Recursive Implementation of Optimal Time VLSi Integer Multipliers

W. K. Luk
Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213
USA

J. E. Vuillemin
INRIA
Rocquencourt
78150 Le Chesnay
FRANCE

August 1983

Appeared in Proceedings, International Conference on VLSI,
Trondheim, August, 1983.

Research supported in part by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Recursive Implementation of Optimal Time VLSI Integer Multipliers

W. K. Luk[1]
Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213
USA

J. E. Vuillemin
INRIA
Rocquencourt
78150 Le Chesnay
FRANCE

Abstract: We present two algorithms suitable for VLSI implementation of very fast log$N$-time multipliers. The first one achieves provably optimal time (to within a constant factor) and-area-period^2 lower bounds. The second multiplier, although theoretical area-time suboptimal, is faster than all previously published ones for practical size, 16×16 and over. It admits a reglular and compact program generated layout, and its area requirements are well within the possibilities of current technologies. Prototypes have been built by a standard 2μm NMOS process, tested and are operational: multiplication of 32 bits can be performed in less than 200ns with a power consumption of about 1W.

Keywords: area-time tradeoff, arithmetic building block, hierarchical design, integer multiplication, modular multiplier, optimal time parallel multiplier, program generated layout, VLSI algorithm, VLSI complexity

1. Introduction

Designing fast binary N-bit integer multipliers has long been of great theoretical and practical interest for both electrical engineers and computer scientists. With the technological progress embedded in VLSI, the problem of finding good layouts for monolithic MOS multipliers has been gaining renewed attention.

1.1. Practical multipliers

Practitioners favor *shift-and-add* serial/parallel multipliers (see e.g. [13]) requiring minimal area $O(N)$ and N clock cycles per multiply, or the *un-folded* parallel/parallel version of the same algorithm, with area $O(N^2)$ and time $O(N)$, executed within a single clock cycle, as described e.g. in [15].

[1]Research supported in part by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

1.2. Area-time trade-offs for multiplication

Theoreticians, on the other side, have been establishing relationships between the following measures of N-bit multipliers layout efficiency:

- the *area* A of the layout (in nanoacres);
- the *time* T (in nanoseconds) required for completing an operation;
- the *period* P (in clock cycles) separating two successive pipelined multiplications through the circuit; it is inversely related to the throughput (i.e. bandwidth, data rate).

Intrinsic *combinatorial* limits on the efficiency of multipliers have been discovered and expressed as trade-offs between A and T or P: for N-bit multipliers, [3] shows that $AT^2 \geq N^2$ and [20] that $A \geq mN + w(N/P)^2$ where m is the size of the unit memory cell, and w the minimal wire width.

These results are obtained under either the constant-delay [19] or the capacitive [16] models for the timing behavior of MOS circuits:

- in the *constant-delay* model, the time delay in transmitting a signal from the output of a logic gate to the input of another gate through the wire connecting them is independent of the length of the wire;
- in the *capacitive* model, the switching time of a gate is proportional to its output wire capacitance.

For technologies such as magnetic bubbles, the speed of signal propagation along wires dominates switching times. A lower bound of $T = \Omega(\sqrt{N})$ has been obtained by [6] for this case, with which the present paper is not concerned.

1.3. Classes of optimal multipliers

1.3.1. Area-time optimal multipliers

Attempts have been made at bridging the gap between existing *practical* designs: $A = O(N)$, $T = O(N)$, or even $A = O(N^2)$, $T = O(N)$; and *theoretical* limits of area-time efficient multipliers: (1) $A = O(N)$, $T = O(\sqrt{N})$, or (2) $A = O(N^2/\log^2 N)$, $T = O(\log N)$, or (3) $A = O(N^2)$, $P = O(1)$.

1.3.2. Optimal multipliers based on Fourier transform

Theoretically efficient multipliers have been proposed: [18] constructs a class of $AT^2 = O(N^2)$ optimal multipliers, where time T can be chosen in the range $O(\log^2 N) \leq T \leq O(\sqrt{N})$; another theoretically optimal design, with area $A = O(N)$ and time $T = O(\sqrt{N})$, is described in [17]. These multipliers, as well as another near-optimal one by [3] all use *Fourier transform* (see [10] for an introduction to the relevance of such transform in computing integer and polynomial products).

As a consequence, although they are asymptotically optimal, such multipliers do not seem competitive for medium size N, say $N=16$ or 32 bits.

1.3.3. Recursive divide and conquer multipliers

Another class of simpler, *recursively* defined multipliers has been proposed: [1] gives a theoretical construction achieving the optimal $A=O(N^2/P^2)$ bound with time $T=O(log^2N)$. [11] independently proposes a complete VLSI layout for a fast (near-optimal) recursive parallel multiplier having time $T=O(log^2N)$ and area $A=O(N^2log^2N)$.

1.3.4. Fast and time-optimal multipliers

Various attempts have been made to speed up multiplication. *Wallace* tree [23] and *Dadda* counting [7, 5] schemes attain the $O(logN)$ time lower bound for integer multiplication. The irregular interconnection makes Wallace tree infeasible for laying out single chip of arbitrary bit size. Furthermore, they are *not* modular in nature, so they can hardly be extended to form regular large multiplier arrays.

Other methods such as the *multiple shift* (e.g. [14]) and the high-radix *string recoding* (e.g. [2]) try to increase the average shift and decrease the average number of additions per multiply. But they require additional hardware and can only improve the $O(N)$ multiply time by a constant factor, i.e. they are not asymptotically optimal and will *not* be competitive for reasonably large size multipliers, say ≥ 16 bits.

To conclude, we see that while the theoretical aspects of designing time-optimal and/or area-time optimal monolithic multipliers of arbitrary bit size were fairly well understood by 1982, a wide gap still exists between theory and practice.

1.4. Contribution

1.4.1. Regular layout for optimal time integer multiplication

In this work, we introduce *two* interesting classes of VLSI multipliers. Both operate in *optimal* $O(logN)$ time, and can be *pipe-lined* in a straight-forward manner to achieve period $P=1$, i.e. one multiplication per clock cycle.

Our first multiplier, named 3M, has area $A=O(N^2)$, which is of *theoretical* interest since it achieves the optimal AP^2 (or AP^2T^2) measure. This is obtained by speeding up the schemes of [1, 11] through systematic use of carry-save representation for intermediate results.

Our second multiplier, named 2M, has asymptotic area $O(N^2logN)$, but it is *faster* than 3M. It is also smaller for reasonable values of N, say $N<1024$. It implements an algorithm proposed by [21], and we

demonstrate here its practical interest.

1.4.2. Layout generation and performance

Layouts of a very regular and fast version of 2M are completely generated by a *program* for arbitrary size parameter. The program takes into account all the *idiosyncrasies* of a given technology, and tailors the circuits to specific separable *test* strategies.

Analysis and simulation indicate that 2M is faster than the existing monolithic multipliers (for size $\geq$16 bits) in terms of the number of stages of carry-save adders:

- The layout occupies an area slightly greater than that of a multiplier built from the straightforward unfolded carry-save adding scheme (e.g. [8]); it is already *2.5* times faster in its 16-bit version.

- Further, 2M is *twice* faster than a 32-bit multiplier which uses the modified Booth's recoding as reported in [15], within less than twice the area. It is indeed competitive in speed with the known fast schemes of Wallace trees.

1.4.3. Test results

8, 16, 32-bit experimental prototypes of 2M were fabricated and tested, all are functioning. Experimental data confirms that the delay along our longest wires can be kept under control, thus maintaining roughly the *logN*-time performance for medium size (16~32 bits) multipliers. Using a 3μm NMOS process, speed for the 8, 16, 32 (2μm)-bit chips are respectively 120, 160 and 220 ns with a power consumption of about 180, 720, 1280 mW.

Although electrical problems in the design of our basic cells prevent our current design to exactly match the theoretical speed, they have been corrected and a second version of our design is currently being fabricated.

To conclude, our optimal and modular scheme proposes a feasible way to handle medium size to ultra large size integer multipliers, when absolute speed is the dominant design factor.

1.5. Outline

Section 2 presents various recursive VLSI algorithms for fast integer multiplication. Section 3 introduces various layout strategies for such recursive algorithms, together with their expressions in a specific geometric description language. Section 4 discusses performance evaluation, verification, simulation and test results.

2. The VLSI algorithms and complexity

We present three simple recursive algorithms for computing integer products in $logN$ time, within reasonably small silicon area. We call these respectively the 4-multiplication ($4M$) version, the 3-multiplication ($3M$) version and the 2-multiplication ($2M$) version.

2.1. Reduction of multiplication to addition

Multiplication is *recursively* divided into a number of multiplications of smaller size followed by *additions* combining the intermediate results. Thus multiplication is recursively reduced to a sequence of additions until arriving at the termination state of the recursion (e.g. a trivial one bit multiplication).

Good adder choice is crucial in the design of such fast multipliers. In all cases, we keep intermediate results coded in *carry-save* form, so that additions of arbitrary length operands can be performed in constant time (independent of the length of such operands) by carry-save adders. See [21] for a more elaborate presentation of carry-save arithmetic. The final result in carry-save form can be converted back to ordinary binary representation via a fast carry-look ahead adder in $logN$ time, such as the one proposed by [4, 22].

2.2. 4-multiplication (4M) version

Let X, Y be two N-bit integers and P be their product; X_0 and X_1 be respectively the least- and most-significant halves of X (similarly for Y); and $P_{ij} = X_i \cdot Y_j$ for $i,j = 0,1$.

The 4M version follows from the divide and conquer paradigm as given by the following equations.

$$P = X \cdot Y = (2^{N/2} \cdot X_1 + X_0) \cdot (2^{N/2} \cdot Y_1 + Y_0) = 2^N \cdot P_{11} + 2^{N/2} \cdot (P_{10} + P_{01}) + P_{00}$$

A complete layout of this version can be found in [11] which occupies $O(N^2 log^2 N)$ area, and runs in $O(log^2 N)$ time. The latter can be improved to being optimal time $O(logN)$ by using carry-save adders. A recursive floorplan illustrating how to map this algorithm onto silicon is shown in Figure 2-1.

2.3. 3-multiplication (3M) version

The second 3M version can be described by the following well known recurrence equations [10].

$$U = (X_1 + X_0) \cdot (Y_1 + Y_0)$$

$$V = X_1 \cdot Y_1$$

$$W = X_0 \cdot Y_0$$

$$P = X \cdot Y = V \cdot 2^N + (U - V - W) \cdot 2^{N/2} + W$$

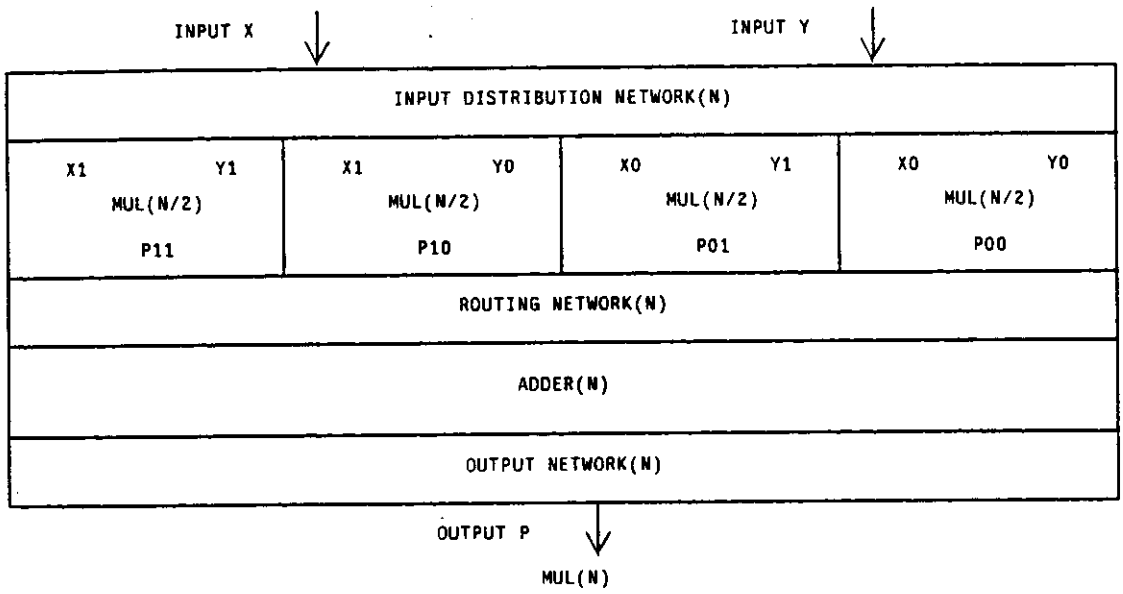


Figure 2-1: Recursive floorplan of 4M multiplier

This version can be embedded onto silicon within an area $O(N^2)$ and runs in $O(logN)$ time. It can be further pipelined at clock rate (period $P=1$). So this algorithm is indeed *optimal* in the sense of running *time* and AP^2 (or AP^2T^2) measure. A recursively defined floorplan implementing this algorithm is shown in Figure 2-2.

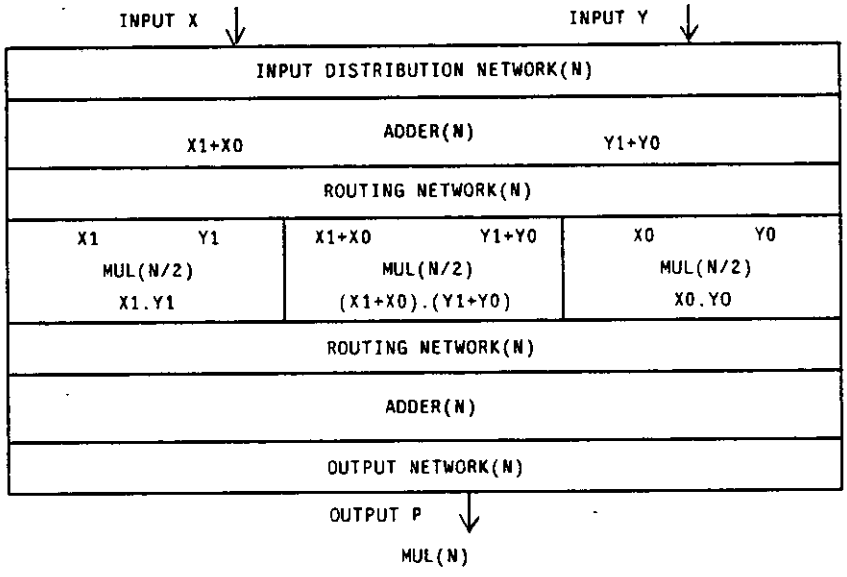


Figure 2-2: Recursive floorplan of 3M multiplier

2.4. 2-multiplication (2M) version

The 3M algorithm, although theoretically optimal, leads to less regular structures (routing and bit testing for performing subtraction). By trading a small factor of $logN$ in the asymptotic area, the following algorithm gives a feasible and nicely regular silicon layout.

[21] proposes a 2-multiplication scheme to do fast multiplication. The algorithm simply recurses on one of the two operands (say X), as indicated below. Figure 2-3 shows how 2M performs the product of Y and $X=X_0+2X_1+4X_2+8X_3$.

$$P=X\cdot Y=(2^{N/2}\cdot X_1+X_0)\cdot Y=2^{N/2}\cdot X_1\cdot Y+X_0\cdot Y$$

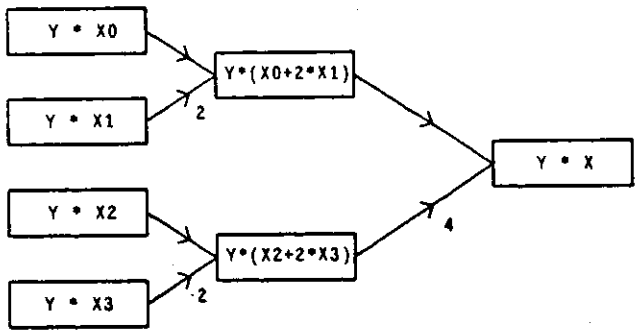


Figure 2-3: Product of Y and $X=X_0+2X_1+4X_2+8X_3$ by algorithm 2M

This version requires an area of $O(NMlogN)$ and time $O(logN)$, M being the length of Y. The area is a factor $logN$ away from being AP^2 optimal.

2.5. Linear array of bit-convolvers (LABC)

For layout purposes, we found convenient to use an *orthogonal* view: the 2M multiplier consists of identical columns of *bit-convolver*, forming a *linear array*. Column k receives as input the bit-wise products $X_i\cdot Y_j$ for $i+j=k$, together with a carry vector from column $k-1$. It sums up all these input bits, as well as the carry vector, to generate the carry vector for column $k+1$ and a pair of sum bits. The bit-summing algorithm used is described as:

Algorithm BITADD:

Input: N,S: integer (with $S=N/2-1$); $A=\{a_0,\ldots,a_{N-1}\}$, $C_1=\{c_{1,0},\ldots,c_{1,S-1}\}$, $C_2=\{c_{2,0},\ldots,c_{2,S-1}\}$: bit-sequence;

Output: s_1,s_2: bit; $D_1=\{d_{1,0},\ldots,d_{1,S-1}\}$, $D_2=\{d_{2,0},\ldots,d_{2,S-1}\}$: bit-sequence;

Comment: A is a N-sequence of input bits. C_1 and C_2 are two S-sequences of carry-in bits. s_1 and s_2 are a pair of sum bits. D_1 and D_2 are two S-sequences of carry-out bits. All bits of A, C_1, C_2, s_1

and s_2 have the same weight; bits of D_1 and D_2 have twice that of the former. The following *invariant* expresses that ***BITADD*** adds up $A+C_1+C_2$, to yield a sum (s_1,s_2) and carries D_1, D_2:

$$S_A+S_{C_1}+S_{C_2}=s_1+s_2+2(S_{D_1}+S_{D_2})$$

where $S_A=\sum_{i=0}^{N-1} a_i$, $S_{C_1}=\sum_{i=0}^{S-1} c_{1,i}$ and similarly for C_2, D_1 and D_2.

Method: Denote by subscripts 0 and 1 respectively the first and second halves of a sequence, and *mid* its middle element (assuming there are odd numbers of elements). ***BITADD***(A,C_1,C_2) recursively divides the bit summing according to:

$$\begin{aligned}
&\textbf{\textit{BITADD}}(A,C_1,C_2)\\
&=\textbf{\textit{BITADD}}(A_0,C_{1,0},C_{2,0})+\textbf{\textit{BITADD}}(A_1,C_{1,1},C_{2,1})+c_{1,mid}+c_{2,mid}\\
&=(s_{1,0},s_{2,0},D_{1,0},D_{2,0})+(s_{1,1},s_{2,1},D_{1,1},D_{2,1})+c_{1,mid}+c_{2,mid}\\
&=(s_{1,0}+s_{2,0}+s_{1,1}+s_{2,1}+c_{1,mid}+c_{2,mid})+(D_{1,0}+D_{1,1})+(D_{2,0}+D_{2,1})
\end{aligned}$$

The final results s_1, s_2, D_1, D_2 are computed as follows:

$$(s_1,s_2;d_{1,mid},d_{2,mid}):=\textit{CSA4}(s_{1,0},s_{2,0},s_{1,1},s_{2,1},c_{1,mid},c_{2,mid})$$

$$D_1:=\{D_{1,0}\ d_{1,mid}\ D_{1,1}\}$$

$$D_2:=\{D_{2,0}\ d_{2,mid}\ D_{2,1}\}$$

Here, ***CSA4*** is a 4-bit carry-save adder module which takes the 4 bits $s_{1,0}$, $s_{2,0}$, $s_{1,1}$, $s_{2,1}$ as inputs, and outputs the 2 bits s_1 and s_2. $c_{1,mid}$ and $c_{2,mid}$ are the two carry-in bits, $d_{1,mid}$ and $d_{2,mid}$ are the two carry-out bits. An *invariant* of ***CSA4*** is

$$s_{1,0}+s_{2,0}+s_{1,1}+s_{2,1}+c_{1,mid}+c_{2,mid}=(s_1+s_2)+2(d_{1,mid}+d_{2,mid})$$

The recursion terminates at $N=4$, a ***CSA4*** carry-save addition.

2.6. Area, time, period complexity and optimality

A detailed analysis of the algorithms can be found in [12]. We summarize the complexity and optimality results on area, time and period in Table 2-1.

version	area	time	period	AP^2	AP^2T^2	remark
lowerbound	N^2	$logN$	1	N^2	N^2log^2N	-
4M	N^2log^2N	$logN$	1	N^2log^2N	N^2log^4N	time-optimal
3M	N^2	$logN$	1	N^2	N^2log^2N	time, AP^2 and AP^2T^2 optimal
2M, LABC	$MNlogN$	$logN$	1	$MNlogN$	$MNlog^3N$	time-optimal and regular layout

Table 2-1: Area, time, period complexity and optimality

All the above algorithms achieve the *absolute* speed (to within a constant factor) for integer multiplication.

The 3M version is indeed AP^2 (or AP^2T^2) *optimal*; and the others are only of logarithmic factors away from being AT^2 and AP^2 optimal.

3. Layout strategies and compiling onto silicon

3.1. Selection of algorithms

Our design objective was to produce the fastest known multiplier, namely algorithm 2M. It also turns out to be more regular, and in fact smaller for practical values than both 4M and 3M.

The 2M algorithm can be laid out recursively, following directly its definition. To form a $M \times N$ multiplier, we *sandwich* a $(M+N)$-bit carry-save adder between two recursively constructed M by $N/2$ multipliers, as shown in Figure 3-1, a recursively generated floorplan and global wiring diagram of its bit-slice version.

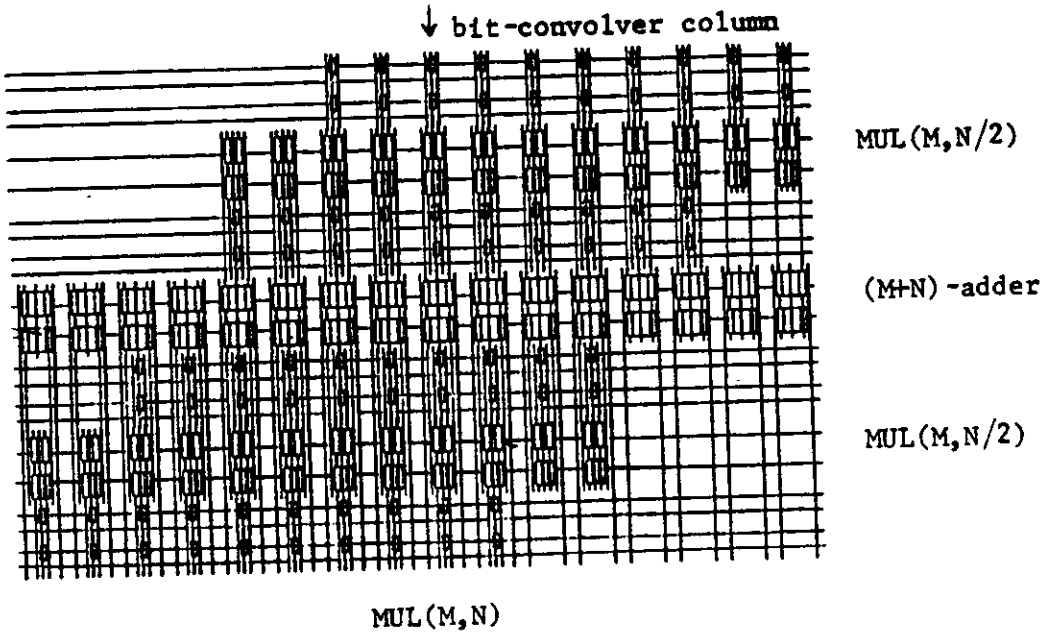


Figure 3-1: Floorplan and global wiring diagram of a bit-slice 2M multiplier

It proves equivalent and possibly more convenient to build such a $M \times N$ multiplier by a linear array of bit-convolvers (*LABC*). Each bit convolver sums the N bits of each column and takes into account how the carries propagate across columns. A generic example is provided by the center column in Figure 3-1. A LABC with proper termination for the carries at the far ends becomes the fast 2M multiplier.

Since the bit-convolver explicitly accounts for the *carries propagation*, it can be directly applied to generate

fast modular multipliers, the basic building block for very large size multiplier array (say 1000 by 1000 bits). Bearing this in mind for future extension, we have macro-generated prototypes multipliers of arbitrary size based on the LABC version.

3.2. Functional partitioning

3.2.1. Linear array of binary trees

The ***BITADD*** algorithm suggests that a *binary tree* is appropriate for summing up the bits in each column. Each *leaf* of the binary tree is a pair of AND gates producing the required bit-wise product for the two vectors X and Y, we call it ***MU2***. The outputs from the leaf nodes (MU2) are directly in carry-save form ready as inputs to the carry-save adder ***CSA4***, a 4-bit carry-save adder. All the *non-leaf* nodes of the binary tree are CSA4's. The *carries* just pass from one side of a binary tree column to the other side. A few columns of this binary tree array (bit-convolver array) are shown in Figure 3-2, decomposition into the basic cells CSA4 and MU2, and further into AND gates and carry-save adders CSA are also included. One can see the regular and nice interconnecting structure of such array.

3.2.2. Layout for binary trees

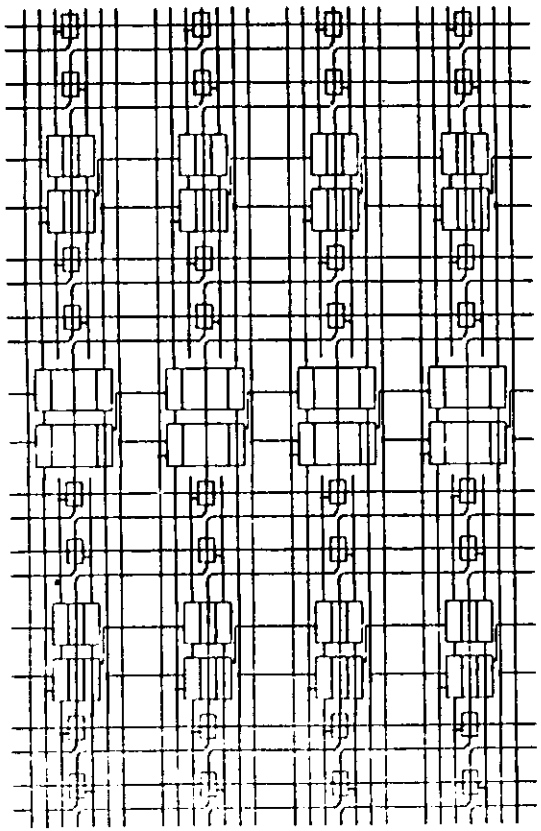
There are a number of ways to layout a binary tree, namely a *V-tree* (Figure 3-3), a *L-tree* (Figure 3-2), a *H-tree* (Figure 3-4) and a *3D-tree*. Their layout generators are:

vtree := λ *N*. if $N=1$ then **leaf** else (*JY* **root**[N] (*JX* (*vtree* [*N*/2]) (*vtree* [*N*/2])))
ltree := λ *N*. if $N=1$ then **leaf** else (*JY* (*ltree* [*N*/2]) **root**[N] (*ltree* [*N*/2]))
htree := λ *N*. if $N=1$ then **leaf** else (*JY* (*ROT*+ (*htree* [*N*/2])) **root**[N] (*ROT*− (*htree* [*N*/2])))
3dtree := λ *N*. if $N=1$ then **leaf** else (*JZ* **root**[N] (*JY* (*3dtree* [*N*/2]) (*3dtree* [*N*/2])))

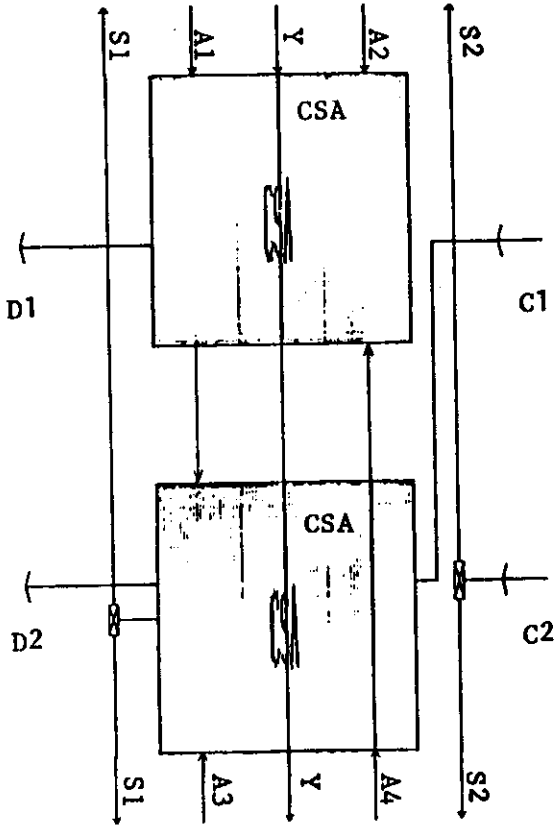
where *JX*, *JY*, *JZ*, *ROT*+, *ROT*− are geometric operators; their semantics can be found in Section 3.4. Two basic cells **leaf** and **root** are sufficient to generate an arbitrary size layout. The parameter N in the **root** cell is used to parameterize the necessary changes such as transistor dimension and driver size, or required by other routines when doing the recursion.

Although it occupies *minimum* area $O(N)$, the H-version do not seem useful since the I/O's are not located on the boundary of the layout. We cannot use the 3-D version because of the planar constraint in the present day technology; it nevertheless proves a convenient conceptual tool, since we can automatically transform 3-D layout into either V or L-trees [21].

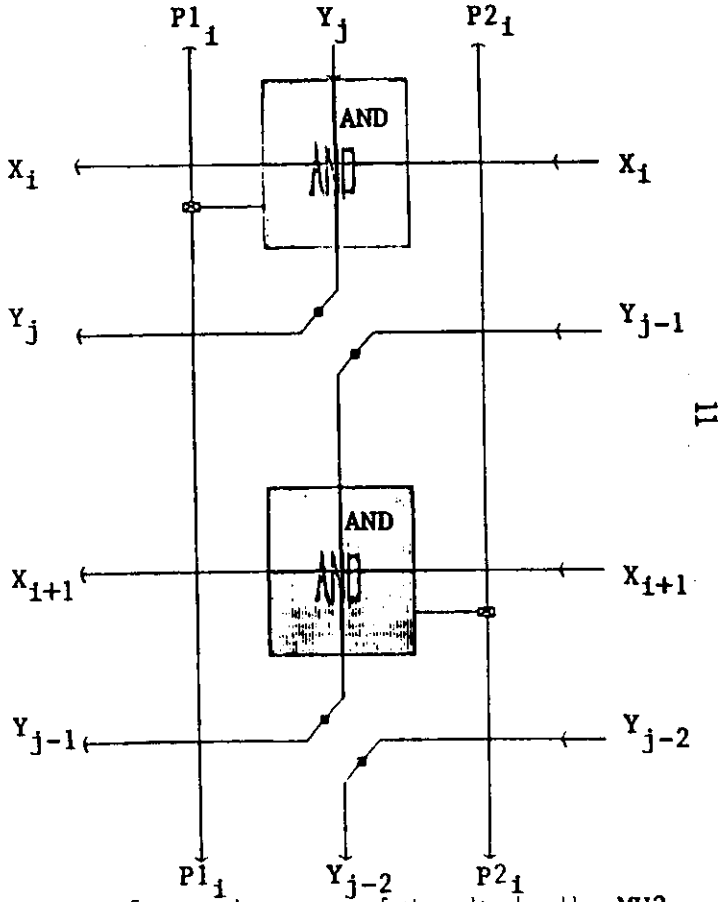
Both the V or L-versions are relevant to our design. For small or medium size layouts where the propagation delay of a signal through long wires may be considered as a constant, the L-version is a *better* choice for it takes an area $N(a+b \cdot logN)$, where b is a small constant (of the order of a few λ's), being much smaller than a, the tree width. When the capacitive effect becomes dominating, *signal amplification* scheme as



4 slots of a 8-bit bit-convolver



Functional structure of the addcell CSA4



Functional structure of the multiplycell MU2

Figure 3-2: Linear array of bit-convolver and layout of L-binary tree

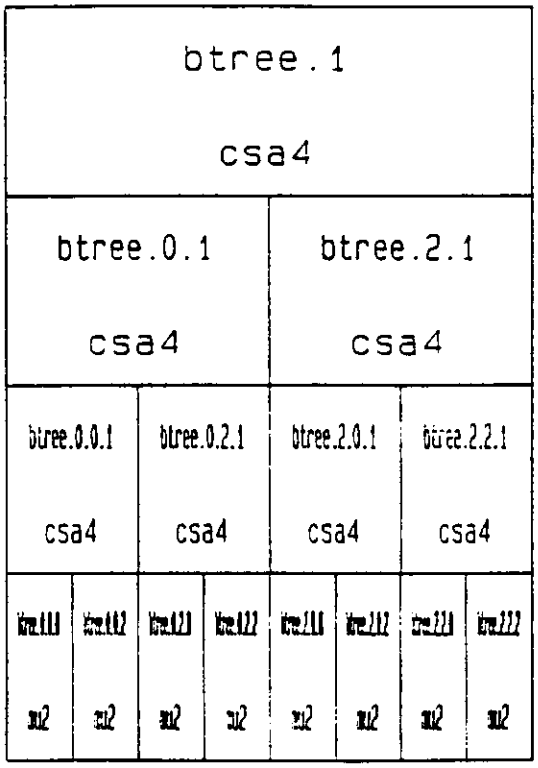


Figure 3-3: Layout of V-binary tree

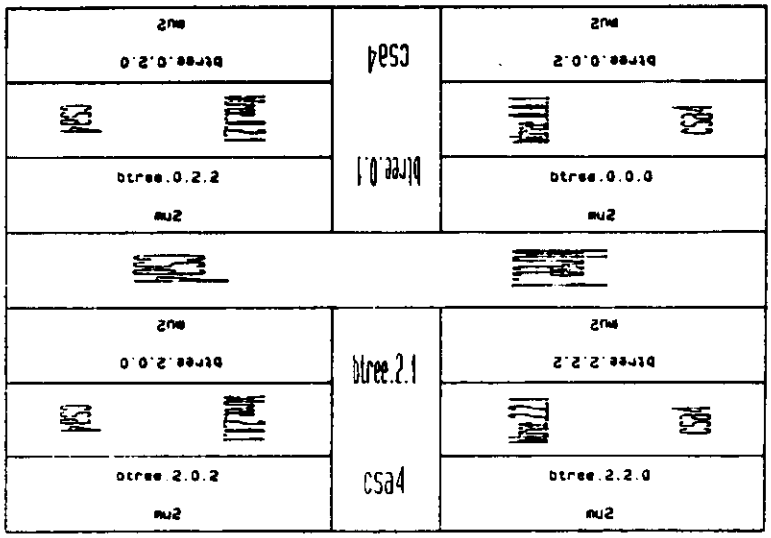


Figure 3-4: Layout of H-binary tree

proposed by [22] in their adder design might be helpful to maintain the *logN*-time performance. V-tree seem to be a reasonable candidate since it provides proportionally larger space for the cells that need to drive longer wires. However, the V-version requires much more actual area than the L-version.

Electrical simulation shows that under the present 3μm n-MOS technology, *distributed RC* and *capacitive* effect is negligible (< 1ns) for wires shorter than 100μm. The equivalent delay along the *longest* path (about 1cm) in our design is about 7ns and 70ns for falling and rising logic transitions. It is unacceptable for *rising* edge delay. However, by using *precharge* techniques, the worst case wire delay can be kept within 7ns which is comparable with the delay in a CSA4 cell.

Based on this analysis, we decided to implement the L-layout, much smaller in area than its V counterpart, without affecting significantly the overall speed performance.

3.3. Logic and circuit descriptions of basic cells

Only two basic cells, namely CSA4 and MU2 are needed to implement the algorithms. We now go down the design hierarchy, from high level functional description to the *logical* and *circuit* specifications. Designing MU2 is straight-forward, but CSA4 requires more attention. Electrical and switch level *simulations* have been extensively used to make critical design decision and verify circuit correctness.

3.3.1. Carry-save adder CSA4

The CSA4 takes 4 input bits and a pair of carry-in bits, generates a pair of sum bits and a pair of carry-out bits as pointed out in the ***BITADD*** algorithm. It is made up by interconnecting two standard carry-save adder cells (see Figure 3-2). Each of these cells is basically that (precharged version) proposed in [15] which has $\approx$ 15ns delay within reasonable area. Its logical description $(C,S) := CSA(a,b,c)$ is:

$$(C,S) := \text{ if } odd(a+b) \text{ then } (c,\neg c) \text{ else } (a,c)$$

where a, b, c are the 3 inputs, and S and C are respectively the sum and carry outputs. Its MOS circuit diagram is shown in Figure 3-5.

Alternative implementations of a carry-save adder $(C,S) := CSA(a,b,c)$ with $a+b+c=2C+S$ have been considered:

- Using two level NOR gates is fast ($\approx$ 10ns) but uses much area:

$$S = NOR(NOR(a,b,c),NOR(a,\neg b,\neg c),NOR(\neg a,b,\neg c),NOR(\neg a,\neg b,c))$$

$$C = NOR(NOR(a,b),NOR(b,c),NOR(c,a))$$

- The following version is small, but slow ($delay_S$ = 20~50ns, $delay_C$ = 20~35ns):

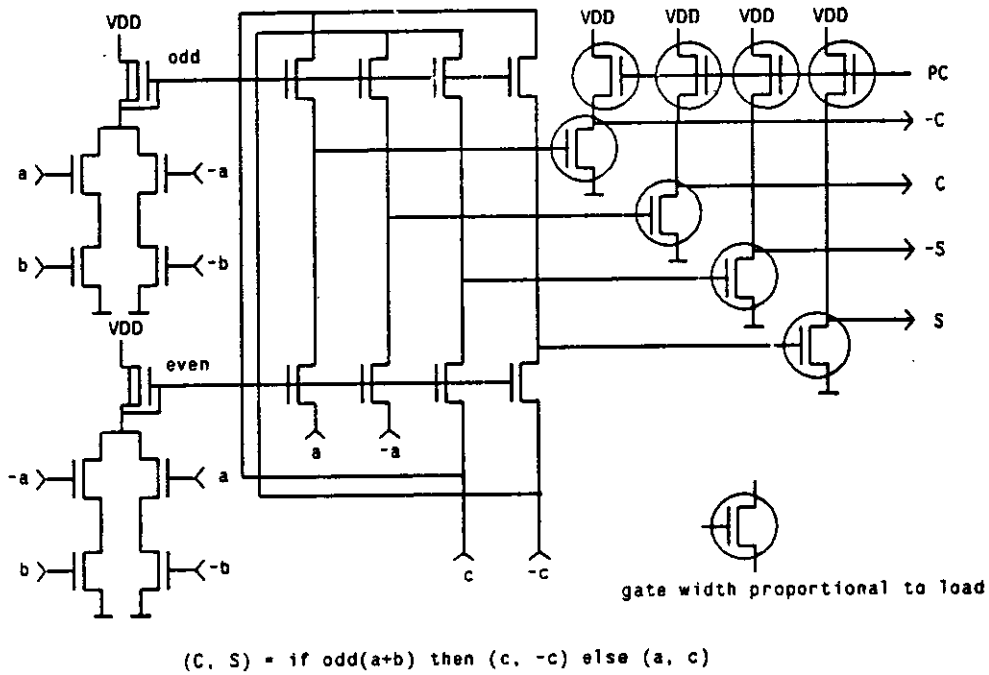


Figure 3-5: CSA MOS circuit diagram

$$S = c \cdot (a \cdot b + \neg C) + (a + b) \cdot \neg C$$

$$C = a \cdot b + c \cdot (a + b)$$

3.3.2. Termination cell MU2

In order to terminate the recursion, the outputs of the leaf cells MU2 should be ready in carry-save form to feed their parent node CSA4 in the binary tree. The logic function of MU2 at position (i, j) of the array is given by $[P_1,P_2]_{ij}=[AND(X_i,Y_j),AND(X_{i+1},Y_{j-1})]$, where P_1 and P_2 are the output bits and X_i, X_{i+1}, Y_j and Y_{j-1} are the input bits at the corresponding positions in the array. MU2 is made up of two precharged AND gates (see Figure 3-2).

3.4. Layout generation

Rather than just producing the layout of one prototype multiplier, we wrote a program ***multiplier(M N)*** which generates the layout of a $M \times N$ multiplier, given the parameters M and N. The mask description target language *LUCIFER* is an extension of LISP developed as part of the design aids at INRIA. The language permits hierarchical and recursive specification of a complete circuit; and also provides a *homogeneous* medium (LISP S-expressions) for handling both the mask descriptions (data) and the programs that manipulate them.

LUCIFER specifies mask descriptions as a structured collection of elementary *rectangles*, assembled by *geometric operators*. A simplified BNF for the language is:

$circuit ::= node$

$node ::= (OP\ node_1\ node_2 \ldots\ node_n)$

$node ::= rectangle$

$rectangle ::= (color\ x\ y\ xdim\ ydim)$

$color ::= ND \mid NP \mid NC \mid NM \mid NI \mid NB \mid NG$

$x,y \in \mathbb{Z},\ xdim,ydim \in \mathbb{N}$

Sample geometric operators *OP* are:

- *JX* juxtaposes a list of nodes along the x direction;
- *JY* juxtaposes a list of nodes along the y direction;
- *JZ* juxtaposes a list of nodes along the z direction (3-dimensional view);
- *ARRAY N* produces *N* juxtaposed copies of a node;
- *ROT+* rotates clockwise by 90 degrees, (*ROT−* by −90 degrees);
- *UN* superposes a list of nodes.

In LUCIFER, our multiplier *generator* has the following structure:

```
BIT-CONVOLVER := lambda [N]
                   if N = 2
                   then (BASIC-CELL MU2)
                   else (JY (BIT-CONVOLVER (div N 2))
                            (BASIC-CELL CSA4 N)
                            (BIT-CONVOLVER (div N 2)))
BIT-CONVOLVER-ARRAY := lambda [M N]
                         (ARRAY M (BIT-CONVOLVER N))
```

The basic cells CSA4 and MU2 are laid out with the help of a graphic editor. Further CSA4 is then parameterized to handle size change and cell alignment required by the recursion. Due to the regular structure of the binary tree, the interconnection (routing) between cells is generated by simple recursive programs. Routing is done based on an *'error-free'* design rule. Details about the complete layout generator can be found in [12]. A complete *program* generated layout of 16×16 bit-convolver is shown in Figure 3-6.

Some general remarks about the program generated layout are:

- The circuits are *parameterized* so that necessary features are customized according to the input size; e.g. the *VDD* and *GROUND* lines widths are adjusted automatically so as to avoid metal migration problems, no matter what input sizes *M* and *N* are.

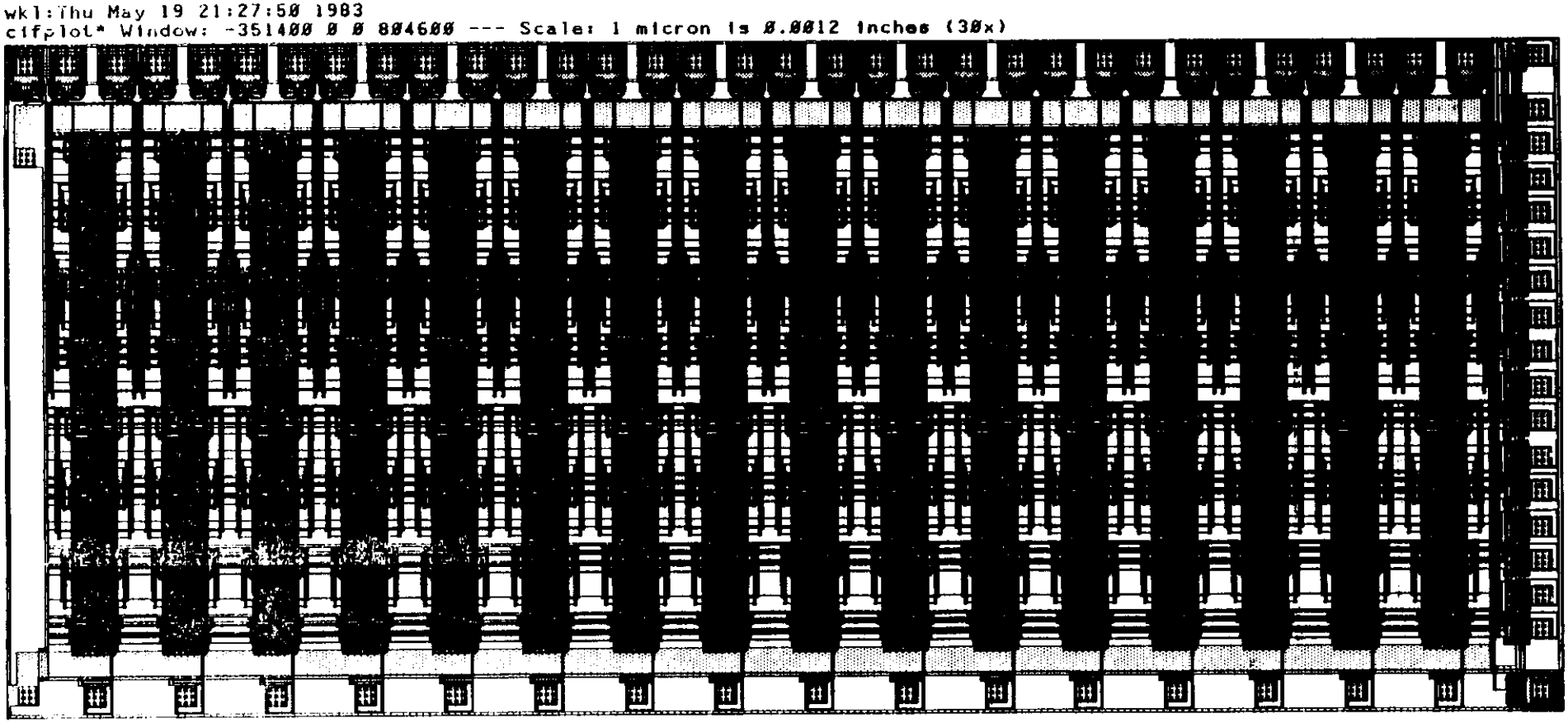


Figure 3-6: Cifplot of a 16×16 bit-convolver

- It is easy to accommodate such a geometry generator to a (non-trivial) *scaling* down of feature sizes; the increase in wiring area turns out to provide just enough room for the input and output pads, which become relatively bigger in scaled technologies.

- Using a 3μm process, we can integrate into a 7mm by 7mm substrate an 8 by 16 bit-convolver array. Little attention has been paid to cells area optimization, and we estimate that a 50% gain could be achieved. For a 1μm process, actual figures indicate that we can integrate, into the same area, a 64-bit LABC.

- A program generating arbitrary size circuits has proved extremely valuable for *testing*. We actually process various experimental prototypes. A small 2×4 array with all external connections present to maximize probing facilities, for testing the correct logic behavior of gates CSA4 and MU2, and their interconnections. A small 3×32 strip, with all inputs tied to logic 0; it provides for a simple test of the actual worst-case speed of 32-bit multiplication. Finally, are the 8, 16, 32-bit LABC's. All these circuits are generated by the same program: the *mode* of execution (logic test, speed test, multiplier) is passed as parameter (together with the sizes of X and Y) to select boundary connections for each case.

4. Performance evaluation, verification and testing

We pay special attention to the time and area performance of our design. It is evaluated analytically with other known fast multipliers. During the various stages of the design, extensive simulations and verifications have been performed. Prototype chips were fabricated and tested to be successfully operational.

4.1. Performance evaluation

4.1.1. Timing analysis

The overall multiply time $T_{mult}(M,N)$ depends on M, N, D_{MU2}, D_{CSA4} (the delay of MU2, CSA4) and the time spent in charging long wires. Note that $D_{CSA4} \leq 2D_{CSA}$, where D_{CSA} is the delay of a standard carry-save adder, and $D_{MU2} = D_{AND}$, the delay of an AND gate:

$$T_{mult}(M,N) = T_{mult}(M,N/2) + D_{CSA4} + D_w(N)$$

$$T_{mult}(M,2) = D_{MU2} + D_w(2)$$

where $D_w(N)$ is the *worst-case* delay along the longest wire of a N-bit multiplier. It is a function of N, and simulation results show that the dominant term in this function grows *linearly* with N: $D_w(N) = a + bN + o(N)$. We can reduce this delay to a constant D_{w_0}, by increasing the width of the gate to amplify the signals in proportion to the output loads. This leads to

$$T_{mult}(M,N) = (D_{CSA4} + D_{w_0})log(N/2) + D_{MU2} + D_w(2)$$

As pointed out earlier, the V-tree is a suitable layout for this scheme. A rough upper bound on the speed of our L-tree layout is given by:

$$D_w(N) \leq D_w(N_{max})$$

$$T_{mult}(N) \leq (D_{CSA4} + D_w(N_{max}))log(N/2) + D_{MU2} + D_w(2)$$

For our target technology using a 3μm process: longest wire = 1cm, $N_{max} = 16$, $D_{CSA4} = 2D_{CSA} = 25ns$, $D_{MU2} = 5ns$, $D_w(2) < 1ns$, $D_w(N_{max}) < 7ns$. So

$$T_{mult}(N) < 32log(N/2) + 5 \quad ns$$

$$T_{mult}(16) < 101ns \quad and \quad T_{mult}(32) < 133ns$$

4.1.2. Comparison with other fast multipliers

We compare the *time* performance and *area* required of our layout with the unfold sequential multiplier, Booth's encoding multiplier [15] and Wallace tree in terms of carry-save adder parameters. Results are given in Table 4-1.

Consider $M \times N$ multipliers, $A_{mult}(M,N)$ denote the multiplier area which is a multiple of the area required by a basic carry-save adder. Let D_{BOOTH} be the delay of the encoding logic used in a Booth's multiplier (note $D_{BOOTH} > D_{AND}$) and $D_w = D_w(N_{max})$.

multiplier	$T_{mult}(M,N)$	$A_{mult}(M,N)$
bit-convolver	$log(N/2)(2D_{CSA} + D_w) + D_{AND}$	$(M + a\ logN)(N-2)$
Wallace tree	$1.7logN(D_{CSA} + D_w) + D_{AND}$	$MN\ logN$
Booth's encoding	$(N/2-1)D_{CSA} + D_{BOOTH}$	$M(N/2-1)$
unfold sequential	$(N-1)D_{CSA} + D_{AND}$	$M(N-1)$

Table 4-1: Area, time performance of fast multipliers

Some simplified figures in terms of the number of carry-save adder stages are shown in Table 4-2.

multiplier	T(8)	T(16)	T(32)	T(64)	T(128)
bit-convolver	4	6	8	10	12
Wallace tree	4	6	8	10	11
Booth's encoding	3	7	15	31	63
unfold sequential	7	15	31	63	127

Table 4-2: Time performance in terms of D_{CSA}

We conclude that the recursive LABC implementation is faster than other known multipliers, except Wallace trees which have a slightly smaller time constant. However, Wallace trees are poorly suited to silicon implementation, and not modular in nature.

4.2. Multi-level simulation and verification

Detail experimental and theoretical results of this work can be found in [12], we summarize briefly the key issues here.

- *Unify structure*: We use a unify tree data structure to handle the descriptions at different abstraction levels (functional, net-list, floorplan, logic, circuit, mask descriptions). Such structure is macro-generated inside *CEYX* [9] (a programming environment for generating, editing and manipulating hierarchical structures).

- *Functional verification*: The complete circuit is functionally verified before detailing into logic design. The verification is carried out using *symbolic* (formal) approach, by manipulating the CEYX structure based on the functional and net-list descriptions.

- *Electrical simulation*: It is extensively used to confirm physical functioning and make critical design decisions (speed optimization, dynamic leakage, timing race, critical path delay, power consumption, ...).

- *DRC and logic simulation*: By the deadline for the design, *hierarchical* versions of the DRC and logic simulator were not available. So the above structure is valuable in the sense that: individual cells are verified by simple local DRC and logic simulator, *uniformity* of the multi-level descriptions guarantees the *global* correctness.

- *Circuit extraction*: Cell layout correctness is verified by circuit extraction followed by logic simulation.

We summarize by giving the approximate amount of time spent in each part of the design (Table 4-3).

algorithm	20%
global structuring	5%
functional verification	5%
circuit design	5%
electrical simulation	20%
layout (programming + graphic)	20%
DRC + logic simulation	10%
circuit extraction	5%
testing	5%
miscellaneous	5%

Table 4-3: Design time distribution

4.3. Test results

NMOS prototypes generated from the same program have been built simultaneously in Grenoble, France and in the USA (*DARPA MOSIS*). 8, 16 and 32-bit test chips of the LABC were fabricated using a 3μm process (2μm for the 32-bit chips). At the time of publication, only the MOSIS chips had been returned for testing, all function accordingly and test results are encouraging.

4.3.1. Testing for speed

We have implemented combinational circuits for the LABC for speed testing. The *multiply* time is measured as the time elapse between the end of the precharge cycle and the instant at which all the root nodes in the LABC complete their logic transitions. Detection for the completion of logic transitions is straightforward in precharged circuitry in which complementary logic states are available.

The delay of a bit-convolver depends on its *input patterns*. The binary tree consists of a precharged chain of CSA4's and a MU2 at the far end. The worst case propagation happens to be the case when all the CSA4's and MU2 have a 1 to 0 transition. This occurs when all the inputs being a logic 0.

The slowest multiply time of the 8, 16 and 32-bit prototypes are found to be respectively 120, 160 and 220ns. The experimental data confirms that the delay along the longest wire can be kept under control, thus roughly achieving the *logN*-time performance for medium size (16-32 bits) multipliers.

It should be pointed out that the basic cell design used in the smaller L-version may not be sufficient to provide enough driving power for all the long wires in the entire operand range (16-64 bits), this becomes increasingly critical for the case of 32 bits or higher. Thus preventing our current design to exactly match the predicted speed, they have been corrected and a second version of our design is currently being fabricated.

4.3.2. Testing for functional correctness

The chips are connected via parallel peripheral interface to a 68000 based tester to test for functional correctness. For the 8 and 16-bits chips, exhaustive test pattern are used and they function correctly. In the 32-bit case, special test patterns are used, and the chips also work accordingly.

4.3.3. Power consumption

The power consumption of the prototype chips are measured. The converted power consumption for the 8×8, 16×16 and 32×32 LABC's are respectively 180, 720 and 1280mW.

5. Conclusion

We present VLSI algorithms and *practical* layouts for *optimal time* parallel multipliers, which are also optimal and near optimal with respect to the previously established lower bounds on AP^2 (or AP^2T^2) tradeoffs.

We show how to *map* such algorithms into recursively defined *regular* silicon structures. Layouts can be automatically generated by parameterized *programs*, which take into account all the *idiosyncrasies* of a given technology, and tailor the circuits to specific *test* strategies. It is easier to run different tests on different circuits than to accommodate all on a single prototype.

Timing evaluation indicates that our design are *faster* than previously published ones for input size $N \geq 16$, and as fast as Wallace trees for medium size (16-64 bits). Although electrical problems in the basic cell prevent our current design to exactly match the predicted speed of about 100ns for 16-bit and 130ns for 32-bit, they have been corrected and a faster second version is currently being fabricated. Beyond this size, our bit-convolver can also be converted into *modular* array to generate *ultra*-large *logN*-time multipliers.

References

[1] Baudet, G. M., Preparata, F. P. and Vuillemin, J. E.
Area-time optimal VLSI circuits for convolution.
Technical Report 30, INRIA, Aug, 1980.
to appear in IEEE Trans. on Computer.

[2] Booth, A. D.
A signed binary multiplication technique.
Mech. and Appl. Math. 4(2):236-240, 1951.

[3] Brent, R. P. and Kung, H. T.
The area-time complexity of binary multiplication.
Journal of the ACM 28(3):521-534, Jul, 1981.

[4] Brent, R. P. and Kung, H. T.
A Regular Layout for Parallel Adders.
IEEE Transactions on Computers C-31(3):260-264, March, 1982.

[5] Capello, P. R. and Steiglitz, K.
A VLSI layout for a pipelined dadda multiplier.
ACM Transactions on Computer Systems 1(2):157-174, May, 1983.

[6] Chazelle, B. and Monier, L.
A model of computation for VLSI with related complexity results.
In *Proceedings of the 13th Annual ACM Symposium on Theory of Computation*, pages 318-325. ACM, May, 1981.

[7] Dadda, L.
Some schemes for parallel multipliers.
Alta Frequenza 34:349-356, Mar, 1965.

[8] Habibi, A. and Wintz, P. A.
Fast multiplier.
IEEE Transactions on Computers C-19(2):153-157, Feb, 1970.

[9] Hullot, J. M.
CEYX: a multiformalism programming environment.
Technical Report, INRIA, Oct, 1982.

[10] Knuth, D. E.
The art of computer programming, Vol.2.
Addison Wesley, 1972.

[11] Luk, W. K.
A regular layout for parallel multiplier of $O(\log^2 N)$ time.
In Kung, Sproull and Steele (editor), *CMU Conference on VLSI Systems and Computations*, pages 317-326. CMU, Pittsburgh, Oct, 1981.

[12] Luk, W. K.
Recursive implementation of optimal multipliers.
Technical Report, INRIA, 1983.
D.Eng. Thesis, University of Paris, Orsay.

[13] Lyon, R. F.
Two's complement pipeline multipliers.
IEEE Trans. Comm. COM-24(4):418-425, Apr, 1976.

[14] MacSorley, O. L.
High-speed arithmetic in binary computers.
Proc. of IRE 49:67-91, Jan, 1961.

[15] Masumoto, R. T.
The design of a 16x16 multiplier.
LAMBDA 1(1, first quarter):15-21, Jan, 1980.

[16] Mead, C. and Rem, M.
Minimun propagation delays in VLSI.
In *Proc. 2nd Caltech. Conference on VLSI.* caltech, 1981.

[17] Preparata, F. P.
A mesh-connected area-time optimal VLSI integer multiplier.
In Kung, Sproull and Steele (editor), *CMU Conference on VLSI Systems and Computations*, pages 311-316. CMU, Pittsburgh, Oct, 1981.

[18] Preparata, F. P. and Vuillemin, J. E.
Area-time optimal VLSI networks for computing integer multiplication and discrete Fourier transform.
In *Proc. ICALP Symposium.* , Haifa, Jul, 1981.

[19] Thompson, C. D.
A complexity theory for VLSI.
PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1980.

[20] Vuillemin, J. E.
A combinatorial limit to the computing power of VLSI circuits.
IEEE Transactions on Computers C-32(3):294-300, Mar, 1983.
also in Proc. 21st Symp. on Foundations of Computer Science, Syracuse, NY, October, 1980, pp.294-300.

[21] Vuillemin, J. E.
A very fast multiplication algorithm for VLSI implementation.
Integration, the VLSI Journal 1(1):39-52, Apr, 1983.

[22] Vuillemin, J. E. and Guibas, L.
A fast n-MOS implementation of addition.
In (editor), *Proc. ICCC82 Conference on Circuits and Computers*, pages 147-150. IEEE, New York, Sep, 1982.

[23] Wallace, C. S.
A suggestion for a fast multiplier.
IEEE Transactions on Computers EC-13(2):14-17, Feb, 1964.