Increasing the Bitlength of a Crypto-Coprocessor

Wieland Fischer and Jean-Pierre Seifert

Infineon Technologies
Security & ChipCard ICs
CC TI Concepts & Innovations
D-81609 Munich
Germany
{wieland.fischer,jean-pierre.seifert}@infineon.com

Abstract. We present a novel technique which allows a virtual increase of the bitlength of a crypto-coprocessor in an efficient and elegant way. The proposed algorithms assume that the coprocessor is equipped with a special modular multiplication instruction. This instruction, called `MultModDiv`(A, B, N) computes $A * B \bmod N$ and $\lfloor (A * B)/N \rfloor$. In addition to the doubling algorithm, we also present two conceivable economic implementations of the `MultModDiv` instruction: one hardware and one software realization. The hardware realization of the `MultModDiv` instruction has the same performance as the modular multiplication presented in the paper. The software realization requires two calls of the modular multiplication instruction. Our most efficient algorithm needs only six calls to an n-bit `MultModDiv` instruction to compute a modular $2n$-bit multiplication. Obviously, special variants of our algorithm, e.g., squaring, require fewer calls.

Keywords: Arithmetical coprocessor, Hardware architecture, Modular multiplication, Hardware/Software codesign.

1 Introduction

Fast modular multiplication algorithms have been extensively studied [Ba,DQ, HP1,HP2,Knu,Mo,Om,Pai,Q,Sed,WQ,Wa]. This is due to the fact that large integer arithmetic is essential for public-key cryptography. Recently, we have seen some progress of integer factorization [C$^+$] which demands for higher RSA bit lengths. On the other hand, for low cost and low power devices (e.g., in Smartcards, PDAs, Cellular Phones, etc.) one has to use hardware which does not provide sufficient bitlengths.

Unfortunately, these two requirements lead to a burden of the system issuer, e.g., the card industry. The source of this burden is the fact that, say 2048-bit RSA, cannot be handled efficiently on a 1024 bit device. Only with some work-around this problem becomes a manageable task. Namely, as it is now commonly known, one can use the Chinese Remainder Theorem for the RSA signature, see

B.S. Kaliski Jr. et al. (Eds.): CHES 2002, LNCS 2523, pp. 71–81, 2003.

[CQ]. To keep the RSA verification also relatively simple, most often the fourth Fermat number is used as public exponent. Only recently it was shown how to efficiently reduce such modular 2048-bit multiplications to 1024-bit modular multiplications, see [HP1,HP2,Pai]. Pailler [Pai] initiated this doubling research topic and formulated the following research problem:

Problem. Find an nk-bit modular multiplication algorithm using a minimal number of n-bit modular operations.

His algorithm needs nine modular multiplications for the case $k = 2$, see [Pai]. In this paper we will provide an answer to his question by presenting a novel doubling algorithm. Our most general and most efficient algorithm needs only six n-bit modular operations to compute a $2n$-bit modular multiplication. The idea for our family of algorithms is based on the fact that the coprocessor is equipped with a special modular multiplication instruction. This instruction is called `MultModDiv` and defined within the next section. An optimal realization is clearly achieved using an enhanced hardware modular multiplication instruction. Nevertheless, an efficient software realization of this instruction is possible. The software realization requires two calls to the modular multiplication instruction. Both realizations of this `MultModDiv` instruction will be presented.

The present paper is organized as follows: The next section gives the necessary definitions of `MultModDiv`. Section 3 explains our basic doubling algorithm, an enhanced version and also our special purpose variants. In section 4 we introduce the simple software emulation of the `MultModDiv` instruction. Finally, in section 5 we show how to realize the `MultModDiv` instruction in hardware.

2 Preliminaries

2.1 The Instructions `MultMod` and $\texttt{MultModInit}_n$

The following definition is the usual modular multiplication.

Definition 1. *For numbers A, B and N, $N > 0$, the* `MultMod` *instruction is defined as*

$$R = \texttt{MultMod}(A, B, N)$$

with

$$R := (A * B) \bmod N.$$

The following extension of the modular multiplication is already a feature of today's existing crypto coprocessors.

Definition 2. *For a fixed integer n and numbers A, B, C and N, $N > 0$, the* $\texttt{MultModInit}_n$ *instruction is defined as*

$$R = \texttt{MultModInit}_n(A, B, C, N)$$

with

$$R := (A * B + C * 2^n) \bmod N.$$

2.2 The Instructions MultModDiv and MultModDivInit$_n$

The following definition is a natural extension of the usual modular multiplication.

Definition 3. *For a fixed integer n and numbers A, B and N, $N > 0$, the* MultModDiv *is defined as*

$$(Q, R) = \texttt{MultModDiv}(A, B, N)$$

with

$$Q := \left\lfloor \frac{A * B}{N} \right\rfloor \quad \textit{and} \quad R := (A * B) - Q * N.$$

Definition 4. *For a fixed integer n and numbers A, B, C and N, $N > 0$, the* MultModDivInit$_n$ *instruction is defined as*

$$(Q, R) = \texttt{MultModDivInit}_n(A, B, C, N)$$

with

$$Q := \left\lfloor \frac{A * B + C * 2^n}{N} \right\rfloor \quad \textit{and} \quad R := (A * B + C * 2^n) - Q * N.$$

3 The Doubling Algorithm

3.1 Modular Multiplication without Initialization

We will start with the easiest of our algorithms, which needs 7 MultModDiv instructions on an n-bit processor.

Theorem 1. *There exists an algorithm to compute $A * B \bmod N$ using seven* MultModDiv *instructions of length n, provided that $2^{2n-1} \leq N < 2^{2n}$ and $0 \leq A, B < N$.*

Proof. We will first present the algorithm.

Basic Doubling Algorithm:
input: $N = N_t 2^n + N_b$ with $0 \leq N_b < 2^n$,
$A = A_t 2^n + A_b$ with $0 \leq A_b < 2^n$,
$B = B_t 2^n + B_b$ with $0 \leq B_b < 2^n$

$(Q^{(1)}, R^{(1)}) := \texttt{MultModDiv}(B_t, 2^n, N_t)$
$(Q^{(2)}, R^{(2)}) := \texttt{MultModDiv}(Q^{(1)}, N_b, 2^n)$
$(Q^{(3)}, R^{(3)}) := \texttt{MultModDiv}(A_t, R^{(1)} - Q^{(2)} + B_b, N_t)$
$(Q^{(4)}, R^{(4)}) := \texttt{MultModDiv}(A_b, B_t, N_t)$
$(Q^{(5)}, R^{(5)}) := \texttt{MultModDiv}(Q^{(3)} + Q^{(4)}, N_b, 2^n)$
$(Q^{(6)}, R^{(6)}) := \texttt{MultModDiv}(A_t, R^{(2)}, 2^n)$
$(Q^{(7)}, R^{(7)}) := \texttt{MultModDiv}(A_b, B_b, 2^n)$

$Q := (R^{(3)} + R^{(4)} - Q^{(5)} - Q^{(6)} + Q^{(7)})$
$R := (R^{(7)} - R^{(6)} - R^{(5)})$
make final reduction on $(Q * 2^n + R)$

output: $Q * 2^n + R$

We will prove that $(R^{(3)}+R^{(4)}-Q^{(5)}-Q^{(6)}+Q^{(7)})*2^n+(R^{(7)}-R^{(6)}-R^{(5)})$ is indeed congruent to $A*B$ modulo N. This can easily be seen from the following, where we use $Z = 2^n$ as abbreviation.

$$
\begin{aligned}
& (A_t Z + A_b) * (B_t Z + B_b) \\
&= A_t B_t Z Z + A_t B_b Z + A_b B_t Z + A_b B_b \\
&= A_t (Q^{(1)} N_t + R^{(1)}) Z + A_t B_b Z + A_b B_t Z + A_b B_b \\
&\equiv A_t R^{(1)} Z - A_t Q^{(1)} N_b + A_t B_b Z + A_b B_t Z + A_b B_b \\
&= A_t R^{(1)} Z - A_t (Q^{(2)} Z + R^{(2)}) + A_t B_b Z + A_b B_t Z + A_b B_b \\
&= A_t (R^{(1)} - Q^{(2)} + B_b) Z - A_t R^{(2)} + A_b B_t Z + A_b B_b \\
&= (Q^{(3)} N_t + R^{(3)}) Z - A_t R^{(2)} + A_b B_t Z + A_b B_b \\
&= (Q^{(3)} N_t + R^{(3)}) Z - A_t R^{(2)} + (Q^{(4)} N_t + R^{(4)}) Z + A_b B_b \\
&\equiv (R^{(3)} + R^{(4)}) Z - (Q^{(3)} + Q^{(4)}) N_b - A_t R^{(2)} + A_b B_b \\
&= (R^{(3)} + R^{(4)}) Z - (Q^{(5)} Z + R^{(5)}) - A_t R^{(2)} + A_b B_b \\
&= (R^{(3)} + R^{(4)}) Z - (Q^{(5)} Z + R^{(5)}) - (Q^{(6)} Z + R^{(6)}) + A_b B_b \\
&= (R^{(3)} + R^{(4)}) Z - (Q^{(5)} Z + R^{(5)}) - (Q^{(6)} Z + R^{(6)}) + (Q^{(7)} Z + R^{(7)}) \\
&= (R^{(3)} + R^{(4)} - Q^{(5)} - Q^{(6)} + Q^{(7)}) Z + (R^{(7)} - R^{(6)} - R^{(5)}) \bmod N
\end{aligned}
$$

The two congruences above are based on the fact that $N_t Z \equiv -N_b \bmod N$. Apart from the fact that this result still has to be reduced modulo N, this completes the proof. □

Practical Implementation Issues

1. Observe that in steps three and five negative numbers may occur. This can be resolved by the fact that for positive numbers A, B and N the equation $(Q, R) = \texttt{MultModDiv}(A, B, N)$ implies $(-Q - 1, N - R) = \texttt{MultModDiv}(A, -B, N)$, if $R \neq 0$.
2. It is possible that the intermediary output (Q, R) is not reduced, i.e., $0 \leq R < 2^n$ and $0 \leq Q < N_t$ is not fulfilled. In this case one has to do a final reduction: first, do $(Q, R) \leftarrow (Q \pm N_t, R \pm N_b)$ until Q is reduced modulo N_t. Then, do $(Q, R) \leftarrow (Q \pm 1, R \mp 2^n)$ until R is reduced modulo 2^n.
3. Using two parallel n-bit processors one only needs the time of four `MultModDiv` instructions.
4. If the given module N has an odd bitlength, then one has to compute with $2 * N$.

3.2 Modular Multiplication with Initialization

By using a $\texttt{MultModDivInit}_n$ instruction we can reduce the number of steps to six.

Theorem 2. *There exists an algorithm to compute* $A * B \bmod N$ *using five* `MultModDiv` *and one* $\texttt{MultModDivInit}_n$ *instruction of length* n*, provided that* $2^{2n-1} \leq N < 2^{2n}$ *and* $0 \leq A, B < N$.

Proof. We first present the algorithm.

Enhanced Basic Doubling Algorithm:
input: $N = N_t 2^n + N_b$ with $0 \leq N_b < 2^n$,
$A = A_t 2^n + A_b$ with $0 \leq A_b < 2^n$,
$B = B_t 2^n + B_b$ with $0 \leq B_b < 2^n$

$(Q^{(1)}, R^{(1)}) := \texttt{MultModDiv}(A_t, B_t, N_t)$
$(Q^{(2)}, R^{(2)}) := \texttt{MultModDivInit}_n(N_b, -Q^{(1)}, R^{(1)}, N_t)$
$(Q^{(3)}, R^{(3)}) := \texttt{MultModDiv}(A_t, B_b, N_t)$
$(Q^{(4)}, R^{(4)}) := \texttt{MultModDiv}(A_b, B_t, N_t)$
$(Q^{(5)}, R^{(5)}) := \texttt{MultModDiv}(A_b, B_b, 2^n)$
$(Q^{(6)}, R^{(6)}) := \texttt{MultModDiv}(Q^{(2)} + Q^{(3)} + Q^{(3)}, N_b, 2^n)$

$Q := (R^{(2)} + R^{(3)} + R^{(4)} + Q^{(5)} - Q^{(6)})$
$R := (R^{(5)} - R^{(6)})$
make final reduction on $(Q * 2^n + R)$

output: $Q * 2^n + R$

We will prove that $(R^{(2)} + R^{(3)} + R^{(4)} + Q^{(5)} - Q^{(6)}) * 2^n + (R^{(5)} - R^{(6)})$ is indeed congruent to $A * B$ modulo N. This can be seen from the following, where we use $Z = 2^n$ as abbreviation.

$$
\begin{aligned}
&(A_tZ + A_b) * (B_tZ + B_b) \\
&= A_tB_tZZ + A_tB_bZ + A_bB_tZ + A_bB_b \\
&= (Q^{(1)}N_t + R^{(1)})ZZ + A_tB_bZ + A_bB_tZ + A_bB_b \\
&\equiv (R^{(1)}Z - Q^{(1)}N_b)Z + A_tB_bZ + A_bB_tZ + A_bB_b \\
&= (Q^{(2)}N_t + R^{(2)})Z + A_tB_bZ + A_bB_tZ + A_bB_b \\
&\equiv (R^{(2)}Z - Q^{(2)}N_b) + A_tB_bZ + A_bB_tZ + A_bB_b \\
&= (R^{(2)}Z - Q^{(2)}N_b) + (Q^{(3)}N_t + R^{(3)})Z + A_bB_tZ + A_bB_b \\
&= (R^{(2)}Z - Q^{(2)}N_b) + (Q^{(3)}N_t + R^{(3)})Z + (Q^{(4)}N_t + R^{(4)})Z + A_bB_b \\
&= (R^{(2)}Z - Q^{(2)}N_b) + (Q^{(3)}N_t + R^{(3)})Z + (Q^{(4)}N_t + R^{(4)})Z + (Q^{(5)}Z + R^{(5)}) \\
&\equiv (R^{(2)} + R^{(3)} + R^{(4)} + Q^{(5)})Z - (Q^{(2)} + Q^{(3)} + Q^{(4)})N_b + R^{(5)} \\
&= (R^{(2)} + R^{(3)} + R^{(4)} + Q^{(5)})Z - (Q^{(6)}Z + R^{(6)}) + R^{(5)} \\
&= (R^{(2)} + R^{(3)} + R^{(4)} + Q^{(5)} - Q^{(6)})Z + (R^{(5)} - R^{(6)}) \bmod N
\end{aligned}
$$

The three congruences above are based on the fact that $N_tZ \equiv -N_b \bmod N$. Apart from the fact that this result still has to be reduced modulo N, this completes the proof. □

Practical Implementation Issues

1. Observe that in steps two and six negative numbers may occur. This can be resolved as shown above.
2. It is possible that the intermediary output (Q, R) is not reduced. This can be resolved as shown above.
3. Using two parallel n-bit processors one only needs the time of three `MultModDiv` instructions.
4. Again, if the given module N has an odd bitlength, then one has to compute with $2 * N$.

3.3 Optimized Special Purpose Variants

Now the basic strategy of our algorithms should be clear. Therefore, we will present the results for special purpose variants.

Squaring

Theorem 3. *There exists an algorithm to compute* $A^2 \bmod N$ *using six* `MultModDiv` *instructions of length* n*, provided that* $2^{2n-1} \leq N < 2^{2n}$ *and* $0 \leq A < N$.

If we consider the algorithm of section 3.2 for the case $A = B$, we see that steps three and four are identical. Therefore, we get the following result:

Theorem 4. *There exists an algorithm to compute* $A^2 \bmod N$ *using four* `MultModDiv` *and one* $\texttt{MultModDivInit}_n$ *instruction of length* n*, provided that* $2^{2n-1} \leq N < 2^{2n}$ *and* $0 \leq A < N$.

Precomputation

If the factor B is known in advance (e.g., square and *multiply* for exponentiation), then the first and second computation of the algorithm of section 3.1 can be carried out in advance. Therefore, the multiplication can be done in five steps.

Theorem 5. *There exists an algorithm to compute* $A * B \bmod N$ *using five* `MultModDiv` *instructions of length* n*, provided that* $2^{2n-1} \leq N < 2^{2n}$ *and* $0 \leq A, B < N$*, where* B *is known in advance.*

Using a completely different idea, one needs only six `MultModDiv` steps. This time, one uses a special representation of A and B. Namely, $A = A_t * N_t + A_b$ and $B = B_t * N_t + B_b$, where $N_t := \lfloor \sqrt{N} \rfloor$.

Theorem 6. *There exists an algorithm to compute* $A * B \bmod N$ *using six* `MultModDiv` *instructions of length* n*, provided that* $2^{2n-1} \leq N < 2^{2n}$ *and* $0 \leq A, B < N$*, where* N *is known in advance..*

4 Software Realization of the `MultModDiv` and $\texttt{MultModDivInit}_n$ Instructions

This section presents a software emulation of the `MultModDiv` and $\texttt{MultModDivInit}_n$ instructions.

Theorem 7. *There exists an algorithm to compute* $\texttt{MultModDiv}(A, B, N)$ *using two* `MultMod` *instructions, provided that* $0 \leq A, B < N$.

Proof. We present the simple algorithm.

Simulation of `MultModDiv`:
input: A, B, N with $0 \leq A, B < N$

$R := \texttt{MultMod}(A, B, N)$
$N' := N + 1$
$R' := \texttt{MultMod}(A, B, N')$
$Q := R - R'$
if $(Q < 0)$ **then**
  $Q := Q + N'$
fi

output: (Q, R)

We will prove that the former algorithm correctly computes the `MultModDiv` instruction. For given inputs A, B and N there exists some Q and R with

$$A * B = Q * N + R \quad \text{and} \quad R = (A * B) \bmod N,$$

where $0 \leq R < N$ and $0 \leq Q < N - 1$. Equivalently, we also have

$$A * B = Q * (N + 1) + (R - Q).$$

For $(R-Q) \geq 0$ this means $(R-Q) = (A*B) \bmod (N+1)$ and for $(R-Q) < 0$ this means $(R-Q)+(N+1) = (A*B) \bmod (N+1)$. Thus, for Q we have

$$Q = ((A*B) \bmod N) - ((A*B) \bmod (N+1))$$

or, if this value is less than zero

$$Q = ((A*B) \bmod N) - ((A*B) \bmod (N+1)) + (N+1).$$

This completes the proof. □

In a similar way the $\texttt{MultModDivInit}_n$ instruction is emulated by the $\texttt{MultModInit}_n$ instruction.

Theorem 8. *There exists an algorithm to compute* $\texttt{MultModDivInit}_n$ (A, B, C, N) *using two* $\texttt{MultModInit}_n$ *instructions, provided that* $2^{n-1} \leq N < 2^n$ *and* $0 \leq A, B, C < N$.

Proof. We present the algorithm.

Simulation of $\texttt{MultModDivInit}_n$:
input: A, B, C, N with $0 \leq A, B, C < N$

$R := \texttt{MultModInit}_{n+2}(2A, 2B, C, 4N)$
$N' := 4N + 1$
$R' := \texttt{MultModInit}_{n+2}(2A, 2B, C, N')$
$Q := R - R'$
if $(Q < 0)$ **then**
  $Q := Q + N'$
fi

output: $(Q, R/4)$

The proof is a derivation of the former one, leaving the modifications to the reader. However, we note that bounding the size of the quotient Q is the crucial point. □

Both algorithms can be extended to algorithms also working for non-reduced A, B and C. This is necessary for our doubling algorithms.

5 Hardware Realization of the MultModDiv and $\texttt{MultModDivInit}_n$ Instructions

We will now sketch how an algorithm for the MultMod instruction can be extended into an algorithm for the MultModDiv instruction. We first consider the textbook MultMod implementation.

Textbook `MultMod` implementation:
input: A, B, N with $0 \leq A, B < N$, and $A = (A_{n-1}, \ldots, A_0)$
$i := n; Z := 0$
repeat
  $i := i - 1$
  case A_i **is**
    0**:** $Z := 2 * Z$
    1**:** $Z := 2 * Z + B$
  end case
  if $(Z \geq N)$ **then**
    $Z := Z - N$
    if $(Z \geq N)$ **then**
      $Z := Z - N$
    fi
  fi
until $(i = 0)$
output: Z

The extension is rather trivial. During the modular multiplication we simply have to "count" the number of subtracted N's. Observe that during a modular multiplication this implicit information is always known to the algorithm/hardware.

`MultModDiv` implementation:
input: A, B, N with $0 \leq A, B < N$ and $A = (A_{n-1}, \ldots, A_0)$
$i := n; Z := 0$
repeat
  $i := i - 1$
  case A_i **is**
    0**:** $Z := 2 * Z$
    1**:** $Z := 2 * Z + B$
  end case
  $Q_i := 0;\ Q'_i := 0$
  if $(Z \geq N)$ **then**
    $Z := Z - N$
    $Q_i := 1;\ Q'_i := 0$
    if $(Z \geq N)$ **then**
      $Z := Z - N$
      $Q_i := 1;\ Q'_i := 1$
    fi
  fi
until $(i = 0)$
$Q := Q + Q'$
output: (Q, Z)

In the paper's full version we will actually show how the former algorithm can be simply integrated into the modular multiplication algorithm due to H. Sedlak [Sed].

The $\texttt{MultModDivInit}_n$ and $\texttt{MultModInit}_n$ are derived from the former ones essentially by exchanging the step $Z := 0$ with $Z := C$.

6 Conclusion

In this paper we have introduced new efficient algorithms to compute $2n$-bit modular multiplications using only n-bit modular multiplications. Using the `MultModDiv` and $\texttt{MultModDivInit}_n$ instructions we were able to improve the results presented by Pailler [Pai]. The question of what is the minimal number of multiplications is still open, as we currently have no proof of the optimality of our algorithm.

Acknowledgments. We would like to thank Holger Sedlak for several valuable discussions on this topic.

References

[Ba] P. Barret, "Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor", *Proc. of CRYPTO '86*, Springer LNCS, vol. 263, pp. 311–323, 1987.

[C$^+$] S. Cavallar *et alii*, "Factoring a 512 bit RSA modulus", *Proc. of EUROCRYPT '00*, Springer LNCS, vol. 1807, pp. 1–19, 2000.

[CQ] C. Couvreur, J.-J. Quisquater, "Fast decipherment algorithm for RSA public-key cryptosystem", *Electronics Letters* **18**(21):905–907, 1982.

[DQ] J.-F. Dhem, J.-J. Quisquater, "Recent results on modular multiplication for smart cards", *Proc. of CARDIS '98* Springer LNCS vol. 1820, pp. 336–352, 1998.

[HP1] H. Handschuh, P. Pailler, "Smart Card Crypto-Coprocessors for Public-Key Cryptography", *CryptoBytes* **4**(1):6–11, 1998.

[HP2] H. Handschuh, P. Pailler, "Smart Card Crypto-Coprocessors for Public-Key Cryptography", *Proc. of CARDIS '98* Springer LNCS vol. 1820, pp. 372–379, 1998.

[Knu] D. E. Knuth, *The Art of Computer Programming, Vol.2: Seminumerical Algorithms*, 3rd ed., Addison-Wesley, Reading MA, 1999.

[MvOV] A. J. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997.

[Mo] P. L Montgomery, "Modular Multiplication without Trial Division", *Math. of Computation*, vol. **44**, pp. 519–521, 1985.

[NMR] D. Naccache, D. M'Raihi, "Arithmetic co-processors for public-key cryptography: The state of the art", *IEEE Micro*, pp. 14–24, 1996.

[Om] J. Omura, "A public key cell design for smart card chips", *Proc. of IT Workshop*, pp. 27–30, 1990.

[Pai] P. Pailler, "Low-cost double size modular exponentiation or how to stretch your cryptocoprocessor", *Proc. of Public Key Cryptography '99*, Springer LNCS, vol. 1560, pp. 223–234, 1999.

[Q] J.-J. Quisquater, "Encoding system according to the so-called RSA method, by means of a microcontroller and arrangement implementing this system", U.S. Patent #5,166,979, Nov. 24, 1992.

[RSA] R. Rivest, A. Shamir, L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Comm. of the ACM* **21**:120–126, 1978.

[Sed] H. Sedlak, "The RSA cryptographic Processor: The first High Speed One-Chip Solution", *Proc. of EUROCRYPT '87*, Springer LNCS, vol. 293, pp. 95–105, 198.

[WQ] D. de Waleffe, J.-J. Quisquater, "CORSAIR, a smart card for public-key cryptosystems", *Proc. of CRYPTO '90*, Springer LNCS, vol. 537, pp. 503–513, 1990.

[Wa] C. Walter, "Techniques for the Hardware Implementation of Modular Multiplication", *Proc. of 2nd IMACS Internat. Conf. on Circuits, Systems and Computers*, vol. **2**, pp. 945–949, 1998.