アルゴリズム 43－4
（1995．1．23）

剰余除算用ハードウェアアルゴリズムについて

高木 直史

名古屋大学工学部情報工学科

剰余除算用のハードウェアアルゴリズムを提案する。拡張ユークリッド互除法を、剰余除算用にさらに拡張する。内部計算に各桁が $\{0,1,-1\}$ の要素からなる冗長2進表現を用い、加算を桁上げの伝搬なしに高速に行う。n を演算数の語長とするとき、n に比例するクロックサイクルで剰余除算を行える。各クロックサイクルの長さは n によらず一定である。提案アルゴリズムに基づく剰余除算器は、ビットスライス式の規則正しいセル配列構造をしており、VLSI化に適している。ハードウェア量は n に比例する。

On a Hardware Algorithm for Modular Division

Naofumi TAKAGI

Department of Information Engineering, Nagoya University

A hardware algorithm for modular division is proposed. The extended Euclidean algorithm is further extended to perform modular division. The redundant binary representation with a digit set $\{0,1,-1\}$ is employed so that additions are performed without carry propagation. A modular division is carried out in $O(n)$ clock cycles where n is the word length of operands. The length of each clock cycle is constant independent of n. A modular divider based on the algorithm has a regular cellular array structure with a bit slice feature and is very suitable for VLSI implementation. Its amount of hardware is proportional to n.

1 Introduction

Modular division in a residue class field is one of the basic operations in abstract algebra. It plays important roles in several applications, such as public-key cryptosystems. For example, we can accelerate decryption in ElGamal public-key system [1], by developing a fast modular divider with a large modulus (e.g., longer than 500-bit).

In this paper, we propose a hardware algorithm for modular division. We further extend the extended Euclidean algorithm (EEA) [2]. In EEA, a procedure for finding the multiplicative inverse is intertwined with that for calculating the greatest common divisor (GCD). We modify the procedure for finding the multiplicative inverse to calculate the quotient. Furthermore, we accelerate the calculation by using a redundant binary (RB) representation with a digit set $\{0, 1, -1\}$. Using RB representation, we can perform addition/subtractions fast without carry propagation.

Recently, we proposed a hardware algorithm for modular inversion based on EEA with RB representation [3]. The algorithm proposed in this paper is an extension of this algorithm. Parikh et al. independently proposed an algorithm for computing the GCD of given two integers based on Euclidean algorithm in which RB representation is also used [4]. Although they suggested that their algorithm can be extended for modular inversion, they did not show any procedure for modular inversion. They mentioned nothing on modular division.

The algorithm proposed in this paper performs modular division within $O(n)$ clock cycles where n is the word length of the operands. The length of each clock cycle is constant independent of n. A modular divider based on the algorithm has a regular cellular array structure with a bit slice feature and is very suitable for VLSI implementation. Its amount of hardware is proportional to n. It seems easy to fabricate a modular divider VLSI based on the proposed algorithm using today's technology.

In the next section, we further extend the extended Euclidean algorithm to perform modular division. In Section 3, we introduce the redundant binary representation and show procedures to perform several modular operations in the redundant binary system. We propose a hardware algorithm for modular division in Section 4. In Section 5, we analyze the number of clock cycles required by the algorithm.

2 Further Extension of EEA

We consider the modular division $X/Y \pmod M$ for positive integers X and Y ($X, Y < M$) in the residue class field with modulus M where M is prime. Namely, we intend to obtain a positive integer Z ($< M$) which satisfies $Z \times Y \equiv X \pmod M$. Although we are assuming the modulus M being prime, the algorithm to be proposed works as long as M is odd and relatively prime with Y.

The algorithm is based on the extended Euclidean algorithm (EEA) [2]. We further extend EEA so that we can perform modular division. EEA consists of two intertwined parts. One is for calculating the greatest common divisor (GCD) of the modulus and the given number and the other is concerned with finding the multiplicative inverse of the number. We modify the procedure for finding the multiplicative inverse to calculate the quotient modulo M. We also make a slight modification for adopting the redundant binary (RB) representation.

The further extended EEA is as follows. Note that $\gcd(M, Y) = 1$.

Algorithm [EEEA]
(Inputs)
  Modulus: M (a prime number)
  Dividend: X (a positive integer, $X < M$)
  Divisor: Y (a positive integer, $Y < M$)
(Output)
  Quotient: Z (a positive integer, $Z < M$, $Z \times Y \equiv X \pmod M$)
(Algorithm)
Step 1:
  $A_0 := M$, $A_1 := Y$, $U_0 := 0$, $U_1 := X$,
  $k := 0$
Step 2:
  while $|A_{k+1}| \neq 1$ do
  begin
    $k := k + 1$
    Choose Q_k so that $|A_{k-1} - A_k \cdot Q_k| < |A_k|$.
    $A_{k+1} := A_{k-1} - A_k \cdot Q_k$
    Choose C_k so that $|U_{k-1} - U_k \cdot Q_k - C_k \cdot M| < M$.
    $U_{k+1} := U_{k-1} - U_k \cdot Q_k - C_k \cdot M$
  end
Step 3:
  if $A_{k+1} = -1$ then $Z' := -U_{k+1}$
  else $Z' := U_{k+1}$
Step 4:
  if $Z' < 0$ then $Z := Z' + M$
  else $Z := Z'$ □

The procedure for calculating GCD is the same as that of EEA except that A_{k+1} may be negative. We choose Q_k so that A_{k+1} satisfies $|A_{k+1}| < |A_k|$, while in EEA, the quotient of $A_{k-1} \div A_k$ is chosen as Q_k and A_{k+1} satisfies $0 \leq A_{k+1} < A_k$. This modification allows us to adopt RB representation into this procedure. $|A_{k+1}|$ for the final k is $\gcd(M, Y)$, which is 1 in our case.

The procedure for calculating the multiplicative inverse is modified so that it calculates the quotient. We set U_1 being X, while it is set being 1 in EEA. We allow U_{k+1} to be negative so that we can also adopt RB representation into this procedure. Z is the quotient of X/Y modulo M. (Note that when $X = 1$, Z is the multiplicative inverse of Y modulo M.) To prove this fact, we first give the following lemma. Hereafter, kf denotes the final k.

[Lemma 1]
$U_k \cdot Y + W_k \cdot M = A_k \cdot X$ holds for any k $(0 \leq k \leq kf + 1)$, where $W_0 = X$, $W_1 = 0$ and $W_{k+1} = W_{k-1} - W_k \cdot Q_k + C_k \cdot Y$.
(Proof)
We prove the lemma by induction on k.
$U_0 \cdot Y + W_0 \cdot M = A_0 \cdot X$ and $U_1 \cdot Y + W_1 \cdot M = A_1 \cdot X$ hold, since $A_0 = M$, $A_1 = Y$, $U_0 = 0$, $U_1 = X$, $W_0 = X$, and $W_1 = 0$.
Assume that $U_{k-1} \cdot Y + W_{k-1} \cdot M = A_{k-1} \cdot X$ and $U_k \cdot Y + W_k \cdot M = A_k \cdot X$ hold for a k $(k \geq 1)$. Then, subtracting Q_k times the latter equation from the former, we get $(U_{k-1} - U_k \cdot Q_k) \cdot Y + (W_{k-1} - W_k \cdot Q_k) \cdot M = (A_{k-1} - A_k \cdot Q_k) \cdot X$. Therefore, $(U_{k+1} + C_k \cdot M) \cdot Y + (W_{k+1} - C_k \cdot Y) \cdot M = A_{k+1} \cdot X$ holds, and hence, $U_{k+1} \cdot Y + W_{k+1} \cdot M = A_{k+1} \cdot X$ holds.
Hence, the lemma has been proved. □

Now, we get the following theorem.

[Theorem 1]
Z obtained by [EEEA] is the quotient, $X/Y \pmod M$.
(Proof)
$U_{kf+1} \cdot Y + W_{kf+1} \cdot M = A_{kf+1} \cdot X$ (from [Lemma 1]), and $|A_{kf+1}| = 1$ $(= \gcd(M, Y))$. When $A_{kf+1} = -1$, $U_{kf+1} \cdot Y + W_{kf+1} \cdot M = -X$ and $Z' = -U_{kf+1}$. When $A_{kf+1} = 1$, $U_{kf+1} \cdot Y + W_{kf+1} \cdot M = X$ and $Z' = U_{kf+1}$. Therefore, $Z' \times Y \equiv X \pmod M$ holds. $|Z'| < M$, because $|Z'| = |U_{kf+1}|$ and $|U_{kf+1}| < M$. When $Z' < 0$, $Z = Z' + M$, and otherwise, $Z = Z'$. Therefore, $Z \times Y \equiv X \pmod M$ and $0 < Z < M$ hold.
Hence, the theorem has been proved. □

Fig. 1 shows a flow of the modular division of $79/108 \pmod{211}$ by [EEEA]. $|A_4| = 1$ $(= \gcd(211, 108))$. Since $A_4 = 1$, $Z' = U_4 = 116$ and since $Z' > 0$, $Z = Z' = 116$. 116 is the quotient. In fact, $116 \times 108 \equiv 79 \pmod{211}$.

$M = 211$, $X = 79$, $Y = 108$

Q_k		A_k		U_k	
		$A_0 =$	211	$U_0 =$	0
		$A_1 =$	108	$U_1 =$	79
$Q_1 =$	2				
		$A_2 =$	−5	$U_2 =$	−158
$Q_2 =$	−22				
		$A_3 =$	−2	$U_3 =$	−21
$Q_3 =$	3				
		$A_4 =$	1	$U_4 =$	116

$Z' = 116$ $(= U_4)$
$Z = 116$ $(= Z')$

Figure 1: A modular division by [EEEA]

3 Modular Operations in RB System

In the algorithm to be proposed, we represent the intermediate results in [EEEA], A_k's and U_k's, in the redundant binary (RB) representation.

RB representation has a fixed radix 2 and a digit set $\{\bar{1}, 0, 1\}$, where $\bar{1}$ denotes -1. An n-digit RB integer $A = [a_1 a_2 \cdots a_n]$ $(a_i \in \{\bar{1}, 0, 1\})$ has the value $\sum_{i=1}^{n} a_i \cdot 2^{n-i}$. Note that there may be several RB numbers which have the same value. Using this redundancy, we can perform addition of two RB numbers without carry propagation.

Let us consider RB addition of $S := A + B$ where $A = [a_1 \cdots a_n]$, $B = [b_1 \cdots b_n]$ and $S = [s_0 s_1 \cdots s_n]$. A carry-propagation-free addition is carried out through two steps. In the first step, we determine the intermediate carry ac_i and the intermediate sum ad_i at each position. In the second step, we add up ad_i and the carry ac_{i+1} from the next lower position and obtain the final sum s_i at each position. In the first step, we determine ac_i and ad_i so that no carry generates in the second step, by looking into the next lower position. Table 1 shows the addition rule. For the details of carry-propagation-free addition, see, e.g., [5].

We can get a negation of an RB number by changing the signs of all nonzero digits in it.

We need no computation for binary to RB conversion, because a binary number itself is an RB

Table 1: The computation rule in the RB addition

Step 1

$a_i\ b_i$ \ ac_i, ad_i	$\bar{1}$	0	1
$\bar{1}$	$\bar{1},0$	$*0,\bar{1}/\bar{1},1$	0,0
0	$*0,\bar{1}/\bar{1},1$	0,0	$*1,\bar{1}/0,1$
1	0,0	$*1,\bar{1}/0,1$	1,0

*: Both a_{i+1} and b_{i+1} are non-negative./ Otherwise.

Step 2

$ad_i\ ac_{i+1}$ \ s_i	$\bar{1}$	0	1
$\bar{1}$	×	$\bar{1}$	0
0	$\bar{1}$	0	1
1	0	1	×

×: Never occurs.

number. On the other hand, we need a carry-propagate addition for RB to binary conversion. In the conversion of an RB number A, we calculate $A^+ - A^-$ where A^+ and A^- are binary numbers formed from the positive and the negative digits of A, respectively.

In the algorithm to be proposed, we need several modular operations on U_k's. Here, we show procedures to perform addition, doubling and halving modulo M in RB system. Let the modulus $M = [1m_2 \cdots m_n]$ $(m_i \in \{0,1\})$ be an n-bit binary (prime) number. $(2^{n-1} < M < 2^n)$ Let $U = [u_0 u_1 \cdots u_n]$, $V = [v_0 v_1 \cdots v_n]$ and $T = [t_0 t_1 \cdots t_n]$ $(u_i, v_i, t_i \in \{\bar{1}, 0, 1\})$ be (n+1)-digit RB integers satisfying $-M < U, V, T < M$.

We perform modular addition $T := MADD(U, V, M)$, i.e. the calculation of T such that $T \equiv U + V \pmod{M}$, through two steps [6]. In the first step, we calculate $S := U + V$ in RB system according to the addition rule shown above. $S = [s_{-1} s_0 s_1 \cdots s_n]$ is an (n+2)-digit RB number. In the second step, we add M or 0 or $M' + 1$ to S, accordingly as the value of the three most significant digits of S, $sv = [s_{-1} s_0 s_1]$, is negative or zero or positive. $M' = [\bar{1}0 m'_2 ... m'_n]$ is the (n+1)-digit RB number where m'_i is 1 or 0 accordingly as m_i is 0 or 1, and has the value $-M - 1$. Note that $-2M < S < 0$ when $sv < 0$, $-2^{n-1} < S < 2^{n-1}$ when $sv = 0$, and $0 < S < 2M$ when $sv > 0$. We also perform this addition in RB system. Since all addend digits except the most significant one are non-negative, the addition in this step is simpler. Table 2 shows the computation rule for this addition. The addend digit, r_i $(\in \{0,1\})$, is m_i or 0 or m'_i, accordingly as sv is negative or zero or positive. We let ad_0 be $2s_{-1} + s_0$ or 0 or $2s_{-1} + s_0 - 1$ also according to sv. We let ac_{n+1} be 1 when sv is positive.

Table 2: The computation rule in the simpler RB addition

Step 1

$s_i\ r_i$ \ ac_i, ad_i	0	1
$\bar{1}$	$0,\bar{1}$	0,0
0	0,0	$1,\bar{1}$
1	$1,\bar{1}$	1,0

Step 2

$ad_i\ ac_{i+1}$ \ t_i	0	1
$\bar{1}$	$\bar{1}$	0
0	0	1

We can perform modular doubling $T := MDBL(V, M)$, i.e. the calculation of T such that $T \equiv 2V \pmod{M}$, by applying the second step of the modular addition to $2V$. We can obtain $2V$ by shifting V with one position to the left.

We perform modular halving $T := MHLV(V, M)$, i.e. the calculation of T such that $T \equiv V/2 \pmod{M}$, through two steps. In the first step, we add M to V when V is odd, i.e. when $v_n \neq 0$. We use the simpler addition rule shown in Table 2. We do nothing when V is even. In the second step, we shift the result of the first step with one position to the right with throwing the least significant digit, which is 0, away. (Recall that M is odd.)

We can perform the procedures $MADD$, $MDBL$ and $MHLV$ in constant time independent of n by means of combinational circuits. A modular adder consists of two RB adders, one of which is simpler and is also used for modular doubling. A modular halving circuit consists of a simpler RB adder.

4 A Hardware Algorithm for Modular Division

The algorithm is based on EEEA shown in Section 2.

We assume six shift registers, REG-A, REG-B, REG-U, REG-V, REG-P and REG-D. Each register can shift its content with one position to the left or right in one clock cycle. We also assume a register, REG-M.

REG-A $(= [.a_1 a_2 \cdots a_n])$ and REG-B $(= [.b_1 b_2 \cdots b_n])$ are n-digit registers and keep A_{k-1}

and A_k represented in RB representation, respectively. To fix the positions which have to be checked at each computation step, we treat A_k's as fractions. Only a few digits around the radix points are checked. REG-A and REG-B can exchange their contents.

REG-U (= $[u_0 u_1 \cdots u_n]$) and REG-V (= $[v_0 v_1 \cdots v_n]$) are (n+1)-digit registers and keep U_{k-1} and U_k also represented in RB representation, respectively. REG-U and REG-V can exchange their contents.

REG-P (= $[.p_1 \cdots p_n]$) is an n-bit register and indicates the position of the least significant digit of A_k in REG-B. Namely, REG-P keeps only a number in the form 2^{-i}, and when its content is 2^{-i}, the least significant digit is at the i-th binary position. REG-D (= $[d_0.d_1 \cdots d_n]$) is an (n+1)-bit register and indicates the difference of the positions of the least significant digits of A_{k-1} in REG-A and that of A_k in REG-B in unary representation. Namely, REG-D also keeps only a number in the form 2^{-i}, and when its content is 2^{-i}, the difference is i.

REG-M (= $[.m_1 \cdots m_n]$) is an n-bit register and keeps the modulus M.

Hereafter, we use A (B, U, V, P, D, and M) to denote both the representation and the value of the content of REG-A (REG-B, REG-U, REG-V, REG-P, REG-D, and REG-M respectively). Note that $a_i, b_i, u_i, v_i \in \{\bar{1}, 0, 1\}$, while $p_i, d_i, m_i \in \{0, 1\}$.

As in our modular inversion algorithm [3] and Parikh et al.'s GCD algorithm [4], "normalization" of an RB fraction is one of the key points. An RB fraction $A = [.a_1 a_2 \cdots a_n]$ is normalized if $a_1 \neq 0$ and $a_2 \neq -a_1$. Note that when A is normalized, $1/4 < |A| < 1$.

The outline of the algorithm is as follows.

Algorithm [MODDIV]
(Inputs)
Modulus: M (an n-bit binary number, $2^{n-1} < M < 2^n$)
Dividend: X (an n-bit binary number, $0 < X < M$)
Divisor: Y (an n-bit binary number, $0 < Y < M$)
(Output)
Quotient: Z (an n-bit binary number, $0 < Z < M$, $Z \times Y \equiv X \pmod{M}$)
(Algorithm)
Step 1: [Initialization]
$A := M \cdot 2^{-n}$; $B := Y \cdot 2^{-n}$;
$U := 0$; $V := X$;
$P := 2^{-n}$; $D := 1$;
$M := M$;
Step 2:
Step 2-1: [Normalization of B]
Shift B to the left until it is normalized.
Double V modulo M, shift P to the left and shift D to the right with the same number of times as B is shifted.
If P becomes 2^{-1}, i.e., B becomes 1 or -1, during the normalization, shift D to the left and halve V modulo M until D becomes 1, and then go to Step 3.
Step 2-2: [Pseudo-division]
Perform pseudo-division until $|A| < |B|$.
(When calculating $A := A - q \cdot B$ in RB system, calculate $U := U - q \cdot V$ modulo M in RB.)
(When shifting A to the left, shift D to the left and halve V modulo M.)
Step 2-3: [Swapping]
Swap A and B. Swap U and V.
Go to Step 2-1
Step 3: [Correction]
If $B = -1$, negate V.
Step 4: [Conversion]
Convert V into binary number Z.
If Z is negative, add M to Z. □

In Step 1, we initialize the registers. We put M (= A_0) and Y (= A_1) into REG-A and REG-B respectively, so that their least significant bits occupy the n-th binary position. We initialize REG-U to 0 (= U_0). We put X (= U_1) into REG-V. We put 2^{-n} and 1 to REG-P and REG-D, respectively. We put M into REG-M.

In Step 2-1, we normalize B. The procedure for normalizing B is as follows. We use the procedures $MDBL$ and $MHLV$ shown in the previous section. $LSHIFT(D)$ and $RSHIFT(D)$ mean one-position left and right shifting of D, respectively.

[Normalization of B]
```
while p1 = 0 and (b1 = 0 or b1 · b2 = −1) do
begin
    if b1 = 0 then b1 := b2 else b1 := b1:
    bi := bi+1 (for i ≥ 2) (/* left-shift */);
    V := MDBL(V, M);
    LSHIFT(P);
    RSHIFT(D);
end;
if p1 = 1 then do
begin
    while d0 = 0 do
```

begin
$LSHIFT(D)$;
$V := MHLV(V, M)$;
end;
go to Step 3
end;
else go to Step 2-2 □

In the main stage, we perform the evaluation of $[.b_1b_2]$, the one-position shifts and $MDBL$ in one clock cycle. In the termination stage, we perform the one-position shift and $MHLV$ in one clock cycle.

In Step 2-2, we perform a pseudo-division, so that the remainder becomes smaller than the divisor, i.e., $|A| < |B|$. The procedure for pseudo-division is as follows. $abs([.a_1a_2])$ denotes the absolute value of $[.a_1a_2]$. $sgn([.a_1a_2])$ denotes the sign of the value of $[.a_1a_2]$, which is -1 or 0 or 1, accordingly as $[.a_1a_2]$ is negative or zero or positive.

[Pseudo-division]
while $d_0 = 0$ do
begin
if $abs([.b_1b_2]) = 3/4$ and $abs([.a_1a_2]) = 2/4$
then do
begin
$V := MHLV(V, M)$;
$LSHIFT(D)$;
$q := a_1 \cdot b_1$;
$A := 2 \cdot A - q \cdot B$;
$U := MADD(U, -q \cdot V, M)$;
end
else do
begin
$q := a_1 \cdot b_1$;
$A := A - q \cdot B$;
$U := MADD(U, -q \cdot V, M)$;
end;
[Normalization of A]
while $d_0 = 0$ and ($a_1 = 0$ or $a_1 \cdot a_2 = -1$) do
begin
if $a_1 = 0$ then $a_1 := a_2$ else $a_1 := a_1$;
$a_i := a_{i+1}$ (for $i \geq 2$) (/* left-shift */);
$V := MHLV(V, M)$;
$LSHIFT(D)$;
end;
end;
[Termination of pseudo-division] (/* $d_0 = 1$ */)
if $abs([.b_1b_2]) = 3/4$ and $abs([.a_1a_2]) \geq 2/4$
then do
begin
$q := a_1 \cdot b_1$;
$A := A - q \cdot B$;
$U := MADD(U, -q \cdot V, M)$
end;
else if $abs([.b_1b_2]) = 2/4$ and $abs([.a_1a_2]) \geq 1/4$
then do
begin
$r := sgn([.a_1a_2])$;
while $sgn([.a_1a_2]) = r$ do
begin
$q := r \cdot b_1$;
$A := A - q \cdot B$;
$U := MADD(U, -q \cdot V, M)$
end;
if $abs([.a_1a_2]) = 2/4$ then do
begin
$q := a_1 \cdot b_1$;
$A := A - q \cdot B$;
$U := MADD(U, -q \cdot V, M)$
end;
end;
go to Step 2-3; □

In the main and the termination stage, we perform the evaluation of $[.a_1a_2]$ and $[.b_1b_2]$, the calculation of A, $MADD$, the possible $MHLV$ and the possible $LSHIFT$ in one clock cycle. In the normalization of A, we perform the evaluation of $[.a_1a_2]$, the one-position shifts and $MHLV$ in one clock cycle.

We perform the addition (subtraction) for calculating A in RB system using the addition rule shown in Table 1 in the previous section except at the most significant position where we adopt a special computation rule to make the integer part of the sum 0. The addend digit at the i-th position is $-q \cdot b_i$. In the case of calculating $2 \cdot A - q \cdot B$, the augend digit at the i-th position is a_{i+1} and we let $ad_1 = a_1$. In the case of calculating $A - q \cdot B$, the augend digit is a_i and we let ad_1 be 0 when $a_1 \neq 0$ and be $-a_2$ ($= -q \cdot b_1$) otherwise. (The case $a_1 = 0$ occurs only in the termination stage.)

From the computation in the termination stage, we can show the following lemma.

[Lemma 2]
$|A| < |B|$ holds at the end of pseudo-division.
(Proof)
Case 1: $abs([.b_1b_2]) = 3/4$
When $abs([.a_1a_2]) \leq 1/4$ at the beginning of the termination stage, since $|A| < 2/4$, $|A| < |B|$ holds. Note that $|B| > 2/4$.
When $abs([.a_1a_2]) \geq 2/4$, we calculate $A := A - q \cdot B$ where $q := a_1 \cdot b_1$. When $sgn([.a_1a_2])$

has not changed by the calculation, $|A| < 2/4$, and therefore, $|A| < |B|$ holds for the updated A. When $sgn([.a_1a_2])$ has become 0, $|A| < 1/4$, and therefore, $|A| < |B|$ holds. When $sgn([.a_1a_2])$ has been negated, $|A| < |B|$ holds.

Case 2: $abs([.b_1b_2]) = 2/4$

When $abs([.a_1a_2]) = 0$ at the beginning of the termination stage, since $|A| < 1/4$, $|A| < |B|$ holds. Note that $|B| > 1/4$.

When $abs([.a_1a_2]) \geq 1/4$, we perform $A := A - q \cdot B$, where q is the initial $sgn([.a_1a_2])$ times b_1, iteratively till $sgn([.a_1a_2])$ changes. When $sgn([.a_1a_2])$ has become 0, $|A| < 1/4$, and therefore, $|A| < |B|$. When $sgn([.a_1a_2])$ has been negated, $abs([.a_1a_2]) \leq 2/4$ from the addition rule. When $abs([.a_1a_2]) = 1/4$, $|A| < |B|$ holds. When $abs([.a_1a_2]) = 2/4$, we add back B to A.

In any case, $|A| < |B|$ holds at the end of pseudo-division. □

In Step 2-3, we swap A and B, and U and V, in one clock cycle.

In Step 3, when $b_1 = \bar{1}$, we negate V in RB system in one clock cycle.

In Step 4, we first convert V into the ordinary binary representation. We need a carry-propagate addition for this conversion. Then, we add M to the converted V in the ordinary binary number system with carry propagation, if it is negative. We can perform a carry-propagate addition bit-serially using a one-bit full adder.

Through [MODDIV], all computations carried out in one clock cycle can be performed within constant time independent of n by means of combinational circuits. Therefore, the length of the clock cycle is constant independent of n.

Fig. 2 shows the first several steps of the computation of 79/108 (mod 211) by [MODDIV]. Only the registers with change are shown.

A modular divider based on the proposed algorithm mainly consists of seven registers, a redundant binary (RB) adder, a modular adder and a modular halving circuit. Recall that the simpler RB adder in the modular adder is also used for modular doubling. The divider has a regular cellular array structure with a bit slice feature and is very suitable for VLSI implementation. Each bit slice mainly consists of 11 flip-flops and four redundant binary addition cells, two of which are simpler. Note that we need two flip-flops to store an RB digit. For the details of the redundant binary addition cells, see, e.g., [6]. The amount of hardware of the divider is proportional to n. It seems easy to fabricate a modular divider VLSI using today's technology.

5 Analysis of the Algorithm

In this section, we show that the number of clock cycles required by [MODDIV] to perform a modular division is proportional to n, the word length of the operands.

In Step 1, if we feed operands to the registers bit-serially, we need n clock cycles.

In Step 2, initially, P is 2^{-n}. Step 2 terminates when P becomes 2^{-1}. P is shifted to the left with several positions in a normalization of B (Step 2-1). D is shifted to the right with the same number of positions as P is shifted. In the succeeding pseudo-division (Step 2-2), D is shifted to the left with the same number of positions as it was shifted in the normalization of B. P is unchanged during this step. We investigate how many clock cycles are required in one pass of Step 2.

When P and D are shifted j positions in one pass of Step 2-1, j clock cycles are executed there except in the final pass. (In the final pass, $2j$ clock cycles are executed there, and then the process exits Step 2.) During the succeeding Step 2-2, D is shifted back j positions. Except in the termination stage, we can show that no more than two cycles are executed succeedingly without a left-shift of D [3]. In many cases, D is shifted with more than one positions during normalization of A. We can also show that no more than three cycles are executed in the termination stage. The number of clock cycles executed in the pass of Step 2-2 is at most about $2j$. In the succeeding swapping step (Step 2-3), one clock cycle is executed. Therefore, the number of clock cycles executed in one pass of Step 2 is at most about $3j$.

We have to consider the case that no shift occurs in Step 2-1. In such case, Step 2-2 immediately enters the termination stage. We can show that no more than two pseudo-divisions without normalization of B are performed succeedingly [3].

Hence, in total, the number of clock cycles required in Step 2 is at most about $3n$. It varies with operands.

For Step 3, we require only one clock cycle.

In Step 4, we need one or two carry-propagate additions. For a carry-propagate addition, we need n clock cycles if we perform it bit-serially using a

one-bit full adder. Of course, we can accelerate the processing by the use of a longer adder.

6 Concluding Remarks

We have proposed a hardware algorithm for modular division in a residue class field. We have further extended the extended Euclidean algorithm and have accelerated it by the use of the redundant binary representation for internal computation.

A modular divider based on the algorithm has a regular cellular array structure with a bit slice feature and is very suitable for VLSI implementation. The amount of hardware of an n-bit modular divider is proportional to n. It carries out a modular division in $O(n)$ clock cycles, where the length of each clock cycle is constant independent of n.

References

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

[2] D. E. Knuth: 'The Art of Computer Programming,' vol. 2, 'Seminumerical Algorithms,' Addison-Wesley, 1969.

[3] N. Takagi: 'A modular inversion hardware algorithm with a redundant binary representation,' IEICE Trans. on Information and Systems, vol. E76-D, no. 8, pp. 863–869, Aug. 1993.

[4] S. N. Parikh and D. W. Matula: 'A redundant binary Euclidean GCD algorithm,' Proc. 10th Symp. Computer Arithmetic, pp. 220-224, June 1991.

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

[6] N. Takagi and S. Yajima: 'Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem,' IEEE Trans. Comput., vol. 41, no. 7, pp. 887–891, July 1992.

Initialization	A	. 1 1 0 1 0 0 1 1	(A_0)
	B	. 0 1 1 0 1 1 0 0	(A_1)
	U	0 0 0 0 0 0 0 0 0	(U_0)
	V	0 0 1 0 0 1 1 1 1	(U_1)
	P	. 0 0 0 0 0 0 0 0 1	
	D	1. 0 0 0 0 0 0 0 0	
	M	1 1 0 1 0 0 1 1	(M)
Norm. B	B	. 1 1 0 1 1 0 0 0	
	V	0 1 1 0 0 1 1 1 1	
	P	. 0 0 0 0 0 0 1 0	
	D	0. 1 0 0 0 0 0 0 0	
$A - B$	A	. 0 0 0 0 1 1 0 1	
	U	0 1 0 1 0 0 1 1 0	
Norm. A	A	. 0 0 0 1 1 0 1 0	
	V	0 0 1 0 1 0 1 1 1	
	D	1. 0 0 0 0 0 0 0 0	
Swap	A	. 1 1 0 1 1 0 0 0	(A_1)
	B	. 0 0 0 1 1 0 1 0	(A_2)
	U	0 0 1 0 1 0 1 1 1	(U_1)
	V	0 1 0 1 0 0 1 1 0	(U_2)
Norm. B	B	. 0 0 1 1 0 1 0 0	
	V	0 1 0 1 0 1 0 0 1	
	P	. 0 0 0 0 0 1 0 0	
	D	0. 1 0 0 0 0 0 0 0	
Norm. B	B	. 0 1 1 0 1 0 0 0	
	V	0 0 0 0 0 0 0 1 1	
	P	. 0 0 0 0 1 0 0 0	
	D	0. 0 1 0 0 0 0 0 0	
Norm. B	B	. 1 1 0 1 0 0 0 0	
	V	0 0 0 0 0 0 1 1 0	
	P	. 0 0 0 1 0 0 0 0	
	D	0. 0 0 1 0 0 0 0 0	
Norm. B	B	. 1 0 1 0 0 0 0 0	
	V	0 0 0 0 0 1 1 0 0	
	P	. 0 0 1 0 0 0 0 0	
	D	0. 0 0 0 1 0 0 0 0	
$A + B$	A	. 0 1 0 0 1 0 0 0	
	U	0 1 0 0 0 0 0 0 0	
Norm. A	A	. 1 0 0 1 0 0 0 0	
	V	0 0 0 0 0 0 1 1 0	
	D	0. 0 0 1 0 0 0 0 0	
$A + B$	A	. 0 1 1 1 0 0 0 0	
	U	0 1 1 1 1 1 1 1 1	
Norm. A	A	. 1 1 1 0 0 0 0 0	
	V	0 0 0 0 0 0 0 1 1	
	D	0. 0 1 0 0 0 0 0 0	
Norm. A	A	. 1 1 0 0 0 0 0 0	
	V	0 1 0 1 1 1 0 1 0	
	D	0. 1 0 0 0 0 0 0 0	
$A - B$	A	. 0 0 1 0 0 0 0 0	
	U	0 0 0 1 1 1 1 0 1	
Norm. A	A	. 0 1 0 0 0 0 0 0	
	V	0 0 1 0 1 1 1 0 1	
	D	1. 0 0 0 0 0 0 0 0	
$A - B$	A	. 1 0 1 0 0 0 0 0	
	U	0 1 0 0 1 1 0 1 1	
$A + B$	A	. 0 1 0 0 0 0 0 0	
	U	0 0 0 1 1 1 1 1 1	
Swap	A	. 1 0 1 0 0 0 0 0	(A_2)
	B	. 0 1 0 0 0 0 0 0	(A_3)
	U	0 0 1 0 1 1 1 0 1	(U_2)
	V	0 0 0 1 1 1 1 1 1	(U_3)

Figure 2: A computation by [MODDIV]