Journal of VLSI Signal Processing 33, 31–35, 2003

On-the-Fly Range Reduction*

VINCENT LEFÈVRE
INRIA, Projet SPACES, LORIA, Campus Scientifique, B.P. 239, 54506 Vandoeuvre-lès-Nancy Cedex, France

JEAN-MICHEL MULLER
CNRS, Projet CNRS/ENS Lyon/INRIA Arenaire, Laboratoire LIP, Ecole Normale Supérieure de Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France

Received October 30, 2000; Revised July 26, 2001

Abstract. In several cases, the input argument of an elementary function evaluation is given bit-serially, most significant bit first. We suggest a solution for performing the first step of the evaluation (namely, the range reduction) *on the fly*: the computation is overlapped with the reception of the input bits. This algorithm can be used for the trigonometric functions sin, cos, tan as well as for the exponential function.

Keywords: range reduction, elementary functions, computer arithmetic

1. Introduction

The algorithms used for evaluating the elementary functions only give a correct result if the argument is within some bounded interval. To evaluate an elementary function $f(x)$ (sine, cosine, exponential, ...) for any x, one must find some "transformation" that makes it possible to deduce $f(x)$ from some value $g(y)$, where

- y, called the *reduced argument*, is deduced from x;
- y belongs to the convergence domain of the algorithm implemented for the evaluation of g.

With the usual functions, the only cases for which reduction is not straightforward are the cases where y is equal to $x - nC$, where n is an integer and C a constant (for instance, for the trigonometric functions, C is a multiple of $\pi/8$).

Example 1 (Computation of the cosine function). Assume that we want to evaluate $\cos(x)$, and that the convergence domain of the algorithm used to evaluate the sine and cosine of the reduced argument contains $[0, +\pi/4]$. We choose $C = \pi/4$, and the computation of $\cos(x)$ is decomposed in three steps:

- Compute y and n such that $y \in [0, +\pi/4]$ and $y = x - n\pi/4$;
- Compute

$$g(y,n) = \begin{cases} \cos(y) & \text{if } n \bmod 8 = 0 \\ \frac{\sqrt{2}}{2}(\cos(y) - \sin(y)) & \text{if } n \bmod 8 = 1 \\ -\sin(y) & \text{if } n \bmod 8 = 2 \\ -\frac{\sqrt{2}}{2}(\cos(y) + \sin(y)) & \text{if } n \bmod 8 = 3 \\ -\cos(y) & \text{if } n \bmod 8 = 4 \\ \frac{\sqrt{2}}{2}(-\cos(y) + \sin(y)) & \text{if } n \bmod 8 = 5 \\ \sin(y) & \text{if } n \bmod 8 = 6 \\ \frac{\sqrt{2}}{2}(\cos(y) + \sin(y)) & \text{if } n \bmod 8 = 7 \end{cases} \quad (1)$$

- Obtain $\cos(x) = g(y, n)$.

*This paper is an extended version of a communication to the SPIE's 45th annual meeting, San Diego, Aug. 2000.

Example 2 (Computation of the exponential function). Assume that we want to evaluate e^x in a radix-2 number system, and that the convergence domain of the algorithm used to evaluate the exponential of the reduced argument contains $[0, \ln(2)]$. We can choose $C = \ln(2)$, and the computation of e^x is then decomposed in three steps:

- Compute $y \in [0, \ln(2)]$ and n such that $y = x - n\ln(2)$;
- Compute $g(y) = e^y$;
- Compute $e^x = 2^n g(y)$.

Unless multiple-precision arithmetic is used during the intermediate calculations, a straightforward computation of y as $x - nC$ is to be avoided, since this operation will lead to catastrophic cancellations (i.e., to very inaccurate estimates of y) when x is large or close to an integer multiple of C. Many algorithms have been suggested for performing the range reduction accurately [1–5].

Now, there are many cases (on special-purpose systems) where the input argument of a calculation is generated most significant digit first. This happens, for instance, when this argument is the result of a division or a square root obtained through a digit-recurrence algorithm [6, 7], the output of an on-line algorithm [8, 9], or when it is generated by an analog-to-digital converter.

In the rest of this paper, we present an adaptation of the Modular Range Reduction Algorithm [3, 10] that accepts such digit serial inputs and performs the range reduction "on the fly": most of the computation is overlapped with the reception of the input bits, and the reduced argument is produced almost immediately after reception of the last input bit. On-the-fly arithmetic algorithms have already been proposed by Ercegovac and Lang for rounding or converting a number from redundant to non-redundant representation [11, 12].

2. Notations

In the rest of the paper, $x = x_h x_{h-1} \cdots x_0 . x_{-1} x_{-2} \cdots x_\ell$ is the input argument, $C = 0.C_{-1}C_{-2}\cdots C_{-p}$ is the constant of the range reduction (with $-p \leq \ell$), and $y = 0.y_{-1}y_{-2}\cdots y_{-p}$ is the reduced argument. We assume $1/2 \leq C < 1$ (this assumption is made to simplify the presentation. Modifying the algorithm for a constant larger than 1 or less than $1/2$ is straightforward). These values satisfy:

- $0 \leq y < C$;
- $n = (x - y)/C$ is an integer.

We also define, for each i, m_i (also called $2^i \bmod C$) as the unique value between 0 and C such that $(2^i - m_i)/C$ is an integer. These notations give some constraints on x and C (e.g., C is less than 1, x is less than 2^{h+1}). One can easily adapt the algorithms given in the rest of the paper to variables belonging to other domains. We chose these constraints to make the presentation of the algorithms simpler.

3. Non-Redundant Algorithm

Algorithm 1 is by far less efficient than the "redundant" algorithm given later. We give it because it is simpler to understand, and because the other algorithm is derived from it. The basic idea is the following: at step i of the algorithm, when we receive input bit x_{h-i} of x, we add $x_{h-i} \times \left(2^i \bmod C\right)$ to an accumulator. If the accumulated value becomes larger than C, we subtract C from it.

Let us call A_{i+1} the value obtained after this operation. One can easily check that $0 \leq A_{i+1} < C$ and $A_{i+1} - x_h x_{h-1} \cdots x_{h-i} \times 2^{h-i}$ is an integer multiple of C. Hence the final value stored in the accumulator is equal to the reduced argument y.

Algorithm 1 Non-redundant algorithm.

$A_0 = 0$
for $i = 0$ **to** $h - \ell$ **do**
  $T_i = A_i + x_{h-i}\, m_{h-i}$
  if $T_i < C$ **then**
    $A_{i+1} = T_i$
  else
    $A_{i+1} = T_i - C$
$y = A_{h-\ell+1}$

A possible variant consists in computing $U_i = A_i + x_{h-i}\,(m_{h-i} - C)$ in parallel with $T_i = A_i + x_{h-i} m_{h-i}$, and then choosing A_{i+1} equal to U_i if $U_i \geq 0$, otherwise T_i.

4. Redundant Algorithm

Now, to accelerate the reduction, we assume that we perform the accumulations with *carry-save* additions. The carry-save number system makes it possible to perform very fast, carry-free additions. On the other hand,

its intrinsic redundancy makes comparisons somewhat more complex. The accumulator will store the values A_i in carry-save. In the previous algorithm, we needed "exact" comparisons between the A_i's and C. Having the A_i's stored in carry-save makes these "exact" comparisons difficult. Instead of that, we will perform comparisons based on the examination of the first three carry-save positions of A_i only. This will not allow us to bound the A_i's by C. Nevertheless, we will show that the A_i's will be upper-bounded by $C + \frac{1}{2}$ (therefore by $\frac{3}{2}$), which will suffice for our purpose. We denote:

$$A_i = \left(\left(A_{i,0}^{(1)}, A_{i,0}^{(2)}\right); \left(A_{i,-1}^{(1)}, A_{i,-1}^{(2)}\right); \left(A_{i,-2}^{(1)}, A_{i,-2}^{(2)}\right); \cdots; \left(A_{i,-p}^{(1)}, A_{i,-p}^{(2)}\right)\right)$$

where $A_{i,j}^{(1)}$ and $A_{i,j}^{(2)}$ are in $\{0, 1\}$ and

$$A_i = \sum_{j=0}^{p} \left(A_{i,j}^{(1)} + A_{i,j}^{(2)}\right) \cdot 2^{-j}.$$

The variable T_i of the non-redundant algorithm is used again, and is also represented in carry-save form:

$$T_i = \left(\left(T_{i,0}^{(1)}, T_{i,0}^{(2)}\right); \left(T_{i,-1}^{(1)}, T_{i,-1}^{(2)}\right); \left(T_{i,-2}^{(1)}, T_{i,-2}^{(2)}\right); \cdots; \left(T_{i,-p}^{(1)}, T_{i,-p}^{(2)}\right)\right)$$

Algorithm 2 Redundant algorithm.

$A_0 = 0$
for $i = 0$ **to** $h - \ell$ **do**
  $T_i = A_i +_{\text{cs}} x_{h-i}\, m_{h-i}$
  $\hat{T}_i = \left(\left(T_{i,0}^{(1)}, T_{i,0}^{(2)}\right); \left(T_{i,-1}^{(1)}, T_{i,-1}^{(2)}\right); \left(T_{i,-2}^{(1)}, T_{i,-2}^{(2)}\right)\right)$
    converted to non-redundant binary using a 3-bit adder
  if $\hat{T}_i < C$ **then**
    $A_{i+1} = T_i$
  else
    $A_{i+1} = T_i -_{\text{cs}} C$
$B = A_{h-\ell+1} -_{\text{cs}} C$
Convert $A_{h-\ell+1}$ and B to non-redundant binary.
if $B < 0$ **then**
  $y = A_{h-\ell+1}$
else
  $y = B$

This gives Algorithm 2.

In the loop, we do not want to waste time with a full comparison to know whether we need to subtract C from T_i or not. Thus we use a rough approximation $\hat{T}_i$ to T_i based on the first three digits of T_i. Since

$$\left(\left(T_{i,-3}^{(1)}, T_{i,-3}^{(2)}\right); \cdots; \left(T_{i,-p}^{(1)}, T_{i,-p}^{(2)}\right)\right) \leq 2 \cdot 2^{-3} + 2 \cdot 2^{-4} + \cdots + 2 \cdot 2^{-p} < \frac{1}{2},$$

we have:

$$\hat{T}_i \leq T_i < \hat{T}_i + \frac{1}{2}$$

We want to ensure that A_i is always positive, that is, $T_i - C$ does not lead to a negative number. Then, the subtraction is performed only when $\hat{T}_i \geq C$. In this case, $T_i - C \geq \hat{T}_i - C \geq 0$.

Now, we want to find an upper bound on all the A_i's (and one on the T_i's). Suppose that for a given i, we have $A_i \leq M$. Thus $T_i \leq M + C$. If $\hat{T}_i < C$, then $A_{i+1} = T_i < \hat{T}_i + \frac{1}{2} < C + \frac{1}{2}$; otherwise, $A_{i+1} = T_i - C \leq M$. If we choose $M = C + \frac{1}{2}$, then $A_{i+1} \leq M$ in both cases. By induction, $A_i \leq C + \frac{1}{2}$ and $T_i \leq 2C + \frac{1}{2}$ for all i.

The final value of y is converted to non-redundant representation using a conventional (i.e., non-redundant) addition. Another, faster, solution is to convert it on-the-fly, during the second loop of the algorithm, using Ercegovac and Lang's on-the-fly algorithm [11, 12] for conversion from redundant to non-redundant representation.

5. An Example: Range Reduction for Computation of cos(1010.111)

We choose $C = \pi/4 \approx 0.1100101$ ($p = 7$). Since $x = 1010.111$, we have $h = 3$ and $\ell = -3$).

The values of the m_i's are:

$$\begin{cases} m_3 = 2^3 \mod \pi/4 \approx 0.0010011 \\ m_2 = 2^2 \mod \pi/4 \approx 0.0001001 \\ m_1 = 2^1 \mod \pi/4 \approx 0.0110111 \\ m_0 = 2^0 \mod \pi/4 \approx 0.0011011 \\ m_{-1} = 2^{-1} \mod \pi/4 = 0.1 \\ m_{-2} = 2^{-2} \mod \pi/4 = 0.01 \\ m_{-3} = 2^{-3} \mod \pi/4 = 0.001 \end{cases}$$

The carry-save representations of the variables T_i and A_i generated by the redundant algorithm are

$x_3 = 1$	$T_0 = \begin{cases} 1.0010011 \\ 0.0000000 \end{cases}$	$0 < C$	$A_1 = \begin{cases} 1.0010011 \\ 0.0000000 \end{cases}$
$x_2 = 0$	$T_1 = \begin{cases} 1.0010011 \\ 0.0000000 \end{cases}$	$0 < C$	$A_2 = \begin{cases} 1.0010011 \\ 0.0000000 \end{cases}$
$x_1 = 1$	$T_2 = \begin{cases} 1.0100100 \\ 0.0100110 \end{cases}$	$0.1 < C$	$A_3 = \begin{cases} 1.0100100 \\ 0.0100110 \end{cases}$
$x_0 = 0$	$T_3 = \begin{cases} 1.0000010 \\ 0.1001000 \end{cases}$	$0.1 < C$	$A_4 = \begin{cases} 1.0000010 \\ 0.1001000 \end{cases}$
$x_{-1} = 1$	$T_4 = \begin{cases} 1.0001010 \\ 1.0000000 \end{cases}$	$1 \geq C$	$A_5 = \begin{cases} 1.0010001 \\ 0.0010100 \end{cases}$
$x_{-2} = 1$	$T_5 = \begin{cases} 1.0100101 \\ 0.0100000 \end{cases}$	$0.1 < C$	$A_6 = \begin{cases} 1.0100101 \\ 0.0100000 \end{cases}$
$x_{-3} = 1$	$T_6 = \begin{cases} 1.0010101 \\ 0.1000000 \end{cases}$	$0.1 < C$	$A_7 = \begin{cases} 1.0010101 \\ 0.1000000 \end{cases}$

We then get $y = 0.1010101$, whereas the exact value of $x \bmod \pi/4$ is $0.10101010001\ldots$.

6. Error Analysis

Performing an error analysis for a range reduction algorithm requires the knowledge of the smallest possible reduced argument for all possible inputs in a given format. Computing this value is rather easy, using an algorithm due to Kahan (a C program that implements the method can be found at `http://http.cs.berkeley.edu/~wkahan`. A Maple program is given in [10]). For instance, a few minutes of calculation suffice to find that the double precision number greater than 1 which is closest to a multiple of $\pi/8$ is

$$\Gamma = 6381956970095103 \times 2^{795}.$$

The distance between Γ and the closest multiple of $\pi/8$ is

$$\epsilon \approx 1.17179 \times 10^{-19}.$$

Assume that the smallest nonzero possible value of the reduced argument is ϵ, that the A_i's are stored in a fixed-point accumulator, that the m_i's are precomputed and stored with p fractional bits of accuracy, and that we want the reduced argument to have at least ν bits of accuracy. In Algorithms 1 and 2, the additions are errorless (they are fixed-point additions), the errors come from the fact that the stored values of the m_i's are approximations to their exact values. Assuming we store rounded-to-nearest values for these constants, the error on each of them is bounded by 2^{-p-1}. Hence, the total error on the reduced argument is $(h - \ell + 1) \times 2^{-p-1}$. Therefore, our accuracy requirement is:

$$-\log_2(|\epsilon|) + \nu \leq p + 1 - \log_2(h - \ell + 1)$$

For instance, in double precision with $C = \pi/8$, with $\nu = 53$, we get $p > 120$. Therefore, a 121-bit fixed point accumulator suffices to make sure that we always get an excellent result in double precision.

7. Conclusion

The redundant algorithm presented in Section 4 allows fast, on-the-fly, range reduction. The accuracy of this method is the same as that of the Conventional Modular range reduction method see [3, 10].

References

1. W. Cody and W. Waite, *Software Manual for the Elementary Functions*, Englewood Cliffs, NJ: Prentice-Hall, 1980.
2. W.J. Cody, "Implementation and Testing of Function Software," in *Problems and Methodologies in Mathematical Software Production*, P.C. Messina and A. Murli (Eds.), Berlin: Springer-Verlag, 1982.
3. M. Daumas, C. Mazenc, X. Merrheim, and J.M. Muller, "Modular Range Reduction: A New Algorithm for Fast and Accurate Computation of the Elementary Functions," *Journal of Universal Computer Science*, vol. 1, no. 3, 1995, pp. 162–175.
4. M. Payne and R. Hanek, "Radian Reduction for Trigonometric Functions," *SIGNUM Newsletter*, vol. 18, 1983, pp. 19–24.
5. R.A. Smith, "A Continued-Fraction Analysis of Trigonometric Argument Reduction," *IEEE Transactions on Computers*, vol. 44, no. 11, 1995, pp. 1348–1351.
6. M.D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Boston: Kluwer Academic Publishers, 1994.
7. J.E. Robertson, "A New Class of Digital Division Methods," *IRE Transactions on Electronic Computers*, vol. EC-7, 1958, pp. 218–222. Reprinted in E.E. Swartzlander, *Computer Arithmetic*, vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990, pp. 159–163.
8. M.D. Ercegovac and T. Lang, "On-Line Arithmetic: A Design Methodology and Applications in Digital Signal Processing," in *VLSI Signal Processing III*, vol. 3, 1988, pp. 252–263. Reprinted in E.E. Swartzlander, *Computer Arithmetic*, vol. 2, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990, pp. 66–77.
9. K.S. Trivedi and M.D. Ercegovac, "On-Line Algorithms for Division and Multiplication," in *3rd IEEE Symposium on Computer Arithmetic*, Dallas, Texas, USA, 1975, pp. 161–167, Los Alamitos, CA: IEEE Computer Society Press.

10. J. Muller, *Elementary Functions, Algorithms and Implementation*, Boston: Birkhauser, 1997.
11. M.D. Ercegovac and T. Lang, "On-the-Fly Conversion of Redundant Into Conventional Representations," *IEEE Transactions on Computers*, vol. C-36, no. 7, 1987. Reprinted in E.E. Swartzlander, *Computer Arithmetic*, vol. 2, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990, pp. 123–125.
12. M.D. Ercegovac and T. Lang, "On-the-Fly Rounding," *IEEE Transactions on Computers*, vol. 41, no. 12, 1992, pp. 1497–1503.

Vincent Lefèvre received the MSC and Ph.D. degrees in computer science from the École Normale Supérieure de Lyon, France, in 1996 and 2000, respectively. Since 2000, he has been an INRIA researcher at the LORIA, France. His research interests include computer arithmetic.
Vincent.Lefevre@inria.fr

Jean-Michel Muller received the Ph.D. degree in computer science from Institut National Polytechnique de Grenoble, France. In 1986, he joined the CNRS (French National Center for Scientific Research). He chairs the LIP Laboratory, located at Ecole Normale Supérieure de Lyon, and the CNRS/ENSL/INRIA Arenaire Project. His research interests are in computer arithmetic. Dr. Muller served as co-program chair of the 13th IEEE Symposium on Computer Arithmetic and general chair of the 14th IEEE Symposium on Computer Arithmetic. He has been an associate editor of the IEEE Transactions on Computers from 1996 to 2000.
Jean-Michel.Muller@ens-lyon.fr