Transactions Briefs

Modified Booth Algorithm for High Radix Fixed-Point Multiplication

Philip E. Madrid, Brian Millar, and Earl E. Swartzlander, Jr.

Abstract— **It is shown that when the standard Booth multiplication algorithm is extended to higher radix (> 2), fixed-point multiplication, incorrect results are produced for some word sizes. A rule which modifies the algorithm to correct this problem is presented. The modification is defined for multipliers of any size, with any power of two radix.**

I. Introduction

As the need for faster computing increases, the need for exploring ways to both quickly and accurately perform multiplication also increases. In 1951, Andrew Booth presented an algorithm to multiply two signed, two's complement numbers without the need for correction cycles [1]. In essence, Booth was implementing a radix 2 recoding of the multiplier number in a redundant number system. Since then, others have sought to extend Booth's algorithm to higher radices to take advantage of the greater redundancy, thereby improving the speed [2]–[5]. Higher radix recoded multipliers appear to be superior from a speed perspective. In general, the higher the radix, the fewer the partial products generated, which translates into fewer cycles necessary to compute the product. For this reason, high radix recoding has found increasing use in modern high-speed arithmetic units and single chip multipliers. Booth's algorithm was developed for radix 2 recoding and can produce incorrect results for radix 4 (or greater) multiplication of fractional (or mixed integer and fractional) operands. As is shown in Section II, there are instances where correction shifts must be performed on the product produced by the Booth algorithm in order to obtain a correct product. These correction shifts were first suggested by MacSorley [2] and Rubinfield [6] and more recently by Vassiliadis [7]. This paper presents a general rule to modify the original algorithm. Section III presents the rule for arbitrary size multipliers and arbitrary radices.

II. Multiplication Algorithm Using High Radix Booth Recoding

Multiplication using the standard Booth method, which is now recognized as recoding the multiplier word in radix 2, is well understood. In addition, the theory and validity of the recoding of binary numbers in redundant number systems has recently been proved for radix 2 and higher radix recoding [5], [7]. However, the application of recoding in radices greater than two, to fixed-point multipliers, is not well understood and does not follow the standard Booth algorithm. The remainder of this section focuses on the modification of the standard rules that allow correct multiplication using higher radix recoding.

Manuscript received December 9, 1992; revised March 1, 1993.

P. E. Madrid and B. Millar are with Motorola Inc., Austin, TX 78735.

E. E. Swartzlander, Jr., is with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712.

IEEE Log Number 9209318.

TABLE I
Radix 4 Recoding Rules [2], [5]

Bit Grouping	Operation
000	$P_{i+1} = P_i/4$
001 and 010	$P_{i+1} = (P_i + B)/4$
011	$P_{i+1} = (P_i + 2B)/4$
100	$P_{i+1} = (P_i - 2B)/4$
101 and 110	$P_{i+1} = (P_i - B)/4$
111	$P_{i+1} = P_i/4$

A. Standard Radix 2 Booth Multiplication Rules

The rules for standard radix 2 Booth recoding are as follows [1]:

1) Append a zero to the right of the LSB of the multipler number, A.
2) Inspect groups of two adjacent bits of A, starting with the LSB and the appended zero.
   - If the pair is 00 or 11, then shift the partial product one bit to the right (i.e., divide the partial product by two).
   - If the pair is 01, then add the multiplicand, B, to the partial product and shift the new partial product one place to the right.
   - If the pair is 10, then subtract B from the partial product and shift the new partial product one place to the right.
3) Proceed with overlapping pairs of bits such that the MSB of a pair becomes the LSB of the next pair. In this manner, one bit of the multiplier number, A is eliminated in each pass through the algorithm.
4) When the last pair of bits is examined, the partial product is updated following the rules in #2 except that no shift is performed.

B. Radix 4 Recoding and Multiplication

Recoding the multiplier in a higher radix is a powerful method of speeding up the standard Booth multiplication algorithm since greater numbers of bits are inspected and "eliminated" during each cycle, effectively reducing the total number of cycles necessary to obtain the product. For instance, the number of bits inspected, n, in radix r is given by

$$n = 1 + \log_2 r. \tag{1}$$

Therefore, if the radix is 4, three bits are inspected on each cycle, and two bits are "eliminated" on each cycle. The rules in Table I show the operations to perform for all possible bit groupings in radix 4 and can be used to compute the product.

Radix 4 multiplication seems trivial in light of the Booth multiplication algorithm and the rules to implement the recoding of the multiplier. However, a simple example, shown in Fig. 1, illustrates the flaw in the algorithm in the radix 4 case, and subsequently, all higher radices.

Notice that the algorithm yields a result of 0.1111 while the correct answer is in fact 0.01111. The algorithm fails because radix 4 recodes the multiplier such that each recoded bit represents an even powered exponent. Therefore, in order to properly weight each partial

		$A = 0.101 = 5/8$			$B = 0.11 = 3/4$	
Cycle	i	a_{i+1}	a_i	a_{i-1}	OPERATION	RESULT
1	0	0	1	0	$P = (P + B)/4$	0.0011
2	2	0	1	0	$P = P + B$	0.1111
		THUS: $P = 0.1111$ (INCORRECT!)				

Note: The sign must be extended by at most $(n - 1)$ bits if the last grouping does not create an n bit group, where n is given by eq. (1)

Fig. 1. Booth radix 4 multiplication example.

TABLE II
RADIX 8 RECODING RULES [2], [5]

Bit Grouping	Operation
0000	$P_{i+1} = P_i/8$
0001 and 0010	$P_{i+1} = (P_i + B)/8$
0011 and 0100	$P_{i+1} = (P_i + 2B)/8$
0101 and 0110	$P_{i+1} = (P_i + 3B)/8$
0111	$P_{i+1} = (P_i + 4B)/8$
1000	$P_{i+1} = (P_i - 4B)/8$
1001 and 1010	$P_{i+1} = (P_i - 3B)/8$
1011 and 1100	$P_{i+1} = (P_i - 2B)/8$
1101 and 1110	$P_{i+1} = (P_i - B)/8$
1111	$P_{i+1} = P_i/8$

product, successive partial products are shifted down two place values (corresponding to a division by 4). Because of this, one can never attain a 5-bit (excluding the sign bit) result for this example since B contains 2 bits and there is one shift of 2-bit positions, resulting in a 4-bit product when 5 bits are necessary to represent the product.

This situation is identical to the decimal equivalent of the problem: $0.625 \times 0.75 = 0.46875$. The problem can be multiplied out disregarding the decimal points and then the proper placement of the point is determined by the total number of significant digits to the right of the decimal point in the multiplier number and the multiplicand (i.e., five places). Therefore, the same reasoning can be applied to the radix 4 example. The multiplier number and multiplicand show that the product must have five significant digits to the right of the binary point, implying that a correction shift of 1 bit (divided by 2), is needed. Performing this final shift on the result $P = 0.1111$ yields $P = 0.01111$ which is indeed the correct answer. This problem does not arise in the radix 2 case since the partial product is always shifted by one bit position, thus allowing any number of significant digits to be represented in the final product.

C. Extension to Radix 8 and Higher Radices

This problem is also evident in higher radix multiplication and the solution is, therefore, extended to these radices. To amplify understanding of the problem, an example using radix 8 is shown. Radix 8 multiplication groups 4 bits of the multiplier number at a time so that the recoded digits have weights $2^0, 2^{-3}, 2^{-6}$, etc., and whose values are $\in \{\bar{4}, \bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3, 4\}$. The rules showing the operation to perform for a particular bit grouping are shown in Table II.

Before presenting the example, an overflow problem with higher radix recoded multiplications should be discussed. In the standard Booth multiplication algorithm the appropriate operations are performed on the partial product and then it is shifted prior to examining the next set of bits (e.g., $P_{i+1} = (P_i + B)/2$. Overflow is always ignored in these cases with no loss of accuracy. However, in radix 8 multiplication, multiplicand multiples of ± 3 and ± 4 may be involved which could cause overflow that cannot be ignored. For example, the operation $P_{i+1} = (P_i + 3B)/8$ may no longer yield an accurate result. This problem is easily overcome by using the associative property. The partial product P, and the multiplicand B, should be shifted by three positions (divide by 8), prior to performing the multiplication by 3 and the addition [e.g., $P_{i+1} = (P_i/8) + 3(B/8)$]. Using this procedure ensures that information is not lost due to overflow.

Since $A = -1/4$ and $B = 7/8$, the product $P = 1.11001$. Clearly, the result from Fig. 2 is incorrect and furthermore it is easy to see that if P is shifted by two places to the right (divide by 4), $P = 1.11001$ which is correct. The reason for this is identical to the radix 4 example. Although the product should contain five significant digits, it only contains three, thus the final divide by four correction cycle must be employed. Now, look at the same example but only inspect the multiplicand B, instead of A:

These examples show that when the operands are of different length, the choice of the multiplier number may affect whether or not the algorithm produces a correct result. In Fig. 3, the choice of the multiplier number resulted in a correct product due to the number of significant digits in A.

D. Modification Rule to Booth's Radix 2 Algorithm

In general, one would employ a correction cycle when the number of significant digits in the product at the completion of the algorithm is not equal to the total number of significant digits in the multiplier number and multiplicand. The correction cycle is a right shift of the product S bit positions, where,

$$\boldsymbol{S} = [\boldsymbol{SD} \text{ of } (\boldsymbol{A}) + \boldsymbol{SD} \text{ of } (\boldsymbol{B}) - \boldsymbol{SD} \text{ of } (\boldsymbol{P})]. \quad (2)$$

The number of significant digits ($\boldsymbol{SD}$) in a fractional number is defined to be the number of digits to the right of the binary point and before the infinite string of zeros. The sign bit is not considered to be a significant digit. This correction cycle, which may or may not be required, and proper use of the associative property for high radix multiplication, are the modifications that must be added to Booth's standard algorithm to ensure the correctness of the product when multiplying two fractional numbers.

III. APPLICATION OF THE MODIFICATION RULE TO IMPLEMENT HIGH RADIX MULTIPLIERS

The form of the rule presented in Section II-D is adequate to understand the basic problem inherent in the Booth algorithm in higher radix multiplication. However, it does not shed much light on how the modification is used in multipliers with operand sizes of 8, 16, 32, 64 bits, etc. In other words, all the examples thus far presented show how the multiplication would be done on paper, not how it would be done in a hardware implementation which would always have fixed length words and the possibility of many trailing zeros in the register where the words are stored. For example, if one was implementing a 32-bit multiplier utilizing radix 8-bit recoding, it should be known beforehand the correction cycle required without having to dynamically determine the number of significant digits required and produced. As is shown, the correction cycles needed follow a regular pattern for each recoding and multiplier size. Note:

			$A = 0.11 = -1/4$			$B = 0.111 = 7/8$	
Cycle	i	a_{i+2}	a_{i+1}	a_i	a_{i-1}	OPERATION	RESULT
1	0	1	1	1	0	$P = P - B$	1.001
			THUS: $P = 1.001$ (INCORRECT!)				

Fig. 2. Booth radix 8 multiplication example (incorrect result).

			$A = 0.111 = -7/8$			$B = 1.11 = -1/4$	
Cycle	i	a_{i+2}	a_{i+1}	a_i	a_{i-1}	OPERATION	RESULT
1	0	1	1	1	0	$P = P - B$	1.001
2	3	0	0	0	1		
			THUS: $P = 1.11001$ (CORRECT!)				

Fig. 3. Booth radix 8 multiplication example (correct result).

TABLE III
RADIX 8 CORRECTION SHIFTS

Multiplier Length	Correction Shift
$3j$	Divide by Four
$3j + 1$	None required
$3j + 2$	Divide by Two
where $j \in I$	

TABLE IV
RADIX 16 CORRECTION SHIFTS

Multiplier Length	Correction Shift
$4j$	Divide by Eight
$4j + 1$	None required
$4j + 2$	Divide by Two
$4j + 3$	Divide by Four
where $j \in I$	

The length of the multiplier, L, is defined as the number of bits used to represent the multiplier word, not including the zero appended in step 1 of Booth's algorithm. L is the length of the hardware register, and is not dependent on specific data patterns.

A. Correction Cycles for Radix 4 Multipliers

In Section II-B it was shown that particular examples of radix 4 multiplication require a final correction shift of 1 bit position to the right, or divide by 2. However, for as many examples where the correction is needed, one can find examples where it is not. It can be shown that for the radix 4 case, a divide-by-2 correction cycle is needed when the length of the multiplier is even, and a correction cycle is not needed when the length of the multiplier is odd. Therefore, correction cycles are required for multipliers of typical sizes (i.e., 8, 16, 32, 64 bits, etc.).

B. Correction Cycles for Radix 8 Multipliers

Radix 8 multiplication requires correction shifts more frequently than radix 4 since the shifts are three bit positions and thus only multipliers whose magnitude lengths (the length of the multiplier not including the sign bit) are evenly divisible by three, yield correct results without final shifting. Therefore, only 33% of all length radix 8 multipliers do not require correction shifts (versus 50% for radix 4).

This pattern shows that all multipliers where the length of the multiplier (including the sign bit) is evenly divisible by three, require a correction shift of 2-bit positions (divide by 4). Similarly, a correction right shift of 1-bit position is always needed for multiplier lengths of 2, 5, 8, 11, $\cdots$ bits. In addition, it can be shown that a correction right shift is never required for multiplier lengths of 1, 4, 7, 10 $\cdots$ bits. In summary, see Table III for a radix 8 fixed-point multiplier.

Like the radix 4 case, there are definite possibilities of correction shifts in an implemented radix 8 multiplier. For instance, the patterns given above show that if a 32-bit radix 8 Booth multiplier was constructed, the multiplier would need to provide a final right shift of one bit position in order to produce a correct product. The pattern also indicates that a 64-bit radix 8 multiplier never requires a final right shift.

C. Correction Cycles for Radix 16 Multipliers

For a radix 16 multiplier, the shift length is four bit positions. Therefore, only 25% of all length radix 16 multipliers do not require correction shifts. Correction shifts are not needed when the length of the multiplier (including the sign bit) is 1, 5, 9, 13, $\cdots$ bits. A final correction right shift of one bit position is needed for multiplier lengths of 2, 6, 10, 14, $\cdots$ bits. Similarly, a final correction right shift of two bit positions is needed for multiplier lengths of 3, 7, 11, 15, $\cdots$ bits. Finally, a correction right shift of three bit positions is needed for multiplier lengths evenly divisible by four. Of course, for typical sizes of multipliers such as 8, 16, 32, 64, etc., correction shifts are needed for radix 16 recoding. In summary, see Table IV for a radix 16 fixed-point multiplier.

D. Correction Cycle Equation

From the analysis presented in this section, a general equation that accounts for all binary compatible radices and any length multiplier can be derived.

If the radix of the multiplier is represented by r, then $Y = \log_2 r$ is the standard shift length between cycles of the multiplier. Then, if the length of the multiplier (including the sign bit), is represented by L, the number of correction right shifts that must be performed on the product produced by the standard radix 2 Booth multiplication algorithm, when multiplying two signed two's complement fractions, to generate correct results is given by

$$\text{Number of correction shifts} = (L - 1) \bmod Y. \tag{3}$$

For example, a 32-bit radix 8 multiplier has $Y = \log_2 8 = 3$ and $L = 32$. Substituting in (3) above yields, $31 \bmod 3 = 1$ right shift (divide by two) necessary to correct the Booth generated product. This agrees with the results in Section III-B.

IV. CONCLUSION

This paper shows that Booth's standard radix 2 algorithm for the multiplication of two signed two's complement fractions can produce incorrect results for some operand sizes when it is extended to higher

radix multiplication. Examples are presented which demonstrate the error when multiplying using radix 4 and 8 recoding schemes. Rules for modifying the original algorithm which produces correct results are presented. Finally, an equation which allows the easy determination of the number of correction shifts required at the end of the Booth process to yield a correct result, is presented as a function of the multiplier word size and radix.

REFERENCES

[1] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, pp. 236–240, 1951. (Reprinted in [8, pp. 100–104.])

[2] O. L. MacSorley, "High speed arithmetic in binary computers," *Proc. IRE*, vol. 49, pp. 67–91, 1961. (Reprinted in [8, pp. 14–38]

[3] J. E. Robertson, "The correspondence between methods of digital division and multiplier recoding procedures," Dep. of Computer Sci., Univ. of Illinois, Urbana, Rep. 252, Dec. 1967.

[4] J. O. Penhollow, "A study of arithmetic recoding with applications to multiplication and division," Dep. of Computer Sci., Univ. of Illinois, Urbana, Rep. 128, Sept. 1962.

[5] H. Sam and A. Gupta, "A generalized multibit recoding of two's complement binary numbers and its proof with applications in multiplier implementations," *IEEE Trans. Computers*, vol. 39, pp. 1006–1015, 1990.

[6] L. P. Rubinfield, "A proof of the modified Booth's algorithm for multiplication," *IEEE Trans. Computers*, vol. C-24, pp. 1014–1015, 1975.

[7] S. Vassiliadis, E. M. Schwartz, and D. J. Hanrahan, "A general proof for overlapped multiple-bit scanning multiplications," *IEEE Trans. Computers*, vol. 38, pp. 172–183, 1989.

[8] E. E. Swartzlander, Jr., Ed., *Computer Arithmetic*, vol. 1, Los Alamitos, CA: IEEE Computer Society Press, 1990.

The Selective Extra Stage Butterfly

S. Konstantinidou

***Abstract*—Multistage interconnection networks (MIN's) have been used extensively as communication networks in parallel machines due to their high bandwidth, low diameter and constant degree switches. The fault-tolerance of multistage networks can be improved by simply adding extra stages to the network. In this paper we suggest a novel method of attaching the extra stages in MIN's such that they are used in the absence of faults but not necessarily by all messages. Messages can *adaptively* select to take the shortest path towards their destination or use one of the longer paths going through the extra stages. We present performance results of our method, obtained through simulation, under various traffic loads both in the presence and absence of faults.**

I. INTRODUCTION

In massively parallel computing the interconnection network, the means through which processors communicate and synchronize, remains one of the most critical components. The network is required to provide high throughput, low latency, reliable communication; these requirements become increasingly important as the number of processors in a parallel computer increases and the cycle time of the processors decreases.

The class of multistage interconnection networks, or *MIN's*, have long been suggested and studied as the communication network for parallel computers. Their main advantages are their high bandwidth, $O(N)$, low diameter, $O(\log N)$, and constant degree switches. Multistage networks have been used in commercial machines, such as the BBN, CM 5 and Meiko, and research projects, such as the IBM RP3 [13], the NYU Ultracomputer [4] and currently the MIT Transit project [6].

A well-known problem with multistage interconnection networks is that in an N node MIN with degree d switches and $\log N/\log d$ stages, there is exactly one path between each pair of input and output nodes. This results in both poor performance and lack of fault-tolerance as the failure of a switch along this path disconnects at least one pair of input and output nodes. Many solutions to provide multiple paths between any source and destination nodes in MIN's have been suggested, invariably requiring some increase in hardware. Typical requirements are extra stages, switches of higher degree and increased wiring complexity between or within stages.

The addition of extra stages is one of the simplest methods to provide fault-tolerance in a multistage interconnection network. The extra stages can be attached to the network through multiplexers and demultiplexers as suggested in [2]. In this case the extra stages are *enabled* in the presence of faults; thus the extra hardware provided for fault-tolerance remains unutilized when the network operates in the absence of faults.

An alternate and equally simple method with respect to switch logic complexity and wiring complexity, is a *multipath* Omega network [1] or *extended* Omega network [12]. In this case the extra stages are always attached to the network. Routing is controlled by the *headers* of messages. The destination address determines n bits of the header and r additional bits select one of the multiple paths. As opposed to the previous method, the extra stages in multipath Omega networks are being utilized even in the absence of faults. Furthermore, if $\log N/\log d$ extra stages are used, a two-phase randomized routing scheme can be implemented where messages are first sent to a random intermediate destination and from there to their final destination [19]. This combination of network and routing scheme can route any permutation of N messages in $O(\log N)$ expected time [19]. The case of continuous routing and the use of r extra stages with $1 \leq r \leq \log_d N$ is examined in [12].

The disadvantage of this method is that it increases the length of the path of every message in the network; messages have to be routed through $n + r$ rather than n stages. Thus the added fault-tolerance can result in increased message latency when the network operates in the absence of faults.

In this paper we suggest a new method of attaching the extra stages in a multistage interconnection network that, in conjunction with a novel, adaptive routing scheme, addresses the problems of the methods previously considered. In our solution the extra stages are permanently connected to the network along with the means to bypass them. Messages can *adaptively* select to take the shortest path towards their destination or use one of the longer paths, going through the extra stages. This results in significant performance improvement over networks of similar hardware requirements, as shown by our simulations. Thus, our network effectively utilizes the extra hardware required to provide fault-tolerance in order to improve performance in the absence of faults.

Manuscript received December 10, 1992; revised February 1, 1993 and February 15, 1993.

The author is with IBM Almaden Research Center, San Jose, CA 95120.

IEEE Log Number 9209319.