A Hardware Algorithm for Modular Multiplication/Division

Marcelo E. Kaihara and Naofumi Takagi, *Senior Member*, *IEEE*

Abstract—A mixed radix-4/2 algorithm for modular multiplication/division suitable for VLSI implementation is proposed. The algorithm is based on Montgomery method for modular multiplication and on the extended Binary GCD algorithm for modular division. Both algorithms are modified and combined into the proposed algorithm so that almost all the hardware components are shared. The new algorithm carries out both calculations using simple operations such as shifts, additions, and subtractions. The radix-2 signed-digit representation is used to avoid carry propagation in all additions and subtractions. A modular multiplier/divider based on the algorithm performs an n-bit modular multiplication/division in $O(n)$ clock cycles where the length of the clock cycle is constant and independent of n. The modular multiplier/divider has a linear array structure with a bit-slice feature and can be implemented with much smaller hardware than that necessary to implement both multiplier and divider separately.

Index Terms—Computer arithmetic, hardware algorithm, modular multiplication, modular division, redundant representation, cryptography.

1 Introduction

The increasing importance of security in computers and communications systems introduced the need for managing several public-key cryptosystems in PCs and mobile devices such as PDAs. Processing these cryptosystems requires a huge amount of computation and there is, therefore, a great demand for developing dedicated hardware to speed up the computations. Modular multiplication and modular division with large modulus are the basic operations in processing many public-key cryptosystems. For example, they are used in the deciphering operation of RSA [21] and ElGamal [6] cryptosystems, in the Diffie-Hellman key exchange protocol [5], and in the DSA digital signature scheme [1]. They can also be used to accelerate the exponentiation operation using addition-subtraction chains [10] and to compute point operations in ECC with curves defined over $GF(p)$ [11].

In this paper, we are investigating modular multiplication/division hardware algorithms for a large modulus suitable to be implemented in compact hardware. Much effort has been devoted to developing specialized hardware for computing fast modular multiplication and modular inversion separately. Many algorithms have been proposed in the literature for computing modular multiplication. Most of them use redundant number systems and perform a high-radix modular multiplication [17], [15], [24], [14], [25] or use Residue Number System (RNS) [20], [2], [9], [7]. For modular inversion, we can cite the works of [19] and [3]. For modular division, however, there are only a few algorithms and these are based on the Euclidean algorithm for computing GCD [22]. None of these works has concentrated on reducing the hardware requirements of modular multiplier and divider by combining them into the same architecture. Enabling the reduction of hardware is important because it allows for the miniaturization of portable devices and reduces fabrication costs.

In this paper, we propose a mixed radix-4/2 algorithm for modular multiplication/division for a large modulus suitable for VLSI implementation. The calculation of modular multiplication is based on the Montgomery multiplication algorithm [16] and the modular division on the extended Binary GCD algorithm [22] because both of these algorithms have similar structures and use similar operations to perform the calculations. We exploit these similarities to modify the algorithms in order to share almost all hardware components for both operations. Other combinations were attempted, but all were found to have inherent differences making them unsuitable for combining.

We have accelerated Montgomery multiplication algorithm as a mixed radix-4/2 algorithm, which processes, when possible, the multiplier in radix-4 per iteration. We have also accelerated the extended Binary GCD algorithm as a mixed radix-4/2 algorithm by transforming a two-step operation into a one-step operation. Thus, when possible, the operands are processed by two digits at each iteration. Redundant representation is used in all additions and subtractions so that they may be carried out without carry propagation.

A modular multiplier/divider based on the proposed algorithm has a linear array structure with a bit-slice feature and is suitable for VLSI implementation. The amount of hardware of an n-bit modular multiplier/divider is proportional to n. It performs an n-bit modular multiplication in a maximum of $\lfloor \frac{2}{3}(n+2) \rfloor + 3$ clock cycles and an n-bit modular division in no more than $2n+5$ clock cycles, where the length of clock cycle is constant, independent of n.

This paper is an extension of [8]. The algorithm for modular multiplication/division has been improved and the reduction of hardware has been determined through

• *The authors are with the Department of Information Engineering, Nagoya University, Nagoya, 464-8603, Japan.*
E-mail: mkaihara@takagi.nuie.nagoya-u.ac.jp, ntakagi@is.nagoya-u.ac.jp.

Manuscript received 24 Sept. 2003; revised 18 May 2004; accepted 5 Aug. 2004; published online 16 Nov. 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0162-0903.

simulation. For the latter, we have designed a modular multiplier based on the Montgomery algorithm with the modification we introduced to accelerate it, a modular divider based on the extended Binary GCD algorithm also with the modification we introduced, and a modular multiplier/divider based on the proposed algorithm. The estimated total circuit area of the modular multiplier/divider resulted much smaller than the total sum of circuit areas when the multiplier and the divider are implemented separately, with critical path delays remaining practically to the same value.

In the next section, we will explain the extended Binary GCD algorithm, the Montgomery multiplication algorithm, and the redundant number representation system we use. Section 3 proposes a hardware algorithm for modular multiplication/division. Section 4 explains the hardware implementation and design. Section 5 considers possible applications to modular exponentiation and cryptography. Section 6 contains our concluding remarks.

2 Preliminaries

2.1 Extended Binary GCD Algorithm for Modular Division

The extended Binary GCD algorithm [10] is an efficient way of calculating modular division. Consider the residue class field of integers with an odd prime modulus M. Let X and $Y(\neq 0)$ be elements of the field. The algorithm calculates $Z(< M)$ where $Z \equiv X/Y \pmod{M}$ (the algorithm also works with M not prime and Y relatively prime to M). It performs modular division by intertwining the procedure for finding the modular quotient with that for calculating $\gcd(Y, M)$.

The algorithm requires four variables, A, B, U, and V. A and B are used for the calculation of $\gcd(Y, M)$ and variables U and V for the calculation of modular quotient. Variables A and B are initialized to Y and M, respectively, and the following properties are applied iteratively to calculate $\gcd(Y, M)$: If A is even and B is odd, then $\gcd(A, B) = \gcd(A/2, B)$; if A and B are both odd, then either $A + B$ or $A - B$ is divisible by 4; in this case, if $A + B$ is divisible by 4, then $\gcd(A, B) = \gcd((A + B)/4, B)$ and $|(A + B)/4| \leq \max(|A/2|, |B/2|)$; otherwise, $A - B$ is divisible by 4, $\gcd(A, B) = \gcd((A - B)/4, B)$, and $|(A - B)/4| \leq \max(|A/2|, |B/2|)$. In order to determine the modular quotient, U and V are initialized to the values of X and 0, respectively; then, the same operations that are performed to A and B are applied to U and V in modulo M.

We show the algorithm below. Note that A and B are integers and are allowed to be negative. δ represents $\alpha - \beta$, where α and β are values such that 2^α and 2^β indicate the upper bounds of $|A|$ and $|B|$, respectively. ρ is introduced to represent $min(\alpha, \beta)$ and the condition $\rho = 0$ assures that $A = 0$.

Algorithm 1 (Algorithm for Modular Division)
Inputs: M: $2^{n-1} < M < 2^n$, $\gcd(M, 2) = 1$, and prime
X, Y: $0 \leq X < M$, $0 < Y < M$
Output: $Z = X/Y \bmod M$
Algorithm:
$A := Y$; $B := M$; $U := X$; $V := 0$; $\rho := n$; $\delta := 0$;
while $\rho \neq 0$ **do**
while $A \bmod 2 = 0$ **do**
$A := A/2$; $U := U/2 \bmod M$;
$\rho := \rho - 1$; $\delta := \delta - 1$;
endwhile
if $\delta < 0$ **then**
$T := A$; $A := B$; $B := T$;
$T := U$; $U := V$; $V := T$;
$\delta := -\delta$;
endif
if $(A + B) \bmod 4 = 0$ **then** $q := 1$ **else** $q := -1$;
$A := (A + qB)/4$; $U := (U + qV)/4 \bmod M$;
$\rho := \rho - 1$; $\delta := \delta - 1$;
endwhile
if $B = 1$ **then** $Z := V$ **else** /* $B = -1$*/ $Z := M - V$;

It can easily be shown that the equivalences $V \times Y \equiv B \times X \pmod{M}$ and $U \times Y \equiv A \times X \pmod{M}$ always hold. Since $\gcd(Y, M) = 1$, when $A = 0$, B is 1 or -1. Hence, in the final step of the algorithm, $Z \times Y \equiv X \pmod{M}$ holds and Z is the quotient of X/Y modulo M.

2.2 Montgomery Modular Multiplication

Montgomery proposed an efficient algorithm for calculating modular multiplication [16]. Consider the residue class ring of integers with an odd modulus M. Let X and Y be elements of the ring. The Montgomery modular multiplication algorithm calculates $Z(< M)$ where $Z \equiv XYR^{-1} \pmod{M}$. R is an arbitrary constant relatively prime to M and it usually takes the value of 2^n when the calculations are performed in radix-2 with an n-bit modulus M.

Montgomery multiplication algorithm in radix-2 is described below. We use the same notation as in the extended Binary GCD algorithm to emphasize the similarities of these algorithms.

Algorithm 2 (Algorithm for Montgomery Modular Multiplication)
Inputs: $M : 2^{n-1} < M < 2^n$ and $\gcd(M, 2) = 1$
$X, Y : 0 \leq X, Y < M$
Output: $Z = XY2^{-n} \bmod M$
Algorithm:
$A := Y$; $U := 0$; $V := X$; $\rho = n$;
while $\rho \neq 0$ **do**
if $A \bmod 2 = 0$ **then** $q := 0$ **else** $q := 1$;
$A := (A - q)/2$; $U := (U + qV)/2 \bmod M$;
$\rho := \rho - 1$;
endwhile
if $U \geq M$ **then** $Z := U - M$ **else** $Z := U$;

The loop finishes after n iterations. Note that U is always bounded by $2M$ throughout the execution of the loop. Therefore, the last correction step assures that the output is correctly expressed in the range $[0, M - 1]$.

2.3 Use of a Redundant Representation

In order to perform additions and subtractions without carry propagation, we represent the internal variables A, B, U, and V as n-digit radix-2 signed digit (SD2) numbers.

The SD2 representation uses the digit set $\{\bar{1}, 0, 1\}$, where $\bar{1}$ denotes -1. An n-digit SD2 integer $A = [a_{n-1}, a_{n-2}, \cdots, a_0]$ $(a_i \in \{\bar{1}, 0, 1\})$ has the value $\sum_{i=0}^{n-1} a_i \cdot 2^i$. The addition of two

TABLE 1
The Rule for Adding Binary SD2 Numbers

a_ib_i	$a_{i-1}b_{i-1}$	c_i	h_i
00	–	0	0
01/10	neither is $\bar{1}$	1	$\bar{1}$
	at least one is $\bar{1}$	0	1
$0\bar{1}/\bar{1}0$	neither is $\bar{1}$	0	$\bar{1}$
	at least one is $\bar{1}$	$\bar{1}$	1
11	–	1	0
$\bar{1}\bar{1}$	–	$\bar{1}$	0
$1\bar{1}/\bar{1}1$	–	0	0

SD2 numbers can be performed without carry propagation. The addition of two SD2 numbers, A and B, is accomplished by first calculating the interim sum h_i and the carry digit c_i and then performing the final sum $s_i = h_i + c_{i-1}$ for each i without carry propagation. To calculate s_i, we need to check the digits a_i, b_i, and their preceding ones, a_{i-1}, b_{i-1}, a_{i-2}, and b_{i-2}. We use the addition rules for SD2 numbers shown in Table 1. All the digits of the result can be computed in parallel. The additive inverse of an SD2 number can be obtained by simply changing the signs of all nonzero digits in it. Subtraction can be achieved by finding the additive inverse and performing addition. We require a carry-propagate addition to convert an SD2 number to the ordinary nonredundant representation.

In applications such as exponentiation, chained multiplications are required. To remove time-consuming SD2 to binary conversion in each multiplication, the input operands X and Y as well as the output result Z are expressed with the same SD2 representation in the range $[-M+1, M-1]$. In this way, the output can be directly fed into the inputs. Note that, in the SD2 system, operands X and Y can still be given in ordinary binary representation.

3 A Hardware Algorithm for Modular Multiplication/Division

We propose a hardware algorithm that performs Montgomery modular multiplication and modular division, which is efficient in execution time and hardware requirements. We first present our accelerated modular division algorithm, then our accelerated Montgomery modular multiplication algorithm, and, finally, the combined mixed radix-4/2 modular multiplication/division algorithm.

3.1 Hardware Algorithm for Modular Division

A hardware algorithm based on the extended Binary GCD algorithm presented in the previous section was proposed in [22]. We modified and accelerated it. We explain first the implementation in [22] and then the modification we introduced.

The algorithm described in [22] performs all basic operations in constant time, independent of n, by a combinational circuit. Internal variables A, B, U, and V are represented as n-digit radix-2 SD2 numbers. The "while" loop of the algorithm is implemented by introducing variable P, which represents a binary number of $n+2$ bits and indicates the minimum of the upper bounds of $|A|$ and $|B|$, i.e., $\min(2^\alpha, 2^\beta)$. Note that P has only one bit of value 1 and the rest of them have the value of 0. In this way, the termination condition of $\rho = 0$, which requires an investigation of all the bits of ρ, is replaced by a check of $P = 1$, which can be carried out by testing the least significant bit of P, i.e., p_0. δ is implemented with a binary number D and a flag $s\,(\in \{0,1\})$. D has $n+2$ bits of length and has the value $D = 2^{(-1)^s \cdot \delta}$. The variable D also has only one bit of value 1 and the rest of them have the value of 0. In this way, the decrement $\delta := \delta - 1$, which may require a borrow propagation, is replaced by a one-bit shift of D. The value of $\delta = 0$ is represented with the values of $D = 1$ and $s = 1$.

In the case when A is divisible by 2, the algorithm performs $A := A/2$ with the operation $U := U/2 \bmod M$.

For the case that $(A+B) \equiv 0 \pmod 4$ (or $(A-B) \equiv 0 \pmod 4$), the calculations of $A := (A+B)/4$ and $U := (U+V)/4 \bmod M$ (or $A := (A-B)/4$ and $U := (U-V)/4 \bmod M$) are performed. When $s = 1$ and $D > 1$, i.e., $\delta < 0$, these calculations are combined with their next swap of A and B and that of U and V. The test condition $(A+B) \bmod 4 = 0$ is carried out by checking if $([a_1a_0] + [b_1b_0]) \bmod 4 = 0$, thus, only the least significant two digits of A and B need to be checked.

The calculation of $U/2$ modulo M is implemented by the operation $MHLV(U, M)$. It is carried out by performing $U/2$ or $(U+M)/2$ accordingly as U is even or odd. Note that only the least significant digit of U needs to be checked to determine whether it is even or odd.

The calculation of $W/4$ modulo M is implemented by the operation $MQRTR(W, M)$. It is carried out by performing the following calculations: If $M \equiv 1 \pmod 4$, it performs $W/4$ or $(W-M)/4$ or $(W+2M)/4$ or $(W+M)/4$, accordingly as $W \bmod 4$ is 0, 1, 2, or 3. If $M \equiv 3 \pmod 4$, it performs $W/4$ or $(W+M)/4$ or $(W+2M)/4$ or $(W-M)/4$, accordingly as $W \bmod 4$ is 0, 1, 2, or 3. Since M is an ordinary binary number, addition of M or $-M$ or $2M$ in $MQRTR$ and addition of M in $MHLV$ are simpler than the ordinary SD2 addition. For the details of the simpler SD2 addition, see, e.g., [23]. Since we are assuming that M is odd, only the second least significant bit of M, i.e., m_1, has to be checked to determine the value of $M \bmod 4$. Operation $(U+V)/4 \bmod M$ and $(U-V)/4 \bmod M$ are then implemented with $MQRTR(U+V, M)$ and $MQRTR(U-V, M)$, respectively. Note that only the least significant two digits of U and V have to be checked to determine the value of $(U+V) \bmod 4$ and $(U-V) \bmod 4$. All results of these basic operations are always in the range from $-M$ to M and no over-flow occurs.

In order to accelerate the calculation, we modify the algorithm and introduce a new testing condition. That is, in the case when A is divisible by 4, instead of performing $A/2$ and $U/2$ modulo M in two different steps, we group two of each operation into the calculations of $A/4$ and $U/4$ modulo M. For the latter, we use operation $MQRTR(U, M)$. Only the least significant two digits of U need to be checked to determine the value of $U \bmod 4$.

Now, we present the accelerated division algorithm. In the algorithm, $\{C1, C2\}$ means that two calculations, $C1$ and $C2$, are performed in parallel. $F >> l$ means a logical shift of F by l positions to the right. Analogously, $F << l$ means a logical shift of F by l positions to the left.

$M = [11111011]_2$ (251), $X = [\bar{1}0010\bar{1}01]_{SD}$ (−115), $Y = [1111111\bar{1}\bar{1}]_{SD}$ (249)

	A		B		P	D	s	U		V	
	$1111111\bar{1}\bar{1}$	(249)	11111011	(251)	1000000000	0000000001	1	$\bar{1}0010\bar{1}01$	(−115)	00000000	(0)
$(A+B)/4,B$	01111101	(125)	11111011	(251)	0100000000	0000000010	1	00100010	(34)	00000000	(0)
$(A+B)/4,A$	01011110	(94)	01111101	(125)	0100000000	0000000001	1	$10001\bar{1}10$	(134)	00100010	(34)
$A/2,B$	00101111	(47)	01111101	(125)	0010000000	0000000010	1	$01000\bar{1}\bar{1}1$	(67)	00100010	(34)
$(A+B)/4,A$	00101011	(43)	00101111	(47)	0010000000	0000000001	1	01011000	(88)	$01000\bar{1}\bar{1}1$	(67)
$(A-B)/4,B$	$00000\bar{1}\bar{1}1$	(−1)	00101111	(47)	0001000000	0000000010	1	01000100	(68)	$01000\bar{1}\bar{1}1$	(67)
$(A-B)/4,A$	$0000\bar{1}\bar{1}00$	(−12)	$00000\bar{1}\bar{1}1$	(−1)	0001000000	0000000001	1	$01000\bar{1}\bar{1}1$	(63)	01000100	(68)
$A/4,B$	$00000\bar{1}\bar{1}\bar{1}$	(−3)	$00000\bar{1}\bar{1}1$	(−1)	0000010000	0000000100	1	$0\bar{1}\bar{1}0\bar{1}\bar{1}\bar{1}$	(−47)	01000100	(68)
$(A+B)/4,A$	$0000000\bar{1}$	(−1)	$00000\bar{1}\bar{1}\bar{1}$	(−3)	0000010000	0000000010	0	01000100	(68)	$0\bar{1}\bar{1}\bar{1}01\bar{1}\bar{1}$	(−47)
$(A+B)/4,B$	$0000000\bar{1}$	(−1)	$00000\bar{1}\bar{1}\bar{1}$	(−3)	0000010000	0000000001	1	01000100	(68)	$0\bar{1}\bar{1}\bar{1}01\bar{1}\bar{1}$	(−47)
$(A+B)/4,B$	$0000000\bar{1}$	(−1)	$00000\bar{1}\bar{1}\bar{1}$	(−3)	0000001000	0000000010	1	01000100	(68)	$0\bar{1}\bar{1}\bar{1}01\bar{1}\bar{1}$	(−47)
$(A+B)/4,A$	$0000000\bar{1}$	(−1)	$0000000\bar{1}$	(−1)	0000001000	0000000001	1	01000100	(68)	01000100	(68)
$(A-B)/4,B$	00000000	(0)	$0000000\bar{1}$	(−1)	0000000100	0000000010	1	00000000	(0)	01000100	(68)
$A/4,B$	00000000	(0)	$0000000\bar{1}$	(−1)	0000000001	0000001000	1	00000000	(0)	01000100	(68)
$-V$										$0\bar{1}000\bar{1}00$	(−68)

$Z = [0\bar{1}000\bar{1}00]_{SD}$ (−68)

Fig. 1. A modular division by **Algorithm 3**.

Algorithm 3 (Hardware Algorithm for Modular Division)
Inputs: $M : 2^{n-1} < M < 2^n$, $\gcd(M, 2) = 1$ and prime
$X, Y : -M < X, Y < M$ $(Y \neq 0)$
Output: $Z = X/Y \bmod M$ $(-M < Z < M)$
Algorithm:

Step 1: $A := Y$; $B := M$; $U := X$; $V := 0$; $M := M$;
$P := 2^{n+1}$; $D := 1$; $s := 1$;

Step 2: **while** $p_0 \neq 1$ **do**
  if $[a_1a_0] = 0$ **then** /* $A \equiv 0 \pmod 4$ */
    $A := A >> 2$; $U := MQRTR(U, M)$;
    if $s = 0$ **then**
      if $d_2 = 1$ **then** $s := 1$;
      if $d_1 = 0$ **then** $D := D >> 2$;
      else $P := P >> 1$; $s := 1$; **endif**
    else /* $s = 1$ */
      $D := D << 2$;
      if $p_1 = 0$ **then** $P := P >> 2$
      else $P := P >> 1$;
    endif
  elseif $a_0 = 0$ **then** /* $A \equiv 2 \pmod 4$ */
    $A := A >> 1$; $U := MHLV(U, M)$;
    if $s = 0$ **then**
      if $d_1 = 1$ **then** $s := 1$;
      $D := D >> 1$;
    else /* $s = 1$ */ $D := D << 1$;
      $P := P >> 1$; **endif**
  else /* $A \equiv 1$ (mod 4) or $A \equiv 3$ (mod 4) */
    if $([a_1a_0] + [b_1b_0]) \bmod 4 = 0$ **then** $q := 1$
      else $q := -1$;
    if $s = 0$ **or** $d_0 = 1$ **then**
      $A := (A + qB) >> 2$;
      $U := MQRTR(U + qV, M)$;
      if $s = 1$ **then**
        $P := P >> 1$; $D := D << 1$;
      else /* $s = 0$ */
        if $d_1 = 1$ **then** $s := 1$;
        $D := D >> 1$;
      endif
    else /* $s = 1$ and $D > 1$ */
      $\{A := (A + qB) >> 2,\ B := A\}$;
      $\{U := MQRTR(U + qV, M),\ V := U\}$;
      if $d_1 = 0$ **then** $s := 0$;
      $D := D >> 1$;
    endif
  endif
endwhile

Step 3: **if** $([b_1b_0] = 3$ **or** $[b_1b_0] = -1)$ **then** $V := -V$;

Step 4: $Z := V$;

The core of the algorithm is described in Step 2. It is divided into three parts corresponding to the cases that $A \equiv 0 \pmod 4$, $A \equiv 2 \pmod 4$, and $A \equiv 1$ or $3 \pmod 4$, respectively.

For the case $A \equiv 0 \pmod 4$, $A := A >> 2$ and $U := MQRTR(U, M)$ are performed.

When $A \equiv 2 \pmod 4$, $A := A >> 1$ and $U := MHLV(U, M)$ are performed.

For the case $A \equiv 1$ or $3 \pmod 4$, and $s = 0$ or $d_0 = 1$, i.e., $\delta \geq 0$, $A := (A + B) >> 2$ and $U := MQRTR(U + V, M)$ or $A := (A - B) >> 2$ and $U := MQRTR(U - V, M)$ are performed. P is shifted by one position to the right, meaning that the upper bound between $|A|$ and $|B|$ is reduced by one digit. In the other case, when $A \equiv 1$ or $3 \pmod 4$, and $s = 1$ and $D > 1$, i.e., $\delta < 0$, swap between the values of A and B and, between U and V, are also performed at the same time.

If $P = 2$ and $a_0 = 0$, although only one-digit operation is required, the algorithm processes two digits, i.e., $A := A >> 2$ and $U := MQRTR(U, M)$, to make the control simple. These operations only update the values of A and U and do not affect the final result nor do they increase the number of iterations needed. No special consideration is required for the termination condition.

In Step 3, B ends with value 1 when, at initialization time, $B \equiv 1 \pmod 4$. Otherwise, it ends with value -1. This happens when $[b_1b_0] = 11$ or $[b_1b_0] = \bar{1}1$ or $[b_1b_0] = 0\bar{1}$ and V is negated in the SD2 system.

In Step 4, V is selected as the output.

Fig. 1 shows an example of a modular division, $-115/249 \bmod 251 = -68 \bmod 251 = 183$, where $n = 8$ by Algorithm 3. The leftmost column shows which calculations have been carried out. For example, "$(A - B)/4, A$" means

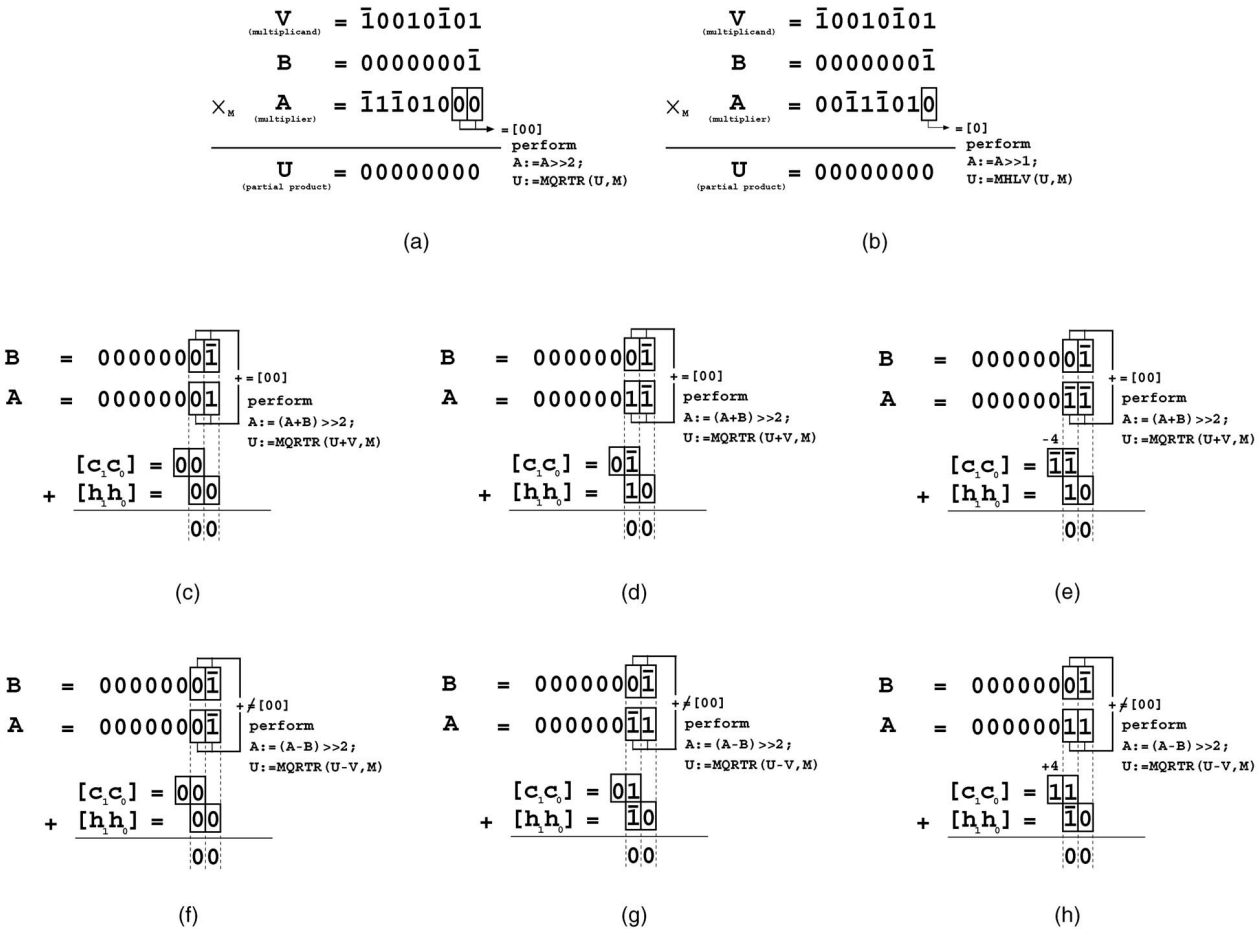


Fig. 2. Diagram showing the transformation process of $[a_1a_0]$.

that $\{A := (A-B)/4, B := A\}$ and $\{U := MQRTR(U-V, M), V := U\}$ have been carried out.

3.2 Hardware Algorithm for Montgomery Modular Multiplication

In order to enable the sharing of the hardware originally used for division and accelerate the calculation process, we modified Montgomery algorithm by introducing SD2 representation in operands, internal calculations, and the output result, and examine the least significant two digits of A, i.e., $[a_1a_0]$, to process them at each iteration when possible. The algorithm follows the same structure of the division algorithm and uses the same test conditions.

For the case $[a_1a_0] = 0$, we perform $U/4$ modulo M and shift down A two digit positions. The calculation of $U/4$ modulo M is performed with $MQRTR(U, M)$. See Fig. 2a.

If $[a_1a_0] = [10]$ or $[\bar{1}0]$, we perform $U/2$ modulo M and shift down A by one position, leaving the digit 1 or $\bar{1}$ that takes place in the least significant digit position to be processed in the next iteration. $U/2$ modulo M is calculated by using $MHLV(U, M)$. See Fig. 2b.

For the remaining cases, we need to determine whether the value of $[a_1a_0]$, i.e., $a_0 + 2 \cdot a_1$, is 1 or -1 or 3 or -3.

In the division algorithm (Algorithm 3), the test condition $([a_1a_0] + [b_1b_0]) \bmod 4$ is used to determine whether $(A+B)$ or $(A-B)$ is divisible by 4. In order to enable the sharing of the hardware, we employ the same variable B as the one used in the division algorithm and use the same test condition to determine the different values of $[a_1a_0]$. We will show that the same operations that are used in the division algorithm can be reused by initializing the variable B with its least significant digit with the value of $\bar{1}$, i.e., $b_0 = \bar{1}$, and the rest of the digits with the value of 0.

Each case is described next. For the case $[a_1a_0] = [01]$ or $[1\bar{1}]$, which means that the value of $[a_1a_0]$ is equal to 1, $([a_1a_0] + [b_1b_0]) = 0 \bmod 4$, so, as in the division algorithm, $U := MQRTR(U+V, M)$ and $A := (A+B) >> 2$ are performed. See Fig. 2c and Fig. 2d.

For the case $[a_1a_0] = [0\bar{1}]$ or $[\bar{1}1]$, which means that the value of $[a_1a_0]$ is equal to -1, the condition $([a_1a_0] + [b_1b_0]) = 0 \bmod 4$ does not hold. Therefore, $U := MQRTR(U-V, M)$ and $A := (A-B) >> 2$ are performed. See Fig. 2f and Fig. 2g.

When $[a_1a_0] = [\bar{1}\bar{1}]$, which means that the value of $[a_1a_0]$ is equal to -3, the condition $([a_1a_0] + [b_1b_0]) = 0 \bmod 4$ holds, so $U := MQRTR(U+V, M)$ and $A := (A+B) >> 2$ are performed as in the case where the value of $[a_1a_0]$ is equal to 1. During the operation of $(A+B)$, a carry digit c_1 is generated with the value of $\bar{1}$, which can be interpreted as an addition of -4 to A. So, this process can be seen as a transformation in the representation of the least significant two digits $[a_1a_0]$ from -3 into $-4+1$. See Fig. 2e.

In the same way, when $[a_1a_0] = [11]$, which means that the value of $[a_1a_0] = 3$, the condition $[a_1a_0] + [b_1b_0] = 0 \bmod 4$ does not hold. Hence, operations $A := (A - B) >> 2$ and $U := MQRTR(U - V, M)$ are performed. During the subtraction, the carry digit c_1 with the value of 1 is generated. This represents an addition of 4 to A. Then, this process can be interpreted as a transformation of the representation of $[a_1a_0]$ from 3 into $+4 - 1$. See Fig. 2h.

As a consequence, all calculations can be performed with the same operations used for the division case, i.e., shifts, $MHLV$ and $MQRTR$ operations. All results are always bounded in absolute value by M.

During the calculations, due to operations $(A + B)$ or $(A - B)$, expansion of maximum two digit positions of A may occur because of addition rules in SD2. For this reason, the algorithm always process $n + 2$ digit positions of A and Montgomery constant R is, therefore, equal to $2^{(n+2)}$.

The "while" loop is implemented with the same variable P as the one used in the division case. It is initialized to the same value, i.e., 2^{n+1}. It indicates the upper bound of A and it is shifted to the right until the final condition of $P = 1$.

We present the accelerated Montgomery multiplication algorithm.

Algorithm 4 (Hardware Algorithm for Montgomery Modular Multiplication)
Inputs: $M : 2^{n-1} < M < 2^n$ and $\gcd(M, 2) = 1$
$X, Y : -M < X, Y < M$
Output: $Z = XY2^{-(n+2)} \bmod M$ $(-M < Z < M)$
Algorithm:
Step 1: $A := Y$; $B := \bar{1}$; $U := 0$; $V := X$; $M := M$;
$P := 2^{n+1}$; $D := 1$; $s := 1$;
Step 2: **while** $p_0 \neq 1$ **do**
if $[a_1a_0] = 0$ **then** /* $A \equiv 0 \pmod 4$*/
$A := A >> 2$; $U := MQRTR(U, M)$;
if $p_1 = 0$ **then** $P := P >> 2$;
else $P := P >> 1$; $s := 0$; **endif**
elseif $a_0 = 0$ **then** /* $A \equiv 2 \pmod 4$ */
$A := A >> 1$; $U := MHLV(U, M)$;
$P := P >> 1$;
else /* $A \equiv 1 \pmod 4$ or $A \equiv 3 \pmod 4$ */
if $([a_1a_0] + [b_1b_0]) \bmod 4 = 0$ **then** $q := 1$
else $q := -1$;
$A := (A + qB) >> 2$;
$U := MQRTR(U + qV, M)$;
if $p_1 = 0$ **then** $P := P >> 2$;
else $P := P >> 1$; $s := 0$; **endif**
endif
endwhile
Step 3: **if** $s = 1$ **then** $U := MHLV(U, M)$;
Step 4: $Z := U$;

In the algorithm, s is initialized to the value of 1. Since the algorithm processes the multiplier by one or two digits, a variable s is used to indicate whether the "while" loop finishes after processing $n + 2$ digits or $n + 1$ digits of A. In the former case, s is set to the value of 0. In the latter case, s retains the same value of 1 indicating that an additional operation is required. It is shown below that, in this case, the unprocessed digit of A always has the value of 0 and $MHLV(U, M)$ is needed to be performed in Step 3.

Proposition 1. *In Algorithm 4, if Step 2 finishes after processing $n + 1$ digits of A, the remaining unprocessed digit of A will always have the value of 0.*

Proof. During the iteration, the operations $A := (A + B) >> 2$ or $A := (A - B) >> 2$ may be performed. If the most significant digit of A at initialization time has the value of 1, this digit can be expanded into $[1\bar{1}]$ or $[10]$ due to the addition rules of SD2 numbers described in Table 1. For the former case, further expansion does not occur because, after updating the value of A, the most significant digit is followed by $\bar{1}$. For the latter case, the digits $[10]$ can in turn be transformed into $[1\bar{1}0]$ and further expansion does not occur (because, again, the most significant digit of the updated value of A is followed by $\bar{1}$). If the most significant digit of A at initialization time has the value of $\bar{1}$, this digit can be transformed into $[\bar{1}1]$ and no further expansion occurs. Therefore, when A is positive, expansion of a maximum of two digits may occur and, when A is negative, expansion of only one digit may occur.

Let us assume that $n - 1$ digits of A have been processed and only the remaining three digits are left to be processed. We call these three digits $[a'_2a'_1a'_0]$, which corresponds to the digits $[a_{n+1}a_na_{n-1}]$ of A at initialization time. If A is positive, only the cases that might leave the digit a'_2 unprocessed with the value different to 0 are when a'_2 is 0 and the digits $[a'_1a'_0]$ are processed together generating a carry digit $c_1 \neq 0$ or a'_2 is 1 and $[a'_1a'_0]$ are processed together without generating any carry digit. We will show that these cases never happen. In the former case, if a'_2 has the value of 0, the only case where the digits $[a'_1a'_0]$ might be processed together generating a carry digit is when $[a'_1a'_0] = [11]$. But, this can never occur because the initial value of A is bounded in absolute value by M. In the latter case, if a'_2 is 1, then $[a'_1a'_0]$ can rather be $[\bar{1}0]$ or $[\bar{1}\bar{1}]$. No other possibilities are left because, again, the initial value of A is bounded in absolute value by M. When $[a'_1a'_0] = [\bar{1}0]$, then the digit a'_0 is processed alone, leaving the digit a'_1 to be processed together with a'_2. When $[a'_1a'_0] = [\bar{1}\bar{1}]$, they are processed together, generating a carry digit c_1 of value $\bar{1}$ that is added to the left digit a'_2 of value 1, leaving this unprocessed digit with the value of 0. Now, let us assume the case when A is negative. Expansion of a maximum of only one digit may occur. So, the digits $[a'_2a'_1a'_0]$ can only have the values of $[0\bar{1}1]$ or $[00\bar{1}]$. None of these cases leaves the digit a'_2 unprocessed with the value different from 0. □

Fig. 3 shows an example of a Montgomery multiplication, $-115 \times 249 \times 2^{-10} \bmod 251 = 137$, where $n = 8$ by Algorithm 4. The leftmost column shows the calculations which have been carried out. For example, "$A >> 1$" means that the operations $A := A >> 1$ and $U := MHLV(U, M)$ have been carried out and "$(A + B) >> 2$" means that $A := (A + B) >> 2$ and $U := MQRTR(U + V, M)$ have been carried out. In this example, Step 2 terminates with $s = 0$, so no extra calculations are needed.

$M = [11111011]_2$ (251), $X = [\bar{1}0010\bar{1}0\bar{1}]_{SD}$ (-115), $Y = [1111111\bar{1}\bar{1}]_{SD}$ (249)

	A		B		P	s	U		V	
	$1111111\bar{1}\bar{1}$	(249)	$0000000\bar{1}$	(-1)	1000000000	1	00000000	(0)	$\bar{1}0010\bar{1}01$	(-115)
$(A+B) >> 2$	$0100001\bar{0}$	(62)	$0000000\bar{1}$	(-1)	0010000000	1	00100010	(34)	$\bar{1}0010\bar{1}01$	(-115)
$A >> 1$	$0010000\bar{1}$	(31)	$0000000\bar{1}$	(-1)	0001000000	1	00010001	(17)	$\bar{1}0010\bar{1}01$	(-115)
$(A-B) >> 2$	$0001\bar{1}000$	(8)	$0000000\bar{1}$	(-1)	0000100000	1	$01\bar{1}0001\bar{1}$	(33)	$\bar{1}0010\bar{1}01$	(-115)
$A >> 2$	$000001\bar{1}0$	(2)	$0000000\bar{1}$	(-1)	0000010000	1	$0100101\bar{1}$	(71)	$\bar{1}0010\bar{1}01$	(-115)
$A >> 1$	$0000001\bar{1}$	(1)	$0000000\bar{1}$	(-1)	0000000010	1	10100001	(161)	$\bar{1}0010\bar{1}01$	(-115)
$(A+B) >> 2$	00000000	(0)	$0000000\bar{1}$	(-1)	0000000001	0	10001001	(137)	$\bar{1}0010\bar{1}01$	(-115)
U							10001001	(137)		

$Z = [10001001]_{SD}$ (137)

Fig. 3. A Montgomery modular multiplication by **Algorithm 4**.

3.3 The Combined Hardware Algorithm

The hardware algorithm for modular multiplication/division is presented here. It consists of four steps. Initialization of variables takes place in Step 1. The core of the algorithm is described in Step 2. Final calculations are performed in Step 3 and, in Step 4, the output result is selected. The input $mode$ is used to select the mode of operation.

Algorithm 5 (Hardware Algorithm for Modular Multiplication/Division)

Inputs: $mode \in \{0, 1\}$
$M : 2^{n-1} < M < 2^n$ and $\gcd(M, 2) = 1$ (prime when $mode = 1$)
$X, Y : -M < X, Y < M$ ($Y \neq 0$ when $mode = 1$)

Output: $mode = 0 : Z = XY2^{-(n+2)} \bmod M$ $(-M < Z < M)$
$mode = 1 : Z = X/Y \bmod M$ $(-M < Z < M)$

Algorithm:

Step 1: $A := Y$; $M := M$; $P := 2^{n+1}$; $D := 1$; $s := 1$;
if $mode = 0$ **then** $B := \overline{1}$; $U := 0$; $V := X$;
else $B := M$; $U := X$; $V := 0$; **endif**

Step 2: **while** $p_0 \neq 1$ **do**
if $[a_1a_0] = 0$ **then** /* $A \equiv 0 \pmod 4$ */
$A := A >> 2$; $U := MQRTR(U, M)$;
if $s = 0$ **then**
if $d_2 = 1$ **then** $s := 1$;
if $d_1 = 0$ **then** $D := D >> 2$;
else $P := P >> 1$; $s := 1$; **endif**
else /* $s = 1$ */
$D := D << 2$;
if $p_1 = 0$ **then** $P := P >> 2$;
else $P := P >> 1$; $s := 0$; **endif**
endif
elseif $a_0 = 0$ **then** /* $A \equiv 2 \pmod 4$ */
$A := A >> 1$; $U := MHLV(U, M)$;
if $s = 0$ **then**
if $d_1 = 1$ **then** $s := 1$;
$D := D >> 1$;
else /* $s = 1$ */ $D := D << 1$; $P := P >> 1$;
endif
else /* $A \equiv 1 \pmod 4$ or $A \equiv 3 \pmod 4$ */
if $([a_1a_0] + [b_1b_0]) \bmod 4 = 0$ **then** $q := 1$
else $q := -1$;
if $mode = 0$ **or** $s = 0$ **or** $d_0 = 1$ **then**
$A := (A + qB) >> 2$;
$U := MQRTR(U + qV, M)$;
if $s = 1$ **then**
if $mode = 0$ **and** $p_1 = 0$
then $P := P >> 2$;
else
if $p_1 = 1$ **then** $s := 0$;
$P := P >> 1$;
endif
$D := D << 1$;
else /* $s = 0$ */
if $d_1 = 1$ **then** $s := 1$;
$D := D >> 1$;
endif
else /* $mode = 1$ and $s = 1$ and $D > 1$ */
$\{A := (A + qB) >> 2,\ B := A\}$;
$\{U := MQRTR(U + qV, M),\ V := U\}$;
if $d_1 = 0$ **then** $s := 0$;
$D := D >> 1$;
endif
endif
endwhile

Step 3: **if** $mode = 0$ **and** $s = 1$ **then** $U := MHLV(U, M)$;
elseif $mode = 1$ **and** $([b_1b_0] = 3$ **or** $[b_1b_0] = -1)$
then $V := -V$;

Step 4: **if** $mode = 0$ **then** $Z := U$;
else $Z := V$;

In Step 1, variables A, M, P, D, and s are initialized to the values Y, M, 2^{n+1}, 1, and 1, respectively. Only the variables B, U, and V are initialized differently accordingly to the mode of operation. In multiplication mode, i.e., mode = 0, they are initialized to values $\overline{1}$, 0, and X, respectively. In division mode, i.e., mode = 1, they are initialized to values M, X, and 0.

The flag s is used in division mode to indicate the sign of δ, whereas, in multiplication mode, it is used to indicate if an extra operation of $MHLV(U, M)$ is required in Step 3.

During Step 2, the least significant two digits of A and B are checked to determine the different cases $A \equiv 0 \pmod 4$, $A \equiv 2 \pmod 4$, or $(A + B) \equiv 0 \pmod 4$ or $(A - B) \equiv 0 \pmod 4$. The operations $A >> 2$, $A >> 1$, $(A + B) >> 2$, and $(A - B) >> 2$, and the corresponding operations $MQRTR(U, M)$, $MHLV(U, M)$, $MQRTR(U + V, M)$, and $MQRTR(U - V, M)$, and the logic that selects the different cases are completely shared for both modes of operation. The logic that controls the operation of P is also shared for the cases that $A \equiv 0 \pmod 4$ and $A \equiv 2 \pmod 4$. It differs only when $A \equiv 1$ or $3 \pmod 4$, where P is shifted two positions in multiplication mode, whereas,

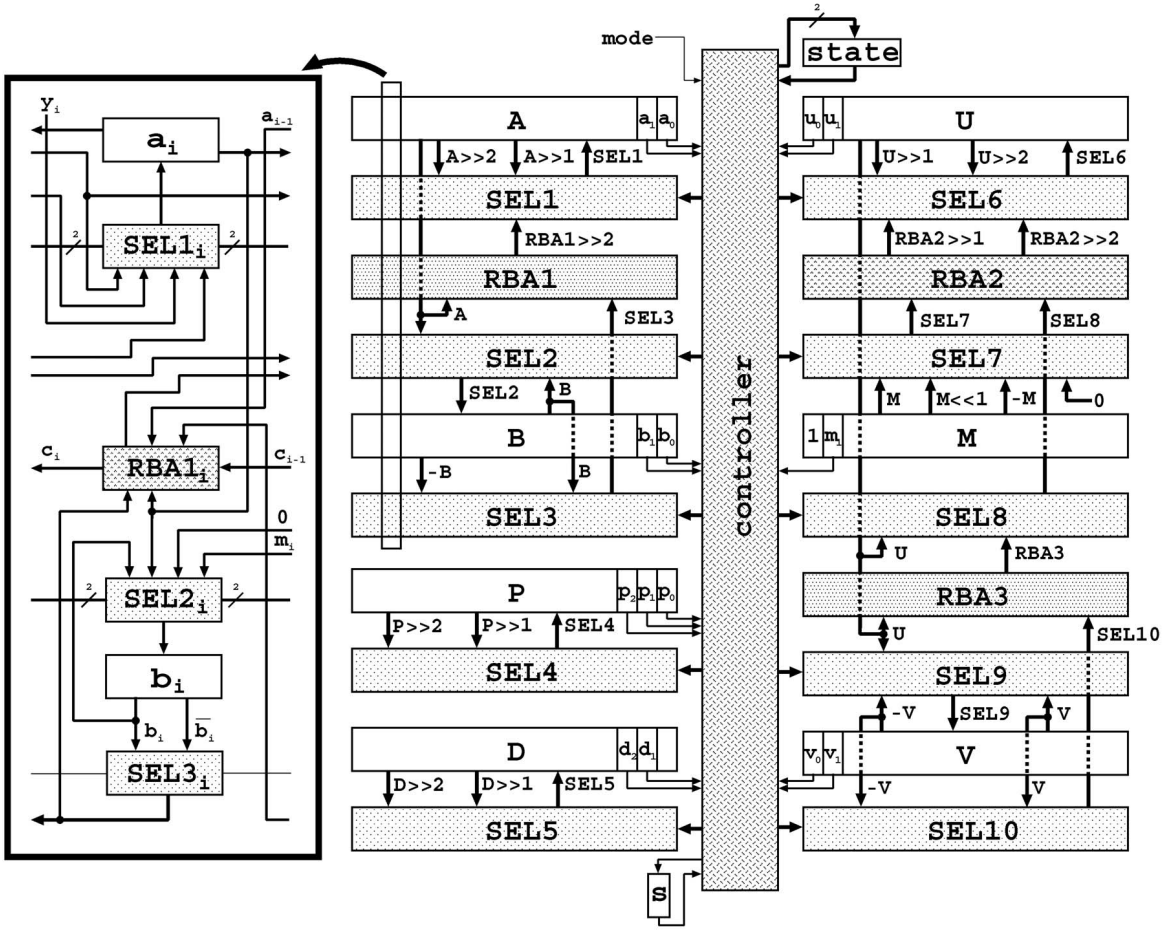


Fig. 4. Block diagram of a multiplier/divider.

in division mode, it is shifted by only one position. Division mode also requires the swapping operations and the logic to control the register D.

In Step 3, additional corrections are performed for both operations.

In Step 4, the output is selected between the values of U and V according to the mode of operation.

In division mode, for the cases $([a_1a_0]) \bmod 4 = 0$, the algorithm processes two digits of the operand. Otherwise, the algorithm processes only one digit. In multiplication mode, for the cases $([a_1a_0]) \bmod 4 = 0$ or 1 or 3, the algorithm processes two digits of the multiplier. For the remaining case $[a_1a_0] \bmod 4 = 2$, the algorithm processes only one digit of the multiplier. That is, the proposed algorithm behaves as a radix-4 algorithm when possible. Otherwise, it behaves as a radix-2 algorithm. We call the proposed algorithm as mixed radix-4/2 algorithm.

4 Hardware Implementation and Design

4.1 Hardware Implementation

We implement each iteration of the "while" loop in Step 2, i.e., one row in Fig. 1/Fig. 3, to be performed in one clock cycle.

A modular multiplier/divider based on Algorithm 5 consists of seven registers for storing A, B, P, D, U, M, and V, selectors, a small control circuit, and three SD2 adders, one of which is simpler. Fig. 4 shows a block diagram of the multiplier/divider.

The controller is a combinational circuit. It takes as inputs the least significant two digits of A, B, U, and V, the bit m_1, the least significant three digits of P, as well as the bits d_2 and d_1, the flag s, the two bits of the register $state$ that stores the number of the step, and one bit of *mode*. The outputs of the controller are signals to all the selectors and the inputs to the flag s and the register *state*.

As an example, we describe the behavior of the circuit components of the expanded diagram of Fig. 4 during an iteration of Step 2. If $[a_1a_0] \bmod 4 = 0$, the controller sends a signal to $SEL1$ to select $A >> 2$. If $[a_1a_0] \bmod 4 = 2$, then $SEL1$ selects $A >> 1$. Otherwise, $SEL1$ selects the output of $RBA1$. Additionally, if $([a_1a_0] + [b_1b_0]) \bmod 4 = 0$, then $SEL3$ selects B so that $A := (A + B) >> 2$ is performed. If not, $-B$ is selected to perform $A := (A - B) >> 2$. Also, if $mode = 1$ and $s = 1$ and $d_0 = 0$, $SEL2$ selects A so that $B := A$ is performed. Otherwise, it selects B, leaving the content of this register unaltered.

The redundant binary adder consists of a combinational circuit whose addition rule is shown in Table 1. To generate the ith digit of $(A + B)$ and c_i, the cell adder takes as inputs a_i, b_i, a_{i-1}, b_{i-1}, and c_{i-1}. $RBA2$ is much simpler than $RBA1$ and $RBA3$ because M is a binary number.

TABLE 2
The Number of Cells, Area, and Delay of a Multiplier/Divider, a Multiplier, and a Divider

n	circuit	#cells	critical path delay [ns]	total max. comp. time [ms]		area[mm^2]
				multiplication	division	
128	MUL/DIV	14259	8.69	0.77341	1.50336	2.232581
	DIV	13862	8.69	—	1.50336	2.180623
	MUL	8902	8.69	0.77341	—	1.419248
256	MUL/DIV	27893	8.76	1.53300	4.52892	4.792391
	DIV	28470	8.69	—	4.49273	4.764970
	MUL	17876	8.43	1.47525	—	3.185108
512	MUL/DIV	57073	8.76	3.02220	9.01404	9.659689
	DIV	53781	8.69	—	8.94201	9.103801
	MUL	34345	8.76	3.02220	—	6.788351

In multiplication mode, D is not used. Also, the algorithm can be implemented in a way that the most significant $n-2$ digits of B and the logic concerned with those digits in the adder are not used. Therefore, these parts can be disconnected during this mode to reduce power consumption. The circuit has a linear array structure with a bit-slice feature. The amount of hardware of the modular multiplier/divider is proportional to n. An n-bit modular multiplication is performed in at most $\lfloor \frac{2}{3}(n+2) \rfloor + 3$ clock cycles and an n-bit modular division in at most $2n+5$ clock cycles. Since the depth of the combinational circuit part is constant, the length of clock cycle is a constant independent of n.

4.2 Hardware Design and Evaluation

We described a modular multiplier/divider, as well as a modular multiplier and a modular divider separately in Verilog-HDL and synthesized them with Synopsys design Compiler using 0.35μm CMOS 3-metal technology provided by VLSI Design and Education Center (VDEC), the University of Tokyo, with the collaboration of Rohm Corporation. Table 2 shows the number of cells, critical path delay, the total maximum computational time (critical path delay $\times$ maximum number of clock cycles), and the area of the described circuits for $n = 128$, 256, and 512. The implemented modular multiplier is based on the Montgomery algorithm with the acceleration we introduced for processing two digits in SD2 system when possible. The modular divider is based on the extended Binary GCD algorithm with the acceleration we introduced for processing two digits of A when it is divisible by 4. As shown in the table, the total circuit area of the multiplier/divider is much smaller than the total sum of circuit areas of the modular multiplier and the modular divider with the critical path delays remaining practically to the same value.

5 Considerations

5.1 Applications to Modular Exponentiation

In applications where chained multiplications are required, such as modular exponentiation, calculations can be performed in Montgomery representation to accelerate the computations. Since the result Z of the modular multiplication always satisfies the condition $|Z| < M$, this can be used as input operands of the succeeding modular multiplication. Only one carry propagate addition is required at the end of the exponentiation to convert the result from the SD2 representation into the binary representation. If the result after conversion is negative, M is added to transform it into the range $[0, M-1]$.

Further acceleration can be obtained by the use of modular division. Consider the operation $X^e \bmod M$. The exponent e can be expressed in SD2 representation and it can be recoded to reduce the Hamming weight and, consequently, the number of operations. Modular exponentiation can be calculated by examining each digit of this exponent from the topmost significant position and performing a modular squaring when the digit has the value 0, a modular squaring and modular multiplication when the digit has the value 1, and a modular squaring and a modular division when the digit has the value $\bar{1}$. Since the output of modular division also satisfies the condition of $|Z| < M$, the output can be directly fed into the inputs of the succeeding operation. All the calculations can be performed in Montgomery representation without any special consideration. The result of the exponentiation can be converted into the binary representation in the same way as described above.

5.2 Applications to Cryptography

The proposed algorithm is efficient in sharing hardware resources and computational speed. However, in cryptographic applications, data dependent timing variation may provide information leakage. Timing and power attacks were initially introduced by Kocher et al. [12], [13] and various countermeasures have been proposed. They are based on the randomization of the private exponent [4], [18] and, on blinding the operands with a secret random number and unblinding it after exponentiation [12]. These countermeasures can be used to increase security of the system. The analysis of the security strength of the proposed algorithm with the different countermeasures is left as future work.

In the case where modular multiplication is required to be computed in constant time, at most $\lfloor \frac{2}{3}(n+2) \rfloor - \lfloor \frac{n+2}{2} \rfloor$ dummy operations have to be inserted. This is approximately $n/6$ clock cycles. Even including these operations, the proposed algorithm performs multiplication in $\lfloor \frac{2}{3}n + 2 \rfloor$ clock cycles, which is faster than performing the multiplication with the conventional radix-2 Montgomery algorithm. In the case where modular division needs to

be calculated in constant time, the testing condition of $[a_1 a_0] = 00$ and the corresponding operations can be omitted. After the termination condition of $P = 1$, D can be shifted to the right until it reaches the end. By doing so, modular division always finishes in exactly $2n + 5$ clock cycles.

In cryptographic applications, the fact that both calculations are carried out in the same hardware and not in separate places is advantageous in the sense that the electromagnetic power radiation emanates from only one source; therefore, no positional information is provided for the different operations.

6 Concluding Remarks

We have proposed a hardware algorithm for modular multiplication/division. It is based on the extended Binary GCD algorithm and on Montgomery modular multiplication, both of which have been modified and combined. The estimated total circuit area and critical path delay of the modular multiplier/divider based on the proposed algorithm show that it can be implemented in much smaller hardware than that necessary to implement multiplier and divider separately. We conclude that, among the various algorithms proposed in literature for calculating modular multiplication and division, the extended Binary GCD algorithm and the Montgomery modular multiplication algorithm seem to be the suitable ones to be combined. Not only the registers that store the operands and the combinational logic involved in the operations can be completely shared. Also, the combinational logic that controls the different operations is able to be shared.

Acknowledgments

The authors would like to thank Associate Professor Kazuyoshi Takagi for his valuable comments and discussions and the VLSI Design and Education Center (VDEC) for providing the library to synthesize the designs.

References

[1] ANSI X9.30, Public Key Cryptography for the Financial Services Industry: Part 1: The Digital Signature Algorithm (DSA). Am. Nat'l Standard Inst. Am. Bankers Assoc., 1997.

[2] J.-C. Bajard, L.-S. Didier, and P. Kornerup, "An RNS Montgomery Modular Multiplication Algorithm," *IEEE Trans. Computers,* vol. 47, no. 7, pp. 766-776, July 1998.

[3] R.P. Brent and H.T. Kung, "Systolic VLSI Array for Linear-Time GCD Computation," *Proc. VLSI '83,* F. Anceau and E.J. Aas, eds., pp. 145-154, 1983.

[4] J.-S. Coron, "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems," *Proc. Workshop Cryptographic Hardware and Embedded Systems,* pp. 292-302, 1998.

[5] W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Trans. Information Theory,* vol. 22, no. 11, pp. 644-654, Nov. 1976.

[6] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Trans. Information Theory,* vol. 31, no. 4, pp. 469-472, July 1985.

[7] W.L. Freking and K.K. Parhi, "Modular Multiplication in the Residue Number System with Application to Massively-Parallel Public-Key Cryptography Systems," *Proc. 34th Asilomar Conf. Signals, Systems, and Computers,* pp. 1339-1343, Oct. 2000.

[8] M.E. Kaihara and N. Takagi, "A VLSI Algorithm for Modular Multiplication/Division," *Proc. 16th IEEE Symp. Computer Arithmetic,* pp. 220-227, June 2003.

[9] S. Kawamura, M. Koike, F. Sano, and A. Shimbo, "Cox-Rower Architecture for Fast Parallel Montgomery Multiplication," *Proc. Advances in Cryptology-EUROCRYPT 2000,* pp. 523-538, May 2000.

[10] D.E. Knuth, *The Art of Computing Programming, Volume 2, Seminumerical Algorithms,* third ed. Reading Mass.: Addison-Wesley, 1998.

[11] N. Koblitz, "Elliptic Curve Cryptosystems," *Math. Computation,* vol. 48, no. 177, pp. 203-209, Jan. 1987.

[12] P.C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," *Proc. Advances in Cryptology-CRYPTO '96,* pp. 104-113, Aug. 1996.

[13] P.C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," *Proc. Advances in Cryptology (CRYPTO '99),* pp. 388-398, 1999.

[14] Ç.K Koç, T. Acar, and B.S. Kaliski Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro,* vol. 16, no. 3, pp. 26-33, June 1996.

[15] P. Kornerup, "High-Radix Modular Multiplication for Cryptosystems," *Proc. 11th IEEE Symp. Computer Arithmetic,* G. Jullien, M.J. Irwin, and E. Swartzlander, eds., pp. 277-283, 1993.

[16] P.L. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation,* vol. 44, no. 170, pp. 519-521, Apr. 1985.

[17] H. Orup, "Simplifying Quotient Determination in High-Radix Modular Multiplication," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W.H. McAllister, eds., pp. 193-199, 1995.

[18] E. Oswald and M. Aigner, "Randomized Addition-Subtraction Chains as a Countermeasure against Power Attacks," *Proc. Cryptographic Hardware and Embedded Systems-CHES 2001,* Ç.K. Koç, D. Naccache and C. Paar, eds., pp. 39-50, May 2001.

[19] S.N. Parikh and D.W. Matula, "A Redundant Binary Euclidean GCD Algorithm," *Proc. 10th Symp. Computer Arithmetic,* pp. 220-224, June 1991.

[20] K.C. Posch and R. Posch, "Modulo Reduction in Residue Number Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 5, pp. 449-454, May 1995.

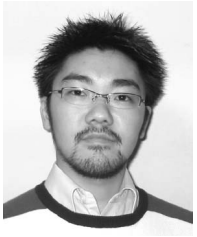
[21] R.L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM,* vol. 21, no. 2, pp. 120-126, Feb. 1978.

[22] N. Takagi, "A VLSI Algorithm for Modular Division Based on the Binary GCD Algorithm," *IEICE Trans. Fundamentals,* vol. E81-A, no. 5, pp. 724-728, May 1998.

[23] N. Takagi and S. Yajima, "Modular Multiplication Hardware Algorithms with a Redundant Representation and Their Application to RSA Cryptosystem," *IEEE Trans. Computers,* vol. 41, no. 7, pp. 887-891, July 1992.

[24] A.F. Tenca, G. Todorov, and Ç.K. Koç, "High-Radix Design of a Scalable Modular Multiplier," *Proc. Cryptographic Hardware and Embedded Systems-CHES 2001,* Ç.K. Koç, D. Naccache, C. Paar, eds., pp. 185-201, 2001.

[25] C.D. Walter, "Systolic Modular Multiplication," *IEEE Trans. Computers,* vol. 42, no. 3, pp. 376-378, Mar. 1993.



Marcelo E. Kaihara received the Ing. Electrónico degree from the University of Buenos Aires, Buenos Aires, Argentina, in 1999 and the ME degree in information engineering from Nagoya University, Nagoya, Japan, in 2003. He is currently working toward the PhD degree in information science at the same university. His research interests include hardware algorithms for cryptography and communications.

Naofumi Takagi (S'82-M'84-SM'03) received the BE, ME, and PhD degrees in information science from Kyoto University, Kyoto, Japan, in 1981, 1983, and 1988, respectively. He joined the Department of Information Science, Kyoto University, as an instructor in 1984 and was promoted to an associate professor in 1991. He moved to the Department of Information Engineering, Nagoya University, Nagoya, Japan, in 1994, where he has been a professor since 1998. His current interests include computer arithmetic, hardware algorithms, and logic design. He received the Japan IBM Science Award and Sakai Memorial Award of the Information Processing Society of Japan in 1995. He was an associate editor of the *IEEE Transactions on Computers* from 1996 to 2000. He is a senior member of the IEEE.