A SIMULATOR FOR ON-LINE ARITHMETIC*

C. S. Raghavendra and M. D. Ercegovac

UCLA Computer Science Department
University of California, Los Angeles

ABSTRACT -- On-line arithmetic is a special class of serial arithmetic where algorithms produce results with the most significant digit first during the serial input of the operands. Speedup of computations can be achieved by overlapping or pipelining successive operations with small delays. This paper describes the design and implementation of a simulator for on-line arithmetic algorithms. The simulator was designed primarily to serve as

1) an experimental tool for synthesis of on-line algorithms;
2) a performance evaluation tool of on-line arithmetic;
3) an on-line calculator in solving some problems involving linear and non-linear recurrences.

The simulator evaluates arithmetic expressions given in a highly functional form. Presently, the set of operations supported include addition, subtraction, multiplication, division, and square root. Several examples are presented in this paper to illustrate the usage of the simulator. The simulator package is implemented in 'C' language on a VAX 11/780 system.

1. INTRODUCTION

There are several approaches for increasing the speed of computations. Speed improvements can be obtained by performing computations concurrently or by pipelining computations. On-line arithmetic is a special class of arithmetic where operations can be overlapped at the digit level to obtain a speed up in the range of 2-16 [ERCE80a] with respect to conventional arithmetic. The algorithms in this class operate systematically in a digit serial manner, beginning with the most significant digit in all operations. In this paper, we describe a simulator for operations and expression evaluation using on-line arithmetic.

On-line algorithms have the special property that, in order to compute the j-th digit of the result it is sufficient to have the operands up to $(j+\delta)$ left most digits [ERCE77, ERCE80b, TRIV77]. The index difference between the result and the input operands, δ, called the on-line delay, is a small integer, typically 1 to 4. These algorithms can be used to speed up computations as they accept operands and compute results in digit serial fashion. In particular, they are suitable for variable precision arithmetic [AVIZ62].

The simulator uses floating point on-line algorithms [WATA80]. The algorithms for addition, subtraction, multiplication, and square root with $\delta = 1$ have been described in the literature [ERCE78, WATA80]. The on-line division algorithm requires $\delta = 2$ to 4 depending on the radix and magnitude of divisor [GORJ80]. The use of redundant number representations of operands is necessary in on-line arithmetic.

* This research is supported in part by the ONR contract N00014-79-C-0866
(Research in Distributed Processing)

The on-line algorithms simulator was designed and implemented with the following objectives:

1) To use the simulator as an experimental tool in synthesis and analysis of on-line arithmetic algorithms;
2) To time it in performance evaluation;
3) To use it as an on-line calculator in solving some special problems involving linear and non-linear recurrences.

The design is modular and the simulator can be conveniently used in an interactive manner. The input expression specification is in a highly functional, prefix form.

Certain design features of the simulator package and the algorithms are presented in the next section. Later, several examples are presented to illustrate the use and operation of the simulator. Finally, methods are discussed to solve recurrence relations using the simulator.

2. DESIGN OF SIMULATOR

The simulator is designed to be highly modular and convenient for interactive usage. The package consists of a main control program and a set of routines for on-line algorithms. The on-line algorithms are written as procedures and they are made as independent as possible from the main program. The program accepts an expression in a functional form with standard symbols or mnemonics for the arithmetic operators. The user can globally specify radix, redundancy ratio, and the precision of the operands, and format of output results. The simulator evaluates a given expression by executing arithmetic operations in parallel using on-line algorithms. For error computations, the given expression is also evaluated using conventional arithmetic. The output represents intermediate results of each arithmetic operation, the time of computation (number of steps), and the errors in comparison with conventional arithmetic results. Several options are provided to the user for formatting the output results as illustrated later in the examples.

The on-line algorithms perform floating point computations. Signed-digit number representation system [AVIZ61] is used for fractional parts and the conventional number system for exponents. Since generally the exponents have fewer digits than mantissa, we assume that complete exponents of operands are available by the time first digit of result is computed. The exponent of the result is computed when the first non-zero digit of mantissa is obtained. Leading zeros, if any, produced in the calculation of the result are suppressed. The floating-point on-line algorithms used in the simulator guarantee that the magnitude of a non-zero mantissa is always greater than r^{-2} [WATA80].

A functional block diagram of the simulator is shown in Figure 1. The input part prompts user to enter the global parameters of the system, namely, radix r of the number system for operands and results, redundancy factor ρ which should be between r/2 for minimal redundancy and r-1 for maximal redundancy, precision m of operands and results, and the print format for output results. The default option is a decimal maximally redundant signed-digit system with 16 digits of precision. The user can also specify a filename when invoking the simulator, in which case the results will be routed to the indicated file.

Next the user will be prompted to enter an arithmetic expression. The expression should be in prefix form with symbols +, -, *, and / or with mnemonics add, sub, mul, div, and sqrt for the arithmetic operators. The expression can be entered in multiple lines and will look like a LISP statement. Presently all operands in the expression are single letter variables, but this limitation can be easily removed. The expression is parsed with some amount of syntax checking and an execution tree, a binary tree, is constructed. The nodes of this tree correspond to on-line arithmetic operators and operands, and the branches indicate data dependencies. A linked list of operand variables is also constructed which is used for requesting values of operands. The user will be prompted to enter the mantissa part first and then the exponent part of each of the operand variables. The number of mantissa digits entered for an operand will be the precision of that operand. If operands are of different length, the default rule is to use the smallest length.

The "productive" phase of the simulator is the computation of result digits using on-line algorithms. The execution tree built for the expression

mul(add(a,b),add(c,d))

is shown in Figure 2. A centralized control mechanism is used in the computations of results where a global clock is maintained and we assume that all the on-line algorithms require unit time to compute one result digit. In each clock time all the nodes are examined, and if the operand digits of a node are available the result digit will be computed and forwarded to destination node. The tree is traversed from left to right and from bottom level to root in each clock step, thus simulating concurrent execution. The expression evaluation is complete when all the significant digits of the root node are computed. The exponent of the result is calculated when the first non-zero result digit is obtained. An alternate approach could be to use asynchronous control mechanism where the arithmetic units exchange tokens when result digits are computed.

The execution part of the simulator is interfaced to a set of on-line arithmetic procedures. These procedures are reentrant and the design is such that it is relatively easy to add modules specifying new on-line algorithms. The interface between the execution part of the simulator and the algorithm procedures is a set of parameters which are used and modified by the algorithms. This way the control structure of the simulator is hidden and modifying an algorithm or implementing a new algorithm is easy. More details of interface and some procedures are presented in Appendix A. The algorithm for addition, multiplication, division, and square-root, currently included in the simulator, are presented in Figures 3, 4, 5, and 6, respectively. In the algorithms presented, i1 and Ex, i2 and Ey, i3 and Ez are index and exponent of operand 1, operand 2, and result, respectively.

In the evaluation of expression, computations are performed as long as significant operand digits are available. The on-line delay used for addition, multiplication, and square-root is 1. For division the delay varies from 2 to 4 depending on the magnitude of the divisor. The numerator is right shifted by two positions internally to make sure that magnitude of divisor is greater than magnitude of dividend. The selection rule used for addition and multiplication is a simple rounding procedure. The selection rules for division and square-root are slightly complicated in that they involve several comparisons [TRIV77, TRIV 78, OKLO78, GORJ80].

Finally, the results are printed according to the user specified format. The results include the total time of computation, the delays encountered by the arithmetic units, result digits and errors in comparison with conventional arithmetic results.

The on-line algorithms simulator is written in 'C' language and runs on a VAX 11/780 system. The size of the program is about 600 lines of 'C' code. Typical execution time is about 0.1 sec for simple arithmetic expressions with 10 nodes are less, including the output of results to a file.

3. EXPERIMENTAL RESULTS

In this section we illustrate typical input and output phases of the simulator. The user will be queried to specify values of radix r, redundancy factor ρ, precision m of of operands, and print option. Then the user will be asked to enter an arithmetic expression. The expression can be entered in as many lines as desired, with two successive carriage returns indicating end of expression. An example of input expression is,

mul(div(a,b),add(c,d))

This can also be entered as,

*(/(a,b),+(c,d))

The simulator has three main output options. In the first option, the printing of result digits follow the corresponding timing diagram as shown in Figure 7. The node labels and exponents are printed as two columns. This option can be used when there is a small number of nodes in the computation tree. If the number of nodes in tree is large, as in solving a recurrence relation, second output option can be used. Here, the delay is displayed in another column and the result digits are not indented. This type of output format is shown in Figures 10 and 11. The third output option is similar to first option where the labels are printed horizontally and result digits vertically.

More detailed results can also be produced. As shown in Figure 8, the intermediate results at each step, including the errors incurred, are displayed. The intermediate results are very useful in the algorithm development phase.

4. RECURRENCE RELATIONS

An on-line arithmetic approach is particularly attractive in solving linear and non-linear recurrence relations. Consider, for example a recurrence relation for square-root of $Y \geqslant 0$,

$$x_{n+1} = \frac{1}{2}[x_n + \frac{Y}{x_n}]$$

x quadratically converges to square-root of Y. Such recurrence relations can be solved using the simulator and the corresponding speedup observed. Since the on-line algorithms produce results in digit serial manner, additional parallelism can be obtained by overlapping computations of different iterations. While solving the above example, we may be computing digits of x_k, x_{k+1}, x_{k+2}, etc. simultaneously. Such observations might be useful in deciding about the design tradeoffs.

In general, it is very difficult to specify recurrence relations as an expression in functional form. Therefore some additional programming is required to build the computation tree. The structure is built by unfolding the recurrence relation to the required number of levels and then closing on itself to form a loop. Using this structure, the simulator performs specified number of iterations by computing digits of as many iterations in parallel as possible. The computation structure used for calculation of cube-root using Newton-Raphson's technique is shown in Figure 9. It was observed that the use of on-line arithmetic does not alter the convergence properties of recurrence relations.

In particular, we experimented with the following classes of problems using the simulator:

1) Square-root, cube-root,..., nineth-root calculation using recurrence relations of Newton-Raphson's technique.
2) Elliptic integral evaluation.
3) Quadratic convergence division of IBM 360/91.
4) Logarithm evaluation.
5) Finding root of a polynomial equation.
   ex: $x^3 - 3x + 1 = 0$
6) Fixed point problem: $f(x) = 0$.
7) Boundary value problems.

The execution traces of some root computations are shown in Figures 10 and 11.

5. CONCLUSIONS

We have described the design and implementation of a highly functional on-line algorithms simulator. It consists of a control part and a library of on-line algorithm procedures. The simulator can evaluate arithmetic expressions with a specified set of global parameters including radix, redundancy ratio, and maximum precision of operands. The time of computation in terms of number of steps and errors in comparison with conventional arithmetic are produced along with the result digits.

The simulator is quite useful for solving recurrence relations, although this requires some additional programming. A useful observation about speed improvement is obtained as digits of different iterations are computed in an overlapped manner.

6. REFERENCES

[AVIZ61] A. Avizienis, "Signed Digit Number Representation for Fast Parallel Arithmetic", IRE Trans. Electron. Computers, Vol. EC-10, 1961, pp 389-400.

[AVIZ62] A. Avizienis, "On a Flexible Implementation of Digital Arithmetic", Proc. IFIP, 1962, pp 664-668.

[ERCE77] M. D. Ercegovac, "A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer", IEEE Trans. on Comput., Vol. C-26, No.7, July 1977, pp 667-680.

[ERCE78] M. D. Ercegovac, "An On-Line Square Rooting Algorithm", Proc. Fourth Symposium on Computer Arithmetic, October 1978, pp 183-189.

[ERCE80a] M. D. Ercegovac, A. L. Grnarov, "On the Performance of On-Line Arithmetic", Proc. IEEE International Conference on Parallel Processing, August 1980.

[ERCE80b] M. D. Ercegovac, "An Universal On-Line Algorithm for Basic Arithmetic Operations", (in preparation), 1980.

[GORJ80] A. Gorji-Sinaki, M. D. Ercegovac, "On-Line Division Algorithms: A Systematic Derivation", (in preparation), 1980.

[OKLO78] V. G. Oklobdzija, "An On-line Higher Radix Square Rooting Algorithm", MS Thesis, UCLA Computer Science Department, 1978.

[TRIV77] K. S. Trivedi, M. D. Ercegovac, "On-Line Algorithms for Division and Multiplication", IEEE Trans. on Comput., Vol. C-26, No. 7, July 1977, pp 681-687.

[TRIV78] K. S. Trivedi, J. G. Rusnak, "Higher Radix On-Line Division", Proc. Fourth IEEE Symposium on Computer Arithmetic, October 1978, pp 164-174.

[WATA80] O. Watanuki, M. D. Ercegovac, "Floating Point On-Line Algorithms", Submitted to the Fifth Symposium on Computer Arithmetic.

APPENDIX A

The execution tree built by the control part of the simulator consists of data structures in the high level language with the following fields:

opcode	-- A small integer specifying the operation.
num	-- A flag to indicate end of operation.
oprd1	-- Pointer to first operand node.
oprd2	-- Pointer to second operand node.
n1	-- Current digit index of first operand.
n2	-- Current digit index of second operand.
n3	-- Current digit index of result.
res	-- Digit vector of result.
expo	-- Exponent value of result.
wj	-- Variable for intermediate result.
w	-- Array for intermediate result.
del	-- On-line delay δ.

The interface between control part and the algorithm consists of a set of parameters. An algorithm is invoked in the following manner:

```
alg-name(ind,x,y,z,exp,pint,preal)
```

where

ind is an array of indices
ind[0] -- Index of result
ind[1] -- Index of operand 1
ind[2] -- Index of operand 2

x, y, z are digit vectors
x -- Digit vector of operand 1
y -- Digit vector of operand 2
z -- Digit vector of result

exp is an array of exponents
exp[0] -- exponent of result
exp[1] -- exponent of operand 1
exp[2] -- exponent of operand 2

pint is an array of integer parameters
pint[0] -- d_j same as z[ind[0]]
pint[1] -- delay δ

preal is an array of real parameters
preal[0] -- w_j
preal[j] -- w[j]

The arithmetic procedures can use all the above parameters to produce result digit. It can modify indices of operands, result exponent, w_j, and delay δ. However, it should not change result index as this requires normalization and will be done in the control part of the simulator. The 'C' language procedures for add and div are presented in Figures 12 and 13 respectively.

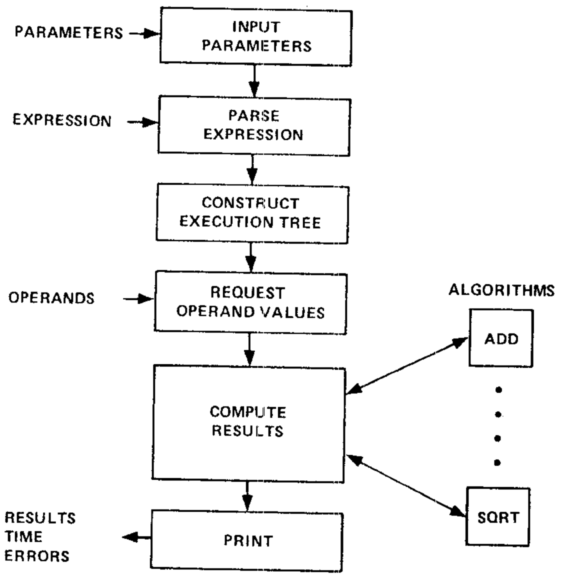

Figure 1. Functional block diagram of the Simulator.

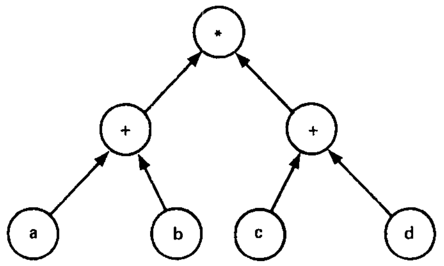

Figure 2. Execution tree of an expression.

Addition/Subtraction

```
Initialization:  i1 = 0   i2 = 0   i3 = 0   $w_0 = 0$

        id = Ex - Ey   sign = 1 for add
                            = -1 for subtraction
        Ez = max(Ex,Ey)

Recursion:  for j = 1,2,3,..... m do
        begin
            if id > 0  i1 = i1 + 1 ; id = id - 1
            if id < 0  i2 = i2 + 1 ; id = id - 1
            if id = 0  i1 = i1 + 1 ; i2 = i2 + 1

            $w_j = r(w_{j-1} - z_{i3}) + r^{-\delta}(x_{i1} + \text{sign}^* y_{i2})$

            $d_j = \text{sign}[w_j] \lfloor |w_j| + 0.5 \rfloor$

            norm($d_j$)
        end

Norm:       if i3 = 0 then
                begin
                    if $d_j \neq 0$ then
                        begin
                            i3 = 1
                            $z_{i3} = d_j$
                            Ez = Ez - (max(i1, i2)-δ-1)
                        end
                end

            else begin
                    i3 = i3 + 1
                    $z_{i3} = d_j$
                end
```

Figure 3. Addition/Subtraction Algorithm.

Multiplication

```
Initialization:  i1 = 0   i2 = 0   i3 = 0

Recursion:  for j = 1,2,3...., m do
        begin
            i1 = i1 + 1 ; i2 = i2 + 1
```

$$X_{j-1} = \sum_{i=1}^{j-1} x_i r^{-i}$$

$$Y_j = \sum_{i=1}^{j} y_i r^{-i}$$

$$w_j = r(w_{j-1} - z_{i3}) + r^{-\delta}(x_{i1} Y_j + y_{i2} X_{j-1})$$

$$d_j = \text{sign}[w_j] \lfloor |w_j| + 0.5 \rfloor$$

```
    norm($d_j$)
end
```

Figure 4. Multiplication algorithm.

Division

Initialization: i1 $= \delta$ i2 $= \delta$ i3 $= 0$

$$P_0 = \sum_{i=1}^{\delta} n_i r^{-i}$$

$$D_0 = \sum_{i=1}^{\delta} d_i r^{-i}$$

$$Q_0 = 0$$

```
Recursion: for j = 1,2,3,...., m do
             begin
               i1 = i1 + 1
               i2 = i2 + 1

               $q_j = \text{select}(rP_{j-1}, D_{j-1})$

               $D_j = D_{j-1} + d_{j+\delta} r^{-j-\delta}$

               $P_j = rP_{j-1} + n_{i1} r^{-\delta} - Q_{j-1} d_{i2} r^{-\delta} - D_j q_j$

               $Q_j = Q_{j-1} + q_j r^{-j}$

               norm($d_j$)
             end
```

Selection: $q_j = \text{i}$ **if**
$rP_{j-1} \epsilon\ [(i-k)D_{j-1} + k(1+k)r^{-\delta+1},\ (i+k)D_{j-1} - k(1+k)r^{-\delta+1}$

$\text{i} \in \{-\rho, \ldots, -1, 0, 1, \ldots, \rho\}$

Figure 5. Division algorithm [GORJ80].

Square-root

Initialization: i1 $= 0$ i3 $= 0$ $Q_0 = 0$

$R_0 = x_1 r^{-2}$ p $= \text{mod}(Ex, 2)$

$$Ez = \left\lceil \frac{Ex}{2} \right\rceil$$

```
Recursion:   for j = 1,2,....,m do
               begin
                 i1 = i1 + 1
                 $d_j = \text{select}(\hat{R}_{j-1}, \hat{B})$
                 $Q_j = Q_{j-1} + d_j r^{-k-p}$
                 $R_j = rR_{j-1} + x_{i1} r^{-2} - 2Q_{j-1} d_j - d_j^2 r^{-k-p}$

                 norm($d_j$)
               end
```

Selection: $\hat{R} = \lfloor r^3 R \rfloor / r^3$

$$S(\hat{R}, \hat{B}) = \begin{cases} k & \text{if } \hat{B}_k < \hat{R} \leqslant \hat{B}_{k+1} \\ 0 & \text{if } \hat{B}_{-1} < \hat{R} \leqslant \hat{B}_1 \\ -k & \text{if } \hat{B}_{-(k-1)} < \hat{R} \leqslant \hat{B}_{-k} \end{cases}$$

$k \in \{-\rho, \ldots, -1, 0, 1, \ldots, \rho\}$

$$\hat{B}_k = A_k \hat{Y}_{j-1} + \lfloor r^3 (D_k r^{-j-1}) \rfloor r^{-3} - x_{i1} r^{-2}$$

$$\hat{B}_{-k} = -A_k \hat{Y}_{j-1} + \lfloor r^3 (D_k r^{-j-1}) \rfloor r^{-3} - x_{i1} r^{-2}$$

$$A_k = \frac{2k-1}{r}$$

$$D_k = \begin{cases} k^2 - k + 1/2 & \text{for } j \leqslant 5 \\ 0 & \textit{otherwise} \end{cases}$$

$$Y_j = \sum_{i=1}^{j-p} z_i r^{-i}$$

Figure 6. Square-root algorithm [OKLO78].

```
mul(add(a,b),add(c,d))


Total Time of Execution = 20

label   expo              mantissa

 d       1     2 4 8 3 2 6 7 3 1 9 4 3 6 7 9 4
 c       1     3 9 2 0 4 1 8 4 6 8 3 9 2 5 6 8
add       2       1 -4 4 0 4 -3 -2 6 -2 -1 -2 -2 3 -1 3 6 2
 b       2     4 9 2 8 5 2 0 5 1 8 4 6 3 8 5 6
 a       3     2 9 0 4 2 8 4 1 9 4 6 8 4 6 3 5
add       3         3 4 0 -3 1 4 -4 2 5 -4 5 3 1 0 2 1
mul       4               2 2 -2 -5 4 2 -1 3 1 0 1 2 3 1 -3
```

Figure 7. Expression evaluation.

```
mul(add(a,b),add(c,d))

mantissa of a = 2 9 0 4 2 8 4 1 9 4 6 8 4 6 3 5
exponent of a = 3

mantissa of b = 4 9 2 8 5 2 0 5 1 8 4 6 3 8 5 6
exponent of b = 2
```

```
mantissa of c = 3 9 2 0 4 1 8 4 6 8 3 9 2 5 6 8
exponent of c = 1

mantissa of d = 2 4 8 3 2 6 7 3 1 9 4 3 6 7 9 4
exponent of d = 1


Total Time of Execution = 20

    operand 1 = 3.920418468392568e+00
    operand 2 = 2.483267319436794e+00

conventional result = 6.403685787829362e+00


  exponent of on-line  result = 2

 j   dj          wj                        error

 1    1    5.000000000000000e-01     -3.596314212170636e+00
 2   -4   -3.700000000000000e+00      4.036857878293642e-01
 3    4    4.000000000000000e+00      3.685787829364129e-03
 4    0    2.999999999999989e-01     3.685787829364129e-03
 5    4    3.599999999999989e+00     -3.142121706358747e-04
 6   -3   -3.300000000000110e+00     -1.421217063590774e-05
 7   -2   -1.500000000001101e+00      5.787829364112262e-06
 8    6    5.699999999988988e+00     -2.121706358382269e-07
 9   -2   -2.300000000110123e+00     -1.217063572145349e-08
10   -1   -1.300000001101235e+00     -2.170635782228203e-09
11   -2   -2.300000011012345e+00     -1.706357277697634e-10
12   -2   -1.800000110123454e+00      2.936428877831077e-11
13    3    2.799998898765464e+00     -6.356026815979021e-13
14   -1   -8.000110123453568e-01     3.643751966819764e-13
15    3    3.499889876546432e+00      6.450395773072159e-14
16    6    6.198898765464318e+00      4.440892098500626e-15
```

Figure 8. Intermediate results of add(c,d).

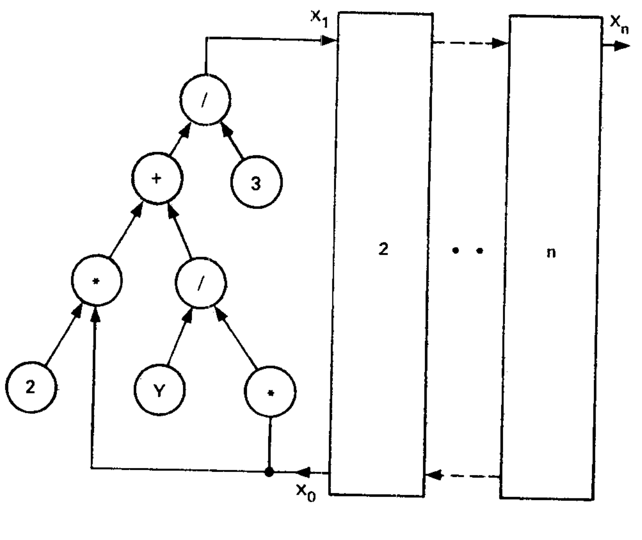

Figure 9. Computation structure for cube-root calculation.

Example 1:

Calculation of cube root using on-line arithmetic.

```
mantissa of y = 4 3 6 2 7 2 8 0 0 0 0 0 0 0 0
exponent of y = 3


 Total Time of Execution = 234

iter   delay   expo                 mantissa

 1      12      3    1  4  6  1 -1  1 -1  3  3  3  3  3  3  3  3  3
 2      22      3    1 -1  7  4  0  0  7  6  5  5  5  5  5  5  5  6
 3      34      2    6  5 -1  5 -1  1  7  2  6  4  4  3  1  6  7  1
 4      44      2    4  3  3  3  4 -1  2  2  2  6  7  5  1  6 -1  4
 5      55      2    3 -1 -1  6  6  7  2  4  3 -2  2  4 -2  6 -1  3
 6      66      2    2 -1  4  8  4  4  6  5  3  0  3  7  1  4  4  0
 7      76      2    1  3  3  7  2  7 -1  7  4  4  4  1  4  4  7  8
 8      87      2    1 -1  7  2  8  3  3  4  4 -1  1 -1  1  1  2 -1
 9      99      1    8  0  2  2  1  5  3 -1  8  2 -2  8  2 -1  0  8
10     109      1    7  6  0  7  8  2  4 -1  3  2  1  2  6  0  1  8
11     120      1    7  6 -2  4  4  4  0 -1  3 -1 -1 -1  8  0  5  8
12     131      1    7  6 -2  4  3  6  7  7 -2  4  2  0  3  8  2  8
13     142      1    7  6 -2  4  3  6  7  7 -2  3  5  1  5  6  5  8
14     153      1    7  6 -2  4  3  6  7  7 -2  3  5  1  5  6  6  1
15     164      1    7  6 -2  4  3  6  7  7 -2  3  5  1  5  6  6  4
```

Figure 10. Calculation of Cube-root.

Example 2:

Calculation of seventh root using on-line arithmetic.

```
mantissa of y = 1 4 6 7 2 8 2 5 9 0 0 0 0 0 0
exponent of y = 3


 Total Time of Execution = 354

iter   delay   expo                 mantissa

 1      19      2    2  1  8  1  8  3  2  2  7  1  4  2  8  5  7  1
 2      35      2    1  8  7  0  1  4  2 -1  6  6  3  7 -1  4  3  6
 3      52      2    1  6  0  3 -1  7  8  8  7  7  3  1  3  3  7  4
 4      70      2    1  3  7  4 -1  8  2  0  1  8  3  9  1  8 -1  3
 5      87      2    1  1  7  7  7 -1  9  1  8  4  4  5  8  4  5  1
 6     105      2    1  0  1 -1  4  5  7  2  3 -1  4  3  4  3  8  7
 7     124      1    8  6  5  2  5  1  0  3  4  8  1  7  8  3  8  2
 8     141      1    7  4  1  6  4  8  7  4 -1  4  4  3  8  6  1  5
 9     157      1    6  3  5  7  1  1  5  1  5  3  3  4  6  4  5  3
10     172      1    5  4  5 -1  2  7  3  2  7  4  3 -1  7 -1  5  9

11     188      1    4  6  7  1  5  9  1  3  7  8 -1  6  8  8  2  5
12     204      1    4  0  0  6  2  3  7  8  1  3  0  2  7  0  8  2
13     221      1    3  4  3  9 -1  8  8 -1  7  6  8  8  0  3  5  9
14     239      1    3 -1  6  0  3  6  1  1  2  3  0  4  0  3  0  9
15     256      1    2  5  6  8  6 -1  4  4  6  8 -1  6  7  3  2  9
16     274      1    2  2  5  8  8 -1  5  2  5  8  2  5  7  7  1  1
17     290      1    2  1 -1  4 -1  2  7  4  8  9  0  7  3  4  4  0
18     306      1    2  0  4  3  4  7  6  4  1  9  2  0  5  8  0  6
19     322      1    1  9 -1  4  4  0  8  3  6 -1  3  2 -1  2  6  3
20     339      1    2  0  7  7  2  7  5  2  7  7  7  7  6  0  2  4
```

Figure 11. Calculation of seventh root.

```
/* On-line Addition Algorithm                    */

     add_on_line(ind,x,y,z,exp,pint,preal)

     int ind[],x[],y[],z[],exp[],pint[];
     double preal[];

     { int i, j, k, dj;
       double wj;

         if(exp[1] > exp[2])
              { ind[1] = ind[1] + 1;
                if(ind[1] == 1)
                        exp[0] = exp[1];

                if(exp[2] + ind[1] > exp[1]) ind[2] = ind[2] + 1;
              }
     else { ind[2] = ind[2] + 1;
            if(ind[2] == 1)
                exp[0] = exp[2];
            if(exp[1] + ind[2] > exp[2]) ind[1] = ind[1] + 1;
          }
      j = ind[1];
      k = ind[2];
      dj = pint[0];
      wj = preal[0];

      wj = r*(wj - dj) + 1.0/r*(x[j] + y[k]);
      preal[0] = wj;

      return(addsel(wj));
     }

   addsel(w)
   double w;
   { int i;
     if(w < 0)  { i = (w - 0.5);
                 if(i >= -ir) return(i);
                      else  return(i+1);
                }
     else     { i = (w + 0.5);
                if(i <= ir) return(i);
                    else  return(i-1);
                }
   }
```

Figure 12. 'C' function for add algorithm.

```
/* On_Line Division Algorithm                     */

  div_on_line(ind,x,y,z,exp,pint,preal)

  int ind[], x[], y[], z[], exp[], pint[];
  double preal[];

   { int i, j, p, dj;
     double qj, rj, wj;

     ind[1] = ind[1] + 1;
     ind[2] = ind[2] + 1;

     j = ind[1];

     if(j < 3) return(0);

     if(j == 3)
       { rj = 0;
         for(i = 1; i < j; ++i)
           rj = rj + powr(r,-i)*y[i];

              if(fabs((rk+1.0)/(rj*(2.0*rk - 1.0))) >= r*r)
                    { exp[0] = exp[1] - exp[2] + 2;
                      pint[1] = 4;
                      preal[0] = powr(r,-4)*x[1];
                      preal[1] = preal[0];
                      return(0);
                    }

         else if(fabs((rk+1.0)/(rj*(2.0*rk - 1.0))) >= r)
                 { exp[0] = exp[1] - exp[2] + 2;
                   pint[1] = 3;
                   preal[0] = powr(r,-3)*x[1];
                   return(0);
                 }


              else { exp[0] = exp[1] - exp[2] + 2;
                   pint[1] = 2;
                   preal[0] = 0;
                   }
       }

     rj = 0;
     for(i = 1; i < j; ++i)
       rj = rj + powr(r,-i)*y[i];

        wj = preal[0];
        dj = divsel(wj,rj,pint[1]);
     rj = rj + powr(r,-j)*y[j];
     qj = 0;
     p = j - ind[0] - pint[1] - 1;

     for(i = 1; i <= ind[0]; ++i)
         qj = qj + powr(r,-p-i)*z[i];


     wj = r*wj + powr(r,-pint[1])*(x[j-2] - y[j]*qj) - rj*dj;

        preal[0] = wj;
        preal[j-pint[1]] = wj;
      return(dj);
   }


divsel(wj,xx,del)

int del;
double wj, xx;

  { int i;
    double yy;

    i = -ir;
    yy = rk*(1.0+rk)*powr(r,-del);

    while(i <= ir)

  { if(r*wj > (fabs(xx)*(i-rk) + yy) && (r*wj <=
                                   (fabs(xx)*(i+rk) - y

         { if(xx >= 0) return(i);
              else  return(-i);
         }
      else i = i + 1;
  }
return(0);

}
```

Figure 13. 'C' function for div algorithm.