FLOATING-POINT ON-LINE ARITHMETIC: ALGORITHMS*

O. Watanuki and M. D. Ercegovac

UCLA Computer Science Department
University of California, Los Angeles

ABSTRACT — For effective application of on-line arithmetic to practical numerical problems, floating-point algorithms for on-line addition/subtraction and multiplication have been implemented by introducing the notion of quasi-normalization. Those proposed are normalized fixed-precision FLPOL (floating-point on-line) algorithms.

1. INTRODUCTION

In designing a high-speed parallel processor it is important to consider features such as system modularity, reconfigurability, LSI realizability, intermodule interconnection bandwidth etc. [AELS77]. For this reason we may consider using on-line arithmetic algorithms [Erce75, Erce77, Erce78, Irwi77, TrEr77, TrRu78], because on-line methods are advantageous from the standpoints of (1) high concurrency, (2) low interconnection bandwidth [Erce78], (3) modularity and (4) VLSI realizability [GrEr80].

The on-line arithmetic algorithms are those algorithms that have the following properties [Erce77]: (1) the computation is performed digit by digit starting with the most significant digit (msd); (2) the j-th digit of the result can be computed as soon as $j+\delta$ digits of the operands are available, where δ is a small integer called on-line delay.

The totally parallel addition proposed by Avizienis [Aviz61, 62] yields an on-line addition algorithm. Irwin [Irwi77] formulated an on-line addition algorithm with on-line delay $\delta=1$. On-line algorithms for multiplication and division were developed by Trivedi and Ercegovac [TrEr77]. The on-line multiplication requires $\delta=1$ and division $\delta=3$ to 5. An on-line square-rooting algorithm with $\delta=1$ was proposed in [Erce78]. The on-line approach can be applied for implementing more complex operations.

Although a variety of on-line algorithms has been proposed, several problems must be solved in order to apply the on-line arithmetic effectively in practical numerical computations. First of all, floating-point algorithms for on-line operations have not been implemented. Furthermore, the properties of redundant mantissa have not been studied and the treatment of normalization problem which results from using a redundant digit set has not been established. Introducing the notion of quasi-normalization, we present floating-point algorithms for on-line addition/subtraction and multiplication in this paper.

2. FLOATING-POINT ON-LINE ARITHMETIC ALGORITHMS

In floating-point on-line (FLPOL) arithmetic algorithms, fixed-point on-line algorithms for addition [Irwi77] and multiplication [TrEr77] are modified and used to compute the result mantissa for arithmetic operations. The exponent processing is similar to the method used in the conventional floating-point algorithms [Coon80, Hwan79, Kuck78, Yohe73]. However, special care must be taken with timing so that operand alignment, treatment of mantissa overflow and post-normalization are performed in an on-line fashion. The proposed normalized fixed-precision FLPOL algorithms can be easily modified into variable precision algorithms. The derivation of the FLPOL algorithm for division is a direct extension.

In order to represent the mantissa of redundant floating-point numbers, signed -digits [Aviz61, 62] are used, while the exponent is represented in the conventional form.

Definition 2.1: The k-digit redundant floating-point number x is represented as

$$x = r^{x_e}\sum_{j=1}^{k} x_j r^{-j}, \qquad (2.1)$$

where r is the radix, x_e is the exponent represented in the conventional form, and the signed-digits of mantissa $x_j \in \{-\rho,\ldots, -1, 0, 1,\ldots, \rho\}$ where $\rho=r/2$ for minimal redundancy and $\rho=r-1$ for maximal redundancy.

In applying the signed digit sets to on-line operations, maximally redundant representation is preferable to minimally redundant representation, because the operands represented using a conventional number system can be directly input to on-line operations. Furthermore, the on-line arithmetic algorithms with maximally redundant digit sets usually require smaller on-line delays, and a faster execution is expected.

However, if a maximally redundant representation is used, it may happen that a redundant mantissa x_f has only one significant digit even though all the digit positions of the mantissa are filled with nonzero digits. We provide the following definitions for the redundant floating-point number system.

Definition 2.2: A non-zero redundant floating-point number $x=x_f r^{x_e}$ with k digits of mantissa represented by a maximally redundant digit set is said to be

(1) normalized if $r^{-1}\leqslant|x_f|<1$;

(2) quasi-normalized if $r^{-2}\leqslant|x_f|<1$;

(3) pseudo-normalized if $r^{-k}\leqslant|x_f|<1$.

* Supported in part by the ONR Contract No. N00014-79-C-0866 (Research in Distributed Processing).

Example: For r=10, ρ=9 and m=4,

(1) normalized $0.11\bar{8}3=0.1023$

(2) quasi-normalized $0.1\bar{9}23=0.0123$

(3) pseudo-normalized $0.1\bar{9}\bar{9}6=0.0004$

As discussed in [WaEr81], the developed FLPOL arithmetic algorithms yield quasi-normalized results, provided that radices higher than two are used. In order to avoid loss of significance due to quasi-normalization, m+1 digits of redundant mantissa must be retained if m significant digits are required. Pseudo-normalized numbers can be only artificially generated, and would not appear in the intermediate results of on-line operations.

Hereafter we shall use the term "normalized" to refer to quasi-normalized redundant floating-point numbers.

Floating-Point On-Line Addition

Given the augend $x=r^{x_e}\sum_{j=1}^{m+1} x_j r^{-j}$ and the addend $y=r^{y_e}\sum_{j=1}^{m+1} y_j r^{-j}$, the result $z=r^{z_e}\sum_{i=1}^{m+1} z_i r^{-i}$ is given by the algorithm below. The mantissa digits are represented in the redundant form, while the conventional representation is used for the exponent. The exponent and the most significant digit (msd) of the mantissa of the result are obtained in the same time step. In the algorithm, i_d represents the difference of the exponents, i is the result digit index, j is the recursion index, and the flag i_n =1 indicates that the algorithm has already output nonzero digits of the result. The vector of signed-digits $s=(s_1, s_2, ..., s_{m+1})$ contains the shifted mantissa digits of the operand with the smaller exponent. Overflow or underflow of the exponent occurs if the final result of e is not in the range of z_e. The treatment of the exceptions follows [Coon80] except for underflow, for which processing follows the algorithm of [Yohe73] in the case of rounding toward zero. In the following algorithms, we use the delimiter || to separate simultaneous assignments.

Algorithm FLPOLA

inputs x_e, y_e **type** integer; x_j, y_j **type** signed-digit; δ, m, r **type** integer constant
/* x_j & y_j become available at step j */
outputs z_e **type** integer; z_i **type** signed-digit
local objects i_d, e **type** integer; d_j **type** signed-digit; s **type** vector of signed-digits; w(j) **type** redundant real
global objects i, j, i_n, x_e, x_j, y_e, y_j, z_e, z_i

begin FLPOLA

/* initialization */

begin INIT
call EXCEPTA;
$i \leftarrow 0$ || $j \leftarrow 1$ || $w(0) \leftarrow 0$ || $d_0 \leftarrow 0$ ||
$i_n \leftarrow 0$ || $i_d \leftarrow x_e - y_e$ || $s \leftarrow \mathbf{0}$
end INIT;

/* negative exponent difference */

if $i_d < 0$ **then**
begin NEGEXP
if $i_d > -(m+1)$ **then**
begin
$e \leftarrow y_e + \delta$;
while $i \leqslant m+1$ **do**
begin
$s_{1-i_d} \leftarrow x_j$ ||
$w(j) \leftarrow r[w(j-1) - d_{j-1}] + r^{-\delta}(s_1 + y_j)$;
$d_j \leftarrow SEL[w(j)]$;
call NORM[e, d_j];
$s \leftarrow SHL[s]$ || $j \leftarrow j+1$
end
end
else
begin
$z_e \leftarrow y_e$ || $z_1 \leftarrow y_1$;
for i=2 **step** 1 **until** m+1 **do**
$z_i \leftarrow y_i$
end
end NEGEXP

/* zero exponent difference */

else if $i_d = 0$ **then**
begin ZEREXP
$e \leftarrow x_e + \delta$;
while $i \leqslant m+1$ **do**
begin
$w(j) \leftarrow r[w(j-1) - d_{j-1}] + r^{-\delta}(x_j + y_j)$;
$d_j \leftarrow SEL[w(j)]$;
call NORM[e, d_j];
$j \leftarrow j+1$
end
end ZEREXP

/* positive exponent difference */

else if $i_d > 0$ **then**
begin POSEXP
if $i_d \leqslant m+1$ **then**
begin
$e \leftarrow x_e + \delta$;
while $i \leqslant m+1$ **do**
begin
$s_{1+i_d} \leftarrow y_j$ ||
$w(j) \leftarrow r[w(j-1) - d_{j-1}] + r^{-\delta}(x_j + s_1)$;
$d_j \leftarrow SEL[w(j)]$;
call NORM[e, d_j];
$s \leftarrow SHL[s]$ || $j \leftarrow j+1$
end
end
else
begin
$z_e \leftarrow x_e$ || $z_1 \leftarrow x_1$;
for i=2 **step** 1 **until** m+1 **do**
$z_i \leftarrow x_i$
end
end POSEXP
end FLPOLA

In the algorithm, SHL is the left shift of one digit position.

The digit selection function SEL[w] is defined as

$$SEL[w] = sign(w)\lfloor |w| + 0.5 \rfloor \qquad (2.2)$$

The function SEL is the same as the one used in the E-method [Erce75, 77].

The on-line delay for fixed-point addition is given as

$$\delta = \left\lceil -\log_r \left[\frac{1}{2} + \frac{(1-r)(1+\Delta)}{4\rho} \right] \right\rceil. \qquad (2.3)$$

In the case of addition, the partial remainder w(j) has a small number of digits and the full precision comparison is efficient. Thus $\Delta = 0$ for on-line addition.

In the algorithm, the procedure EXCEPTA generates the results when at least one of the operands is zero or special operand. EXCEPTA is defined as below. We assume zero or NaN ("Not-a-Number" generated as a result of invalid operations [Coon80]) can be detected by checking the exponent and the msd of the mantissa.

```
procedure EXCEPTA

begin EXCEPTA
  if (|x|<∞ and |y|=∞)
  or (x≠NaN and y=NaN) then
    begin YERR
      z_e←y_e || z_1←y_1;
      for i=2 step 1 until m+1 do
        z_i←y_i;
      error exit
    end YERR

  else if (|x|=∞ and |y|<∞) or
       (x=NaN and y≠NaN) then
    begin XERR
      z_e←x_e || z_1←x_1;
      for i=2 step 1 until m+1 do
        z_i←x_i;
      error exit
    end XERR

  else if (|x|=∞ and |y|=∞) then
    begin INF
      output +∞ or -∞ or NaN,
      depending on the operands
      and the mode;
      error exit
    end INF

  else if (x=NaN and y=NaN) then
    begin NAN
      output NaN depending on
      the operands;
      error exit
    end NAN

  else if (x=0 and 0<|y|<∞) then
    begin XZERO
      z_e←y_e || z_1←y_1;
      for i=2 step 1 until m+1 do
        z_i←y_i;
      exit
    end XZERO

  else if (0<|x|<∞ and y=0) then
    begin YZERO
      z_e←x_e || z_1←x_1;
      for i=2 step 1 until m+1 do
        z_i←x_i;
      exit
    end YZERO

  else if (x=0 and y=0) then
    begin ZERO
      z_e←0 || z_1←0;
      for i=2 step 1 until m+1 do
        z_i←0;
      exit
    end ZERO

  else return

end EXCEPTA
```

The normalization procedure NORM is defined as follows:

```
procedure NORM[e, d_j]

begin NORM

  if i_n=1 then
    begin
      z_{i+1}←d_j || i←i+1
    end;

  else if d_j=0 then
    begin
      if j≤m+δ+1 then e←e-1
      else
        begin
        z_e←0 || z_1←0;
        for i=2 step 1 until m+1 do
          z_i←0;
        exit
        end
    end;

  else
    begin
      i_n←1 || i←1;
      call CHECK[e];
      z_i←d_j
    end;
  return

end NORM
```

CHECK is a procedure to check Overflow or Underflow of the exponent, defined by

```
procedure CHECK[e]

begin CHECK

  if e overflows z_e then
    begin
      output +∞ or -∞ depending on
      the operands and the mode;
      error exit
    end;

  else if e underflows z_e then
    begin
      z_e←0 || z_1←0;
      for i=2 step 1 until m+1 do
        z_i←0;
      error exit
    end;

  else z_e←e;
  return

end CHECK
```

This algorithm is also valid in case the signs of the operands are different. If FLPOL subtraction is needed, the algorithm is trivially realized by changing the sign of the addend y.

Floating-Point On-Line Multiplication

Given the multiplicand $x=r^{x_e}\sum_{j=1}^{m+1} x_j r^{-j}$ and the multiplier $y=r^{y_e}\sum_{j=1}^{m+1} y_j r^{-j}$, the product $z=r^{z_e}\sum_{i=1}^{m+1} z_i r^{-i}$ is produced by the algorithm shown below. Again we use a signed-digit number representation for the mantissa and the conventional representation for the exponent. In the algorithm, i is the result digit index, j is the recursion index, i_n is the normalization indicator.

Algorithm FLPOLM

inputs x_e, y_e **type** integer; x_j, y_j **type** signed-digit; δ, m, r **type** integer constant
/* x_j & y_j become available at step j */
outputs z_e **type** integer; z_i **type** signed-digit
local objects i_d, e **type** integer; d_j **type** signed-digit; s **type** vector of signed-digits; w(j) **type** redundant real
global objects i, j, i_n, x_e, x_j, y_e, y_j, z_e, z_i

begin FLPOLM

/* initialization */

begin INIT
call EXCEPTM;
$i \leftarrow 0$ || $j \leftarrow 1$ || $w(0) \leftarrow 0$ || $d_0 \leftarrow 0$ || $i_n \leftarrow 0$ ||
$X_{-1} \leftarrow 0$ || $Y_0 \leftarrow 0$
end INIT

/* recursion */

begin REC
while $i \leqslant m+1$ **do**
begin
if j=1 **then** $e \leftarrow x_e + y_e + \delta$ **else skip;**
$Y_j \leftarrow CAT[Y_{j-1}, y_j]$ ||
$X_{j-1} \leftarrow CAT[X_{j-2}, x_{j-1}]$;
$w(j) \leftarrow r[w(j-1) - d_{j-1}] + r^{-\delta}(x_j Y_j + y_j X_{j-1})$;
$d_j \leftarrow SEL[w(j)]$;
call NORM[e, d_j];
$j \leftarrow j+1$
end
end REC

end FLPOLM

In the algorithm, δ, SEL and NORM are the same as defined in the floating-point on-line addition algorithm. $X_{j-1}=\sum_{k=1}^{j-1} x_k r^{-k}$, and $Y_j=\sum_{k=1}^{j} y_k r^{-k}$, which are formed by the concatenation CAT.

The procedure EXCEPTM is defined as below. We again assume special operands can be detected by merely checking the exponent and the msd of the mantissa.

procedure EXCEPTM

begin EXCEPTM

if (x=0 **and** |y|=∞)
or (|x|=∞ **and** y=0)
or (x=NaN **and** y=NaN) **then**
begin NAN
output NaN depending on
the operands;
error exit
end NAN

else if (|x|>0 **and** |y|=∞)
or (|x|=∞ **and** |y|>0)
then
begin INF
output +∞ or -∞ depending on
the operands and the mode;
error exit
end INF

else if (x=NaN **and** $y \neq NaN$) **then**
begin XERR
$z_e \leftarrow x_e$ || $z_1 \leftarrow x_1$;
for i=2 **step** 1 **until** m+1 **do**
$z_i \leftarrow x_i$;
error exit
end XERR

else if ($x \neq NaN$ **and** y=NaN) **then**
begin YERR
$z_e \leftarrow y_e$ || $z_1 \leftarrow y_1$;
for i=2 **step** 1 **until** m+1 **do**
$z_i \leftarrow y_i$;
error exit
end YERR

else if (x=0 **or** y=0) **then**
begin ZERO
$z_e \leftarrow 0$ || $z_1 \leftarrow 0$;
for i=2 **step** 1 **until** m+1 **do**
$z_i \leftarrow 0$;
exit
end ZERO

end EXCEPTM

Time Requirement

From the above algorithms, we obtain the following on-line delays for a maximally redundant number system and a higher radix assuming the exponent processing time is equal to the digit computation time. In the following discussions, we use the notation δ_a for the on-line delay of floating-point addition and δ_m for multiplication.

$$0 \leqslant \delta_a \leqslant m+3, \tag{2.4}$$

where $\delta_a=0$ if at least one of the operands is zero, $\delta_a=1$ or 2 for addition of the operands with similar signs, and $\delta_a=2$ to m+3 for subtraction.

Using the above results, the total number of time steps to yield a result of m+1 digits can be given. If at least one of the operands is zero, the total number of steps is m+1. For on-line addition of operands with similar signs, the m+1-digit result is obtained in m+2 to m+3 steps, and for on-line subtraction m+3 to 2m+4 steps. If we assume that alignment shift or normalization shift of one digit position in conventional floating-point arithmetic takes one time step, then conventional floating-point addition and subtraction takes m+1 to 2m+2 steps including one time step for exponent processing. The excessive delay which occurs in subtraction can be improved by using unnormalized arithmetic or significance arithmetic [BiMe77, MeAs58].

Since the absolute value of quasi-normalized operands are in the interval $[r^{-2},1)$, the absolute value of the product should lie in the range $[r^{-4},1)$. That is, the number of leading zeros is at most three. Thus the total delay of FLPOL multiplication is bounded by the following inequality:

$$0 \leqslant \delta_m \leqslant 4, \tag{2.5}$$

where δ_m=0 if at least one of the operands are zero.

Thus the total number of steps to yield m+1-digit product becomes m+2 to m+5. In particular, if at least one of the operands is zero, the result is obtained in m+1 steps.

3. COMPUTATION USING ON-LINE ARITHMETIC

As stated previously, the exponent and the msd of the result mantissa are obtained at the same time in on-line arithmetic, and the subsequent operations can be initiated as soon as both of the msd's of the operands are available. We shall illustrate this process using an example. Consider the expression

$$z = a + (bx + cy), \tag{3.1}$$

where a, b and c are constants represented in the conventional form, and x and y are variables represented in the redundant form. The msd's of bx and cy are produced after a small number of delays. Then addition bx+cy can be initiated as soon as the msd's of both bx and cy become available. Thus the computation is carried out in a digit-serial fashion, starting with the msd of the mantissa.

A numerical example is shown below.

Numerical Example: r=10, ρ=9, m=8,

$$z = a + (bx + cy), \quad \textit{where}$$

a=0.32751613E+00, b=0.14752103E+00,
c=0.71253192E-01,
x=0.60$\bar{4}$115$\bar{2}\bar{1}\bar{5}E$−01, y=0.1$\bar{4}$3301$\bar{2}\bar{5}$3E+01.

time	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
b_e	0
b_j	$\bar{1}$ 4 7 5 2 1 0 3 0
x_e	$\bar{1}$
x_j	6 0 $\bar{4}$ 1 1 5 $\bar{2}$ 1 $\bar{5}$
bx	$\bar{1}$
	$\bar{1}$ $\bar{1}$ $\bar{2}$ $\bar{1}$ 4 $\bar{1}$ 5 $\bar{3}$ 0
c_e	$\bar{1}$
c_j	7 1 2 5 3 1 9 2 0
y_e	1
y_j	1 $\bar{4}$ $\bar{3}$ 3 0 1 $\bar{2}$ $\bar{5}$ 3
cy	$\bar{1}$
	4 5 1 0 4 $\bar{2}$ 1 $\bar{3}$ 1
bx+cy	0
	1 $\bar{5}$ 4 $\bar{1}$ 0 $\bar{2}$ $\bar{2}$ $\bar{5}$ 4
a_e	0
a_j	3 2 7 5 1 6 1 3 0
z_e	0
z_j	4 $\bar{2}$ 1 4 1 4 $\bar{1}$ $\bar{2}$ 4

The computing time of expression (3.1) using on-line arithmetic becomes 14 digit steps using 4 FLPOL arithmetic processors. In conventional normalized floating-point operations, the exponent must be adjusted after the msd of the mantissa is obtained. Therefore, floating-point addition can not be initiated until the whole procedure of the preceding operations is completed. Assuming that the exponent processing time and the digit computation time of conventional arithmetic are equal to the digit computation time of on-line arithmetic, it takes 27 time steps to yield the same result using two conventional floating-point processors. In this example, the computing time using on-line arithmetic is approximately 52% of that of conventional arithmetic. It should be also noted that the computing time of on-line arithmetic is less dependent on the precision.

4. CONCLUSION

Introducing the notion of quasi-normalization, floating-point algorithms for on-line addition and multiplication have been presented. Those developed are normalized, fixed precision algorithms. Because of quasi-normalization, m+1 digits must be computed to retain m significant digits. The exponent is represented using the conventional representation, while the redundant representation is used for the mantissa. One of the advantages of on-line arithmetic is that on-line addition need not wait for the completion of alignment shift, because this can be done in parallel with the recursion for computing the mantissa digits.

Since the exponent and the msd of the mantissa of the result are obtained at the same time in FLPOL operations, the subsequent operations can be initiated as soon as the msd's of the both operands become available. Therefore, a highly overlapped digit-serial computation can be performed using FLPOL arithmetic. Furthermore, its computation time is less dependent on the precision.

5. REFERENCES

[AELS77] Avizienis, A., Ercegovac, M. D., Lang, T., Sylvain, P. and Thomasian, A., "An investigation of fault-tolerant architecture for large-scale numerical computing," High Speed Computer and Algorithm Organization, ed. D. J. Kuck et al., Academic Press, 1977.

[Aviz61] Avizienis, A., "Signed-digit number representation for fast parallel arithmetic," IRE Trans. Electron. Comput., vol. EC-10, pp. 389-400, Sept. 1961.

[Aviz62] Avizienis, A., "On a flexible implementation of digital computer arithmetic," Proceedings of IFIP Congress 62, pp. 664-678, North Holland Publishing Co., 1963.

[BiMe77] Bivins, R. L. and Metropolis, N. C., "Significance arithmetic: application to a partial differential equation," IEEE Trans. Comput., vol.C-26, no. 7, July 1977.

[Coon80] Coonen, J. T., "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," Computer, vol. 13, no. 1, pp. 68-79, Jan. 1980.

[Erce75] Ercegovac, M. D., "A general method for evaluation of functions and computations in a digital computer," Ph.D. dissertation, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Tech. Rept. 750, July 1975.

[Erce77] Ercegovac, M. D., "A general hardware-oriented method for evaluation of functions and computations in a digital computer," IEEE Trans. Comput., vol. C-26, no. 7, pp. 667-680, July 1977.

[Erce78] Ercegovac, M. D., "An on-line square rooting algorithm," Proceedings of the 4th Symposium on Computer Arithmetic, pp. 183-189, Oct. 1978.

[ErGr80] Ercegovac, M. D. and Grnarov, A. L., "On the performance of on-line arithmetic," Proceedings of the 1980 International Conference on Parallel Processing, pp. 55-61, August 1980.

[GrEr80] Grnarov, A. and Ercegovac, M. D., "VLSI-Oriented Iterative Networks for Array Computations," Proceedings of IEEE International Conference on Circuits and Computers, 1980.

[Hwan79] Hwang, K., "Computer arithmetic," Chapter 10, J. Wiley & Sons, N.Y., 1979.

[Irwi77] Irwin, M. J., "An arithmetic unit for on-line computation," Ph.D. dissertation, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, Tech. Rept. 873, May 1977.

[Kuck78] Kuck, D. J., "The structure of computers and computations," volume 1, Wiley, N.Y., 1978.

[MeAs58] Metropolis, N. and Ashenhurst, R. L., "Significant digit arithmetic," IRE Trans. Electron. Comput., vol. EC-7, pp. 265-267, December 1958.

[TrEr77] Trivedi, K. S. and Ercegovac, M. D., "On-line algorithms for division and multiplication," IEEE Trans. Comput., vol. C-26, no. 7, pp. 681-687, July 1977.

[TrRu78] Trivedi, K. S. and Rusnak, J. G., "Higher radix on-line division." Proceedings of the 4th Symposium on Computer Arithmetic, pp. 164-174, Oct. 1978.

[WaEr81] Watanuki, O. and Ercegovac, M. D., "Floating-point on-line arithmetic: error analysis," Proceedings of 5th Symposium on Computer Arithmetic, 1981.

[Yohe73] Yohe, J. M., "Roundings in floating-point arithmetic," IEEE Trans. Comput., vol. C-22, no. 6, pp. 577-586, June 1973.