Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware

Kris Gaj[1], Soonhak Kwon[2], Patrick Baier[1], Paul Kohlbrenner[1], Hoang Le[1], Mohammed Khaleeluddin[1], and Ramakrishna Bachimanchi[1]

[1] Dept. of Electrical and Computer Engineering, George Mason University, Fairfax, Virginia 22030, USA
{kgaj, pkohlbr1, hle7, mkhaleel, rbachima}@gmu.edu, districtline@gmx.net
[2] Inst. of Basic Science, Sungkyunkwan University, Suwon 440-746, Korea
shkwon@skku.edu

Abstract. A novel portable hardware architecture for the Elliptic Curve Method of factoring, designed and optimized for application in the relation collection step of the Number Field Sieve, is described and analyzed. A comparison with an earlier proof-of-concept design by Pelzl, Šimka, et al. has been performed, and a substantial improvement has been demonstrated in terms of both the execution time and the area-time product. The ECM architecture has been ported across three different families of FPGA devices in order to select the family with the best performance to cost ratio. A timing comparison with a highly optimized software implementation, GMP-ECM, has been performed. Our results indicate that low-cost families of FPGAs, such as Xilinx Spartan 3, offer at least an order of magnitude improvement over the same generation of microprocessors in terms of the performance to cost ratio.

Keywords: Cipher-breaking, factoring, ECM, FPGA.

1 Introduction

The fastest known method for factoring large integers is the Number Field Sieve (NFS), invented by Pollard in 1991 [1,2]. It has since been improved substantially and developed from its initial "special" form (which was only used to factor numbers close to perfect powers, such as Fermat numbers) to a general purpose factoring algorithm. Using the Number Field Sieve, an RSA modulus of 663 bits was successfully factored by Bahr, Boehm, Franke and Kleinjung in May 2005 [3]. The cost of implementing the Number Field Sieve and the time it takes for such an implementation to factor a b-bit RSA modulus provide an upper bound on the security of b-bit RSA.

In order to factor a big integer N, such as an RSA modulus, NFS requires the factorization of a large number of moderately sized integers created at run time, perhaps of size 200 bits. Such numbers can be routinely factored in very little time. However, because an estimated 10^{10} such factorizations are necessary for NFS to succeed in factoring a 1024 bit RSA modulus, it is of crucial importance

L. Goubin and M. Matsui (Eds.): CHES 2006, LNCS 4249, pp. 119–133, 2006.

to perform these auxiliary factorizations as fast and efficiently as possible. Even tiny improvements, once multiplied by 10^{10} factorizations, would make a significant difference in how big an RSA modulus we can factor. The Elliptic Curve Method (ECM), which is the main subject of this paper, is a sub-exponential factoring algorithm, with expected run time of $O(\exp(c\sqrt{\log p \log\log p})\, M(N))$ where $c > 0$, p is a factor we aim to find, and $M(N)$ denotes the cost of multiplication (mod N). ECM is the best method to perform the kind of factorizations needed by NFS, for integers in the 200-bit range, with prime factors of up to about 40 bits [16,17].

The contribution of this paper is an implementation of the elliptic curve method in hardware (FPGAs). We describe in detail how to optimize the design and compare our work both to an existing software implementation (GMP-ECM)[4,5] and an earlier hardware implementation [6,7].

2 Elliptic Curve Method

2.1 ECM Algorithm

Let K be a field with characteristic different from $2, 3$. An elliptic curve can be represented by a homogeneous equation $Y^2Z = X^3 + AXZ^2 + BZ^3$ with $X, Y, Z \in K$ not all zero, where A, B are in K with $4A^3 + 27B^2 \neq 0$, together with a special point $O = (0, 1, 0)$ called a "point at infinity". Points of the curve E together with the addition operation form an abelian group which is denoted by $E(K)$, where O is the identity element of the group [8].

The Elliptic Curve Method of factoring was originally proposed by Lenstra [9] and subsequently extended by Brent [10] and Montgomery [11,12]. The original part of the algorithm proposed by Lenstra is typically referred to as Phase 1 (or Stage 1), and the extension by Brent and Montgomery is called Phase 2 (or Stage 2). The pseudocode of both phases is given below as Algorithm 1.

Algorithm 1. ECM Algorithm

Require: N: composite number to be factored, E: elliptic curve, $P_0 = (x_0, y_0, z_0) \in E(\mathbb{Z}_N)$: initial point, B_1: smoothness bound for Phase 1, B_2: smoothness bound for Phase 2, $B_2 > B_1$.
Ensure: q: factor of N, $1 < q \leq N$, or FAIL.

Phase 1.
1: $k \leftarrow \prod_{p \leq B_1} p^{\lfloor \log_p B_1 \rfloor}$
2: $Q_0 \leftarrow kP_0$
$\{Q_0 = (x_{Q_0}, y_{Q_0}, z_{Q_0})\}$
3: $q \leftarrow \gcd(z_{Q_0}, N)$
4: **if** $q > 1$ **then**
5: return q
6: **else**
7: go to Phase 2
8: **end if**

Phase 2.
9: $d \leftarrow 1$
10: **for** each prime $p = B_1$ to B_2 **do**
11: $(x_{pQ_0}, y_{pQ_0}, z_{pQ_0}) \leftarrow pQ_0$.
12: $d \leftarrow d \cdot z_{pQ_0} \pmod N$
13: **end for**
14: $q \leftarrow \gcd(d, N)$
15: **if** $q > 1$ **then**
16: return q
17: **else**
18: return FAIL
19: **end if**

2.2 Implementation Issues

An efficient algorithm for computing scalar multiplication was proposed by Montgomery [11] in 1987, and is known as the Montgomery ladder algorithm.

This algorithm is especially efficient when an elliptic curve is expressed in the Montgomery form, $E : by^2 = x^3 + ax^2 + x$. This form is obtained by a suitable change of variables [4] from the standard Weierstrass form. The corresponding expression in projective coordinates is

$$E : by^2z = x^3 + ax^2z + xz^2, \tag{1}$$

with $b(a^2 - 4) \neq 0$.

When one uses the Montgomery ladder algorithm with the Montgomery form of elliptic curve given in (1), all intermediate computations can be carried on using only x and z coordinates. As a result, we denote the starting point P_0 by $(x_0 : \ : z_0)$, intermediate points P, Q, by $(x_P : \ : z_P)$, $(x_Q : \ : z_Q)$, and the final point kP_0 by $(x_{kP_0} : \ : z_{kP_0})$. The pseudocode of the Montgomery ladder algorithm is shown below as Algorithm 2., and its basic step is defined in detail as Algorithm 3..

Algorithm 2. Montgomery Ladder Algorithm

Require: $P_0 = (x_0 : \ : z_0)$ on E with $x_0 \neq 0$, an s-bit positive integer $k = (k_{s-1}k_{s-2} \cdots k_1k_0)_2$ with $k_{s-1} = 1$
Ensure: $kP_0 = (x_{kP_0} : \ : z_{kP_0})$
1: $Q \leftarrow P_0, \quad P \leftarrow 2P_0$
2: **for** $i = s - 2$ downto 0 **do**
3: **if** $k_i = 1$ **then**
4: $Q \leftarrow P + Q, \quad P \leftarrow 2P$
5: **else**
6: $Q \leftarrow 2Q, \quad P \leftarrow P + Q$
7: **end if**
8: **end for**
9: return Q

Algorithm 3. Addition and Doubling using the Montgomery's Form of Elliptic Curve

Require: $P = (x_P : \ : z_P)$, $Q = (x_Q : \ : z_Q)$ with $x_Px_Q(x_P - x_Q) \neq 0$, $P_0 = (x_0 : \ : z_0) = (x_{P-Q} : \ : z_{P-Q}) = P - Q$, $a_{24} = \frac{a+2}{4}$, where a is a parameter of the curve E in (1)
Ensure: $P + Q = (x_{P+Q} : \ : z_{P+Q})$, $2P = (x_{2P} : \ : z_{2P})$
1: $x_{P+Q} \leftarrow z_{P-Q}[(x_P - z_P)(x_Q + z_Q) + (x_P + z_P)(x_Q - z_Q)]^2$
2: $z_{P+Q} \leftarrow x_{P-Q}[(x_P - z_P)(x_Q + z_Q) - (x_P + z_P)(x_Q - z_Q)]^2$
3: $4x_Pz_P \leftarrow (x_P + z_P)^2 - (x_P - z_P)^2$
4: $x_{2P} \leftarrow (x_P + z_P)^2(x_P - z_P)^2$
5: $z_{2P} \leftarrow (4x_Pz_P)\left((x_P - z_P)^2 + a_{24} \cdot (4x_Pz_P)\right)$

A careful analysis of formulas in Algorithm 3 indicates that point addition $P+Q$ requires 6 multiplications, and point doubling 5 multiplications. Therefore, a total of 11 multiplications are required in each step of the Montgomery ladder algorithm. In Phase 1 of ECM, the initial point, P_0, can be chosen arbitrarily. Choosing $z_0 = 1$ implies $z_{P-Q} = 1$ throughout the entire algorithm, and thus reduces the total number of multiplications from 11 to 10 per one step of the algorithm, independent of the i-th bit k_i of k. This optimization is not possible in Phase 2, where the initial point Q_0 is the result of computations in Phase 1, and thus cannot be chosen arbitrarily.

2.3 Implementation of Phase 2

Phase 1 computes one scalar multiplication kP_0, and the implementation issues are relatively easy compared with Phase 2. For Phase 2, we follow the basic idea of the standard continuation [11] and modify it appropriately for efficient FPGA implementation. Choose $2 < D < B_2$, and let every prime p, $B_1 < p \leq B_2$, be expressed in the form

$$p = mD \pm j \tag{2}$$

where m varies between $M_{MIN} = \lfloor (B_1 + \frac{D}{2})/D \rfloor$ and $M_{MAX} = \lceil (B_2 - \frac{D}{2})/D \rceil$, and j varies between 1 and $\lfloor \frac{D}{2} \rfloor$. The condition that p is prime implies that $\gcd(j, D) = 1$. Thus, possible values of j form a set $J_S = \{j : 1 \leq j \leq \lfloor \frac{D}{2} \rfloor, \gcd(j, D) = 1\}$, of the size of $\phi(D)/2$, and possible values of m form a set $M_T = \{m : M_{MIN} \leq m \leq M_{MAX}\}$, of the size $M_N = M_{MAX} - M_{MIN} + 1$, where M_N is approximately equal to $\frac{B_2 - B_1}{D}$. Then, the condition $pQ_0 = O$, implies $(mD \pm j)Q_0 = O$, and thus $mDQ_0 = \pm jQ_0$.

Writing $mDQ_0 = (x_{mDQ_0} : \ : z_{mDQ_0})$ and $jQ_0 = (x_{jQ_0} : \ : z_{jQ_0})$, the condition $mDQ_0 = \pm jQ_0 \in E(\mathbb{Z}_q)$ is satisfied if and only if $x_{mDQ_0} z_{jQ_0} - x_{jQ_0} z_{mDQ_0} \equiv 0 \pmod{q}$. Therefore existence of such pair m and j implies that one can find a factor of N by computing

$$\gcd(d, N) > 1, \quad \text{where } d = \prod_{m,j} (x_{mDQ_0} z_{jQ_0} - x_{jQ_0} z_{mDQ_0}) \tag{3}$$

In order to speed up these computations, one precomputes one of the sets $S = \{jQ_0 : j \in J_S\}$ or $T = \{mDQ_0 : m \in M_T\}$. Typically, the first of these sets, S, is smaller, and thus only this set is precomputed. One then computes the product d in the (3) for a current value of mDQ_0, and all precomputed points jQ_0, for which either $mD + j$ or $mD - j$ is prime. For each pair, (m, j), where $j \in J_S$ and $m \in M_T$, we can precompute a bit value: prime_table$[m, j] = 1$ *when* $mD + j$ *or* $mD - j$ *is prime, and* 0 *otherwise*. This table can be reused for multiple iterations of Phase 2 with the same values of B_1 and B_2, and is of the size of $M_N \cdot \phi(D)/2$ bits. Similarly, we can precompute a bit table: GCD_table$[j] = 1$ *when* $j \in J_S$, *and* 0 *otherwise*. This table will have $D/2$ bits for odd D and $D/4$ for even D (no need to reserve bits for even values of j). The exact pseudocode of the algorithm used in our implementation of Phase 2 is given in Algorithm 4.

The value B_1 is usually chosen as $B_1 \approx e^{\sqrt{\frac{1}{2} \log q \log \log q}}$ where q is unknown prime we want to find, and the value B_2 is between $50B_1$ and $100B_1$ depending on the computational resources for Phase 2. In our case, like Šimka et al. [6,7], we choose $B_1 = 960$ and $B_2 = 57000$ to find a 40-bit prime divisor of 200-bit integers. Note that one has $e^{\sqrt{\frac{1}{2} \log q \log \log q}} \approx 988$ by setting $q = 2^{41}$ which is close to 960, and the ratio B_2/B_1 is $57000/960 \approx 59$. In general, the larger values of B_1 and B_2 increase the probability of success in Phase 1 and Phase 2 respectively (and thus decrease the expected number of trials), but at the same time, increase the execution time of these phases. Values of $D = 30 = 2 \cdot 3 \cdot 5$ and $D = 210 = 2 \cdot 3 \cdot 5 \cdot 7$ are the two most natural choices for D as they

minimize the size of sets J_S and S and as a result of the amount of memory storage and computations required for Phase 2. The larger D, the larger the amount of Precomputations in Algorithm 4., but the smaller M_N, and thus the smaller number of iterations of the outer loop during Main computations in Algorithm 4.. A theoretical analysis of the optimal parameter choices is given in [19], with a view towards software implementations. The techniques developed there - which use Dickman's function to estimate the probability of success of the Elliptic Curve Method - can be adapted to a hardware setting and make it possible to determine optimal parameter choices via numerical approximations to Dickman's function. While our choices are not strictly optimal, they are fairly good and allow for direct comparsion with Šimka et al. [6,7].

Algorithm 4. Standard Continuation Algorithm of Phase 2

Require: N: number to be factored, E: elliptic curve, $Q_0 = kP_0$: initial point for Phase 2 calculated as a result of Phase 1, B_1: smoothness bound for Phase 1, B_2: smoothness bound for Phase 2, $B_2 > B_1$, D: parameter determining a trade-off between the computation time and the amount of memory required; D is assumed even in this version of the algorithm.
Ensure: q: factor of N, $1 < q \leq N$ or FAIL

Precomputations:

1: $M_{MIN} \leftarrow \lfloor (B_1 + \frac{D}{2})/D \rfloor$
2: $M_{MAX} \leftarrow \lceil (B_2 - \frac{D}{2})/D \rceil$
3: clear GCD_table, clear J_S
4: **for** each $j = 1$ to $\frac{D}{2}$ step 2 **do**
5: **if** $\gcd(j, D) = 1$ **then**
6: GCD_table$[j] = 1$
7: add j to J_S
8: **end if**
9: **end for**
10: clear prime_table
11: **for** each $m = M_{MIN}$ to M_{MAX} **do**
12: **for** each $j = 1$ to $\frac{D}{2}$ step 2 **do**
13: **if** ($mD + j$ or $mD - j$ is prime) **then**
14: prime_table$[m, j] = 1$
15: **end if**
16: **end for**
17: **end for**
18: $Q \leftarrow Q_0$
19: **for** $j = 1$ to $\frac{D}{2}$ step 2 **do**
20: **if** GCD_table$[j] = 1$ **then**
21: store Q in S
$\{Q = jQ_0 = (x_{jQ_0} : \ : z_{jQ_0})\}$
22: **end if**
23: $Q \leftarrow Q + 2Q_0$
24: **end for**

Main computations:

25: $d \leftarrow 1,\ Q \leftarrow DQ_0,\ R \leftarrow M_{MIN}Q$
26: **for** each $m = M_{MIN}$ to M_{MAX} **do**
27: **for** each $j \in J_S$ **do**
28: **if** prime_table$[m, j] = 1$ **then**
29: retrieve jQ_0 from table S
30: $d \leftarrow d \cdot (x_R z_{jQ_0} - x_{jQ_0} z_R)$
$\{R = (x_R : \ : z_R)\}$
31: **end if**
32: **end for**
33: $R \leftarrow R + Q$
34: **end for**
35: $q \leftarrow \gcd(d, N)$
36: **if** $q > 1$ **then**
37: return q
38: **else**
39: return FAIL
40: **end if**

3 ECM Architecture

3.1 Top-Level View: ECM Units

Our ECM system consists of multiple ECM units working independently in parallel, as shown in Figure 1. Each unit performs the entire ECM algorithm for one number N, one curve E and one initial point P_0. All units share the same global control unit and the same global memory. All components of the system are located on the same integrated circuit, either an FPGA or an ASIC, depending on the choice of an implementation technology. The exact number of ECM

units per integrated circuit depends on the amount of resources available in the given integrated circuit. Multiple integrated circuits may work independently in parallel, on factoring a single number, or factoring different numbers. All integrated circuits are connected to a central host computer, which distributes tasks among the individual ECM systems, and collects and interprets results.

The operation of the system starts by loading all parameters required for Phase 1 of ECM from the host computer to the global memory on the chip. These parameters include:

1. Number to be factored, N, coordinates of the starting point P_0, and the parameter a_{24} dependent on the coefficient a of the curve E - all of which can be separate for each ECM unit.
2. Integer k, used as an input in the ECM Phase 1 (see Algorithm 1.), its size k_N, and the parameter $n = \lfloor \log_2 N_{MAX} \rfloor + 2$, related to the size of the largest N, N_{MAX}, processed by the ECM units - all of which are common for all ECM units.

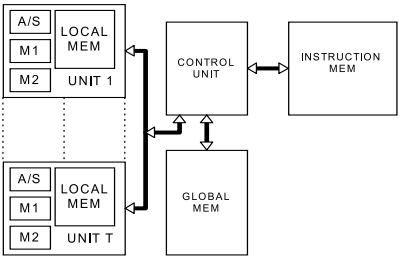

Fig. 1. Block diagram of the top-level unit. Notation: MEM-memory; $M1$, $M2$-multipliers 1 and 2; A/S adder/subtractor.

In the next step, N, the coordinates of P_0, and the parameters a_{24} and n are loaded to the local memories of the respective ECM units, and the operation of these units is started. All units operate synchronously, on different data sets, performing all intermediate calculations exactly at the same time.

The results of these calculations are the coordinates x_{Q_0} and z_{Q_0} of the ending point $Q_0 = kP_0$, separate for each ECM unit. These coordinates are downloaded to the host computer, which performs the final calculations of Phase 1, $q = \gcd(z_{Q_0}, N)$.

If no factor of N was found, the ECM system is ready for Phase 2. The values of N, the parameters a_{24} and n, and the coordinates of the points Q_0 obtained as a result of Phase 1 are already in the local memories of each ECM unit. The host computer calculates and downloads to the global memory of the ECM system the following parameters dependent on B_2 and D: M_{MIN}, M_N, GCD_table, and prime_table, defined in Section 2.3. The Phase 2 is then started simultaneously in all ECM units, and produces as final results, the accumulated product d (see (3)). These final results are then downloaded to the host computer, where the final calculations $q = \gcd(d, N)$ are performed.

Note that with this top level organization, there is no need to compute greatest common divisor or division in hardware. Additionally, the overhead associated with the transfer of data between the ECM system and the host computer, and the time of computations in software are both typically insignificant compared to the time used for ECM computations in hardware, even in the case of a relatively slow interface and/or a slow microprocessor.

3.2 Medium-Level View: Operations of the ECM Unit

Medium-Level Operations. The primary operation constituting Phase 1 of ECM is a scalar multiplication $Q_0 = kP_0$. As discussed in Section 2.2, this operation can be efficiently implemented in projective coordinates using Algorithm 2.

In Phase 1, one coordinate of P_0 can be chosen arbitrarily, and therefore the computations can be simplified by selecting $z_{P_0} = z_{P-Q} = 1$. The remaining computations necessary to simultaneously compute $P + Q$ and $2P$ can be interleaved, and assigned to three functional units working in parallel, as shown in Table 1. The entire step of a scalar multiplication, including both point addition and doubling can be calculated in the amount of time required for 2 modular additions/subtractions and 5 modular multiplications. Please note that because the time of an addition/subtraction is much shorter than the time of a multiplication, two *sequential* additions/subtractions can be calculated in parallel with two multiplications.

Table 1. One step of a scalar multiplication, including the concurrent operations $P+Q$ and $2P$, for the case of $z_{P-Q} = 1$. Notation: A: operations used for addition only, D: operations used for doubling only, A/D: operations used for addition and doubling.

Adder/Subtractor	Multiplier 1	Multiplier 2
A/D: $a_1 = x_P + z_P$ $s_1 = x_P - z_P$		
A/D: $a_2 = x_Q + z_Q$ $s_2 = x_Q - z_Q$	D: $m_1 = s_1^2$	D: $m_2 = a_1^2$
D: $s_3 = m_2 - m_1$	A: $m_3 = s_1 \cdot a_2$	A: $m_4 = s_2 \cdot a_1$
A: $a_3 = m_3 + m_4$ $s_4 = m_3 - m_4$	D: $x_{2P} = m_5 = m_1 \cdot m_2$	D: $m_6 = s_3 \cdot a_{24}$
D: $a_4 = m_1 + m_6$	A: $x_{P+Q} = m_7 = a_3^2$	A: $m_8 = s_4^2$
	A: $z_{P+Q} = m_9 = m_8 \cdot x_{P-Q}$	D: $z_{2P} = m_{10} = s_3 \cdot a_4$

The storage used for temporary variables $a_1, \ldots, a_4, s_1, \ldots, s_4$, and $m_1, \ldots, m_{10}$ can be reused whenever any intermediate values are no longer needed. With the appropriate optimization, the amount of local memory required for Phase 1 has been reduced to 11 256-bit operands, i.e., 88 32-bit words. The remaining portion of this memory is used in Phase 2 of ECM.

In Phase 2, the initial computation

$$D \cdot Q_0 \quad \text{and} \quad M_{MIN} \cdot (D \cdot Q_0)$$

can be performed using an algorithm similar to the one used in Phase 1. The only difference is that now, $P - Q = Q_0$ cannot be chosen arbitrarily, and thus,

$z_{P-Q} = z_{Q_0} \neq 1$. As a result, the computations will take the amount of time required for 2 modular additions/subtractions and 6 modular multiplications.

The second type of operation required in Phase 2 is a simple point addition $P + Q$. This operation can be performed in the time of 6 additions/subtractions and 3 modular multiplications.

Finally, the last medium level operation required in Phase 2 is the accumulation of the product d, as defined in (3). We can rewrite the expression for d as

$$d \equiv \prod_{i,n} d_{in} \equiv \prod_{i,n} (x_n z_i - x_i z_n) \pmod{N} \quad (4)$$

where, $(x_i, z_i) \in \{(x, z) \colon (x \colon : z) = jQ_0\}$, $(x_n, z_n) \in \{(x, z) \colon (x \colon : z) = mDQ_0\}$ and GCD_table[j]=1 and prime_table[m, j]=1. In Table 2, we show how these operations can be distributed in an optimum way among three arithmetic units working in parallel. As shown in Table 2, after the initial delay of one multiplication, the time required to compute and accumulate any *two* subsequent values of a partial product $x_{mDQ_0} z_{jQ_0} - x_{jQ_0} z_{mDQ_0}$ is equal to the time of three multiplications.

Table 2. Accumulation of the partial results $\prod_{i,n} (x_n z_i - x_i z_n) \pmod{N}$ in Phase 2 (for fixed n and moving i)

Adder/Subtractor	Multiplier 1	Multiplier 2
	$m_1 = x_n \cdot z_0$	$m_2 = x_0 \cdot z_n$
$d_{0n} = m_1 - m_2$	$m_3 = x_n \cdot z_1$	$m_4 = x_1 \cdot z_n$
$d_{1n} = m_3 - m_4$	$d = d \cdot d_{0n}$	$m_1 = x_n \cdot z_2$
	$d = d \cdot d_{1n}$	$m_2 = x_2 \cdot z_n$
$d_{2n} = m_1 - m_2$	$m_3 = x_n \cdot z_3$	$m_4 = x_3 \cdot z_n$
$d_{3n} = m_3 - m_4$	$d = d \cdot d_{2n}$	$m_1 = x_n \cdot z_4$
	$d = d \cdot d_{3n}$	$m_2 = x_4 \cdot z_n$
.....	.....	.....

Instructions of the ECM Unit. Each ECM unit is composed of two modular multipliers, one adder/subtractor, and one local memory. The local memory size is 512 32-bit words, equivalent to 64 256-bit registers. In Phase 1, only 11 out of 64 256-bit registers are in use. In Phase 2, with $D = 210$, the entire memory is occupied.

Every ECM unit forms a simple processor with its own instruction set. Since all ECM units perform exactly the same instructions at the same time, the instructions are stored in the global instruction memory, and are interpreted using the global control unit, as shown in Figure 1. Three sequences of ECM instructions describe three kinds of medium-level operations:

1. One step of a scalar multiplication kP ($P = 2P$, $Q = P + Q$) in Phase 1, i.e., with $z_{P_0} = 1$ (see Table 1).
2. One step of a scalar multiplication kP ($P = 2P$, $Q = P + Q$) in Phase 2, i.e., with $z_{P_0} \neq 1$.
3. Addition $P + Q$ in Phase 2, i.e., with $z_{P_0} \neq 1$.

Since only 11 256-bit registers are necessary to perform each of the sequences of instructions given above, only 4 bits are required to encode each input/output address.

The operation performed by each instruction is determined based on the position of the instruction in the instruction sequence, and thus does not need to be encoded in the instruction body. In particular, a group of four instructions corresponds to one row of Table 1 and is stored in the order: Multiplication 2, Multiplication 1, Subtraction, and Addition. These four consecutive instructions are fetched serially, but executed in parallel. The processor progresses to the next group of four instructions only when all instructions of the previous group have been completed. If the given arithmetic unit should remain inactive in the given sequence of four instructions, this inactivity is described using the zero value of a special flag in the body of the respective instruction.

3.3 Low-Level View: Modular Multiplication and Addition/Subtraction

The three low level operations implemented by the ECM unit are Montgomery modular multiplication [13], modular addition, and modular subtraction. Modular addition and subtraction are very similar to each other, and as a result they are implemented using one functional unit, the adder/subtractor. For 256-bit operands, both addition and subtraction take 41 clock cycles.

In order to simplify our Montgomery multiplier, all operations are performed on inputs X, Y in the range $0 \leq X, Y < 2N$, and return an output S in the same range, $0 \leq S < 2N$. This is equivalent to computing all intermediate results modulo $2N$ instead of N, which increases the size of all intermediate values by one bit, but shortens the time of computations, and leads to exactly the same final results as operations $\bmod N$.

In our implementation we have adopted the Radix-2 Multiplier Algorithm with Carry Save Addition, reported earlier in [14]. With this algorithm applied, the total execution time of a single Montgomery multiplication is equal to $n+16$ clock cycles. For a typical use within ECM, n is greater than 100, and thus one addition followed by one subtraction can easily execute in the amount of time smaller than the time of a single Montgomery multiplication.

4 Implementation Results

Our ECM system has been developed entirely in RTL-level VHDL, and written in a way that provides portability among multiple families of FPGA devices and standard-cell ASIC libraries. In the case of FPGAs, the code has been synthesized using Synplicity Synplify Pro v. 8.0, and implemented on FPGAs using Xilinx ISE v. 6.3, 7.1 and 8.1. Three different families of FPGA devices have been targeted, including high-performance families, Virtex E and Virtex II, as well as a low-cost family, Spartan 3. The design has been debugged and verified using a test program written in C, and using GMP-ECM [4,5].

Table 3. Execution time of Phase 1 and Phase 2 in the ECM hardware architecture for 198-bit numbers $N, B_1 = 960$ (which implies number of bits of $k, k_N = 1375$), $B_2 = 57000$, and $D = 30$ or $D = 210$

Operation	Notation	Formula	# clk cycles $D = 30$	# clk cycles $D = 210$
Elementary operations				
Modular addition	T_A		**41**	
Montgomery multiplication	T_M	$T_M = n + 16$	**216**	
Point addition and doubling (Phase 1)	T_{AD1}	$T_{AD1} = 5T_M + 2T_A + 50$	**1212**	
Point addition and doubling (Phase 2)	T_{AD2}	$T_{AD2} = 6T_M + 2T_A + 50$	**1428**	
Point addition (Phase 2) (Phase 2)	T_{ADD2}	$T_{ADD2} = 3T_M + 6T_A + 30$	**924**	
Phase 1				
Phase 1 (estimation)	$T_{P1\,est}$	$T_{P1} \approx k_N \cdot T_{AD1}$	**1,666,500**	
Phase 1 (simulation)	$T_{P1\,sim}$		**1,713,576**	
Phase 2				
Precalculating jQ_0	T_{jQ}	$T_{jQ} \approx 2T_{AD2}$ $+(\lfloor D/4 \rfloor - 2)T_{ADD2}$	**7476** (**0.19%**)	**49,056** (**2.56%**)
DQ_0	T_{DQ}	$T_{DQ} \approx \lceil log_2(D+1) \rceil T_{AD2}$	**7140** (**0.18%**)	**11,424** (**0.60%**)
$M_{MIN}DQ_0$	$T_{M_{min}DQ}$	$T_{M_{min}DQ} \approx$ $\lceil log_2(M_{MIN}+1) \rceil T_{AD2}$	**8568** (**0.22%**)	**4284** (**0.22%**)
Calculating mDQ_0 **for** $M_{MIN} < m \leq M_{MAX}$	T_{mDQ}	$T_{mDQ} \approx (M_N - 2)T_{ADD2}$	**1,725,108** (**44.29%**)	**244,860** (**12.78%**)
Number of ones in the prime_table	$n_{\text{prime\_table}}$		**4531**	**4361**
Calculating accumulated product d	T_d	$T_d \approx \lceil 1.5 \cdot n_{\text{prime\_table}} \rceil (T_M + 12)$ $+M_N(T_M + T_A)/2$	**1,789,883** (**45.95%**)	**1,525,886** (**79.67%**)
Phase 2 (estimation)	$T_{P2\,est}$	$T_{P2} \approx T_{jQ} + T_{DQ} + T_{M_{min}DQ}$ $+T_{mDQ} + T_d$	**3,538,175** (**90.84%**)	**1,835,510** (**95.84%**)
Phase 2 (simulation)	$T_{P2\,sim}$		**3,895,013** (**100%**)	**1,915,219** (**100%**)

The execution times of Phase 1 and Phase 2 in the ECM hardware architecture are shown in Table 3. The generic formulas for major component operations are provided, together with the estimated values of the execution times for the case of 198-bit numbers N, and the smoothness bounds $B_1 = 960$ and $B_2 = 57000$. The estimated values are compared with the accurate values obtained from simulation. The difference is less than 10%, and can be attributed to the time needed for control operations and data movements within local memories, and between global memory and local memories. Two values of the parameter D are considered for Phase 2, $D = 30$ and $D = 210$. The table proves that the choice of the parameter $D = 210$, reduces the execution time of Phase 2 in our architecture by a factor of two compared to the case of $D = 30$. As confirmed by exhaustive search, the choice of $D = 210$ results in the smallest possible execution time for Phase 2 for the given values of the smoothness bounds

$B_1 = 960$ and $B_2 = 57000$, assuming execution times of basic operations given in Table 3. For $D = 210$, the largest contribution to Phase 2, around 80%, comes from the calculation of the accumulated product d.

In order to estimate an overhead associated with the transfer of control and data between a microprocessor and an FPGA, the ECM system with 10 ECM units has been ported to a reconfigurable computer SRC 6 from SRC Computers [18], based on 2.8 GHZ Xeon microprocessors and Xilinx Virtex II XC2V6000-6 FPGAs running at a fixed clock frequency of 100 MHz. The data and control transfer overheads have been experimentally measured to be less than 4% of the end-to-end execution time for the combined Phase 1 and Phase 2 calculations.

In Table 4, we compare our ECM architecture to an earlier design by Pelzl, Šimka, et al., presented at SHARCS 2005, and described in subsequent publications [6,7]. Every possible effort was made to make this comparison as fair as possible. In particular, we use an identical FPGA device, Virtex 2000E-6. We also do not take into account any limitations imposed by an external microcontroller used in the Pelzl/Šimka architecture. Instead, we assume that the system could be redesigned to include an on-chip controller, and it would operate with the maximum possible speed reported by the authors for their ALUs [6,7], i.e., 38 MHz (clock period = 26.3 ns). We also ignore a substantial input/output overhead reported by the authors, and caused most likely by the use of an external microcontroller.

In spite of these equalizing measures, our design outperforms the design by Pelzl, Šimka, et al. by a factor of 9.3 in terms of the execution time for Phase 1, by a factor of 7.4 in terms of the execution time for Phase 2 with the same value of parameter D, and by a factor of 15.0 for Phase 2 with the increased value of $D = 210$, not reported by Pelzl/Šimka. The main improvements in Phase 1 come from the more efficient design for a Montgomery multiplier (a factor of 5 improvement) and from the use of two Montgomery multipliers working in parallel (a factor of 1.9 improvement). An additional smaller factor is the ability of an adder/subtractor to work in parallel with both multipliers, as well as, the higher clock frequency.

One might expect that such improvement in speed comes at the cost of substantial sacrifices in terms of the circuit area and cost. In fact, our architecture is bigger, but only by a factor of 2.7 in terms of the number of CLB slices. Additionally, the design reported in [6,7] has a number of ECM units per FPGA device limited not by the number of CLB slices, but by the number of internal on-chip block RAMs (BRAMs). If this constraint was not removed, our design would outperform the design by Pelzl/Šimka in terms of the amount of computations per Xilinx Virtex 2000E device by a factor of $9.3 \cdot 2.33 = 22$ for Phase 1 and 35 for Phase 2. If the memory constraint is removed, the product of time by area still improves compared to the design by Pelzl and Šimka by a factor of $9.3/2.7 = 3.4$ for Phase 1 and 5.6 for Phase 2.

In Table 5, we show the results of porting our design to three families of Xilinx FPGAs. For each family, a representative device is selected and used in our implementation. For each device we determine the exact amount of resources

Table 4. Comparison with the design by Pelzl, Šimka, et al., both implemented using Virtex 2000E-6

Part 1: Execution Time						
	Pelzl, Šimka, et al.		**Our design**		**Ratio Pelzl, Šimka / ours**	
	# clk cycles	**Time**	**# clk cycles**	**Time**	**# clk cycles**	**Time**
Clock period		26.3 ns		18.5 ns		
Modular addition	16	0.62 μs	41	0.78 μs	0.6	0.8
Modular subtraction	24	0.42 μs	41	0.78 μs	0.4	0.5
Montgomery multiplication	796	20.7 μs	216	4.1 μs	3.7	**5.0**
Point addition & doubling (Phase 1)	8200	213.2 μs	1212	23.0 μs	6.8	**9.3**
Phase 1	**11,266,800**	**292.9** ms	**1,713,576**	**31.7** ms	**6.6**	**9.3**
Point addition & doubling (Phase 2)	8998	233.9 μs	1428	27.1 μs	5.6	8.6
Point addition (Phase 2)	4920	127.9 μs	924	17.6 μs	4.8	**7.3**
Calculation and accumulation of two values of d_{in} (Phase 2)	4776	124.2 μs	648	12.3 μs	6.2	**10.1**
Phase 2 ($D = 30$)	**20,276,060**	**527.2** ms	**3,895,013**	**72.1** ms	**5.2**	**7.4**
Phase 2 ($D = 210$)	-	-	**1,915,219**	**35.5** ms	**10.6**	**15.0**

Part 2: Resource usage per one ECM unit					
	Pelzl, Šimka, et al.		**Our design ($D = 210$)**		**Ratio Ours / Pelzl, Šimka**
Number of	#	%	#	%	
CLB slices	N/A	6.0	3102	16	**2.7**
LUTs	1754	4.5	4933	13	2.8
FFs	506	1.25	3129	8	6.2
BRAMs	44	27	2	1.25	**0.045**
Maximum number of ECM units per chip	3 (limited by BRAMs)		7 (limited by CLB slices)		**2.33**

needed per single ECM unit, the maximum number of ECM units per chip, the maximum clock frequency, and then the maximum amount of ECM computations (Phase 1 and Phase 2) per unit of time. Finally, we normalize the performance by dividing it by the cost of a respective FPGA device. From the last row in the table one can see that the low-cost FPGA devices from the Spartan 3 family outperform the high-performance Virtex II devices by a factor of 16, and thus are more suitable for cost effective code breaking computations.

In Table 6, we compare the execution time of Phase 1 and Phase 2 between the two representative FPGA devices and a highly optimized software implementation (GMP-ECM) running on Pentium 4 Xeon, 2.8 GHz. GMP-ECM is one of the most powerful software implementations of ECM and contains multiple optimization techniques for both Phase 1 and Phase 2 [4,5]. Additionally, we run our own test program in C that mimics almost exactly the behavior of

Table 5. Results of the FPGA implementations (resources and timing for one ECM unit per FPGA device, execution time of Phase 1 and Phase 2 for 198-bit numbers $N, B_1 = 960$, $B_2 = 57000$, $D = 210$)

Results	Virtex XCV2000E-6	Virtex II XC2V6000-6	Spartan 3 XC3S5000-5
Resources for one ECM unit			
- CLB slices	**3102 (16%)**	**3197 (9%)**	**3322 (10%)**
- LUTs	**4933 (13%)**	**5025 (7%)**	**5134 (8%)**
- FFs	**3129 (8%)**	**3102 (5%)**	**3130 (5%)**
- BRAMs	**2/160**	**2/144**	**2/104**
Maximum number of ECM units per FPGA device	**7**	**10**	**10**
Technology	0.15/0.12 μm	0.15/0.12 μm	90 nm
Cost of an FPGA device[a]	**$1230**	**$2700**	**$130**
Maximum clock frequency for one ECM unit	**54 MHz**	**123 MHz**	**100 MHz**
Time for Phase 1 and Phase 2	**67.2** ms	**29.5** ms	**36.3** ms
# of ECM computations per second with the maximum number of ECM units	**104**	**339**	**276**
# of ECM computations per second per $100 with the maximum number of ECM units	**8**	**13**	**212**

[a] Approximate cost per unit for a batch of 10,000+ devices

Table 6. Comparison of the execution time between 2.8 GHz Xeon Pentium 4 (w/512KB cache) and two types of FPGA devices Virtex II XC2V6000-6 and Spartan 3 XC3S5000-5 (198-bit number $N, B_1 = 960$, $B_2 = 57000$, $D = 210$, maximum number of ECM units per FPGA device)

	Virtex II XC2V6000-6	Spartan 3 XC3S5000-5	Pentium 4 (testing program)	Pentium 4 (GMP-ECM)
Clock frequency	**123 MHz**	**100 MHz**	**2.8 GHz**	
No. of parallel ECM computations	**10**	**10**	**1**	
Time of Phase 1	**13.9** ms	**17.1** ms	**18.3** ms	**11.3** ms
Time of Phase 2	**15.6** ms	**19.2** ms	**18.6** ms	**13.5** ms
Time of Phase 1 & Phase 2	**29.5** ms	**36.3** ms	**36.9** ms	**24.8** ms
# of Phase 1 computations per second	**718**	**584**	**55**	**89**
# of Phase 2 computations per second	**642**	**522**	**54**	**74**
# of Phase 1 & 2 computations per second	**339**	**276**	**27**	**40**

hardware, except for using calls to the multiprecision GMP library for all low level operations, such as modular multiplication and addition. One can see that the algorithmic optimizations used in GMP-ECM matter, and reduce the overall execution time for Phase 1 from 18.3 ms to 11.3 ms (38%), and Phase 2 from 18.6 ms to 13.5 ms (27%).

Interestingly, the execution time for an ECM unit running on Virtex II, 6000E is only slightly greater than the execution time of GMP-ECM on a Pentium

4 Xeon. At the same time, since this FPGA device can hold up to 10 ECM units, its overall performance is about 8.5 times higher for combined Phase 1 and Phase 2 computations. However, the current generation of high-end FPGA devices cost about 10 times as much as comparable microprocessors. Therefore, the advantage of Virtex II over Pentium 4 disappears when cost is taken into account. In order to get an advantage in terms of the performance to cost ratio, one must use a low-cost FPGA family, such as Xilinx Spartan 3. In this case, the ratio of the amount of computations per chip is about 7 in favor of the biggest Spartan 3. Additionally this device is actually cheaper than the state-of-the-art microprocessor, so the overall improvement in terms of the performance to cost ratio exceeds a factor of 10.

5 Conclusions

A novel hardware architecture for the Elliptic Curve Method of factoring has been proposed. The main differences as compared to an earlier design by Pelzl, Šimka, et al. [6,7] include the use of an on-chip optimized controller for Phase 1 and Phase 2 (in place of an external controller based on an ARM processor), substantially smaller memory requirements, an optimized architecture for the Montgomery multiplier, the use of two (instead of one) multipliers, and the ability of all arithmetic units (two multipliers and one adder/subtractor) to work in parallel. When implemented on the same Virtex 2000E-6 device, our architecture has demonstrated a speed-up by a factor of 9.3 for ECM Phase 1 and 15.0 for ECM Phase 2, compared to the design by Pelzl/Šimka, et al. At the same time, memory requirements have been reduced by a factor of 22, and the requirements for CLB slices have increased by a factor of 2.7. If the same optimizations regarding the memory usage and the use of an internal controller were applied to the design by Pelzl/Šimka, our architecture would still retain an advantage in terms of the performance to cost ratio by a factor of 3.4 for Phase 1 and 5.6 for Phase 2.

Our architecture has been implemented targeting two additional families of FPGA devices, Virtex II and Spartan 3. Our analysis revealed that the low-cost Spartan 3 devices outperformed the high-performance Virtex II devices in terms of the performance to cost ratio by a factor of about 16.

We have also compared the performance of our hardware architecture implemented using Virtex II XC2V6000-6 and Spartan 3 XC3S5000-5 with the optimized software implementation running on Pentium 4 Xeon, with a 2.8 GHz clock. Our analysis shows that the high performance FPGA device outperforms the same generation microprocessor by a factor of about 8.5, but looses its advantage when the cost of both devices is taken into account. On the other hand, the low-cost FPGA device Spartan 3 achieves about an order of magnitude advantage over the same generation Pentium 4 processor in terms of both performance and performance to cost ratio. This feature makes low-cost FPGA devices an appropriate basic building block for cost-optimized hardware for breaking cryptographic systems, which is consistent with the conclusions of other research groups reported earlier in the literature [15].

References

1. J.M. Pollard, "Factoring with cubic integers", *Lecture Notes in Mathematics 1554*, pp. 4-10, Springer, 1993.
2. A.K. Lenstra and H.W. Lenstra, *The Development of the Number Field Sieve*, *Lecture Notes in Mathematics 1554*, Springer, 1993.
3. Factorization of RSA-200, F. Bahr, M. Boehm, J. Franke, T. Kleinjung, http: crypto-world.com/announcements/rsa200.txt.
4. P. Zimmermann, "20 years of ECM," *preprint*, 2005, http://www.loria.fr/∼ zimmerma/papers/ecm-submitted.pdf.
5. J. Fougeron, L. Fousse, A. Kruppa, D. Newman, and P. Zimmermann, "GMP-ECM", http://www.komite.net/laurent/soft/ecm/ecm-6.0.1.html, 2005.
6. M. Šimka, J. Pelzl, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovsky, V. Fischer, and C. Paar, "Hardware factorization based elliptic curve method", *IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM'05*, Napa, CA, USA, 2005.
7. J. Pelzl, M. Šimka, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovsky, V. Fischer, and C. Paar, "Area-time efficient hardware architecture for factoring integers with the elliptic curve method", *IEE Proceedings on Information Security,* vol . 152, no. 1, pp. 67-78, 2005.
8. D. Hankerson, A.J. Menezes, and S.A. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer–Verlag, 2004.
9. H.W. Lenstra, "Factoring integers with elliptic curves", *Annals of Mathematics*, vol. 126, pp. 649–673, 1987.
10. R.P. Brent, "Some integer factorization algorithms using elliptic curves", *Australian Computer Science Communications*, vol. 8, pp. 149–163, 1986.
11. P.L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization", *Mathematics of Computation*, vol. 48, pp. 243–264, 1987.
12. P.L. Montgomery, "An FFT extension of the elliptic curve method of factorization", *Ph.D. Thesis*, UCLA, 1992.
13. P.L. Montgomery, "Modular multiplication without trivial division", *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.
14. C. McIvor, M. McLoone, J. McCanny, A. Daly, and W. Marnane, "Fast Montgomery modular multiplication and RSA cryptographic processor architectures", *Proc. 37th IEEE Computer Society Asilomar Conference on Signals, Systems and Computers*, Monterey, USA, pp. 379-384, Nov. 2003.
15. S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, A. Rupp, M. Schimmler, "How to break DES for 8,980 Euro", *2nd Workshop on Special-purpose Hardware for Attacking Cryptographic Systems - SHARCS 2006*, Cologne, Germany, April 3-4, 2006.
16. J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, and C. Stahlke, "SHARK : A realizable special hardware sieving device for factoring 1024-bit integers", *Cryptographic Hardware and Embedded Systems - CHES 05*, LNCS 3659, pp. 119–130, Springer-Verlag, 2005.
17. W. Geiselmann, F Januszewski, H Koepfer, J. Pelzl, and R. Steinwandt, "A simpler sieving device: Combining ECM and TWIRL", *Cryptology ePrint Archive*, http://eprint.iacr.org/2006/109.
18. SRC Computers, Inc., http://www.srccomp.com.
19. R.D. Silverman and S.S. Wagstaff,"A practical analysis of the elliptic curve factoring algorithm", *Mathematics of Computation,* vol. 61, no. 203, pp. 465-462, 1993.