2010 International Conference on Field Programmable Logic and Applications

High-Performance Integer Factoring with Reconfigurable Devices

Ralf Zimmermann, Tim Güneysu and Christof Paar
Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
Email: {zimmermann,gueneysu,cpaar}@crypto.rub.de

***Abstract*—We present a novel FPGA-based implementation of the Elliptic Curve Method (ECM) for the factorization of medium-sized composite integers. More precisely, we demonstrate an ECM implementation capable to determine prime factors of up to 2,424 151-bit integers per second using a single Xilinx Virtex-4 SX35 FPGA. Using this implementation on a cluster like the COPACOBANA is beneficial for attacking cryptographic primitives like the well-known RSA cryptosystem with advanced methods such as the Number Field Sieve (NFS).**

To provide this vast number of integer factorizations per FPGA, we make use of the available DSP blocks on each Virtex-4 device to accelerate low-level arithmetic computations. This methodology allows the development of a time-area efficient design that runs 24 ECM cores in parallel, implementing both phase 1 and phase 2 of the ECM. Moreover, our design is fully scalable and supports composite integers in the range from 66 to 236 bits without any significant modifications to the hardware. Compared to the implementation by Gaj *et al.*, who reported an ECM design for the same Virtex-4 platform, our improved architecture provides an advanced cost-performance ratio which is better by a factor of 37.

***Index Terms*—Factorization, elliptic curve method, reconfigurable hardware, COPACOBANA.**

I. Introduction

In 1987, the Elliptic Curve Method (ECM) was introduced by H. W. Lenstra [1] as a new method for integer factorization, generalizing the concept of Pollard's $p-1$ and Williams' $p+1$ method [2], [3]. Although the ECM is known not to be the fastest method for factorization with respect to asymptotical time complexity, it is widely used to factor composite numbers up to 200 bits due to its very limited requirements on memory.

The most prominent application that relies on the hardness of the factorization problem is the RSA cryptosystem. An attacker on RSA has to find the factorization of a composite number n which consists of two large primes p, q. More precisely, the RSA security parameter n is larger than 1024 bits and hence out of reach of the ECM. Up to date, such large bit sizes are preferably attacked with the most powerful methods known so far, such as the *Number Field Sieve* (NFS). However, the complex NFS[1] involves the search of relations in which many mid-sized numbers need to be tested if they are "smooth", i.e., composed only of small prime factors not larger than a fixed boundary B. In this context, ECM is an important tool to determine the smoothness of such integers (i.e., if they can be factored into small primes), in particular due to its moderate resource requirements.

The fastest ECM implementations for retrieving factors of composite integers are software-based; a state-of-the-art system is the GMP-ECM software published by P. Zimmermann *et al.* [5] and has been extended for use with GPUs by Bernstein *et al.* [6]. As a promising alternative, efficient hardware implementations of the ECM were first proposed in 2005: Šimka *et al.* [7] demonstrated the feasibility to implement the ECM in reconfigurable hardware by presenting a first proof-of-concept implementation. Their results were improved by Gaj *et al.* [8], [9], who also showed a complete hardware implementation of ECM phase 2. However, the low-level arithmetic in these implementations were only implemented using straightforward techniques within the configurable logic which yet leaves room for further improvements. To fill this gap, de Meulenaer *et al.* [10] proposed an unrolled Montgomery multiplier based on a two-dimensional pipeline on Xilinx Virtex-4 FPGAs to accelerate the field arithmetic. However, due to limitations in area and the long pipeline design, their design only efficiently supports the first phase of the ECM.

Contribution: In this work we propose a novel ECM architecture for Xilinx Virtex FPGAs making use of DSP blocks for the computationally intensive arithmetic. Our focus is to accelerate the underlying field arithmetic of the ECM on FPGAs without sacrificing the option to combine both phase 1 and 2 in a single core. Thus, we adopt some high-level decisions like memory-management and the use of SIMD instructions from [8] which also supports both phases on the same hardware. To improve the field arithmetic, we place fundamental arithmetic functions like adders and multipliers in embedded DSP blocks of modern FPGAs. For factoring large amounts of numbers, we finally describe our factorization setup based on a variant of COPACOBANA (Cost Optimized PArallel COde Breaker) - a cluster system based on FPGAs [11], [12].

Outline: We start with a short review on the mathematical background and the concept of the ECM. In Section III we first describe the cluster system COPACOBANA, which represents the target platform of our work, and then discuss the architecture of an ECM core and its corresponding arithmetic components. Finally, we present our factorization results in Section IV before we conclude with Section V.

[1]The NFS comprises of four steps, the polynomial selection, relation finding, a linear algebra step and finally the square root step. The relation finding step is most time-consuming, taking roughly 90% of the runtime. For more information on the NFS refer to [4].

978-0-7695-4179-2/10 $26.00 © 2010 IEEE
DOI 10.1109/FPL.2010.26

II. Mathematical Background

We start with a brief introduction of the $p-1$ method for factorization to motivate the concept of the ECM. Let $k \in \mathbb{N}$ and n be the composite to be factored. Furthermore, let $p|n$ with $p \in \mathbb{P}$, $a \in \mathbb{Z}$ and n be co-prime, i.e., $gcd(a,n) = 1$. Now take Fermat's little Theorem [13, Fact 2.127] with $a^{p-1} \equiv 1 \bmod p$. The extension by the k-multiple of $(p-1)$ leading to $a^{k(p-1)} \equiv 1 \bmod p$ holds as well since $(a^{p-1})^k \equiv 1^k = 1 \bmod p$. Then, with $e = k(p-1)$ we have

$$\begin{aligned} & a^{k(p-1)} = a^e \equiv 1 \bmod p \\ \Rightarrow \quad & a^e - 1 \equiv 0 \bmod p \\ \Rightarrow \quad & p|(a^e - 1) \end{aligned}$$

Hence, if $a^e \neq 1 \bmod n$ we know

$$1 < gcd(a^e - 1, n) < n.$$

In this case, we found a non-trivial divisor of n. However, we are not able to compute $e = k(p-1)$ without knowledge of p. Thus, we assume that $p-1$ decomposes solely into small prime factors p_i less than a defined bound $B1$ (in this case, $p-1$ is called $B1$-smooth). Now we choose e as product of all prime powers p_i^r lower than $B1$ and hope that e is a multiple of $p-1$:

$$e = \prod_{p_i \in \mathbb{P}; p_i < B1} p_i^{\lfloor \log_{p_i}(B1) \rfloor} \tag{1}$$

By computing $d = gcd(a^e - 1, n)$ we finally hope to find a divisor d of n. Now, we transfer this idea to elliptic curves $\mathcal{E}$. Let us assume q to be a factor of n and $|\mathcal{E}(\mathbb{Z}_q)|$ is $B1$-smooth so that e, according to the construction in Equation (1), is a multiple of $q-1$. Note that point multiplication by $|\mathcal{E}(\mathbb{Z}_q)|$ (or multiples of $|\mathcal{E}(\mathbb{Z}_q)|$) returns the point at infinity, e.g., $Q = eP = \mathcal{O}$ for an arbitrary point P and resulting point Q. Recall that the resulting point $Q = \mathcal{O}$ implies a prior impossible division by z_Q with $z_Q \equiv 0 \bmod q$. Note that we actually perform all point operations in $\mathcal{E}(\mathbb{Z}_n)$ since we do not know q. Hence, we compute $Q = eP$ in $\mathcal{E}(\mathbb{Z}_n)$ and hope to yield a point Q with coordinate $z_Q \neq 0 \bmod n$ but $z_Q \equiv 0 \bmod q$. Then, the factor q of n is obtained by $q = gcd(z_Q, n)$.

From an algorithmic point of view, we can discover a factor q of n as follows: in the first phase of the ECM, we compute $Q = eP$, where e is a product of prime powers $p^i \leq B1$ with appropriately chosen smoothness bounds. The second phase increases the success probability by checking for each prime $B1 < p \leq B2$ whether pQ reduces to the neutral element in $E(\mathbb{Z}_q)$. This second phase increases the chance to find a factor of n as shown by [14]. Algorithm 1 summarizes all necessary steps of both phases of the ECM.

If we consider a single curve only, the properties of the method are closely related to those of Pollard's $(p-1)$-method that can fail by returning a trivial divisor, such as 1 or n. The advantage of ECM is apparent with the possibility of choosing another curve (and thus group order) after each unsuccessful trial, increasing the probability of retrieving factors of n. If the final gcd of the product s and n satisfies $1 < gcd(s,n) < n$, a factor is found. Note that the parameters $B1$ and $B2$ control the probability of finding a divisor q. More precisely, if the order of P factors into a product of prime powers (each $\leq B1$) and at most one additional prime between $B1$ and $B2$, the prime factor q is discovered.

Algorithm 1 The Elliptic Curve Method

Input: Composite $n = f_1 \cdot f_2 \cdot \ldots \cdot \ldots \cdot f_n$.
Output: Factor f_i of n.
1: **Phase 1:**
2: Choose arbitrary curve $E(\mathbb{Z}_n)$ and random point $P \in E(\mathbb{Z}_n) \neq \mathcal{O}$.
3: Choose smoothness bounds $B1, B2 \in \mathbb{N}$.
4: Compute $e = \prod_{p_i \in \mathbb{P}; p_i < B1} p_i^{\lfloor \log_{p_i}(B1) \rfloor}$
5: Compute $Q = eP = (x_Q; y_Q; z_Q)$.
6: Compute $d = gcd(z_Q, n)$.
7: **Phase 2:**
8: Set $s := 1$.
9: **for** each prime p with $B1 < p \leq B2$ **do**
10: Compute $pQ = (xpQ : ypQ : zpQ)$.
11: Compute $d = d \cdot zpQ$.
12: **end for**
13: Compute $f_i = gcd(d, n)$.
14: **if** $1 < f_i < n$ **then**
15: A non-trivial factor d is found.
16: return f_i
17: **else**
18: Restart from Step 2 in Phase 1.
19: **end if**

Note that in the last years, most designs for the ECM preferably use elliptic curves in Montgomery or Edwards form [15], [16] since it allows to compute points with simplified formulas, e.g., by only using the x- and z-coordinates. In this work, we will use Montgomery curves since they allow highly regular computation patterns which are favorable for hardware circuits.

III. Designing the ECM System

To factor a large amount of mid-sized numbers as required by the NFS, we chose COPACOBANA as the target platform for our ECM implementation. COPACOBANA [11] is a parallel low-cost cluster system that can be populated with up to 128 FPGAs in a single housing. More precisely, we modified the COPACOBANA architecture for this work to host Virtex-4 FPGA devices distributed among 16 plug-in cards which are all connected to single backplane. Each Virtex-4 SX device provides 192 DSP blocks which can be cascaded to support pipelined, arithmetic intensive computations as shown in Section III-B. Note that latest Spartan-6 FPGAs (which also come with a large number of DSP blocks) are another promising alternative due to their cost-performance ratio as target device. However, they are currently not available in sufficiently large quantities.

A. Architecture of an ECM Core

We will now describe the layout of our implementation. As it adopts some of the design choices by Gaj *et al.*, we will begin with a short summary of the basic system and then outline our improvements.

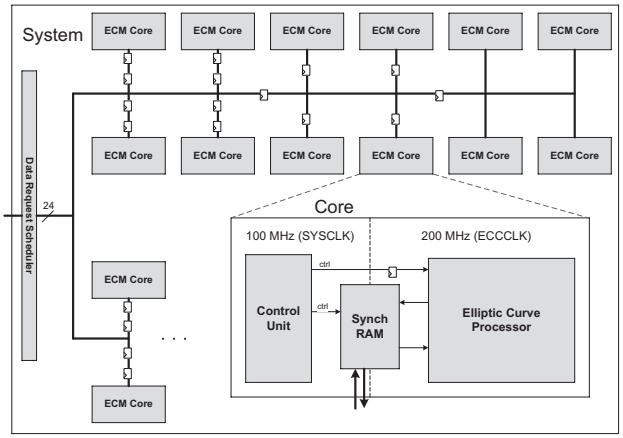

Fig. 1: Layout of the ECM system and ECM core

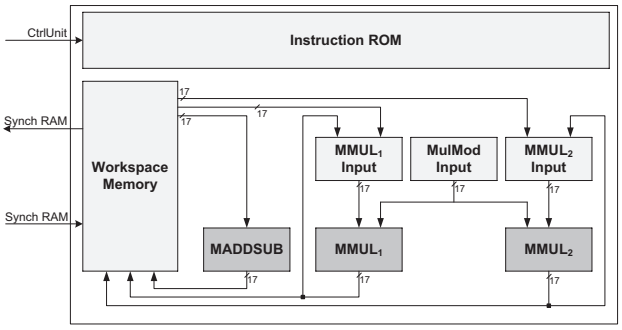

Fig. 2: Layout of the Elliptic Curve Processor

Figure 1 shows the components of our design: It consists of a buffered system bus which is fed by a data request scheduler which manages the I/O communication and timing dependencies and a fixed number ECM cores. The COPACOBANA is connected to a host PC which takes care of the necessary pre- and post-processing. Depending on the size of the communication interface (not included in the figure), up to 24 ECM cores can be placed on a Virtex-4 SX35 FPGA with the number of DSPs as the limiting factor.

Previous versions of COPACOBANA did not require a fast communication interface since applications performing an exhaustive key search rarely exchange data. However, parameters and results for ECM operations need to be transferred between the cluster and the host-PC. In particular, some operations like gcd computations and the generation of elliptic curves and corresponding parameters are costly in hardware and are performed on the host system.

In the lower part of Figure 1, we outline an ECM core in detail. It consists of a control unit, the Elliptic Curve Processor (ECP) and a memory buffer to synchronize between clock domains. Each core is divided into two clock domains: The control unit operates at a 100 MHz clock, controlling the ECP and setting up the necessary computation parameters, while the ECP itself runs at a 200 MHz clock performing the arithmetic intensive operations. In order to cross these domains (i.e., for receiving the ECM parameters or sending the results) a dual-ported memory buffer is used, each port operating at the respective clock speed.

As our FPGA has a total of 192 memory modules, each ECM core contains a local memory storing parameters and instructions for phase 1 and 2 (control unit) and each ECP contains a dedicated instruction ROM. This redundancy is beneficial to reduce the critical path length in order to enable the fast ECP clock domain.

Gaj *et al.* suggested an efficient way to implement both phases of the ECM using two modular multiplication and one modular addition/subtraction unit (cf. [8]). We adopt this proposal for an high-level abstraction but implement a DSP-based ECP, as shown in Figure 2, for improved performance. The processor contains a large instruction ROM, controlling four dedicated memory blocks and three underlying arithmetic units. It implements the sequences for point addition, point doubling, one step of the Montgomery Ladder [15] as well as optimized instructions to process the second phase.

To implement the second phase of the ECM, we chose the standard continuation method with a different precomputation and memory layout to that used in [8]. To continue with the previous point result from phase 1 and additional primes, we first need to store these primes in a $primetable$. Furthermore, we use a time-memory trade-off (dependent on a parameter $D = 210$) to store differences between individual primes represented using an interval scanning technique. For this, we calculate a set of 24 fixed multiples of the resulting point Q_0 as well as the boundary points DQ_0 and $M_{MIN}(B_1)DQ_0$ ($M_{MIN}(B_1) \geq 2$). As the parameter D is fixed for a given implementation, we computed an addition chain[2] which returns all points for the chosen B_1. Using the suggested boundary $B_1 = 960$, our addition chain needs to perform 30 point additions and doubles 7 times instead of 1 point doubling and 52 point additions as reported in [8].

The computation suggested by Gaj *et al.* for phase 2 uses two multiplication and one addition/subtraction unit to perform subsequently three modular multiplications and corresponding additions per round after an initial delay of one leading multiplication. We remark that the initial delay exists in every round (and thus $M_{MAX} - M_{MIN} + 1$ times in total). Additionally, in case if this is an odd number, the final computation requires special handling since only one point is used instead of two. Algorithm 2 shows an improved version of the standard continuation which provides remedy to both issues and only requires a single initial multiplication per run.

We work on the variables r, s and d and delay the computation of $r - s$ to the next iteration. Thus, we require an initialization (line 4) using the first prime we found in the $primetable$. Afterwards, these variables are updated in each iteration. For round m, we find two distinct points aQ_0 and bQ_0 and process these subsequent values (lines 11 to 14). At the end of the iteration, we renew the variables d, r and s. As

[2]Note that the chain must ensure that for any point addition $R+Q$ we have already calculated (and stored) the corresponding difference point $R - Q$.

Algorithm 2 Computation of d in Phase 2

Input: Bit table $primetable$ and points $R = (R_x :: R_z) = M_{MIN}DQ_0$, $Q = DQ_0$ and jQ_0
Output: Output d of second phase

1: $m \leftarrow M_{MIN}$, $pt \leftarrow primetable_m$
2: $i \leftarrow$ `findNextPosition(pt)`
3: $A \leftarrow j_iQ_0$
4: $\mathbf{r} \leftarrow A_zR_x$, $\mathbf{s} \leftarrow A_xR_z$, $\mathbf{d} \leftarrow 1$
5: **while** $m \leq M_{MAX}$ **do**
6: $i \leftarrow$ `findNextPosition(pt)`
7: **if** $i \neq$ END **then**
8: $A \leftarrow j_iQ_0$
9: $i \leftarrow$ `findNextPosition(pt)`
10: **if** $i \neq$ END **then**
11: $B \leftarrow j_iQ_0$
12: $t \leftarrow A_zR_x$, $u \leftarrow A_xR_z$, $d_1 \leftarrow r - s$
13: $\tilde{d} \leftarrow \tilde{d} \cdot d_1$, $\mathbf{r} \leftarrow B_zR_x$, $d_2 \leftarrow t - u$
14: $\mathbf{d} \leftarrow \tilde{d} \cdot d_2$, $\mathbf{s} \leftarrow B_xR_z$
15: **else**
16: $\mathbf{r} \leftarrow A_zR_x$, $\mathbf{s} \leftarrow A_xR_z$, $d_1 \leftarrow r - s$
17: $\mathbf{d} \leftarrow d \cdot d_1$
18: **end if**
19: **end if**
20: **if** i = END **then**
21: $m \leftarrow m + 1$, $pt \leftarrow primetable_m$
22: $R \leftarrow R + Q$
23: **end if**
24: **end while**
25: $d_1 \leftarrow r - s$
26: $\mathbf{d} \leftarrow d \cdot d_1$

soon as we reach the end of $primetable_m$, we have one of two cases: either we found only aQ_0, or no point since the last iteration. We cannot process aQ_0 with the first prime found in $primetable_{m+1}$, because we need to update the point R for the next iteration. Therefore, we update the variables r, s and d using two multiplications (lines 16 and 17). The second case does not provide a problem and requires no special handling. In both cases we increase m and modify R according to the basic algorithm. When we reach $m = M_{MAX}$, we process the values remaining in r, s and return d (lines 25 and 26).

B. Modular Arithmetic Units

In this section we discuss the implementation of modular arithmetic units for elliptic point computations relying on the fast arithmetic using DSP blocks.

1) Modular Multiplication Unit: The modular multiplication $A \cdot B \bmod M$ implemented in this work is a variant of the modular multiplication algorithm without trial division proposed by Montgomery [17]. Orup published several modifications for this method to simplify the quotient handling in [18]. Straightforward quotient handling is particularly important as it allows consecutive arithmetic operations in the DSP blocks which require a rather fixed realignment of operands. In this context, Orup's improvements only demand for word-wise (i.e., k-bit) multiplications, additions and shifting which are natively supported by the DSP blocks of Virtex-4 FPGAs. Hence, using Orup's variant *Modular Multiplication with Quotient Pipelining*, all arithmetic operations can be performed by sequential instructions in DSP blocks without need of additional resources in the configurable logic.

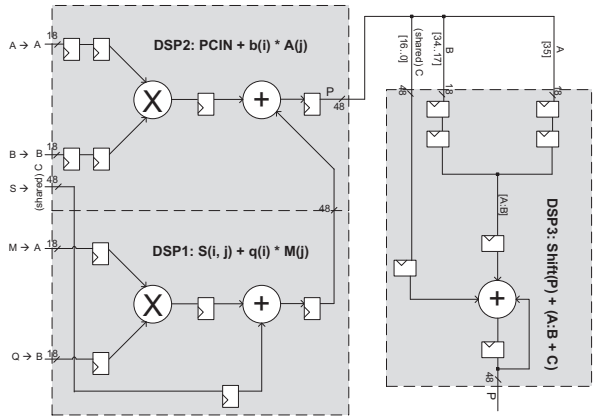

Fig. 3: DSP configuration of the modular multiplication unit

We will now shortly revise Orup's modification to the Montgomery multiplication. It needs $n+2+d$ steps for a fixed delay parameter d. We set the pipelining parameter $d = 0$ and simplify the algorithm as shown in Algorithm 3.

Algorithm 3 Modular Multiplication with Quotient Pipelining and $d = 0$

Input: Modulus $M > 2$ with $\gcd(M, 2) = 1$, $k, n \in N : 4\check{M} < 2^{kn}$. $\check{M} := (\tilde{M} + 1)$ div 2^k, $\tilde{M} := (M' \bmod 2^k)M$, with $(-MM')$ mod $2^k = 1$. Integers A, B in Montgomery Form with $2^{kn}R^{-1} \equiv 1$ mod M and $0 \leq A \leq 2\check{M}$, $0 \leq B \leq 2\check{M}$. b_i refers to the i-th k-bit block of B.
Output: $S_{n+2} \equiv ABR^{-1} (\text{mod } M)$ and $0 \leq S_{n+2} < 2\check{M}$

$S_0 := 0$
for $i := 0$ to n **do**
$q_i := S_i \bmod 2^k$
$S_{i+1} := S_i$ div $2^k + q_i \cdot \check{M} + b_iA$;
end for
$S_{n+2} := 2^kS_{n+1}$

The definition of $\check{M}$ represents the relation between the number of blocks n and the size of the modulus M: $M < 2^{k(n-1)-2}$. To use dedicated DSP cores, we use the static word size of $k = 17$. In turn, if we implement $n = 10$, we allocate 170 bits for each intermediate result to perform computations on a 151-bit modulus. This overhead is required because the algorithm does not compute the final reduction to save time.

Our implementation uses a fixed pipeline of three DSP cores to sequentially calculate the result. Using a sequential approach that processes individual word segments provides several advantages for low-level optimizations, e.g., the efficient use of counters to address input and output memory.

Figure 3 shows the configuration of these three DSP cores. The index i denotes the round while the index j represents the current 17-bit block of the input. Note that an additional feedback path which connects the result block to the q_i input is not included in the figure for the sake of clarity. Our multiplier needs a total of $n(n+2)+6$ clock cycles to compute and return the result.

2) Modular Addition/Subtraction Unit: The modular addition/subtraction $A+B \bmod M$ is no critical operation in terms of clock cycles. However, the proposed formulae for elliptic

curve computations require that two modular additions must not exceed the time of one modular multiplication. Again, we use the arithmetic power of DSP blocks for implementing the modular addition, significantly reducing the required amount of configurable logic resources.

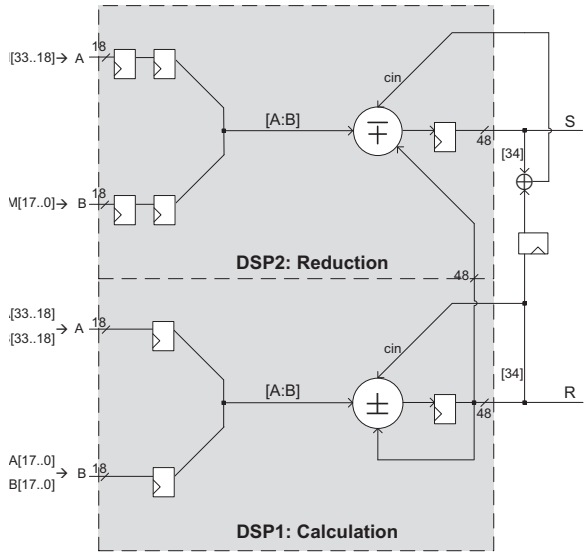

Fig. 4: DSP configuration of the modular addition/subtraction unit.

Figure 4 depicts the DSP configuration for the addition/subtraction unit, consisting of two pipelined and cascaded DSP cores. Each core supports the concatenation of the (signed) 18 bit inputs A and B. We use this to load two 17-bit blocks of the first value A_j into the result register and use the feedback path to add two 17-bit blocks of the second value B_j. This requires a dynamic operation mode configuration, as the DSP alternates between loading and adding the feedback path[3].

After processing the block j, the first DSP passes the intermediate result $R_j = A_j \pm B_j (+ \text{ CIN})$ to the second DSP for reduction $S_j = R_j \mp M_j (+ \text{ CIN})$. Both operations work in parallel, only shifted by one clock cycle. At the end of the computation, the unit evaluates the borrow bit of the subtraction (either by the main or reduction DSP) and determines the correct output, i.e., R or S, respectively.

As we decided to feed results from a modular multiplication directly to this unit, we have to process n blocks with 17-bit each and subsequently reduce them by $2\check{M}$. With respect to the pipelining registers, this leads to a total of $2n + 3$ clock cycles to compute and return the result.

IV. Results

We present the results of our implementation based on Xilinx ISE Foundation 11 for synthesis and implementation. All results are based on ECM parameters $B_1 = 960$, B_2 =57,000, $D = 210$ and the fixed scalar e of 1,323 bit. These parameters were used in most of the related work and allow a fair comparison.

Our ECM system is designed to allow straightforward scalability and to factor composite integers with smaller or larger bit sizes. It supports bit length of $L = 17(b-1)-2$ with $b = 5..15$ without notably hardware changes. This corresponds to an input bit range from 66 to 236 bits. To adjust the bit length, we have to place and route the ECM system with modified generics increasing or decreasing the internal size of the output buffers of the elliptic curve processor without changing the rest of the architecture. Afterwards, we generate a new instruction sequence and load it into the instruction ROM.

FPGA Type	Virtex-4 SX 35
Number of Cores	24
Arithmetic Clock Frequency	200 MHz
# Cycles (Phase I)	945,746
# Cycles (Phase II)	1,024,029
# Cycles (Phase I + II)	1,969,775
Time (Phase I)	4.73 ms
Time (Phase II)	5.12 ms
Phase I per Second	5,064
Phase I + II per Second	2,424

TABLE I: Implementation results of our ECM system factoring integers up to 151 bit.

Table I shows the results for the block size $b = 10$, factoring integers up to 151 bit in 4.73 ms (phase 1) and 5.12 ms (phase 2). This corresponds to 5,064 computations per second for the first phase and 2,424 for phase 1+2.

We compare our result to the Virtex-4 implementation by Gaj *et al.* and de Meulenaer *et al.* in Table II. For a fair comparison, we scaled our design to support the same bit lengths as reported in [9], [10]. For a consistent pricing model, we refer to the official Xilinx distributor Avnet Electronics Marketing[4]. The information are as of January, 2010 for single purchase without discount, rounded up to the next dollar.

Note that comparing to the architecture proposed by Gaj *et al.* is possible since performance figures for same platform (Xilinx Virtex-4) and scope of implementation (ECM phase 1 and phase 2) are available. However, we like to emphasize that this does not work for the design by de Meulenaer *et al.* The latter design only implements phase 1 of the ECM what provides a higher performance at the cost of a reduced success probability to find a prime factor of the composite integer. Hence, this design is listed for completeness but does not allow an expressive comparison.

Considering the reported work by Gaj *et al.* our design provides the significant better cost-performance ratio by a factor of 37 for ECM phase 1+2 on the same class of Virtex-4 FPGAs. This ratio is likely to be improved when using cost-optimized Spartan-6 devices instead of Virtex-4 FPGAs.

[3]Note that a static operation mode concept is possible and requires less logic, but does not work on Virtex-4 FPGAs very effectively. On Virtex-4 FPGAs, two DSP cores share one input port. This restriction does not exist for other FPGA families, such as Virtex-5 or Virtex-6.

[4]see http://em.avnet.com

		Gaj *et al.*	This work	Factor	de Meulenaer *et al.*	This work	Factor
Design	FPGA Device	V4LX200-11	V4SX35-10		V4SX25-10	V4SX35-10	
	FPGA Cost	7,564 $	468 $	0.06	298 $	468 $	1.57
	Supported Bits	198	202	1.02	135	134	0.99
	Max. ECM Cores	24	24	1.00	1	24	24
	Max. Frequency	104 MHz	200 MHz	1.92	220 MHz	200 MHz	0.91
Timing	Cycles for Addition	41	29	0.71	1	21	21
	Cycles for Multiplication	216	201	0.93	1	105	105
	Cycles for Phase 1	1,666,500	1,473,596	0.88	13,750	797,288	59.98
	Time for Phase 1	16 ms	7.37 ms	0.46	63 μs	3.99 ms	63.78
Results	# Phase 1 per Second	1,448	3,240	2.24	16,000*	6,000	0.375
	# Phase 1 per Second per 100 $	19	692	36.42	5,369*	1,282	0.239
	# Phase 1+2 per Second	696	1,560	2.24	-	2,880	-
	# Phase 1+1 per Second per 100 $	9	333	37	-	615	-

TABLE II: Comparison of our results using $b = 13$ (L = 202 bit) and $b = 9$ (L = 134 bit) with Virtex-4 LX 200 (Gaj *et al.*) and Virtex-4 SX 25 (de Meulenaer *et al.*) implementations. Note that figures with asterisk solely represent a design implementing ECM phase 1 with reduced success probability.

V. Conclusion

In this work, we presented an implementation of phase 1 and 2 of the Elliptic Curve Method for integer factorization. Our efficient design allows to run up to 24 parallel ECM cores on a single Virtex-4 SX35 FPGA and can be scaled to factor 66 to 236 bit integers without considerable hardware changes. Picking up the ideas by Šimka *et al.* and Gaj *et al.* we implemented the underlying arithmetic function using the support of DSP blocks and used additional optimizations such as an advanced memory management and improved addition chains for point multiplication.

For 151-bit integers, our implementation can provide 2,424 factorizations per second per FPGA running both phases of the ECM. Given an interface fast enough to provide the I/O data, our implementation scales to a total of 310,272 factorizations per second on a fully equipped COPACOBANA cluster populated with 128 Virtex-4 devices. Compared to the implementation by Gaj *et al.* who reported a similar design for the same Virtex-4 platform, our architecture provides a cost-performance ratio which is better by a factor of 37.

Acknowledgements

The work described in this paper has been supported by the Federal Office for Information Security (BSI), Germany, and the European Commission through the ICT program under contract ICT-2007-216676 ECRYPT II.

References

[1] H. Lenstra, "Factoring Integers with Elliptic Curves," *Annals of Mathematics*, vol. 126, pp. 649–673, 1987.

[2] J. M. Pollard, "Theorems on factorization and primality testing," in *Proceedings of the Cambridge Philosophy Society*, 1974, pp. 521–528.

[3] H. C. Williams, "A $p + 1$ method of factoring," *Mathematics of Computation*, vol. 39, pp. 225–234, 1982.

[4] A. K. Lenstra and H. W. J. Lenstra, Eds., *The Development of the Number Field Sieve*, ser. LNM. Springer-Verlag, 1993, vol. 1554.

[5] P. Zimmermann, A. Kruppa, B. Gladman, J. Papadopoulos, J. Feltin, L. Fousse, P. Gaudry, R. Cosset, and T. Kleinjung, "GMP-ECM Website," June 2010, available at https://gforge.inria.fr/projects/ecm/.

[6] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang, "ECM on graphics cards," in *Eurocrypt 2009*, ser. LNCS, vol. 5479, 2009, pp. 483–501.

[7] M. Šimka, J. Pelzl, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, V. Fischer, and C. Paar, "Hardware Factorization Based on Elliptic Curve Method," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005)*, J. Arnold and K. L. Pocek, Eds. IEEE Computer Society, April 18-20 2005, pp. 107–116.

[8] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi, "Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware," in *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2006)*, ser. LNCS, L. Goubin and M. Matsui, Eds., vol. 4249. Springer-Verlag, 2006, pp. 119–133.

[9] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, R. Bachimanchi, and M. Rogawski, "Area-time efficient implementation of the elliptic curve method of factoring in reconfigurable hardware for application in the number field sieve," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, 2009.

[10] G. de Meulenaer, F. Gosset, M. M. de Dormale, and J.-J. Quisqater, "Integer Factorization Based on Elliptic Curve Method: Towards Better Exploitation of Reconfigurable Hardware," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. IEEE Computer Society, 2007, pp. 197–206.

[11] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp, "Cryptanalysis with COPACOBANA," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1498–1513, November 2008.

[12] T. Güneysu, C. Paar, G. Pfeiffer, and M. Schimmler, "Enhancing COPACOBANA for advanced applications in cryptography and cryptanalysis," in *Proceedings of the Conference on Field Programmable Logic and Applications (FPL 2008)*, 2008, pp. 675–678.

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. New York: CRC Press, 1996.

[14] R. P. Brent, "Some Integer Factorization Algorithms Using Elliptic Curves," *Australian Computer Science Communications*, vol. 8, pp. 149–163, 1986.

[15] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987.

[16] H. M. Edwards, "A normal form for elliptic curves," *Bulletin-American Mathematical Society*, vol. 44, no. 3, p. 393, 2007, http://www.ams.org/bull/2007-44-03/S0273-0979-07-01153-6/S0273-0979-07-01153-6.pdf.

[17] P. L. Montgomery, "Modular Multiplication Without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, April 1985.

[18] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *ARITH '95: Proceedings of the 12th Symposium on Computer Arithmetic*. Washington, DC, USA: IEEE Computer Society, 1995, p. 193.