AREA-TIME EFFICIENT ARITHMETIC ELEMENTS FOR VLSI SYSTEMS

Ramautar Sharma

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Algorithms for the high speed binary arithmetic operations of addition and multiplication in a VLSI environment are analyzed for area-time efficiency. It is shown that some schemes for addition and multiplication, although good for stand-alone designs, fail to provide both area and time efficiencies simultaneously. Solutions that yield area-time efficient practical implementations of these arithmetic functions are described.

1. INTRODUCTION

The advanced VLSI capability available to system designers gives them the responsibility to evaluate and choose algorithms that yield regularity in the design and give rise to overall higher system performance. It is not sufficient to choose algorithms that maximize the speed for the individual modules of the system; rather, the system must be considered as a whole. This is analogous to finding a globally optimal solution in the presence of many local optima. For example, to design a high performance and regular data-path with its arithmetic elements connected to data buses as shown in Fig. 1, one has to consider the flow of control signals, buses passing through the elements, etc. Therefore, the floor-plan for each element may not be chosen freely; instead, it is often determined by other factors as shown in Fig. 2 where the data-buses and the control signals constrain the floor-plan of the multiplier in Fig. 1. Similar considerations hold for all other units too.

In this paper we will consider mainly the problem of choosing the appropriate algorithms for the design of multipliers. But adders, as will be shown later, often limit the speed of multiplication as well as other arithmetic and logical unit (ALU) operations, we will also discuss them. Many authors have considered the design of stand-alone multipliers [BrKu80], [PrVu81], and [AbAn80]. Similarly, the problem of adder design has been addressed by [BrKu82], [OnAt83], [OkBa85], and [NgIr85]. In these studies, the authors have assumed that all the gates have identical delay irrespective of their complexity so as to simplify the derivations of area and time bounds for the algorithms. However, the above assumptions can lead to erroneous conclusions about timing.

Here we will focus our attention on the analysis of the schemes for implementing multipliers using the modified Booth's algorithm [Hwan79]. In Section 2, we analyze and compare the area-time complexity of fixed-point multipliers and show that the final adder used to form the product is the main speed limiting element. In Section 3, we study the problem of fast addition in a VLSI environment and show that the carry-lookahead scheme for a properly chosen number of bits provides an area-time efficient practical solution. The main results of our study are summarized in Section 4.

2. MULTIPLICATION: ALGORITHM ANALYSIS AND IMPLEMENTATION

To multiply two numbers $X = \{x_{n-1}, x_{n-2}, \ldots\ldots, x_1, x_0\}$ and $Y = \{y_{m-1}, y_{m-2}, \ldots\ldots, y_1, y_0\}$ represented in 2's complement notation, using the modified Booth's algorithm, $\frac{(n+1)(m+1)}{2}$ partial products are generated [Hwan79] and then added to get an $(m+n)$-bit product. Although higher radix algorithms can be derived, they are not attractive from the point of view of layout, since they require the generation of terms that are not powers of 2 and need more logic. Therefore, we will not consider such algorithms here.

For the modified Booth's algorithm, the partial product is recursively given by (1)

$$R_i = R_{i-1} + X f_i \,, \text{ and}$$
$$f_i = y_{2i-1} + y_{2i} - 2\, y_{2i+1} \,, \qquad (1)$$

where $i = 0,1,2,\ldots\ldots,\frac{m-1}{2}$ and $R_{-1} = y_{-1} = 0$. Note that R_i is the $(n+1)$-bit partial product at the i^{th} stage of recursion. The algorithm generates the complete product in $\frac{m}{2}$ recursions with 2 bits being produced at every stage and $(n+1)$ bits generated in the final fast adder. To implement this recursive algorithm sequentially, the three least significant bits of Y are examined, the value of the function f_i is determined, and one argument for the adder is then generated; the other input to the is adder is formed by R_{i-1}. While the addition is being performed, the next three least significant bits of Y are examined and the process continues until all the bits of Y have been used. The interconnection pattern for this implementation is simple and yields a regular and area efficient layout [Shar85]. But, it is slow since it takes $\frac{m}{2}$ iterations to complete the multiplication. The multiplication time can be reduced by generating all the partial products and their sums in parallel as described in the next section.

2.1 Parallel Implementations

All the partial product bits pp_{ij}, as given by (2), can be generated simultaneously.

$$pp_{ij} = f_i \cdot x_j \,, \qquad (2)$$

where f_i and x_j are defined above and pp_{ij} are the individual bits of R_i, i.e., $R_i = \{pp_{i,n+1}, pp_{i,n}, pp_{i,n-1}, \ldots\ldots, pp_{i,1}, pp_{i,0}\}$. These partial products can be summed in many different ways: Wallace's carry-save adder tree [Wall64], Dadda's parallel counters [Dadd65], recursive carry-save adders [Kuck78], etc.

2.1.1 Wallace and Dadda Schemes:

In the Wallace and Dadda schemes, a large number of inputs are reduced successively to a smaller number until only two inputs are in every column. The main difference between these is that Wallace's scheme uses full adders whereas Dadda's scheme employs multi-input adders (counters). If counters are used that take d-inputs and produce e-outputs, we have found that the general relation between the number of inputs and outputs (< inputs) for any stage is given by (3)

$$outputs = e \left\lfloor \frac{inputs}{d} \right\rfloor + inputs \bmod d . \quad (3)$$

Here, $\lfloor x \rfloor$ ($\lceil x \rceil$) represents the smallest (greatest) integer $\geqslant$ ($\leqslant$) x, respectively. However, the use of complex parallel counters to realize multipliers requires more wiring, which would further increase with the incorporation of control signals and buses in a VLSI system. Therefore, a simpler counter such as a full adder should be used. Under this condition, Dadda's parallel counter scheme reduces to the Wallace tree. A block diagram of a multiplier using the Wallace tree is shown in Fig.3. For the Wallace tree, the number of inputs and outputs at every stage and the number of stages needed for final reduction can be obtained from the following algorithm.

```
$k_0 = K$ ;        /* initial number of inputs */
$s = 0$ ;          /* initialization of stage counter */

/* $n_c$[ ] is the number of adders (counters) at $s^{th}$ stage */

/* Assume $d = 3$ */

while($k_s > 2$) {
    $n_c[\, s \,] = \left\lfloor \frac{k_s}{3} \right\rfloor$ ;
/* $k_s$ is the number of inputs to $s^{th}$ stage */
    $k_{s+1} = 2\, n_c[\, s \,] + k_s \bmod 3$ ;
/* $k_{s+1}$ is the number of outputs from $s^{th}$ stage */
    $s^{++}$ ;
/* $s$ gives the number of stages needed for final reduction */
}
```

The flow of signals for a 2-bit slice in the middle of the tree to accomplish the summation of the partial products is shown in Fig.4. The complexity of the signal flow increases with m. To reduce all the partial products to final summation, a tree of $> \left\lfloor \frac{(n+1)(m-1)}{2} \right\rfloor$ adders with a depth of log m is needed. A final adder of length $(n + m - \log m)$ bits is required to generate the product. Thus following a typical approach [BrKu80], it can be seen that the multiplication time for this algorithm is $O(max(m,n))$ and it needs $O(mn)$ adders. The overall area-time efficiency for this algorithm is $O(max(m,n)mn)$.

2.1.2 Recursive Schemes:

Other schemes for reducing the partial products to final summation use recursive algorithms [Kuck78]. Here we describe a recursive algorithm that yields a regular layout and can accommodate the buses and control signal gracefully. The addition is both distributive and associative. Therefore Kuck's algorithm can be modified to the following recursive algorithm (R-algorithm), given by (4), to reduce the partial products to the final summation.

$$q_{ij} = pp_{ij} \oplus q_{i-1,j+2} \oplus c_{i-1,j+1} , \text{ and}$$
$$c_{ij} = pp_{ij} \cdot q_{i-1,j+2} + (pp_{ij} + q_{i-1,j+2}) \cdot c_{i-1,j+1} . \quad (4)$$

In (4) q_{ij} and c_{ij} are the bit-level partial sum and carry, respectively. Note that $q_{-1,j} = c_{-1,j} = 0$ for all j's. Eq.(4) is a nonlinear recurrence in $q_{i-1,j+2}$ and $c_{i-1,j+1}$, with the terms pp_{ij} regarded as known constants. We can solve it directly using an array of $\frac{(n+1)(m-1)}{2}$ bit level adders as shown in Fig.5a. The signal flow for this scheme is shown in Fig.5b. A typical cell contains the logic for generating pp_{ij} as well as a full adder as shown in Fig.6a; its layout is shown in Fig.6b. In Fig.5a, the final add operation is done using a fast adder similar to the one used for the Wallace tree. Note that the recurrence is solved using $O(mn)$ adders in $O(max(m\,,\,n))$ time. Thus, the area-time efficiency for this design is $O(max(m\,,\,n)mn)$.

2.2 Comparison of R-Algorithm and Wallace Tree Schemes

Area-time efficiency estimates are useful for reducing the choices in algorithm design, but when two algorithms have nearly equal $O()$'s, a careful analysis of the details is needed. For example, it is generally believed that the Wallace tree is the fastest design for a multiplier [Hwan79], and one such implementation has been reported [GGAG86]. However, from the above area-time analysis we see that the R-algorithm and the Wallace tree algorithm have the same $O()$'s. Further, we will show that the R-algorithm yields a more area-time efficient design in practice, especially when constrained layout is to be done. This is demonstrated as follows:

1. From Figs. 4 and 5b, it is seen that the signal flow for the R-algorithm is much simpler than that for the Wallace tree and hence leads to a simpler realization.
2. The number of carry-save adders for the R-algorithm and Wallace tree, N_R and N_W respectively, are given by (5)

$$N_R = \frac{(n+1)(m-1)}{2} ,$$
$$N_W > \left\lfloor \frac{(n+1)(m-1)}{2} \right\rfloor . \quad (5)$$

   It is clear that $N_W \geqslant N_R$ and therefore that the Wallace tree will occupy more area than the R-algorithm.
3. The depth of the Wallace tree is log m whereas that for the R-algorithm is $\frac{(m-1)}{2} \geqslant \log m$. However, the latter can generate $2\left\lceil \frac{m-1}{2} \right\rceil$ products as compared to log m for the Wallace tree.
4. The Wallace tree algorithm needs an $(n + m - \log m)$-bit fast adder whereas the R-algorithm needs an $(n+1)$-bit fast adder and thus generates the final product faster. The multiplication times for these two algorithms are given by (6).

$$T_W(n \times m) = max(T_{MBE}, T_x) + T_{MUX} + \sum_{i=1}^{\log m} T_{CSAi} + T_{(n+m-\log m)} ,$$

$$T_R(n \times m) = \max(T_{MBE}, T_x) + T_{MUX} + \max\left\{ \sum_{i=1}^{\left\lceil \frac{m-1}{2} \right\rceil} T_{CSAi} \,,\, T_{2\left\lceil \frac{m-1}{2} \right\rceil} \right\} + T_{(n+1)} . \quad (6)$$

In (6), T_{MBE} is the time needed for the modified Booth's encoder, T_x is the time needed to stabilize the X-input, T_{MUX} is the time for generating one of the inputs to the carry-save adder, T_{CSAi} is the adder delay for the i^{th} stage, and $T_{(i)}$ is the time needed to do fast addition of i-bits. Note that for the R-algorithm, the T_{CSAi} are identical for all stages; this is generally not the case for the Wallace tree.

Assuming that the first two terms for $T_W(n \times m)$ and $T_R(n \times m)$ are equal, all the carry-save adders take the same time of 0.5nsec, the fast adder adds in an average time of 0.3nsec/bit, and the number of bits in X and Y are equal, i.e., m=n, then we can plot the last two terms of (6) as shown in Fig.7. From the figure it is clear that $T_R(n \times m)$ is always less than $T_W(n \times m)$. Thus, the R-algorithm leads to a faster and more regular multiplier design.

It should also be noted that the fast adder required to generate the final product is crucial for the overall speed of the multiplication process. We will analyze the fast addition in the next section.

3. *ADDITION : ALGORITHMS AND IMPLEMENTATION*

The add operation is primitive but important since the performance of many complex modules like ALUs, multipliers, etc., depends on the speed of addition. Many authors have analyzed the problem and suggested some solutions for speeding up the add operation [BrKu82], [OnAt83], [OkBa85], and [NgIr85]. For fast addition, the carry-lookahead (CLA) scheme is the most obvious, but if carried out directly it results in an irregular design with large fan-in and fan-out gates. To overcome this difficulty, a tree based CLA scheme is suggested in [BrKu82] that solves the large fan-in and fan-out problem. Another solution with area-time efficiency of $O(n\ (log\ n)^2)$ is given in [NgIr85] where negative logic is used to generate block carries. In all these $O()$ calculations, one generally assumes that the gate delays are the same irrespective of gate complexities. Because of this the results obtained do not hold good in practice. The approach of using a variable number of bits for different groups as suggested in [OkBa85] is also not very practical from the layout point of view.

We consider the adder design problem under the following conditions:

1. The design is to be implemented in either NMOS or CMOS high performance technologies.
2. No more than two devices are to be connected in series from either supply rail to the output node so that circuits can be used with low supply voltages (2-3.5V).
3. The design is to be highly modular to increase the layout efficiency.
4. The signal propagation delay along silicide wires is proportional to the square of the length. This is in contrast to the assumptions made in [NgIr85] when metal is used to route the signals. Thus, increased device sizes cannot make the delay along wires constant.

With the above constraints, we will analyze the binary addition using the carry-lookahead carry-select adder scheme and show that it provides a design that is highly modular and area-time efficient.

Let $A = \{a_{n-1}a_{n-2}.....a_1a_0\}$ and $B = \{b_{n-1}b_{n-2}.....b_1b_0\}$ be two binary numbers that are to be added to produce their sum $S = \{s_{n-1}s_{n-2}.....s_1s_0\}$. The sum and carry from the i^{th} bit position can be expressed by (7).

$$\begin{aligned} c_{-1} &= C_{in} \\ c_i &= g_i + p_i\ c_{i-1} \\ s_i &= a_i \oplus b_i \oplus c_{i-1} \end{aligned} \qquad (7)$$

where $g_i = a_i\ b_i$, and $p_i = a_i + b_i$ are the generate and propagate terms, respectively. In (7), c_i is expressed recursively. It can be used as such to design adders, but this leads to a slower adder [OnAt83]. The fastest way to add is to generate the carry and sum for all the bits simultaneously. However, this is obviously not a practical solution, and a compromise must be made in terms of the number of bits the carry is looked ahead; typically it is chosen between 4 to 8 bits. As the number of bits are increased in the group, the fan-in and fan-out problems become important. For example, the maximum fan-in or fan-out for a gate is $(m + 1)$ when m bits are used in the group.

In (7), the c_i's can be expanded to see the effects of fan-in and fan-out.

$$\begin{aligned} c_0 &= p_0\ c_{-1} + g_0\ , \\ c_1 &= p_1\ p_0\ c_{-1} + p_1\ g_0 + g_1\ , \\ c_2 &= p_2\ p_1\ p_0\ c_{-1} + p_2\ p_1\ g_0 + p_2\ g_1 + g_2\ , \\ c_3 &= p_3\ p_2\ p_1\ p_0\ c_{-1} + p_3\ p_2\ p_1\ g_0 + p_3\ p_2\ g_1 + p_3\ g_2 + g_3\ , \\ s_0 &= a_0 \oplus b_0 \oplus c_{-1}\ , \\ s_1 &= a_1 \oplus b_1 \oplus c_0\ , \\ s_2 &= a_2 \oplus b_2 \oplus c_1\ , \text{ and} \\ s_3 &= a_3 \oplus b_3 \oplus c_2\ . \end{aligned} \qquad (8)$$

In (8), the maximum fan-out is from the c_{-1} signal, and the maximum fan-in is to generate the c_3 signal. In both cases it is 5.

Although we can use multi-level carry generation logic to speed up the operation, it results in a disproportionate increase in the area and the wire length. The binary tree structure of [BrKu82] has a similar problem of routing; moreover the number of gates in its signal path is not different from the case when we use one level of carry-lookahead over a 4-bit group only. The logic for the *3rd* bit position as given by (8) is shown in Fig.8. Here, we have used 2-input NAND gates, multi-input NOR gates, 2-input XOR gates, and inverters only. The carry-out from the *3rd* bit becomes the input for the next 4-bit group, and so on.

A 32-bit adder using this 4-bit carry-lookahead scheme has been designed. It requires 7 levels to generate the sums in bit positions 28 through 31, which is the same as for the Brent-Kung adder [BrKu82]. However, their scheme has the distinct feature that the carries for bit positions 2^i, where $i = 1, 2, ...$, can be generated faster than others. This can be used with advantage to design large modulo adders, say for 64, 128 or 256 bits. We can also mix the binary tree adders and the fixed carry-lookahead schemes to take advantage of both.

For the fixed length carry-lookahead scheme, the area-time efficiency is $O(\left\lceil \frac{n}{m} \right\rceil^2)$ as compared to $O(n\ (log\ n)^2)$ reported in [NgIr85] and $O(n \log n)$ in [BrKu82]. The area-time efficiency for our design is not affected whether it is used in the constrained or unconstrained layouts; this is not true for the other designs. From Fig.8 it can also be shown that the adder delay T_n is given by (9)

$$\begin{aligned} T_n = \max\ (T_{NR2}\ ,\ T_{ND2}) + \left\lceil \frac{n}{m} \right\rceil \left(2T_{NRm+1} + T_{inv}\right) \\ + 2\ (T_{XOR} + T_{inv}) \end{aligned} \qquad (9)$$

where T_{NRi} is the gate delay for an i-input NOR gate, T_{ND2} is the gate delay for a 2-input NAND gate, T_{XOR} is the gate delay for a 2-input XOR gate, and T_{inv} is the delay for an inverter. Note that in CMOS technology we would use n-channel devices to implement the logic and a p-channel device for the load in multi-input NOR gates in order to satisfy the assumptions made above.

4. *CONCLUSIONS*

In this paper we have analyzed the algorithms and the implementations of two important arithmetic elements: the binary adder and multiplier. The analysis has been done to help determine which algorithm is best suited for VLSI system designs. It has been shown that

1. The structures and algorithms good for discrete module design may not provide efficient design for VLSI systems.
2. The theoretical studies for area-time efficiency can only give general direction.
3. For multiplication, the R-algorithm is better suited for large VLSI system designs and results in a higher performance solution than the Wallace tree.

4. Carry-lookahead over a small and fixed number of bits, e.g. 4-bit slices, results in a regular layout and high speed adders. To improve the speed for larger size adders , say 64 to 256 bits, this should be used along with the binary tree carry generation scheme of [BrKu82].

ACKNOWLEDGEMENTS

I would like to thank A.D.Lopez for the cell layout in gate-matrix style, and H.T.Weston for his careful review of the manuscript.

REFERENCES

[AbAn80]: Abelson, H. and Andreae, P., "Information Transfer and Area-Time Trade-offs for VLSI Multiplication", Communications of ACM, Vol. 23, No. 1, pp.20-22; January 1980.

[BrKu80]: Brent, R.P. and Kung, H.T., 'The Chip Complexity of Binary Arithmetic", Proc. of 12th ACM Symp. on theory of Computing, (Los Angeles), pp.190-200; May, 1980.

[BrKu82]: Brent, R.P. and Kung, H.T., "A Regular Layout for Parallel Adders", IEEE Trans. Computers, Vol. C-31, pp.260-264; March 1982.

[Dadd65]: Dadda, L., "Some schemes for Parallel Multipliers", Alta Frequenza, Vol. 34, pp.349-356; 1965.

[GGAG86]: Gamal, A.E., Gluss, D., Ang, P-H., Greene, J., and Reyneri, J., "A CMOS 32b Wallace Tree Multiplier-Accumulator", ISSCC Digest Tech. Papers, pp.194-195,345; Feb.19-21, 1986.

[Hwan79]: Hwang, K., Computer Arithmetic, New York: John Wiley & Sons, 1979.

[Kuck78]: Kuck, D.J., The Structure of Computers and Computations, Vol.1, New York: John Wiley & Sons, 1978.

[NgIr85]: Ngai, T-F. and Irwin, M.J., "Regular, Area-Time Efficient Carry-Lookahead Adders", Proc. 7th Symp. Computer Arithmetic, pp.9-15; May 1985.

[OkBa85]: Oklobdzija, V.G. and Barnes, E.R., "Some Optimal Schemes for ALU Implementation in VLSI Technology", ibid., pp.2-8; May 1985.

[OnAt83]: Ong, S. and Atkins, D.E., "A Comparison of ALU Structures For VLSI Technology", Proc. 5th Symp. Computer Arithmetic, pp.10-15; May 1983.

[PrVu81]: Preparata, F.P. and Vuillemin, J., "Area-Time Optimal VLSI Networks for Computing Integer Multiplication and Discrete Fourier Transforms", Proc. I.C.A.L.P. Symp., Haifa, Israel; July, 1981.

[Shar85]: Sharma, R., "Architecture Design of a High-Quality Speech Synthesizer Based on the Multipulse LPC Technique", IEEE Jour. Selected Areas in Communications, Vol. SAC-3, pp.377-383; March 1985.

[Wall64]: Wallace, C.S., "A Suggestion for a Fast Multiplier", IEEE Trans. Electronic Computers, Vol. EC-13, pp.14-17; Feb. 1964.

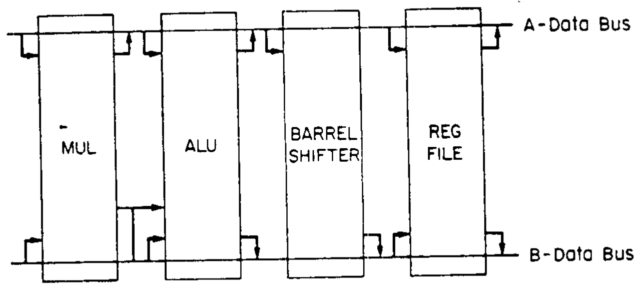

FIG.1: A Typical Data-Path of an Arithmetic Processor

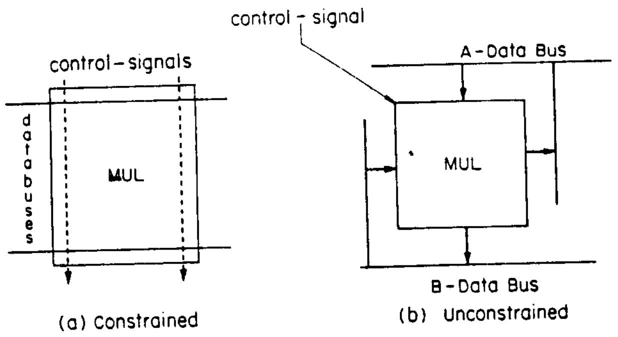

FIG.2: Floor-Plans for Multipler

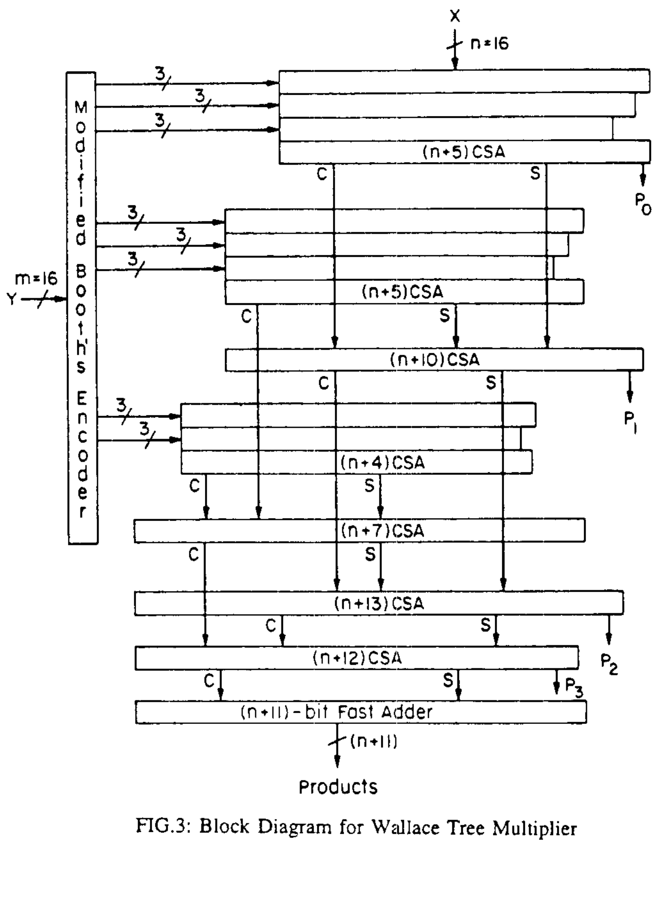

FIG.3: Block Diagram for Wallace Tree Multiplier

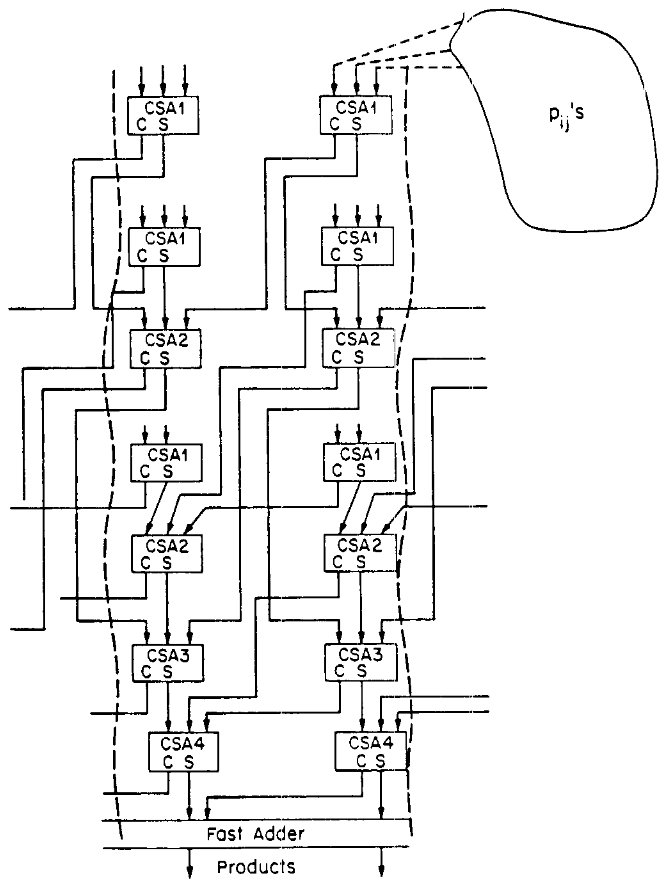

FIG.4: Signal-Flow for Wallace Tree Multiplier

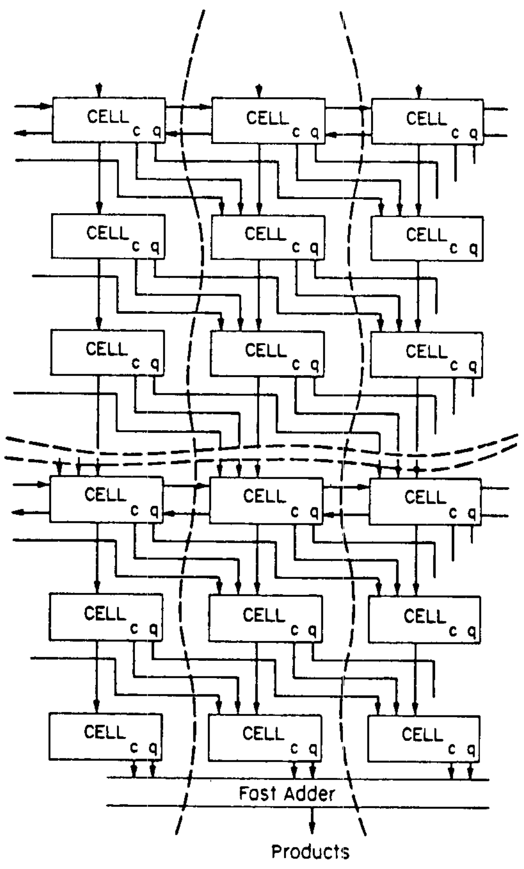

FIG.5b: Signal-Flow for R-Algorithm Multiplier

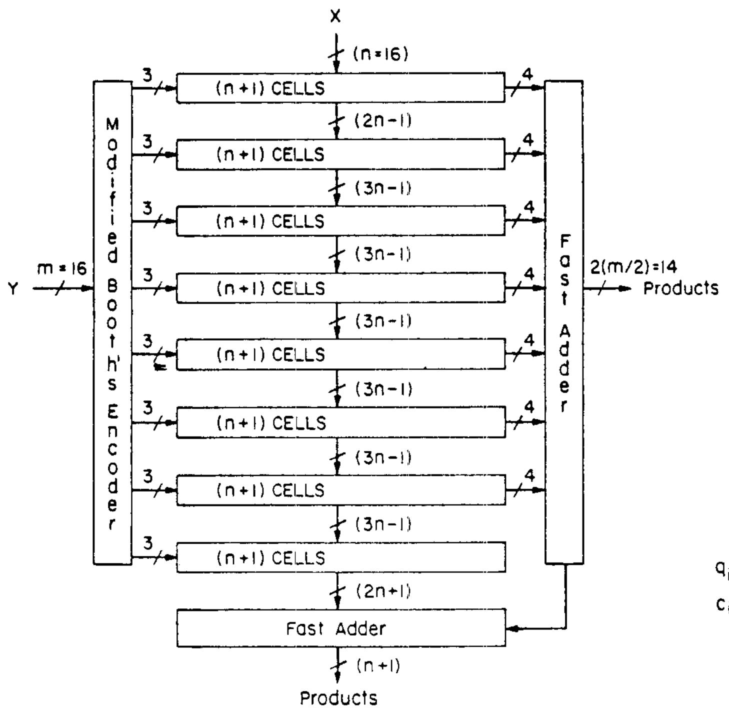

FIG.5a: Block Diagram for R-Algorithm Multiplier

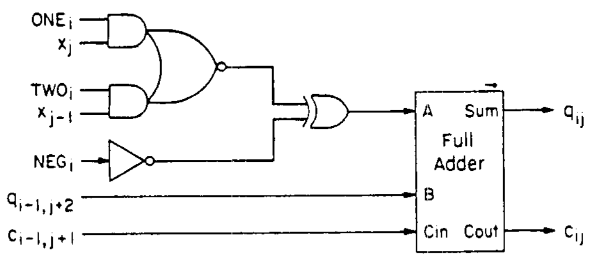

FIG.6a: Logic Diagram of a Typical Cell

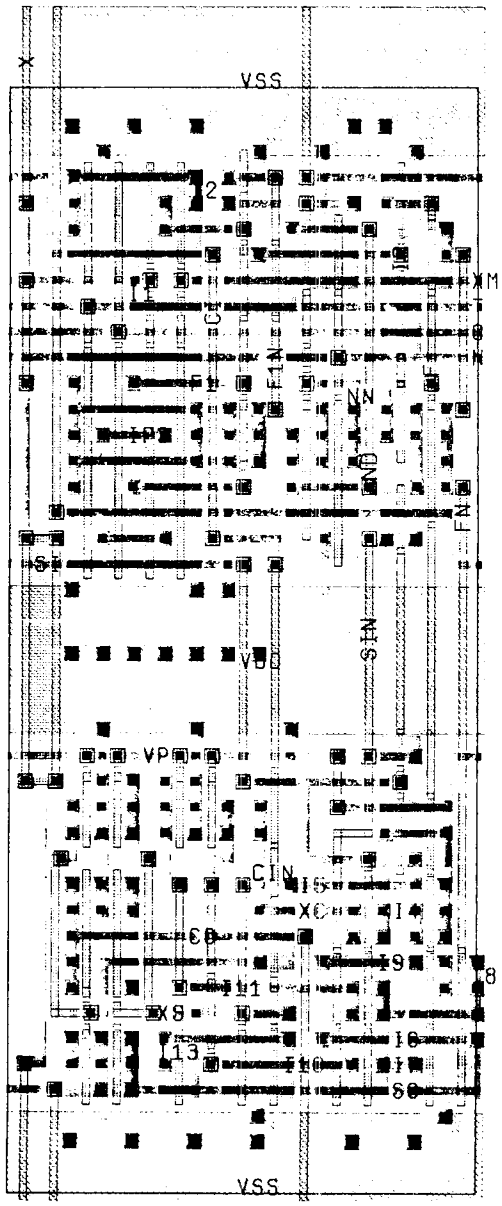

FIG.6b: The Cell Layout

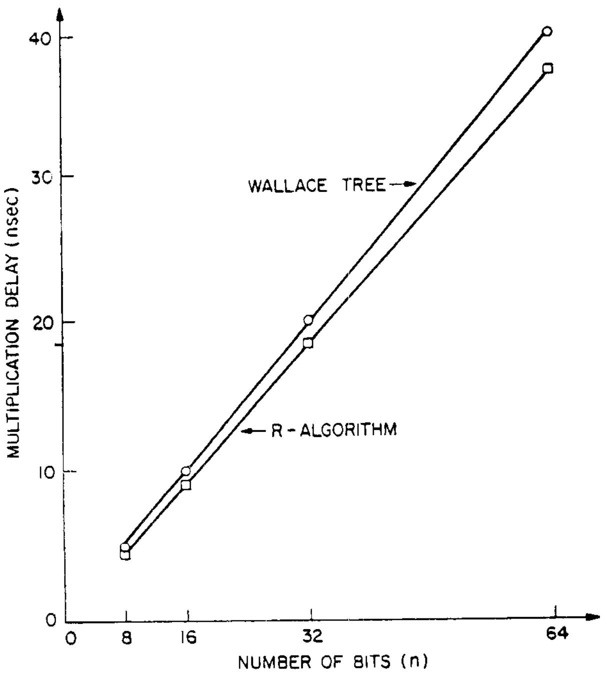

FIG.7: Multiplication Times

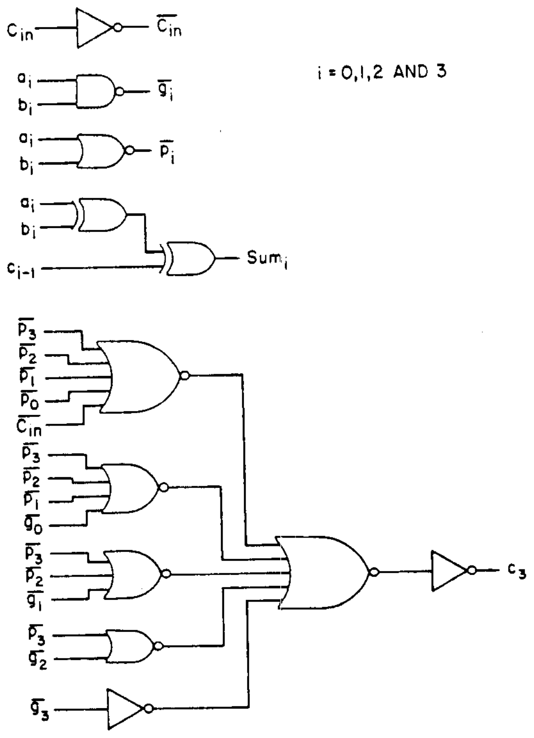

FIG.8: Logic Diagram for Carry Output Stage