Architectures for Montgomery's multiplication

O. Nibouche, A. Bouridane and M. Nibouche

Abstract: Many public key cryptographic algorithms require modular multiplication of very large operands as their core arithmetic operation. One method to perform this operation reasonably fast is to use specialised hardware. However, larger sizes are often required to increase security. This comes at the expense of either reducing the clock rate or dramatically increasing the size and hence the cost of the system. Therefore, techniques that allow efficient and fast computation of this operation at the algorithmic level are desired. An algorithm/structure for the computation of Montgomery's modular multiplication is presented. The modified algorithm splits the original algorithm into two multiplication operations, which can be executed in parallel. The derived architectures can be pipelined to the bit-level by interleaving multiple modular multiplication operations onto the same structure.

1 Introduction

Recently, there has been growing interest and considerable research activity related to developing algorithms and architectures for modular multiplication for use in cryptography. Several factors have fuelled this work, among them, the security requirement of the booming electronic commerce sector, where the safety of the transactions is becoming a major concern.

Privacy and fraud concerns can be addressed through the use of various cryptographic primitives such as data encryption and user/message authentication, which can be used with the appropriate protocols in order to construct secure and trusted communication systems. One of the main and best-known public key systems is the RSA cryptosystem. This cryptosystem performs fast modular exponentiation operations on numbers with a size of hundreds of bits [1], which in turn, can be broken down into a series of modular multiplication operations. Therefore, the implementation of such systems, which requires efficient architectures to compute the modular product, has motivated the development of a number of modular multiplication algorithms and architectures [2–9].

A clear distinction between the algorithms used for modular multiplication can be based upon the data format. Two major classes of algorithms can be distinguished: most significant bit first (MSBF) algorithms and least significant bit first (LSBF) algorithms. The MSBF class of modular algorithms are either comparison-/subtraction-based algorithms or look up table (LUT)-based algorithms. Alternatively, Montgomery's algorithm [10] makes the LSBF approach useful when performing numerous successive modular multiplication operations, such as modular exponentiation. The algorithm is used to speed up both the modular multiplication and squaring operations required during the exponentiation process. Rather than calculating the residue of the division operation as do the MSBF methods, the algorithm computes the modular product of two numbers multiplied by a scaling factor, which is relatively prime to the modulus. This allows the algorithm to perform divisions by a power of two, which is a shift operation, thus making the design of modulo multipliers easier.

When implementing Montgomery's algorithm, avoiding the global broadcast of data is very important. Such a global broadcast can lower the clock frequency [2–8]. As the operands size is large, a digit implementation of the algorithm can be the right solution to avoid large hardware usage of the parallel-based implementations [11]. Pipelining the digit structures is critical. As a matter of fact, by changing the digit size and the level of pipelining, the designer can find the best speed-area usage trade-off [11].

A modified version of Montgomery's multiplication algorithm will be presented. The main feature of the proposed method is that the algorithm is split into two concurrent multiplication operations. The first one consists of a conventional multiplication operation of the two operands while the second is a reduction operation, which is needed to ensure that the partial results remain in the range required by the algorithm. This reduction operation is nothing more than a multiplication operation carried out in such a way that the least significant part of the result is the two's complement of the least significant part of the first multiplication operation. The derived architectures can be pipelined to the bit-level by interleaving multiple modular multiplication operations onto the same structure. Therefore, a fully systolic bit-serial parallel structure can be designed for interleaving just two of Montgomery's multiplication operations onto the same multiplier. The systolicity of the structures is an important feature to achieve high frequencies of operation [12]. This is achieved by implementing the multiplication algorithm in a pipelined fashion and by avoiding the global broadcast of the data lines [2–8].

IEE Proceedings online no. 20030567

doi: 10.1049/ip-cdt:20030567

Paper first received 13th December 2002 and in revised form 22nd April 2003

O. Nibouche is with the Faculty of Informatics, Ulster University at Magee, Northland Road, Londonderry BT48 7JL, UK

A. Bouridane is with the School of Computer Science, Queen's University Belfast, Belfast BT7 1NN, UK

M. Nibouche is with the Faculty of Computing, Engineering and Mathematical Sciences, University of the West of England, Bristol BS16 1QY, UK

2 Modular exponentiation and RSA cryptography

In 1977, Rivest *et al.* [13] introduced what has become one of the most successful and widely used public key cryptosystem based on computing modular exponentiations. The level of security offered by RSA derives from the difficulty of efficiently factorising large numbers. As an example [1], hundreds of computers were used worldwide for a number of months in order to factorise a 513-bit integer. For the purpose of keeping the data as secure and secret as possible, current implementations of the RSA use a 1 Kb key wordlength or longer [1], which appears to be safe within the current state-of-the-art of the technology.

As was proposed by [13], the modulus M of the RSA algorithm is the product of two suitably generated secret prime numbers P and Q: $M = P \times Q$. The public exponent (also known as the encryption key) E is randomly chosen such that it is prime to $(P-1)(Q-1)$. The secret exponent (also known as the decryption key) D is computed using the extended Euclidean algorithm:

$$\langle E \times D \rangle_{(P-1)(Q-1)} = \langle 1 \rangle_{(P-1)(Q-1)} \tag{1}$$

where $\langle\,\rangle_M$ denotes a modulo M operation. The two numbers P and Q are no longer needed. They should be discarded but never revealed.

A $n = km$ bits message is divided into m blocks of k-bit integers each. A k-bit word A is encrypted as enc(A) defined as follows: $\text{enc}(A) = \langle A^E \rangle_M$.

The decryption operation is defined by dec(B) defined as follows: $\text{dec}(B) = \langle B^D \rangle_M$.

The encryption and decryption operations are mutual inverses, i.e.:

$$\text{enc}(\text{dec}(A)) = \text{dec}(\text{enc}(A)) = A \quad \text{with} \quad 0 \leqslant A < M$$

This indicates that the original data can be recovered through the RSA encrypt/decrypt process.

A standard way to perform the modular exponentiation is by using repeated modular multiplication and squaring operations, where the exponentiation is carried out iteratively, as shown in algorithm 1 for the case of a LSBF scheme. The algorithm computes B, which is the remainder of the division of A^E by the modulus M. In each iteration, two modular multiplications are carried out, which can be done in parallel or pipelined using the same multiplier structure. Many RSA crypto-system implementations have been proposed in the literature [2–9]. Ultimately, the speed of the system depends on the speed of the modular multiplier; therefore a high performance Montgomery's modular multiplier structure is crucial for such systems.

Algorithm 1:

$B = \langle A^E \rangle_M$
$E = \sum_{i=0}^{n-1} e_i 2^i, P_0 = 1, B_0 = A, B = P_n$
for $i = 0$ to $n-1$
{
$B_{i+1} = \langle B_i^2 \rangle_M$
if $e_i = 1$ then $P_{i+1} = \langle P_i B_i \rangle_M$
else $P_{i+1} = P_i$
}

3 Montgomery's multiplication: background

Let the modulus M be an integer within the range $[2^{n-1}, 2^n]$ and let R be 2^n. Montgomery's multiplication algorithm 2 requires R and M to be relatively prime, i.e. $\gcd(R, M) = \gcd(2^n, M) = 1$, which is satisfied if M is odd as is required by the algorithm. By exploiting this property, the Montgomery reduction algorithm introduces an efficient multiplication scheme, which computes the modular product, P, of two given integers, A and B, as follows [10]:

$$P = \langle ABR^{-1} \rangle_M \tag{2}$$

where R^{-1} is the inverse of R modulo M. In order to describe his algorithm, Montgomery introduced the quantity, M', which is the inverse of $-M$ modulo R, i.e.:

$$M' = \langle -M^{-1} \rangle_R \tag{3}$$

The computation of the Montgomery multiplication is carried out using algorithm 2 as follows:

Algorithm 2:

{
$T = AB$
$P = (T + \langle TM' \rangle_R M)/R$
if $P \geqslant M$ then $P = P - M$
}

The algorithm uses the multiplication modulo R and the division by R, which are faster and simpler than the computation of AB mod M that involves division by M. The algorithm is only efficient when multiple operations are carried out, such as in the modular exponentiation operation once broken into modular multiplication operations.

For a hardware implementation, a systolic array can be derived from the bit-wise version of the Montgomery multiplication as shown in algorithm 3 [10]:

Algorithm 3:

$0 < A = \sum_{i=0}^{n} a_i 2^i < M, 0 < B < R, P_{-1} = 0$
{
for $i = 0$ to $n-1$
$q_i = \langle P_{i-1} + a_i B \rangle_2$
$P_i = P_{i-1} + a_i B + q_i M/2$
}
if $P_{n-1} \geqslant M$ then $P_{n-1} = P_{n-1} - M$

Algorithm 3 interleaves the multiplication steps with the reduction steps. As shown in the algorithm, the LSB of the partial result of the previous iteration, P_{i-1}, together with the bit-product $a_i b_0$, directly selects the modular reduction value, which is either zero or M. Therefore, the LSB of P_i equals zero. The partial result is then shifted one position to the right. After n iterations of the algorithm, the scaling factor is equal to 2^{-n}. Therefore, the final result is $\langle AB2^{-n} \rangle_M$ The partial results fall in the range [0, $2M$[and as such an operation of comparison/subtraction is necessary at the end of the algorithm [10].

Several structures have been presented in the literature for the implementation of Montgomery's algorithm [2–9]. These architectures, which implement the interleaved algorithm 3, are either based on the use of carry save adders (CSAs) or carry propagate adders (CPAs). Fig. 1 depicts a bit parallel implementation of this algorithm based on a CSA multiplication scheme. Each iteration of this algorithm is implemented using two rows of gated full adders (gFAs). The first row is used for the multiplication part of the algorithm while the second part implements the reduction step. The bit serial parallel structure is derived from the bit parallel multiplier by a linear projection as shown in Fig. 2. An additional control signal is added so that

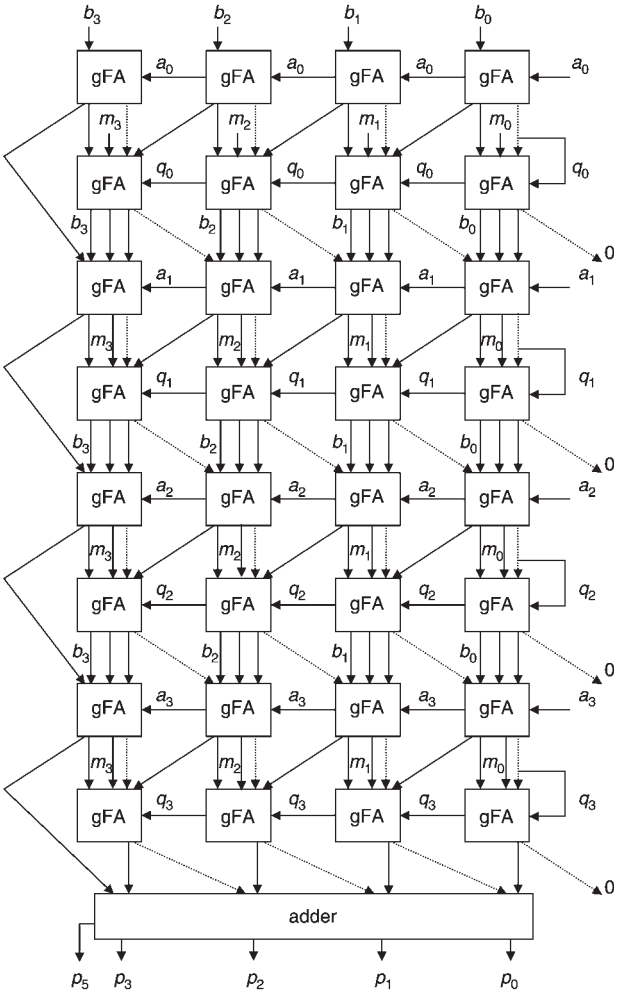

Fig. 1 *A parallel Montgomery's multiplier*

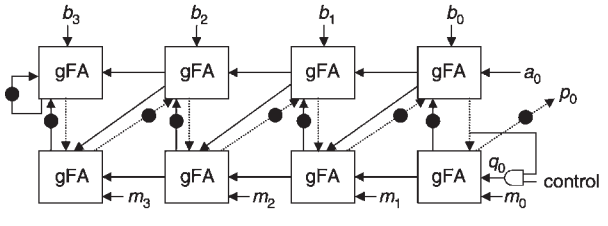

Fig. 2 *A bit-serial Montgomery's multiplier*

no reduction operation is carried out after the first n cycles since only the result and carry bits already generated are propagated to compute the correct result. However, the loops within the bit serial structure suggest a non-pipelined unfolded digit structure. Furthermore, the serial inputs are broadcast to all the cells. Such a global distribution lowers the clock frequency, and therefore should be eliminated. In the following sections architectures for Montgomery's modular multiplication are presented where the multiplication operation is carried out separately from the reduction operation but in a concurrent fashion. This has the merit of reducing the longest path of the multiplier to just one gFA.

4 A modified non-interleaved Montgomery's multiplier

As shown by Algorithm 3, Montgomery's modular multiplication algorithm is equivalent to two interleaved conventional multiplication operations, where the second one is a reduction operation which is required to keep the partial results within the range $[0, M + B]$.

In this Section, a modified Montgomery's algorithm and its structure are presented. This is achieved by splitting the initial algorithm into two multiplication operations that can be simultaneously carried out. Let T be the product of $A \times B$, i.e.:

$$T = AB = T_0 + T_1 R. \tag{4}$$

where T_0 and T_1 are the least significant word and most significant word of T, respectively.

To divide the modular multiplication into two concurrent multiplication operations, a function f is defined as:

$$0 = \langle f(T_0, M) \rangle_R \tag{5}$$

The function f performs a multiplication operation-controlled by T_0 of the modulus M in such a way that the result modulo R is zero. This operation can be carried out sequentially using the proposed algorithm 4, as follows:

Algorithm 4: A proposed technique

$P_{-1} = 0$
for $i = 0$ to $n - 1$
$\{q_i = P_{i-1,0} \oplus T_{0,i}$
$P_i = P_{i-1} + q_i M + T_{0,i}/2\}$
$P = P_{n-1} + T_1$

where $\oplus$ is the bit-wise exclusive OR operation, $T_{0,i}$ and $P_{i-1,0}$ are the ith and the LSB of T_0 and P_{i-1}, respectively. The reduction value, which is either M or zero, is selected by q_i in such a way the LSB of $P_i + q_I M$ is *0*. By simultaneously carrying out the multiplication of the product AB and the calculation of the function f of (5) using Algorithm 4, the result of (2) is computed and is equal to $P_{n-i} + T_1$. An implementation of this algorithm requires two multipliers and an adder. The first multiplier is used to implement the product AB while the second one is used to implement the function f. Therefore, this multiplier must be a carry propagate (CP) multiplier. In this particular case, the carry save (CS) multiplier cannot be used because at each LSB position, 4 bits need to be reduced to a result-bit and a carry-bit, which cannot be carried out by CS multipliers. Finally, an adder is used to accumulate the most significant part of the result from both multipliers. The final result falls within the range specified in [10], therefore an additional subtraction operation might be required in order to reduce the result back into the required range $[0, M[$.

A modification of the above algorithm, which allows an implementation using either CP multipliers and/or CS multipliers, is shown by algorithm 5. In contrast with algorithm 4, the term T_0 is not added to the partial results so that the LSB of the word P_i is low. Instead T_0 is subtracted, thus, the LSB of the partial results is equal to $\bar{T}_{0,i}$, where $\bar{T}_0$ is the two's complement of T_0. Therefore, when using a carry save multiplication scheme, only 3 bits are to be accumulated at every LSB position.

Algorithm 5: A modified version of algorithm 4 for implementation with CP and CS multipliers

$P'_{-1} = 0, c_0 = 1, P''_i = 0$
for $i = 0$ to $n - 1$
{
$P'_i = P'_{i-1} + 2a_i B$
$P'_{i,i} + c_i = \bar{T}_{0,i} + 2c_{i+1}$
$q_i = P''_{i,i} \oplus \bar{T}_{0,i}$

$P''_i = P''_{i-1} + q_i M\}$
$P = (P'_{n-1} + P''_{n-1})/2^n$
if $P \geqslant M$ then $P = P - M$
}

As previously shown in algorithm 4, the modular multiplication in this algorithm is split into two concurrent multiplication operations and computes the Montgomery multiplication by using the two's complement of T_0 to select the reduction value. The results from the two multipliers are then added and a division by R follows. The algorithm starts by computing the product AB (T). It then uses $\bar{T}_0$, the two's complement of T_0 to compute the reduction value. In fact, the algorithm uses a function f given by:

$$\bar{T}_0 = R - T_0 = \langle f(T_0, M) \rangle_R \tag{6}$$

which is the two's complement of T_0.

It is easy to demonstrate that this novel algorithm implements Montgomery's multiplication. Let M'' and R' be defined by:

$$MM'' = 1 + R'R \tag{7}$$

By comparing (2) and (3), the following holds:

$$M'' = \langle -M' \rangle_R \tag{8a}$$

and

$$R' = \langle -R^{-1} \rangle_M \tag{8b}$$

The function f can then be written as:

$$f(T_0, M) = -T_0 MM'' = -T_0(1 + R'R) \tag{9}$$

Therefore, the result of (2) is obtained by adding the results of the multiplication operation and the reduction operation followed by a division by R. The final reduction operation is carried out to ensure that the final result is in the range $[0, M[$. A parallel implementation of the proposed modified algorithm (Algorithm 5) is depicted in Fig. 3. The first multiplier uses a conventional multiplier while the least significant (LS) cells of the second multiplier are built around a half adder (HA) with some additional gates. The LSB cell at the ith row of the second multiplier receives the bit $\bar{T}_{0,i}$ and computes the bit q_i, which is broadcast to the remaining cells of the row. The LSB cell also computes a carry bit which is fed to the row below. However, the overall delay of the cell does not exceed the delay of a gFA. In addition to the two multipliers, the parallel structure uses a two's complement circuit and an adder to compute the final result. A bit serial parallel implementation of algorithm 5 is shown in Fig. 4. The algorithm is implemented using two bit-serial multipliers, a serial adder for use to perform the addition of the results from the multiplication and the reduction operations, and a serial two's complement circuit that is used to two's complement the product AB.

The serial input a_i is fed to the first multiplier during the n first cycles. Then, n zeros are fed during the remaining n cycles of the multiplication process, during which the result and carry bits are propagated to generate the result of the multiplication. The second multiplier operates in almost a similar fashion, except that the LSB cell is different from the remaining cells. During the n first cycles, the LSB cell generates the bit q_i to select the reduction value and a carry bit that is fed back to the same cell while the result bit is equal to $\bar{T}_{0,i}$. During the remaining n cycles, the LSB cell operates as a HA and the bits q_i are set low. To achieve this, a control signal has been added to the LSB cell together with a multiplexer and two AND gates. The control signal is set

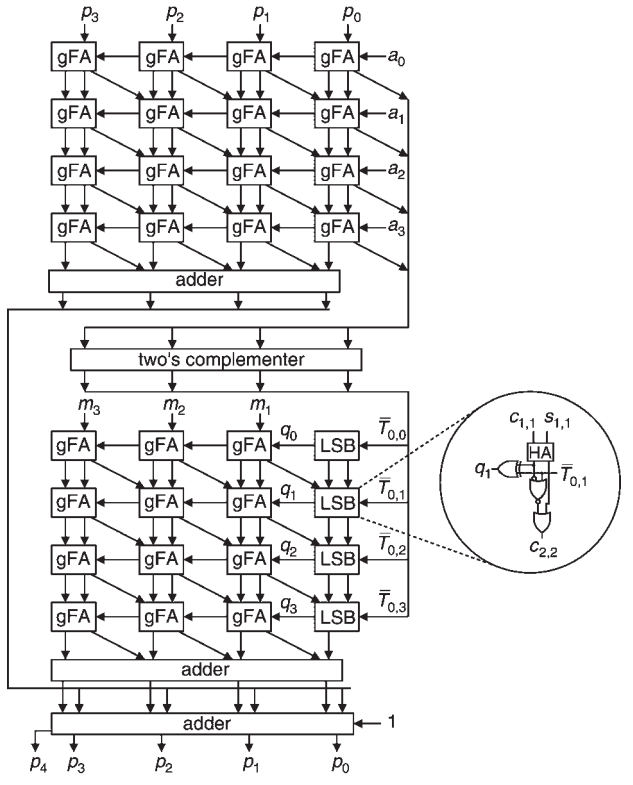

Fig. 3 *The parallel Montgomery's multiplier of algorithm 5*

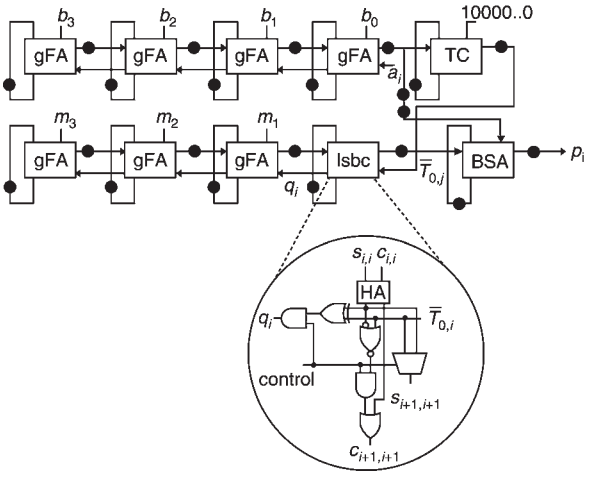

Fig. 4 *A bit-serial implementation of proposed algorithm 5*

high during the first n cycles and then low during the remaining n cycles so that the bit q_i is set low and the carry and result bits from the HA are selected as carry and result bit of the LSB cell, respectively. However, the delay within this cell does not exceed the delay within a gFA. The result from the first multiplier is generated two cycles before the result of the second multiplier, thus three latches are added to the signal from the LSB FA of the first serial multiplier to the bit serial adder to cope with the different latencies of the two bit serial multipliers.

5 Bit-level architectures for modular multiplication

The clock path of the bit serial multiplier of Fig. 4 is equivalent to that of a gFA and a latch. However, the data lines are broadcast to all the cells and as such a global distribution lowers the clock frequency. In this Section, systolic architectures that implement efficiently the

proposed technique of algorithm 5 are proposed. A bit-serial architecture is proposed in this section. This structure is fully systolic and the global distribution lines are avoided. The design of this architecture has been carried out by interleaving two modular multiplication operations onto the same structure and by pipelining the feedback loops strategy, as used in many works [8, 14]. The number of operations interleaved onto the same structure depends on the number of latches added to the feedback loops.

New digit structures based on algorithm 5 are also proposed in this Section. Usually, the concept of digit arithmetic has been proposed as a compromise between the bit serial and the bit parallel arithmetic to avoid the inconvenient too slow bit serial multipliers and the too large parallel multipliers. Since the digit size is variable, the digit structures provide the designer with more flexibility in finding the best trade-off between hardware cost and sample rate. However, based on the use of the feedback loop pipelining technique, the aim of the work is to design digit structures that are pipelined at the bit level with the minimum number of operations interleaved onto the same structure.

The bit level digit structures proposed in the literature [8] are an extension of the work presented in [14] by extensively pipelining the parallel multiplier and then using the folding transformation in order to obtain the digit structures. While this has been carried out for the interleaved algorithm, the digit structures proposed in this Section are based on the non-interleaved algorithm. This has the merit of reducing the number of interleaved modular multiplication operations required on the same bit level pipelined structure.

First let us recall the basic of the feedback pipelining technique as it was shown in [8]. Fig. 5 shows the dependence graph (DG) of a conventional multiplier. The CS multiplication scheme has been adopted in this Figure. However, the CP multiplication scheme or any other valid multiplication scheme can also be used. A bit-serial architecture is derived by projecting the DG onto the direction [0,1]. The scheduling vector $S^T[s_i, s_j]$ is chosen in such a way that the bit-serial multiplier is fully systolic; i.e., no global line distribution and a bit level of pipelining has to be achieved. From Fig. 5, s_j latches are added to every edge from a node at (i, j) to a node $(i, j+1)$. Also s_i latches are added to every edge from a node at (i, j) to a node $(i+1, j)$. To edges from node (i, j) to node $(i+1, j-1)$, $s_i - s_j$ latches are added. To derive a systolic multiplier form the DG of Fig. 5, the following conditions must be satisfied:

$$s_i > 0 \tag{10a}$$

$$s_j > 0 \tag{10b}$$

$$s_i - s_j > 0 \tag{10c}$$

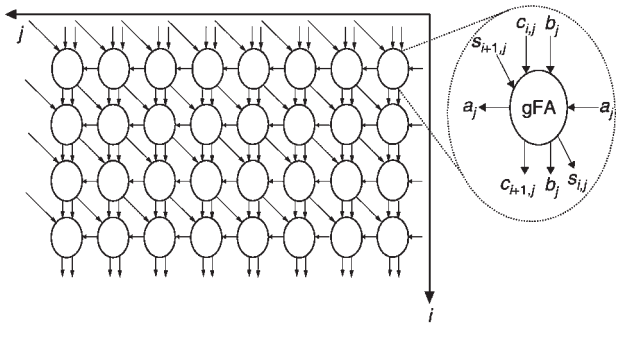

Fig. 5 *The DG of a CS multiplier*

A sufficient condition is to have:

$$s_i = 2 \tag{11a}$$

and

$$s_j = 1 \tag{11b}$$

In this way for every edge from a node at (i, j) to a node $(i, j+1)$ one latch is added while two latches are added to every edge from a node (i, j) to a node $(i+1, j)$. Finally, one latch is added to edges from node (i, j) to node $(i+1, j-1)$.

Taking this scheduling vector into account, the bit-serial structure which implements algorithm 5 is shown in Fig. 6. The same basic cells used in the multiplier of Fig. 4 are used to build this systolic multiplier structure, except that the overall number of latches has increased. It is worth noting that the multiplier only uses nearest neighbour communications and that there are two latches in the feedback loops. In fact the multiplier interleaves two modular multiplication operations into the same structure. It is also worth noting that there are two latches in the loop of the serial adder to cope with feeding two different pairs of operands to the multiplier.

It was suggested by [14] that the two interleaved modular multiplication operations are the two operations involved in the modular exponentiation. In this way, the serial multiplier can be used as a bit serial-parallel modular exponentiation structure.

Many scheduling vectors can also be chosen. Among these, the scheduling vector $[J, 2J]$ can be considered as the general case of the scheduling vectors.Therefore, $2J$ latches are required in the carry feedback loop, while J latches are needed in the other connections and $2J$ operations have to be interleaved. This scheduling vector can also be applied to the parallel multiplier. In [14], the scheduling vector [1, 2] was applied to the parallel multiplier before folding it to get digit modular multiplier structures pipelined at the bit level. For this purpose, the interleaved algorithm was used and the basic cell and the DG of the modular multiplication are shown in Fig. 7.

The unfolding transformation can also be used to design these structures. However, if the scheduling vector $[J, 2J]$ and the unfolding factor J are selected, the transformation leads to J separate serial multipliers. In fact, by taking into account that for $i<J, \lfloor (i+J)/J \rfloor = 1$ and $(i+J)/J = i$, and applying the transformation on a serial multiplier is equivalent to:

- for each node U in the original flow graph (FG), draw the J nodes $U_0 U_1 U_2 \ldots U_{J-1}$;
- for each edge $U \rightarrow V$ with J latches in the original FG, draw the J edges $U_i \rightarrow V_i$ with one latch.

Therefore, this requires J separate bit serial structures for the J operations interleaved onto the original structure. However, digit structures obtained by unfolding the bit serial multiplier and by linear projection (the scheduling vector [1,0] is adopted) can be pipelined by adding latches to the loops of the structure.

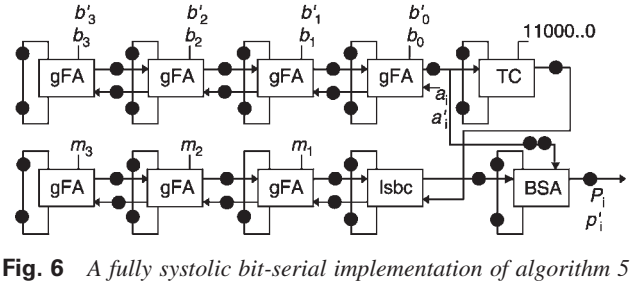

Fig. 6 *A fully systolic bit-serial implementation of algorithm 5*

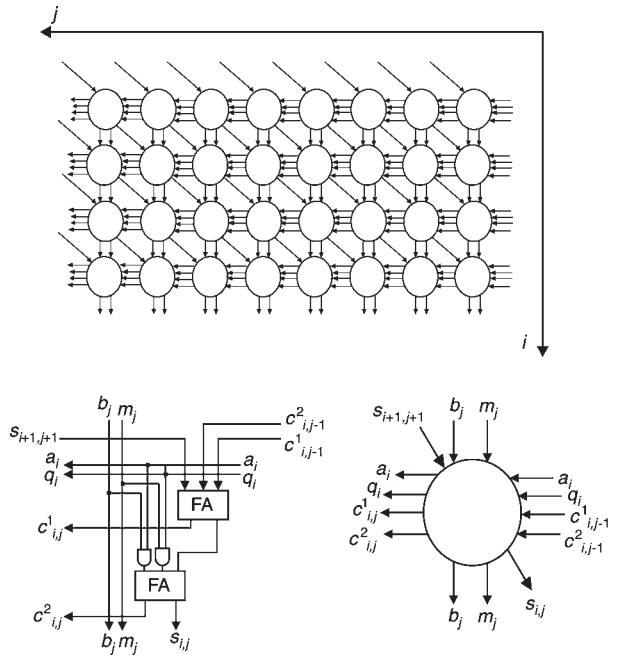

Fig. 7 *The DG of the multiplier structure in [8]*

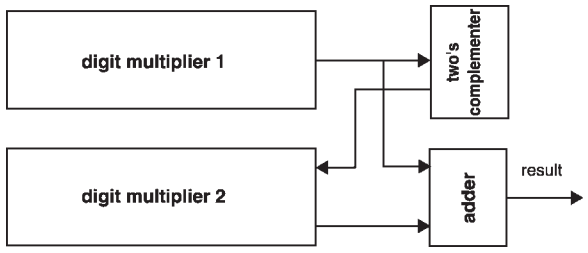

Fig. 8 *The proposed digit structure*

The Montgomery's digit modular multiplier is shown in Fig. 8. It uses two digit multipliers, a digit adder, and a digit complement circuit. The structure of the digit adder is similar to those described in [15], while LSB cells of the second digit are similar to that of the bit serial multiplier of Fig. 2.

The starting point of the design is based on the digit multiplier of Fig. 9 (step 1) or a digit size of four (or an unfolding factor J of four). The second step is to add $2J-1$ latches (or seven latches) to the loops of the multiplier. Note that the number of latches in the upward connections is equal to $2J$ (or eight latches). In the third and fourth steps of Fig. 9, the retiming of the structure is carried out. In step 3,

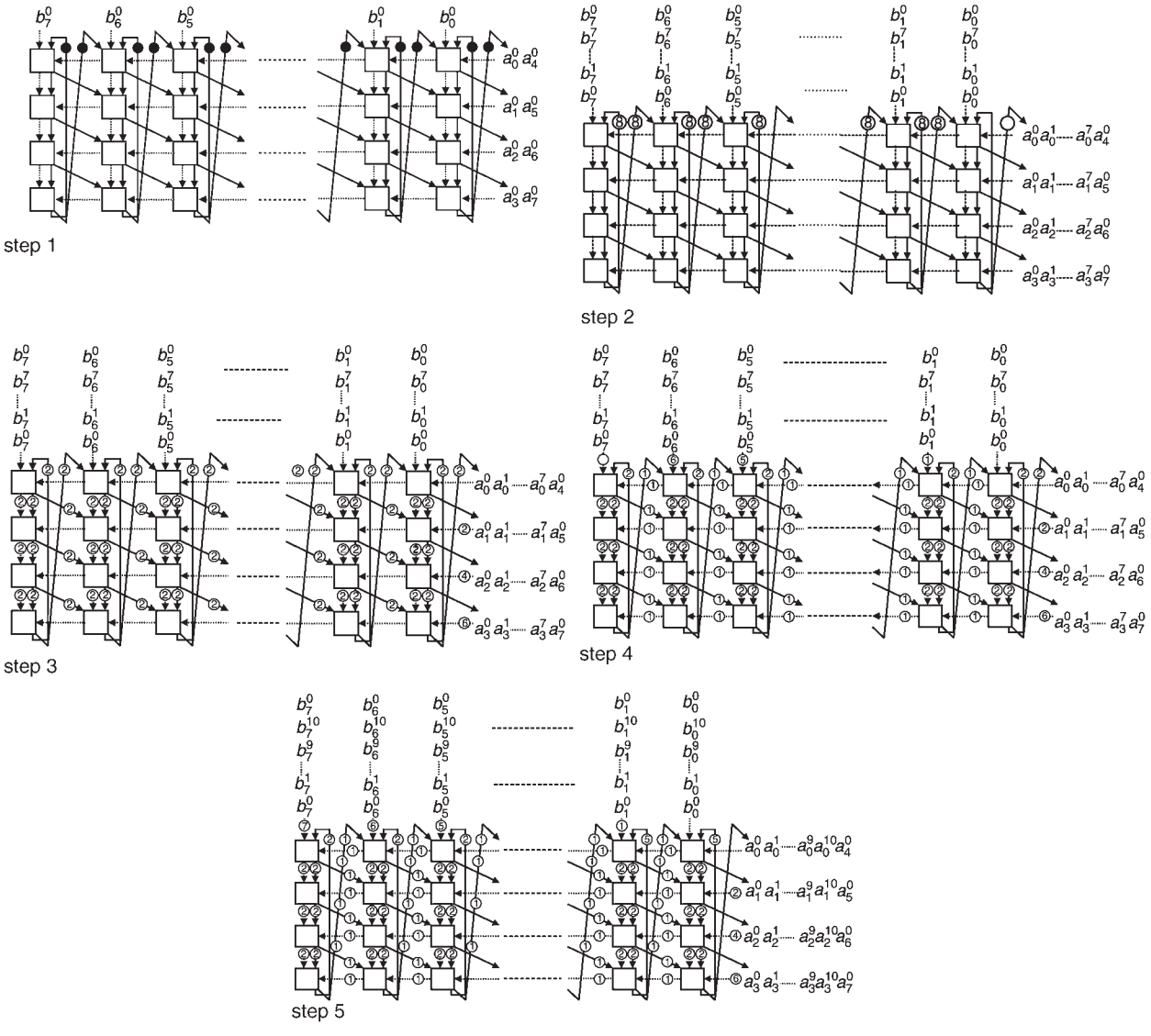

Fig. 9 *Internal architecture of the digit multipliers used to implement algorithm 5*

Table 1: Task assignments and scheduling in the multiplier cells

Cell 8	Cell 7	Cell 6	Cell 5	Cell 4	Cell 3	Cell 2	Cell 1		
$a_0^0 b_7^0$	$a_0^0 b_6^0$	$a_0^0 b_5^0$	$a_0^0 b_4^0$	$a_0^0 b_3^0$	$a_0^0 b_2^0$	$a_0^0 b_1^0$	$a_0^0 b_0^0$	row 1	Cycle 1
								row 2	
								row 3	
								row 4	
$a_0^1 b_7^1$	$a_0^1 b_6^1$	$a_0^1 b_5^1$	$a_0^1 b_4^1$	$a_0^1 b_3^1$	$a_0^1 b_2^1$	$a_0^1 b_1^1$	$a_0^1 b_0^1$	row 1	Cycle 2
								row 2	
								row 3	
								row 4	
$a_0^2 b_7^2$	$a_0^2 b_6^2$	$a_0^2 b_5^2$	$a_0^2 b_4^2$	$a_0^2 b_3^2$	$a_0^2 b_2^2$	$a_0^2 b_1^2$	$a_0^2 b_0^2$	row 1	Cycle 3
$a_1^0 b_7^0$	$a_1^0 b_6^0$	$a_1^0 b_5^0$	$a_1^0 b_4^0$	$a_1^0 b_3^0$	$a_1^0 b_2^0$	$a_1^0 b_1^0$	$a_1^0 b_0^0$	row 2	
								row 3	
								row 4	
$a_0^3 b_7^3$	$a_0^3 b_6^3$	$a_0^3 b_5^3$	$a_0^3 b_4^3$	$a_0^3 b_3^3$	$a_0^3 b_2^3$	$a_0^3 b_1^3$	$a_0^3 b_0^3$	row 1	Cycle 4
$a_1^1 b_7^1$	$a_1^1 b_6^1$	$a_1^1 b_5^1$	$a_1^1 b_4^1$	$a_1^1 b_3^1$	$a_1^1 b_2^1$	$a_1^1 b_1^1$	$a_1^1 b_0^1$	row 2	
								row 3	
								row 4	
$a_0^4 b_7^4$	$a_0^4 b_6^4$	$a_0^4 b_5^4$	$a_0^4 b_4^4$	$a_0^4 b_3^4$	$a_0^4 b_2^4$	$a_0^4 b_1^4$	$a_0^4 b_0^4$	row 1	Cycle 5
$a_1^2 b_7^2$	$a_1^2 b_6^2$	$a_1^2 b_5^2$	$a_1^2 b_4^2$	$a_1^2 b_3^2$	$a_1^2 b_2^2$	$a_1^2 b_1^2$	$a_1^2 b_0^2$	row 2	
$a_2^0 b_7^0$	$a_2^0 b_6^0$	$a_2^0 b_5^0$	$a_2^0 b_4^0$	$a_2^0 b_3^0$	$a_2^0 b_2^0$	$a_2^0 b_1^0$	$a_2^0 b_0^0$	row 3	
								row 4	
$a_0^5 b_7^5$	$a_0^5 b_6^5$	$a_0^5 b_5^5$	$a_0^5 b_4^5$	$a_0^5 b_3^5$	$a_0^5 b_2^5$	$a_0^5 b_1^5$	$a_0^5 b_0^5$	row 1	Cycle 6
$a_1^3 b_7^3$	$a_1^3 b_6^3$	$a_1^3 b_5^3$	$a_1^3 b_4^3$	$a_1^3 b_3^3$	$a_1^3 b_2^3$	$a_1^3 b_1^3$	$a_1^3 b_0^3$	row 2	
$a_2^1 b_7^1$	$a_2^1 b_6^1$	$a_2^1 b_5^1$	$a_2^1 b_4^1$	$a_2^1 b_3^1$	$a_2^1 b_2^1$	$a_2^1 b_1^1$	$a_2^1 b_0^1$	row 3	
								row 4	
$a_0^6 b_7^6$	$a_0^6 b_6^6$	$a_0^6 b_5^6$	$a_0^6 b_4^6$	$a_0^6 b_3^6$	$a_0^6 b_2^6$	$a_0^6 b_1^6$	$a_0^6 b_0^6$	row 1	Cycle 7
$a_1^4 b_7^4$	$a_1^4 b_6^4$	$a_1^4 b_5^4$	$a_1^4 b_4^4$	$a_1^4 b_3^4$	$a_1^4 b_2^4$	$a_1^4 b_1^4$	$a_1^4 b_0^4$	row 2	
$a_2^2 b_7^2$	$a_2^2 b_6^2$	$a_2^2 b_5^2$	$a_2^2 b_4^2$	$a_2^2 b_3^2$	$a_2^2 b_2^2$	$a_2^2 b_1^2$	$a_2^2 b_0^2$	row 3	
$a_3^0 b_7^0$	$a_3^0 b_6^0$	$a_3^0 b_5^0$	$a_3^0 b_4^0$	$a_3^0 b_3^0$	$a_3^0 b_2^0$	$a_3^0 b_1^0$	$a_3^0 b_0^0$	row 4	
$a_0^7 b_7^7$	$a_0^7 b_6^7$	$a_0^7 b_5^7$	$a_0^7 b_4^7$	$a_0^7 b_3^7$	$a_0^7 b_2^7$	$a_0^7 b_1^7$	$a_0^7 b_0^7$	row 1	Cycle 8
$a_1^5 b_7^5$	$a_1^5 b_6^5$	$a_1^5 b_5^5$	$a_1^5 b_4^5$	$a_1^5 b_3^5$	$a_1^5 b_2^5$	$a_1^5 b_1^5$	$a_1^5 b_0^5$	row 2	
$a_2^3 b_7^3$	$a_2^3 b_6^3$	$a_2^3 b_5^3$	$a_2^3 b_4^3$	$a_2^3 b_3^3$	$a_2^3 b_2^3$	$a_2^3 b_1^3$	$a_2^3 b_0^3$	row 3	
$a_3^1 b_7^1$	$a_3^1 b_6^1$	$a_3^1 b_5^1$	$a_3^1 b_4^1$	$a_3^1 b_3^1$	$a_3^1 b_2^1$	$a_3^1 b_1^1$	$a_3^1 b_0^1$	row 4	
$a_4^0 b_7^0$	$a_4^0 b_6^0$	$a_4^0 b_5^0$	$a_4^0 b_4^0$	$a_4^0 b_3^0$	$a_4^0 b_2^0$	$a_4^0 b_1^0$	$a_4^0 b_0^0$	row 1	Cycle 9
$a_1^6 b_7^6$	$a_1^6 b_6^6$	$a_1^6 b_5^6$	$a_1^6 b_4^6$	$a_1^6 b_3^6$	$a_1^6 b_2^6$	$a_1^6 b_1^6$	$a_1^6 b_0^6$	row 2	
$a_2^4 b_7^4$	$a_2^4 b_6^4$	$a_2^4 b_5^4$	$a_2^4 b_4^4$	$a_2^4 b_3^4$	$a_2^4 b_2^4$	$a_2^4 b_1^4$	$a_2^4 b_0^4$	row 3	
$a_3^2 b_7^2$	$a_3^2 b_6^2$	$a_3^2 b_5^2$	$a_3^2 b_4^2$	$a_3^2 b_3^2$	$a_3^2 b_2^2$	$a_3^2 b_1^2$	$a_3^2 b_0^2$	row 4	

the retiming lines are horizontal while in the fourth step the retiming lines are vertical. The tasks assigned to the cells of the multiplier after the fourth step are shown in Table 1. The notation x_j^i is the jth bit of the ith set of data x.

However, the level of pipelining achieved is such that there are latches at every output of the cells, the connections from the last row of the digit multiplier to the top row are not local connections. To circumvent this problem, latches were added to the feedback loops. For an unfolding factor of J, $J-1$ latches (three latches in Fig. 9) are added to each loop so that the upward lines are no longer broadcast to the top cells, as it is shown in the fifth step and as such, the resulting structure has its connections fully localised. The behaviour of the digit structure changes from one step to another. In step 1, the digit multiplier carries out a single operation. In steps 2,3 and 4 the multiplier interleaves $2J$ (eight in Fig. 9) multiplication operations while in step 5, $3J-1$ (11 in Fig. 9) operations are time multiplexed onto the digit structure.

6 Results and performances

A comparison between the two modular multipliers that use the structures of Fig. 4 is shown in Table 2. The comparison is made in terms of the hardware usage for the same wordlength and digit size for the digit structures derived from algorithm 5 and the structures proposed in [8], which are based on the implementation of the interleaved Montgomery's modular multiplier. The path in Table 2 is the length of the wires between two neighbour cells. The number of operations is also taken into account in the comparison.

Multiplier 1 is built using the digit multiplier of Fig. 9 (step 4) and uses JAFAs and $8J$ latches per bit of operand for a digit size of J. The longest path of the multiplier is $J-1$ and it interleaves $2J$ modular multiplication operations. Multiplier 2 uses the digit multiplier of Fig. 9 (step 5). It employs J FAs and $12J$ latches per bit of operand. However, the longest path is one. Thus, the multiplier is

Table 2: Comparison of performances

	Area usage	Longest path	Number of operations
Multiplier 1	J(2 FAs + 8 latches)	$J-1$	$2J$
Multiplier 2	J(2 FAs + 12 latches)	1	$3J-2$
Multiplier 3 [93]	$3J^2+5J$ latches + $2J$ FAs	1	$J(J-1)$
Multiplier 4 [93]	$19J$ latches + $2J$ FAs	$J-2$	$5J$

strictly systolic. It interleaves $3J-2$ multiplication operations. To achieve the same speed as in the structure proposed in [8], $3J^2+5J$ latches per bit of operand are required to interleave $J(J-1)$ modular multiplication operations (see multiplier 3 in Table 2). If some relaxation in the systolicity of the multiplier is tolerated so that there is a latch at every output of the FAs within the structure, $19J$ latches are required per bit of operand to interleave $5J$ operations (see multiplier 4 in Table 2). This clearly underlines that the proposed structure can be pipelined at the bit level with a minimum number of added latches while the number of interleaved multiplication operation can be increased by extensively adding latches to the feedback loop.

7 Conclusions

An algorithm and a set of architectures to implement Montgomery's multiplication have been presented. The main feature of this algorithm is that the modular computation is broken into two multiplication operations. Fully systolic architectures derived from the proposed algorithm have been proposed. The longest path of these architectures is one FA. The implementation of these algorithms yields to scalable architectures that are very well suited for VLSI implementations. The proposed structures can also be pipelined at the bit level by interleaving fewer operations than previously proposed in the literature while using less FFs or latches. An advantage offered by the feedback pipelining technique is that the number of latches added to the unfolded multiplier can vary from zero, for multiplier 1, to $J-2$ for multiplier 2. Therefore, the number of latches added to the multiplier is speed- dependent, which is left to the designer to fix. To every latch added to the unfolded multiplier, a multiplication operation is added to those already performed by the multiplier. In RSA cryptography the data is broken into multiple blocks and each block is encrypted separately, therefore, the proposed structures can be used to interleave such encryption operations.

8 References

1 Shnad, M., and Vuillemin, J.: 'Fast implementation of RSA cryptography'. Proc. 11th IEEE Symp. on Computer arithmetic, Windsor, ONT, July 1993, pp. 252–259
2 Kornerup, P.: 'A systolic, linear-array multiplier for a class of right-shift algorithms', *IEEE Trans. Comput.*, 1994, **43**, (8), pp. 892–898
3 Guo, J., and Wang, C.: 'A novel digit-serial systolic array for modular multiplication'. Proc. Int. Symp. on Circuits and systems (ISCAS), Monterey, CA, 31 May–3 June 1998, pp. 177–180
4 Walter, C.D.: 'Systolic Modular Multiplication', *IEEE Trans. Comput.*, 1993, **42**, (3), pp. 376–378
5 Tsai, W.T., Shung, C.B., and Wang, S.: 'Two Systolic Architectures for Modular Multiplication', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2000, **8**, (1), pp. 103–107
6 Tenca, A.F., and Koc, C.K.: 'A scalable architecture for Montgomery multiplication. Cryptographic Hardware and Embedded Systems', *Lect. Notes Comput. Sci. No. 1717*, 1999, pp. 94–108
7 Freking, W.L., and Parhi, K.K.: 'Performance-scalable array architectures for modular multiplication'. Proc. IEEE Int. Conf. on Application-specific systems, architectures, and processors (ASAP), Boston, MA, 10–12 July 2000, pp. 149–162
8 Freking, W.L., and Parhi, K.K.: 'Ring-Planarized Cylindrical Arrays with Application to Modular Multiplication'. IEEE Workshop on Signal processing systems design & implementation (SIPS), Lafayette, LA, USA, 11–13 October 2000, pp. 497–506
9 Nibouche, O., Bouridane, A., and Nibouche, M.: 'New Iterative Algorithms and Architectures of Modular Multiplication for Cryptography'. Proc. 8th Int. IEEE Conf. on Electronics, circuits, and systems, (ICECS), Malta, 2–5 September 2001
10 Montgomery, P.L.: 'Modular multiplication without trial division', *Math. Comput.*, 1985, **44**, pp. 519–521
11 Nibouche, O., Bouridane, A., Nibouche, M., and Crookes, D.: 'A New Pipelined Digit Serial-Parallel Multiplier'. Proc. IEEE Int. Symp. on Circuits and systems (ISCAS), Geneva, 28–31 May 2000, pp. 12–15
12 Nibouche, O.: 'High Performance Computer Arithmetic Architectures for Image and Signal Processing Applications'. PhD Thesis, School of Computer Science, Queen's University Belfast
13 Rivest, R.L., Shamir, A., and Adelman, L.: 'A method of obtaining digital signatures and public-key cryptosystems', *Commun. ACM*, 1978, **21**, pp. 120–126
14 Chiou, C.D., and Yang, T.C.: 'Iterative modular multiplication algorithm without magnitude comparison', *Electron. Lett.*, 1994, **30**, (30), pp. 2017–2018
15 Nibouche O., and Nibouche, M.: 'On Designing Digit Multipliers'. Proc. 9th Int. IEEE Conf. on Electronics, circuits, and systems (ICECS), Dubrovnik, 15–18 September 2002, pp. 951–954