Area-Time Efficient Implementation of the Elliptic Curve Method of Factoring in Reconfigurable Hardware for Application in the Number Field Sieve

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

***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, 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 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, without the use of embedded FPGA resources, such as embedded multipliers.**

***Index Terms*— Cipher-breaking, factoring, ECM, FPGA, NFS**

I. Introduction

The fastest known method for factoring large integers is the Number Field Sieve (NFS), invented by Pollard in 1991 [17], [25]. 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 [8]. 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 [13], [17], [24]. Such numbers can be quickly factored and a smoothness test applied. However, because an estimated 10^{10} such factorizations may be 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.

Kris Gaj, Paul Kohlbrenner, and Marcin Rogawski are with the ECE Department, George Mason University, 4400 University Drive, Fairfax, VA 22030, USA, e-mails: {kgaj, pkohlbr1, mrogawsk}@gmu.edu. Soonhak Kwon is with the Department of Mathematics, Sungkyunkwan University, Suwon 440-746, Korea, e-mail: shkwon7@gmail.com. Patrick Baier is with Siemens PLM Software, e-mail: districtline@gmx.net. Hoang Le is with the Department of Electrical Engineering, University of Southern California, 3740 McClintock Ave, Electrical Engineering Building - EEB246, Los Angeles, CA-90089, USA, e-mail: hoangle@usc.edu. Mohammed Khaleeluddin is with Hughes Network Systems, Germantown, MD, USA, e-mail: mdkhaleel@gmail.com. Ramakrishna Bachimanchi is the Thomas Jefferson National Accelerator Facility (Jefferson Lab), e-mail: bachiman@jlab.org. This research was done when all co-authors were associated with George Mason University as faculty members, visiting scholars, or graduate students.

We therefore review existing algorithms which can be used to factor medium-size numbers. Most 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 may be considered as a first step in factoring random numbers. While there are asymptotically faster deterministic methods, in practice these are surpassed by simple probabilistic methods.

Three other probabilistic factoring methods are also of exponential run time, but with a much smaller overhead than the sub-exponential algorithms, 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 (see for example [5] for a general introduction to elementary factoring algorithms).

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 $(\mathrm{mod}\ N)$. 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 [10], [13].

The use of above mentioned smoothness tests has been considered in the context of the quadratic sieve (see 4.15 in [18]), the number field sieve [6], [14], and special purpose factoring hardware (TWIRL [13], or NFS in hardware [2]). Bernstein [3] has pointed out potentially large performance improvements possible from using a combination of these techniques with "early aborts" of less promising candidates.

The contribution of this paper is an implementation in hardware (FPGAs) of the elliptic curve method of integer factoring, originally proposed by H.W. Lenstra [16] in 1987. We describe in detail how to optimize the design and compare our work both to an earlier hardware implementation [24], [27], as well as state-of-the-art software implementation, GMP-ECM [9], [31].

II. Elliptic Curve Method

Let K be a field with characteristic different from 2 and 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 $(\mathrm{mod}\ q)$. An elliptic curve E over K is defined as a set of points $(X, Y) \in K^2$

satisfying

$$Y^2 = X^3 + AX + B, \tag{1}$$

where $A, B \in K$, $4A^3 + 27B^2 \neq 0$, 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,

$$Y^2Z = X^3 + AXZ^2 + BZ^3. \tag{2}$$

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 [21] studied elliptic curves of the form, $E : by^2 = x^3+ax^2+x$, which allows a more efficient implementation of 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

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

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. The choice of parameters a and b for the given above curve can be simplified using Suyama's parametrization, which expresses a, b, and the coordinates (x, y, z) of a point on the curve P, as a function of a single parameter σ, as described in detail in [31].

Let N be a composite integer we want to factor. The ECM Method [4], [21], [31] considers elliptic curves in Montgomery form (3), 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.

A. ECM Algorithm

The Elliptic Curve Method (ECM) was originally proposed by Lenstra [16] and subsequently extended by Brent [4] and Montgomery [21]. 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 pseudo code of both phases is given below as Algorithm (1). Recall that an integer is called B-smooth (or simply smooth if the value of B is implicit) if it has no prime divisors exceeding B.

Let q be an unknown factor of N. For any point P_0 belonging to the curve E, we have $|E(\mathbb{Z}_q)|P_0 = O$, where $|E(\mathbb{Z}_q)|$ is the order of the curve E, i.e., the number of points on the curve E with operations performed $\pmod q$. This order might be a smooth number, and we have a good chance of finding an integer $k \in \mathbb{Z}$ (by multiplying many small primes) so that $k = l \cdot |E(\mathbb{Z}_q)|$ for some l. 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 [21], [22] and Brent [4] 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 p 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 [16] $O(\exp((\sqrt{2}+o(1))\sqrt{\log q \log\log q})M(N))$. 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.

Example: We want to factor $N = 40586929$, using $B_1 = 20$ and $B_2 = 50$. Suyama's parametrization with $\sigma = 6$ gives us the point $P_0 = (29791 : 48335 : 13824)$ on the curve $E = 3150302y^2z = x^3 + 3371272x^2z + xz^2 \pmod N$. The product of maximal prime powers below $B_1 = 20$ is $k = 2^4 \cdot 3^2 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 = 232792560$. In Phase 1, we compute $Q_0 = kP_0 = (x_{Q_0} :: z_{Q_0}) = (3177782 :: 33732517)$ but do not succeed in recovering any factors from N since $\gcd(z_{Q_0}, N) = 1$ $(= \gcd(x_{Q_0}, N))$. (It can be shown that $y_{Q_0} = 37451505$, but the implementation described in Algorithm 3 does not compute the y-coordinates at all). In Phase 2 we compute $pQ_0 = (x_{pQ_0} :: z_{pQ_0})$ for all large primes p in the range $B_0 = 20 < p \leq B_1 = 50$, i.e., $p \in \{23, 29, 31, 37, 41, 43, 47\}$, and set $d = \prod_{B_0<p\leq B_1} z_{pQ_0} \mod N$. We find $d = 20600066$, and the Euclidean algorithm reveals a factor $q' = \gcd(d, N) = 8887$. With $q'' = N/q' = 4567$ we have found a factorization $N = 8887 \cdot 4567$, with both factors easily shown to be prime.

In this simple setting we can have a glance at what happens inside the algorithm. Reducing the curve E and the point P_0 modulo the two factors $q' = 8887$ and $q'' = 4567$ we get two elliptic curves $E' = 4304y^2z = x^3 + 3099x^2z + xz^2$ and $E'' = 3639y^2z = x^3 + 826x^2z + xz^2$ with points $P_0' = (3130 : 550 : 4937)$ and $P_0'' = (2389 : 666 : 123)$ on them. Schoof's algorithm can be used to show that the number of points on these curves is $|E'/\mathbb{F}_{q'}| = 8928$ and $|E''/\mathbb{F}_{q''}| = 4572$, where $8928 = 2^5 \cdot 3^2 \cdot 31$ and $4572 = 2^2 \cdot 3^2 \cdot 127$. Here we see the divisor 12 of the group orders due to the fact the Suyama curves have a torsion subgroup of order 12. Moreover, an explicit calculation shows that $o' = \mathrm{ord}[P_0' : E'] = 1116 = 2^2 \cdot 3^2 \cdot 31$ and $o'' = \mathrm{ord}[P_0'' : E''] = 762 = 2 \cdot 3 \cdot 127$. (As an aside, this again reveals the orders of E' and E'' because the only multiple of o' in the Hasse interval $[q' + 1 - 2\lfloor\sqrt{q'}\rfloor, q' + 1 + 2\lfloor\sqrt{q'}\rfloor] = [8699, 9076]$ is $8o' = 8928$, and the same argument works for E''.) From the definition of k, we see that $Q_0 = kP_0$ is of order 31 over E' and of order 127 over E''. In both cases, the order of Q_0 is prime, but because $31 < B_2 < 127$, for $p = 31$ in Phase 2 we find $31Q_0 = O_{E'}$ over E', while $pQ_0 \neq O_{E''}$ for all $p < B_2$ over E''. Thus q', but not q'' shows up as a factor in d and we are able to recover the divisor $8887 = \gcd(N, d)$ in Phase 2.

B. Operations on an Elliptic Curve

The hierarchy of major operations used in the ECM algorithm is shown in Figure 1. Scalar multiplication, kP, is the basic elliptic curve operation used in ECM.

An efficient algorithm for computing scalar multiplication was proposed by Montgomery [21] in 1987, and is known as the Montgomery ladder algorithm. This algorithm is especially useful when an elliptic curve is expressed in Montgomery form (see Eq. (3)), 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, except for the sign, 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.

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**

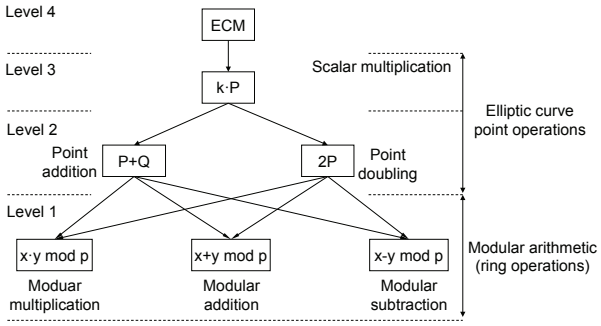

Fig. 1. Hierarchy of Elliptic Curve Method Operations

Algorithm 2 Montgomery Ladder Algorithm

Require: $P_0 = (x_0 : : z_0)$ on E with $x_0 \neq 0$, an s-bit positive integer $k = (k_{s-1}k_{s-2} \cdots k_1 k_0)_2$ 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

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 pseudo code of the Montgomery ladder algorithm is shown as Algorithm 2, and its basic step is defined in detail as Algorithm 3. 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.

Algorithm 3 Addition and Doubling using 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. (3)

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: $z_{2P} \leftarrow (4x_P z_P)\left((x_P - z_P)^2 + a_{24} \cdot (4x_P z_P)\right)$

A careful analysis of the 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.

C. 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 [20] 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.

Despite the initial conversion cost, if we do many Montgomery multiplications followed by an inverse conversion from the Mont-

gomery domain back to the ordinary domain, as in RSA, we obtain an advantage over ordinary multiplication. In fact, in the ECM method, the inverse conversion is not necessary because $\gcd(X', N) = \gcd(X2^n \pmod N, N) = \gcd(X, N)$ for an arbitrary X, and odd N.

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: $\quad q_i \leftarrow S[i] + X_i Y_0 \pmod 2$
4: $\quad S[i+1] \leftarrow (S[i] + X_i Y + q_i N) \text{ div } 2$
5: **end for**
6: return $S[n]$

Algorithm 4 uses an improvement over Montgomery's original method which avoids the need for an additional conditional subtraction at the end, see [30] and [1]. This improvement was first developed with the goal of avoiding the side-channel risks inherent in conditional statements, but it also improves performance in hardware which makes it beneficial in our context. Algorithm 4 shows the pseudo code 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 [20] originally used. This modified algorithm makes all the inputs and output in the same range, i.e., $0 \le X, Y, S[n] < 2N$. Therefore it is possible to implement Algorithm 4 repeatedly without any reduction unlike the original algorithm [20], 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.

D. 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 [21] and modify it appropriately for efficient FPGA implementation. Choose $0 < D < B_2$, and let every prime p, $B_1 < p \le B_2$, be expressed in the form

$$p = mD \pm j \tag{4}$$

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 $J_S = \{j : 1 \le j \le \lfloor \frac{D}{2} \rfloor, \gcd(j, D) = 1\}$, of the size of $\phi(D)/2$, and possible values of m form a set $M_T = \{m : M_{MIN} \le m \le M_{MAX}\}$, 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 $\gcd(d, N) > 0$, where

$$d = \prod_{m,j} (x_{mDQ_0} z_{jQ_0} - x_{jQ_0} z_{mDQ_0}). \tag{5}$$

In order to speed up these computations, one precomputes one of the sets $S = \{jQ_0 : j \in J_S\}$ or $T = \{mDQ_0 : m \in M_T\}$.

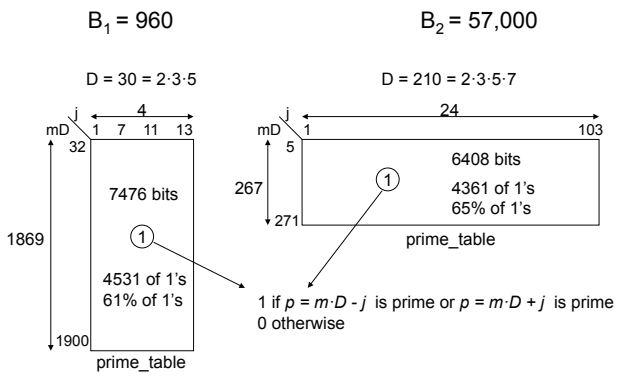

Fig. 2. Dimensions of prime_table and the number of 1's in this table for $D = 30$ and $D = 210$

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. (5) 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:

$$\text{prime\_table}[m, j] = \begin{cases} 1 & \Leftrightarrow \quad mD + j \text{ or } mD - j \text{ is prime} \\ 0 & \Leftrightarrow \quad \text{else.} \end{cases}$$

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:

$$\text{GCD\_table}[j] = \begin{cases} 1 & \Leftrightarrow \quad j \in J_S \\ 0 & \Leftrightarrow \quad \text{else.} \end{cases}$$

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, for the case of even D, is given in Algorithm 5. 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. As a result, they minimize the amount of memory storage and computations required for Phase 2. In Figure 2, we show the dimensions of prime_table and the number of 1's in this table for these choices of D. The total size of prime_table in bits determines the memory requirements of the implementation, while the number of 1's in the table affects the computation time of Phase 2.

E. 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 $B_1 \approx e^{\sqrt{\frac{1}{2}\log q \log\log q}}$ [16], where log is the natural logarithm. However the precise value of $o(1)$ term is difficult to estimate. The choice of the bound B_1 is closely related with the Dickman-de Brujin function $\rho(u)$ [22], 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 sets B_2 between $50B_1$ and $100B_1$ depending on the computational resources for Phase 2. For example, Šimka et al. [27] choose $B_1 = 960$ and $B_2 = 57000$ to find a 40-bit prime divisor of 200-bit integers. By

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, 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:

$M_{MIN} \leftarrow \lfloor (B_1 + \frac{D}{2})/D \rfloor$
$M_{MAX} \leftarrow \lceil (B_2 - \frac{D}{2})/D \rceil$
clear GCD_table, clear J_S
for each $j = 1$ to $\frac{D}{2}$ step 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 $\frac{D}{2}$ step 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

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 [27] 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 curves), but at the same time, increase the execution time per curve of these phases.

A theoretical analysis of the optimal parameter choices is given in [26], 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 comparison with Šimka et al. [24], [27].

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 $D + B_2/D \approx \sqrt{B_2}$. 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 always 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. The larger D, the larger the amount of Precomputations in Algorithm 5, but the smaller M_N, and thus the smaller number of iterations of the outer loop during Main computations in Algorithm 5.

III. ECM Architecture

A. Top-level view: ECM units

Our ECM system consists of multiple ECM units working independently in parallel, as shown in Figure 3. 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) The number to be factored, N, the coordinates of the starting point P_0, and the parameter a_{24} which depends on the coeffi-

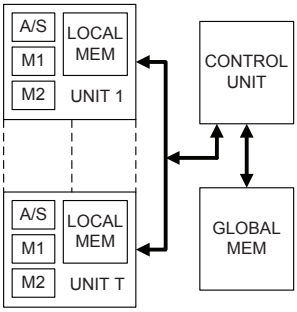

Fig. 3. Block diagram of the top-level unit. Notation: MEM-memory; $M1, M2$-multipliers 1 and 2; A/S-adder/subtractor.

cient 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 4a.

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.

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, $q_i = \gcd(z_{Q_0}, N)$. If $q_i = 1$, no factor was found by a given ECM unit. If $q_i > 1$ and $q_i \neq N$, then a non-trivial factor of N, q_i, was found. If q_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, GCD_table, and prime_table, as defined in Section II-D.

The contents of the global memory after initialization for Phase 2 is shown in Figure 4b. 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. (5)). 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. 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. Additionally, software and hardware computations can be done in parallel.

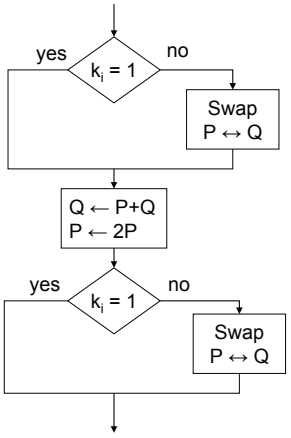

Fig. 5. Implementation of a basic step of the Montgomery ladder algorithm.

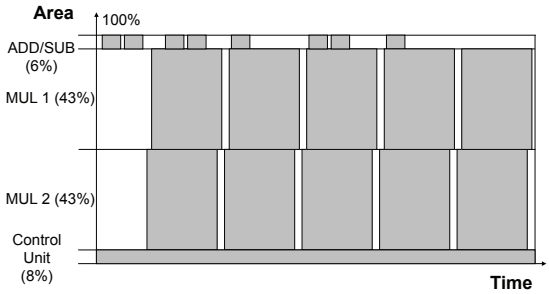

Fig. 6. Utilization of resources as a function of time during the execution of Phase 1.

B. Medium-level View: Operations of the ECM Unit

1) Medium-level operations: The primary operation constituting Phase 1 of ECM is a scalar multiplication $Q_0 = kP_0$. As discussed in Section II-B, this operation can be efficiently implemented in projective coordinates using Algorithm 2.

The two branches of the if statement in Algorithm 2 can be calculated using exactly the same sequence of instructions, with a conditional swap of input and output variables, as shown in Figure 5.

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 I. 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 two multiplications. As a result, as shown in Figure 6, we obtain over 90% utilization of the area $\times$ time space, which is crucial from the point of view of minimizing the area $\times$ time product.

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.

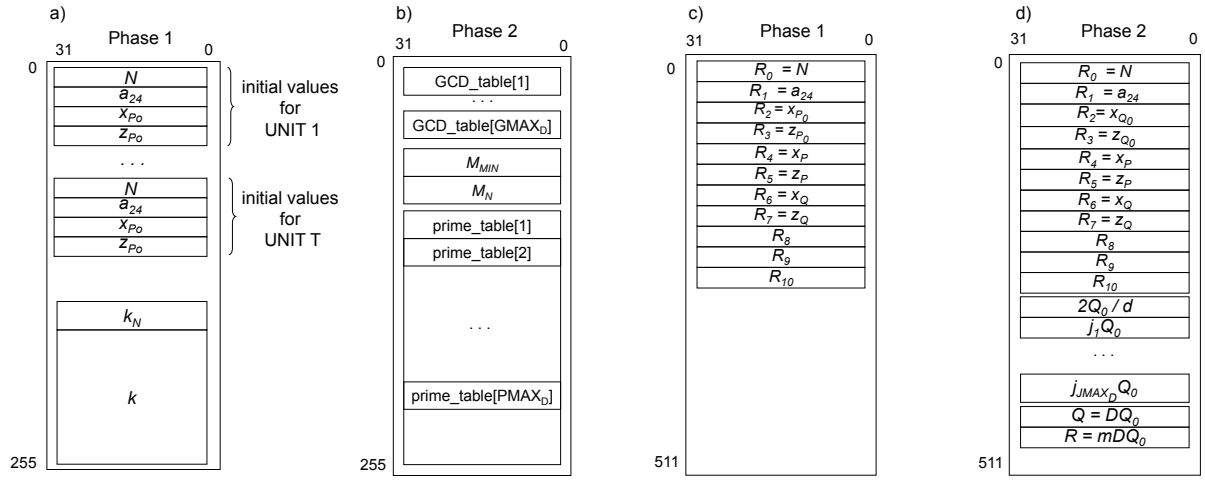

Fig. 4. Contents of the global memory in a) Phase 1, b) Phase 2. Contents of the local memories in c) Phase 1, d) Phase 2.

TABLE I

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: $z_{2P} = m_{10} = s_3 \cdot a_4$

In Phase 2, the initial computation

$$D \cdot Q_0 \quad \text{and} \quad M_{MIN} \cdot (D \cdot Q_0) \tag{6}$$

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 II.

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 III.

Finally, the last medium level operation required in Phase 2 is the accumulation of the product d as defined in Eq. (5). We can rewrite the expression for d as

$$d \equiv \prod_{i,n} d_{in} \equiv \prod_{i,n} (x_n z_i - x_i z_n) \pmod{N} \tag{7}$$

where

$$(x_i, z_i) \in \{(x,z)\colon (x,z) = jQ_0\}, \tag{8}$$
$$(x_n, z_n) \in \{(x,z)\colon (x,z) = mDQ_0\} \tag{9}$$

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

TABLE IV

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 IV, 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.

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 4c and 4d, 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 3.

TABLE II

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: $x_{P+Q} = m_{11} = m_7 \cdot z_{P-Q}$

TABLE III

ADDITION OF POINTS $P+Q$

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

C. 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 II-C), 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**

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**

The block diagram of the adder/subtractor unit implementing both algorithms is shown in Figure 7. 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 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

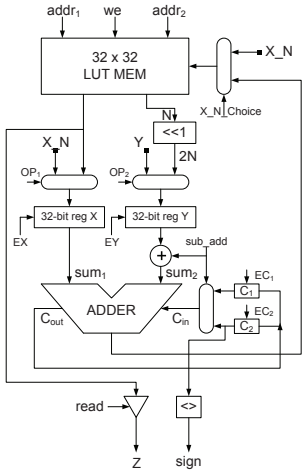

Fig. 7. Block diagram of the adder-subtractor

(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 II-C. 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 8b, 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 $0 \le X, Y < 2N$

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: $\quad q_i \leftarrow (C[i]_0 + S[i]_0 + X_i \cdot Y_0) \pmod 2$
5: $\quad (C[i+1], S[i+1]) \leftarrow CSA(C[i], S[i], X_i \cdot Y, q_i \cdot N) \text{ div } 2$
6: **end for**
7: $C = 0$
8: **for** $j = 0$ to 7 **do**
9: $\quad (C, Z^{(j)}) \leftarrow C[n]^{(j)} + S[n]^{(j)} + C$
10: $\quad$ return $Z^{(j)}$
11: **end for**

The modified algorithm, based on carry save addition (CSA) is shown as Algorithm 8. This algorithm has been described earlier in [19]. The block diagram of the circuit implementing Algorithm 8 is shown in Figure 8a. 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.

IV. 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-ECM [9], [31].

In Table V, 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 4cd. 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 $1 \le j \le \lfloor D/2 \rfloor$, 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. In the older families of Xilinx FPGAs, such as Virtex, the size of a single Block RAM is equal to 4 kbits, which translates to the memory of the size 256x16 bits or 512x8 bits. As a result, a larger number of BRAMs is required to implement local memory for Phases 1 and 2, as shown in the last but one column of Table V.

Global memory is the memory used by the global control unit, and its map is shown in Figure 4ab. 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, as shown in Figure 2. 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.

The execution times of Phase 1 and Phase 2 in the ECM hardware architecture are shown in Table VI. 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

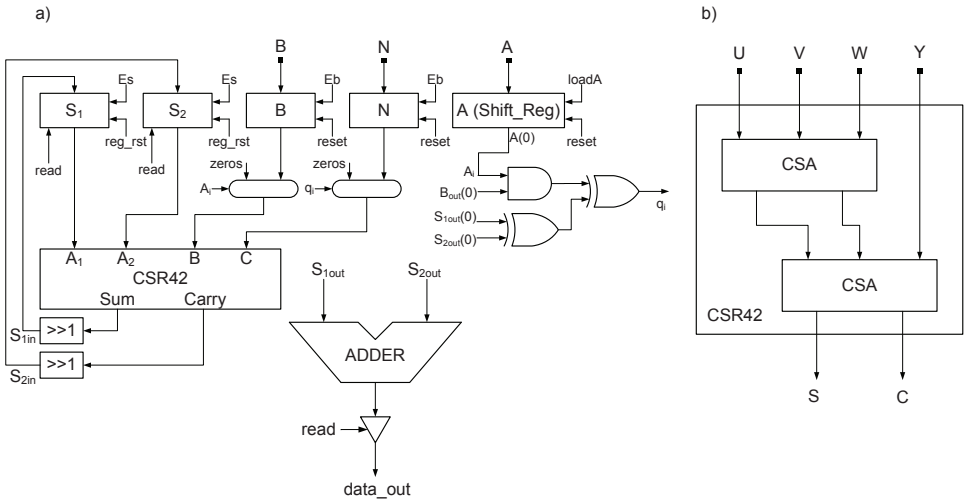

Fig. 8. a) Block diagram of a Montgomery multiplier; b) a cascade of two carry save adders

TABLE V
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**

TABLE VI

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 (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%)**

TABLE VII

EXECUTION TIME OF PHASE 1 AND PHASE 2 USING SRC 6 RECONFIGURABLE COMPUTER HOLDING 9 ECM UNITS FOR 198-BIT NUMBERS N; $B_1 = 960$ (WHICH IMPLIES NUMBER OF BITS OF k; $k_N = 1375$), $B_2 = 57000$, AND $D = 210$

	Phase 1	**Phase 2**	**Phases 1 & 2**
One-time pre-computations			
Pre-computations by the microprocessor (common to all integers to be factored)	2,249 μs		
Generation of 9 integers to be factored			
Generation of numbers to be factored	7,902 μs		
ECM computations for one set of 9 integers to be factored			
Pre-computations by the microprocessor (specific to a given set of integers to be factored)	1,368 μs	0 μs	1,368 μs
Transfer of data-in (from the microprocessor memory to the on-board-memory of the FPGA-based processor	51 μs	0 μs	51 μs
Calculations performed by the FPGA-based processor	17,136 μs	19,152 μs	36,289 μs
Transfer of data-out (from the on-board-memory of the FPGA-based processor to the microprocessor memory)	19 μs	19 μs	19 μs
Function call overhead (overhead associated with the transfer of control between the microprocessor and the FPGA)	252 μs	252 μs	252 μs
Post-computations by the microprocessor (final GCD computation)	81 μs	81 μs	81 μs
Total end-to-end execution time	18,907 μs	19,504 μs	38,060 μs
Percentage of the total end-to-end execution time			
Pre-computations and postcomputations by the microprocessor	8.03%	0.51%	3.99%
Function call and data transfer overheads	1.33%	1.29%	0.66%
FPGA board computations	90.63%	98.20%	95.35%

TABLE VIII

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**

TABLE IX

RESULTS OF THE FPGA IMPLEMENTATIONS: RESOURCES AND TIMING FOR THE MAXIMUM NUMBER OF ECM UNITS PER FPGA DEVICE. EXECUTION TIME OF PHASE 1 AND PHASE 2 FOR 198-BIT NUMBERS N, $B_1 = 960$, $B_2 = 57,000$, $D = 210$

Results	**Virtex XCV2000E-6**	**Virtex II XC2V6000-6**	**Spartan 3 XC3S5000-5**	**Spartan 3E XC3S1600E-5**	**Virtex 4 XC4VLX200-ll**
Max # of ECM units	**7**	**13**	**13**	**5**	**24**
- CLB slices	**18,756 (94%)**	**33,790 (99%)**	**32,278 (99%)**	**13,915 (94%)**	**63,220 (70%)**
- LUTs	**28,976 (73%)**	**53,970 (79%)**	**53,880 (80%)**	**21,703 (71%)**	**98,332 (55%)**
- FFs	**19,270 (50%)**	**35,146 (52%)**	**35,141 (52%)**	**14,092 (47%)**	**65,052 (36%)**
- BRAMs	**32/160**	**14/144**	**14/104**	**6/36**	**26/336**
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**
Max clock frequency	**48 MHz**	**120 MHz**	**80 MHz**	**96 MHz**	**104 MHz**
Max clock frequency for single ECM unit	**54 MHz**	**123 MHz**	**100 MHz**	**98 MHz**	**135 MHz**
Time for Phase 1 and 2	**75.5** ms	**30.2** ms	**45.3** ms	**37.7** ms	**35.19** ms
# of ECM computations/s	**93 ECM operations/s**	**430 ECM operations/s**	**287 ECM operations/s**	**133 ECM operations/s**	**682 ECM operations/s**
# of ECM computations/s per $100	**8 ECM operations/s per $100**	**16 ECM operations/s per $100**	**221 ECM operations/s per $100**	**380 ECM operations/s per $100**	**22 ECM operations/s per $100**

[a]cost per unit for a batch of 10,000+ devices as of Nov. 2006

TABLE X

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	**120 MHz**	**80 MHz**	**2.8 GHz**	
No. of parallel ECM computations	**13**	**13**	**1**	
Time of Phase 1	**14.2** ms	**21.3** ms	**18.3** ms	**11.3** ms
Time of Phase 2	**15.9** ms	**24** ms	**18.6** ms	**13.5** ms
Time of Phase 1 & Phase 2	**30.2** ms	**45.3** ms	**36.9** ms	**24.8** ms
# of Phase 1 computations per second	**915**	**610**	**55**	**89**
# of Phase 2 computations per second	**818**	**542**	**54**	**74**
# of Phase 1 & 2 computations per second	**430**	**287**	**27**	**40**

TABLE XI

RESULTS OF THE ASIC IMPLEMENTATIONS USING SYNOPSYS 90 nm GENERIC LIBRARY FOR TEACHING IC DESIGN

Number of ECM units	**1**	**2**	**5**	**10**	**13**	**20**	**24**
Clk frequency in MHz	**350**	**333**	**325**	**300**	**275**	**250**	**225**
Area in $\mathbf{mm}^2$	**1.119**	**1.691**	**2.900**	**5.156**	**6.483**	**9.762**	**11.474**
Area in gate equivalents	**202,733**	**305,875**	**524,536**	**932,403**	**1,172,505**	**1,765,422**	**2,074,976**

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 VI. For $D = 210$, the largest contribution to Phase 2, around 80%, comes from the calculation of the accumulated product d.

In order to estimate the overhead associated with the transfer of control and data between a microprocessor and an FPGA, an ECM system with 9 ECM units has been ported to the reconfigurable computer SRC 6 from SRC Computers [28], based on 2.8 GHz Xeon microprocessors and Xilinx Virtex II XC2V6000-4 FPGAs running at a fixed clock frequency of 100 MHz. The execution times for all phases of the ECM computations performed using this reconfigurable computer are summarized in Table VII. The data transfer and function call overheads have been experimentally measured to be less than 1% for the combined Phase 1 and Phase 2 computations. The precomputations and postcomputations by the microprocessor amounted to about 4% of the total execution time of the combined Phases 1 and 2, and their overhead can be practically eliminated by overlapping computations in the FPGA and the microprocessor.

In Table VIII, we compare our ECM architecture to an earlier design by Pelzl, Šimka, et al., presented at SHARCS 2005, and described in subsequent publications [24], [27]. 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 [24], [27], 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 [24], [27] 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 $9.3 \cdot 2.33 \approx 22$ 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 $9.3/2.7 \approx 3.4$ for Phase 1 and 5.6 for Phase 2.

In Table IX, 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 13.8 and 14.1 respectively, and thus are more suitable for cost effective code breaking computations.

Thus, assuming that only CLB slices and block RAMs are used for computations, low-cost FPGAs, such as Spartan and Spartan 3E are more cost-effective for the implementation of ECM. The

situation substantially changes when embedded FPGA multipliers are employed for the implementation of the most time consuming operation of ECM, Montgomery modular multiplication. These multipliers are present in both low-cost and high-performance FPGA devices, but their number and the maximum clock frequency is greater for high-performance FPGAs. In [23], [12], preliminary results for two alternative designs based on the use of embedded multipliers in Virtex 4 SX FPGAs are presented. In both papers, only the implementation of Phase 1 of ECM is reported. Both papers demonstrate a potential for a substantial improvement in terms of the throughput and the throughput to cost ratio based on the use of embedded multipliers. However, neither paper attempts to quantify the difference between the performance of high-end FPGAs and low-cost FPGAs under the new assumption that all embedded multipliers can be employed in ECM computations for both kinds of FPGAs. In order to answer this question, a comprehensive analysis would need to be performed, and the effect of a novel pipelined hardware architecture employed in [23], would need to be separated from the effect of employing embedded multipliers. This analysis is beyond the scope of this paper, and is proposed as an interesting future study.

In Table X, 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 [9], [31]. 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 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 13 ECM units, its overall performance is about 11 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.

Further gains in terms of absolute performance and cost-effectiveness of the hardware implementation of ECM can be reached by employing standard-cell ASIC technology. In order to estimate this effect, we have ported our VHDL code to Synopsys 90nm Generic Library for Teaching IC Design [29]. Synopsys Design Compiler v. A-2007.12-SP4 was used for synthesis and static timing analysis. The results of our implementation are summarized in Table XI. The gain in terms of the clock frequency by a factor of 3.44 was demonstrated, for a circuit composed of 13 ECM units, when compared to Spartan 3, implemented using similar $90\ nm$ technology. This gain is consistent with the ASIC to FPGA speed gains reported earlier in [15] for a representative set of digital system benchmarks, representing areas of cryptography, DSP, and communications.

While there is considerable uncertainty about the cost of factoring a 1024 bit RSA number, it might be interesting to project our results to such an effort. The SHARK device proposed by Franke et. al. [10] was designed to complete the sieving for the factorization of a 1024 bit RSA integer in one year at a cost of $200 million (in 2005). SHARK uses 2300 identical machines built with conventional ASICs and a special transport system for communication. Like the ECM architecture proposed here, it uses technologies available today. In one year, SHARK produces $1.7 \cdot 10^{14}$ sieving reports which need to be processed by a factoring device such as the one proposed here for smoothness testing and to obtain a complete factorization.

Two of the FPGA families shown in Table IX, Virtex 4 and Spartan 3E, can perform 682, respectively 133 ECM operations per second, or $2.15 \cdot 10^{10}$, respectively $4.19 \cdot 10^{9}$ such operation per year. If we perform 20 ECM operations on each sieving report, we need about 158,086 Virtex 4 FPGAs, or 810,626 Spartan 3Es to process the $1.7 \cdot 10^{14}$ sieving reports generated by SHARK in one year. The price for those is about $492 million, respectively $29 million.

Thus, combining the results of SHARK with our architecture, we estimate the total cost for the relation collection step to be $229 million to finish in one year, using the most cost efficient FPGA technology (Spartan 3E). In case ASICs are used for implementing ECM, the product of area by time can be reduced by a factor of over 100, as shown for a representative suite of basic data processing units in [15]. In this case, the cost of the ECM part (around $290k plus about $1 million of non-recurring costs associated with preparing ASIC masks) would become an almost insignificant fraction of the cost required for the sieving step of NFS.

V. 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. [24], [27] include the following

a) 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)
b) Substantially smaller memory requirements, an optimized architecture for the Montgomery multiplier
c) The use of two (instead of one) multipliers,
d) 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 four additional families of FPGA devices, including high-performance families (Virtex II and Virtex 4), as well as low-cost families (Spartan 3 and Spartan 3E). Our analysis revealed that within the two recent generations of FPGA families (older: Spartan3, Virtex II; and the more recent: Spartan 3E, Virtex 4), the low-cost devices outperform the high-performance devices in terms of performance to cost by a factor of ≈ 14.

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 11, 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 the performance to cost ratio.

All of the described results have been reached under the assumption that only CLB slices and Block RAMs are used in computations. Future work will include the comprehensive analysis of influence of embedded multipliers and DSP units on the results of ECM implementations using both low-cost and high-performance FPGAs. Additionally, the choice of optimum hardware architectures (e.g. iterative [11] vs. fully pipelined [23]), best matching all resources available in modern FPGAs and standard-cell ASICs will be further investigated.

ACKNOWLEDGMENT

The authors would like to thank Peter Alfke and Amir Zeineddini from Xilinx, Inc., and Dan Poznanovic from SRC Computers, Inc. 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] L. Batina and G. Muurling, Montgomery in Practice: How to Do It More Efficiently in Hardware, *Topics in Cryptology - CT-RSA - The Cryptographers' Track at the RSA Conference*, Lecture Notes in Computer Science, vol. 2271, pp. 40-52, 2002.

[2] D.J. Bernstein. Circuits for integer factorization: a proposal. `http://cr.yp.to/papers.html#nfscircuit`

[3] D.J. Bernstein. `http://cr.yp.to/talks/2004.07.29/slides.pdf`

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

[5] H. Cohen. *A Course in Computational Algebraic Number Theory* Springer Graduate Texts in Mathematics, 2nd ed., 1993.

[6] D. Coppersmith, Modifications to the number field sieve, Journal of Cryptology **6** (1993) 169-180.

[7] R. Crandall and C. Pomerance, *Prime Numbers - A Computational Perspective*, Springer, New York, 2001.

[8] F. Bahr, M. Boehm, J. Franke, T. Kleinjung, Factorization of RSA-200, `http://crypto-world.com/announcements/rsa200.txt`.

[9] 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.`

[10] 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.

[11] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, R. Bachimanchi, Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware, *Proc. Special Purpose Hardware for Attacking Cryptographic Systems, SHARCS 2006*, Cologne, Germany, Apr. 3-4, 2006.

[12] T. Güneysu, C. Paar, G. Pfeiffer, M. Schimmler, Enhancing COPACOBANA for advanced applications in cryptography and cryptanalysis, *International Conference on Field Programmable Logic and Applications, FPL 2008*, pp. 675-678, 8-10 Sept. 2008.

[13] 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`.

[14] R. Golliver, A. K. Lenstra, and K. McCurley. Lattice sieving and trial division. Proceedings of the ANTS-I conference, Lecture Notes in Computer Science, vol. 877, Springer-Verlag, 1994, pp. 18-27.

[15] I. Kuon and J. Rose, Measuring the Gap Between FPGAs and ASICs, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 62, no. 2, Feb 2007.

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

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

[18] A.K. Lenstra, H.W. Lenstra, Jr., Algorithms in number theory, chapter 12 in Handbook of theoretical computer science, Volume A, algorithms and complexity (J. van Leeuwen, ed.), Elsevier, Amsterdam (1990)

[19] 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.

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

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

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

[23] G. de Meulenaer, F. Gosset, G.M. de Dormale, J.-J. Quisquater, Integer Factorization Based on Elliptic Curve Method: Towards Better Exploitation of Reconfigurable Hardware, *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2007*, pp.197-206, 23-25 April 2007.

[24] J. Pelzl, J. 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.

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

[26] 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.

[27] 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.

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

[29] Synopsys 90nm Generic Library for Teaching IC Design, available at `http://www.synopsys.com/Community/UniversityProgram/Pages/Library.aspx`

[30] C.D. Walter, Precise Bounds for Montgomery Modular Multiplication and some Potentially Insecure RSA Moduli, *Topics in Cryptology - CT-RSA - The Cryptographers' Track at the RSA Conference*, Lecture Notes in Computer Science, vol. 2271, pp. 30-39, 2002.

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

Kris Gaj received the M.Sc. and Ph.D. degrees in electrical engineering from Warsaw University of Technology in Warsaw, Poland. In 1998, he joined George Mason University, where he currently works as an Associate Professor, doing research and teaching courses in the area of cryptographic engineering and reconfigurable computing. His research projects center on new hardware architectures for secret key ciphers, hash functions, public key cryptosystems, and factoring; benchmarking of cryptographic hardware, as well as development of specialized libraries and application kernels for high-performance reconfigurable computers.

Soonhak Kwon received the B.Sc. degree in mathematics from KAIST (Korea Advanced Institute of Science and Technology), Korea, in 1990 and the Ph.D. degree in mathematics from The Johns Hopkins University, Baltimore, MD, in 1997. Since 1998 he has been with the Department of Mathematics, Sungkyunkwan University, Korea, where he is now an Associate Professor. His current research interests include computational number theory, cryptographic hardware, and efficient implementation of elliptic curve cryptography.

Patrick Baier studied mathematics in Freiburg, Germany, and Cambridge and Oxford, UK, where he received his D.Phil. in 2001. He is currently working for Siemens PLM Software and develops cryptographic systems for various health care companies in the US.

Mohammed Khaleeluddin received his B.Tech degree in Electronics and Communication Engineering from Jawaharlal Nehru Technological University, Hyderabad, AP, India in May 2003 and the Masters degree in Electrical Engineering from George Mason University, Fairfax VA in August 2006. He currently works at Hughes Network Systems, Germantown, MD.

Paul Kohlbrenner received the BS from Carnegie Mellon University, Pittsburgh, Pennsylvania in 1984 and the MS in Computer Engineering from George Mason University, Fairfax, Virginia in 2004. He is currently working towards a Ph.D. degree at George Mason University. He is also a member of the professional staff at Pinnacle Solutions LLC in Fairfax, Virginia where he specializes in computer systems engineering and information security. His research interests include network security, emergent systems design, and multi-agent systems engineering. He has been a member of the IEEE since 1987.

Ramakrishna Bachimanchi received the B.Engg degree in electronics and communication engineering from Osmania Unversity, India, in 2004 and MS degree in computer engineering from George Mason University in 2007. He is currently with Thomas Jefferson National Accelerator Facility (Jefferson Lab).

Hoang Le received the BS and MS degrees in computer engineering from George Mason University, USA, in 2005 and 2007, respectively. He is currently a Ph.D candidate at University of Southern California, USA.

Marcin Rogawski received the MS degree in computer science from Military University of Technology, Poland, in 2003. In 2008, he joined George Mason University, where he currently works toward his PhD, doing research in the area of cryptographic engineering and reconfigurable computing.