A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over $\mathbb{GF}(2^n)$

M. Ernst[1], M. Jung[1], F. Madlener[1], S. Huss[1], and R. Blümel[2]

[1] Integrated Circuits and Systems Lab., Computer Science Department,
Darmstadt University of Technology, Germany
{ernst | mjung | madlener | huss}@iss.tu-darmstadt.de

[2] cv cryptovision GmbH, Gelsenkirchen, Germany
rainer.bluemel@cryptovision.com

Abstract. The performance of elliptic curve based public key cryptosystems is mainly appointed by the efficiency of the underlying finite field arithmetic. This work describes two generic and scalable architectures of finite field coprocessors, which are implemented within the latest family of Field Programmable System Level Integrated Circuits FPSLIC from Atmel, Inc. The HW architectures are adapted from Karatsuba's divide and conquer algorithm and allow for a reasonable speedup of the top-level elliptic curve algorithms. The VHDL hardware models are automatically generated based on an eligible operand size, which permits the optimal utilization of a particular FPSLIC device.

Keywords. Elliptic Curve cryptography, $\mathbb{GF}(2^n)$ arithmetic, Karatsuba multiplication, VHDL model generator, coprocessor synthesis, FPGA hardware acceleration, Atmel FPSLIC platform.

1 Introduction

Today there is a wide range of distributed systems, which use communication resources that can not be safeguarded against eavesdropping or unauthorized data alteration. Thus cryptographic protocols are applied to these systems in order to prevent information extraction or to detect data manipulation by unauthorized parties. Besides the widely-used RSA method [1], public-key schemes based on elliptic curves (EC) have gained more and more importance in this context. In 1985 elliptic curve cryptography (ECC) has been first proposed by V. Miller [2] and N. Koblitz [3]. In the following a lot of research has been done and nowadays ECC is widely known and accepted. Because EC methods in general are believed to give a higher security per key bit in comparison to RSA, one can work with shorter keys in order to achieve the same level of security [4]. The smaller key size permits more cost-efficient implementations, which is of special interest for low-cost and high-volume systems. Because ECC scales well over the whole security spectrum, especially low-security applications can benefit from ECC.

Each application has different demands on the utilized cryptosystem (e.g., in terms of required bandwidth, level of security, incurred cost per node and number of communicating partners). The major market share probably is occupied by the low-bandwidth, low-cost and high-volume applications, most of which are based on SmartCards or similar low complexity systems. Examples are given by the mobile phone SIM cards, electronic payment and access control systems. In case of access control systems, ECC allows to use *one device and one key-pair per person* for the entire application. A very fine granular control is possible and in contrast to present systems, which are mostly based on symmetric ciphers, there is no problem regarding the key handling.

Depending on the application, the performance of genuine SW implementations of ECC is not sufficient. In this paper two generic and scalable architectures of *Finite Field* coprocessors for the acceleration of ECC are presented. The first one, which is mainly composed of a single combinational Karatsuba multiplier (CKM), allows for a significant speed-up of the finite field multiplication while spending only a small amount of HW resources. The second one is a finite field coprocessor (FFCP), implementing field multiplication, addition and squaring completely within HW. The proposed multi-segment Karatsuba multiplication together with a cleverly selected sequence of intermediate result computations permits high-speed ECC even on devices offering only approx. 40K system gates of HW resources. A variety of fast EC cryptosystems can be built by disposing the proposed system partitioning. Running the EC level algorithms in SW allows for algorithmic flexibility while the HW accelerated finite field arithmetic contributes the required performance.

Recently, Atmel, Inc. introduced their new AT94K family of FPSLIC devices (Field Programmable System Level Integrated Circuits). This architecture integrates FPGA resources, an AVR microcontroller core, several peripherals and SRAM within a single chip. Based on HW/SW co-design methodologies, this architecture is perfectly suited for System on Chip (SoC) implementations of ECC.

The mathematical background of elliptic curves and finite fields is briefly described in the following section. In Sec. 3 the architectures of the proposed finite field coprocessors are detailed. Sec. 4 introduces the FPSLIC hardware platform. Finally, we report on our implementation results give some performance numbers and conclusions.

2 Mathematical Background

There are several cryptographic schemes based on elliptic curves, which work on a subgroup of points of an EC over a finite field. Arbitrary finite fields are approved to be suitable for ECC. In this paper we will concentrate on elliptic curves over the finite field $\mathbb{GF}(2^n)$[1] and their arithmetics only. For further information we refer to [5] and [6].

2.1 Elliptic Curve Arithmetic

An elliptic curve over $\mathbb{GF}(2^n)$ is defined as the cubic equation

$$E: \quad y^2 + xy = x^3 + ax^2 + b \tag{1}$$

with $a, b, x, y \in \mathbb{GF}(2^n)$ and $b \neq 0$. The set of solutions $\{(x, y) \mid y^2 + xy = x^3 + ax^2 + b\}$ is called the points of the elliptic curve E. By defining an appropriate addition operation and an extra point $\mathcal{O}$, called the point at infinity, these points become an additive, abelian group with $\mathcal{O}$ the neutral element. The EC point multiplication is computed by repeated point additions such as

$$\underbrace{P + P + \ \dots\ + P + P}_{k \text{ times}} = k \cdot P = R$$

[1] In the context of cryptographic applications n should be prime, in order to be safeguarded against *Weil decent* attacks [7].

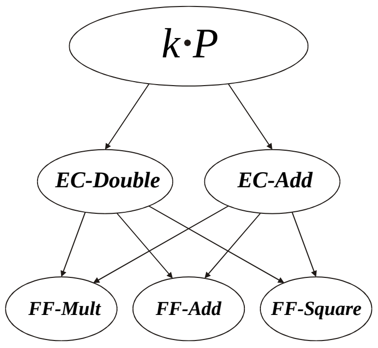

Fig. 1. EC arithmetic hierarchy

with $k \in \mathbb{N}$ and $P, R \in E$.

The hierarchy of arithmetics for EC point multiplication is depicted in Fig. 1. The top level $k{\cdot}P$ algorithm is performed by repeated *EC-Add* and *EC-Double* operations. The EC operations in turn are composed of basic operations in the underlying finite field (FF). The proposed FF coprocessor (see Sec. 3.2 and Sec. 5.2) is capable to compute the *FF-Add* and *FF-Square* operations within one clock cycle. The operation *FF-Mult* is more costly. The number of clock cycles for its computation depends on the particular architecture of the FF multiplier. Compared to *FF-Add*, *FF-Square* and *FF-Mult* the FF inversion is a very expensive operation and is therefore not implemented in the coprocessor as a basic operation. In order to avoid computing inverses, projective coordinates are used during the computation of the EC operations. By exploiting a projective coordinate representation only one FF inversion is required to perform a complete EC point multiplication. This single FF inversion has to be done at the end of the $k{\cdot}P$ algorithm for the conversion back to affine coordinates.

2.2 Finite Field Arithmetic

As previously mentioned, the EC arithmetic is based on a FF of characteristic 2 and extension degree n: $\mathbb{GF}(2^n)$, which can be viewed as a vector space of dimension n over the field $\mathbb{GF}(2)$. There are several bases known for $\mathbb{GF}(2^n)$. The most common bases, which are also permitted by the leading standards concerning ECC (IEEE 1363 [8] and ANSI X9.62 [9]) are *polynomial bases* and *normal bases*.

The representation treated in this paper is a polynomial basis, where field elements are represented by binary polynomials modulo an irreducible binary polynomial (called reduction polynomial) of degree n. Given an irreducible polynomial $P(x) = x^n + \sum_{i=0}^{n-1} p_i x^i$, with $p_i \in \mathbb{GF}(2)$; an element $A \in \mathbb{GF}(2^n)$ is represented by a bit string $(a_{n-1}, \ldots, a_2, a_1, a_0)$, so that

$$A(x) = \sum_{i=0}^{n-1} a_i x^i = a_{n-1}x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

is a polynomial in x of degree less than n with coefficients $a_i \in \mathbb{GF}(2)$.

By exploiting a field of characteristic 2, the addition is reduced to just XOR-ing the corresponding bits. The sum of two elements $A, B \in \mathbb{GF}(2^n)$ is given by

$$C(x) = A(x) \oplus B(x) = \sum_{i=0}^{n-1} (a_i \oplus b_i) x^i \tag{2}$$

and therefore takes a total of n binary XOR operations.

The multiplication of two elements $A, B \in \mathbb{GF}(2^n)$ is equivalent to the product of the corresponding polynomials:

$$C(x) = A(x) \cdot B(x) = \sum_{i=0}^{2n-2} c_i x^i \text{ denoting } c_k = \sum_{i=0}^{k} a_i b_{k-i} \text{ for } 0 \le k \le 2n-2, \tag{3}$$

with $a_i = 0$ and $b_i = 0$ for $i \ge n$. At the bit level the multiplication in $\mathbb{GF}(2)$ is performed with boolean AND operation.

Squaring is a special case of multiplication. For $A \in \mathbb{GF}(2^n)$ the square is given by:

$$A^2(x) = \sum_{i=0}^{n-1} a_i x^{2i}. \tag{4}$$

In the case of multiplication and squaring a polynomial reduction step has to be performed, which is detailed in Sec. 2.2.

Karatsuba Multiplication. In 1963 A. Karatsuba and Y. Ofman discovered that multiplication of two n bit numbers can be done with a bit complexity of less than $O(n^2)$ using an algorithm now known as Karatsuba multiplication [10]. For multiplication in $\mathbb{GF}(2^n)$ the Karatsuba multiplication scheme can be applied as well. Therefore, a polynomial $A \in \mathbb{GF}(2^n)$ is subdivided into two segments and expressed as

$$A = A_1 x^{n/2} \oplus A_0 .$$

For polynomials $A, B \in \mathbb{GF}(2^n)$ the n-bit multiplication $C = A \cdot B$ is subdivided into $n/2$-bit multiplications as follows:

$$\begin{aligned} C &= A \cdot B \\ &= (A_1 x^{n/2} \oplus A_0) \cdot (B_1 x^{n/2} \oplus B_0) \\ &= A_1 \cdot B_1 x^n \oplus (A_1 \cdot B_0 \oplus A_0 \cdot B_1) x^{n/2} \oplus A_0 \cdot B_0 . \end{aligned}$$

By defining some additional polynomials

$$\begin{aligned} T_1 &= A_1 \cdot B_1 \\ T_2 &= (A_1 \oplus A_0) \cdot (B_1 \oplus B_0) = A_1 B_0 \oplus A_0 B_1 \oplus A_1 B_1 \oplus A_0 B_0 \\ T_3 &= A_0 \cdot B_0 \end{aligned}$$

one gets

$$A \cdot B = T_1 x^n \oplus (T_2 \ominus T_1 \ominus T_3) x^{n/2} \oplus T_3$$

and since $\ominus$ and $\oplus$ are equal in $\mathbb{GF}(2^n)$

$$A \cdot B = T_1 x^n \oplus (T_2 \oplus T_1 \oplus T_3) x^{n/2} \oplus T_3 . \tag{5}$$

This results in a total of three $n/2$-bit multiplications and some extra additions (XOR operations) to perform one n-bit multiplication.

Multi-Segment Karatsuba Multiplication. The fundamental Karatsuba multiplication for polynomials in $\mathbb{GF}(2^n)$ is based on the idea of *divide and conquer*, since the operands are divided into two segments. One may attempt to generalize this idea by subdividing the operands into more than two segments. [11] reports on such an implementation with a fixed number of three segments denoted as *Karatsuba-variant multiplication*. The Multi-Segment Karatsuba (MSK) multiplication scheme, which is detailed subsequently, is more general because an arbitrary number of segments is supported. Disregarding some slight arithmetic variations, the Karatsuba-variant multiplication is a special case of the MSK approach.

Two polynomials in $\mathbb{GF}(2^n)$ are multiplied by a k-segment Karatsuba multiplication (MSK_k)[2] in the following way: It is assumed that $n \mod k = 0$; if not, the polynomials are padded with the necessary number of zero coefficients. A polynomial $A \in \mathbb{GF}(2^n)$ is divided into k segments such that $A = \bigoplus_{i=0}^{k-1} A_i \cdot \hat{x}^i$, with $\hat{x} = x^{n/k}$. With Eqn. 6 holds $C = A \cdot B = MSK_k(A, B)$ for any polynomials $A, B \in \mathbb{GF}(2^n)$:

$$MSK_k(A,B) = \left(\bigoplus_{i=1}^{k} S_{i,0}(A,B) \cdot \hat{x}^{i-1}\right) \oplus \left(\bigoplus_{i=1}^{k-1} S_{k-i,i}(A,B) \cdot \hat{x}^{i-1+k}\right) \quad (6)$$

with

$$S_{m,l}(A,B) = \left(\bigoplus_{i=1}^{m-1} S_{i,l}(A,B)\right) \oplus \left(\bigoplus_{i=1}^{m-1} S_{i,l+m-i}(A,B)\right) \oplus M_{m,l}(A,B)\,, \quad (7)$$

$$S_{1,l}(A,B) = M_{1,l}(A,B) \text{ and } M_{m,l}(A,B) = \left(\bigoplus_{i=l}^{l+m-1} A_i\right) \cdot \left(\bigoplus_{i=l}^{l+m-1} B_i\right)$$

According to Eqn. 6 the entire product $C = A \cdot B = MSK_k(A, B)$ is composed of the partial sums $S_{m,l}(A, B)$. Each partial sum consists of partial products $M_{m,l}(A, B)$ according to Eqn. 7. The total number $N(k)$ of required n/k-bit multiplications in order to perform one n-bit multiplication using the MSK_k scheme is given by

$$N(k) = \sum_{i=1}^{k} i = \frac{(k+1) \cdot k}{2}\,. \quad (8)$$

The application of the above equations for a MSK_3 multiplication, made up of six $n/3$-bit multiplications, is illustrated in the appendix of this paper.

Polynomial Reduction. For $A, B \in \mathbb{GF}(2^n)$, the maximum degree of the resulting polynomial $C(x) = A(x) \cdot B(x)$ is $2n - 2$. Therefore, in order to fit into a bit string of size n, $C(x)$ has to be reduced. The polynomial reduction process modulo $P(x)$ is based on the equivalence

$$x^n \equiv \sum_{i=0}^{n-1} p_i x^i \mod P(x). \quad (9)$$

[2] A k-segment Karatsuba multiplication is subsequently termed as MSK_k.

Implementations of the reduction can especially benefit from hard-coded reduction polynomials with low Hamming weight such as trinomials, which are typically used in cryptographic applications.
Given such a trinomial as prime polynomial $P(x) = x^n + x^b + 1$ the reduction process can be performed efficiently by using the identities:

$$\begin{aligned} x^n &\equiv x^b + 1 \mod P(x) \\ x^{n+1} &\equiv x^{b+1} + x \mod P(x) \\ &\vdots \\ x^{2n} &\equiv x^{b+n} + x^n \mod P(x) \end{aligned}$$

This leads to

$$\begin{aligned} C(x) &= \sum_{i=0}^{2n-2} c_i x^i \\ &\equiv \sum_{i=0}^{n-1} c_i x^i + \sum_{i=n}^{2n-2} c_i (x^{b+i-n} + x^{i-n}) \mod P(x) \\ &= \sum_{i=0}^{n-1} c_i x^i + \sum_{i=0}^{n-1-b} c_{i+n} x^{b+i} + \sum_{i=n-b}^{n-1} c_{i+n} x^{b+i} + \sum_{i=0}^{n-1} c_{i+n} x^i \\ &\equiv \sum_{i=0}^{n-1} c_i x^i + \sum_{i=0}^{n-1-b} c_{n+i} x^{b+i} + \sum_{i=n-b}^{n-1} c_{n+i} (x^{2b+i-n} + x^{b+i-n}) + \sum_{i=0}^{n-1} c_{i+n} x^i \mod P(x) \\ &= \underbrace{\sum_{i=0}^{n-1} c_i x^i}_{(1)} + \underbrace{\sum_{i=0}^{n-1-b} c_{i+n} x^{b+i}}_{(2)} + \underbrace{\sum_{i=0}^{b-1} c_{2n-b+i} x^{b+i}}_{(3)} + \underbrace{\sum_{i=0}^{b-1} c_{2n-b+i} x^i}_{(4)} + \underbrace{\sum_{i=0}^{n-1} c_{n+i} x^i}_{(5)} \qquad (10) \end{aligned}$$

which results in a total of $2n + b$ binary XOR operations for one polynomial reduction. The particular terms (1...5) of the final equation are structured according to Fig. 2 in order to perform the reduction. With respect to the implementation in Sec. 3 a single n-bit register is sufficient to store the resulting bit string.

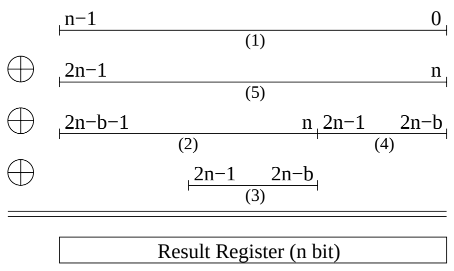

Fig. 2. Structure of the polynomial reduction

3 Hardware Architecture

An ideal HW/SW partitioning targeting a reconfigurable HW platform for an EC based cryptosystem depends on several parameters. As stated before, the FF arithmetic is the most time critical part of an EC cryptosystem. Depending on the utilized key size and the amount of available FPGA resources the FF operations can not inevitably be performed completely within HW. Therefore, flexibility within the HW design flow is essential, in order to achieve the maximum performance from a specific FPGA device. In order to ensure this flexibility, the HW design flow is based on the hardware description language VHDL, which is the *de-facto* standard for abstract modeling of digital circuits. A VHDL generator approach (similar to that one documented in [12]) was exploited to derive VHDL models for both of the subsequently described FF coprocessors. In Sec. 3.1 the combinational Karatsuba multiplier (CKM) is illustrated and Sec. 3.2 details the architecture of the entire finite field coprocessor (FFCP).

3.1 Combinational Karatsuba Multiplier (CKM)

As stated in Sec. 2.2 and shown in Fig. 3a the multiplication over $\mathbb{GF}(2)$ is computed by a single AND operation. According to Eqn. 5 the multiplication of two polynomials of degree m can be computed with three $m/2$-bit multiplications and some XOR operations to determine interim results and to accumulate the final result. This leads immediately to a recursive construction process, which builds CKMs of width $m = 2^i$ for arbitrary $i \in \mathbb{N}$ (see Fig. 3). With slight modifications this scheme can be generalized to support arbitrary bit widths. Exploiting the VHDL generator, CKM models for arbitrary $m \in \mathbb{N}$ can be automatically generated.

To determine the number of gates that constitute an m-bit CKM, we take a look at Fig. 4. In addition to the resources of the three $m/2$-bit multipliers, $2(m/2) = m$ 2-input XOR's are needed to compute the sub-terms $(A_1 \oplus A_0)$ and $(B_1 \oplus B_0)$ of T_2.

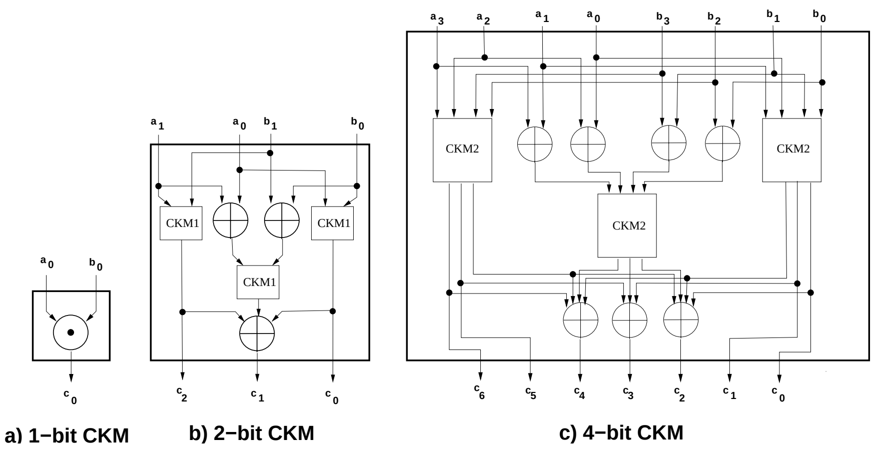

Fig. 3. Recursive construction process for CKM

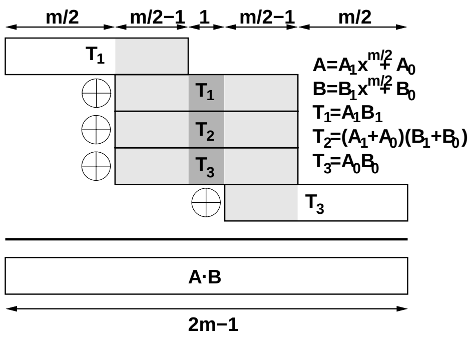

Fig. 4. Karatsuba multiplication

As can be seen from Fig. 4, $2(m/2-1)=(m-2)$ 4-input XOR's (light gray) and one 3-input XOR (dark gray) are in addition necessary to sum up the product. Thus, we can calculate the number of gates of an m-bit CKM with the following recurrences:

$$XOR_2(m)=\begin{cases}0 & m=1\\ m+3\cdot XOR_2(m/2) & m>1\end{cases}\quad XOR_3(m)=\begin{cases}0 & m=1\\ 1+3\cdot XOR_3(m/2) & m>1\end{cases}$$

$$AND_2(m)=\begin{cases}1 & m=1\\ 3\cdot AND_2(m/2) & m>1\end{cases}\quad XOR_4(m)=\begin{cases}0 & m=1\\ m-2+3\cdot XOR_4(m/2) & m>1\end{cases}$$

With the master method [13] it can easily be shown that all of these recurrences belong to the complexity class $\Theta(m^{\log_2 3})$. Explicit gate counts for CKM of various bit widths are summarized in the Tab. 1.

Table 1. CKM gate counts

Bit Width	1	2	4	8	16	32	64
AND_2	1	3	9	27	81	243	729
XOR_2	0	2	10	38	130	422	1330
XOR_3	0	1	4	13	40	121	364
XOR_4	0	0	2	12	50	180	602
SUM	1	6	25	90	301	966	3025

3.2 Finite Field Coprocessor (FFCP)

This section presents a generic and scalable FFCP architecture, which accelerates field multiplication, addition and squaring. Addition and squaring are operations, which require only a few logical resources and hence can be implemented by combinational logic. In contrast, the multiplication can not reasonably be implemented by combinational logic only. By the use of the proposed MSK multiplication scheme (see Sec. 2.2) and a cleverly selected sequence of intermediate result computations, the resulting datapath has only modest requirements on logic resources and at the same time a low cycle count for a complete field multiplication.

The datapath is build around a low complexity m-bit CKM as detailed in Sec. 3.1, but of course any other multiplier design would also do. By application of the sequential MSK_k multiplication algorithm, $k{\cdot}m$ bit wide operands can be processed. With respect to the implementation in Sec. 5.2 and for reasons of easy illustration we assume $k = 5$ in the following, but the scheme applies and scales in a nice way for arbitrary $k > 1$.

Eqn. 6 evaluated for $k = 5$ (MSK_5) is illustrated in Fig. 5a. Each rectangle denotes the result of an m-bit multiplication. As one would expect, these products are as wide as two segments. The labels in the rectangles determine the indices of the segments, whose sums have been multiplied. E.g., the label "234" represents the term $(A_2 \oplus A_3 \oplus A_4) \cdot (B_2 \oplus B_3 \oplus B_4)$, which is denoted $M_{3,2}(A, B)$ in Eqn. 7. The horizontal position of a rectangle denotes the exponent i of the associated factor $\hat{x}^i$. E.g., the rectangle in the lower left edge labeled "4" together with its position denotes the term $(A_4 \odot B_4) \cdot \hat{x}^8$. The result $A \odot B$ is computed by summing up (XORing) all the terms according to their horizontal position. This product is $2k$ segments wide, as one would expect. The partial

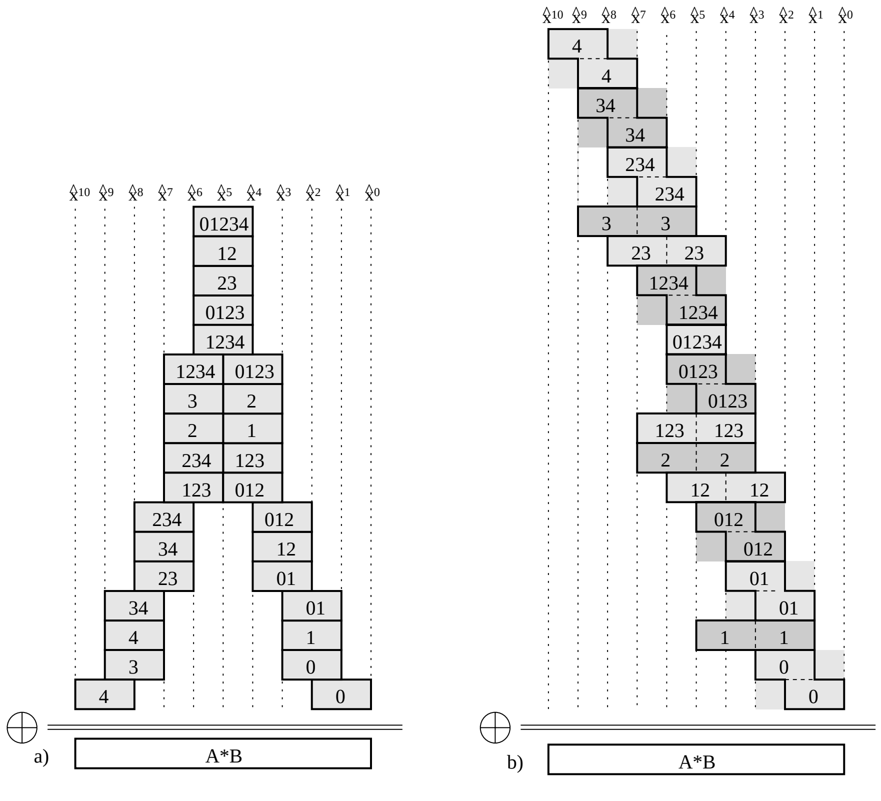

Fig. 5. 5-Segment Karatsuba multiplication and operand reordering

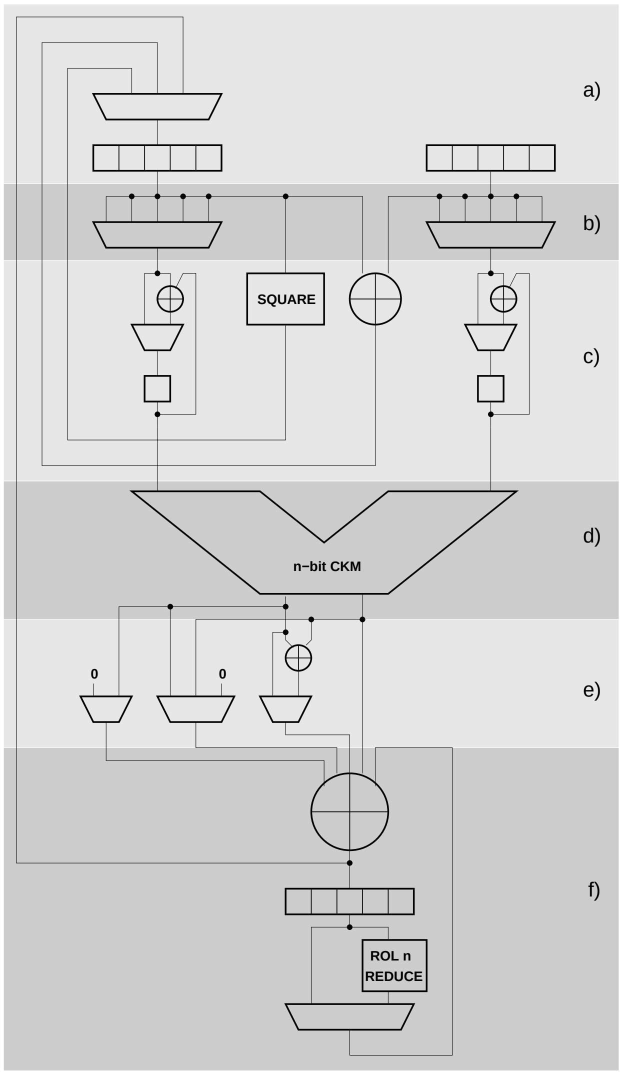

Fig. 6. Generic datapath architecture

products can be reordered as shown in Fig. 5b. This order was achieved in consideration of three optimization criteria.

First, most partial products are added two times in order to compute the final result. They can be grouped together and placed in one of three patterns, which are indicated in Fig. 5b. This is true for all instances of the multi-segment Karatsuba algorithm. In the datapath, these patterns are computed by some additional combinational logic, which is connected to the output signals of the CKM (see part (c) of Fig. 6).

Second, the resulting patterns are ordered by descending i of their factor $\hat{x}^{\,i}$. In this way, the product can be accumulated easily in a shift register.

Table 2. FFCP gate count

Datapath	FF	XOR_2	AND_2[3]	$MUX_{2:1}$	$MUX_{3:1}$	$MUX_{4:1}$
part a)	$2l$				l	
part b)						$2m$
part c)	$2m$	$2m+l+$ $2n+b$		$2m$		
part e)		m	$2m$	$2m$		
part f)	n	$4m$		n		
SUM	$2m+$ $2l+n$	$7m+l+$ $2n+b$	$2m$	$4m+n$	l	$2m$

n, b:	according to $P(x)=x^n+x^b+1$
m	: CKM bit width
l	: input Reg width

As the third optimization criterion the remaining degree of freedom is taken advantage of in the following way: The patterns are once more partially reordered, such that when iterating over them from top to bottom, one of the two following conditions holds: Either the current pattern is constructed out of a single segment product (e.g. $A_4 \odot B_4$), or the set of indices of the patterns segments differs only by one index from its predecessor (as in the partial products $(A_0 \oplus A_1) \cdot (B_0 \oplus B_1)$ and $(A_0 \oplus A_1 \oplus A_2) \odot (B_0 \oplus B_1 \oplus B_2)$). In Fig. 5b this criterion is met for all but one iteration step (namely it is not met for the step from “23” to “1234”). Thus, based on the datapath in Fig. 6 the computation of the partial product “1234” takes a total of two clock cycles, which is one more compared to all other iteration steps. The number of additional clock cycles due to the fact that this third criterion can not be met increases slowly with the number of segments k.

By applying the third optimization criterion to the pattern sequence, the partial product computations can be performed as follows: By placing m-bit accumulator registers at the inputs of the CKM, the terms $M_{m,l}(A, B)$ can be computed iteratively. This results in a two stage pipelined design for the complete datapath and yields a total of 17 clock cycles to perform one field multiplication utilizing the MSK_5.

The complete datapath is depicted in Fig. 6. In part a) the two operand registers of width $l = k \cdot m$ are shown as well as their partitioning into five segments. Both are implemented as shift-registers in order to allow data exchange with the external controller. The multiplexors in part b) select one from the five segments of the operands. They can both be controlled by the same set of signals, since they are always operating synchronously. Besides the combinational addition and squaring blocks, part c) illustrates the two accumulator registers. Both can either be loaded with a new segment, or they can accumulate intermediate segment sums. Section d) of Fig. 6 consists of the CKM. Part e) covers the pattern generation stage, which is mainly composed of multiplexors. Finally, in part f) the multiplication accumulator register is shown. It can either hold its value or the current pattern can be added to it in each cycle. Each time the intermediate result is shifted left by m bit, an interleaved reduction step according to Eqn. 10 is performed. This way, the accumulator needs only to be n bits wide, where n is the degree of the reduction polynomial. Furthermore, the necessary number of logic elements for the reduction step is minimized and no additional clock cycle is needed.

[3] $MUX_{2:1}$ components with constant zero inputs have been optimized to AND_2 gates.

In order to reduce the amount of communication between the controller and the FFCP, the result of the current computation is fed back to one of the operand registers. Thus, interim results need not inevitably be transferred several times between controller and FFCP.

Tab. 2 gives an overview of the amount of structural and logical components, which are required to implement the proposed datapath (excluding the CKM resources, please refer to Sec. 3.1 for the CKM implementation complexity). The number of states of the finite state machine, which controls the datapath, is in the order of Eqn. 8. Thus, logic resources for the FSM are negligibly small.

4 Atmel FPSLIC Hardware Platform

For the implementation of the previously detailed FF coprocessors the AT94K FPSLIC hardware platform from Atmel, Inc. is used within this work [14]. This product family integrates FPGA resources, an AVR 8-bit RISC microcontroller core, several peripherals and up to 36K Bytes SRAM within a single chip. The AVR microcontroller core is a common embedded processor, e.g., on SmartCards and is also available as a stand-alone device. The AVR is capable of 129 instructions, most of which can be performed within a single clock cycle. This results in a 20+ MIPS throughput at 25 MHz clock rate.

The FPGA resources within the FPSLIC devices are based on Atmel's AT40K FPGA architecture. A special feature of this architecture are FreeRam[4] cells which are located at the corners of each 4x4 cell sector. Using these cells results in minimal impact on bus resources and by that in fast and compact FPGA designs. The FPGA part is connected to the AVR over an 8-bit data bus. The amount of available FPGA resources ranges from about 5K system gates within the so-called μFPSLIC to about 40K system gates within the AT94K40.

Both, the AVR microcontroller core and the FPGA part are connected to the embedded memory separately. Up to 36K Bytes SRAM are organized as 20K Bytes program memory, 4K Bytes data memory and 12K Bytes that can dynamically be allocated as data or program memory.

Atmel provides a complete design environment for the FPSLIC including tools for software development (C Compiler), tools for hardware development (VHDL synthesis tools) and a HW/SW co-verification tool, which supports the concurrent development of hardware and software.

For the implementations detailed subsequently the Atmel ATSTK94 FPSLIC demonstration board is used. This board comes with a AT94K40 device and is running at 12 MHz clock rate. The FPGA part consists of 2304 logic cells and 144 FreeRam cells, which is equivalent to approx. 40K system gates.

5 Implementation

Three different prototype implementations were built in order to evaluate the architectures detailed in Sec. 3. Due to the restrictions in terms of available FPGA resources

[4] Each FreeRam cell is organized as a 32x4 bit dual-ported RAM block.

these implementations support 113 bit EC point multiplication only. This is certainly not sufficient for high-security applications, but can be applied in low-security environments.

The following sections present some implementation details and performance numbers for a purely software based implementation, a design that is accelerated with a 32-bit CKM and another one, which applies the FFCP. Furthermore an extension to the FFCP design is proposed and performance numbers for this extended version are estimated.

5.1 Pure Software without HW Acceleration

The software variant is entirely coded in assembler and has been optimized regarding the following design criteria:

– High performance.
– Resistance against side channel attacks.
– Easy SW/HW exchange of basic FF operations.

Concerning the performance, special effort has been spent at FF level in optimizing the field multiplication and reduction, which is the performance critical part of the entire $k{\cdot}P$ algorithm. At the EC level the so-called *2P Algorithm* documented in [15] is utilized to perform the EC point multiplication. This algorithm takes only 4 multiplications, 1 squaring and 2 additions in the underlying FF for one EC-Add computation. One EC-Double takes only 2 multiplications, 4 squaring and 1 addition. Summing up, this $k{\cdot}P$ implementation is about 2 times faster compared to standard *Double-and-Add* implementations. Furthermore, the *2P Algorithm* is inherently resistent against pertinent timing resp. power attacks, since in every iteration of its inner loop both operations (EC-Add and EC-Double) have to be computed, regardless of the binary expansion of k. Thus, besides some pre- and postprocessing overhead, one $k{\cdot}P$ computation over $\mathbb{GF}(2^n)$ takes exactly n EC-Add and n EC-Double operations. At the FF level countermeasures against side-channel attacks based on randomization and avoidance of conditional branches are applied as well [16].

Table 3. SW performance values

Operation	Bit Width	Clock Cycles		
		Computation	Overhead	Total
FF-Mult	16	96	NA	96
FF-Mult	32	$3 * 96 = 288$	131	419
FF-Mult	64	$3 * 419 = 1.257$	383	1.640
FF-Mult	128	$3 * 1.640 = 4.920$	489	5.409
FF-Square	128	340	NA	340
FF-Add	128	160	NA	160
FF-Reduce	113	420	NA	420
EC-Double	113	15.300	NA	15.300
EC-Add	113	25.200	NA	25.200
$k{\cdot}P$	113	4.752.000	NA	4.752.000

Tab. 3 summarizes the performance of the implementation on FF level as well as on EC level. The analysis of the $k{\cdot}P$ algorithm identifies the field multiplication as the most time consuming operation, which amounts to about 85% of the overall cycle count.

5.2 Hardware Acceleration

The subsequently detailed FPGA designs have been implemented by using the design tools which are packaged with the utilized demonstration board. For hardware synthesis this is *Leonardo v2000.1b* from Mentor, Inc. The FPGA mapping is done with *Figaro IDS v7.5* from Atmel, Inc. Also from Atmel, Inc. there is the top-level design environment called *System Designer v2.1*, which is required to build up the entire design based on the AVR and the FPGA part.

Acceleration based on CKM. The genuine SW implementation can be accelerated by utilizing a CKM as presented in Sec. 3.1, which is implemented in the FPGA part of the AT94K40 device. Matching to the particular bit width m of the raw CKM, two m-bit input registers and a $2m$-bit output register is added on the HW side. In order to allow a reasonable communication over the fixed 8-bit interface, the input registers are designed as 8-bit shift-in and parallel-out registers. Accordingly, the output register is parallel-in and 8-bit shift-out.

Table 4. 32-bit CKM performance values

Operation	Bit	Clock Cycles		
	Width	Computation	Overhead	Total
FF-Mult	32	1	16	17
FF-Mult	64	$3 * 17 = 51$	383	434
FF-Mult	128	$3 * 434 = 1.302$	489	1.791
EC-Double	113	8.100	NA	8.100
EC-Add	113	10.700	NA	10.700
$k{\cdot}P$	113	2.201.000	NA	2.201.000

Tab. 4 summarizes the performance of the combined HW/SW implementation based on a 32-bit CKM. The 32-bit CKM takes about 53% of the FPGA resources. At the FF level this results in a speed-up of about 3 and for the $k{\cdot}P$ algorithm there is still a speed-up factor of about 2.2 compared to the values given in Tab. 3.

The CKM architecture is of special interest for HW platforms offering only a small amount of FPGA resources, such as the μFPSLIC (see Sec. 4). This device is still sufficient for the implementation of an 8-bit CKM, which results in 3384 cycles for one 128-bit field multiplication. This is still a speed-up of about 1.6 compared to the genuine SW implementation.

Acceleration based on FFCP. Utilizing the FFCP architecture detailed in Sec. 3.2 instead of the stand-alone CKM design allows for a further significant performance gain.

Table 5. FFCP performance values

Operation	Bit Width	FFCP Clock Cycles best case	FFCP Clock Cycles worst case	extended FFCP est. Clock Cycles
FF-Mult	113	32	152	19
FF-Add	113	16	136	3
FF-Square	113	1	91	3
EC-Double	113		493	53
EC-Add	113		615	85
$k \cdot P$	113		130.200	16.380

For the implementation presented here, the particular design parameters are fixed to 113-bit operand width, 24-bit CKM and 5-segment Karatsuba multiplication (MSK_5). This results in a FPGA utilization of 96% for the entire FFCP design.

Due to the fact that the result of each operation is fed back into one of the operand registers, the cycle count of a particular operation (I/O overhead plus actual computation) differs regarding to data dependencies. The corresponding best- and worst-case value for each FF operation is denoted in Tab. 5.

Tab. 5 unveils that the major part of cycles is necessary to transfer 113-bit operands over the fixed 8-bit interface between AVR and FPGA. These transfers can be avoided almost completely with an additional register file on the FFCP and an extended version of the finite state machine, which interprets commands given by the software running on the AVR. Assuming a 2-byte command format (4 bit opcode, 12 bit to specify the destination and the source registers) results in cycle counts according to the right column of Tab. 5. With respect to the FPSLIC architecture and their special FreeRAM feature, such a register file can be implemented without demand on additional logic cells. The extended version of the FFCP is currently under development on our site.

5.3 Performance Comparison

There are several FPGA based hardware implementations of EC point multiplication documented in the literature [12] [17] [18] [19]. The performance values of these state-of-the-art implementations are given in Tab. 6. Additionally, Tab. 6 comprises the particular figures of the previously described FPSLIC based implementations.

A performance comparison of hardware implementations against each other is in general not straight forward. This is mostly because of different key sizes and due to the fact that different FPGA technologies are used for their implementation.

A basically scalable HW architecture is common to all implementations referenced in Tab. 6. In contrast to our SoC approach, the implementations in [12] [17] [18] and [19] are mainly focusing on high-security, server based applications. Their functionality is entirely implemented within a relatively large FPGA and no arrangements against side-channel attacks are documented.

In [12] and [17] the underlying field representation is an optimal normal basis. Both implementations are based on FPGAs from Xilinx, Inc. Furthermore, VHDL module generators are used in both cases to derive the particular HW descriptions. The approach in [17] allows for a parameterization of the key size only. Parallelization, which

Table 6. Performance comparison

Target Platform	Bit Width	$k{\cdot}P$
FPGA (XCV300, 45 MHz) [17]	113	3.7 ms
FPGA (XC4085XLA, 37 MHz) [12]	155	1.3 ms
FPGA (EPF10K, 3 MHz) [18]	163	80.7 ms
FPGA (XCV400E, 76.7 MHz) [19]	167	210 μs
FPSLIC pure SW (AT94K40, 12 MHz)	113	396 ms
FPSLIC with 32-bit CKM (AT94K40, 12 MHz)	113	184 ms
FPSLIC with FFCP (AT94K40, 12 MHz)	113	10.9 ms
FPSLIC with ext. FFCP (AT94K40, 12 MHz)	113	1.4 ms (est.)

is essential in order to achieve maximum performance from a specific FPGA, is additionally supported by the design in [12]. For the implementation in [17] a XCV300 FPGA with a complexity of about 320K system gates is used. The design in [12] is based on a XC4085XLA device with approx. 180K system gates.

The implementations in [18] and [19] are both designed for polynomial bases and the field multiplications are in principle composed of partial multiplications.

The design in [18] is based on an Altera Flex10k family device with a complexity of about 310K system gates. The architecture is centered around a w_1-bit$\times w_2$-bit partial multiplier. Due to the flexibility in w_1 and w_2 it is shown, that the architecture scales well, even for smaller FPGA platforms.

The best performing implementation, representing the current benchmark with respect to $k{\cdot}P$ performance, is described in [19]. It is highly optimized, exploiting both pipelining and concurrency. The field multiplication is performed with a digit-serial multiplier. A Xilinx XCV400E FPGA with a complexity of about 570K system gates, running at 76.7 MHz is used for the implementation. Compared to our design this signifies a factor of more than 10 in space and a factor of about 6 in speed.

6 Conclusion

Speeding up the most time critical part of EC crypto schemes enables the use of these methods within combined HW/SW systems with relatively low computing power. Running the EC level algorithms in SW facilitates algorithmic flexibility while the required performance is contributed by dedicated coprocessors.

Two generic and scalable architectures of FF coprocessors (CKM and FFCP) which are qualified for SoC implementations have been illustrated in this paper. While CKM supports only multiplication, the FFCP architecture implements multiplication, addition and squaring completely within HW. The proposed multi-segment Karatsuba multiplication scheme, which is the core of the FFCP architecture, permits fast and resource saving HW implementations. By exploiting the presented coprocessor architectures a considerable speed-up of EC cryptosystems can be achieved.

Acknowledgment. This work was sponsored by and has been done in cooperation with cv cryptovison GmbH, Gelsenkirchen, Germany.

References

1. R. L. Rivest, A. Shamir and L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, Feb 1978.
2. V. Miller, "Use of elliptic curves in cryptography," *Advances in Cryptology, Proc. CRYPTO'85*, LNCS 218, H. C. Williams, Ed., Springer-Verlag, pp. 417–426, 1986.
3. N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.
4. A. Lenstra and E. Verheul, "Selecting Cryptographic Key Sizes," *Proc. Workshop on Practice and Theory in Public Key Cryptography*, Springer-Verlag, ISBN 3540669671, pp. 446–465, 2000.
5. A. J. Menezes, "Elliptic Curve Public Key Cryptosystems," Kluwer Akademic Publishers, 1993.
6. J. H. Silverman, "The Arithmetic of Elliptic Curves," *Graduate Texts in Mathematics*, Springer-Verlag, 1986.
7. S. Galbraith and N. Smart, "A cryptographic application of Weil descent," *Codes and Cryptography*, LNCS 1746, Springer-Verlag, pp. 191–200, 1999.
8. IEEE 1363, "Standard Specifications For Public Key Cryptography," http://grouper.ieee.org/groups/1363/, 2000.
9. ANSI X9.62, "Public key cryptography for the financial services industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)," (available from the ANSI X9 catalog), 1999.
10. A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Sov. Phys.-Dokl (Engl. transl.), vol. 7, no. 7*, pp. 595–596, 1963.
11. D. V. Bailey and C. Paar, "Efficient Arithmetic in Finie Field Extensions with Application in Elliptic Curve Cryptography," *Journal of Cryptology*, vol. 14, no. 3, pp. 153–176, 2001.
12. M. Ernst, S. Klupsch, O. Hauck and S. A. Huss, "Rapid Prototyping for Hardware Accelerated Elliptic Curve Public-Key Cryptosystems," *Proc. 12th IEEE Workshop on Rapid System Prototyping (RSP01)*, Monterey, CA, 2001.
13. J. L. Bentley, D. Haken and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *SIGACT News*, vol. 12(3), pp. 36–44, 1980.
14. Atmel, Inc. "Configurable Logic Data Book," 2001.
15. J. Lopez and R. Dahab, "Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 99)*, LNCS 1717, C.K. Koc and C. Paar Eds., Springer-Verlag, pp. 316–327, 1999.
16. J. Coron, "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 99)*, LNCS 1717, C.K. Koc and C. Paar Eds., Springer-Verlag, pp. 292–302, 1999.
17. K.H. Leung, K.W. Ma, W.K. Wong and P.H.W. Leong, "FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor," *Proc. IEEE FCCM 2000*, pp. 68–76, Napa Valley, 2000.
18. S. Okada, N. Torii, K. Itoh and M. Takenaka, "Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, LNCS 1965, C.K. Koc and C. Paar Eds., Springer-Verlag, pp. 25–40, 2000.
19. G. Orlando and C. Paar, "A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, LNCS 1965, C.K. Koc and C. Paar Eds., Springer-Verlag, pp. 41–56, 2000.

Appendix: 3-Segment Karatsuba Multiplication

For any polynomials $A, B \in \mathbb{GF}(2^n)$ the product $C = A \cdot B = MSK_3(A,B)$ using the 3-segment Karatsuba multiplication is according to Eqn. 6 given by:

$$MSK_3(A,B) = \left(\bigoplus_{i=1}^{3} S_{i,0}(A,B) \cdot \hat{x}^{i-1} \right) \oplus \left(\bigoplus_{i=1}^{2} S_{3-i,i}(A,B) \cdot \hat{x}^{i+2} \right)$$

$$\begin{array}{ll}
= S_{1,0}(A,B) & \hat{x}^0 \oplus \\
 S_{2,0}(A,B) & \hat{x}^1 \oplus \\
 S_{3,0}(A,B) & \hat{x}^2 \oplus \\
 S_{2,1}(A,B) & \hat{x}^3 \oplus \\
 S_{1,2}(A,B) & \hat{x}^4 \\
\\
= M_{1,0}(A,B) & \hat{x}^0 \oplus \\
 (S_{1,0}(A,B) \oplus S_{1,1}(A,B) \oplus M_{2,0}) & \hat{x}^1 \oplus \\
 (S_{1,0}(A,B) \oplus S_{2,0}(A,B) \oplus S_{1,2}(A,B) \oplus S_{2,1}(A,B) \oplus M_{3,0}(A,B)) & \hat{x}^2 \oplus \\
 (S_{1,1}(A,B) \oplus S_{1,2}(A,B) \oplus M_{2,1}(A,B)) & \hat{x}^3 \oplus \\
 M_{1,2}(A,B) & \hat{x}^4 \\
\\
= M_{1,0}(A,B) & \hat{x}^0 \oplus \\
 (M_{1,0}(A,B) \oplus M_{1,1}(A,B) \oplus M_{2,0}(A,B)) & \hat{x}^1 \oplus \\
 (M_{1,0}(A,B) \oplus S_{1,0}(A,B) \oplus S_{1,1}(A,B) \oplus M_{2,0}(A,B) \oplus M_{1,2}(A,B) \oplus & \\
 S_{1,1}(A,B) \oplus S_{1,2}(A,B) \oplus M_{2,1}(A,B) \oplus M_{3,0}(A,B)) & \hat{x}^2 \oplus \\
 (M_{1,1}(A,B) \oplus M_{1,2}(A,B) \oplus M_{2,1}(A,B)) & \hat{x}^3 \oplus \\
 M_{1,2}(A,B) & \hat{x}^4 \\
\\
= M_{1,0}(A,B) & \hat{x}^0 \oplus \\
 (M_{1,0}(A,B) \oplus M_{1,1}(A,B) \oplus M_{2,0}(A,B)) & \hat{x}^1 \oplus \\
 (M_{1,0}(A,B) \oplus M_{1,0}(A,B) \oplus M_{1,1}(A,B) \oplus M_{2,0}(A,B) \oplus M_{1,2}(A,B) \oplus & \\
 M_{1,1}(A,B) \oplus M_{1,2}(A,B) \oplus M_{2,1}(A,B) \oplus M_{3,0}(A,B)) & \hat{x}^2 \oplus \\
 (M_{1,1}(A,B) \oplus M_{1,2}(A,B) \oplus M_{2,1}(A,B)) & \hat{x}^3 \oplus \\
 M_{1,2}(A,B) & \hat{x}^4 \\
\\
= M_{1,0}(A,B) & \hat{x}^0 \oplus \\
 (M_{1,0}(A,B) \oplus M_{1,1}(A,B) \oplus M_{2,0}(A,B)) & \hat{x}^1 \oplus \\
 (M_{2,0}(A,B) \oplus M_{2,1}(A,B) \oplus M_{3,0}(A,B)) & \hat{x}^2 \oplus \\
 (M_{1,1}(A,B) \oplus M_{1,2}(A,B) \oplus M_{2,1}(A,B)) & \hat{x}^3 \oplus \\
 M_{1,2}(A,B) & \hat{x}^4
\end{array}$$

with

$$\begin{array}{l}
M_{1,0}(A,B) = A_0 \cdot B_0 \\
M_{1,1}(A,B) = A_1 \cdot B_1 \\
M_{1,2}(A,B) = A_2 \cdot B_2 \\
M_{2,0}(A,B) = (A_0 \oplus A_1) \cdot (B_0 \oplus B_1) \\
M_{2,1}(A,B) = (A_1 \oplus A_2) \cdot (B_1 \oplus B_2) \\
M_{3,0}(A,B) = (A_0 \oplus A_1 \oplus A_2) \cdot (B_0 \oplus B_1 \oplus B_2)
\end{array}$$