

# A PCI-Card for Accelerating Elliptic Curve Cryptography

Johannes Wolkerstorfer and Wolfgang Bauer

Institute for Applied Information Processing and Communications (IAIK)  
Graz University of Technology  
Inffeldgasse 16a, 8010 Graz, AUSTRIA  
[Johannes.Wolkerstorfer@iaik.at](mailto:Johannes.Wolkerstorfer@iaik.at)

## Abstract

*In this article we present a PCI-card for accelerating Elliptic Curve Cryptography. The card has application in e-Commerce and e-Government where Internet servers establish secure and authenticated communication with clients over public networks. All elliptic-curve operations are processed on the card to liberate the software from this computational intensive task. The hardware is optimized for a particular class of elliptic curves. A medium-sized FPGA serves as target technology, which allows to update the card in the field and to customize the hardware quickly to other curves or even improved algorithms. The current implementation is optimized for an 191-bit elliptic curve over  $GF(2^{191})$  and is able to calculate 1000 scalar multiplications per second on a Xilinx Spartan-II device. A highly parallel radix-256 digit-serial multiplier accounts for the high throughput. The maximum clock frequency is 70 MHz.*

## 1 Introduction

Information security is one of the main aspects of e-Commerce and e-Government. In this fast-growing area new services only find acceptance when they provide a sufficient level of security in terms of authentication, confidentiality, data integrity, and non-repudiation. For instance, the *Weißbuch Bürgerkarte* [1] focuses on security aspects of the Austrian e-Government activities and emphasizes the role of secure digital-signatures as a key technology. Elliptic Curve Cryptography (ECC) is a technology that fulfills all the demands and requires only moderate resources [2].

Digital signatures as well as key establishment algorithms, which are required to establish encrypted channels in insecure networks, belong to the asymmetric cryptography [3]. ECC can implement asymmetric cryptography very efficiently because it is based on the Elliptic-Curve-Discrete-Logarithm-Problem (ECDLP). No algorithm with sub-exponential running time is known to solve ECDLP. Alternatives to ECC are methods based on the discrete logarithm problem or the factorization problem. Both re-

quire bitlengths of 2048-bit to obtain adequate security. ECC is content with much smaller bitlengths to achieve the same level of security and is thus favored in resource-constricted devices like smartcards. ECC covers all relevant asymmetric cryptographic primitives like digital signatures and key agreement algorithms and is standardized by many organizations. A relevant standard for this article is the ANSI X9.62 standard. It defines the Elliptic Curve Digital Signature Algorithm (ECDSA) [5]. ANSI X9.63 defines cryptographic primitives for key agreement [6].

The main portion of these primitives is calculated by the PCI-card to accelerate ECC. Such a dedicated hardware solution offers the advantage to process data at the full wordsize of  $m$  bits. In this article the wordsize is fixed to 191 bits, although the developed hardware-description model has a scalable wordsize. Operating at the full wordsize will turn out to be the main contributor to outperform software solutions, which have to operate on 32-bit words. Calculations on elliptic curves involve doubling and addition of curve points. These operations are calculated by operations in the finite field  $GF(2^m)$ . The overall performance of an elliptic-curve processor is mainly determined by the speed to multiply field elements. Hence, the multiplier will be the most important building block of such a processor and parallelization of the multiplication will be a key issue in the design of the architecture. If the costly inversion of field elements is avoided, the enlarged effort to control EC-operations needs special attention in a hardware implementation to prevent the critical path from being in the control unit.

The remainder of this article is structured as follows. Section 2 highlights the application of the ECC-accelerator-card to point out the motivation for this work and section 3 gives an overview over related work that takes both software implementations and hardware implementations into account. Section 4 briefly presents the mathematical background, which is useful to explain the design considerations of the hardware architecture in section 5. The PCI interface and the software level are discussed in section 6. Section 7 presents results, and conclusions are drawn in section 8.

## 2 Application

The PCI-card for accelerating ECC has an application in e-Commerce and e-Government. It accelerates the EC-operations on the server side as depicted in Figure 1.

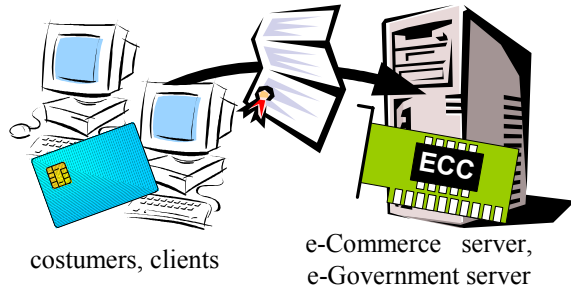


Figure 1: Application

The performance of EC-operations on the client side of such an application is not critical. The authentication, the negotiation of encryption keys, and the signing of documents may take up to a second and can be done efficiently with smartcards or similar devices. The server side on the other hand has to process EC-operations for a multitude of clients and to perform other operations like database queries. A dedicated ECC-accelerator-card will relieve the computational effort of the server.

## 3 Previous work

Up to our knowledge there are no ECC-accelerator-cards commercially available by now. Scientific results for accelerating EC-operations with hardware will be given below. A rough comparison of the throughput can be made by looking at the performance of hardware-accelerator-cards for asymmetric cryptography based on the RSA algorithm. For instance, the *AEP SSL Accelerator*<sup>TM</sup> card from AEP Systems has a throughput of up to 2000 encryptions per second for 1024-bit bitlength [7]. This will reduce to approximately 500 encryptions for 2048 bits, which is comparable to 191-bit ECC.

Darrel Hankerson et al. presented the fastest known software implementation of EC-operations in [8]. Their assembler optimized implementation on a 400 MHz Pentium-II processor reached 594 EC-operations per second for a 163-bit curve and 252 operations for 233-bit. This compares roughly to 414 EC-operations for a 191-bit curve and it should be considered that the CPU load for these values is 100%. Nothing else can be computed without deteriorating the throughput.

One of the first ECC hardware implementations was reported 1993 by G. B. Agnew et al. in [9]. Their implemen-

tation is a mere coprocessor for operations in the finite field  $GF(2^{155})$  and EC-operations are controlled by a multi-purpose processor. This ASIC implementation is basically a bit-serial multiplier with a clock frequency of 40 MHz.

James Goodman and Anantha P. Chandrakasan published their so-called Domain-Specific Reconfigurable Cryptographic Processor in [10]. Their versatile design can operate with bitlengths from 4-bit up to 1024-bit and can calculate modular integer-arithmetic besides operations in the finite field  $GF(2^m)$ . This ASIC circuit can be clocked with 50 MHz and calculates approximately 125 191-bit EC-operations per second.

Gerardo Orlando and Christof Paar reported the fastest known EC-processor in [12]. Their FPGA implementation is also optimized for a particular class of elliptic curves (fixed bitlength), has a digit-serial multiplier, and an extra square unit to exploit a shortcut offered by  $GF(2^m)$ -arithmetic. A large internal memory allows the use of algorithms, which rely on pre-computations. This implementation should have a throughput of about 2000 EC-operations per second when it is clocked with 70 MHz and when a radix-256 multiplier is used.

M. Ernst et al. presented in [11] an FPGA-based PCI-card for ECC-acceleration. Contrary to the widely used polynomial-basis representation of  $GF(2^m)$ -elements, they use an optimal-normal basis representation that cannot be directly compared with a polynomial representation. A 191-bit version of their PCI-card with a radix-32 multiplier can be clocked with 36 MHz and has a throughput of 431 EC-operations. The card can be accessed from a personal computer running under Windows NT via a C++-interface.

## 4 Mathematical background

ECC is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP). The ECDLP states that it is a hard problem to find the scalar  $k$  in  $Q=k \cdot P$  when the curve points  $Q$  and  $P$  are given. No sub-exponential-time algorithm is known to solve the ECDLP for non-super-singular elliptic curves [3]. The scalar-multiplication  $Q=k \cdot P$  is therefore the basic operation in all elliptic-curve cryptographic primitives like the Elliptic Curve Digital Signature Algorithm (ECDSA) [5]. The scalar-multiplication can be calculated by repeated point additions

$$\begin{aligned} Q = k \cdot P &= P + P + \dots + P = \\ &= k_0 P + k_1 2P + k_2 4P + \dots + k_{m-1} 2^{m-1} P \end{aligned}$$

or more efficiently by the double-and-add algorithm, which iteratively doubles the point  $P$  ( $P, 2P, 4P, \dots$ ) and accumulates those multiples of  $P$  where the according bit  $k_i$  of the binary representation of the scalar  $k$  is 1. This procedure requires functions to double points and to add them. De-

tailed information about the mathematical background of ECC can be found in [4].

A point  $P$  on an elliptic curve over a finite field of characteristic 2 is a tuple  $P=(x, y)$  where  $x$  and  $y$  are elements of the finite field  $GF(2^m)$ —in our case  $m$  is fixed to 191. Every point on an elliptic curve fulfills the affine equation of the elliptic curve

$$EC: y^2 + xy = x^3 + ax^2 + b, \quad a, b \in GF(2^m)$$

where the parameters  $a$  and  $b$  define a particular curve over  $GF(2^m)$ . In addition to all tuples fulfilling the curve equation, the so-called point-at-infinity  $\mathcal{O}$  belongs to the curve. The set of all points together with a general point addition form a commutative group. The general point addition  $P_1+P_2$  distinguishes between the cases that  $P_1 \neq P_2$  (addition),  $P_1=P_2$  (double), and  $P_1=-P_2$  (point-at-infinity  $\mathcal{O}$ ). The point-at-infinity is the neutral element for the general point addition:  $P_1+\mathcal{O}=P_1$ . Addition and doubling of points is calculated by field operations on the coordinates of the tuple. Point doubling is given by

$$\begin{aligned} P_1 + P_2 &= (x_1, y_1) + (x_2, y_2) = (x_3, y_3), & P_1 \neq P_2 \\ \lambda &= (y_1 + y_2)(x_1 + x_2)^{-1} \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1. \end{aligned}$$

Point addition is given by

$$\begin{aligned} 2P &= 2(x, y) = (x_3, y_3) \\ \lambda &= yx^{-1} + x \\ x_3 &= \lambda^2 + \lambda + a \\ y_3 &= (x + x_3)\lambda + x_3 + y. \end{aligned}$$

From the formulas for point addition and point doubling it can be seen that the finite-field operations addition, multiplication, squaring and inversion are required to calculate EC-operations. Inversion is the most complex operation. It can (nearly) be avoided if projective coordinates are used to represent points. An in-depth description of this topic is omitted here, but interested readers should refer to [4]. The trick of projective coordinates is to avoid inversion by representing a point by a triple  $(x,y,z)$  of field elements. This comes at the cost of an increased number of multiplications for point addition and point doubling but will pay off, as long as the cost of an inversion is more than ten times higher than those of a multiplication. Projective coordinates require only one inversion to transform a projective result  $(x,y,z)$  back into an affine representation  $(x',y')$ .

## 4.1 Operations in the finite field $GF(2^m)$

Elements of the finite field  $GF(2^m)$  can be represented as polynomials of degree smaller than  $m$  with binary coefficients:

$$\begin{aligned} a(x) &= a_0 + a_1x + a_2x^2 + \dots + a_{m-1}x^{m-1}, \quad a(x) \in GF(2^m) \\ &= \sum_{i=0}^{m-1} a_i x^i, & a_i \in \{0,1\}. \end{aligned}$$

Elements of  $GF(2^m)$  are fully determined by their binary coefficients which allows to represent them as strings of  $m$  bits. In example, an element of  $GF(2^{191})$  can be stored in a 191-bit register. Operations on field elements of  $GF(2^m)$  are derived from calculations with polynomials. The addition  $a(x)+b(x)$  of two elements  $a(x), b(x) \in GF(2^m)$  is done coefficient-wise:

$$a(x) + b(x) = \sum_{i=0}^{m-1} a_i x^i + \sum_{i=0}^{m-1} b_i x^i = \sum_{i=0}^{m-1} (a_i \oplus b_i) x^i.$$

The symbol  $\oplus$  denotes the addition in the field  $GF(2)$ , which is an integer addition modulo 2. This corresponds to the Boolean XOR-function. A hardware implementation of the coefficient-wise addition scheme can be realized with  $m$  XOR-gates, which add corresponding bits of the bitstring representation of  $a(x)$  and  $b(x)$ . Contrary to integer addition, addition in  $GF(2^m)$  has a constant complexity of  $O(1)$  for arbitrary  $m$  because no carry propagation occurs. A noticeable feature of addition in  $GF(2^m)$  is the fact that an element is its own additive identity:  $a(x)+a(x)=0$ . This observation accounts for the identity of addition and subtraction:  $a(x)+b(x)=a(x)-b(x)$ .

Multiplication in  $GF(2^m)$  is more complex than addition and has some similarities to modular integer multiplication. A pure polynomial multiplication  $a(x) \cdot b(x)$  will result a polynomial of degree less or equal than  $2(m-1)$ :

$$a(x) \cdot b(x) = a(x) \cdot \left( \sum_{i=0}^{m-1} b_i x^i \right) = \sum_{i=0}^{m-1} (a(x)b_i) \cdot x^i.$$

In case, the result has a degree higher than  $m-1$  it is not an element of  $GF(2^m)$  anymore. It is necessary to apply a modular reduction step, which calculates the remainder of a polynomial division by an irreducible polynomial. The irreducible polynomial has degree  $m$  and defines the polynomial-basis representation of the finite field  $GF(2^m)$ . The irreducible polynomial in our  $GF(2^{191})$ -example is  $f(x) = x^{191} + x^9 + 1$ ; as defined in Example 1 of Annex J4.3 in [5]. It is possible to avoid intermediate results that have nearly double the bitlength by interleaving the modular reduction in the multiplication. A MSB-to-LSB scheme of this procedure is shown in Algorithm 1. In every iteration a partial product  $a(x)b_i$  is added to the intermediate result  $c(x)$ . In case  $c(x)$  has degree  $m$ , the irreducible polynomial  $f(x)$  is subtracted to ensure  $c(x)$  is an element of  $GF(2^m)$ .

**Input:**  $a(x), b(x) \in \text{GF}(2^m)$ ,  
irreducible polynomial  $f(x)$  of degree  $m$

**Output:**  $c(x) = a(x) \cdot b(x) \bmod f(x)$

```

1:    $c(x) \leftarrow 0$ 
2:   for  $i = m-1$  to  $0$  do
3:      $c(x) \leftarrow c(x) \cdot x + a(x)b_i$ 
4:      $c(x) \leftarrow c(x) + f(x)c_m$ 
5:   end for
6:   return  $c(x)$ 

```

**Algorithm 1: Multiplication in  $\text{GF}(2^m)$**

A multiplication of a polynomial by a coefficient ( $a(x)b_i$ ) corresponds to a bitwise Boolean AND-function:  $a(x)1 = a(x)$ ;  $a(x)0 = 0$ . Multiplication of polynomials by a power of  $x$  ( $a(x) \cdot x^i$ ) corresponds to a shift-left operation where the bitstring representation of the element is shifted  $i$  positions.

Squaring in  $\text{GF}(2^m)$  can either be calculated by a multiplication  $a(x) \cdot a(x) \bmod f(x)$ . Another possibility is to exploit the fact that squaring in  $\text{GF}(2^m)$  is a linear operation:  $(a(x) + b(x))^2 \equiv a^2(x) + b^2(x) \bmod f(x)$ . This allows to square  $a(x)$  by calculating

$$a^2(x) = \left( \sum_{i=0}^{m-1} a_i x^i \right)^2 = \sum_{i=0}^{m-1} a_i x^{2i}.$$

This corresponds to an insertion of 0 after each binary coefficient in the bitstring representation of  $a(x)$ . A subsequent modular reduction of the expanded bitstring will yield the desired result.

The inverse of an  $\text{GF}(2^m)$ -element satisfies the equation  $a(x) \cdot a^{-1}(x) \equiv 1 \bmod f(x)$ . It can either be calculated with the Extended Euclidean algorithm for polynomials (see [3]) or by exponentiation:

$$a^{-1}(x) = a^{2^m - 2}(x) \bmod f(x).$$

The calculation of the Extended Euclidean algorithm includes comparing the degree of polynomials. In a hardware implementation this requires additional logic resources and complicates the architecture. Calculating the inverse by exponentiation relies on multiplication and squaring as shown in Algorithm 2. The time-complexity of Algorithm 2 is determined by the speed of multiplication. The performance will usually be slower than those of the Extended Euclidean algorithm but will not deteriorate the overall performance of an EC-operation because only a single inversion is needed when projective coordinates are used.

**Input:**  $a(x) \in \text{GF}(2^m)$ ,  
irreducible polynomial  $f(x)$  of degree  $m$

**Output:**  $c(x) = a^{-1}(x) \bmod f(x)$

```

1:    $c(x) \leftarrow 1$ 
2:   for  $i = 0$  to  $m-2$  do
3:      $a(x) \leftarrow a(x)^2 \bmod f(x)$ 
4:      $c(x) \leftarrow c(x) \cdot a(x) \bmod f(x)$ 
5:   end for
6:   return  $c(x)$ 

```

**Algorithm 2: Inversion in  $\text{GF}(2^m)$**

## 5 Hardware architecture

The objective of the PCI-card for accelerating ECC is to calculate the scalar-multiplication of elliptic-curve points:  $k \cdot P$ . From sight of the server where the card is plugged in, the card does this operation with affine coordinates. Internally, the calculations are processed with projective coordinates to avoid time-consuming inversions. The output is transformed on the card into affine coordinates. So the server's software is completely liberated from EC-operations and has only to manage IO and to calculate a small number of modular integer operations to implement ECC-primitives like the digital-signature algorithm.

The PCI-card in use is an off-the-shelf product. The Csys XC2S\_EVAL-card [13] consists mainly of two chips. The first one is an Infineon PITA-2 PCI-bridge that handles the communication with the PCI bus [14]. It is interconnected via a micro-controller interface with the second chip, a Xilinx Spartan-II XC2S200 FPGA [15]. The whole functionality of the EC-processor resides within this FPGA. Other resources of the board like SRAM are not used.

The overall architecture of the EC-processor is shown in Figure 2. The most prominent part of the architecture is the arithmetic unit, which comprehends a highly parallel  $\text{GF}(2^m)$ -multiplier and a square unit. The register-file is a storage unit for 191-bit intermediate results and curve parameters. The arithmetic unit and the register-file receive their control signals from the ECC-control-unit. This unit controls the scalar-multiplication sequence that is compound of point additions and doublings. A small interface unit passes input and output data from the PCI-bridge to the arithmetic unit. Commands and status information is transferred between the PCI-bridge and the ECC-control-unit.

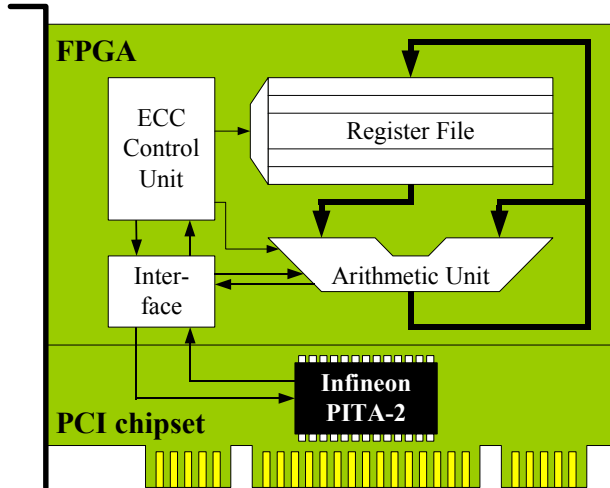


Figure 2: Hardware architecture

### 5.1 Datapath

The datapath of the EC-processor consists of the arithmetic unit and the register-file. The communication between these two units is 191-bit wide. 191-bit buses can transfer intermediate results and curve parameters in a single clock cycle. These values are stored in the register-file, which has a capacity of 16 191-bit words. The capacity is sufficient for all EC-operations and has a convenient size for implementation with Xilinx FPGAs. The basic building blocks of Xilinx FPGAs are Configurable Logic Blocks (CLB) that can be configured as 16x1-bit memory—191 such blocks make the register-file. The register-file has a read port and a write port, which share a common address decoder. The read port is always active; the write port is active when the according control signal is set. A true dual-ported register-file would increase the overall throughput unperceivable at the cost of a more complicated control unit. Therefore, this idea was abandoned.

All operations in the finite field  $GF(2^m)$  are calculated by the arithmetic unit shown in Figure 3. The arithmetic unit performs a number of operations: load  $a(x)$ , hold  $c(x)$ , add  $a(x)+c(x) \bmod f(x)$ , multiply  $a(x) \cdot m(x) \bmod f(x)$ , and square  $c^2(x) \bmod f(x)$ . In addition to these arithmetic operations, the arithmetic unit also does the IO for the EC-processor. The arithmetic unit has two 191-bit registers. Register  $C$  stores the intermediate result of a calculation. This value serves as 191-bit output of the arithmetic unit and is fed back externally to the input  $b(x)$ . The second 191-bit register  $M$  stores the second argument  $m(x)$  required for multiplication. This register is not directly accessible from outside the unit.

Further components of the arithmetic unit are a digit-serial multiplier, an adder with an integrated modular re-

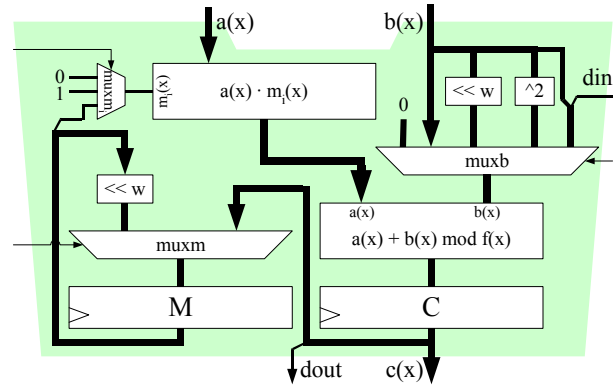


Figure 3: Arithmetic unit

duction unit, a square unit, two shifters, and three multiplexers to select the desired operation.

The arithmetic unit is optimized for high data-throughput. All operations except multiplication are executed in a single clock cycle. The number of cycles needed for multiplication depends on the degree of parallelism of the multiplier unit and is discussed in detail after an analysis of the single-cycle operations.

**Hold:** The simplest single-cycle operation is *hold*: This operation sustains the value  $c(x)$  in register  $C$ . The value  $c(x)$  is externally fed back to the input  $b(x)$  and passes through the multiplexer  $muxb$  to the adder. The adder does not alter the value if the multiplier output is 0. This can be achieved by selecting the multiplier input  $m_i(x)=0$ .

**Load:** The *load- $a(x)$* -operation is similar to the hold operation. The multiplexer  $muxb$  selects 0 as input value for the adder to cut off the  $b(x)$ -input. The multiplier passes its input value  $a(x)$  to the adder by selecting  $m_i(x)=1$ .

**IO.** The IO-operation is basically a shift-left operation of register  $C$  where new data is loaded into the least-significant bits and the most-significant bits are dropped. The wordsize of input and output data is scaleable and is  $d$ -bit wide. The register  $C$  is shifted  $d$  positions to the left by selecting the rightmost input of multiplexer  $muxb$  (see Figure 3). The input  $din$  provides the lowest  $d$  bits. The highest  $d$  bits of register  $C$  are visible from outside at the bus  $dout$ .

**Addition.** Addition sums up input  $a(x)$  and the content of register  $C$ . This is an addition in  $GF(2^m)$  which corresponds to a bitwise XOR function. The input  $a(x)$  is passed to the adder in the same way as during *load*. The feedback mechanism for the content of register  $C$  is the same as for the *hold*-operation.

**Multiplication.** Multiplication is a multi-cycle operation. It is performed in a digit-serial manner where the mul-

tiplicand  $a(x)$  is assigned at full precision and the multiplier  $m(x)$  is processed in digits. The wordsize of the digits is scaleable. So the EC-processor can be optimized for high throughput (i.e. 16-bit or 32-bit digits) or for low gate count (i.e. 4-bit digits). Scaling of the multiplier is accomplished by a resynthesis of the HDL source-code. Actually, 8-bit digits are used which is a fair tradeoff between throughput and gate count ( $w=8$ ). The resulting radix-256 multiplier needs 25 clock cycles to multiply 191-bit values. In the first clock cycle the content of register  $C$  is loaded into register  $M$  to become the multiplier  $m(x)$ . During the next 24 clock cycles  $m(x)$  is assigned digit-after-digit to the  $m_i(x)$ -input of the multiplier (see Figure 3). The most-significant digit is processed in the first cycle, the least-significant in the last cycle. The register  $M$  generates these digits by shifting its content in every cycle  $w$  positions to the left and assigning the highest  $w$  bits to the multiplier. During the whole multiplication the register-file outputs the multiplicand  $a(x)$ . The multiplier generates the partial products  $a(x) \cdot m_i(x)$ , which are accumulated in register  $C$ . Intermediate results in register  $C$  have to be aligned to new partial products. This is done by shifting the feedback  $b(x)$   $w$  positions to the left before addition with the new partial product. In the first clock cycle the feedback is cut off to set register  $C$  to its initial value  $c(x)=0$ .

The generation of a partial product  $p(x)=a(x) \cdot m_i(x)$  is shown in Figure 4 where a  $GF(2^{191})$ -multiplier is depicted. For convenience, a radix-4 version of the multiplier is shown which limits  $m_i(x)$  to two input bits. The partial product  $p(x)$  is the sum of two basic partial-products that are generated by masking the multiplicand  $a(x)$  with AND-gates. The addition of these two values is a bitwise XOR-function of the corresponding bits. When the multiplier is scaled to achieve higher throughput, the number of basic partial-products is increased. The VHDL model supports scaling with powers of two, which effect the most efficient circuits because the XOR-gates for summing up the partial products can be arranged in a tree structure. This guarantees that the critical path grows very slowly:  $O(\log_2 \#basic\_partial\_products)$ .

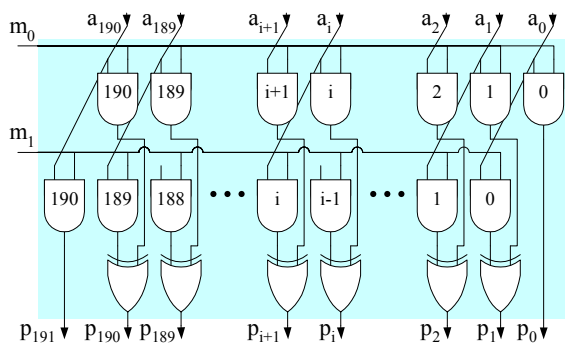


Figure 4:  $GF(2^{191})$ -multiplier

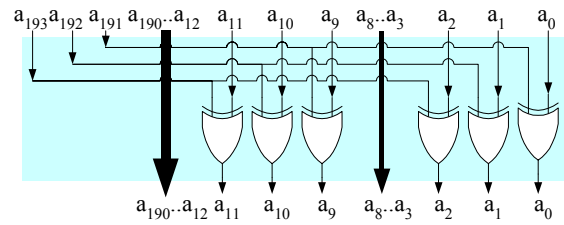


Figure 5: Modular reduction in  $GF(2^{191})$

**Modular reduction.** Multiplication and squaring produces intermediate results that might be longer than  $m$  bits. Therefore, the adder unit has an integrated modular reduction unit, which shortens intermediate results to  $m$  bits. Modular reduction for arbitrary irreducible polynomials will result in a bulky circuit. Therefore, the modular reduction is optimized for a distinct trinomial or pentanomial. Figure 5 shows the reduction unit for the irreducible polynomial  $f(x) = x^{191} + x^9 + 1$  which is used in our  $GF(2^{191})$ -example.  $f(x)$  is a trinomial because it has three terms. The number of terms influences the complexity of the reduction unit. The width  $w$  of the multiplier is another parameter for the complexity: The circuit in Figure 5 shows the reduction unit for  $w=4$  where three bits have to be reduced. The HDL description for the reduction unit can be parameterized with any trinomial/pentanomial and any width  $w$  to customize the hardware quickly for other fields and multiplier sizes.

**Squaring.** The arithmetic unit has an extra square unit because this operation takes only one clock cycle. The square unit resides in the feedback path of the arithmetic unit. It actually calculates  $a^2(x) \bmod x^{w-1} \cdot f(x)$  which is advantageous because the modular reduction unit will fully reduce this intermediate result to  $a^2(x) \bmod f(x)$ . The calculation is limited to trinomials or pentanomials ( $f(x)=x^m+x^j+x^k+1$ ). The architecture of the square unit is shown in Figure 6. It exploits the fact that  $a(x)$  can be split into two polynomials which can be squared individually  $a(x) = a_h(x) \cdot x^{(m-1)/2} + a_l(x)$ . Only the higher half  $a_h(x)$  will be affected by modular reduction. In case of our  $GF(2^{191})$ -example the square operation reduces almost to an addition of the expanded halves:  $a_l^2(x) + a_h^2(x) \cdot (x^9 + 1)$ .

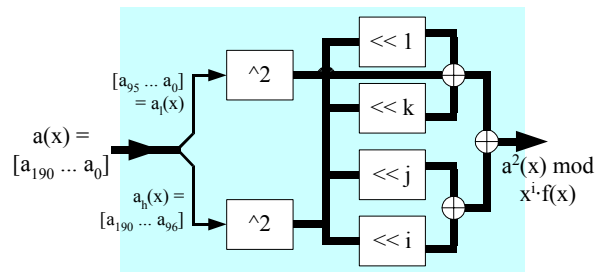


Figure 6: Square unit for  $GF(2^{191})$

## 5.2 Controlpath

The controlpath of the EC-processor spans the ECC-control-unit and the interface unit. The ECC-control-unit sequences the scalar-multiplication  $Q=k \cdot P$  of an elliptic-curve point. This operation is similar to Algorithm 1: It doubles the point  $Q_i$  in every iteration and conditionally adds the point  $P$  if the actual bit  $k_i$  of the scalar  $k$  is set. Initially,  $Q_i$  is set  $Q_i=P$ , which requires detecting the highest bit set in  $k$ . Doubling and adding of points uses a projective representation of curve points and are non-trivial operations: Point doubling comprises 4 additions, 5 square operations, and 5 multiplications in  $\text{GF}(2^m)$ ; Point addition involves 8 additions, 4 square operations, and 10 multiplications. The scalar-multiplication concludes by converting the point  $Q$  into affine coordinates which involves 1 inversion (Algorithm 2), 1 square operation, and 2 multiplications in  $\text{GF}(2^m)$ .

All these operations have to be sequenced by the ECC-control-unit. The sequence was optimized to eliminate idle cycles of the datapath and to minimize communication between the arithmetic unit and the register-file. A complete scalar-multiplication for our  $\text{GF}(2^{191})$ -example takes more than 50,000 clock cycles when a radix-256 multiplier is used. The performance of an EC-processor can easily be jeopardized by the control logic. A straightforward implementation of the control unit will surely result in a circuit where the critical path of the whole processor is in the control unit. To prevent this undesired event, the ECC-control-unit is subdivided into a state machine and a microprogram. The state machine is realized with multi-level logic and controls major phases of the scalar-multiplication: Preshifting the scalar  $k$  to detect the highest bit set, the double-and-add iterations, and the final projective-to-affine conversion. The microprogram generates the control signals for multi-cycle operations like point addition or point doubling. These operations are controlled by patterns stored in a ROM table. The microprogram also contains control patterns used for IO. IO-operations can either transfer an  $m$ -bit value from a specified address of the register-file to the arithmetic unit or vice versa. Another IO-operation initiates a shift operation of the arithmetic unit where data from the PCI-interface is shifted in. The microprogram contains a 128x10-bit ROM where the patterns are stored and a 7-bit address counter. A 10-bit ROM-word contains 2 bits for sequence control, 5 bits for controlling the register-file (4-bit address, 1-bit write signal), and 3 bits for selecting the desired operation of the arithmetic unit.

The chosen architecture of the controlpath is an efficient compromise between bulky multi-level logic and a complex but freely programmable micro-controller. For an FPGA implementation this approach seems to be ideal, as ROMs are fast and small, and costly Boolean logic is kept small.

## 6 PCI interface and software

The Cesiums XC2S\_EVAL-card [13], which is used as hardware platform, already contains a PCI-bridge. Therefore, it is only necessary that the EC-processor has an interface to the PCI-chip Pita-II [14]. This interface distinguishes between data transfer and command invocation by separate addresses. Data transfer (either reading or writing) initiates an IO-operation of the processor, which shifts the content of the arithmetic unit to the left and places input data at the least-significant bits. The highest bits can be retrieved by a read operation. Command invocation is limited to a small number of operations to keep the software interface concise. Commands supported by the elliptic processor are summarized in Table 1.

Command	Name	Operation
0000AAAA	READ	Register $C \leftarrow \text{regfile}[AAAA]$
0010AAAA	WRITE	$\text{Regfile}[AAAA] \leftarrow \text{register } C$
I100xxxx	MULT	Scalar-multiplication
I110xxxx	ADD	Point addition

**Table 1: Command summary**

The termination of a command can either be determined by polling the *busy* flag in the status register or a software interrupt is triggered. Interrupt notification is activated by setting the *I*-flag during command invocation.

Software that uses the PCI-card for accelerating EC-cryptography communicates with the card via drivers. The card itself is shipped with a kernel-mode driver for Windows NT. This driver provides access to the card via the PCI-bus and allows configuring the FPGA with a Xilinx bitstream that turns the FPGA into the EC-processor. Furthermore, the driver offers routines for communicating with the FPGA. A dynamic link library (DLL) written in C++ uses these routines to group them to new functions. The new functions are meaningful for writing elliptic-curve crypto-primitives, as they allow multiplying curve points by scalars and adding points after scalar-multiplication.

For demonstration and evaluation purposes we integrated the elliptic-curve accelerator in the Java™ Cryptography Extension (JCE) implemented by IAIK [16]. Applications based on this cryptographic toolkit will be accelerated substantially. If no card is present, the toolkit automatically uses the slower software implementation. Even if such a toolkit is not available, the integration of the card is simple as it offers all elliptic-curve specific functions for implementing digital-signature or key-agreement algorithms. The remaining task for the software is to call the DLL routines and to calculate a few integer operations.

## 7 Results

The EC-processor was modeled with VHDL. The VHDL-code was written in a style that it is well suited for synthesis. Special care was taken that the register-file will be implemented as RAM and that the  $w$ -bit XOR function of the multiplier will be synthesized as a tree structure to keep the critical path short.

An important issue of the VHDL model is its scalability. All parameters, which are important for scaling the EC-processor for performance (multiplier radix) or to optimize it for different finite fields (field size and irreducible polynomial) are adjustable. A re-synthesis of the VHDL code with new parameters will produce an optimized hardware in minutes.

Size $m=191$	Complexity [LUT]	$k \cdot P$ [cycles]	$f_{\max}$ [MHz]	$k \cdot P$ / sec @66 MHz
$w = 8$	2,563	62,296	74.6	1059,4
$w = 16$	6,454*	36,905	71.3	1788,3
$w = 32$	9,881*	24,205	70.4	2726,9

**Table 2: Results for different multiplier sizes**

Table 2 summarizes the synthesis results for different multiplier sizes. The complexity of the circuit is measured in Look-Up Tables (LUT), which are the basic elements of Xilinx FPGAs. LUT-values marked with an asterisk indicate that the circuit is too large to fit the target FPGA Spartan-II XC2S200. The maximum clock frequency is 70 MHz. When the card is clocked with a 66 MHz oscillator, more than 1,000 scalar-multiplications can be calculated per second. These figures already include IO.

## 8 Conclusions

The presented PCI-card is a novel solution to speed up elliptic curve cryptography. Its application in e-Commerce will accelerate digital signatures and key-agreement algorithms. The hardware realization of the elliptic curve processor is based on a medium sized FPGA. The processor is able to calculate the computational-intensive scalar multiplication of points on elliptic curves over the finite field  $GF(2^m)$ . Internally, it operates with projective coordinates but it even performs the back transformation to affine coordinates. The processor is optimized for a particular finite field. The core of its performance-optimized arithmetic unit is a digit-serial multiplier. As all other relevant parameters of the circuit, its degree of parallelism can be adjusted in the scaleable VHDL model. The current 191-bit version with a radix-256 multiplier achieves a throughput of 1000 scalar-multiplications per second when clocked at 66 MHz.

## References

- [1] R. Posch, G. Karlinger, D. Konrad, A. Leiningen-Westerburg, Th. Menzel: *Weißbuch Bürgerkarte (German)*, A-SIT white paper, [www.buergerkarte.at](http://www.buergerkarte.at), Mai 2002.
- [2] J. Großschädl, G. A. Kamendje, E. Oswald, R. Posch: *Elliptic Curve Cryptography in Practice - The Austrian Citizen Card for e-Government Applications*, Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science and Medicine on the Internet, Jan. 2002.
- [3] A. Menezes, P. van Oorschot, S. Vanstone: *Handbook of Applied Cryptography*, CRC Press, New York, 1997.
- [4] I. Blake, G. Seroussi, N. Smart: *Elliptic Curves in Cryptography*, London Mathematical Society Lecture Note Series, vol. 265, Cambridge University Press, 1999.
- [5] ANSI X9.62: *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, 1999.
- [6] ANSI X9.63, *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Key Transport Protocols*, 1999.
- [7] AEP Systems: *AEP SSL Accelerator™*, data sheet, [www.aep.ie](http://www.aep.ie), 2002.
- [8] D. Hankerson, J. L. Hernandez, A. Menezes: *Software Implementation of Elliptic Curve Cryptography Over Binary Fields*, Proceedings of CHES 2000 Workshop, Springer-Verlag, LNCS 1965, 2000.
- [9] G. B. Agnew, R. C. Mullin, S. A. Vanstone: *An Implementation of Elliptic Curve Cryptosystems over  $F_{2^{155}}$* , IEEE Journal on Selected Areas in Communications, Vol. 11, No. 5, Jun. 1993.
- [10] J. Goodman, A. P. Chandrakasan: *An Energy-Efficient Reconfigurable Public-Key Cryptography Processor*, IEEE Journal of Solid-State Circuits, Vol. 36, No. 11, Nov. 2001.
- [11] M. Ernst, S. Klupsch, O. Hauck, S. A. Huss: *Rapid Prototyping for Hardware Accelerated Elliptic Curve Public-Key Cryptosystems*, 12<sup>th</sup> Workshop on Rapid System Prototyping, Monterey, CA, Jun. 2001.
- [12] G. Orlando, Ch. Paar: *A High-Performance Reconfigurable Elliptic Curve Processor for  $GF(2^m)$* , Proceedings of CHES 2000 Workshop, Springer-Verlag, LNCS 1965, 2000.
- [13] Cesium GmbH: *XC2S\_EVAL User Manual*, User Manual, Version 1.3, [www.cesium.com](http://www.cesium.com), Aug 2001.
- [14] Infineon Technologies: *PITA-2 – PCI Interface for Telephony/Data Applications*, Preliminary Data Sheet, 1.0, [www.infineon.com](http://www.infineon.com), 2000.
- [15] Xilinx Inc.: *Spartan-II FPGA Family*, Preliminary Product Specification, v2.3, [www.xilinx.com](http://www.xilinx.com), Nov 2001.
- [16] IAIK: *IAIK JCE-Toolkit*, Institute for Applied Information Processing and Communications, TU-Graz, [jcewww.iaik.at](http://jcewww.iaik.at).