A very fast multiplication algorithm for V.L.S.I. implementation

J. Vuillemin

IRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 183

A VERY FAST MULTIPLICATION ALGORITHM FOR VLSI IMPLEMENTATION

Jean VUILLEMIN

Janvier 1983

A VERY FAST MULTIPLICATION ALGORITHM

FOR VLSI IMPLEMENTATION

Jean Vuillemin

INRIA - Rocquencourt

Abstract :

We present a simple, recursive algorithm for multiplying two binary N-bit numbers in parallel O(log N) time. The simplicity of the design allows for a regular layout. The area requirement of this algorithm is comparable with that of much slower designs classically used in monolithic multipliers and in signal processing chips, hence the construction has definite practical impact.

Resumé :

Nous décrivons un algorithme récursif simple permettant de multiplier deux entiers binaires de N-bits en un temps parallèle O(logN). La simplicité de sa conception rend possible une disposition régulière du plan du circuit des masques. La surface de ce circuit est comparable à celle requise par les algorithmes habituellement utilisés dans les multiplicateurs monolithiques et dans les processeurs de traitement du signal. Cette construction présente un intérêt pratique direct.

1 Introduction

While O(log N) time algorithms for computing the product of two N-bits binary numbers have been known for close to twenty years ((Wallace 64),(Dadda 65),...) they have not much been used in modern integrated circuits. Because of their complex nature, such algorithms have generally been discarded by designers of integrated multipliers.

Designers have chosen instead to implement a comparatively slow ((TRW 77), (Matsumoto 80),(Cand-Scan-Ros 82),...) O(N) time seq-mult algorithm, which trades theoretical speed for regularity of design and silicon area.

The O(log N) time multiplication algorithm par-mult proposed here is simple enough to admit a regular layout whose area requirement is only marginally bigger than that of seq-mult. It's recursive definition makes it ideally suited for macro-generating regular mask descriptions from a high level algorithmic specification (Luk 83). We thus propose par-mult as a practical alternative to current VLSI multiplier design, whenever speed is the dominant design criterion.

Indeed, we estimate that par-mult computes 16 (resp. 32) bits products 2 (resp. 3) times faster than seq-mult ; yet it only uses 30 % (resp. 40 %) more area.

A prototype circuit and complete analysis of par-mult is performed by (Luk 83).

2 Description of the algorithm

We describe the algorithms Par-Mult and Seq-Mult in this section, postponing layout considerations until section 3.

2.1 Binary notation and statement of the problem

Let A = <a(n-1),...,a(0)> be a n-bit binary sequence. Such a sequence denotes the natural number val{A}=SUM{0=<i<n:a(i)*(2**i)}. Function val, which maps binary sequences {0,1}**n into natural numbers {a:0<=a<(2**n)} defines the semantics of binary notation.

A binary multiplier is an algorithm for transforming two binary sequences A and B into P=MULT(A,B) , the binary representation of the product val{A}*val{B}. If A and B are respectively m and n-bit sequences, P is a (m+n)-bit sequence. For the algorithm to be correct, sequence P must satisfy:

(1) val{P}=SUM{k>=0:p(k)*(2**k)}
=SUM{i,j>=0:a(i)*b(j)*(2**(i+j))}
=val{A}*val{B}.

Multipliers realized as integrated circuits encode bit values in the physical world as states of a bi-stable electronic device. For example, in MOS technologies, bits are represented by the presence or absence of electrical charges on localised capacitive wires. Inputs and outputs to such circuits must appear in this representation.

To implement a multiplication algorithm on silicon, we must decompose its description to the point where it is entirely made of atomic actions, whose functionality is exactly matched by an electronic realisation (gate) in the technology. Designing such algorithms is conceptually no different from producing machine code from high-level descriptions.

We attempt here to describe the silicon-structure of the proposed multiplier, by successively refining a high-level description of the algorithm, in a top down (and ultimately error free) manner.

2.2 Primitive building blocs for multiplication

If we represent 1 by *true* and 0 by *false*, the product a*b of bits a and b is implemented as the logical *and*(a,b) an atomic gate of the technology.

Using m copies of the primitive *and*, we can construct the operator *bit-product*, computing the product of a m-bit binary number A by bit b:

(2) *bit-product*(A,b)=<*and*(a(m-1),b),... *and*(a(0),b)>
val{*bit-product*(A,b)}=val{A}*val{b}
=*if* b=0 *then* 0 *else* A.

In hardware implementations, a parallel broadcast of bit b on a common bus (wire) ensures that *bit-product* introduces a (small) delay, independant of m, the number of bits in A.

Another useful primitive is *shift*(i,A), which adds i zeroes to the right (least significant bit) of sequence A:

(3) *shift*(i,A)=<a(m-1),...,a(0),0,. .0>
val{*shift*(i,A)}=val(A)*(2**i).

In parallel multipliers, the *shift* operator is "hard-wired" so as to introduce zero delay.

We shall fix later (section 2.4) the structure of the adders from which we assemble our multiplier. It will suffice for the time being to consider addition as a primitive operator *add* satisfying:

(4) val{*add*(A,B)}=val{A}+val{B}.

2.3 Algorithms Seq-Mult and Par-Mult

Formula (1), defining multiplication, can be rewritten as

(5) val{P}=val{A}*val{B}=SUM{j>=0:val{A}*b(j)*(2**j)}.

It follows that we can compute P=MULT(A,B) by initially forming all summands val{A}*b(j)*(2**j), and adding them all.

The classical algorithm Seq-Mult essentially sums up the operands in n-1 sequential stages.

We propose, for Par-Mult to sum up the operands in a parallel tree, 2 by 2, then 4 by 4, ... , in log(n) parallel stages.

The following recursive definitions of both algorithms show their similarities and differences :

(Seq-Mult)

```
SEQ-MULT(A,B)::=
        if 0=<val{B}=<1
          then bit-product(A,b(0))
          else add(shift(k,SEQ-MULT(A,B1)),SEQ-MULT(A,B0))
```

where k=1,
B=<b(n-1),...,b(0)>, B1=<b(n-1),...,b(1)>, B0=<b(0)>.

(Par-Mult)

```
PAR-MULT(A,B)::=
        if 0=<val{B}=<1
          then bit-product(A,b(0))
          else add(shift(k,PAR-MULT(A,B1)),PAR-MULT(A,B0))
```

where k=n/2+1
B=<b(n-1),...,b(0)>, B1=<b(n-1),...,b(k)>, B0=<b(k-1),...,b(0)>.

The correctness of both algorithms, as expressed by (1) and (5), follows from (2),(3), and (4), together with

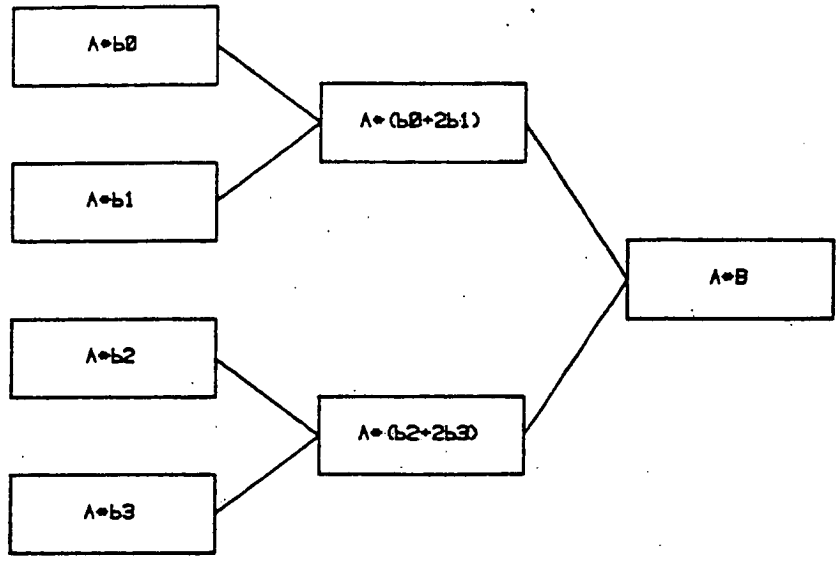
(7) val{B}=val{B1}*(2**k),

this last expression holding true for both algorithms (k=1 or k=n/2+1).

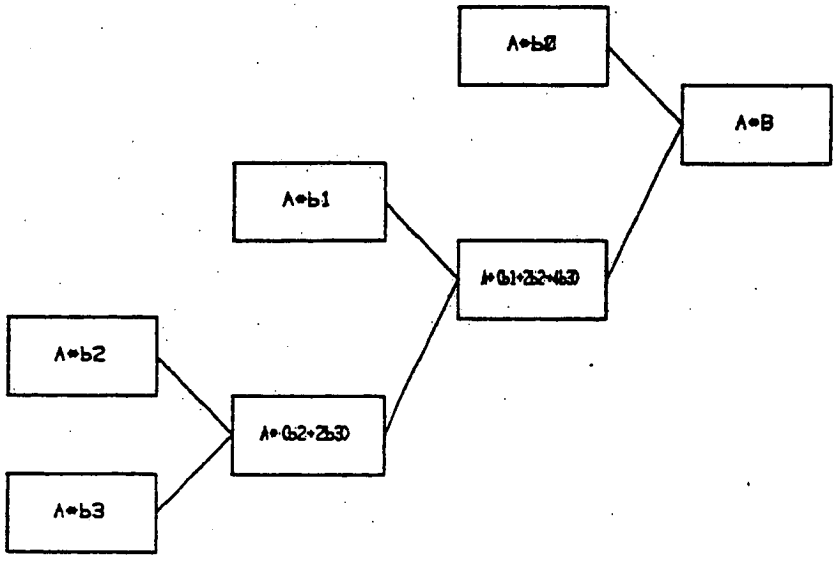
Both algorithms perform exactly n-1 additions in the course of multiplying a m-bit sequence A by the n bits of B. Because of

parallelism, these additions are completed within log(n) (baes 2 logarithm) stages in Par-Mult, while Seq-Mult requires n-1 such stages.

Each algorithm can be depicted by a tree of successive additions, as in Figure 1.



Par-Mult (A, <b3, b2, b1, b0>)



Seq-Mult (A, <b3, b2, b1, b0>)

Fig. 1. Addition trees for *Seq-Mult* and *Par-Mult*.

2.4 Using carry save representation for intermediate products.

A carry save number M is a pair (R,S) of binary sequences R and S. The value of M is the sum val{M}=val{R}+val{S} of the values if its components R and S. A convenient representation of carry save numbers by sequences is M=<m(n-1),...,m(0)>, where digit i is m(i)=r(i)+s(i), with value in 0,1,2, as shown in Figure 2.

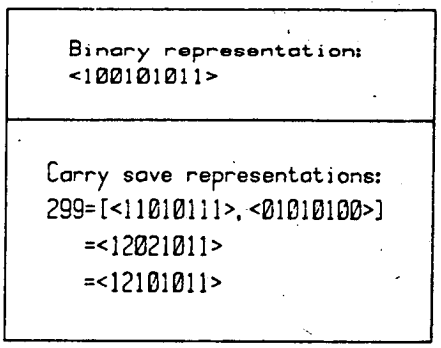


Fig. 2. Carry save representation.

The addition add-csb((R,S),T) of a carry save number M=(R,S) with a binary integer T, yielding a carry save result (R',S') can be performed in a bit-wise manner:

(8) add-csb((R,S),T)::=(R',S')
where
S'=<fa-sum(r(n-1),s(n-1),t(n-1)),...,fa-sum(r(0),s(0),t(0))>
and
R'=shift(1,<fa-carry(r(n-1),s(n-1),t(n-1)),...,fa-carry(r(0),s(0),t(0)

The bit-wise full adder is defined by:

(9) s'=fa-sum(r,s,t)::={(r+s+t) mod 2}
and r'=fa-carry(r,s,t)::={(r+s+t)>1},
so that (2r'+s')=(r+s+t).

The sum add-cscs((R,S),(T,U)) of two carry save numbers M=(R,S) and N=(T,U) is computed in two steps:

(10) add-cscs((R,S),(T,U))::=add-csb(add-csb((R,S),T),U)

It follows directly from (8),(9) and (10) that:

(11) val{add-cscs(M,N)}=val{M}+val{N}.

The delay Tadd introduced by the parallel computation of add-cscs is independant of the length of the operands, namely:

(12) Tadd-cscs=2*Tadd-csb=2*Tfa

where Tfa is the delay introduced by one stage of full adder.

2.5 Par-Mult and Seq-Mult revisited

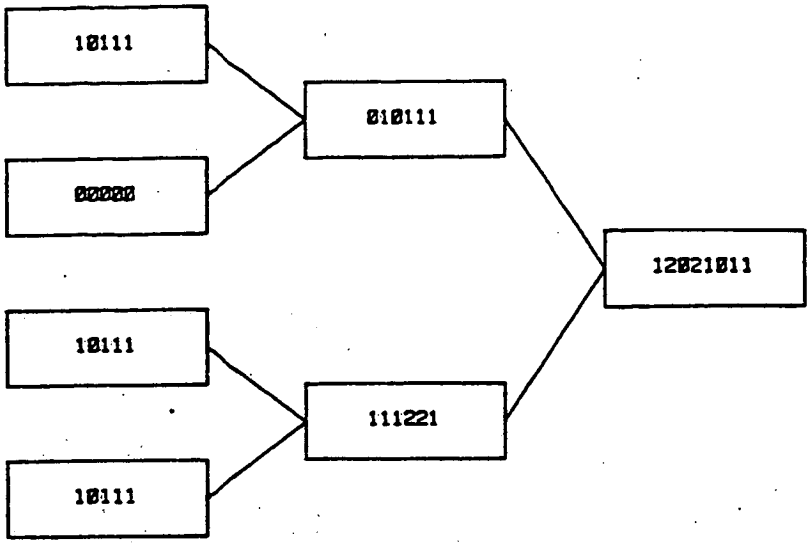
Assuming the operands A and B of multiplication to be initially presented in binary form, the first level of addition in the trees of Figure 1 is used to obtain the intermediate sums (A*b(2j),shift(1,A*b(2j+1))) in carry save form. There is no delay involved in the initial conversion form-cs which is a mere convention.

In the case of Seq-Mult, all subsequent additions combine a carry save and a binary number, thus we use add-csb.

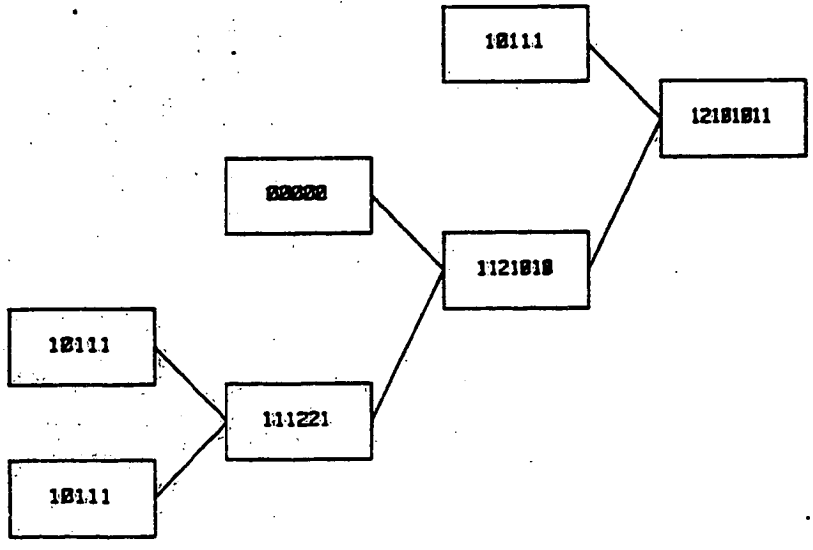
For Par-Mult, all intermediate results after the first stage are in carry save form, and they are combined two by two using add-cscs.

In both cases, the final addition stage yields a carry save representation of the product P=A*B. The two binary sequences forming this carry save product must then be added, in order to produce the final result in true binary form. For this purpose, we use a fast carry-look-ahead adder cla-add, such as described for example by (Vuill-Guib 82).

Computation of the numerical product 23*13 with both algorithms is shown in Figure 3.



Par-Mult(13, 23)=299=<100101011>



Seq-Mult(13, 23)=299=<100101011>

Fig. 3. Computation of $13 * 23$ with *Seq-Add* and *Par-Add*.

2.6 Timing analysis

If m and n are the number of bits of A and B respectively, the time required by Seq-Mult and Par-Mult for computing the (n+m) bits of product P=A*B is given by:

(13) Tseq-mult(m,n)=T0+(n-2)*Tfa+Tcla-add(n+m);

(14) Tpar-mult(m,n)=T0+2(log(n)-1)*Tfa+Tcla-add(n+m).

Here, T0 represents the time required for input distribution and bit-wise products. It is shown by (Vuill-Guib 82) that integrated carry look ahead adders can be designed with logarithmic delay Tcla-add(p)=T1*log(p). It follows that Par-Mult also has a logarithmic computing time, and it proves faster than Seq-Mult for all values of n and m.

To be fair, we must point out that Seq-Mult is almost invariably implemented in conjunction with Booth's recoding of operand B, as described for example in (Matsumoto 80). Such a recoding divides the number of summands by two, improving the speed by almost the same factor.

Booth's recoding can be used in conjunction with Par-Mult just as well, dividing by two the number of initial summands. The speed gain in that case is marginal, since it only amounts to reducing by one the depth of the tree of adders, at the expense of recoding. All this is not worth the complexity and area increase. As it stands, Par-Mult without Booth's recoding remains faster than Seq-Mult with Booth's recoding for all operand lengths.

3 Laying out Par-Mult on silicon

We describe an efficient layout strategy for Par-Mult. Although our target technology is MOS, the construction is general enough to apply to other technologies, such as Bipolar, ...

Our technique for laying out Par-Mult(m,n) uses a rectangular array of (m+n) columns and n rows. Row i initially contains the summand b(i)*A, suitably shifted by 2**i positions, so that column k contains the "bit-slice" of partial products b(i)*a(j), with i+j=k.

For the sake of description clarity, it proves convenient to first define a 3 dimensional layout, suitable for ideal technologies endowed with log(n) levels of connexions. This 3-D layout is then mapped into an ordinary 2-D floorplan, so as to fit within the stringent constraints of current technologies, limited to one (or two) levels of connexion.

We also limit ourselves to describing the "carry save" part of our multiplier: conversion from carry save to true binary at the last stage of the algorithm uses circuitry which is descibed elsewhere (Vuill-Guib 82).

3.1 A 3-D recursive layout

Let n=2**k, with k=log(n) be the length of multiplicand B. Our 3-D layout is a parallelogram made of k superposed plane structures.

To formally describe such structures, we introduce the operators JX,JY,JZ: let P and P' be parallelograms of respective dimensions dx,dy,dz and dx',dy',dz'; the operation JX(P,P'), which is only defined when dy'=dy and dz'=dz, constructs a parallelogram of dimensions dx+dx',dy,dz by juxtaposition of P and P' along the x axis. Operators JY and JZ are defined "mutatis mutandis".

A n*m 3-D multiplier layout is constructed by invoquing the function par-mult-layout(0,log(n)), which is recursively defined by:

```
(15)    par-mult-layout(i,k)::=
            if k=0 then bit-product-layout(i)
                   else JZ(REC-MULT,add-cscs-layout(k))

where REC-MULT=JY(par-mult-layout(i,k-1),
                  par-mult-layout(i+2**(k-1),k-1)).
```

A pictorial view of the 3-D multiplier is given in Figure 4.

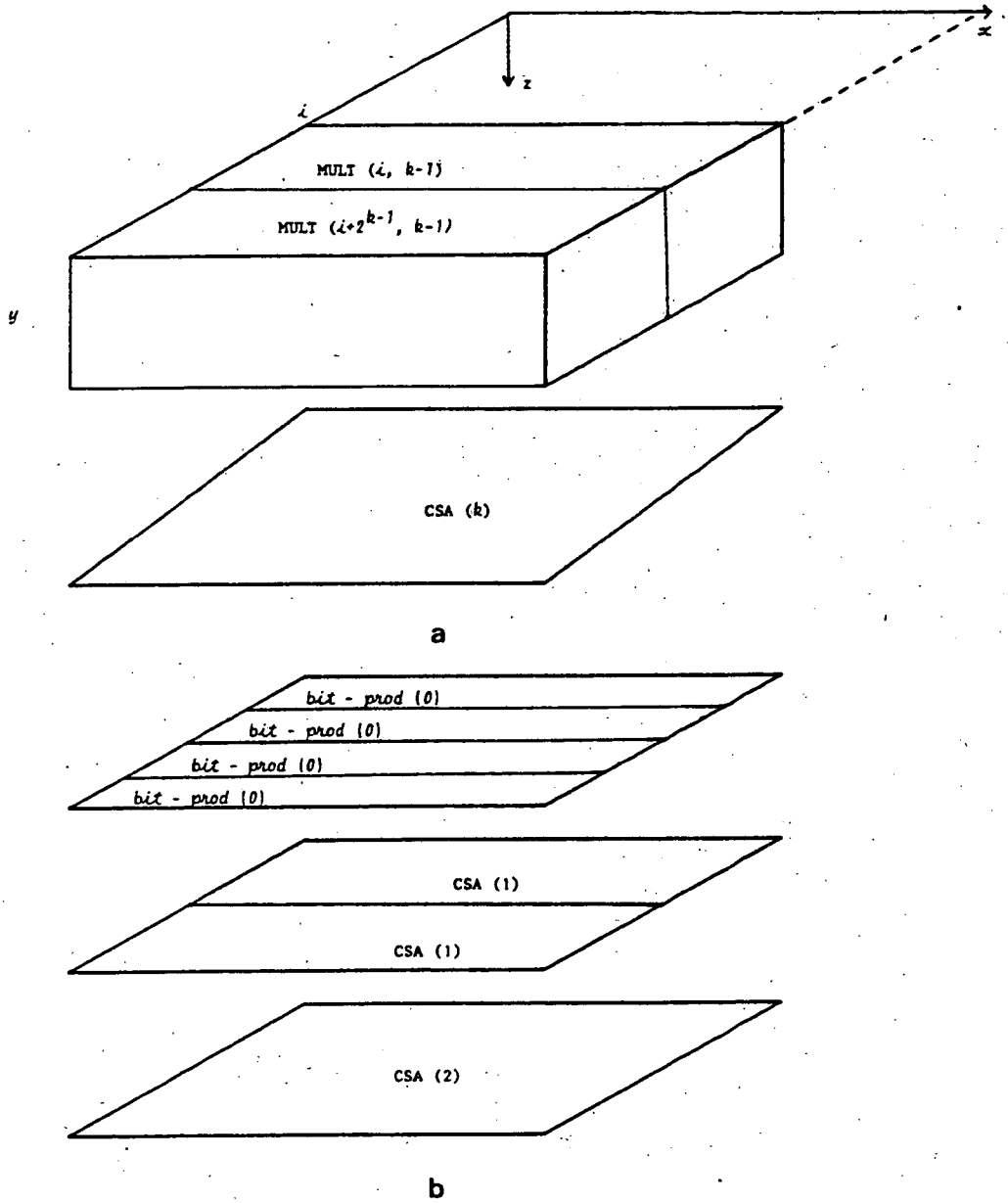


Fig. 4. 3-D layout of *Par-Mult.* (a) Recursive view of MULT(i, k); (b) Unfolded 3-D version of MULT(0, 2).

3.1.1 Layout of bit-product

The primitive operation bit-product-layout(i) generates the planar layout of a rectangular circuit, formed by a row of (n+m) identical cells. Each cell has a horizontal (x-axis) input b(i), a diagonal (x&y-axis) input a(j), and computes the product and(b(i),a(j)) along the third (z-axis) dimension, as in Figure 5. The index j of a(j) runs from n+m-i-1 to j=-i, with the convention a(j)=0 whenever j is outside the range 0=<j<m.

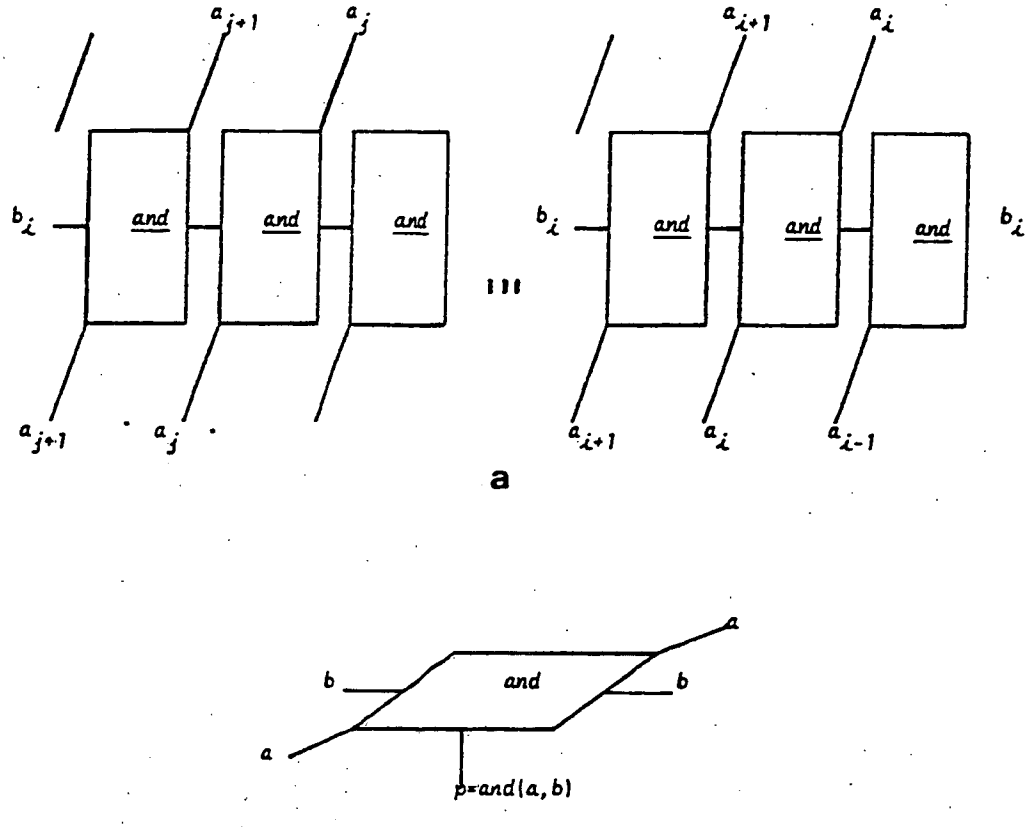


Fig. 5. Bit-product layout. (a) Result of *bit-product-layout(i)*; (b) A 3-D view of the *and* cell.

Although Figure 5 uses diagonal connections, it is easy enough to replace these by connections which are all parallel to one of the coordinate axis, x,y or z.

3.1.2 Carry save adder layout

The operator add-cscs-layout(k) generates the layout of a circuit for adding two carry-save numbers, entirely made of interconnected full-adders fa. There are two such fa's in each bit-slice. The first fa receives its three operands s(i),t(i),u(i) from level (k-1), producing two outputs q(i+1),p(i) such that s(i)+t(i)+u(i)=2*q(i+1)+p(i). The second fa receives operand r(i) from level (k-1), operands p(i) and q(i) from level k. It delivers its outputs n(i+1) and m(i) at level (k+1), as shown in Figure 6.

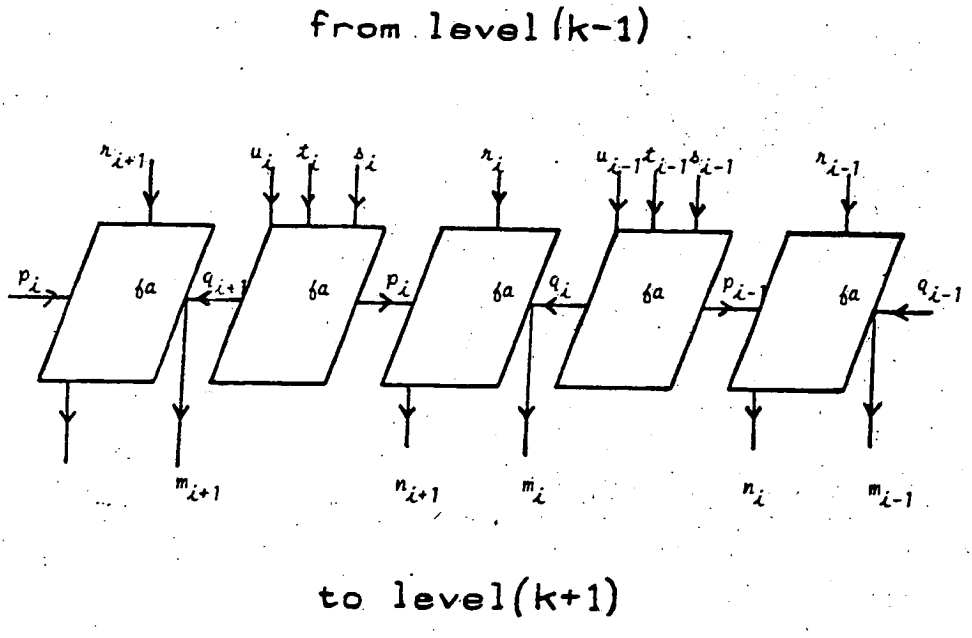


Fig. 6. Layout of add-cscs.

As indicated in section 2.5, the adder add-cscs-layout(1) at level 1 is not performing any operation: it merely combines two binary numbers originating at level 0, into a pair, representing the carry-save input to level 2. Thus, level 1 comprises only routing of signals.

3.2 Mapping 3-D into 2-D layout

In order to imbed our layout in a planar technology, we use the following rules:

Rule 1: We keep the row structure, by alternating bit-products and additions, according to Figure 7, obtained by the recursive definition:

```
(16)    par-mult-2D-layout(i,k)::=
            if k=0 then bit-product-layout(i)
                   else JY(par-mult-2D-layout(i,k-1),
                           add-cscs-layout(k),
                           par-mult-2D-layout(i+2**(k-1),k-1)).
```

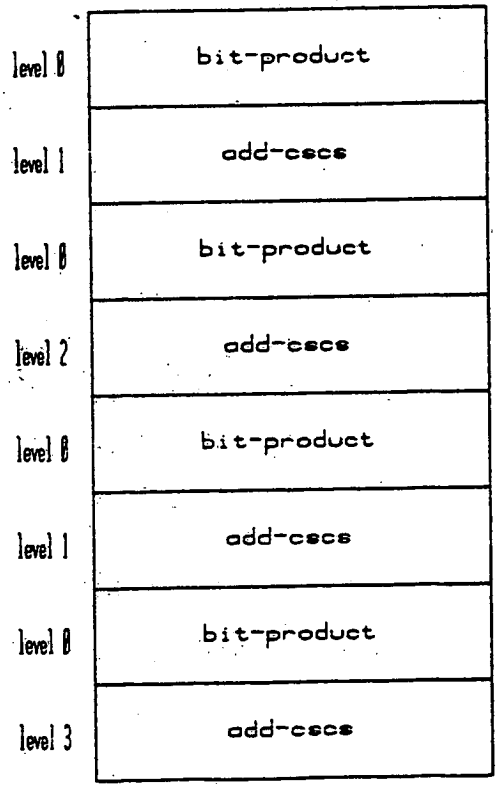


Fig. 7. Row-layout of *Par-Mult.*

Rule 2: We keep the bit-slice structure of columns by allocating k=log(n) vertical channels within each column. Channel j is dedicated to the layout of level j.

Rule 3: (x) Horizontal (x-axis) wires are mapped into horizontal wires within the same row.

(y) Vertical (y-axis) wires are mapped into vertical wires within the corresponding bit-slice channel.

(z) Third dimension (z-axis) wires between levels i and i+1 run horizontally between the i-th and the i+1-rst channel of the corresponding row and column.

Figure 8 shows the channel structure of a bit-slice, and the corresponding routing strategy.

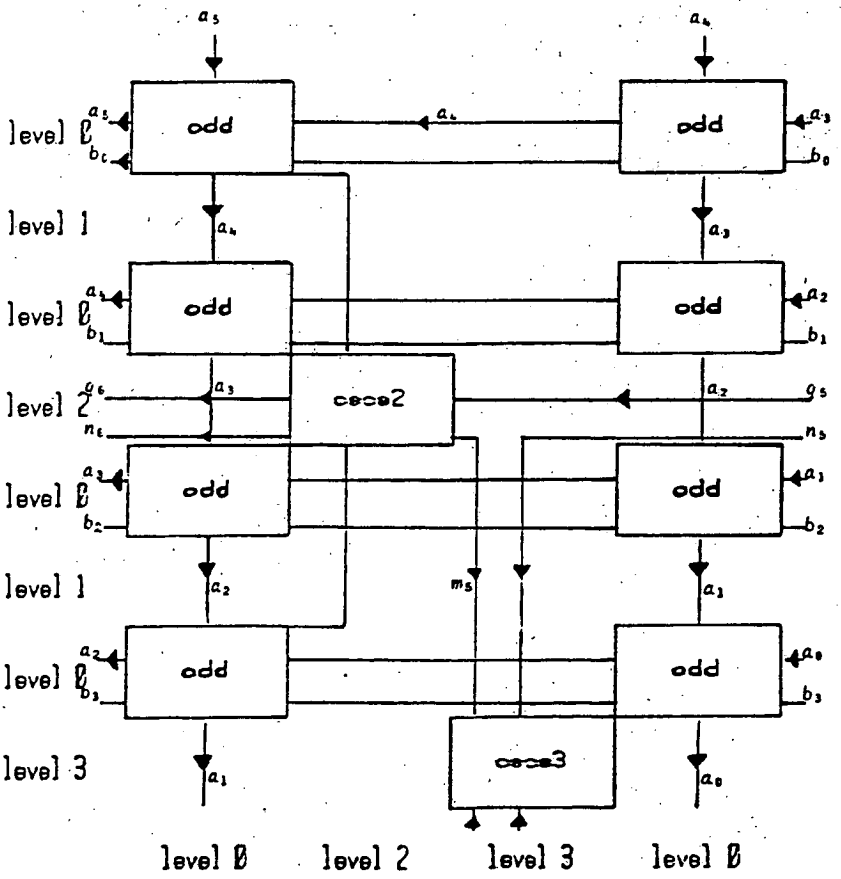


Fig. 8. Column layout of *Par-Mult.*

Such a layout keeps the tree structure adopted in Par-Mult for summing up partial products apparent in each bit-slice, as in Figure 8. The area penalty incured over Seq-Mult thus only arises because of the O(log(n)) extra wires required by the tree structure connexion. For n up to 32, this area is less than half of that devoted to the full-adder logic layout. Hence, Par-Mult can be layed out within 150% of the area of Seq-Mult.

3.3 On truly large time O(log(n)) multipliers

When n gets very large (n>64), the tree structure of Figure 8 no longer performs addition in time O(log(n)): although signal only traverses log(n) gates, gates no longer possess unit delay, because their parasitic capacitance increases with the length of output wires. As a consequence, gates become slower as they get closer to the root of the tree. Indeed, wire lengths double with each level up in the tree.

A radical solution to this problem, as presented in (Vuill-Guib 82) or (Mead-Rem 80), uses the layout of Figure 9, which has signal amplification built into the tree structure. Tree nodes at level h+1 have twice the speed and size of tree nodes at level h. This means that transistor gates at level h+1 are twice longer than their homologue at level h. They are therefore able to drive output capacitance twice bigger, within the same time constant.

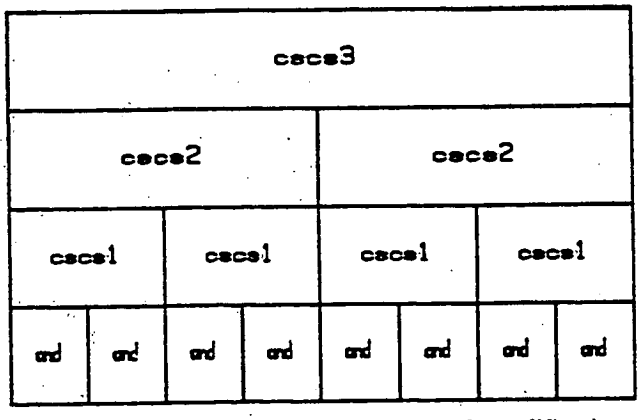


Fig. 9. Bit-slice layout of Fig. 8 with signal amplification.

Using this technique allows to design tree structures in which all gates have the same timing characteristic, regardless of their heigth in the tree. Consequently, it is possible to design time O(log(n)) multipliers in MOS technologies, for arbitrary large values of n. This requires however two levels of interconnect, since connexions of Figure 8 still have to overlap the gates of Figure 9.

4 Conclusion

We feel that the algorithm Par-Mult and its silicon layout are interesting from at least two points of view:

(a) It provides a mathematically elegant and easy to program technique for multiplying two N-bit binary integers in time O(logN).

(b) It provides an attractive practical alternative to Seq-Mult, in designing fast integrated multipliers, for 8 bits and over. This holds true of a large class of technologies, including n-MOS, c-MOS, and bipolar.

5 References

(Wallace 64) Wallace C.S., "A Suggestion for Parallel Multipliers," IEEE Trans. Electronic Computers, Vol.EC-13, Feb. 1964, pp. 14-17.

(Dadda 65) Dadda L., "Some Schemes for Parallel Multipliers" Alta Frequenza, Vol. 34, March 1965, pp. 349-356.

(TRW 77) TRW, "MPY-LSI Multipliers : AJ 8x8, 12x12 and 16x16," LSI Products, TRW, Redondo Beach, Calif., March 1977.

(Matsumoto 80) Matsumoto R.T., "The Design of a 16x16 Multiplier," LAMBDA, first quarter, 1980, pp. 15-21.

(Cand-Scan-Ros 82) Cand M., Le Scan P. and Rosset A., "A Single Chip Digital Signal Processor", Proc. IEEE, Vol. 429, Sept. 1982, pp. 356-359.

(Luk 83) Luk W.K., "Silicon Compilation of a Fast Parallel Multiplier" These de 3eme Cycle, Universite de Paris 1983.

(Vuill-Guib 82) Vuillemin J. and Guibas L., "On Fast Binary Addition in MOS Technologies" Proc. IEEE, Vol. 429, Sept. 1982, pp. 147-150.

(Mead-Rem) Mead C. and Rem M., "Minimum Propagation Delays in VLSI", Proc. 2nd Caltech Conference on VLSI, 1981.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique