Available online at www.sciencedirect.com
SCIENCE DIRECT®
Microprocessors and Microsystems 28 (2004) 253–260
MICROPROCESSORS AND MICROSYSTEMS
www.elsevier.com/locate/micpro

An FPGA implementation of a GF(p) ALU for encryption processors

Alan Daly[a,*,1], William Marnane[a], Tim Kerins[a], Emanuel Popovici[b]

[a]*Department of Electrical and Electronic Engineering, University College Cork, Cork, Ireland*
[b]*Department of Microelectronic Engineering, University College Cork, Cork, Ireland*

Received 20 October 2003; revised 24 February 2004; accepted 4 March 2004

Abstract

Secure electronic and internet transactions require public key cryptosystems to establish and distribute shared secret information for use in the bulk encryption of data. For security reasons, key sizes are in the region of hundred's of bits. This makes cryptographic procedures slow in software. Hardware accelerators can perform the computationally intensive operations far quicker. Field-Programmable Gate Arrays are well-suited for this application due to their reconfigurability and versatility. Elliptic Curve Cryptosystems over GF(p) have received very little attention to date due to the seemingly more attractive finite field GF(2^m). However, we present a GF(p) Arithmetic Logic Unit which can perform 160-bit arithmetic at clock speeds of up to 50 MHz.

Keywords: GF(p); Elliptic curve cryptosystems; Elliptic curve; Cryptography; Arithmetic

1. Introduction

As the popularity of mobile internet devices increases, so too does the need for secure and reliable electronic communication over insecure channels. Encryption provides the confidentiality, authentication, data integrity and non-repudiation required for electronic transactions. Bulk data is generally encrypted using private key systems, where both parties share the same common secret key. However, the establishment and exchange of these secret keys is normally achieved via public key cryptosystems. Generally, these public key systems are more computationally intensive and slower than their private key counterparts [1].

Field-Programmable Gate Arrays (FPGAs) are an ideal platform to provide hardware arithmetic acceleration for use in many cryptographic applications. Their reconfigurability means that they can be re-programmed to perform the more computationally intensive operations of a range of ciphers depending on security and application requirements.

Elliptic Curve Cryptosystems (ECC) were independently proposed in the mid-eighties by Victor Miller [2] and Neil Koblitz [3] as an alternative to existing public key systems such as RSA and DSA. ECC is quickly establishing itself due to it's potential to provide equivalent security to existing public key cryptosystems at reduced key sizes. This is particularly attractive for use in constrained applications such as smart cards, mobile phones and palm-top devices, where chip area and memory storage are limited resources. The security of this scheme relies on the difficulty of the discrete logarithm problem in the group formed by the points on an elliptic curve over a finite field. Unlike the ordinary discrete logarithm problem, no sub-exponential algorithm is known to date to solve the discrete logarithm problem on a suitably chosen elliptic curve. It is estimated that a 160-bit ECC cryptosystem has security equivalent to RSA with a bit length of 1024 [4].

Two types of finite field are popular for use in elliptic curve public key cryptography: GF(p) with p a 'large' prime, and GF(2^m) with m a positive integer. Few Elliptic curve cryptosystems over GF(p) have been reported in the literature to date, due to the requirement of carry propagation in the addition operation [5–10]. Addition in GF(2^m) is performed bitwise modulo 2, and thus has a shorter critical path. However, the more complex arithmetic operations of multiplication, inversion and division over GF(2^m) do not map well to the underlying FPGA structure. The resulting large combinational structures can reduce performance as significantly as the GF(p) carry-chain adder

* Corresponding author. Tel.: +353-21-4903156; fax: +353-21-4271698.
E-mail address: aland@rennes.ucc.ie (A. Daly).
[1] http://rennes.ucc.ie/~aland

0141-9331/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2004.03.006

structures which are optimised on FPGA. Many of the existing ECC implementations over GF(2^m) are also constrained to fixed register sizes once the field has been selected and the target device has been configured. Whereas, a GF(p) implementation with a register size of m bits can operate with any prime p up to $(2^m - 1)$ without reconfiguration. A GF(2^m) processor which can operate on different key sizes without reconfiguration has previously been presented in Ref. [11], but it is the belief of the authors that the more versatile GF(p) processors can be implemented as efficiently as GF(2^m) processors on reconfigurable platforms.

2. Elliptic curve cryptography over GF(p)

An elliptic curve over the finite field GF(p) is defined as the set of points (x, y), which satisfy the elliptic curve equation

$$y^2 = x^3 + ax + b$$

where x, y, a and b are elements of the field, and $4a^3 + 27b^2 \neq 0$.

To encrypt data, it is represented as a point $P(x_P, y_P)$ on the chosen curve over the finite field. The fundamental encryption operation is point scalar multiplication, i.e. point P is added to itself k times, to get point $Q(x_Q, y_Q)$.

$$Q = kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}$$

Recovery of k, through knowledge of the elliptic curve equation, base point P, and end point Q, is called the Elliptic Curve Discrete Logarithm Problem (ECDLP). It is relatively easy to calculate Q given k and P, but very difficult to determine k given P and Q.

In order to compute kP, a double and add method is used with k represented in binary form and scanned right to left from LSB to MSB, performing a double at each step and an addition if k_i is 1. Therefore the multiplier will require $(m - 1)$ point doublings and an average of $(m - 1)/2$ point additions, where m is the bitlength of the field prime, p. These may be performed concurrently, resulting in a total execution time per point multiplication of $(m - 1)$ times that of a point doubling or addition (whichever is greater).

The point addition/doubling formulae in *affine coordinates* are given below. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then $P_3 = (x_3, y_3) = P_1 + P_2$ is given by:

$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$
$$\lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2 \text{ (Point Addition)} \\ \dfrac{3x_1^2 + a}{2y_1} & \text{if } P_1 = P_2 \text{ (Point Doubling)} \end{cases}$$

Point multiplication operations on elliptic curves over GF(p) are performed by modular addition, subtraction, multiplication and inversion. In *affine coordinates*, point addition requires three modular multiplications, 1 inversion and 6 addition/subtraction operations. Point doubling requires 4 modular multiplications, 1 inversion and 7 additions/subtractions.

Different coordinate systems can be used to represent elliptic curve points, and it is possible to eliminate inversions by representing the points in *projective coordinates*. However, this results in a large increase in the number of modular multiplications required per point scalar multiplication (16 per point addition, 10 per point doubling) [4]. GF(p) ECC processors have been presented in Refs. [5–9] which perform point scalar multiplication in projective coordinates using only modular multiplication.

It should be noted that when performing point scalar multiplication in projective coordinates, inversion over GF(p) is still necessary to convert the final result back to affine coordinates. This can be achieved through modular exponentiation given by Fermat's Little theorem which states $A^{-1} = A^{M-2} \bmod M$, and requires on average another $(3(m - 1))/2$ modular multiplications.

An ECC Processor which performs modular inversion by the Extended Euclidean Algorithm (EEA) is presented in Ref. [10]. This processor uses a redundant carry-save representation in its computations which is not well suited to the sign detection and magnitude comparison requirements of the EEA. This results in the time per inversion being 70 times that of a multiplication.

3. Modular arithmetic functions

All arithmetic in the Galois Field GF(p) is performed modulo the field prime, p. In order to perform the point addition and doubling computations outlined in Section 2, modular addition, subtraction, multiplication and inversion/division are required. In this section the modular arithmetic functions are listed, and algorithms and architectures for each are presented. In Section 4, these architectures are combined into a single GF(p) Arithmetic Logic Unit which is capable of performing any of the required operations. In the following descriptions, the m-bit modulus is represented by M, and for most applications will be equal to the field prime p.

3.1. Modular addition

The modular addition operation adds two inputs, A and $B(A, B \in [0, M - 1])$, and subtracts the modulus M from the sum if $(A + B) \geq M$. An architecture to perform modular addition is illustrated in Fig. 1. The carry-propagate adders used must be at least $(m + 1)$-bits long to represent the intermediate result $(A + B)$, which could be greater than the m-bit modulus. To subtract M, a carry-propagate adder is

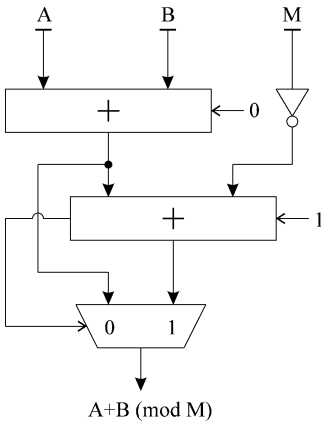

Fig. 1. Modular addition.

used with the sum and bitwise inverted modulus as inputs, and the carry-in tied to '1', thus performing two's compliment subtraction. The carry-out of this adder is then an indication that $(A + B)$ is greater than or equal to M. This signal controls the multiplexer which selects whether $(A + B)$ or $((A + B) - M)$ is the correct result.

3.2. Modular subtraction

To perform modular subtraction, input B is bitwise inverted and added to input A with a carry-in of '1'. If the result is negative (i.e. the carry-out is low) then the modulus must be added to produce an output in the range $[0, M - 1]$. An architecture to perform modular subtraction is illustrated in Fig. 2. In this architecture, the value of $((A - B) + M)$ is computed while the relative magnitude of A and B is being determined. By this method, both possible results are computed in slightly more time than a single m-bit addition, and the correct result is selected depending on the carry out bit of the $(A - B)$ stage. This eliminates the necessity to await a full m-bit magnitude comparison before deciding whether to add M or not.

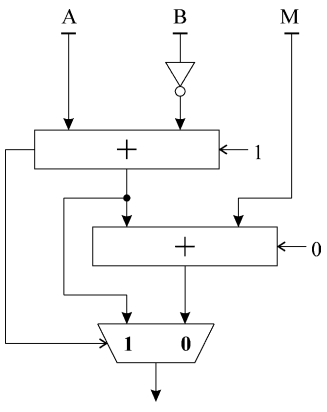

Fig. 2. Modular subtraction.

3.3. Modular negation

Modular negation may be performed by using the modular subtraction architecture illustrated in Fig. 2. Using only input B, and setting input A to zero, the input is negated modulo M :

$$R = (-B) \bmod M$$

3.4. Modular doubling

Modular doubling may be performed using the modular addition hardware architecture of Fig. 1. By setting both inputs to 'A', the output is given by $2A \bmod M$. Modular doubling is required in the Montgomery domain inverse operation as described in Section 3.8.

3.5. Modular multiplication

An efficient algorithm for multiplying two integers modulo M was proposed by Montgomery in 1985 [12]. It computes the Montgomery product through an iterative process of simple additions and right-shifts. This eliminates the need for trial division by M, and keeps the intermediate result bounded by $(m + 2)$-bits throughout the calculation. The Montgomery modular product is given by the expression:

$$R = \mathrm{MonPro}(A, B, M) = AB2^{-m} \bmod M$$

To obtain the correct modular product, this result must be Montgomery multiplied again by $(2^{2m} \bmod M)$ to remove the extra factor of 2^{-m}. The algorithm is given in Algorithm 1, and a hardware architecture is given in Fig. 3. Note that the bitlength of the adders used must be equal to $(m + 2)$ to manipulate the intermediate result at each iteration. The output however, is guaranteed to be in the range $[0, 2M - 1]$. One modular correction is then required to guarantee the output is in the range $[0, M - 1]$, taking only 1 extra clock cycle.

The number $\hat{A}$ is said to be the Montgomery domain representation of A if it is expressed as:

$$\hat{A} = \mathrm{MonPro}(A, (2^{2m} \bmod M), M) = A2^m \bmod M$$

Algorithm 1
Montgomery multiplication

MonPro(A, B, M)
Input: $A, B \in [0, M - 1]$ and M
Output: R, where $R = AB2^{-m} (\bmod M)$ and $R \in [0, 2M - 1]$

```
01  R_0 ⇐ 0;
02  for i = 0 to (m − 1) do
03      q_i ⇐ (R_i + b_i A) mod 2;
04      R_(i+1) ⇐ (R_i + b_i A + q_i M)/2;
05  end for;
06  return R_m;
```

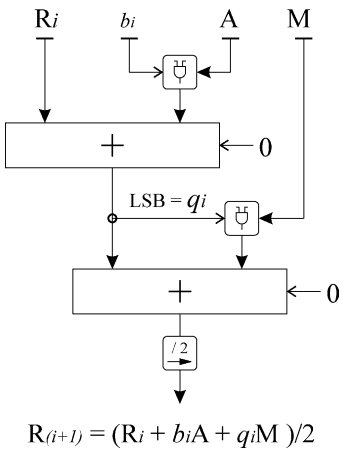

Fig. 3. Modular multiplication.

To map the number back to the integer domain, it must be Montgomery multiplied by '1':

$$A = \text{MonPro}(\hat{A}, 1, M) = (A2^m)2^{-m} \bmod M$$

The benefit of representing numbers in this way is that the Montgomery product of two Montgomery domain numbers is itself in the Montgomery domain, and hence the extra tasks of mapping to and from the Montgomery domain can be negligible when performing many operations in the domain.

3.6. Modular correction

Modular correction is the process of subtracting the modulus from an input if it is greater than or equal to the modulus. The output is in the range $[0, M-1]$, assuming the input is in the range $[0, 2M-1]$. This adjustment is necessary after a modular multiplication to bound the result before further calculations. Modular correction may be performed using the modular addition hardware architecture in Fig. 1, by setting input 'B' to zero. The input 'A' is then corrected if it is greater than or equal to the modulus.

3.7. Modular reduction

Modular reduction is the process of halving the input modulo M. If the input is odd (i.e. the LSB = '1') then the modulus must be added before right-shifting to divide by 2. Reduction is required in phase II of the modular inversion algorithm as described in Section 3.8. The multiplication architecture of Fig. 3 may be used to output the result $R = A2^{-1} \bmod M$, by setting R_i to zero and b_i to '1'.

3.8. Modular inversion

The modular multiplicative inverse $A^{-1} \pmod M$ of an integer A, exists if, and only if, A and M are relatively prime, and hence their greatest common divisor (gcd) = 1. The Montgomery inverse of A is defined as:

$$R = A^{-1}2^m \bmod M$$

An algorithm to compute the inverse and Montgomery inverse of an integer was described by Kaliski in Ref. [13]. The algorithm is split into two phases. Phase I is presented here in Algorithm 2 and is termed the 'Almost Montgomery Inverse' [14]. The AlmMonInv(A, M) operation takes k clock cycles to output the result $R = A^{-1}2^k \bmod M$, where k is between m and $2m$. Phase II repeatedly reduces the intermediate result modulo M to produce either the actual inverse, $A^{-1} \bmod M$, or the Montgomery inverse, $A^{-1}2^m \bmod M$.

In order to resist cryptanalysis attacks based on timing, each inversion operation should take the same (maximum) number of clock cycles to complete, independent of the inputs. Therefore, when the almost Montgomery operation is complete, another $(2m - k)$ clock cycles are spent doubling the result modulo M as described in Algorithm 3. The output of the Montgomery Domain Inverse operation is then given by $R = A^{-1}2^{2m} \bmod M$.

Algorithm 2
Almost Montgomery inverse

AlmMonInv(A, M)
Input: $A \in [1, M-1]$ and M
Output: R and k, where $R = A^{-1}2^k \pmod M$ and $m \le k \le 2m$

```
01  U ⇐ M; V ⇐ A; R ⇐ 0; S ⇐ 1; k ⇐ 0;
02  while (V > 0)
03     if (U even) then U ⇐ U/2; S ⇐ 2S;
04     else if (V even) then V ⇐ V/2; R ⇐ 2R;
05     else if (U > V) then U ⇐ (U − V)/2; R ⇐ R + S; S ⇐ 2S;
06     else if (V ≥ U) then V ⇐ (V − U)/2; S ⇐ S + R; R ⇐ 2R;
07     k ⇐ k + 1;
08  if (R ≥ M) then R ⇐ R − M;
09  return R ⇐ M − R;
10  return k;
```

Algorithm 3
Montgomery domain inverse

MonDomInv(A, M)
Input: $A \in [1, M-1]$ and M
Output: R, where $R = A^{-1}2^{2m} \pmod M$

```
01  R, k ⇐ AlmMonInv(A, M);
02  while (k < 2m)
03     R ⇐ 2R;
04     if (R ≥ M) then R ⇐ R − M;
05     k ⇐ k + 1;
06  return R;
```

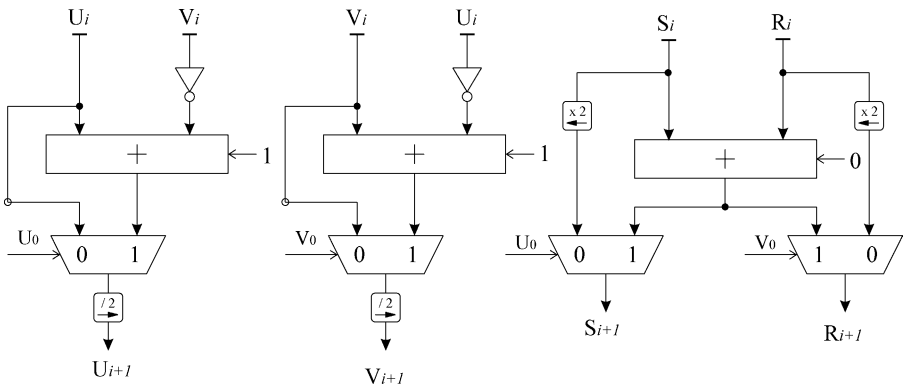

Fig. 4. Modular inversion.

Interestingly, if the input $\hat{A}$ to the MonDomInv$(\hat{A}, M)$ operation is in the Montgomery domain, so too is the output.

$$\begin{aligned} R &= \text{MonDomInv}(\hat{A}, M) \\ &= \text{MonDomInv}(A2^m, M) \\ &= (A\ 2^m)^{-1}2^{2m} \bmod M \\ &= A^{-1}2^m \bmod M \\ &= \hat{A}^{-1} \bmod M \end{aligned}$$

The inversion architecture of Fig. 4 performs lines 03-06 of Algorithm 2. Two m-bit subtractors and an m-bit adder are required in the architecture, along with multiplexers controlled by the parity and relative magnitude of temporary registers U and V. Four registers in total are required in the architecture; U, V, R and S.

Lines 08 and 09 of Algorithm 2 can be combined into a single modular subtraction performed using the architecture of Fig. 2, by setting input A to the modulus, and input B to the intermediate result, R. Lines 03 and 04 of Algorithm 3 perform modular doubling and can be realised using the addition architecture of Fig. 1 as described in Section 3.6.

3.9. Modular division

Very few dedicated GF(p) modular division implementations have been reported in the literature to date [15]. However, modular division may be achieved through modular inversion followed by modular multiplication.

As stated in Section 3.8, the Montgomery domain inverse operation takes a constant $2m$ clock cycles in order to resist timing cryptanalysis attacks. In order to compute the actual inverse $(A^{-1} \bmod M)$ of an integer input A, the same number of clock cycles must be spent reducing the result modulo M, meaning a total of $4m$ clock cycles per integer inversion. However, one Montgomery modular multiplication takes m clock cycles, while simultaneously reducing the product. After performing the Montgomery domain inversion on an input C, the output equals $C^{-1}2^{2m} (\bmod M)$.

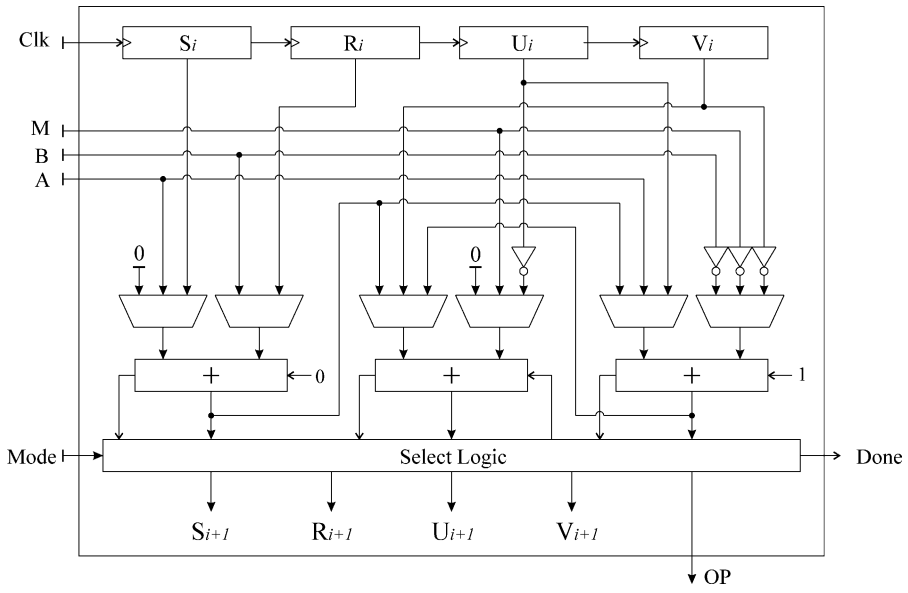

Fig. 5. GF(p) Arithmetic logic unit.

Table 1
Input selection for the 3 adders for each mode of operation

Mode	Adder 1			Adder 2			Adder 3		
	IP_A	IP_B	C_i	IP_A	IP_B	C_i	IP_A	IP_B	C_i
Addition	A	B	0	OP_1	$\overline{M}$	1	*	*	*
Subtraction	*	*	*	OP_3	M	0	A	$\overline{B}$	1
Multiplication	b_iA	R_i	0	OP_1	q_iM	0	*	*	*
Inversion	S_i	R_i	0	V_i	$\overline{U_i}$	1	U_i	$\overline{V_i}$	1

By Montgomery multiplying the result by A, the output equals $A/C\ 2^m \pmod M$, and by further multiplying by B, the result is $\frac{AB}{C} \pmod M$. Thus, by performing a constant $2m$ clock cycles per Montgomery inversion, two multiplications may be performed in the time that would otherwise be wasted in correcting the inversion result.

It is worth noting that this ($\frac{AB}{C}$) calculation may be performed on Montgomery domain numbers to output a result in the Montgomery domain also.

4. The GF(p) arithmetic logic unit

The proposed GF(p) ALU illustrated in Fig. 5 combines all the functions listed in Section 3 into one architecture. Three $(m+2)$-bit carry-propagate adders are used, and two levels of switching to select the inputs and outputs. The switching multiplexers are controlled by operation mode and signals such as carry bits, parity bits, and b_i, the LSB of B, generated by a shift-register for the multiplication operation. The carry-in of an adder is set to '1' when a two's compliment subtraction is required. Table 1 lists the inputs for each of the three adders for the four fundamental operations. The remaining operations can be implemented using those fundamental operations. Also required are four registers to store intermediate variables U, V, R and S.

The almost Montgomery inverse calculation takes between m and $2m$ clock cycles to complete. The 'Done' signal goes high when $U = V$, and the correct output is latched on the next rising clock edge. The Montgomery multiplication operation takes m clock cycles to complete. The output must be registered on the correct rising clock edge determined by an external counter. Modular reduction, addition and subtraction (and hence correction, doubling and negation) take only 1 clock cycle to complete.

It should be noted that the GF(p) ALU presented could also perform ECC point operations in projective coordinates as it is not limited to one coordinate system, thus making it even more versatile. However, unless input and output data to the ALU were already in projective coordinates, the affine operations can be performed in fewer clock cycles.

Procedure 1
Generic ($\frac{AB}{C}$) operation

$ABC(A, B, C, M)$
Input: $A, B, C \in [1, M-1]$ and M, where g.c.d.$(C, M) = 1$
Output: R, where $R = \frac{AB}{C} \pmod M$

01 $R \Leftarrow \text{MonDomInv}(C, M)$;
02 $R \Leftarrow \text{MonPro}(R, A, M)$;
03 if $(R \geq M)$ then $R \Leftarrow R - M$;
04 $R \Leftarrow \text{MonPro}(R, B, M)$;
05 if $(R \geq M)$ then $R \Leftarrow R - M$;
06 return R;

5. GF(p) ALU application example

As an example, the generic operation ($\frac{AB}{C}$) was implemented utilizing the GF(p) ALU and tested on a Celoxica RC1000 prototyping PCI card. The card includes a Xilinx Virtex2000e FPGA and 8 Mb of SRAM, and communicates with the host PC over a PCI interface.

The procedure for the ($\frac{AB}{C}$) operation is given in Procedure 1. All the required functions to perform the operation can be performed in the GF(p) ALU. Step 01 is performed in inversion, subtraction and addition modes as described in Section 3.8, steps 02 and 04 in multiplication mode and steps 03 and 05 in addition mode.

The generic operation ($\frac{AB}{C}$) can be used to perform inversion, division, squaring and multiplication. These are required in the point addition and point doubling operations outlined in Section 2.

6. Results

Post-Synthesis and Post-Place and Route results for the GF(p) ALU are presented in Table 2. The target device for these results was the Xilinx Virtex2 xc2v2000-6 FPGA. The post-synthesis speed figures indicate clock frequencies that the ALU could operate at as a component within

Table 2
Results for the GF(p) ALU on Xilinx Virtex2 xc2v2000-6 FPGA

Word length	Post-synthesis results (ALU component only)				Post-P and R
	Slices	% of device	Equiv gates	Max Freq (MHz)	Max Freq (MHz)
32-bit	371	3.5	7583	74.63	68.05
64-bit	723	6.7	14,688	67.24	61.84
96-bit	1103	10.3	21,654	61.18	51.10
128-bit	1617	15.0	29,489	56.93	45.17
160-bit	1854	17.2	35,835	52.33	40.28
192-bit	2347	21.8	45,654	50.64	35.99
224-bit	2721	25.3	53,183	45.62	33.06
256-bit	3109	28.9	60,805	44.42	31.92

Table 3
Implementation results for the application example of $(\frac{AB}{C})$ mod M

Word length	Post-place and route results $(\frac{AB}{C})$ Application)			
	Slices	% of device	Equiv gates	Max Freq (MHz)
32-bit	751	3.9	27,565	36
64-bit	1415	7.4	38,513	30
96-bit	2097	10.9	49,602	27
128-bit	2766	14.4	60,743	23
160-bit	3434	17.9	71,758	21
192-bit	4135	21.5	83,236	19
224-bit	4808	25.0	94,126	16
256-bit	5477	28.5	105,300	14

Target device: Xilinx Virtex2000e-6 FPGA on Celoxica RC1000 board.

another system. The post-place and route results were acquired from a design which included memory interfacing shift registers to input and output data in 32-bit words, and also signal routing to and from I/O pins on the FPGA. This accounts for the performance difference between columns 5 and 6 in Table 2.

Speed and area results for the $(\frac{AB}{C})$ implementation are given in Table 3. The target device is the Xilinx Virtex 2000e chip. These results have been verified on the Celoxica RC1000 board.

The differences in performance results between Tables 2 and 3 are explained by the lower specification Virtex E device used for implementation on the Celoxica board. Also, memory interfacing and communications over the PCI bus limit the performance further.

Using the ALU, and by performing the affine coordinate calculations in the Montgomery domain, point addition will require exactly $(5m + 9)$ clock cycles, and point doubling will require exactly $(6m + 12)$ clock cycles. Since the double and add algorithm performs point additions and doublings concurrently, the greater of the two will dictate the computation time per point operation. Therefore, far fewer clock cycles will be required to perform point scalar multiplications using the GF(p) ALU in affine coordinates than in projective coordinates (approximately 6m clock cycles compared with $16m$).

At a clock speed of 91.3 MHz, the systolic multiplier of Örs et al. takes $(3m + 4)$ clock cycles per Montgomery modular multiplication, $((9/2)m^2 + 6m)$ cycles per inversion, $(40m + 38)$ per point doubling and $(42m + 56)$ per point addition [6]. This gives an execution time of 70 and 74 μs per point doubling and addition respectively for a 160-bit design.

Table 4
Clock cycles and execution time for ECC operations in GF(p) ALU

Operation	No. of clock cycles	Execution time (μs)	
		50 Mhz Clock	20 MHz Clock
Mont multiplication	$m + 1$	3.22	8.05
Inversion (MonDomInv)	$2m$	6.40	16.00
Point addition	$5m + 9$	16.18	40.45
Point multiplication	$6m + 12$	19.44	48.60

The design presented here, operating at 50 MHz on the Virtex2 FGPA will perform the same operations in 19.44 and 16.18 μs respectively, and on the VirtexE FPGA at 20 MHz, will perform the operations in 48.6 and 40.45 μs. Execution times for the ECC operations are summarised in Table 4.

7. Conclusions

Very few GF(p) arithmetic processors have been reported in the literature to date. Those that have been reported, use projective coordinates to perform point scalar multiplication and focus mainly on the modular multiplication operation, neglecting modular inversion and division. In projective coordinates, point addition requires approximately $16m$ clock cycles and point doubling requires approximately $10m$ clock cycles.

Here, a versatile GF(p) Arithmetic Logic Unit capable of performing all necessary modular calculations for an ECC implementation in affine (or projective) coordinates has been presented.

Post-synthesis results have indicated that a 160-bit GF(p) ALU which could be used in an ECC processor with security equivalent to that of 1024-bit RSA, could operate at a clock frequency of 50 MHz in a carefully designed system. The design has been successfully verified and tested on a Xilinx Virtex2000e-6 device at a frequency of 20 MHz.

Acknowledgements

This work is funded by a research innovation project from Enterprise Ireland.

References

[1] B. Schneier, Applied Cryptography, second ed., Wiley, New York, 1996.
[2] V.S. Miller, Use of elliptic curves in cryptography, Adv. Cryptogr. Crypto'85 218 (1985) 417–426.
[3] N. Koblitz, Elliptic curve cryptosystems, Math. Comp. 48 (1987) 203–209.
[4] I. Blake, G. Seroussi, N. Smart, Elliptic Curves in Cryptography, London Mathematical Society Lecture Note Series 265, Cambridge University Press, 2000.
[5] G. Orlando, C. Paar, A Scalable GF(p) elliptic curve processor architecture for programmable hardware. Cryptographic Hardware and Embedded Systems—CHES 2001, (LNCS 2162), May 2001, pp. 348–363.
[6] S.B. Örs, L. Batina, B. Preneel, J. Vandewalle, Hardware implementation of elliptic curve processor over GF(p), Proceedings of

the Application-Specific Systems, Architectures, and Processors—ASAP, 2003, pp. 433–443.
[7] J. Goodman, A. Chandrakasan, An energy-efficient reconfigurable public-key cryptography processor, IEEE J. Solid-State Circuits 36 (11) (2001) 1808–1820.
[8] A.A. Gutub, M.K. Ibrahim, High radix parallel architecture for GF(p) elliptic curve processor, IEEE Conf. Acoustics Speech Signal Process.—ICASSP (2003) 625–628.
[9] M. Feldhofer, T, Trathnigg, B. Schnitzer, Self-timed arithmetic unit for elliptic curve cryptography, Proceedings of Euromicro Symposium on Digital System Design—DSD, 2002, pp. 347–350.
[10] J. Wolkerstorfer, Dual-field arithmetic unit for GF(p) and GF(2^m), Cryptographic Hardware and Embedded Systems—CHES 2002, (LNCS 2523), August 2002, pp. 500–514.
[11] T. Kerins, E. Popovici, W. Marnane, P. Fitzpatrick, Fully parameterizable elliptic curve cryptography processor over GF(2^m), Field-Programmable Logic and Applications—FPL 2002, (LNCS 2438), September 2002, pp. 750–759.
[12] P.L. Montgomery, Modular multiplication without trial division, Math. Comput. 44 (1985) 519–521.
[13] B.S. Kaliski Jr., The Montgomery inverse and it's applications, IEEE Trans. Comput. 44 (8) (1995) 1064–1065.
[14] E. Savas, Ç.K. Koç, The Montgomery modular inverse—Revisited, IEEE Trans. Comput. 49 (7) (2000) 763–766.
[15] A. Daly, W. Marnane, T. Kerins, E. Popovici, Fast modular division for application in ECC on reconfigurable logic. Field-Programmable Logic and Applications—FPL 2003, (LNCS 2778), 2003, pp. 786–795.

Alan Daly received BE (Elec) degree in electrical and electronic engineering from University College Cork, Ireland in 2000. He is currently working towards his PhD degree. His primary research interests are reconfigurable logic devices and applications, cryptography, and large integer arithmetic.

William Peter Marnane received BE degree in electrical engineering from National University of Ireland, Cork in 1984 and the DPhil degree from the University of Oxford in 1989. He is a senior lecturer in the Department of Electrical and Electronic Engineering at the National University of Ireland, Cork since 1999. His research interests include digital design for DSP, coding theory and cryptography.

Tim Kerins received his BSc degree in physics from University College Cork, Ireland in 2000. He is currently working towards his PhD degree in electrical engineering at University College Cork. His primary research areas are flexible architectures for the implementation of cryptographic algorithms based over Galois fields and the efficient implementation of public key cryptography protocols based on elliptic curves.

Emanuel Mihai Popovici received Dipl. Ing. Degree in computer engineering from University Politehnica Timisoara, Romania, in 1997 and the PhD degree in microelectronic engineering from the National University of Ireland, Cork, in 2002. He has been a Lecturer in the Department of Microelectronic Engineering at the National University of Ireland, Cork since 2002. His research interests include coding theory, cryptography and their applications, design automation and test.