Superscalar Coprocessor for High-speed Curve-based Cryptography *

K. Sakiyama, L. Batina, B. Preneel and I. Verbauwhede

Katholieke Universiteit Leuven / IBBT
Department Electrical Engineering - ESAT/SCD-COSIC
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{ksakiyam, lbatina, preneel, iverbauw}@esat.kuleuven.be

Abstract. We propose a superscalar coprocessor for high-speed curve-based cryptography. It accelerates scalar multiplication by exploiting instruction-level parallelism (ILP) dynamically and processing multiple instructions in parallel. The system-level architecture is designed so that the coprocessor can fully utilize the superscalar feature. The implementation results show that scalar multiplication of Elliptic Curve Cryptography (ECC) over GF(2^{163}), Hyperelliptic Curve Cryptography (HECC) of genus 2 over GF(2^{83}) and ECC over a composite field, GF($(2^{83})^2$) can be improved by a factor of 1.8, 2.7 and 2.5 respectively compared to the case of a basic single-scalar architecture. This speed-up is achieved by exploiting parallelism in curve-based cryptography. The coprocessor deals with a single instruction that can be used for all field operations such as multiplications and additions. In addition, this instruction only allows one to compute point/divisor operations. Furthermore, we provide also a fair comparison between the three curve-based cryptosystems.

Keywords: Superscalar, instruction-level parallelism, coprocessor, curve-based cryptography, scalar multiplication, HECC, ECC

1 Introduction

Public-key cryptosystems form an essential building block for digital communication. Unlike secret-key algorithms that allow for a fast encryption of a large bulk of data, the importance of Public-Key Cryptography (PKC) is to have secure communications over insecure channels without prior exchange of a secret key. In addition, PKC enables digital signatures as an important cryptographic service. Diffie and Hellman introduced the idea of PKC [1] in the mid 70's.

Implementing PKC is a challenge for most application platforms varying from software to hardware. The reason is that one has to deal with very long numbers in conditions that are often constrained in area and power. For the choice of the implementation platform, several factors have to be taken into account.

* Kazuo Sakiyama and Lejla Batina are funded by FWO projects (G.0450.04, G.0141.03). This research has been also supported by IBBT-QoE and the EU IST FP6 projects SCARD, SESOC, ECRYPT.

Hardware solutions provide the speed and more physical security, but the flexibility is limited. For that property software solutions are needed, but a pure software solution is not a feasible option in most resource-limited environments. Hardware/software co-design potentially allows an efficient design platform that explores trade-off between cost, performance and security.

The most popular and most widely used public-key cryptosystems are RSA [2] and ECC [3, 4]. In embedded systems, ECC is considered a more suitable choice than RSA because ECC obtains higher performance, lower power consumption, and smaller area on most platforms. Another appealing candidate for PKC is HECC. Recently many good results appear for software and hardware implementations of HECC at the same time more theoretical work has shown HECC to be also secure in the case of curves with a small genus [5].

A considerable amount of work has been reported on improving the performance of Elliptic Curve (EC) scalar multiplication. The work can be classified into following categories: First of all, mathematical investigation has been done for various types of elliptic curves such as Koblitz curves. Secondly, various algorithms for scalar multiplication have been proposed and criteria for improvements include performance as well as side-channel security. One of the best-known examples that meet requirements for both is the Montgomery's powering ladder [6]. Lastly, architecture-level improvements can be considered from a hardware implementations' point of view. Our interest in this paper mainly lies at this level.

The contribution of this paper is in accelerating curve-based cryptosystems by deploying a superscalar architecture. The solution is algorithm-independent and can be applied for any scalar multiplication algorithm. Some previous work reported parallel use of modular arithmetic units for accelerating scalar multiplication [7–12]. In those papers, point/divisor doubling and addition are reformulated so that they can take advantage of the parallel processing. One original contribution is that our proposed architecture embeds an instruction scheduler that explores the best level of parallelism and assigns tasks for the processing units in an optimal way. In this way the parallelism within the operations can be found *on-the-fly* by *dynamically* checking the data dependency in the instructions. We provide also a fair comparison between three cryptosystems, ECC, HECC and ECC over a composite field. Namely, it is known that for HECC of genus 2 one has the ability to work in the field of a size two times smaller than the one for ECC obtaining the same level of security. On the other hand using ECC over $\mathrm{GF}((2^p)^2)$, we end up with the same field arithmetic as HECC. In this way, another contribution of this paper lies in the system architecture of three curve-based cryptosystems enabling one to use the same amount of area.

The remainder of this paper is as follows. Section 2 gives a survey of relevant previous work for curve-based cryptography implementations. In Section 3, some background information on ECC and HECC is given. In Section 4 the architecture for our proposed coprocessor is explained. The details of our implementation are introduced in Section 5 and the results are shown for various implementation options in Section 6. Section 7 concludes the paper.

2 Previous Work

This section lists some relevant previous work. As already mentioned, there is a considerable amount of work done on hardware implementations, especially for ECC [13, 14], but more recently also some on HECC. Recent improvements on HECC divisor operations' formulae [15–17] resulted in several hardware implementations featuring efficient HECC performances [18, 11]. The first result showing that HECC performance is comparable to the one of ECC is the work of Pelzl *et al.* [19].

In 1989 Agnew *et al.* reported the first result for performing the elliptic curve operations on hardware [20]. Since then a substantial amount of work dealt with hardware implementations of ECC, the majority of that over binary fields. In 2000 Orlando and Paar proposed a scalable elliptic curve processor architecture which operates over finite fields $GF(2^n)$ in [13]. Gura *et al.* [14] have introduced a programmable hardware accelerator for ECC over $GF(2^n)$, which can handle arbitrary field sizes up to 255.

There is not much previous work on hardware implementations of HECC. The first complete hardware implementation of HECC was given by Boston *et al.* [21]. They designed a coprocessor for genus two curves over $GF(2^{113})$ and implemented it on a Xilinx Virtex-II FPGA. The algorithm of Cantor was used for all computations on Jacobians. On the other hand, the work of Elias *et al.* [18] used Lange's explicit formulae. The results reported were the fastest in hardware at the time. Wollinger *et al.* investigated an HECC implementation on a VLSI coprocessor. They compared coprocessors using affine and projective coordinates and concluded that the latter should be preferred for hardware implementations [11].

While ECC applications are highly developed and widely used in practice, the use of HECC is still mainly for research purposes. Previous work on exploring the parallelism between the point/divisor operations has been done for both ECC and HECC. Smart [7] showed that up to three field operations could be executed in parallel for the Hessian form of an elliptic curve. On the other hand, the work of Mischra investigated parallelism between divisor operations [10], both purely on algorithmic level.

3 Curve-based Cryptography

Here, we consider some background information for curve-based cryptography over binary fields; for hyperelliptic curves we are interested only in genus 2 curves. We mention the basic algorithms and the structure of the operations. Good references for the mathematical background are [22–24].

The main operation in any curve-based primitive is scalar multiplication. The general hierarchical structure for operations required for implementations of curve-based cryptography is given in Fig. 1(a). Point/divisor multiplication is at the top level. At the next (lower) level are the point/divisor group operations. The lowest level consists of finite field operations such as addition, multiplication

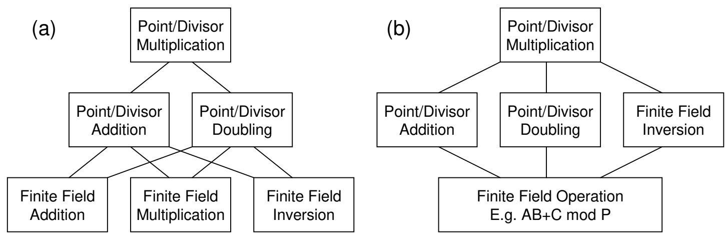

Fig. 1. Scheme of the hierarchy for ECC/HECC operations.

and inversion required to perform the group operations. The only difference between ECC and HECC is in the middle level that in this case consists of different sequences of operations. Those for HECC are more complex when compared with the ECC point operation, but they use shorter operands. One can perform inversion also with a chain of multiplications [25] and only provide hardware for finite field multiplication and addition. The corresponding hierarchy is illustrated in Fig. 1(b). We use this structure for our proposed coprocessor.

3.1 ECC over a binary field

ECC relies on a group structure induced on an elliptic curve. A set of points on an elliptic curve (with one special point added, the so-called point at infinity $\mathcal{O}$) together with a point addition as a binary operation has the structure of an abelian group. As we consider a finite field of characteristic 2, *i.e.* GF(2^n), a non-supersingular elliptic curve E over GF(2^n) is defined as the set of solutions $(x, y) \in$GF$(2^n)\times$GF(2^n) of the equation: $y^2 + xy = x^3 + ax^2 + b$, where $a, b \in$GF$(2^n), b \neq 0$, together with $\mathcal{O}$.

3.2 HECC

Let $\overline{\mathrm{GF}}(2^n)$ be an algebraic closure of the field GF(2^n). Here we consider a hyperelliptic curve C of genus $g = 2$ over GF(2^n), which is given with an equation of the form:

$$C : y^2 + h(x)y = f(x) \quad in \quad \mathrm{GF}(2^n)[x, y], \tag{1}$$

where $h(x) \in$GF$(2^n)[x]$ is polynomial of degree at most g ($deg(h) \leq g$) and $f(x)$ is a monic polynomial of degree $2g + 1$ ($deg(f) = 2g + 1$). Also, there are no solutions $(x, y) \in \overline{\mathrm{GF}}(2^n)\times\overline{\mathrm{GF}}(2^n)$ which simultaneously satisfy the equation (1) and the equations: $2v + h(u) = 0, h'(u)v - f'(u) = 0$. These points are called singular points. For the genus 2, in the general case the following equation is used $y^2 + (h_2x^2 + h_1x + h_0)y = x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$.

A divisor D is a formal sum of points on the hyperelliptic curve C *i.e.* $D = \sum m_P P$ and its degree is $deg(D) = \sum m_P$. Let Div denotes the group of all divisors on C and Div_0 the subgroup of Div of all divisors with degree zero. The

Jacobian J of the curve C is defined as quotient group $J = Div_0/P$. Here P is the set of all principal divisors, where a divisor D is called principal if $D = div(f)$, for some element f of the function field of C ($div(f) = \sum_{P \in C} ord_P(f)P$). The discrete logarithm problem in the Jacobian is the basis of security for HECC. In practice, the Mumford representation according to which each divisor is represented as a pair of polynomials $[u, v]$ is usually used. Here, u is monic of degree 2, $deg(v) < deg(u)$ and $u|f - hv - v^2$ (so-called reduced divisors). For implementations of HECC, we need to implement the multiplication of elements of the Jacobian *i.e.* divisors with some scalar.

3.3 ECC over a composite field

With respect to cryptographic security it is typically recommended to use fields GF(2^p) where p is a prime. As an example we consider the case where $p = 163$. As already mentioned, HECC on a curve of a genus 2 allows one to work in a finite field where bit-lengths are shorter with a factor 2, when compared with ECC. That means, for the equivalent level of security we should choose GF(2^{83}). A similar situation we get when considering ECC over a field of a quadratic extension of GF(2^{83}), so GF($(2^{83})^2$) =GF(2^{83})$[y]/g(y)$ and $deg(g) = 2$. In this way one can obtain a speed-up and benefit even more from the parallelism. The reason is that in composite field each element is represented as $c = c_1 t + c_0$ where $c_0, c_1 \in$GF(2^{83}) and the multiplication in this field takes 3 multiplications and 4 additions in GF(2^{83}) [26].

3.4 Algorithms for our implementations

In our implementations scalar multiplication is achieved by use the NAF algorithm [23]. In this way the scalar is decomposed as a NAF and scalar multiplication is done with a series of addition/subtractions of elliptic curve points. We also use projective coordinates for all implementations.

Furthermore, we have rewritten the formulae from [23, 16] for EC point operations and HECC divisor doubling, respectively to obtain an optimal usage of our new datapath. We use the same approach to get the formulae for HECC divisor addition in the case of mixed coordinates. Our datapath performs one basic operation, $AB + C$ or $A(B + D) + C$ over a binary field. This operation can be used for the sequence of point/divisor operations. For example, by using $A(B + D) + C$ operation the formulae for HECC divisor addition include 48 instructions instead of 44 multiplications and a lot of additions.

4 Architecture of the curve-based coprocessor

4.1 System Architecture

The proposed architecture of the curve-based cryptosystems is composed of the main controller, several Modular Arithmetic Logic Units (MALUs) and the coprocessor memory that shares intermediate variables between the MALUs (*i.e.*

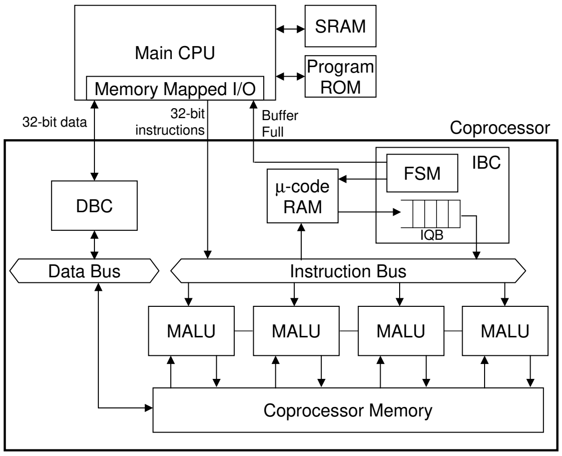

Fig. 2. Block Diagram for the system architecture with the curve-based coprocessor.

the so-called shared memory). The block diagram of the cryptosystem is illustrated in Fig. 2. The configuration of the coprocessor is flexible to provide from the smallest to the fastest implementation depending on a target application. Some components can be added or removed as will be explained next.

The main CPU communicates with the coprocessor through memory-mapped I/O (*e.g.* SRAM interface) and has three types of 32-bit in- and outputs; one of them is a signal that tells the controller to stop sending instructions when the instruction buffer is full. A 32-bit input/output passes data back and forward between the main CPU and the coprocessor and a 32-bit output is used to send instructions. The data transfer between the main CPU and the coprocessor is controlled by a Data Bus Controller (DBC). When using SRAM attached to the main CPU for storing intermediate variables for HECC/ECC operations, the coprocessor can be constructed without use of the coprocessor memory. Alternatively, for the purpose of reducing the I/O transfer overhead, the data memory can be embedded in the coprocessor. In this case, the path through the DBC is only activated when an initial point and the parameters of an elliptic curve are sent to the RAM, or when the result is retrieved.

Instructions are sent to the MALU either from the main CPU or from pre-set micro codes in the μ-code RAM. When the main CPU is in charge of dispatching instructions, the IBC block can be detached from the coprocessor. In this case, it occurs that the throughput of issuing instructions is not high enough for the MALU(s) to be utilized effectively. On the contrary, when the μ-code RAM is used for assisting the main CPU, the Instruction Bus Controller (IBC) can handle one instruction per cycle. For instance, the sequence of point doubling is stored in the μ-code RAM and the main CPU calls it as an instruction. Thus multiple MALUs can be activated in parallel without any instruction stalls. During point multiplication, the IBC keeps on reading instructions from the μ-code RAM and stores them to an Instruction Queue Buffer (IQB) unless the

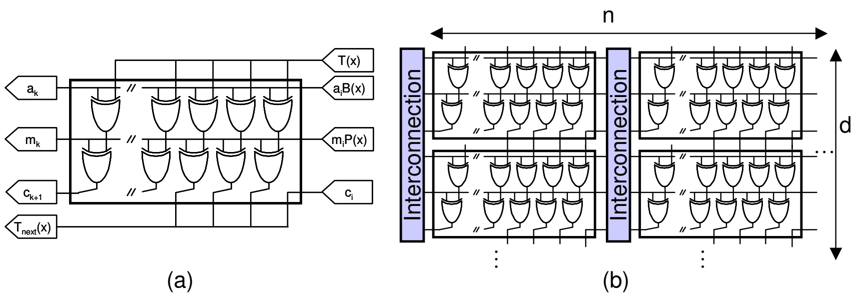

Fig. 3. Reconfigurable datapath for GF(2^n) operation. (a) MSB-first bit-serial polynomial-basis multiplier. (b) Scalability of the MALU.

IQB is full. The IBC checks if there is instruction-level parallelism (ILP) by checking the data-dependency of instructions in the IQB and forwards them to the MALU(s) (see Section 4.2 and 4.4).

4.2 Modular Arithmetic Logic Unit

In this section the architecture for the MALU is briefly explained. The datapath of the MALU is an MSB-first bit-serial polynomial-basis GF(2^n) multiplier as illustrated in Fig. 3(a). This is a hardware implementation that computes $A(x)B(x) + C(x) \bmod P(x)$ where $A(x) = \sum a_i x^i$, $B(x) = \sum b_i x^i$, $C(x) = \sum c_i x^i$ and $P(x) = \sum p_i x^i$. The proposed MALU computes $A(x)B(x) + C(x) \bmod P(x)$ by following the steps: The MALU sums up three types of inputs which are $a_iB(x), m_iP(x)$ and $T(x)$, and then outputs the intermediate result, $T_{next}(x)$ by computing $T_{next}(x) = (T(x) + a_iB(x) + m_iP(x))x + c_{i-1}$ where $m_i = t_n \oplus a_ib_n$. By providing T_{next} as the next input T and repeating the same computation for n times, one can obtain the result. The detailed explanation is also discussed in [27]. Moreover, by providing $B(x) + D(x)$ in place of $B(x)$, an operation, $A(x)(B(x) + D(x)) + C(x) \bmod P(x)$ can be also supported. This operation requires additional XORs and selector logics for registers storing the coefficients of $B(x)$ or $(B(x) + D(x))$.

The proposed datapath is scalable in the digit size d (in vertical direction in Fig. 3(b)) which can be decided by exploring the best combination of performance and cost. The field size n is determined by the key-length. It can be achieved also by interconnecting several MALUs in horizontal direction. Hence, various implementation options can be chosen with the MALU. For instance, the coprocessor can support arbitrary field sizes up to 335 when using four sets of the MALU whose field size is 83.

4.3 The MALU Instruction

Here, a new instruction called MALU_n is defined. It is worth mentioning that this is the only instruction that operates on the datapath.

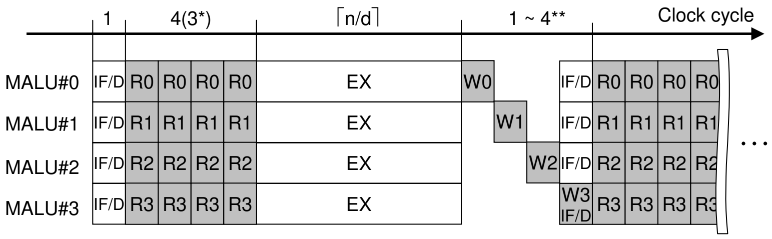

Fig. 4. Example of four parallel issue of instructions in case of allocating four MALUs. (IF/D: Instruction Fetch/Decode, EX: Execution of MALU, R/W: Read/Write from/to the coprocessor memory). *The read cycle differs from the type of operation. **The write cycle depends on the number of instructions issued in parallel.

$$\mathrm{MALU}_n(A, B, C, D) = A(x)(B(x) + D(x)) + C(x) \bmod P(x). \tag{2}$$

When using $A(x)B(x) + C(x) \bmod P(x)$ operation, one can ignore $D(x)$ as $D(x) = 0$. The whole procedure to execute MALU_n starts from an instruction fetch and decode (IF/D). Then, variables for $A(x), B(x), C(x)$ and $D(x)$ are loaded via RAM (R) for the succeeding execution stage. The result is stored to RAM (W) in the last step. Note that the data at different addresses can be read in parallel for the different MALU by replicating RAM (*i.e.* four clones of single-port RAMs in case of using four MALUs). The write cycle is determined by the number of instructions that can be issued in parallel. When using multiple MALUs, the write operations from every MALU are done at the different cycle to escape memory-write conflicts. This is illustrated in Fig. 4.

4.4 Dynamic Scheduling

ILP is exploited for all instructions as long as two or more instructions are buffered in the IQB. Here, we introduce our strategy to find ILP. A MALU_n instruction has four source operands and outputs the result to RAM, *i.e.* MALU_n deals with five types of addresses in the case of operating $A(x)(B(x) + D(x)) + C(x) \bmod P(x)$. Here, let A, B, C, D be the addresses for four inputs and R be the address where the result is stored. They are expressed as follows:

$$\mathrm{MALU}_n : R = A, B, C, D. \tag{3}$$

The MALU_n also refers to $P(x)$ that is stored in RAM. Including out-of-order execution, the following two types of dependencies are possible between two instructions, MALU_n^i and MALU_n^j (i and j are labels indicating order of instruction in the IQB). By checking the following two dependencies for all i and j that satisfy $i < j < ILP_D$, where ILP_D is the size of the instruction window, one can determine the number of instructions to be issued in parallel.

Table 1. Primary instructions for the coprocessor.

INSTRUCTION	DESCRIPTION	OPERATION
`STORE(@dst)`	Data storing to the coprocessor	`R@dst <= din;`
`LOAD(@src)`	Data loading from the coprocessor	`dout <= R@src;`
`MALU(@dst,@src1-4)`	Operate MALU_n	`R@dst <= MALU(R@src1-4)`
`HECCPD()`	HECC divisor doubling	`P <= 2P`

Read-After-Write (RAW) Dependency check for in-order execution ($R^i = A^j$, $R^i = B^j$, $R^i = C^j$, $R^i = D^j$): If the result of the instruction MALU_n^i, R^i is input for the following instructions, the instruction MALU_n^i cannot be issued until the preceding instruction completes the operation.

RAW Dependency check for out-of-order execution ($R^j = A^i$, $R^j = B^i$, $R^j = C^i$, $R^j = D^i$): In case that all conditions are not true, the instruction MALU_n^j cannot be issued until the instruction MALU_n^i finishes. The example using the actual sequence of EC point doubling is shown in the Appendix.

The proposed architecture needs no check for Write-After-Read and Write-After-Write dependencies contrary a general superscalar machine. This is because MALU_n is a fixed-length multi-cycle instruction and hence we can skip those dependencies in the sequence of point/divisor operations. Suppose the size of the instruction window is ILP_D, the number of conditions to check becomes $4(ILP_D - 1)^2$. The hardware complexity for ILP expands with a large ILP_D, but instead further parallelism can be expected.

5 Implementation

5.1 Instruction Sets for the Coprocessor

Table 1 shows some of the primary instructions for the co-processor. The input registers of the MALU are set via data-bus ports. In case of using a 32-bit CPU such as the ARM, setting a register whose address is `src1` requires three `STORE(@dst)` instructions for HECC over $GF(2^{83})$. After all operands are set in corresponding registers, a `MALU(@dst,@src1-4)` operation is executed. When using the μ-code configuration, it is possible to define an instruction that consists of a series of `MALU(@dst,@src1-4)` operations. In this paper, point/divisor operations are all composed of the MALU instruction (see the Appendix).

5.2 System Configurations

The system configurations are explored in two steps. First, in order to make the best use of the superscalar coprocessor, four different coprocessor configurations are explored as listed in Fig. 5(a). This is the so-called vertical exploration of the hardware/software co-design. Secondly, the performance comparison is made with HECC, ECC and ECC over a composite field by changing the number of MALUs. Thus the coprocessor is also investigated from a parallel processing point of view (horizontal exploration).

(a)

	# of MALUs	μ-code RAM	Copro. Mem.
TYPE I	1		
TYPE II	1	X	
TYPE III	1		X
TYPE IV	1	X	X

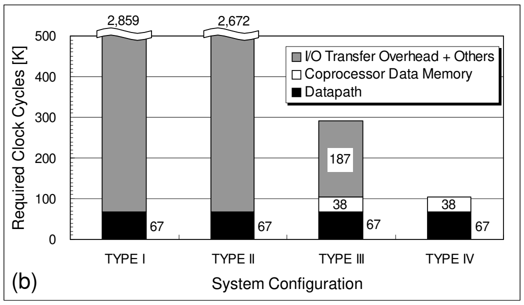

Fig. 5. (a) Coprocessor configurations for the vertical exploration. (b) Required clock cycles of HECC scalar multiplication for different coprocessor configuration ($d = 12$).

5.3 Design Environment

The proposed design is constructed on GEZEL hardware/software co-design environment with the ARM Instruction Set Simulator (ISS) [28].

The platform provides cycle-accurate simulations for various hardware/software system configurations. As mentioned in Section 4, the coprocessor is attached to the memory-mapped interface of the ARM. Thus, various types of system configurations are examined to verify the functionality and estimate the performance in a system-level. The GEZEL codes are automatically translated into VHDL codes that can be used for an FPGA prototype.

6 Results

6.1 Vertical Exploration of System Architecture with Coprocessor

Fig. 5(b) compares the performance of HECC scalar multiplication for different system configurations. For the case of the TYPE I and II, the I/O transfer overhead between the main CPU and the coprocessor is the majority of the cycles (about 97%). The reason for this is that the temporary data variables are stored in the memory of the main CPU and travel through the CPU to the coprocessor for processing. As for the TYPE III, the I/O transfer overhead is reduced significantly due to the effect of the data memory allocated in the coprocessor. However, the I/O overhead is still dominant because the main CPU issues instructions via the slow communication channel. The parallel processing feature is hence useless to improve the performance in such system settings. Note that the ratio of the I/O transfer overheads is reduced ostensibly by introducing smaller d since the datapath performs in more clock cycles. In this way, it is important to find the best digit size, d that can hide the I/O transfer overhead with the TYPE III. This paper, however, focuses on the TYPE IV for a deeper investigation of the parallelism in order to obtain high performance. Because the TYPE IV assures the highest parallelism regardless of the value of d.

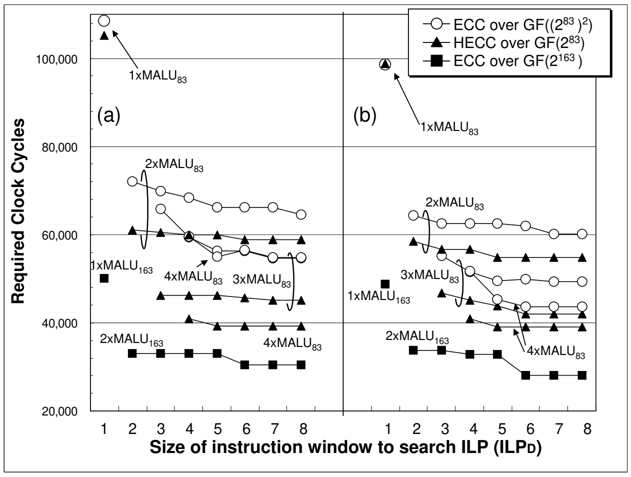

Fig. 6. Required clock cycles of scalar multiplication for different ILP_D ($d = 12$). (a) Operation form is $AB + C$. (b) Operation form is $A(B + D) + C$.

6.2 Performance Comparison between Three Cryptosystems

Fig. 6 shows the required cycles for various implementations based on the TYPE IV configuration. The building block of the datapath is the MALU whose field size is 83 or MALU_{83}. Up to four clones of the MALU_{83} are embedded in the coprocessor to observe the performance improvement with the superscalar architecture. For ECC, a pair of MALU_{83} is equivalent to one MALU_{163} in terms of hardware cost. The overall performance improves as increasing the number of MALU_{83} for both of the operation type. Also a large ILP_D helps exploiting more parallelism and leads to a higher performance. The results show the effectiveness of an operation whose form is $A(B+D)+C$ especially for the ECC over a composite field. In our case, the performance of ECC is better than others on equivalent hardware resources. The results are also summarized in Table 2.

Table 2. Required clock cycles of scalar multiplication for $d = 12$ and $ILP_D = 6$. Figures in parenthesis are the speed-up ratio based on the smallest configuration.

Coprocessor Configuration	Operation: $AB + C$			$A(B + D) + C$		
	HECC $GF(2^{83})$	ECC $GF(2^{163})$	ECC $GF((2^{83})^2)$	HECC $GF(2^{83})$	ECC $GF(2^{163})$	ECC $GF((2^{83})^2)$
$1\times\text{MALU}_{83}$	105,237 (1.00)	-	108,603 (1.00)	98,856 (1.06)	-	98,688 (1.10)
$2\times\text{MALU}_{83}$ $=1\times\text{MALU}_{163}$	58,917 (1.79)	50,112 (1.00)	66,193 (1.64)	54,909 (1.92)	48,849 (1.03)	61,941 (1.75)
$3\times\text{MALU}_{83}$	45,606 (2.31)	-	56,267 (1.93)	42,029 (2.50)	-	49,849 (2.18)
$4\times\text{MALU}_{83}$ $=2\times\text{MALU}_{163}$	39,247 (2.68)	30,396 (1.65)	56,437 (1.92)	39,115 (2.69)	27,981 (1.79)	43,594 (2.49)

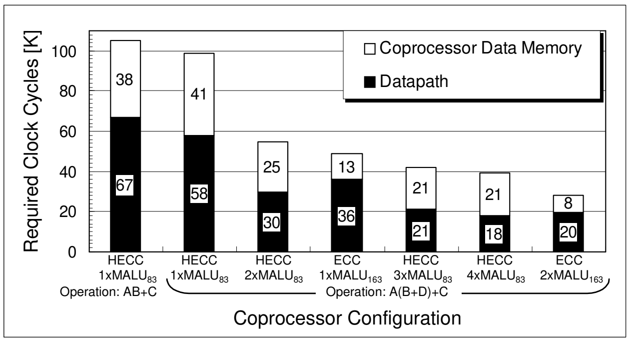

Fig. 7. The profile graphs of the required clock cycles in ECC/HECC scalar multiplication for different hardware settings of the coprocessor ($d = 12$).

In order to investigate the performance bottle-neck of HECC and ECC, the required clock cycles in scalar multiplication is split into two factors; one is for the memory access and another is for the data processing of the datapath. As can be seen from the Fig. 7, operation form, $A(B + D) + C$ introduces more memory accesses while the data can be processed in less clock cycles. Overall the proposed superscalar feature can reduce the clock cycles in both of the coprocessor memory access and the datapath operation. The memory accesses of HECC become dominant as introducing more parallelism. On the other hand the memory accesses in ECC is less than 30 % of the total clock cycles. This fact explains the reason that scalar multiplication of HECC is eventually slower than that of ECC on equivalent hardware resources.

6.3 Prototype Results on FPGA

Based on the performance observation, the coprocessor is prototyped with the system configuration of $d = 12$ and $ILP_D = 6$ on Virtex-II PRO (XC2VP30). The operation that the MALU supports is $A(B + D) + C$. The the coprocessor memory consist of several 32×84-bit single-port RAMs and each RAM is assigned to each MALU_{83}. The μ-code program is implemented as an LUT ROM. As shown in Table 3, our HECC results show a better trade-off between cost and performance than the previous work. With regard to ECC implementation, our result is based on the IEEE-P1363 compliant sequence [23] and is not as fast as some previous work [13, 29]. However considering the flexibility in our proposed coprocessor, the difference can be regarded as small.

7 Conclusions

This paper introduced a superscalar coprocessor that could deal with three different curve-based cryptosystems. The implementation results showed that scalar multiplication of ECC over GF(2^{163}), HECC of genus 2 over GF(2^{83}) and ECC

Table 3. Performance Comparison of HECC/ECC implementations on FPGAs

Ref. Design	Field	Target Platform	Area [$slices/gates$]	f_{max} [MHz]	Perform. [μsec]	Polynomial $P(x)$	Comments
HECC							
This work	GF(2^{83})	Virtex-II Pro	2,446 4,749 6,586	100.0	989 549 420	Arbitrary	$1\times$MALU$_{83}$ $2\times$MALU$_{83}$ $3\times$MALU$_{83}$
[11]	GF(2^{81})	Virtex-II Pro	4,039 7,737	57.0 60.7	787 387	Fixed	$2\times$MULT,$1\times$INV $3\times$MULT,$2\times$INV
ECC							
This work	GF(2^{163})	Virtex-II Pro	4,749 8,450	100.0	488 280	Arbitrary	$1\times$MALU$_{163}$ $2\times$MALU$_{163}$
[14]	GF(2^{163})	Virtex E	19,508	66.5	1,554 143	Arbitrary Fixed: $x^{163}+x^7$ $+x^6+x^3+1$	López-Dahab scalar mult.
[13]	GF(2^{167})	Virtex E	3,002 (+ 10 BRAMs)	76.7	210	Fixed: $x^{167}+x^6+1$	López-Dahab scalar mult.
[29]	GF(2^{191})	Virtex E	19,626 (+ 26 BRAMs)	9.99	59.26	Fixed: $x^{191}+x^9+1$	López-Dahab scalar mult.

over a composite field, GF$((2^{83})^2)$ was improved by a factor of 1.8, 2.7 and 2.5 respectively compared to the case of a basic single-scalar architecture. This speed-up was achieved by vertical and horizontal exploration of the system architecture to exploit parallelism in curve-based cryptography. In our design, ECC showed better performance than others on the same amount of hardware resource. All operations in three curve-based cryptosystems were performed with only one instruction that could be flexibly defined as $AB+C$ or $A(B+D)+C$.

Acknowledgement

The IBBT - QoE project is co-funded by the IBBT (Interdisciplinary Institute for BroadBand Technology), a research institute founded by the Flemish Government in 2004, and the involved companies and institutions [30].

References

1. W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.
2. R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
3. N. Koblitz. Elliptic curve cryptosystem. *Math. Comp.*, 48:203–209, 1987.
4. V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology: Proceedings of CRYPTO'85*, number 218 in LNCS, pages 417–426. Springer-Verlag, 1985.
5. N. Thériault. Index calculus attack for hyperelliptic curves of small genus. In C. S. Laih, editor, *Proceedings of Advances in Cryptology - ASIACRYPT: 9th International Conference on the Theory and Application of Cryptology and Information Security*, number 2894 in LNCS, pages 75–92. Springer-Verlag, 2003.

6. P. Montgomery. Speeding the pollard and elliptic curve methods of factorization.
7. N.P. Smart. The Hessian form of an elliptic curve. In Ç.K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of 3rd International Workshop on Cryptograpic Hardware and Embedded Systems (CHES)*, number 2162 in LNCS, pages 121–128. Springer-Verlag, 2001.
8. M. Joye and S.-M. Yen. The Montgomery powering ladder. In B.S. Kaliski Jr., Ç.K. Koç, and C. Paar, editors, *Proceedings of 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2523 in LNCS, pages 291–302. Springer-Verlag, 2002.
9. T. Izu and T. Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In D. Naccache and P. Paillier, editors, *Proceedings of 5th International Workshop on Practice and Theory in Public Key Cryptosystems (PKC 2002)*, number 3027 in LNCS, pages 280–296. Springer-Verlag, 2002.
10. P. K. Mishra and P. Sarkar. Parallelizing explicit formula for arithmetic in the jacobian of hyperelliptic curves. In J. Hartmanis G. Goos and J. van Leeuwen, editors, *Proceedings of ASIACRYPT 2003*, number 2894 in LNCS, pages 93–110. Springer-Verlag, 2003.
11. T. Wollinger. *Software and Hardware Implementation of Hyperelliptic Curve Cryptosystems.* PhD thesis, Ruhr-University Bochum, 2004.
12. A. Hodjat, L. Batina, D. Hwang, and I. Verbauwhede. A hyperelliptic curve crypto coprocessor for an 8051 microcontroller. In *Proceedings of The IEEE 2005 Workshop on Signal Processing Systems (SIPS'05)*, pages 93–98, 2005.
13. G. Orlando and C. Paar. A high-performance reconfigurable elliptic curve processor for GF(2^m). In Ç.K. Koç and C. Paar, editors, *Proceedings of 2nd International Workshop on Cryptograpic Hardware and Embedded Systems (CHES)*, number 1965 in LNCS, pages 41–56. Springer-Verlag, 2000.
14. N. Gura, S.C. Shantz, H. Eberle, D. Finchelstein, S. Gupta, V. Gupta, and D. Stebila. An end-to-end systems approach to elliptic curve cryptography. In B. Kaliski Jr., Ç.K. Koç, and C. Paar, editors, *Proceedings of 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2523, 2002.
15. T. Lange. Formulae for arithmetic on genus 2 hyperelliptic curves. *Applicable Algebra in Engineering, Communication and Computing*, 15(5):295–328, 2005.
16. B. Byramjee and S. Duquesne. Classification of genus 2 curves over F_{2^n} and optimization of their arithmetic. Cryptology ePrint Archive: Report 2004/107.
17. T. Lange and M. Stevens. Efficient doubling on genus two curves over binary fields. In H. Handschuh and M.A. Hasan, editors, *In Selected Areas in Cryptography: SAC 2004*, volume 3357 of *LNCS*, pages 170–181. Springer-Verlag, 2004.
18. G. Elias, A. Miri, and T. H. Yeap. High-performance, FPGA based hyperelliptic curve cryptosystem. In *In Proceedings of the 22nd Biennial Symposium on Communications*, 2004.
19. J. Pelzl, T. Wollinger, J. Guajardo, and C. Paar. Hyperelliptic curve cryptosystems: Closing the performance gap to elliptic curves. In C. Walter, Ç.K. Koç, and C. Paar, editors, *Proceedings of 5th International Workshop on Cryptograpic Hardware and Embedded Systems (CHES)*, number 2779 in LNCS, pages 351–365. Springer-Verlag, 2003.
20. G.B. Agnew, R.C. Mullin, and S.A. Vanstone. A fast elliptic curve cryptosystem. In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology: Proceedings of EUROCRYPT'89*, number 434 in LNCS, pages 706–708. Springer-Verlag, 1989.

21. N. Boston, T. Clancy, Y. Liow, and J. Webster. Genus two hyperelliptic curve coprocessor. In B.S. Kaliski Jr., Ç.K. Koç, and C. Paar, editors, *Proceedings of 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2523 in LNCS, pages 400–414. Springer-Verlag, 2002.
22. N. Koblitz. *Algebraic Aspects of Cryptography.* Springer-Verlag, first edition, 1998.
23. I. Blake, G. Seroussi, and N.P. Smart. *Elliptic Curves in Cryptography.* London Mathematical Society Lecture Note Series. Cambridge University Press, 1999.
24. A. Menezes, Y.-H. Wu, and R. Zuccherato. *An Elementary Introduction to Hyperelliptic Curves - Appendix*, pages 155–178. Springer-Verlag, 1998. N. Koblitz: Algebraic Aspects of Cryptography.
25. T. Itoh and S. Tsujii. Effective recursive algorithm for computing multiplicative inverses in GF(2^m). *Electronics Letters*, 24(6):334–335, 1988.
26. R. Lidl and H. Niederreiter. *Finite fields*, volume 20 of *Encyclopedia of Mathematics and its Applications.* Cambridge University Press, second edition, 2000.
27. K. Sakiyama, B. Preneel, and I. Verbauwhede. A fast dual-field modular arithmetic logic unit and its hardware imlementation. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'06)*, pages 787–790, 2006.
28. P. Schaumont. Gezel version 2. http://rijndael.ece.vt.edu/gezel2/.
29. Nazar A. Saqib, Francisco Rodríguez-Henriquez, and Arturo Díaz-Pérez. A reconfigurable processor for high speed point multiplication in elliptic curves. In *International Journal of Embedded Systems 2005*, volume 1, No. 3/4, pages 237 – 249, 2005.
30. https://projects.ibbt.be/qoe/.

A Dynamic Scheduling for EC point doubling

The first two instructions have a RAW dependency with t_1. ECDB04 has no RAW dependency upon the first three instructions in in-order and out-of-order execution, and therefore it can be issued prior to the first three instructions.

Table 4. Example of parallelized out-of-order instruction sequence for EC point doubling in case of three consecutive point doublings (*i.e.* $P \Leftarrow 2^3P$, where $P(X_1, Y_1, Z_1)$). The ECDBs in italic are instructions from preceding and succeeding point doublings.

Original Sequence						Parallelized Out-of-order Sequence
Address:	**R**	**A**	**B**	**C**	**D**	
ECDB01: $\text{MALU}_n($	t_1,	X_1,	X_1,	0,	$0)$	*ECDB08* & ECDB04
ECDB02: $\text{MALU}_n($	t_2,	t_1,	t_1,	0,	$0)$	*ECDB09* & ECDB06
ECDB03: $\text{MALU}_n($	t_4,	Y_1,	Z_1,	t_1,	$0)$	*ECDB10* & ECDB01
ECDB04: $\text{MALU}_n($	t_3,	Z_1,	Z_1,	0,	$0)$	ECDB02 & ECDB03
ECDB05: $\text{MALU}_n($	Z_1,	X_1,	t_3,	0,	$0)$	ECDB05 & ECDB07
ECDB06: $\text{MALU}_n($	t_5,	d_6,	t_3,	X_1,	$0)$	ECDB08 & *ECDB04*
ECDB07: $\text{MALU}_n($	t_3,	t_5,	t_5,	0,	$0)$	ECDB09 & *ECDB06*
ECDB08: $\text{MALU}_n($	X_1,	t_3,	t_3,	0,	$0)$	ECDB10 & *ECDB01*
ECDB09: $\text{MALU}_n($	t_1,	X_1,	Z_1,	0,	$t_4)$	*ECDB02* & *ECDB03*
ECDB10: $\text{MALU}_n($	Y_1,	t_2,	Z_1,	t_1,	$0)$	*ECDB05* & *ECDB07*