An Alternative Algorithm for High Speed Multiplication and Addition Using Growing Technique

Reza Hashemian

Northern Illinois University, Electrical Engineering Dept.

Abstract-*An alternative algorithm is presented for multiplication/addition of variable bit-size operands. The algorithm is shown to be fast, and the computational time is variable and dependent on the accuracy requested. The growing nature of the product term, during the course of operation, gives the method some unique computational properties. The algorithm is implemented for the design of 32 X 32-bit multiplier.*

I. Introduction

A new algorithm is presented for simultaneous addition and multiplication of two integer numbers. The algorithm is also extended to include floating-point numbers. Although the method is originally developed for the multiplication operation, it is shown that, the sum of the operands is also obtained as a bi-product in this implementation, and without spending almost any extra effort, both in terms of time and hardware. Due to the gradual build up of both sum and product terms, starting from the most significant bits (MSB), the algorithm is suitable for any bit size operands, and it is, therefore, very flexible with respect to data size; and thus very portable. Another important property of this gradual build up is the possibility of terminating the multiplication in any specified position during the course of operation. This allows variable precision multiplication; thus, depending on the application, multiplications with different precision as well as different computational timing is available through this method. This is a very useful property, for example, manipulating certain data such as locating the boundary lines in a graphic image, or calculating a data address, may need much more accurate multiplication (and/or addition) than it is needed in a case of, for instance, coefficient manipulation in a digital filter design problem. This, so called *dynamic configurability* property of the proposed multiplication/addition procedure adds a new dimension to computer data manipulations. In traditional methods of multiplication a pair of multiplier and multiplicand are multiplied and the product is either located in a similar word size or in a "double" sized location.

In the first case some accuracy is lost, due to the truncation of the product term, while the computation is carried out for the entire range of accuracy. In the second case, however, it is often the job of software to double the space to accommodate the product, and this "doubling" the size can not certainly be continued for ever in a sequence of many multiplication operations.

The proposed multiplication procedure is shown to be fast. This speed is the result of three separate properties of the algorithm: First, the process jumps over multiple zeros with just shifting the product to the left (similar to the Booth's algorithm [1]), and it performs addition plus shifting, in the cases of non-zero entries. Second, the operation starts from single bit (MSB) operation and then gradually builds up to include full size operands; therefore, a logical conclusion is that the average operation (iteration) is equivalent to half of that for a full size operand. In anther words, if, in a worst situation, we assume proportionality between the operational time and the word size the multiplication speed in this method is expected to double, just due to the later property, alone. Third, the addition operation implemented in this algorithm is also new and it is shown to be fast. The speed in this addition results from the fact that the addition is basically reduced to a number of incrementing operations; and, as it has been reported earlier [2], incrementing can be performed with very high speed.

As discussed before, another useful property in this method is the presence of the sum term without any extra effort or time being spent. There are certainly many applications where both sum and product terms are needed, for which this algorithm is an ideal one to implement.

The proposed algorithm is implemented for the design of a multiplier/adder with variable size operands up to 32 X 32-bits. The *dynamic configurability*, discussed earlier, allows for flexibility in speed/accuracy expected from the multiplier. To manage such a flexibility, a timer (clock pulse counter) unit is set by the instruction to cut off and terminate the multiplication/addition operation for that particular application.

II. Algorithm Development

The algorithm is based on the following simple mathematical procedure:

Let $A_i = a_0 \cdot a_1\ a_2 \ldots a_{i-1}$, and $B_i = b_0\ b_1\ b_2\ b_{i-1}$; where, a_0 (b_0) is MSB, and a_{i-1} (b_{i-1}) is LSB of A_i (B_i). Also, let $A_{i,0}$ ($A_{i,1}$) represent A_i after a single shift to *the left with LSB replaced by a 0 (1) digit. Similarly* let $M_{i,00}$ denote M_i after two position shift to the left. Next, if we assume

$$M_i = A_i * B_i, \text{ and } S_i = A_i + B_i. \qquad (1)$$

Then for one digit higher operands A_{i+1} and B_{i+1} we can write

$$M_{i+1} = A_{i+1}*B_{i+1} = (A_{i,0} + a_i)*(B_{i,0} + b_i). \qquad (2)$$

Equation (2) can simply be expanded as:

$$M_{i+1} = A_{i,0}*B_{i,0} + a_i*B_{i,0} + b_i*A_{i,0} + a_i*b_i. \qquad (3)$$

An alternative representation of Eq. (3) is the truth table, shown in Table 1.

Table 1

Table of Summation and Multiplication

a_i b_i	S_{i+1}	M_{i+1}
0 0	$S_{i,0}$	$M_{i,00}$
0 1	$S_{i,1}$	$M_{i,00} + A_{i,0}$
1 0	$S_{i,1}$	$M_{i,00} + B_{i,0}$
1 1	$S_{i,1} + 1$	$M_{i,00} + S_{i,1}$

Table 1 presents some of the interesting properties of multiplication as well as addition, as presented in this algorithm:

1- In all three cases where (a_i, b_i) are equal to (0, 0), (0, 1), or (1, 0) the addition is progressed by applying a single shift to the previous sum term. In the case where both a_i and b_i are 1 an extra increment is required after the shift is performed.

2- Multiplication is performed by either a pair of shifting alone, shifting plus adding one of the operands, or shifting plus adding the sum term to the product. It is interesting to note that in no case the present sum term, that is, S_{i+1} is required for performing multiplication. Thus, the summation and the multiplication can be proceeded simultaneously.

III. Hardware Implementation

Multiplication/Addition (M/A) normally starts from a single bit M/A; in this case, it starts from the MSBs of the operands, i.e., a_0 and b_0. This situation, as we will discuss shortly, holds for the floating point numbers as well as integers with equal bit-size operands. Figure 1(a) shows the hardware implementation for a single bit M/A, which also serves as the initial stage for the general case of multi-bit operands. Notice that addition of a single wire through the *bus expander/shifter* satisfies the left shift for the carry term. Figure 1(b) shows a generalized *bus expander/shifter* through which two separate buses are merged, and, as a result, a wider bus is formed. In Fig. 2 a schematic representation of an adder is shown. The iteration process begins by allowing S_1, obtained from Fig. 1(a), to enter the loop via terminal S_{in}, and after certain predetermined number of iterations the operation terminates and the summand is accessed through S_{out}. The hardware implementation for the multiplier is shown in Fig. 3. Again note that the iteration process starts by entering M_1, from Fig. 1(a), through terminal M_{in}, and the iteration terminates by accessing the output from the terminal M_{out}. Also, note that the term S_i is the result of the intermediate summation, obtained in the adder circuit (Fig. 2). This indicates that summation and multiplication progress simultaneously, as we pointed out before. An alternative hardware implementation to remove the dependency of the product term M_{i+1} on the sum term S_i is shown in Fig. 4. Although two adders, instead of one, being used in this case, but the data is allowed to bypass the adders, if the condition is met. This slightly speeds up the process.

As it was pointed out, the procedure just discussed is applicable to equal bit-size operands, or floating point numbers. In the case of operands with different bit-size the procedure must be slightly modified. In this case, the MSBs of the two numbers do not, obviously, match (not in the same order of magnitude), and, therefore, we can not start the procedure from the two MSBs of the operands, as we did before. The way to get around this difficulty is to right adjust the two numbers such that they match in LSBs, and then begin the iteration from the MSB position of the smaller number. More precisely, let us consider two numbers A_k and B_j, and let A_k=<A_0><A_j> such that A_j and B_j have the same bit size; where, < > means concatenation. Now, in order to find the sum and product of A_k and B_j we may still use the relations given in Table 1 with exception that we start with i = 0, and we must assume that $S_0 = A_0$, $M_0 = 0$, and $B_0 = 0$. It is interesting to note that with this modification the hardware structure, as described in Figs. 2, 3, and 4, are basically unchanged. The iteration process starts from i = 0, in this case, and it terminates for i = j-1. In effect, we have been able to perform the multiplication (or summation) in j iterations rather than in k iterations (the number of bits in A_k). Clearly, the problem of right adjustment do not arise any difficulty in the case of floating point operations. This is because, in a floating point operand, we can indefinitely extend the fraction to the right without basically effecting its value; and thus, the right adjustment seams to be meaningless in this case (or it is always right adjusted). For floating point numbers the decimal point is adjusted which is identical to the left adjustment of the mantissa. For the fraction part, the addition or multiplication operation could arbitrarily be stopped at any stage of iteration without much effecting the result. The only consequence of this operation is the loss of relative accuracy in return for a higher speed.

IV. Parallel Operations

Although the gradual growth of the sum and product terms, discussed in this algorithm, make the method fast compared to other methods dealing with full size operands, but the iterative nature of this procedure is delaying the operation considerably, specially in the case of high density operands. To overcome this problem, we use operand partitioning. Suppose we wish to multiply two numbers A and B to get the product

$$M = A * B \qquad (4)$$

One way to reduce the number of iterations in this multiplication (or addition) is to first partition one, or both, of the operands into two or more parts, do partial multiplications concurrently, and then add up the results for the final product term. For simplicity, let us just partition B into B_0 and B_j such that $B = <B_0><B_j>$. Now, let

$$M_0 = A * B_0, \text{ and } M_j = A * B_j \qquad (5)$$

It is easy to prove that

$$M = <M_0><0_j> + M_j \qquad (6)$$

where, 0_j means j number of zero bits.

Now if we try to partition B into equal parts in size; i.e., B_0 and B_j have equal number of bits, then we have been able to reduce a multiplication of n X n into two multiplications of n/2 X n/2, plus an addition. In fact, one can further partition both B_0 and B_j into smaller parts in size and, therefore, reduce the size of multiplications. The price to pay is clearly the number of multiplications in parallel plus the increase in number of additions of partial products. The ultimate partitioning of the operand (B) is, naturally, bit size which then lead this method close to Booths Algorithm [1].

V. Design of 32 X 32 Bit Multiplier

For the design of a fast 32 X 32 bit multiplier/adder (for simplicity, we only consider the multiplier, in this discussion) we first partition one operand (multiplier) into four equal bit-size parts; each part being an 8-bit number. In the second stage we simultaneously multiply each 8-bit number by the 32-bit number using the growing technique, discussed above. At last we apply proper shifts on the partial products, so obtained, and then add them to get the final product term.

To be more specific, consider two 32-bit numbers A and B. We first partition B into $B = <B_1><B_2><B_3><B_4>$, where each B_i, for i = 1,2,3, and 4 represents an 8-bit number. We also partition A into two parts as; $A = <A_0><A_1>$, where A_0 represents the 24 MSBs, and A_1 represents the low 8-bit portion of A. Now, if we apply the growing algorithm and simultaneously multiply A by B_1, B_2, B_3, and B_4 we get:

$$M_1 = A * B_1, \quad M_2 = A * B_2,$$
$$M_3 = A * B_3, \text{ and } M_4 = A * B_4 \qquad (7)$$

As it was developed earlier (Eq.(6)), the final product term is obtained as:

$$M = <M_1><0_8><0_8><0_8> + <M_2><0_8><0_8> + <M_3><0_8> + <M_4> \qquad (8)$$

Where, 0_8 means eight zero bits.

For a graphic representation let us assume an 8-bit a byte, and represent it by a unit line. Evidently each of the partial products M_1, M_2, M_3, and M_4 are, in general, five bytes long. Figure 5(a) shows the graphic representation of all four partial products, shifted due to the extended zeros, as given in Eq. (8).

In a regular procedure, in order to obtain the sum M of the partial products, given by Eq.(8), we basically need to add four 64-bit numbers, or, in effect, to perform three double 64-bit additions. To reduce both the hardware and the computational time we propose a new scheme for addition, which reduces the operation (EQ. (8)) into three double 32-bit addition, each followed by an 8 or 16-bit carry propagation (incrementing). This technique, called *Add and Carry extend* method, is well demonstrated in Figs. 5 (b), (c), and (d). As shown in Fig. 5(b), two 32-bit adders (or one adder used sequentially) are implemented to do the additions depicted by Q_1 and Q_2. The carry out from Q_2 and Q_2 are extended to P_2 and P_2,respectively, to fulfill the additions. This expansion of carry could be no action or an increment procedure. The result of this partial addition is shown in Fig. 5(c). Similar to the previous case, we perform a 32-bit addition, designated by Q_3 in the figure, and extend the carry bit to P_3 and P_4 bytes. This concludes the operation and the 8-byte (64-bit) multiplication result, M, is obtained, as shown in Fig. 5(d).

The following step by step algorithm explains the procedure for multiplying two 32-bit numbers. To simplify, the numbers are assumed to be unsigned integers.

Algorithm:

1. Given two 32-bit numbers A and B, partition A into A = $<A_0><A_1>$, and B into B = $<B_1><B_2><B_3><B_4>$, where, A_0 is a 24-bit, and A_1, B_1, B_2, B_3, and B_4 are all 8-bit numbers.

2. Apply the *Growth Algorithm* (Fig. 4) to get the partial products M_1, M_2, M_3, and M_4, as given in Eq. (7).

3. Use 32-bit adder and incrementer and apply the *Add and carry extend* method to obtained the final product term.

Figure 5 shows an overall block representation of the algorithm for 32 X 32 bit multiplication.

VI. Conclusion

The proposed algorithm is implemented for the design of a multiplier/adder with variable size operands up to 32 X 32-bits. The *dynamic configurability*, discussed earlier, allows flexibility in the computational time as well as the accuracy of the multiplication. To manage such a flexibility, a timer (clock pulse counter) unit is set by the instruction to cut off and terminate the multiplication/addition operation for the specified computational application.

VII. References:

[1] A. D. Booth,"A signed binary multiplication technique," Quart. J. Mech. Appl. Math., vol. 4, part 2, 1951.

[2] R. Hashemian and C. P. Chen,"A NEW PARALLEL TECHNIQUE FOR DESIGN OF DECREMENT/INCREMENT AND TWO'S COMPLEMENT CIRCUITS," IEEE, Proc. 34th Midwest Symposium on Circuits and Systems, Monterey, CA, May 1991.

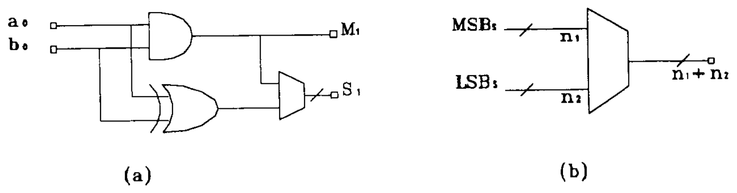

Fig. 1 - (a) Multiplication/summation of single bit operands, and
(b) bus expander/shifter.

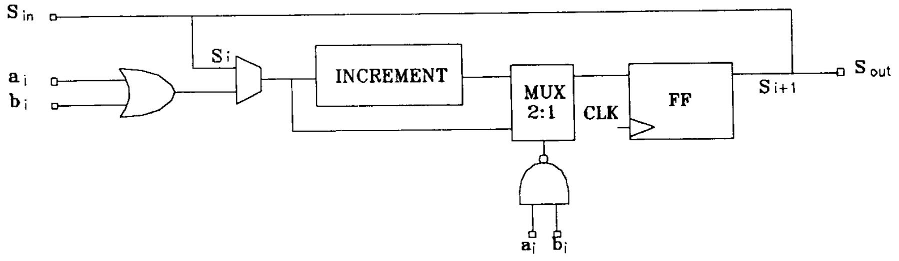

Fig. 2 - Summation using growing technique.

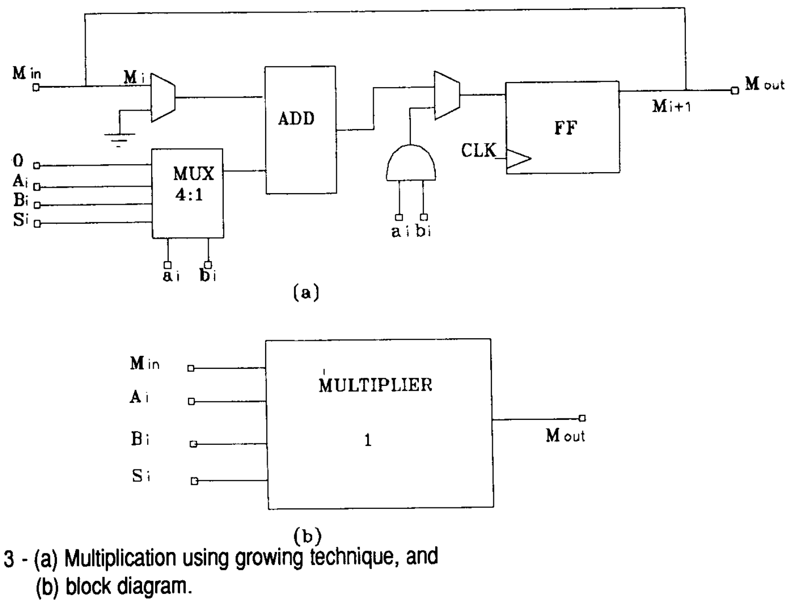

Fig. 3 - (a) Multiplication using growing technique, and
(b) block diagram.

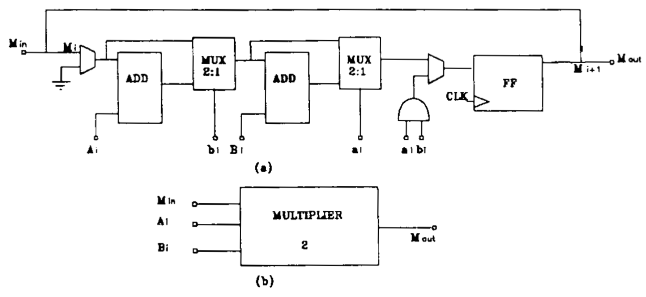

Fig. 4 - (a) Multiplication using growing technique, and (b) block diagram.

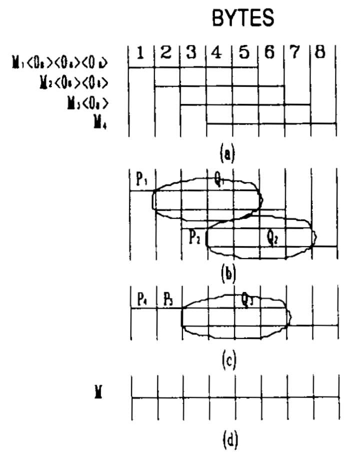

Fig. 5 - Add and carry extend method diagram.

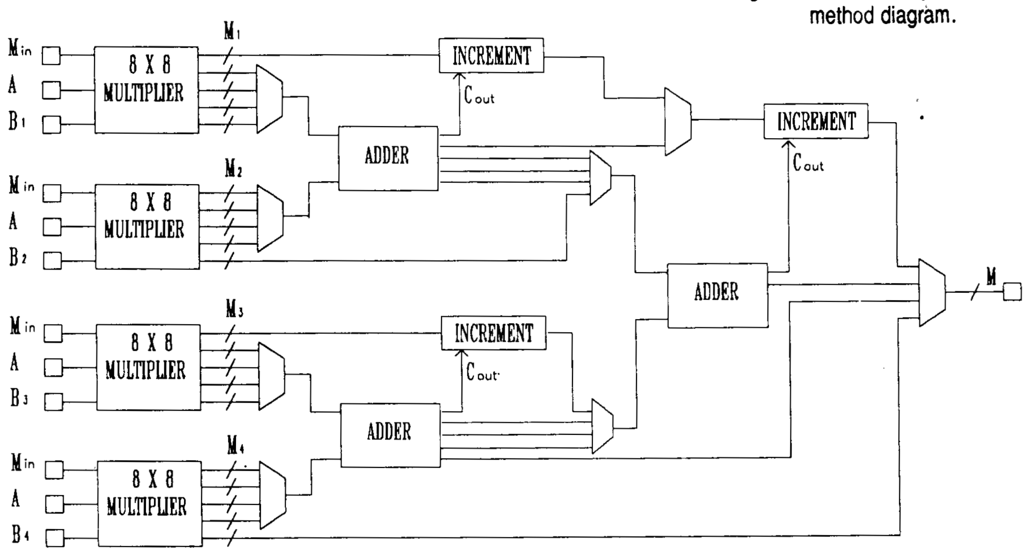

Fig. 6 - Parallel 32 X 32 bit multiplier using growing technique.