Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware

Kris Gaj, Soonhak Kwon, Patrick Baier,
Paul Kohlbrenner, Hoang Le, Mohammed Khaleeluddin, Ramakrishna Bachimanchi

George Mason University

{kgaj, skwon1}@gmu.edu,
districtline@gmx.net,
{pkohlbr1, hle7, mkhaleel, rbachima}@gmu.edu

ABSTRACT

A novel portable hardware architecture of 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, Simka, 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 five different families of FPGA devices in order to select the family with the best performance to cost ratio. A timing comparison with the highly optimized software implementation, GMP-ECM, has been performed. Our results indicate that low-cost families of FPGAs, such as Spartan-3 and Spartan-3E, offer at least an order of magnitude improvement over the same generation of microprocessors in terms of the performance to cost ratio.

1. INTRODUCTION

The fastest known method for factoring large integers is the Number Field Sieve (NFS), invented by Pollard in 1991 [14]. 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. 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 during 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 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.

We therefore review existing algorithms which can be used to factor medium-size numbers. The only practically useful algorithms are probabilistic (Monte-Carlo) methods. There is no guarantee that a probabilistic algorithm will terminate successfully, but the probability of a successful outcome is large enough that the expected time needed to factor a given number is considerably lower than that of any deterministic algorithm. In particular, all known deterministic factoring methods have exponential asymptotic run time. In practice, they are at best used to remove the smallest prime factors from the number to be factored.

Trial division by at most a few hundred small primes with asymptotic run-time $O(N^{1/2+\varepsilon})$ (with ε covering the cost of integer arithmetic as N increases) may be considered as a first step in factoring random numbers. The fastest known deterministic factoring method is due to Pollard and Strassen and has an asymptotic run time $O(N^{1/4+\varepsilon})$, but is not recommended in practical applications because it is easily surpassed by simple probabilistic methods.

Three other probabilistic factoring methods are also of exponential run time, but with a much smaller overhead than their sub-exponential colleagues, so that within a certain range they are efficient factoring tools. These are Pollard's $p-1$ method, the similar $p+1$ method due to Williams, and Pollard's ρ-method.

Finally, 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 $\pmod N$. 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.

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 hardware implementation [16, 18] and a software implementation (GMP-ECM) [7, 15].

2. ELLIPTIC CURVE METHOD

Let K be a field with characteristic different from $2, 3$. For example, $K = \mathbb{Z}_q$ with a prime $q > 3$, which is a set of integers $\{0, 1, \ldots, q-1\}$ with addition and multiplication $\pmod q$. An elliptic curve E over K is defined as a set of points $(X, Y) \in K^2$ satisfying

$$Y^2 = X^3 + AX + B, \quad \text{where } A, B \in K,\ 4A^3 + 27B^2 \neq 0, \tag{1}$$

together with a special point called "the point at infinity" and denoted O. Two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ can be added together to give a third point $R = P + Q = (x_R, y_R)$, where $x_R = f_1(x_P, y_P, x_Q, y_Q)$ and $y_R = f_2(x_P, y_P, x_Q, y_Q)$ for some K-rational functions f_1 and f_2. The point at infinity, O, is an identity element of this operation, i.e., $P + O = P = O + P$. Points of the curve E (including the point at infinity) together with aforementioned addition form a group, which is denoted by $E(K)$. The representation of elliptic curve points using two coordinates $P = (x_P, y_P)$ is called the affine representation.

In order to increase the computational efficiency of point addition, one may prefer the representation of E in homogeneous (projective) coordinates of E,

$$Y^2Z = X^3 + AXZ^2 + BZ^3.$$

With this change, (X, Y, Z) with $Z \neq 0$ represents $(\frac{X}{Z}, \frac{Y}{Z})$ in affine coordinates. If $Z = 0$, then we have the point at infinity O which is represented by $(0, 1, 0)$ in projective coordinates.

Montgomery [2] studied elliptic curves of the form, $E : by^2 = x^3 + ax^2 + x$, to further speed up elliptic curve operations in software and hardware. This form is obtained by the change of variables, $X = \frac{3x+a}{3b}, Y = \frac{y}{b}, A = \frac{3-a^2}{3b^2}, B = \frac{2a^3-9a}{27b^3}$, from Eq. 1. The corresponding expression in projective coordinates is

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

with $b(a^2 - 4) \neq 0$. Using the above form of elliptic curves, Montgomery derived an addition formula for P and Q which does not need any y-coordinate information, assuming that the difference $P - Q$ is already known.

Let N be a composite integer we want to factor. The ECM Method [2, 5, 7] considers elliptic curves in Montgomery form, given in Eq. 2, and involves elliptic curve operations $\pmod N$, where the elements in $\mathbb{Z}$ are reduced $\pmod N$. Since N is not a prime, E over $\mathbb{Z}_N$ is not really an elliptic curve but we can still do point additions and doublings as if $\mathbb{Z}_N$ was a field.

2.1. ECM Algorithm

The Elliptic Curve Method (ECM) was originally proposed by Lenstra [4] and subsequently extended by Brent [5] and Montgomery [2]. 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.

Let q be an unknown factor of N. Then the order of the curve E, $|E(\mathbb{Z}_q)|$, i.e., the number of points on the curve E with operations performed $\pmod q$, might be a smooth number that divides k. In that case,

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.

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

Phase 2.

$d \leftarrow 1$.
for each prime $p = B_1$ to B_2 **do**
  $(x_{pQ_0}, y_{pQ_0}, z_{pQ_0}) \leftarrow pQ_0$.
  $d \leftarrow d \cdot z_{pQ_0} \pmod N$
end for
$q \leftarrow gcd(d, N)$
if $q > 1$ **then**
  return q
else
  return FAIL
end if

we have $k = l \cdot |E(\mathbb{Z}_q)|$ for some l. For any point P_0 belonging to the curve E, $|E(\mathbb{Z}_q)|P_0 = O$, therefore $kP_0 = l \cdot |E(\mathbb{Z}_q)|P_0 = O$. Thus, $z_{kP_0} \equiv 0 \pmod q$, and the unknown factor of N, q, can be recovered by taking $\gcd(z_{kP_0}, N)$.

Montgomery [2, 3] and Brent [5] independently suggested a continuation of Phase 1 if one has $kP_0 \neq O$. Their ideas utilize that fact that even if one has $Q_0 = kP_0 \neq O$, the value of k might miss just one large prime divisor of $|E(\mathbb{Z}_q)|$. In that case, one only needs to compute the scalar multiplication by p to get $pQ_0 = O$. A second bound B_2 restricts the size of possible values of p.

Let $M(N)$ be the cost of one multiplication $\pmod N$. Then Phase 1 of ECM finds a factor q of N with the conjectured time complexity [4] $O(\exp((\sqrt{2} + o(1))\sqrt{\log q \log\log q})M(N))$. Phase 2 speeds up Lenstra's original method by the factor $\log q$ which is absorbed in the $o(1)$ term of the complexity, but is significant for small and medium size factors q.

2.2. Operations on an Elliptic Curve

Scalar multiplication, kP, is a basic elliptic curve operation used in the ECM method. It is also a fundamental operation of a majority of Elliptic Curve Cryptosystems [10], and therefore it has been studied extensively in the past from the point of view of efficient implementations in software and hardware. Scalar multiplication is defined as an addition, $kP = P + ... + P$ (k times), where k is an integer and P is a point on an elliptic curve.

An efficient algorithm for computing scalar multiplication was proposed by Montgomery [2] in 1987, and is known as the Montgomery ladder algorithm. This algorithm is applicable to any curve, and is independent of the point representation, i.e., it can be executed in both affine and projective coordinates. However, it is especially useful when an elliptic curve is expressed in Montgomery form (see Eq. 2), in projective coordinates. In this case, all intermediate computations can be carried on using only x and z coordinates, and the y-coordinate of the result can be retrieved (if needed) from the x and z coordinates of the final point. In the ECM method, the y-coordinate of the result is not needed, so this final computation is unnecessary.

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 $x_0 \neq 0$, $k = (k_{s-1}k_{s-2}\cdots k_1k_0)_2 > 0$, an s-bit positive integer
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 $x_P x_Q (x_P - x_Q) \neq 0$, $P_0 = (x_0 :: z_0) = (x_{P-Q} :: z_{P-Q}) = P - Q$, $a_{24} = \frac{a+2}{4}$, where a is a parameter of the curve E in Eq. 2
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: $4x_P z_P \leftarrow (x_P + z_P)^2 - (x_P - z_P)^2$
4: $x_{2P} \leftarrow (x_P + z_P)^2 (x_P - z_P)^2$
5: $z_{2P} \leftarrow (4x_P z_P)\left((x_P - z_P)^2 + a_{24} \cdot (4x_P z_P)\right)$

The algorithm is constructed in such a way that the difference between the intermediate points P and Q, $P - Q$, is always constant, and equal to the value of the initial point P_0. Therefore, x_{P-Q} and z_{P-Q} in the formulas in Algorithm 3 can be replaced by x_0 and z_0 respectively.

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 independently 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. Parametrization of an Elliptic Curve

Let E be an elliptic curve in Montgomery form and let q be a prime with $q \not| 2b(a^2 - 4)$. Then it is well known that the order $|E(\mathbb{Z}_q)|$ is divisible by 4. By choosing E such that $|E(\mathbb{Z}_q)|$ is divisible by a factor larger than 4, one may get a slight advantage during the procedure of ECM, since ECM tries to find q such that $|E(\mathbb{Z}_q)|$ is a product of small primes. One popular way of having an elliptic curve with such property is to use a Suyama's parametrization [7]. Let σ be any integer > 5. Suyama's parametrization defines an elliptic curve in Montgomery form as follows: Let $u = \sigma^2 - 5, v = 4\sigma$. Then define the curve parameters a_σ, b_σ by

$$a_\sigma = \frac{(v-u)^3(3u+v)}{4u^3 v} - 2, \quad b_\sigma = \frac{u}{v^3}, \tag{3}$$

and define an initial point $P_\sigma = (x_\sigma, y_\sigma, z_\sigma)$ by

$$x_\sigma = u^3 = (\sigma^2 - 5)^3, \;\; y_\sigma = (\sigma^2 - 1)(\sigma^2 - 25)(\sigma^4 - 25), \;\; z_\sigma = v^3 = 4^3\sigma^3. \tag{4}$$

The advantage of using Suyama's parametrization is as follows. First, the elliptic curve with Suyama's parametrization always admits $|E(\mathbb{Z}_q)|$ divisible by 12. Thus one might expect a slight advantage in finding a prime q such that the order of $E(\mathbb{Z}_q)$ is smooth. Second, using Suyama's parametrization, an elliptic curve E_σ and an initial point P_σ are completely determined by the value of σ. Therefore we can simultaneously choose different σ to generate different curves E_σ, which is very convenient for parallel implementation of many curves. Because the curve E_σ and the initial point P_σ are defined explicitly by the single parameter σ, experiments can easily be replicated and compared.

Choosing a coordinate z_0 of the initial point P_0 to be equal to $z_0 = 1$ is also possible with Suyama's parametrization. This is done by assigning $x_0 \equiv x_\sigma/z_\sigma \pmod N$ and $z_0 \equiv z_\sigma/z_\sigma \equiv 1 \pmod N$. Note that this initialization is done with no additional inversion cost because $x_0 \equiv x_\sigma/z_\sigma \equiv (\frac{u}{v})^3 \equiv (4u^4 \cdot \frac{1}{4u^3v})^3 \pmod N$, where the inverse of $4u^3v \pmod N$ is already known in the expression of a_σ of Suyama's parametrization.

2.4. Montgomery Multiplication

Let $N > 0$ be an odd integer. In many cryptosystems such as RSA, computing $XY \pmod N$ is a crucial operation. Taking the reduction of $XY \pmod N$ is a more time consuming step than the multiplication XY without reduction. Montgomery [1] introduced a method for calculating products $\pmod N$ without the costly reduction $\pmod N$, known as Montgomery multiplication. Montgomery multiplication of X and Y, $MP(X, Y, N)$, is defined as $XY2^{-n} \pmod N$ for some fixed integer n.

Since Montgomery multiplication is not an ordinary multiplication, there is a process of conversion between the ordinary domain (with ordinary multiplication) and the Montgomery domain. The conversion between the ordinary domain and the Montgomery domain is given by the relation $X \longleftrightarrow X'$ with $X' = X2^n \pmod N$ and the corresponding diagram is shown below.

Ordinary Domain	$\Longleftrightarrow$	Montgomery Domain
X	$\leftrightarrow$	$X' = X2^n \pmod N$
Y	$\leftrightarrow$	$X' = X2^n \pmod N$
XY	$\leftrightarrow$	$(XY)' = XY2^n \pmod N$

The table shows that the conversion is compatible with multiplications in each domain, since

$$MP(X', Y', N) \equiv X'Y'2^{-n} \equiv (X2^n)(Y2^n)2^{-n} \equiv XY2^n \equiv (XY)' \pmod N.$$

The conversion between each domain can be done using the same Montgomery operation, in particular $X' = MP(X, 2^{2n} (\mathrm{mod}\, N), N)$ and $X = MP(X', 1, N)$, where $2^{2n} (\mathrm{mod}\, N)$ can be precomputed. Despite the initial conversion cost, if we do many Montgomery multiplications followed by a conversion as in the ECM method or in RSA, we obtain an advantage over ordinary multiplication.

Algorithm 4 Radix-2 Montgomery Multiplication

Require: $N, n = \lfloor \log_2 N \rfloor + 2, X = \sum_{j=0}^{n-1} X_j 2^j$, $Y = \sum_{j=0}^{n-1} Y_j 2^j$ with $0 \le X, Y < 2N$
Ensure: $Z = MP(X, Y, N) = XY2^{-n} \pmod N < 2N$
1: $S[0] \leftarrow 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad q_i \leftarrow S[i]_0 + X_i Y_0 \pmod 2$
4: $\quad S[i + 1] \leftarrow (S[i] + X_i Y + q_i N)$ div 2
5: **end for**
6: return $S[n]$

Algorithm 4 shows the pseudocode for radix-2 Montgomery multiplication where we choose $n = \lfloor \log_2 N \rfloor + 2$. It should be mentioned that our n is slightly different from $\lfloor \log_2 N \rfloor + 1$ which Montgomery [1] originally used.

This modified algorithm makes all the inputs and output in the same range, i.e., $0 \leq X, Y, S[n] < 2N$. Therefore it is possible to implement Algorithm 4 repeatedly without any reduction unlike the original algorithm [1], where one has to take reduction $\pmod N$ at the end of the algorithm to make the output value in the same range as the input values.

The verification of the above algorithm is given below: Let us define $S[i]$ as $S[i] \equiv \frac{1}{2^i}\left(\sum_{j=0}^{i-1} X_j 2^j\right) Y \pmod N$ with $S_0 = 0$. Then, $S[n] \equiv XY2^{-n} \pmod N = MP(X, Y, N)$. Thus, $S[n]$ can be computed iteratively using dependence:

$$\begin{aligned} S[i+1] &\equiv \frac{1}{2^{i+1}}\left(\sum_{j=0}^{i} X_j 2^j\right) Y \equiv \frac{1}{2^{i+1}}\left(\sum_{j=0}^{i-1} X_j 2^j + X_i 2^i\right) Y \\ &\equiv \frac{1}{2}\left(\frac{1}{2^i}\left(\sum_{j=0}^{i-1} X_j 2^j\right) Y + X_i Y\right) \equiv \frac{1}{2}(S[i] + X_i Y) \pmod N. \end{aligned}$$

Therefore depending on the parity of $S[i] + X_i Y$, we compute $S[i+1]$ as

$$S[i+1] = \frac{S[i] + X_i Y}{2} \quad \text{or} \quad \frac{S[i] + X_i Y + N}{2}, \tag{5}$$

to make the numerator divisible by 2. Since $n = \lfloor \log_2 N \rfloor + 2$, one has $0 \leq S[i] < 2N$ by the Eq. 5.

2.5. Implementation of Phase 2

Phase 1 computes one scalar multiplication kP_0, and the implementation issues are relatively easy compared to Phase 2. For Phase 2, we follow the basic idea of the standard continuation [2] and modify it appropriately for efficient FPGA implementation. Choose $0 < D < B_2$, and let every prime p, $B_1 < p \leq B_2$, be expressed in the form

$$p = mD \pm j \tag{6}$$

where m changes between $M_{MIN} = \lfloor (B_1 + \frac{D}{2})/D \rfloor$ to $M_{MAX} = \lceil (B_2 - \frac{D}{2})/D \rceil$, 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 $M_N = M_{MAX} - M_{MIN} + 1$. 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

$$\gcd(d, N) > 0, \quad \text{where } d = \prod_{m,j} (x_{mDQ_0} z_{jQ_0} - x_{jQ_0} z_{mDQ_0}) \tag{7}$$

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 Eq. 7 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 table:

$$\text{prime\_table}[m, j] = 1 \text{ when } mD + j \text{ or } mD - j \text{ is prime, and 0 otherwise}$$

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:

$$\text{GCD\_table}[j] = 1 \text{ when } j \in J_S, \text{ and 0 otherwise}$$

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). This leads to the following version of the algorithm for Phase 2, in which we assume D is even. 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 precomputations and memory storage required for Phase 2.

Algorithm 5 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, $B1$: smoothness bound for Phase 1, $B2$: smoothness bound for Phase 2, $B2 > B1$, D: parameter determining a trade-off between the computation time and the amount of memory required

Ensure: q: factor of N, $1 < q \leq N$ or FAIL

Precomputations:

$M_{MIN} \leftarrow \lfloor (B_1 + \frac{D}{2})/D \rfloor, \quad M_{MAX} \leftarrow \lceil (B_2 - \frac{D}{2})/D \rceil$
clear GCD_table, clear J_S
for each $j = 1$ to $\frac{D}{2}$ **do**
  if $\gcd(j, D) = 1$ **then**
    GCD_table$[j] = 1$
    add j to J_S
  end if
end for
clear prime_table
for each $m = M_{MIN}$ to M_{MAX} **do**
  for each $j = 1$ to $\frac{D}{2}$ **do**
    if ($mD + j$ or $mD - j$ is prime) **then**
      prime_table$[m, j] = 1$
    end if
  end for
end for
$Q \leftarrow Q_0$
for $j = 1$ to $\frac{D}{2}$ step 2 **do**
  if GCD_table$[j] = 1$ **then**
    store Q in S $//Q = jQ_0 = (x_{jQ_0} :: z_{jQ_0})//$
  end if
  $Q \leftarrow Q + 2Q_0$
end for

Main computations:

$d \leftarrow 1, \quad Q \leftarrow DQ_0, \quad R \leftarrow M_{MIN}Q$
for each $m = M_{MIN}$ to M_{MAX} **do**
  for each $j \in J_S$ **do**
    if prime_table$[m, j] = 1$ **then**
      retrieve jQ_0 from table S
      $d \leftarrow d \cdot (x_R z_{jQ_0} - x_{jQ_0} z_R)$ $//R = (x_R :: z_R)//$
    end if
  end for
  $R \leftarrow R + Q$
end for
$q \leftarrow \gcd(d, N)$
if $q > 1$ **then**
  return q
else
  return FAIL
end if

2.6. Choice of B_1, B_2 and D

The subexponential time complexity $O(\exp((\sqrt{2}+o(1))\sqrt{\log q \log\log q})M(N))$ of ECM is achieved by choosing the theoretical bound $B_1 \approx e^{\sqrt{\frac{1}{2}\log q \log\log q}}$ [4], where log is a natural logarithm. However the precise value of $o(1)$ term is difficult to estimate. Choice of the bound B_1 is closely related with Dickman-de Brujin function $\rho(u)$ [3], which gives the probability that a randomly chosen integer X is $X^{\frac{1}{u}}$-smooth. As with the case of B_1, an optimal bound B_2 is related with certain numerical integrations involving Dickman-de Brujin type functions. However, it seems that predicting precise values of theoretical optimal bounds, B_1 and B_2, is rather difficult. Instead, one usually determines B_1 first (which is more or less close to $e^{\sqrt{\frac{1}{2}\log q \log\log q}}$) and set B_2 between $50B_1$ and $100B_1$ depending on the computational resources for Phase 2. For example, Simka et al. [18] choose $B_1 = 960$ and $B_2 = 57000$ to find a 40-bit prime divisor of 200-bit integers. By setting $q = 2^{41}$, we have $e^{\sqrt{\frac{1}{2}\log q \log\log q}} \approx 988$ which is close to 960. The ratio B_2/B_1 in [18] is $57000/960 \approx 59$.

In Phase 2, one needs at most D point additions for the computation of the set S and at most B_2/D additions for the table T. Thus the time complexity of finding tables of S and T is $O(D + B_2/D)$. By choosing $D \approx \sqrt{B_2}$, one minimizes $O(D + B_2/D) = O(\sqrt{B_2})$. Also one may choose D in such a way that it has many prime factors so that the size of the set S can be further reduced. However in memory constrained hardware devices, choosing $D \approx \sqrt{B_2}$ is not possible because the table S (or at least one of S and T) should be precomputed and needs to be saved. For hardware purposes, one may choose D sufficiently small such as $D = 30$ or 210 and use the precomputed table S.

2.7. Software Implementation of ECM and Summary of GMP-ECM

The ECM method is implemented in many educational software packages such as Maple or Mathematica as well as in the educational program for learning cryptology, CrypTool [11]. On the other hand, massively parallel implementations of the ECM method are reported in [8, 9]. One of the most popular and powerful ECM tools is GMP-ECM [15]. It contains various optimization techniques for Phase 1 and Phase 2 and exploits many ideas known in the literature. Especially, it uses elliptic curves in Montgomery form with Suyama's parametrization. Let us briefly explain some of the optimization techniques in GMP-ECM.

Table 1. Summary of GMP-ECM

	Phase 1	Phase 2
Elliptic Curve	Montgomery form: $by^2z = x^3 + ax^2z + xz^2$	Weierstrass form: $Y^2 = X^3 + AX + B$
Coordinate	Projective	Affine
Optimization Techniques (Reducing time)	Lucas chain (PRAC algorithm)	Fast polynomial multiplication Montgomery's D_1D_2 method
Optimization Techniques (Increasing probability)		Brent-Suyama extension
Porting Optimizations to Hardware	Possible with precomputations in software	Inverter required Large amounts of memory required

Phase 1 in GMP-ECM : The main operation here is to compute kP_0 for a fixed point P_0 on the curve E. Projective coordinates are used. To minimize the number of necessary point additions and doublings, GMP-ECM uses Lucas chains for k instead of using a Montgomery ladder algorithm with binary expansion of k. A typical example of Lucas sequence is a Fibonacci sequence; $1, 1, 2, 3, 5, 8, 13, 21, \cdots$. For instance, the computation of $21P$ can be done by the chain of the points $P, 2P, 3P, 5P, 8P, 13P$ and $21P$. Since the i-th point P_i satisfies $P_i = P_{i-1} + P_{i-2}$ and $P_{i-1} - P_{i-2} = P_{i-3}$, the difference of two points is also in the sequence and therefore we can use the addition formula in Algorithm 3. With Lucas chains, one needs 5 additions and 1 doubling to compute $21P$. However, if we use a Montgomery ladder algorithm with $21 = (10101)_2$, we need 4 additions and 4 doublings because both of addition and doubling are performed in each step. The Lucas chain idea might be useful in hardware, assuming that the best possible Lucas chain is first precomputed in software.

Phase 2 in GMP-ECM : Phase 2 requires a substantial amount of memory for the tables S and T. Moreover to use fast polynomial arithmetic, GMP-ECM uses affine coordinates with the Weierstrass form of elliptic curve

to compute $G(x) = \prod_{(x_\tau, y_\tau) \in T}(x - x_\tau)$ using the product tree algorithm [6]. Then GMP-ECM applies the POLYEVAL algorithm to compute $\prod_{(x_\sigma, y_\sigma) \in S} G(x_\sigma)$ where the polynomial multiplications are done using the Schönhage-Strassen algorithm [7].

Another approach possible in Phase 2 is the Brent-Suyama extension. It is based on the idea that, for any polynomial f with integer coefficients and for any integer i and j, one has $i \pm j | f(i) \pm f(j)$. Therefore a point Q satisfying $(i \pm j)Q = O$ also satisfies $(f(i) \pm f(j))Q = O$ but the converse might not be true. That is, one may expect that Q is killed by $f(i) \pm f(j)$ even though $(i \pm j)Q \neq O$. GMP-ECM uses $f(x) = x^s$ or Dickson polynomial. Computing $f(i)Q$ from the information of $f(i-1)Q$ requires one to compute certain points Q_1, Q_2 and $Q_1 + Q_2$ where $Q_1 - Q_2$ is not known. Therefore a standard Weierstrass form of elliptic curve with affine coordinates is used instead of the Montgomery form of elliptic curve with projective coordinates. It should be mentioned that the Brent-Suyama extension increases the probability in Phase 2 by slightly sacrificing the speed.

Another optimization technique in Phase 2 of GMP-ECM is the so called $D_1 D_2$ method of Montgomery. Montgomery [3] suggested that one may use D_1, D_2 instead of using one D. By choosing $D_1, D_2 < B_2$ with $\gcd(D_1, D_2) = 1$, any integer $p < B_2$ is represented as

$$p = iD_1 + jD_2,$$

where $0 \leq j < D_1$ (equivalently, $jD_2 \equiv p \pmod{D_1}$) and $i = \frac{p - jD_2}{D_1}$. In this case, the table $S = \{jD_2Q\}$ is same as the table in the previous case with $D_2 = 1$. However the size of the table T is reduced by the factor $1 - \frac{\phi(D_2)}{D_2}$ as follows. Since $iD_1 + jD_2 = p$ is a prime, the table $T = \{iD_1Q\}$ has the restriction $\gcd(i, D_2) = 1$ with $-D_2 < i < \frac{B_2}{D_1}$. If $0 < D_2 < D_1 << B_2$, then the lower bound of i is almost same as the previous case with $D_2 = 1$ and the condition $\gcd(i, D_2) = 1$ implies that the new table T excludes the indices i with $\gcd(i, D_2) > 1$. Therefore the size of a new table T is $\approx \frac{B_2}{D_1} \cdot \frac{\phi(D_2)}{D_2}$ and it is reduced by the factor of $1 - \frac{\phi(D_2)}{D_2}$ compared with the original construction of the case $D_2 = 1$.

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 different 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.

The contents of the global memory after initialization for Phase 1 is shown in Figure 2.

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

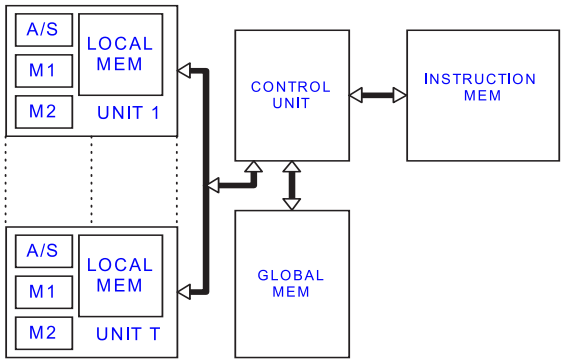

Figure 1. Block diagram of the top-level unit. Notation: MEM-memory; $M1, M2$-multipliers 1 and 2; A/S-adder/subtractor.

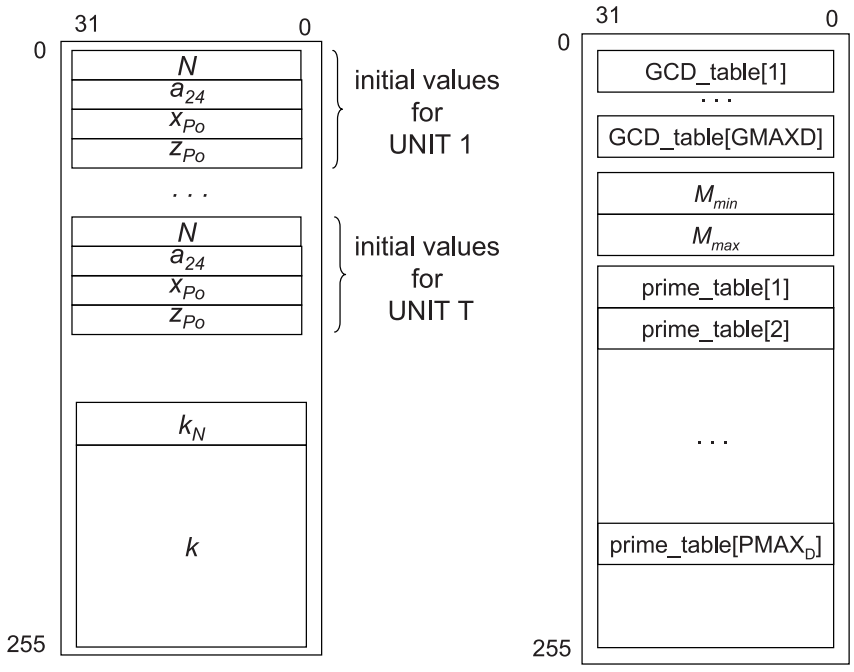

Figure 2. Contents of the Global Memory in a) Phase 1, b) Phase 2.

The results of these calculations are 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 calculation of Phase 1, $p_i = \gcd(z_{Q_0}, N)$. If $p_i = 1$, no factor was found by a given ECM unit. If $p_i > 1$ and $p_i \neq N$, then a non-trivial factor of N, p_i, was found. If p_i is equal to N for all ECM units working on the same N, then the computations of Phase 1 need to be repeated for a smaller value of the bound B_1.

If no factor of N was found, the ECM system is ready for Phase 2. The values of N, parameters of the curves a_{24}, 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}, $M_N = M_{MAX} - M_{MIN} + 1$; bit table called the GCD_table, which for every odd $j \leq D/2$, determines whether $\gcd(j, D) = 1$; bit table called the prime_table, which for every $M_{MIN} \leq m \leq M_{MAX}$ and $0 < j \leq D/2$ determines whether either $m \cdot D + j$ or $m \cdot D - j$ is a prime.

The contents of the global memory after initialization for Phase 2 is shown in Figure 2b. Note that the previous contents of the global memory used for Phase 1 can be overwritten because the inputs to Phase 1 are either no longer needed (P_0, k), or have been already loaded to the local memories (N, a_{24}). Phase 2 is then started simultaneously on all ECM units, and produces as final results, the accumulated products d (see Eq. 7). These final results are then download to the host computer, where the final calculations $\gcd(d, N)$ are performed.

Note that with this top level organization, there is no need to compute greatest common divisors or divisions 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

3.2.1. 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.

The two elementary steps of this algorithm consist of the computations

$$\text{Step 1:} \quad \begin{aligned} P &= 2P \\ Q &= P + Q \end{aligned}$$

$$\text{Step 2:} \quad \begin{aligned} P &= P + Q \\ Q &= 2Q. \end{aligned}$$

The second step can be calculated using the exact operations of the first step with a simple swap of input and output variables:

$$\begin{aligned} P' &= Q, \\ Q' &= P \\ \hline P' &= 2P' \\ Q' &= P' + Q' \\ \hline P &= Q' = P + Q \\ Q &= P' = 2Q. \end{aligned}$$

Thus, only one of these two steps, and a conditional swap of variables at the input and output needs to be implemented.

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

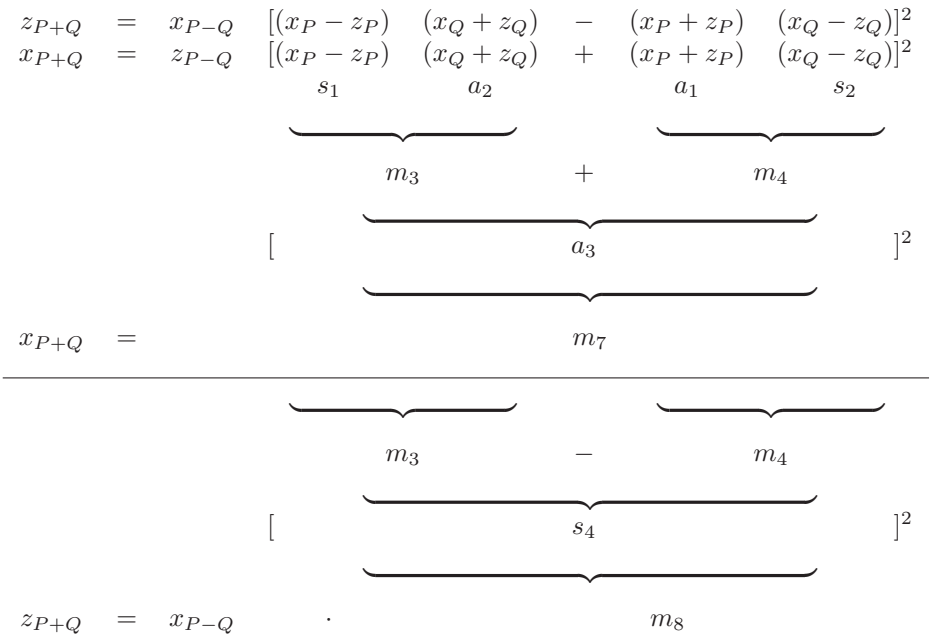

Figure 3. Computation flow for a step of scalar multiplication computing $P + Q$

$2P$ can be interleaved, and assigned to three functional units working in parallel, as shown in Table 2. For example, in Figure 3, we show the computational flow for $P + Q$. 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. 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 a single multiplication.

Table 2. One step of a scalar multiplication, including the concurrent operations $P+Q$ and $2P$, for the case of $z_{P-Q} = 1$. Notation: A: operation used for addition only, D: operation used for doubling only, A/D: operation 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 a similar algorithm to the one used in Phase 1. The only difference is that now, $P - Q = Q_0$, cannot be chosen arbitrarily, and thus, $z_{P-Q} = z_{Q_0} \neq 1$ in general. As a result, the computations

will take the amount of time required for 2 modular additions/subtractions and 6 modular multiplications, as shown in Table 3.

Table 3. One step of a scalar multiplication, including the concurrent operations $P+Q$ and $2P$, for the case of $z_{P-Q} \neq 1$. Notation: A: operation used for addition only, D: operation used for doubling only, A/D: operation 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: $m_7 = a_3^2$	A: $m_8 = s_4^2$
		A: $z_{P+Q} = m_9 = m_8 \cdot x_{P-Q}$	D: $z_{2P} = m_{10} = s_3 \cdot a_4$
			A: $x_{P+Q} = m_{11} = m_7 \cdot z_{P-Q}$

The second type of operation required in Phase 2 is a simple point addition $P + Q$. This operation can be performed using the time of 6 additions/subtractions and 3 modular multiplications, as shown in Table 4.

Table 4. Addition of points $P + Q$

Adder/Subtractor	Multiplier 1	Multiplier 2
$a_1 = x_P + z_P$ $s_1 = x_P - z_P$		
$a_2 = x_Q + z_Q$ $s_2 = x_Q - z_Q$		
	$m_3 = s_1 \cdot a_2$	$m_4 = s_2 \cdot a_1$
$a_3 = m_3 + m_4$ $s_3 = m_3 - m_4$		
	$m_7 = a_3^2$	$m_8 = s_4^2$
	$z_{P+Q} = m_{10} = m_8 \cdot x_{P-Q}$	$x_{P+Q} = m_{11} = m_7 \cdot z_{P-Q}$

Finally, the last medium level operation required in Phase 2 is the accumulation of the product d:

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

where

$$\begin{aligned} (x_i, z_i) &\in \{(x, z) \colon (x, z) = jQ_0\}, \\ (x_n, z_n) &\in \{(x, z) \colon (x, z) = mDQ_0\} \end{aligned}$$

and GCD_table[j]=1 and prime_table[m, j]=1. The repetitive sequence of such operations is shown in Table 5.

Table 5. 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
	$m_1 = x_n \cdot z_0$	$m_2 = x_0 \cdot z_n$
$d_{0n} = m_1 - m_2$	$m_3 = x_n \cdot z_1$	$m_4 = x_1 \cdot z_n$
$d_{1n} = m_3 - m_4$	$d = d \cdot d_{0n}$	$m_1 = x_n \cdot z_2$
	$d = d \cdot d_{1n}$	$m_2 = x_2 \cdot z_n$
$d_{2n} = m_1 - m_2$	$m_3 = x_n \cdot z_3$	$m_4 = x_3 \cdot z_n$
$d_{3n} = m_3 - m_4$	$d = d \cdot d_{2n}$	$m_1 = x_n \cdot z_4$
	$d = d \cdot d_{3n}$	$m_2 = x_4 \cdot z_n$
.....	.....	.....

As can be seen from Table 5, after the initial delay of one multiplication, the time required to compute and accumulate any two subsequent values of d_{in} is equal to the time of three multiplications.

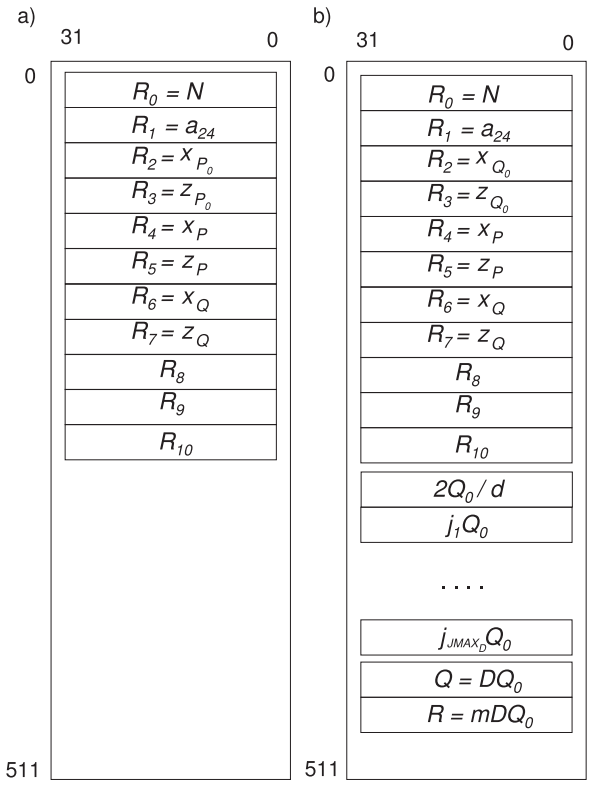

Figure 4. Contents of the local memories in a) Phase 1, b) Phase 2.

3.2.2. Instructions of the ECM unit

Each ECM unit is composed of two modular multipliers, one adder/subtractor, and one local memory. The local memory is 512 32-bit words in size, equivalent to 64 256-bit registers. The contents of the local memory during the execution of Phase 1 and Phase 2 are shown in Figures 4a and 4b, respectively. 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 execute 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 2).
2. One step of a scalar multiplication kP ($P = 2P$, $Q = P + Q$) in Phase 2, i.e., with $z_{P_0} \neq 1$ (see Table 3).
3. Addition $P + Q$ in Phase 2, i.e., with $z_{P_0} \neq 1$ (see Table 4).

Each instruction describes the operation of one of the functional units corresponding to Tables 2, 3 and 4, and encodes the addresses of locations in the local memory used to store input 1, input 2, and output of the given operation, as shown in Figure 5. 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 the input/output address.

The operation performed by each instruction is determined based on the position of the instruction in the instruction sequence, and thus no opcode is needed in the instruction body. In particular, a group of four instructions corresponds to one row of Tables 2, 3 and 4, and is listed in the order: Multiplication 2, Multiplication

SWAP	ON	INPUT 1	INPUT 2	OUTPUT
XX	X	XXXX	XXXX	XXXX

Figure 5. Format of an ECM system instruction

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.

In each table there exist empty fields, which means that the given arithmetic unit should remain inactive in the given step, this inactivity is described using the flag ON, in the instruction body. Additionally, for the scalar multiplications, two SWAP bits determine which instructions swap inputs or outputs between P and Q, depending on the current bit of a scalar k. Swapping inputs affects only the first four additions/subtractions in Tables 1 and 2, i.e., the instructions that use x_P, z_P, x_Q, and z_Q as inputs. Swapping outputs affects only the four instructions that compute outputs x_{P+Q}, z_{P+Q}, x_{2P}, and z_{2P} in Tables 2, 3. The SWAP bits are set to "00" (no swap) for all instructions of the point addition in Phase 2 (Table 4). In the optimized version of our design, the number of ON flags has been increased to two, which allowed us to overlap sequences of instructions corresponding to Tables 2, 3, 4 and minimize the total number of instructions in the instruction memory to 32.

3.3. Low-level View: Modular multiplication and addition/subtraction

The three low level operations implemented by the ECM unit are Montgomery modular multiplication (defined in Section 2.4), 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, adder/subtractor.

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 $\pmod N$. The algorithms for modular addition and subtraction are shown as Algorithms 6 and 7 respectively. In both algorithms, S is a result variable, T is a temporary variable, and C_1, C_2 are two carry bits.

Algorithm 6 Modular addition

Require: $N, X, Y < 2N$, all expressed using e 32-bit words, $X^{(j)}, Y^{(j)}, N^{(j)}$, $j = 0, \ldots, e-1$
Ensure: $Z = X + Y \mod 2N$
1: **for** $j = 0$ to $e - 1$ **do**
2: $(C_1, T^{(j)}) \leftarrow C_1 + X^{(j)} + Y^{(j)}$
3: **end for**
4: **for** $j = 0$ to $e - 1$ **do**
5: $(C_2, S^{(j)}) \leftarrow C_2 + T^{(j)} - (2N)^{(j)}$
6: **end for**
7: **if** $S < 0$ **then**
8: return T
9: **else**
10: return S
11: **end if**

The block diagram of the adder/subtractor unit implementing both algorithms is shown in Figure 6. The modulus N is loaded to the adder/subtractor, using input X_N, one time, during the initialization stage of Phase 1, and does not need to be changed until the next run of Phase 1 for another number N. This modulus is stored in the internal 32×32-bit memory, used to hold three numbers N, S, and T, all up to 256 bits wide. The 32-bit words of operands X and Y are loaded in parallel, starting from the least significant word, and immediately added or subtracted, depending on the value of the control input sub_add (with sub_add = 1

Algorithm 7 Modular Subtraction

Require: N, X, $Y < 2N$, all expressed using e 32-bit words
$X^{(j)}$, $Y^{(j)}$, $N^{(j)}$, $j = 0, \ldots, e-1$
$X^{(j)}, Y^{(j)}, N^{(j)}$, $j = 0, \ldots, e-1$
Ensure: $Z = X - Y \mod 2N$
1: **for** $j = 0$ to $e-1$ **do**
2: $(C_2, S^{(j)}) \leftarrow C_2 + X^{(j)} - Y^{(j)}$
3: **end for**
4: **for** $j = 0$ to $e-1$ **do**
5: $(C_1, T^{(j)}) \leftarrow C_1 + S^{(j)} + (2N)^{(j)}$
6: **end for**
7: **if** $S < 0$ **then**
8: return T
9: **else**
10: return S
11: **end if**

denoting subtraction). The result is stored in the internal memory as variable T for addition i.e. $X + Y$, and S for subtraction i.e. $X - Y$. This first operation is followed by the second operation of the respective algorithm, involving the previously computed value and the modulus $2N$ computed on the fly, with the result stored back to the memory. Finally, depending on the sign of S, stored in the flip-flop C_2, either T or S is returned as a final result. For 256-bit operands, the entire operation takes 41 clock cycles (including writing data back to local RAM), the same amount for addition and subtraction.

The radix-2 version of the Montgomery Multiplication algorithm, which calculates the Montgomery product of X and Y is specified as Algorithm 4 in section 2.4. This algorithm assumes that all words of the inputs X, Y, and M, are already available inside of the multiplier, and can be accessed at the same time. The second instruction inside of the for loop involves the addition of three long words. If implemented directly in hardware the operation would result in a long critical path and a very low clock frequency. In order to prevent that, this addition is performed using carry save adders, and the result $S[i+1]$ is stored in the carry save form. Using carry save adders, the sum of three numbers U, V, W is reduced to the sum of two numbers S (sum) and C (carry), such that $U+V+W = C+S$. Similarly, using a cascade of two carry save adders, as shown in Figure 7, the sum of four numbers, U, V, W, and Y can be reduced to the sum of two numbers S and C, such that $U + V + W + Y = C + S$. Each carry save adder is composed of a row of n Full Adders working in parallel, so it introduces a delay of just a single Full Adder (i.e., a delay of a single stage of a basic ripple-carry adder).

Algorithm 8 Radix-2 Montgomery Multiplication with Carry Save Addition

Require: $N, n = \lfloor \log_2 N \rfloor + 2, X = \sum_{j=0}^{n-1} X_j 2^j$, $Y = \sum_{j=0}^{n-1} Y_j 2^j$ with $0 \leq X, Y < 2N$
Ensure: $Z = MP(X, Y, N) = X \cdot Y \cdot 2^{-n} \pmod N < 2N$; $Z^{(j)}$,$C[n]^{(j)}$, $S[n]^{(j)}$ denote a j-th word of $Z, C[n]$ and $S[n]$ respectively.
1: $S[0] \leftarrow 0$
2: $C[0] \leftarrow 0$
3: **for** $i = 0$ to $n-1$ **do**
4: $q_i \leftarrow (C[i]_0 + S[i]_0 + X_i \cdot Y_0) \pmod 2$
5: $(C[i+1], S[i+1]) \leftarrow CSA(C[i], S[i], X_i \cdot Y, q_i \cdot N)$ div 2
6: **end for**
7: $C = 0$
8: **for** $j = 0$ to 7 **do**
9: $(C, Z^{(j)}) \leftarrow C[n]^{(j)} + S[n]^{(j)} + C$
10: return $Z^{(j)}$
11: **end for**

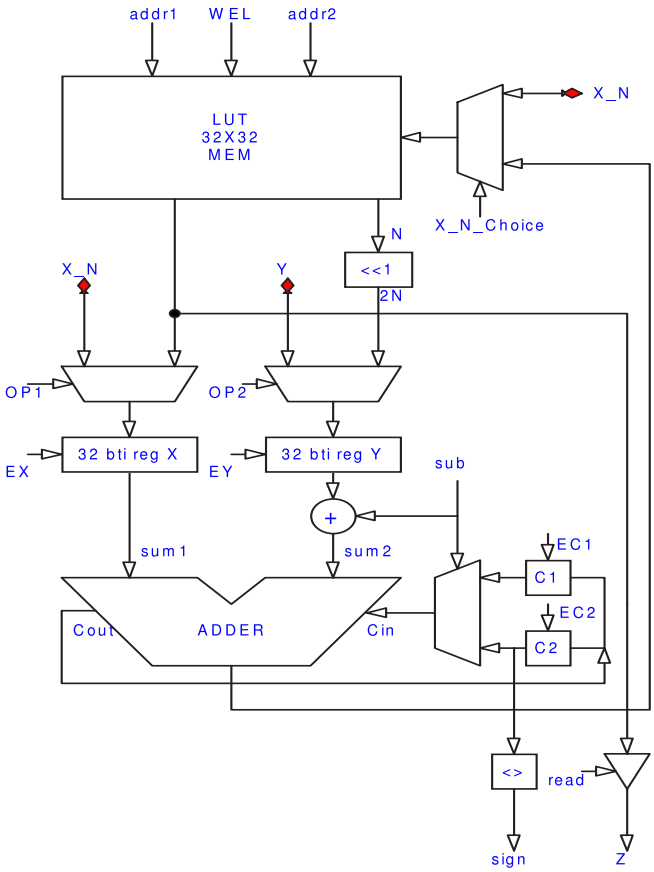

Figure 6. Block diagram of the adder-subtractor

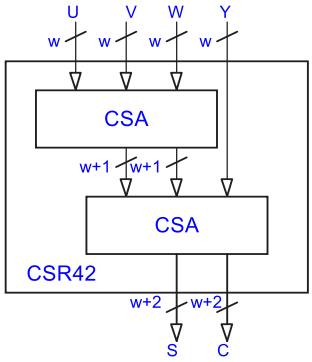

Figure 7. A cascade of two carry save adders, reducing four operands to two

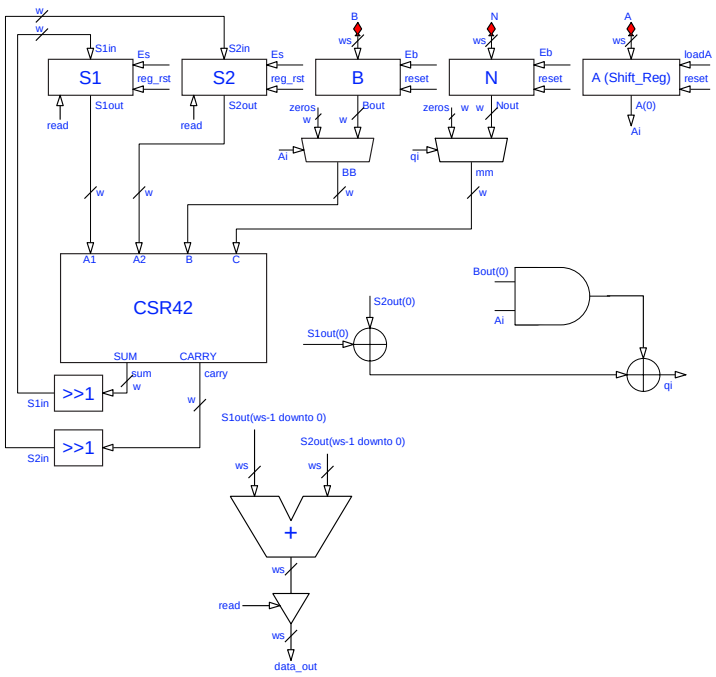

Figure 8. Block diagram of a Montgomery multiplier

The modified algorithm, based on carry save addition (CSA) is shown as Algorithm 8. This algorithm has been described earlier in [25]. The block diagram of the circuit implementing Algorithm 8 is shown in Figure 8. The modulus N and the parameter n are loaded in to the multiplier once at the beginning of Phase 1, and do not need to be changed until the beginning of Phase 1 for another number N. At the beginning of multiplication, the inputs X and Y are first loaded in parallel, in 32-bit words, to internal 256-bit registers X and Y. In the following n clock cycles, the circuit executes n iterations of the for loop. Finally, in the last 8 clock cycles, the final result is computed word by word, starting from the least significant word, and transferred to the output. 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 an amount of time significantly 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. Five different families of FPGA devices have been targeted, including the high-performance families, Virtex E, Virtex II, and Virtex 4, as well as low-cost families, such as Spartan 3 and Spartan 3E. The entire design has been thoroughly verified using test vectors generated by a special test program written in C and by comparison with the results of GMP-C.

In Table 6, we summarize the memory requirements of our ECM hardware architecture. The local memory represents memory located within each ECM unit, with a memory map shown in Figure 4. In Phase 1, only 11 256-bit registers are required, taking a total of 88 memory words, and thus a 128x32 bit memory is sufficient to hold all inputs, outputs, and temporary values. In Phase 2, the same registers are required, and an additional precomputed table S of points of the form jQ_0, where $1 \leq j \leq \lfloor D/2 \rfloor$, and $\gcd(j, D) = 1$. Clearly, the size of

this table depends on D, and as a result the total size of the local memory is equal to 256x32 for $D = 30$ and 512x32 for $D = 210$. In the modern families of FPGA devices, such as Spartan 3 and Virtex II, the smallest size of BRAM (Block RAM) that can be allocated to a local memory is 512x32, and smaller memories can be implemented only using distributed RAMs available within CLB slices. Thus, one BRAM is sufficient to hold local memory for both Phase 1 and 2.

Global memory is the memory used by the global control unit, and its map is shown in Fig. 2. The size of this memory in Phase 1 is determined primarily by the number of ECM units. In Phase 2, this memory can be completely overwritten by new values. It is worth noting that the size of its main component, prime_table, is almost independent of the value of D. For 6 ECM units per each control unit, memory requirements in Phase 1 and Phase 2 match, and amount to 256 32-bit words, or one BRAM in modern families of FPGAs. The size of global memory has been minimized by the use of bit tables, GCD_table and prime_table, defined in Section 2.5.

Table 6. Amount of memory required by the ECM Hardware Architecture (for the number of bits of N, $n < 254$)

Objects	**# objects**	**# 32-bit words per objects**	**# of words**	**# of bits**	**Memory size**	**# BRAMs in Virtex (256 × 16 or 512 x 8)**	**# BRAMs in Spartan 3 & Virtex 2 (512 × 32)**
Local memory - Phase 1							
Registers	11	8	**88**	2816	**128 x 32**	**1**	**1**
Local memory - Phase 1 & 2, $D = 30$							
Registers	11	8	88	2816			
jQ_0	4	16	64	2048			
DQ_0, mDQ_0	2	16	32	1024			
			184	5888	**256 x 32**	**2**	**1**
Local memory - Phase 1 & 2, $D = 210$							
Registers	11	8	88	2816			
jQ_0	24	16	384	12288			
DQ_0, mDQ_0	2	16	32	1024			
			504	16128	**512 x 32**	**4**	**1**
Global memory - Phase 1 (6 ECM units, $B_1 = 960$**)**							
ECM unit init values	6×4	8	192	6144			
k_N	1	1	1	32			
k	1	43	43	1376			
			236	7552	**256 x 32**	**2**	**1**
Global memory, Phase 2 ($D = 30$**)**							
GCD_table	1	1	1	32			
M_{min}, M_N	2	1	2	64			
prime_table	1	234	234	7488			
			237	7584	**256 x 32**	**2**	**1**
Global memory, Phase 2 ($D = 210$**)**							
GCD_table	1	2	2	64			
M_{min}, M_N	2	1	2	64			
prime_table	1	201	201	6432			
			205	6560	**256 x 32**	**2**	**1**

The execution times of Phase 1 and Phase 2 in our ECM hardware architecture are derived in Table 7. The generic formulas for all component operations are shown, together with the values of the execution times for the case of 198-bit numbers N, and the smoothness bounds $B_1 = 960$ and $B_2 = 57000$. 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 46% compared to the case of $D = 30$. It also makes this time comparable to the execution time of Phase 1. This reduction is accomplished primarily by a significant reduction in the time required to compute values of mDQ_0 for $m = M_{MIN} + 1$ to M_{MAX}. This time is the largest contributor to the total time of Phase 2 for the case of $D = 30$. The value of $D = 210$ is close to the theoretical optimum $\sqrt{B_2} = 239$, discussed in Section 2.6, and at the same time, the choice of $D = 2 \cdot 3 \cdot 5 \cdot 7$

Table 7. 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)	T_{ADD2}	$T_{ADD2} = 3T_M + 6T_A + 30$	**924**	
Phase 1				
Phase 1	T_{P1}	$T_{P1} = k_N \cdot T_{AD1}$	**1,666,500**	
Phase 2				
Precalculating jQ_0	T_{jQ}	$T_{jQ} = 2T_{AD2} + (\lfloor D/4 \rfloor - 2)T_{ADD2}$	**7476** (**0.23%**)	**49,056** (**2.85%**)
DQ_0	T_{DQ}	$T_{DQ} = \lceil \log_2(D+1) \rceil T_{AD2}$	**7140** (**0.22%**)	**11,424** (**0.66%**)
$M_{MIN}DQ_0$	$T_{M_{min}DQ}$	$T_{M_{min}DQ} = \lceil \log_2(M_{MIN}+1) \rceil T_{AD2}$	**8586** (**0.27%**)	**4284** (**0.25%**)
Calculating mDQ_0 **for** $m = M_{MIN} + 1...M_{MAX}$	T_{mDQ}	$T_{mDQ} = (M_N - 2)T_{ADD2}$	**1,725,108** (**53.63%**)	**244,860** (**14.21%**)
Number of ones in the prime_table	n_{prime_table}		**4531**	**4361**
Calculating accumulated product d	T_d	$T_d = (\lceil 1.5 \cdot n_{prime\_table} \rceil + 1)T_M$	**1,468,368** (**45.65%**)	**1,413,072** (**82.03%**)
Phase 2	T_{P2}	$T_{P2} = T_{jQ} + T_{DQ} + T_{M_{min}DQ} + T_{mDQ} + T_d$	**3,216,660** (**100%**)	**1,722,696** (**100%**)

Table 8. 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 ns		19 ns		
Modular addition	24	0.62 μs	41	0.78 μs	0.6	0.8
Modular subtraction	16	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	**1,666,500**	**31.7** ms	**6.8**	**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_i (Phase 2)	4776	124.2 μs	648	12.3 μs	6.2	**10.1**
Phase 2 ($D = 30$)	**20,276,060**	**527.2** ms	**3,216,660**	**61.1** ms	**6.3**	**8.6**
Phase 2 ($D = 210$)	-	-	**1,722,912**	**32.7** ms	**11.8**	**16.1**

Part 2: Resource usage per one ECM unit					
	Pelzl, Šimka, et al.		**Our design ($D = 210$)**		**Ratio Ours / Pelzl, Šimka**
Number of	#	%	#	%	
CLB slices		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**

(a product of small primes) helps to somewhat reduce the size of the precomputed table S. For the given values of the smoothness bounds $B_1 = 960$ and $B_2 = 57,000$, the choice of $D = 210$ results in the smallest possible execution time of Phase 2. For this value of D, the largest contribution to this execution time, 82%, comes from the calculation of the accumulated product d.

In Table 8, we compare our ECM architecture to an earlier design by Pelzl, Šimka, et al., presented at SHARCS 2005, and described in subsequent publications [16, 18]. 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 [16, 18], i.e., 38 MHz (clock period = 26 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 8.6 in terms of the execution time for Phase 2 with the same value of parameter D, and by a factor of 16.1 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.

Table 9. 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**	**Spartan 3E XC3S1600E-5**	**Virtex 4 XC4VLX200-ll**
Resources with one ECM unit					
- CLB slices	**3102 (16%)**	**3197 (9%)**	**3322 (10%)**	**3463 (23%)**	**3224 (4%)**
- LUTs	**4933 (13%)**	**5025 (7%)**	**5134 (8%)**	**5086 (17%)**	**5047 (3%)**
- FFs	**3129 (8%)**	**3102 (5%)**	**3130 (5%)**	**3105 (11%)**	**3077 (2%)**
- BRAMs	**2/160**	**2/144**	**2/104**	**3/336**	**3/336**
Maximum number of ECM units per FPGA device	**7**	**10**	**10**	**4**	**27**
Technology	0.15/0.12 μm	0.15/0.12 μm	90 nm	90 nm	90 nm
Cost of an FPGA device[a]	**$1230**	**$2700**	**$130**	**$35**	**$3000**
Maximum clock frequency for one ECM unit	**54 MHz**	**123 MHz**	**100 MHz**	**93 MHz**	**135 MHz**
Time for Phase 1 and 2	**62.8** ms	**27.6** ms	**33.9** ms	**36.5** ms	**25.2** ms
# of ECM computations per second (with the maximum number of ECM units)	**111 ECM operations/s**	**362 ECM operations/s**	**295 ECM operations/s**	**109 ECM operations/s**	**1073 ECM operations/s**
# of ECM computations per second per $100 (with the maximum number of ECM units)	**9 ECM operations/s per $100**	**13 ECM operations/s per $100**	**227 ECM operations/s per $100**	**311 ECM operations/s per $100**	**36 ECM operations/s per $100**

[a] cost per unit for a batch of 10000+ devices

Table 10. 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 + GMP library)**	**Pentium 4 (GMP-ECM with Phase 1 optimizations turned off**	**Pentium 4 (GMP-ECM with all optimizations turned on)**
Clock frequency	**123 MHz**	**100 MHz**	**2.8 GHz**		
No. of parallel ECM computations	**10**	**10**	**1**		
Time of Phase 1	**13.6** ms	**16.7** ms	**18.3** ms	**13.5** ms	**11.3** ms
Time of Phase 2	**14.0** ms	**17.2** ms	**18.6** ms	**13.5** ms	**13.5** ms
Time of Phase 1 & Phase 2	**27.6** ms	**33.9** ms	**36.9** ms	**27.0** ms	**24.8** ms
# of Phase 1 computations per second	**735**	**600**	**55**	**74**	**89**
# of Phase 2 computations per second	**714**	**580**	**54**	**74**	**74**
# of Phase 1 & 2 computations per second	**362**	**295**	**27**	**37**	**40**

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 [16, 18] 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 37.5 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 6.0 for Phase 2.

In Table 9, we show the results of porting our design to five families of Xilinx FPGAs. For each family, a representative device is selected and used in our implementations. For each ECM device, we determine the exact amount of resources needed for a factoring circuit with one ECM unit, the maximum number of ECM units per chip, the maximum clock frequency, and then the maximum number 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 and Spartan3E device families outperform the high-performance devices, such as Virtex II and Virtex 4 by a factor of about 17.5 and 8.6 respectively, and thus are more suitable for cost effective code breaking computations. Additionally, as expected, the newest generations of devices, Spartan 3E and Virtex 4, have the best results, in their respective categories, in terms of the amount of computations per unit of cost.

In Table 10, 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 [7, 15]. For GMP-ECM we consider two cases, one with all optimizations enabled, and second with the Lucas chain optimization for Phase 1 (see Section 2.7) disabled. Additionally, we run our own test program in C that mimics almost exactly the behavior of hardware, except for using calls to the multiprecision GMP library for the low level operations, such as modular multiplication and addition. One can see that the algorithmic optimizations 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%). In particular, GMP-ECM Phase 1 optimizations (Lucas chains [7]) reduce the execution time of Phase 1, for our specific choice of parameters, by a factor of 16%.

Interestingly, the execution time for an ECM unit running on Virtex II, 6000 is almost identical to 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 10 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 7.4 in favor of the biggest Spartan 3. Additionally this device is actually cheaper than a state-of-the-art microprocessor, so the overall improvement in terms of the performance to cost ratio exceeds a factor of 10.

5. SOFTWARE EXPERIMENTS

The two graphs, shown in Figure 9 and 10, illustrate the probability of success for software experiments of ECM with Pentium 4 Xeon, applied to 200-bit numbers with 40-bit prime factors. For these experiments, we define "success" in two different ways: Firstly, we call a trial successful, if p_1, one particular 40-bit prime factor of our candidate number $N = p_1p_2p_3p_4p_5$, is found. Secondly, we impose a more stringent requirement that all five prime factors need to be found independently and the number completely factored. This is illustrated in Figures 9 and 10, for the following choices of B_1 and B_2.

trial	B_1	B_2
1	800	32470
2	960	57000
3	1100	51114

The parameters for trial 2 are taken from Šimka, et al [18] and the values of B_1 in trials 1 and 3 are chosen for comparison. In those cases, the value of B_2 is determined as in GMP-ECM by the relation $B_2 = \left(\frac{11}{6}B_1\right)^{1.424828748}$. From the graphs, it can be deduced that the following average numbers of curves are needed to achieve probabilities of success of 50%, 90%, 95% and 99%.

p	factor p_1 found			All factors found		
B_1	800	960	1100	800	960	1100
50%	10	8	8	28	22	21
90%	33	26	24	57	42	38
95%	42	34	29	69	50	43
99%	62	48	44	87	69	52

These are, of course, experimental values, based on a sample of 1000 test numbers, and therefore they are subject to sampling error. However, the results clearly show that the probability of success (for both definitions) increases rapidly to about 90% with the number of curves. Beyond 90%, it becomes more costly to further improve the success rate.

It is worthwhile noting that Graphs 1 and 2 can be explained in terms of certain exponential functions. That is, they are approximated by the following equations,

$$\text{Equation of Graph 1} : y = 100(1 - (1 - \lambda)^x),$$
$$\text{Equation of Graph 2} : y = 100(1 - (1 - \lambda)^x)^5,$$

for some constant λ. Recall that, in our experiments, 200-bit integer N is a product of 40-bit random primes p_i, i.e., $N = p_1p_2p_3p_4p_5$. Intuitively, λ can be considered as the probability that a fixed p_i is found after one try of ECM with one curve. Our experiments indicate that, while λ is dependent on B_1 and B_2, it is independent of each value of p_i. We used the following three pairs of $(B_1, B_2) = (800, 32470), (960, 57000), (1100, 51114)$ for experiments. If we use x curves which are chosen randomly, the probability that p_1 (or any fixed p_i) is not

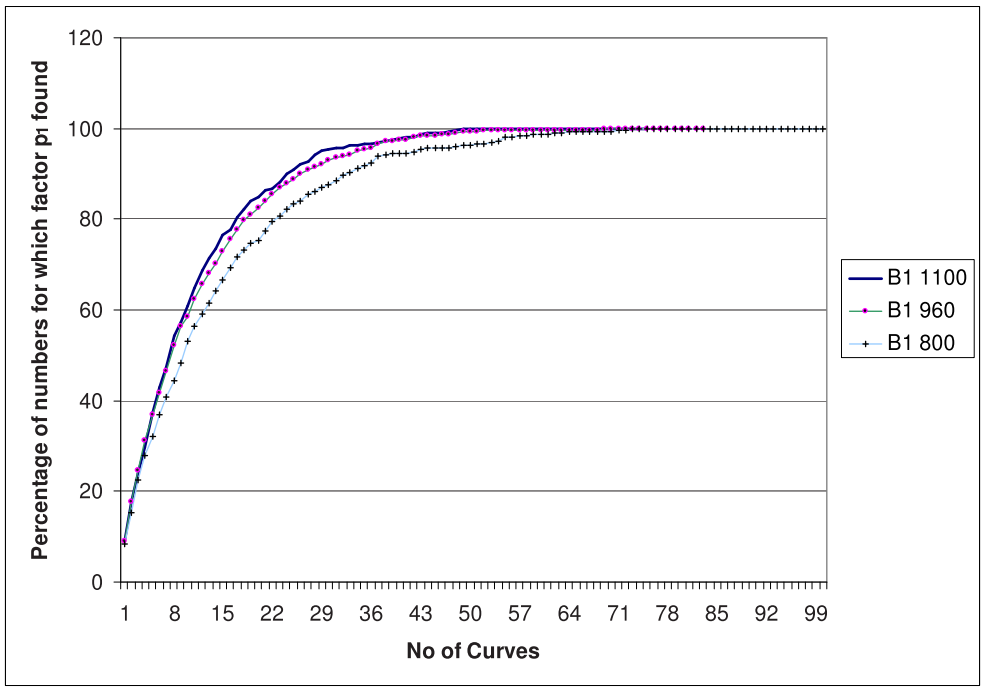

Figure 9. Percentage of the numbers of curves for which the factor p_1 found as a function of the number of curves tried

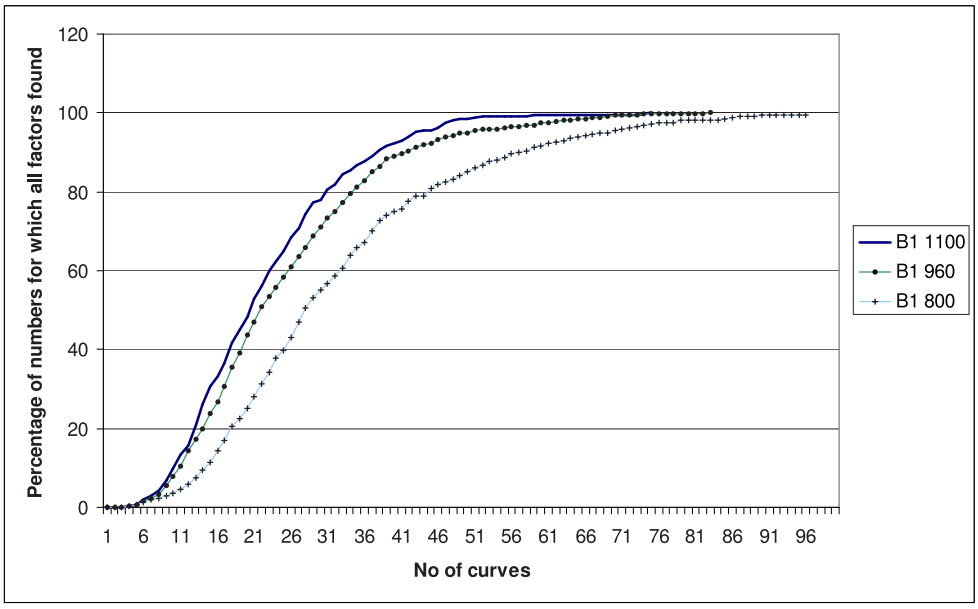

Figure 10. Percentage of the numbers for which all factors were found as a function of the number of the curves tried

found is $(1-\lambda)^x$. Thus $1-(1-\lambda)^x$ is the probability that p_1 is found at least once among the x tries of ECM, and $y = 100(1-(1-\lambda)^x)$ gives the percentage of the numbers for which the factor p_1 is found when x curves are used. Also, since N is a product of five random primes p_i of the same size, the probability that all of p_i are at least once found among the x tries of ECM is $(1-(1-\lambda)^x)^5$. Thus $y = 100(1-(1-\lambda)^x)^5$ gives the percentage of the numbers for which all the five factors p_i are found when x curves are used. For the three values of $B_1 = 800, 960, 1100$, it seems that $\lambda = 0.071, 0.088, 0.092$ give resonably good approximations for the graphs.

6. CONCLUSIONS AND FUTURE WORK

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. [16, 18], 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, optimized architecture for the Montgomery multiplier, the use of two (instead of one) multipliers, and the ability of all arithmetic units (2 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 16.1 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 6.0 for Phase 2.

Our architecture has been implemented targeting four additional families of FPGA devices, including high-performance families (Virtex II and Virex 4), as well as low-cost families (Spartan 3 and Spartan 3E). Our analysis revealed that within the two most recent generations of FPGA families (older: Spartan3, Virtex II; and the most recent: Spartan 3E, Virtex 4), the low-cost devices outperform the high-performance devices in terms of performance to cost ratio by a factor greater than 8.

We have also compared the performance of our hardware architecture implemented using Virtex II XC2V6000-6 and Spartan 3 XC3S5000-5 with an optimized software implementation running on a 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 10, but looses its advantage when the cost of both devices is taken into account. On the other hand, the low-cost FPGA device achieves over an order of magnitude advantage over the same generation Pentium 4 processor in terms of the 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 [26].

Future research directions of our group include the comparison of software and FPGA implementations of ECM with the standard-cell ASICs, estimation and optimization of the time taken by ECM when used as a part of the Number Field Sieve, and porting of our design to two selected general-purpose high-performance reconfigurable computers [27, 28, 30]. As a part of this last project, we will investigate the advantages and drawbacks of using high-level programming languages, such as C and Mitrion-C [27, 29] to describe the behavior of complex cryptanalytical systems in hardware.

ACKNOWLEDGMENT

The authors would like to thank Peter Alfke from Xilinx, Inc., Dan Poznanovic from SRC Computers, Inc., and Amir Zeineddini from SGI, for the insightful materials and discussions, and the members of our extended research team, Deapesh Misra and Chang Shu, for extensive help with typesetting of this paper.

References

[1] P.L. Montgomery, "Modular multiplication without trivial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[2] P.L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, pp. 243–264, 1987.

[3] P.L. Montgomery, "An FFT extension of the elliptic curve method of factorization," *Ph.D. Thesis*, UCLA, 1992.

[4] H.W. Lenstra, "Factoring integers with elliptic curves," *Annals of Mathematics*, vol. 126, pp. 649–673, 1987.

[5] R.P. Brent, "Some integer factorization algorithms using elliptic curves," *Australian Computer Science Communications*, vol. 8, pp. 149–163, 1986.

[6] D.J. Bernstein, "Scaled remainder trees," *preprint*, 2004, http://cr.yp.to/arith/scaledmod-20040820.pdf.

[7] P. Zimmermann, "20 years of ECM," *preprint*, 2005, http://www.loria.fr/∼zimmerma/papers/ecm-submitted.pdf.

[8] B. Dixon and A. Lenstra, "Massively parallel elliptic curve factoring," *Eurocrypt 92*, LNCS 658, pp. 183–193, 1993.

[9] E. Wolski, J.G.S. Filho, and M.A.R. Dantas, "Parallel implementation of elliptic curve method for integer factorization using message-passing interface (MPI')," *13th Symposium on Computer Architecture*, Pirenopolis, Sepetember, 2001.

[10] D. Hankerson, A.J. Menezes, and S.A. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer–Verlag, 2004.

[11] CrypTool: *eLearning Program for Cryptology*, http://www.cryptool.com.

[12] R. Crandall and C. Pomerance, "*Prime Numbers - A Computaional Perspective*," Springer, New York, 2001.

[13] A.K. Lenstra and H.W. Lenstra, "*The Development of the Number Field Sieve*," *Lecture Notes in Mathematics 1554*, Springer, 1993.

[14] J.M. Pollard, "Factoring with cubic integers", *Lecture Notes in Mathematics 1554*, pp. 4-10, Springer, 1993.

[15] 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.

[16] J. Pelzl, J. Simka, 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.

[17] J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, and C. Stahlke, "SHARK - A realizable hardware architecture for factoring 1024-bit composites with the GNFS", *Cryptographic Hardware and Embedded Systems - CHES'05*, LNCS 3659, Springer-Verlag, 2005.

[18] M. Simka, 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.

[19] J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, and C. Stahlke, "SHARK - A realizable hardware architecture for factoring 1024-bit composites with the GNFS", *1st Workshop on Special-purpose Hardware for Attacking Cryptographic Systems - SHARCS'05*, Paris, France, 2005.

[20] J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, M. Simka, and C. Stahlke, "An efficient hardware architecture for factoring large numbers with the elliptic curve method", *1st Workshop on Special-purpose Hardware for Attacking Cryptographic Systems - SHARCS'05*, Paris, France, 2005.

[21] S. Bajracharya, D. Misra, K. Gaj, and T. El-Ghazawi, "Reconfigurable hardware implementation of mesh routing in number field sieve factorization", *Special Purpose Hardware for Attacking Cryptographic Systems, SHARCS'05*, Paris, France, 2005

[22] S. Bajracharya, C. Shu, K. Gaj, and T. El-Ghazawi, "Implementation of elliptic curve cryptosystems over $GF(2^n)$ in optimal normal basis on a reconfigurable computer", *14th International Conference on Field Programmable Logic and Applications - FPL'04*, pp. 1001-1005, Antwerp, Belgium, 2004.

[23] N. Nguyen, K. Gaj, D. Caliga, and T. El-Ghazawi, "Implementation of elliptic curve crypto systems on a reconfigurable computer", *Proc. IEEE International Conference on Field-Programmable Technology - FPT'03*, pp. 60-67, Tokyo, Japan, 2003.

[24] O. D. Fidanci, D. Poznanovic, K. Gaj, T. El-Ghazawi, and N. Alexandridis, "Performance and overhead in a hybrid reconfigurable computer", *Proc. Reconfigurable Architecture Workshop - RAW'03*, 2003.

[25] 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.

[26] 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.

[27] SRC Computers, Inc., http://www.srccomp.com.

[28] SGI, http://www.sgi.com.

[29] Mitrionics AB, http://www.mitrionics.com.

[30] Cray Corp., http://www.cray.com.