A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$*

Gerardo Orlando[1] and Christof Paar[2]

[1] General Dynamics Communication Systems
77 A St., Needham MA 02494-2892, USA
gerardo.orlando@gd-cs.com
[2] ECE Department, Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609, USA
christof@ece.wpi.edu

Abstract. This work proposes a processor architecture for elliptic curves cryptosystems over fields $GF(2^m)$. This is a scalable architecture in terms of area and speed that exploits the abilities of reconfigurable hardware to deliver optimized circuitry for different elliptic curves and finite fields. The main features of this architecture are the use of an optimized bit-parallel squarer, a digit-serial multiplier, and two programmable processors. Through reconfiguration, the squarer and the multiplier architectures can be optimized for any field order or field polynomial. The multiplier performance can also be scaled according to system's needs. Our results show that implementations of this architecture executing the projective coordinates version of the Montgomery scalar multiplication algorithm can compute elliptic curve scalar multiplications with arbitrary points in 0.21 msec in the field $GF(2^{167})$. A result that is at least 19 times faster than documented hardware implementations and at least 37 times faster than documented software implementations.

1 Introduction

This work proposes a scalable elliptic curve processor architecture (ECP) which operates over finite fields $GF(2^m)$. One of its key features is its suitability for reconfigurable hardware. Unlike traditional VLSI hardware, reconfigurable devices such as Field Programmable Gate Arrays (FPGA) do not possess fixed functionality after fabrication but can be reprogrammed during operation. The scalability of the ECP architecture and the flexibility of reconfigurable hardware afford implementations the following benefits:

Architecture Efficiency. The complexity of finite field arithmetic architectures depends greatly on whether arithmetic for one specific field is being implemented, or for general finite fields. The most dramatic example is perhaps squaring in $GF(2^m)$ using standard basis. For a specific field, squaring can be performed in one clock cycle, whereas a general architecture usually

* This research was supported in part by NFS CAREER award CCR-9733246.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 41–56, 2000.

requires $m/2$ clock cycles (where $m \geq 160$ for elliptic curves cryptosystems) [BG89]. Consequently, one algorithmic option that we explore in this paper relies on the bit-parallel computation of squares, resulting in extremely efficient implementations. The use of reconfigurable hardware allows applications to use an optimized squarer for every finite field.

Scalability. Depending on the application, different levels of security may be required. The main factor that determines the security of elliptic curve cryptosystem is the size of the underlying finite field. For instance, NIST announced recently a list of curves ranging from 163–571 bits [NIS99]. Realizing such a wide operand range efficiently in traditional hardware is a major challenge, whereas the ECP's architectural scalability and the FPGAs reconfigurability allow optimized processor instantiations for any field size. Moreover, the fine-grained scalability of the ECP's architecture provides a wide range of time-area, performance-cost architectural options. Section 5 provides some examples.

Algorithm Agility. It is a design paradigm of modern security protocols that cryptographic algorithms can be negotiated on a per-session basis. With the proposed ECP, it is possible through reconfiguration to (1) switch algorithm parameters and (2) to switch to another type of public-key algorithm.

Resource Efficiency. The vast majority of security protocols use public-key algorithms in the beginning of a session for tasks such as entity authentication and key establishment and private-key algorithms for bulk data encryption after that. With reconfigurable platforms, it is possible to reuse the same device for both tasks.

The remainder of the paper is structured as follows. Section 2 summarizes the previous works on elliptic curve implementations. Section 3 provides the most crucial mathematical and algorithmic background needed to understand the ECP. Section 4 describes the ECP architecture and its main components. Section 5 describes prototype implementations and results. Section 6 summarizes the conclusions.

2 Previous Work

A number of software and hardware implementations have been documented for the computation of point multiplication, which is the basic operation used by elliptic curve cryptographic systems. Among the most significant hardware implementations are [AMV93,Ros98,GSS99,SES98]. The ones in [AMV93,SES98] use off-the-shelf processors to perform elliptic curve operations and accelerators to perform finite field arithmetic. The implementation in [AMV93] uses an ASIC accelerator and the one in [SES98] uses an FPGA accelerator. The implementations in [Ros98,GSS99] are standalone elliptic curve processors in FPGAs. Both [Ros98,GSS99] define roadmaps for full-size, secure elliptic curve implementations but do not document successful implementations of them.

The implementations in [AMV93,GSS99,SES98] use normal basis representation. They use bit-serial multipliers, which require about m clock cycles to

compute a multiplication in $GF(2^m)$ and compute squares with cyclic shifts. (The use of digit-serial multipliers, which are used in this work, is mentioned in [GSS99] but the documented implementations use bit-serial multipliers.)

The hardware implementation documented in [Ros98] uses standard basis representation. This implementation is suitable for composite fields $GF((2^u)^v)$ where $u * v = m$. Its core-processing element is a hybrid multiplier which computes a multiplication in v clock cycles. This multiplier is also used to compute squares. It should be pointed out that recent developments demonstrate that some forms of composite fields give rise to elliptic curves that posses cryptographic weaknesses [GHS00].

Among the best performing software implementations which are reported in open literature are [SOOS95,LD99]. The performance of these implementations, as demonstrated in Section 5, rival that of the traditional hardware implementations previously mentioned. The main reasons for their high performance are their use of very efficient algorithms that are optimized for modern processors and the availability of processors with wide words that operate at very high clock rates.

The elliptic curve processor architecture introduced in this work exhibits the features of the aforementioned hardware and software implementations. Its hardware architecture is scalable and its processing units, like the ones used by the software implementations, are programmable. In addition, its architecture is neither restricted to use polynomials on extension degrees of a special form, as is the case for [Ros98], nor it favors particular fields, as is the case for [AMV93,GSS99,SES98] that favor fields for which Gaussian normal bases exist. It is also, to the authors' knowledge, the only standalone elliptic curve processor architecture that has been rendered into a full-size, secure elliptic curve implementation in FPGA technology.

3 Mathematical Background

3.1 Elliptic Curves Algorithms and Choice of Field Representation

This section provides a brief description of the elliptic curve algorithms used by the elliptic curve processor (ECP). The first algorithm is the double-and-add algorithm for scalar multiplications using projective coordinates as defined in [P1398]. The other algorithm is the projective coordinates version of the Montgomery scalar multiplication method described in [LD99]. The distinctive characteristics of these two algorithms are that the double-and-add algorithm adds and doubles elliptic curve points, while the Montgomery method adds and doubles only the x coordinates of two points, P_1 and P_2, where $P_2 = P_1 + P$ and P is the point that is being multiplied. Since the relationship between P_1 and P_2 is maintained throughout the multiplication, the addition of P_1 and P_2 yields the point $2P_1 + P$. From this detail and Algorithm 2 in the Appendix, one can verify that the intermediate points P_1 obtained during the computation of kP correspond to the intermediate points obtained with the double-and-add algorithm. At the end of the multiplication process, the x coordinate of kP is given

by the x coordinate of P_1 and the y coordinate is derived from the x coordinates of P_1 and P_2 and from the coordinates of P. The two multiplication methods previously discussed are presented in Algorithm 1 and 2 in the Appendix. Note that these algorithms, as the rest of this document, assume that the elliptic curve equation is defined as $y^2 + xy = x^3 + ax^2 + b$. These algorithms also assume that the binary representation of k is given by $k = \sum_{i=0}^{l-1} k_i 2^i$ with $k_{l-1} \neq 0$. The computational complexity of these algorithms is summarized in Table 1.

Table 1. Complexity of point multiplication in $GF(2^m)$ $(a, b \neq 0)$

Complexity	Montgomery	Double-and-Add (average)
#Squares	$5(m-1)+3$	$7(m-1)+1$
#Mult.	$6(m-1)+10$	$10.5(m-1)+3$
#Inverses	1	1

From Table 1 it is clear that an efficient method for squaring will have a considerable impact on the overall performance. Through the use of reconfigurable hardware it is possible to compute a square in one clock cycle for any field order even though a standard basis representation is being used. It appears very difficult to achieve the same behavior with traditional ASIC hardware platforms. An alternative is a normal basis representation, but this comes at the cost of a more complex multiplication architecture. In particular, normal basis multipliers can be prohibitively expensive for fields for which optimum normal bases do not exist. For an ECP with flexible finite field support, normal basis representation appear not to be the best choice.

It is important to note that the point multiplication algorithms consist of a main function, the double_and_add or the montgomery_scalar_multiplication functions in the algorithms shown in the Appendix. These main functions call point addition, point multiplication, coordinate conversion, and other functions as subroutines. In turn, these subroutines call finite field arithmetic subroutines. This hierarchical view is helpful for understanding the processor architecture described in Section 4.

3.2 $GF(2^m)$ Field Arithmetic

This section provides a brief introduction to $GF(2^m)$ finite field arithmetic. The reader is referred to [LN94] for in-depth study of this topic.

For all practical purposes, the computation of elliptic curve point double and a point addition is realized with algorithms involving field additions, squares, multiplications, and inversions. This work considers arithmetic in fields of characteristic two, $GF(2^m)$, using a standard basis representation. This basis representation is also known as polynomial or canonical basis representations. A field $GF(2^m)$ is isomorphic to $GF(2)[x]/(F(x))$, where $F(x) = x^m + \sum_{i=0}^{t} f_i x^i$ is a

monic irreducible polynomial of degree m with coefficients $f_i \in \{0, 1\}$. Here each residue class is represented by the polynomial of least degree in its class.

A standard basis representation uses the basis defined by the set of elements $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$, where α is a root of the irreducible polynomial $F(x)$. In this basis, field elements are represented as polynomials in α of degree less than m with coefficients 0 or 1; for example, an element A is represented as $A = \sum_{i=0}^{m-1} a_i \alpha^i$ with coefficients $a_i \in \{0, 1\}$. In hardware, the field elements are represented by binary m-tuples as in $(a_{m-1}, a_{m-2}, \ldots a_0)$.

The addition of two elements requires the modulo 2 addition of the coefficients of the field elements. In hardware, a bit-parallel adder requires m XOR gates, and an addition can be generally computed in one clock cycle.

The squaring of a field element $A = \sum_{i=0}^{m-1} a_i \alpha^i$ is ruled by Equation (1). A bit-parallel realization of this squarer requires at most $(r-1)(m-1)$ gates [Wu99,PFSR99], where r represents the number of non-zero coefficients of the field polynomial.

$$A^2 \equiv \sum_{i=0}^{m-1} a_i \alpha^{2i} \bmod F(\alpha) \tag{1}$$

The multiplication of two field elements A and B can be expressed as shown in Equation (2). This equation is arranged so that it facilitates the understanding of the digit-serial multiplier used by the ECP. This multiplier is of the type introduced in [SP97], and it is described here in Section 4.

In Equation (2), B is expressed in k_D digits ($1 \leq k_D \leq \lceil m/D \rceil$) as follows: $B = \sum_{i=0}^{k_D-1} B_i \alpha^{Di}$, where $B_i = \sum_{j=0}^{D-1} b_{Di+j} \alpha^j$ and D is the digit size in bits. Note that when m/D is not an integer, B is extended to an integer number of digits ($k_D = \lceil m/D \rceil$) by setting its most significant coefficients to 0 ($b_m = b_{m+1} = \ldots = b_{k_D * D-1} = 0$).

$$\begin{aligned} AB &\equiv (A \sum_{i=0}^{k_D-1} B_i \alpha^{Di}) \bmod F(\alpha) \\ &\equiv (\sum_{i=0}^{k_D-1} B_i (A\alpha^{Di} \bmod F(\alpha))) \bmod F(\alpha) \end{aligned} \tag{2}$$

The ECP lacks inversion circuitry. This work recommends the computation of inversions with repeated multiplications using the algorithms described in [IT88,Van99]. These algorithms compute inverses with $\lfloor \log_2(m-1) \rfloor + W(m-1) - 1$ multiplications [BSS99], where $W(m-1)$ represents the number of non-zero coefficients in the binary representation of $m-1$.

4 Processor Architecture

To compute kP efficiently one needs a blend of efficient algorithms and hardware architectures. Efficient algorithms are needed to compute point multiplication

and field operations. One also needs a platform that supports the efficient computation of such algorithms. This work proposes a processor architecture optimized for the use of efficient elliptic curve algorithms, which is also well suited for implementations in reconfigurable hardware.

The elliptic curve processor (ECP), shown in Figure 1, consists of three main components. These components are the main controller (MC), the arithmetic unit controller (AUC), and the arithmetic unit (AU). The MC is the ECP's main controller. It orchestrates the computation of kP and interacts with the host system. The AUC controls the AU. It orchestrates the computation of point additions, point doublings, and coordinate conversions. The AU performs the $GF(2^m)$ field additions, squares, multiplications, and inversions under AUC control. For the point multiplication algorithms given in the Appendix, the MC executes the double_and_add and the montgomery_scalar_multiplication functions, the AUC performs all the other subroutines, and the AU is the hardware that computes the finite field operations.

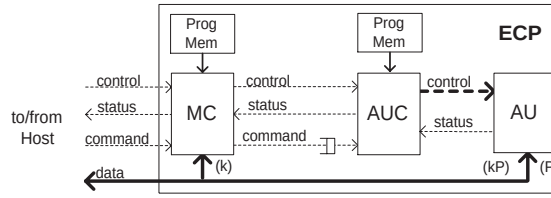

Fig. 1. Elliptic curve processor architecture

The following is a typical sequence of steps for the computation of kP in the ECP using the double-and-add algorithm and projective coordinates. First, the host loads k into the MC, loads the coordinates of P into the AU, and commands the MC to start processing. Then, the MC does its initialization, which includes finding the most significant non-zero coefficient of k. The MC then commands the AUC to perform its initialization, which includes the conversion of P from affine to projective coordinates. During the computation of kP, the MC scans one bit of k at time starting with the second most significant coefficient and ending with the least significant one. In each of these iterations, the MC commands the AU/AUC to do a point double. If the scanned bit is a 1, it also commands the AU/AUC to do a point addition. For each of these point operations, the AUC generates the control sequence that guides the AU through the computation of the required field operations. After the least significant bit of k is processed, the MC commands the AU/AUC to convert the result back to affine coordinates. When the AU/AUC finishes this operation, the MC signals to the host the completion of the kP operation. Finally, the host reads the coordinates of kP from the AU.

The ECP incorporates a set of techniques that maximizes resource utilization and speed. The most evident feature is concurrency. The ECP uses two loosely

coupled controllers, the MC and the AUC controllers, that execute their respective operations concurrently. These are very simple processors that execute one instruction per clock cycle. The AU uses concurrency. The AU incorporates a multiplier, a squarer, and a register file, all of which can operate in parallel on different data.

Another technique is pipelining. The regular architecture of the ECP allows it to use pipeline stages to reduce the critical path delay of the hardware and thus increase its operational frequency. The ECP incorporates pipelining in the AU and assures its maximum utilization with the AUC. The AUC maximizes pipeline utilization by minimizing pipeline fills and flushes. For example, the AUC can start loading operands for the next multiplication before the current one finishes.

The last main technique is the use of a large register set. The ECP's large register set supports algorithms that rely on precomputations. There are many such algorithms. Here we consider two examples. An example is the fixed window point multiplication algorithm. This algorithm requires on average $m + 2^{w-1}$ point doubles, $\lfloor m/w \rfloor + 2^{w-1}$ point additions, and the storage of 2^w points. Another algorithm is an adaptation of a fixed base exponentiation method introduced in [BGMW93] for operations involving a fixed point. This algorithm requires on average $\lfloor m/w \rfloor + 2^w$ point additions, the storage of $\lceil m/w \rceil$ points, and no point doubles. In the previous expressions, w is the window size, which is a measure of the number of bits of k processed in parallel. It must be pointed out that these optimizations can be used with the projective coordinate equations for point double and point addition defined in [P1398] but not with the ones defined in [LD99]. As this later algorithm requires that the relationship $P2 = P1 + P$ be maintained throughout the point multiplication process, while the aforementioned optimizations rely on precomputing absolute multiples of a point; for example, $1P$, $2P$, ... , $(2^w - 1)P$.

To illustrate the benefits of precomputation, consider an implementation for $GF(2^{167})$ using the projective coordinates defined in [P1398] and $w = 4$. Compared to the traditional double-and-add algorithm, the fixed window algorithm is approximately 1.1 times faster and the fixed point algorithm is over 2.5 times faster.

4.1 Arithmetic Unit

The AU, shown in Figure 2, is the unit responsible for field arithmetic. It consists of a register file, a least significant digit first (LSD) multiplier, a squarer, an accumulator, and a zero test circuit. The AU arranges these components in a streamlined, pipelined configuration that exhibits low fan out. The architecture contains two feedback paths that allow fast availability of operands to the multiplier, the squarer, and the register file.

The AU components operate under AUC control. The AUC's control extends to all the components shown in Figure 2. This fine control allows the AUC to extract maximum throughput from the AU by paralleling functions and managing pipeline delays.

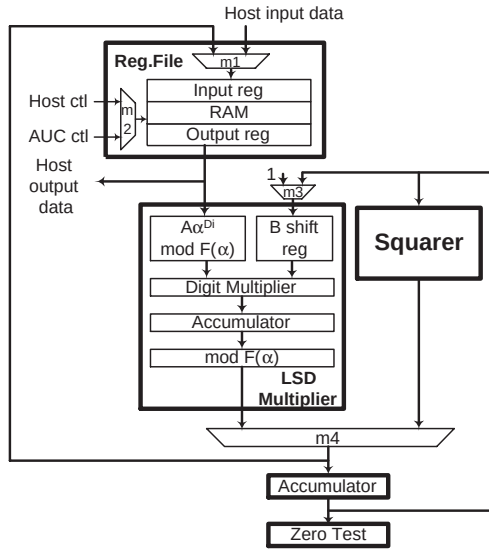

Fig. 2. Arithmetic unit

The multiplier and the squarer support the computation of field additions, squares, multiplications, and inversions. The addition of A and B is done by first computing $A * 1$ and then adding to it the product $B * 1$. The 1 operand can be supplied by the multiplexer m3 or the register file. This addition method exploits the ability of the LSD multiplier to accumulate products and eliminates the need for an adder. Field inversions are computed with repeated multiplications using the inversion algorithms described in [IT88,Van99].

The register file stores operands, precomputed values, and temporary values. It accepts input operands, such as the coordinates of P and the elliptic curve parameters a and b from the host system. It also accepts the results from the multiplier or the squarer selected by the multiplexer m4. It outputs operands to the multiplier and results to the host system. The basic components of the register file are the input and output registers and the RAM memory. RAM memory supports a large number of registers and the input and output register resolve access contentions to it.

The accumulator stores results from the multiplier and the squarer. It supplies the input operand of the squarer and one of the input operands of the multiplier. The zero test circuit, upon command, samples the content of the accumulator and compares it with zero. It maintains its result until another test is issued.

The AU employs a bit-parallel squarer [Wu99,PFSR99]. In the ECP's architecture, this squarer is capable of computing a square in one clock cycle. This squarer is a rendition of Equation (1) using XOR gates. For the field polynomials recommended for cryptographic applications [P1398,ANS98,ANS99], the

squarer complexity is at most $(m+t+1)/2$ gates for irreducible trinomials $F(x) = x^m + x^t + 1$ and $4(m-1)$ gates for pentanomials $F(x) = x^m + x^{t1} + x^{t2} + x^{t3} + 1$ [Wu99]. Moreover, for trinomials the critical path delay is at most two gate delays [Wu99].

The AU uses an LSD multiplier of the type introduced in [SP97]. This semi-systolic multiplier computes products according to Equation (2) using Algorithm 3. This multiplier computes a product sum $AB + C \bmod F(\alpha)$ within $\lceil m/D \rceil$ clock cycles. More precisely, the product is computed in k_D clock cycles, where k_D $(1 \leq k_D \leq \lceil m/D \rceil)$ represents the number of digits of B. The performance and consequently complexity of this multiplier is a function of the digit size D [SP97].

Algorithm 3: LSD multiplication

Inputs: $A = \sum_{i=0}^{m-1} a_i \alpha^i$ $B = \sum_{i=0}^{k_D-1} B_i \alpha^{Di}$, where $B_i = \sum_{j=0}^{D-1} b_{Di+j} \alpha^j$ Output: $C = (AB + C) \mod F(\alpha)$

$C = 0$ or the previous value of C
for $i = 0$ to $k_D - 1$ do
$C = B_i(A\alpha^{Di} \mod F(\alpha)) + C$
end for
$C = C \mod F(\alpha)$

As previously described, the ECP takes advantage of the accumulation property of its multiplier to compute additions. The addition $A+B$ requires two clock cycles when it is necessary to compute $A*1$ and then add to it the product $B*1$. It requires only one clock cycle when adding to the result of the previously computed multiplication or addition. In this last case one of the operands is already in the multiplier's accumulator.

A block diagram of the LSD multiplier is included in Figure 2 along with the other components of the AU. Its components are the B shift register, the $A\alpha^{Di} \mod F(\alpha)$, the digit multiplier, the accumulator, and the mod $F(\alpha)$ circuits. The B shift register delivers one digit of the B operand in each clock cycle. The $A\alpha^{Di} \mod F(\alpha)$ circuit computes an element $A\alpha^{Di} \mod F(\alpha)$ in each clock cycle from A for $i = 1$ or from the previously computed $A\alpha^{D(i-1)} \mod F(\alpha)$ for $i = 2, \ldots, k_D - 1$. The digit multiplier computes a product $B_i(A\alpha^{Di} \mod F(\alpha))$ in each clock cycle and the accumulator adds it to the cumulative sum of the previously computed products. The accumulated result is reduced by the mod $F(\alpha)$ circuit. The architecture of the multiplier is regular with only the reduction operations (mod $F(\alpha)$) dependent on the field polynomials.

The complexity of this multiplier, assuming no pipelining of the digit multiplier circuit, is approximately $2Dm + 7m$ gates and $3m$ registers for $m >> D$. The digit multiplier circuit is a main contributor to the complexity and performance of the multiplier. Its gate complexity is proportional to the digit size,

$2Dm$ gates, and, when it is implemented with binary trees, its critical path delay is $\lceil \log_2 2D \rceil$ gate delays.

Note that all the estimates given in this section assume 2-input gates, account for system I/O, and assume optimum field polynomials according to the definition given in [SP97]. These are field polynomials $F(x) = x^m + \sum_{i=0}^{t} f_i x^i$ for which $m - t \geq D$. Over 99% of the field polynomials in [P1398,ANS98,ANS99] satisfy this condition for digit sizes up to $D = 50$ and fields in the range $160 \leq m \leq 1024$.

5 Prototype Implementations

Three ECP prototypes were built to verify the suitability of the ECP architecture for reconfigurable FPGA logic. These prototypes support elliptic curves over the field $GF(2^{167})$, which is an attractive field for secure cryptosystems, with this field being defined by the field polynomial $F(x) = x^{167} + x^6 + 1$. However, we would like to stress that the ECP can be reconfigured with optimized architectures for any field $GF(2^m)$.

Each prototype used a 16-bit MC processor with 256 words of program memory, a 24-bit AUC processor with 512 words of program memory, and 128 registers, each of which is 167 bits wide. They also provided 32-bit I/O interface to the host system. To verify the scalability of the ECP architecture, each of the prototypes used an LSD multiplier with a different digit size. The prototypes used LSD multipliers with digit sizes equal to 4, 8, and 16. To verify the ECP's ability to handle multiple algorithms, the operation of the prototypes was verified with the two elliptic curve algorithms described in the Appendix. The implementation of these two algorithms demonstrates the ability of the ECP to adopt new, highly efficient algorithms. For example, an ECP can be deployed with one algorithm today and then updated with a better algorithm in the future.

The prototypes were implemented using the Xilinix's XCV400E-8-BG432 (Virtex E) FPGA. The prototypes were coded in VHDL. They were synthesized with Synopsis' FPGA Express 3.0 and Xilinx's Design Manager M2.1i. The details of these prototype implementations are discussed in the following subsections.

5.1 ECP Algorithms and Programming

The ECP prototypes were tested with two programs. One of the programs implemented the projective coordinates version of the Montgomery scalar multiplication algorithm and the other the projective coordinates version of the traditional double-and-add algorithm, none of which relies on precomputations. It should be noted that use of algorithms that rely on precomputation is supported by the ECP and their use will typically result in faster implementations than the ones documented here.

The number of clock cycles required to compute kP for each of the programs is summarized in Table 2. Because each step of the Montgomery algorithm requires the computation of a point addition and a point double, this table groups

these two operations in a single row. For the double-and-add algorithm, independent rows for point addition and point double are provided because each step of the algorithm requires a point double but not necessarily a point addition.

Note that the entries in Table 2 contain terms multiplied by $\lceil 167/D \rceil$, where D is the digit size of the multiplier being used. These terms reflect the number of $GF(2^{167})$ multiplications, each of which is executed in $\lceil 167/D \rceil$ clock cycles. The constant terms in the table account for squares, additions and processing overhead. Each square is computed in one clock cycle. Each additions is computed in one clock cycle if one of the operands is already in the multiplier's accumulator or in two clock cycles if that is not the case. The overhead processing time varies with each operation and it is accounted for each operation in the table. The times for coordinate conversions includes the computation of inverses using the inversion algorithm described in [Van99].

For both elliptic curve algorithms, the MC program used 56% of the MC's program memory. The AUC program used 90–98% of the AUC's program memory depending on the algorithm and the digit size. The high AUC memory utilization is due to the in-line coding of the point double and point add functions, which are by far the most frequently used operations. This is evident from the low overhead reported in Table 2 for these functions. To conserve memory, in-line coding was not used for infrequently executed functions such as coordinate conversion. Consequently, these operations exhibit high overhead.

Table 2. Number of clock cycles required to compute kP over $GF(2^{167})$

Operation	Double-and-Add # Clock Cycles	Montgomery # Clock Cycles
Point Double	$5\lceil 167/D \rceil + 25$	$6\lceil 167/D \rceil + 17$
Point Add	$11\lceil 167/D \rceil + 31$	
Coor. Conv., etc.,	$13\lceil 167/D \rceil + 575$	$20\lceil 167/D \rceil + 764$
kP	$(10.5\lceil 167/D \rceil + 47.5) * 166 + 13\lceil 167/D \rceil + 575$	$(6\lceil 167/D \rceil + 24) * 166 + 20\lceil 167/D \rceil + 764$

Table 3 approximates the number of cycles required for the computation of point multiplication for arbitrary $GF(2^m)$ fields. The approximations are based exclusively on the number of multiplications and the number of clock cycles required to compute them with an LSD multiplier with digit size D. This table assumes that inverses are computed using one of the algorithms defined in [IT88,Van99]. The inversion is assumed to require $\lfloor \log_2(m-1) \rfloor + W(m-1) - 1$ multiplications [BSS99], where $W(m-1)$ represents the number of non-zero coefficients in the binary representation of $m-1$.

Table 3. Number of clock cycles required to compute kP over $GF(2^m)$

Operation	Double-and-Add # Clock Cycles	Montgomery # Clock Cycles
Point Double	$5\lceil m/D \rceil$	$6\lceil m/D \rceil$
Point Add	$11\lceil m/D \rceil$	
Coor. Conv.	$(3 + (\lfloor \log_2(m-1) \rfloor + W(m-1) - 1))\lceil m/D \rceil$	$(10 + (\lfloor \log_2(m-1) \rfloor + W(m-1) - 1))\lceil m/D \rceil$
kP	$(10.5(m-1) + 3 + (\lfloor \log_2(m-1) \rfloor + W(m-1) - 1))\lceil m/D \rceil$	$(6(m-1) + 10 + (\lfloor \log_2(m-1) \rfloor + W(m-1) - 1))\lceil m/D \rceil$

5.2 Performance and Comparisons

This section summarizes the performance of the ECP prototype implementations and shows how it compares against leading software and hardware implementations.

Table 4 summarizes the performance of the ECP prototypes for the two elliptic curve algorithms. The results in this table illustrate that the Montgomery method is about 1.7 times faster than the traditional double-and-add algorithm. One can deduce from Table 1 that this is a direct result of the number of multiplications required by each algorithm ($\approx 10.5/6$), as the processing time for additions, squares, and inversions is almost negligible.

Table 4 also shows that the speedup increases as the digit size increases. The increase is not proportional to the digit size. What happens is that as the digit size increases, the multiplication processing time decreases proportionally. Consequently, the additions, the squarings, and the overhead processing costs increase relative to that of multiplications. Another contributing factor is the modest reduction in clock rate as the digit size increases and thus the size of the ECP. For the prototypes, an appreciable reduction in clock rate occurs as the digit size increased from 4 to 8. The clock rate remained fairly constant as the digit size increased from 8 to 16.

Table 4. Point multiplication performance of ECP prototypes

Digit Size	Clock (MHz)	Montgomery (msec)	double-and-add (msec)	Speedup rel. to $D = 4$
4	85.7	0.55	0.96	1
8	74.5	0.35	0.61	1.8
16	76.7	0.21	0.36	3.0

Table 5 lists the performance of leading published software (SW) and hardware (HW) implementations along with that of the fastest ECP prototype im-

plementation. The data in this table correspond to k values whose binary representation contains roughly the same number of 1's and 0's. Table 5 shows that the performance of software implementations on platforms with wide words and high clock rates rival that of traditional hardware implementations. It also shows that the performance of the fastest ECP implementation is at least 19 times faster than that of traditional hardware implementations and 37 times faster than software implementations.

Table 5. Performance of leading software and hardware implementations

Implementation	SW/ HW	Fields	Platform	Point Mult. (msecs)	Speedup rel. to ECP $D = 16$
Montgomery [LD99]	SW	$GF(2^{163})$	UltraSparc 64-bit,300MHz	**13.5**	64
Almost Inv. [SOOS95]	SW	$GF(2^{155})$	DEC Alpha 64-bit,175MHz	**7.8**	37
ASIC Coprocessor [AMV93]	HW	$GF(2^{155})$	VLSI 40 MHz	**3.9** est.	19
FPGA Coprocessor [SES98]	HW	$GF(2^{155})$	Xilinx FPGA XC4020XL,15 MHz	**18.4** est.	88
Composite fields [Ros98]	HW	$GF(((2^4)^2)^{21})$ $GF((2^8)^{21})$	Xilinx FGPA XC4062,16MHz	**4.5** est.	21
ECP $D = 16$	HW	$GF(2^{167})$	Xilinx FPGA XCV400E,76.7MHz	**0.21**	1

5.3 Logic Complexity

The logic complexity of the ECP prototypes is summarized in Table 6 in terms of the main components of modern FPGAs. These components are lookup tables (LUT) which are used as programmable gates, flip-flops (FF), and Block RAM which are configurable 4k-bit RAMs [Xil99]. The normalized complexity of the ECP prototypes is approximately $228+6.6m+(\lceil 2D/3 \rceil -1)m$ LUTs, $224+9.2m$ FF, and $4+\lceil m/32 \rceil$ 4k-bit Block RAMs for $m >> D$, 4-input LUTs, 32-bit Block RAMs, and D a multiple of 4. Note that the complexity is a function of the digit size D, which as mentioned previously is the main parameter that defines the performance and complexity of the ECP, and the size of the finite field (m). Interestingly, of all the logic elements only LUT logic complexity varies largely as a function of D. The multiplier's digit multiplier circuit is responsible for this variability as its size varies proportionally with the digit size.

The prototype implementations used between 15% and 28% of the LUTs (depending on the digit size), 16% of the FFs, and 25% of the Block RAMs available in the XCV400E-8-BG432 FPGA. Together, the AUC and the MC

Table 6. Logic complexity of ECP prototypes

Digit Size	#LUT	#FF	# Block RAM
4	1627	1745	10
8	2136	1753	10
16	3002	1769	10

processors, ignoring the complexity of the register that holds the k operand, used less than 13% of the logic resources and 40% of the memory elements. In turn, the AU used 76–87% of the LUTs, 59% of the flip-flops, and 60% of the memory elements. The remaining resources were used by system I/O logic. This breakdown shows that the ECP prototype implementations devoted most of its resources to arithmetic processing.

6 Conclusions

This work introduced a new elliptic curve processor architecture. This is a scalable and programmable processor architecture that exploits reconfigurability to deliver optimized solutions for different elliptic curves and finite fields. The ECP architecture is characterized by two loosely coupled processors responsible for the algorithmic functions of point multiplication and by a streamlined, pipelined finite field arithmetic unit that can be optimized for each finite field.

This work demonstrated that the ECP can attain high processing speeds in FPGA logic with three prototype implementations. The fastest prototype implementation was capable of computing a point multiplication in the field $GF(2^{167})$ at least 19 times faster than documented hardware implementations and 37 times faster than documented software implementations. Moreover, because the ECP is programmable as well as configurable, these prototype implementations can be programmed to use future, more efficient elliptic curve algorithms, and their size and performance can be tailored, through reconfiguration, to meet future needs.

References

AMV93. G.B. Agnew, R.C. Mullin, and S.A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

ANS98. ANSI X9.62-1999. Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), January 1998. Approved January 7, 1999.

ANS99. ANSI X9.63-1999. Public Key Cryptography For The Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography, January 1999. Working Draft.

BG89. T. Beth and D. Gollmann. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications*, 7(4):458–466, 1989.

BGMW93. E.F. Brickell, D.M. Gordon, K.S. McCurley, and D.B. Wilson. Fast exponentiation with precomputation. In *Lecture Notes in Computer Science 658: Advances in Cryptology — EUROCRYPT '92*, pages 200 – 207. Springer-Verlag, Berlin, 1993.

BSS99. I. Blake, G. Seroussi, and N.P. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, Cambridge, UK, first edition, 1999.

GHS00. P. Gaundry, F. Hess, and N.P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. available at `http://www.hpl.hp.com/techreports/2000/HPL-2000-10.html`, January 2000.

GSS99. L. Gao, S. Shrivastava, and G. Sobelman. Elliptic curve scalar multiplier design using FPGAs. In C. Koc and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, volume LNCS 1717. Springer-Verlag, August 1999.

IT88. T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78:171–177, 1988.

LD99. J. Lopez and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In C. Koc and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, volume LNCS 1717. Springer-Verlag, August 1999.

LN94. R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, Cambridge, UK, revised edition, 1994.

NIS99. NIST. Recommended elliptic curves for federal government use. available at `http://csrc.nist.gov/encryption`, May 1999.

P1398. P1363. *Standard Specifications for Public-key Cryptography (Draft Version 8)*. IEEE, October 1998.

PFSR99. C. Paar, P. Fleischmann, and P. Soria-Rodriguez. Fast arithmetic for public-key algorithms in Galois fields with composite exponents. *IEEE Transactions on Computers*, 48(10):1025–1034, October 1999.

Ros98. M. Rosner. Elliptic curve cryptosystems on reconfigurable hardware. Master's thesis, ECE Dept., Worcester Polytechnic Institute, Worcester, USA, May 1998.

SES98. S. Sutikno, R. Effendi, and A. Surya. Design and implementation of arithmetic processor $F_{2^{155}}$ for elliptic curve cryptosystems. In *The 1998 IEEE Asia-Pacific Conference on Circuits and Systems*, pages 647–650, November 1998.

SOOS95. R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptogaphy, Crypto 95*, volume LNCS 963. Springer-Verlag, 1995.

SP97. L. Song and K. K. Parhi. Low-energy digit-serial/parallel finite field multipliers. *Journal of VLSI Signal Processing Systems*, 2(22):1–17, 1997.

Van99. S.A. Vanstone. Efficient implementation of elliptic curve cryptography, June 1999. Certicom Corporation Seminar.

Wu99. H. Wu. Low complexity bit-parallel finite field arithmetic using polynomial basis. In C. Koc and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, volume LNCS 1717. Springer-Verlag, August 1999.

Xil99. Xilinx. *The Programmable Logic Data Book*. Xilinx, Inc., 1999.

A Relevant Algorithms

Algorithm 1: Double-and-add scalar multiplication using projective coordinates

```
double_and_add(x, y, k)
  (X, Y, Z) = conv_projective(x,y)
  (X_0, Y_0, Z_0) = (X, Y, Z) /* P_0 = P */
  for i = l − 2 downto 0 do
    (X, Y, Z) = double(X, Y, Z) /* P = 2P */
    if k_i = 1 then /* P = P + P_0 */
      (X, Y, Z) = add(X_0, Y_0, Z_0, X, Y, Z)
    end if
  end for
  (x, y) = conv_affine(X, Y, Z)
return (x, y)
```

```
double(X, Y, Z)
  /* if P = O then return O */
  if (X, Y, Z) = O then return(O)
  else /* P ≠ O return 2P */
    Z_2 = X * Z^2
    X_2 = (X + b^{1/4} Z^2)^4
    U = Z_2 + X^2 + YZ
    Y_2 = X^4 Z_2 + U X_2
  endif
return(X_2, Y_2, Z_2)
```

```
conv_projective(x, y)
return (X = x, Y = y, Z = 1)
```

```
conv_affine(X, Y, Z)
return(x = X/Z^2, y = Y/Z^3)
```

```
add(X_0, Y_0, Z_0, X_1, Y_1, Z_1)
  /* if P_1 = O then return P_0 */
  if (X_1, Y_1, Z_1) = O then
    return(X_0, Y_0, Z_0)
  /* else if P_0 = −P_1 then return O */
  else if (X_0, Y_0, Z_0) = −(X_1, Y_1, Z_1)
    then return(O)
  /* else if P_0 = P_1 then return 2P_0 */
  else if (X_0, Y_0, Z_0) = (X_1, Y_1, Z_1) then
    (X_2, Y_2, Z_2) = double(X_0, Y_0, Z_0)
  else /* return P_2 = P_0 + P_1 */
    U_0 = X_0 Z_1^2
    S_0 = Y_0 Z_1^3
    U_1 = X_1 Z_0^2
    W = U_0 + U_1
    S_1 = Y_1 Z_0^3
    R = S_0 + S_1
    L = Z_0 W
    V = R X_1 + L Y_1
    Z_2 = L Z_1
    T = R + Z_2
    X_2 = a Z_2^2 + TR + W^3
    Y_2 = T X_2 + V L^2
  endif
return(X_2, Y_2, Z_2)
```

Algorithm 2: Montgomery scalar multiplication using projective coordinates

```
montgomery_scalar_multiplication(x, y, k)
  /* P_1 = P,  P_2 = 2P */
  (X_1, Z_1, X_2, Y_2) = conv_projective(x, y)
  for i = l − 2 downto 0 do
    if k_i = 1 then
      /*P_1.X = P_1.X + P_2.X, P_2.X = 2P_2.X*/
      (X_1, Z_1) = madd(X_1, Z_1, X_2, Z_2, x)
      (X_2, Z_2) = mdouble(X_2, Z_2)
    else
      /*P_2.X = P_1.X + P_2.X, P_1.X = 2P_1.X*/
      (X_2, Z_2) = madd(X_2, Z_2, X_1, Z_1, x)
      (X_1, Z_1) = mdouble(X_1, Z_1)
    end if
  end for
return(compute_xy(X_1, Z_1, X_2, Z_2, x, y))
```

```
conv_projective(x, y)
 X_1 = x;   Z_1 = 1
 X_2 = x^4 + b;   Z_2 = x^2
return(X_1, Z_1, X_2, Z_2)
```

```
madd(X_1, Z_1, X_2, Z_2, x)
 X_1 = X_1 Z_2 X_2 Z_1 + x(X_1 Z_2 + X_2 Z_1)^2
 Z_1 = (X_1 Z_2 + X_2 Z_1)^2
return(X_1, Z_1)
```

```
mdouble(X, Z)
return(X = X^4 + bZ^4, Z = X^2 Z^2)
```

```
compute_xy(X_1, Z_1, X_2, Z_2, x, y)
 x_k = X_1/Z_1
 y_k = ((X_1/Z_1 + x)(X_2/Z_2 + x)+
       x^2 + y) * (X_1/Z_1 + x)/x + y)
return(x_k, y_k)
```