert between the ordinary integer domain and the Montgomery domain or vice versa. These operations are often required when performing modular arithmetic in public-key cryptosystems. The unified algorithm is also more efficient than the previous method described in [22] in computing the Montgomery modular inverse of an integer already in the Montgomery domain. As discussed, Algorithm 5 requires a maximum of three multiplications to calculate this whereas the new approach requires at most only two and thus a minimum 33% saving in Montgomery multiplication operations is achieved. This unified algorithm is therefore well suited for hardware (and indeed software) implementations, as a single circuit architecture can serve to compute both types of inverse, allowing for an efficient and compact inverter implementation. A saving of up to 50% (when compared with using Algorithms 4 and 5) on the number of gates required in

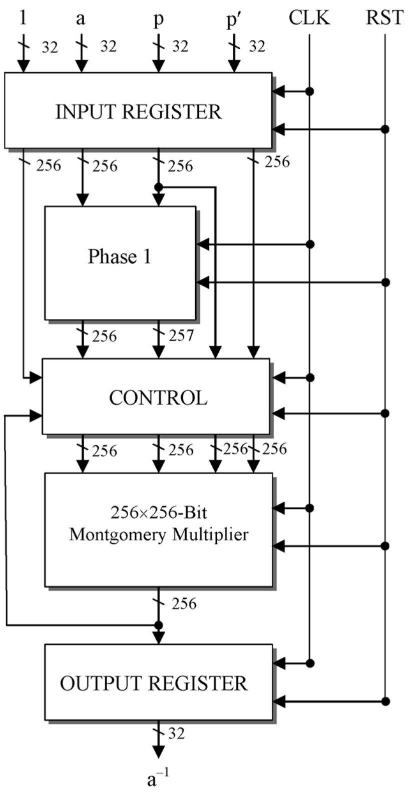

Fig. 2. A 256-bit unified inversion architecture.

an implementation can be achieved. This is discussed further in Section III-B.

A 256-bit modular inversion hardware architecture based on the unified algorithm is shown in Fig. 2 for input values a and 1 (to compute the classical inverse). If one wishes to compute the Montgomery modular inverse of an integer already in the Montgomery domain then the values $a^{2k} (\mathrm{mod}\ p)$ and R^2 should be input instead. Here, the inputs are registered into the circuit 32 bits per clock cycle over 8 cycles, as shown in Fig. 2. The values a and p are then fed into the *Phase1* component, which comprises a 256-bit adder and subtractor. These additions and subtractions are performed sequentially in accordance with Algorithm 1 using a state machine approach. The resulting values *Phase1(a)* and z are then stored in the control unit of the circuit. Here, the value 2^{2k-z} is determined. A comparison between z and k is also performed to determine the inputs to the 256-bit Montgomery multiplier, according to Algorithm 6. This is the 256-bit version of the Montgomery multiplier architecture described in Section III-C. If $z = k$ then only one modular multiplication is required. If not, then two multiplications are needed and r_1 (from Algorithm 6) is fed back into the control unit to be reused as an input for the second multiplication, as shown in Fig. 2. Once the necessary modular multiplications have been completed, a^{-1} is output from the circuit 32 bits per clock cycle over eight cycles. It should be noted that although Fig. 2 depicts a 256-bit inverter, the design is general and can be reconfigured to produce architectures of any operand length.

B. Modular Inversion Implementation

The inversion architecture shown in Fig. 2 has been captured in VHDL and implemented on a Xilinx Virtex2 Pro XC2VP125-7-ff1696 FPGA [27] for demonstration purposes, using a 256-bit operand length. The additions and subtractions required have been implemented using the fast carry chains located on the Virtex2 Pro devices. Also, the multiplications are performed using the 256-bit full-word Montgomery multiplication implementation given in Section III-D. For comparative purposes, inverter implementations based on Algorithms 4 and 5 have also been implemented in a similar manner. Table II provides performance results for these implementations, obtained using Xilinx Foundation software v6.1.03i.

It will be noted that no significant speed-up is achieved if the unified algorithm is used when compared with Algorithms 4 and 5. However, the reduction in the number of slices used is 49.9%, as a single inverter circuit is used rather than the two separate circuits previously required. This is calculated as the percentage of the difference between the total number of slices used to implement Algorithms 4 and 5 and the number of slices used in the implementation of Algorithm 6 over the total number of slices used for Algorithms 4 and 5. Similar speed-ups and savings in source code/silicon area are attainable if the algorithms are implemented in software or alternative hardware media, such as modern ASIC devices. The unified inversion algorithm affords this, due to its inherently less complicated structure.

It should be acknowledged that there have been several alternative modular inversion implementations recently reported in the literature including those in [28]–[32]. In the most recent implementation, given in [28], the authors use a modified version of Algorithm 1 to compute *Phase1(a)* and then an iterative process involving division by two (similar to Algorithms 2 and 3) to compute the final result $r_1 = a^{-1}2^k\ (\mathrm{mod}\ p)$. This modified algorithm is hardware orientated and is written in such a way as to reduce the number of operations required by taking into account the targeted device (Xilinx Virtex-E and Xilinx Spartan3) constraints. A direct comparison between this and the unified implementation is difficult, as the functionality of the underlying algorithms used for these are different. The algorithm presented in [28] can only compute the Montgomery inverse of an ordinary integer, whilst that presented here computes the wider range of Montgomery functions described above. i.e., the unified algorithm can compute the classical modular inverse of an integer, the Montgomery modular inverse of an integer in the Montgomery domain, and the Montgomery inverse of an ordinary integer or the classical inverse of an integer in the Montgomery domain. For similar reasons, the implementations given in [29]–[32] cannot be directly compared with the unified inversion implementation presented.

C. Full-Word Montgomery Multiplication Architectures

Modular multiplication is also a key operation needed in public-key cryptography. Within this context, Montgomery modular multiplication [24] is one of the most efficient modular multiplication algorithms available. This section describes the original full-word Montgomery multiplication algorithm [23] and corresponding hardware architectures for this, which have been used to create the ECC processor described in

TABLE II
PERFORMANCE RESULTS FOR INVERTER IMPLEMENTATIONS

Algorithm	Clock (MHz)	Area (Slices)	Clock Cycles	Inversion Time (μs)	Function (Inverse Type)	% Speed-up using Algorithm 6 instead	% Area Saved using Algorithm 6 instead
4	40.46	14,723	585	14.46	Classical	- 1.2 %	49.9 %
5	40.68	14,844	619	15.22	Montgomery	3.8 %	
6	40.04	14,800	586	14.64	Classical & Montgomery	N/A	N/A

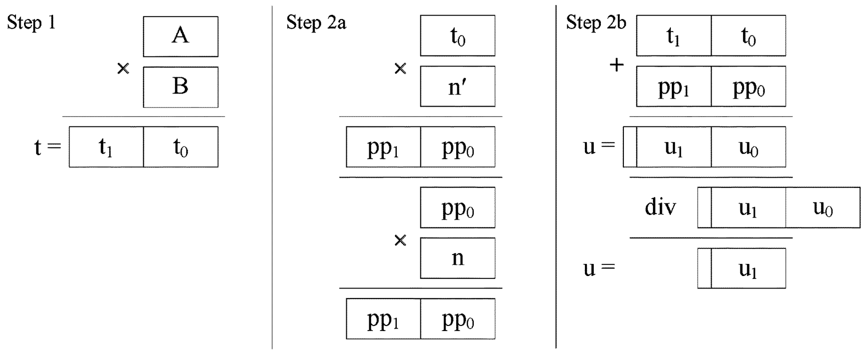

Fig. 3. Analysis of full-word Montgomery multiplication.

Section III-E. The full-word version of Montgomery's multiplication algorithm, which calculates the Montgomery product of k-bit n-residues A and B, is summarised as Algorithm 7, where n is the k-bit modulus, $r = 2^k$, and $(r \times r^{-1} - n \times n\prime) = 1$.

The main arithmetic functions to be performed include three full-word multiplications, one full-word addition, and a conditional full-word subtraction. Modulo and division operations by $r = 2^k$ are also required, which can be implemented in hardware as left and right shift operations, respectively. Fig. 3 provides an analysis of how this algorithm works.

Firstly, A and B are multiplied together to compute their product t, as shown in step 1 of Fig. 3. The product $t \times n\prime (\mathrm{mod}\, r)$ is then calculated by multiplying together $t (\mathrm{mod}\ r) = t_0$ and $n\prime$, as shown in step 2a. By reducing this product *modulo r* we obtain the least significant partial product pp_0, which is multiplied by n to obtain the product $(t \times n\prime (\mathrm{mod}\ r) \times n)$ required at step 2 of Algorithm 7.

The value $n\prime$ is calculated so that when $(t \times n\prime (\mathrm{mod}\ r) \times n)$ is added to t, the sum is exactly divisible by r to obtain $u = A \times B \times r^{-1}$, as shown in step 2b of Fig. 3 and in the following equation (remember $(r \times r^{-1} - n \times n\prime) = 1$ and $t = A \times B$):

$$\begin{aligned}
&[t + (t \times n\prime \times n)]\ \mathrm{div}\, r \\
&\quad = [t + (t \times ((r \times r^{-1} - 1)/n) \times n)]\mathrm{div}\, r \\
&\quad = [t + (t \times (r \times r^{-1} - 1))]\mathrm{div} r \\
&\quad = [t + (-t + t \times r \times r^{-1})]\mathrm{div}\, r \\
&\quad = [t \times r \times r^{-1}]\mathrm{div}\, r \\
&\quad = [A \times B \times r \times r^{-1}]\mathrm{div}\, r \\
&\quad = A \times B \times r^{-1}. \qquad (12)
\end{aligned}$$

A conditional subtraction of the modulus is then performed to obtain the value $u = A \times B \times r^{-1} (\mathrm{mod}\ n)$, as required.

To the authors' knowledge, no hardware architectures or implementations have been described for this. Previous hardware Montgomery multipliers have traditionally been based on a bitwise version of Algorithm 7 [24], [25] or variants of this. The attraction of using full-word length multipliers and adders is that the potential number of clock cycles required can be much less than required previously, while still maintaining a competitive critical path delay. This, in turn, should allow for higher throughput rates to be achieved. Thus, new hardware architectures for performing Algorithm 7 have been developed. An example of this is presented in Fig. 4, which is based on a 256-bit operand length.

In this, the inputs are registered into the circuit 32 bits per clock cycle for eight cycles, with this combination perhaps varying dependent on the technology used. This allows significant reductions in the number of inputs and outputs required. The control unit then determines the order in which the multiplications, addition and conditional subtraction are performed according to Algorithm 7, as shown in Table III.

The *t-REG/UPDATE REG/CONTROL* component stores $t = A \times B$ and the results of the other multiplications and addition, which are then fed back into the control unit to be re-used as inputs to the 256 × 256-bit multiplier or *Addition/Subtraction* component. These are conventional multipliers and adders. The *t-REG/UPDATE REG/CONTROL* component also performs the trivial *mod* and *div* operations in Algorithm 7. Once the conditional subtraction is performed, u is output from the circuit 32 bits at a time over eight clock cycles. Again, the design is general and can be reconfigured to produce architectures of any operand length.

The conventional multipliers and adders shown in Fig. 4 may be implemented using well-known methods, such as Booth recoding (for multiplication), or alternatively they can be tech-

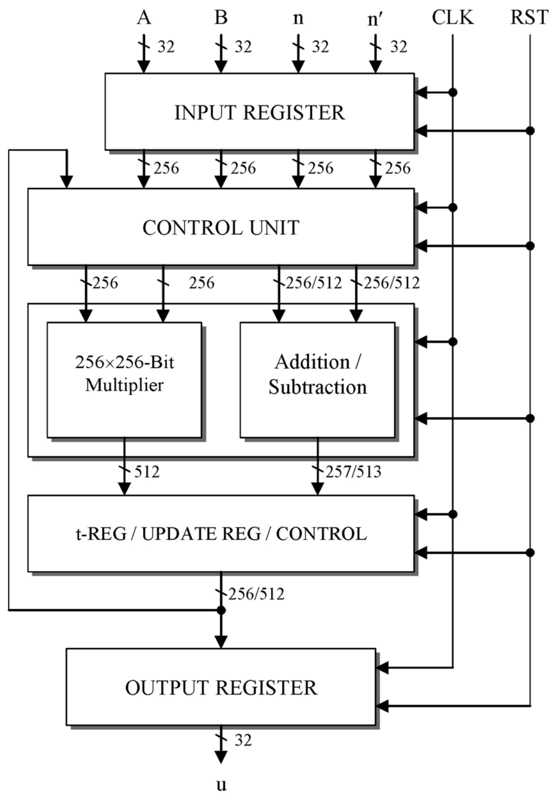

Fig. 4. A 256-bit Montgomery multiplier architecture.

TABLE III
COMPONENTS REQUIRED BY 256-BIT MONTGOMERY ARCHITECTURE

State	Component	Inputs	Outputs
0	*256×256 Mult*	A, B	$t^{(0)}$
1	*256×256 Mult*	$t^{(0)}$ (mod r), n′	$t^{(1)}$
2	*256×256 Mult*	$t^{(1)}$ (mod r), n	$t^{(2)}$
3	*512-bit Add*	$t^{(0)}$, $t^{(2)}$	$u^{(0)}$
4	256-bit Subtract	$u^{(0)}$ (div r), n	$u^{(1)}$

nology specific and comprise of embedded arithmetic functions located on modern FPGAs, for example. Thus, the architectures are general and suitable for implementation on a variety of different hardware platforms.

D. Full-Word Montgomery Multiplication Implementation

The architecture described in Section III-C has been captured in VHDL and used to create an implementation using the Xilinx Virtex2 Pro family of FPGAs [27]. The full-word addition and subtraction, required in the multiplier architecture shown in Fig. 4, are computed using the fast carry chains located on these devices and two's-complement addition, respectively. To perform the multiplications for $k = 256$ bit, it is necessary to develop a 256×256-bit multiplier, by cascading numerous 16×16-bit unsigned multipliers, as shown in Fig. 5.

The larger multipliers have been developed in a systematic fashion. For instance, the partial products of the 32×32-bit multiplier are calculated using the 16×16-bit unsigned multiplier blocks. These partial products are then added together using the fast look-ahead carry chains to obtain the 64-bit final product. This process is continued until the desired multiplier size is attained, as shown in Fig. 5. The calculation and addition of the partial products is a fully pipelined process, designed to take full advantage of the small critical path delay of the 16×16-bit multiplier blocks and the fast carry chains. Therefore, it takes 3, 5, 7, and 9 clock cycles to complete a full 32-, 64-, 128-, and 256-bit multiplication, respectively. The 256×256-bit multiplier, as shown in Fig. 5, is used in the 256-bit Montgomery multiplier implementation.

Table IV provides post place and route performance results for this implementation, obtained using Xilinx Foundation software v6.1.03i. The 256-bit Montgomery multiplier has been implemented in the XC2VP125-7-ff1696 FPGA. To the authors' knowledge, this is the fastest FPGA 256-bit modular multiplication time reported to date. These results demonstrate that by using Algorithm 7 to perform Montgomery multiplication then impressive throughput rates can be achieved, when implemented in hardware. This shows that a competitive critical path can be maintained when using full-word computations (rather than the traditional bit-wise approach) and a reduced number of clock cycles to compute a Montgomery multiplication.

E. ECC Processor

By combining the modular inverter and Montgomery multiplier architectures described in Sections III-A and III-C, respectively, a new ECC hardware processor has been developed and implemented, which displays impressive data throughput rates. A block diagram of this for a 256-bit operand length is shown in Fig. 6.

In this, the inputs are registered into the circuit 32 bits per clock cycle over eight cycles. The values of the inputs A and B depend on which arithmetic function is to be performed. For example, if one wishes to compute a classical modular inverse, then $A = a$ and $B = 1$ (see Algorithm 6). The *CONTROL1* component then determines which arithmetic function is to be performed, according to the *MODE* signal, and inputs the appropriate values to the 256-bit modular arithmetic unit, as shown in Fig. 6. This 2-bit *MODE* signal allows four choices: "00" for Montgomery multiplication; "01" for classical or Montgomery inversion; "10" for modular addition; "11" for modular subtraction. The processor described requires a single Montgomery multiplier and this takes the form of a 256-bit version of the architecture described in Section III-C. As discussed, the modular inverter part of the circuit is based on the unified inversion architecture described in Section III-A. For this, Phase1 is computed in a similar way to that described in Algorithm 1. One or two Montgomery multiplications are then required (see Algorithm 6) depending on the outputs of Phase1. These are performed using the 256-bit Montgomery multiplier. The *CONTROL2* component determines the number of modular multiplications needed and in turn feeds the correct operands into the modular multiplier, as shown in Fig. 6.

The modular additions and subtractions are quite trivial operations to perform. They are calculated using Algorithms 8 and

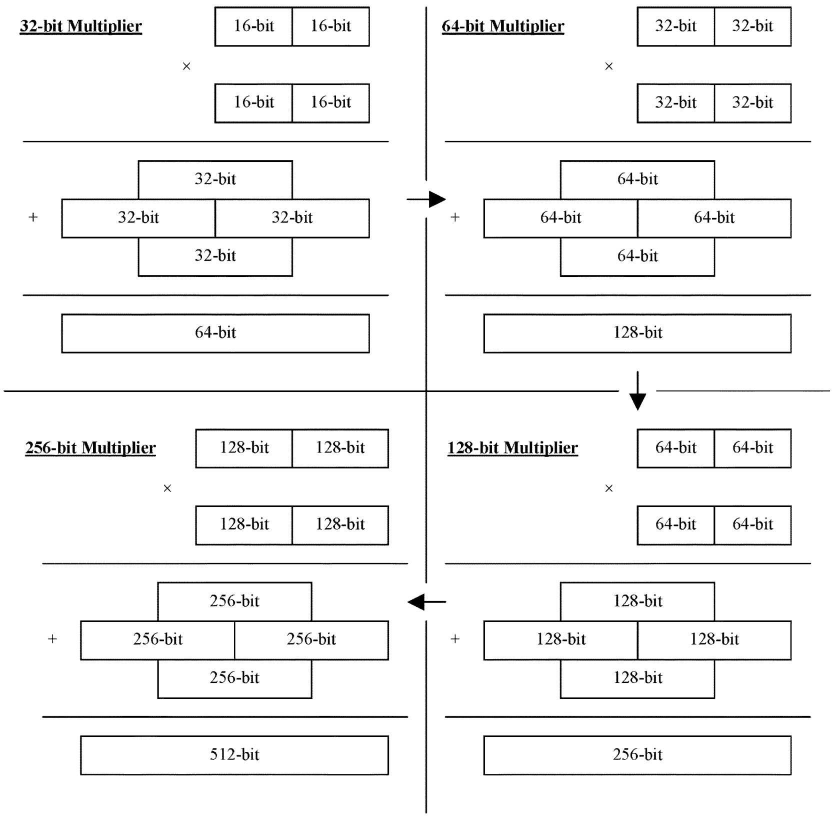

Fig. 5. Cascading 16 × 16-bit unsigned multipliers.

TABLE IV
PERFORMANCE RESULTS FOR FULL-WORD MULTIPLIER IMPLEMENTATION

Multiplier	Clock (MHz)	Area (Slices)	Mult 18×18	Clock Cycles	Data Rate (Mb/s)	Multiplication Time (μs)
MontMult_256Bit	45.68	11,992	256	32	365.44	0.70

9, respectively. X and Y are positive integers and $X, Y < p$, where p is the k-bit modulus.

Once the requested arithmetic function has been performed (controlled by the *MODE* signal), the result is output from the chip 32-bits per clock cycle over eight cycles. Although Fig. 6 depicts a 256-bit processor, the design is general and can be reconfigured to produce architectures of any appropriate operand length.

As discussed above, the processor uses the unified inversion algorithm and multiplier architectures described in Sections III-A and III-C, respectively, which in turn require Montgomery multiplication. Combining these therefore results in a circuit with many distinct advantages over those previously reported, including improved modular inversion computation time and functionality.

F. Performance Results

The ECC processor described has been captured in VHDL and a 256-bit operand length used to implement this on a Xilinx Virtex2 Pro XC2VP125-7-ff1696 FPGA [27]. The modular inverter and multiplier used are based on the implementations described in Sections III-B and III-D, respectively. The addition and subtraction operations in Algorithms 8 and 9 are performed using the fast carry logic on the Virtex2 Pro devices [27]. The FPGA design presented is highly adaptable and easily reprogrammable for varying prime field sizes. This is a very desirable characteristic when using ECC, due to the large number of different *secure* curves, prime fields and cryptographic primitives currently available [17]. Future FPGA implementations should allow advantage to be taken of the PowerPC RISC processing units available on the Virtex2 Pro devices. These can be programmed to control the arithmetic components, enabling the arithmetic functions to be utilised in a predefined order, in accordance with a specific cryptographic primitive. Thus, any one of a number of elliptic curve cryptographic primitives can be performed with ease, using this hardware/software co-design methodology.

Table V provides post place and route performance results for this ECC processor, again obtained using Xilinx Foundation software v6.1.03i. The critical delay through this is 25.34 ns, giving a maximum clock speed of 39.46 MHz. The circuit requires 15,755 CLB slices and uses 256 18 × 18-bit embedded multipliers. As shown in Table V, a 256-bit scalar point multiplication over $\mathrm{GF}(p)$ takes 3.86 ms, using the double-and-add algorithm and projective coordinates.

The performance results presented in Table V are not easily comparable to the previous hardware implementations over

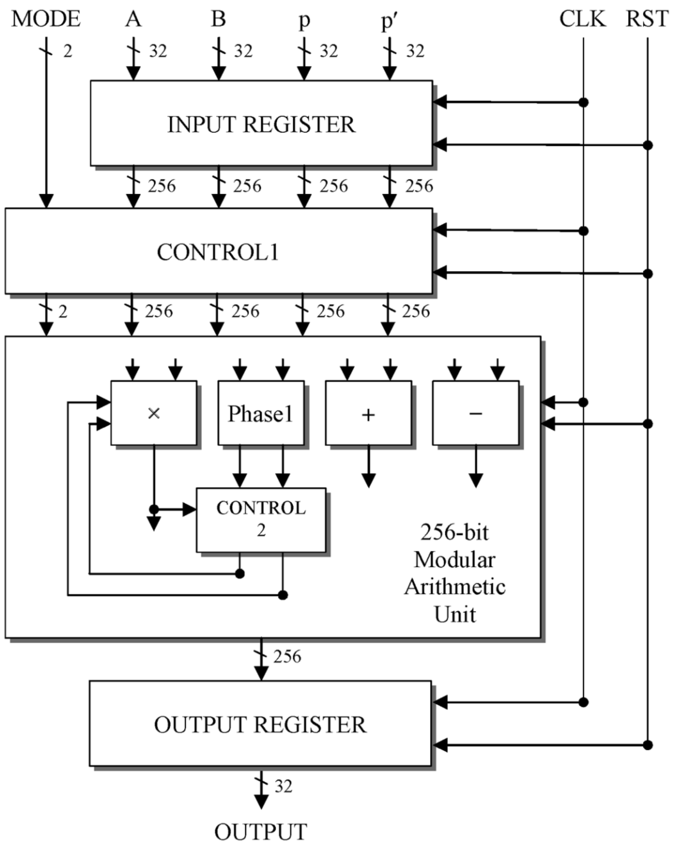

Fig. 6. A 256-bit ECC processor.

TABLE V
PERFORMANCE RESULTS FOR 256-BIT GF(P) ECC PROCESSOR

Function	Clock Cycles	Throughput Rate	Time for One Operation
Montgomery Multiplication	32	315.68 Mb/s	0.81 μs
Montgomery Inversion	586	17.24 Mb/s	14.85 μs
Modular Addition	2	5.05 Gb/s	51 ns
Modular Subtraction	2	5.05 Gb/s	51 ns
Point Addition	526	19.20 Mb/s	13.33 μs
Point Doubling	328	30.80 Mb/s	8.31 μs
Point Multiplication	151,360	66.74 kb/s	3.86 ms

GF(p), reported in [18]–[20]. These are also not easily comparable to one another. This is because the implementation platforms, the prime field sizes used and also the functionality of both the processor presented and the previous implementations differ somewhat. However, a broad overview of the approaches used and performance results is given for each of these.

In [20] the authors used an FPGA systolic array Montgomery multiplier and requires a relatively high number of clock cycles to compute a modular multiplication. They reported a 160-bit scalar point multiplication time of 14.414 ms using a Xilinx Virtex-E V1000E-BG580-8 FPGA running at 91.308 MHz and using 6 055 slices. The relatively slow point multiplication time is a reflection of the relatively large number of clock cycles needed to perform both Montgomery multiplication (bit-wise systolic) and therefore modular inversion, as Fermat's Little Theorem is used to perform this. As discussed in Section I, this is equivalent to performing a modular exponentiation, which is a very computationally intensive calculation to compute in comparison to using the unified algorithm described here.

In [19] Orlando and Paar proposed an architecture that uses a high-radix Montgomery multiplier and Fermat's Little Theorem to tackle the problem of modular inversion. It is stated that if it were possible to extract 100% throughput from the multiplier, then a 192-bit point multiplication time would take an estimated 3 ms, using the double-and-add algorithm and projective coordinates. This design was implemented on a Xilinx Virtex-E XCV1000E-BG680 FPGA running at 40 MHz and using 11 416 LUTs, 5 735 Flip-Flops, and 35 BlockRAMs. This also uses a predetermined prime modulus $p = 2^{192} - 2^{64} - 1$, limiting the flexibility of the design.

In [18], Satoh and Takano proposed an elliptic curve cryptographic processor using a Montgomery multiplier architecture comprising a Wallace tree and carry propagation adders. They implemented their architectures using a 0.13-μm CMOS standard cell ASIC library. A 256-bit scalar point multiplication over GF(p) takes 2.68 ms operating at a frequency of 137.7 MHz (using approximately 120 259 gates). A relatively large number of clock cycles are required to compute a Montgomery multiplication and modular inversion. Again, their work does not fully deal with the problem of modular inversion and this is calculated using Fermat's Little Theorem making it inherently more complex.

The 256-bit scalar point multiplication time of 3.86 ms presented in this paper is the only one reported for a GF(p) FPGA implementation in the literature to date. It is 1.18 ms slower than the implementation time reported in [18] but this is expected as ASIC implementations are traditionally much faster than designs implemented on FPGA, where flexibility is gained at the expense of higher clock speeds. It should also be noted that the implementations presented in [18]–[20] are not well-suited to ECCs wherein the point addition and doubling are performed using affine coordinates, and thus require a large number of modular inversions. This is because by using Fermat's Little Theorem to perform this would result in a very large number of clock cycles to perform a full scalar point multiplication. Therefore, the application area of the architectures in [18]–[20] is limited to ECCs using projective coordinates and is relatively narrow in comparison with the design presented here. Moreover, the processor created takes advantage of the new full-word multipliers described in Section III-C. Our design also exploits the unified inversion algorithm described and is more efficient than previous ECC designs, which use Fermat's Little Theorem to perform modular inversion. The inherent benefits of using these in the processor implementation include high throughput rates, as demonstrated above. In addition, the architecture described is also the only ECC processor to offer four different types of modular inversion (see Section III-A), thus broadening the flexibility and application areas of this.

IV. Conclusion

In this paper, a new hardware ECC processor architecture over $\mathrm{GF}(p)$ has been introduced. This can perform the main prime field arithmetic functions needed in these cryptosystems and in particular comprises modular inversion and multiplication components. It uses a combination of a new unified inversion algorithm and novel full-word multiplier architectures, which display impressive throughput rates when implemented efficiently in hardware. This has been demonstrated through FPGA implementation. In addition, the processor described is the only one to date that offer four different types of modular inversion allowing it to be applied to many different ECC applications.

References

[1] V. S. Miller, "Use of elliptic curves in cryptography," in *Proc. Adv. Cryptolog. (Crypto'85)*, 1986, pp. 417–426.

[2] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comput.*, vol. 48, pp. 203–209, 1987.

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[4] US National Institute of Standards and Technology (NIST), Cryptographic Toolkit Jun. 2002, pp. 83–84 [Online]. Available: Available: http://csrc.nist.gov/CryptoToolkit/kms/guideline-1.pdf

[5] Certicom Corporation, The Basics of ECC 2006 [Online]. Available: Available: http://www.certicom.com/index.php?action=res,ecc_faq

[6] G. Harper, A. Menezes, and S. Vanstone, "Public-key cryptosystems with very small key lengths," in *Proc. Adv. Cryptolog. (Eurocrypt'92)*, 1992, vol. 658, pp. 163–173.

[7] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck, "Fast key exchange with elliptic curve cryptosystems," in *Proc. Adv. Cryptolog. (Crypto'95)*, 1995, vol. 963, pp. 43–56.

[8] E. D. Win, A. Bosselaers, and S. Vandenberghe, "A fast software implementation for arithmetic operations in $\mathrm{GF}(2^n)$'," in *Proc. Adv. Cryptolog. (Asiacrypt'95)*, 1996, vol. 1163, pp. 65–76.

[9] J. Guajardo and C. Paar, "Efficient algorithms for elliptic curve cryptosystems," in *Proc. Adv. Cryptolog. (Crypto'97)*, 1997, vol. 1294, pp. 342–356.

[10] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," in *Proc. Adv. Cryptolog. (Asiacrypt'98)*, 1998, vol. 1514, pp. 51–65.

[11] R. Doud, "Hardware crypto solutions boost VPN," *EE Times*, pp. 57–64, Apr. 1999.

[12] A. Shamir and N. Van Someren, "Playing hide and seek with stored keys," in *Proc. 3rd Int. Conf. Fin. Cryptography*, 1999, pp. 118–124.

[13] G. Agnew, R. Mullin, I. Onyszchuk, and S. Vanstone, "An implementation of elliptic curve cryptosystems over F_2^{155}," *IEEE J. Select. Areas Commun.*, vol. 11, pp. 804–813, Jun. 1993.

[14] S. Sutikno, R. Effendi, and A. Surya, "Design and implementation of arithmetic processor F_2^{155} for elliptic curve cryptosystems," in *Proc. IEEE Asia-Pacific Conf. Circuits Syst. (APCCAS'98)*, Nov. 1998, pp. 647–650.

[15] G. Orlando and C. Paar, "A high-performance reconfigurable elliptic curve processor for $\mathrm{GF}(2^m)$," in *Proc. Cryptographic Hardware Embedded Syst. (CHES'00)*, Aug. 2000, pp. 41–56.

[16] S. Okada, N. Torii, K. Itoh, and M. Takenaka, "Implementation of elliptic curve cryptographic coprocessor over $\mathrm{GF}(2^m)$ on an FPGA," in *Proc. Cryptographic Hardware Embedded Syst. (CHES'00)*, Aug. 2000, pp. 25–40.

[17] *Standard Specifications for Public-key Cryptography*, IEEE Standard P1363, 2000 [Online]. Available: http://grouper.ieee.org/groups/1363

[18] A. Satoh and K. Takano, "A scalable dual-field elliptic curve cryptographic processor," *IEEE Trans. Comput.*, vol. 52, no. 4, pp. 449–460, Apr. 2003.

[19] G. Orlando and C. Paar*, A Scalable* $\mathrm{GF}(p)$ *Elliptic Curve Processor Architecture for Programmable Hardware*, C. K. Koc, D. Naccache, and C. Paar, Eds. : , 2001, pp. 356–371, CHES 2001, LNCS 2162.

[20] S. B. Ors, L. Batina, and B. Preneel, "Hardware implementation of elliptic curve processor over $\mathrm{GF}(p)$," in *Proc. 14th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'03)*, Jun. 2003, pp. 433–443.

[21] C. McIvor and J. McCanny, "Method of Calculating a Modular Inverse," British Patent Application No. 0412084.6, Filed, May 2004.

[22] E. Savas and C. K. Koc, "The Montgomery modular inverse—Revisited," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 763–766, Jul. 2000.

[23] C. K. Koc, T. Acar, and B. S. Kaliski, "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.

[24] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, pp. 519–521, 1985.

[25] C. D. Walter, "Montgomery exponentiation needs no final subtractions," *Electron. Lett.*, vol. 35, no. 21, pp. 1831–1832, Oct. 1999.

[26] B. S. Kaliski, "The Montgomery inverse and its applications," *IEEE Trans. Comput.*, vol. 44, no. 8, pp. 1064–1065, Aug. 1995.

[27] Xilinx, Inc., Xilinx Data Sheets 2005 [Online]. Available: Available: http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp

[28] G. M. de Dormale, P. Bulens, and J.-J. Quisquater, "An improved Montgomery modular inversion targeted for efficient implementation on FPGA," in *Proc. IEEE Int. Conf. Field-Programm. Technol. (FPT'04)*, Dec. 2004, pp. 441–444.

[29] A. A.-A. Gutub, A. Tenca, and C. K. Koc, "Scalable VLSI architecutre for $\mathrm{GF}(p)$ Montgomery modular inverse computation," in *Proc. IEEE Comput. Soc. Annual Symp. VLSI*, Apr. 2002, pp. 53–58.

[30] ——, "Efficient scalable hardware architecture for Montgomery inverse computation in $\mathrm{GF}(p)$," in *Proc. IEEE Workshop Signal Process. Syst. (SIPS'03)*, Aug. 2003, pp. 93–98.

[31] T. Zhou, X. Wu, G. Bai, and H. Chen, "New algorithm and fast VLSI implementation for modular inversion in galois field $\mathrm{GF}(p)$," in *Proc. IEEE Int. Conf. Commun. Circuits Syst. West Sino Expo.*, Jul. 2002, vol. 2, pp. 1491–1495.

[32] L. A. Tawalbeh and A. Tenca, "An algorithm and hardware architecture for integrated modular division and multiplication in $\mathrm{GF}(p)$ and $\mathrm{GF}(2^n)$," in *Proc. IEEE Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP'04)*, Sep. 2004, pp. 247–257.

Ciaran J. McIvor was born in Belfast, Northern Ireland, U.K., in 1978. He received the Bachelor of Science (Hon) degree in mathematics and the Ph.D. degree in electrical and electronic engineering from Queen's University, Belfast, Northern Ireland, U.K., in 2000 and 2005, respectively.

He is currently a Research Fellow at the Institute of Electronics, Communications, and Information Technology (ECIT) at Queen's. His main research interests include efficient algorithms, hardware architectures, and computer arithmetic techniques for public-key cryptography. He has numerous peer-reviewed conference and journal publications on these topics.

Máire McLoone (M'03) received the Master of Engineering degree (with distinction) in electrical and electronic engineering and the Ph.D. degree in digital signal processing from Queen's University, Belfast, Northern Ireland, U.K., in 1999 and 2002, respectively.

She is currently at the Institute of Electronics, Communications, and Information Technology, Queen's, in her third year of a 5-year Royal Academy of Engineering research fellowship conducting research into cryptographic algorithms and architectures for system-on-chip (SoC). She has authored a research book *System-on-Chip Architectures and Implementations for Private-Key Data Encryption* (Kluwer, 2003), and has over 30 peer-reviewed conference and journal publications. Her research interests include generic silicon architectures for symmetric and asymmetric cryptographic algorithms, security for wireless and ad hoc networks, hardware/software cryptographic SoC architectures and cryptography for constrained environments.

Dr. Mcloone was presented with a U.K. Science Engineering and Technology (SET) Student of the Year award for best electronic engineering student in 1999, and in 2000, she received a Beijing Institute of Technology (BIT) International Outstanding Student of the Year award. In 2004, she was presented with the Vodafone award for her work in high-speed data security at the Britain's Younger Engineers event held at the House of Commons, London. She is a member of the Institution of Electrical Engineers, U.K., and IACR.

John V. McCanny (M'86–SM'95–F'99) received the Bachelor's degree in physics from the University of Manchester, Manchester, U.K., the Ph.D. degree in physics from the University of Ulster, Ulster, U.K., and the D.Sc. (higher doctorate) degree in electrical and electronics engineering from Queen's University, Belfast, Northern Ireland, U.K., in 1973, 1978, and 1998, respectively

He is an international authority in the design of silicon integrated circuits for Digital Signal Processing; having made many pioneering contributions to this field. He has co-founded two successful high technology companies, Audio Processing Technology Ltd., and Amphion Semiconductor Ltd. a leading supplier of systems-on-chip (SoC) cores for video compression. He is currently Director of the Institute of Electronics, Communications and Information Technology at Queen's University, Belfast. He has published 250 major journal and conference papers, holds 25 patents and has published five research books.

Prof. McCanny is a Fellow of the Royal Society, the Royal Academy of Engineering, the Institution of Electrical Engineers ,and the Institute of Physics. He has won numerous awards, including a Royal Academy of Engineering Silver Medal for outstanding contributions to U.K. engineering leading to commercial exploitation (1996), an IEEE Third Millennium medal and the Royal Dublin Society/Irish Times Boyle Medal (2003). In 2002, he was awarded a CBE for his contributions to engineering and higher education.