Studies on Modular Arithmetic Hardware Algorithms for Public-key Cryptography

Marcelo Emilio Kaihara

Graduate School of Information Science

Nagoya University

January 2006

Dedicated to my father.

Abstract

Public-key cryptography plays an important role in digital communication and storage systems. Processing public-key cryptosystems requires huge amount of computation, and, there is therefore, a great demand for developing dedicated hardware to speed up the computations. In this thesis, we focus on modular arithmetic hardware algorithms for public-key cryptosystem since these two operations are the computationally most intensive parts in encryption and decryption processes.

After reviewing major algorithms for computing modular multiplication and division in Chapter 2, we present in Chapter 3, a mixed radix-$4/2$ algorithm for modular multiplication/division suitable for VLSI implementation. The hardware algorithm is based on the Montgomery multiplication algorithm for modular multiplication and the Extended Binary GCD algorithm for modular division. These two algorithms are combined into the proposed algorithm in order to share hardware components. 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. A modular multiplier/divider based on this hardware algorithm 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.

Chapter 4 presents a hardware algorithm for modular multiplication/division based on the extended Euclidean algorithm. This hardware algorithm performs modular division, Montgomery multiplication, and ordinary modular multiplication. In order to calculate Montgomery multiplication, we propose a new computation method that consists of processing the multiplier from the most significant digit first. The ordinary modular multiplication is based on the interleaved modular multiplication algorithm. Each of these three operations is carried out through the iteration of simple operations such as shifts and additions/subtractions. In order 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.

Chapter 5 presents a new fast method for calculating modular multiplication named Bipartite Modular Multiplication. The calculation is performed using a new representation of residue classes modulo M that enables the splitting of the multiplier into two parts. These two parts are then processed separately, in parallel, potentially doubling the calculation speed. The upper part and the lower part of the multiplier are processed using the interleaved modular multiplication algorithm and the Montgomery algorithm respectively. Conversions back and forth between the original integer set and the new residue system can be performed at speeds up to twice that of the Montgomery method without the need for precomputed constants. This new method is suitable for both hardware implementation; and software implementation in a multiprocessor environment.

A fast hardware algorithm for calculating modular multiplication based on this method is presented at the end of this chapter. In this hardware algorithm, the addition of the partial products to the intermediate accumulated product is pipelined in order to reduce the critical path delay. A radix-4 version of the hardware algorithm is then given and its

hardware implementation is discussed.

Finally, in Chapter 6 we conclude that taking advantage of similarities and symmetries is a good technique for reducing hardware requirement and for speeding up the calculations.

Contents

Acknowledgments

First of all, I would like to thank to my supervisor, Prof. Naofumi Takagi, for his careful guidance during my study years and motivating me to research. He gave me the opportunity to enroll as a Master student and a Ph.D. student under the Monbusho scholarship at Nagoya University. From him I learned the art of the scientific research. I am thankful to Prof. Kazuyoshi Takagi, for all the time he spent for discussing and teaching me to view at problems from different perspectives. I am also grateful to Dr. Kazuhiro Nakamura for all the support during my stay at Nagoya University.

Marcelo E. Kaihara

Nagoya

January 25, 2006

Chapter 1

Introduction

In recent years, we have experienced a great development in the field of digital communication technologies which brought together a great concern about security in computers and communications systems. Several public-key cryptosystems were proposed in order to enable the encryption of messages using a public encryption key e without a prior communication of a secret key. The secrecy relies on the fact that decryption key is computationally infeasible to deduce from the public encryption key. Then, the only person who can decrypt the ciphertext is the receiver, who knows the secret decryption key d.

Most of the widely used public-key cryptosystems heavily rely on operations in modular arithmetic. They are based on two difficult problems: one is the factoring of large integers, and the other one is the computation of the discrete logarithm for finite fields.

We explain the most known public-key cryptosystems and see that the most computationally intensive part of the encryption and decryption processes heavily relies on the performance of calculating modular multiplication and division.

The RSA cryptosystem (Rivest et al. 1978) is the most known cryptosystem based on the difficulty of factoring large integers. The RSA cryptosystem is described as follows: A public key e, and the modulus M, which is a product of two large primes p and q ($M = p \cdot q$) are published. The prime factors of M are kept secret. M is a large number of around thousand bits in length. The secret key d that satisfies $e \cdot d \equiv 1 \pmod{\phi(M)}$, where $\phi(M) = lcm((p-1) \cdot (q-1))$ is then computed. To encrypt a message X ($X \in Z_M$), the sender computes $Y = e_K(X) = X^e \mod M$ using the public key e and M.

The decryption of Y can be performed by the person who possesses the secret key d or the factorization of M, and it can be performed by computing $d_K(Y) = Y^d \mod M = X^{e \cdot d} \mod M = X$.

The RSA cryptosystem can be used to provide digital signatures. A message can be signed using the secret key d and the decryption algorithm as the signing algorithm, i.e. $sig_K = d_K$, and anyone can verify its authenticity by using the encryption algorithm as a public verification algorithm, i.e. $ver_K = e_K$.

As noted above, both the encryption and the decryption operations in RSA are modular exponentiations. Computation of the modular exponentiation can be performed as a series of modular multiplications. If the well-known "square-and-multiply" approach is used, modular exponentiation, $X^c \mod M$, can be calculated in at most $2 \cdot w$ modular multiplications where w is the number of bits in the binary representation of c (Stinson 1995). Note that for decrypting messages in RSA, this number is around thousand of bits. Further acceleration can be accomplished if the modular division operation is available. Then, the calculation of modular exponentiation can be decomposed as a series of modular multiplications and divisions. The details of this approach are explained later in Section 3.6. From this discussion, we see that the performance of the entire cryptosystem relies on the efficiency of calculating modular multiplication and division.

Another well-known public-key cryptosystem is ElGamal cryptosystem (ElGamal 1985) and is based on the difficulty of the discrete logarithm problem for prime field. The cryptosystem can be described as follows: Let p be a prime such that the discrete $\log$ problem in Z_p is intractable, and let $\alpha \in Z_p^*$ be a primitive element. Let p, α, β and a be integers such that $\beta \equiv \alpha^a (\mod p)$. The values of p, α and β are public, and the value of a is secret. Let $k \in Z_{p-1}$ be a secret random number. Then, the encryption rule for a message $x \in Z_p^*$ is $e_K(x, k) = (\alpha^k \mod p, x\beta^k \mod p)$. For decrypting a message composed by $(y_1, y_2) \in Z_p^* \times Z_p^*$, one performs $d_K(y_1, y_2) = y_2 / y_1^a \mod p$. Here, we see again that the encryption and decryption operations requires modular exponentiations and a modular division.

The Elliptic Curve Cryptosystems (ECC) is another well-known public-key cryp-

tosystem that was proposed by Miller and Koblitz (Miller 1985, Koblitz 1987). It is based on the discrete logarithm problem on elliptic curves.

An elliptic curve over the finite field $GF(p)$ is defined as a set of solutions $(x, y) \in Z_p \times Z_p$, which satisfy the elliptic curve equation

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

where x, y, a and b are elements of the field such that $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$, together with a special point $\mathcal{O}$ called the point at infinity.

To encrypt data, it is represented as a point on the chosen curve over the finite field. Point scalar multiplication is the fundamental encryption operation, i.e. a point P is added to itself k times.

$$Q = kP = \underbrace{P + P + \cdots + P}_{\text{(k times)}}$$

In order to compute kP, a double and add method is used. In this method, the value of k represented in binary form is scanned right to left from the least significant bit to the most significant bit. Then, a double and an addition are performed when k_i is 1, where k_i is the i-th bit of k. Hence, a multiplier requires $(m-1)$ point doublings and an average of $(\frac{m-1}{2})$ point additions, where m is the bit length of the field prime p.

Point multiplication operations on elliptic curves over $GF(p)$ are performed by modular addition, subtraction, multiplication and inversion.

Below, we show the point addition/doubling formulae in affine coordinates. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then $P_3 = (x_3, y_3) = P_1 + P_2$ can be computed as:

$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$
$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & if \quad P_1 \neq P_2 \\ \frac{3x_1^2 + a}{2y_1} & if \quad P_1 = P_2 \end{cases}$$

The performance of an ECC processor relies on the efficiency of the finite field computations required to perform the point addition and doubling operations. Point addition requires 2 modular multiplications, 1 modular division and 6 modular addition/subtraction operations. Point doubling requires 3 modular multiplications, 1 modular division and 7 modular additions/subtractions. The critical operation is the modular division and the speed of the overall processor depends on a high-speed modular divider.

In this thesis, we are investigating fast hardware algorithms for modular arithmetic focusing on modular multiplication and division with large modulus suitable to be implemented in compact hardware. Speeding up the computation using specialized hardware enables the use of larger keys in public-key cryptosystems. This is translated into an increase of the security of the system. Also, this enables the speedup of a secure link between two distant points using an insecure channel, which is critical in real-time systems. The reduction of the hardware amount is another important aspect when implementing in dedicated hardware because it allows for the miniaturization of portable devices and reduces fabrication costs.

Many hardware algorithms have been proposed in the literature for computing modular multiplication. Most of them use redundant number systems and performs a high-radix modular multiplication (Orup 1995, Kornerup 1993, Tenca et al. 2001, Koç et al. 1996, Walter 1993) or use Residue Number System (RNS) (Posch and Posch 1995, Bajard et al. 1998, Kawamura et al. 2000, Freking and Parhi 2000). Two major approaches for calculating modular multiplication are the interleaved modular multiplication algorithm (Blakley 1983, Brickell 1983, Kornerup 1993, Morita 1990, Sloan 1985, Takagi 1990) where the multiplier is processed from the most significant digit position and the Montgomery multiplication algorithm (Montgomery 1985, Orup 1995, Tenca et al. 2001, Walter 1993) where the multiplier is processed from the least significant digit position. For modular inversion, we can cite the works of (Parikh and Matula 1991) and (Brent and Kung 1983). For modular division, however, there are only a few algorithms and these are based on the Euclidean algorithm for computing GCD (Takagi 1998, Takagi 1996). These algorithms have been developed for each operation separately.

In this thesis, we apply a technique that exploits algorithmic similarities between algorithms in order to share hardware components reducing hardware requirements. We derived two different hardware algorithms for calculating modular multiplication/division which have no precedence in this research field.

One is the hardware algorithm for modular multiplication/division that is based on the extended Binary GCD algorithm for modular division and the Montgomery algorithm for modular multiplication. These two algorithms process the operands from the least significant digit position and perform similar operations. This algorithmic similarity is used in order to modify and combine them into the proposed hardware algorithm so that almost all the hardware components are shared. We will see that a modular multiplier/divider based on this hardware algorithm can be implemented with much smaller hardware than that necessary to implement both multiplier and divider separately.

The other hardware algorithm for modular multiplication/division is based on the extended Euclidean algorithm and calculates modular division, Montgomery multiplication and ordinary modular multiplication. The extended Euclidean algorithm and the interleaved modular multiplication algorithm used for calculating ordinary modular multiplication process the operands from the most significant position and perform similar operations. This similarity is used to modify and combine them into the proposed hardware algorithm. The Extended Euclidean algorithm and the Montgomery multiplication algorithm seem not to be similar but opposite in the way they process the operands. This apparent inconvenient is overcame by introducing a new computation method for calculating Montgomery multiplication which consists of processing the multiplier from the most significant digit first. Then, the resulting hardware algorithm calculates the three operations sharing almost all hardware components. Additionally, the original hardware algorithm for modular division based on the extended Euclidean algorithm (Takagi 1996) has been modified to further reduce hardware requirement. We will see that the resulting hardware algorithm can be implemented using a hardware amount only slightly larger than that of the modular divider.

This thesis also presents a method called Bipartite Modular Multiplication for speed-

ing up modular multiplication that combines algorithms that are symmetrical in the way they process the operands. This symmetry is used to divide the calculation into two parts which can then be processed in parallel reducing the calculation time. One part of the split calculation can be performed using the interleaved modular multiplication algorithm while the the other part can be performed using the Montgomery multiplication algorithm. Since the split calculation are independent to each other, the techniques for speeding up the interleaved modular multiplication algorithm and the Montgomery multiplication algorithm can be applied separately. Not only we can apply developed techniques but also those that may eventually be devised in the future.

This thesis is organized as follows: In Chapter 2, we will review the algorithms for computing modular multiplication and division. Chapter 3 presents a hardware algorithm for modular multiplication/division based on the extended Binary GCD algorithm for modular division and the Montgomery algorithm for modular multiplication. Chapter 4 presents a hardware algorithm for modular multiplication/division based on the extended Euclidean algorithm. Chapter 5 presents the Bipartite Modular Multiplication method. The hardware implementation is then discussed and a pipelined version of the algorithm which enables to reduce the critical path delay is given at the end of this chapter. A radix-4 version of the hardware algorithm is also presented and its hardware implementation is discussed. Finally, Chapter 6 presents our conclusions.

Chapter 2

Preliminaries

2.1 Modular reduction

The operation "mod" applied to the integer number B and the modulus M, denoted as $B \bmod M$, calculates the residue of $B \pmod{M}$.

$$A = B \bmod M = B - \lfloor \tfrac{B}{M} \rfloor \cdot M$$

where $\frac{B}{M}$ denotes the quotient of the integer division of B by M. The number M is called the modulus. For cryptographic applications, this modulus is usually a large integer and odd.

Two integer numbers A and B are said to be congruent modulo M, if A and B have the property that their difference is divisible by an integer number M. i.e. $M|(A - B)$. The statement "A is congruent to B (modulo M)" is denoted in this thesis as follows.

$$A \equiv B \pmod{M}$$

2.2 Modular Multiplication

2.2.1 Ordinary Modular Multiplication

Given a modulus M, and two elements X and Y of the residue class ring of integers modulo M, the ordinary modular multiplication between X and Y, i.e. $X \times Y$, is defined as the residue of the product $X \cdot Y$ modulo M and is mathematically written as:

$$X \times Y \triangleq X \cdot Y \bmod M$$

The ordinary modular multiplication with large numbers can be computed efficiently by interleaving the multiplication and modular reduction phases reducing storage space.

Let the modulus M be an n-digit number in radix-r. The i-th digit $(i = 0, 1, \cdots, n-1)$ of Y is denoted by y_i. Namely, $Y = \sum_{i=0}^{n-1} y_i \cdot r^i$. Then, the procedure of interleaving the multiplication and the modular reduction can be explained as follows:

$$X \cdot Y \mod M = X \cdot (\textstyle\sum_{i=0}^{n-1} y_i \cdot r^i) \mod M$$
$$= (X \cdot y_{n-1} \cdot r^{n-1} + X \cdot y_{n-2} \cdot r^{n-2} + \cdots + \cdots + X \cdot y_1 \cdot r + X \cdot y_0) \mod M$$
$$= \Big(\cdots \big((X \cdot y_{n-1} \mod M) \cdot r + X \cdot y_{n-2} \mod M \big) \cdot r + \cdots + X \cdot y_1 \mod M \Big) \cdot r + X \cdot y_0 \mod M$$

The radix-r algorithm for calculating the ordinary modular multiplication is shown below (Blakley 1983, Sloan 1985).

[Algorithm 2.1]
(Interleaved Modular Multiplication Algorithm)
Input: $M : r^{n-1} < M < r^n$
$X, Y : 0 \le X, Y < M$
Output: $Z = X \cdot Y \bmod M$
Algorithm:

$Z := 0$;
for $i := n - 1$ **downto** 0 **do**
$Z := r \cdot Z + y_i \cdot X$;
$Z := Z - \lfloor \frac{Z}{M} \rfloor \cdot M$;
endfor

In the algorithm, the digits of the multiplier are processed from the most significant position first. The evaluation of the quotient $\lfloor \frac{Z}{M} \rfloor$ is time consuming and various researches have been directed to simplify this calculation. This is accomplished by using fast estimation techniques using few digits of the most significant positions of Z and M (Brickell 1983, Knuth 1998a, Barrett 1987, Walter 1991, Morita 1990, Takagi 1990).

Compared to the straightforward method of computing the multiplication and then reducing the result modulo M, the interleaved modular multiplication algorithm requires much less storage space with little extra computation effort.

2.2.2 Montgomery Multiplication Algorithm

Montgomery introduced a method for speeding up the calculation of modular multiplication (Montgomery 1985). This method uses a representation of residue classes where the ordinary modular multiplication is replaced by a Montgomery multiplication without affecting the modular addition and subtraction operations.

Given an n-digit odd modulus M and an integer U in radix-r and in the range $[0, M-1]$, the image, or the M-residue of U is defined as $X = U \cdot R_M \mod M$ where R_M is a constant relatively prime to M and $R_M > M$. In order to reduce computation effort, this constant is usually set to the value of r^n. If X and Y are the images of U and V respectively, the Montgomery multiplication of these two images, $X * Y$, is defined as:

$$X * Y \triangleq X \cdot Y \cdot R_M^{-1} \mod M$$

Here, R_M^{-1} is an integer that satisfies $R_M^{-1} \cdot R_M \equiv 1 \pmod{M}$. The result of the above operation is the image of $U \cdot V \mod M$. The mapping between the original residue system and the M-residue system is depicted in Fig 2.1.

The procedure for calculating the Montgomery multiplication can be seen as follows.

$$X \cdot Y \cdot r^{-n} \mod M = X \cdot \left(\sum_{i=0}^{n-1} y_i \cdot r^i\right) \cdot r^{-n} \mod M$$

$$= (X \cdot y_{n-1} \cdot r^{-1} + X \cdot y_{n-2} \cdot r^{-2} + \cdots + \cdots + X \cdot y_1 \cdot r^{-(n-1)} + X \cdot y_0 \cdot r^{-n}) \mod M$$

$$= \Bigg(\Big(\Big(\cdots \big(X \cdot y_0 \cdot r^{-1} \mod M + X \cdot y_1 \big) \cdot r^{-1} \mod M + \cdots + X \cdot y_{n-2} \Big) \cdot r^{-1} \mod M + X \cdot y_{n-1} \Big) \cdot r^{-1} \mod M \Bigg) \mod M$$

From this expression, we can see that, in contrast to the interleaved modular multiplication algorithm, the multiplier can be processed in the reverse order.

Below, we describe the radix-r Montgomery multiplication algorithm for computing the Montgomery multiplication.

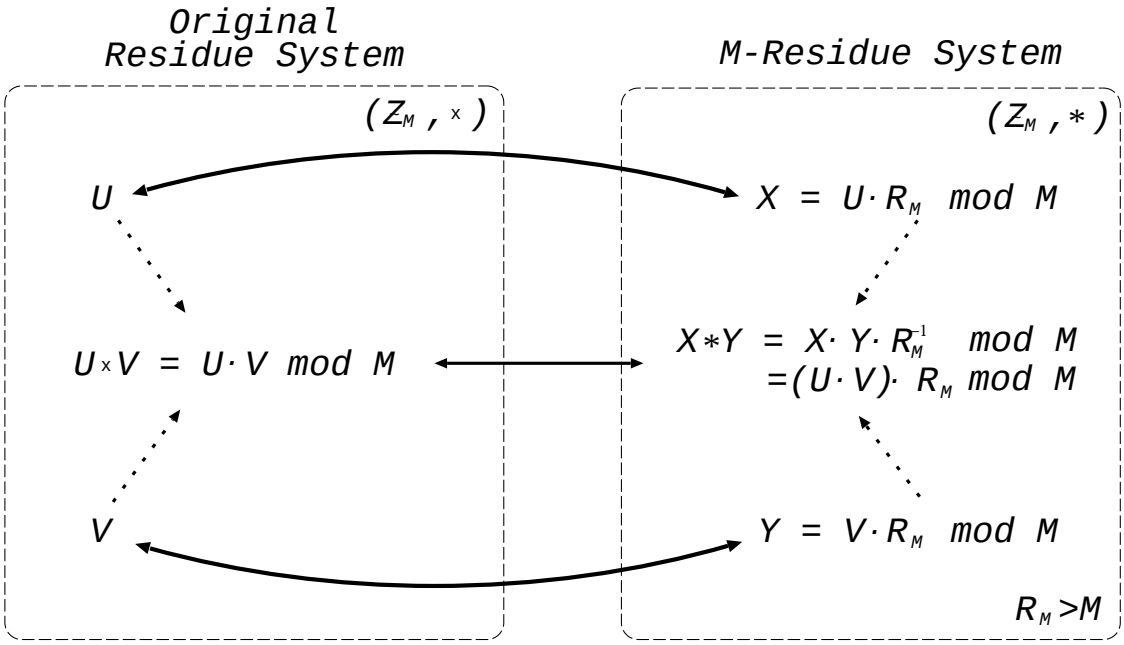

Figure 2.1: *Mapping between the original residue system and the M-residue system*

[Algorithm 2.2]

(Montgomery Multiplication Algorithm)

Input: $M : r^{n-1} < M < r^n$ and $\gcd(M, r) = 1$

$X, Y : 0 \leq X, Y < M$

Output: $Z = X \cdot Y \cdot r^{-n} \bmod M$

Algorithm:

$Z := 0$;

for $i := 0$ **to** $n - 1$ **do**

$Z := Z + y_i \cdot X$;

$Z := (Z + (-Z \cdot M^{-1} \mod r) \cdot M)/r$;

endfor

if $Z \geq M$ **then** $Z := Z - M$

In the algorithm, M^{-1} is a residue modulo r that satisfies $M \cdot M^{-1} \equiv 1 \pmod r$. The multiplication by the term r^{-1} is replaced in the algorithm by an integer division which

is indicated with the symbol "/". For the latter, a multiple of M is calculated and added to make $Z + y_i \cdot X$ divisible by r. We can see that r divides $Z + (-Z \cdot M^{-1} \mod r) \cdot M$, since $(Z+(-Z\cdot M^{-1} \mod r)\cdot M) \mod r = 0$. Then, if the i-th digit of M is denoted as m_i, i.e. $M = \sum_{i=0}^{n-1} m_i \cdot r^i$, and the number that represents the partial products is denoted as $Z = \sum_{i=0}^{n-1} z_i \cdot r^i$, the evaluation of $-Z \cdot M^{-1} \mod r$ can be computed using only the least significant digits of Z and M, i.e. as $-z_0 \cdot m_0^{-1} \mod r$, where m_0^{-1} is the inverse modulo r of m_0. For digital systems, r is usually a power of 2, and M is odd in cryptographic applications. These two conditions guarantees the existence of $m_0^{-1} \bmod r$.

This is an improvement compared to the quotient determination of the interleaved modular multiplication algorithm because the carry propagation need not to be considered in the calculation of $-z_0 \cdot m_0^{-1} \mod r$. In order to simplify the evaluation of $-z_0 \cdot m_0^{-1} \mod r$, similar techniques to those developed for the interleaved modular multiplication have been proposed (Eldridge and Walter 1993, Kornerup 1993).

During the execution of the loop in the algorithm, Z is always bounded by $2 \cdot M$. Therefore, the last correction step assures that the output is in the range $[0, M-1]$.

The transformations back and forth between the original representation and the M-residue representation can be performed using the same algorithm provided that the constant $R_M^2 \mod M$ is precomputed. An integer U can be transformed to the M-residue representation by applying the Montgomery multiplication algorithm to this integer and the constant $R_M^2 \mod M$. Transformation of an image X back into the original integer set can be done by applying the Montgomery multiplication algorithm to this image and the number 1.

2.3 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. Modular division of X by Y, i.e. $X/Y \mod M$, is defined as the residue of the product $X \cdot Y^{-1} \mod M$ where Y^{-1} is an integer satisfying $Y \cdot Y^{-1} \equiv 1 \,(\mathrm{mod} M)$.

$$X/Y \mod M \triangleq X \cdot Y^{-1} \mod M$$

2.3.1 Extended Euclidean Algorithm for Modular Division

One method for calculating modular division is the extended Euclidean algorithm (Knuth 1998b). We describe first the Euclidean algorithm for computing the GCD of two integers and then the extended Euclidean algorithm for computing the modular division. The Euclidean algorithm described below calculates the GCD of two integers through a series of integer divisions. Variable A and B are initialized to X and Y and the algorithm uses the property that $\gcd(A, B) = \gcd(A - \lfloor \frac{A}{B} \rfloor \cdot B, B)$ when $A \geq B$. In the loop, A' stores the remainder of the integer division of A by B.

[Algorithm 2.3]

(Euclidean Algorithm for GCD)

Inputs: X, Y: $X, Y \geq 0$, $X > Y$

Output: $Z = GCD(X, Y)$

Algorithm:

$A := X$; $B := Y$;

while $A > 0$ **do**

$q := \lfloor \frac{A}{B} \rfloor$;

$A' := A - q \cdot B$;

$A := B$; $B := A'$;

endwhile

$Z := B$;

This algorithm is extended to perform modular division by intertwining a procedure for finding the modular quotient with that for calculating $\gcd(Y, M)$. Below is the

extended Euclidean algorithm for computing the modular division. M is the modulus, and X and $Y(\neq 0)$ are two elements of the residue class field of integers modulo M. The algorithm also works with elements of a residue class ring modulo an integer not prime if $\gcd(Y, M) = 1$.

[Algorithm 2.4]

(Extended Euclidean Algorithm for Modular Division)

Inputs: M, X, Y: $0 \leq X < M$, $0 < Y < M$, $\gcd(Y, M) = 1$

Output: $Z = X/Y \bmod M$

Algorithm:

$A := M$; $B := Y$; $U := 0$; $V := X$;

while $B \neq 1$ **do**

$q := \lfloor \frac{A}{B} \rfloor$;

$A' := A - q \cdot B$;

$U' := U - q \cdot V \bmod M$;

$A := B$; $B := A'$;

$U := V$; $V := U'$;

endwhile

$Z := V$;

Variables U, V and U' are used for the calculation of the modular division. Since the same operations that are performed to A and B are also applied to U and V in modulo M, the congruence $V \times Y \equiv B \times X \pmod M$ always holds. Since $\gcd(Y, M) = 1$, the final B is equal to 1. The loop finishes with the condition $Z \times Y \equiv X \pmod M$, resulting Z as the quotient of X/Y modulo M.

2.3.2 Extended Binary GCD Algorithm for Modular Division

The extended Binary GCD algorithm (Knuth 1998b) is another efficient way of calculating modular division. This algorithm is an extension of the Binary GCD algorithm for calculating the GCD of two integers. The following properties are applied iteratively: When A is even and B is odd, then $\gcd(A, B) = \gcd(A/2, B)$. When A and B are odd and $A \geq B$, the equality $\gcd(A, B) = \gcd(A - B, B)$ holds and $A - B$ is even.

Below, we give the Binary GCD algorithm.

[Algorithm 2.5]

(Binary GCD Algorithm)

Inputs: X, Y: $X, Y > 0$, $\gcd(Y, 2) = 1$

Output: $Z = GCD(X, Y)$

Algorithm:

$A := X$; $B := Y$;

while $A > 0$ **do**

while $A \mod 2 = 0$ **do** $A := A/2$; **endwhile**

if $A \geq B$ **then** $A := A - B$;

else $T := A$; $A := B - A$; $B := T$;

endif

endwhile

$Z := B$;

As in the extended Euclidean algorithm, the extended Binary GCD algorithm calculates the 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. 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 respectively.

We show the algorithm below.

[Algorithm 2.6]

(Extended Binary GCD Algorithm for Modular Division)

Inputs: M: $\gcd(M, 2) = 1$

X, Y: $0 \leq X < M,\ 0 < Y < M,\ \gcd(Y, M) = 1$

Output: $Z = X/Y \bmod M$

Algorithm:

$A := Y$; $B := M$; $U := X$; $V := 0$;

while $A > 0$ **do**

while $A \bmod 2 = 0$ **do**

$A := A/2$; $U := U/2 \bmod M$;

endwhile

if $A \geq B$ **then**

$A := A - B$; $U := U - V \bmod M$;

else

$T := A$; $A := B$; $B := T$;

$T := U$; $U := V$; $V := T$;

endif

endwhile

$Z := V$;

The equivalence $V \times Y \equiv B \times X \pmod M$ always holds. Since $\gcd(Y, M) = 1$, the loop of the algorithm finishes with the conditions $A = 0$ and $B = 1$. 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.

2.4 Binary Signed-Digit Number Representation

Fast hardware algorithms for arithmetic operations can be obtained by the use of binary signed-digit representation (SD2) when additions or subtractions are heavily used. 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 $\sum_{i=0}^{n-1} a_i \cdot 2^i$. Addition of two 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 2.1. 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 non-redundant representation.

Table 2.1: *The rule for the first step of addition in binary SD2 representation*

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

Chapter 3

A Hardware Algorithm for Modular Multiplication/Division Based on the Binary GCD Algorithm

3.1 Introduction

In this chapter, a mixed radix-$4/2$ algorithm for modular multiplication/division for a large modulus suitable for VLSI implementation (Kaihara and Takagi 2003, Kaihara and Takagi 2005b) is presented. The calculation of modular multiplication is based on the Montgomery multiplication algorithm (Montgomery 1985) and the modular division on the extended Binary GCD algorithm (Takagi 1998). The reason for basing on these two algorithms is that they have similar structures and use similar operations to perform the calculations. These similarities are exploited in order to modify these two algorithms and share almost all hardware components for both operations.

We have accelerated the 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 in (Takagi 1996) into a one-step operation. Thus, when possible, the operands are processed by two binary digits at each iteration. Redundant representation is used in all additions and subtractions so that they can be carried out without carry propagation.

A modular multiplier/divider based on the algorithm that we present in this chapter has a linear array structure with a bit-slice feature and is suitable for VLSI implementa-

tion. 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.

The amount of hardware that can be saved using the proposed algorithm has been estimated through simulation. For this, we have designed a modular multiplier based on 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 delay remaining practically to the same value.

In the next section, we will review and compare the extended Binary GCD algorithm and the Montgomery multiplication algorithm. Section 3.3 presents accelerated hardware algorithms for modular multiplication and division. Section 3.4 presentes the combined hardware algorithm for modular multiplication and division based on the extended binary GCD algorithm. Section 3.5 explains the hardware implementation and design. Section 3.6 considers possible applications to modular exponentiation and cryptography. Section 3.7, contains the concluding remarks for this chapter.

3.2 Similarity Between the Extended Binary GCD Algorithm and the Montgomery Multiplication Algorithm

In this section, we further modify the extended binary GCD algorithm explained in the previous chapter in order to accelerate the calculation. We compare it to the Montgomery multiplication algorithm also explained in the previous chapter in order to emphasize the similarity between these two algorithms.

The division algorithm described in Section 2.3.2 requires four variables, A, B, U and V. A and B are initialized to the values of Y and M and are used for the calculation of

$\gcd(Y, M)$. 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.

In order to further extend this algorithm we consider the following properties: 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 $|(A - B)/4| \leq \max(|A/2|, |B/2|)$.

If we also allow A and B to take negative values and represent with δ the difference $\alpha - \beta$, where α and β are values such that 2^α and 2^β indicates the upper bounds of $|A|$ and $|B|$ respectively, then the resulting algorithm is described overleaf on the left. Note that ρ is introduced to represent $min(\alpha, \beta)$, and the condition $\rho = 0$ assures that $A = 0$.

If we describe the Montgomery multiplication algorithm using the same variable names, we can see the extended Binary GCD algorithm and the Montgomery multiplication algorithm have similar algorithmic structure and perform similar operations.

Both algorithms require a check of the least significant digit position of the variable A and a right-shift of the variable A. For modular arithmetic, both algorithms require a the operation $U/2 \bmod M$. The division algorithm requires the operation $(U + V)/4 \bmod M$ while the Montgomery multiplication algorithm the operation $(U + V)/2 \bmod M$. We will see in the next section that these two algorithms can be completely combined into a new algorithm.

[Algorithm 3.1]

(Algorithm for Mod. Div.)

Inputs: M: $2^{n-1} < M < 2^n$, odd 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 /* $B = -1$ */ $Z := M - V$;

[Algorithm 3.2]

(Algorithm for Montgomery Mult.)

Inputs: $M : 2^{n-1} < M < 2^n$ and odd

$X, Y : 0 \leq X, Y < M$

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 $Z := U$;

3.3 Hardware Algorithms for Modular Multiplication and Division Based on the Extended Binary GCD Algorithm

In this section, we present hardware algorithms that performs Montgomery modular multiplication and modular division that are suitable to be combined and are efficient in execution time and hardware requirements. We first describe our accelerated modular division algorithm, then our accelerated Montgomery modular multiplication algorithm.

3.3.1 An Accelerated Hardware Algorithm for Modular Division Based on the Binary GCD Algorithm

A hardware algorithm based on the extended Binary GCD algorithm presented in the previous section was proposed in (Takagi 1998). We modified and accelerated it. We first explain the implementation of (Takagi 1998) and then the modification we introduced.

The algorithm described in (Takagi 1998), 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 a 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 the condition 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 $U/2 \bmod M$.

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 $U := (U+V)/4 \mod M$ (or $A := (A-B)/4$ and $U := (U-V)/4 \mod M$) 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 examined.

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 examined 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., (Takagi and Yajima 1992). Since we are assuming that M is odd, only the second least significant bit of M, i.e. m_1, has to be examined to determine the value of $M \bmod 4$. Operation $U := (U+V)/4 \mod M$ and $U := (U-V)/4 \mod M$ 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 examined 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.

[Hardware Algorithm 3.3]

(Modified Hardware Algorithm for Modular Division)

Inputs: $M : 2^{n-1} < M < 2^n$, $\gcd(M, 2) = 1$ and prime

$X, Y : -M < X, Y < M \quad (Y \neq 0)$

Output: $Z = X/Y \bmod M \quad (-M < Z < M)$

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 $[a_1 a_0] = 0$ **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 /* $A \equiv 1 \pmod 4$ or $A \equiv 3 \pmod 4$ */
if $([a_1a_0] + [b_1b_0]) \bmod 4 = 0$ **then** $q := 1$ **else** $q := -1$;
if $s = 0$ **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 in 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 updates 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 B is initialized to a value so that $B \equiv 1 \pmod 4$. Otherwise it ends with value -1. This happens when $[b_1 b_0] = 11$ or $[b_1 b_0] = \bar{1}1$ or $[b_1 b_0] = 0\bar{1}$ and V is negated in the SD2 system.

In Step 4, V is selected as the output.

Fig. 3.1 shows an example of a modular division, $-115/249 \bmod 251 = -68 \bmod 251 = 183$ where $n = 8$ by [Hardware Algorithm 3.3]. The leftmost column shows which calculations have been carried out. For example, '$(A - B)/4$, A' means that { $A := (A - B)/4$, $B := A$ } and { $U := MQRTR(U - V, M)$, $V := U$ } have been carried out.

$M = [11111011]_2$ (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	
	$111111\bar{1}\bar{1}$	(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	$000000\bar{1}1$	(-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)	$000000\bar{1}1$	(-1)	0001000000	0000000001	1	$01000\bar{1}\bar{1}1$	(63)	01000100	(68)
$A/4$,B	$000000\bar{1}\bar{1}$	(-3)	$000000\bar{1}1$	(-1)	0000010000	0000000100	1	$0\bar{1}1\bar{1}01\bar{1}\bar{1}$	(-47)	01000100	(68)
$(A+B)/4$,A	$0000000\bar{1}$	(-1)	$000000\bar{1}\bar{1}$	(-3)	0000010000	0000000010	0	01000100	(68)	$0\bar{1}1\bar{1}01\bar{1}\bar{1}$	(-47)
$(A+B)/4$,B	$0000000\bar{1}$	(-1)	$000000\bar{1}\bar{1}$	(-3)	0000010000	0000000001	1	01000100	(68)	$0\bar{1}1\bar{1}01\bar{1}\bar{1}$	(-47)
$(A+B)/4$,B	$0000000\bar{1}$	(-1)	$000000\bar{1}\bar{1}$	(-3)	0000001000	0000000010	1	01000100	(68)	$0\bar{1}1\bar{1}01\bar{1}\bar{1}$	(-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)

Figure 3.1: *A modular division example by [Hardware Algorithm 3.3]*

3.3.2 A Hardware Algorithm for Montgomery Modular Multiplication

In this subsection, we describe a hardware algorithm for Montgomery multiplication algorithm. The Montgomery multiplication algorithm described in Section 3.2 is modified in order to enable the sharing of the hardware originally used for division, and accelerate the calculation process. For this, we initialize the variables V and A with the values of X and Y respectively. The variable U which is used to store the partial products is initialized to 0. Also, we introduce SD2 representation in operands, internal calculations and the output result and examine the least significant two digits of A, i.e. $[a_1 a_0]$, to process them at each iteration when possible. The algorithm follows the same structure of the division algorithm and use the same test conditions.

For the case $[a_1 a_0] = 0$, 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. 3.2 (a).

If $[a_1 a_0] = [10]$ 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. 3.2 (b).

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

In the division algorithm ([Hardware Algorithm 3.3]), the test condition $([a_1 a_0] + [b_1 b_0]) \bmod 4$ 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 $[a_1 a_0]$. 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 $[a_1 a_0] = [01]$ or $[1\bar{1}]$, which means that the value of $[a_1 a_0]$ is equal to 1, $([a_1 a_0] + [b_1 b_0]) = 0 \bmod 4$, so, as in the division algorithm,

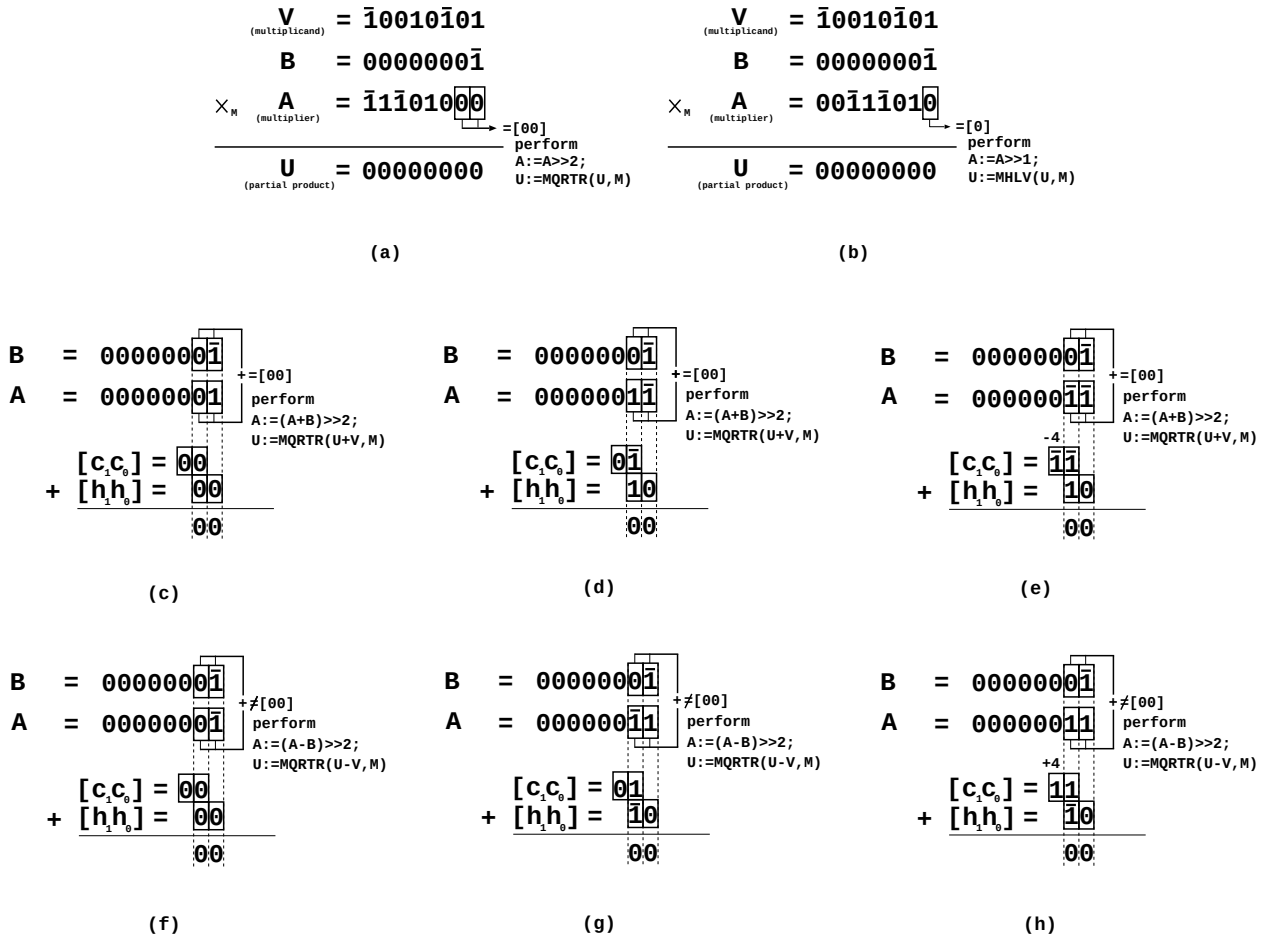

Figure 3.2: *Diagram showing the transformation process of* $[a_1a_0]$

$U := MQRTR(U + V, M)$ and $A = (A + B) >> 2$ are performed. See Fig. 3.2 (c) and (d).

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. 3.2 (f) and (g).

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. 3.2 (e).

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. 3.2 (h).

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_M, 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 is shifted to the right until the final condition of $P = 1$.

We present the accelerated Montgomery multiplication algorithm.

[Hardware Algorithm 3.4]

(Accelerated 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: $Z = XY2^{-(n+2)} \bmod M \quad (-M < Z < M)$

Algorithm:

Step 1: $A := Y$; $B := \overline{1}$; $U := 0$; $V := X$; $M := M$;

$P := 2^{n+1}$; $D := 1$; $s := 1$;

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 $([a_1 a_0] + [b_1 b_0]) \bmod 4 = 0$ **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 has always the value of 0 and $MHLV(U, M)$ is needed to be performed in Step 3.

Proposition 1: In [Hardware Algorithm 3.4], 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 2.1. 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 as $[a'_2a'_1a'_0]$, 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 that 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 to 0.

∎

Fig. 3.3 shows an example of a Montgomery multiplication, $-115 \times 249 \times 2^{-10} \bmod 251 = 137$ where $n = 8$ by [Hardware Algorithm 3.4]. 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), $X = [\bar{1}0010\bar{1}01]_{SD}$ (-115), $Y = [111111\bar{1}\bar{1}]_{SD}$ (249)

	A		B		P	s	U		V	
	$111111\bar{1}\bar{1}$	(249)	$0000000\bar{1}$	(-1)	1000000000	1	00000000	(0)	$\bar{1}0010\bar{1}01$	(-115)
$(A+B) >> 2$	$010000\bar{1}0$	(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)	0000010000	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)	0000000100	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)

Figure 3.3: *A Montgomery modular multiplication by [Hardware Algorithm 3.4]*

3.4 The Combined Hardware Algorithm for Modular Multiplication/Division Based on the Extended Binary GCD Algorithm

In this section, a combined mixed radix-$4/2$ hardware algorithm for modular multiplication/division is presented. It consists of 4 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.

[Hardware Algorithm 3.5]

(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: $mode = 0 : Z = XY2^{-(n+2)} \bmod M \quad (-M < Z < M)$

$mode = 1 : Z = X/Y \bmod M \quad (-M < Z < M)$

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** $([b_1 b_0] = 3$ **or** $[b_1 b_0] = -1)$ **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 $\bar{1}$, 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 in division mode, it is shifted only by 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 process 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. Hence, the proposed hardware algorithm is a mixed radix-$4/2$ algorithm.

3.5 Hardware Implementation

3.5.1 A Description of the Hardware

We implement each iteration of the 'while' loop in Step 2, i.e., one row in Fig. 3.1/Fig. 3.3, to be performed in one clock cycle.

A modular multiplier/divider based on [Hardware Algorithm 3.5] consists of 7 registers for storing A, B, P, D, U, M and V, three SD2 adders, one of which is simpler, multiplexors and a small control circuit. Fig. 3.4 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 store 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 $state$.

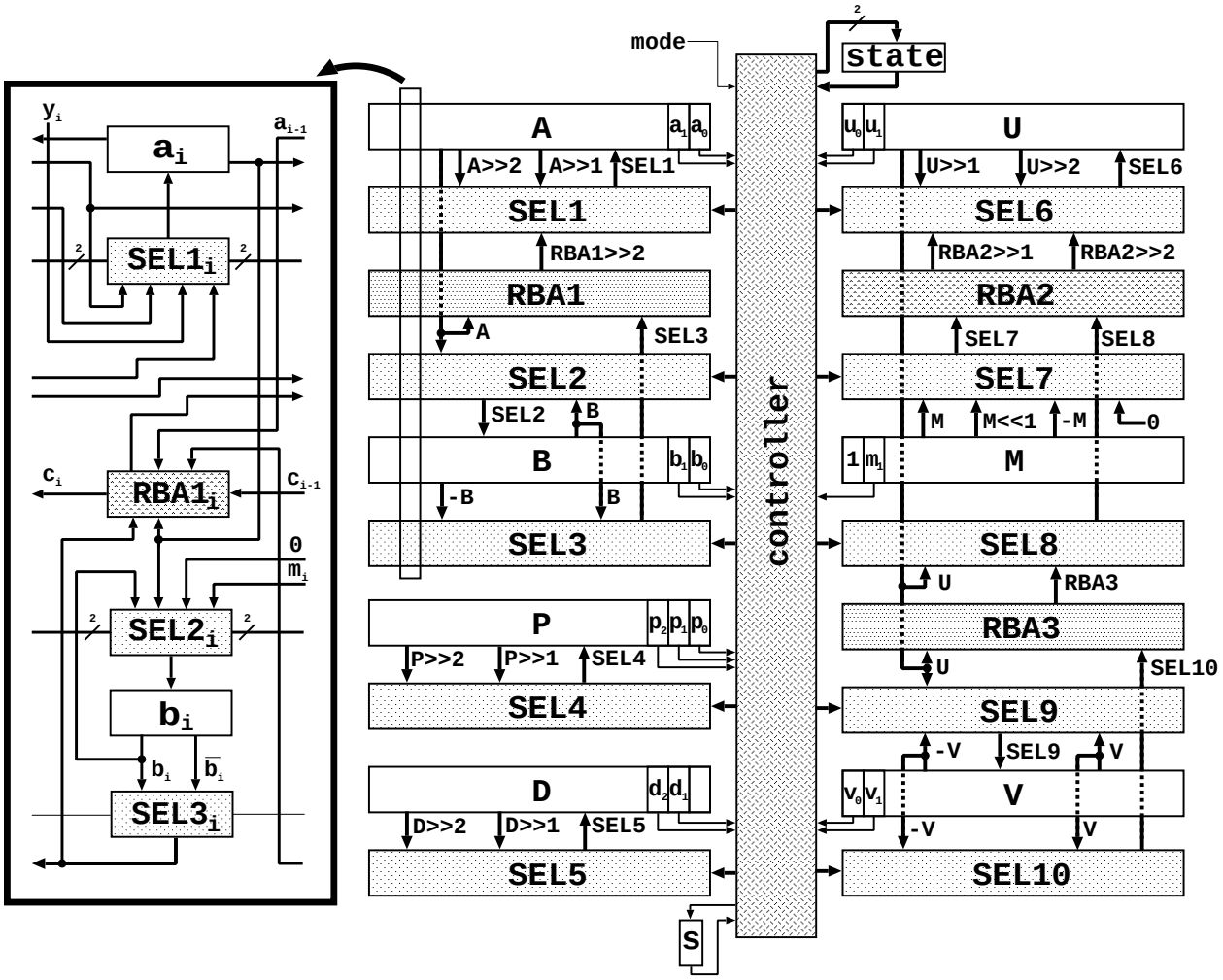

Figure 3.4: *Block diagram of a multiplier/divider*

As an example, we describe the behavior of the circuit components of the expanded diagram of Fig. 3.4 during an iteration of Step 2. If $[a_1a_0] \bmod 4 = 0$, the controller sends 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 SD2 adder consists of a combinational circuit whose addition rule is shown in

Table 2.1. To generate the i-th 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.

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 to 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.

3.5.2 Hardware Implementation 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 3.1 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 Montgomery algorithm with the acceleration we introduced for processing two digits in SD2 system when possible. The modular divider is based on 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.

Table 3.1: *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 [μs]		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

3.6 Considerations

3.6.1 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 $X^e \mod M$. 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.

3.6.2 Applications to Cryptography

The proposed combined algorithm is efficient in sharing hardware resources and computational speed. However, in cryptographic applications, data dependent timing variation may provide information leakage and need to be considered. Timing and power attacks were initially introduced by Kocher (Kocher 1996, Kocher et al. 1999) and various countermeasures have been proposed. They are based on the randomization of the private exponent (Coron 1998, Oswald and Aigner 2001), and, on blinding the operands with a secret random number and unblinding it after exponentiation(Kocher 1996). In order to increase the security of the systems, these countermeasures can be applied.

In case that 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 presented 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 case that modular division is needed to be calculated in constant time, 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 a sense that the electromagnetic power radiation emanates from only one source, therefore, no positional information is provided for the different operations.

3.7 Concluding Remarks

We have presented 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 suitable 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 are able to be shared.

Chapter 4

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

4.1 Introduction

In the previous chapter, we presented 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 chapter, we present 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 (Kaihara and Takagi 2005c). To the best of our knowledge, there is no other work proposed in literature that combines these three operations.

In the hardware algorithm that we describe in this chapter, modular division is based on the extended Euclidean algorithm. We improve the hardware algorithm for modular division originally presented in (Takagi 1996) 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 interleaved modular multiplication algorithm. 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 binary 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 review the extended Euclidean algorithm, Montgomery multiplication algorithm and the ordinary modular multiplication algorithm. We also explain basic operations in the SD2 system. In section 4.3, we propose a hardware algorithm that performs modular division which is efficient in execution time and hardware requirements. We also present an improved algorithm for calculating Montgomery multiplication suitable for combining to this division algorithm. Section 4.4 presents a hardware algorithm for modular multiplication/division based on the extended Euclidean algorithm. In section 4.5, we discuss hardware implementation. Finally, in section 4.6, we present our concluding remarks of this chapter.

4.2 Preliminaries

4.2.1 Similarity Between the Extended Euclidean Algorithm and the Interleaved Modular Multiplication Algorithm

In this section, a further modified extended Euclidean algorithm is explained and compared to the interleaved modular multiplication algorithm explained in Chapter 2 in order to emphasize the similarity between these two algorithms.

The further extended Euclidean algorithm requires four variables, $\mathcal{A}$, $\mathcal{B}$, U and V. $\mathcal{A}$ and $\mathcal{B}$ are initialized to M and Y respectively and are used for the calculation of

$\gcd(Y, M)$. The variables U and V are initialized to the values of X and 0 respectively and are used to calculate the modular quotient. The further modified extended Euclidean algorithm is presented below on the left. Note that $\mathcal{A}$ ($\mathcal{A}'$) and $\mathcal{B}$ and also in U (U') and V are allowed to take negative values. The equivalence $V \times Y \equiv \mathcal{B} \times X \pmod{M}$ always holds and 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.

The interleaved modular multiplication algorithm is shown below on the right. In the algorithm, a_i denotes the i-th digit of the variable A.

[Algorithm 4.1]
(Algorithm for Modular Division)
Inputs: M: 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** $Z := Z'$;

[Algorithm 4.2]
(Interleaved Modular Multiplication)
Input: $M : r^{n-1} < M < r^n$
$X, Y : 0 \leq X, Y < M$
Output: $Z = X \cdot Y \bmod M$
Algorithm:
$A := Y$; $U := 0$; $V := X$;
for $i := n - 1$ **downto** 0 **do**
$Q := -a_i$;
$U := r \cdot U - V \cdot Q \bmod M$;
endfor
$Z := U$

If we compare the explained division algorithm to the interleaved modular multiplication algorithm, we can see that these two algorithms process the operands from the

most significan digit position and perform similar operations. We will see in the following section that this similarity can be used to combine these two operations in order to share hardware components.

4.2.2 The Extended Euclidean Algorithm and the Montgomery Multiplication Algorithm

In this section, we review the algorithm introduced by Montgomery and explained in Section 2.2.2. If we compare the Montgomery multiplication algorithm shown below to the extended Euclidean algorithm, we can see that they perform similar operations but in opposite direction. The extended Euclidean algorithm process the operands from the most significant digit position while the Montgomery multiplication algorithm process from the least significant digit position. We will see in Section 4.3.2 how to overcome this apparent inconvenient in order to share almost all the hardware components.

[Algorithm 4.2] (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:

$U := 0$;

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** $Z := U$;

4.2.3 Basic Operations in SD2 System

The hardware algorithm for modular division requires a doubling procedure for an SD2 integer without overflow (Takagi 1996). 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 ($s_{n-1} = -s_n$), $A = [s_n s_{n-2}s_{n-3} \cdots s_1 s_0 0]$.

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 $[w_{n+1}w_n w_{n-1}]$, 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 (Takagi and Yajima 1992).

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.

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

4.3 Hardware Algorithms for Modular Multiplication and Division Based on the Extended Eucliden Algorithm

We present a hardware algorithm that performs modular division which is efficient in execution time and hardware requirements. We also present an improved algorithm for calculating Montgomery multiplication suitable for combining to the division algorithm.

4.3.1 An Improved Hardware Algorithm for Modular Division Based on the Extended Euclidean Algorithm

We introduce two modifications to improve the hardware algorithm for modular division based on the extended Euclidean algorithm proposed in (Takagi 1996). In the hardware algorithm proposed in (Takagi 1996), variables $\mathcal{A}$ and $\mathcal{B}$ of [Algorithm 4.1] 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 (Takagi 1996), 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 the representation of them 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 (Takagi 1996) is eliminated.

The second modification that we introduce is related to the order of the steps. The structure of the hardware algorithm proposed in (Takagi 1996) follows the structure of [Algorithm 4.1]. 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 (Takagi 1996).

The improved hardware algorithm is described below:

[Hardware Algorithm 4.1]

(Modular Division)

Inputs: $M : 2^{n-1} < M < 2^n$ and odd prime

$X, Y : -M < X, Y < M, \ \ Y \neq 0$

Output: $Z : Z \equiv X/Y (\mathrm{mod} M)$ 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 (Takagi 1996) 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 (Takagi 1996).

A numerical example of a modular division performed by [Hardware Algorithm 4.1] is given overleaf in Fig.4.1. The calculation of $205/199 \bmod 251$ is performed using $n = 8$. Lines $1 - 3$ represent the equivalent to the first iteration of [Algorithm 4.1]. 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 $Z = -86 \equiv 165 \pmod{251}$.

$M = [11111011]_2$ (251), $X = [1\bar{1}10\bar{1}0\bar{1}01]_{SD}$ (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	$A := A - B$	00101$\bar{1}$0100	10$\bar{1}$000111	000000001	0000000001	001$\bar{1}$1100$\bar{1}$0	1$\bar{1}$1010$\bar{1}$01
4	*Step* 2-1	$NORM(A)$	0101$\bar{1}$01000	10$\bar{1}$000111	000000010	0000000010	0$\bar{1}$0$\bar{1}$000001	1$\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}$111$\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}$111$\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}$$\bar{1}1\bar{1}$01	001$\bar{1}$$\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}$$\bar{1}1\bar{1}$01	000$\bar{1}$$\bar{1}$$\bar{1}$110
9	*Step* 2-1	$NORM(A)$	0$\bar{1}$100$\bar{1}$000	10$\bar{1}$010000	000001000	0000000010	0$\bar{1}$$\bar{1}0\bar{1}00\bar{1}$1	000$\bar{1}$$\bar{1}$$\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}$	000$\bar{1}$$\bar{1}$$\bar{1}$110
11	*Step* 2-1	$NORM(A)$	$\bar{1}$00$\bar{1}$00000	10$\bar{1}$010000	000100000	0000001000	$\bar{1}$111$\bar{1}$01$\bar{1}$$\bar{1}$	000$\bar{1}$$\bar{1}$$\bar{1}$110
12	*Step* 2-2	$SWAP$	10$\bar{1}$010000	$\bar{1}$00$\bar{1}$00000	000100000	0000001000	000$\bar{1}$$\bar{1}$$\bar{1}$110	$\bar{1}$$\bar{1}$$\bar{1}$$\bar{1}01\bar{1}$$\bar{1}$
13	*Step* 2-3	$A := DBL(A + B)$	$\bar{1}$10$\bar{1}$00000	$\bar{1}$00$\bar{1}$00000	000100000	0000000100	01101$\bar{1}$0$\bar{1}$0	001010100
14	*Step* 2-3	$A := DBL(A)$	$\bar{1}$0$\bar{1}$000000	$\bar{1}$00$\bar{1}$00000	000100000	0000000010	01101$\bar{1}$0$\bar{1}$0	000101010
15	*Step* 2-3	$A := DBL(A - B)$	00$\bar{1}$000000	$\bar{1}$00$\bar{1}$00000	000100000	0000000001	00$\bar{1}$0$\bar{1}$00001	000010101
16	*Step* 2-1	$NORM(A)$	0$\bar{1}$0000000	$\bar{1}$00$\bar{1}$00000	001000000	0000000010	0$\bar{1}$$\bar{1}0\bar{1}$$\bar{1}$$\bar{1}$$\bar{1}$	000010101
17	*Step* 2-1	$NORM(A)$	$\bar{1}$00000000	$\bar{1}$00$\bar{1}$00000	010000000	0000000100	00$\bar{1}$000$\bar{1}$$\bar{1}$	000010101
18	*Step* 2-2	$SWAP$	$\bar{1}$00$\bar{1}$00000	$\bar{1}$00000000	010000000	0000000100	000010101	00$\bar{1}$000$\bar{1}$$\bar{1}$
19	*Step* 2-3	$A := DBL(A - B)$	00$\bar{1}$000000	$\bar{1}$00000000	010000000	0000000010	0$\bar{1}$0$\bar{1}$0$\bar{1}$$\bar{1}$$\bar{1}$1	001100$\bar{1}$01
20	*Step* 2-3	$A := DBL(A)$	0$\bar{1}$0000000	$\bar{1}$00000000	010000000	0000000001	0$\bar{1}$0$\bar{1}$0$\bar{1}$$\bar{1}$$\bar{1}$1	010111$\bar{1}$100
21	*Step* 2-1	$NORM(A)$	$\bar{1}$00000000	$\bar{1}$00000000	100000000	0000000010	0$\bar{1}$$\bar{1}1\bar{1}001\bar{1}$	010111$\bar{1}$100
22	*Step* 2-2	$SWAP$	$\bar{1}$00000000	$\bar{1}$00000000	100000000	0000000010	010111$\bar{1}$100	0$\bar{1}$1$\bar{1}$1001$\bar{1}$
23	*Step* 2-2	$MHLV(V)$	$\bar{1}$00000000	$\bar{1}$00000000	100000000	0000000001	010111$\bar{1}$100	001011$\bar{1}$$\bar{1}$10
24	*Step* 3	$V := -V$	$\bar{1}$00000000	$\bar{1}$00000000	100000000	0000000001	010111$\bar{1}$100	00$\bar{1}$0$\bar{1}$$\bar{1}$$\bar{1}$10
		Z						00$\bar{1}$0$\bar{1}$$\bar{1}$$\bar{1}$0

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

Figure 4.1: *A modular division by [Algorithm 4.1]*

4.3.2 A New Algorithm for Computing Montgomery Multiplication

In order to implement Montgomery multiplication using the same hardware components needed for modular division, we modify [Algorithm 4.2] 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 4.4]

(Modified Montgomery Multiplication)

Inputs: $M : 2^{n-1} < M < 2^n$ and $\gcd(M, 2) = 1$

$X, Y : 0 < X, Y < M$

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

$Z := U;$

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

In this section, a new hardware algorithm for calculating modular multiplication/division 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 4.1], [Algorithm 4.3] and [Algorithm 4.4], 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 the same way in multiplication mode, i.e. $mode = 1$ and $mode = 2$, to store the multiplier, the partial product and the multiplicand respectively.

[Hardware Algorithm 4.2]

(Modular Multiplication/Division)

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: $Z : Z \equiv X/Y \bmod M \quad (if\ mode = 0)$

$Z \equiv X \cdot Y \cdot 2^{-n} \bmod M \ (if\ mode = 1)$

$Z \equiv X \cdot Y \bmod M \quad (if\ mode = 2)$

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 4.1]. 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 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 Multiplication/ Ordinary Modular Multiplication)

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 describe how the algorithm works for calculating ordinary modular multi-

plication. 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.

4.5 Hardware Implementation

An n-bit modular multiplier/divider based on [Hardware Algorithm 4.2] consists of seven registers for storing A, B, P, D, U, V and M, and a combinational circuit part.

We assume that each iteration of the 'while' loops are performed in one clock cycle. Thus, in Step 2-1, operations $DBL(A)$, 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

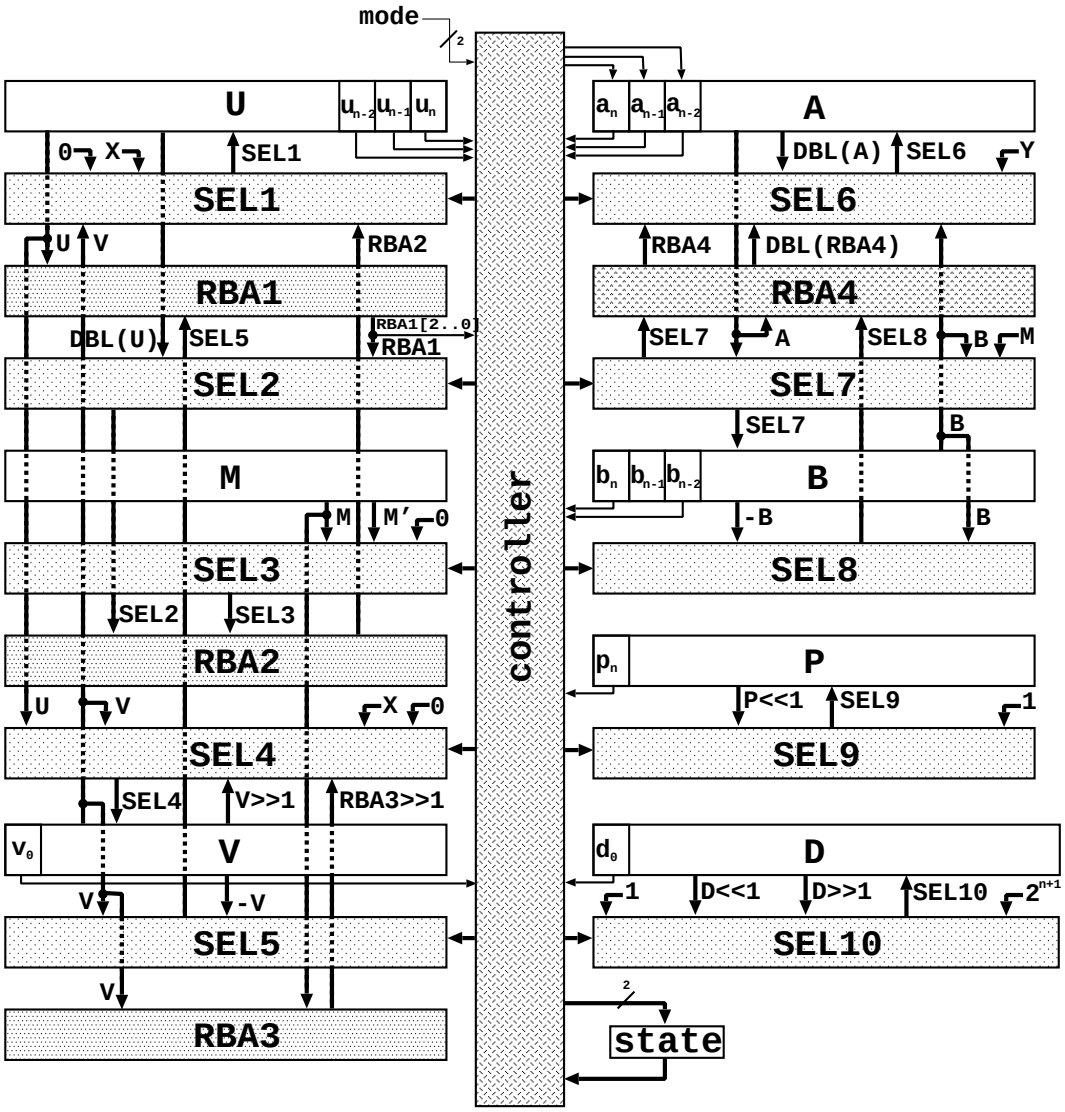

Figure 4.2: *Block diagram of a multiplier/divider*

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 there-

fore, 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 (Takagi 1996) 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.

4.6 Concluding Remarks

We have presented a hardware algorithm for modular multiplication/division based on the extended Euclidean algorithm. We first improved the hardware algorithm for modular division proposed in (Takagi 1996) to reduce hardware requirements. Then, we modified 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, Mont-

gomery 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 Montgomery domain by multiplying the operand with the value of 2^n without the need for any precomputed constants.

Following the discussion of the previous chapter (see 3.6), 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 (Kaihara and Takagi 2003).

Chapter 5

Bipartite Modular Multiplication

5.1 Introduction

This chapter presents a fast method for calculating modular multiplication that takes advantage of the techniques developed to speed up the interleaved modular multiplication algorithm (Blakley 1983, Brickell 1983, Kornerup 1993, Morita 1990, Sloan 1985, Takagi 1990) and the Montgomery algorithm (Montgomery 1985, Orup 1995, Tenca et al. 2001, Walter 1993). This method is called Bipartite Modular Multiplication (BMM) (Kaihara and Takagi 2005a). The key that enables the linking of these two approaches is the representation of residue classes modulo M called KT-residue. Assuming that M is an n-digit odd integer, and the radix of each digit is $r = 2^k$, this representation maps an integer U in the range $[0, M-1]$ to the number $U \cdot R \mod M$ in the same range. R is a constant of value $r^{\alpha n}$, relatively prime to M, where α is a rational number such that $0 < \alpha < 1$, and, αn is an integer. The set of these images forms a complete residue system called KT-residue system. The novelty in this representation is that the transformation constant R has a value less than the modulus M, a condition not allowed by the Montgomery representation. Modular multiplication is then performed in the KT-residue system. The new values for the transformation constant enable the splitting of the multiplier into two parts which can then be processed separately, in parallel. The upper part and the lower part of the multiplier can be processed using the interleaved modular multiplication algorithm and the Montgomery algorithm, respectively. The possibility of selecting the parameter α between the values 0 and 1, encompasses the application of this method to all combinations of algorithms of different performance

derived from the interleaved modular multiplication algorithm and the Montgomery algorithm including those that may eventually be devised in the future. If applied to algorithms with similar performance and the multiplier is split into two equal parts, it is theoretically possible to achieve the maximum speed of twice that of these two algorithms when performed individually. The latter condition is represented with the value of the parameter α so that $\alpha n = \lceil \frac{n}{2} \rceil$.

Two other advantages of this new method are: Firstly, compared to the Montgomery method, conversion speed between the original integer set and the new residue system is potentially doubled; and secondly, precomputation of constants is no longer necessary.

Due to the parallel processing, the proposed method is suitable for hardware implementation and also for software implementation in a multiprocessor environment.

This chapter also presents a new hardware algorithm based on the BMM method which enables the pipelining of the intermediate calculations in a natural way. This is accomplished by applying the shift operations, the modular reduction and the Montgomery reduction to the multiplicand instead of applying them to the intermediate accumulated product. A radix-4 version of the algorithm that can be implemented with similar performance and with less hardware requirements to that of a pipelined radix-16 Montgomery multiplier (Orup 1995) is presented and discussed.

The remainder of this chapter is organized as follows: Section 5.2 reviews the interleaved modular multiplication algorithm and the Montgomery algorithm and explains the symmetry between these two algorithms. The BMM method is presented in Section 5.3. Section 5.4 explains hardware implementation of the method. Section 5.5 presents pipelined hardware algorithm based on the BMM method. In Section 5.6 a radix-4 version of the pipelined hardware algorithms is explained. Section 5.7 contains our concluding remarks of this chapter.

5.2 Symmetry Between the Interleaved Modular Multiplication Algorithm and the Montgomery Multiplication Algorithm

Given a modulus M, and two elements of the residue class ring of integers modulo M, X and Y, the ordinary modular multiplication calculates $X \cdot Y \mod M$. Given an n-digit odd modulus M and an integer U in the range $[0, M-1]$, the image, or the M-residue of U is defined as $X = U \cdot R_M \mod M$ where R_M is a constant relatively prime to M and $R_M > M$. In order to reduce computation effort, this constant is usually set to the value of r^n. If X and Y are the images of U and V respectively, the Montgomery multiplication of these two images, $X * Y$, results as $X \cdot Y \cdot R_M^{-1} \mod M$ which is the image of $U \cdot V \mod M$.

Let the modulus M be an n-digit number (odd for Montgomery multiplication), where the radix of each digit is $r = 2^k$. If the i-th digit of M is denoted as m_i, then $M = \sum_{i=0}^{n-1} m_i \cdot r^i$. The i-th digit $(i = 0, 1, \cdots, n-1)$ of Y is denoted by y_i. Namely, $Y = \sum_{i=0}^{n-1} y_i \cdot r^i$. In a similar way, if the number that represents the intermediate accumulated products is denoted as $Z = \sum_{i=0}^{n-1} z_i \cdot r^i$, the interleaved modular multiplication algorithm. i.e. [Algorithm 2.1], and the Montgomery multiplication algorithm, i.e. [Algorithm 2.2], can be rewritten as below.

[Algorithm 5.1]

(Interleaved Mod. Mul. Algorithm)

Input: $M : r^{n-1} < M < r^n$

$X, Y : 0 \le X, Y < M$

Output: $Z = X \cdot Y \mod M$

Algorithm:

$Z := 0$;

for $i := n-1$ **downto** 0 **do**

$Z := r \cdot Z + y_i \cdot X$;

$q_C := \lfloor \frac{Z}{M} \rfloor$;

[Algorithm 5.2]

(Montgomery Algorithm)

Input: $M : r^{n-1} < M < r^n$ and $\gcd(M, 2) = 1$

$X, Y : 0 \le X, Y < M$

Output: $Z = X \cdot Y \cdot r^{-n} \mod M$

Algorithm:

$Z := 0$;

for $i := 0$ **to** $n-1$ **do**

$Z := Z + y_i \cdot X$;

$q_M := (-z_0 \cdot m_0^{-1}) \mod r$;

$Z := Z - q_C \cdot M;$

endfor

$Z := (Z + q_M \cdot M)/r;$

endfor

if $Z \geq M$ **then** $Z := Z - M$

If we compare these two algorithms, we can see that the multiplier Y is processed from the most significant position in the interleaved modular multiplication algorithm whereas in the Montgomery multiplication algorithm the multiplier is processed from the least significant position. Also, the number that stores the intermediate accumulated product Z is multiplied by r in the interleaved modular multiplication algorithm which can be implemented by a left-shift operation. In constrast, the partial product Z is divided by r in the Montgomery algorithm which can be implemented by a right-shift operation. Finally, the quotient q_C can be determined using the most significant few digits of Z and M, whereas q_M can be determined using the least significant digit of Z and M. We will see in the next section how we can exploit this symmetry to speed up the calculation of modular multiplication.

5.3 Bipartite Modular Multiplication Method

In this section, a new fast method for calculating modular multiplication is presented. The calculation is performed using a new representation of residue classes modulo M. In contrast to the M-residue representation introduced by Montgomery which requires the constant R_M to be relatively prime to M and greater in value than M, we have changed the condition of $R_M > M$ and defined a new residue class representation using a new constant $R = r^{\alpha n}$, where R is relatively prime to M, and, α is a rational number so that $0 < \alpha < 1$ and αn is an integer. The resulting image of an integer U is $X = U \cdot r^{\alpha n} \mod M$. Given X and Y, two images of integers U and V respectively, multiplication modulo M in the new residue system is defined as:

$$X \circledast Y \triangleq X \cdot Y \cdot r^{-\alpha n} \mod M \tag{5.1}$$

The existence of $r^{-\alpha n} \mod M$ is assured by the condition that $r^{\alpha n}$ is realatively prime to M. Since M is odd for cryptographic applications, by utilizing $r = 2^k$, the primalty condition is satisfied.

Transformation from the original representation to the new residue system is accomplished by performing conventional modular multiplication between the integer value and the constant $r^{\alpha n}$. The inverse transformation from the new residue system back to the original representation can be performed by multiplying either of the images with the constant $r^{-\alpha n}$ in modulo M, which can be done using the Montgomery algorithm as explained at the end of this section. That the new multiplication modulo M over the images of U and V results in a image of $U \cdot V \mod M$ can easily be demonstrated as follows.

$$\begin{aligned} & X \cdot Y \cdot r^{-\alpha n} \mod M \\ = & (U \cdot r^{\alpha n}) \cdot (V \cdot r^{\alpha n}) \cdot r^{-\alpha n} \mod M \\ = & (U \cdot V) \cdot r^{\alpha n} \mod M \end{aligned} \tag{5.2}$$

Isomorphism between the original integer set $\mathcal{Z}_{\mathcal{M}}$ with the operation $\times$, and the KT-residue system $\mathcal{Z}'_{\mathcal{M}}$ with the operation $\circledast$, holds as illustrated in Fig. 5.1.

As we will now show, modular multiplication can be efficiently computed using the KT-representation of residue class. Let us consider the multiplier Y split into two parts Y_H and Y_L, i.e. $Y = Y_H \cdot r^{\alpha n} + Y_L$. Then, the multiplication modulo M of the images X and Y can be computed as follows:

$$X \circledast Y = (X \cdot Y_H \bmod M + X \cdot Y_L \cdot r^{-\alpha n} \bmod M) \mod M \tag{5.3}$$

The left term, $X \cdot Y_H \mod M$, can be calculated using the interleaved modular multiplication algorithm that processes the split multiplier Y_H, while the second term, $X \cdot Y_L \cdot r^{-\alpha n} \mod M$, can be calculated using the Montgomery algorithm which processes the other split multiplier Y_L. These two calculations are performed in parallel. Since the split operands Y_H and Y_L are shorter in length than Y, the calculations $X \cdot Y_H \mod M$ and

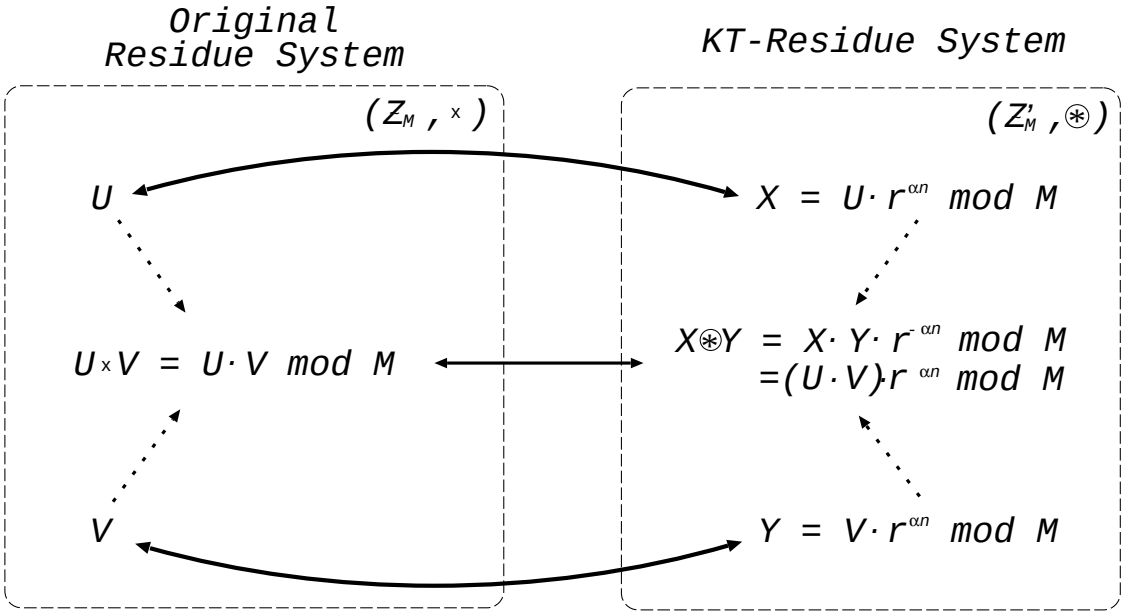

Figure 5.1: *Mapping between the original residue system and the KT-residue system*

$X \cdot Y_L \cdot r^{-\alpha n}$ are performed faster than the individual execution of the interleaved modular multiplication algorithm and the Montgomery algorithm with the unsplit operands.

The correctness of the above formula can be seen using the following equality:

$$\begin{aligned}
&(X \cdot Y_H \bmod M + X \cdot Y_L \cdot r^{-\alpha n} \bmod M) \bmod M \\
&= X \cdot (Y_H \cdot r^{\alpha n} \cdot r^{-\alpha n} + Y_L \cdot r^{-\alpha n}) \bmod M \\
&= X \cdot (Y_H \cdot r^{\alpha n} + Y_L) \cdot r^{-\alpha n} \bmod M \\
&= X \cdot Y \cdot r^{-\alpha n} \bmod M = X \circledast Y
\end{aligned} \tag{5.4}$$

Below is the algorithm that computes modular multiplication using the BMM method. In this algorithm, W is a variable that stores the multiplicand; A and B are variables that store the upper and the lower parts of the multiplier respectively. $Interleaved\_modmul$ (W, A, M) is a function that calculates $W \cdot A \bmod M$ by using the interleaved modular multiplication algorithm. $Montgomery\_modmul$ (W, B, M) is a function that calculates $W \cdot B \cdot r^{-\alpha n} \bmod M$ by using the Montgomery algorithm. $\{C1; C2; \}$ means that two calculations, $C1$ and $C2$, are performed in parallel.

[Algorithm 5.3]

(Bipartite Modular Multiplication)

Input: $M : r^{n-1} < M < r^n$, $\gcd(M, r) = 1$ and $r = 2^k$

$X, Y : 0 \leq X, Y < M$

Output: $Z = X \cdot Y \cdot r^{-\alpha n} \bmod M$

Algorithm:

Step 1: $W := X$; $M := M$; $S := 0$; $T := 0$;

$A := Y_H$; $B := Y_L$ $\quad /* \, Y = Y_H \cdot r^{\alpha n} + Y_L \, */$

Step 2: $\{S := Interleaved\_modmul(W, A, M)$;

$T := Montgomery\_modmul(W, B, M); \}$

Step 3: $Z := (S + T) \bmod M$;

When using the interleaved modular multiplication algorithm and the Montgomery algorithm of similar performance, α can be set to the value so that $\alpha n = \lceil \frac{n}{2} \rceil$; the two split parts of the multiplier, Y_H and Y_L, are at most $\lceil \frac{n}{2} \rceil$-words wide. This means that, it is theoretically possible to obtain a maximum acceleration of twice the speed of the original algorithms performed individually, when these conditions are met. Fig. 5.2 shows the multiplication procedure with the parameter $\alpha = 1/2$.

Transformation of an integer U from the original integer set to the new residue system can be performed by executing $X = Interleaved\_modmul(U, r^{\alpha n}, M)$ or $X = BMM(U, r^{2\alpha n}, M)$ where $r^{2\alpha n}$ is a precomputed constant. The inverse transformation of an image X from the new residue class representation back to the original integer set can be accomplished by executing $U = Montgomery\_modmul(X, 1, M)$ and processing only the αn digits of the multiplier, or by performing $BMM = (X, 1, M)$. When α is set to the value so that $\alpha n = \lceil \frac{n}{2} \rceil$, either of these transformations can be completed theoretically in half the time required by the Montgomery method without the need for precomputed constants. Note also that these transformation can be performed using the bipartite modular multiplication.

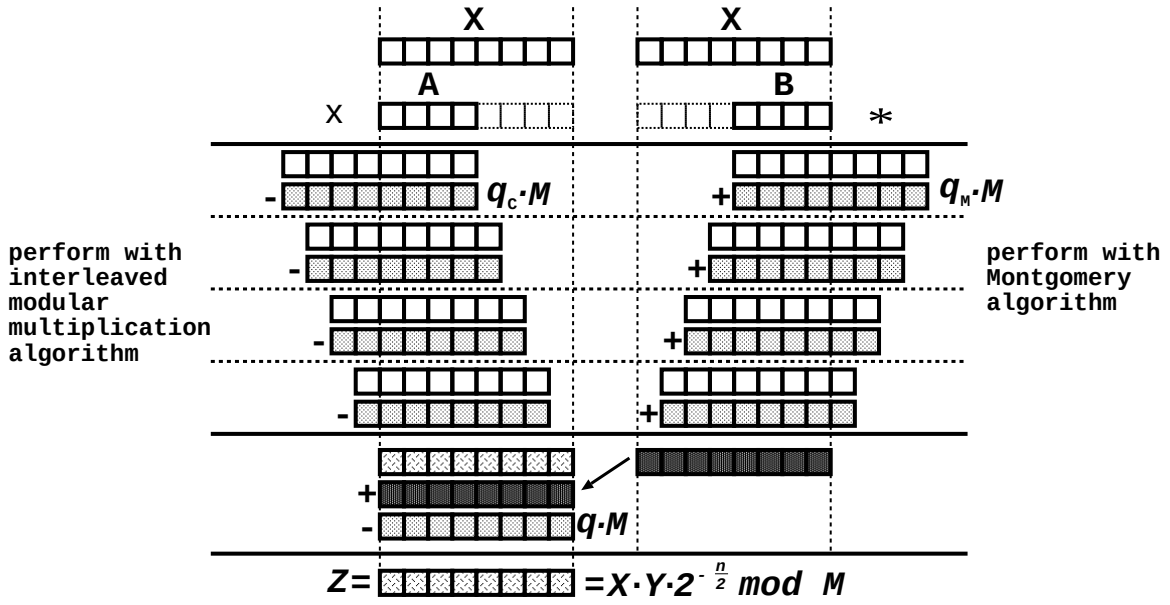

Figure 5.2: *Modular multiplication using BMM method with* $\alpha = 1/2$

5.4 Hardware Implementation

A modular multiplier based on the algorithm presented in the previous section consists of six registers, an interleaved modular multiplier, a digit-serial Montgomery multiplier, a modular adder, and a multiplexer. The registers are: W, which stores the multiplicand; A and B which are shift registers and store the upper and the lower parts of the multiplier respectively; M, which stores the modulus M; and, S and T, which store the partial results. A block diagram of this circuit is shown in Fig. 5.3.

Various implementations of the interleaved modular multiplier and the Montgomery multiplier are possible depending on the techniques used for speeding up the calculation. Most of these techniques use redundant representation and increase the radix, and the different combinations of the multipliers allow for a wide range of trade-offs between speed and hardware requirements.

When a radix-r interleaved modular multiplier is jointly used with a radix-r Montgomery multiplier with similar critical path delays, and n is even, the parameter α can be set to the value $1/2$ and registers of equal length can be used for A and B, thus halving the processing time compared to an individual execution of the interleaved modular

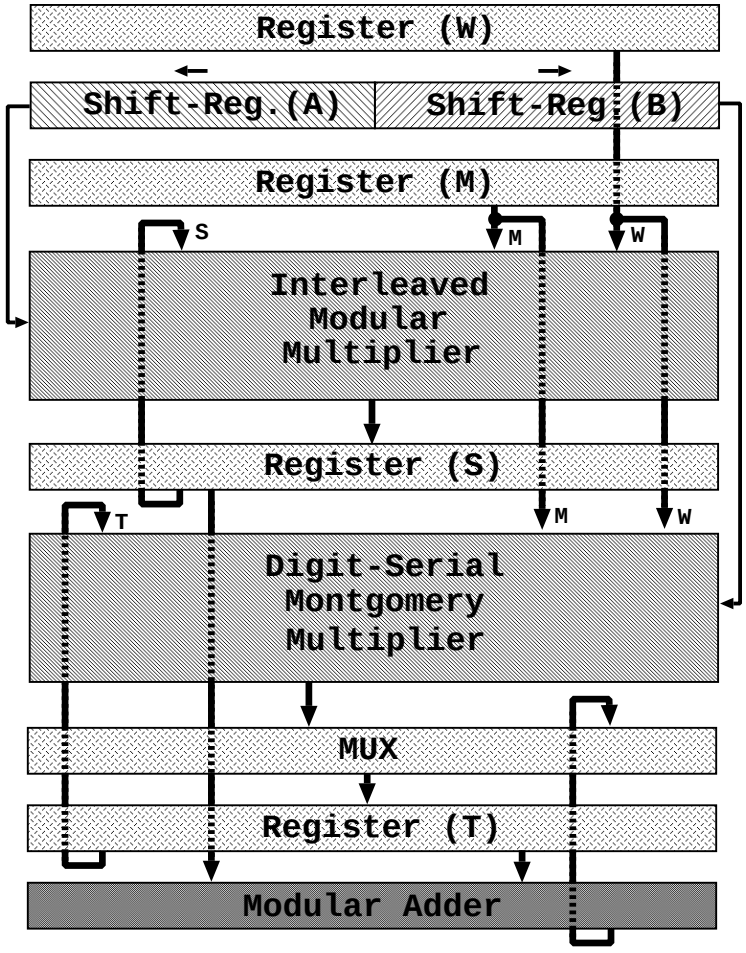

Figure 5.3: *Block diagram of a multiplier*

multiplier or the Montgomery multiplier with the unsplit operands.

Transformation from the ordinary integer set to the new residue class representation can be performed with the same hardware provided that the hardware module which computes the interleaved modular multiplication iterates one extra cycle compared to that required for modular multiplication. Inverse transformation from the new residue class representation back to the original integer set can be performed using the hardware module that computes the Montgomery multiplication.

The value of the parameter α can be displaced around $1/2$ enabling the use of multipliers of different performance. In this case, the multiplier is then split into two parts of unequal length that can be stored in two registers A and B of different length. The value of α can be determined from the difference of performance between the multi-

pliers. For example, if a radix-2 interleaved modular multiplier is used with a radix-4 Montgomery multiplier with similar critical path delays, then α can be set to a number where $\alpha n = \lceil \frac{2}{3} n \rceil$. Then, modular multiplication can be accomplished in about $\lceil \frac{n}{3} \rceil$ clock cycles. Transformation from the original integer set to the new residue system, can be performed with the same hardware by executing the interleaved modular multiplication module twice. In the first execution, an integer U is multiplied by $r^{\lceil \frac{n}{3} \rceil}$. In the second execution, the result of the first execution is multiplied by $r^{\lceil \frac{2}{3} n \rceil - \lceil \frac{n}{3} \rceil}$. Alternatively, the transformation can be performed faster by applying the bipartite modular multiplication to the integer to be transformed and a precomputed constant $r^{\lceil \frac{2}{3} n \rceil 2} \bmod M$. Inverse transformation from the new residue system to the original integer set can be performed by the same hardware by executing the Montgomery multiplication module once.

The amount of hardware of the proposed multiplier is proportional to n. Compared to an individual interleaved modular multiplier or a Montgomery multiplier, the new modular multiplier requires an extra digit modular multiplier, an extra register, a modular adder and related multiplexers.

The space and time trade-offs for high radix modular multiplications based on the classical interleaved algorithm and the Montgomery algorithm are detailed in (Walter 1997). For both algorithms, increasing k, i.e. the number of bits of the radix, to values greater than $\log(n)$, where n is the number of digits, results in a penalty in time for producing the quotient bits q_C or q_M for the next modular reduction in time that makes this approach unattractive. By contrast, by using our algorithm, a speedup can be achieved for such values of radices, since the multiplication and the modular reduction for the split multiplier can be performed by two separate high-radix digit-serial multipliers processing in parallel. Thus, the number of iteration is reduced without increasing the time requirements for each cycle.

Furthermore, as there is no need to increase the radix, the design can remain relatively simple compared to hardware designed using higher radix.

In cryptographic applications, such as in RSA, this approach is much more attractive than implementing two modular multiplier processors separately because it has the advantage of producing the outputs sequentially.

5.5 Pipelined Bipartite Modular Multiplication

In this section we present a hardware algorithm for modular multiplication based on the BMM method. The feature of this hardware algorithm is that it enables the pipelining of the intermediate calculations reducing the clock cycle time. This is accomplished by initially storing copies of the multiplicand into two variables and performing a shift operation and a modular reduction on one variable and a shift operation and a Montgomery reduction on the other variable instead of applying a shift operation and either of the reductions to the intermediate accumulated product. For simplicity, we describe the algorithm for the parameter $\alpha = 1/2$. We assume hereafter that each word is composed by one digit, and that n is an even number. When n is odd, the algorithm that we will describe works by concatenating a 0 digit on the most significant position of the multiplier.

In order to enable the pipelining of the intermediate operations, we need two variables of n digits in length, L and R, which are initialized to the value of the multiplicand X. Then, at each iteration, L is shifted to the left by one digit position and reduced modulo M. Similarly, R is shifted to the right by one digit position and reduced modulo M using the Montgomery reduction. We also need two variables A and B for storing the two split parts of the multiplier. A and B initially store Y_H and Y_L respectively. Then, L and A are used for generating the partial products of $X \cdot Y_H \bmod M$, whereas R and B are used for generating the partial products of $X \cdot Y_L \cdot r^{-\frac{n}{2}} \bmod M$. A and B have $\frac{n}{2}+1$ digits in length. The i-th digit $(i = 0, 1, \cdots, \frac{n}{2})$ of A is denoted by a_i. Namely, $A = \sum_{i=0}^{\frac{n}{2}} a_i \cdot r^i$. Similarly, $B = \sum_{i=0}^{\frac{n}{2}} b_i \cdot r^i$. The digits of A are scanned from the least significant position while in B, the digits are scanned from the most significant position. The scanning procedures can be implemented using shift operations. We use a variable C of n digits in length to store the result of the addition between the generated partial products. A variable D of n digits of length is used to store the intermediate accumulated product.

Below is the radix-r pipelined hardware algorithm based on the BMM method. In the algorithm, $\{0|Y_H\}$ and $\{0|Y_L\}$ mean that a 0 digit has been concatenated to the most significant position of Y_H and Y_L respectively.

[Hardware Algorithm 5.1]

(Pipelined Bipartite Modular Multiplication)

Inputs: $M : r^{n-1} < M < r^n$, $\gcd(M, r) = 1$ and $r = 2^k$

$X, Y : 0 \leq X, Y < M$

Output: $Z = X \cdot Y \cdot r^{-\frac{n}{2}} \bmod M$

Algorithm:

Step 1: $L_0 := X$; $R_0 := X$; $M := M$;

$A := \{0|Y_H\}$; $B := \{0|Y_L\}$;

$/* \; Y = Y_H \cdot r^{\frac{n}{2}} + Y_L \; */$

Step 2:

Step 2-1: $D_0 := 0$;

$T_1^{2-1} : L_1 := L_0 \cdot r \bmod M$;

$R_1 := R_0/r \bmod M$;

$C_0 := (a_0 \cdot L_0 + b_{\frac{n}{2}} \cdot R_0) \bmod M$

Step 2-2: **for** $j := 1$ **to** $\frac{n}{2}$ **do**

$T_1^{2-2} : L_{j+1} := L_j \cdot r \bmod M$;

$R_{j+1} := R_j/r \bmod M$;

$C_j := (a_j \cdot L_j + b_{\frac{n}{2}-j} \cdot R_j) \bmod M$;

$T_2^{2-2} : D_j := (C_{j-1} + D_{j-1}) \bmod M$;

endfor

Step 2-3: $T_2^{2-3} : D_{\frac{n}{2}+1} := (C_{\frac{n}{2}} + D_{\frac{n}{2}}) \bmod M$;

Step 3: $Z := D_{\frac{n}{2}+1}$;

In Step 1, initialization of variables takes place. In Step 2, we process the upper part of the multiplier, i.e. A, from the least significant position and the lower part of the multiplier, i.e. B, from the most significant position. In contrast to the interleaved modular multiplication algorithm and the digit-serial Montgomery multiplication algorithm where the intermediate accumulated product is shifted and reduced modulo M at each iteration, we shift the variables that initially stores the multiplicand instead. In Step 2,

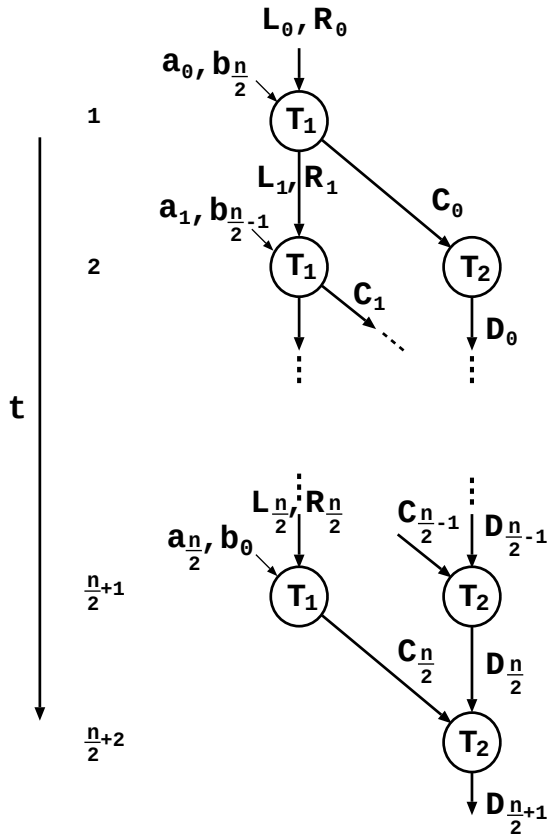

Figure 5.4: *Dependency graph of the tasks in Step* 2 *of Hardware Algorithm 5.1*

two atomic tasks are performed in parallel: Task T_1 executes the shifting of the variable L by one digit position toward the most significant position and the modular reduction on this shifted value. It also executes the shifting of the variable R by one digit position toward the least significant position and the Montgomery reduction on this shifted value. This task also generates the partial products, adds them and applies the modular reduction to this added value. The result is then placed into variable C. Task T_2 executes the addition of the previous result of C to the value of the intermediate accumulated product stored in variable D. Then, modular reduction is applied to the result of this addition and the result is placed into D. One step, i.e. Step 2-1, is required until the two tasks are executed fully in parallel. One extra step, i.e. Step 2-3, is necessary for obtaining the final result placed into D. Step 2-2 is the core of the algorithm. In this step, the two tasks are executed in parallel. The dependency graph for the tasks in Step 2 is depicted in Fig. 5.4.

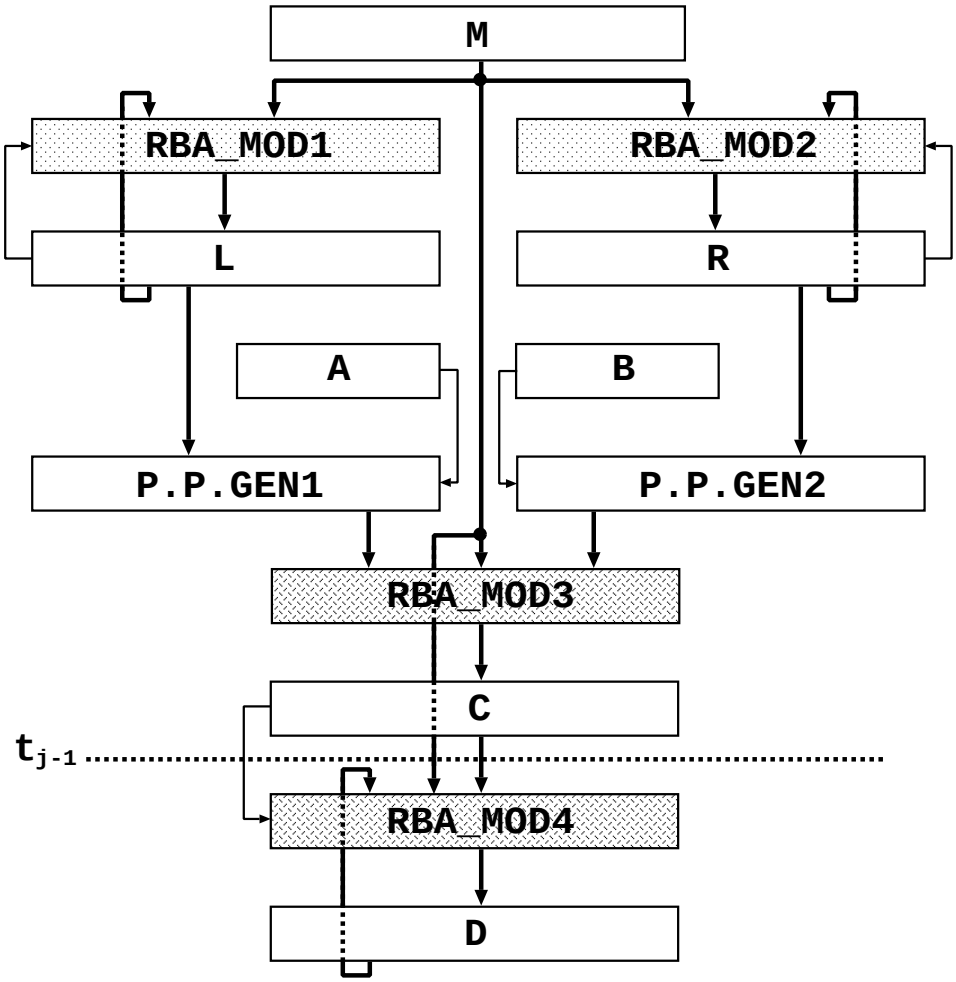

Figure 5.5: *Block diagram of a PBMM multiplier*

In Step 3, the result is obtained from the variable D. If we implement the hardware algorithm so that all the operations contained in one single iteration of each step are executed in one clock cycle, a modular multiplication is calculated in $\frac{n}{2} + 4$ clock cycles.

A modular multiplier based on the new hardware algorithm consists of seven registers and four adders. The rest of the components are multiplexers. A block diagram of a radix-r digit multiplier based on this hardware algorithm is given in Fig. 5.5.

Compared to the conventional interleaved modular multiplication algorithm and the digit-serial Montgomery multiplication, where the modular reduction is performed after shifting the result of the addition between the generated partial product and intermediate accumulated product, the proposed hardware algorithm does not require a shift operation neither on the addition of the generated partial products nor on the accumu-

lation of the partial products. Thus, modular reduction can be applied to the results of these operations independently. This reduces the number of candidates for selecting the multiples of the modulus M required for performing the modular reduction, thus, reducing the critical path delay.

5.6 A Radix-4 Implementation Example

In this section we present a radix-4 implementation example of the hardware algorithm described in the previous section. In this example, the radix $r = 4$. We denote with n' the number of bits required to represent the modulus M. Then, $2^{n'-1} < M < 2^{n'}$ and $n = \lceil n'/2 \rceil$. The binary signed-digit representation (SD2) is employed to all additions and subtractions so that they can be performed without carry propagation. In cryptographic applications, modular multiplication are usually required successively. In order to enable the direct feedback of the output into the inputs and avoid the conversion from the SD2 representation into the binary representation in each multiplication, we represent the inputs and the output using the same SD2 representation. Then, X, Y and Z are SD2 numbers of $n' + 1$ digits in the range $[-M + 1, M - 1]$.

To accelerate the computations, 2-bit signed digit Booth's recoding algorithm (Takagi 1990) is applied to the variables A and B. Note that an $(\frac{n'}{2})$-bit binary multiplier is recoded into a radix-4 number of $\frac{n}{2} + 1$ digits. The i-th recoded digit of A is denoted as $\hat{a}_i$. The i-th recoded digit of B is denoted as $\hat{b}_i$. The recoding rule for the variable B is shown in Table 5.1. In the table, $\bar{1}$ and $\bar{2}$ means -1 and -2 respectively. The same recoding rule can be applied to variable A. Although, it is not shown here, the rule can be further simplified if the direction of the carry propagation in the recoding process is taken into account.

Following the notation of the previous section, the variable L involved in the calculation of $X \cdot Y_H \bmod M$, represents a number $\gamma \in Z_M$ as an SD2 number of $(n' + 1)$ digits which satisfies $-d \cdot M < L < d \cdot M$ and $L \equiv \gamma \pmod{M}$. d is a parameter and can be any number that satisfies $\frac{5}{8} \leq d \leq \frac{2}{3}$. We explain how these values are derived. The shifting procedure and the modular reduction are based on the following recurrence.

Table 5.1: *The recoding rule for B in Step 2*

(a) Phase 1

$b_{2i+1}b_{2i}$	b_{2i-1}	bc_{i+1}	bt_i
$\bar{1}1$	–	$\bar{1}$	1
$\bar{1}0$	non-negative	0	$\bar{2}$
	negative	$\bar{1}$	2
$\bar{1}\bar{1}$	–	$\bar{1}$	1
$0\bar{1}/\bar{1}1$	–	0	$\bar{1}$
00	–	0	0
$01/1\bar{1}$	–	0	1
10	non-negative	1	$\bar{2}$
	negative	0	2
11	–	1	$\bar{1}$

(b) Phase 2

$bt_i \setminus bc_i$	$\hat{b}_i$ $\bar{1}$	0	1
$\bar{2}$	$\times$	$\bar{2}$	$\bar{1}$
$\bar{1}$	$\bar{2}$	$\bar{1}$	0
0	$\bar{1}$	0	1
1	0	1	2
2	1	2	$\times$

$\times$: Never occurs

$$L_{j+1} := 4 \cdot L_j - \hat{p}_j \cdot M$$

We select $\hat{p}_j$ from $\{\bar{2}, \bar{1}, 0, 1, 2\}$ so that modular reduction can be performed by adding 0 or $\pm M$ or $\pm 2M$ which can be obtained from shifts and complements of the modulus. In order that L_j stays in the range $-d \cdot M < L < d \cdot M$, the following inequality must hold:

$$4 \cdot d \cdot M - 2 \cdot M \leq d \cdot M$$

From this inequality, $d \leq \frac{2}{3}$. Now, we calculate how many digits we need to examine for determining $\hat{p}_j$. Assume that we compare $4 \cdot L_j$ with $\pm\frac{M}{2}$ and $\pm\frac{3}{2}M$ down to the k-th position. We need to find the largest k that satisfies the following inequality:

$$2 \cdot 2^k \leq (d - \tfrac{1}{2}) \cdot M$$

Since M can be $2^{n'-1} + 1$, the inequality $k \leq n' - 2 + \log_2(d - \frac{1}{2})$ must hold. Since $d \leq \frac{2}{3}$, k is at most $n' - 5$. Conversely, k is $n' - 5$ when $d \geq \frac{5}{8}$. The Robertson's diagram for the modular reduction procedure can be found in Fig. 5.6.

We will now explain the procedure of calculating the modular reduction. The selection of $\hat{p}_j$ from $\{\bar{2}, \bar{1}, 0, 1, 2\}$ is performed by comparing $4 \cdot L_j$ with $\pm\frac{M}{2}$ and $\pm\frac{3}{2}M$ down to the $(n-5)$-th position. If we define $top(W)$ as a function that evaluates the most significant digits down to the $(n-5)$-th position of a number W, then the rule for selecting the values for $\hat{p}_j$ is as follows.

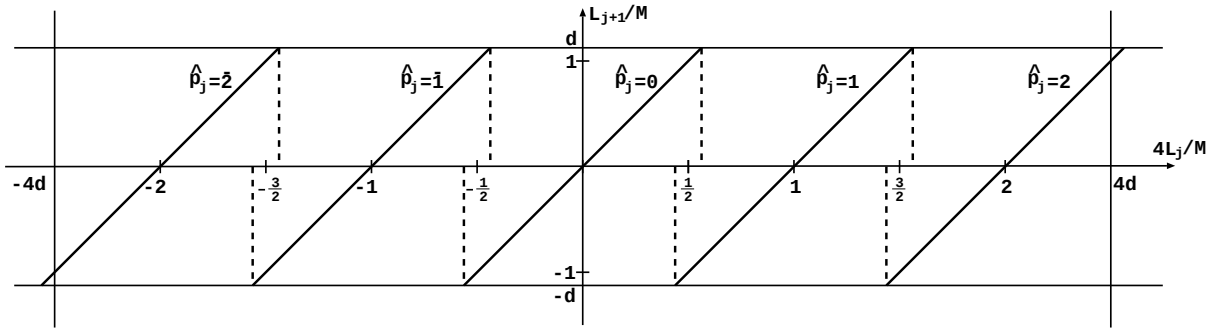

Figure 5.6: *Robertson's diagram for the modular reduction on L*

$$\hat{p}_j = \begin{cases} \bar{2} & \text{if} \quad top(L_j) < -top(\frac{3}{2}M) \\ \bar{1} & \text{if} \quad -top(\frac{3}{2}M) \leq top(L_j) < -top(\frac{M}{2}) \\ 0 & \text{if} \quad -top(\frac{M}{2}) \leq top(L_j) < top(\frac{M}{2}) \\ 1 & \text{if} \quad top(\frac{M}{2}) \leq top(L_j) < top(\frac{3}{2}M) \\ 2 & \text{if} \quad top(L_j) \geq top(\frac{3}{2}M) \end{cases}$$

$top(L_j)$ can be calculated from the most significant 6 digits of L_j. We can calculate $top(\frac{M}{2})$ from the most significant 4 bits of M. We can calculate $top(\frac{3}{2}M)$ from the most significant 5 bits of M so that $|\frac{3}{2}M - top(\frac{3}{2}M)| \leq 2^{n'-4}$. Since M is a binary number, the addition of $\pm M$ or $\pm 2M$ for obtaining L_{j+1} is simpler than the conventional SD2 addition. For details, see (Takagi 1990).

The variable R involved in the calculation of $X \cdot Y_L \cdot 4^{-\frac{n}{2}} \bmod M$ represents an SD2 number of $n' + 2$ digits which satisfies $-2M < R < 2M$. The shifting operation and the Montgomery reduction, i.e. the calculation of $R/4 \bmod M$, is performed by using the function $MQRTR(R, M)$. For the details of how this function is implemented, see Section 3.3.1.

For the addition of the partial products. i.e. $C_j := (a_j \cdot L_j + b_{\frac{n}{2}-j} \cdot R_j) \bmod M$, we use a more redundant representation to make the modular reduction procedure simple. The variable C that stores the result of the addition after performing the modular reduction represents an SD2 number of $n' + 2$ digits which satisfies $-2M < C < 2M$. After adding the partial products, we add $2M$ or 0 or $2M'$, accordingly as the value of the number

formed by the three most significant digits of $\hat{a}_j \cdot L_j + \hat{b}_j \cdot R_j$ is negative or zero or positive. $M' = [\bar{1}0m'_{n'-2}...m'_1 1]$ 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 $-M$. For the details of this modular addition procedure, see (Takagi and Yajima 1992).

The result of the previous addition is added to the intermediate accumulated product stored in variable D. D also represents an SD2 number of $(n'+2)$ digits which satisfies $-2M < D < 2M$. Modular reduction is performed in the same way as described for the addition of the partial products. The result obtained in Step 3 can be reduced into the range $-M < D < M$ by executing a shift operation and a modular reduction using the same rule. Then, the final result can be obtained from the most significant $n'+1$ digits of D.

The radix-4 version of the hardware algorithm described here can be implemented using four redundant modular adder, where two of them are simpler, three registers for storing SD2 numbers, one SD2 shift register for storing the intermediate accumulated product, one register for storing the modulus M and two SD2 shift registers for storing the two split parts of the multiplier. Taking into account that the SD2 adder can be implemented with carry save adders (Kornerup 2002), and that the modular reduction procedure used in our implementation requires an extra delay of one AND gate, a modular multiplier based on the proposed hardware algorithm can be implemented with similar performance to a radix-16 pipelined Montgomery multiplier (Orup 1995).

In our implementation, the modular reduction of the variables that initially store the multiplicand can be performed in parallel to the addition of the generated partial products without increasing the clock cycle time. This enables the reduction of the number of stages for pipelining, and thus, the number of registers to latch the intermediate results between the stages. For obtaining similar performance, the proposed modular multiplier can be implemented with two less stages for pipelining and requires one less redundant adder, three less n-digit registers for storing redundant binary numbers and less complex multiplexers compared to the radix-16 modular multiplier of (Orup 1995).

With our implementation, transformation back and forth between the ordinary integer set and KT-residue class representation can be performed with the same hardware.

Furthermore, precomputation of the modulus is no longer necessary and postprocessing is simplified.

5.7 Concluding Remarks

In this chapter, we have presented a fast method for computing modular multiplication called bipartite modular multiplication (BMM). We have defined a residue class representation called KT-residue which enables the splitting of the multiplier into two parts which can be processed by using the interleaved modular multiplication algorithm and the Montgomery algorithm in parallel, potentially doubling the speed. Transformations back and forth between the original integer set and the new residue system can be performed at a maximum of twice the speed of the Montgomery method without the need for precomputed constants. The dual processing makes it suitable for software implementation in a multiprocessor environment as well as for hardware implementation as discussed in Section 5.4.

We have also presented a fast hardware algorithm for calculating modular multiplication based on the BMM method. The addition of the partial products to the intermediate accumulated product is pipelined in order to reduce the critical path delay. We have presented a radix-4 implementation example of the hardware algorithm as an efficient alternative for calculating modular multiplication.

Chapter 6

Conclusion

In this study, we have presented hardware algorithms for modular arithmetic focusing on modular multiplication and division. It has been shown that taking advantage of similarities and symmetries is a good technique for reducing hardware requirement and for speeding up the calculations.

Similarities between modular multiplication and division algorithms can be exploited in order to develop new hardware algorithms which can calculate both operations with much less hardware amount than that required when these operations are implemented separately. This is the case of the hardware algorithm for modular multiplication/division based on the Binary GCD algorithm presented in Chapter 3, and the hardware algorithm for modular multiplication/division based on the extended Euclidean algorithm presented in Chapter 4. Similarities are not always evident at a first glance and further modification is required as in the case of combining the extended Euclidean algorithm with the Montgomery algorithm.

The symmetry between algorithms can also be used to develop fast calculation methods. The Bipartite Modular Multiplication method (BMM) presented in Chapter 5 was inspired in the symmetry between the interleaved modular multiplication algorithm and the Montgomery algorithm. This enables the multiplier to be split into two parts which can then be processed in parallel, reducing calculation time. Pipelining of the calculation is also possible when the BMM method is used and the two parts of the split multiplier are processed in reverse order. This reduces the critical path delay improving the overall performance of the multiplier.

Throughout this study, we used redundant binary signed digit (SD2) representation

in additions/subtractions to avoid carry propagation. The use of the SD2 representation enables the application of the techniques developed in (Takagi 1990) and (Takagi and Yajima 1992) to speed up the modular reduction with reduced hardware amount. Also, with this representation, systematic analysis of convergence in recurrence equations can easily be made as the one performed in Section 5.6 of Chapter 5.

In the future, we can expect more connectivity between portable devices, and these devices may carry money as a digital form, reinforcing the need of efficient public-key cryptosystems. The results of this thesis contribute to the process of miniaturization and improvement of efficiency in public-key cryptosystems.

Bibliography

Bajard, J.-C., Didier, L.-S. and Kornerup, P.: 1998, An rns montgomery modular multiplication algorithm, *IEEE Trans. Computers* **47**(7), 766–776.

Barrett, P.: 1987, Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor, *in* A. Odlyzko (ed.), *Advances in Cryptology - CRYPTO'86 (LNCS)*, Vol. 263, Springer, pp. 311–323.

Blakley, G. R.: 1983, A computer algorithm for calculating the product *ab* modulo *m*, *IEEE Trans. Computers* **C-32**(5), 497–500.

Brent, R. P. and Kung, H. T.: 1983, Systolic vlsi array for linear-time gcd computation, *in* F. Anceau and E. J. Aas (eds), *Proc. VLSI'83*, Elsevier Science Publishers, pp. 145–154.

Brickell, E. F.: 1983, A fast modular multiplication algorithm with application to two key cryptography, *Advances in Cryptology, Proc. CRYPTO'82* **C-32**(5), 497–500.

Coron, J.-S.: 1998, Resistance against differential power analysis for elliptic curve cryptosystems, *Proc. Workshop Cryptographic Hardware and Embedded Systems*, pp. 292–302.

Eldridge, S. E. and Walter, C. D.: 1993, Hardware implementation of montgomery's modular multiplication algorithm, *IEEE Trans. Computers* **42**(6), 693–699.

ElGamal, T.: 1985, A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. Information Theory* **IT-31**(4), 469–472.

Freking, W. L. and Parhi, K. K.: 2000, 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.

Kaihara, M. E. and Takagi, N.: 2003, A vlsi algorithm for modular multiplication/division, *Proc. 16th IEEE Symp. Comp. Arith.*, pp. 220–227.

Kaihara, M. E. and Takagi, N.: 2005a, Bipartite modular multiplication, *Proc. Cryptographic Hardware and Embedded Systems - CHES 2005 (LNCS 3659)*, pp. 201–210.

Kaihara, M. E. and Takagi, N.: 2005b, A hardware algorithm for modular multiplication/division, *IEEE Trans. Computers* **54**(1), 12–21.

Kaihara, M. E. and Takagi, N.: 2005c, A hardware algorithm for modular multiplication/division based on the extended euclidean algorithm, *IEICE Trans. Fundamentals* **E88-A**(12), 1–8.

Kawamura, S., Koike, M., Sano, F. and Shimbo, A.: 2000, Cox-rower architecture for fast parallel montgomery multiplication, *Advances in Cryptology - EUROCRYPT 2000 (LNCS 1807)*, pp. 523–538.

Knuth, D. E.: 1998a, *The Art of Computing Programming, Volume 2, Seminumerical Algorithms*, third edn, Reading Mass.: Addison-Wesley, p. 271.

Knuth, D. E.: 1998b, *The Art of Computing Programming, Volume 2, Seminumerical Algorithms*, third edn, Reading Mass.: Addison-Wesley, pp. 523–538.

Koblitz, N.: 1987, Elliptic curve cryptosystems, *Math. of Computation* **48**(177), 203–209.

Koç, Ç. K., Acar, T. and Jr., B. S. K.: 1996, Analyzing and comparing montgomery multiplication algorithms, *IEEE Micro* **16**(3), 26–33.

Kocher, P. C.: 1996, Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems, *Advances in Cryptology - CRYPTO '96 (LNCS 1109)*, pp. 104–113.

Kocher, P., Jaffe, J. and Jun, B.: 1999, Differential power analysis, *Proc. Advances in Cryptology-CRYPTO '99*, pp. 388–398.

Kornerup, P.: 1993, High-radix modular multiplication for cryptosystems, *in* G. Jullien, M. J. Irwin and E. Swartzlander (eds), *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 277–283.

Kornerup, P.: 2002, Reviewing 4-to-2 adders for multi-operand addition, *Proc. The IEEE Int. Conf. Application-Specific Systems, Architectures, and Processors, ASAP 2002*, pp. 218–229.

Miller, V.: 1985, Use of elliptic curves in cryptography, *Proc. Advances in Cryptography-CRYPTO '85*, Vol. 218, pp. 417–426.

Montgomery, P.: 1985, Modular multiplication without trial division, *Mathematics of Computation* **44**, 519–521.

Morita, H.: 1990, A fast modular multiplication algorithm with application to two key cryptography, *in* G. Brassard (ed.), *Proc. Advances in Cryptology – CRYPTO'89 (LNCS)*, Vol. 435, Germany: Springer-Verlag, pp. 387–399.

Orup, H.: 1995, Simplifying quotient determination in high-radix modular multiplication, *in* S. Knowles and W. H. McAllister (eds), *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 193–199.

Oswald, E. and Aigner, M.: 2001, Randomized addition-subtraction chains as a countermeasure against power attacks, *in* D. N. Ç. K. Koç and C. Paar (eds), *Cryptographic Hardware and Embedded Systems - CHES 2001 (LNCS 2162)*, pp. 39–50.

Parikh, S. N. and Matula, D. W.: 1991, A redundant binary euclidean gcd algorithm, *Proc. 10th Symp. Computer Arithmetic*, pp. 220–224.

Posch, K. C. and Posch, R.: 1995, Modulo reduction in residue number systems, *IEEE Trans. on Parallel and Distributed Systems* **6**(5), 449–454.

Rivest, R. L., Shamir, A. and Adleman, L.: 1978, A method for obtaining digital signatures and public-key cryptosystems, *Commun. ACM* **21**(2), 120–126.

Sloan, K. R.: 1985, Comments on 'a computer algorithm for calculating the product *ab* modulo *m*', *IEEE Trans. Computers* **C-34**(3), 290–292.

Stinson, D. R.: 1995, *Cryptography Theory and Practice*, CRC Press, p. 127.

Takagi, N.: 1990, A radix-4 modular multiplication hardware algorithm for modular exponentiation, *IEEE Trans. Computers* **41**(8), 949–956.

Takagi, N.: 1996, A hardware algorithm for modular division based on the extended euclidean algorithm, *IEICE Trans. Information and Systems* **E79-D**(11), 1518–1522.

Takagi, N.: 1998, A vlsi algorithm for modular division based on the binary gcd algorithm, *IEICE Trans. Fundamentals* **E81-A**(5), 724–728.

Takagi, N. and Yajima, S.: 1992, Modular multiplication hardware algorithms with a redundant representation and their application to rsa cryptosystem, *IEEE Trans. Computers* **41**(7), 887–891.

Tenca, A. F., Todorov, G. and Koç, Ç. K.: 2001, High-radix design of a scalable modular multiplier, *in* Ç. K. Koç, D. Naccache and C. Paar (eds), *Cryptographic Hardware and Embedded Systems - CHES 2001 (LNCS)*, pp. 185–201.

Walter, C. D.: 1991, Faster modular multiplication by operand scaling, *Advances in Cryptology - CRYPTO '91 (LNCS)*, Vol. 576, Springer, pp. 313–323.

Walter, C. D.: 1993, Systolic modular multiplication, *IEEE Trans. Computers* **42**(3), 376–378.

Walter, C. D.: 1997, Space/time trade-offs for higher radix modular multiplication using repeated addition, *IEEE Trans. Computers* **46**(2), 139–141.

Author's Publications

Refereed Papers

1. Kaihara, M.E., Takagi, N., "A Hardware Algorithm for Modular Multiplication/Division Based on the Extended Euclidean Algorithm," IEICE Transactions on Fundamentals, vol. E88-A, No.12, Dec. 2005.

2. Kaihara, M.E., Takagi, N., "Bipartite Modular Multiplication," Proceedings Cryptographic Hardware and Embedded Systems - CHES 2005, LNCS 3659, Springer, 201-210, Aug./Sep. 2005.

3. Kaihara, M.E., Takagi, N., "A Hardware Algorithm for Modular Multiplication/Division," IEEE Transactions on Computers, vol.54, No. 1, Jan. 2005.

4. Kaihara, M.E., Takagi, N., "A VLSI Algorithm for Modular Multiplication/Division," Proceedings 16th IEEE Symp. Computer Arithmetic (ARITH-16), pp.220-227, Jun. 2003.

Technical Reports

1. Kaihara, M.E., Takagi N., "Pipelined Bipartite Modular Multiplier," Technical Report of IEICE, VLD2005-61$\tilde{7}$5, vol.105, No.442, pg. 37-42., Dec. 2005.

2. Kaihara, M.E., Takagi N., "Fast Modular Multiplication by Processing the Multiplier from Both Sides in Parallel," Technical Report of IEICE, VLD2004-124, pg. 1-4, Mar. 2005.

3. Kaihara, M.E., Takagi N., "A Multiplier/Divider for Modular Arithmetic Based on the Extended Euclidean Algorithm," Technical Report of IEICE, VLD2004-1, vol.104, No.78, pg. 1-6., May 2004.

4. Kaihara, M.E., Takagi N., "A Multiplication Division VLSI Algorithm for Modular Arithmetic," LA Symposium, Evolutionary Advancement in Fundamental Theories of Computer Science, pg. 201-207, May 2004.

5. Kaihara, M.E., Takagi N., "A Modular Multiplication/Division Algorithm for VLSI," CS Sessions at 2003 IEICE Gen. Conf., Mar. 2003.

6. Kaihara, M.E., Takagi N., "A Modulo M Multiplier/Divider," Technical Report of IEICE, VLD2002-109, vol. 102, No. 476, pg. 163-168, Nov. 2002.