Reconfigurable Implementation of Elliptic Curve Crypto Algorithms

M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, J. Teich
University of Paderborn, Paderborn, Germany
{bednara, daldrup, teich}@date.upb.de, {gathen, jamshid}@upb.de

Abstract

For FPGA based coprocessors for elliptic curve cryptography, a significant performance gain can be achieved when hybrid coordinates are used to represent points on the elliptic curve. We provide a new area/performance tradeoff analysis of different hybrid representations over fields of characteristic two. Moreover, we present a new generic cryptoprocessor architecture that can be adapted to various area/performance constraints and finite field sizes, and show how to apply high level synthesis techniques to the controller design.

1 Introduction

Currently, there are two popular kinds of cryptographic protocols, namely public key and symmetric key protocols. In the symmetric key protocols, a common key is used by both communication partners and for both encryption and decryption. Among them are DES, IDEA and AES. These systems provide high speed but have the drawback that a common key must be established for each pair of participants. In public key protocols we have two keys, one is kept private by the owner and used either for decryption (confidentiality) or encryption (signature) of messages. The other key is published to be used for the reverse operation. RSA, ElGamal and DSA are examples of public key systems. These systems are slower than the symmetric ones, but they provide arbitrarily high levels of security and do not require an initial private key exchange.

In real applications, both types are used. The public key algorithm first establishes a common symmetric key over a insecure channel. When this key expires after some time, a new key is established via a public key algorithm. Then the symmetric system is used for secure communication with high throughput.

Due to the comparative slowness of the public key crypto algorithms, dedicated hardware support is desirable. In this paper, we present an area versus performance tradeoff for FPGA based implementations of a cryptographic coprocessor using elliptic curve crypto algorithms. FPGA based cryptoprocessors avoid a series of drawbacks of ASIC based systems:

- A cryptography algorithm is secure as long as no effective attack is found. If this happens, the algorithm must be replaced. FPGAs facilitate a fast and cost effective way of exchanging the algorithm, in particular of switching to a higher key length.
- In electronic commerce servers, cryptographic algorithms must be changed often for the purpose of adaption to the current workload, depending on the type of cryptography that is mainly used (public key or symmetric). This can be done by exploiting the FPGAs reconfiguration facility.
- Elliptic curve cryptosystems facilitate several degrees of freedom like Galois field characteristic, extension degree, elliptic curve parameters or the fixed point generating the working subgroup on the curve (see Section 7). FPGAs allow for an effortless adaption to changing security requirements.
- The empirical results of testing various approaches on an FPGA may later be of help in designing an efficient ASIC, where such experiments would be much more costly.

The paper is organized as follows. Section 2 discusses some previous work. In Section 3 we give a short introduction into the theory of elliptic curve cryptography. Section 4 is a survey of the required finite field operations along with an area/performance analysis of different field multiplier architectures. Sections 5 and 6 give a performance analysis of various combinations of point multiplication algorithms and coordinate representations. In Section 7 we present a flexible and generic cryptoprocessor architecture and some experimental and theoretical results. Section 8 summarizes the results.

This work has been supported by DFG Sonderforschungsbereich 376 “Massive Parallelität.”

2 Related work

FPGA based elliptic curve coprocessor implementations have been reported in [6], [21], [5], and [23].

[6] is probably the first of these implementations. It uses a normal basis to represent the finite field and the Massey-Omura multiplier to multiply two elements of the field. The implementation in [21] is the fastest reported implementation. It uses the Montgomery method for point multiplication which is introduced in [17] and modified for fields of characteristic 2 in [14]. Here in each iteration only the X and Z coordinates of the intermediate points are computed. The computation of the Y coordinate of the result is postponed to the last iteration of the algorithm. Such an implementation is also strong against side channel attacks, because the number of operations does not depend on the multiplier bits. On the other hand, the performance of the method cannot be improved by using addition-subtraction chains. Another point of this article is to exploit the reconfigurable structure of FPGAs for fast squaring in polynomial bases, which was believed to be slow compared to squaring in normal bases. This has been theoretically analyzed in [26]. [5] is another implementation of the Montgomery method in the normal basis representation. [23] discusses the Hessian form of an elliptic curve, another representation of elliptic curves which can be used to parallelize the point multiplication algorithm. Unfortunately this method can only be applied to curves whose groups of points have a cyclic subgroup of order 3. This restricts the selection of elliptic curves. E.g. the elliptic curves which are suggested by NIST cannot be implemented with this method.

In this work we compare several possibilities for hardware implementation of elliptic curve point multiplication. Our comparison will include different point representations like the mixed coordinate representation. In addition we analyze the effect of parallelism on the overall performance. This topic has been already briefly analyzed in [23].

3 Elliptic Curve Cryptography

This section gives a short introduction into the application of elliptic curves in the area of cryptography [1].
The points of an elliptic curve defined over a finite field form a finite group, and the group operation is point addition. The basic operation in elliptic curve cryptosystems is the computation of $m\mathcal{P}$, where $\mathcal{P}$ is a point and m a (large) integer. The computation of $m\mathcal{P}$ is done as a sequence of repeated point additions and doublings. Elliptic curve cryptosystems (ECCs) rely on the fact that solving the discrete logarithm problem on an elliptic curve is a hard task. That means, for a given $\mathcal{P}$ and m, computing $m\mathcal{P}$ is of polynomial complexity, but computing m from only $\mathcal{P}$ and $m\mathcal{P}$ is, in general, assumed to be infeasible in polynomial time [16]. Some care has to be exercised in order to avoid special curves with easy discrete logarithms. For a field of characteristic two, the minimum number of bits required to represent the finite field elements is recommended to be larger than 160 to resist "generic" attacks. ECCs defined over such fields are assumed to be as secure as RSA systems with 1024 bits [1]. The short keys make elliptic curve cryptosystems attractive in communication systems with tight bandwidth limitations.
Fig. 1 shows how the ultimate goal, namely point multiplication, naturally decomposes into a hierarchy of three levels. The top two ones use essentially the subroutines provided at the level just below it. Each level can be optimized in order to meet the given area/performance constraints.

Point multiplication:
Double and add method, addition subtraction chains.

Point addition and doubling:
Selection of point representation method.
Affine, projective, Jacobian, or mixed representation.

Finite field arithmetic:
Selection of basis, multiplier and inverter structures.

Figure 1. Hierarchical levels of elliptic curve point multiplication.

The register transfer level could be seen as a fourth level below the field arithmetic level. However, logic optimization is beyond the scope of this paper.

4 Finite field arithmetic

Every finite field $GF(2^{m \cdot n})$ can be expressed as a vector space over $GF(2^m)$. We consider here the special case of $m = 1$. In this case every element of $GF(2^n)$ is a vector of length n with coefficients in $GF(2)$, i.e either 0 or 1. Such a representation is suitable for hardware implementation of arithmetic in $GF(2^n)$. Every element can be represented with n one-bit registers.

There are several types of vector space bases that can be used to represent the finite field. They have various advantages and drawbacks for efficient implementation of arithmetic. In this section we consider each of the field operations addition, multiplication, squaring, and division with respect to different bases.

4.1 Addition

The addition of two elements of $GF(2^n)$ is the XOR of two n-bit sequences, independent of the choice of vector space basis over $GF(2^n)$.

4.2 Multiplication

One of the most resource consuming operations in finite field arithmetic is multiplication. Division can be (and is normally) implemented using consecutive multiplications. Therefore there is a lot of attention to finite field multiplication in the literature. Multiplication can be implemented either in parallel or serially. Parallel multipliers perform the total multiplication in one clock cycle but require more space to be implemented. Since this clock cycle propagates through a long path due to large area implementation, pipelining can be used to break the path into more logic stages, allowing higher clock rates at the cost of more clock cycles. But here again each result can be computed with only one clock cycle if the pipeline stages are fed with data at appropriate times. Examples of parallel multipliers can be found in [24], [22].

Serial multipliers on the other hand require smaller area and have a less complex structure, but generate one or a few bits of the result in each clock cycle ([20], [25]).

Parallel multipliers are especially suitable for small finite fields ($n < 64$) since they require a large area on the order of n^2 for school method and $n^{1.59}$ for Karatsuba multipliers.

Indeed there are asymptotically fast multiplication algorithms which can improve the software performance (see [8]). These fast multiplication methods have been used to factorize a polynomial of degree over $1,000,000$ ([2]). But to our knowledge there is no efficient hardware implementation of them. For elliptic curve systems to be secure, we need $n > 160$, and will use $n = 191$. Here serial multipliers are more attractive and we consider only these kinds of multipliers in this section.

- **Polynomial basis LFSR[1] multiplier**: To generate a polynomial basis for the field $GF(2^n)$, it suffices to find a polynomial $f(x)$ of degree n which is irreducible over $GF(2)$. Such a polynomial will have a root ω in $GF(2^n)$ ([15], [12]), and the n distinct powers $\{1, \omega, \omega^2, \cdots, \omega^{n-1}\}$ of ω constitute a basis for $GF(2^n)$ over $GF(2)$. LFSR multipliers use such a basis; see [20] for an FPGA implementation. It can be implemented as bit-level multiplier which generates the result in n clock cycles or as a word-level multiplier of length D which multiplies two elements in $m = \lceil \frac{n}{D} \rceil$ clock cycles using $(n-D)+(D-1)(mD-1) + 2\sum_{j=0}^{D-2} w(\omega^{n+j}) + w(\omega^{n+D-1})$ XOR gates and mD^2 AND gates, where $w(\omega^k)$ is the Hamming weight or number of nonzero coefficients in the representation of ω^k.

- **Normal basis multiplier**: Normal bases are attractive since squaring is very easy in such a basis. Because of this characteristic, it is desirable to multiply two elements represented in this basis with as few resources as possible ([19], [7], [24]). One normal basis multiplier is the Massey-Omura multiplier [19] which is very flexible according to time and area. One can implement D similar pieces and perform the total multiplication in $m = \lceil \frac{n}{D} \rceil$ clock cycles.

  The Massey-Omura multiplier requires at least $D(2n-1)$ AND gates and $D(2n-2)$ XOR gates.

- **Dual basis:** There are also multipliers for the dual basis of polynomial basis like the Berlekamp multiplier [15]. We have not considered them.

- **Comparison of different multipliers:** We have compared the number of logic elements to implement both LFSR polynomial basis multiplier and normal basis multiplier for $GF(2^{191})$ in Figure 2, and also the propagation times. There is no general formula to compute the propagation delay of a LFSR multiplier. Values have been measured by analyzing the schematic diagram of this multiplier. The normal basis is assumed to be an optimal normal basis. The LFSR multiplier is generated with the irreducible polynomial $f(x) = x^{191} + x^9 + 1$.

As can be seen from the figures, a LFSR multiplier always requires fewer logic elements. The propagation delay is smaller than of the Massey-Omura for a word length less than 18. For the word lengths between 18 and 32, which correspond to a number of clock cycles between 6 and 12, both multipliers have the same propagation delay. Since circuits with larger word length require a lot of logic elements, we have used a bit-level LFSR multiplier. The design is so modular that changing the multiplier can be done easily.

4.3 Squaring

Squaring is a special case of multiplication, we have to multiply an element by itself. But there are algorithms that perform squaring much better than multiplication. So we examine the cases of normal bases and polynomial bases separately, and have an element γ represented by $(c_0, \ldots, c_{n-1}) \in GF(2)^n$.

- **Normal basis:** In this case γ^2 will be represented by $(c_{n-1}, c_0, \ldots, c_{n-2})$. It means that squaring in normal basis is only a circular shift [16].

[1] Linear feedback shift register

- **Polynomial basis:** In this case we have :

$$\gamma^2 = \sum_{i=0}^{n-1} c_i \omega^{2i}. \qquad (1)$$

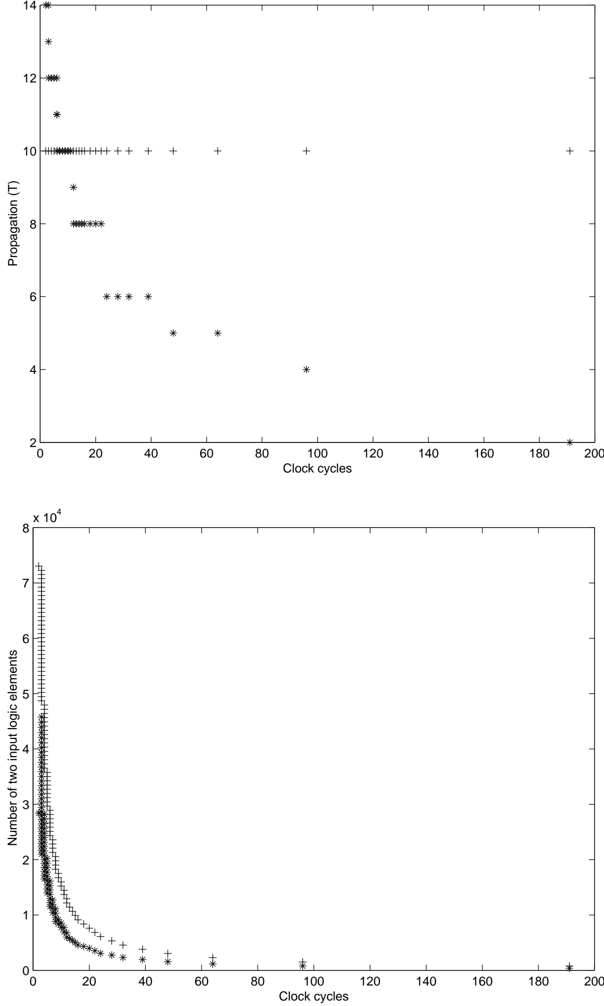


Figure 2. Propagation delay (above) and number of 2-input logic gates (below) of both LFSR (*) and normal basis multiplier (+) for one multiplication in $GF(2^{191})$ (assuming $T_{and} = T_{xor} = T$).

We have to reduce the polynomial corresponding to (1), of degree at most $2n - 2$, modulo $f(x)$. This can be done with $(r-1)(n-1)$ two-input gates, where r is the Hamming weight of $f(x)$ ([26], [10]). It is conjectured that we can always choose $r \leq 5$, and often $r = 3$ ([10]). So squaring will consist of a permutation followed by a modular reduction. The later can be reimplemented separately for each finite field using the reconfigurable structure of FPGAs.

Table 1. Sequence of multiplications and squarings for inversion in $GF(2^{191})$

0:	$y_1 \leftarrow x$	$\{x^{2^1-1}\}$
1:	$y_2 \leftarrow y_1^2 \cdot y_1$	$\{x^{2^2-1}\}$
2:	$y_3 \leftarrow y_2^2 \cdot y_1$	$\{x^{2^3-1}\}$
3:	$y_5 \leftarrow y_3^{2^2} \cdot y_2$	$\{x^{2^5-1}\}$
4:	$y_{10} \leftarrow y_5^{2^5} \cdot y_5$	$\{x^{2^{10}-1}\}$
5:	$y_{20} \leftarrow y_{10}^{2^{10}} \cdot y_{10}$	$\{x^{2^{20}-1}\}$
6:	$y_{40} \leftarrow y_{20}^{2^{20}} \cdot y_{20}$	$\{x^{2^{40}-1}\}$
7:	$y_{80} \leftarrow y_{40}^{2^{40}} \cdot y_{40}$	$\{x^{2^{80}-1}\}$
8:	$y_{85} \leftarrow y_{80}^{2^5} \cdot y_5$	$\{x^{2^{85}-1}\}$
9:	$y_{95} \leftarrow y_{85}^{2^{10}} \cdot y_{10}$	$\{x^{2^{95}-1}\}$
10:	$y_{190} \leftarrow y_{95}^{2^{95}} \cdot y_{95}$	$\{x^{2^{190}-1}\}$
11:	Output $\leftarrow y_{190}^2$	$\{x^{2^{191}-2}\}$

4.4 Inversion

There are two ways to perform inversion in a finite field. One approach is via the Extended Euclidean Algorithm, and another via Fermat's theorem:

- **Euclidean algorithm:** This algorithm requires $O(n^2)$ bit operations to perform an inversion in $GF(2^n)$.
- **Fermat's theorem:** For every element $\gamma \in GF(2^n)^\times$, we have $\gamma^{2^n-1} = 1$, and hence $\gamma^{-1} = \gamma^{2^n-2}$.

  The binary representation of $2^n - 2$ has n digits and $n-1$ of them are equal to 1. So it requires $n-1$ multiplications and n squarings by double and add method. This cost can be reduced to n squarings and $O(\log(n))$ multiplications ([9]). The sequence of 10 multiplications and 190 squarings for the case $n = 191$ is shown in Table 1.

5 Point addition and doubling

Any point multiplication will be done with a sequence of point additions, so to minimize the total cost one should consider both the point addition algorithm and the sequence in which the operations will be performed.

In this section we consider three different representations of a point on an elliptic curve and study their effect on point addition and doubling costs.

5.1 Coordinate Representations

Most point multiplication methods are based on repeated addition/subtraction and doubling of points (see Section 6). In an addition, one of the two arguments is fixed. This allows a simpler and faster computation logic in certain cases.

In this subsection, we introduce different representations of points on elliptic curves and show how some hybrid forms of them can be used to enhance the computation logic in dependence of various area and performance constraints. Hybrid coordinate representations have been discussed in [4] and [13].
The most popular coordinate representation is the affine representation which is based on two coordinates (x, y). Other representations are projective, Jacobian and López-Dahab representation [13] each of which uses three coordinates (X, Y, Z). Transforming affine coordinates into one of the other representations is almost trivial, but not vice versa since the back transformation requires (expensive) field inversions. Table 2 gives the formulas for mapping into affine coordinates.

Coordinate Representation	Mapping to Affine Coordinates
Projective	$x = X/Z,\ y = Y/Z$
Jacobian	$x = X/Z^2\ \ y = Y/Z^3$
López-Dahab	$x = X/Z\ \ y = Y/Z^2$

Table 2. Mapping of Projective, Jacobian, and López-Dahab representation into affine coordinates.

We have analyzed two different hybrid coordinate representations with respect to the suitability for performance enhancement of the computation logic, namely affine/Jacobian and affine/López-Dahab. We call these representations *mixed coordinate I* and *mixed coordinate II*, respectively. Moreover, we analyze the Montgomery method ([17], [14]) for point addition and doubling which is actually not a hybrid coordinate representation but uses the Jacobian coordinates. The Montgomery method is based on the observation that the x coordinate of the sum of two points $\mathcal{P}+\mathcal{Q}$ depends only on the x and z coordinates of the two points $\mathcal{P}$ and $\mathcal{Q}$, if $\mathcal{Q} = \mathcal{P} + \mathcal{S}$ for a fixed point $\mathcal{S}$. This method keeps the condition $\mathcal{Q} = \mathcal{P}+\mathcal{S}$ in each iteration step of the point multiplication, and thus only works with very special addition chains.

Table 3 compares the methods for point addition and point doubling in mixed coordinate I and II and the Montgomery method. The table gives the number of multiplications $\mathcal{M}$, squarings $\mathcal{S}$ and additions $\mathcal{A}$ in the underlying finite field.

The number of field operations in Table 3 gives only a coarse estimation for the time and area requirements of the computation logic. A more detailed analysis is given in the next subsection.

Representation	Addition	Doubling
Mixed coord. I	$10\mathcal{M} + 5\mathcal{S} + 8\mathcal{A}$	$5\mathcal{M} + 5\mathcal{S} + 4\mathcal{A}$
Mixed coord. II	$14\mathcal{M} + 4\mathcal{S} + 10\mathcal{A}$	$6\mathcal{M} + 4\mathcal{S} + 4\mathcal{A}$
Montgomery	$4\mathcal{M} + 1\mathcal{S} + 2\mathcal{A}$	$2\mathcal{M} + 4\mathcal{S} + 1\mathcal{A}$

Table 3. Costs of different point addition and doubling methods in terms of field operations, in characteristic 2.

6 Point Multiplication Algorithms

The task of point multiplication is to compute $m\mathcal{P}$ from m and $\mathcal{P}$, assuming that we know how to add two points and also how to double a point [2].

By Hasse's bound, the elliptic curve has at most $k = 2^n + 2^{n/2+1}$ points, and we can always reduce m to satisfy $0 \leq m < k$.

The standard method for multiplication by an integer m is the double and add method which uses the binary representation of m.

In this method all the bits in the binary representation of m except the first one are traversed from left to right. For each 0 a doubling will be performed, and for each 1 a doubling and an addition with the original point. Since for a random n bit number m, on average $\frac{n}{2}$ bits equal 1, the total number of operations for a complete point multiplication is about n doublings and $\frac{n}{2}$ additions.

The number of operations can be reduced by using better addition chains [3] or addition subtraction chains [11], [18]. If the bit sequence is processed to produce an addition subtraction chain, and considering that computing inverse of a point is a trivial task we require only $\frac{n}{3}$ additions but the number of doublings will be kept fixed.

The number of clock cycles required for a point multiplication (plus the conversion into affine coordinates) is shown in Figure 3. All multipliers are serial LFSR multipliers in $GF(2^{191})$. Each triple of bars corresponds to a complete point multiplication and a subsequent coordinate transformation using the above mentioned methods and a fixed number of multipliers.

The number of required clock cycles reduces as more multipliers are used until the maximum degree of parallelism in the point multiplication algorithm is reached. For the Montgomery representation, the maximum required number of multipliers is 2, for the López-Dahab representation it is 3, and for the affine/Jacobian representation 4. These diagrams hold only for multipliers with a latency of more than 5 clock cycles. For smaller values, the latencies of memory accesses and other functional units limit the performance.

[2]The formula for doubling is not that for addition with two identical arguments.

7 Architecture

In this section, we present a generic datapath architecture that can be easily adapted to various performance/area constraints by exploiting several degrees of freedom, namely:

- number of functional units in the data path,
- degree of parallelism in the multiplier units,
- type of point multiplication algorithm,
- implementation type of the controller,
- coordinate representation.

The impact of the number of functional units and the coordinate representation on the overall performance is shown in Fig. 3. An area and performance estimation of multipliers of different degrees of parallelism is already given in Section 4.2. For the controller we can use either a hardwired finite state machine or a microprogrammable controller. For performance measures, we have implemented a sample architecture on a prototyping system with a Xilinx Virtex FPGA (XCV1000-BG560-4).

7.1 Datapath Architecture

The structure of the datapath architecture is shown in Figure 4. We use two squarers, two adders and four sequential LFSR multipliers in our sample architecture. Each of these arithmetic units (AU) can get operands from a dual-port operand memory, a register, or directly from the output of another arithmetic unit.

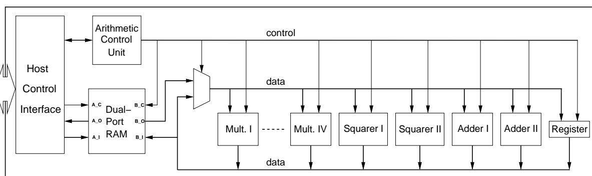


Figure 4. Datapath structure.

The arithmetic control unit (ACU) generates control signals for all AUs, the operand memory and the register.
The second port of the operand memory is used by a host interface, thus allowing for host data transfer while a point multiplication is being performed.

The AUs consist of one or two operand registers, one output register and the core functional unit.

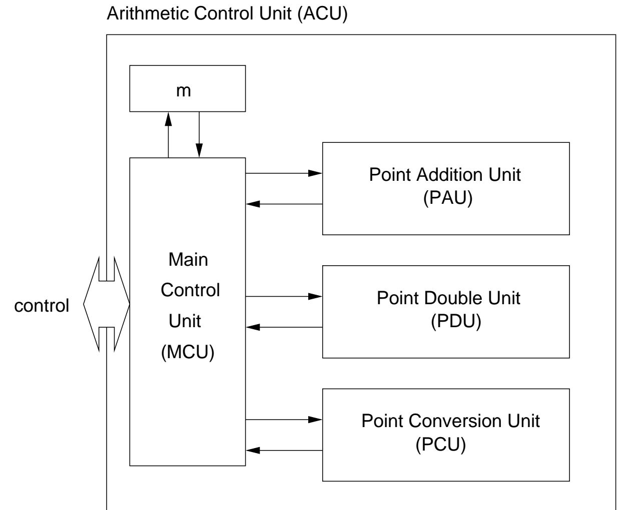


Figure 5. The Arithmetic Control Unit (ACU).

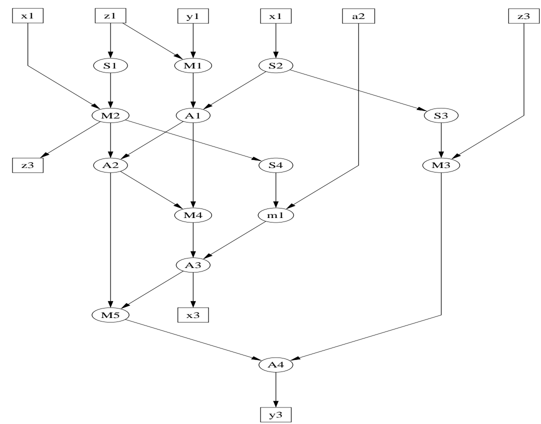


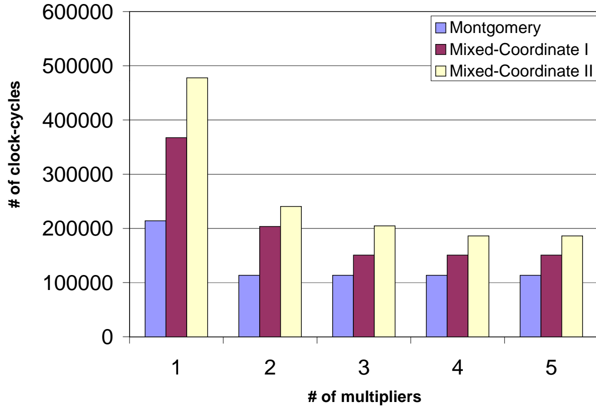
Figure 6. Data dependency graph for point doubling in mixed coordinates type II.

7.2 Controller Architecture

For our sample implementation, we use a hardwired controller state machine. The FSM is structured hierarchically as shown in Figure 5, which reduces the state set and increases the design reusability.

The PAU and PDU units generate control signals for the AUs in order to perform a point addition or point doubling, respectively. The algorithm implemented by the PDU is given exemplarily as a data dependency graph in Figure 6. The A, S, and M gates perform an addition, squaring, and multiplication, respectively.

The PCU generates control signals to perform a coordinate transformation for Jacobian/affine conversion, which requires also a field inversion operation (Section 4.4).



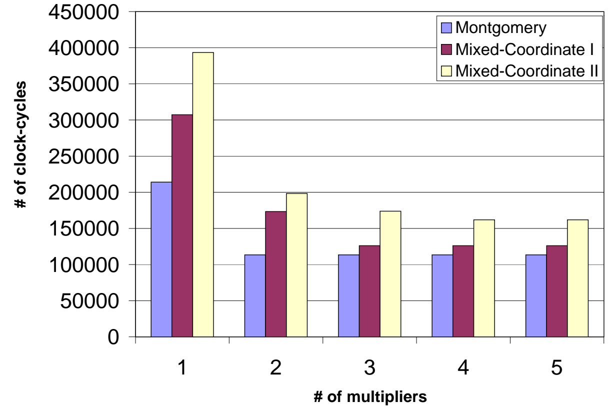


Figure 3. Required number of clock cycles for point multiplication in double and add method (left) and addition subtraction chains (right).

The point multiplication algorithm according to the double-and-add method (Section 6) is performed by the MCU. For this purpose, MCU provides control signals for PAU, PDU and PCU.

7.3 Merits of the Modular Structure

The modular structure of the datapath and the hierarchical design of the controller allows a high design reusability when adapting the design to given area and performance constraints.

- Replacing the sequential multipliers by parallel units requires no modifications to the controller.
- Changing the point multiplication algorithm requires only modification of the MCU.
- Changing the coordinate representation requires only modifications of PAU, PDU, and PCU.
- Changing the degree of the field extension (i.e. the key length) requires no modifications to PAU, PDU, and PCU.

The PAU, PDU, and PCU were designed using high level synthesis techniques. For generating the resource constrained schedules, we used an LP based scheduler. From these automatically generated schedules the state machines can easily be derived and coded in VHDL.

7.4 Performance Measures

Our prototype implementation on a Xilinx Virtex FPGA (XCV1000-BG560-4) operates at a clock period of 20ns. The total time for a point multiplication in our implementation is given in the first row of Table 4 as well as theoretical

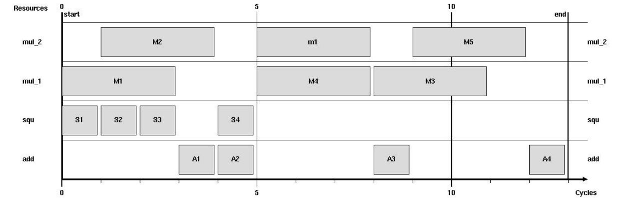


Figure 7. Schedule for the point doubling operation with two multipliers.

performance values for some other selected configurations using serial LFSR multipliers (assuming 20ns clock period for all configurations).

Configuration	PMult.	Conv .	Total [ms]
4Mult,Double&Add, Mixed Coordinate II	183742	2482	3.72
2Mult,Double&Add, Mixed Coordinate I	201220	2290	4.07
1Mult,AddSubChain, Montgomery Coord.	210673	3438	4.28
2Mult,AddSubChain, Montgomery Coord.	110207	3248	2.27

Table 4. LFSR Latency of point multiplication, conversion and total operation

Table 5 gives performance values when a parallel Massey-Omura multiplier is used instead of a LFSR multiplier. Our Massey-Omura implementation requires 7 clock cycles for one multiplication but an extremely large CLB area (about 48300 LUTs), so Table 5 covers only implementations with a single multiplier at clock period of 27.8ns.

Configuration	PMult.	Conv .	Total [ms]
1Mult,Double&Add, Mixed Coordinate II	17954	274	0.50
1Mult,Double&Add, Mixed Coordinate I	16331	268	0.46
1Mult,AddSubChain, Montgomery Coord.	9550	310	0.27

Table 5. Massey-Omura Latency of point multiplication, conversion and total operation

As in Section 6, we assume for both tables that the number of 1-bits in the multiplier m is $\frac{n}{2}$ for the double and add algorithm coordinates and $\frac{n}{3}$ for addition subtraction chains, where $n = 191$ is the number of bits of m.

8 Conclusion and summary

Our work gives an area/speed tradeoff analysis for hardware implementations of elliptic curve cryptography algorithms. We have shown that in each case, the Montgomery representation gives the fastest point multiplication method even when there is no limitation for the area. In this method the maximum number of multipliers that can be used are two multipliers. The best choice of representation basis is the polynomial basis. The normal bases are advantageous only when the available area is very large, so that a Massey-Omura multiplier with a high throughput can be used.

Our tradeoff analysis concerns different characteristic two field multipliers with respect to area and performance. Moreover, we analyzed the tradeoff between different elliptic curve point multiplications using various hybrid coordinate representations and multiplication algorithms. In conclusion, we have presented a new generic architecture for an FPGA based elliptic curve coprocessor that can be easily adapted to area and performance constraints and designed using high level techniques. Experimental results give comparison of different configurations of our coprocessor.

References

[1] Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*. Number 265 in London Mathematical Society Lecture Note Series. Cambridge University Press, 1999.

[2] Olaf Bonorden, Joachim von zur Gathen, Jürgen Gerhard, Olaf Müller, and Michael Nöcker. Factoring a binary polynomial of degree over one million. *ACM SIGSAM Bulletin*, 35(1):16–18, March 2001.

[3] A. Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45:736–739, 1939.

[4] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *ASIACRYPT 1998*, number 1514 in LNCS, pages 51–65, 1998.

[5] Markus Ernst and Sorin Huss. Ein FPGA basierter Elliptic-Curve Kryptoprozessor mit variabler Schlüssellänge für hohen Datendurchsatz. In *Entwurf Integrierter Schaltungen und Systemen*, Dresden, April 2001.

[6] Lijun Gao, Sarvesh Shrivastava, and Gerald E. Sobelman. Elliptic Curve Scalar Multiplier Design Using FPGAs. In *CHES '99*, number 1717 in LNCS, pages 257–268, 1999.

[7] Shuhong Gao, Joachim von zur Gathen, Daniel Panario, and Victor Shoup. Algorithms for Exponentiation in Finite Fields. *Journal of Symbolic Computation*, 29(6):879–889, June 2000.

[8] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, UK, 1999.

[9] Joachim von zur Gathen and Michael Nöcker. Computing special powers in finite fields: Extended abstract. In Sam Dooley, editor, *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation ISSAC '99,* Vancouver, Canada, pages 83–90. ACM Press, 1999.

[10] Joachim von zur Gathen and Michael Nöcker. Exponentiation using addition chains for finite fields. submitted, 2002.

[11] Daniel M. Gordon. A survey of Fast Exponentiation Methods. *Journal of Algorithms*, 27:129–146, 1998.

[12] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. Number 20 in Encyclopedia of Mathematics and its Applications. Addison-Wesley, Reading MA, 1983.

[13] Julio López and Ricardo Dahab. Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$. In *Selected Areas in Cryptography*, number 1556 in LNCS, pages 201–212. 1998.

[14] Julio López and Ricardo Dahab. Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation. In *CHES '99*, number 1717 in LNCS, pages 316–327, 1999.

[15] Robert J. McEliece. *Finite Fields for Computer Scientists and Engineers*. The Kluwer International Series in engineering and computer science. Kluwer Academic Publishers, 1987.

[16] Alfred J. Menezes, Ian F. Blake, XuHong Gao, Ronald C. Mullin, Scott A. Vanstone, and Tomik Yaghoobian. *Applications of finite fields*. Kluwer Academic Publishers, 1993.

[17] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, January 1987.

[18] François Morain and Jorge Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *Informatique théorique et Applications/Theoretical Informatics and Applications*, 24(6):531–544, 1990.

[19] Jimmy K. Omura and James L. Massey. Computational method and apparatus for finite field arithmetic. *United States Patent 4,587,627*, 1986.

[20] G. Orlando and C. Paar. A Super-Serial Galois Fields Multiplier for FPGAs and its Application to Public-Key Algorithms. In *FCCM '99*, April 1999.

[21] G. Orlando and C. Paar. A High-Performance Reconfigurable Elliptic Curve Coprocessor for $GF(2^m)$. In *CHES '2000*, number 1965 in LNCS, pages 41–56, 2000.

[22] Christof Paar. A New Architecture for a Parallel Finite Field Multiplier with Low Complexity on Composite Fields. *IEEE Transactions on Computers*, 45(7):856–861, July 1996.

[23] N. P. Smart. The Hessian form of an elliptic curve. In *CHES '2001*, number 2162 in LNCS, pages 118–125, 2001.

[24] B. Sunar and Ç. K. Koç. Mastrovito Multiplier for All Trinomials. *IEEE Transactions on Computers*, 48(5):522–527, July 1999.

[25] C. C. Wang, T. K. Truong, H. M. Shao, L. J. Deutsch, J. K. Omura, and I. S. Reed. VLSI Architectures for Computing Multiplications and Inverses in $GF(2^m)$. *IEEE Transactions on Computers*, C-34:709–717, 1985.

[26] Huapeng Wu. On Computation of Polynomial Modular Reduction. Technical report, Centre of Applied Cryptographic Research, University of Waterloo, June 2000.