PAPER

# A Hardware Algorithm for Modular Multiplication/Division Based on the Extended Euclidean Algorithm

**Marcelo E. KAIHARA**[†a)], ***Nonmember and*** **Naofumi TAKAGI**[†b)], ***Member***

**SUMMARY** A hardware algorithm for modular multiplication/division which performs modular division, Montgomery multiplication, and ordinary modular multiplication is proposed. The modular division in our algorithm is based on the extended Euclidean algorithm. We employ our newly proposed computation method that consists of processing the multiplier from the most significant digit first to calculate Montgomery multiplication. Finally, the ordinary modular multiplication is based on shift-and-add multiplication. Each of these three operations is carried out through the iteration of simple operations such as shifts and additions/subtractions. To avoid carry propagation in all additions and subtractions, the radix-2 signed-digit representation is employed. A modular multiplier/divider based on the algorithm has a linear array structure with a bit-slice feature and carries out $n$-bit modular multiplication/division in $O(n)$ clock cycles, where the length of the clock cycle is constant and independent of $n$. This multiplier/divider can be implemented using a hardware amount only slightly larger than that of the modular divider.

***key words:*** *modular arithmetic, modular multiplication, modular division, Montgomery multiplication, extended Euclidean algorithm, hardware algorithm*

## 1. Introduction

Modular multiplication and modular division are basic operations in abstract algebra and play important roles in processing many public-key cryptosystems. For example, modular multiplication is used in the RSA cryptosystem [9] and in the Diffie-Hellman key exchange protocol [3]. Modular multiplication and modular division are required in the ElGamal [4] cryptosystem, in the DSA digital signature scheme [1] and to compute point operations in elliptic curve cryptosystem with curves defined over $GF(p)$ [7].

In applications where long chained multiplications are required, such as in the deciphering process of RSA, the Montgomery method [8] has an outstanding performance. On the other hand, in applications where few modular multiplications are required, as in the enciphering process of RSA, performing the calculations using the ordinary modular multiplication may be faster.

Considering the need of personal computers and mobile devices to manage several security protocols, and the great demand in technology to shrink hardware to reduce fabrication costs, it is important to develop an algorithm for calculating modular multiplication/division that can be implemented in compact hardware.

In a previous publication [5], we proposed a hardware algorithm for modular multiplication/division that calculates modular division and Montgomery multiplication where the calculation of the modular division is based on the extended Binary GCD algorithm. In this paper, we propose a hardware algorithm for modular multiplication/division which calculates modular division, Montgomery multiplication, and also, ordinary modular multiplication with similar hardware resources to that necessary to calculate modular division only. To the best of our knowledge, there is no other work proposed in the literature that combines these three operations.

In the hardware algorithm to be proposed, modular division is based on the extended Euclidean algorithm. We improve the hardware algorithm for modular division originally presented in [10] to reduce hardware requirements by sharing the modular reduction hardware component. Montgomery multiplication is based on our newly proposed computation method that consists of processing the multiplier from the most significant digit first. This method makes it possible to compute Montgomery multiplication using almost the same hardware required for computing the modular division. The ordinary modular multiplication is based on the classical shift-and-add multiplication. We modify this algorithm in order to use almost the same hardware. Hence, the three operations share almost all the hardware components reducing the required hardware resources considerably. Each of the three operations is carried out through the iteration of shifts and additions/subtractions. In order to avoid carry propagation in all additions and subtractions, the radix-2 signed-digit (SD2) representation is employed.

A modular multiplier/divider based on our algorithm has a linear array structure with a bit-slice feature and is suitable for VLSI implementation. The hardware amount of an $n$-bit modular multiplier/divider is proportional to $n$ and is slightly larger than that of the modular divider. It performs an $n$-bit modular multiplication/division in $O(n)$ clock cycles where the length of a clock cycle is constant independent of $n$.

In the next section, we will explain the extended Euclidean algorithm, the Montgomery multiplication algorithm and the ordinary modular multiplication algorithm. We also explain basic operations in the SD2 system. In Sect. 3, we propose a hardware algorithm for modular multiplication/division. In Sect. 4, we discuss hardware imple-

Manuscript received October 5, 2004.
Manuscript revised June 14, 2005.
Final manuscript received August 15, 2005.
[†]The authors are with the Department of Information Engineering, Nagoya University, Nagoya-shi, 464–8603 Japan.
a) E-mail: mkaihara@takagi.nuie.nagoya-u.ac.jp
b) E-mail: ntakagi@is.nagoya-u.ac.jp
DOI: 10.1093/ietfec/e88–a.12.3610

mentation. Finally, in Sect. 5, we present our concluding remarks.

## 2. Preliminaries

### 2.1 Extended Euclidean Algorithm for Modular Division

One of the well known methods for calculating modular division is the Extended Euclidean Algorithm [6].

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 $Z(< M)$ so that $Z \times Y \equiv X \pmod{M}$.

The algorithm performs modular division by intertwining a procedure to find the modular quotient with that for calculating $\gcd(Y, M)$.

**[Algorithm 1]** (Modular Division)

```
Inputs: $M$: $2^{n-1} < M < 2^n$, and odd prime
        $X, Y$: $0 \leq X < M$, $0 < Y < M$
Output: $Z = X/Y \bmod M$
Algorithm:
  $\mathcal{A} := M; \mathcal{B} := Y; U := 0; V := X;$
  while $|\mathcal{B}| \neq 1$ do
    Choose integer $Q$ so that $|\mathcal{A} - \mathcal{B} \cdot Q| < |\mathcal{B}|$;
    $\mathcal{A}' := \mathcal{A} - \mathcal{B} \cdot Q$;
    Calculate $U'$ which satisfies
      $U' \equiv U - V \cdot Q \pmod{M}$ and $|U'| < M$;
    $\mathcal{A} := \mathcal{B}; \mathcal{B} := \mathcal{A}'$;
    $U := V; V := U'$;
  endwhile
  if $\mathcal{B} = -1$ then $Z' := -V$ else $Z' := V$;
  if $Z' < 0$ then $Z := Z' + M$ else $Z := Z'$;
```

$\mathcal{A}$ ($\mathcal{A}'$) and $\mathcal{B}$ are involved in the calculation of GCD and are allowed to be negative. $U$ ($U'$) and $V$ are used for calculating the quotient and are also allowed to be negative. $V \times Y \equiv \mathcal{B} \times X \pmod{M}$ always holds. Since the final $\mathcal{B}$ satisfies $|\mathcal{B}| = 1$, then $Z' \times Y \equiv X \pmod{M}$. Also, as the condition of $|V| < M$ always holds, then $-M < Z' < M$. Therefore, $Z \times Y \equiv X \pmod{M}$ and $0 \leq Z < M$ hold, resulting $Z$ as the quotient of $X/Y$ modulo $M$.

### 2.2 Montgomery Multiplication

Montgomery introduced an efficient algorithm for calculating modular multiplication [8]. Consider the residue class ring of integers with an odd modulus $M$, and let $X$ and $Y(= [y_{n-1}y_{n-2} \cdots y_0])$ be elements of the ring. The Montgomery multiplication algorithm now calculates $Z(< M)$ so that $Z \equiv X \times Y \times R^{-1} \pmod{M}$ where $R$ is an arbitrary constant $R > M$ and relatively prime to $M$. This constant $R$ usually takes the value of $2^n$ when the calculations are performed in radix-2 with an $n$-bit modulus $M$.

The radix-2 Montgomery multiplication algorithm is described below.

**[Algorithm 2]** (Montgomery Multiplication)

```
Inputs: $M : 2^{n-1} < M < 2^n$ and odd
        $X, Y : 0 \leq X, Y < M$
Output: $Z = X \cdot Y \cdot 2^{-n} \bmod M$
Algorithm:
  $U := 0$;
  for $i := 0$ to $n - 1$ do
    $U := (U + y_i \cdot X)/2 \bmod M$;
  endfor
  if $U \geq M$ then $Z := U - M$ else $Z := U$;
```

### 2.3 Ordinary Modular Multiplication

The usual method for calculating the ordinary modular multiplication, i.e. the calculation of $Z$ such that $Z \equiv X \times Y \pmod{M}$, is known as shift-and-add multiplication and it is shown below. In this method, the bits of the multiplier are scanned from the most significant position first.

**[Algorithm 3]** (Ordinary Modular Multiplication)

```
Inputs: $M : 2^{n-1} < M < 2^n$
        $X, Y : 0 \leq X, Y < M$
Output: $Z = X \cdot Y \bmod M$
Algorithm:
  $U := 0$;
  for $i := n - 1$ downto $0$ do
    $U := (2 \cdot U + y_i \cdot X) \bmod M$;
  endfor
  $Z := U$;
```

### 2.4 Basic Operations in SD2 System

In order to perform additions and subtractions without carry propagation, we represent the internal variables as $(n + 1)$-digit radix-2 signed digit (SD2) numbers [2] where $n$ is the bit length of the modulus $M$. The SD2 representation has a fixed radix 2 and a digit set $\{\bar{1}, 0, 1\}$, where $\bar{1}$ denotes $-1$. An $(n + 1)$-digit SD2 integer $A = [a_n a_{n-1} \cdots a_0]$ ($a_i \in \{\bar{1}, 0, 1\}$) has the value $\sum_{i=0}^{n} a_i \cdot 2^i$.

The algorithm requires a doubling procedure for an SD2 integer without overflow [10]. In order to illustrate how this procedure is implemented, let us take two $(n + 1)$-digit SD2 integers and call them $A$ and $S$. The doubling procedure $A := DBL(S)$, i.e., the calculation of $A$ so that $A = 2 \cdot S$, is only necessary when $S$ satisfies $s_n = 0$ or $s_{n-1} = -s_n$ and is performed as follows. When $s_n = 0$, $A = [s_{n-1}s_{n-2}s_{n-3} \cdots s_1 s_0 0]$ and otherwise ($s_{n-1} = -s_n$), $A = [s_n s_{n-2} s_{n-3} \cdots s_1 s_0 0]$.

Procedures for addition, doubling and halving in modulo $M$ in the SD2 system are also required and are described below. See also [10], [11].

Let the modulus $M$ $(= [1m_{n-2} \cdots m_1 1])$ be an $n$-bit binary odd integer satisfying $2^{n-1} < M < 2^n$. Let $U$, $V$ and $T$ be $(n + 1)$-digit SD2 integers satisfying $-M < U, V, T < M$.

A modular addition $T := MADD(U, V, M)$, i.e., the calculation of $T$ so that $T \equiv U + V \pmod{M}$, is performed in two steps. In the first step, we calculate $W := U + V$ in the SD2 system where $W$ results in an $(n+2)$-digit SD2 number. In the second step, if the value of the number formed by the three most significant digits of $W$, i.e. the value of $[w_{n+1}w_nw_{n-1}]$, is negative or zero or positive, we add $M$ or 0 or $M'$ to $W$, respectively. $M' = [\bar{1}0m'_{n-2}...m'_1 1]$ is a $(n+1)$-digit SD2 number where $m'_i$ is 1 or 0 accordingly as $m_i$ is 0 or 1, and has the value $-M$. This addition is also performed in the SD2 system. Since all the digits of the addend are non-negative except the most significant one, the addition in this step is simpler than the addition of two SD2 numbers. For the details of the modular addition procedure, see [11].

Modular doubling $T := MDBL(U, M)$, i.e., the calculation of $T$ so that $T \equiv 2 \cdot U \pmod{M}$, can be performed by applying the second step of the modular addition to $2 \cdot U$, which is obtained by shifting $U$ one position to the left.

Modular halving $T := MHLV(V, M)$, i.e., the calculation of $T$ so that $T \equiv V/2 \pmod{M}$, is performed through two steps. In the first step, we add $M$ to $V$ when $V$ is odd, i.e. when $v_0 \neq 0$. No action is required when $V$ is even. In the second step, we shift the result of the first step one position to the right throwing away the least significant digit, which is 0. (Recall that $M$ is odd.)

Procedures *MADD*, *MDBL* and *MHLV* can be performed in a constant time independent of $n$ by means of combinational circuits.

## 3. A Hardware Algorithm for Modular Multiplication/Division

We propose a hardware algorithm that performs modular division, Montgomery multiplication and ordinary modular multiplication, which is efficient in execution time and hardware requirements. We present first our improved modular division algorithm and then our newly proposed algorithm for computing Montgomery multiplication, and finally, the combined modular multiplication/division algorithm.

### 3.1 Improved Hardware Algorithm for Modular Division

We introduce two modifications to improve the hardware algorithm for modular division based on the extended Euclidean algorithm proposed in [10]. In the hardware algorithm proposed in [10], variables $\mathcal{A}$ and $\mathcal{B}$ of [Algorithm 1] are represented by using two $n$-digit SD2 integers, $A$ and $B$, and two $n$-digit binary integers of the form $2^i$, $P$ and $D$, so that $\mathcal{A} = A/(P/D)$ and $\mathcal{B} = B/P$. Note that $P$ and $D$ have only one bit of value 1. $U$ and $V$ are $(n+1)$-digit SD2 integers satisfying $-M < U, V < M$. The calculation of GCD is performed as a series of integer divisions where the divisor and the remainder of each of these divisions are taken as the new dividend and the new divisor of the next integer division respectively. To simplify the quotient digit determination, the variable $B$ which stores the divisor is strongly normalized. This means that $B$ is rewritten to change the representation in the SD2 system without modifying the numerical value so that $b_{n-1} \neq 0$ and $b_{n-2} = 0$, i.e., $[b_{n-1}b_{n-2}] = [10]$ or $[\bar{1}0]$. Then, integer division is performed as a series of subtractions $A - q \cdot B$ in the SD2 system.

The first modification that we introduce is related to the length and the representation of the input operands. In the original algorithm [10], after finishing the calculations in the SD2 system, the result of $(n+1)$-digits is converted into the binary representation and then reduced to the range $[0, M-1]$. This step involves a carry propagation addition, a full bit comparison and another possible carry propagation addition. All these operations are time consuming. In order to enable the feed back of the output represented in SD2 directly into the inputs and avoid this time consuming step, we represent the variables $\mathcal{A}$ and $\mathcal{B}$ with the same digit length as $U$ and $V$ and in the same SD2 representation. Note that in the SD2 system, operands $X$ and $Y$ can still be given in the ordinary binary representation. Increasing the length of the operands and allowing them to be represented in the SD2 system do not alter the structure of the algorithm. The only correction that is required in the algorithm is the displacement by one of the subindex positions in the variables where the digits are examined to control the different operations. As the input operands are now represented as numbers of $(n+1)$-digits, and $|Y| < M$, $b_{n-1}$ can never take the value of 1 when $b_n$ is equal to 1 at initialization time. Therefore, Step 2-0 of the original hardware algorithm [10] is eliminated.

The second modification that we introduce is related to the order of the steps. The structure of the hardware algorithm proposed in [10] follows the structure of [Algorithm 1]. Integer division is performed between $A$ and $B$ which contains the dividend $\mathcal{A}$ and the divisor $\mathcal{B}$ respectively. After performing each division, the divisor $\mathcal{B}$ and the remainder $\mathcal{A}'$ are taken as the new dividend and the new divisor, respectively, for the next integer division. In order to accomplish this, a swap operation between the variables $A$ and $B$ is performed. Normalization is then applied to variable $B$. To reduce hardware requirements, we change the order of the steps. After finishing the integer division, variable $A$, which contains the remainder, is normalized. Then, a swap operation is performed so that integer division is performed between $A$ and $B$ in the same way as the original hardware algorithm. Consequently to the change of the order, the variables $A$ and $B$ need to be initialized with the values interchanged. Equivalent modifications are required to the variables $U$ and $V$. As a result, this modification enables the operation *DBL* to be applied only to variable $A$ and enables the sharing of the modular reduction hardware component between the operations *MDBL* and *MADD* since these two operations are now applied to the same variable $U$. This means that, the proposing algorithm is more efficient in terms of hardware requirements than the one proposed in [10].

The improved hardware algorithm is described below:

**[Hardware Algorithm 1]** (Modular Division)
*Inputs:* $M : 2^{n-1} < M < 2^n$ and odd prime
$X, Y : -M < X, Y < M, \quad Y \neq 0$

```
Output: $Z : Z \equiv X/Y \pmod{M}$ and $-M < Z < M$
Algorithm:
  Step 1:
    $A := Y; B := M; U := X; V := 0; M := M;$
    $P := 1; D := 1;$
  Step 2:
    Step 2-1: [Normalization of A]
      while $p_n = 0$ and $[a_n a_{n-1}] \neq [10]$
        and $[a_n a_{n-1}] \neq [\bar{1}0]$ do
          if $[a_n a_{n-1} a_{n-2}] = [1\bar{1}1]$ or
            $[a_n a_{n-1} a_{n-2}] = [011]$ then
              $[a_n a_{n-1} a_{n-2}] := [10\bar{1}]$;
          elseif $[a_n a_{n-1} a_{n-2}] = [\bar{1}1\bar{1}]$ or
            $[a_n a_{n-1} a_{n-2}] = [0\bar{1}\bar{1}]$ then
              $[a_n a_{n-1} a_{n-2}] := [\bar{1}01]$;
          else
            $A := DBL(A)$;
            $U := MDBL(U, M)$;
            $P := 2 \cdot P; D := 2 \cdot D$;
          endif
      endwhile
    Step 2-2: [Swapping]
      $SWAP(A, B); SWAP(U, V)$;
      if $p_n = 1$ then
        while $d_0 = 0$ do
          $D := D/2; V := MHLV(V, M)$;
        endwhile
        goto Step 3;
      endif
    Step 2-3: [Integer Division]
      / * Main Stage * /
      while $d_0 = 0$ do
        if $a_n = 0$ or $a_{n-1} = -a_n$ then $S := A$;
        else
          $q := a_n \cdot b_n$;
          $S := A - q \cdot B$;
          $U := MADD(U, -q \cdot V, M)$;
        endif
        if $(s_n = 0$ or $s_{n-1} = -s_n)$ then
          $A := DBL(S); D := D/2$;
          $V := MHLV(V, M)$;
        else
          $A := S$;
        endif
      endwhile
      / * Termination Stage * /
      $r := sgn([a_n a_{n-1}])$;
      while $sgn([a_n a_{n-1}] = r$ and
        $(abs([a_n a_{n-1} a_{n-2}]) \geq 3$ or
        $(b_{n-2} = -b_n$ and $abs([a_n a_{n-1} a_{n-2}]) = 2))$
      do
        $q := r \cdot b_n$;
        $A := A - q \cdot B$;
        $U := MADD(U, -q \cdot V, M)$;
      endwhile
      goto Step 2-1;
  Step 3: [Correction]
    if $b_n = \bar{1}$ then $V := -V$;
  Step 4:
    $Z := V$;
```

In Step 2-3, we perform an integer division. We perform the subtraction of $A - q \cdot B$ in the SD2 system. In this subtraction, we use the special addition rule detailed in [10] at the most significant two positions. We can show that in the main stage, no two successive SD2 additions are performed without doubling $A$ $(DBL(S))$. The termination stage, at the end of the integer division, avoids the situation where the final $|A|$ $(|\mathcal{A}'|)$ is very near to $|B|$ $(|\mathcal{B}|)$ which makes the convergence of the whole computation very slow. For the details, see [10].

A numerical example of a modular division performed by [Hardware Algorithm 1] is given overleaf in Fig. 1. The calculation of 205/199 mod 251 is performed using $n = 8$. Lines 1 − 3 represent the equivalent to the first iteration of [Algorithm 1]. In line 3, the remainder $\mathcal{A}'$ is $\mathcal{A}'= 52$ and the divisor $\mathcal{B}$ is $\mathcal{B}= 199$. Lines 4 − 8 represent the equivalent to the second iteration. In line 8, $\mathcal{A}'= -9$ and $\mathcal{B}= 52$. Lines 9 − 15 represent the equivalent to the third iteration. In line 15, $\mathcal{A}'= -2$ and $\mathcal{B}= -9$, and so on. In line 23, $p_n = 1$, and $\mathcal{B}= -1$, which represent the end of the series of divisions. As the divisor is normalized during division, this divisor needs to be restored so that its most significant position is aligned to the most significant position of the dividend. This difference of positions is represented by the variable $D$. Since we are only interested in the result of the modular divisions, we divide $V$ by means of $V := MHLV(V, M)$ until $D = 1$. Line 23 represents this operation. As $\mathcal{B}= -1$, in line 24, $V$ is negated in the SD2 system. The result is $Z = -86 \equiv 165 \pmod{251}$.

## 3.2 New Algorithm for Computing Montgomery Multiplication

In order to implement Montgomery multiplication using the same hardware components needed for modular division, we modify [Algorithm 2] so that the multiplier is processed from the most significant digit. Here, instead of halving the intermediate result, we halve the multiplicand in modulo M. We present the modified Montgomery algorithm. Here, $y_n = 0$.

**[Algorithm 4]** (Modified Montgomery Multiplication)

```
Inputs: $M : 2^{n-1} < M < 2^n$ and $\gcd(M, 2) = 1$
        $X, Y : 0 \leq X, Y < M$
Output: $Z = X \cdot Y \cdot 2^{-n} \bmod M$
Algorithm:
  $U := 0; V := X$;
  for $i := n$ downto 0 do
    $U := (U + y_i \cdot V) \bmod M$;
    $V := V/2 \bmod M$;
  endfor
  $Z := U$;
```

$M = [11111011]_2$ (251), $X = [1\bar{1}1010\bar{1}01]_{SD}$ (205), $Y = [011000111]_{SD}$ (199)

| | | | $A$ | $B$ | $P$ | $D$ | $U$ | $V$ |
|---|---|---|---|---|---|---|---|---|
| | | | 011000111 | 011111011 | 000000001 | 0000000001 | $1\bar{1}1010\bar{1}01$ | 000000000 |
| 1 | *Step* 2-1 | *NORM*(*A*) | $10\bar{1}000111$ | 011111011 | 000000001 | 0000000001 | $1\bar{1}1010\bar{1}01$ | 000000000 |
| 2 | *Step* 2-2 | *SWAP* | 011111011 | $10\bar{1}000111$ | 000000001 | 0000000001 | 000000000 | $1\bar{1}1010\bar{1}01$ |
| 3 | *Step* 2-3 | $A := A - B$ | $0010\bar{1}0100$ | $10\bar{1}000111$ | 000000001 | 0000000001 | $001\bar{1}100\bar{1}0$ | $1\bar{1}1010\bar{1}01$ |
| 4 | *Step* 2-1 | *NORM*(*A*) | $010\bar{1}01000$ | $10\bar{1}000111$ | 000000010 | 0000000010 | $0\bar{1}0\bar{1}00001$ | $1\bar{1}1010\bar{1}01$ |
| 5 | *Step* 2-1 | *NORM*(*A*) | $10\bar{1}010000$ | $10\bar{1}000111$ | 000000100 | 0000000100 | $0\bar{1}100\bar{1}11\bar{1}$ | $1\bar{1}1010\bar{1}01$ |
| 6 | *Step* 2-2 | *SWAP* | $10\bar{1}000111$ | $10\bar{1}010000$ | 000000100 | 0000000100 | $1\bar{1}1010\bar{1}01$ | $0\bar{1}100\bar{1}11\bar{1}$ |
| 7 | *Step* 2-3 | $A := DBL(A - B)$ | $000\bar{1}100\bar{1}0$ | $10\bar{1}010000$ | 000000100 | 0000000010 | $001\bar{1}\bar{1}1\bar{1}01$ | $0011\bar{1}1100$ |
| 8 | *Step* 2-3 | $A := DBL(A)$ | $00\bar{1}100\bar{1}00$ | $10\bar{1}010000$ | 000000100 | 0000000001 | $001\bar{1}\bar{1}1\bar{1}01$ | $00011\bar{1}110$ |
| 9 | *Step* 2-1 | *NORM*(*A*) | $0\bar{1}100\bar{1}000$ | $10\bar{1}010000$ | 000001000 | 0000000010 | $0\bar{1}\bar{1}0\bar{1}00\bar{1}1$ | $00011\bar{1}110$ |
| 10 | *Step* 2-1 | *NORM*(*A*) | $\bar{1}100\bar{1}0000$ | $10\bar{1}010000$ | 000010000 | 0000000100 | $\bar{1}1\bar{1}10\bar{1}01\bar{1}$ | $00011\bar{1}110$ |
| 11 | *Step* 2-1 | *NORM*(*A*) | $\bar{1}00\bar{1}00000$ | $10\bar{1}010000$ | 000100000 | 0000001000 | $\bar{1}11\bar{1}10\bar{1}1\bar{1}$ | $00011\bar{1}110$ |
| 12 | *Step* 2-2 | *SWAP* | $10\bar{1}010000$ | $\bar{1}00\bar{1}00000$ | 000100000 | 0000001000 | $00011\bar{1}110$ | $\bar{1}11\bar{1}10\bar{1}1\bar{1}$ |
| 13 | *Step* 2-3 | $A := DBL(A + B)$ | $\bar{1}10\bar{1}00000$ | $\bar{1}00\bar{1}00000$ | 000100000 | 0000000100 | $0110110\bar{1}0$ | 001010100 |
| 14 | *Step* 2-3 | $A := DBL(A)$ | $\bar{1}0\bar{1}000000$ | $\bar{1}00\bar{1}00000$ | 000100000 | 0000000010 | $0110110\bar{1}0$ | 000101010 |
| 15 | *Step* 2-3 | $A := DBL(A - B)$ | $00\bar{1}000000$ | $\bar{1}00\bar{1}00000$ | 000100000 | 0000000001 | $00\bar{1}0\bar{1}0001$ | 000010101 |
| 16 | *Step* 2-1 | *NORM*(*A*) | $0\bar{1}0000000$ | $\bar{1}00\bar{1}00000$ | 001000000 | 0000000010 | $01\bar{1}10\bar{1}11\bar{1}$ | 000010101 |
| 17 | *Step* 2-1 | *NORM*(*A*) | $\bar{1}00000000$ | $\bar{1}00\bar{1}00000$ | 010000000 | 0000000100 | $00\bar{1}0000\bar{1}1$ | 000010101 |
| 18 | *Step* 2-2 | *SWAP* | $\bar{1}00\bar{1}00000$ | $\bar{1}00000000$ | 010000000 | 0000000100 | 000010101 | $00\bar{1}0000\bar{1}1$ |
| 19 | *Step* 2-3 | $A := DBL(A - B)$ | $00\bar{1}000000$ | $\bar{1}00000000$ | 010000000 | 0000000010 | $0\bar{1}0\bar{1}0\bar{1}1\bar{1}1$ | $001100\bar{1}01$ |
| 20 | *Step* 2-3 | $A := DBL(A)$ | $0\bar{1}0000000$ | $\bar{1}00000000$ | 010000000 | 0000000001 | $0\bar{1}0\bar{1}0\bar{1}1\bar{1}1$ | $01011\bar{1}100$ |
| 21 | *Step* 2-1 | *NORM*(*A*) | $\bar{1}00000000$ | $\bar{1}00000000$ | 100000000 | 0000000010 | $0\bar{1}1\bar{1}1001\bar{1}$ | $01011\bar{1}100$ |
| 22 | *Step* 2-2 | *SWAP* | $\bar{1}00000000$ | $\bar{1}00000000$ | 100000000 | 0000000010 | $01011\bar{1}100$ | $0\bar{1}1\bar{1}1001\bar{1}$ |
| 23 | *Step* 2-2 | *MHLV*(*V*) | $\bar{1}00000000$ | $\bar{1}00000000$ | 100000000 | 0000000001 | $01011\bar{1}100$ | $001011\bar{1}10$ |
| 24 | *Step* 3 | $V := -V$ | $\bar{1}00000000$ | $\bar{1}00000000$ | 100000000 | 0000000001 | $01011\bar{1}100$ | $00\bar{1}0\bar{1}\bar{1}1\bar{1}0$ |
| | | $Z$ | | | | | | $00\bar{1}0\bar{1}\bar{1}1\bar{1}0$ |

$Z = [00\bar{1}0\bar{1}\bar{1}1\bar{1}0]_{SD}$ (−86)

**Fig. 1** A modular division by [Algorithm 1].

### 3.3 A Hardware Algorithm for Modular Multiplication/Division

In this subsection, the new hardware algorithm is presented. The algorithm has three modes of operation. In mode=0, the algorithm performs modular division. In mode=1, it performs Montgomery multiplication, and in mode=2, it performs ordinary modular multiplication.

In order to remove time-consuming SD2 to binary conversion in each multiplication/division, the input operands $X$ and $Y$, as well as the output result $Z$ are expressed as (n+1)-digit SD2 numbers in the range $[-M+1, M-1]$. Because of the structural similarity between the main stage of Step 2-3 of [Hardware Algorithm 1], [Algorithm 3] and [Algorithm 4], we implement Montgomery multiplication and the ordinary modular multiplication in this stage. Variable $D$ is used in division mode to indicate the number of digits that the divisor is shifted in order to align its most significant digit to that of the dividend. When performing Montgomery multiplication and ordinary modular multiplication, variable $D$ is used to implement the 'for' loop. In this case, $D$ is required to be an $(n+2)$-bit variable which is initialized to the value of $2^{n+1}$ and shifted to the right at each iteration step until is equal to 1. The variables $A$, $U$ and $V$ are used in both multiplication modes, i.e. $mode = 1$ and $mode = 2$, to store the multiplier, the partial product and the multiplicand respectively.

**[Hardware Algorithm 2]** (Modular Mult./Div.)

```
Inputs: $mode \in \{0, 1, 2\}$
    $M : 2^{n-1} < M < 2^n$ and odd
        (prime when $mode = 0$)
    $X, Y : -M < X, Y < M$
        ($Y \neq 0$ when $mode = 0$)
Output: $Z : Z \equiv X/Y \pmod{M}$   ($if\ mode = 0$)
    $Z \equiv X \cdot Y \cdot 2^{-n} \pmod{M}$   ($if\ mode = 1$)
    $Z \equiv X \cdot Y \pmod{M}$   ($if\ mode = 2$)
        and $-M < Z < M$
Algorithm:
  Step 1:
    $A := Y$; $B := M$; $P := 1$; $M := M$;
    if $mode = 0$ then
      $U := X$; $V := 0$; $D := 1$;
    else
      $U := 0$; $V := X$; $D := 2^{n+1}$;
      goto Step 2-3;
    endif
  Step 2:
    Step 2-1: [Normalization of A]
      while $p_n = 0$ and $[a_n a_{n-1}] \neq [10]$
          and $[a_n a_{n-1}] \neq [\bar{1}0]$ do
        if $[a_n a_{n-1} a_{n-2}] = [1\bar{1}1]$ or
            $[a_n a_{n-1} a_{n-2}] = [011]$ then
              $[a_n a_{n-1} a_{n-2}] := [10\bar{1}]$;
        elseif $[a_n a_{n-1} a_{n-2}] = [\bar{1}1\bar{1}]$ or
            $[a_n a_{n-1} a_{n-2}] = [0\bar{1}\bar{1}]$ then
              $[a_n a_{n-1} a_{n-2}] := [\bar{1}01]$;
        else
```

```
            $A := DBL(A)$; $P := 2 \cdot P$; $D := 2 \cdot D$;
            $U := MDBL(U, M)$;
          **endif**
        **endwhile**
      **Step 2-2:** [Swapping]
        $SWAP(A, B)$; $SWAP(U, V)$;
        **if** $p_n = 1$ **then**
          **while** $d_0 = 0$ **do**
            $D := D/2$; $V := MHLV(V, M)$;
          **endwhile**
          **goto Step 3**;
        **endif**
      **Step 2-3:** [Multiplication/Integer Division]
        $/ * \ Main\ Stage\ (MUL/DIV) * /$
        **while** $d_0 = 0$ **do**
          **if** $a_n = 0$ **or** $a_{n-1} = -a_n$ **then** $S := A$;
          **else**
            **if** $mode = 0$ **then**
              $q := a_n \cdot b_n$;
              $S := A - q \cdot B$;
            **else**
              $q := -a_n$;
              $S := A$;
              **if** $mode = 1$ **then** $s_n := 0$;
            **endif**
            $U := MADD(U, -q \cdot V, M)$;
          **endif**
          **if** ($s_n = 0$ **or** $s_{n-1} = -s_n$) **then**
            $A := DBL(S)$; $D := D/2$;
            **if** $mode = 2$ **and** $d_0 = 0$ **then**
              $U := MDBL(U, M)$;
            **else** $V := MHLV(V, M)$;
          **else**
            **if** $mode = 2$ **then** $s_n := 0$;
            $A := S$;
          **endif**
        **endwhile**
        **if** $mode \neq 0$ **then goto Step 4**;
        $/ * \ Termination\ Stage\ (DIV) * /$
        $r := sgn([a_n a_{n-1}])$;
        **while** $sgn([a_n a_{n-1}] = r$ **and**
          $(abs([a_n a_{n-1} a_{n-2}]) \geq 3$ **or**
          $(b_{n-2} = -b_n$ **and** $abs([a_n a_{n-1} a_{n-2}]) = 2))$
        **do**
          $q := r \cdot b_n$;
          $A := A - q \cdot B$;
          $U := MADD(U, -q \cdot V, M)$;
        **endwhile**
        **goto Step 2-1**;
    **Step 3:** [Correction]
      **if** $b_n = \bar{1}$ **then** $V := -V$;
    **Step 4:**
      **if** $mode = 0$ **then** $Z := V$;
      **else** $Z := U$;
```

The hardware algorithm is divided in 4 steps. Initialization of variables takes place in Step 1. In division mode, i.e. mode= 0, variables $U$ and $V$ are initialized to $X$ and 0 respectively whereas in the multiplication mode, i.e. mode=1 and 2, they are initialized to 0 and $X$. The core of the algorithm is described in Step 2. It follows the same structure of [Hardware Algorithm 1]. Only Step 2-3 is modified to include the calculations of Montgomery multiplication and ordinary modular multiplication. A correction is performed in Step 3, and in Step 4 the output result is selected. In modular division, the output is $V$ and in multiplication modes, the output is $U$.

We will now describe in detail how the hardware algorithm works for calculating Montgomery multiplication and the ordinary modular multiplication. To clarify the explanation, we present the portion of codes used only for these operations from the main stage of Step 2-3.

```
[Montgomery Mult./ Ord. Mod. Mult.]
  **while** $d_0 = 0$ **do**
    **if** $a_n = 0$ **or** $a_{n-1} = -a_n$ **then** $S := A$;
    **else**
      $q := -a_n$;
      $S := A$;
      **if** $mode = 1$ **then** $s_n := 0$;
      $U := MADD(U, -q \cdot V, M)$;
    **endif**
    **if** ($s_n = 0$ **or** $s_{n-1} = -s_n$) **then**
      $A := DBL(S)$; $D := D/2$;
      **if** $mode = 2$ **and** $d_0 = 0$ **then**
        $U := MDBL(U, M)$;
      **else** $V := MHLV(V, M)$;
    **else**
      **if** $mode = 2$ **then** $s_n := 0$;
      $A := S$;
    **endif**
  **endwhile**
```

Firstly we explain how the algorithm computes Montgomery multiplication. When $a_n = 0$, we set the temporary variable $S$ to $A$ and perform $A := DBL(S)$ in order to shift the multiplier to the left. Nothing is added to $U$. When $[a_n a_{n-1}] = 1\bar{1}$ or $\bar{1}1$, we perform the same operations since they represent a 0 at the most significant digit of $A$. When $a_n = 1$ and $[a_n a_{n-1}] \neq 1\bar{1}$ nor $\bar{1}1$, we add the multiplicand $X$ (divided several times by 2) stored in $V$ to $U$, and then, in order to throw this digit away, we set the temporary variable $S$ to $A$ and set the digit $s_n$ to 0 so that operation $A := DBL(S)$ is performed. In the same way, when $a_n = \bar{1}$ and $[a_n a_{n-1}] \neq 1\bar{1}$ nor $\bar{1}1$, we perform the same operations except that we subtract the multiplicand $X$ stored in $V$ from $U$. In all these cases, $V := MHLV(V, M)$ is also performed and $D$ is shifted to the right by one position indicating that one digit of the multiplier is processed. The 'while' loop continues until $d_0 = 1$. At this point, the $n + 1$ digits of the multiplier are processed and the Montgomery constant equals to the value of $2^n$.

Next we will describe how the algorithm works for calculating ordinary modular multiplication. Since variable $U$, which stores the partial product, is initialized to 0, instead of doubling $U$ and then adding the multiplicand, we proceed in reverse order. For the case $a_n = 0$ or $[a_n a_{n-1}] = 1\bar{1}$ or $\bar{1}1$, temporary variable $S$ is set to $A$ in order to double $A$ by means of $DBL(S)$ and $U$ by $MDBL(U, M)$. For the cases $[a_n a_{n-1}] = 11$ or $10$ or $\bar{1}\bar{1}$ or $\bar{1}0$, we need to perform the operations $U := MADD(U, -q \cdot V, M)$ and $U := MDBL(U, M)$. However, since the modular reduction hardware component are shared for both operations to reduce hardware resources, these operations can never be executed at the same iteration. Therefore, we split the calculation of $MDBL(U, M)$ and $MADD(U, -q \cdot V, M)$ into two different iteration steps using simple control signals. For this, we set the temporary variable $S$ with the value of $A$ and perform $U := MADD(U, -q \cdot V, M)$ depending on the value of $a_n$. Note that in this case, $s_n \neq 0$ and $[s_n s_{n-1}] \neq 1\bar{1}$ nor $\bar{1}1$, therefore no action is performed to $A$, $D$ and $V$. At the end of the loop we leave the most significant position of $A$ in 0. In this way, in the next iteration, $A$ is shifted to the left by $DBL(A)$ and $U$ is doubled in modulo $M$ by $MDBL(U, M)$. Since we reverse the order of doubling the partial product and adding the multiplicand, $MDBL(U, M)$ is not performed in the last iteration.

## 4. A Multiplier/Divider for Modular Arithmetic

An $n$-bit modular multiplier/divider based on [Hardware Algorithm 2] consists of seven registers for storing $A$, $B$, $P$, $D$, $U$, $V$ and $M$, and a combinational circuit part. A block diagram of the modular multiplier/divider is given in Fig. 2.

We assume that each iteration of the ‘while’ loops are performed in one clock cycle. Thus, in Step 2-1, operations

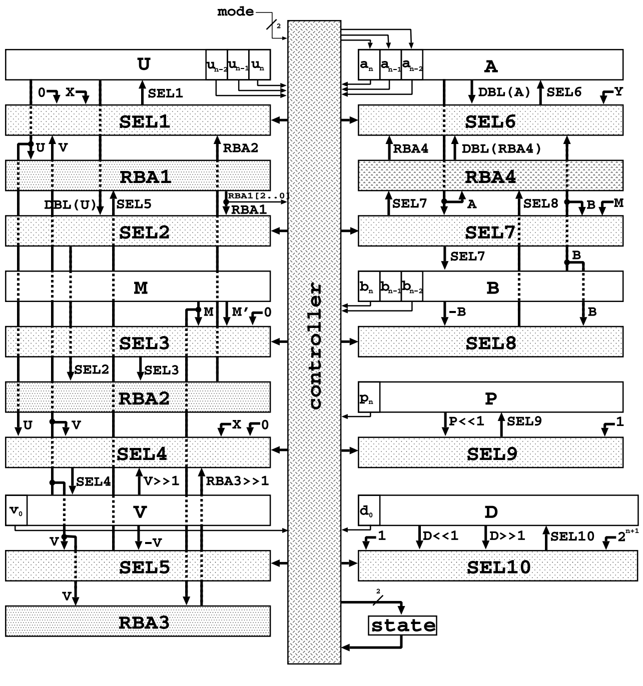

Fig. 2 Block diagram of a multiplier/divider.

$DBL(A)$, one-bit-shift of $P$ and $D$ and $MDBL(U, M)$ are performed in one clock cycle. In the ‘while’ loop of Step 2-2, one-bit-shift of $D$ and $MHLV(V, M)$ are performed in one clock cycle. In Step 2-3, in the main stage, $A := DBL(A)$ or $A := A - q \cdot B$ or $A := DBL(A - q \cdot B)$ is executed in one clock cycle. For modular arithmetic, operations $U := MADD(U, -q \cdot V, M)$ and $V := MHLV(V, M)$ or the operation $U := MDBL(U, M)$ are executed in one clock cycle. In the termination stage, operations $A := A - q \cdot B$ and $U := MADD(U, -q \cdot V, M)$ are executed in one clock cycle. Additionally, initialization step, i.e., Step 1; the swapping, i.e. Step 2-2; the correction step, i.e., Step 3; and the selection of the output, i.e. Step 4; takes one clock cycle each.

The combinational circuit part of the multiplier/divider (for Steps 2 and 3) mainly consists of an SD2 adder (with an operand negator), a modular adder (with an operand negator), a modular halving circuit, an SD2 negator and selectors. The modular adder consists of two SD2 adders where one of these is simpler. The modular doubling circuit and the modular halving circuit consist of simpler SD2 adders.

The depth of the combinational circuit part is a constant independent of $n$, and therefore, the length of the clock cycle is constant independent of $n$. The modular multiplier/divider has a bit-slice structure and is suitable for VLSI implementation.

The amount of hardware of the modular multiplier/divider is proportional to $n$. The hardware components used in this architecture are practically the same as those needed in the construction of the individual modular divider. Only a slight increase of the hardware is required to enable the computation of Montgomery multiplication and the ordinary modular multiplication. This little extra hardware is due to the selectors of Step 1 and Step 4 and the additional logic gates to select the different operations in Step 2-3.

In division mode, we can show from the discussion in [10] that in Step 2-3, no two successive clock cycles are executed without doubling $A$ ($DBL(S)$) in the main stage, and no more than three cycles are executed in the termination stage. We can also show that if $DBL(A)$ is not performed in an execution of Step 2-1 (normalization of $A$), $DBL(A)$ must be performed in the next execution of Step 2-1. Hence, the number of clock cycles executed in Step 2 is at least $2n + 1$ and at most about $3n$ depending on the operands.

Montgomery multiplication is performed in Step 2-3 in exactly $n + 1$ clock cycles. Ordinary modular multiplication is performed in at least $n + 1$ and at most $2n + 1$ clock cycles. This varies with the multiplier.

## 5. Concluding Remarks

We have proposed a hardware algorithm for modular multiplication/division. We first improved the hardware algorithm for modular division proposed in [10] to reduce hardware requirements. Then, we modified the Montgomery multiplication algorithm and the ordinary modular multiplication algorithm to accelerate them by the use of the SD2 redundant representation for internal computation and enabled these operations to share almost all the hardware components with those required for computing a single modular division.

Although two operations are available for calculating modular multiplication, Montgomery multiplication has an outstanding performance when long chained multiplications are required. Therefore, in applications such as in modular exponentiation, multiplications can be performed in the Montgomery representation to accelerate the calculations. The inclusion of the ordinary modular multiplication in the same hardware and the expansion of the inputs in one digit position allows for performing not only a few single modular multiplications but also for transforming the operands into the Montgomery domain by multiplying the operand with the value of $2^n$ without the need for any precomputed constants.

Modular divisions can be jointly used with Montgomery multiplications to accelerate modular exponentiation. The classical methods consist of decomposing the calculation of modular exponentiation as a series of modular multiplications. By representing the exponent as an SD2 number, it is possible to decompose the calculation into either a series of modular multiplications or a mixture of multiplications and divisions. As the number of multiplications and divisions are proportional to the weight of the exponent, acceleration can be accomplished by representing the exponent as a minimum weight SD2 number. Finally, it is worth noting that these operations can still be computed using the Montgomery representation as described in [5].

**Acknowledgments**

The authors would like to thank Associate Professor Kazuyoshi Takagi for his valuable comments and discussions and to Miss Grith Christensen for her help in preparing this paper.

## References

[1] ANSI X9.30. Public Key Cryptography for the Financial Services Industry: Part 1: The Digital Signature Algorithm (DSA), American National Standard Institute, American Bankers Association, 1997.
[2] A. Avizienis, “Signed-digit number representations for fast parallel arithmetic,” IRE Trans. Electronic Comput., vol.EC-10, pp.389–400, 1961.
[3] W. Diffie and M.E. Hellman, “New directions in cryptography,” IEEE Trans. Inf. Theory, vol.IT-22, no.11, pp.644–654, Nov. 1976.
[4] T. ElGamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” IEEE Trans. Inf. Theory, vol.IT-31, no.4, pp.469–472, July 1985.
[5] M.E. Kaihara and N. Takagi, “A VLSI algorithm for modular multiplication/division,” Proc. 16th IEEE Symp. Comp. Arith., pp.220–227, Santiago de Compostela, Spain, June 2003.
[6] D.E. Knuth, The Art of Computing Programming, vol.2, Seminumerical Algorithms, third ed., Addison-Wesley, Reading, MA, 1998.
[7] N. Koblitz, “Elliptic curve cryptosystems,” Math. Comput., vol.48, no.177, pp.203–209, Jan. 1987.
[8] P.L. Montgomery, “Modular multiplication without trial division,” Math. Comput., vol.44, no.170, pp.519–521, April 1985.
[9] R.L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” Commun. ACM, vol.21, no.2, pp.120–126, Feb. 1978.
[10] N. Takagi, “A hardware algorithm for modular division based on the extended Euclidean algorithm,” IEICE Trans. Inf. & Syst., vol.E79-D, no.11, pp.1518–1522, Nov. 1996.
[11] N. Takagi and S. Yajima, “Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem,” IEEE Trans. Comput., vol.41, no.7, pp.887–891, July 1992.

**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** received the B.E., M.E., and Ph.D. 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 the Sakai Memorial Award of the Information Processing Society of Japan in 1995.