Rendiconti
Accademia Nazionale delle Scienze detta dei XL
Memorie di Scienze Fisiche e Naturali
127° (2009), Vol. XXXIII, P. II, t. I, pp. 47-62

LUIGI DADDA*

A new notation for arithmetic array design and its application to binary parallel multipliers

I - Introduction

The design of arithmetic operators (binary or decimal multi-operand adders, multipliers) is often done by using some kind of arithmetic array. The design process starts with an initial array composed from a number of rows that is then processed in a sequence of stages in which the number of rows is gradually reduced, until an array of two rows is obtained. The process performed in each stage is done via carry-free addition, and a single carry propagating addition obtains the final result. This is called the *reduction process.*

A known notation for such task is the *dot notation* introduced in 1965 [1]. It has been used by other authors [4, 6-8]. In 2005 a variant of such notation, the compact dot notation has been proposed and used [2, 3]. It leads to an array of relatively small numbers and consequently to the adoption of spreadsheets for the implementation of design algorithms.

In the reduction process it happens that in some stages the number of output rows is slightly greater than the one expected, leading to an increase of the total number of stages of the reduction process, consequently to a greater total delay. We will examine the origin of this *row overflow* showing that it happens in the reduction algorithm when some peculiar values of the involved parameters, not easily perceivable, occur. The scope of this paper is to analyze such problem and to show a solution.

The method will be applied to a notable example, the binary parallel multiplier.

Since we will adopt a notation not commonly used, we present first a short survey of the existing notation, as an introduction to new proposed notations.

* Uno dei XL. Politecnico di Milano. E-mail: luigi.dadda@polimi.it

II - Dot Notation, Compact Dot Notation, Cell Notation, Spreadsheet

The *Dot Notation* is used in fig.1a for representing the addition of 10 bit of same weight, i.e. a column of 10 bit.

In fig. 1b the same problem is shown with a different representation, the *Compact Dot Notation*. A single dot, associated with an integer *N* is used to represent a column of variables, *N=10* in the figure. A single full adder associated with an integer F, represents F full adders. The second stage in fig.1a is composed with 3 full adders; in fig. 1b with a single full adder associated with 3.

Note that, for practical reasons, the associated number is reproduced twice, close to the Sum and to the Carry full adder's outputs. Note that a single full adder can be represented without number or with an associated 1. It is clear from fig. 1b why this method was named *Compact Dot notation*.

In fig. 1a the basic rule is: partition the input dots (10) in triplets and feed each full adder in the next stage with a triplet; transfer with no processing the remaining dot(s).

In fig. 1b the rule is: divide the number of the input dots by 3 and place the integer quotient close to the Sum and to the Carry outputs of a full adder in the second stage; the remainder, if any, being represented by a dot (with a 2 if the corresponding remainder is 2). An integer quotient of N/3 can be written as $\lfloor N/3 \rfloor$ and the remainder as N_{MOD3}. A third stage is needed to obtain two rows, using two full adders.

The non-redundant final binary number is obtained via a Carry Propagating Adder, represented in the figure by a thick line joining all the output Sum bits.

It is important to understand that the above schemes represent the hardware composition of a circuit generating four bits representing the number of inputs valued 1. The scheme doesn't represent the numbers themselves.

Fig. 1c represents an adder of 10 binary numbers of 4 bit. The most important difference from fig.1a is the handling of carries. We need 6 stages instead of 3. Note also that the scheme has an interesting property: 3 of the final 7 bit result are already computed in the stages preceding the output stage. The length of the binary carry propagating adder is composed from 4 stages only (note that the most significant bit is represented by the carry from the fourth stage of the parallel adder).

A third method for designing such a system is represented in fig. 2a, where we use an array composed by the "multiplicity factors" of the various groups of variables of fig. 1b, avoiding the explicit graphical symbols for the full (or half) adders. We can operate on such an array as in the case of Compact Dot schemes. The numerical operations are executed "by hand", as in the compact dot-notation scheme.

In fig. 2b the fig. 1b scheme is represented on a spreadsheet. For addressing each cell on a spreadsheet it is used a method based on alphabetic letters to indicate a column and on decimal integers to indicate a row. As an example cell, AW67 contains a 3.

The operations on such an array are performed via programs written in the

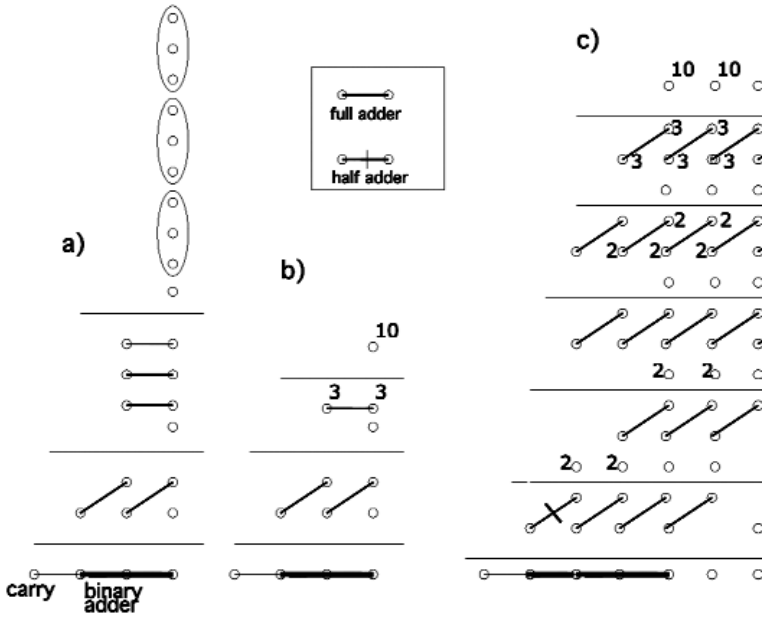


Fig. 1. Examples of dot-notation: adding 10 bit; a) in simple dot-notation; b) in compact dot-notation; c) adding 10, 4bit binary numbers, in Compact dot-notation.

spreadsheet language. We show in fig. 2b the expressions to be used for implementing the first stage of the reduction algorithm.

The arrays seen in the above two schemes are identical. Note that each reduction stage is composed from 4 rows.

Fig. 2c is the spreadsheet representation of the compact dot scheme of fig. 1c. The equation of fig. 2b are repeated for each column and stage of fig. 2c.

II - The row overflow generation and detection in compression schemes

In order to describe the row overflow problem we can consider it in a simple situation, i.e. the addition of a small number if columns, as shown in fig. 3A where we consider the addition of numbers composed from 7 bit and represented with columns of different height: 5 for the three least significant bits in columns 1, 2, 3; 4 bit in column 4 and 3 bits in column 5, 6, 7. The first stage of their addition is shown in fig. 3A (see fig. 2c). The stage output is in row 5: 2223443. The output value $n1_i$ in column i is:

$$n1_i = \lfloor n0_i/3 \rfloor + \lfloor n0_{i-1}/3 \rfloor + n0_{imod3} \qquad (1)$$

$n0_i$ and $n0_{i-1}$ being the input values in column i and $i-1$.

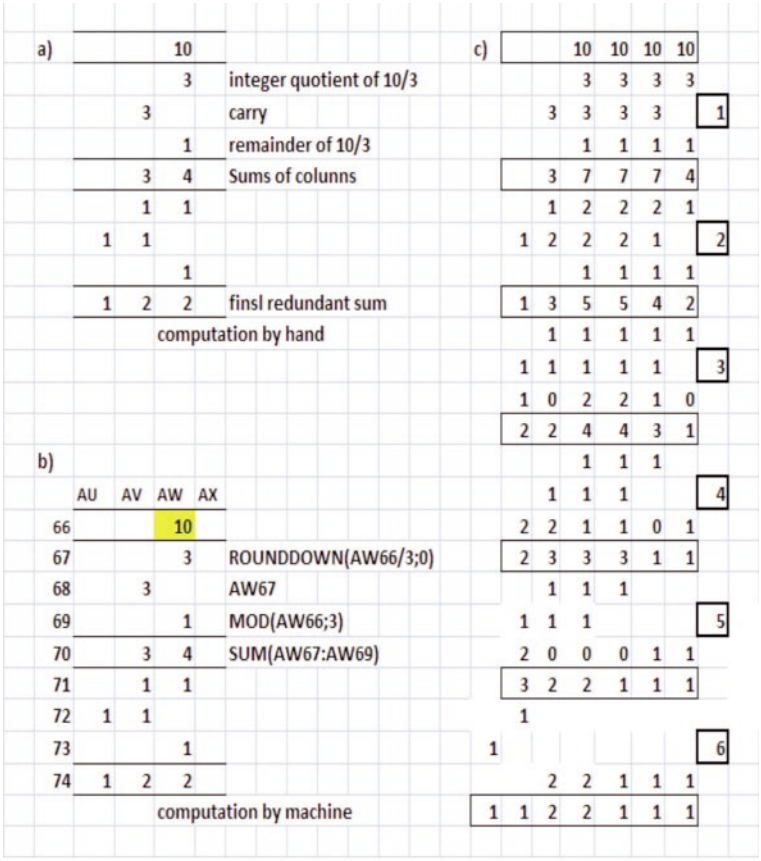


Fig. 2. Cell schemes for same problem of fig. 1b. a) written by hand; b) represented on spreadsheet, operations being done by a program; c) on spreadsheet, same problem of fig. 1c.

The same can be said for fig. 3B.

Let's consider now the case of fig. 3C. Row 1 values change from 6 to 5 (in column 4), to 4. A value, 5 is noticed in column 4, row 5. This value is generated in the transition from a column input of 6 (creating in column 4 a carry valued 2) and a column input of 5, creating in column 4 a quotient 1 and a remainder of 2 for a total output of 5 in column 4 row 5.

Such a value is higher than any other value in the same row 5, an abnormal situation if compared with the two preceding cases. This could require an additional stage in the reduction process.

Thing can be, anyway, corrected. We can feed a half adder (represented in fig. 3C, row 4, with the usual symbol) with the remainder 2 in row 4. The 2 in column 4 is then replaced with a 1 (the output Sum of the half adder), while the 1 in

column 5 (output from the Carry) is increased by 1, see row 6 in same figure. We get in the output row 7 the corrected value of 4 in column 4 and 5.

In fig. 3D we see a case similar to fig. 3C: more precisely it can be derived from fig. 3C by adding 3 to all input values The correction is the same.

The case in fig. 3E differs from the one in fig. 3C for having two adjacent input columns valued 5.

This leads to a correction requiring two half adders.

In fig. 3F we have three adjacent columns with input values 5 and therefore three half adders will be needed. We can obtain successive similar cases.

Note that the cases to be considered are only those of columns adjacent to the maximum in the respective row, since if the columns have smaller heights, an overflow from them will not generate a row overflow in the stage.

Taking care of all the relevant cases, the *Rule* for identifying and correcting the row overflows in each stage of a compression array can be expressed as follows:

Obtain the maximum input value(s) n_{max} (more than one maximum could be found).

If n_{max} >5 and is a multiple of 3, look at the next column input.

If its value is n_{max}*-1* the remainder (valued 2) is fed to a half adder, whose Sum output is written in the same column, in the 4^{rh} row, while the Carry output is placed in next column, in a new 5th row. See fig. 4A.

If more than one adjacent columns are valued n_{max}*-1* a corresponding group of correcting half adders is needed, as in fig. 4B and fig. 4D examples.

Note in fig. 4 that he 5th row contains only the carries of the correcting half adder.

The application of the *Rule* doesn't therefore require to generate the row affected from row-overflow and then to correct it, as done in fig. 3. We instead generate directly the correct row, immune from row overflow, as shown in fig. 4. The *Rule* permits to decide if a column is a source of row overflow and to provide its correct composition: a "regular" one in case of no-row overflow, a column with a half adder if it is needed to avoid row-overflow: see fig. 4 examples.

The *Rule* has been written in the spreadsheet programming language and used in file *A.xlsx*, that can be downloaded from [10].

IV - Designing a binary multiplier immune from row-overflows

We consider a preliminary problem: deciding if the number of rows generated from a stage is not affected by overflow. We need a method that must be as simple as possible. For example, we could compare the number of rows effectively generated in a given scheme with the number of rows from an "ideal" or "standard" stage that, due to its architecture, cannot generate overflow rows. We propose to consider as a standard stage the one that is composed by a very large number of columns, ideally infinite. We can easily simulate such a stage, implementing a single

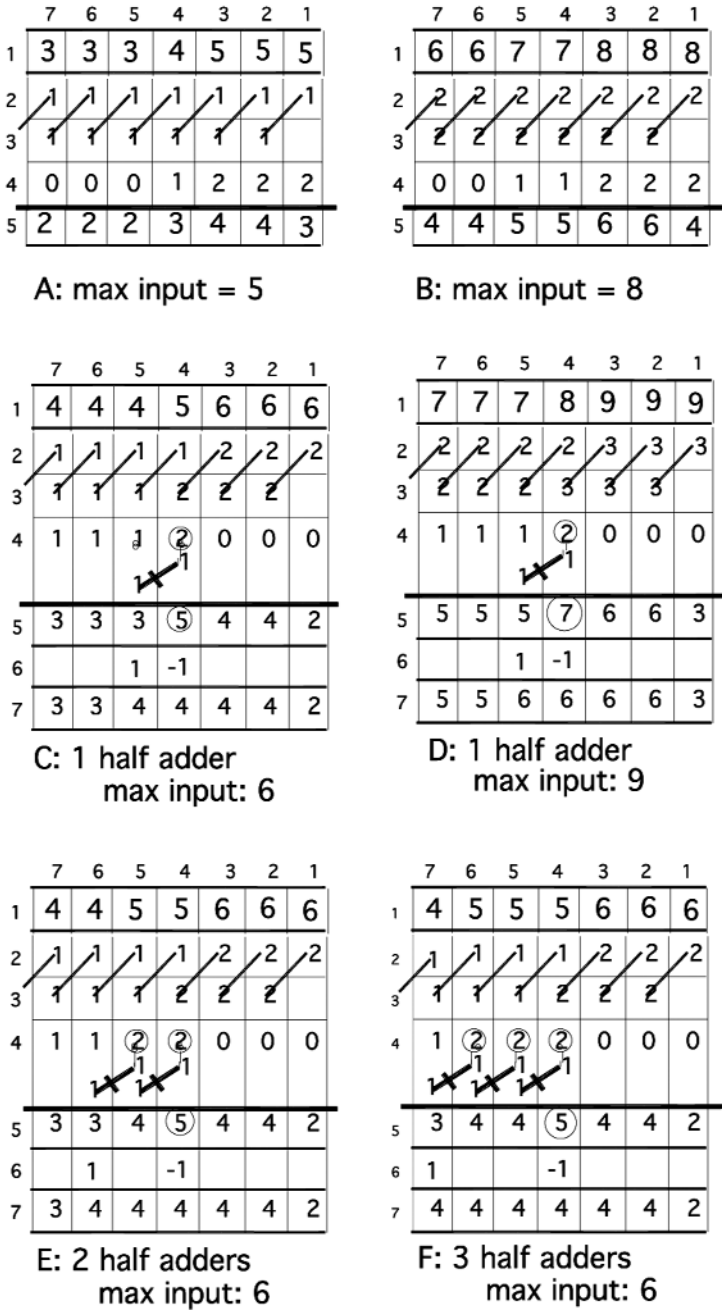


Fig. 3. A and B: two cases with no row overflow. C and D: two cases of overflow requiring a single half adder for correction; E: a case requiring 2 half adders. F: a case requiring 3 half adders.

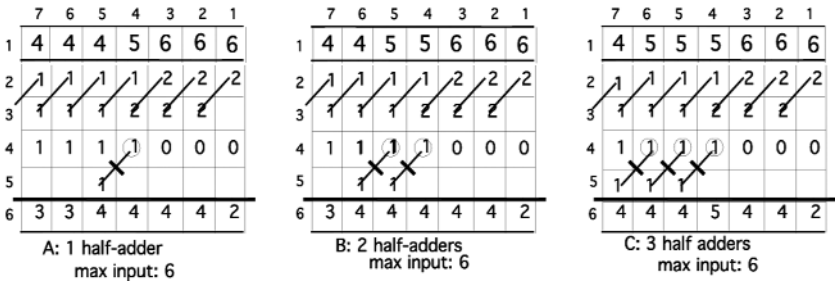


Fig. 4. Compression stages immune from row-overflow for cases A, B, C (see fig. 3).

column and feeding it with carries identical to the carries generated from the column itself. A stage where the columns are identical cannot generate overflow rows due to the over flow generation mechanism requiring a suitable difference among adjacent columns, as seen in the preceding chapter. In other words, the anomaly of row-overflows in such a case is simply impossible.

The input N to a standard scheme will be reduced to an output of $n1$ rows. From N we compute the integer quotient $\lfloor N/3 \rfloor$ and the remainder of N/3, i.e. the value of N_{mod3}. The $n1$ output is:

$$n_1 = 2^{*}\lfloor N/3 \rfloor + N_{mod3} \tag{2}$$

The $n2$ rows output from a second stage, is given by the above relation in which N is replaced with n_1. A sequence of such n_i values is obtained until the value 2 is reached.

In fig. 5 we show the design of a 9×9 bit multiplier reduction stage in two versions: in the first, fig.5a, the design was done ignoring the *Rule* just given for obtaining a row-overflow-free scheme. In fig. 5b instead such *Rule* has been applied. The first scheme is composed from 5 stages, the second from 4 stages.

Note that in fig. 5 we adopt the method for addressing a cell described for fig. 2b. We will develop cases of reduction by executing the required operation "by hand", not by programs written with the spreadsheet language.

Row 1 in scheme 5a is filled by digits 1,2,...8,9,8,…2,1 for representing the partial product array of a 9×9 multiplier. The first reduction stage can be filled column after column starting from the rightmost column, applying the fig. 2a algorithm. Row 5 is the output of the first stage. It is also the input to the second stage, whose output is in row 9. The third stage output is in row 13, followed by the fourth stage whose output is in row 17. Note that its maximum value is 3. This means that the following is the final stage, whose output is composed from values 1 and 2. Note that the max values of the output rows of the various stages are: 7, 5, 4, 3, 2.

Note also that in fig. 5a we use in each of the first 4 stages a single half adder for reducing to 1 the input value 2 in the right part of the array. This reduces the length (and therefore the cost) of the final Carry Propagating Adder, see [1, 8].

The values given by (2) are: 6, 4, 3, 2. i.e. we should have one less stage. Consequently we see that in fig. 5a we must have a row overflow, the cause of the unwanted stage. In order to show the procedure of building an array with no overflow we repeat the exercise in fig. 5b. Looking in first row, the input row, we notice in column I the value 9 followed by 8 in next column H. According to the *Rule*, we know that the remainder in the column with input 8, which is 2 in fig. 5a, must be fed to a half adder, whose Sum output in row 4 is represented in column H by 1, and the Carry output is also represented by a 1 in row 5 in next column G, see figure 5b.

Row 6 contains the sum of rows 2, 3, 4, 5: it is the output of the first stage and the input to the second stage, built with the same algorithm. We notice that the maximum in such a row is 6, in three columns. Starting from the right, we have the first 6 in column J, followed by 5 in next column I. We have therefore an overflow in column I, avoided with a half adder represented by two 1s: the first in row 9, column I, the second (the carry) by a 1 in row 10, column H, i.e. in cell H10.

In row 6 we notice two more adjacent 6, in columns H and G, followed by 4 in column F: this last value tells that there will not be row-overflow.

The third and the fourth stages don't show any overflow, because the max values of their inputs are smaller than 6.

We notice in fig.5a scheme, in column R some "service" cells. In R2, R6, R10, R14 and R18 we find the number of full adders used in the corresponding stages. The counting can easily be performed automatically in spreadsheet.

Same kind of cells is used in fig. 5b.

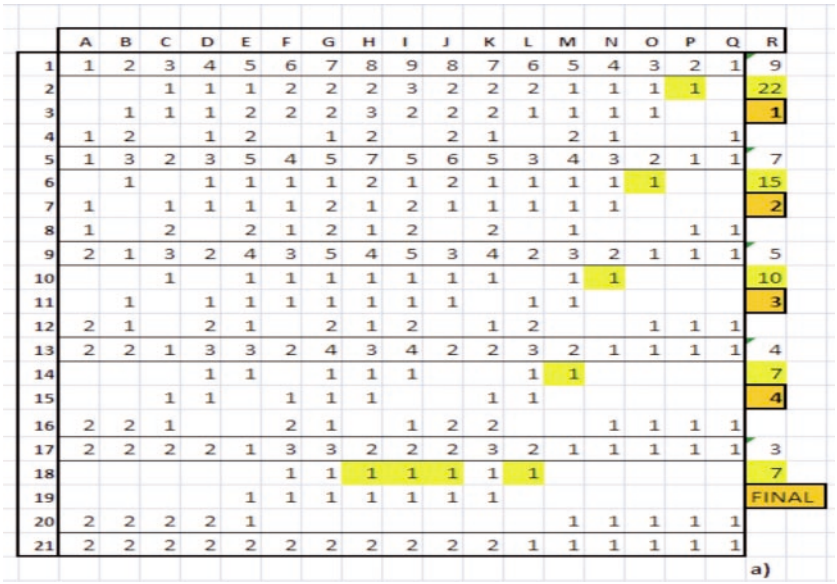
Design tools

Using spreadsheets, we have designed and implemented tools permitting the automatic design of multipliers fo *N* in range 3<N<67. It is not the scope of this writing to illustrate in detail such designs, but rather to use them for a comparison of three different architectures. We invite the interested reader to download such programs from [10].

Two programs generate schemes similar to those of fig. 5, giving also a set of parameters at the basis of the comparison developed in next chapter.

File *A.xlsx* permits the design of multiplier schemes of fig. 5b type, i.e. immune from row overflow.

File *B.xls* generates schemes of fig. 5a type, i.e. affected by row overflow. It will be used here only for comparing the results with the preceding tool, in the said range of *N*. (In reality, this has been the first program developed, that allowed us to realize the existence of the row-overflow problem)



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2	1	9
2			1	1	1	2	2	2	3	2	2	2	1	1	1	1		22
3		1	1	1	2	2	2	3	2	2	2	1	1	1	1			1
4	1	2		1	2		1	2		2	1		2	1			1	
5	1	3	2	3	5	4	5	7	5	6	5	3	4	3	2	1	1	7
6		1		1	1	1	1	2	1	2	1	1	1	1	1			15
7	1		1	1	1	1	2	1	2	1	1	1	1	1				2
8	1		2		2	1	2	1	2		2		1			1	1	
9	2	1	3	2	4	3	5	4	5	3	4	2	3	2	1	1	1	5
10			1		1	1	1	1	1	1	1		1	1				10
11		1		1	1	1	1	1	1	1		1	1					3
12	2	1		2	1		2	1	2		1	2			1	1	1	
13	2	2	1	3	3	2	4	3	4	2	2	3	2	1	1	1	1	4
14				1	1		1	1	1			1	1					7
15			1	1		1	1	1			1	1						4
16	2	2	1			2	1		1	2	2			1	1	1	1	
17	2	2	2	2	1	3	3	2	2	2	3	2	1	1	1	1	1	3
18						1	1	1	1	1	1	1						7
19					1	1	1	1	1	1	1							FINAL
20	2	2	2	2	1								1	1	1	1	1	
21	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	

a)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2	1	9
2			1	1	1	2	2	2	3	2	2	2	1	1	1	1		22
3		1	1	1	2	2	2	3	2	2	2	1	1	1	1			1
4	1	2		1	2		1	1		2	1		2	1			1	
5							1											1
6	1	3	2	3	5	4	6	6	5	6	5	3	4	3	2	1	1	6
7		1		1	1	1	2	2	1	2	1	1	1	1	1			16
8	1		1	1	1	2	2	1	2	1	1	1	1	1				1
9	1		2		2	1			1		2		1			1	1	
10								1										1
11	2	1	3	2	4	4	4	4	4	3	4	2	3	2	1	1	1	4
12			1		1	1	1	1	1	1	1		1	1				10
13		1		1	1	1	1	1	1	1		1	1					1
14	2	1		2	1	1	1	1	1		1	2			1	1	1	
15																		
16	2	2	1	3	3	3	3	3	3	2	2	3	2	1	1	1	1	3
17				1	1	1	1	1	1	1	1	1	1					10
18			1	1	1	1	1	1	1	1	1	1						FINAL
19	2	2	1											1	1	1	1	
20	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	

b)

Fig. 5. Reduction schemes for a 9×9 bit multiplier: a) ignoring the row overflow problem, 5 stages; b) adopting the no-row-overflow generation rule, 4 stages.

File *C.xlsx* implements multipliers according to a rather different architecture, proposed first in [1] based on a prescribed set of "stage heights" (2, 3, 4, 6, 9, 13, 19, 28, 42, 63, 94,…., the *reference sequence*) and offering the smallest number of full adders of all the binary multipliers so far proposed. It is immune from row overflow since its compression algorithm is not aimed at using as many full adders as possible (as in cases *A* and *B*), but only those needed to reach a prescribed column height, namely a *reference* height.

It will be of some interest to compare the performances of the three architectures.

From the examples of fig. 5 we can notice that a characteristic of the algorithms implemented in them is to use in each stage as many full adders as possible in each column: in effect the remainders are all smaller than 3. We can then say that the reduction from *N* (the number of rows of the Partial Product Array) to *3* rows has an optimal solution.

The Final stage, for obtaining the 2-rows output can be done with different architectures and it involves necessarily a number of half adders.

An architecture of this kind (characterized also by the adoption of the reference height sequence), has been proposed by *Bickerstaff, Schulte* and *Swartzlander* in 1995 [8]. They have called such an architecture as *Reduced Area Multipliers*, since it has the smallest area of all schemes so far proposed.

The final stage output is composed from 2 rows. The product is then represented in a 2-redundant digits format. The product in non redundant form requires a Carry Propagating Adder, usually a carry-look-ahead adder for speed reason. In order to reduce its length and therefore its cost, it is important to generate the final 2-redundant result with a number of least significant digits in non redundant form, i.e. with a single bit per column.

Such a result was first shown in [1] as a variation of the Wallace multiplier, by adopting a small number of half adders for reducing to 1 a column composed from 2 bits only. The same result has been shown possible in the Reduced Area Multiplier.

Such a variation has been adopted in our design tools, which gives also the number of non-redundant least significant bits of the generated product, the *final 1s.*

The case of two's-complement factors

It is well known [9] that a multiplier for factors in two's complement format cn be obtained by a transformation of the partial products array as shown in fig. 6a) example. The terms of the bottom row and of the leftmost diagonal are complemented; two 1s are added in the central column; a 1 is placed at the column to the left of the leftmost column (composed from a single term). The complementation of the terms is done in the *partial product generator*; the addition of the two 1s in the central column is a task for the *compression stage*; the single most significant 1 involve the *final Carry Propagating Adder*.

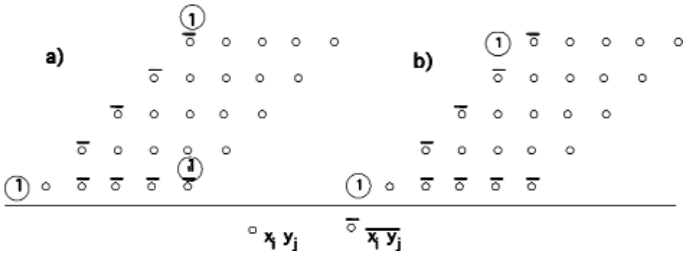


Fig. 6. a) multiplier array for two's-complement factors. The two 1s in central column are needed for complementing the bottom row and the leftmost diagonal; the leftmost 1 derives from such complementation and generates the correct sign for the product when affected by the carry from the preceding column; b) an equivalent more convenient version.

We consider here the second task, suggesting a simple solution, easy to implement in the Partial Products Compression. We first add the two 1s in central column, see fig. 6b). The result is a single 1 in the next (to the left) column. We increase by 1 the corresponding term in the input to the first compression stage. In fig. 5b the term in cell H1 is increased from 8 to 9.

In such an example, the full adders in column H will become 3 (instead of 2); the correcting half adder will not be anymore needed (due to the *Rule*). All the rest of the stages will not be affected. The total effect on the hardware will be: the addition of a full adder and the removal of a correcting half adder.

This solution, valid for all cases, has been tested by simulation.

V - Comparison

Using the above mentioned tools, we have drawn Table I, and Table II, the first containing data on the three multipliers *A, B, C* just commented. For each multiplier Table I shows in each row:

– In column 1: *N,* the factor's length, in bit: $3<N<67$, selected values.

– In columns 2, 3, 4: the number of stages *stg* for *A*, *B*, and *C* multiplier respectively.

– In columns 5, 6, 7*:* the numbers of full adders, *fa*, and half adders, *ha* , in two sub-columns for *B* and *C*, in three sub-columns for *A*, the third one, *hac,* for the number of correcting half adders (included in *ha*).

– In column 8, two sub-columns *A-B* and *A-C* for the differences of (*fa+ha/2*) in the corresponding multipliers. (*fa+ha/2*) is assumed as an acceptable approximation of the area of the Compression stage. We assume *fa+ha/2* as "equivalent full adders".

In Table II we have more detailed data concerning the *A* multiplier, precisely:

– In column *N,* the factors length, in bit: $3<N<67$, selected values. In the associated sub-column: the number of correcting half adder in the 1st stage

– In column 2: the height of the 2nd stage and in the associated sub-column, the number of the correcting half adders, if any.

– In columns 3 to 10: data as in column 2 for the corresponding stages.

– In column *hac:* the total number of correcting half adders for each *N*.

– In columns *stg* and *1s*: the number of stages and the number of *final 1s* respectively.

Some properties of the described multipliers can be read from the given tables.

In few cases, no correction is required for row overflow (*hac* = 0). This happens for *N* = 4, 5, 7, 10, 14, 20, 29, 43, 64. In those cases *N* is 1 more than the value characterizing the *reference sequence* adopted in *C* multipliers and in the *RA* multipliers [8].

From the number of stages *stg* we can derive the delay in the compression stages by multiplying it with the full adder Sum delay. We note in Table I that in a number of cases, precisely in those written in italic, 20 in total, the delays of the three multipliers are equals. In those cases the correction for row overflow is therefore ineffective. We note also that in multipliers *A* and *B* the numbers of full adders and of half adders are identical. However, such multipliers are different due to the corrective half adders present only in *A*.

In most of the cases, 43 over 63, *A* and *B* have different number of stages. The correcting half adders are effective, reducing the total delay of the compression process in multiplier *A*. The number of such correcting half adders in shown in column *hac* in Table II for multiplier *A*. Table I, column 8, sub-column *A-B* shows the differences of the equivalent full adders in the two circuits. In all cases the difference is -1: the corrected circuit *A* uses fewer full adders than *B*. In other words the correcting half adders induce a reduction of full adders, though very small.

Note that in *B* half adders are present in the right side of the compression stages and in the final stage, while in *A* additional *hac* half adders are placed in the central part of the compression stages.

In all cases, the first correcting half adder appears in the column at the left of the central column of the Partial Product arrays. Subsequent half adders (if any) appear in the following stages in adjacent columns at the right of the first one.

Multiplier *C* in smaller in all cases, though by a rather small amount. Multiplier *C* has only a single *final 1* in its product. Multipliers *A* and *B* have a number of *final 1s*, as shown in Table II: such a number is smaller by 1 for multiplier *A*: this is due to the fact that it has 1 less stages in most cases and that in each stage we cannot obtain more than 1 column with a single output bit.

We can also compare the results presented here with similar published results. A specific case is given in [8]. In addition to what has been already said in chapter IV, we note that the *Reduced Area* scheme described in such a paper is similar, but not identical to our scheme. For each stage the reduction is obtained using as many full-adders as possible. A half adder is used in each stage, at the rightmost column

containing a 2, in order to obtain a final 1 in the product. This is what we do in fig. 5a. Moreover, a number of half-adders are placed in a stage when required to reduce the stage height to the values prescribed in the reference sequence. Such a choice implies that the first stage height is, for a number of the cases, higher that the heights of the columns generated from the assigned *N*, except for *N* values equal to one of the reference values.

In our scheme, instead, the reference height sequence is used only for *N* equal to a reference value, i.e. only in a few cases.

In Reduced Area schemes, consequently, row overflow can happen without being noticed. Fortunately, however, they migrate in the following stages where they can be noticed and corrected by half adders. With such a procedure, the Reduced Area scheme can be considered immune from row overflow. An interesting result, obtained in a time when row overflow was ignored.

In our scheme, we, instead, place few half-adders in order to avoid the birth of row overflow. The reference stage height sequence is used only for the cases characterize by *N* equal to values belonging the reference sequence (63,43,28,29, 13,9,6). Note that in Table II the reference stage height sequence is affected by a row overflows requiring a relatively high number of half adders. We have chosen to adopt in each case an height sequence derived by the corresponding *N* in order to implement in each stage a tight control on the row-overflows.

The comparison of the full adders and half adders numbers in *RA* with the corresponding values in our schemes shows that they are identical for the three cases given for *RA* schemes, (N = 8, 12, 16), confirming that the two schemes are very similar. We could transfer to our scheme most of the properties underlined in [8] for the *RA* scheme.

Looking at Table II we note that the cases corresponding to the reference sequence of stages (63, 42, 28, 19, 13, 9, 6,….) correspond to a local maximum of correcting half adders *hac.* Moreover the corresponding *N* is the maximum for a given number of stages. This can justify the adoption in [8] of the reference sequence of stages.

As previously said the cases 64, 43, 29, 20, 14, 10, 7), i.e. those following the reference cases, don't need any row overflow correction, i.e. *hac* = 0. As shown in Table II *hac* increases gradually with *N*, reaching a maximum when the next reference value is reached.

For obtaining a multiplier for two's-complement factors it is suggested in [8] to adopt fig. 6a) scheme. For the addition of the two 1s in the central column of the partial product array it is proposed to use a couple of half adders introduced previously.

Such half adders are first transformed into special half adders generating an output equal to the one of a normal half adder plus 1. This is possible since a half adder is a non-saturated counter.

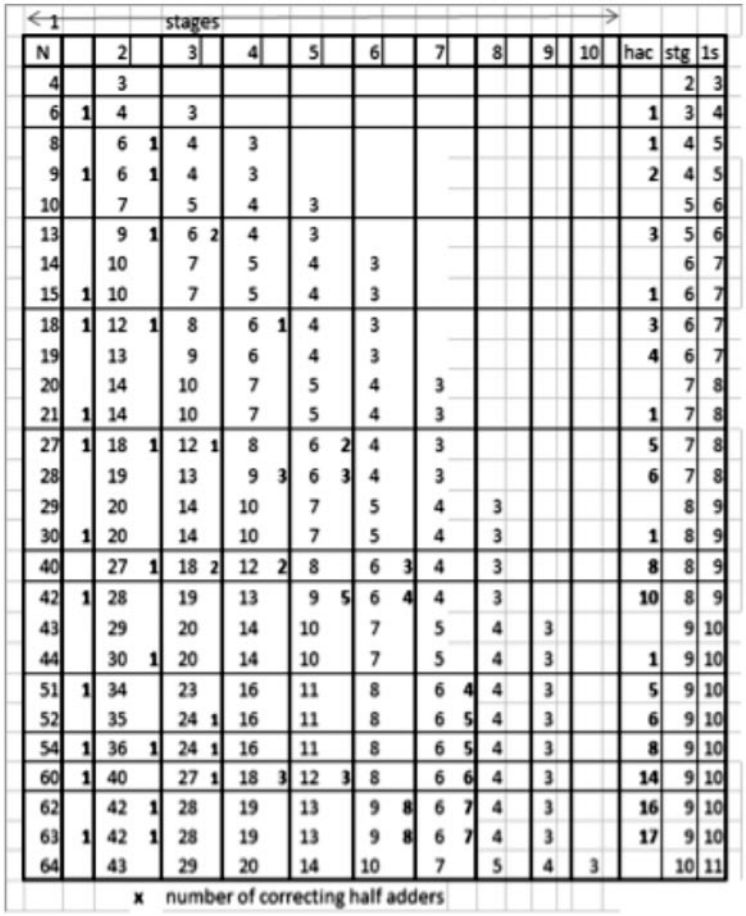
1	2	3	4	5			6		7		8	9
	A	B	C	A			B		C		area diff.	
N	stg	stg	stg	fa	ha	cr	fa	ha	fa	ha	A-C	A-B
4	2	2	2	5	3		5	3	4	2	1.5	
6	3	4	3	18	5	**1**	19	5	16	4	2.5	-1
8	4	5	4	39	7	**1**	40	7	36	6	3.5	-1
9	4	5	4	52	8	**2**	53	8	49	7	3.5	-1
10	5	5	5	68	9		68	9	64	8	4.5	
13	5	6	5	125	12	**3**	126	12	121	11	4.5	-1
14	6	6	6	149	13		149	13	144	12	5.5	
15	6	6	6	174	14	**1**	174	14	169	13	5.5	
18	6	7	6	261	17	**3**	262	17	256	16	5.5	-1
19	6	7	6	294	18	**4**	295	18	289	17	5.5	-1
20	7	7	7	330	19		330	19	324	18	6.5	
21	7	7	7	367	20	**1**	367	20	361	19	6.5	
27	7	8	7	631	26	**5**	632	26	625	25	6.5	-1
28	7	8	7	682	27	**6**	683	27	676	26	6.5	-1
29	8	8	8	736	28		736	28	729	27	7.5	
30	8	8	8	791	29	**1**	791	29	784	28	7.5	
40	8	9	8	1451	39	**8**	1452	39	1444	38	7.5	-1
42	8	9	8	1607	41	**10**	1608	41	1600	40	7.5	-1
43	9	9	9	1689	42		1689	42	1681	41	8.5	
44	9	9	9	1772	43	**1**	1772	43	1764	42	8.5	
51	9	10	9	2409	50	**5**	2410	50	2401	49	8.5	-1
52	9	10	9	2508	51	**6**	2509	51	2500	50	8.5	-1
54	9	10	9	2712	53	**8**	2713	53	2704	52	8.5	-1
60	9	10	9	3372	59	**14**	3373	59	3364	58	8.5	-1
62	9	10	9	3608	61	**16**	3609	61	3600	60	8.5	-1
63	9	10	9	3729	62	**17**	3730	62	3721	61	8.5	-1
64	10	10	10	3853	63		3853	63	3844	62	9.5	

A.xlsx

Table I. Parameters of multipliers A. B and C.

We have tested on our design tool *A.xlsx* and found that the requested half adder exist (for the row overflow correction) in a majority of cases, but not all of them can satisfy the need of the described two's-complement solution. In particular it would not be possible to apply such a method for those cases, that don't need overflow correction (7 cases over 66).

The method for handling two's-complement factors, described in chapter IV, can be applied to all cases, requiring also a smaller increase of the hardware.



N		2		3		4		5		6		7		8		9		10		hac	stg	1s
4		3																			2	3
6	1	4		3																1	3	4
8		6	1	4		3														1	4	5
9	1	6	1	4		3														2	4	5
10		7		5		4		3													5	6
13		9	1	6	2	4		3												3	5	6
14		10		7		5		4		3											6	7
15	1	10		7		5		4		3										1	6	7
18	1	12	1	8		6	1	4		3										3	6	7
19		13		9		6		4		3										4	6	7
20		14		10		7		5		4		3									7	8
21	1	14		10		7		5		4		3								1	7	8
27	1	18	1	12	1	8		6	2	4		3								5	7	8
28		19		13		9	3	6	3	4		3								6	7	8
29		20		14		10		7		5		4		3							8	9
30	1	20		14		10		7		5		4		3						1	8	9
40		27	1	18	2	12	2	8		6	3	4		3						8	8	9
42	1	28		19		13		9	5	6	4	4		3						10	8	9
43		29		20		14		10		7		5		4		3					9	10
44		30	1	20		14		10		7		5		4		3				1	9	10
51	1	34		23		16		11		8		6	4	4		3				5	9	10
52		35		24	1	16		11		8		6	5	4		3				6	9	10
54	1	36	1	24	1	16		11		8		6	5	4		3				8	9	10
60	1	40		27	1	18	3	12	3	8		6	6	4		3				14	9	10
62		42	1	28		19		13		9	8	6	7	4		3				16	9	10
63	1	42	1	28		19		13		9	8	6	7	4		3				17	9	10
64		43		29		20		14		10		7		5		4		3			10	11

Table II. Stages and row overflows correcting half adders in scheme A multiplier.

Conclusion

We have presented an extension of the known dot-notation, for the representation of arithmetic arrays, intended to obtain more compact schemes, proposing multiple-dots and cells, consequently spreadsheets.

An overlooked problem, i.e. the generation of row overflow, causing unwanted delay, can be identified and solved. Finally we have shown how a class of binary multiplier, immune from row overflows, can be implemented, comparing it to known design algorithms.

The design methodology and the related tools presented in this paper is derived from the dot notation methodology. It is helpful in the conception and in the first development phase of arithmetic problems requiring the evaluation of

large arrays. It is necessarily followed by the use of languages like VHDL or Verilog for obtaining a final layout. For this reason we are working in the development of methods and tools for obtaining the transcription of spreadsheet results into the above languages.

Abstract - In the design of multi-operand adders and of multipliers it is useful to represent the array of the addends or of the partial products in a synthetic way for an easier description of the algorithms for obtaining their evaluation through carry free addition. We recall the known dot notation, showing the usefulness of a more compact representation with the adoption of dots associated to a number of logical variables. Adopting a cell instead of a dot, it is easier to execute the calculations for implementing any algorithm. A further step is the adoption of the spreadsheet. In dealing with operation on partial product arrays of binary multipliers the carry free addition reveals an unwanted behavior due to the generation of "overflow" rows that cause additional delay. The scope of this paper is also to clarify the origin of such overflow and to propose a method for their identification and elimination of their effects. As an example we present a program for designing parallel binary multipliers not affected by row overflows.

REFERENCES

[1] Dadda, L., *Some schemes for parallel multipliers*, Alta Frequenza, vol. 19, pp. 349-356, March 1965.
[2] Dadda, L., *A compact dot notation for designing sch binary multi operand adders and binary multipliers*, USI-ALaRI report, 2005.
[3] Nannarelli, A., *A variant of a combinational parallel decimal multiplier*, ISCAS, 2008.
[4] Habibi, A., Wintz, P.A., *Fast Multipliers*, IEEE Transaction on Computers, Febr. 1970.
[5] Dadda, L., *On parallel multipliers*. Alta Frequenza, vol. 45 pp. 574-580, 1976.
[6] Swartzlander E.E., *Merged Arithmetic for Signal Processing*. Proceedings of the 4th IEEE Trans. action on Computer Arithmetic vol. C-29, 946-950, 1980.
[7] Gajski, D.D., *Parallel Compressors*, IEEE Trans is in IEEE Transaction on Computer Arithmetic.
[8] Bickerstaff, A.C., Schulte M.J., Swartzlander, E.E., *Parallel Reduced Area Multipliers*, Journal of VLSI Signal Processing, 9, 181-191, 1995.
[9] Baugh, C.R., Wooley, B.A., *A Two's Complement Parallel Array Multiplication*, IEEE Transaction on Computer, Vol. C-22, pp. 1045-1047, 1973.
[10] Dadda L., *Tools for designing binary parallel multipliers*, http://www.alari.ch/people/dadda/home/BinMult/BinMult.htm