Realizing Arbitrary-Precision Modular Multiplication with a Fixed-Precision Multiplier Datapath*

Johann Großschädl
University of Luxembourg
johann.groszschaedl@uni.lu

Erkay Savaş
Sabanci University, Turkey
erkays@sabanciuniv.edu

Kazim Yumbul
Gebze Institute of Technology
kyumbul@gyte.edu.tr

Abstract

Within the context of cryptographic hardware, the term scalability refers to the ability to process operands of any size, regardless of the precision of the underlying datapath or registers. In this paper we present a simple yet effective technique for increasing the scalability of a fixed-precision Montgomery multiplier. Our idea is to extend the datapath of a Montgomery multiplier in such a way that it can also perform an ordinary multiplication of two n-bit operands (without modular reduction), yielding a 2n-bit result. This conventional ($n \times n \rightarrow 2n$)-bit multiplication is then used as a "sub-routine" to realize arbitrary-precision Montgomery multiplication according to standard software algorithms such as Coarsely Integrated Operand Scanning (CIOS). We show that performing a 2n-bit modular multiplication on an n-bit multiplier can be done in 5n clock cycles, whereby we assume that the n-bit modular multiplication takes n cycles. Extending a Montgomery multiplier for this extra functionality requires just some minor modifications of the datapath and entails a slight increase in silicon area.

1. Introduction

Tenca and Koç [18] define an arithmetic unit as scalable *if the unit can be reused or replicated in order to generate long-precision results independently of the datapath precision for which the unit was originally designed*. For instance, a modular multiplier originally dimensioned for a precision of 1024 bits should also allow one to perform modular arithmetic on 2048-bit operands without the need to re-design the multiplier. Scalability is a highly desirable feature of cryptographic hardware as different applications generally require different levels of security.

The simplest way to implement a multiplier in hardware is by means of a serial/parallel architecture in which one of the operands is processed serially (e.g. bit by bit), while the other operand is processed fully parallel. In each step, a partial product is added to a running sum and then the sum is shifted by one bit in order to align it for the next partial product. Modular reduction can be easily integrated into the multiplication steps by subtracting (or adding) a multiple of the modulus so that either the most or least significant bit(s) of the sum becomes zero. A *bit-serial Montgomery multiplier* determines the multiple of the modulus M to be subtracted by inspecting the least significant bit of the running sum [5]. Algorithm 1 describes the basic principle of Montgomery multiplication for a number representation radix of 2 [8]. High-radix Montgomery multiplication can be realized in a similar way [2].

Algorithm 1. Montgomery multiplication (radix-2 version)

Input: n-bit modulus $M = (m_{n-1}, \ldots, m_1, m_0)_2$ with $m_0 = 1$ (i.e. M is an odd number), two operands $A = (a_{n-1}, \ldots, a_1, a_0)_2$ and $B = (b_{n-1}, \ldots, b_1, b_0)_2$ with $0 \leq A, B < M$.
Output: Montgomery product $Z = A \cdot B \cdot 2^{-n} \bmod M$.

```
Z ← 0
for i from 0 to n − 1 do
    Z ← Z + A · b_i      { addition of partial product A · b_i }
    Z ← Z + M · z_0      { addition of M · z_0 (z_0 is the LSB of Z) }
    Z ← Z/2              { 1-bit right-shift of Z (z_0 = 0 before the shift) }
end for
if Z ≥ M then Z ← Z − M end if
return Z
```

Bit-serial Montgomery multipliers are simple to design and implement, but suffer from serious limitations in terms of scalability. A modular multiplier with a fixed-precision datapath (e.g. a bit-serial/parallel Montgomery multiplier) does not scale beyond a certain limit (e.g. 1024 bits), which means that the modular multiplier has to be re-designed when the need for dealing with operands of higher precision arises. Poor scalability is even more problematic from the security perspective. Fundamental advances in cryptanalysis could, hypothetically speaking, make 1024-bit RSA keys substantially less secure than expected today. In such case

*The research described in this paper was supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under project number 105E089 (TUBITAK Career Award).

the key-size needs to be extended (e.g. from 1024 to 2048 bits), which is only possible when the crypto hardware is flexible enough to process the longer operands.

1.1. Related Work

Previous attempts to improve the scalability of modular multipliers can be broadly divided into two categories. The first approach utilizes advanced algorithmic techniques to virtually "stretch" the bit-length of a modular multiplier [15, 6, 3], while the second approach realizes scalability in hardware on basis of special multiplier architectures able to process operands of arbitrary size [18, 1, 19]. Paillier's [15] technique to overcome the scalability constraints of a conventional n-bit modular multiplier falls into the former category. Taking advantage of a Residue Number System (RNS), Pailler presented an algorithm that accomplishes a $2n$-bit modular multiplication by performing exactly nine n-bit modular multiplications (or only six if one of the two operands and the modulus are given in advance). Another method for performing a $2n$-bit modular multiplication via n-bit arithmetic was introduced by Fischer and Seifert in [6]. Their method requires a minimal modification of the hardware so that the modular multiplier actually performs an Euclidean multiplication, i.e. an arithmetic operation returning not only the remainder $A \cdot B \bmod N$, but also the integer quotient $A \cdot B \operatorname{div} N$. Fischer and Seifert describe two algorithms for performing a $2n$-bit modular multiplication, whereby the more efficient one requires six Euclidean multiplications of n-bit precision each. This result was then further improved using a specific (i.e. modulus-dependent) representation of the operands [3].

Scalable cryptographic systems can be implemented in a completely different way by taking advantage of dedicated hardware architectures into which scalability is built in by design and not just added as an afterthought. The scalable modular multiplier architecture by Tenca and Koç serves as a good example for this approach [18]. Their design is based on a special word-level algorithm for Montgomery multiplication and utilizes an array of word-size processing elements (PEs) organized in a pipeline. The architecture is highly scalable because a fixed-area multiplier can handle operands of any size. Moreover, the word-size of the PE as well as total the number of pipeline stages can be selected according to the desired trade-off between area and performance. However, such a high degree of scalability does not come for free, but implies increased complexity in the design and an area overhead due to extra control logic and registers, especially when compared to a standard bit-serial Montgomery multiplier. Scalable Montgomery multipliers based on the concept of systolic arrays can be realized in a similar way, see e.g. [20, 1]. Radix-4 designs of scalable Montgomery multipliers were presented in [19] and [17].

1.2. Our Approach

We present the architecture of a simple modular multiplier that comprises a fixed-precision datapath, but nevertheless allows one to perform modular multiplications on operands exceeding the size of the datapath. Our approach aims to find a trade-off between the simplicity of a bit-serial architecture and the perfect scalability of Tenca and Koç's multiplier design. The architecture we devised is basically a bit-serial Montgomery multiplier with some additional functionality to perform a multiplication without reduction operation. Our multiplier consists of two n-bit carry-save adders (CSAs) and a set of n-bit registers to accommodate the operands. It is able to execute an $(n \times n)$-bit modular multiplication in n clock cycles, whereas the conventional $(n \times n)$-bit multiplication (yielding a $2n$-bit product) takes $n/2$ cycles. A modular multiplication on n-bit operands is carried out as shown in Algorithm 1: the first CSA is used to add a partial product $A \cdot b_i$, while the second CSA serves to reduce the intermediate sum Z by addition of a multiple of the modulus. However, when the length of the operands exceeds n bits, the modular multiplication is performed via a sequence of conventional $(n \times n)$-bit multiplications according to software algorithms such as Coarsely Integrated Operand Scanning [7, 11]. When executing a conventional multiplication (without reduction), both CSAs are used to add partial products; therefore the $2n$-bit result is available after $n/2$ cycles. A complete $2n$-bit modular multiplication requires ten conventional $(n \times n)$-bit multiplications, which take some $5n$ clock cycles on our n-bit multiplier.

2. Long Integer Modular Arithmetic

Modular exponentiation is the core operation of various public-key cryptosystems, including RSA, DSA, and Diffie-Hellman. There exist a number of different algorithms for computing a modular exponentiation, but in the end they all require to carry out a sequence of modular multiplications and modular squarings. The main challenge when implementing public-key cryptosystems is the mismatch between the length of the operands and the wordsize of today's processors. Typical operand lengths for public-key systems based on the integer factorization problem (e.g. RSA) or the discrete logarithm problem (e.g. DSA or Diffie-Hellman) range from 1024 to 4096 bits, which means that they exceed the wordsize of current-generation processors by one or two orders of magnitude. Another crucial implementation issue is the algorithm for performing a modular reduction. Most practical implementations use either Barrett's algorithm or the well-known Montgomery reduction method [12] since they allow one to avoid costly divisions in the reduction operation. Both techniques are suitable for implementation in hardware and software.

In the following, we briefly summarize the basic concepts of Montgomery's reduction method. Let M be an odd modulus of length n bits, i.e. $2^{n-1} \leq M \leq 2^n - 1$, and let A and B denote two unsigned integers within the range of $[0, M-1]$. Furthermore, let us assume that the product $P = A \cdot B$ is $2n$ bits long. Montgomery's technique is based on the simple fact that we can freely add multiples of M to the product P without changing the value of P modulo M. A straightforward implementation of Montgomery reduction performs shift and add operations as follows: One inspects the least significant bit p_0 of the product P and adds the modulus M to P if $p_0 = 1$. This addition converts P into an even number if it was odd before[1], but does not change the value of P mod M. Now we can shift P one bit to the right without destroying any information. Repeating these steps n times leads to a result that is at most $n+1$ bits long and congruent to P mod M (see [12] for more details). A final subtraction of M brings the result into the desired range of $[0, M-1]$. However, Montgomery's algorithm computes $P \cdot 2^{-n}$ mod M instead of the actual residue P mod M; the factor 2^{-n} is due to the n right shifts carried out during a Montgomery reduction.

In most practical implementations of Montgomery's algorithm, the modular reduction is interleaved with a multiplication or squaring, instead of performing it as a separate operation thereafter. An interleaving of multiplication and reduction steps, as shown in Algorithm 1, is more economic in the use of memory resources since doing the reduction after completion of the multiplication would necessitate to temporarily store the $2n$-bit product $P = A \cdot B$. References [5] and [7] describe several techniques for computing the *Montgomery product* $A \cdot B \cdot 2^{-n}$ mod M by combining the multiplication of A by B and the reduction modulo M into a single operation:

$$\text{MonPro}(A, B) = A \cdot B \cdot 2^{-n} \bmod M \qquad (1)$$

The Montgomery product carries the factor 2^{-n}, which is referred to as *Montgomery radix* [8]. Some simple pre- and post-calculations are necessary to deal with this factor. The pre-calculation transfers the operands A, B into a special representation, the so-called *M-residue* [8] or *Montgomery image* representation (see [12] and [7] for a more detailed treatment of this operand conversion).

2.1. Montgomery Multiplication in Hardware

The main issue one has to deal with when designing a modular multiplier for public-key cryptography is the size of the operands (typically ≥ 1024 bits for RSA). Hardware implementations can efficiently cope with operands of such size by providing registers and/or datapaths of appropriate precision. When classifying hardware architectures for long integer Montgomery multiplication, two basic approaches can be identified. The first approach is based on a bit- or digit-serial multiplier datapath where one of the operands is scheduled bit by bit (or digit by digit), while the other operand is scheduled fully parallel. A second approach for performing Montgomery multiplication is to employ one or more word-level multipliers and accomplish the long integer arithmetic by means of conventional $(w \times w)$-bit multiplications according to software-oriented algorithms such as Coarsely Integrated Operand Scanning [7]. The selection of the wordsize w enables different trade-offs between area and performance; typical values for w are 16 and 32 [16].

Bit-serial or digit-serial architecture: A conventional bit-serial Montgomery modular multiplier can be implemented as illustrated in Algorithm 1. Given a modulus length of n bits, a bit-serial multiplier requires exactly n clock cycles to accomplish a Montgomery multiplication (excluding a possible final subtraction). Both the multiplier datapath and the registers for storing the operands and the result have a precision of n bits. An n-bit multiplier can also be used for operands of smaller size (e.g. $k < n$ bits) by right-aligning all operands in the registers and returning the result after k cycles. On the other hand, it is generally not possible to use this n-bit multiplier for larger operands, which means that the bit-serial architecture is only scalable up to a certain limit, namely the precision it has been designed for.

Word-level architecture: A word-level architecture consists of one or several "small" integer multipliers that are able to accomplish a $(w \times w)$-bit multiplication, whereby w refers to the word-size of the architecture. In general, the word-size is much smaller than the operand length; typical values for w are 16, 32, or 64 bits [11, 13, 16]. Due to the mismatch between operand length and word-size, an n-bit Montgomery multiplication can not be directly executed on a "small" w-bit multiplier. Hence, the long operands are processed in w-bit words according to software algorithms such as Coarsely Integrated Operand Scanning [7]. Besides the multiplier, a word-level architecture also comprises a set of registers for storing operands and results, as well as a state machine for control. Word-level architectures have the advantage that the size w of the multiplier is naturally "decoupled" from the operand length n, which facilitates the implementation of a scalable architecture for Montgomery multiplication (see [16, 17] for exemplary designs).

2.2. Montgomery Multiplication in Software

All software implementations of Montgomery multiplication have in common that they store the long operands in arrays of w-bit words, with w referring to the processor's

[1] The moduli used in cryptographic applications are either primes or products of primes, which means that M is generally an odd number.

wordsize. Algorithms for multiple-precision arithmetic operate on the single-recision words in these arrays using the instructions provided by the processor. A common software algorithm for Montgomery multiplication is the Coarsely Integrated Operand Scanning (CIOS) method, which can be efficiently implemented in many programming languages including C and Java. The CIOS method has a nested loop structure with a relatively simple inner loop, each iteration of which performs a single-precision multiplication (i.e. a $(w \times w)$-bit multiply instruction) and four single-precision additions [7]. Executing an $(n \times n)$-bit CIOS Montgomery multiplication on a w-bit processor requires performing a total of $2k^2 + k$ single-precision multiplications (k denotes the number of w-bit words an n-bit long operand consists of, i.e. $k = \lceil n/w \rceil$). An in-depth description and analysis of the CIOS method can be found in [7].

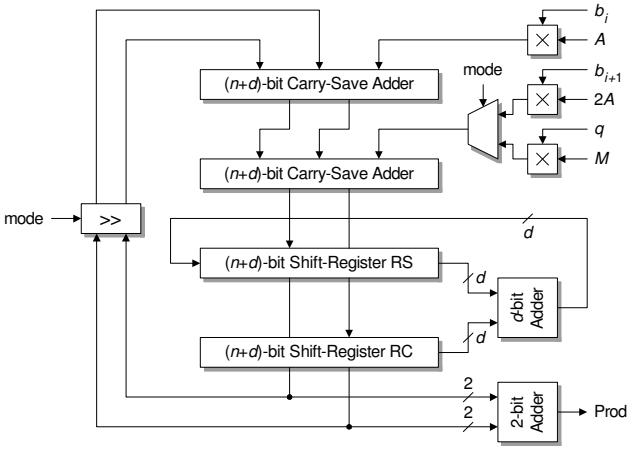

Figure 1. Datapath of our bit-serial multiplier

3. Scalable Bit-Serial Montgomery Multiplier

As noted in Section 1.2, our scalable multiplier is a bit-serial Montgomery multiplier with the ability of performing not just modular multiplications, but also conventional multiplications without modular reduction. Figure 1 shows the multiplier datapath, which essentially consists of two carry-save adders (CSAs) and two basic shift-registers. In each clock cycle, the CSAs add a partial product and a multiple of the modulus to a running sum. However, both CSAs add partial products when the multiplier performs a conventional multiplication without reduction. The main advantage of CSAs is that they avoid carry propagation by using a redundant representation, the so-called carry-save form, which makes the multiplier's critical path delay relatively independent of the precision of its datapath. Besides the CSAs, the datapath also contains two registers, *RS* and *RC*, in which the sum and carry vector of the running sum are held. A modular multiplication can be executed in this datapath in a similar way as shown Algorithm 1; the main difference is that we get the result in carry-save form. The result can be converted from carry-save to standard binary form with help of the d-bit Adder in Figure 1, which is simply a carry-propagation adder. Typical values for the width d of this adder are 8, 16, and 32 bits.

The CSAs and shift-registers shown in Figure 1 have a length of $n+d$ bits, whereby n refers to the operand-size the multiplier is dimensioned for. Modular multiplications with operands of up to n bits can be directly performed in the datapath; modular multiplications with longer operands (i.e. operands exceeding n bits) must be realized through a sequence of conventional multiplications. Even though our architecture does not pose any restrictions on n and d, the implementation gets easier when n is a multiple of d. We use a datapath of $(n+d)$-bit precision (instead of a more intuitive n-bit datapath) because this "extended" precision allows one to avoid the final subtraction of the modulus in the Montgomery multiplication (line 7 of Algorithm 1), as described in [21]. The registers *RS* and *RC* are supposed to be able to carry out d-bit right shift operations.

3.1. Execution of a Montgomery Multiplication

The multiplier datapath shown in Figure 1 supports two modes of operation; in the first mode it is able to perform a modular multiplication, whereas the second mode allows for performing an ordinary multiplication. When operating in the former mode, the datapath is similar to that of the Four-to-Two CSA architecture from [10] and executes a Montgomery multiplication according to Algorithm 1. The upper CSA adds a partial product of the form $A \cdot b_i$ to a running sum given in carry-save representation. This partial product can be easily generated via a logical AND of each bit of the multiplicand A and the multiplier-bit b_i, an operation that requires just an array of AND gates. The second CSA adds $q \cdot M$ (i.e. a multiple of the modulus M) to the output of the first CSA. Since q is either 0 or 1 when using the binary (i.e. radix-2) version of Montgomery's reduction technique, the generation of $q \cdot M$ is simply a matter of logical AND operations [5]. The sum and carry output of the second CSA are buffered in the *RS*, *RC* register pair for one cycle and then fed back to the first (i.e. upper) CSA. This feedback path is implemented in such a way that the sum and carry vector are shifted right by one position to align them properly for the addition of the next partial product.

After n clock cycles, when all partial products have been added, the result is available in the *RS*, *RC* register pair in carry-save representation. Using the d-bit carry-propagation adder (CPA), the result can be converted to standard binary form in $(n+d)/d$ cycles, following the approach described in [10]. In each clock cycle, a d-bit word of the sum and carry vector is shifted out from the *RS*, *RC* registers, starting

with the least significant word. The words are added up by the CPA, and the resulting d-bit sum is shifted back into register *RS*. A straightforward implementation of bit-serial Montgomery multiplication according to Algorithm 1 often requires a final subtraction of M to ensure that the result is completely reduced. However, this final subtraction can be avoided by modifying Algorithm 1 such that it computes $A \cdot B \cdot 2^{-(n+2)} \bmod M$ instead of the standard Montgomery product $A \cdot B \cdot 2^{-n} \bmod M$. This implies an increase in the number of loop iterations from n to $n+2$ and necessitates a slightly longer multiplier datapath, but allows one to use an incompletely reduced result in the range $[0, 2M-1]$ as input for the next modular multiplication (see [1, 22] for a more detailed description).

A datapath with a precision of $n+d$ bits also eliminates the need of performing final subtractions, but simplifies the implementation compared to an $(n+2)$-bit datapath. The execution time of a Montgomery multiplication producing $A \cdot B \cdot 2^{-(n+d)} \bmod M$ as result is $n+d+(n+d)/d$ cycles (including conversion to binary form). For example, when $n = 1024$ and $d = 32$, we get an execution time of 1,089 cycles, the vast majority of which is used for the addition of partial products. The time spent on redundant-to-binary conversion is almost negligible for typical values of n and d, i.e. an n-bit Montgomery multiplication requires roughly n clock cycles.

When using the radix-2 (i.e. bit-serial) version of Montgomery multiplication shown in Algorithm 1, the multiple of M being added in line 4 is determined by the LSB of the running sum Z after addition of the partial product $A \cdot b_i$. In our multiplier, Z is given in carry-save form; therefore, the bit q in Figure 1 is nothing else than the logical XOR of the LSB of the sum and carry vector representing Z. The generation of $q \cdot M$ lies on the critical path of a typical bit-serial Montgomery multiplier, but this delay can be reduced via some slight modifications of the datapath, such as the ones described in [4] and [5].

3.2. Execution of a Conventional Multiplication

An ordinary multiplication (without reduction) is carried out in a very similar way as the Montgomery multiplication described before, except that both CSAs are used to add partial products. In each clock cycle, the first (i.e. upper) CSA adds the partial product $A \cdot b_i$ (which is generated as explained in Section 3.1) to a running sum given in carry-save representation, whereas the second CSA adds a partial product of the form $2A \cdot b_{i+1}$. The factor of 2 in the second partial product, which is necessary since bit b_{i+1} has twice the weight of b_i, is realized in terms of a "hard-wired" left shift by one position. After the second CSA, the sum and carry vector are buffered for one cycle in the registers *RS* and *RC*, and then fed back to the first CSA. However, the feedback path includes a 2-bit right-shift operation (instead of the 1-bit right-shift that is carried out when executing a modular multiplication) in order to align the sum and carry vector for the addition of the next partial product. The two LSBs in the registers *RS* and *RC* constitute a part of the final result (i.e. the $2n$-bit product $A \cdot B$); they are summed up by the 2-bit Adder shown in Figure 1 and shifted into a register[2]. After all n partial products have been added, the lower half (i.e. the n LSBs) of the $2n$-bit product $A \cdot B$ is available in said register in binary representation, whereas the upper n-bit half is stored in the *RS*, *RC* register pair in carry-save form. The conversion from carry-save to binary form can be accomplished with help of the d-bit adder as described in the previous subsection[3]

Implementation	Time	Basic operation
Paillier [15]	$9n$	Modular multiplication
Fischer and Seifert [6]	$6n$	Euclidean multiplication
Our scalable multiplier	$5n$	Standard multiplication

Table 1. Timings of double-size mod. mult.

A multiplication of two n-bit operands using a bit-serial multiplier requires the generation and addition of n partial products. Our multiplier processes two partial products per clock cycle; consequently, an n-bit multiplication takes $n/2$ clock cycles. The conversion of the upper n-bit half of the product from carry-save into binary representation requires $(n+d)/d$ cycles, which results in $n/2+(n+d)/d$ cycles altogether. However, as mentioned in Section 3.1, the time needed for redundant-to-binary is almost negligible, thus the execution time of an n-bit multiplication is roughly $n/2$ clock cycles. A "double-size" Montgomery multiplication (i.e. a $2n$-bit Montgomery multiplication performed on an n-bit multiplier) consists of ten n-bit multiplications when using the CIOS method as explained in Section 2.2, which amounts to $5n$ clock cycles when ignoring other operations such as n-bit additions or operand transfers. However, since n-bit additions require just a fraction of the execution time of n-bit multiplications, they are often ignored in high-level performance evaluations (as is the case with most previous work, e.g. [3, 6]). Also the cost of operand transfers can be ignored if the multiplier provides some local storage or is equipped with a fast interface to the system's RAM [9].

Table 1 compares the execution time for a double-size modular multiplication of our multiplier with the execution times of [6, 15] as described in Section 1.1 (n refers to the execution time of a "normal-size" modular multiplication).

[2]Note that Figure 1 does not show the registers holding the operands A and B, as well as the register in which the modulus M is stored when the multiplier executes a Montgomery multiplication. However, this register is not needed when a conventional multiplication is performed; thus it can be used to store the lower part of the $2n$-bit product $A \cdot B$.

[3]The last addition of low-half bits in the 2-bit Adder may generate a "carry-out," which must be considered in the addition of the higher part.

4. Conclusions

We presented a new approach to overcome the operand-length restrictions of a conventional bit-serial architecture for Montgomery multiplication. The bit-serial multiplier we propose is able to perform Montgomery multiplications as well as ordinary multiplications without modular reduction operation. When dimensioned for a precision of n bits, our multiplier executes a Montgomery modular multiplication with operands of length (at most) n bits "directly" in circa n clock cycles. A double-length (i.e. $2n$-bit) Montgomery multiplication can be performed through a sequence of ten ordinary n-bit multiplications, which takes about $5n$ cycles altogether (i.e. roughly the five-fold execution time of an n-bit Montgomery multiplication). In both execution modi (i.e. Montgomery multiplication and ordinary multiplication), the CSAs and register infrastructure of our bit-serial architecture are utilized in an optimal way. Therefore, the extra hardware cost in relation to a conventional bit-serial Montgomery multiplier is very low.

References

[1] L. Batina and G. Muurling. Montgomery in practice: How to do it more efficiently in hardware. In *Topics in Cryptology — CT-RSA 2002*, vol. 2271 of LNCS, pp. 40–52. Springer Verlag, 2002.

[2] T. Blum and C. Paar. High-radix Montgomery modular exponentiation on reconfigurable hardware. *IEEE Transactions on Computers*, 50(7):759–764, July 2001.

[3] B. Chevallier-Mames, M. Joye, and P. Paillier. Faster double-size modular multiplication from Euclidean multipliers. In *Cryptographic Hardware and Embedded Systems — CHES 2003*, vol. 2779 of LNCS, pp. 214–227. Springer Verlag, 2003.

[4] A. Cilardo, A. Mazzeo, L. Romano, and G. Saggese. Carry-save Montgomery modular exponentiation on reconfigurable hardware. In *Proceedings of the 7th Conference on Design, Automation and Test in Europe (DATE 2004), Designers' Forum*, pp. 206–211. IEEE Computer Society Press, 2004.

[5] A. Daly and W. Marnane. Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic. In *Proceedings of the 10th ACM Symposium on Field Programmable Gate Arrays (FPGA 2002)*, pp. 40–49. ACM Press, 2002.

[6] W. Fischer and J.-P. Seifert. Increasing the bitlength of a crypto-coprocessor. In *Cryptographic Hardware and Embedded Systems — CHES 2002*, vol. 2523 of LNCS, pp. 71–81. Springer Verlag, 2002.

[7] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[8] Ç. K. Koç and C. D. Walter. Montgomery arithmetic. In *Encyclopedia of Cryptography and Security*, pp. 394–398. Springer Verlag, 2005.

[9] M. Koschuch et al. Hardware/software co-design of elliptic curve cryptography on an 8051 microcontroller. In *Cryptographic Hardware and Embedded Systems — CHES 2006*, vol. 4249 of LNCS, pp. 430–444. Springer Verlag, 2006.

[10] C. McIvor, M. McLoone, J. V. McCanny, A. Daly, and W. Marnane. Fast Montgomery modular multiplication and RSA cryptographic processor architectures. In *Conference Record of the 37th Asilomar Conference on Signals, Systems, and Computers*, vol. 1, pp. 379–384. IEEE, 2003.

[11] C. McIvor, M. McLoone, and J. V. McCanny. FPGA Montgomery multiplier architectures – A comparison. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, pp. 279–282. IEEE Computer Society Press, 2004.

[12] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.

[13] K. Mukaida, M. Takenaka, N. Torii, and S. Masui. Design of high-speed and area-efficient Montgomery modular multiplier for RSA algorithm. In *Digest of Technical Papers of the 18th Symposium on VLSI Circuits*, pp. 320–323. IEEE, 2004.

[14] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In *Proceedings of the 12th IEEE Symposium on Computer Arithmetic (ARITH '95)*, pp. 70–77. IEEE Computer Society Press, 1995.

[15] P. Paillier. Low-cost double-size modular exponentiation or how to stretch your cryptoprocessor. In *Public Key Cryptography — PKC '99*, vol. 1560 of LNCS, pp. 223–234. Springer Verlag, 1999.

[16] A. Satoh and K. Takano. A scalable dual-field elliptic curve cryptographic processor. *IEEE Transactions on Computers*, 52(4):449–460, Apr. 2003.

[17] H.-K. Son and S.-G. Oh. Design and implementation of scalable low-power Montgomery multiplier. In *Proceedings of the 22nd IEEE International Conference on Computer Design (ICCD 2004)*, pp. 524–531. IEEE Computer Society Press, 2004.

[18] A. F. Tenca and Ç. K. Koç. A scalable architecture for Montgomery multiplication. In *Cryptographic Hardware and Embedded Systems*, vol. 1717 of LNCS, pp. 94–108. Springer Verlag, 1999.

[19] A. F. Tenca and L. A. Tawalbeh. An efficient and scalable radix-4 modular multiplier design using recoding techniques. In *Conference Record of the 37th Asilomar Conference on Signals, Systems, and Computers*, vol. 2, pp. 1445–1450. IEEE, 2003.

[20] E. Trichina and A. Tiountchik. Scalable algorithm for Montgomery multiplication and its implementation on the coarse-grain reconfigurable chip. In *Topics in Cryptology — CT-RSA 2001*, vol. 2020 of LNCS, pp. 235–249. Springer Verlag, 2001.

[21] C. D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–378, Mar. 1993.

[22] C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 38(21):1831–1832, Oct. 1999.