An Algorithm for Multiplication Modulo (2^N-1)

Zhongde Wang, G.A. Jullien and W.C. Miller

VLSI Research Group
University of Windsor, Windsor, Ontario, Canada N9B 3P4

Abstract

This paper propose an efficient algorithm for multipliers modulo $(2^N - 1)$ *. To achieve high speed, the Wallace tree is adopted for the multipliers. The proposed Wallace tree multipliers exhibit much more resular structure than binary Wallace tree cultipliers. Comparison with a previous counterpart shows favorable to our multiplier in both speed and hardware..*

1. Introduction

Residue number systems (RNS) possesses the advantages of parallel, carry-free and thus high speed processing It. has found applications in digital signal processing [1][2][3]. Among different moduli that are suitable for RNS application, modulus $(2^N - 1)$ is one of the major choice That is why out of the twelve RNS classes suggested by [1] and [2] for signal processing applications, ten include $(2^N - 1)$ as one or two of its moduli. Since the operation of multiplication is of major importance for almost all kind of processor, efficient implementation of multiplication modulo $(2^N - 1)$ is important for the application of RNS.

Conventionally, look-up-table is the only method for modulo arithmetic, especially for modulo multiplication [4]. Unfortunately, this approach has a major disadvantage. The size of the look-up table grows exponentially with the size of the modulus. When N is large, this approach becomes unrealizable.

Recently, a new multiplier modulo $(2^N - 1)$ was proposed [5], which reduce the size of the look-up table by a factor up to 48, at the expense of some additions and square operations. Although the size of the look-up table has been considerably reduced, the look-up table is still a necessity for the approach proposed in [5], and the trend, that the look-up table grows exponentially with the size of the modulus, is not changed. Therefore, the approach in [5] is still not applicable for large modulus.

In this paper, we propose an approach for multiplication modulo $(2^N - 1)$. Our approach simply follows the lines of binary multipliers, but exhibits a more regular structure than a binary multiplier. The Wallace tree is employed achieve high speed. In recognizing the unequal delay of a full adder, an algorithm for delay optimization of the Wallace tree in intorduced.

This paper is organized as follows. Section 2 introduces the algorithm for multiplication modulo $(2^N - 1)$. Section 3 discusses the application of Wallace tree to multiplier modulo $(2^N - 1)$. The unequal delay of a full adder is discussed in section 4. An algorithm for delay optimization of the Wallace tree is introduced in section 5. A comparison of our approach to a recent reparted approach is made in section 6, and the conclusions are followed.

2. Algorithm for multiplication modulo $(2^N - 1)$

Let

$$A = \sum_{k=0}^{n-1} a_k 2^k \tag{1}$$

or alternatively, $A = \{a_{n-1} a_{n-2} ...... a_0\}$, be the multiplicand. Then it is easy to show that A multiplied by a power of 2, 2^j, results a left cyclic shifting of j bits.

$$\begin{aligned} A \cdot 2^j &= \sum_{k=0}^{n-1} a_k 2^{k+j} \\ &= \sum_{k=0}^{n-1-j} a_k 2^{k+j} + \sum_{k=0}^{j-1} a_{n-j+k} 2^k \ \ \mathrm{Mod}(2^N - 1) \end{aligned} \tag{2}$$

or, in bit representation,

$$A \cdot 2^j = \{a_{n-1-j} a_{n-2-j} ...... a_0 a_{n-1} a_{n-2} ...... a_{n-j}\} \ . \qquad \mathrm{Mod}(2^N - 1) \ . \tag{3}$$

Now let

$$B = \sum_{m=0}^{n-1} b_m 2^m \tag{4}$$

be the multiplier. Then

$$AB = \sum_{m=0}^{n-1} b_m A \ 2^m \ \ \mathrm{Mod}(2^n - 1) \tag{5}$$

Similar to the binary multiplier, $b_m A \ 2^m \ \ \mathrm{Mod}(2^N - 1)$ can be treated as a row of partial products, which is the ANDing of b_m and the bit row of Eq. (). The multiplication is

thus converted to the summation of the partial product array together to get the final product. There is a major difference between the binary multiplier and the modulo $(2^N - 1)$ multiplier. The partial product array of the binary multiplier is shaped as a triangle [9].. While the partial product array of the modulo $(2^N - 1)$ multiplier is shaped as a rectangle. All rows of partial products for the modulo $(2^N - 1)$ multiplier are located on the same range of bit position. Since the height of all columns of the partial product array of he modulo $(2^N - 1)$ multiplier is the same, N, the column size compression by Wallace tree is easy to apply.

3. Wallace Tree

Similar to the binary multiplier, after the partial product array is obtained, how to sum up the partial product array to get the final result is of great importance for the speed of the multiplier, because this is the major portion of the time for the multiplication. It is well known that the Wallace tree [6], or alternately the Dadda method [7], is an efficient method to reduce the column size for binary multipliers. Unfortunately, when Wallace tree is applied, the structure of the binary multiplier becomes complicated and the irregular interconnect between cells (full adders) becomes a major obstacle. That is why the VLSI implementation of Wallace-tree-type binary multiplier [8] appears long after the method was suggested.

For a binary multiplier the column size for different column is different. This is the reason which causes the complicated structure and interconnection of the Wallace-tree-type binary multipliers.

In contrast, for a modulo $(2^N - 1)$ multiplier , the column size of any column is the same. The Wallace tree is very regular and much easier to be implemented, The only thing have to be kept in mind is that the carry out of the most significant bit should be shifted to the least significant bit position

A numerical example, 155(10011011) x 109(01101101) = 65(01000001) Mod $(2^8 - 1)$, using Wallace tree structure, is shown in Fig. 3.

In Figure 1, every three rows of either partial products, or intermediate sums or carries, are compressed into two rows using full adders. The sum row and carry row are marked as s and c, respectively. Each carry out from the MSB is underlined and is shifted to the LSB position on the same row. The shifted bit is also underlined. **A** is the step for generating the partial product array. **B** through **E** are steps for column compression by Wallace tree, using full adders only, to compress the column size from eight up to two. Each step in this stage needs time of a full adder delay. The total delay of this stage is four full adder delays. The final two steps, **F** and **G**, represent a fast adder modulo 255, where **F** represents a 8 bit fast adder, and **G** represents the carry correction for modulo 255 operation.

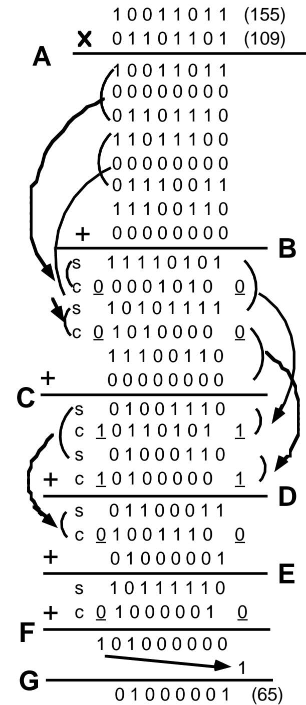

Figure 1 A numerical example of multiplication modulo $(2^8 - 1)$

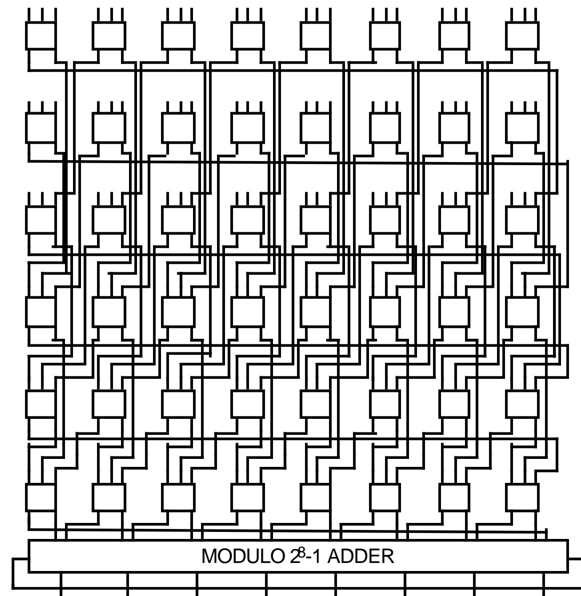

Figure 2 An architecture for multiplier modulo $(2^8 - 1)$.

The architecture of multiplier modulo $(2^8 - 1)$, using above approach, is shown in Figure 2, where each block represents a full adder, with carry-out from the left hand side of the bottom, and sum from right. The final step, the modulo $(2^8 - 1)$ adder is achieved by directly feeding the carry-out of a fast 8-bit adder to its carry-in.

It can be seen from Figure 2 that, except the adders at the MSB position, The architecture of adders in any column is identical. This feature provides a big advantage for layout, while the counterpart of the binary multiplier, as shown in [8], does not possess this advantage.

4. Comparison

Since Skavantzos and Rao has shown that their approach for multiplication modulo $(2^N - 1)$ is more efficient than conventional approach [5], we shall compare our approach with their approach only.

In order to make a fair comparison, we have to make a brief review of the approach in [5], which may be summarized as follows:

1. Decomposing the multiplicand A and the multiplier B into k N/k-bit bytes x_i and y_i, i=0,1,...,k-1. Typical choice is k=4.

2. Computing 12 intermediate variables, a_4, b_4,..., k_4, and l_4, which are either four or eight element combinations of x_i and y_i.

3. Squaring those 12 intermediate variables.

4. Computing another group of four intermediate variables P, Q, R, and S, which are four element combinations of the 12 squared variables.

5. Computing z_i, i=0,1,...,k-1, which are two element combinations of P, Q, R, or S.

6. Evaluate the final result from k terms of summation modulo $(2^N - 1)$. For k=4,

$$\langle A \cdot B \rangle_{2^N - 1} = \langle z_0 + z_1 2^{N/4} + z_2 2^{N/2} + z_3 2^{3N/4} \rangle_{2^N - 1} \quad (6)$$

where $\langle . \rangle_m$ represents modulo m operation.

In the following we will show that the proposed scheme for multiplication modulo $(2^N - 1)$ is more advantageous, in terms of both hardweare and speed, than Skavantzos and Rao's scheme.

Let us compare the case for N=16 and k=4. The final evaluation (Eq. (6)) is a four term summation modulo $(2^{16} - 1)$. [5] did not give the detail of the procedure of evaluation. We assume it is evaluated by our approach. Then we only have to compare the rest part of the two approaches. Actually, evaluation of Eq. (6) is equivalent to steps E, F, G, and H in Figure 2. Thus, the procedure of steps 1 through 5 of the approach in [5] is equivalent to the partial product generation and part of the Wallace tree of our approach, which compresses the column size from sixteen to four, identified as steps A, B, C, and D in Figure 2.

The comparison is made on two aspects, the delay and the hardware requirement.

First, we estimate the delay for evaluation of z_i form x_i and y_i, i=0, 1, 2, and 3, for the approach in [5]. We assume that no delay is caused for the decomposing the multiplicand and the multiplier into 4 bit bytes. Each of x_i or y_i is a 4 bit word. The sum of two of them should be represented by 5 bits. The sum of four of them should be represented by 6 bits. And the sum of eight of them should be represented by 7 bits. From Fig. 1 of [5] it is clear that the delay for evaluating a_4, b_4, c_4, and d_4 from x_i and y_i, the step 2 of the approach in [5], is one 4 bit adder delay, plus one 5 bit adder delay, plus one 6 bit adder delay. The delay for squaring (step 3) is the delay of a look-up table reading. The square of a 7 bit word should be represented by 14 bits. It is also clear from Fig. 1 of [5] that the delay of step 4, to evaluate P, Q, R, and S form the squared variables a_4^2, b_4^2,...... is one 14 bit adder delay plus one 15 bit adder delay. It is not difficult to verify that P and Q are 16 bit words. But R and S are 15 bit words. Thus, to get 16 times z_0, z_1, z_2, and z_3 from P, Q, 2R, and 2S, as illustrated by Fig. 2 in [5], it requires a delay time of a 16 bit adder delay. Therefore, the total delay to evaluate z_i form x_i and y_i, for the approach in [5], is the delay of 6 adders with lengths of 4, 5, 6, 14, 15, 16 bits, respectively, and the delay of a look-up table reading.

Now we estimate the delay of our approach for compressing the column size from sixteen up to four. The delay for generating the partial product array is simply an AND gate delay. From Figure 2 it is clear that the delay for compressing the column size from sixteen to four is only the delay of four full adders. A full adder delay is definitely shorter than the delay of any adder of length more that one, and should be much shorter than an adder with large length, e. g. length 15. Therefore, one may judge that the delay of our approach is much shorter than the delay of the approach in [5].

Now we estimate the hardware requirement of the two approaches.

First, we estimate the hardware requirement for evaluation of z_i form x_i and y_i, i=0, 1, 2, and 3, for the approach in [5]. To evaluate P and Q form x_i and y_i, as indicated by Fig. 1 in [5], it requires four 4 bit adders, four 5 bit adders, four 6 bit adders, four look-up tables, two 14 bit adders and two 15 bit adders. In a similar manner, we may estimate that to evaluate R and S from x_i and y_i, it requires eight 4 bit adders, eight 5 bit adders, eight look-up tables, four 12 bit adders, and two 13 bit adders. To evaluate z_i from P, Q, R, and S, as indicated by Fig. 2 in [5], it requires four 16 bit adders. The total hardware required for evaluating z_i from x_i and y_i is

listed in Table 1.

Table 1 hardware requirement to evaluate z_i form x_i and y_i by approach of [5]

4 bit adder	5 bit adder	6 bit adder	12 bit adder	13 bit adder	14 bit adder	15 bit adder	look-up table
12	12	4	4	2	2	2	12

The carry ripple adder, which implement an N bit adder by cascading N full adders, is the slowest adder. But it requires the least hardware. Therefore, the lowest bound of hardware to implement all the adders in table 1 is equivalent to 12x(4+5)+4x(6+12)+2x(13+14+15) =264 full adders. In addition, this approach requires 26x2^9 ROM bits for the look-up tables, as given by table 1 in [5].

[5] made a comparison on the size of the look-up table for their method and the conventional method. The implication is that the look-up table is the major concern for hardware consideration. It is the major part of hardware for the multiplication. Consequently, the hardware for the adders mentioned above is comparatively smaller.

The estimation of the hardware for our approach is, on the other hand, much more easier. The generation of partial product array for a multiplier modulo $(2^{16}-1)$ requires 16x16=256 AND gates. To compress the column size from 16 to 4 requires 12x16=192 full adders. The purpose of our estimation of hardware requirement is for comparison only. It is not necessary to make a precise estimation. Therefore, we may assume that AND, NAND, or XOR gate occupies the same amount of silicon area. In its simplist form, the sum of a full adder requires two XOR gates. The carry requires two AND gates and tow OR gates. Thus, we may assume that a full adder is roughly equivalent to six AND gates. Consequently, the hardware required to convert the multiplication of two 16 bit numbers modulo $(2^{16}-1)$ into the summation modulo $(2^{16}-1)$ of four 16 bit numbers for our approach is 192+256/6=235 full adders, which is below the lowest bound of the hardware required by the approach of [5] even if the hardware of look-up table is not counted. Therefore, we may safely to say that our approach needs less hardware than the approach in [5], because the hardware for our approach is even less than the minor part of the hardware of the multiplier given in [5]. Since the size of the look-up table increase exponentially with the increase of the word length, N, the comparison for hardware will be more favorable to our approach when N becomes large.

5. Conclusion

In conclusion, a new approach for multiplication modulo (2^N-1) is proposed. Similar to the binary multiplier, the generation of the partial products is accomplished by AND gates. Wallace tree is applied to optimize the speed for compression of column size from N to two. To completely utilize the unequal delay of a full adder, an algorithm for delay optimization of the Wallace tree is developed. The proposed approach exhibits superior performance, in terms of either speed of hardware requirement, in comparison with a recent counterpart for the same purpose. In addition, the proposed multiplier modulo (2^N-1) shows an extremely regular structure and is very suitable for VLSI implementation.

6. Acknowledgments

The authors acknowledge support from the Natural Science and Engineering Research Council of Canada, and the Micronet Network of Centres of Excellence for funding this work. Design tools have been provided by the Canadian Microelectronics Corporation.

7. References

[1] W. K. Jenkins: “Recent advances in residue number techniques for recursive digital filtering” *IEEE Trans. Acoust. Speech, Signal Processing*, vol ASSP-27, pp. 19-30, 1979.

[2] M. H. Etzel and W. K. Jenkins: “The design of specialized residue classes for efficient recursive digital filter realization” *IEEE Trans. Acoust. Speech, Signal Processing*, vol ASSP-30, pp. 370-380, 1982.

[3] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, Eds. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*, IEEE Press, New York, 1986.

[4] M. A. Bayoumi, G. A. Jullien and W. C. Miller: “A look-up table VLSI design methodology for RNS structures used in DSP application”,.*IEEE Trans. Circuits and Systems,* vol CAS-34, pp. 604-616, 1987

[5] A. Skavantzos and P. B. Rao: “New multipliers modulo (2^N-1).”, *IEEE Trans. Computers.*, vol. C-41, pp. 957-961, 1992.

[6] C. S. Wallace: “A suggestion for a fast multiplier”, *IEEE Trans. Electronic Computers*, vol. EC-13, pp. 14-17, 1964.

[7] L. Dadda: “Some schemes for parallel multipliers”, Acta Frequenza, vol. 45, pp. 574-580, 1966.

[8] D. G. Crawley and G. A. J. Amaratunga: 8x8 bit pipelined Dadda multiplier in CMOS, IEE Proc. vol. 135 Pt. G, pp. 231-240, 1988.

[9] Zhongde Wang, G. A. Jullien, and W. C. Miller: “A new design technique for column compression multipliers”, *IEEE Trans. Computers*., vol. C-44, pp. 962-970, 1995.