A Fast Algorithm for Modular Reduction [*]

Ç. K. Koç [†]
Electrical and Computer Engineering
Oregon State University
Corvallis, Oregon 97331, USA

C. Y. Hung
DSP R&D Center
Texas Instruments Inc.
Dallas, Texas 75265, USA

Abstract

We present an algorithm for computing the residue $R = X \bmod M$. The algorithm is based on a *sign estimation* technique that estimates the sign of a number represented by a carry-sum pair produced by a carry save adder. Given the $(n + k)$-bit X and the n-bit M, the modular reduction algorithm computes the n-bit residue R in $O(k + \log n)$ time, and is particularly useful when the operand size is large. We also present a variant of the algorithm that performs modular multiplication by interleaving the shift-and-add and the modular reduction steps. The modular multiplication algorithm can be used to obtain efficient VLSI implementations of exponentiation cryptosystems.

Key Words: Carry save adder, sign estimation technique, division, modular multiplication.

1 Introduction

Arithmetic operations with long operands are often carried out using redundant representations in order to avoid a full-length carry propagation delay. In the modular reduction operation $X \bmod M$, we need to compare X with some integer multiple q of M, or equivalently, calculate the sign of $(X - qM)$. However, in a redundant representation, the sign of a number may not be readily available. In particular, when the carry save addition technique is used, the precise determination of the sign of an n-bit number represented by a carry-sum pair requires the computation of the total sum, which takes $O(n)$ time with a carry propagate adder.

We present an improved version of the *sign estimation technique* [13] for detecting the sign of a number represented in the form of a carry-sum pair. The sign estimation technique was used to obtain fast algorithms for multi-operand modular addition [13] and modular multiplication [12, 14] operations. The improved sign estimation algorithm presented here correctly computes the sign of the number when the number is large enough in magnitude. The sign cannot be determined only when the number is small. We then develop a modular reduction algorithm based on the sign estimation technique, which is useful for division and modular multiplication operations.

Most of the previous work on division and modular multiplication can be classified into three categories based on the representation of the numbers:

- nonredundant binary number representation combined with table lookup [2],

[*] *IEE Proceedings: Computers and Digital Techniques*, 145(4):265-271, July 1998.
[†] Supported in part by Intel Corporation.

- nonredundant binary number representation, with the approximate quotient obtained by multiplying the dividend by the divisor reciprocal [1, 3].
- signed digit number representation [16, 18, 21], and
- carry save representation and its variations [23, 5, 12, 13, 14].

In the worst case, addition of nonredundant numbers requires carry propagation over the full length of the operands, and thus, this approach may not be efficient for large operand size. Multiplying by the divisor reciprocal is very effective when a fast multiplier is available, for example in the context of single precision division in residue number systems. Both signed digit number and carry save representations eliminate the problem of carry propagation. However, signed digit addition usually requires more logic per bit than carry save addition. We advocate carry save representation which leads to more efficient VLSI implementation: Our modular reduction and modular multiplication algorithms exclusively use carry save adders.

The division algorithm proposed by Zurawski and Gosling uses the borrow save representation, which is a variation of the carry save representation [23]. However, this algorithm performs restoring division by having two data paths computing in parallel, and thus, the amount of logic required is roughly doubled. The modular multiplier proposed by Brickell is based on half adders [5]. Brickell's method computes modular multiplication of two redundantly represented numbers, and thus, conversions between redundant and nonredundant representations can be skipped. However, the complexity of the logical circuit required for each bit is quite high. We also point out that modular reduction algorithms in the context of residue number systems have been designed, assuming small X (about 32 bits) and much smaller M; for example, see [1].

A completely different approach, proposed by Montgomery [17], is to perform modular reduction *right-to-left*, i.e., scanning the dividend X from the least significant digit on to the more significant digits. An array processor for modular multiplication based on this concept has strictly right-to-left communications, and thus leads to very efficient systolic implementation [22]. However, the Montgomery algorithm produces an extra factor on the result. The correction involves some precomputation based on the divisor and an extra modular multiplication, and offsets the benefits the algorithm provides.

The modular reduction algorithm proposed in this paper is suitable for very large X and M, in the order of hundreds of bits. Applications are found in cryptosystems based on modular exponentiation, for example, the RSA algorithm [19], the ElGamal signature scheme [9], and the recently proposed digital signature standard (DSS) of the National Institute for Standards and Technology [10]. These cryptosystems use modular exponentiation, which in turn requires modular multiplication as the core operation. The challenge comes from the fact that, due to security requirements, the number of bits of the operands are in the order of several hundreds, sometimes up to a thousand. Fast implementations of these cryptosystems are crucial since such systems will be used in high-speed computer/communication networks (from 64 Kbits/s up to 140 Mbits/s) [4]. Current software [7] and conventional hardware implementations [6] of the RSA algorithm provide a speed of about 20 Kbits/s. Specialized hardware architectures, such as those using programmable gate arrays, which encrypt/decrypt at a speed of 200–600 Kbits/s have been designed [20]. However, there is still a need for algorithmically and technologically efficient, cost-effective implementations. The modular reduction algorithm presented here provides an efficient solution, requiring very little hardware resources.

2 The Sign Estimation Technique

Redundant representations of numbers are frequently used for performing arithmetic operations with large operands. Of the many existing redundant representations, the carry save technique is rather inexpensive to implement, and thus, widely used. The carry save representation, produced by carry save addition, is in the form of two binary numbers usually called the carry and the sum. The value of the number is equal to the sum of these two numbers.

The sign of a number in one's or two's complement representations is indicated by the most significant bit. However, in the carry save representation, the most significant bit (the sign bit) is not readily available. Nor is it trivial to compute: In the worst case, we have to compute the total sum of these two binary numbers in order to obtain the sign, which takes takes $O(\log n)$ time with a carry lookahead adder, or $O(n)$ time with a carry propagate adder. The time spent for calculating the exact sign bit may dominate the overall time complexity for computations that require the sign bit in the innermost loop, e.g., division and modular reduction operations. We attack this problem by substituting the *exact sign* with an *estimated sign*. The estimated sign which is less expensive to compute is used in the bulk of the computation. The exact sign is used only as the last resort. Given two m-bit binary numbers S and C representing a signed number, the estimated sign function $ES(S, C)$ returns with one of these three answers:

1. The sign is positive $(+)$,
2. The sign is negative $(-)$,
3. The sign is unsure $(\pm)$.

When positive or negative signs are returned, the signed number is guaranteed to be positive (≥ 0) or negative (< 0), respectively. When the sign is determined to be unsure, then the magnitude of the signed number is guaranteed to be either too large (close to 2^{m-1}) or too small (close to zero). The sign estimation technique is more useful when we have *a priori* knowledge that the signed number is never too large; an unsure sign would then always indicate a small number.

In the following presentation, we use the notation:

- Range notation of the form $[x, y]$, $[x, y)$, $(x, y]$, and (x, y) is used. Brackets denote closed boundaries while parentheses denote open boundaries. For example,

$$z \in [x, y) \text{ is equivalent to } \quad x \leq z < y \ .$$

- We use $|x|_y$ to represent the residue of x modulo y. By definition, x and y are real, and $y > 0$. If $r = |x|_y$, we must have $r \in [0, y)$, and there exists an integer k such that $r = x - ky$.

- An m-bit binary number X is represented by the binary string

$$X_{m-1}X_{m-2} \cdots X_1X_0 \ ,$$

where X_{m-1} is the most significant and sign bit, and X_0 is the least significant bit. We use the same symbol X to represent the (signed) value, and use X_U to represent the *unsigned* value of the binary string. The usual two's complement format is used. Therefore we have

$$X = -2^{m-1}X_{m-1} + \sum_{i=0}^{m-2} 2^i X_i \ , \quad X_U = \sum_{i=0}^{m-1} 2^i X_i \ .$$

Note that $X = -2^m X_{m-1} + X_U$ and $X_U \in [0, 2^m)$, therefore we have

$$X_U = |X|_{2^m} \ .$$

- The truncation function $T(X)$ is defined as the operation which replaces the least significant t bits of X with zeros, i.e., if X is an m-bit number,

$$T(X) = X_{m-1}X_{m-2}\cdots X_t \underbrace{0\cdots0}_{t\text{ zeros}} \ .$$

  The following two inequalities can easily be proven:

$$\begin{aligned} T(X) \leq &\quad X &< T(X) + 2^t \ , \qquad (1)\\ T(S) + T(C) \leq &\quad S + C &< T(S) + T(C) + 2^{t+1} \ . \qquad (2)\end{aligned}$$

- Two's complement addition of two m-bit numbers is specified in the form

$$Z := X + Y, \qquad (3)$$

  even though arithmetically it is not always true. The actual arithmetic operation is an unsigned addition and ignores the carry out of the sign bit. Therefore, we have

$$Z_U = |X_U + Y_U|_{2^m} = |X + Y|_{2^m}. \qquad (4)$$

  Equation(3) does not hold when there is an overflow, i.e., two positive numbers sum up to be negative or vice versa. Unfortunately, in the carry save representation, summing the carry and sum vectors to obtain the value may cause intentional overflow that should be *ignored*.

- Carry-save addition is denoted as $(S, C) := X + Y + Z$, where X, Y and Z are operands, and S and C are the sum and carry. Arithmetically, the two's complement sum of S and C is equivalent to the two's complement sum of X, Y, and Z.

Given the sum S and carry C vectors representing a number X, the sign estimation function takes bits t through $m-1$ of S and C as input, and returns a sign for X as $(+)$, $(-)$, or $(\pm)$. In the last case, X is guaranteed to be in the range

$$-2^t \leq X < 2^t \ .$$

The parameter t controls the cost and quality of sign estimation. The higher value of t indicates that the sign estimation algorithm uses fewer bits of C and S, and the probability of computing the sign as $(\pm)$ is higher. The algorithm requires that the magnitude of the signed number X is not too large:

$$|X| \leq 2^{m-1} - 2^{t+1} \ . \qquad (5)$$

Let $T(S)$ and $T(C)$ denote the truncated S and C, with $t \leq m-2$. The sign estimation algorithm first computes Y as the two's complement sum of $T(S) + T(C)$. Then, the estimated sign is determined by comparing Y against some constants:

$$ES(S, C) = \begin{cases} (+) & \text{If} \quad Y \geq 0 \ , \\ (-) & \text{If} \quad Y < -2^t, \\ (\pm) & \text{Otherwise.} \end{cases}$$

Theorem 1 *If the sign estimation algorithm computes $ES(S,C) = (+)$ or $(-)$, we must have $X \geq 0$ or $X < 0$, respectively. If $ES(S,C) = (\pm)$, then $-2^t \leq X < 2^t$.*

Proof Let y be an underestimate of x: $y \leq x < y + \delta$ and the error $\delta < z$. We have the following relations:

$$0 \leq |y|_z \leq z - \delta \quad \text{implies} \quad |y|_z \leq |x|_z < |y|_z + \delta \ . \tag{6}$$

$$z - \delta \leq |y|_z < z \quad \text{implies} \quad |y|_z \leq |x|_z < z \ \text{ or } \ 0 \leq |x|_z < |y|_z + \delta - z \ . \tag{7}$$

Figure 1 shows two examples, one for each case. The closed "[" and open ")" boundary notations are carried over to the figure.

Figure 1 goes here

Since X is the two's complement sum of S and C, we have

$$X_U = |S + C|_{2^m}.$$

similarly for the intermediate value Y, we have

$$Y_U = |T(S) + T(C)|_{2^m}.$$

Note that $T(S) + T(C)$ is an underestimate of $S + C$, with the two expressions related by (2). The bound (5) on the input X implies

$$0 \leq \quad X_U \quad \leq 2^{m-1} - 2^{t+1} \ \text{ or} \tag{8}$$

$$2^{m-1} + 2^{t+1} \leq \quad X_U \quad < 2^m \ . \tag{9}$$

We now prove the theorem for each case.

- When $Y \geq 0$, we know $0 \leq Y_U < 2^{m-1} \leq 2^m - 2^{t+1}$. Applying (6), we have

$$Y_U \leq X_U < Y_U + 2^{t+1} \ . \tag{10}$$

  It follows that

$$0 \leq X_U < 2^{m-1} + 2^{t+1} \ .$$

  The inequalities (8) and (9) further restrict the above to

$$0 \leq X_U \leq 2^{m-1} - 2^{t+1} \ ,$$

  thus X is positive.

- The sign estimation algorithm returns $(-)$ when $Y < -2^t$, which implies $2^{m-1} \leq Y_U < 2^m - 2^t$. Since both $T(S)$ and $T(C)$ are multiples of 2^t, we have

$$2^{m-1} \leq Y_U \leq 2^m - 2^{t+1} \ . \tag{11}$$

  Again (6) applies and (10) follows. Combining (10) and (11), we have

$$2^{m-1} \leq X_U < 2^m,$$

  thus X is negative.

- Finally, when $-2^t \leq Y < 0$, the sign estimation returns $(\pm)$. Since $T(S)$ and $T(C)$ are multiples of 2^t, Y must be exactly -2^t, which implies $Y_U = 2^m - 2^t$. The relation (7) leads to

$$\begin{array}{rll} 2^m - 2^t \leq & X_U & < 2^m \quad \text{or} \\ 0 \leq & X_U & < 2^t \ . \end{array}$$

Thus, we have $-2^t \leq X < 2^t$ as claimed.

□

Figure 2 summarizes the proof by showing the ranges of $X = S + C$ and $Y = T(S) + T(C)$ for the 3 cases. In the figure, $\epsilon = 2^t$, and the ranges are labeled as X^* and Y^* where $*$ is $+, -$ and $\pm$, corresponding to the computed sign.

Figure 2 goes here

Although the sign estimation seems to require the two's complement sum $Y = T(S) + T(C)$ in order to compare it to some constants, the full summation need not be carried out. We only need to know the sign bit of the full sum, and also to be able to detect if the sum is equal to -2^t. The carry lookahead technique can be used to quickly generate the sign bit and detect -2^t. We can formulate the carry look-ahead computation as follows: Let p_i and g_i be the carry propagate and carry generate functions, defined as $p_i = S_i \oplus C_i$ and $g_i = S_i C_i$. We compute the carry into the $(m-1)$st bit using

$$C_{\text{in}} = g_{m-2} + g_{m-3}p_{m-2} + g_{m-4}p_{m-3}p_{m-2} + \cdots + g_t p_{t+1} p_{t+2} \cdots p_{m-2} \ .$$

The sign bit is computed as

$$\text{Sign} = S_{m-1} \oplus C_{m-1} \oplus C_{\text{in}} = p_{m-1} \oplus C_{\text{in}} \ .$$

We also define the boolean variable SmallNumber which is equal to 1 if the sum is equal to -2^t, and 0 otherwise. We examine the bit distribution of -2^t as

$$\begin{array}{rcccccccccc} & m-1 & m-2 & \cdots & t+1 & t & t-1 & \cdots & 1 & 0 \\ 2^t = & 0 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 \\ -2^t = & 1 & 1 & \cdots & 1 & 1 & 0 & \cdots & 0 & 0 \end{array}$$

and conclude that SmallNumber is equal to 1 if all p_i from $i = t$ to $m-1$ are equal to 1. This gives the simple formulae:

$$\text{SmallNumber} = \prod_{i=t}^{m-1} p_i \ .$$

The boolean variables Sign and SmallNumber relate to the estimated sign in the following way:

$$ES(S, C) = \begin{cases} (+) & \text{Sign} = 0 \ . \\ (-) & \text{Sign} = 1 \ \text{ and } \ \text{SmallNumber} = 0 \ . \\ (\pm) & \text{SmallNumber} = 1 \ . \end{cases}$$

3 The Modular Reduction Algorithm

Given the $(n+k)$-bit and n-bit positive integers X and M, respectively, the following algorithm computes the remainder $R = X \bmod M$.

```
1.  $S := 2^{-k-1}X$, $C := 0$, and $t = n-2$
2.  for $i = 1, 2, 3, \ldots, k+1$ do begin
3.      if $ES(S,C) = (+)$ then $(S,C) := 2S + 2C - M$
4.      elseif $ES(S,C) = (-)$ then $(S,C) := 2S + 2C + M$
5.      else $(S,C) := 2S + 2C$
6.  end
7.  $(\hat{S}, \hat{C}) := S + C + M$
8.  $R := S + C$ and $\hat{R} := \hat{S} + \hat{C}$
9.  if $R \geq 0$ return $R$ else return $\hat{R}$.
```

The modular reduction algorithm consists of $k+1$ reduction steps (lines 2–6) and the final summation and selection (lines 7–9). The reduction steps use carry save addition, while the algorithm by which line 8 is executed is not specified. It can be carried out using a carry propagate adder, or a faster method, e.g., a carry look-ahead adder. Overall structure of the algorithm resembles the SRT division (see, e.g., [11, pages 226–229]). The input number X is scaled down $k+1$ bits initially, and is shifted up one bit and modularly reduced in each of the $k+1$ reduction steps. In each reduction step a sign estimation (with parameter $t = n-2$), and a carry-save addition are performed.

We shall prove the correctness of the algorithm. First, we claim that

$$-M \leq S + C < M \tag{12}$$

after each iteration is executed. Before the first iteration, immediately after X is scaled down and loaded in S, we have the relation

$$0 \leq S + C < 2^{-k-1}2^{n+k} = 2^{n-1} \leq M \ .$$

Here we assume that M requires exactly n bits to represent, i.e.,

$$2^{n-1} \leq M < 2^n \ .$$

Thus, the inequality (12) is true. Next, suppose before some iteration i, relation (12) is true; we want to show that it still holds after lines 3, 4 and 5 are executed for that iteration. The sign estimation algorithm produces one of three outcomes:

- $ES(S,C) = (+)$: In this case $S + C \geq 0$, and thus $S + C$ is in the range $[0, M)$. After the left-shifts and the subtraction by M, we have $S + C \in [-M, M)$.
- $ES(S,C) = (-)$: In this case $S + C < 0$, and thus $S + C$ is in the range $[-M, 0)$. After the left-shifts and the addition with M, we have $S + C \in [-M, M)$.
- $ES(S,C) = (\pm)$. In this case, the sign estimation procedure guarantees that $S + C \in [-2^t, 2^t) \equiv [-2^{n-2}, 2^{n-2}) \subseteq [-M/2, M/2)$. The left-shifts bring $S + C$ to $[-M, M)$.

Therefore we conclude that relation (12) is true after every iteration. As the algorithm exits the **for** loop, we have $-M \leq S + C < M$. In line 7, we produce another carry-sum pair $(\hat{S}, \hat{C})$ by adding M to (S, C). Both of the pairs are kept as candidates for the correct residue. We compute the sums R and $\hat{R}$, and choose one of them. Since $R \in [-M, M)$ and $\hat{R} = R + M$, we have $\hat{R} \in [0, 2M)$. If $R \geq 0$, then R is in the range $[0, M)$, and thus, is the correct residue. Otherwise, $R \in [-M, 0)$ and $\hat{R} \in [0, M)$, i.e., $\hat{R}$ is the correct residue.

Since scaling up $k+1$ bits on line 1 cancels out with the subsequent $k+1$ shifts in the **for** loop, the net effect is adding a multiple of M to X to produce a residue in $[0, M)$. Therefore, the algorithm correctly computes $X \bmod M$.

Next we would like to assess the cost of the algorithm, particularly in carry-save additions and sign estimations. During the entire execution of the algorithm, the number the carry-sum pair represents is in the range $[-M, M) \subseteq (-2^n, 2^n)$. Thus, seemingly we need just $n+1$ bits to the left of the radix point to represent the numbers S and C, and the same length for the adders since fractional part is never added. However, this may not be sufficient. The sign estimation algorithm requires that $S + C$ not use the full range of representation, as shown in (5). If we pick $m = n+1$, the requirement posed by (5) that

$$|S + C| \leq 2^{m-1} - 2^{t+1} = 2^n - 2^{n-1} = 2^{n-1}$$

is not guaranteed to be met. Next, we try $m = n + 2$, and the requirement that

$$|S + C| \leq 2^{m-1} - 2^{t+1} = 2^{n+1} - 2^{n-1}$$

is satisfied. Therefore, we need $n + 2$ bits for the adders. The $(k+1)$-bit fraction of S can be stored in a shift register. The sign estimation procedure looks at bits $n+1, n, n-1$, and $n-2$ of S and C. It can be implemented by a 4-bit adder and a few extra gates to encode the three possible outcomes.

4 The Modular Multiplication Algorithm

The modular reduction algorithm presented in the previous section leads to an efficient modular multiplication algorithm. The sign estimation procedure is tuned to yield more precise determination of the sign. With proper scaling of operands, the shift-and-add steps of multiplication are interleaved with modular reduction steps. The following algorithm computes $P = AB \bmod M$, where M is an n-bit number, $2^{n-1} \leq M < 2^n$, and $0 \leq A, B < M$.

```
1.  $M := 8M_{\text{input}}$, $S := 0$, $C := 0$, and $t = n - 1$
2.  **for** $i = n-1, n-2, \ldots, 0, -1, -2, -3$ **do begin**
3.      **if** $ES'(S, C) = (+)$ **then** $(S, C) := 2S + 2C + A_i B - M$
4.      **elseif** $ES'(S, C) = (-)$ **then** $(S, C) := 2S + 2C + A_i B + M$
5.      **else** $(S, C) := 2S + 2C + A_i B$
6.  **end**
7.  $(\hat{S}, \hat{C}) := S + C + M$
8.  $P := S + C$ and $\hat{P} := \hat{S} + \hat{C}$
9.  **if** $P < 0$ then $P := \hat{P}$
10. **return** $P/8$.
```

Line 1 scales up M by a factor of 8. That makes the added terms A_iB in each iteration small relative to M, so the partial product $S + C$ stays in a range. Lines 2 to 6 contain the **for** loop that performs shift-and-add and modular reduction. Note that a variant of the sign estimation procedure, $ES'()$ is used. The iteration count i goes down to -3. We define A_{-1}, A_{-2} and A_{-3} to be zeros, as A is an integer. Thus, each of the last three iterations is the same as an iteration in the modular reduction algorithm; just shifting and adding/subtracting M. The three extra modular reduction steps compensate for the use of scaled up modulus. A by-product of these extra steps is a factor of 8 on the carry-sum pair, which is corrected on line 10. Lines 7 to 10 are the final summation and selection steps similar to lines 7 to 9 of the modular reduction algorithm. The only difference is that the result is scaled back by 8 before being returned.

The new sign estimation procedure is tuned to reduce the shift-and-add form of accumulation as opposed to just shifting in our modular reduction algorithm. The procedure computes value Y as the sum of truncated S and C with the precision parameter t. For convenience, we define $\epsilon = 2^t$. The numbers Y is compared against to determine the sign are different:

$$ES'(S, C) = \begin{cases} (+) & \text{If} \quad Y \geq 2\epsilon \ , \\ (-) & \text{If} \quad Y \leq -4\epsilon \ , \\ (\pm) & \text{Otherwise.} \end{cases}$$

A counterpart to Theorem 1 for the new procedure is as follows.

Theorem 2 *If $ES'(S, C)$ computes $ES'(S, C) = (+)$, we must have $X \geq 2\epsilon$; if it computes $ES'(S, C) = (-)$, we have $X < -2\epsilon$; and if $ES'(S, C) = (\pm)$, then $-3\epsilon \leq X < 3\epsilon$, where $\epsilon = 2^t$, X is the m-bit integer (S, C) pair represents, and $|X| < 2^{m-1} - 2\epsilon$.*

The proof is very similar to the proof for Theorem 1, and is therefore omitted. Figure 3 summarizes the ranges of X and Y. The new procedure also requires the represented m-bit number X to satisfy condition (5).

Figure 3 goes here

To prove the correctness of the modular multiplication algorithm, we first show that throughout the **for** loop, we have

$$S + C \in [-\frac{3M}{4}, \frac{7M}{8}). \tag{13}$$

On line 1, S and C are initialized to zero. Therefore, condition (13) is true before entering the first iteration. Next, assume that condition (13) is true before the execution of some iteration. We want to show that (13) is still true after at the completion of that iteration. The scaled up M is related to $\epsilon = 2^{n-1}$ by

$$8\epsilon \leq M < 16\epsilon.$$

The 3 cases of sign estimation outcome are:

- $ES'(S, C) = (+)$: In this case $S+C \geq 2\epsilon > M/8$, and thus $S+C$ is in the range $(M/8, 7M/8)$. After the shifting, addition of A_iB, and subtraction by M, we have $S'+C' \in (-3M/4, 7M/8)$.
- $ES'(S, C) = (-)$: In this case $S + C \leq -2\epsilon < -M/8$, and thus $S + C$ is in the range $[-7M/8, -M/8)$. After the shifting, and addition of A_iB and M, we have $S' + C' \in [-3M/4, 7M/8)$.

- $ES'(S,C) = (\pm)$. In this case, the sign estimation procedure guarantees that $S + C \in [-3\epsilon, 3\epsilon) \subseteq [-3M/8, 3M/8)$. The shifting and addition of A_iB lead to $S' + C' = S + C \in [-3M/4, 7M/8)$.

So far, we have shown that condition (13) holds throughout the $n+3$ iterations. At the end of the **for** loop, we have $S + C \in [-3M/4, 7M/8) \subset [-M, M)$, and so the final summation and selection of lines 7 to 9 finds a result in $[0, M) \equiv [0, 8M_{\text{input}})$. The scaling down of line 10 finds a value (not yet proven to be an integer) in $[0, M_{\text{input}})$.

By expanding the computation inside the $n+3$ iterations, we can see that, after the **for** loop, we have

$$\begin{aligned} S + C &= ((A_{n-1}B + q_{n-1}M) \cdot 2 + A_{n-2}B + q_{n-2}M) \cdot 2 + \ldots + A_{-3}B + q_{-3}M \\ &= 8AB + QM, \end{aligned}$$

where q_i is 0, 1, or -1, and $Q = \sum 2^{i+3} q_i$. The scaling down of line 10 corrects the factor of 8, and does not produce a fraction regardless of whether P or $\hat{P}$ is chosen in line 9, because both $8AB + QM$ and $8AB + (Q+1)M$ are divisible by 8. So far, we know that the returned P is an integer, is AB plus some multiple of M_{input}, and is in $[0, M_{\text{input}})$. Therefore, it must be the correct result for $AB \bmod M_{\text{input}}$. We conclude that the algorithm is correct.

The hardware cost of the algorithm is analyzed as follows. We claim that $(n+4)$-bit carry-save adders are sufficient. The largest magnitude the carry-sum pair represents is $7M/8 < 2^{n+3}$. The additional requirement imposed by the sign estimation is also met: Taking $m = n+4$, we observe that $2^{m-1} - 2^{t+1} = 2^{n+3} - 2^n = (7/8)2^{n+3} > 7M/8$. The sign estimation procedure therefore looks at bits $n-1$ through $n+3$ of S and C. The procedure can be implemented with a 5-bit adder and a few extra gates.

5 Implementation and Application

The modular reduction and multiplication algorithms presented here can be efficiently implemented with one-dimensional and two-dimensional array processors, semi-systolic or systolic. Since the literature already contains systematic procedures to map dependency DAG (directed acyclic graph) onto array processor (see, e.g., Kung [15]), we shall show how the arithmetic operations in both algorithms can be organized into regular DAGs to facilitate hardware implementation. We shall concentrate on the bulk of the computation, i.e., the **for** loop of each algorithm.

The modular reduction algorithm of Section 2 has shift-and-add in each iteration of the loop. The item to be added, qM, $q = \{1, -1, 0\}$, depends on the outcome of sign estimation, which in turn depends on the leading 4 bits of S and of C, which are produced by the 5 leftmost bits of carry-save adders. Figure 4 shows a section of the DAG. The "L" cell represents sign estimation procedure, and the X cell represents the carry-save addition and the logic to provide the qM term.

Figure 4 goes here

To make a simpler and more regular DAG, we take the same approach as in [14]: in each row we group the L cell with the 3 leftmost X cells into a supercell, and pair up the rest of the X cells. The resulting DAG is shown in Figure 5.

Figure 5 goes here

The DAG for the modular multiplication algorithm is slightly more complicated. The 4-term summation in lines 3 and 4 needs two levels of carry-save adders to reduce to a carry-sum pair, and there is a carry transfer between these two levels. As depicted in Figure 6(a), the carry transfer is signified by the left shift of carry vector between the 2 CSAs. These carries prevent merging of the two levels of adders, so the corresponding DAG has $2(n+3)$ rows.

To facilitate more efficient implementations, we use (S, C_1, C_2), i.e., sum and double carry vectors, to represent the partial product. We now have 5-term summation, and still need 2 levels of carry-save additions, but there is no carry transfer between the 2 levels. Figure 6(b) shows the modified summation. Note that the validity of sign estimation depends on the carry-sum representation of the partial product, thus the 3-vector representation, (S, C_1, C_2), has to sum up to a 2-vector format before sign estimation takes place. The output of the first CSA provides the 2 vectors needed for the sign estimation. With these modifications, we can now have a DAG that has just $n+3$ rows, one for each iteration. The DAG has almost the same dependence pattern as the one in Figure 4, except that each L cell depends on the 7 leftmost X cells one row above. The sign estimation is again represented as the L cell, and the X cell represents the two levels of single-bit CSA and the logic to produce qM and A_iB. Grouping of each L cell with the 5 leftmost X cells in each row and pairing up the remaining X cells again yields a simpler DAG similar to the one in Figure 5. Efficient semisystolic and systolic array processors can be designed for the algorithms using these DAGs.

Figure 6 goes here

The modular multiplication algorithm can be realized by a semisystolic, one-dimensional, two-stage pipelined design for use in modular-exponentiation-based computations. The first stage computes the shift-add-reduce of the **for** loop. The second stage completes the modular multiplication by performing the final summation and selection. One n-bit modular multiplication is completed every $n+3$ clock cycles, and the pipelining depth of 2 matches 2 multiplications per iteration of exponentiation. For k-bit exponent and n-bit modulus, $2k(n+3)$ clock cycles are required to compute a modular exponentiation.

We are currently investigating a variant of the modular multiplication algorithm that accepts multiplicand and multiplier in carry-sum pair format, as in [5] and [8]. Such a formulation will eliminate the need for final summation and selection stage in the context of modular exponentiation, and may reduce the latency and the hardware cost.

References

[1] G. Alia and E. Martinelli. A VLSI structure for X (mod m) operation. *Journal of VLSI Signal Processing*, 1:257–264, 1990.

[2] D. E. Atkins. Higher-radix division using estimates of the divisor and partial remainders. *IEEE Transactions on Computers*, 17(10):925–934, October 1968.

[3] F. Barsi. Mod m arithmetic in binary systems. *Information Processing Letters*, 40(6):303–309, December 1991.

[4] T. Beth and D. Gollmann. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications*, 7(4):458–466, May 1989.

[5] E. F. Brickell. A fast modular multiplication algorithm with application to two key cryptography. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *Advances in Cryptology, Proceedings of Crypto 82*, pages 51–60. New York, NY: Plenum Press, 1982.

[6] E. F. Brickell. A survey of hardware implementations of RSA. In G. Brassard, editor, *Advances in Cryptology — CRYPTO 89, Proceedings*, Lecture Notes in Computer Science, No. 435, pages 368–370. New York, NY: Springer-Verlag, 1989.

[7] S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT 90*, Lecture Notes in Computer Science, No. 473, pages 230–244. New York, NY: Springer-Verlag, 1990.

[8] S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, June 1993.

[9] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[10] National Institute for Standards and Technology. Digital signature standard (DSS). *Federal Register*, 56:169, August 1991.

[11] K. Hwang. *Computer Arithmetic, Principles, Architecture, and Design*. New York, NY: John Wiley & Sons, 1979.

[12] Ç. K. Koç and C. Y. Hung. Carry save adders for computing the product AB modulo N. *Electronics Letters*, 26(13):899–900, 21st June 1990.

[13] Ç. K. Koç and C. Y. Hung. Multi-operand modulo addition using carry save adders. *Electronics Letters*, 26(6):361–363, 15th March 1990.

[14] Ç. K. Koç and C. Y. Hung. Bit-level systolic arrays for modular multiplication. *Journal of VLSI Signal Processing*, 3(3):215–223, 1991.

[15] S. Y. Kung. *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[16] S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi, and N. Takagi. Design of high speed MOS multiplier and divider using redundant binary representation. In M. J. Irwin and R. Stefanelli, editors, *Proceedings, 8th Symposium on Computer Arithmetic*, pages 80–86, Como, Italy, May 19–21, 1987. Los Alamitos, CA: IEEE Computer Society Press, 1987.

[17] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[18] F. P. Preparata and J. E. Vuillemin. Practical cellular dividers. *IEEE Transactions on Computers*, 39(5):605–614, May 1990.

[19] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[20] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In E. Swartzlander, Jr., M. J. Irwin, and G. Jullien, editors, *Proceedings, 11th Symposium on Computer Arithmetic*, pages 252–259, Windsor, Ontario, June 29– July 2, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[21] N. Takagi and S. Yajima. Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem. *IEEE Transactions on Computers*, 41(7):887–891, July 1992.

[22] C. D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–378, March 1993.

[23] J. H. P. Zurawski and J. B. Gosling. Design of high-speed digital divider units. *IEEE Transactions on Computers*, 30(9):691–699, September 1981.

Figure Captions

Figure 1: Examples for the lemma used to prove Theorem 1

Figure 2: Summary of Theorem 1 for the sign estimation procedure

Figure 3: Summary of Theorem 2 for the tuned sign estimation

Figure 4: A section of DAG for the modular multiplication algorithm

Figure 5: A section of DAG for modular multiplication after grouping of cells

Figure 6: Two-level carry-save additions for the modular multiplication algorithm. The "$\leftarrow$" denotes left-shift by one bit.

Figures

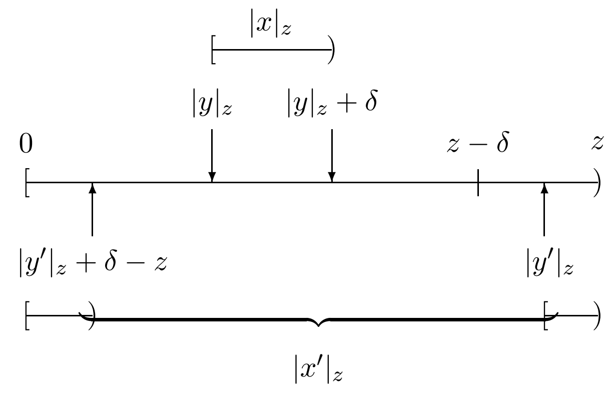


Figure 1: Examples for the lemma used to prove Theorem 1

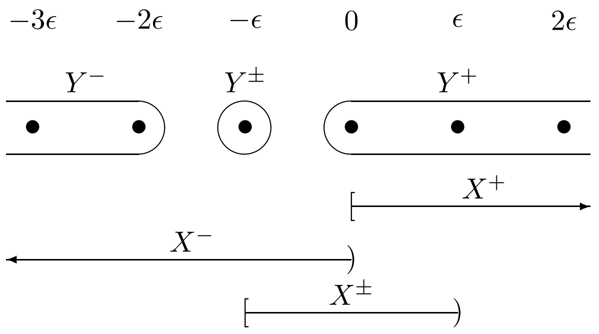


Figure 2: Summary of Theorem 1 for the sign estimation procedure

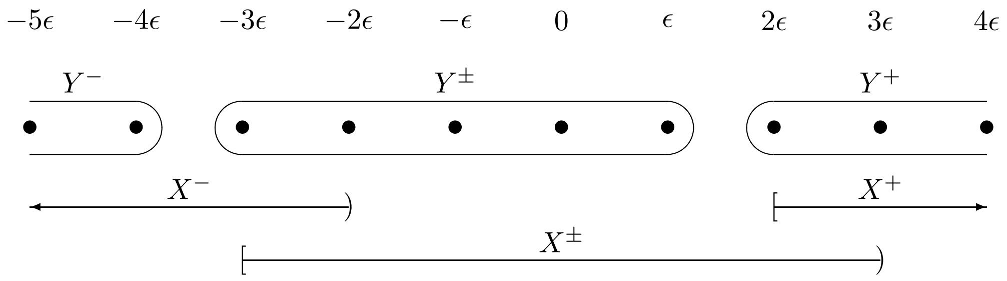


Figure 3: Summary of Theorem 2 for the tuned sign estimation

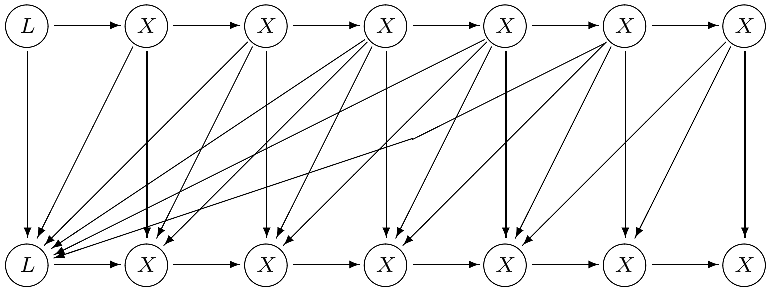


Figure 4: A section of DAG for the modular multiplication algorithm

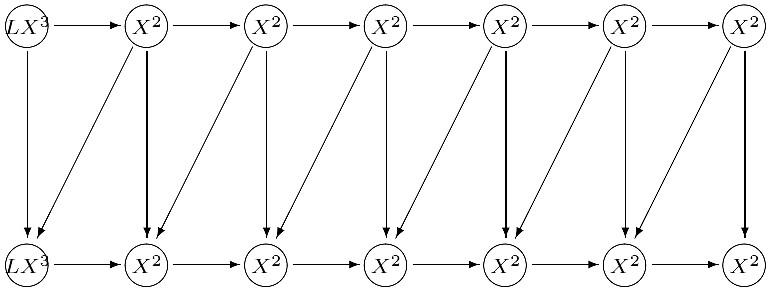
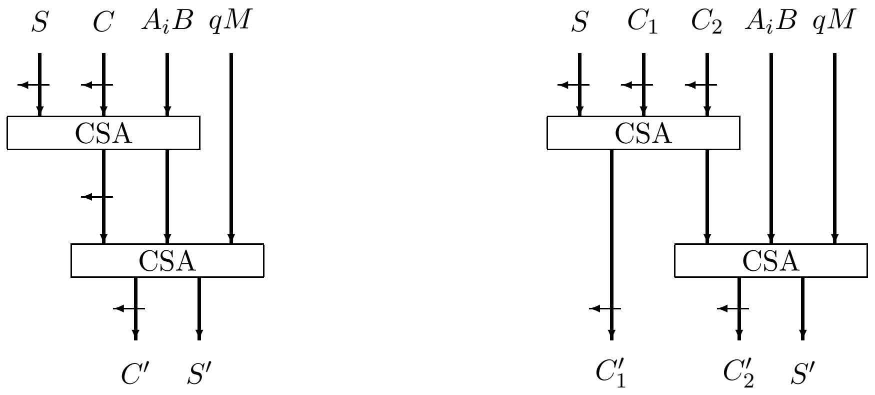


Figure 5: A section of DAG for modular multiplication after grouping of cells



(a) Four-to-two summation (b) Five-to-three summation

Figure 6: Two-level carry-save additions for the modular multiplication algorithm. The ”$\leftarrow$” denotes left-shift by one bit.