Rendiconti
Accademia Nazionale delle Scienze detta dei XL
Memorie di Scienze Fisiche e Naturali
124° (2006), Vol. XXX, P. II, pp. 51-80

LUIGI DADDA*

Parallel Decimal Multipliers: a hybrid approach[1]

Abstract – An algorithm for decimal parallel multiplication is presented, in which the basic steps are: the creation of 4*4 arrays representing the products of a multiplicand BCD digit and of a multiplier digit (digit partial products), the calculation of the binary sums of such arrays lying on each digit- column of the total array, their conversion in decimal, the creation of an array of "column partial products", composed by 3 or 4 "major partial products", whose sum is the product. Three main points are discussed: the design of the adders of a number of digit partial products (using a compact dot notation and showing a design tool based on a spreadsheet), the parallel binary-decimal converters composed by a planar array of identical cells, the parallel addition of the major partial products obtaining the product. The area of this multiplier is shown to be slightly higher than the area of a parallel binary multiplier, with factors of same bit-length. The total delay is approximately twice the delay of a binary parallel multiplier.

Keywords: parallel arithmetic, high speed arithmetic, logic arrays, design aids, automatic synthesis.

I - Introduction

Considerable interest has arisen since few years in decimal arithmetic, in particular in decimal multipliers [12, 13, 14, 16]. The related research appears to be rather exploratory, since the problems are more complex than it could seem at a first glance, and no definitive real good solution have not been yet found. The

* Uno dei XL. Dipartimento di Elettronica e Informazione, Piazza Leonardo da Vinci, 32, Politecnico di Milano, 20133 Milano. E-mail: dadda@elet.polimi.it; ALaRI- Università della Svizzera Italiana, 6900 Lugano, CH, E-mail: dadda@alari.ch

[1] A paper of same title has been submitted to IEEE Transaction on Computers at 12 February 2006 and withdrawn at 25 January 2007 for lack of decision. This paper is a revision of such text, with the addition of a new chapter VI on Time and Area.

approach followed by most Decimal Arithmetic schemes so far proposed assumes decimal digits represented in one of the many binary codes available, from BCD to Signed Codes. Moreover the proposed decimal multipliers are based on parallel-serial operation. Elementary arithmetic operators and any parts of them involve often some form of re-coding. A particular attention has been paid to decimal codes offering the possibility of implementing addition with limited or no carry propagation [12, 13].

We will show that it is possible and convenient to assume in some parts of a decimal multiplier, a pure binary code. The advantage is a simpler and faster binary processing. The disadvantage consists in the need for radix conversion, i.e. binary to BCD.

A radical application of such approach consists in using a purely binary multiplier (e.g. a fully parallel one) "surrounded" by suitable binary to decimal (DB) and BD converters. Such a scheme offers a high throughput (comparable to the one offered by the binary multiplier) at the cost of a considerable latency and greater area required by the converters.

We consider here the case of a fully parallel decimal multiplier where the binary approach is assumed only for well defined parts of it (precisely, to the addition of single "digit-columns" of the products array).

II - A basic scheme

In a binary multiplier each element of the initial product array generates a single binary digit obtained as a logical AND of a multiplicand digit with a digit of the multiplier.

The product is the value of the total array and the various schemes are characterized by the algorithms used in transforming the multiplication array into a single equivalent binary number.

The generation of the multiplication array in decimal multipliers differs from the binary case due to the fact that the factors are decimal numbers, composed by decimal digits coded in binary in various ways: we assume to adopt the binary coding of decimal digits (BCD). Each *Digit Partial Product* is obtained from a digit of the multiplicand (Md) and another from the Multiplier (Mr). The first step of the multiplier algorithm consists in obtaining each *Digit Partial Product* from the two digits. This is an expensive operation, if the digit partial product (i.e. the decimal product of two decimal digits) has to be obtained as a decimal two digits number. It can be obtained through a table or through the direct synthesis of each partial product combinational functions (8 with BCD coding).

Due to the fact that all digit partial products require two decimal digits, two different algorithms can be used for their accumulation. In the first each digit partial product is added to another digit partial product (of same column in the product array) by considering both digits (of weights 10^i and 10^{i+1}) composing it. In a second algorithm the most significant digit is associated with the least significant digit of a digit partial product belonging to the following (*i+1*) column: two digits of same weight are added instead of two digits of different (*i, i+1*) adjacent columns.

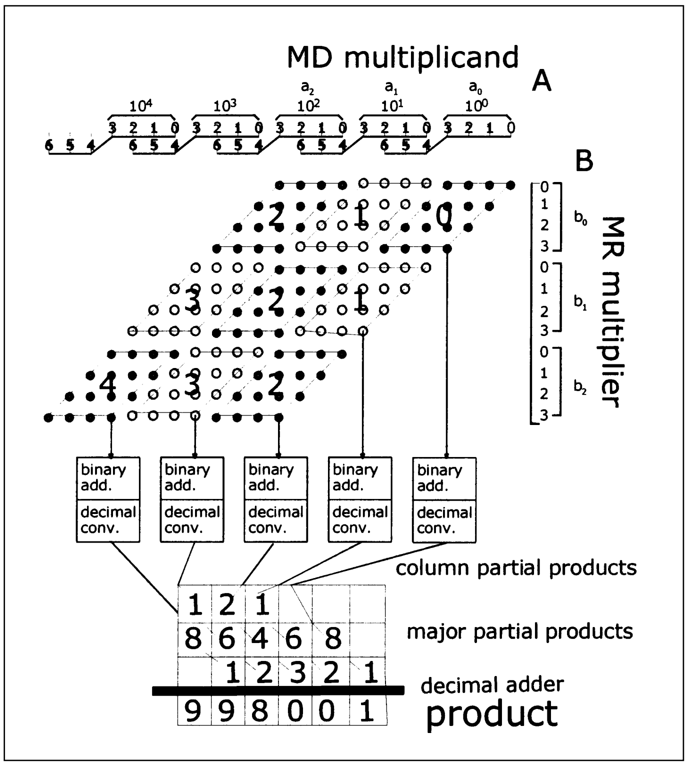


Fig. 1. The basic scheme of a 3*3 multiplier: the values in the bottom part assume that all MR and MD digits are equal to 9.

We propose to avoid any encoding of the digit partial products, assuming them to be represented simply by the 16 bits generated by 16 two-input AND gates that appear arranged in a "digit-array" as shown in fig. 1. Each digit-array is enclosed in a parallelogram, and an array of 3*3 digits is shown decomposed in 9 digit arrays.

Let's consider as a working example the product of 3*3 decimal digits A and B. Each of the three decimal digits a_2, a_1, a_0 and b_2, b_1, b_0, is encoded with bits having weights equal, for a_2, a_1, a_0, to: $a_{2,3} = 800$, $a_{2,2} = 400$, $a_{2,1} = 200$, $a_{2,0} = 100$; $a_{1,3} = 80$, $a_{1,2} = 40$, $a_{1,1} = 20$, $a_{1,0} = 10$; $a_{0,3} = 8$, $a_{0,2} = 4$, $a_{0,1} = 2$, $a_{0,0} = 1$, and similarly for b_2, b_1, b_0.. The numbers A and B are not binary, since the weights of the successive bits are not (except for the 4 least significant ones) integer powers of the base 2 (as: $2^4 = 16_{10}$, $2^5 = 32_{10}$, …).

Suppose now that the two numbers are treated as in parallel binary multipliers, by forming the array of fig. 1, where each line starting from A's bits of a multiplicand digit and each line from the B's bits of a multiplier digit crosses in 16 points, the 12 lines starting from the multiplicand A being directed diagonally to down-left direction. A two-input AND gate in each of the 12*12 crossings produces a bit of the product array.

The array can be partitioned, as shown in Fig. 1, into digit-arrays representing the product arrays of two BCD digits of 4 bits each, that we have call Digit Partial Products. It is important to note that in each digit-arrays each of the 7 columns is characterized by weights from $2^0 = 1$ (for the rightmost one containing the upper-right corner of the digit-array), to $2^6 = 64$ (for the leftmost column containing the lower-left corner). The intermediate columns contain 2 or 3 or 4 bits of the digit-array.

It is also important to note that the 3*3 decimal digit-arrays can be identified by the couple *i, j* of integers given by the index *i* of the multiplicand and *j* of the multiplier.

The product array can be seen as an array composed by digit-arrays as shown in the figure: the digit-arrays are arranged in columns, each containing all digit-arrays with the same $i+j$ value.

Each Digit Partial Product is computed as a binary number with weights from 2^0 to 2^6, whose maximum value is: $9*9=81_{10}$ or : $101\ 0001_2$. Each digit partial product is multiplied by $10^{(i+j)}$, where *i* and *j* are the subscript's of A and B digits. The product is equivalent to the sum of all the digit partial products composing the array of fig. 1 (the product array).

Note that in each of the product array column, the weight of each bit is not the same, since it depends from the digit-array to which it belongs. More precisely, it is necessary, as done in fig. 1, to mark the successive weights for each digit-array column from 2^0*10^i to 2^6*10^i, where 2^4*10^i, 2^5*10^i and 2^6*10^i overlap 2^0*10^{i+1}, 2^1*10^{i+1} and 2^3*10^{i+1}, valid for the successive $i+1$ digit-array columns.

Obtaining such a sum, expressed with a decimal BCD number, is our problem.

A further step is to split the weights of each bit of the product array in two factors, the first being the binary weight 2^k ($0<=k<7$), the second the decimal one, 10^{i+j}.

The main reason for partitioning the product array into digit arrays columns is given by the fact that all bits belonging to a digit-column are characterized by successive power of two. It becomes then possible to evaluate the value of each digit-column as can be done in purely binary multipliers [2].

The binary numbers thus obtained must be multiplied by $10^{(i+j)}$. The successive digit-arrays columns are characterized by successive values of $(i+j)$.The addition of such numbers will lead to the product: this requires that before the addition, each column value, obtained in binary, is converted in decimal form. Consequently the multiplication with a power of ten and the alignment for their addition can be done very easily, through suitable wiring.

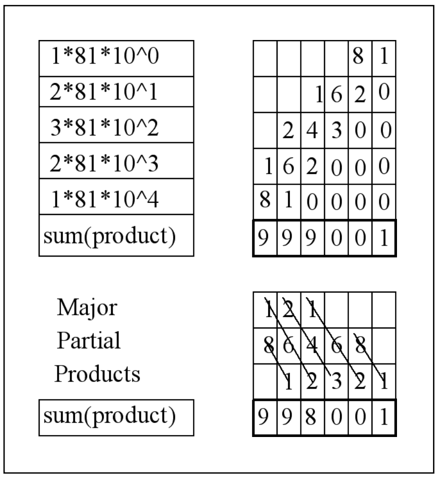
Details about the above problems will be given in the following chapter.

Meanwhile, it will be of some interest to show the main intermediate results, concerning the product: 999*999 = 998001.

An example

Assuming the input digits equal to 9, all digit-array values will be 81_{10} i.e. 1010001_2. The (maximum) value of the successive digit-arrays columns will be: 2*81*10=1620; 3*81*100=24300 (see Fig. 2). For the two columns following the third one (see fig. 1) we get the values: 2*81*1000=162000 and: 810000. The sum of those numbers is: 998001, i.e. the product.

A better way for obtaining the product through the columns values is to write these as in fig. 1 and fig. 2, i.e. in "skew-tiled" form: we then obtain three decimal numbers, whose sum is the product: we call this three decimal numbers the *Major Partial Products*.



1*81*10^0					8	1
2*81*10^1			1	6	2	0
3*81*10^2		2	4	3	0	0
2*81*10^3	1	6	2	0	0	0
1*81*10^4	8	1	0	0	0	0
sum(product)	9	9	9	0	0	1

Fig. 2. The addition of the Column Partial Products in Fig. 1 scheme.

It can be shown that three numbers will suffice for multipliers of up to twelve digit factors, since 81*12=972 (3 digit) and 81*13=1052 (4 digit).

Four numbers will suffice for factors of up to 123 decimal digits: 81*123=9963 (4 digit), 81*124=10044 (5 digit).

We discuss now with more details the three main parts of the general scheme of Fig. 1, i.e. the computation of the Column Partial Products, the conversion in decimal of their binary outputs, the summation of the Major Partial Products obtaining the Product.

Meanwhile, we invite the reader to look and to test a 8*8 and a 16*16 decimal multipliers, implementing the algorithm just described in a pure decimal form, using for the purpose a spreadsheet: they can be downloaded from the Web, from the site whose URL is given at [18]. This Web page contains the links to various other spreadsheets programs, recalled later.

III - Computing the binary value of digit-arrays columns ("Column Partial Products") via combinational networks of full-adders

A Column Partial Product has been defined in the preceding chapter as the sum of the digit arrays composing a digit-arrays column, as shown in fig. 1

The 7 vertical lines defined by each 4*4 digit array are assumed to carry the weights: 2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6 or: 1, 2, 4, 8, 16, 32, 64. The factors $10^{(i+j)}$ (see fig. 1) will be taken into account later, when dealing with the addition of the Column Partial Products through the Major Partial Products Array.

The evaluation of the Column Partial Products can be done using the same method used for designing binary multipliers [2, 3, 7, 8, 15]. This design method is an abstract representation of the real circuitry and will be applied as a first example to the evaluation of a Column Partial Product composed by a single digit array: see fig. 3 ($p=1$). The first line gives in the 7 column the values: 1, 2, 3, 4, 3, 2, 1 representing the number of binary variables in each column of the "dot" diagrams in fig. 3A.

In the rightmost column we see: a single dot under the 1 value, in the 3rd line of the 2nd array. A crossed slash joins a dot in the 2nd column (1st line of 2nd array, under the input value 2), with a dot in 3rd column (2nd line of 2nd array): the crossed slash represents a half adder, whose two inputs are the 2 variables in 2nd column of 1st input row.

The value 3 in third (from right) column represents variables input to a full adder, represented by a slash joining the two outputs. The value 4 in the 4th column corresponds to a full adder, using 3 variables, the fourth being simply transmitted with to the 3rd line of 2nd array.

Note that the two inputs in column 6th are not processed, being simply moved to the second array with a dot marked with 2 in the third row. The reason why the two inputs in 2nd column are processed via a half adder is to produce a single output bit in 2nd column: see more comment later.

The second "compression" stage transforms, with the rules given above, the three lines of the 2nd array into the two lines of the 3rd array.

Finally, it is assumed to adopt a carry propagating adder (usually, for speed reason, a carry look ahead adder) for obtaining the binary number equivalent to the original 4*4 product array.

An important note: the product of two BCD decimal digits has a maximum value of $9*9=81$, while the product of two generic 4 bit numbers is $15^2=225$. In binary, 7 bit are required to represent 81_{10} and 8 bit to represents 225_{10}. Moreover, the compression process adopted obtains the final result with three bits (the less significant) fully computed. As a consequence, the final adder is composed with 4 stages only, with no carry needed from the leftmost stage, since at most 7 bits in total are needed.

Consider now the case $p=2$, see fig. 3A. The first line represents the input variables in the different column as: 4, 8, 12, 16, 12, 8, 4, twice the values of the preceding case. We apply here a variation of the method used so far, leading to "compact" dot-schemes. Schemes, that with the standard methods [2] may be difficult to trace and to read, for a too large number of rows, can be handled much more easily.

In order to compress a given column, the number d of variables of same weight is divided by 3 and the quotient q written close to the segment joining the two outputs, so that the necessary full adders are represented by a single *full adder* with the "multiplicity factor" q close to it or to the carry and sum dots, placed in two adjacent columns. The remainder (valued 0 or 1 or 2) is written below the sum output in a third row: see fig. 3A. In another version, in case of remainder equal to 2, a *half adder* can be used to obtain a sum and carry output placed in two adjacent columns.

All the fig. 3A dot-schemes have been constructed with the just described procedure, with two additional points.

The first is the use of a half adder at the right end on each compression stage if the input variables are 2: this is done with the purpose of obtaining in the last 2-rows stage a number of least significant single bits. This requires a smaller carry propagating final adder, and therefore a smaller delay in the final addition.

The second point concerns the number of bits of the final output, due to the fact that the 4*4 digit products are bound to operate on 4-bits factors representing BCD digit and not generic 4-bit binary numbers. For the reason said before, the 8th bit is not generated in fig. 3A ($p=1$) scheme.

In a similar way this check has been done for all the other fig. 3A schemes, knowing the values of the multiples of 81_{10}. These are listed in Appendix B Table, in column 3 with their binary equivalents.

It can be noted, in the last 2-line arrays of schemes $p=3$, $p=5$ and $p=6$, a half adder with a X at the carry output, placed in the leftmost position. This symbol represents a 2-input XOR for obtaining the Sum output [11]. The carry is not produced, since no carry can be generated at those points.

The above rules allow the design of rather complex computing schemes based on a network of full (and half) adders, or (3,2) and (2,2) elementary counters.

The schemes are not yet wiring schemes: rather they represent a first step for arriving to wiring schemes. For this reason, the three input variables of a full adders, represented by a slash joining two dots in adjacent columns, are not specified, being assumed that they have to be found in the column of the sum output.

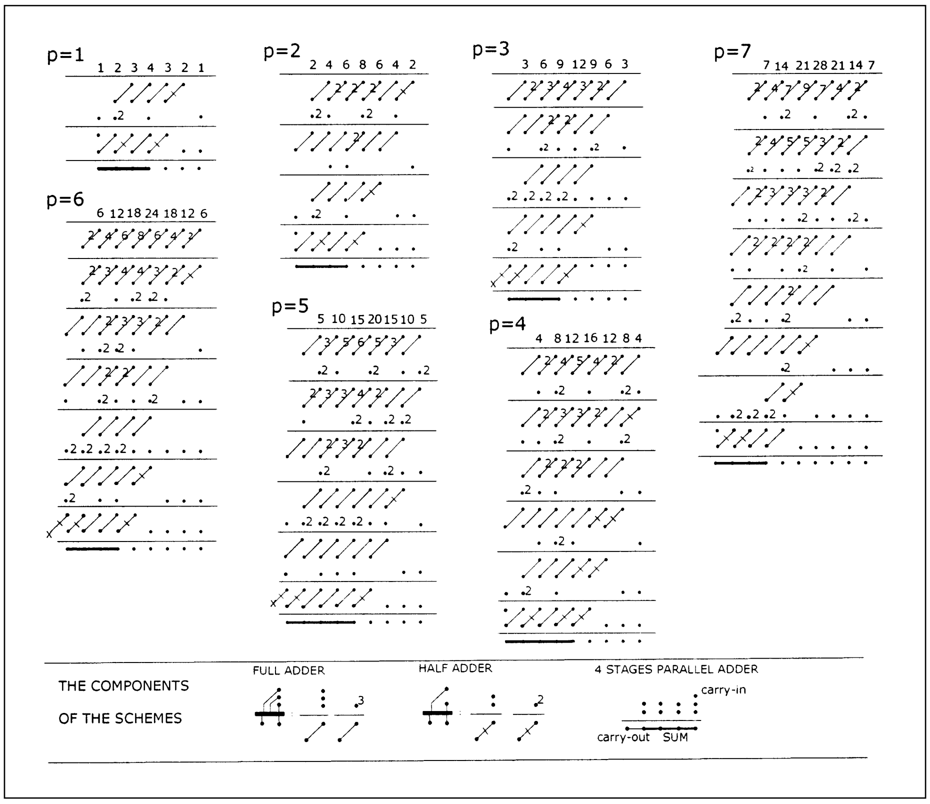


Fig. 3A. Dot schemes for obtaining the binary Column Partial Products for up to p=7.

We recall here that in the future IEEE Standard for decimal floating point [17] three length values are prescribed for the characteristic: 7, 16 and 34 BCD digits. The schemes of fig. 3A are therefore sufficient for the implementation of a parallel decimal multiplier for 7 digit factors. They could also be used for implementing smaller sized multipliers, as could be needed in generic embedded systems.

For larger characteristic's values, more schemes would be needed. It has been found convenient for such cases to provide a tool for obtaining their design by computer. This in order to get a faultless design: it is in effect rather easy to make mistakes in drawing the dot schemes by hand.

In the report [15], dedicated to the compact dot design method described previously, it is also shown that efficient design tools can be built for adders of a large number of binary addends, for parallel binary multipliers and for adders of products. The latter is what is needed for the design of Column Partial Products: such a program, for up to *p*=45, can be found in a web page [18].

Fig. 3A schemes have been compared with the schemes generated through the spreadsheet program that can be found in the said web page. The corresponding schemes have been found identical.

The algorithm implemented for the final stage has the role of compressing the 3 input rows to value 2, obtaining the value 1 for a number of adjacent columns, starting from the rightmost, least significant one. Starting from the right side we can obtain this result using a full adder for the first found 3 or a half adder for the first found 2. If the inputs following this full or half adder are 3 or 2, we must use full or respectively half adders for "making room" for the carries. If a 1 is found, it can accept a carry. The process is repeated for the following string of bits. The additional cost of the single bit generated in the last stage is given by the half adders placed for the first group of 2s.

An example of 2 initial half adders can be seen in fig. 3A $p=4$.

The same program computes also two important parameters: the total number compression stages (knowing the VLSI technology adopted it is then possible to compute the delay of each Column Partial Product) and the total number of full and half adders (allowing the computation of the silicon area). For simplicity reason and also for the small number of half adders, these are counted as full adders.

The following fig. 3B shows the dot schemes for computing the Column Partial Products for $p=8$ to $p=16$, to be used with fig.3A schemes for the design of a $16*16$ digit multiplier.

In Appendix B Table a number of parameters concerning the Column Partial Products for up to $p=34$ are shown.

IV - Converting the Column Partial Products to decimal

The Column Partial Products obtained as shown in the preceding chapter must be converted to decimal (BCD) in order to add them for obtaining the product: see fig. 1 general scheme.

The problem of converting numbers in a given base or radix into another one has been extensively treated , particularly in the early years of the computer era [1].

In our case we need a conversion algorithm that can be implemented in a rather fast circuit. We must therefore look for algorithms that operate with a high parallelism.

Searching in the literature, we found a suitable algorithm in a paper by Nicoud [4]. The algorithm can be implemented with a planar architecture, composed by identical cells, each connected only to the nearest neighboring cells or to the output-input nodes.

Fig. 4 shows a set of binary to decimal converters for decimal multipliers for factors of up to 37 digit. Each converter converts all Column Partial Products of a given length (in bit). See Appendix B Table, 3rd column and compare with fig. 4 schemes.

The operation of the BD converters is briefly exposed in Appendix A.

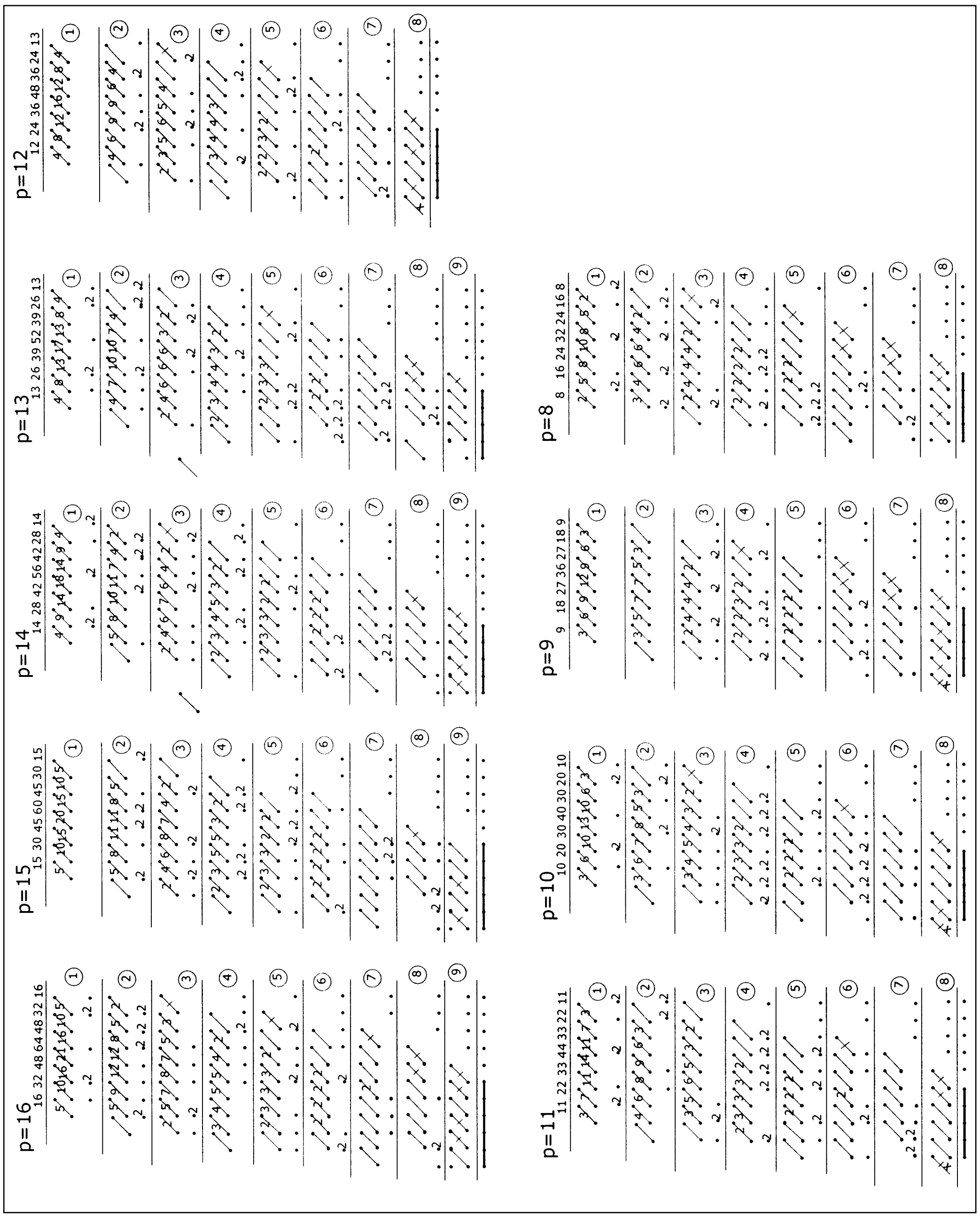


Fig. 3B. The dot schemes for computing the Column Partial Products for p=8 to p=16.

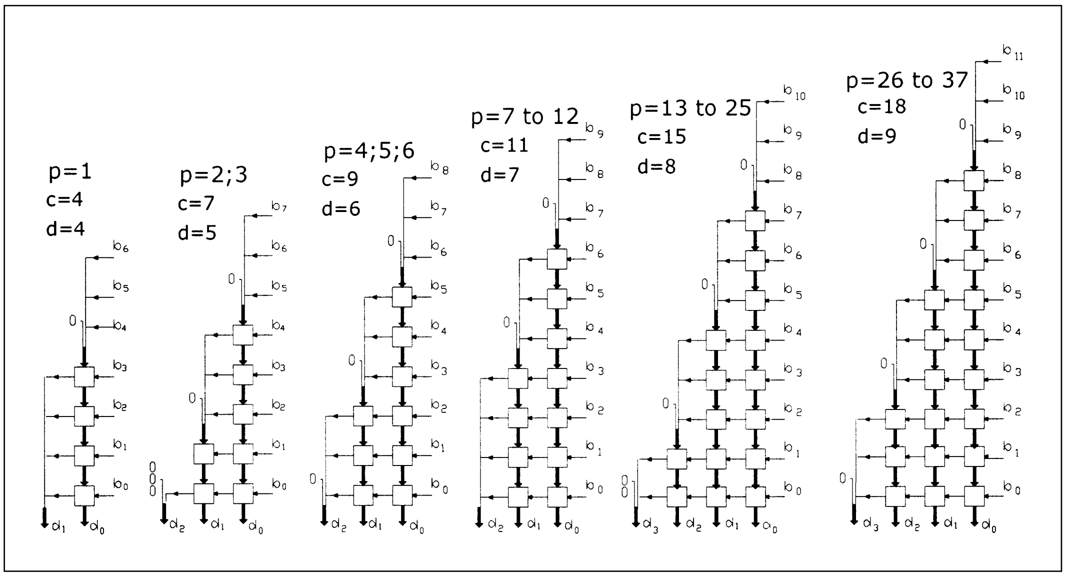


Fig. 4: The set of binary to decimal converters for decimal multipliers of up to 37 digits. **p**: number of digit partial products; **c**: number of cells; **d**: maximum number of cell-delays.

The range of *p* (the number of Digit Products composing a column of the multiplier array) is found in each scheme, along with *c* (the number of cells) and *d* (the number of cell delays included in the critical paths).

The silicon area of each cell is less than twice the area of a full adder. Its delay is approximately a half the delay of a full adder, see chapter VI. The total area of a converter is therefore rather small if compared with the area of the associated Column Partial Product generator, due to the fact that the number of BD cells is much smaller that the number of full adders needed to implement a Column Partial Product generator: from *13.8%* for *n=7* to *4.5%* for *n=34.* See the data in Appendix B Table.

The delay of each converter can be easily evaluated, due to its regular simple structure. The critical path in each converter can be expressed by the maximum number of cells connecting the most significant bits at the top of Fig. 4 schemes to the d_0 decimal digits at their bottom [4]. The numbers of such "levels" is written as *d* in same Fig. 4[2].

It can be seen that the delay introduced by the binary to decimal converters is smaller than the delays of the respective Column Partial Products: see chapter VI for numerical data.

[2] Fig. 4 has been derived from fig. 7 in the Nicoud paper [4], with a more compact symbolism.

It can be shown that fig.4 schemes are redundant: for instance input numbers composed by all 1's will never be applied, see Appendix B Table, 3rd column.

In Appendix A we give an algorithm allowing the design of converters whose conversion ranges match closely the needed ranges. This leads to the fig. A4 schemes. They offer smaller delays and number of cells: in Appendix B Table, columns headed "BD Conv B", we give for each *n* the following data: *number of cells, total number of cells in an n*n multiplier, number of cascaded stages.* In column "BD Conv A" the data are given for fig. 4 schemes.

The design of such new set of schemes has been obtained with the help of a design tool implemented with a spreadsheet. Such tool can be downloaded via the URL at [18].

V - Adding the Major Partial Products

As said in chapter II the Major Partial Products are generated by adding all the Column Partial Products. As shown in fig. 2 it is convenient to write the Column Partial Product in skew-tiled form, thus avoiding to write the 0's needed for their correct alignment and obtaining the product simply as the sum of three, or at most four, decimal numbers.

Appendix C shows the major partial products for multipliers of up to 18 digits factors, all digits in those factors having the value 9, in order to determine the maximum possible value of products. The products can be easily checked, their values (except for $n=1$) being composed by $n-1$ 9s followed by an 8 and by $n-1$ 0s and a final *1*.

Appendix B table shows the values of Column Partial Products in decimal and in binary form (for factors digit = 9). It is possible to compose with them the set of the Major Partial Products for any $n<35$, as done in the construction of Appendix C for $n<19$

Fig. 5 shows the Major Partial Products for $n=7$ and for $n=16$.

It can be seen that the number of Major Partial Products increases from *3* to *4* for multipliers of $n=13$. Note that 12*81=972 and that 13*81=1053. The fourth major partial product is composed by 0s and 1s for $12<p<25$. For $24<n<37$ the fourth Major Partial Product will be composed by 0s, 1s and 2s.

This is important when considering the problem of adding all the Major Partial Products for obtaining the final Product.

We have shown in [16] that the sum of several decimal numbers can be obtained with a hybrid approach similar to the one followed in this paper.

In the fig. 5, $n=16$ example we consider the set of Major Partial Products composed by 4 decimal numbers: three of them are composed by decimal digits that can have values from 0 to 9. The fourth number is composed from a central part with all digits that can at most be equal to 1, and two adjacent parts with all zeros.

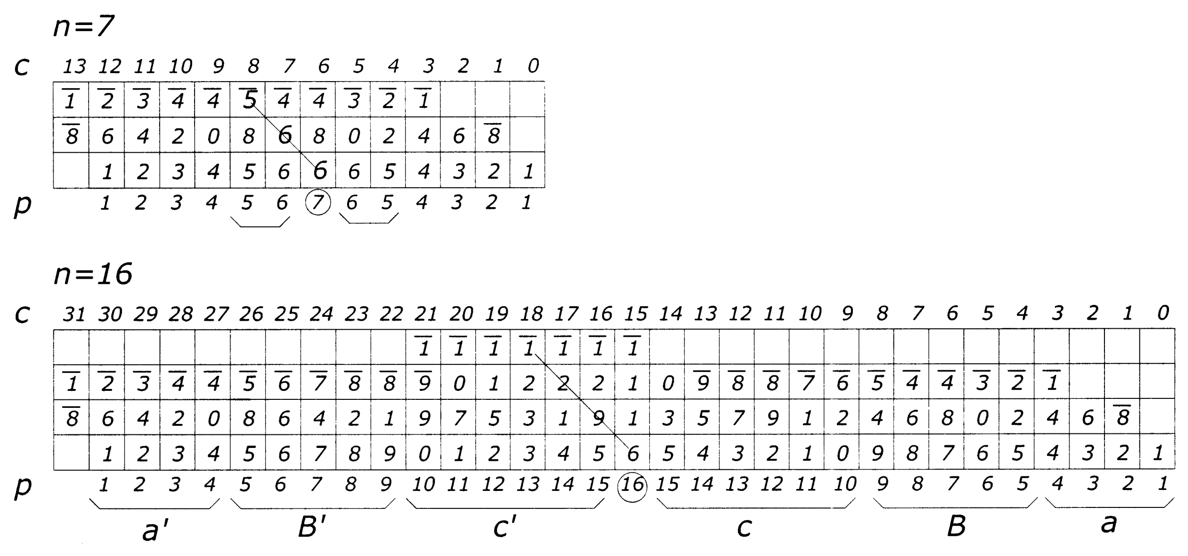


Fig. 5. The Major Partial Products arrays for n=16 and for n=7 (factor's digits equal to 9).

Note that in both cases in fig. 5 (*n=7* and *n=16*) the least significant digit of the Product is known (column *c=0*).

Columns *c=1* and *c=2* are composed by *2* digits. Columns *3* to *14* are composed by *3* digits; columns *15* to *21* by *4* digits; columns *22* to *30* by *3* digits; column *31*, the last, most significant, by *2* digits.

An important remark: when we consider the worst case (all digits of both factors equal to 9) it is certainly true that the most significant digit of each column sum is the maximum: for any other values of the factors digits those values will be at most equal to the worst case ones. This cannot be said for the other digits of the column sums: they can assume values greater or smaller than the their worst case one.

This can be easily checked through the spreadsheets that can be downloaded via the URL at [18], where two multipliers (8*8 and 16*16) have been simulated, showing also the set of Major Partial Products.

The most significant digits of the column sums are overlined in fig. 5. Such values will be used for optimizing the schemes obtaining the sums of the columns in the Major Product Arrays, see fig. 7.

Note that in column *2* and column *14* (fig. 5, *n=16*) the topmost digit is not overlined: they are not the most significant digits of a column sum, but the second

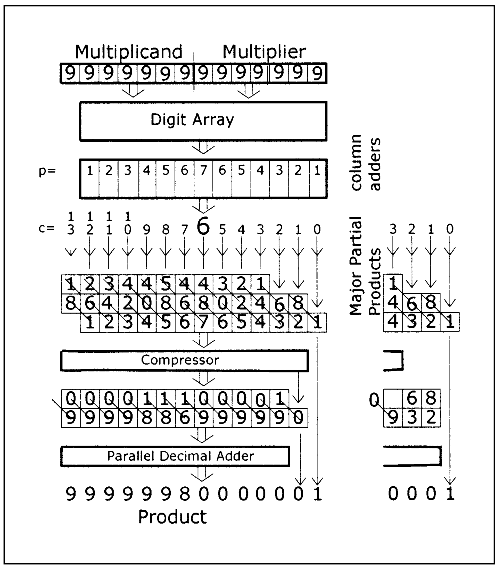


Fig. 6. A scheme for a 7*7 multiplier, where the addition of the Major Partial Products is shown in detail **p**: number of digit products; **c**: the weight of the column is 10^c. At the right: a variation in which the column c=1 and c=2 of the MPP array are unchanged. Each arrows at the output of column adders points to the least significant digit of the column sum placed in the bottom line of the Major Partial Products array.

one. The value assigned to them will therefore be 9 when considering the design of the Major Partial Product Adder schemes.

A scheme for adding the Major Partial Product is shown in fig. 6 for a $7*7$ digit multiplier.

Such a scheme shows in its upper part an array of $2n\text{-}1=13$ column adders (including the respective BD Converters), followed by the array of 3 Major Partial Products, whose sum is the Product.

The Product least significant digit coincides with the least significant digit of the third Major Partial Product, column $c=0$. Each of the remaining columns of the MPP array feeds a "decimal compressor". This generates a binary sum that is converted in decimal, as shown in the various schemes of fig. 7, with two decimal digits: d_0 with the same weight 10^k of the input digits, d_1 with weight 10^{k+1}.

Fig. 7 shows the schemes of four column compressors, where the components are the same as in fig. 3A with the addition of the basic BD Conversion cell (see Appendix A).

In all compressor schemes we start with a first phase in which, in all binary columns in parallel we add the binary values of the input digits (of same weight 10^k). The output is composed from two digits: d_0 with the same weight 10^k of the input digit, d_1 with a weight 10^{k+1}.

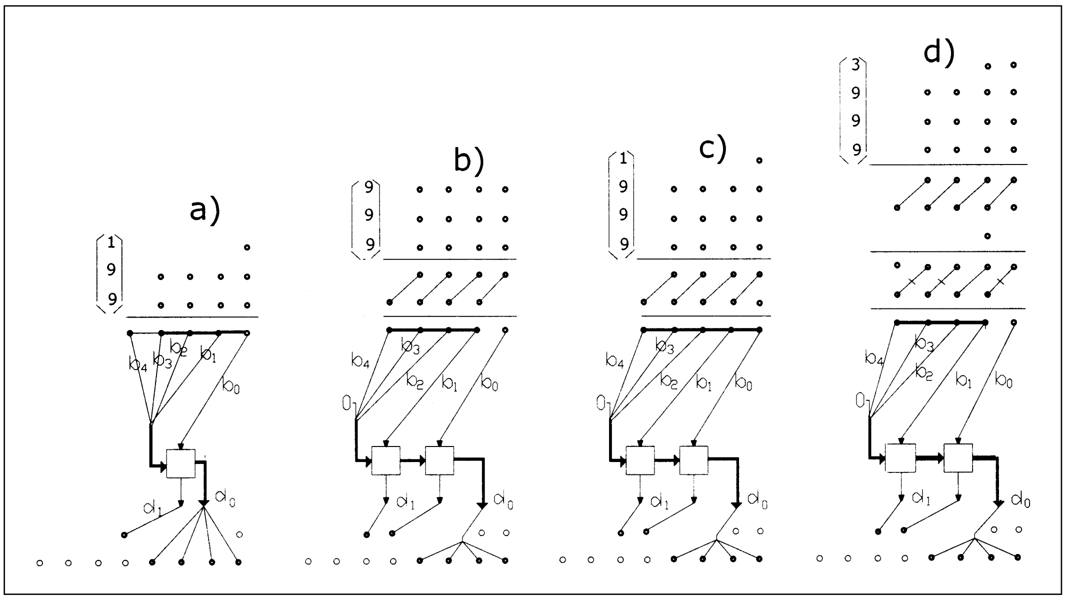


Fig. 7. Dot Schemes for compressing 3 or 4 digits columns of the Major Partial Products Array into decimal numbers.

In fig. 7a) scheme we need only a 4 bit parallel adder. We must add two BCD digits and a third one (the topmost) composed of a single bit, i.e. a digit that can assume only the values *0* or *1*. The maximum values of the three digits (1,9,9) is written at the left-top of the figure. The minimum sum value can be *0,* the three digits being all zeros, and it has to be noticed that in the case the value of the binary sum of the three digit is smaller than 10_{10} the action of the BD cell will be simply to generate a d_0 equal to the sum and a $d_1=0$.

Fig.7b) shows a compressor whose maximum input is (9, 9, 9): the corresponding binary sum will be $3*9=27=11011_2$. A 2-stage BD converter is needed.

Fig. 7c) has 4 digits input, the top one having at most the value *1*. The maximum digit values are (1,9,9,9): the corresponding binary value being $1+3*9=28=11100_2$. The BD conversion scheme is identical to the one in fig. 7b). The difference of the two schemes is in the binary adder: in fig. 7b) scheme we need a 4-stages adder, while a 5-stages adder is needed in fig. 7c) scheme. Note that in fig. 7b) scheme the carry-save stage generates a final "single bit": this doesn't happen in fig. 7c scheme.

In the last fig. 7d) scheme the maximum values of the three digits are (3,9,9,9). The corresponding sum is: $3+3*9=30=11110_2$. We have here two carry save stages, generating a single least significant bit (as in fig. 7b). The BD conversion is identical to the two previous schemes.

The above described decimal compressors can be used for transforming a set of 3 or 4 Major Partial Products into a set of two decimal numbers whose sum is the Product. We will therefore need, see fig. 6, a final Parallel Decimal Adder. For speed reason this will adopt a carry-look-ahead scheme.

It is possible to derive from fig. 7b) two schemes, having a slight smaller area, assuming that the top input digit requires only *3* or *2* bits, representing the values *7* or *3*. The corresponding circuits will require *1* or *2* half adders to replace full adders in the first reduction stage.

Note that generating the maximum values of Column Partial Product does not mean that the maximum values in the column of the Major Partial Products array reach the maximum 3*9+1 value. This value is certainly the upper limit in the Major Partial Product sum. Determining for all cases its real maximum could lead to some simplification. This problem will not be treated here.

VI - Delay and area

We give now data on delay and area on a 7*7 digit multiplier. This offers an opportunity to show how the various functional components are chosen: column adders from fig. 3A, BD Converters from fig. 4 or fig. A4, decimal compressors from fig. 7. They will be assembled as shown in fig. 1 and fig. 6.

Such data, and the following ones, have been obtained from the *STM 0.18mm library of standard cells and Synopsis Design Compiler.*

Delay and Area of the main component of the schemes are given in the following Table A:

	NOR	FA	HA	BD	cla4	cla5	cla6	cla7
ns	0.04				0.31	0.36	0.38	0.41
/um^2	197	90	50	168	614	823	1409	2999

NOR: generating the bits in the product array
FA, HA: Full and Half Adders
BD: Binary-to-Decimal conversion cell
Cla4: carry-look-ahead binary adders with 4 cells

Table A

The critical path

Let's consider fig. 8, where the scheme of fig. 6 has been redrawn representing in a single block the 13 column adders, the Major Partial Product Set and the Compressor. In each column we find: the number *p* of the digit product, the exponent *c* of the base 10 and, at the bottom, the delay; the *c=13* column was added to represent (see fig. 6) the delay of the last compressor: since the result of the addition cannot be greater than 9, the d_1 output of this compressor is always zero.

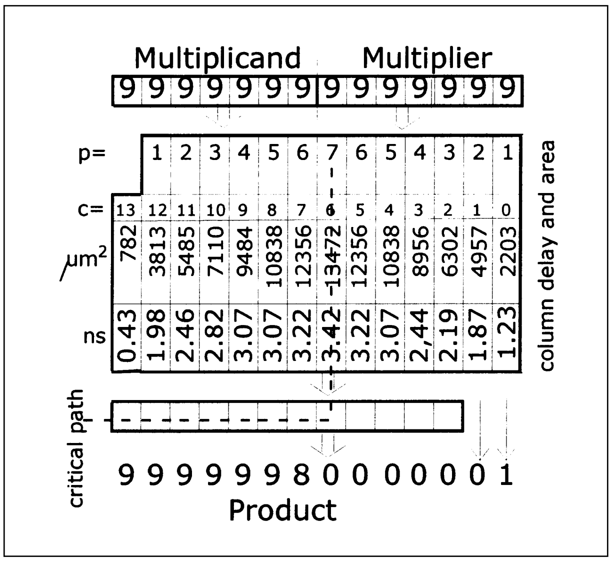


Fig. 8. The critical path is given by the column p=7 and the 8 most significant 8 stages of the Decimal Parallel Adder.

Since the two least significant digits are already known, we start the Decimal Adder from column *c=2*, the last stage being at column *c=13*. We compare the delays from column *c=2* to *c=7*, i.e. the central column with the maximum delay (3.42 ns). We compute the delay in the Decimal Adder from the first stage to the 5th stage (in column *c=7*). If this delay is smaller then the delay difference from the columns *7* and *3*, the critical path is composed of the central column and by the 8 most significant stages of the Decimal Adder. In the case considered here and using the data obtained from the said cell library the total delay in the critical path is *4.45 ns.*

In case the above condition is not met, the critical path is composed by one of the columns at the left of the central one, c_k, whose delay added to the delay of the stage of the Decimal Adder corresponding to the same column, is maximum.

The matching of the delays pattern of the column adders with the final decimal adder has only been sketched. For a better understanding simulation could help.

The area

The total area, excluding the final decimal adder (of 12 stages, see fig. 8) is $108953\mu m^2$. The area of a carry-look-ahead decimal adder of 12 digit is $23223\mu m^2$

Delay and area for the n=16 multiplier

Fig. 9 is a general scheme of a 16*16 digit decimal multiplier, similar to the preceding fig. 6 scheme. The outputs of the 31 column adders compose an array of 4 decimal numbers shown in fig. 5 and reproduced in fig. 9. The outputs of columns 12 to 18 require 4 digits each, the most significant being at most *1*.

As a consequence the compressors for columns 15 to 21 must adopt the scheme of fig. 7C (requiring a 5 stage carry-look-ahead binary adder). The final decimal parallel adder will require 30 stages. The following fig. 8B contains for each column of the multiplier the delay and the area of circuits of the Digit Array (simple NORs), of the Column Adders (shown in fig. 3A and fig. 3B) and the associated BD converters fig. 4A) and of the Compressor section (fig. 7).

Note that column *c=31* is composed only of a compressor, acting on the carries generated in the Major Partial Products.

The product is obtained via the Decimal Parallel Adder, assumed to be a carry-look-ahead type for speed reason. The area of the multiplier part represented in fig. 9 is $511638\mu m^2$; the area of the 30 stages final decimal adders is $56660\mu m^2$; the total area is therefore $568298\mu m^2$.

The maximum delay at the inputs to the decimal adder is the one in column *c=15*, of *4.01 ns.* The total delay of the decimal adders in 30 stages is 1.59*ns*. Due to the delays in the columns output from *c=2* to *c=15* it can be assumed, as done for fig.8 that the critical path is composed of two parts: the path through column *c=15* (4.01*ns*) and the paths fin the last 17 stages of the decimal adder (*1.38ns*) for a total of *5.39ns*.

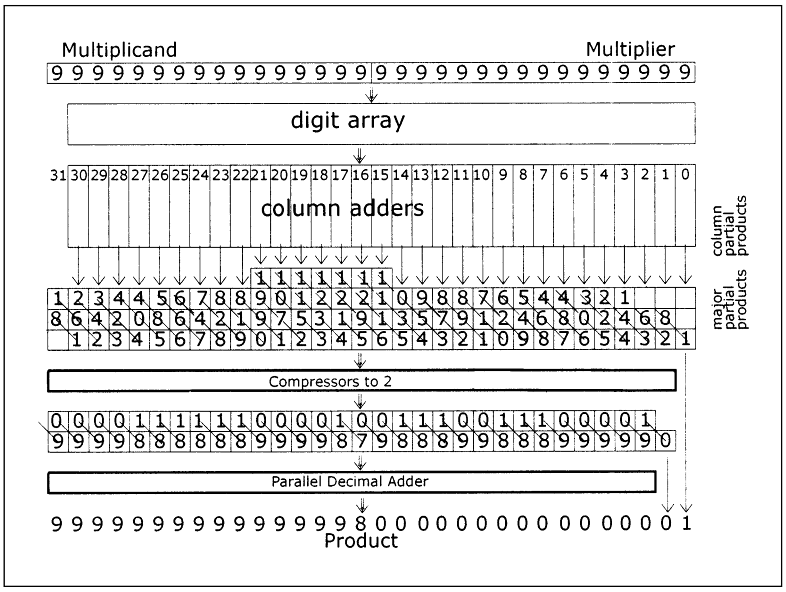


Fig. 9. The scheme of a 16*16 digit multiplier where the compression of the Major Partial Products to two equivalent BCD numbers is shown, followed by a parallel decimal multiplier obtaining the Product.

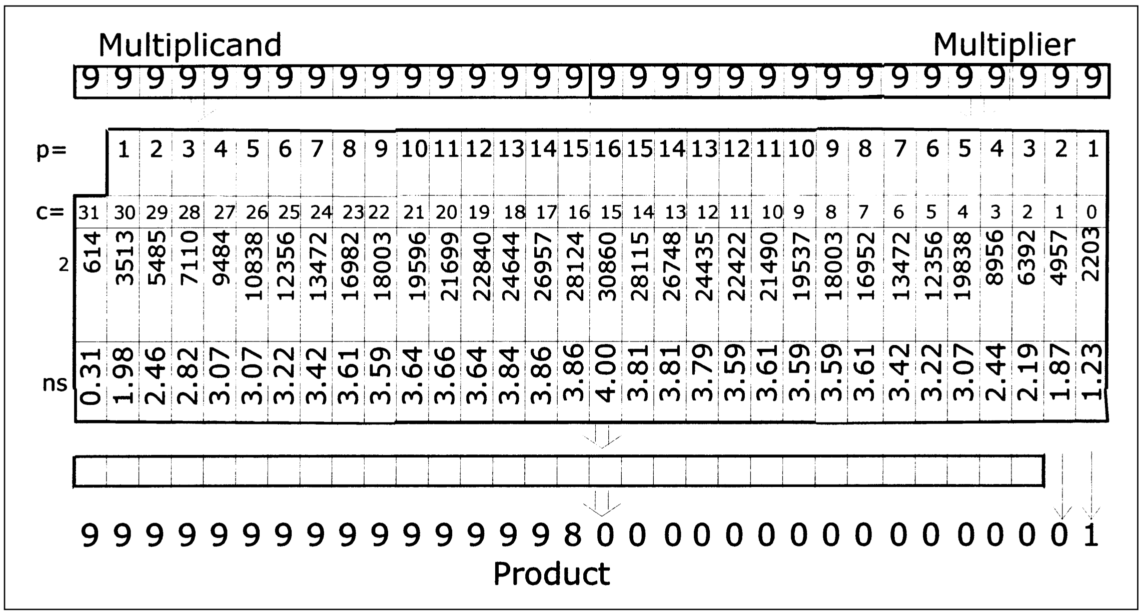


Fig. 10. Delay and area in a 16*16 digit multiplier.

Comparisons: a first comparison can be made with a binary parallel multiplier for factors with an equal number of bits. The main data, i.e. the total number of full adder and the number of stages in shown in the Appendix B Table. Our decimal multiplier uses a slightly larger area (less that 10%) and gives a delay about 50% higher. Taking into account the totally different architectures the decimal scheme seems quite acceptable.

A parallel decimal multiplier has been recently presented [19]. This multiplier was designed with a purely decimal arithmetic. The factors digits are recoded for reducing the number of partial product pre-computations. The design is done in a *90nm* technology. The delay on the critical path is *2.65ns* and the total area is *300,000*μm^2. This multiplier and the one described in this paper are, to our knowledge, the only two, so far published, proposing a parallel decimal scheme. Their architectures appear totally different. Comparing the data of our scheme, based on a 0,18μm CMOS technology with the data of the just mentioned authors, based on a 0.09μm technology doesn't seem possible. We will redesign in the near future our scheme on the latter technology.

VII - Conclusion

We have shown how a fully parallel decimal multiplier can be designed, with a hybrid approach using the binary technology for adding the values of Digit Partial Products, composing Columns Partial Products. Binary to Decimal converters are used for converting to decimal their binary output.

The summation of such Partial Product in order to obtain the Product is composed by two phases, the first obtaining the set of Major Partial Products by "tiling" the "skewed" Column Partial Products, the second by adding two or at most four decimal numbers.

Some software tools have been built for obtaining the Column Partial Products.

It has been found, see Appendix B Table, that the size of a decimal multiplier is slightly larger than the size of a binary multiplies for factors of equal bit-size. The total area and delay has been determined to be 568298μm^2 and 5.39*ns.*

It has been pointed out that computing the Column Partial Products requires a considerable silicon area, while a relatively smaller one is required by the other functions. The Binary to Digital converters require a relatively small area and delay.

Further problems have been identified such as: the partition into pipelined sections in order to increase the throughput (accepting the connected latency); the composition of small *n* multipliers for obtaining multiplications of larger factors, the partition of a large multiplier into smaller parts to be used for the same purpose.

A faster BD conversion scheme is highly desirable.

Acknowledgment

The author is grateful to Alberto Nannarelli, at the Danish Technical University, Kongens Lyngby, Denmark, for providing advices and most of the area and delay data of the basic components, and to students of the ALaRI Institute, University of Italian Switzerland, Lugano, for simulation work.

REFERENCES

[1] Couleur, J.F.: *BIDEC - A Binary-to-Decimal or Decimal-to-Binary Converter,* IRE Trans. on Electronic Computers, vol. C-20, pp. 313-316, Dec. 1958.

[2] Dadda, L.: *Some schemes for parallel multipliers*, Alta Frequenza, vol. 19, pp. 349-356, March, 1965.

[3] Habibi, A., Wintz, P.A.: *Fast Multipliers*, IEEE Trans on Computers, Febr. 1970.

[4] Nicoud, J.D.: *Iterative Arrays for Radix Conversion,* IEEE Trans. Computers, pp.1479-1489, vol. C-20, n.12, Nov. 1971.

[5] *Computer Design Development Principal Papers,* E.E. Swartzlander, Editor, Hayden Book Co, Rochelle Park, N.J., 1976.

[6] Dadda, L.: *On parallel multipliers*. Alta Frequenza, vol. 45, pp. 574-580, 1976.

[7] Earl E. Swartzlander, Jr.: *Merged Arithmetic for Signal Processing*, Proceedings of the 4th Symposium on Computer Arithmetic, pp. 239-244, 1978. A revised version (*Merged Arithmetic)* is in IEEE Transaction on Computer Arithmetic, vol. C-29, 946-950, 1980.

[8] Gajski, D.D.: *Parallel Compressors,* IEEE Trans. on Computers, Vol. C-29, n. 5, May 1980.

[9] Muller, J.M., *Arithmétique des ordinateurs*, Ed. Masson, Paris, 1989.

[10] Koren, I: *Computer Arithmetic Algorithms,* Prentice Hall, Englewood Cliffs, 1993.

[11] Gok, M., Schulte, M.J., Balzola. P. I.: *Efficient Integer Multiplication Overflow Detection Circuits,* Proc. of the Thirty Fifth Asilomar Conference on Signal, Systems and Computers, pp. 1661-1665, 2001.

[12] Erle, M.A., Schulte, M.J.: *Decimal Multiplication Via Carry-Save Addition*, Proc. Applicatio-Specific Systems, Architectures and Processors (ASAP'03).

[13] Erle, M.A., Schwarz, E.M. Schulte, M.J.: *Decimal Multiplication with efficient Partial Product Generation*, Proc. 17th Symposium on Computer Arithmetic, Cape Cod, MA, June 27-29, 2005.

[14] Kenney, R.D., Schulte, M.J.: *High-Speed Multioperand Decimal Adders*, IEEE Trans. on Computers, Vol. 54, n. 8, Aug. 2005.

[15] Dadda, L.: *A compact dot notation for the design of binary adders, multipliers and adders of products.* AlaRI internal report, Dec. 2005.

[16] Dadda, L.: *Multioperand Decimal Parallel Adders: a mixed binary-BCD approach,* ALaRI internal report, Jan. 2006.

[17] Hursley: *Decimal Arithmetic Specification,* http://www2.hursley.ibm.com/decimal/decbits.html

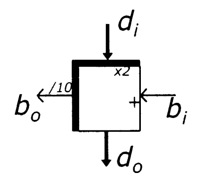
[18] Dadda, L.: *Spreadsheet tools for the design of a parallel decimal multiplier*, AlaRI internal report, Dec. 2005. http://www.alari.ch/people/dadda/DecimalMultiplier.htm

[19] Lang, T., Nannarelli, A.: *A radix-10 Combinational Multiplier*, 40th Asilomar Conference on Signals, Systems and Computers. Oct. 29 - Nov. 1, 2006.

Appendix A: ***Binary to Decimal Conversion***

In Chapter IV we have described the binary to decimal converters used for obtaining the Column Partial Products. We expose here, for completeness, the background of such converters. The exposition differs from the original one by Nicoud [4] mostly for adopting a different more compact notation, allowing an easier design and checking of the operation. We also prove directly that a linear array of basic cells obtains, in binary, the division of a given binary integer by ten. It is well known that for converting a number N_s in base s into its equivalent N_t in another base t, the basic step is to divide N_s by t, the remainder being a digit of N_t.

The basic cell for our case (s=2, t=10_{10}) is depicted in Fig. A1:



d_i is the input decimal digit; we assume that it is a BCD digit (it could be coded in any other way)
d_o is the output decimal digit
b_i is the binary input digit
b_o is the binary output digit

Fig. A1. The basic conversion cell

The algorithm implemented in the cell operates as follows.

```
– The sum S of the inputs is: $S = 2*d_i + b_i$.
– If $S > 9$
        $S = S - 10;\ b_o = 1;\ d_o = S$
  If $S<10_{10}$
        $b_o = 0;\ d_o = S$
```

The above algorithm can easily by represented by a truth table [4] and implemented with a combinational network.

A converter based on a single cell, valid for input numbers of only 4 bits, is shown in Fig. A2, a). The three most significant input bits, b_3, b_2 and b_1 compose, with an additional *0* the BCD decimal input digit of the cell, b_0 being applied at the binary input. The decimal output d_0 is the least significant decimal digit, the binary output, associated with three *0*'s is the most significant output digit. The operation of the circuit can be easily verified: e.g. (b_3 b_2 b_1 b_0) = (1 1 1 1) generate 15.

Fig. A2, b) to d), represent converters for numbers: $10000_2=16_{10}$ to $10011_2=19_{10}$. The input numbers are characterized by the first four bits representing 8 or 9: they can therefore be used as the decimal input to the converters

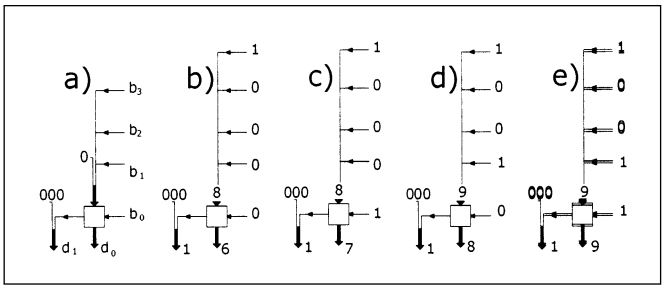


Fig. A2. Monocell BD converters, for number from 00_{10} to 19_{10}.

Fig. A3 shows some converter schemes each composed by a column of four cells.

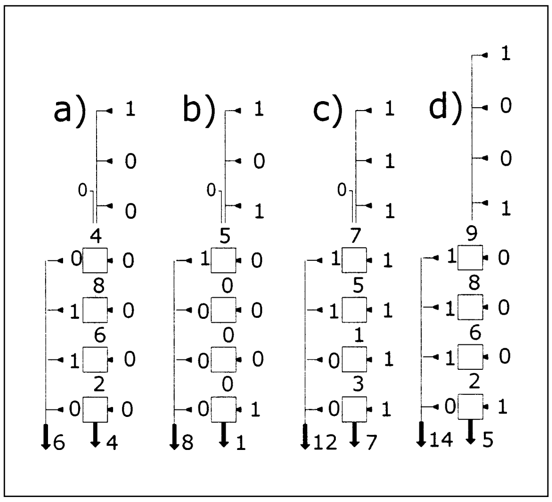


Fig. A3. **a**) a linear array of 4 cells obtain the quotient 0110_2, i.e. 6_{10}, and a remainder of 4_{10} from the number 100 0000_2, i.e. 64_{10}; **b**) the input 101 0001_2 is the value 9*9 of a BCD digit array; **c**), **d**) two more examples.

In fig. A3, a) the binary input is 100 0000_2. The converter operation is shown by writing the values at the inputs and at the outputs of each cell, according to the described algorithm. The four binary outputs represents the digit $6=0110_2$. The first 1 represents a number equal to $10*2^2$, the second the value $10*2^1$, both subtracted to the input binary input to the array. The total of the number subtracted, 40+20=60, is the *maximum multiple of the base ten smaller that 64, i.e. the quotient of 64/10, since the remainder 4 is smaller than ten. This property holds for any binary number, of any length (provided a suitable number of cells is used in the array).*

We show also an algorithm for obtaining a set of schemes, using a minimum number of cells.and to be used instead of the fig. 4 schemes.

Fig. A3, b) shows the case of input equal to 101 0001_2 = 81_{10}, i.e. the conversion of a single digit array column partial product.

Consider now fig. A3, c): the output binary is, in decimal, 12_{10}, the quotient of 127/10. In order to obtain the two digits 1 and 2, a further converting column is required, composed as in fig. A2, a) scheme. The same holds for fig. A3, d) scheme.

Fig. 4 schemes (chapter IV) were obtained on the basis of the bit length *n* of the Column Partial Products, assuming a scheme capable of converting a binary number composed of *n* 1s, i.e. the maximum numbers of such length, obviously

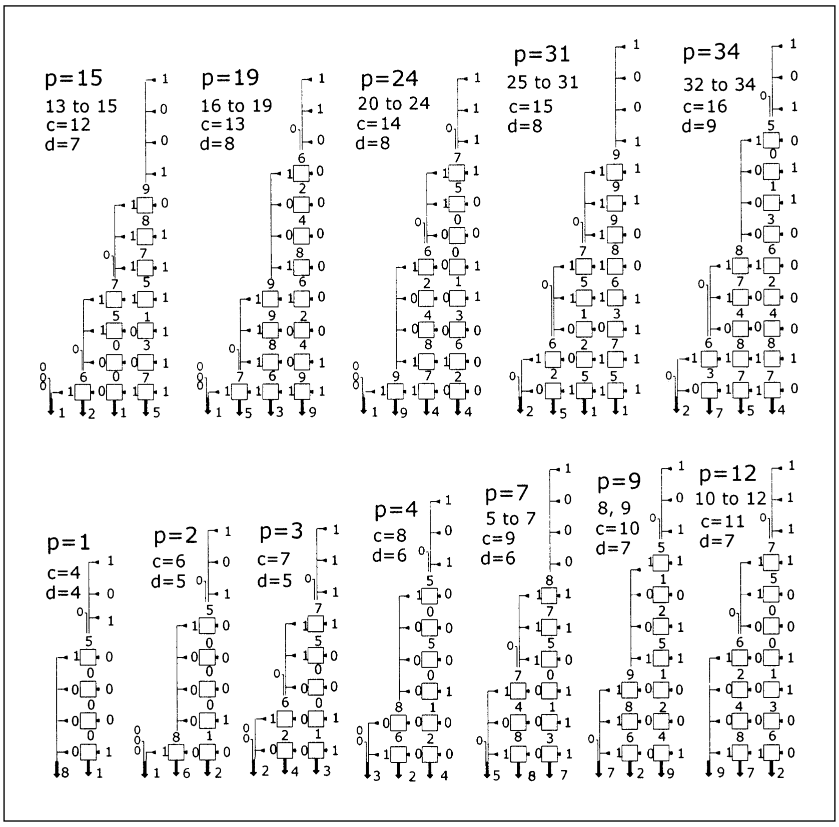


Fig. A4. The set of minimal binary to decimal converters for decimal multipliers of up to 34 digits; **p**: number of digit partial products; **c**: number of cells; **d**: maximum delay in number of cell-delays. The range of p=12 includes p=11 and p=10. Similarly for all other schemes.

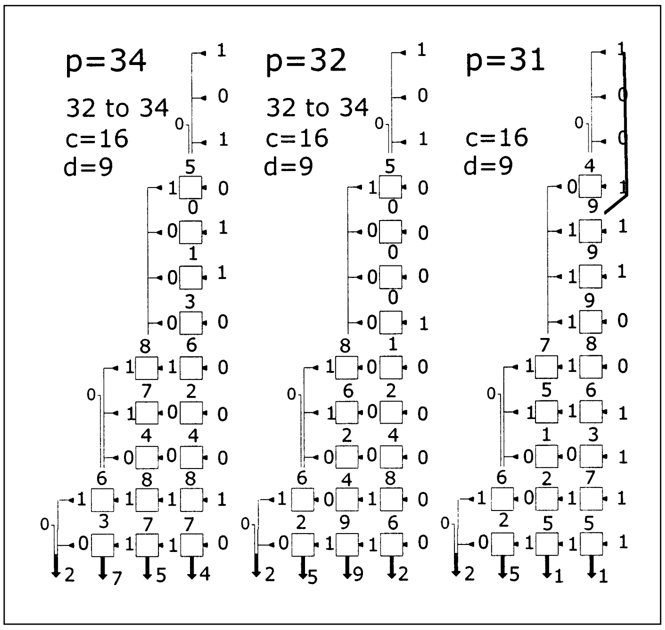


Fig. A5. Obtaining an optimal BD converter for p=31 from a BD converters for p=34.

applicable to all Column Partial Products with such length. Another criterion is to adopt converters on the basis of their actual maximum values, listed in Appendix B Table, in column "binary". This allows to determining schemes that can be implemented with smaller number of cells. The algorithm to be used can be described as follows:

– consider all the columns of binary numbers generated within the converters, see in fig. A4 the *p=34* scheme: the first (from the right) column is the input number, the second is the quotient of the division by ten of such number, the third is the quotient of the division by ten of the second number, etc. The final remainders represent the result, i.e. the decimal equivalent of the binary input.

– in each column consider the first four topmost digits: if their value is 1001_2 or 1000_2, the corresponding variables compose the BCD digit to input to the first topmost cell. Otherwise (this is the case in *p=34* scheme), send to the same cell the first three bits and add a zero bit to compose the BCD digit.

The above algorithm must be applied to all 34 Column Partial Products of Appendix B Table, obtaining the above fig. A4 schemes.

In order to help this process we have developed a simulation of the converters using a spreadsheet: it can be downloaded from [18]. In this process we start with the highest *p=34* case as given by the top-right scheme in fig. A4. It has been drawn for the input from Appendix A Table *p=34* row, 3rd column, with the above rules. Replace in it the input with *p=33* from the table, obtain all the new values, see that the rules are still satisfied. So with the *p=32* values.

With *p=31* such rules are not more satisfied. Keep the previous scheme, valid for *p=34 to 32*, and transform the scheme by deleting the top cell: the input value being 1001 it must be input in the new first cell. Compute all the remaining values obtaining the *p=31* scheme; check that it's valid for *n=25 to 31*.

For *p=24* it will be necessary to remove a cell from the leftmost cell column, obtaining a single cell in it.

A cell will be removed any time a new scheme is created.

The selective removal of cells is the characteristic of the method, that leads to a sequence of schemes ending with the case *p=1*.

The set of fig. A4 schemes can be considered an optimal solution to the conversion problem.

Appendix B

Columns Product Sums /Column Partial Products (factor's digits=9)									BD Conv A			BD Conv B			Binary *multiplier*	
n	dec.	binary	n bits	Stages	last row length	last 1's	full adders	mult. tot. fa	cells	mult. tot. cells	stages	cells	mult. tot. cells	stages	mult. tot. fa	stages
1	81	101 0001	7	2	7	3	8	8	4	4	4	4	4	4	*6*	*2*
2	162	1010 0010	8	4	8	4	24	40	6	14	5	6	14	5	*42*	*5*
3	243	1111 0011	8	5	9	4	38	102	7	27	5	7	27	5	*110*	*6*
4	324	1 0100 0100	9	6	9	4	58	198	8	42	6	8	42	6	*210*	*7*
5	405	1 1001 0101	9	6	10	4	69	325	9	59	6	9	59	6	*342*	*7*
6	486	1 1100 0110	9	7	10	5	86	480	9	77	6	9	77	6	*506*	*8*
7	567	10 0011 0111	10	8	10	6	102	668	11	97	7	9	95	6	*702*	*8*
8	648	10 1000 1000	10	8	10	4	122	892	11	119	7	10	114	7	*930*	*8*
9	729	10 1101 1001	10	8	11	5	138	1152	11	141	7	10	134	7	*1190*	*9*
10	810	11 0010 1010	10	8	11	5	149	1439	11	163	7	11	155	7	*1482*	*9*
11	891	11 0111 1011	10	8	11	4	164	1752	11	185	7	11	177	7	*1806*	*9*
12	972	11 1100 1100	10	8	11	5	181	2097	11	207	7	11	199	7	*2162*	*10*
13	1053	100 0001 1101	11	9	11	6	197	2475	15	233	8	12	222	7	*2550*	*10*
14	1134	100 0110 1110	11	9	11	5	214	2886	15	263	8	12	246	7	*2970*	*10*
15	1215	100 1011 1111	11	9	11	6	227	3327	15	293	8	12	270	7	*3422*	*10*
16	1296	101 0001 0000	11	9	11	5	248	3802	15	323	8	13	295	8	*3906*	*10*
17	1377	101 0110 0001	11	10	11	6	264	4314	15	353	8	13	321	8	*4422*	*11*
18	1458	101 1011 0010	11	10	12	6	282	4860	15	383	8	13	347	8	*4970*	*11*
19	1539	110 0000 0011	11	10	12	6	295	5437	15	413	8	13	373	8	*5550*	*11*
20	1620	110 0101 0100	11	9	12	5	308	6040	15	443	8	14	400	8	*6162*	*11*
21	1701	110 1010 0101	11	10	12	6	325	6673	15	473	8	14	428	8	*6806*	*11*
22	1782	110 1111 0110	11	10	12	5	338	7336	15	503	8	14	456	8	*7482*	*11*
23	1863	111 0100 0111	11	10	12	5	357	8031	15	533	8	14	484	8	*8190*	*11*
24	1944	111 1001 1000	11	10	12	6	373	8761	15	563	8	14	512	8	*8930*	*11*
25	2025	111 1110 1001	11	11	12	7	388	9522	15	593	8	15	541	8	*9702*	*11*
26	2106	1000 0011 1010	12	11	12	7	405	10315	18	626	9	15	571	8	*10506*	*11*
27	2187	1000 1000 1011	12	11	12	6	416	11136	18	662	9	15	601	8	*11342*	*11*
28	2268	1000 1101 1100	12	11	12	6	438	11990	18	698	9	15	631	8	*12210*	*11*
29	2349	1001 0010 1101	12	11	12	6	450	12878	18	734	9	15	661	8	*13110*	*11*
30	2430	1001 0111 1110	12	11	12	7	467	13795	18	770	9	15	691	8		
31	2511	1001 1100 1111	12	11	12	6	480	14742	18	806	9	15	721	8		
32	2592	1010 0010 0000	12	11	12	5	504	15726	18	842	9	16	752	9		
33	2673	1010 0111 0001	12	11	12	6	517	16747	18	878	9	16	784	9		
34	2754	1010 1100 0010	12	11	12	6	535	17799	18	914	9	16	816	9		
123	9963	10 0110 1110 1011														

The above Table summarizes data on the Parallel Decimal Multiplier described in the main text.

The data concern multipliers for up to 34 BCD digits in each factor (34 digits being the maximum length of the characteristic prescribed in the new (draft) IEEE Standard for floating point decimal numbers).

- 1st column **n** gives is the length of the factors.
- 2nd column "decimal" shows the maximum value of the Column Partial Products, for each **n**, i.e. the values corresponding to factors composed by all 9's.
- 3rd column **"binary"** gives the same values in binary.
- 4th column **"bits"** gives the number of bits of the Column Partial Product.
- 5th column **"stages"** gives the number of cascaded logical stages of the compression process, corresponding to a delay of a full adder for each stage.
- 6th column **"last row length"** gives the length of the last rows of the output of *Products Adders* in fig. 3A and fig. 3B, obtaining the binary values of each Column Partial Products.
- 7th column **"last 1s"** shows the number of the non-redundant least significant part of the Column Partial Products, generated by the compression stages, composed by single bits only. The remaining most significant part has to be computed by carry propagating binary adders.
- 8th column **"full adders"** gives the number of full adders (and half adders) composing the compression stages of each *Column Partial Product*, of length **n.**
- 9th column **"mult. tot. fa"** gives the total numbers of full (and half) adders needed in a decimal **n*n** multiplier (see Fig. 1).

The following three columns under the heading **"BD Conv. A"** give the main data on the *binary-to-decimal converters* used to obtain the decimal values of the Column Partial Products, see Fig. 4.

- 10th column **"cells"** gives the number of cells composing a converter, for each **n**
- 11th column **"mult. tot. cells"** contains the total number of cells for a decimal **n*n** multiplier.
- The 10th column **"stages"** gives the number of logical stage in each converter.

The above two data allows a reasonable good evaluation of the *area* and of the *delay* for each converter and for the whole multiplier.

The following three columns under the heading "BD **Conv, B"** give the same data for the BD converters illustrated in Appendix A, fig. A4.

The last two columns give data concerning binary parallel multipliers for factors of 4***n** bits each. The data show that the total number of full adders for computing all the Column Partial Products is slightly smaller that the number of full adder in a parallel binary multipliers.

The decimal multiplier needs also the Adder of the Major Partial Products, generating the *2*n* digit products. The amount of hardware needed can be obtained from [14] for each of the three main methods there suggested, or from fig. 6 for the method outlined in chapter V.

An **“anomaly”** can be noticed in 5th column (stages) row n=20: the value 9 is preceded by three values 10 and followed by four values 10. Is this an error in the program? An accurate analysis of the programs doesn’t reveal any error: the result is obtained via a proved algorithm. A possible objection: the program for n=20 could be used for n=17, 18 and 19, obtaining schemes with 9 stages in the compression stages, i.e. a faster circuit. This can certainly be done, but at a cost given by the greater number of full adders. Similar anomalies has been found in other programs as well.

The said anomaly is caused by the remainders of the division by three in the last compression stages, where the values of the quotients have values 1 or 2.

Appendix C: *Major Partial Products of multipliers of up to 18 digit factors*

1 **8** **1**	***10*** 1234456**7**876544321 864208642**1**246802468 123456789**0**987654321
2 **1** 8**6**8 1**2**1	***11*** 12344567**8**8876544321 864208642**1**9**1**246802468 1234567890**1**0987654321
3 1**2**1 86**4**68 12**3**21	***12*** 123445678**8**9**8**876544321 8642086421**9**7**9**1246802468 12345678901**2**10987654321
4 12**3**21 864**2**468 123**4**321	***13*** **1** 12344567889**0**98876544321 86420864219**7**5**7**91246802468 123456789012**3**210987654321
5 123**4**321 8642**0**2468 1234**5**4321	***14*** 1**1**1 123445678890**1**098876544321 864208642197**5**3**5**791246802468 1234567890123**4**3210987654321
6 1234**4**4321 86420**8**02468 12345**6**54321	***15*** 11**1**11 1234456788901**2**1098876544321 8642086421975**3**1**3**5791246802468 12345678901234**5**43210987654321
7 12344**5**44321 8642086**8**02468 123456**7**654321	***16*** 111**1**111 12344567889012**2**21098876544321 86420864219753**1**9**1**35791246802468 0123456789012345**6**543210987654321
8 123445**6**544321 8642086**4**6802468 1234567**8**7654321	***17*** 1111**1**1111 123445678890122**3**221098876544321 864208642197531**9**7**9**135791246802468 1234567890123456**7**6543210987654321
9 1234456**7**6544321 86420864**2**46802468 12345678**9**87654321	***18*** 11111**1**11111 1234456788901223**4**3221098876544321 8642086421975319**7**5**7**9135791246802468 12345678901234567**8**76543210987654321

For obtaining the set of Major Partial Products for n>18, see the spreadsheet program designed for the purpose downloadable from [18].