A $\theta(logn)$ ALGORITHM FOR MODULO MULTIPLICATION*

K. M. Elleithy and M. A. Bayoumi

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504, U.S.A.

ABSTRACT

A $\theta(log\ n)$ algorithm for large moduli multiplication for Residue Number System(RNS) based architectures is proposed. The proposed modulo multiplier is much faster than previously proposed multipliers and more area efficient. The implementation of the multiplier is modular and is based on simple cells which leads to efficient VLSI realization. A VLSI implementation using 3 micron CMOS process shows that a pipelined n-bit modulo multiplication scheme can operate with a throughput of 30 M operation per second.

1. Introduction

Recently, RNS is receiving an increased attention due to its ability to support high-speed concurrent arithmetic[1]. Applications such as fast fourier transform, digital filtering, and image processing utilize the high speed RNS arithmetic operations; addition and multiplication, they do not require the difficult RNS operations such as division and magnitude comparison. The technological advantages offered by VLSI have added a new dimension in the implementation of RNS-based architectures[2]. Several high speed VLSI special purpose digital signal processors have been successfully implemented[3-5].

The scope of this paper is modulo multiplication. In general, look-up tables and PLA's[6,7] have been the main logical modules used when the data granularity is the word. It has been found that such structure is only efficient for small size moduli. For medium size and large moduli, bit-level structures are more efficient, where the data granularity is the bit[8]. Most of the reported works for large moduli multipliers are based on using special set of moduli. In [6,9] the moduli should be prime numbers. In [10,11] the moduli should be from the set $(2^n-1\ ,\ 2^n\ ,\ 2^n+1)$. Due to the constraints imposed on the chosen moduli, such approaches have limited applications.

In this paper, we present a modulo multiplier for medium size and large moduli. The multiplier is based on using a $\theta(1)$ modulo adder[12]. It is configured as a two dimensional array of very simple cells (modulo adder). The modulo multiplication is performed in $\theta(logn)$ steps with no constraints imposed on the chosen moduli.

2. Residue Number System

In RNS, an integer , X, can be represented by N-tuple of residue digits,

$$X = (r_1,\ r_2,\ .....,\ r_N)$$

where $r_i = \left|X\right|_{m_i}$, with respect to a set of N moduli $\{m_1,\ m_2,\ .....,\ m_N\}$. In order to have a unique residue representation, the moduli must be pairwise relatively prime, that is,

$$GCD(m_i,\ m_j) = 1, \qquad \text{for } i \neq j$$

then it is shown that there is a unique representation for each number in the range of $0 \leq X < \prod_{i=1}^{N} m_i = M$ where N is the number of moduli.

The arithmetic operation on two integers A and B is equivalent to the arithmetic operation on its residue representation, that is,

$$\left|A \cdot B\right|_M = \left(\left|\left|A\right|_{m_1} \cdot \left|B\right|_{m_1}\right|_{m_1}, \left|\left|A\right|_{m_2} \cdot \left|B\right|_{m_2}\right|_{m_2}, ..., \left|\left|A\right|_{m_N} \cdot \left|B\right|_{m_N}\right|_{m_N} \right)$$

where '$\cdot$' can be addition, subtraction, or multiplication. It is desired to convert binary arithmetic on large integers to residue arithmetic on smaller residue digits in which the operations can be parallelly executed, and there is no carry chain between residue digits.

§2.1 The Modulo Multiplication

Generally, multiplication modulo m has $2^n - m$ ($n = \lceil \log m \rceil$) incorrect residue states. These states are in the range $[m\ ,\ 2^n - 1]$ which may be called overflow states. The corrected residue numbers can be obtained by two methods; employing a binary adder or a correction table. In the first method, a constant $(2^n - m)$ is added to correct the overflow residue states (generalized end-round carry) as shown in Figure 1. The multiplication is performed as follows:

$$y = \left|x_1 * x_2\right|_m = \begin{cases} x_1 * x_2 & \text{if } x_1 * x_2 < m \\ x_1 * x_2 - m & \text{if } x_1 * x_2 \geq m \end{cases}$$

A n−bit multiplier and an inner-product cell are used; the multiplier computes $x_1 * x_2$, while the inner-product cell computes $x_1 * x_2 - m$. The carry bit generated from the second unit indicates whether or not $x_1 * x_2$ is greater than m. A multiplexer, controlled by the carry, selects the correct output. In the second method, a look-up table is used to correct the incorrect residue states $(2^n - m)$, Figure 2. The first algorithm of modulo multiplication is slow, and the second algorithm is not suitable for medium and large moduli.

3. Modulo Adder

The modulo adder is the basic kernel used in performing the modulo multiplication. The modulo adder is based on representing a number as a *Carry* and a *Sum* to obtain a scheme that has a constant speed which does not depend on the number of bits[12]. The modulo adder is used to add two numbers A and B in modulo m. Figure 3 shows that A is represented as a pair of numbers $(A_S\ ,\ A_C)$, B is also represented as $(B_S\ ,\ B_C)$, and the output C is represented as $(C_S\ ,\ C_C)$. Each number is represented as a group of *Sum* bits and *Carry* bits. There is no unique representation for A_S and A_C. The condition that need to be satisfied is:

$$\left|A_S + A_C\right|_m = \left|A\right|_m$$

One possible representation is:

$$A_S = \left|A\right|_m \qquad A_C = 0$$

The choice of a representation has no implication on the complexity of the design. With such representation, four numbers $(A_S\ ,\ A_C\ ,B_S\ ,B_C)$ need to be added, two steps of CSA are required. After the addition process we need to detect if $-M$ or $2*(-M)$ is required to adjust the result. The adjusting process takes at most three steps. Since the adder has a fixed number of steps; five; no matter how long is A and B, it can be used in a multioperand pipelined addition scheme[13] and in the implementation of a pipelined architecture of the CRT[12].

§3.1 The Modulo Addition Algorithm

The proposed algorithm for modulo m addition of two numbers can be described as follows:

Algorithm modulo_add (A , B , Result)

Input: Two variables A and B in modulo m, A is represented as A_S and A_C. B is represented as B_S and B_C. All variables are n bit numbers $(2^{n-1} \leq m \leq 2^n)$.

*This work was supported in part by NSF grant No. MIP-8809811.

89CH2785-4/90/0000-0353$01.00 © 1990 IEEE

Authorized licensed use limited to: University of Bridgeport. Downloaded on February 24,2010 at 12:52:51 EST from IEEE Xplore. Restrictions apply.

Output: Variable *Result* represented as $Result_C$ and $Result_S$. The relation between A, B, and *Result* is: $Result = \left| A + B \right|_m$.

Procedure:

```
begin
 Do in parallel
   begin
     Call Sum(temp1 , AS , AC , BS)
     Call Carry(temp2 , AS , AC , BS)
   end
 Do in parallel
   begin
     Call Sum(temp3 , temp1 , temp2 , BC)
     Call Carry(temp4 , temp1 , temp2 , BC)
   end
 Case ( temp2[n+1]+temp4[n+1] ) of
   0: Do in parallel
        begin
          ResultS := temp3
          ResultC := temp4
        end
      exit
   1: Do in parallel
        begin
          Call Sum(temp5 , temp3 , temp4 , (2^n−m))
          Call Carry(temp6 , temp3 , temp4 , (2^n−m))
        end
   2: Do in parallel
        begin
          Call Sum(temp5 , temp3 , temp4 , 2*(2^n−m))
          Call Carry(temp6 , temp3 , temp4 , 2*(2^n−m))
        end
 end case
 Case ( temp6[n+1] ) of
   0: Do in parallel
        begin
          ResultS := temp5
          ResultC := temp6
        end
      exit
   1: Do in parallel
        begin
          Call Sum(temp7 , temp5 , temp6 , (2^n−m))
          Call Carry(temp8 , temp5 , temp6 , (2^n−m))
        end
 end case
 Case ( temp8[n+1] ) of
   0: Do in parallel
        begin
          ResultS := temp7
          ResultC := temp8
        end
   1: Do in parallel
        begin
          Call Sum(temp9 , temp7 , temp8 , (2^n−m))
          Call Carry(temp10 , temp7 , temp8 , (2^n−m))
        end
      Do in parallel
        begin
          ResultS := temp9
          ResultC := temp10
        end
 end case
end.

Sum (A , B , C , D)
begin
  Do in parallel (1≤i≤n)
     A[i] :=(B[i]∧C[i]) ∨ (B[i]∧D[i]) ∨(C[i]∧D[i])
end

Carry (A , B , C , D)
begin
  A[1] := 0
  Do in parallel (1≤i≤n)
     A[i+1] :=B[i] ⊕ C[i] ⊕ D[i]
end
```

An implementation of the algorithm is shown in Figure 4. The proof that the modulo adder scheme for adding two n-bit numbers in modulo m has an asymptotic time complexity $\theta(1)$ is shown in [12] with an example for the addition operation.

4. The Proposed Modulo Multiplier

The modulo adder presented in the previous section is the main kernel of the modulo multiplier. The multiplier consists of two stages. In the first stage (Figure 5) an array of *AND* gates is used to obtain the partial products. The second stage of the adder is a binary tree of modulo adders used to perform the addition of the n partial products (Figure 6). The correctness of the operation is established by the following theorem.

Theorem 1: *Adding n numbers $(y_1 , y_2 , \cdots , y_n)$ in modulo M is equivalent to :*

[1] Adding (y_1 , y_2) modulo M ,....,$(y_i , y_{i+1}) , \ldots ,$ and (y_{n-1} , y_n) gives $y_{12} , \ldots , y_{(n-1)n}$.

[2] Step [1] is repeated on (y_{12} , y_{34}) ,..., $(y_{(n-3)(n-2)} , y_{(n-1)n})$.

[3] Step [2] is repeated for $\lceil \log N \rceil - 2$ times to obtain one final output represented as a sum and carry.

Proof: To add two numbers a and b in modulo M we have the following cases:

(i) $a < M$ and $b < M$ then $a = \left| a \right|_M$ and $b = \left| b \right|_M$, then:

$$\left| a+b \right|_M = \left| a_M + b_M \right|_M \quad (1)$$

(ii) $a > M$ and $b < M$ then $b = \left| b \right|_M$ and $a = M + x$, then:

$$\left| a+b \right|_M = \left| M+x+b \right|_M = \left| x+b \right|_M \quad (2)$$

Since $x < M$ and $b < M$, then from (1) and (2):

$$\left| a + b \right|_M = \left| a_M + b_M \right|_M \quad (3)$$

(iii) $a > M$ and $b < M$ like case (ii).

(iv) $a > M$ and $b > M$ then $a = M+x$ and $b = M+y$, then:

$$\left| a+b \right|_M = \left| M+x+M+y \right|_M = \left| x+y \right|_M = \left| a_M + b_M \right|_M \quad (4)$$

From the previous four cases:

$$\left| a+b \right|_M = \left| a_M + b_M \right|_M \quad (5)$$

Since addition is associative then:
Using (5) we have:

$$= \left| \left| y_1 + \ldots + y_{\frac{n}{2}} \right|_M + \left| y_{\frac{n}{2}+1} + \ldots + y_n \right|_M \right|_M$$

We can further expand this expression using the same method to get the addition process in the right hand side in terms of only two operands added in modulo M. □

Theorem 1 means that adding n numbers in modulo M can be performed using a binary tree consists of units that are capable of adding only two numbers in modulo M. The modulo adders are used as those building blocks to perform the addition process. Since the modulo addition requires that inputs be represented in the form of sum and carry, then this form should be enforced at all levels. The form will be enforced automatically for levels ≥ 2, because the outputs of the previous levels are in the correct form. For first level we have the following:

$$Y_{iS} = y_i \ , \ Y_{iC} = 0 \quad \forall\, 1 \leq i \leq n$$

For the second stage the output is in the form of sum and carry which is exactly the same form we have using the CSAs.

5. Modulo Multiplier Evaluation

§5.1 Asymptotic Complexity

Using the VLSI model of computation for asymptotic complexity[14], a comparative study for the proposed multiplier is analyzed. For multiplier I (Figure 1) the complexity measures will be as follows:

$$A = = \theta(n^2)$$

$$T = \theta(n)$$

$$AT^2 = \theta(n^4)$$

Authorized licensed use limited to: University of Bridgeport. Downloaded on February 24,2010 at 12:52:51 EST from IEEE Xplore. Restrictions apply.

For multiplier *II* (Figure 2), using the complexity analysis of the correction table of[14]:

$$A = O(n^2 + 2^n\ n) = O(n\ 2^n)$$
$$T = O(n)$$
$$AT^2 = O(n^3 2^n)$$

For the proposed multiplier, the first stage consists of n^2 *AND* gates. The area of this stages is $\theta(n^2)$. The partial products are obtained after a constant time (The *AND* gate delay). For the second stage number of adders required to perform this stage is:

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots\ldots + 1 = n - 1 = \theta(n)$$

Since each modulo adder has $\theta(n)$ full adders, then the total area required for this stage is $\theta(n^2)$. The time required to perform the addition process is the delay of the modulo adder multiplied by the depth of the tree $\theta(logn)$.

$$A = \theta(n^2)$$
$$T = \theta(\log\ n)$$
$$AT^2 = \theta(n^2(log\ n)^2)$$

From the previous analysis it is clear that the proposed multiplier is superior than previously proposed schemes for medium and large moduli.

§5.2 Layout Complexity

An 8-bit multiplier has been implemented based on Domino Logic using a double metal 3 micron CMOS process. Table I summerizes the area and time of different components of the design. For the modulo adder we use three cell types. Type I is used in the implementation of stages one and two, Type II is used in the implementation of stage three, and Type III is used in the implementation of stages four and five. Type I is simply a n full-adders, Type II consists of a three input multiplexer and n full-adders, and Type III consists of two input multiplexer and n full-adders. The longest delay among the three types is for type III. The length of type III delay determines the clock period. The clock period is 32.40 ns which gives a throughput of 31 M modulo multiplication operation per second.

Cell	Area	Time Delay
And array	37 * 452	1.45 ns
Latch	232 * 73	5.30 ns
Type I	154 * 164	15.83 ns
Type II	215 * 164	32.40 ns
Type III	298 * 164	17.20 ns
Modulo adder	1557 * 3397	124.96 ns
Multiplier	7569 * 14547	249.92 ns

Table I. Area and Time Delay of the 8 bit multiplier

We can generalise the previous figures for n-bit modulo multiplication. The total delay consists of the *AND* gates array delay and the tree's delay. The array has a delay of 1.45 ns and each adder has a delay of 124.9 ns, then:

Total delay = 124.9 * log n + 1.45 ns.

The clock period is constant regardless of n, then:

Clock period = 32.40 ns.

Throughput = 30 M modulo-multiplication/s.

6. Conclusions

The modulo multiplier introduced in this paper has a total time-delay complexity of $\theta(log\ n)$ for multiplying two n-bit numbers in modulo m. Based on the analysis of section 5.1, this adder is the fastest and the most area efficient for large moduli. The VLSI implementation using double metal 3 micron shows that the pipelined multiplier can operate with a clock rate of 32.4 ns, which leads to a throughput of 30 M modulo multiplication operation per second. The proposed design has the following advantages:

1) It does not have any limitation on the size of the modulus.
2) It is quite modular, it is a 2-D array of two cell types (modulo adder , ROM).
3) It is easy to pipeline yielding a very high throughput.
4) It is very fast and area-efficient compared with other schemes.

References

[1] F. J. Taylor, "Residue Arithmetic: A Tutorial with Examples," IEEE Computer Magazine, pp. 50-62,May 1984.

[2] M. A. Bayoumi, G. A. Jullien, W. C. Miller, "A Look-up Table VLSI Design Methodology for RNS Structures Used in DSP Applications," IEEE Trans. on Circuits and Systems, pp. 604-616, Vol 34, No. 6, June 1987.

[3] M. A. Bayoumi, "A High Speed VLSI Complex Digital Signal Processor Based on Quadratic Residue Number System," VLSI Signal Processing II, pp. 200-211, IEEE Press, 1986.

[4] M. A. Bayoumi, "Digital Filter VLSI Systolic Arrays over Finite Fields for DSP Applications," Proc. of the 6th IEEE Annual Phoenix Conference on Computers and Communications, pp. 194-199, Feb. 1987.

[5] W. Jenkins and E. Davidson, "A custom-designed Integrated Circuit for the Realization of Residue Number Digital Filters," Proc. ICASSP 1985, pp. 220-223, March 1985.

[6] M. A. Soderstrand and C. Vernia, "A High-Speed Low-Cost Modulo P_i Multiplier with RNS Arithmetic Application," Proc. IEEE, vol. 68, pp. 529-532, Apr. 1980.

[7] M. A. Soderstrand and E. L. Fields, "Multipliers and Residue Number Arithmetic Digital Filters," Electronic letters, vol. 13, no. 6, pp. 164-166, Mar. 1977.

[8] K. M. Elleithy, "On Bit-Parallel Processing for Modulo Arithmetic," VLSI Technical Report TR86-8-1, The Center for Advanced Computer Studies, University of Southwestern Louisiana, 1986.

[9] G. A. Jullien, "Implementation of Multiplication Modulo a Prime Number with Applications to Number Theoretic Transforms," IEEE Trans. Comput., vol. C-29, no. 10, pp. 899-905, Oct. 1980.

[10] F. J. Taylor, "Large Moduli Multipliers for Signal Processing," IEEE Trans. Circuits and Systems, vol. CAS-28, no. 7, July 1981.

[11] F. J. Taylor and C. H. Huang, "An Autoscale Residue Multiplier," IEEE Trans. Comput., vol. C-31, pp. 321-325, Apr. 1982.

[12] K. M. Elleithy, M. A. Bayoumi, and K. P. Lee, "$\theta(logN)$ Architectures for RNS Arithmetic Decoding," Proc. of the 9th symp. on computer arithmetic, pp. 202-209, 1989.

[13] K. M. Elleithy, "On the Bit-Parallel Implementation for the Chinese Remainder Theorem," VLSI Technical Report TR87-8-1, The Center for Advanced Computer Studies, University of Southwestern Louisiana, 1987.

[14] Magdy A. Bayoumi, "Lower Bounds for VLSI Implementation of Residue Number System Architectures," Integration, the VLSI Journal, vol. 4, No. 4, pp. 263-269, Dec. 1986.

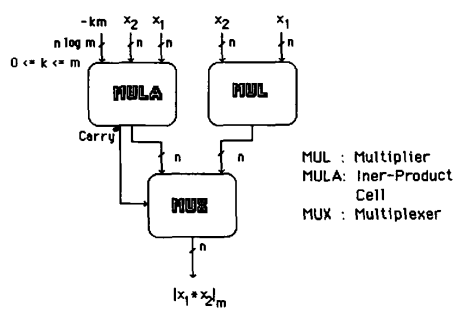


Figure 1. Modulo Multiplication Using a Multiplier and an Iner-Product Cell.

Authorized licensed use limited to: University of Bridgeport. Downloaded on February 24,2010 at 12:52:51 EST from IEEE Xplore. Restrictions apply.

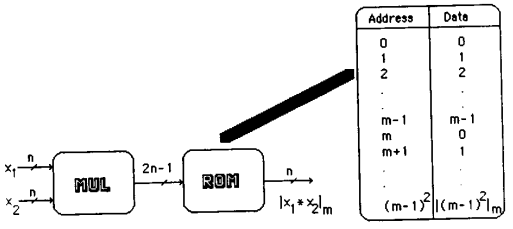


Figure 2. Modulo Multiplication Using Look-up table.

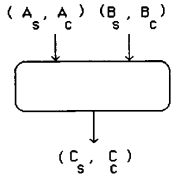


Figure 3. A Modulo Sum Adder

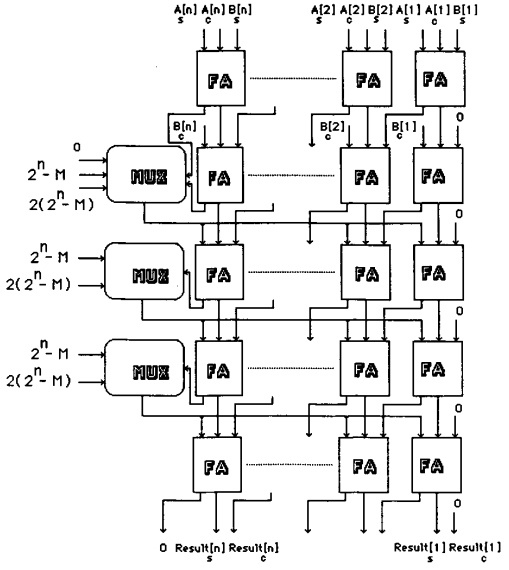


Figure 4. Different Stages of The Modulo Adder.

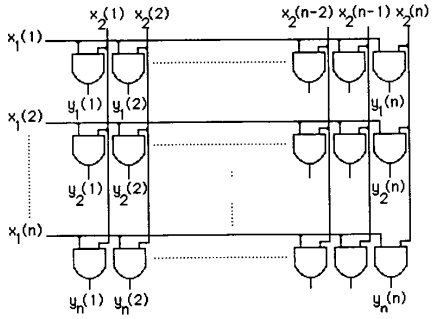


Figure 5. Implementation of Partial Products Generator.

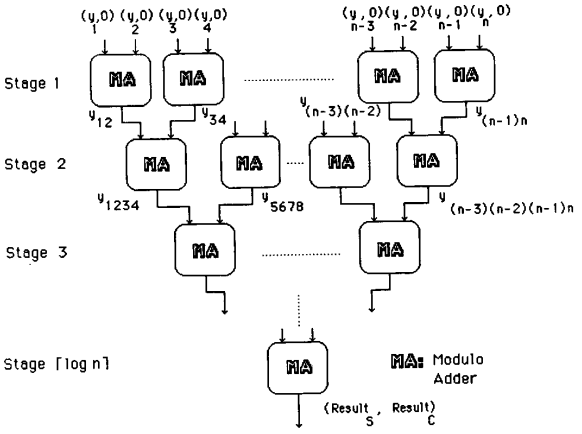


Figure 6. Addition of the Partial Products using Modulo Adders

Authorized licensed use limited to: University of Bridgeport. Downloaded on February 24,2010 at 12:52:51 EST from IEEE Xplore. Restrictions apply.