Hardware Factorization Based on Elliptic Curve Method

Martin Šimka[1], Jan Pelzl[2], Thorsten Kleinjung[3],
Jens Franke[3], Christine Priplata[4], Colin Stahlke[4],
Miloš Drutarovský[1], Viktor Fischer[5], Christof Paar[2]

[1] Department of Electronics and Multimedia Communications, Technical University of Košice,
Park Komenského 13, 04120 Košice, Slovak Republic
{Martin.Simka,Milos.Drutarovsky}@tuke.sk

[2] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
{cpaar,pelzl}@crypto.rub.de

[3] University of Bonn, Department of Mathematics, Beringstraße 1, D-53115 Bonn, Germany
{thor,franke}@math.uni-bonn.de

[4] EDIZONE GmbH, Siegfried–Leopold–Straße 58, D-53225 Bonn, Germany
{priplata,stahlke}@edizone.de

[5] Laboratoire Traitement du Signal et Instrumentation, Unité Mixte de Recherche CNRS 5516,
Université Jean Monnet, 10, rue Barrouin, 42000 Saint-Etienne, France
fischer@univ-st-etienne.fr

Abstract

The security of the most popular asymmetric cryptographic scheme RSA depends on the hardness of factoring large numbers. The best known method for factorization large integers is the General Number Field Sieve (GNFS). Recently, architectures for special purpose hardware for the GNFS have been proposed [5, 12]. One important step within the GNFS is the factorization of mid-size numbers for smoothness testing, an efficient algorithm for which is the Elliptic Curve Method (ECM). Since the smoothness testing is also suitable for parallelization, it is promising to improve ECM via special-purpose hardware. We show that massive parallel and cost efficient ECM hardware engines can improve the cost-time product of the RSA moduli factorization via the GNFS considerably.

The computation of ECM is a classical example for an algorithm that can be significantly accelerated through special-purpose hardware. In this work, we present an efficient hardware implementation of ECM to factor numbers up to 200 bits, which is also scalable to other bit lengths. For proof-of-concept purposes, ECM is realized as a software-hardware co-design on an FPGA and an embedded microcontroller. This appears to be the first publication of a realized hardware implementation of ECM, and the first description of GNFS acceleration through hardware-based ECM.

1 Introduction

Up to now, several efficient algorithms for factoring integers have been proposed. Each algorithm is appropriate for a different situation. For instance, the Generalized Number Field Sieve (GNFS, see [7]) is the best for factoring numbers with large factors (hundreds of bits) and, hence, can be used for attacking the RSA cryptosystem. As an intermediate step, the GNFS requires a method to efficiently factor lots of smaller numbers (factorization of the rests). An appropriate choice for this task is Multiple Polynomial Quadratic Sieve (MPQS) or Elliptic Curve Method (ECM, see [8]). The current world record in factoring a random RSA modulus is 576 bits and was achieved with a complete software implementation of the GNFS in 2003 [4], using MPQS for the factorization of the rests. For larger moduli it will become crucial to use special hardware for factoring. A new hardware design for the sieving step in GNFS is called SHARK [5] and has been proposed very recently.

SHARK distinguishes itself from other proposed architectures for RSA factorization (see, e.g., [12]) by relying on conventional technologies and a very high modularity. However, SHARK (and possibly other innovative GNFS realizations) will need highly efficient support units for smoothness testing.

It appears that the use of ECM rather than MQPS is the better choice for this task, since the MQPS requires a larger silicon area and irregular operations. On the other hand, ECM is an almost ideal algorithm for dramatically improving the time-cost product through special purpose hardware. First, it performs a very high number of operations on a very small set of input data, and is, thus, not very I/O intensive. Second, it requires relatively little memory. Third, the operands needed for supporting GNFS are well beyond the width of current computer buses, arithmetic units, and registers, so that special purpose hardware can provide a much better fit. Lastly, it should be noted that the nature of the smoothness testing in the GNFS allows for a very high degree of parallelization. Hence, the key for efficient ECM hardware lies in fast arithmetic units. Such units for modular addition and multiplication have been studied thoroughly in the last few years, e.g., for the use in cryptographic devices using Elliptic Curve Cryptography (ECC), see, e.g., [6, 10]. Therefore, we could exploit the well developed area of ECC architectures for our ECM design.

In this work, we present an efficient hardware implementation of ECM to factor numbers up to 200 bits, which is also scalable to other bit lengths. For proof-of-concept purposes, ECM is realized as a software-hardware co-design on an FPGA and an embedded microcontroller. Our design has a good scalability to larger and smaller bit lengths. In some range, both the time and the silicon area depend linearly on the bit length. One should notice that while direct application of the ECM to factor numbers with length of several hundreds bits[1] is uneffective and the method is asymptotically slow, the ECM is highly suitable for effective processing of smoothness testing step within GNFS that requires factorization up to 200-bit operands.

There are many possible improvements of the original ECM. Based on these improvements, we adapted the method to the very restrictive memory requirements of efficient hardware, thus, minimizing the AT product (area time product). The parameters used in our implementation are best suited to find factors of up to about 40 bits.

For the implementation, a highly efficient modular multiplication architecture described by Tenca and Koç (see [13]) is used. We describe a controlling unit that synchronously feeds multiple ECM units with programming steps, such that the ECM algorithm does not need to be stored in every single unit. In this way we keep the overall area small and still do not need much bandwidth for communication.

Section 2 introduces ECM and some optimizations relevant for our implementation. The choice of parameters and arithmetic algorithms, the design of the ECM unit and a possible parallelization are described in Section 3. Section 4 presents our FPGA implementation. The last section collects results and conclusions.

2 Elliptic Curve Method

The principles of ECM are based on Pollard's $(p-1)$-method [11]. We describe H. W. Lenstra's Elliptic Curve Method (ECM) [8].

2.1 The Algorithm

Let N be an integer without small prime factors which is divisible by at least two different primes, one of them q. Such numbers appear after trial division and a quick prime power test.

Let $E/\mathbb{Q}$ be an elliptic curve with good reduction at all prime divisors of N (this can be checked by calculating the gcd of N and the discriminant of E which very rarely yields a prime factor of N) and a point $P \in E(\mathbb{Q})$. Let $E(\mathbb{Q}) \to E(\mathbb{F}_q), Q \mapsto \overline{Q}$ be the reduction modulo q. If the order o of $\overline{P} \in E(\mathbb{F}_q)$ satisfies certain smoothness conditions described below, we can discover the factor q of N as follows:

- *Phase 1*: Calculate $Q = kP$, where $k = \prod_{p \le B_1 \text{ prime}} p^{e_p}$ and $e_p = \left[\frac{\log B_1}{\log p}\right]$.

- *Phase 2*: Check for each prime $B_1 < p \le B_2$ whether pQ reduces to the neutral element in $E(\mathbb{F}_q)$. This can be done, e.g. using the Weierstraß form and projective coordinates $pQ = (x_{pQ} : y_{pQ} : z_{pQ})$, by testing whether $\gcd(z_{pQ}, N)$ is bigger than 1.

All calculations are done modulo N. If an inversion is not possible (e.g., using affine coordinates

[1]To break RSA-based cryptosystems one has to factor the moduli with length of currently used minimum 1024 bits.

in the Weierstraß form) a divisor is found. The parameters B_1 and B_2 control the probability of finding a divisor q. More precisely, if o factors into a product of coprime prime powers (each $\leq B_1$) and at most one additional prime between B_1 and B_2, the prime factor q is discovered.

The procedure will be repeated for other elliptic curves and starting points. To generate them one commences with the starting point P and constructs an elliptic curve such that P lies on it. It is possible that more than one or even all prime divisors of N are discovered simultaneously. This happens rarely for reasonable parameter choices and can be ignored by proceeding to the next elliptic curve. Note that we can avoid all gcd computations but one at the expense of one modular multiplication per gcd by accumulating the numbers to be checked in a product modulo N and doing one final gcd.

2.2 The Elliptic Curves

Apart from the Weierstraß form there are various other forms for the elliptic curves. We use Montgomery's form (1) and compute in the set $S = E(\mathbb{Z}/N\mathbb{Z})/\{\pm 1\}$ only using the x- and z-coordinates.

$$By^2z = x^3 + Ax^2z + xz^2 \tag{1}$$

The residue class of $\overline{P+Q}$ in this set can be computed from $\overline{P}$, $\overline{Q}$ and $\overline{P-Q}$ using 6 multiplications (2). A duplication, i. e. $\overline{2P}$, can be computed from $\overline{P}$ using 5 multiplications (3). Since we are only interested in checking whether we obtain the point at infinity for some prime divisor of N computing in S is no restriction. In the following we will not distinguish between $E(\mathbb{Z}/N\mathbb{Z})$ and S, and pretend to do all computations in $E(\mathbb{Z}/N\mathbb{Z})$.

$$\begin{aligned}
&\textit{Addition}:\\
x_{P+Q} &= z_{P-Q}[(x_P - z_P)(x_Q + z_Q) + \\
&\quad (x_P + z_P)(x_Q - z_Q)]^2\\
z_{P+Q} &= x_{P-Q}[(x_P - z_P)(x_Q + z_Q) - \\
&\quad (x_P + z_P)(x_Q - z_Q)]^2 \qquad (2)\\
&\textit{Duplication}:\\
4x_Pz_P &= (x_P + z_P)^2 - (x_P - z_P)^2\\
x_{2P} &= (x_P + z_P)^2(x_P - z_P)^2\\
z_{2P} &= 4x_Pz_P[(x_P - z_P)^2 + \\
&\quad 4x_Pz_P(A+2)/4] \qquad (3)
\end{aligned}$$

2.3 The First Phase

If the triple $(P, nP, (n+1)P)$ is given we can compute $(P, 2nP, (2n+1)P)$ or $(P, (2n+1)P, (2n+2)P)$ by one addition and one duplication in Montgomery's form. Thus $Q = kP$ can be calculated using $[\log_2 k]$ additions and duplications, amounting to $11[\log_2 k]$ multiplications. In the case $z_P = 1$ we can reduce this to $10[\log_2 k]$ multiplications.

By handling each prime factor of k separately and using optimal addition chains the number of multiplications can be decreased to roughly $9.3[\log_2 k]$ (see [9]). The addition chains can be precalculated.

2.4 The Second Phase

The standard way to calculate the points pQ for all primes $B_1 < p \leq B_2$ is to precompute a (small) table of multiples kQ where k runs through the differences of consecutive primes in the interval $[B_1, B_2]$. Then, p_0Q is computed with p_0 being the smallest prime in that interval and the corresponding table entries are added successively to obtain pQ for the next prime p. Two major improvements have been proposed for ECM [2, 9]. Using Montgomery's form, the procedure is difficult to implement but can be improved as follows.

The improved standard continuation uses a parameter $2 < D < B_1$. First, a table T of multiples kQ of Q for all $1 \leq k < \frac{D}{2}$, $\gcd(k, D) = 1$ is calculated. Each prime $B_1 < p \leq B_2$ can be written as $mD \pm k$ with $kQ \in T$. Now, $\gcd(z_{pQ}, N) > 1$ if and only if $\gcd(x_{mDQ}z_{kQ} - x_{kQ}z_{mDQ}, N) > 1$. Thus, we calculate the sequence mDQ (which can easily be done in Montgomery's form) and accumulate the product of all $x_{mDQ}z_{kQ} - x_{kQ}z_{mDQ}$ for which $mD - k$ or $mD + k$ is prime.

The memory requirements for the improved standard continuation are $\frac{\varphi(D)}{2}$ points for the table T and the points DQ, $(m-1)DQ$, mDQ for computing the sequence, altogether $\varphi(D) + 6$ numbers. The computational costs consist of the generation of T and the calculation of mDQ which amounts to at most $\frac{D}{4} + \frac{B_2}{D} + 7$ elliptic curve operations (mostly additions) and at most $3(\pi(B_2) - \pi(B_1))$ modular multiplications, $\pi(x)$ being the number of primes up to x. The last term can be lowered if D contains many small prime factors since this will increase the number of pairs (m, k) for which both $mD - k$ and $mD + k$ are prime. Neglecting space considerations a good choice for D is a number around $\sqrt{B_2}$ which is divisible by many small primes.

3 Methodology

This section discusses the parametrization and design of our ECM unit.

3.1 Parametrization of ECM

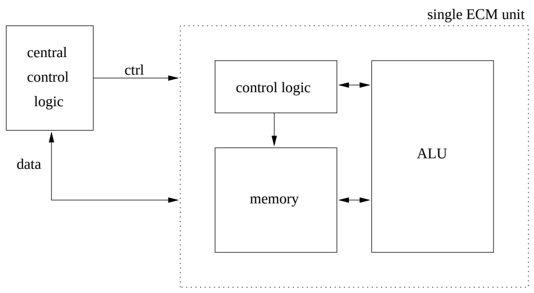
Our implementation focusses on the factorization of numbers up to 200 bits with factors of up to around 40 bits. Thus, “good” parameters for B_1, B_2, and D have to be found, yielding a high probability of success and a relatively small running time and area consumption. With the running time depending on the size of the (unknown) factors to be found, optimal parameters cannot be known beforehand. Hence, good parameters can be found by experiments with different prime bounds.

Deduced from software experiments, we choose $B_1 = 960$ and $B_2 = 57.000$ as prime bounds. The value of k has 1375 bits and, assuming the simple binary method, 1374 point additions and point duplications for the execution of phase 1 are required. Due to the use of Montgomery coordinates, the coordinate z_P of the starting point P can be set to 1, thus, addition takes only 5 multiplications instead of 6. The improved phase 1 (with optimal addition chains) has to use the general case, where $z_P \neq 1$. For the sake of simplicity and a preferably simple control logic, we choose the simple method for the time being. For the chosen parameters, the computational complexity of phase 1 is 13740 modular multiplications and squarings[2].

According to Equation (3), duplicating a point $2P_A = P_C$ involves the input values x_A, z_A, A_{24} and n, where $A_{24} = (A + 2)/4$ is computed from the curve parameter A (see Equation (1)) in advance and should be stored in a fixed register. A point addition $P_C = P_A + P_B$ handles the input values $x_A, z_A, x_B, z_B, x_{A-B}, z_{A-B}$ and n (see Equation 2). Notice that during phase 1 the values n, A_{24}, x_{A-B} and z_{A-B} do not change. Furthermore, $z_{A-B} = z_1$ can be chosen to be 1. Thus, no register is required for z_{A-B}. The output values x_C and z_C can be written to certain input registers to save memory. If we assume that the ECM unit does not execute addition and doubling in parallel, at most 7 registers for the values in $\mathbb{Z}/n\mathbb{Z}$ are required for phase 1. Additionally, we will require 4 temporary registers for intermediate values. Thus, a total of 11 registers is required for phase 1.

[2] In this contribution, squarings and multiplications are considered to have an identical complexity since the hardware will compute a squaring with the multiplication circuit.

Figure 1. Overview of one ECM Unit



For phase 2, the value $D = 30$ is chosen to keep the size of the table low at the cost of an increased running time. Since $\varphi(D) = 8$ is small, only 8 additional registers are required to store all coordinates in the table. Unlike in phase 1, we have to consider the general case where $z_{A-B} \neq 1$. Hence, an additional register for this quantity is needed. For the product Π of all $x_A \cdot z_B - z_A \cdot x_B$, one more register is necessary. The temporary registers from phase 1 suffice to store the intermediate results $x_A \cdot z_B$, $z_A \cdot x_B$ and $x_A \cdot z_B - z_A \cdot x_B$. Hence, a total of 21 registers is necessary for both phases. The total computational complexity of phase 2 is 1881 point additions 10 point doublings. Together with the 13590 modular multiplications for computing the product Π, 24926 modular multiplications and squarings are required.

For a high probability of success ($> 90\%$), software experiments showed for the parameters given that we need to run ECM on approximately 20 different curves.

3.2 Design of the ECM Unit

The ECM unit mainly consists of three parts: the ALU (arithmetic logic unit), the memory part (registers) and an internal control logic (see Figure 1). Each unit has a very low communication overhead with central control logic since all input operands and results are stored *inside* the unit during computation. Before the computation starts, all required initial values are assigned to memory cells of the unit. Only at the very end when the product Π has been computed, the result is read from the unit’s memory to the processor for subsequent computations. Commands are generated and timed by a central control logic outside the ECM unit(s).

3.2.1 Central Control Logic

The central control logic is connected to each ECM unit via a control bus (*ctrl*). It coordinates the data exchange with the unit before and after computation and starts each computation in the unit by a special set of commands stored in a control register. The commands contain an instruction for the next computation to be performed (i.e., add, subtract, multiply, double), including the in- and output registers to be used (R1-R21). The start of an operation is invoked by setting the *start*-bit to '1'. The control bus has to offer the possibility to specify which input register(s) and which output register is active. Only certain combinations of in- and output registers occur, offering the possibility to reduce the complexity of the logic and the width of the control bus by compressing the necessary information. For simplicity and clarity, we skipped the further optimization of the commands. Instead, we use a clearly understandable structure for the commands. A command consists of 16 bits which are assigned as follows (LSB is left):

start	operation	input 1	input 2	output
X	XX	XXXX	XXXX	XXXXX

If several ECM units work in parallel, only one central control register is needed. All commands are sent in parallel to all units and computations start in the same time. Only in the beginning and in the end, the unit's memory cells have to be written and read out separately. Once the computations in all units are done, an LSB of the central status register is set to '0' to indicate the units' availability for further commands.

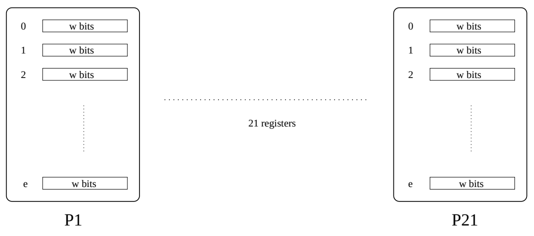
3.2.2 Internal Control Logic

Each unit possesses an internal control logic in order to coordinate the data in- and output from and to the registers, respectively, and to control the computation process of ALU. Once a command with the corresponding start bit is set, the chosen operation (addition/subtraction or multiplication/squaring) inside the unit is started. The arithmetic is coordinated inside the ALU and communicated with the internal control logic.

3.2.3 Memory

The addresses specified above refer to relative addresses inside each unit since we want to address the same register in multiple units in parallel. For reading from/ writing to a single register in a specific unit, the unit needs to be addressed separately.

Figure 2. Memory Management of the ECM Unit



In combination with a unique address for each unit, a register has an unique hardware address and can be addressed from outside the unit by the processor. This is imperative since the central control logic writes data from the processor to these registers before phase 1 starts and it reads data from one of the registers after phase 2 has been finished. Each register can contain n bits and is organized in $e = \lceil \frac{n+1}{w} \rceil$ words of size w (see Figure 2). Memory access is performed word wise. Reasonable values for w are $w = 4, 8, 16, 32$ but are, though, not mandatory. Optimal word width w depends on memory block units configuration inside chosen family of FPGAs.

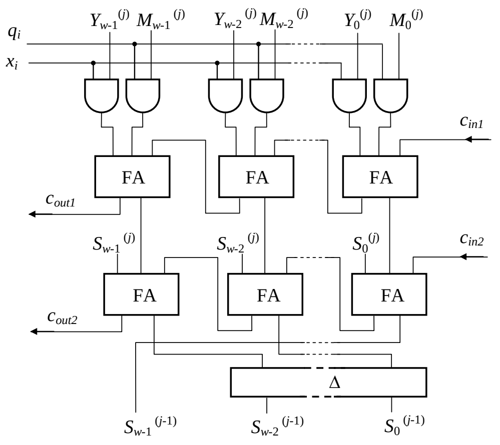
3.2.4 Arithmetic Logic Unit (ALU)

The ALU performs the basic arithmetic operations in mod $2M$ (for the following part of paper we use M for modulus), i.e., modular multiplication, modular squaring, modular addition and modular subtraction. Objectives for the choice of implemented algorithms are mentioned in the following subsections.

3.3 Choice of the Arithmetic Algorithms

The main purpose of the design is to synthesize an area-time efficient implementation of ECM. Hence, all algorithms are chosen to allow for low area and relatively high speed. Low area consumption can be achieved by structures, which allow for a certain degree of serialization and, hence, do not require much memory. For ECM, we have chosen a set of algorithms which seem to be very well suited for our purpose. In the following, we briefly describe the algorithms for modular addition, subtraction, and multiplication to be implemented for the ALU. Squaring is done with the multiplication circuit since a separate hardware circuit for squaring would increase the overall AT

Figure 3. Multiplier Stage with Carry-Propagate Adders and Non-redundant Representation



product. Similarly, subtraction can be computed with a slightly modified adder circuit. Since we do not require parallel running of modular operations (what would bring a demand for larger memory block), sharing the same part of ALU for two similar modular operations offers significant decrease of area.

3.3.1 Modular Multiplication

An efficient Montgomery multiplier, highly suitable for our design is described in [13]. The presented multiplier architecture allows a word wise multiplication and is scalable regarding operand sizes, word size, and pipelining depth. The internal word additions are performed by simple adders. Figure 3 shows the architectures of one stage. While in [13] a structure with carry-save adders and redundant representation of operands has been implemented, we have chosen a configuration with carry-propagate adders and non-redundant representation that makes possible more effective implementation especially when target platform supports fast carry chain logic.

Recent FPGAs contain high speed interconnect lines between adjacent logic blocks which are designed to provide an efficient carry propagation. The multiplier architecture presented in this paper is optimal for implementation on any FPGA that has dedicated carry logic capability (e.g. modern Altera and Xilinx FPGAs). A detailed analysis and comparison of both structures can be found in [3].

The depicted hardware performs a slightly modified Multiple Word Radix-2 Montgomery Multiplication (Algorithm 1). Instead of more expensive word-wise addition in step 3 we have used only bit operations. In Algorithm 1, word and bit vectors are represented as:

$$\begin{aligned} M &= (0, M^{(e-1)}, \ldots, M^{(1)}, M^{(0)}), \\ Y &= (0, Y^{(e-1)}, \ldots, Y^{(1)}, Y^{(0)}), \\ S &= (0, S^{(e-1)}, \ldots, S^{(1)}, S^{(0)}), \\ X &= (x_{n-1}, \ldots, x_1, x_0), \end{aligned}$$

where words are marked with superscripts and bits are marked with subscripts. The final reduction step of originally proposed Montgomery multiplication can be omitted when the following condition is fulfilled:

$$4M < 2^n \quad (4)$$

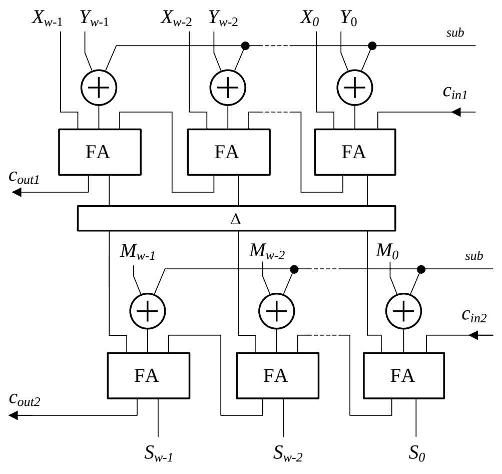
Algorithm 1 Multiple Word Radix-2 Montgomery Multiplication [13]

```
1:  $S = 0$
2:  for $i = 0$ to $n - 1$ do
3:    $q_i := x_i Y_0^{(0)} + S_0^{(0)}$
4:    if $q_i = 1$ then
5:      for $j = 0$ to $e$ do
6:        $(C_a, S^{(j)}) := C_a + x_i Y^{(j)} + M^{(j)}$
7:        $(C_b, S^{(j)}) := C_b + S^{(j)}$
8:        $S^{(j-1)} := (S_0^{(j)}, S_{w-1\ldots1}^{(j-1)})$
9:      end for
10:   else
11:     for $j = 0$ to $e$ do
12:       $(C_a, S^{(j)}) := C_a + x_i Y^{(j)}$
13:       $(C_b, S^{(j)}) := C_b + S^{(j)}$
14:       $S^{(j-1)} := (S_0^{(j)}, S_{w-1\ldots1}^{(j-1)})$
15:     end for
16:   end if
17:   $S^{(e)} = 0$
18: end for
```

In this case if the input values are bounded by $X, Y < 2M$, then also the output value is bounded by $S < 2M$ and can be used as an input value for the following computation (for more details see [14]).

A minimal AT product of the multiplier can be achieved with a word width of 8 bits and a pipelining depth of 1 ($w = 8, p = 1$, see [13]). However, for our ECM architecture, the AT product does not only depend on the AT product of the multiplier. In fact, the multiplier only takes a comparably small part of the overall area. On the other hand, the overall speed relies primarily on the speed of the multiplier. Thus, for implementation we choose a pipelining depth of $p = 2$ for

Figure 4. Universal Addition and Subtraction Unit for Operands with Word Width w



word width $w = 32$ bits, in order to achieve a higher speed and better connection with embedded microcontroller that has 32-bit wide data interface. Thus, data can be read and written directly by 32-bit words.

3.3.2 Modular Addition and Subtraction

Addition and subtraction unit is implemented as one circuit. As with the multiplication circuit, the operations are done word wise and the word size and number of words can be chosen arbitrary. Since the same memory is used for input and output operands, we choose the same word size as for the multiplier. The subtraction relies on the same hardware, only one input bit has to be changed ($sub = 1$) in order to compute a subtraction rather than an addition (see Figure 4). All operations are done in mod $2M$. The actual computation is done in a word wise manner using a word size of $w = 32$. Algorithms 2 and 3 show the elementary steps of a modular addition and subtraction, respectively.

If $X+Y > 2M$ a reduction can be applied by simple subtraction of the double modulus $2M$. Hence, following algorithm is used for the modular addition: S contains the result and T is a (temporary) register inside the ALU. Both values are written directly to the specified output register in the ECM unit's memory. A comparison $T < 2M$ takes the same amount of time than a subtraction $S = T - 2M$. Thus, we compute the subtraction in all cases and decide by the sign of the values, which one to take as the result (S or T). If S is

Algorithm 2 Modular addition

```
Require: Two integers $X, Y < 2M$
Ensure: Sum $S = X + Y \bmod 2M$
1: for $j = 0$ to $e$ do
2:     $(C_1, T^{(j)}) := C_1 + X^{(j)} + Y^{(j)}$
3:     $(C_2, S^{(j)}) := C_2 + T^{(j)} - 2M^{(j)}$
4: end for
5: if $S < 0$ then
6:     copy $T$ to $S$
7: else
8:     done
9: end if
```

negative, the content of T has to be copied to the memory in $(e+1)$ clock cycles (since in the meanwhile the content of the output memory was rewritten by the value S, see step 3 in Algortihm 2). If S has positive value, then the result stored in memory is correct and computation is done.

For a modular addition, we need at most

$$T_{add} = 3(e+1) \tag{5}$$

clock cycles, where e is the number of words (for implemented non-redundant form of operands $e = \lceil \frac{n+1}{w} \rceil$). On average, we would only have to copy temporary register T every other time, in case $X + Y < 2M$ and no reduction is needed. However, since the control of ECM phase 1 and phase 2 is parallelized for many units, we have to assume the worst case running time which is given by Equation (5).

The operation of subtraction in addition unit can be accomplished by the addition of two's complement, that is computed by inversion of the operand (by exclusive-OR addition, $X \oplus 1$) and addition of 1. The addition of 1 is simply achieved by setting the first carry bit to one ($c_{in} = 1$) for $j = 0$. For the other words of operands ($j \neq 0$), the carry bit c_{in} has the value of the previous c_{out}.

For modular subtraction one can use a bit simpler algorithm in comparison to addition. First the subtraction of input operands is computed. In case the result is negative, the double modulus has to be added as it is described in the Algorithm 3. In step 2, both memory cells S and T obtain the same value, which can be done in hardware in parallel at the same time without any additional overhead. After the computation of the difference, one can check for the correctness of the result. Hence, subtraction can be performed more efficiently than addition and requires in the worst case

$$T_{sub} = 2(e+1) \tag{6}$$

Algorithm 3 Modular subtraction

```
Require: Two integers $X, Y < 2M$
Ensure: Difference $S = X - Y \bmod 2M$
 1: for $j = 0$ to $e$ do
 2:     $(C_1, T^{(j)}) := C_1 + X^{(j)} - Y^{(j)}$
 3: end for
 4: if $T < 0$ then
 5:     for $j = 0$ to $e$ do
 6:         $(C_2, S^{(j)}) := C_2 + T^{(j)} + 2M^{(j)}$
 7:     end for
 8: else
 9:     done
10: end if
```

clock cycles.

To achieve computation in mod $2M$ one has to add/subtract double value of the value M stored in the ECM unit's memory. Since operands are read word wise, word M needs to be shifted and MSB of computed word is stored and concatenated as LSB for the subsequent word of M. For the sake of simplicity this operation is not depicted in Figure 4.

3.4 Parallel ECM

ECM can be perfectly parallelized by using different curves in parallel since the computations of each unit are completely independent. For the control of more than one ECM unit, it is essential to know that the both phases, phase 1 and phase 2, are controlled completely identical, independent of the composite to be factored. Solely the curve parameter and eventually the modulus of the units and, hence, the coordinates of the initial point differ. Thus, all units have to be initialized differently which is done by simply writing the values into the corresponding memory locations sequentially[3].

During the execution of both phases, exactly the same commands can be sent to all units in parallel by central control logic. Since running time of multiplication/squaring is constant (does not rely on input values) and for addition/subtraction differs at most in $2(e + 1)$ clock cycles, all units can execute the same command at exactly the same time. After phase 2, the results are read from the units one after another.

The required time for this data IO is neglectable for one ECM unit since the computation time of both phases dominates. For several units in parallel, the computation time does not change, but the time for data IO scales linearly with the number of units. Hence, not too many units should be controlled by one single logic. For massively parallel ECM in hardware, the ECM units can be segmented into clusters, each with an own control unit.

[3]At a later stage, we might think of some modifications to improve the data IO of the registers.

4 Implementation

In this section we provide results in timing and area consumption together with estimation of running times for all operations done within the ECM.

4.1 Hardware Platform

The ECM implementation at hand is a hybrid design. It consists of an ECM unit implemented on an FPGA (Xilinx Virtex2000E-6) and a control logic implemented in software on an embedded microcontroller (ARM7TDMI, 25MHz, see [1]). The ECM unit is coded in VHDL and was synthesized for a Xilinx FPGA (Virtex2000E-6, see [15]). For the actual VHDL implementation, memory cells have been realized with the FPGA's internal block RAM memory. For the word width $w = 32$ bits 2 blocks with $e = \lceil \frac{n+1}{w} \rceil$ words are used for each register.

The unit, as implemented, listens for commands which are written to control register accessible by the FPGA. Required point coordinates and curve parameters are loaded into the unit before the first command is decoded. For this purpose, these memory cells are accessible from the outside by a unique address. Internal registers, which are only used as temporary registers during the computation are not accessible from the outside.

The control of the unit(s) is done by a microcontroller present on the board and controls the data transfer from an to the units an issues the commands for all steps in phase 1 and phase 2.

Remark, that the design is done for $n = 198$ bits composites. The whole design is fully scalable and bitlengths from 100 to 300 bits can be easily accomplished. In this case, the AT product will de-/ increase according to the size of n^2. For suitable implementation in selected platform one can choose the word width w, number of words e (length of operands), level of pipeline inside the multiplier p and number of ECM units. Although presented implementation was realized on Xilinx Virtex-E FPGA, proposed algorithms and design architecture can be implemented on any FPGA that includes fast carry chain logic and on-chip dual-port memory blocks.

Table 1. Running Times of the ECM Implementation (195 bits modulus, $w = 32$, Xilinx Virtex2000E-6, 25MHz)

Operation	Time
modular addition	2.00 μs
modular subtraction	1.68 μs
modular multiplication	64.5 μs
modular squaring	64.5 μs
point addition (phase 1, $z_Q = 1$)	333 μs
point addition (phase 2)	397 μs
point doubling	330 μs
Phase 1	912 ms
Phase 2	1879 ms

4.2 Results

After the synthesis and place and route, the binary image was loaded onto the FPGA and clocked with a frequency of 25MHz that is common for both, the ARM processor and the FPGA with the ECM design. Hence, the cycle length of the ALU performing the modular arithmetic is 40ns. Table 1 shows the timings of the implementation. Hardware factorization design includes full support for all operations needed during the ECM phases 1 and 2. The timings for phase 1 and 2 are obtained after timing measurements on a testing board. The time for the initialization and reading from the memories is not taken into account, since it only delays the computation at the very beginning and the very end.
The ECM unit including the full support for the phase 1 and 2 of the ECM with the word width $w = 32$ bits, number of words $e = 7$, and pipeline level $p = 2$ has the following area requirements: 1754 lookup-tables (LUT), 506 flip-flops and 44 Blocks RAM. Minimum clock period is 26.225ns (maximum clock frequency: 38.132MHz). Further improvements in data configuration and better management of input and output operands for the ALU inside the ECM unit should yield higher performance of the whole design.

4.3 Estimation of the Running Time

We can determine the running time of both phases on basis of the underlying ring arithmetic. The upper bounds for the number of clock cycles of a modular addition and a modular subtraction for a setting with $n = 198$, $w = 32$, $p = 2$ and $e = 7$ are determined by $T_{add} = 3(e+1) = 24$ and $T_{sub} = 2(e+1) = 16$ cycles. According to [13], the implemented multiplier requires

$$T_{mul} = \left\lceil \frac{n}{p} \right\rceil (e+1) + 2p \qquad (7)$$

clock cycles per multiplication. Hence, a multiplication requires $T_{mul} = 796$ cycles. For each operation we should include $T_{init} = 2$ clock cycles for initialization of ALU at the beginning of each computation.
For the group operations for phase 1 we obtain

$$\begin{aligned} T_{Padd} &= 5T_{mul} + 3T_{add} + 3T_{sub} + 11T_{init} \\ &= 4122 \qquad (8) \end{aligned}$$

and

$$\begin{aligned} T_{Pdbl} &= 5T_{mul} + 2T_{add} + 2T_{sub} + 9T_{init} \\ &= 4078 \qquad (9) \end{aligned}$$

clock cycles.
For phase 2, T_{Padd} changes to $T'_{Padd} = 4920$ since $z_{A-B} \neq 1$ in most cases, hence, we have to take the multiplication with z_{A-B} into account.
The total cycle count for both phases is

$$\begin{aligned} T_{Phase\ 1} &= 1374(T_{Padd} + T_{Pdbl}) \\ &= 11071692 \qquad (10) \end{aligned}$$

and

$$\begin{aligned} T_{Phase\ 2} &= 1881T'_{Padd} + 50T_{Pdbl} + \\ &\quad 13590T_{mul} \\ &= 20276060 \qquad (11) \end{aligned}$$

clock cycles. Excluding the time for pre- and post-processing, a unit needs approximately $31.3 \cdot 10^6$ clock cycles for both phases on one curve.
From the comparison to the timings obtained from the implemented hardware factorization design (in Section 4.2) we can conclude that the control of the computations and communication between ARM processor and FPGA is not optimal yet and represents significant part of the total computation time. Therefore this part of design needs further improvements.

5 Conclusions

The work at hand presents the first hardware implementation of factorization based on ECM. The implementation is a hardware-software codesign and has been implemented on an FPGA and an ARM microcontroller for factoring integers of a size of up to 200 bits. We implemented a variant of ECM which allows for a very low area time product and, hence, is very cost effective. Our implementation impressively shows that due to very low area requirements and low data IO, ECM is predestinated for the use in hardware. A single unit for factoring composites of up to 195 bits requires 1754 lookup-tables, 506 flip-flops and 44 Blocks RAM (less than 6% of logic and 27% of memory resources of the Xilinx Vertex2000E device).

As demonstrated, ECM can be perfectly parallelized and, thus, an implementation at a larger scale can be used to assist the GNFS factoring algorithm by carrying out all required smoothness tests. The concept of the GNFS architecture SHARK counts with around 3.7 millions of the ECM units, therefore a low cost ASIC implementation of ECM can decrease the overall costs of SHARK, as shown in [5].

As future steps, variants of phase 2 can be examined in order to achieve the lowest possible AT product. To achieve higher maximal clock frequency of the ECM unit, we need to focus on better structure of the control logic inside the unit. Furthermore, most of the computation time is spent for modular multiplications so its improvement will directly affect the overall running time. With the VHDL source code at hand, the next logic step is the design and simulation of a full custom ASIC containing the logic, which currently is implemented in the FPGA. For an ASIC implementation, a parallel design of many ECM units is preferable. The control can still be handled outside the ASIC by a small microcontroller, as it is the case with the work at hand. Alternatively, a soft core of a small microcontroller can be adopted to the specific needs of ECM and be implemented within the ASIC. With a running ECM ASIC, exact cost estimates for the support of algorithms such as the GNFS can be obtained.

References

[1] ARM Limited. ARM7TDMI (Rev 3) — Technical Reference Manual. `http://www.arm.com/pdfs/DDI0029G_7TDMI_R3_trm.pdf`, 2001.

[2] R. P. Brent. Some Integer Factorization Algorithms Using Elliptic Curves. In *Australian Computer Science Communications 8*, pages 149–163, 1986.

[3] M. Drutarovský, V. Fischer, and M. Šimka. Comparison of two implementations of scalable Montgomery coprocessor embedded in reconfigurable hardware. In *Proceedings of the XIX Conference on Design of Circuits and Integrated Systems — DCIS 2004*, pages 240–245, Bordeaux, France, Nov. 24–26, 2004.

[4] J. Franke, T. Kleinjung, F. Bahr, M. Lochter, and M. Bohm. E-mail announcement. `http://www.crypto-world.com/announcements/rsa576.txt`, Dec. 2003.

[5] J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, and C. Stahlke. SHARK — A Realizable Special Hardware Sieving Device for Factoring 1024-bit Integers. In *1st Workshop on Special-purpose Hardware for Attacking Cryptographic Systems — SHARCS 2005*, Paris, France, Feb. 24–25, 2005.

[6] N. Gura, S. Chang, H. 2, G. Sumit, V. Gupta, D. Finchelstein, E. Goupy, and D. Stebila. An End-to-End Systems Approach to Elliptic Curve Cryptography. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002*, volume LNCS 2523, pages 349–365. Springer-Verlag, 2002.

[7] A. K. Lenstra and H. W. Lenstra. *The Development of the Number Field Sieve*. Lecture Notes in Math. Volume 1554. 1993.

[8] H. W. Lenstra. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126(2):649–673, 1987.

[9] P. Montgomery. Speeding up the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[10] G. Orlando and C. Paar. A Scalable $GF(p)$ Elliptic Curve Processor Architecture for Programmable Hardware. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001*, volume LNCS 2162, pages 348–363. Springer-Verlag, May 14–16, 2001.

[11] J. Pollard. A Monte Carlo Method for Factorization. *Nordisk Tidskrift for Informationsbehandlung (BIT)*, 15:331–334, 1975.

[12] A. Shamir and E. Tromer. Factoring Large Numbers with the TWIRL Device. In *Advances in Cryptology — Crypto 2003*, volume 2729 of *LNCS*, pages 1–26. Springer, 2003.

[13] A. Tenca and Ç.K. Koç. A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm. *IEEE Trans. Comput.*, 52(9):1215–1221, 2003.

[14] C. D. Walter. Precise Bounds for Montgomery Modular Multiplication and Some Potentially Insecure RSA Moduli. In B. Preneel, editor, *Proceedings of Topics in Cryptology- CT-RSA 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 30–39, 2002.

[15] Xilinx. Virtex-E 1.8V Field Programmable Gate Arrays — Production Product Specification. `http://www.xilinx.com/bvdocs/publications/ds022.pdf`, June 2004.