Fast Parallel Algorithms for Binary Multiplication and Their Implementation on Systolic Architectures

BHABANI P. SINHA, MEMBER, IEEE, AND PRADIP K. SRIMANI, MEMBER, IEEE

***Abstract*—Two algorithms for parallel multiplication of two n-bit binary numbers have been presented in this paper. The first algorithm computes the product in $(n + \lceil \log_2 n \rceil)$ units of addition time of a single-bit full adder and is easily implementable on the $n \times n$ systolic SIMD architecture. The second algorithm is still faster and requires approximately $3\lceil \log_2 n \rceil$ units of time of a single-bit full adder. Both the algorithms require almost regular interconnection between only two types of cells and hence are very suitable for VLSI implementation. Both of them can also be easily modified to handle two's complement numbers with constant difference in time.**

***Index Terms*—Binary multiplication, column compression, precarry addition, regular interconnection, VLSI chip design.**

I. Introduction

MULTIPLICATION of two n-bit numbers plays a very important role in many applications and with the advent of VLSI technology, parallel algorithms for multiplication have become increasingly important. The ordinary straightforward multiplication scheme [1] can multiply two n-bit numbers in $2n$ units of time (a unit of time is defined as the time of a single-bit full adder). This execution time for parallel multiplication has been decreased substantially by using the technique of partial product matrix reduction or column compression [2]–[4]. This technique of column compression leads to less execution time than iterative array algorithms but it requires a large number of irregular interconnections between different types of cells. The pipelined architecture of [5] for multiplication using carry save adders has also been used in some commercially available systems like CDC Star 100 [6] to achieve substantial speed in multiplication. Recently, two algorithms for iterative array multiplication have been reported in [7] both of which require only n units of time for multiplication of two n-bit numbers and involve almost regular interconnection structures of the multiplier array cell elements which are ideal for VLSI implementation. Authors in [11] have also reported an $O(\log_2 n)$ multiplication scheme using redundant binary trees.

In this paper, we propose two new parallel algorithms for multiplication of two n-bit numbers both of which use the technique of column compression to increase the speed of execution. The first algorithm, $M1$, requires $(n + \lceil \log_2 n \rceil)$ units of time and is realizable on an $n \times n$ systolic SIMD architecture with $O(n^4)$ AT^2 value [8]. The next algorithm, $M2$, which is an improved version of $M1$, exploits further parallelism present in the operations resulting in an execution time of approximately $3\lceil \log_2 n \rceil$ units and $O(n^2(\log_2 n)^2)$ AT^2 value. A systolic architecture to multiply two 64-bit numbers using Algorithm $M2$ has been shown as an implementation example and the corresponding data flow in various stages has been shown. Both the algorithms require almost regular interconnection between two types of cells: single-bit full adders and precarry generators (a two-level circuit of seven gates) and hence are very suitable for single-chip VLSI implementation.

Manuscript received September 15, 1986; revised July 15, 1987.

The authors are with the Department of Computer Science, Southern Illinois University, Carbondale, IL 62901.

IEEE Log Number 8825679.

II. Preliminaries

Let U and V be the two numbers to be multiplied whose binary representations are as follows:

$$U = u_{n-1}u_{n-2} \cdots u_1u_0$$

$$V = v_{n-1}v_{n-2} \cdots v_1v_0$$

where u_0 and v_0 are the least significant bits. Let $W = UV$, where W can be expressed as $W = w_{2n-1}w_{2n-2} \cdots w_1w_0$. Hence, we can write

$$W = \sum_{i=0}^{n-1} 2^i \sum_{j=0}^{i} u_{i-j}v_j + \sum_{i=n}^{2n-2} 2^i \sum_{j=i-n+1}^{n-1} u_{i-j}v_j \tag{1}$$

where the first term accounts for the least significant n bits $w_0 \cdots w_{n-1}$ and the second term for the most significant n bits $w_n \cdots w_{2n-1}$ of W. Thus, any w_i can be obtained by computing the appropriate summation and adding carry bits that are propagated from the less significant bit positions. Denoting u_iv_j by a_{ij}, we rewrite (1) as

$$W = \sum_{i=0}^{n-1} 2^i \sum_{j=0}^{i} a_{i-j,j} + \sum_{i=n}^{2n-2} 2^i \sum_{j=i-n+1}^{n-1} a_{i-j,j}. \tag{2}$$

Addition of Two n-Bit Numbers in $\lceil \log_2 n \rceil$ Time

As we will see later, the final steps in both of our proposed algorithms, $M1$ and $M2$, are to add two binary numbers of equal length and this we plan to do in $\lceil \log_2 n \rceil$ units of time using the concept of precarry introduced first in [9]. We

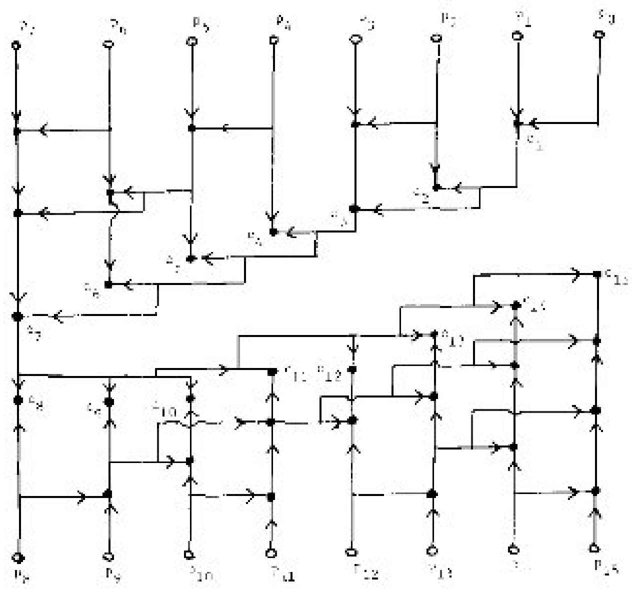

Fig. 1. Carry generation from precarry vector for $n = 16$.

describe the method briefly for the sake of completeness and then propose a simple hardware adder to implement the precarry concept. We also study the complexity of the module that computes the sum of two numbers in $\lceil \log_2 n \rceil$ time units.

Consider two binary numbers $A = a_{n-1} \cdots a_0$ and $B = b_{n-1} \cdots b_0$ which are to be added to produce the sum $S = s_n s_{n-1} \cdots s_0$. Define a *precarry* vector $P = p_{n-1} \cdots p_0$ where $p_0 = 0$ and for $i > 0$

$$p_i = \begin{cases} 0, & \text{if } a_{i-1} = b_{i-1} = 0 \\ 1, & \text{if } a_{i-1} = b_{i-1} = 1 \\ 2, & \text{otherwise.} \end{cases}$$

Define a binary operator \$ as

$$x \,\$\, y = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } x = 1 \\ y, & \text{if } x = 2 \end{cases}$$

where the operands x or y can assume values 0, 1, and 2. It is to be noted that the operator \$ is associative but not commutative. Now consider the carry bits generated in successive bit positions while adding the two numbers A and B. This carry vector $C = c_{n-1} \cdots c_0$ can be defined as

$$c_i = \begin{cases} 0, & \text{if } i = 0 \\ a_{i-1}b_{i-1} + a_{i-1}c_{i-1} + b_{i-1}c_{i-1}, & \text{if } i > 0 \end{cases}$$

and the sum S can be defined as

$$s_i = \text{EXOR}(a_i, b_i, c_i), \qquad 0 \le i \le n \qquad [\text{assuming } a_n = b_n = 0].$$

It can be readily seen that the carry vector can be generated from the precarry vector as

$$c_i = p_i \,\$\, (p_{i-1} \,\$\, (p_{i-2} \cdots (p_1 \,\$\, p_0)) \cdots), \qquad 0 \le i \le n-1.$$

Evidently these \$ operators can be executed in parallel and hence if we have a sufficient number of hardware modules that execute \$ operation on two given operands, then the carry vector can be generated from the precarry vector in $\lceil \log_2 n \rceil$ units of time. Two bits are needed to represent each element of the precarry vector. If b_{i1} and b_{i2} are the bits to represent p_i, then we may assign $b_{i1} = b_{i2} = 0$ for $p_i = 0$, $b_{i1} = b_{i2} = 1$ for $p_i = 1$ and $b_{i1} = \bar{b}_{i2}$ for $p_i = 2$. The following two equations specify the logic module that executes the operation $p_k = p_i \,\$\, p_j$:

$$b_{k1} = b_{i1}b_{i2} + (b_{i1} \oplus b_{i2}) \cdot b_{j1}$$

$$b_{k2} = b_{i1}b_{i2} + (b_{i1} \oplus b_{i2}) \cdot b_{j2}.$$

The number $N(n)$ of such modules (precarry generators) required to compute the precarry vector C, for $n = 2^k$, is given by the recurrence

$$N(2^k) = 2N(2^{k-1}) + 2^{k-1}$$

whose solution is given by $N(2^k) = k \cdot 2^{k-1}$. An example of interconnecting all these modules to generate the carry vector is shown in Fig. 1 for $n = 16$. The logic modules have been shown by black dots. Evidently the precarry logic modules can be designed as two-level AND–OR circuits just like single-bit full adders and hence the above logic operations require the same amount of time as that of a single-bit full adder. We can say that two n-bit numbers can be added in $\lceil \log_2 n \rceil + 1$ time units using this precarry technique and henceforth we call it an n-bit precarry adder. It should be noted that the precarry adder requires a maximum of $\lceil n/2 \rceil$ broadcasting communications.

III. The Parallel Algorithm $M1$

We describe here Algorithm $M1$ that computes W from the above equation assuming that all the a_{ij}'s are known in

```
          begin
init :      for i = 0 to (2n-2) do in parallel
              if i < n then
                begin
                  $S_i^0 \leftarrow a_{i,0}$ ; $C_i^0 \leftarrow 0$;
                end
              else
                begin
                  $S_i^{i-n} \leftarrow a_{n-1,i-n+1}$; $C_i^{i-n} \leftarrow 0$;
                end;
            $S_{2n-1}^{n-1} \leftarrow 0$;
loop1 :     for i = 1 to (2n-3) do in parallel
              compute $l_i$ and $n_i$;
loop2 :       for k = $l_i$ to $n_i$ do
                begin
                  $S_i^k \leftarrow \mathcal{S}(S_i^{k-1}, C_i^{k-1}, a_{i-k,k})$;
                  $C_{i+1}^k \leftarrow \mathcal{C}(S_i^{k-1}, C_i^{k-1}, a_{i-k,k})$;
                end
output1 :   for i = 0 to (n-1) do
              $w_i \leftarrow S_i^1$ ;
            $c_n \leftarrow 0$;
precarry :  for i = n to (2n-2) do in parallel
              begin
                $c_{i+1} \leftarrow SP(S_i^{n-1} S_{i-1}^{n-1} .. S_n^{n-1}, C_i^{n-1} C_{i-1}^{n-1} .. C_n^{n-1})$;
output2 :       for i = n to (2n-1) do in parallel
                  $w_i \leftarrow \mathcal{S}(S_i^{n-1}, C_i^{n-1}, c_i)$;
          end.
```

Fig. 2. Algorithm $M1$.

advance. The main idea is that we pass on the carry to the next higher bit position as soon as it is generated; and after $(n - 1)$ cycles of additions (and subsequent carry propagations) we are left with a $(2n - 1)$ bit sum vector $S = s_{2n-2} \cdots s_0$ and an n-bit carry vector $C = c_{2n-1} \cdots c_n$. The product W will then be obtained by a parallel addition of C with the most significant $(n - 1)$ bits $s_{2n-2}s_{2n-1} \cdots s_n$ of S. This parallel addition can be done in $\lceil \log_2 n \rceil$ steps using an n-bit precarry adder.

The complete Algorithm $M1$ has been given in Fig. 2. To describe the algorithm first we define two integers l_i and n_i as follows:

$$l_i = \begin{cases} 1, & \text{for } i < n \\ i - n + 1, & \text{otherwise} \end{cases}$$

$$n_i = \begin{cases} i, & \text{for } i < n \\ n - 1, & \text{otherwise.} \end{cases}$$

The variables S_i^k and C_i^k in Algorithm $M1$ indicate the sum and carry bits, respectively, generated at the ith bit position of W after $k - l_i + 1$ iterations of the loop "loop2." The symbols $\mathcal{S}$ and $\mathcal{C}$ denote the sum and carry functions, respectively, in a single-bit binary full adder. The function SP over the two bit vectors $S_i^{n-1} S_{i-1}^{n-1} \cdots S_n^{n-1}$ and $C_i^{n-1} C_{i-1}^{n-1} \cdots C_n^{n-1}$ gives the carry bit c_{i+1} for the $(i + 1)$th position (by the already described precarry technique) to be used in generating the sum of these two vectors.

Since Algorithm $M1$ is the direct implementation of (2), the proof of its correctness is obvious. Execution of loop2 requires $(n - 1)$ steps for $i = n - 1$. Also the precarry function SP over two bit vectors of length n each requires $\lceil \log_2 n \rceil$ steps. Consequently we get the following theorem.

Theorem 1: The time complexity of Algorithm $M1$ is $(n + \lceil \log_2 n \rceil)$.

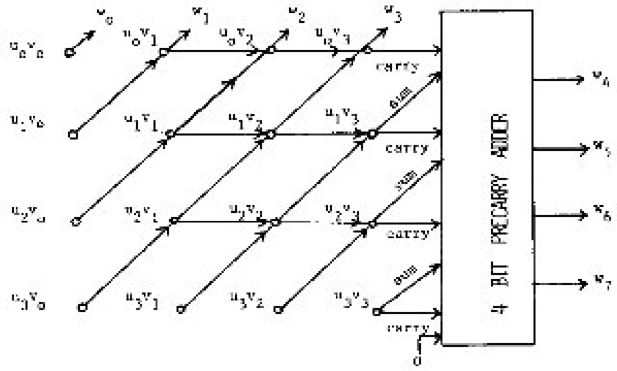

Fig. 3. Systolic array for $n = 4$.

Implementation of Algorithm M1 on a Systolic Array

Algorithm $M1$ can be easily implemented on a systolic $n \times n$ processor array where each processor element of the array is a single-bit full adder. Fig. 3 shows such a systolic array for $n = 4$. Initially all the a_{ij}'s are stored on the processor elements P_{ij}'s, $0 \leq i, j \leq n - 1$, as shown in the figure. We note that in loop2 of Algorithm $M1$ the sum of the indexes of a_{ij}'s required to compute S_i^k and C_{i+1}^k for a given value of i is always equal to i for all values of k. This indicates that there should be a diagonal flow (from bottom to top) of the sum bits S_i^k's and a horizontal flow (from left to right) of the carry bits C_{i-1}^k's for the operation with the next higher value of k. The operations on each diagonal are so synchronized that at the kth step, $1 \leq k \leq n - 1$, all the processors P_{ik}'s, $0 \leq i \leq n - 1$ only at the kth column perform the addition operation. After $(n - 1)$ cycles of sum and carry propagation and subsequent additions, the processor elements $P_{0,1}, P_{0,2} \cdots, P_{0,n-1}$ store the bits $w_0, w_1, \cdots, w_{n-1}$ of W. The processor elements $P_{1,n-1}, P_{2,n-1}, \cdots, P_{n-1,n-1}$ now contain the most significant parts of the sum vector and the carry vector to be finally added in $\lceil \log_2 n \rceil$ steps to generate the n most significant bits of the final sum W using an n-bit precarry adder.

IV. Algorithm $M2$

Let x be a positive integer, $x \geq 1$, and x' be another integer defined as

$$x' = \begin{cases} x, & \text{if } x = 3p \\ x + 1, & \text{if } x = 3p + 2 \\ x + 2, & \text{if } x = 3p + 1 \end{cases} \quad (3)$$

where p is any integer. Let T be a function defined on a set of x bits $B = \{b_1, b_2, \cdots, b_x\}$ in the following way where we assume $b_{x+1} = b_{x+2} = 0$.

$$T : \{b_1, b_2, \cdots, b_x\} \rightarrow \{\mathcal{S}(b_j, b_k, b_l), \mathcal{C}(b_j, b_k, b_l), T(B - \{b_j, b_k, b_l\}) \mid 1 \leq j, k, l \leq x', j \neq k \neq l\} \quad (4)$$

```
        begin
            $w_{2n-1} \leftarrow 0$;
loop1 :     for i = 1 to (2n-2) do in parallel
              begin
                $(b_{i1}, b_{i2}, \ldots, b_{ix_i}) \leftarrow (a_{m,i-m}, a_{m-1,i-m+1}, \ldots, a_{i-m,m})$;
                set $y_i \leftarrow \lceil x_i/3 \rceil$;
                set t to the smallest integer such that $f^t(n) = 2$;
loop2 :         for k = 1 to (t+2) do
                  begin
                    $(s_{i,1}, \ldots, s_{i,y_i}, c_{i,1}, \ldots, c_{i,y_i}) \leftarrow T(b_{i1}, b_{i2}, \ldots, b_{ix_i})$;
                    $(b_{i1}, b_{i2}, \ldots, b_{i,y_i+y_{i-1}}) \leftarrow (s_{i1}, s_{i2}, \ldots, s_{i,y_i},$
                                                 $c_{i-1,1}, c_{i-1,2}, \ldots, c_{i-1,y_{i-1}})$;
                    $w_{2n-1} \leftarrow w_{2n-1} + c_{2n-2,1}$;
                    $x_i \leftarrow y_i + y_{i-1}$; $y_i \leftarrow \lceil x_i/3 \rceil$
                  end
              end;
            $w_0 \leftarrow a_{00}$; $c_1 \leftarrow 0$;
loop3 :     for i = 1 to 2n-2 do in parallel
            $c_{i+1} \leftarrow SP((b_{i1}, b_{i-1,1}, \ldots, b_{11}), (b_{i2}, b_{i-1,2}, \ldots, b_{12}))$;
            for i = 1 to 2n-2 do in parallel
            $w_i \leftarrow (b_{i1}, b_{i2}, P_i)$;
            $w_{2n-1} \leftarrow w_{2n-1} + c_{2n-1}$
        end.
```

Fig. 4. Algorithm $M2$.

where $\mathcal{S}(b_j, b_k, b_l)$ and $\mathcal{C}(b_j, b_k, b_l)$ are, respectively, the sum and carry bits in a single-bit binary full adder with inputs b_j, b_k, and b_l. The function T operates on x number of bits to generate $\lceil 2x/3 \rceil$ bits as output. Let m be an integer defined as

$$m = \begin{cases} i, & \text{if } i < n \\ n-1, & \text{if } i \geq n \end{cases} \tag{5}$$

and x_i be another integer defined as

$$x_i = \begin{cases} i+1, & \text{if } i < n \\ 2n-i-1, & \text{if } i \geq n. \end{cases} \tag{6}$$

It follows from (2) that w_i can be obtained by adding x_i bits $a_{m,i-m}, a_{m-1,i-m+1}, \cdots, a_{i-m,m}$ together with the final carry bit propagated from the $(i-1)$th bit position. This addition job at each bit position i to generate the final sum bit has to be done by repeated application of the T operator; after each T operation, the generated bits $\mathcal{S}$ are to be retained in this bit position, the generated $\mathcal{C}$ bits are to be transmitted to the next higher bit position, and the generated $\mathcal{C}$ bits generated at the preceding bit position are to be collected at this position to be ready for the next T operation. The central idea of our proposed Algorithm $M2$ is as follows.

For each i, the T operation is done on all bits to be added to get w_i and the generated carry bits $\mathcal{C}(b_j, b_k, b_l)$ are propagated to the next higher order bit position $(i+1)$. However, if $b_k = b_l = 0$, $\mathcal{C}(b_j, 0, 0) = 0$ and this carry bit is not propagated. At the subsequent step, the T operation is performed at each bit position on the sum bits $\mathcal{S}(b_j, b_k, b_l)$ generated at this bit position and the propagated carry bits $\mathcal{C}(b_j, b_k, b_l)$ from the lower order bit position. This process continues until we get only two bits, one sum bit and one carry bit, at each bit position. All these operations are done in parallel for all i. At this point, the resulting sum vector $s_{2n-2}s_{2n-3} \cdots s_1 s_0$ and the carry vector $c_{2n-1}c_{2n-2} \cdots c_1 c_0$ are added together by using a $2n$-bit precarry adder as before. The complete pseudocode of the algorithm appears in Fig. 4. The function $f(n)$ used in the algorithm is defined as

$$f(n) = \lceil 2n/3 \rceil.$$

If t is the minimum integer such that $f^t(n) = 2$, then after executing the T operation t times we get two sum bits and two carry bits. And hence we need two more T operations to reduce them to one sum bit and one carry bit. This explains $(t+2)$ times execution of loop2. Hence, this integer t for any given n will evaluate the complexity of the algorithm.

To estimate the time complexity of Algorithm $M2$ we need the following lemma.

Lemma 1: Given that $f(n) = \lceil 2n/3 \rceil$, the smallest integer t such that $f^t(n) = 2$ satisfies the inequality

$$t \leq \lceil 1.71 \log_2 n \rceil - 1.$$

Proof: Let us form a tree of all natural numbers greater than or equal to 2, with 2 as the root of the tree in such a way that every number N is the parent of some number(s) N' whenever $N = \lceil 2N'/3 \rceil$. The first few levels of the tree will look as in Fig. 5. The levels of the nodes increase downward, with that of the root node equal to zero. We see that from level 3 onwards, there will be more than one node at a given level. Also, since $\lceil (2/3)(3p-1) \rceil = \lceil (2/3)(3p) \rceil = 2p$ and $\lceil (2/3)(3p+1) \rceil = 2p+1$, it follows that all the even integers in the tree will have two children while the odd integers will have only one. If the number of nodes at a level L (≥ 3) is q, the number of nodes at level $(L+1)$ is then equal to $\lceil 3q/2 \rceil$ or $\lfloor 3q/2 \rfloor$. Let us now consider the series

$$S = 4 + 6 + 9 + 13 + 21 + 31 + 47 + 70 + 105 + \cdots,$$

where the rth term t_r of S denotes the total number of nodes at a level $L = r + 4$.

Let us also consider the following G.P. series.

$$S' = 4 + 6 + 9 + 13.5 + 20.25 + 30.375 + 45.5625 + \cdots.$$

If t'_r is the rth term of the series S', the series is defined by the relation $t'_{r+1} = (3/2)t'_r$. Hence, if $t_r > t'_r + 1$, we get

$$\begin{aligned} t_{r+1} &\geq \lfloor (3/2)(t'_r + 1) \rfloor \geq \lfloor (3/2)t'_r \rfloor + 1 \\ &> (3/2)t'_r \\ &= t'_{r+1}. \end{aligned}$$

We see that $t_7 > t'_7 + 1$ and hence $t_r \geq t'_r$ for all r except $r = 4$ when $t_4 < t'_4$. Hence, the sum of the first r terms $(r > 4)$ of S is always greater than or equal to the sum of the first r terms of S', which is equal to $8[(3/2)^{r-1} - 1]$. This implies that the total number of nodes in r consecutive levels $(r > 4)$ starting from level $L = 5$ is greater than or equal to $8[(3/2)^{r+1} - 1]$. However, since $t_4 - t'_4 < 1$ and the nodes numbered up to 9 are included in the levels from $L = 0$ to $L = 4$, we can easily check that the level of a given integer N is the minimum value of L satisfying the inequality

$$N - 9 \leq 8[(3/2)^{L-4} - 1]$$

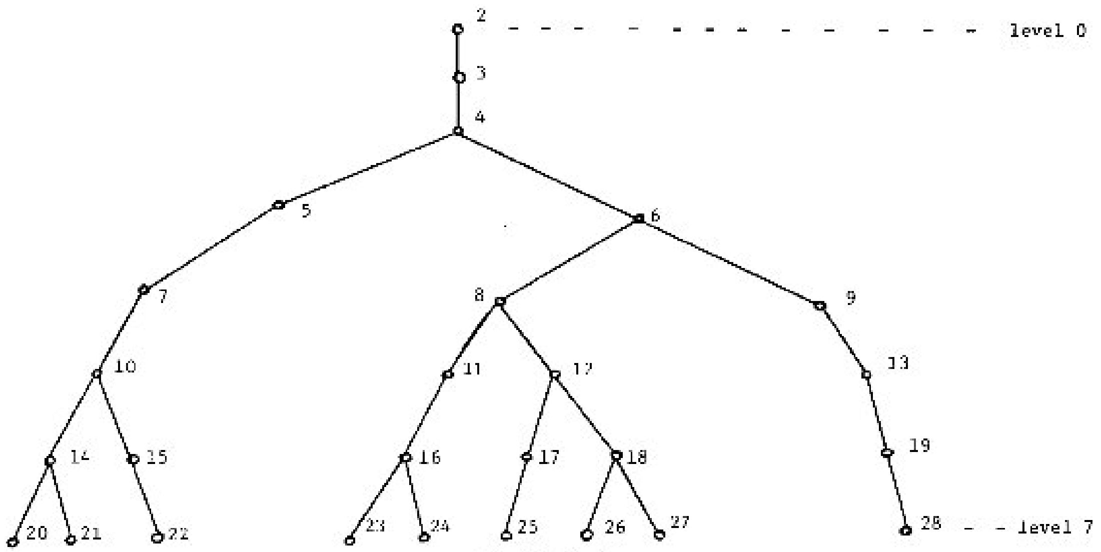

Fig. 5. The tree.

or

$$L-4 \geq \log_{3/2}[(N-1)/8]$$

or

$$L \geq 1.7095 \log_2(N-1) - 1.1285.$$

So the value of t is equal to the level of the integer n in the above tree and hence

$$t \leq \lceil 1.71 \log_2 n \rceil - 1. \qquad \text{Q.E.D.}$$

We now have the following theorem.

Theorem 2: The time complexity $T(n)$ of Algorithm $M2$ is given by

$$T(n) \leq \lceil 2.71 \log_2 n \rceil + 3.$$

Proof: Loop2 of Algorithm $M2$ requires $(t + 2)$ steps and loop3 requires $\lceil \log_2(2n) \rceil = \lceil \log_2 n \rceil + 1$ steps of additions in a single-bit full adder. Hence,

$$\begin{aligned} T(n) &= t + \lceil \log_2 n \rceil + 3 \\ &\leq \lceil 1.71 \log_2 n \rceil + \lceil \log_2 n \rceil + 2 \\ &\leq \lceil 2.71 \log_2 n \rceil + 3. \end{aligned} \qquad \text{Q.E.D.}$$

An Implementation Example

We now discuss the processor architecture and the associated data flow among the processors implementing a 64×64 bit multiplication using Algorithm $M2$. We first divide the total number of processors into 127 bit planes, each for one bit position of the product W, excepting the most significant bit. The bit planes are numbered from 0 to 126, the ith bit plane corresponds to w_i and has x_i processors. For the most significant bit, we need to have a latch which is initially zero and is set whenever a carry 1 is propagated from the immediately preceding bit position. To describe the overall architecture, let us first concentrate on the 63rd bit plane containing 64 processors P_{jk}'s, $1 \leq j, k \leq 8$, arranged in an 8×8 array. The necessary interconnections for this bit plane are shown in Fig. 6.

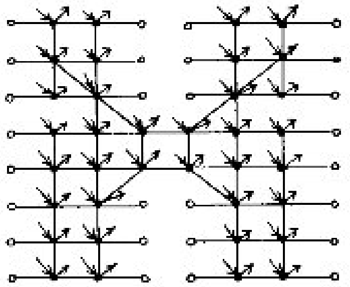

Fig. 6. Processor interconnection for 63rd bit plane. (The incoming arrows are from the corresponding positions of 62nd bit plane and the outgoing arrows are towards the corresponding positions of 64th bit plane.)

Now consider two processors $P_{j1,k1}$ and $P_{j2,k2}$ so that $P_{j1,k1}$ takes part upto q_1th parallel step of Algorithm $M2$ (either as a source of input data or as a generator of output data or both) and $P_{j2,k2}$ takes part up to the q_2th parallel step. We define an ordering between $P_{j1,k1}$ and $P_{j2,k2}$ as "$P_{j1,k1} \leq P_{j2,k2}$" if $q_1 \leq q_2$. Clearly "$\leq$" will be a total ordering on the set of all 64 processors. As we go to other bit planes on either side of the 63rd bit plane, the number of processors will be gradually reduced and we remove the processors from those positions in a given bit plane which correspond to the tail end of the above ordering

$$P_{j1,k1} \leq P_{j2,k2} \leq P_{j3,k3} \leq \cdots.$$

Finally, each processor in the ith plane is connected to the processors at the corresponding positions of the $(i-1)$th and $(i+1)$th planes, when they exist. This connection is necessary for carry propagation after each T operation from one bit plane to another.

Assuming that P_{jk}'s initially store all the relevant a_{jk}'s for

```
Step 1 :
        P_j2 <--- (a_j1, a_j2) }
        P_j3 <--- (a_j3, a_j4) }     for all j, 1 ≤ j ≤ 8
        P_j6 <--- (a_j5, a_j6) }
        P_j7 <--- (a_j7, a_j8) }
Step 2 :
        P_j3 <--- (S_j2, S_j3, C_j3) }     for all j, 1 ≤ j ≤ 8
        P_j5 <--- (S_j7, S_j6, C_j5) }
        P_2k <--- (C_1k, C_2k, C_3k) }
        P_5k <--- (C_4k, C_5k, C_6k) }     for k = 2, 7
        P_7k <--- (C_7k, C_8k)       }
Step 3 :
        P_2k <--- (S_1k, C_1k, S_2k) }
        P_4k <--- (S_3k, C_3k, S_4k) }
        P_5k <--- (C_4k, S_5k, C_5k) }     for k = 3, 6
        P_6k <--- (S_6k, C_6k, S_7k) }
        P_7k <--- (C_7k, S_8k, C_8k) }
        P_33 <--- (S_22, C_22, C_23)
        P_36 <--- (S_27, C_27, C_26)
        P_5k <--- (S_5k)             }     for k = 2, 7
        P_6k <--- (C_5k, S_7k, C_7k) }
Step 4 :
        P_3k <--- (S_2k, C_2k, S_3k) }
        P_4k <--- (S_4k, C_4k, C_3k) }     for k= 3, 6
        P_5k <--- (S_5k, C_5k, S_6k) }
        P_6k <--- (S_7k, C_7k, C_5k) }
        P_5k <--- (S_5k, S_6k, C_6k)       for k = 2, 7
Step 5 :
        P_43 <--- (S_43, C_43)             P_46 <--- (S_46, C_46)
        P_53 <--- (S_52, C_52, S_53)       P_56 <--- (S_57, C_57, S_56)
        P_44 <--- (S_33, C_33)             P_45 <--- (S_36, C_36)
        P_54 <--- (S_63, C_63, C_53)       P_55 <--- (S_66, C_66, C_56)
Step 6 :
        P_4k <--- (S_4k, C_4k)             for k = 3, 6
        P_4k <--- (S_4k, C_4k, C_5k)       for k = 4, 5
        P_54 <--- (S_53, C_53, S_54)
        P_55 <--- (S_56, C_56, C_55)
Step 7 :
        P_44 <--- (S_43, C_43, S_44)       P_45 <--- (S_46, C_46, S_45)
        P_54 <--- (S_54, C_54, C_44)       P_55 <--- (S_55, C_55, C_45)
Step 8 :
        P_44 <--- (S_44, C_44, S_54)
        P_55 <--- (S_55, C_55, C_54)
        P_45 <--- (S_45, C_45)
Step 9 :
        P_44 <--- (S_44, C_44, S_45)
        P_45 <--- (S_55, C_55, C_45)
Step 10 :
        P_44 <--- (S_44, C_44, S_45)
        P_45 <--- (C_45)
Step 11 :
        P_44 <--- (S_44, C_44, C_45)
```

Fig. 7. Data flow diagram.

the corresponding bit positions, the data flow can be described as in Fig. 7, where the symbol "$P_{j1,k1} \leftarrow (b_{j2,k2}, b_{j3,k3}, b_{j4,k4})$" means that the processor $P_{j1,k1}$ performs the addition of bits $b_{j2,k2}$, $b_{j3,k3}$, $b_{j4,k4}$ that are specified within the parentheses. The bits S_{jk}'s represent the sum bits generated at the (j, k)th position of the same bit plane while C_{jk}'s represent the carry bits generated at the (j, k)th position of the previous bit plane and subsequently propagated to the current plane. In this data flow diagram, we deviate a little from Algorithm $M2$, changing only the first parallel step which adds only pairs of bits instead of triplets of bits. This change accounts for one more parallel step than that in Algorithm $M2$.

The processor positions where additions are performed are marked by dots, while the rest have been denoted by circles in Fig. 6. By proper initialization of the processors at the dot positions, it is possible to do away with the processors at the circled positions, thus saving a lot of hardware cost and complexity. For large n, the number of processors (single-bit full adders) will be asymptotically equal to

$$N_P(n) = 2 \cdot 2[1+2+3+\cdots+(q-1)] + q$$

$$[\text{with } n = 2q]$$

$$= n(n+1)/2.$$

The AT^2 value of such a realization comes out to be $n^2(\log_2 n)^2$. Similar (although not identical) processor architecture and data flow diagram can be obtained for other values of n. It can be shown that even when $n = 256$, the running time is well below the bound of Theorem 2 and from VLSI chip design point of view one may not be interested in values of n beyond 256. When n is even larger, for the implementation to achieve the $O(\log_2 n)$ speed of Theorem 2, we may need a few communication links between some nonadjacent processors.

V. Two's Complement Multiplication

Both Algorithms $M1$ and $M2$ can be readily modified to perform the multiplication of two signed integers in two's complement form with a constant difference of time. Following the results in [10], we note that when U and V are given in two's complement form, their product W in two's complement

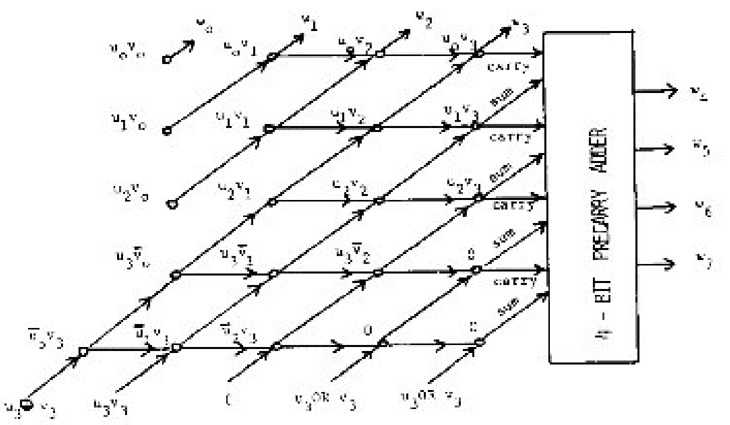

Fig. 8. Two's complement multiplication by $M1$.

form can be written as

$$W=\sum_{i=0}^{n-2}2^i\sum_{j=0}^{i}u_{i-j}v_j+\sum_{i=n-1}^{2n-4}2^i \cdot\left[\sum_{j=0}^{2n-i-4}u_{n-j-2}v_{i-(n-j-2)}+u_{n-1}\bar{v}_{i-n+1}+\bar{u}_{i-n+1}v_{n-1}\right]+2^{2n-3}[u_{n-1}\bar{v}_{n-2}+\bar{u}_{n-2}v_{n-1}]+2^{n-1}\sum_{i=0}^{n}2^ib_i, \qquad (7)$$

where

$$b_0=u_{n-1}\oplus v_{n-1}$$

$$b_1=u_{n-1}v_{n-1}$$

$$b_{n-1}=b_n=u_{n-1}\text{ OR }v_{n-1}$$

and all other b_i's are zero.

To perform two's complement multiplications, the following changes are to be done in Algorithm $M1$.

1) a_{ij}'s are to be redefined as follows:

$$a_{ij}=\begin{cases}u_iv_j, & \text{for } 0\le i<n-1,\ 0\le j\le n-1\\ u_{n-1}\bar{v}_j, & \text{for } i=n-1,\ 0\le j<n-1\\ \bar{u}_{j+1}v_{n-1}, & \text{for } i=n,\ -1\le j<n-2\end{cases}$$

and $a_{n-1,n-1}=a_{n,n-2}=a_{n,n-1}=0$

2) We need to redefine l_i as follows:

$$l_i=\begin{cases}1, & \text{for } i<n-1\\ i-n, & \text{otherwise.}\end{cases}$$

3) The following initializations are to be done before "loop1" in place of the present steps in Fig. 2.

```
for $i = 0$ to $(2n - 2)$ do in parallel
begin
  if $i < n - 1$ then
    begin
      $S_i^0 \leftarrow a_{i,0}$; $C_i^0 \leftarrow 0$;
    end
  else
    begin
      $S_i^{i-n-1} \leftarrow b_{i-n+1}$; $C_i^{i-n-1} \leftarrow 0$;
    end
end
compute $l_i$ and $n_i$;
```

4) The upper limit of the index i in the "for" statement at "loop1" should be $(2n - 1)$.

The corresponding changes in the implementation scheme of Algorithm $M1$ have been shown in Fig. 8 for $n = 4$. Here we have used one additional row of processor elements for the additional terms in (7). The initial values at each processor element have also been shown in the figure. After the initialization of the inputs, the product in two's complement form can be obtained in $(n + 2 + \lceil \log_2 n \rceil)$ units of time.

Algorithm $M2$ can also be modified in a similar manner by properly a) redefining x_i and the a_{ij}'s as above, and b) initializing the b_{ix}'s and w_{2n-1}. Changes in implementation of Algorithm $M2$ are also reflected in appropriately initializing the inputs to different processor elements in different bit planes. Due to the additional bits in $(n - 1)$th, nth, $(2n - 2)$th, and $(2n - 1)$th bit positions the number of parallel steps needed to perform the two's complement multiplication may be at most one more than that for unsigned multiplication. Moreover, to generate w_{2n-1} we need to have a processor (single-bit full adder) instead of a latch as in the case of unsigned multiplication.

VI. Conclusion

We have proposed two parallel algorithms for multiplication of two n-bit numbers; one requires $(n + \lceil \log_2 n \rceil)$ while the other approximately $3\lceil \log_2 n \rceil$ units of time. The first one is implementable on an SIMD systolic architecture, while the second can be implemented on an MIMD architecture. The AT^2 [8] measures of the two implementations are $O(n^4)$ and $O(n^2(\log_2 n)^2)$, respectively, where A is measured in terms of the number of single-bit full adder units and precarry logic modules. Both the algorithms have basically utilized the concept of column compression to reduce execution time. Since both the algorithms involve almost regular interconnections between only two types of cells, they are very suitable for single-chip VLSI implementation.

Acknowledgment

The authors are grateful for the valuable comments from the referees which greatly improved the presentation of the paper.

References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 2.* Reading, MA: Addison-Wesley, 1969.

[2] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 114–117, Feb. 1964.

[3] L. Dadda, "On parallel digital multipliers," *Alta Frequenza*, vol. 45, pp. 574–580, 1976.

[4] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia, "A compact high speed parallel multiplication scheme," *IEEE Trans. Comput.*, vol. C-26, pp. 948–957, Oct. 1977.

[5] J. P. Hayes, *Computer Architecture and Organization.* New York: McGraw-Hill, 1978.

[6] R. G. Hintz and D. P. Tate, "Control data STAR-100 processor design," in *Proc. 6th Annu. IEEE Comput. Soc. Int. Conf. (COMPCON 72)*, CA, Sept. 1972, pp. 1–4.

[7] S. Nakamura, "Algorithms for iterative array multiplication," *IEEE Trans. Comput.*, vol. C-35, pp. 713–719, Aug. 1986.

[8] C. D. Thompson, "A complexity theory for VLSI," Ph.D. dissertation, Dep. Comput. Sci. CMU, Aug. 1980.

[9] T. Herman, "Linear algorithms that are efficiently parallelized to time O(log n)," Tech. Rep. TR-85-17, Dept. Comput. Sci., Univ. Texas at Austin, Sept. 1985.

[10] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Comput.*, vol. C-22, pp. 1045–1059, Dec. 1973.

[11] N. Tagaki, H. Yassura, and S. Yajima, "High speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Trans. Comput.*, vol. C-34, pp. 789–796, Sept. 1985.

Bhabani P. Sinha (M'84) received the B.Sc. degree in physics, the B. Tech. and M. Tech. degrees in radiophysics and electronics, and the Ph.D. degree from the University of Calcutta, India, in 1970, 1973, 1975, and 1979, respectively.

He is currently visiting the Department of Computer Science, Southern Illinois University, Carbondale, IL, as an Associate Professor. He is on leave from Indian Statistical Institute, Calcutta, where he was an Associate Professor. His recent research interests include parallel algorithms and computer networks. During 1979–1981 he visited the Informatik Kolleg, GMD, Bonn, West Germany, as an Alexander von Humboldt Fellow.

Pradip K. Srimani (M'87) was born in Calcutta, India, in 1951. He received the B. Tech., M. Tech., and Ph.D. degrees in 1973, 1975, and 1978 from the Institute of Radiophysics and Electronics, University of Calcutta, India.

He has worked with Indian Statistical Institute, Calcutta, Gesselschaft fuer Mathematik und Datenverarbeitung, Bonn, West Germany and Indian Institute of Management, Calcutta. Since 1984 he has been at the Southern Illinois University, Carbondale, IL, where he is presently a Professor in the Department of Computer Science. His current research interests include parallel algorithms, fault-tolerant computing, and application of graph theory.