OPTIMUM ARRAY-LIKE STRUCTURES FOR HIGH-SPEED ARITHMETIC*

by

Dharma P. AGRAWAL, Member IEEE
Mini and Microcomputer Laboratory
Swiss Federal Institute of Technology
Ch. de Bellerive 16, CH-1007 Lausanne
Switzerland

SUMMARY

Array-like structures for high-speed multiplication, division, square and square-root operations have been described in this paper. In these designs the division and square-rooting time have been made to approach to that of multiplication operation. These structures are optimum from speed and versatility point of view. Most of the cellular arrays described in the literature are adequately slow. The time delay is particularly significant in the division and square-rooting operations due to the ripple effect of the carries. Though the carry-save technique has been widely utilized for multiplication operation, it has been only recently employed by Cappa et.al. in the design of a non-restoring divider array. This requires sign-bit detection that makes the array non-uniform. Such an array has been named as an array-like structure. The carry-save method has been extended here for restoring division operation. Due to sign-detection and over-flow correction requirements, the restoring method is slightly complex. But the main advantage of such restoring array is in its simple extension for multiplication operation. The array for the two operations, when pipelined, will have more computing power than all other multiplier-divider arrays. Suggestions have also been included for further speed improvement.

The technique applied for division operation is as well applicable for the square-rooting and an array-like structure for square-square-rooting operations has also been given. For performing any one of the four operations, the only manipulation to be done is to combine the two arrays; one for multiplication-division and another for square-square-rooting. Possible methods of combining the two arrays have been indicated and their relative advantages and disadvantages have been mentionec. Finally, a generalized pipeline array-like structure with complete internal details and for 4-bit operation, has been shown. Due consideration has also been given to the possibility of large-scale-integration of different arrays presented in this paper.

*This work is a part of D.Sc.Tecn. thesis and was supported by the Fond National Suisse and the Federal Institute of Technology, Lausanne.

I INTRODUCTION

Since the inception of computers, much effort has been made in search of fast arithmetic techniques. The ripple carry adder, which uses minimum number of gates, forces a long delay in producing the sum as the carry must be propagated through the entire number. The break through came with the introduction of carry-look-ahead adders[1] and the conditional-sum adders[2]. These are the two most economical and yet universally adopted fast adders. While the first one has the problem of fan-in with larger word-length, the second type is asynchronous. The circuits design for most arithmetic operations utilize these adders in one stage or the other.

The design techniques for binary functions have almost approached to the ultimate and much attention has been paid to circuit design of hardware in the form of a regular pattern of special circuits. This idea has become particularly attractive at the present time since this type of design is most suitable for its large-scale-integration. In this respect, Hennie[3] was the first to point out that, with the advancement of microelectronics, it would be worthwhile to consider logical systems composed of a number of identical subnetworks or cells connected in a regular array and defined these as "iterative arrays". Since then, a large amount of papers have appeared dealing with the design of cellular arrays for various arithmetic operations like multiplication, division, square and square-root. These arrays have been designed from regular or semi-regular structure point of view and they utilize arithmetic cells performing two or more functions amongst addition, subtraction and transfer (no arithmetic). Several modifications and improvements have also been suggested. The complexity of these arithmetic cells and their corresponding arrays vary greatly. Due to their iterative or near-iterative nature, these arrays are suitable for the large-scale-integration. But, as fast adders[1,2] have not been included, their operating speed is relatively slow (particularly division and square-rooting). To speed up the multiplication operation, the carry-save[4] addition has been utilized in the design of high speed arrays[5-7]. Very recently, this technique has been employed in an adequately fast divider array[8]. Such a divider array is non-uniform since, in each row of the array, a carry-look-ahead circuit has to be added for computing the sign bit. However, this seems to be the best possible solution from the speed point of view. In this paper, the carry-save technique, the carry-look-ahead circuit for sign-bit computation in each

row and the carry-look-ahead adders for the addition of final two summends have been employed. The speed up techniques utilized here reduces the division and square-root time to approximately equal to that of multiplication operation. But they make the array depart from strict uniformity. That is why, these arrays are hereafter called as array-like structures. Also, to increase the effective efficiency by continous utilization of the digital hardware, application of pipelining technique [9, 10] has been examined in detail.

II MULTIPLIER ARRAY-LIKE STRUCTURES

Based on different algorithms, many cellular arrays have been described in the literature. The application of Booth's algorithm[11] for multiplication leads to an array[12-13] for this function. But this array is unsuitable for its extension as a multipurpose array (see section IV). The usual method of multiplication utilizes "shift" and "add" process. In this method, the multiplication can be started either from the least-significant-bit or from the most-significant-bit. Numerical examples for the two methods are shown in Table I and II and the corresponding arrays are given in Fig. 2 and 3 respectively. These arrays utilize the cell of Fig. 1. The internal structure of the cells shown in Fig. 1a and 1b are the same and only their way of representation is different. Their boolean expressions can be given as :

$$s = a \oplus c_1 \oplus (bp) \qquad (1)$$

and

$$c_2 = ac_1 + (a+c_1)\,bp \qquad (2)$$

where $\oplus$ represents the Exclusive-OR or modulo 2 operation.

TABLE I :Left-shift method of multiplication

multiplicand 1 0 1 1 X multiplier 1 1 0 1

```
  1 0 1 1
0 0 0 0
1 0 1 1
1 0 1 1
```

TABLE II :Right-shift method of multiplication

multiplicand 1 0 1 1 X multiplier 1 1 0 1

```
  1 0 1 1
  1 0 1 1
0 0 0 0
  1 0 1 1
```

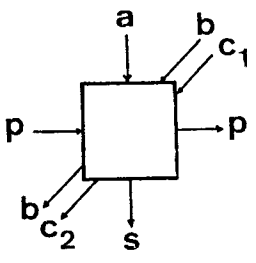

Fig. 1a Arithmetic cell for Fig. 2

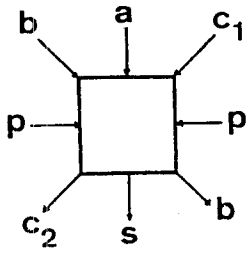

Fig. 1b Arithmetic cell for Fig. 3

Hence, "b" is added only when "p" bit is "1", otherwise only the partial sums and carries are added. This "add" or "transfer" function can be realized with an AND gate and a full-adder. In Fig. 2 and 3, the cells are connected in a carry-save mode and the final two partial-sums are added by fast carry-look-ahead adders[14]. The redundant cells are not included in Fig. 3 and instead, another row of cells are utilized as carry-save adders. This technique reduces the number of cells in the array. The array of Fig. 2 has already been described [5-7, 10]. It is included here for the completeness of the paper.

In multiplying two "n" bit numbers, let the time-delay for the multipliers of Fig. 2 and 3 be denoted by T_L and T_R respectively, then they can be given by expressions :

$$T_L = n\,T + T_{CLA} \qquad (3)$$

and

$$T_R = (n+1)T + T_{CLA} \qquad (4)$$

where T is the delay in each arithmetic cell and T_{CLA} is the delay in the last stage of carry-look-ahead adders.

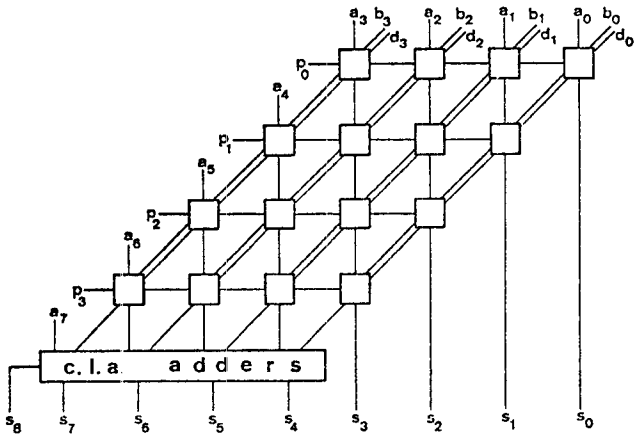

Fig. 2 Left-shift multiplier array-like structure
s = a + bp + d

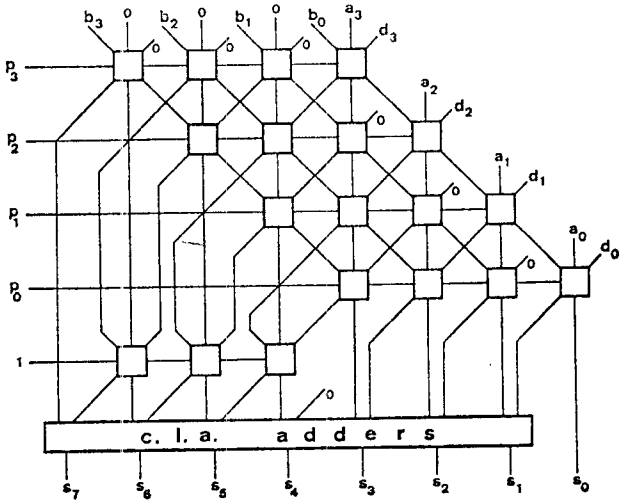

Fig. 3 Right-shift multiplier array-like structure
s = a + bp + d

Now, it can be easily said that the array of Fig. 2 is better, as it is slightly faster and requires lesser number of cells and carry-look-ahead adders for lesser number of bits. But the division operation requires arithmetic to be started from the most-significant-bit[15] (also see section IV), and if it is to be performed by the same array, the configuration of Fig. 3 (with modified cells) is the only choice. Habibi et. al.[7] have shown that the schemes of Wallace[16] and Dadda[17] are nearly 30% faster than the carry-save scheme and suitable integrated circuits[18] are now commercially available. Very recently, Meo[19] has described the analysis and synthesis tools for the design of more complex elementary units of multipliers, each unit of which can process input signals of different weights. But these schemes cannot be extended for other arithmetic operations. Another reason to substantiate the choice of carry-save iterative array is that, when pipelined, this is more efficient[10] than the Wallace multiplier.

III DIVIDER ARRAY-LIKE STRUCTURE

Many arrays for division operation have been proposed and they can be classified in two categories : (i) restoring and (ii) non-restoring. In restoring division, the divisor is subtracted from the dividend (or from the previous remainder); if the remainder is negative, the previous dividend is restored and the quotient bit is taken as zero. Otherwise the quotient bit is one and the process in continued without any change. In non-restoring method, the division process is carried out without restoring the previous divident irrespective of the sign of the result. The organization of two types of dividers are quite similar and only the design of basic cells are slightly different. It was thought by Majithia[20] that the non-restoring dividing arrays are faster than the restoring dividing arrays. But later on Gardiner et.al.[21] showed that the speed of the two types of arrays are almost equal and the restoring technique gives a true remainder. But when the cells are connected in a carry-save mode, the delay in obtaining the sum-bit of each cell is more in the restoring array. The comparison of the restoring array proposed here and the existing non-restoring divider array[8] is given later on.

In a divider array the subtraction can be achieved either directly or by adding 2's complement of the divisor. The functional requirements of different division techniques (partly given in Deegan[22]) are shown in Table III. When an array is to be designed explicitely for division operation, any one of these four possibilities can be considered for its basic cell design. But, when the same array is to be used for multiplication, the restoring and 2's complement addition technique must be preferred. This type of cell requires only two functions of "add"-"transfer", while other combinations require three functions (as multiplication needs "add" - "transfer" functions). An additional advantage of this selection is that the sign-bit detection logic is needed only for 2's complement addition. For all other combinations, either much more complex or two different sign-bit detection logic circuits are needed.

TABLE III :Functional requirements of different division techniques

Technique	Direct subtract	2's complement addition
1 Restoring	Subtract-transfer	Add-transfer
2 Non-restoring	Add-subtract	Add-add(without 2's complement)

One such array-like structure is shown in Fig. 5. This array utilizes the basic cell of Fig. 4. In this cell, the signal "b_i" is added to the summation of "a_i" and "c_i" only when the signal "q_j" is "1". But "a_i", "b_i" and "c_i" are added to compute the expected carry-out "e_{i+1}" and the two carry-look-ahead terms "G_{i+1}" and "P_{i+1}". The boolean expression for the cell can be given as :

$$s_i = (a_i \oplus c_i) \oplus (q_j b_i) \quad (5)$$

$$c_{i+1} = a_i c_i + (a_i + c_i)\, q_j b_i \quad (6)$$

$$e_{i+1} = a_i c_i + (a_i + c_i)\, b_i \quad (7)$$

$$G_{i+1} = (a_i \oplus b_i \oplus c_i)\, e_i \quad (8)$$

and $$P_{i+1} = a_i \oplus b_i \oplus c_i + e_i \quad (9)$$

In Fig. 5, the carry-look-ahead circuits are shown as a block marked with c.l.a. and can be implemented with two

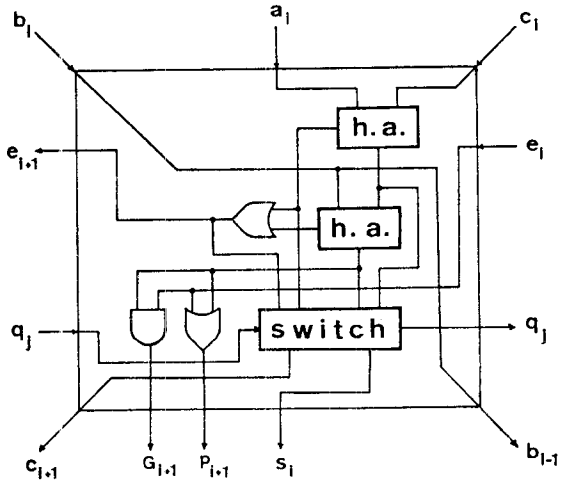

Fig. 4 Divider cell

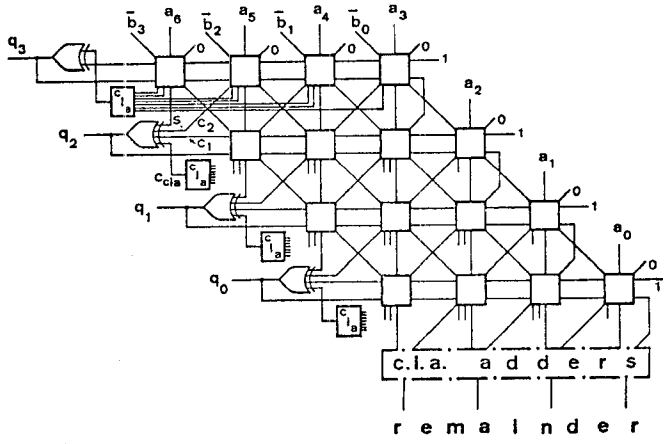

Fig. 5 Divider array-like structure

level NAND gates. But for larger word-length, multilevel carry-look-ahead circuits[14] have to be utilized due to the fan-in limitation.The array of Fig. 5 can divide "a" (having 7 bits word-length) by "b" (4 bits) and the quotient bits are obtained at "q" terminals. The inclusion of 4-bits carry-look-ahead adders after the last step gives the remainder. The quotient bit is taken as "1" when the remainder, after 2's complement addition of the divisor "b", remains positive. Otherwise, the bit-value is taken as "0" and the original value of the remainder is to be restored. In the array of Fig. 5, the restoring operation is slightly different than the usual array[23]. The carry-save technique employed here gives two partial sums in each row of the array and if restoring or "no-arithmetic" is desired, only sums and carries of previous partial remainders (without adding 2's complement of the divisor) are allowed to appear as inputs to the next row. This serves the same purpose. The 2's complement addition is achieved by adding the complement of all individual divisor bits and taking the expected carry-in input "e_i" of the right-most cell as "1" and if the quotient bit is obtained as one, this carry-in "1" is added in the next row. Thus, this "1" has also been taken in to account in the carry-save mode.

Another significant difference in this array is the logic for sign-detection. In the usual arrays[21-22] positive sign is indicated by the presence of the carry-out from the cell of most-significant-bit position. Here, as carry-save gives two remainders, the test for such "carry-overflow" is not sufficient. Also, in the previous row, if the quotient digit is "1" and the carry-overflow has not taken place (rather still present in the previous remainder available in the form of two partial sums), a correction factor has to be considered while evaluating the sign of the remainders under consideration. The truth table for this is given in Table IV. If the correction is required, it is denoted by "1". Similarily "1" indicates the presence of positive remainder. The signals "s", "c_1", "c_2" and "c_{cla}" are the inputs to the 4-input Exclusive-OR gate (also refer to the second row of Fig. 5). The following points are considered while preparing this table :

(a) When no correction is required, more than one variables amongst s, c_1, c_2, and c_{cla} can not be "1" and the presence of any one of them indicates the remainders with positive sign.

(b) When overflow correction is required, at least two signals and at the most three signals amongst s, c_1, c_2, and c_{cla} are in "1" state and when three of them are "1", the remainders are positive.

(c) When c_{cla} is present, the correction will be required in the next step.

All "ϕ" indicates don't cares and minimization gives the logic for positive sign (and the quotient bit q) as:

$$\text{Positive sign, } q = s \oplus c_1 \oplus c_2 \oplus c_{cla} \quad (10)$$

and next step correction requirement $\equiv CR' = c_{cla}$ (11)

It may be noted that the expression (10) is the most compact form and is independent of overflow correction But this correction is equal to the value of c_{cla} in the previous row and the implementation of modulo 2 operation requires more number of gates and a longer delay is introduced due to this. Hence, another appropriate simplification of the Table IV gives

TABLE IV :Sign detection logic for partial remainders

CR,correction required	s	c_1	c_2	c_{cla}	positive sign	CR',next step correction requirement
0	0	0	0	0	0	0
0	0	0	0	1	1	1
0	0	0	1	0	1	0
0	0	0	1	1	ϕ	ϕ
0	0	1	0	0	1	0
0	0	1	0	1	ϕ	ϕ
0	0	1	1	0	ϕ	ϕ
0	0	1	1	1	ϕ	ϕ
0	1	0	0	0	1	0
0	1	0	0	1	ϕ	ϕ
0	1	0	1	0	ϕ	ϕ
0	1	0	1	1	ϕ	ϕ
0	1	1	0	0	ϕ	ϕ
0	1	1	0	1	ϕ	ϕ
0	1	1	1	0	ϕ	ϕ
0	1	1	1	1	ϕ	ϕ
1	0	0	0	0	ϕ	ϕ
1	0	0	0	1	ϕ	ϕ
1	0	0	1	0	ϕ	ϕ
1	0	0	1	1	0	1
1	0	1	0	0	ϕ	ϕ
1	0	1	0	1	0	1
1	0	1	1	0	0	1
1	0	1	1	1	1	0
1	1	0	0	0	ϕ	ϕ
1	1	0	0	1	0	1
1	1	0	1	0	0	1
1	1	0	1	1	1	0
1	1	1	0	0	0	1
1	1	1	0	1	1	0
1	1	1	1	0	1	1
1	1	1	1	1	ϕ	ϕ

ϕ indicates don't care

$$\text{Positive sign} = CR[s\,c_1\,c_2 + s\,c_1\,c_{cla} + s\,c_2\,c_{cla} + c_1\,c_2\,c_{cla}] + \overline{CR}[s + c_1 + c_2 + c_{cla}] \quad (12)$$

and this can be implemented by three-level NAND gates. In fact this delay can be reduced to two levels, if bi-polar circuits are used. For the first row, a further simplification is possible and putting s, c_2 and CR as zero makes the expression (12) as :

$$\text{Sign bit for the first row} = c_1 + c_{cla} \quad (13)$$

It may be noted that it is difficult to show the circuit-implementation of relation (12) in each row of Fig. 5. Hence, for the convenience of drawing (though it is also valid), 4-input Exclusive-OR gates are shown in all diagrams of the following sections.

The array of Fig. 5 can be easily extended for larger word-length. The time delay for such an array will be proportional to the number of quotient bits to be evaluated and for "n" bits, the expression can be given as :

$$T_D = n\,(T_c + T_{ex} + T_{cla}) \quad (14)$$

where T_c is the delay in each cell, T_{ex} is the delay in each Exclusive-OR gate and T_{cla} is the delay in carry-look-ahead circuits. This value of T_D is comparable with the delay in the multiplier array (see expressions 3 and 4) and this array is much faster than other divider schemes [20-26]. Thus, the extra cost of carry-look-ahead circuits for the sign-bit is justified. **Now the comparisor of the non-restoring array of Cappa et. al.[8] and the restoring array of Fig. 5 can be taken up.**

TABLE V :Comparison of component requirements for "n" bits of the quotient and the divisor

Item	Non-restoring array of Cappa et. al.	Restoring array proposed in Fig 5
1. No of arithmetic cells	n^2	n^2
2. No of sign-bit cells	n	-
3. Sign-bit carry-look-ahead circuits for no of bits	n-1	n
4. Gates needed for quotient bits computation		
i. No of Exclusive-OR gates	n	-
ii No of NAND gates	-	9n

TABLE VI :NAND/NOR implementation of different arithmetic cells

Type of gate	Cell of Cappa et.al.	Cell of Fig. 4	Modified cell of Fig. 6
8-input NAND	1	-	1
6-input NAND	-	-	1
5-input NAND	1	-	-
4-input NAND	8	1	10
3-input NAND	4	11	10
2-input NAND	2	5	5
2-input NOR	1	1	1
Inverter	5	6	5
TOTAL	22	24	33

TABLE VII :Comparison of delays in obtaining different signals in each row of the array

Delay in terms of number of gates	Array of Cappa et. al.	Array of Fig. 5	Array using modified cell of Fig. 6
1. For G and P terms	4	4	4
2. For sign-bit carry-look-ahead circuits	2	2	2
3. For sign-bit detection	3	2	2
4. For quotient-bit	9	8	8
5. For carry term/terms	3	10	4
6. For sum term/terms	4	11	4
7. TOTAL delay in each row	9	11	8

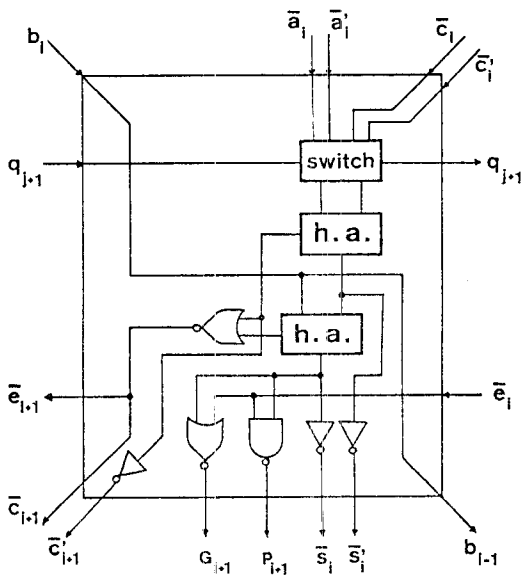

Fig. 6 Modified divider cell

The array proposed here does not require the cell for the sign-bit, but the sign-bit carry-look-ahead circuit is required for one larger number of bit. Table V and Table VI shows the no. of NAND/NOR gates needed for the arithmetic cell of Cappa et. al.[8] and that of Fig. 4. When these two cells are implemented with M.O.S. Technology, the cell of Cappa et. al. requires 22 resistors and 68 transistors, while 24 resistors and 55 M.O.S. transistors are needed for the cell of Fig. 4. Thus, the cell of Fig. 4 needs about 10% lesser than the non-restoring cell of Cappa et. al..

The time delay in obtaining different signals in each row of the two arrays, are given in Table VII. These values are based on the implementation of the two cells with NAND/NOR gates. The array of Fig. 5 is slightly slower as the restoration process has to be kept postponed till the corresponding quotient bit is available. This time delay can be reduced if the restoration function is included in the cell of next row. This necessitates certain alterations and the modified arithmetic cell is shown in Fig. 6. The boolean relations for this cell, in a simplified form, can be given as :

$$\bar{s}'_i = q_{j+1}(a_i \odot c_i) + \bar{q}_{j+1}(a'_i \odot c'_i) \quad (15)$$

$$\bar{s}_i = [q_{j+1}(a_i \odot c_i) + \bar{q}_{j+1}(a'_i \odot c'_i)]\bar{b}_i + [q_{j+1}(a_i \oplus c_i) + \bar{q}_{j+1}(a'_i \oplus c'_i)]b_i \quad (16)$$

$$\bar{c}'_{i+1} = (\bar{a}_i + \bar{c}_i)q_{j+1} + (\bar{a}'_i + \bar{c}'_i)\bar{q}_{j+1} \quad (17)$$

$$\bar{c}_{i+1} = q_{j+1}(\bar{a}_i\bar{b}_i + \bar{b}_i\bar{c}_i + \bar{a}_i\bar{c}_i) + \bar{q}_{j+1}(\bar{a}'_i\bar{b}_i + \bar{b}_i\bar{c}'_i + \bar{a}'_i\bar{c}'_i) \quad (18)$$

and $\bar{e}_{i+1} = \bar{c}_{i+1}$ (19)

where $\odot$ represents the complement of the Exclusive-OR operation ($\odot$ is also sometimes called as the coincidence operation).

As clear from relations (15)-(19), these expressions are more complex than those given by (5)-(9) and hence the modified cell of Fig. 6 requires more components (nearly 65% more) than the cell of Fig. 4. The details are included in Table VI. Also, when implemented with M.O.S. Technology, 33 resistors and 101 transistors are needed for the modified cell. The operating speed of the modified divider cell is shown in Table VII. Thus, at the cost of hardware and increased cellular interconnections, the array becomes marginally faster than non-restoring array of Cappa et. al.[8]. The main advantage of the restoring array is in its simple extension for multiplication operation and is discussed in the next section. It is also worth mentioning that the array of Fig. 5 (and that of Cappa et. al.) can be used to obtain the division of sum of two 6-bit numbers, i.e. q = (a + c)/b can be performed by applying proper values of "c" inputs along with "a" input bits.

IV MULTIPLIER-DIVIDER ARRAY-LIKE STRUCTURE

Most multiplier-divider arrays described in the literature [15, 22, 24, 27] are designed from fully iterative point of view. Division speed-up technique of Cappa et. al. have been discussed in section III and if we try to include multiplication operation in their array, one additional control line has to be included in each arithmetic cell. The restoring-array described in section III, can be easily extended for multiplication operation. This modified form is shown in Fig. 7. The external control line "x" allows the selection of

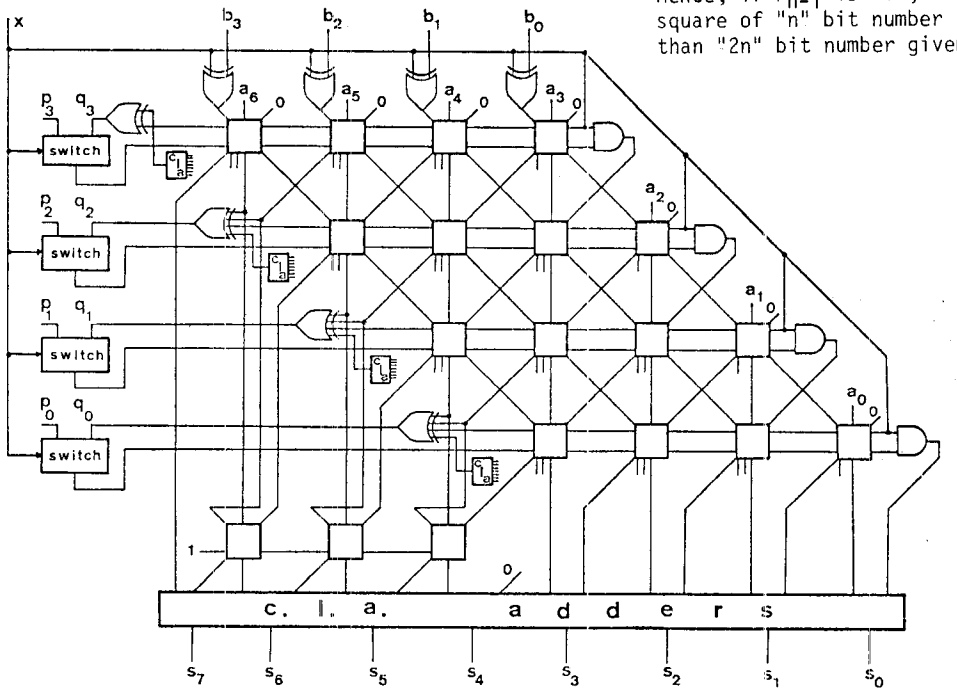

Fig. 7 Multiplier-divider array-like structure
x = 0 for multiplication (s=bp)
x = 1 for division (q=a/b)

multiplication or division (x = 0 for multiplication and x = 1 for division). Proper modifications are done to add 2's complement in division operation. "x" line also controls the multiplexer switches to pass on either "p" inputs or "q" signals. The product "bp" is obtained at "s" terminals and after fourth row of the array, three more cells are needed as carry-save full-adders. It is worth mentioning that the sign-bit carry-look-ahead circuits are not needed for multiplication and actually speaking, the array of Fig. 7 is a combination of Fig. 3 and 5. But due to additional delays in switches and input Exclusive-OR gates, the time delay for division operation will be more than that of the array shown in Fig. 5. The time required for "n" bit by "n" bit multiplication can be given as

$$T_m = (n+1)T_c + T_{ex} + T_{sw} \qquad (20)$$

where T_{sw} is the delay in each multiplexer switch.

Considering the hardware requirement, speed of operation and suitability to large-scale-integration, this multiplier-divider array seems to be optimal design. To include square and square-root functions, this array is further modified and is discussed in section V.

V ARRAY-LIKE STRUCTURE FOR SQUARE-ROOTING AND SQUARING

The process of square-rooting is similar to that of division, except that in division operation, the subtrahend remains the same while in square-rooting, the successive subtrahend changes. This change in restoring method occurs in a particular fashion[9] and can be easily derived. Let the square-root of a "2n" bit number "a" be "r" and given as :

$$\sqrt{a} = r_{n-1}\, r_{n-2} \cdots r_2\, r_1\, r_0 \qquad (21)$$

Hence, if r_{n-1} is "1", "a" must be at least equal to square of "n" bit number (1 0 0 . . 0 0), i.e. greater than "2n" bit number given by (0 1 0 0 . . . 0 0).

Hence, the very first subtrahend is "01" and this is to be subtracted from two most significant bits of "a". If the remainder is positive, then r_{n-1} is "1". Similarily, if r_{n-2} is "1", "a" must be greater than or equal to the square of "n" bit number (r_{n-1} 1 0 0 . . . 0 0). The square of "n" bit number (r_{n-1} 0 0 . . 0) has already been subtracted. Hence an additional quantity to be subtracted, can be easily calculated and this comes out to be "2n" bit number (0 r_{n-1} 0 1 0 0 . . . 0 0). In a similar way, the successive subtrahends for lower significant bits can be obtained and these are shown in Table VIII.

Both restoring [28, 29] and non-restoring [30, 31] arrays have been described in the literature. The interconnection pattern of these arrays are not regular because of the changes in successibe subtrahends. Devries et. al.[32] used an additional signal to control the changes in subtrahend bits and designed a fully-iterative array. Majithia[31] proposed an array for square and square-root and obtained the square by adding the subtrahends of Table VIII. Here an array-like structure for square and square-root will be designed. This array differs from the design of Majithia[3] in two respects. First, to speed up square-rooting, this array utilizes the technique of carry-

TABLE VIII :Subtrahends at different levels in the process of square-rooting

bit to be evaluated	Subtrahend
r_{n-1}	0 1
r_{n-2}	0 r_{n-1} 0 1
r_{n-3}	0 0 r_{n-1} r_{n-2} 0 1
r_{n-4}	0 0 0 r_{n-1} r_{n-2} r_{n-3} 0 1
.	. . . .
.	
.	

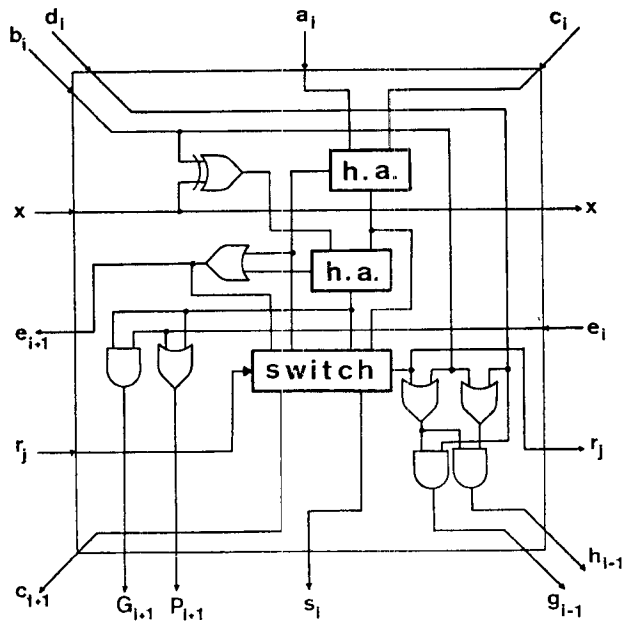

Fig. 8 Cell for square-root and square
x = 0 for square
x = 1 for square-root

save and sign-bit carry-look-ahead circuits described in sections III. The second difference is that cell to cell interconnections are fully iterative [9] and this leads to its possible extension for multiplication and division operations (see section VI).

The basic cell used in the proposed array is shown in Fig. 8. This cell is almost similar to the divider cell of Fig. 4 and two more lines "d" and "h" are added. Also, to achieve 2's complement addition, the Exclusive-OR gate is brought into the cell at its "b" input (rather than putting at the top of the array, as done in multiplier-divider array of Fig. 5). This is done because of the changes in successive subtrahend. The output-input logic equations for the cell of Fig. 8 can be given as :

TABLE IX :Truth table for equations 27 and 28

b_k	d_k	g_{k-1}	h_{k-1}
0	0	0	0
0	1	r_k	r_k
1	0	0	1
1	1	1	1

$$s_i = a_i \oplus c_i \oplus [r_j(b_i \oplus x)] \quad (22)$$

$$c_{i+1} = a_i c_i + (a_i + c_i) r_j (b_i \oplus x) \quad (23)$$

$$e_{i+1} = a_i c_i + (a_i + c_i)(b_i \oplus x) \quad (24)$$

$$G_{i+1} = (a_i \oplus b_i \oplus c_i \oplus x) e_i \quad (25)$$

$$P_{i+1} = a_i \oplus b_i \oplus c_i \oplus x + e_i \quad (26)$$

$$g_{i-1} = b_i d_i + d_i r_j \equiv d_i(b_i + r_j) \quad (27)$$

$$h_{i-1} = b_i + d_i r_j \equiv (b_i + d_i)(b_i + r_j) \quad (28)$$

Here, "b" is the subtrahend and the next level subtrahend is denoted by "g". The variables "d" and "h" are used to change the subtrahend in the desired manner[32]. For ease in understanding, the truth table for these variables is given in Table IX.

A careful study of Table VIII reveals that all the bits below the leading diagonal are zero, while among others, an initial "0" changes to "r_k"; "1" changes to "0" and "r_k" remains unaltered. The maximum successive change is "1" to "0" and "0" to "r_k" and this can be achieved by selecting the value of "b" and "d" bits as "1" and "0" respectively (see Table IX). If no change is needed, then "b" and "d" inputs are made the same. Thus, the changes in successive subtrahends can be achieved by selecting proper values of the variables "b" and "d". The cell of Fig. 8 performs three functions : addition, complement addition and transfer (or restore).

In square operation, the subtrahends of Table VIII are generated and only those subtrahends are added whose corresponding bits are "1". The array for obtaining the square-root of 8-bits or square of 4-bits, is shown in Fig. 9. The basic structure of this array is similar to that of multiplier-divider array of Fig. 7 and hence much explanation is not given here. The number of cells in each row of Fig. 9 increases as the word-length of successive subtrahend in this process changes. The control line "x" is made "1" for square-rooting and "0" for square. Proper values of "b" and "d" inputs are also shown in the figure.

From the hardware requirement point of view, the arithmetic cell of Fig. 8 needs more number of gates than that of Fig. 4. But the array of Fig. 9 is equally fast and the time delay for square and square-rooting operations can be seen equal to the delay in the array of Fig. 5 in performing multiplication and division respectively.

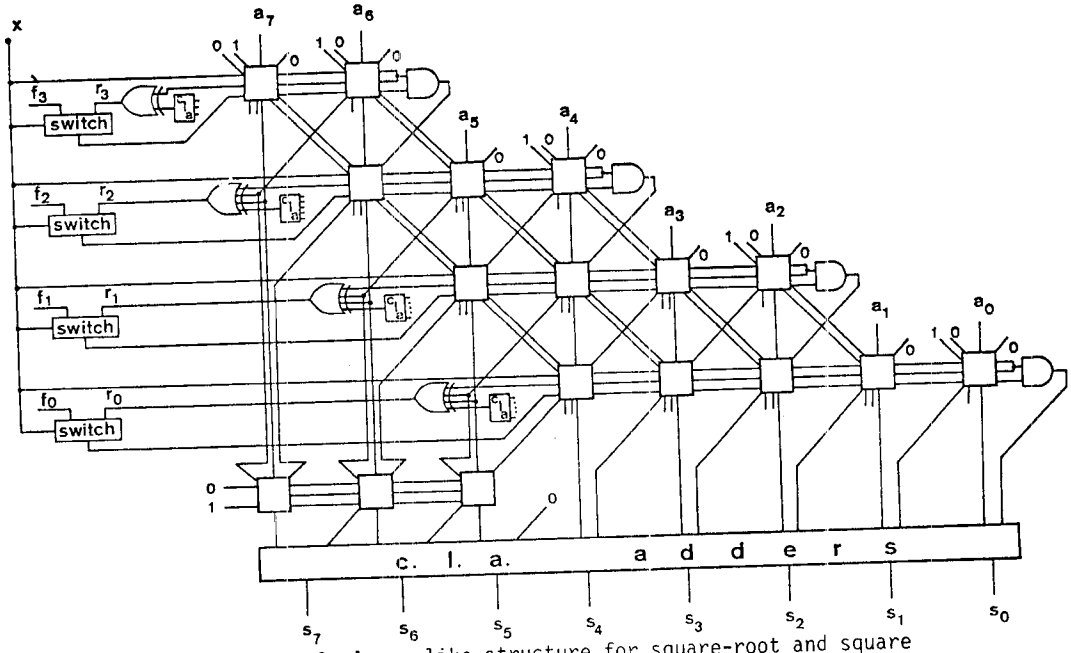

Fig. 9 Array-like structure for square-root and square
x = 0 for square ($s=f^2$)
x = 1 for square-root ($r=\sqrt{a}$)

VI GENERALIZED ARRAY-LIKE STRUCTURE, PIPELINING AND EFFICIENCY

The computing power of a system depends on many factors. To determine this power, two terms are frequently used and they are : bandwidth and latency[33]. Bandwidth is the number of tasks that can be performed in a unit time interval and latency is the length of time required to perform a single task. For a system that operates on only one task at a time, latency is the inverse of bandwidth. Most increase in the bandwidth of arithmetic units has been achieved by reducing the latency. Pipelining is a technique to provide further increase in bandwidth by allowing simultaneous execution of many tasks[34, 35]. In a pipelined unit, a new task is started before the previous task is complete. The key to pipelining a system is the use of temporary storage or latch circuits between two stages of the system. As soon as the output of a stage is set, it can be latched and a new task can be started at the inputs to the stage. The latches keep the value of the old task which is the input to the next row. Hence the bandwidth of a pipelined unit is the inverse of the latency per stage and not the latency of the entire unit. But this is true on the troughput basis, i.e. with every clock-pulse, the operands are available and fed to the system.

Earle[36] designed the carry-save adders with latch circuits. In the field of arithmetic arrays, Deverell[37] and Dean et. al.[38] proposed the inclusion of D-type flip-flops in each arithmetic cell. Their design concept is exactly similar to a pipeline system. Majithia[29] proposed a pipeline array for square-root extraction. Very recently Deverell[10] has given a detailed consideration of the effect of pipelining various multiplier arrays and a combined multiplier-divider array. In fact, most of the arrays described in the literature, can be pipelined and their effective efficiencies can be increased. As the efficiency also depends on the rate of occurance of the operands, a system which can perform a number of operations will be, in general, more efficient. Hence, to obtain an increase in the versatility of the array, a significant step was taken by Agrawal et. al.[39]. Their iterative array for multiplication, division and square-rooting can not perform direct square operation (without using multiplication) as in their array, they have generated 2's complement of the subtrahends shown in Table VIII. Further improvement in their array was suggested by Kamal et. al.[9] by generating the subtrahends of Table VIII and utilizing the concept of pipelining. This array can as well perform square operation (using the subtrahends generated in the process of square-rooting). It may be noted that the direct square operation does not add anything to the complexity of the cell and the array. All these arrays are designed from iterative structure point of view. Here, as carry-save and carry-look-ahead techniques have been utilized, only array-like structure is possible.

A pipelined array for four operations of multiplication, division, square and square-rooting can accept two operands for any one of these functions at every clock-pulse. In order to design such an array, the only problem left is to combine the two arrays shown in Fig. 7 and 9. The square-root - square cell shown in Fig. 8 is more general than the divider cell of Fig. 4 and this has to be utilized in the combined array. The process of square-rooting requires that the subtrahends change in a particular fashion (as given in Table VIII). Hence a direct combiration of the two arrays shown in Fig. 7 and 9 is not possible. One solution is to use the square-root - square array[9, 39] and let the start of multiplication-division operation be postponed till the number of cells in a row at least equals word-length of the operands used for multiplication-division operation. If this type of solution is chosen, the array of Fig. 9 can multiply 3 bits by 3 bits; or can divide 5 bits by 3 bits of quotient. In general, the array for obtaining "n' bits of square-root (or for obtaining square of a "n" bit number) can multiply two (n+2)/2

bit numbers [(n+1)/2 by (n+3)/2 bit numbers, if n is odd]. But if, the word-length is to be kept the same for all the four operations (i.e. multiplication of n X n bits or division by "n" bits with quotient length of "n" bits), additional cells have to be added. These cells have to be placed at the end of the square-root - square array and for n=4, the cell pattern is shown in Fig. 10. The additional cells are shown with back-slash. Another alternative is to add the cells on the left-side of the square-root - square array, so that multiplication and division can be started from the very first row. This design cell pattern is shown in Fig. 11. In these two designs, few cells are redundant for each arithmetic operation. Table X gives the comparison of the two design techniques.

It is clear from Table X that the two solutions require equal number of arithmetic cells and the second design contains lesser number of row with much larger word-length. Hence, its carry-look-ahead circuit will be more complex. Both these designs when pipelined, will have the same latency. The bandwidth of the first will be larger than the second, as the first has more number of steps. But the actual time required to obtain the result is larger in the first solution than in the second. So, whenever there is a recursive relation to be computed by the array, the second design must be preferred. An alternative which combines the two designs, is to shift the multiplier-divider array partly towards left and partly towards downward. The cell pattern for such a design is given in Fig. 12 (for n = 8). The cells with back-slash are needed for multiplication-division and the cells with slash are needed for square-root - square operation. It may be noted that the number of cells required remains unaltered.

The complete design for such an array for the word-length of 4 is shown in Fig. 13. An additional control line "y" is included to decide, whether multiplication-division is required (y = 0); or square-square-rooting is to be performed (y = 1). Values of "b" and "d" inputs are shown at the top of the array and to select them, "y" signal can be again used. Other functions of this array is similar to those described in the previous sections. One important point worth mentioning is that while performing division, the right-most cell is not the cell for the least-significant-bit and normally "1" is forced as a carry-in to l.s.b. position while performing 2's complement addition. It can be easily seen that this addition is automatically achieved[9]. Also, after the irregularity has occured in the structure of the array (such as the 5th row of Fig. 13 and 9th row of the Fig. 12), the look-ahead terms for the missing cells have to be considered for division operation. The value of these terms depends on the quotient bit just before the occurance of the irregularity. If this is zero, all the terms are zero. But when this quotient bit is "1", all look-ahead terms for the missing cells have to be taken as the same as in the row just before the irregularity.

It is also worth mentioning that, for larger word-length, carry-look-ahead circuits become much more complex and due to fan-in consideration, a multi-level carry-look-ahead[14] has to be utilized. This problem can be partly overcome (up to word-length of 16 for a fan-in of 8) by dividing the carry-look-ahead adders in two parts and moving up the right-half portion by one row. This can also be done in the various arrays shown in Fig. 3, 5, 7 and 9. In Fig. 13, the number of latch

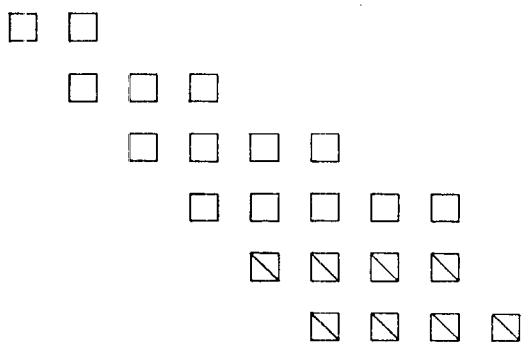

Fig. 10 Arithmetic cell pattern for 4-bit generalized array

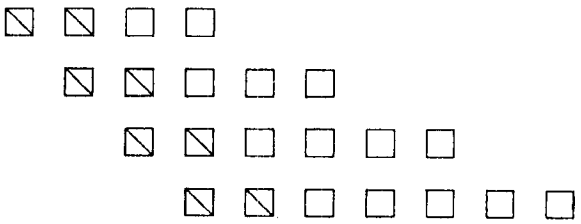

Fig. 11 Another solution for 4-bit generalized array

TABLE X :Comparison of the two designs shown in Figs. 10 and 11 for even values of "n"

Item	First design illustrated in Fig. 12	Second design illustrated in Fig. 13
1. No of arithmetic cells	n(3n-1)/2	n(3n-1)/2
2. No of rows	(2n-2)	n
3. No of cells in the first row	2	n
4. No of cells in the last row	n	(2n-1)
5. Maximum no of cells in a row	(n+1)	(2n-1)
6. No of columns made by arithmetic cells	(3n-3)	(3n-2)
7. Start of square and square-root	*1st row*	*1st row*
8. End of square and square-root	*nth row*	*nth row*
9. Start of multiplication and division	*(n-1)th row*	*1st row*
10. End of multiplication and division	*(2n-2)th row*	*nth row*

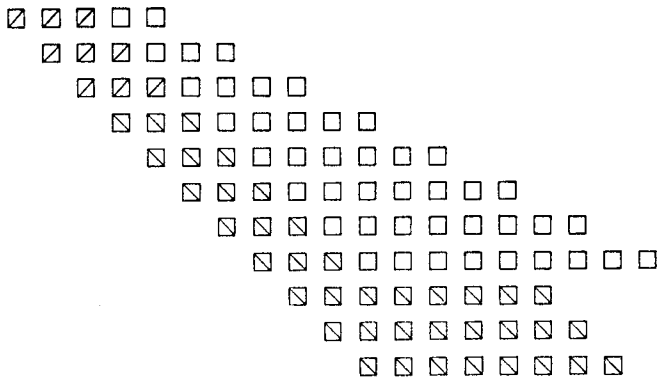

Fig. 12 Midway solution for 8-bits generalized array

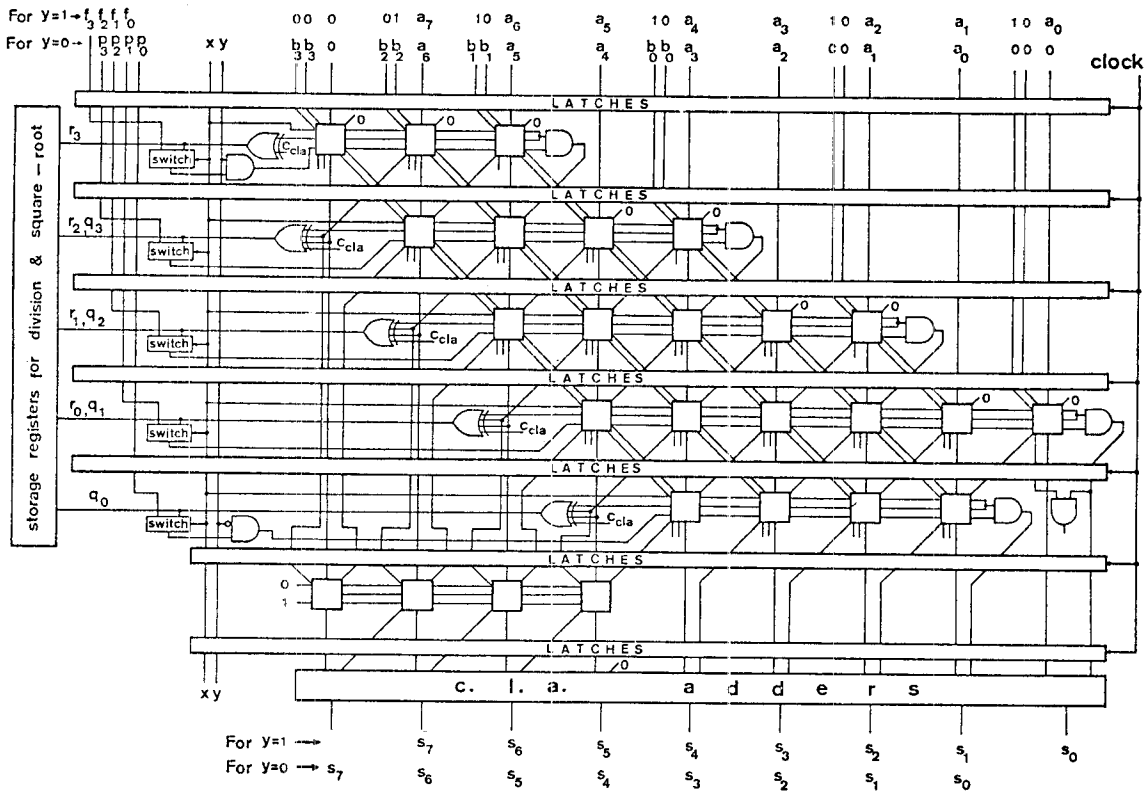

Fig. 13 Generalized pipeline array-like structure for 4-bit word-length

Operation	y	x	result
multiplication	0	0	s=bp
division	0	1	q=a/b
square	1	0	$s=(f)^2$
square-root	1	1	$r=\sqrt{a}$

circuits are different in different rows. At the output terminals, "y" line can be again used for a proper adjustement of the output. An extension for fractional numbers is easily possible. This generalized array is more versatile and efficient than all those arrays presently available in the literature.

VII L.S.I. CONSIDERATION

The design of array-like structure for all the four binary operations (i.e. multiplication, division, square-rooting and square) have been considered in sections II to VI of this paper. All these arrays utilize arithmetic cells in a carry-save mode. The carry-look-ahead circuits for sign-bit and/or carry-look-ahead adders for final addition of two addends, make these arrays non-uniform but these arrays are faster than all existing fully-iterative and near-iterative arrays. The pipelining technique further increases their efficiencies. Approximate number of transistors needed by these arrays with and without pipelining, have been shown in Table XI and XII and with the present trends in integrated circuit technology, their hardware implementation seems to be feasible. The transistor count is based on the application of n-channel M.O.S. gates with a maximum of 9 M.O.S. transistors for 8-input NAND gate. The arithmetic cells are also been considered to have utilized minimum number of transistors (41 for the cell of Fig. 4 and 50 for the cell of Fig. 8).

TABLE XI : Transistor requirements of the multiplier-divider structure for different word-length

No. of bits	No. of transistors needed	No. of transistors needed when pipelined
2	400	600
3	700	1150
4	1350	2200
5	1850	3175
6	2550	4400
7	3375	5825
8	4025	7175

TABLE XII : Transistor requirement of various array-like structures for the word-length of 4 bits

No. of transistors needed	Multiplier-divider structure	Square-root-square structure	Multiplier-divider-square-roor-square structure
Without pipelining	1325	1375	2000
With pipelining	2200	2200	3800

From Table XII it is evident that a pipeline multiplier-divider-square-root-square array requires slightly lesser hardware than the use of two separate arrays (one multiplier-divider and another square-root - square). This is due to the fact that only "2n" cells

are effectively utilized for all the four operations while additional latch circuits are required and the complexity of sign-bit carry-look-ahead circuits increases. Hence, for the computers having distributed arithmetic units or having multi-arithmetic units, the combined array-like structure for all the four operations will not be much advantageous. For such applications it is better to have two separate units. Howsoever, for applications like desk-calculators, a combined unit (without pipeline) will be useful.

VIII CONCLUSION

In this paper, the present state of arithmetic arrays, has been briefly investigated and the best solutions in the form of array-like structures have been given. These structures utilize the techniques of carry-save and sign-bit carry-look-ahead (valid only for division and square-rooting) and carry-look-ahead adders for final addition of two partial sums or remainders. The advantages and disadvantages of restoring and non-restoring techniques of binary division and square-root have been described and the ultimate selection of subtrahend-controlled "add - 2's complement add - transfer" type of cell is justified by its suitability to a generalized pipeline array design. This generalized array can perform any of the four arithmetics amongst mjltiplication, division, square and square-rooting. Further, the array is pipelined with the help of latch circuits. This technique helps in getting the overlapping philosophy implemented and allows more than one operation to occur in the array during the same time-interval. Thus the operands for any of the four arithmetic operations can be fed in to the array at every clock-pulse. On the throughput basis, the effective time delay has been thus reduced to the summation of delays in only one arithmetic cell, one latch circuit and sign-bit carry-look-ahead circuit, Thus, a significant improvement has been achevied with the help of continous utilization and more general design of digital hardware and the application of carry-save and carry-look-ahead techniques. Due to speed-up technique utilized here, the arrays are not fully-iterative. But their structures are array-like and there is every possibility of large-scale-integration of the arrays presented in this paper.

REFERENCES

1. O.L. MacSorley, "High speed arithmetic in binary computers," Proceedings of the I.R.E., vol. 49, pp. 67-91, January 1961.

2. J. Sklansky, "Conditional-sum addition logic," I.R.E. Transactions on Electronic Computers, vol. EC-9, pp. 226-231, June 1960.

3. F.C. Hennie III, "Iterative arrays of logical circuits," The M.I.T. Press and John Wiley Sons Inc, 1961.

4. E.L. Braun, "Digital computer design," New York : Academic Press, 1963.

5. D.P. Burton and D.R. Noaks, "High speed iterative multiplier," Electronics Letters, vol. 4, p. 262, 28th June 1968.

6. K.J. Dean, "Versatile multiplier arrays," Electronics Letters, vol. 4, pp. 333-334, 9th August 1968.

7. A. Habibi and P.A. Wintz, "Fast multipliers," I.E.E.E. Transactions on Computers, vol. C-19, pp. 153-157, February 1970.

8. M. Cappa and V.C. Hamacher, "An augmented iterative array for high-speed binary division," I.E.E.E. Transactions on Computers, vol. C-22, pp. 172-175, February 1973.

9. A.K. Kamal, H. Singh and D.P. Agrawal, "A generalized pipeline array," I.E.E.E. Trarsactions on Computers, vol. C-23, pp. 533-536, May 1974.

10. J. Deverell, "Pipeline iterative arithmetic arrays," I.E.E.E. Transactions on Computers, vol. C-24, pp. 317-322, March 1975.

11. A.D. Booth, "A signed binary multiplication technique," Quart. J. Mech. Appl. Mathl, vol. 4, pp. 236-240, 1951.

12. J.C. Majithia and R. Kitai, "An iterative array for multiplication of signed binary numbers," I.E.E.E. Transactions on Computers, vol. C-20, pp. 214-216, February 1971.

13. S. Bandyopadhyay, S. Basu and A.K. Choudhary, "An iterative array for multiplication of signed binary numbers," I.E.E.E. Transactions on Computers, vol. C-21, pp. 921-922, August 1972.

14. I. Flores, "The logic of computer arithmetic," New Jersey : Prentice Hall, 1963.

15. A. Gex, "Multiplier-divider cellular array," Electronics Letters, vol. 7, pp. 442-444, 29th July 1971.

16. C.S. Wallace, "A suggestion for a fast multiplier," I.E.E.E. Transactions on Electronic Computers, vol. EC-13, pp. 14-17, February 1964.

17. L. Dadda, "Some schemes for parallel multipliers," Alta Frequenza, vol. 34, pp. 349-356, May 1965.

18. Texas instruments incorporated, "Supplement to the TTL data book for design engineers," 1974.

19. A.R. Meo, "Arithmetic networks and their minimization using a new line of elementary units," I.E.E.E. Transactions on Computers, vol. C-24, pp. 258-280, March 1975.

20. J.C. Majithia, "Nonrestoring binary division using a cellular array," Electronics Letters, vol. 6, pp. 303-304, 14th May 1970.

21. A.B. Gardiner and J. Hont, "Comparison of restoring and non-restoring cellular array dividers," Electronics Letters, vol. 7, pp. 172-173, 22nd April 1971.

23. K.J. Dean, "Binary division using a data dependent iterative arrays," Electronics Letters, vol. 4, pp. 283-284, 12th July 1968.

24. D.P. Agrawal and H. Singh, "An iterative array for multiplication and division," Journal of Institution of Electronics and Telecommunication Engineers, vol. 21, pp. 207-209, February 1975.

25. H.H. Guild, "Some cellular logic arrays for non-restoring binary division," The Radio and Electronic Engineer, vol. 39, pp. 345-348, June 1970.

26. R. Stefanelli, "A suggestion for a high-speed parallel binary divider," I.E.E.E. Transactions on Computers, vol. C-21, pp. 42-55, January 1972.

27. G. White, "A versatile cellular array for binary arithmetic," The Radio and Electronic Engineers, vol. 41, pp. 463-464, October 1971.

28. K.J. Dean, "Cellular logical array for extracting square-root," Electronics Letters, vol. 4, pp. 314-315, 26th July 1968.

29. J.C. Majithia, "Pipeline array for square-root extraction," Electronics Letters, vol. 9, pp. 4-5, January 1973.

30. H.H. Guild, "Cellular logical array for non-restoring square-root extraction," Electronics Letters, vol. 6, pp. 66-67, 5th February 1970.

31. J.C. Majithia, "Cellular array for extraction of squares and square-roots of binary numbers," I.E.E.E. Transactions on Computers, vol. C-21, pp. 1023-1024, September 1972.

32. R.C. Devries and M.H. Chao, "Fully iterative array for extracting square-roots," Electronics Letters, vol. 6, pp. 255-256, 16th April 1970.

33. T.G. Hallin and M.J. Flynn, "Pipelining of arithmetic functions," I.E.E.E. Transactions on Computers, vol. C-21, pp. 880-886, August 1972.

34. L.W. Cotten, "Circuit implementation of high-speed pipeline systems," AFIPS Conference Proceedings, vol. 27, pp. 489-504, 1965.

35. L.W. Cotten, "Maximum-rate pipeline systems," Spring Joint Computer conference, AFIPS Conference Proceedings, vol. 34, pp. 581-586, 1969.

36. J.G. Earle, "Latched carry-save adder," IBM Technical disclosure bulletine, vol. 7, pp. 909-910, March 1965.

37. J. Deverell, "Sequential generalized array," Electronics Letters, vol. 8, pp. 9-10, 13th January 1972.

38. K.J. Dean and J. Deverell, "General iterative array,' Electronics Letters, vol. 8, pp. 100-102, 24th February 1972.

39. D.P. Agrawal and H. Singh, "An iterative array for square-root multiplication and division," presented in 8th annual convention of Computer Society of India, New Delhi, 26th February-1st March 1973.