数理解析研究所講究録 1375 巻 2004 年 201-207

A Multiplication/Division VLSI Algorithm for Modular Arithmetic

名古屋大学情報科学研究科
貝原 マルセロ (Marcelo E. Kaihara) 高木 直史 (Naofumi Takagi)
Department of Information Engineering,
Nagoya University

Abstract

A multiplication/division VLSI algorithm for modular arithmetic is proposed. The algorithm performs modular division, Montgomery's modular multiplication and ordinary modular multiplication. Modular division is based on the extended Euclidean algorithm (EEA). Montgomery's modular multiplication is performed using a new method consisting of processing the multiplier from the most significant digit first. Ordinary modular multiplication is performed using the conventional doubling and adding procedures. The algorithm enables the three operations to share almost all hardware components reducing considerably hardware requirements. It carries out these 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. 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. A modular multiplier/divider based on the algorithm has a linear array structure with a bit-slice feature and is suitable for VLSI implementation.

1 Introduction

Modular multiplication and modular division are basic operations in abstract algebra. They play important roles in processing many public-key crytopsystems. For example, they are used in RSA[1] and ElGamal [2] cryptosystems, in the Diffie-Hellman key exchange protocol [3] and in the DSA digital signature scheme [4]. They can also be used to accelerate the exponentiation operation using addition-subtraction chains [5] and to compute point operations in ECC with curves defined over $GF(p)$ [6]. Considering also the demand in technology to shrink hardware and to increase computation capacity at the same time, it is important to develop an algorithm for calculating modular multiplication/division that can be implemented with reduced hardware requirements.

In this paper, we propose an algorithm for modular multiplication/division for a large modulus suitable for VLSI implementation. Modular division is based on the extended Euclidean algorithm. We have improved the hardware algorithm proposed in [7] to reduce hardware requirements. We further modified it so that it can perform Montgomery's modular multiplication as well as ordinary modular multiplication without increasing hardware requirements considerably. Montgomery's modular multiplication is performed using a new method, which consists of processing the multiplier from the most significant digit first while the multiplicand is halved each time using modular arithmetic. Ordinary modular multiplication is performed with a conventional multiplication algorithm, which is based on doubling and adding procedures. The multiplier is scanned from the most significant position first. The intermediate result is doubled and the multiplicand is added to the partial product every time the scanned digit of the multiplier has the value of one.

A modular multiplier/divider based on the 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/division in $O(n)$ clock cycles where the length of clock cycle is constant independent of n.

In the next section, we will explain the extended Euclidean algorithm, the ordinary multiplication algorithm and Montgomery's multiplication algorithm. In Section 3, we will look at the redundant representation number system and procedures for basic operations in this number system. In Section 4, we propose a hardware algorithm for modular multiplication/division. In Section 5, we will explain the hardware implementation and design. In Section 6, we present concluding remarks.

2 Preliminaries

2.1 Extended Euclidean Algorithm for Modular Division

Extended Euclidean Algorithm [7] 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 $Z(< M)$ such that $Z \equiv X/Y \pmod{M}$ (the algorithm also works with an odd not prime M and Y relatively prime to M).

It performs modular division by intertwining a procedure for finding the modular quotient with that for calculating $\gcd(Y, M)$.

[Algorithm 1]
(Extended Euclidean Algorithm)
Function: Modular Division
Inputs: M: $2^{n-1} < M < 2^n$
X, Y: $0 \leq X < M$, $0 < Y < M$
Output: $Z = X/Y \bmod M$
Algorithm:
$A := M; B := Y; U := 0; V := X;$
while $B \neq 1$ **do**
Choose Q so that $|A - B \cdot Q| < |B|$;
$A' := A - B \cdot Q$;
Calculate U' which satisfies
$U' \equiv U - V \cdot Q \pmod{M}$ and $|U'| < M$;
$A := B; B := A'$;
$U := V; V := U'$;
endwhile
if $B = -1$ **then** $Z' := -V$; **else** $Z' := V$; **endif**
if $Z' < 0$ **then** $Z := Z' + M$ **else** $Z := Z'$; **endif**
output Z as the result;

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

2.2 Modular Multiplication

2.2.1 Ordinary Modular Multiplication

Consider the residue class ring of integers with an odd modulus M. Let X and Y be elements of the ring. Modular multiplication is defined as Z such that $0 \leq Z < M$ and $Z \equiv XY \pmod{M}$. In ordinary modular multiplication method, the digits of the multiplier are scanned from the most significant position. For each digit that is processed, the partial product is doubled. If the scanned digit has the value of one, the multiplicand is then added to the partial product, otherwise, none is done. The multiplier is then shifted one position to the left to allow the next digit to be scanned. The multiplication algorithm is described below. Note that A is n-digits long and the most significant one is represented as a_{n-1}.

[Algorithm 2]
(Modular Multiplication Algorithm)
Function: Modular Multiplication
Inputs: $M : 2^{n-1} < M < 2^n$
$X, Y : 0 \leq X, Y < M$
Output: $Z = XY \bmod M$
Algorithm:
$A := Y; U := 0; V := X;$
for $i := 1$ **to** n **do**
$U := 2 \cdot U \bmod M$;
$q := a_{n-1}$;
$A := A << 1; U := (U + qV) \bmod M$;
end for
$Z := U$;
output Z as the result;

2.2.2 Montgomery's Modular Multiplication

Montgomery introduced an efficient algorithm for calculating modular multiplication [8]. Consider the residue class ring of integers with an odd modulus M. Let X and Y be elements of the ring. Montgomery's modular multiplication algorithm calculates $Z(< M)$ such that $Z \equiv XYW^{-1} \pmod{M}$ where W is an arbitrary constant relatively prime to M. The value of W is usually set to 2^n when the calculations are performed in radix-2 with an n-bit modulus M.

The radix-2 Montgomery's multiplication algorithm is described below.

[Algorithm 3]
(Montgomery's Multiplication Algorithm)
Function: Montgomery's Modular Multiplication
Inputs: $M : 2^{n-1} < M < 2^n$
$X, Y : 0 \leq X, Y < M$
Output: $Z = XY2^{-n} \bmod M$
Algorithm:
$A := Y; U := 0; V := X;$
for $i := 1$ **to** n **do**
if $A \bmod 2 = 0$ **then** $q := 0$;
else $q := 1$; **end if**
$A := (A - q)/2; U := (U + qV)/2 \bmod M$;
end for
if $U \geq M$ **then** $Z := U - M$;
else $Z := U$; **end if**
output Z as the result;

Note that U is always bounded by $2M$ throughout all iterations. Therefore, the last correction step assures that the output is correctly expressed in modulo

M.

3 Use of a Redundant Representation

The radix-2 signed-digit representation (SD2) has a fixed radix 2 and a 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 $\sum_{i=0}^{n-1} a_i \cdot 2^i$. We can perform addition of two SD2 numbers without carry propagation. For the details of carry-propagation-free addition, see, e.g., [10]. We can get a negation of an SD2 number by changing the signs of all nonzero digits in it.

In the algorithm to be proposed, input operands X and Y as well as the output result Z are assumed to be n-digit radix-2 signed-digit (SD2) integers in the range $[-M+1, M-1]$. Intermediate results are also represented in the SD2 representation.

The algorithm requires a doubling procedure for a SD2 integer without overflow. Let A and B be n-digit SD2 integers. Assume A satisfies $a_{n-1} = 0$ or $a_{n-2} = -a_{n-1}$. A doubling $B := DBL(A)$, i.e., the calculation of B such that $B = 2 \cdot A$ is performed as follows. When $a_{n-1} = 0$, $B = [a_{n-2}a_{n-3}a_{n-4} \cdots a_1a_00]$ and otherwise ($a_{n-2} = -a_{n-1}$), $B = [a_{n-1}a_{n-3}a_{n-4} \cdots a_1a_00]$.

Procedures for addition, doubling and halving modulo M in the SD2 system are also required and are described next.

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 such that $T \equiv U + V \pmod{M}$, is performed through two steps. In the first step, we calculate $S := U + V$ in the SD2 system. S is an $(n+2)$-digit SD2 number. In the second step, we add M or 0 or M' to S, accordingly as the value of the number formed by the three most significant digits of S, i.e. the value of $[s_{n+1}s_ns_{n-1}]$, is negative or zero or positive. $M' = [\bar{1}0m'_{n-2}...m'_1 1]$ is the $(n+1)$-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. For the details of the modular addition procedure, see [10].

Modular doubling $T := MDBL\,(U, M)$, i.e., the calculation of T such 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 by one position to the left.

Modular halving $T := MHLV(V, M)$, i.e., the calculation of T such 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 $v_0 \neq 0$. Nothing is performed when V is even. In the second step, we shift the result of the first step by 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.

4 A Hardware Algorithm for Modular Multiplication/Division

We propose a hardware algorithm that performs modular division, Montgomery's modular multiplication and ordinary modular multiplication which is efficient in hardware requirements.

4.1 Division Mode

A hardware algorithm based on the extended Euclidean algorithm presented in the previous section was proposed in [7]. We modified it to reduce hardware requirements and to make it easy to merge with the multiplication algorithms. This modification also simplify the initial normalization procedure.

In Step 1 of the proposing algorithm, initialization of variables takes place. In division mode, A, B, U and V are initialized to the values of Y, M, X and 0 respectively. P and D are two n-bit binary numbers of the form 2^i. They are initialized to the value of 1. At the beginning of each iteration of Step 2, $D = 1$ and $b_{n-1} \neq 0$. If we denote with $\mathcal{A}$, $\mathcal{B}$, $\mathcal{U}$ and $\mathcal{V}$ the represented values in binary of the numbers stored in variables A, B, U and V respectively, $A = \mathcal{A} \cdot 2^{n-k}$, $B = \mathcal{B} \cdot 2^{n-k}$ and $P = 2^{n-k}$. $U = \mathcal{U}$ and $V = \mathcal{V}$ when $\mathcal{B}$ is effectively k-bit long.

At each iteration of Step 2, we first strongly normalize A which stores the divisor during Step 2-1. A is strongly normalized if $a_{n-1} \neq 0$ and $a_{n-2} = 0$, i.e., $[a_{n-1}a_{n-2}] = [10]$ or $[\bar{1}0]$. During the normalization, A is doubled several times by means of DBL shown in Section 3. At the same time that A is doubled, D and P are also doubled, and U is doubled modulo M by means of $MDBL$ shown in Section 3. In Step 2-2, A and B, and U and V are swapped respectively. At this time, $A = \mathcal{A} \cdot (P/D)$, $B = \mathcal{B} \cdot P$ and $V \equiv \mathcal{V} \cdot D \pmod{M}$.

Then, we perform an integer division. We produce Q as a sequence of q's where $q \in \{-1, 1\}$ and $Q = \sum q \cdot D$. Note that D is updated during the integer division. As the production of q, we calculate $A - q \cdot B$ in the SD2 system without carry propagation ($A - q \cdot B$ corresponds to $\mathcal{A} - (q \cdot D) \cdot \mathcal{B}$). Concurrently, we calculate $U - q \cdot V$ modulo M in the SD2 system by means of $MADD$ shown in Section 3. During the integer division, we double A several times by means of DBL. When we double A, we halve D, and halve V modulo M by means of $MHLV$ shown in Section 3. After the integer division, $D = 1$, $A = \mathcal{A}' \cdot P$, $B = \mathcal{B} \cdot P$ ($b_{n-1} \neq 0$), $U = \mathcal{U}'$ and $V = \mathcal{V}$.

Note that in the algorithm proposed in [7], the values of A and B, and U and V are initialized respectively interchanged. In the algorithm proposed in [7], B is normalized. During the procedure, B is doubled and V is doubled modulo M. After the integer division, variables A and B, and variables U and V are swapped respectively. In the proposing algorithm, A is normalized. During the procedure, A is doubled and U is doubled modulo M. In this way, first, we eliminate the hardware component for doubling B. Second, we enable to reuse the hardware component for modular reduction needed for $MDBL(U, M)$ in the calculation of $MADD(U, -q \cdot V, M)$ eliminating the need of extra hardware. And finally, we eliminate the swap operation performed in [7] for the special case that B cannot be normalized. Hence, the proposing algorithm is much more efficient in terms of hardware requirements than that of [7].

4.2 Multiplication Mode

The proposing algorithm performs both Montgomery's modular multiplication and ordinary modular multiplication. In the proposing algorithm, A and V are initialized to the values of the multiplier Y and the multiplicand X respectively. U is used to store the partial product of the multiplication.

In Section 2.2.2, we described Algorithm 3 which performs Montgomery's modular multiplication. It examines the least significant bit of A to determine whether to add or not V to U. The result is then divided by 2 using modular arithmetic and A is shifted one position to the left.

In order to implement Montgomery's multiplication operation using the same hardware components required by the division mode, we introduce SD2 representation in operands, internal calculation and the output result and we examine the multiplier from the most significant position first, i.e. a_{n-1}. For a nonzero value of the digit, we add or subtract the multiplicand X stored in V to U depending on the sign of it. If it is positive, we add. Otherwise we subtract. Then, instead of dividing the partial product U by 2 modulo M, we divide the multiplicand V by 2 modulo M. For this operation we use $MHLV(V, M)$ described in Section 3. If the value of the digit a_{n-1} is 0, we perform a $DBL(A)$ to shift to the left the multiplier and $MHLV(V, M)$ to divide the multiplicand by 2 modulo M. The same operations are performed for the cases that $[a_{n-1}a_{n-2}] = 1\bar{1}$ or $\bar{1}1$ since they represent a 0 at the most significant digit of A. The 'for' loop is implemented using variable D which is initialized with 2^n and it is shifted to the right at each iteration step until it is equal to 1.

We also implement the ordinary modular multiplication described in Section 2.2.1. We also perform the operations in SD2 representation. Since variable U, which stores the partial product, is initialized with 0, instead of doubling U and then adding the multiplicand, we proceed in the reverse order. For the case that $a_{n-1} = 0$ or $[a_{n-1}a_{n-2}] = 1\bar{1}$ or $1\bar{1}$, we just double A by means of $DBL(A)$ and U by $MDBL(U, M)$. For the cases that $[a_{n-1}a_{n-2}] = 11$ or 10 or $\bar{1}\bar{1}$ or $\bar{1}0$, in order to share hardware components without increasing the calculation delay, we split the calculation of $MDBL(U, M)$ and $MADD(U, -q \cdot V, M)$ into two different iteration steps. We first perform $U := MADD(U, -q \cdot V, M)$ depending on the value of a_{n-1} and we leave the most significant position of A in 0. By doing so, A is shifted to the left by $DBL(A)$ and U is doubled modulo M by $MDBL(U, M)$ in the next iteration. Since we reverse the order of doubling the partial product and adding the multiplicand, $MDBL(U, M)$ is not performed in the last iteration.

4.3 The Hardware Algorithm

The hardware algorithm is presented here. It is divided in 4 steps. Initialization of variables takes place in Step 1. The core of the algorithm is described in Step 2. A correction is performed in Step 3, and in Step 4 the output result is selected.

[Algorithm 4]
(A VLSI Algorithm for modular multiplication and modular division)

Function: Modular Multiplication and Modular Division
Inputs: $M : 2^{n-1} < M < 2^n$
$X, Y : -M < X, Y < M$
Output: $mode = 0 : Z \equiv X/Y \bmod M$
$mode = 1 : Z \equiv XY2^{-n} \bmod M$
$mode = 2 : Z \equiv XY \bmod M$
Algorithm:
Step 1:
$A := Y; B := M; P := 1; M := M;$
if $mode = 0$ **then**
$D := 1; U := X; V := 0;$
else
$D := 2^n; U := 0; V := X;$
goto Step 2-4;
endif
Step 2:
Step 2-0:
if $[a_{n-1}a_{n-2}] = [11]$ **or** $[\bar{1}\bar{1}]$ **then**
$q := a_{n-1} \cdot b_{n-1};$

```
    A := A − q · B;
    U := MADD(U, −q · V, M);
  endif
Step 2-1:
  while p_{n−1} = 0 and
    [a_{n−1}a_{n−2}] ≠ [10] and [a_{n−1}a_{n−2}] ≠ [1̄0];
    if [a_{n−1}a_{n−2}a_{n−3}] = [11̄1] or
      [a_{n−1}a_{n−2}a_{n−3}] = [011] then
      [a_{n−1}a_{n−2}a_{n−3}] := [101̄];
    elseif [a_{n−1}a_{n−2}a_{n−3}] = [1̄11̄] or
      [a_{n−1}a_{n−2}a_{n−3}] = [01̄1̄] then
      [a_{n−1}a_{n−2}a_{n−3}] := [1̄01];
    else
      A = DBL(A); P := 2 · P; D := 2 · D;
      U = MDBL(U, M);
    endif
  endwhile
Step 2-2:
  T := A; A := B; B := T;
  T := U; U := V; V := T;
Step 2-3:
  if p_{n−1} = 1 then
    while d_0 = 0 do
      D := D/2; V := MHLV(V, M);
    endwhile
    goto Step 3;
  endif
Step 2-4:
  / * Main Stage (MUL/DIV) * /
  while d_0 = 0 do
    if a_{n−1} = 0 or a_{n−2} = −a_{n−1} then
      S := A;
    else
      if mode = 0 then
        q := a_{n−1} · b_{n−1};
        S := A − q · B;
      else
        q := −a_{n−1};
        S := A;
        if mode = 1 then s_{n−1} := 0;
      endif
      U := MADD(U, −q · V, M);
    endif
    if (s_{n−1} = 0 or s_{n−2} = −s_{n−1}) then
      A := DBL(S); D := D/2;
      if mode = 2 and d_0 = 0 then
        U := MDBL(U, M);
      else
        V := MHLV(V, M);
      endif
    else
      if mode = 2 then s_{n−1} := 0;
      A := S;
    endif
  endwhile
  if mode ≠ 0 then goto Step 4;
  / * Termination Stage (DIV) * /
  r := sgn([a_{n−1}a_{n−2}]);
  while sgn([a_{n−1}a_{n−2}] = r and
    (abs([a_{n−1}a_{n−2}a_{n−3}]) ≥ 3 or
      (b_{n−3} = −b_{n−1} and
      abs([a_{n−1}a_{n−2}a_{n−3}] = 2))
  do
    q := r · b_{n−1};
    A := A − q · B;
    U := MADD(U, −q · V, M);
  endwhile
  goto Step 2-1;
Step 3:
  if b_{n−1} = 1̄ then V := −V;
Step 4:
  if mode = 0 then Z := V;
  else Z := U;
  output Z as the result;
```

$sgn([a_{n-1}a_{n-2}])$ is -1 or 0 or 1, accordingly as the value of $[a_{n-1}a_{n-2}]$ is negative or zero or positive. $abs([a_{n-1}a_{n-2}a_{n-3}])$ means $|4a_{n-1}+2a_{n-2}+a_{n-3}|$.

Step 2-0 is executed only when the most significant two bits of A, i.e., the divisor Y, have both the value of 1. Note that when its most significant two bits have both value of 1, A cannot be strongly normalized. We perform the subtraction of $A - B$ in the SD2 system using an ordinary SD2 addition rule, e.g., [10]. Since the most significant two bits of B, i.e., the modulus M, have also the value of 1 in this case, the most significant digit of the updated A has as result the value of 0.

In Step 2-1, we strongly normalize A. At the beginning of this stage, $[a_{n-1}a_{n-2}] \neq [11]$ nor $[\bar{1}\bar{1}]$, from Step 2-0 and the SD2 addition rule in the termination stage shown below. It never become [11] nor $[\bar{1}\bar{1}]$ during this stage, because we rewrite $[1\bar{1}1]$ and [011] to $[10\bar{1}]$ and $[\bar{1}1\bar{1}]$ and $[0\bar{1}\bar{1}]$ to $[\bar{1}01]$. In Step 2-2, we swap A and B and also U and V. In Step 2-3, when p_{n-1} becomes 1, i.e., P becomes 2^{n-1}, it means that $|B|$ has become 1. Then, we divide V by D modulo M by means of $MHLV$, because V has been multiplied by D modulo M. Then we terminate Step 2.

In Step 2-2, we perform an integer division, a Montgomery's modular multiplication or an ordinary modular multiplication. During the integer division, we perform the subtraction of $A - q \cdot B$ in the SD2 system. In this subtraction, we use the special addition rule shown in Table 1 at the most significant two positions. Note that when an addition is performed, $[a_{n-1}a_{n-2}]$ is not [00]. Furthermore, in the main stage, when an addition is performed, $[a_{n-1}a_{n-2}]$ is not $[\bar{1}1]$ nor $[0\bar{1}]$ nor [01] nor $[1\bar{1}]$. (The rule for these cases in the table is for the addition in the termination stage.) Note also that $[b_{n-1}b_{n-2}]$ is [10] or $[\bar{1}0]$ and that $q = sgn([a_{n-1}a_{n-2}]) \cdot b_{n-1}$ ($= an - 1 \cdot b_{n-1}$ in the main stage). In the table, c_{n-3} is the intermediate

Table 1. A special SD2 addition rule

	$s_{n-1}s_{n-2}$					
	$a_{n-1}a_{n-2}$					
c_{n-3}	$\bar{1}\bar{1}$	$\bar{1}0$	$\bar{1}1,0\bar{1}$	$01,1\bar{1}$	10	11
$\bar{1}$	$\bar{1}0$	$0\bar{1}$	00	$\bar{1}0$	$0\bar{1}$	00
0	$0\bar{1}$	00	01	$0\bar{1}$	00	01
1	00	01	10	00	01	10

carry from the third significant position. We use an ordinary SD2 addition rule, e.g., that in [10], at the other positions. Note that $[s_{n-1}s_{n-2}]$ never become [11] nor $[\bar{1}\bar{1}]$.

Because of the strongly normalization of A, the computation in this step is simple. We can show that in the main stage, no two successive SD2 additions are performed without doubling A $(DBL(S))$.

In the termination stage, we use a bit complicated condition for termination, in order to avoid the situation that the final $|A|$ $(|\mathcal{A}'|)$ is very near to $|B|$ $(|\mathcal{B}|)$. Note that this situation makes the convergence of the whole computation very slow. By the use of the complicated condition, we can make the final $|A|$ significantly smaller than $|B|$. Hence, we can guarantee that when A is not doubled in an execution of Step 2-1, the updated A must be doubled in the next execution of Step 2-1. We can show that no more than three SD2 additions are performed in the termination stage. Note that the final $[a_{n-1}a_{n-2}]$ is not [11] nor $[\bar{1}\bar{1}]$, from the SD2 addition rule.

In Step 3, when $b_{n-1} = \bar{1}$, we negate V in the SD2 system.

In Step 4, we select the output depending on the mode of operation.

5 Hardware Implementation

An n-bit modular multiplier/divider based on Algorithm 4 consists of seven registers for storing A, B, P, D, U, V and M, and a combinational circuit part. P and D can be implemented with 1-hot counters. Further improvements in circuit area can be accomplished by using two-level 1-hot counters or by using binary counters. For details, see [11].

We assume that we perform each of Step 2-0, or one iteration of Step 2-1,or one iteration of Step 2-2, or one iteration of Step 2-3, or one iteration of main or termination stage of Step 2-4, or Step 3 in one clock cycle. Then, in one clock cycle, the modular divider mainly performs a SD2 addition of $A - B$ and $MADD(U, -V, M)$ in Step 2-0, or $DBL(A)$, one-bit-shifts of P and D and $MDBL(U, M)$, or swaps of A and B, and U and V, in Step 2-2, or a one-bit-shift of D and $MHLV(V, M)$ in Step 2-3, or a $DBL(A)$ and $MDBL(U)$ or $MHLV(V)$ and a one-bit-shift of D or a SD2 addition of $A - q \cdot B$ or a reset of the most significant digit of A and $MADD(U, -q{\cdot}V, M)$, and a possible $DBL(S)$, a one-bit-shift of D and $MHLV(V, M)$, in Step 2-4, or a negation of V in the SD2 system, in Step 3.

The combinational circuit part of the 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 doubling, a modular halving circuit, an SD2 negator and selectors. The modular adder consists of two SD2 adders one of which is simpler. The modular doubling and the modular halving circuit consist of simpler SD2 adders. These circuits have bit-slice structure.

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 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.

In division mode, from the discussion in the previous section, we can show that in Step 2-4, 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, we can show that the number of clock cycles executed in Step 2 is at least $2n$ and at most about $3n$. It varies with the operands.

Montgomery's multiplication is performed in Step 2-4 in exactly n clock cycles. Ordinary modular multiplication is performed in at least n and at most $2n$ clock cycles. It varies with the multiplier.

6 Concluding Remarks

We have proposed a hardware algorithm for modular multiplication/division. The algorithm performs modular division, Montgomery's modular multiplication and ordinary modular multiplication. Modular division is based on the extended Euclidean algorithm. Montgomery's modular multiplication is based on a new method consisting of processing the multiplier from the most significant digit first. Ordinary modular multiplication is performed using the conventional multiplication algorithm which is based on doubling and adding procedures. The algorithm has almost all hardware components shared for these three operations reducing considerably hardware requirements. It carries out these calculations using simple operations such as shifts, additions and subtractions. 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. A modular multiplier/divider based on the algorithm has a linear array structure with a bit-slice feature and is suitable for VLSI implementation.

References

[1] 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.

[2] T. ElGamal, 'A public key cryptosystem and a signature scheme based on discrete logarithms,' *IEEE Trans. Information Theory*, vol. IT-31, no. 4, pp. 469–472, July 1985.

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

[4] 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.

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

[6] N. Koblitz, 'Elliptic Curve Cryptosystems,' *Math. of Computation*, vol. 48, no. 177, pp. 203–209, Jan. 1987.

[7] N. Takagi, 'A hardware algorithm for modular division based on the extended Euclidean algorithm,' *IEICE Trans. Information and Systems*, vol. E79-D, no. 11, pp. 1518–1522, Nov. 1996.

[8] P. L. Montgomery, 'Modular Multiplication without Trial Division' *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

[9] N. Takagi, H. Yasuura and S. Yajima, 'High-speed VLSI multiplication algorithm with a redundant binary addition tree,' *IEEE Trans. Computers*, vol. C-34, no. 9, pp. 789–796, Sep. 1985.

[10] 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.

[11] Y. Watanabe and N. Takagi, 'A VLSI Algorithm for Division in GF(2^m) Based on Extended Binary GCD Algorithm,' *IEICE Trans. Fundamentals*, vol. E85-A, no. 5, pp. 994–999, May 2002.