$M \times N$ Booth Encoded Multiplier Generator Using Optimized Wallace Trees

Jalil Fadavi-Ardekani, *Senior Member, IEEE*

***Abstract*—In this paper the architecture and the design method of an M-bit by N-bit Booth encoded parallel multiplier generator are discussed. A new algorithm for reducing the delay inside the branches of the Wallace tree section is explained. The final stage of adding two $N + M - 1$-bit numbers is done by an optimal carry select adder stage. The algorithm for optimal partitioning of the $N + M - 1$-bit adder is also presented.**

I. Introduction

FAST multipliers are essential parts of digital signal processing systems. General purpose DSP's can afford to have customized high performance multipliers (16 by 16 < 25 ns in 0.9 mm CMOS). Applications specific DSP's (ASDSP), on the other hand, implement algorithms as direct realization of their flow graphs, thus having several multipliers, possibly each one with different specifications. It is not economical to have custom layout for every conceivable multiplier. Besides, experience shows that in ASDSP systems the desired parts are not actually multipliers; rather more often they are composition of multipliers, and adders, and pipeline registers. Therefore, manipulation of prefabricated custom layouts for accommodating any such requirements turns out to be another "custom" task.

There are other methods to quickly build various multipliers and at the same time keep some margin of flexibility for further alterations. One method is to use layout generators [2]. One major task in this approach is the extensive amount of "qualification and regression testing" needed to accept each incarnation of the generator for a real world application. Our solution is a multiplier netlist generator. This tool, written in AWK [1], produces a description of a high performance production ready multiplier in terms of the circuits available in a standard cell catalog. One other advantage of this approach is the flexibility of the final product. Although our generator program produces a hierarchical netlist of a multiplier, it is very practical to textually modify the logic to obtain the desired function (e.g., truncate, merge two multipliers, etc.), and the product may be manufactured as before. Fig. 1 shows comparisons of two different approaches, standard cell netlist generator, and layout generator. The netlist generator has been chosen by us because of its architectural flexibility.

The multiplier architecture chosen for the implementation of this generator is modified Booth recoding [4], [6], Wallace tree summation of partial products [10], and final addition of the result by an optimal carry select adder [3]. Booth recoding with Wallace tree adders have been often used in fast multipliers [5]. However, in many other instances, the merit of using Wallace tree adders in multipliers has been traded off with the structural regularity obtained from using carry save adders [7], or using a sub-"Wallace tree" structure [9]. An important contribution of the present work is the automatic formation of the Wallace tree adders, and the heuristic algorithm for reducing the delays inside the Wallace trees. In this paper, the architecture of the multipliers designed by this generator program will be discussed. Then the simulation and testing of these multipliers will be presented.

	Standard cell netlist generator	Layout generator
Architectural flexibility	♠♠♠	♠
Speed	♠♠ (60 ns for 16×16 in 1.25 μCMOS)	♠♠♠ (40 ns for 16×16 in 1.25 μCMOS)
Area	♠ (3mm^2 for 16×16 in 1.25 μCMOS)	♠♠♠ (1mm^2 for 16×16 in 1.25 μCMOS)
Technology independence	♠♠♠	♠♠

♠ is a unit of goodness. Wherever actual data has not been available, subjective rating by various designers has been used.

Fig. 1. Comparison of netlist, and layout generators.

II. Multiplier Architecture

The multiplier is composed of three blocks: the modified Booth encoder and multiplicand selector block for formation of the partial products; next is the Wallace tree section, which adds all of the partial products simultaneously to eventually produce two numbers; the third block is the carry select adder section which adds the two numbers, obtained from the Wallace tree section, as fast as possible. The block diagram of a 12 × 12 parallel multiplier is shown in Fig. 3.

A. Modified Booth recoding

Modified Booth 3-bit recoding was proposed by MacSorley [6]. By applying this method of recoding to an M-bit two's complement binary number:

$$X\left(= -2^{M-1}x_{M-1} + \sum_{i=0}^{M-2} x_i 2^i\right)$$

Manuscript received December 8, 1992; revised February 19, 1993 and March 18, 1993.

The author is with AT&T Bell Laboratories, Allentown, PA 18103.

IEEE Log Number 9209314.

$x(2i+1)$	$x(2i)$	$x(2i-1)$	d_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	−2
1	0	1	−1
1	1	0	−1
1	1	1	0

Fig. 2. Modified Booth encoder truth table.

an equivalent base 4 redundant sign digit representation is obtained:

$$X = \sum_{i=0}^{(M/2)-1} d_i 4^i.$$

The digits d_i's are chosen from the set $\pm 2, \pm 1, 0$, according to the rules in Fig. 2. At each step $i, (i = 0, 1, ., M/2 - 1)$, 3 bits of X, $x_{2i+1}x_{2i}x_{2i-1}$, are examined and the table in Fig. 2 is referred to for selecting d_i. X is always appended on the right with zero $(x_{-1} = 0)$; and M is always even (X is sign extended if needed). For proof of correctness see [8].

In the M-bit by N-bit multiplication, the M-bit multiplier number X is "modified Booth" recoded. The product XY is then obtained by adding $M/2$ partial products, $P_i = d_iY, i = 0, \cdots, M/2$, that can be calculated easily by shifting and/or complementing the multiplicand Y:

$$XY = \sum_{i=0}^{(M/2)-1} d_i 4^i Y$$
$$= \sum_{i=0}^{(M/2)-1} P_i 4^i.$$

B. Sign Extension Prevention

When adding the $M/2$ partial products, P_i's, the P_{i+1}th partial product is placed two bits to the left of P_ith partial product. However, all P_i's should be sign extended to the MN binary position. This sign extension can be avoided if we accumulate P_i's, one at a time. However, in the case of Wallace tree adders, we add all of the partial products in parallel. Thus such a sign extension becomes very costly. In order to prevent that, we use the following technique. We assume that all of the partial products are negative. In this case, the sum of all sign extensions can be precalculated:

$$\text{signs} = \sum_{i=0}^{(M/2)-1} \left((-1)2^N\right)4^i$$
$$= 2^N(-1)\left(\frac{2^M - 1}{3}\right).$$

The above relation can be interpreted to mean that a fixed number, $(-1)\left[(2^M - 1)/3\right]$, should be added to the (unextended) partial products, starting from the Nth binary position leftwards. This number, if expressed in binary is equal to $10101 \cdots 01011$, where there are $M/2 - 1$ zeros. If it turns out that a partial product, $P_i = d_iY$ is indeed positive, we simply replace its sign bit with a one to undo the effect of our earlier assumption about the negativeness of the partial product P_i. Example 1 (see below) illustrates the method of multiplication using the modified Booth recoding, and the sign extension prevention.

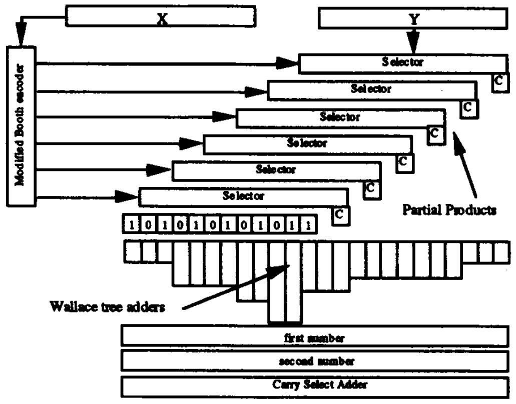

Fig. 3. Block diagram of the parallel multiplier.

In Fig. 3, the modified Booth recording circuits generate six partial products each of length 13 bits. The fixed number is added to the 13th (Nth, counting from 0) column and leftwards. As was mentioned earlier, prevention of the sign

Example 1: $X = 22 (= 010110)) \times Y = -3 (= 1101)$:

```
0 |                                 (1) 0   1   0   1
0 | -2Y                                             (1)
1 |       |             (0) 1   0   1   0
1    2Y   |                             (0)
0 |       |     (0) 1   1   0   1
1 |  Y                      (0)
0 |          1   0   1   0   1   1

             1   1   1   0   1   1   1   1   1   0  =  (-66)
```

extension is particularly important in multiplier architectures based on Wallace tree adders.

In a carry–save adder (CSA) architecture, one adds the bits in each column of the first three partial products independently (by full adders). From there on, the resulting arrays of sum and carry bits and the next partial product are added by another array of full adders. This continues until all of the partial products are condensed into one array of sum bits and one array of carry bits. A fast adder (carry select or look-ahead) is finally used to produce the final answer. The advantage of this method is the the possibility of regular custom layout. The disadvantage of the CSA method is the amount of delay of producing the final answer. Because, the critical path is equivalent to first traversing all CSA arrays and then going through the final fast adder. In contrast, in a Wallace tree architecture, all the bits of all of the partial products in each column are added together in parallel and independent of other columns. Then, a fast adder is used to produce the final result similar to the CSA method. The advantage of Wallace tree architecture is speed. Because, the addition of the partial products is now $O(\log)$. This advantage becomes more pronounced for multipliers of bigger than 16 bits. However, building a regular layout becomes a challenge in this case. It can be seen (from Fig. 3) that changing the Wallace tree multiplier into a multiplier/accumulator is quite simple. One needs to include the incoming data for accumulation in the set of partial products at the input of the Wallace tree section; and the Wallace tree will treat it as another partial product. Also, merging multiple parallel multipliers and adders is as simple. One only needs to include all partial product bits in the same column in the inputs to the Wallace tree adders. This is a property that we have greatly taken advantage of in designing high speed data paths.

III. Generator Algorithms

In this section we explain the method of generating the logic description of an M-bit by N-bit Wallace tree multiplier. However, we will first provide some techniques for estimating the speed of the multiplier, in terms of the delays of full adders used. In order to estimate the speed of the Booth encoded Wallace tree section of the multiplier, we need to identify the column (C) with the most number of bits in it (n_C) including the carry bits from the previous columns. The worst delay in generating the results of the Wallace tree adders is then proportional to $DW = \lceil \log_3(n_C) \rceil$. The worst case delay of the final result for an M-bit by N-bit multiplication, excluding the fixed overheads, will be proportional to $DW + M + N - C$. This can be contrasted to a rough proportionality constant of $N + M/2$ for the CSA architectures.

A. Calculating the Number of Participating Bits in Each Column

Define:

N_{pp_i} number of input bits to the Wallace tree adders in column i from the partial products, and the sign-extension prevention number. Then,

$$\begin{aligned} N_{pp_i} = &\sum_{k=0}^{(M/2)-1} [u(i-2k) - u(i-2k-N-1)] \\ &+ [(i+1) \bmod (2)]u(M-2-i) \\ &+ [(i-N) \bmod (2)]u(i-N) \\ &+ d(i-N), \qquad i = 0, N+M-2. \end{aligned}$$

Here, the functions $u(x)$ is the step function and $d(x)$ is the impulse function of (x).

$N_{cr_i} \triangleq$ number of carries sent from column i to column $i+1$

$N_{tl_i} \triangleq$ total number of bits to be added in column i.

Therefore, $N_{tl_i} = N_{pp_i} + N_{cr_{i-1}}$.

In each column we will set aside the last carry generated in that column to be considered as an input to the carry select adder stage. We will label this carry as non-summable (NS). The total number of bits to be summed up in each column and thus the number of carries generated and transmitted in each column are calculated as follows.

```
$N_{cr_{-1}} = 0$;     /* no carry from column -1 */
for($i = 0; i \leq 2N + M - 2; i++$) {
total = $N_{pp_i} + N_{cr_{i-1}}$;
$N_{tl_i}$ = total;
$N_{cr_i}$ = 0;
  /* no carry is generated first */
do {
    $N_{cr_i} = N_{cr_i} + \lfloor \frac{\text{total}}{3} \rfloor$
    remain = total $\bmod(3) + \lfloor \frac{\text{total}}{3} \rfloor$
    switch (remain) {
    case '1':
           break; /* $N_{cr_i}$ is correct */
    case '2':
             $N_{cr_i}++$; break;
    case '3':
             $N_{cr_i}++$; break;
    default:
             total = remain ; break;
    } while( total > 3);
    if ($N_{cr_i} \geq 1$) {
           $N_{cr_i} = N_{cr_i-1}$;
    } /* take out the NS carry */
}
```

Therefore, by finding the biggest N_{tl_i}, the speed of the multiplier can be estimated as explained above.

B. Wallace Tree Adders Optimization

The Wallace tree section of the generator accepts as input variables the list of partial products for each column, and the delay associated with each bit. At the present time, rather than keeping track of both the fall- and the rise-time delays of the bits, only an average delay is calculated, and saved at any given step. It is assumed that all of the partial product bits arrive at the Wallace tree section simultaneously. This is a fair assumption, because of the fact that all of the Booth encoding

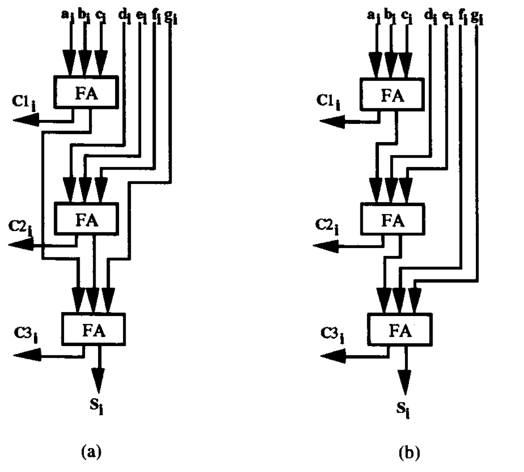

Fig. 4. Two examples of Wallace tree addition of 7 bits.

sections work in parallel. Next we start adding partial product bits in each column in Wallace tree fashion. Fig. 4 shows two ways of adding 7 bits of equal positional values, bits a_i through g_i.

The main difference between the two methods of adding these 7 bits is the time delay in calculating the sum bit S_i. This in turn is related to the amount of delay of each incoming bit, and the delay produced by each full adder. The delays of the sum bit, and the carry bit of the full and half adders are provided to the generator program as user parameters. Therefore, in the Wallace tree section, the heuristic algorithm groups the partial product bits, the resulting sum bits, and the carry over bits of each column for being added in full or half adders. The goal is to result in the minimum delay for the final sum bit. To achieve this, initially a delay is associated with every bit in a column. As we produce sum bits which have to be added to the same column (such as shown in Fig. 4), the time delay needed for their computation is also calculated and saved. Also, the time delay for the carry bits are calculated and saved with them; so that when we are in the next column we will use this information to obtain optimal grouping of bits in that column. Whenever 3 bits are added together via a full adder, they are removed from the list of bits waiting to be added, and the resulting sum bit is appended to the list. Then, the list is sorted in ascending order with respect to the delays of arrival of the bits. The algorithm stops stacking adders in a column when no more bits are left. At that time, the sum bit as well as the last carry bit are saved for addition in the carry select adder stage. Below, the algorithm is explained in detail. The variables d_s and d_c are delays in calculating the final sum and the carry bits, respectively. Here, we assume that all of the previously defined variables: N_{cr_i}, N_{pp_i}, and N_{tl_i} are known. In addition, variable $p(i,j)$ means the delay of (Wallace tree adder) input in column i, and jth position. Also, W_{si} and W_{ci} are the delays of calculating the sum and carry bits that are input to the final carry select adder section in column i, respectively. Initially, N_{cr-1}, and all $p(i,j) = 0$.

```
remain_0 = $N_{pp0} + N_{cr-1}$;
for($i = 0; i \le N + M - 2; i++$){
    remain_{i+1} = $N_{pp_{i+1}}$;
    while(remain_i > 3){
        sort $p(i,j)$;      /* sort bits according to their delays. */
        if($p(i,0) == p(i,2) || p(i,0) \neq p(i,1)$) {
            combine the top three bits by a full adder;
            $delay_s = \max\{p(i,0), p(i,1), p(i,2)\} + ds$;      /* $d_s$ is the delay in calculating
                        the sum bit.*/
            $delay_c = \max\{p(i,0), p(i,1), p(i,2)\} + dc$; /* $d_c$ is the delay in calculating
                        the carry bit.*/
            $p(i,0)$ = delays;
            for($j = 3; j < remain_i; j++$)$p(i,j-2) = p(i,j)$;      /* shift two places up */
            remain_i = remain_i - 2;
    }
    else {
            combine the top two bits by a half adder;
            $delay_s = p(i,1) + ds$;
            $delay_c = p(i,1) + dc$;
            $p(i,0) = delay_s$;
            for($j = 2; j < remain_i; j++$){$p(i,j-1) = p(i,j)$}; /* shift one place up */
            remain_i = remain_i - 1;
    }
    $p(i+1, ++remain_{i+1}) = delay_c$;
}
switch (remain_i) {
        case '3':
                combine the three remaining bits by a full adder;
        case '2':
                combine the three remaining bits by a half adder;
        case '1':
                this bit is directly input to the carry select adder;
        all cases:
                label the sum and carry bits as input to the carry select adder;
                calculate $W_{si}$ and $W_{ci+1}$;
    }
}
```

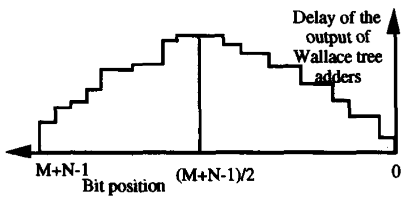

Fig. 5. A typical delay distribution of the output of Wallace tree section.

C. Carry Select Adder

Adding two $N + M - 1$ bit numbers is accomplished in a carry select adder stage with optimal partitioning of the $N + M - 1$ bits into several parallel adder blocks. The time delay for calculating the sum of two numbers using a carry select adder depends on the initial delay of arrival of the operands, and the delay introduced in each partitioned block. Because all of the partitioned adders of the carry select adder operate in parallel, all of the resulting bits should approximately fall out simultaneously. However, in the case of our multiplier, the input bits of the carry select adder come out of the Wallace tree adder; and do not arrive at the same time. A typical plot of the delay of arrival of the bits from the Wallace tree section is shown in Fig. 5.

The switch level simulator has verified the "bell" shape of the delay function. The procedure explained in the Wallace tree section, automatically provides this delay information for the carry select adder. Our goal in the carry select adder generator is therefore to find the optimal number of partitions, and number of bits per partition. It was shown in the previous section that the variables W_{ci} and W_{si} show the time of arrival of Wallace tree output bits in column i. We use the variable $W_i = \max\{W_{si}, W_{ci}\}$ as the delay in calculation of the inputs to the carry select adder in column i. The algorithm for partitioning the $N + M - 1$ bit adder is as follows.

```
$L = 1$;      /* number of partitions. */
for($j = 0; j \le N + M - 2; j++$) {
      $X_L = W_j$;   /* delay of the first bit
          in partition $L$. */
      $B_L = 1$;      /* number of bits
          in partition $L$. */
while( $XL <$ desired speed AND $j \le N+M-2$){
            $X_L = X_L + d_s$;
            $B_L++$;
      }
            $j = j + BL$;
            $L++$;
}
```

Because of the extra delay due to the combinational block in front of the multiplexers of each block of the carry select adder, the actual speed of the carry select adder will be equal to the delay of the first block, and the accumulation of delays of all of the combinational blocks. In order to alleviate this problem, we should reduce the user's requested "maximum delay" of each partition by the amount of delays of the combinational blocks existing between that partition and the partition containing the MSB bit. For exact results, a recursive algorithm should be implemented. For the time being, users can manually iterate the generator until satisfied with the result. Different requested delays will result in different partitioning of the carry select adder. For the following multipliers the carry select adder partitions for the fastest "requested speed" are shown. Here each bracket indicates the bits grouped together in one stage of the carry select adder.

For 16 × 16 multiplier:
[31:25], [24:20], [19:16], [15:13], [12:9], [8:6], [5:0]

For 12 × 9 multiplier:
[20:18], [17:15], [14:13], [12:11], [10:8], [7:6], [5:4], [3:0]

For 24 × 24 multiplier:
[47:42], [41:34], [33:29], [28:24], [23:19], [18:14], [13:9], [8:0].

IV. Using the Generator

Users can obtain a variety of multipliers by changing the sizes of the operands. When variations in the architecture due to the changes in full and half adder performance, or the required speeds are also included, the possibilities quickly rise to many thousands of multipliers. Below, a sample "specification file" created by interface section of the generator is shown.

```
# Generator: mulwalgen
NAME: mul #name of the multiplier
WIDTH:
    24 #Width of the multiplicand
```

TABLE I

Multiplier	Speed (ns)	Number of Transistors	Estimated power (mW)
12 by 9	40	4172	80
16 by 16	60	7858	100
24 by 24	90	16 252	140

```
        24 #Width of the multiplier
    IOLATCH: yes #Pipelines at the input and output
    DELAY:
        3.8 #Delay of sum bit in full adder
        2.0 #Delay of carry bit in full adder
        2.2 #Delay of sum bit in half adder
        1.1 #Delay of carry bit in half adder
    MAXDLY: 30 #Maxi. delay of multiplication**
    ** Wallace tree plus carry select adder only.
```

V. Summary of Results

Many instances of the multiplier generator have been created, and tested. The following instances have been quantified in 1.25 mm standard library under worst case, (125 C, 4.2v, slow process) for speed measurements. For power measurements, operation at 5 V and their corresponding fastest speeds have been considered. These results are obtained by testing the immediate output of the generator and no further tuning of the circuits has taken place and shown in Table I.

VI. Conclusion

The techniques for doing fast M by N multiplication using Wallace tree were presented in this paper. New heuristic algorithms for formation of the Wallace tree adders, and the subsequent carry select adder section were detailed. Several instances of this M-bit by N-bit multiplier generator were simulated, and results presented. Currently, many instances of this generator are used in customer specified designs. This program has created fast ASIC (standard cell based) multipliers that have been easily modified into fast data paths. The method can equally well be applied to other technologies such as gate arrays. Because there are only a limited types of cells used in the whole multiplier, custom designers can also enjoy the result of these algorithms.

Acknowledgment

The author would like to acknowledge the encouragements of Howard Moscovitz and Craig Garen.

References

[1] A. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*. Addison-Wesley, 1988.

[2] AT&T Miceoelectronics, "High-speed HS900C CMOS Standard-Cell Library," *Data Book*, Jan. 1992.

[3] O. J. Bedrij, "Carry-select adders," *IRE Trans. Electron. Comp.*, vol. EC-11, pp. 340–346, June 1962.

[4] A. D. Booth, "A signed binary multiplication technique", *Quart. J. Math.*, vol. IV, pt. 2, 1951.

[5] Y. Kaji *et al.*, "A 45 ns 16X16 CMOS multiplier," in *IEEE ISSCC Dig. Tech. Papers*, pp. 32–33, 1983.

[6] O. L. MacSorley, "High speed arithmetic in binary computers," *Proc. IRE*, vol. 49, Jan. 1961.

[7] Y. Oowaki *et al.*, "A sub-10-ns 16x16 multiplier using 0.6mm cmos technology," *IEEE J. Solid-State Circuits*, vol. SC-22, pp. 762–767, Oct. 1987.

[8] L. P. Rubinfield, "A proof of the modified booth's algorithm for multiplication," *IEEE Trans. Computers*, vol. 37, 1988.

[9] M. Santoro and Horowitz, M., "A pipelined 64x64 iterative array multiplier," in *IEEE ISSCC Dig. Tech. Papers*, pp. 36–37, 1988.

[10] C. S. Wallace, "A suggestion for a fast multiplier", *IEEE Trans. Electron. Comp.*, vol. EC-13, pp. 14–17, Feb. 1964.

Jalil Fadavi-Ardekani (S'78–M'80–SM'88) received his Ph.D. degree in electrical and computer engineering from the University of California, Santa Barbara, in 1980.

He joined AT&T Bell Laboratories, Allentown, PA, in 1985 in the Digital Signal Processing Department. He first worked on automatic synthesis of application specific DSP's and developed automatic digital filter layout tools, as well as tools for high speed data path netlist generation. He has published several papers in this area. Later on, he became involved in the architecture and implementation of AT&T HDTV IC's, while the activities were located in Allentown, PA. He is now with the Multimedia group, working on AT&T next generation floating point DSP and peripherals. He has contributed to two books: *Miniaturized and Integrated Filters* (New York: Wiley, 1989) and *Digital Signal Processing* (New York: Wiley, 1993).

Dr. Fadavi-Ardekani is the 1993 chairman of the IEEE DSP technical committee of the Circuits and Systems society. He is a member of Sigma Xi.