Journal of Universal Computer Science, vol. 1, no. 3 (1995), 162-175
submitted: 4/1/95, accepted: 1/3/95, appeared: 28/3/95, ©Springer Pub. Co.

Modular Range Reduction: a New Algorithm for Fast and Accurate Computation of the Elementary Functions

Marc Daumas
(Lab. LIP, Ecole Normale Supérieure de Lyon
Marc.Daumas@lip.ens-lyon.fr)

Christophe Mazenc
(Lab. LIP, Ecole Normale Supérieure de Lyon
Christophe.Mazenc@lip.ens-lyon.fr)

Xavier Merrheim
(Lab. LIP, Ecole Normale Supérieure de Lyon
Xavier.Merrheim@lip.ens-lyon.fr)

Jean-Michel Muller
(CNRS, Lab. LIP, Ecole Normale Supérieure de Lyon
Jean-Michel.Muller@lip.ens-lyon.fr)

Abstract: A new range reduction algorithm, called *Modular Range Reduction (MRR)*, briefly introduced by the authors in [Daumas et al. 1994] is deeply analyzed. It is used to reduce the arguments to exponential and trigonometric function algorithms to be within the small range for which the algorithms are valid. MRR reduces the arguments quickly and accurately. A fast hardwired implementation of MRR operates in time $O(\log(n))$, where n is the number of bits of the binary input value. For example, with MRR it becomes possible to compute the sine and cosine of a very large number accurately. We propose two possible architectures implementing this algorithm.

Key Words: Computer Arithmetic, Elementary Functions, Range Reduction
Category: B.2, G.1.0

1 Introduction

The algorithms used for evaluating the elementary functions (polynomial or rational approximations [Cody and Waite 1980, Remes 1934], Taylor expansions, shift-and-add algorithms — see [Ercegovac 1973],[DeLugish 1970],[Walther 1971] and [Asada et al. 1987], table lookup methods...) only give a correct result if the argument is within a given bounded interval. In order to evaluate an elementary function $f(x)$ for any x, one must find some "transformation" that makes it possible to deduce $f(x)$ from some value $g(x^*)$, with:

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

In practice, there are two different kinds of reduction:

1. *Additive reduction.* x^* is equal to $x - kC$, where k is an integer and C a constant (for instance, for the trigonometric functions, C is a multiple of $\pi/4$).

2. *Multiplicative reduction.* x^* is equal to x/C^k, where k is an integer and C a constant (for instance, for the logarithm function, a convenient choice for C is the radix of the number system).

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 $[-\pi/4, +\pi/4]$. We choose $C = \pi/2$, and the computation of $\cos(x)$ is decomposed in three steps:

– Compute x^* and k such that $x^* \in [-\pi/4, +\pi/4]$ and $x^* = x - k\pi/2$
– Compute $g(x^*, k) =$

$$\begin{cases} \cos(x^*) & \text{if } k \bmod 4 = 0 \\ -\sin(x^*) & \text{if } k \bmod 4 = 1 \\ -\cos(x^*) & \text{if } k \bmod 4 = 2 \\ \sin(x^*) & \text{if } k \bmod 4 = 3 \end{cases} \tag{1}$$

– Obtain $\cos(x) = g(x^*, k)$

The previous reduction mechanism is an *additive reduction.* Let us examine another example of additive reduction.

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 choose $C = \ln(2)$, and the computation of e^x is decomposed in three steps:

– Compute $x^* \in [0, \ln(2)]$ and k such that $x^* = x - k\ln(2)$.
– Compute $g(x^*) = e^{x^*}$
– Compute $e^x = 2^k g(x^*)$

The radix-2 number system makes the final multiplication by 2^k straightforward.

Another way of performing the range reduction for the exponential function (with an algorithm whose convergence domain is $[0, 1]$) is to choose $x^* = x - \lfloor x \rfloor$, $k = \lfloor x \rfloor$, and $g(x^*) = e^{x^*}$. Then, $e^x = g(x^*) \times e^k$, and e^k can either be evaluated by performing a few multiplications — since k is an integer — or by table-lookup. Usually, this latter method is preferable, since there is no loss of accuracy when computing x^*. With the range reduction algorithm we give in the following, the former choices for x^*, g, and k become interesting, for several reasons:

– we will be able to compute x^* very accurately,
– the required convergence interval is smaller, which means that to reach the same accuracy, we need a smaller number of coefficients for a polynomial or rational approximation,
– there will not be any error when deducing e^x from $g(x^*)$.

Anyway, range reduction is more a problem for trigonometric functions than for exponentials, since, in practice, we never have to deal with exponentials of very large numbers: they merely are overflows!

Example 3 Computation of the logarithm function. Assume that we want to evaluate $\ln(x)$, $x > 0$, in a radix-2 number system, and that the convergence domain of the algorithm used to compute the logarithm of the reduced argument contains $[1/2, 1]$. We choose $C = 2$, and the computation of $\ln(x)$is decomposed in three steps:

- Compute $x^* \in [1/2, 1]$ and k such that $x^* = x/2^k$ (if x is a normalized radix-2 floating-point number, x^* is its mantissa, while k is its exponent).
- Compute $g(x^*, k) = \ln(x^*)$
- Compute $\ln(x) = g(x^*, k) + k \ln(2)$

The previous mechanism is a *multiplicative* reduction.

In practice, multiplicative reduction is not a problem: when computing the usual mathematical functions, it only occurs with logarithms and n-th roots. With these functions, as in the example above, C can be chosen equal to a power of the radix of the number system. This makes the computation of x/C^k straightforward. Therefore, in the following, we concentrate on the problem of *additive* range reduction only.

2 The Modular Range Reduction Algorithm

We focus on the following problem: we assume that we have an algorithm able to compute the function g in an interval I of the form $[-C/2 - \epsilon, +C/2 + \epsilon]$ (we call this case *"symmetrical reduction"*) or $[-\epsilon, C + \epsilon]$ (we call this case *"positive reduction"*), with $\epsilon \geq 0$. We want to compute $x^* \in I$ and an integer k such that:

$$x^* = x - kC \tag{2}$$

If $\epsilon > 0$, then x^* and k are not uniquely defined by Eq. 2. In such a case, the problem of deducing these values from x will be called *"redundant range reduction"*. For example, if $C = \frac{\pi}{2}, I = [-1, 1]$ and $x = 2.5$, then $k = 1$ and $x^* = 0.929203\ldots$ or $k = 2$ and $x^* = -0.641592\ldots$ are possible values. If $\epsilon = 0$, this problem is called *"non-redundant range reduction"*. As in many fields of computer arithmetic, redundancy will allow faster algorithms. Table 1 sums-up the different possible cases.

	$I = [-C/2 - \epsilon, C/2 + \epsilon]$	$I = [-\epsilon, C + \epsilon]$
$\epsilon = 0$	symmetrical non-redundant	positive non-redundant
$\epsilon \neq 0$	symmetrical redundant	positive redundant

Table 1. The different cases in additive range-reduction

It is worth noticing that:

1. In some practical cases, it is not necessary to fully compute k. For instance, for the trigonometric functions, if $C = \pi/2$, then one just needs to know $k \bmod 4$. If $C = 2\pi$, there is no need for any information about k.
2. With the usual algorithms for evaluating the elementary functions, one can assume that the length of the convergence domain I is greater than C, i.e. that we can perform a *redundant* range reduction. For instance, with the CORDIC algorithm, when performing rotations (see [Walther 1971]), the convergence domain is $[-1.743\ldots, +1.743\ldots]$, which is much larger than $[-\pi/2, +\pi/2]$. With polynomial or rational approximations, the convergence domain can be enlarged by adding one coefficient to the approximation.

Let us define $\lfloor x \rceil$ as the nearest integer to x. Usually, the range reduction is done by:

- Computing $k = \lfloor x/C \rfloor$ (in the case of a positive reduction), or $k = \lfloor x/C \rceil$ (in the case of a symmetrical reduction) by the means of multiplication or division.
- Computing the reduced argument $x^* = x - kC$ by the means of a multiplication by k or a table-lookup if the values kC are pre computed and stored followed by a subtraction.

The above process may be rather inaccurate (for large values of x, the final subtraction step leads to a catastrophic cancellation — although cunning tricks have been proposed to limit this cancellation [Cody and Waite 1980]). In [Daumas et al. 1994] we briefly proposed a new algorithm, called the *modular reduction algorithm* (MRR), that performs the range reduction quickly and accurately. In the following we deeply analyze that algorithm in terms of speed and accuracy, and we propose architectures implementing it.

2.1 Fixed-point reduction

First of all, we assume that the input operands are *Fixed-point radix-2 numbers*, less than 2^N. These numbers have a N-bit integer part and a p-bit fractional part. So the digit chain:

$$x_{N-1}x_{N-2}x_{N-3}\ldots x_0.x_{-1}x_{-2}\ldots x_{-p}, \quad \text{where } x_i \in \{0,1\}$$

represents the number $\sum_{i=-p}^{N-1} x_i 2^i$.
We assume that we should perform a *redundant* range reduction, and we call ν the integer such that $2^\nu < C \leq 2^{\nu+1}$.
Let us define, for $i \geq \nu$ the number $m_i \in [-C/2, C/2)$ such that $\frac{2^i - m_i}{C}$ is an integer (in the following, we will write "$m_i \equiv 2^i \bmod C$"). The Modular Range Reduction (MRR) algorithm consists in performing the two following steps:

First reduction We compute the number[1]:

$$r = (x_{N-1}m_{N-1}) + (x_{N-2}m_{N-2}) + (x_{N-3}m_{N-3}) + \ldots + (x_\nu m_\nu) \\ +x_{\nu-1}x_{\nu-2}x_{\nu-3}\ldots x_0.x_{-1}x_{-2}\ldots x_{-p} \quad (3)$$

Since the x_i's are equal to 0 or 1, this computation is reduced to the sum of $N-\nu+1$ terms. The result r of this first reduction is between $-(N-\nu+2)C/2$ and $+(N-\nu+2)C/2$. This is a consequence of the fact that all the $x_i m_i$ have an absolute value less than $C/2$, and

$$x_{\nu-1}x_{\nu-2}x_{\nu-3}\ldots x_0.x_{-1}x_{-2}\ldots x_{-p}$$

has an absolute value less than 2^ν, which is less than C.

Second reduction Define the r_i's as the digits of the result of the first reduction:

$$r = r_\ell r_{\ell-1} r_{\ell-2}\cdots r_0.r_{-1}r_{-2}\cdots$$

where $\ell = \lfloor \log_2(N-\nu+2)\rfloor$.

Let us define $\hat{r}$ as the number obtained by truncating the binary representation of r after the $\lceil -\log_2(\epsilon)\rceil$-th bit, that is (using the relation $-\lceil -x\rceil = \lfloor x\rfloor$):

$$\hat{r} = r_\ell r_{\ell-1} r_{\ell-2}\cdots r_0.r_{-1}r_{-2}\cdots r_{\lfloor \log_2(\epsilon)\rfloor}$$

$\hat{r}$ is an m-digit number, where $m = \lfloor \log_2(N-\nu+2)\rfloor + \lceil -\log_2(\epsilon)\rceil$ is a very small number in all practical cases (see the example below). If we define k as $\lfloor \frac{\hat{r}}{C}\rceil$ (resp. $\lfloor \frac{\hat{r}}{C}\rfloor$) then $r - kC$ will belong to $[-C/2-\epsilon, +C/2+\epsilon]$ (resp. $[-\epsilon, C+\epsilon]$), i.e. it will be the correct result of the symmetrical (resp. positive) range reduction.

Proof

1. *In the symmetrical case.* We have $|k - \frac{\hat{r}}{C}| \le \frac{1}{2}$, therefore $|\hat{r} - kC| \le C/2$. From the definition of $\hat{r}$, $|r-\hat{r}| \le 2^{\lfloor \log_2(\epsilon)\rfloor} \le \epsilon$, therefore:
$$|r - kC| \le \frac{C}{2} + \epsilon$$
2. *In the positive case.* We have $k \le \frac{\hat{r}}{C} < k+1$, therefore $0 \le \hat{r} - kC < C$, therefore $-\epsilon \le r - kC < C + \epsilon$.

Since k can be deduced from $\hat{r}$, this second reduction step will be implemented by looking up the value kC in a 2^m-bit entry table at the address constituted by the bits of $\hat{r}$. Fig. 1 Sums up the different steps of MRR.

During this reduction process, we perform the addition of $N-\nu+1$ terms. If these terms (namely the m_i's and the value kC of the second reduction step) are represented in fixed-point with q fractional bits (i.e. the error on each of these term is bounded by 2^{-q-1}), then the difference between the result of the computation and the exact reduced argument is bounded by $2^{-q-1}(N-\nu+1)$. In order to obtain the reduced argument x^* with the same absolute accuracy as the input argument x (i.e. p significant fixed-point fractional digits), one needs to store the m_i's and the values kC with $p + \lceil \log_2(N-\nu+1)\rceil$ fractional bits.

[1] This formula looks correct only for positive values of ν. It would be more correct, although maybe less clear, to write: $r = \sum_{i=\nu}^{N-1} x_i m_i + \sum_{i=-p}^{\nu-1} x_i 2^i$

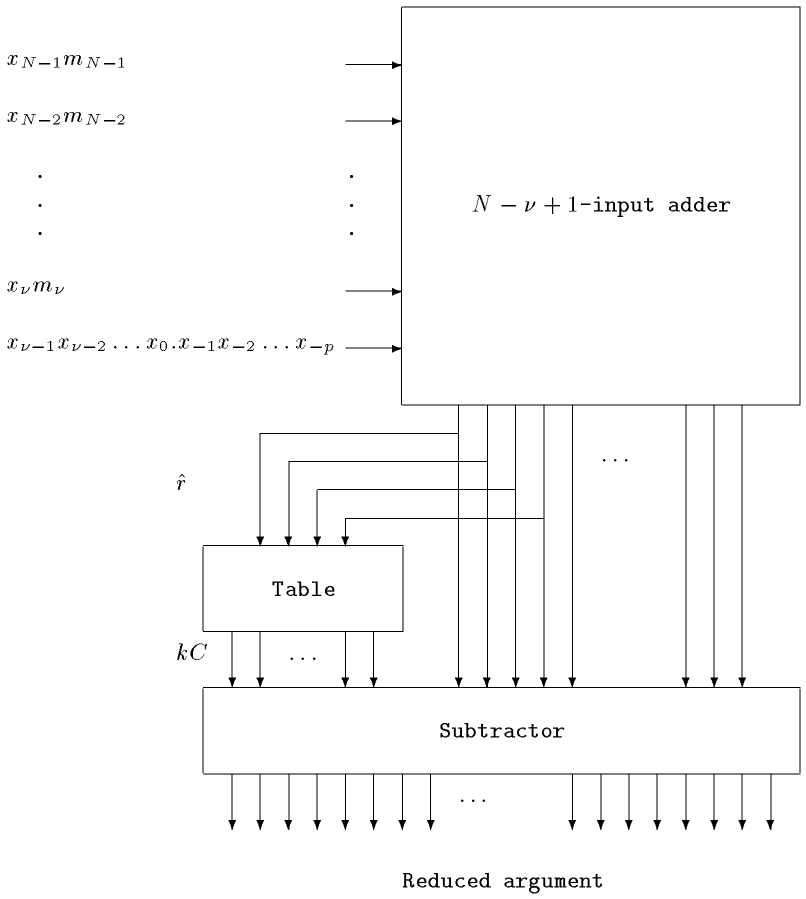

Fig. 1. The Modular Reduction Algorithm

Example 4. Assume we need to compute sines of angles between -2^{20} and 2^{20}, and that the algorithm used with the reduced arguments is CORDIC [Volder 1959], [Walther 1971]. The convergence interval I is $[-1.743\dots,+1.743\dots]$, therefore (since $1.743 > \frac{\pi}{2}$) we have to perform a *symmetrical redundant* range reduction, with $C=\pi$ and $\epsilon = +1.743\dots - \frac{\pi}{2} = 0.172\dots > 2^{-3}$. We immediately get the following parameters:

- $N = 20$ and $\nu = 2$
- The first range reduction consists of the addition of 19 terms
- $r \in [-10\pi, +10\pi]$, therefore, since $10\pi < 2^5$, the second reduction step requires a table with $5 + \lceil -\log_2 \epsilon \rceil = 8$-address bits.

– To obtain the reduced argument with p significant fractional bits, one needs to store the m_i's and the values kC with $p+5$ bits.

Assume we compute the sine of 355. The binary representation of 355 is 101100011. Therefore during the first reduction, we have to compute $m_8+m_6+m_5+m_1+1$, where:

– $m_8 = 256 - 81 \times \pi = 1.530995059226747684\ldots$
– $m_6 = 64 - 20 \times \pi = 1.1681469282041352307\ldots$
– $m_5 = 32 - 10 \times \pi = 0.5840734641020676153\ldots$
– $m_1 = 2 - \pi = -1.1415926535897932384 62\ldots$

We get $m_8+m_6+m_5+m_1+1 = 3.14162279794315729 21\ldots$ The second reduction consists in subtracting π from that result, which gives $0.000030144353364053 72\ldots$, the sine of which is $0.000030144353359488449\ldots$
Therefore, $\sin(355) = -0.000030144353359488449\ldots$

2.2 Floating-point reduction

Now, assume that the input value x is a radix-2 floating-point number:

$$x = 0.x_1x_2x_3\ldots \times x_n 2^{exponent}$$

The range reduction can be performed exactly as in the fixed-point case. During the first reduction, we replace the addition of the terms m_i by the addition of the terms $m_{exponent-i}$. As previously, $m_i \equiv 2^i \bmod C$ is the number belonging to $[-C/2, C/2)$ such that $\frac{2^i-m_i}{C}$ is an integer. The main difference between this reduction method and the other ones is that during the reduction process, we just add numbers (the m_i's) of the same order of magnitude, represented in fixed-point. This makes the reduction very accurate. One can easily show that if the m_i's and the terms kC of the second reduction are represented with q fractional bits then the *absolute* error on the reduced argument is bounded by $(n+1)2^{-q-1}$. Thus, for instance it is possible to compute with good accuracy the sine and cosine of a huge floating-point number. In floating point number systems, one would prefer informations on the *relative* error: this will be discussed later.

3 Architectures for Modular Reduction

The first reduction consists of adding $N - \nu + 1$ numbers. This addition can be performed in a redundant number system (carry-save or signed-digit) in order to benefit from the carry-free ability of such a system, and/or in an arborescent way. This problem is obviously closely related to the problem of multiplying two numbers (multiplying $x = \sum_{i=0}^{q} x_i 2^i$ by $y = \sum_{j=0}^{q} y_i 2^j$ reduces to computing the sum of the $q+1$ terms $y_j 2^j x$). Therefore, almost all the classical architectures proposed in the multiplication literature (see for instance [Braun 1963], [Dadda 1965], [Harata et al. 1987], [Nakamura 1986], [Takagi et al. 1985], [Wallace 1964]), can be slightly modified in order to be used for range reduction. For instance, the architecture shown Fig. 2 is obtained from Braun's cellular array multiplier [Braun 1963], while the logarithmic-time architecture shown in Fig. 4 is a

Wallace tree [Wallace 1964]. In Fig. 3, m_{ij} is the digit of weight 2^{-j} of m_i. This similarity between the Modular Range Reduction algorithm and the multiplication makes it possible to perform both operations with the same hardware, which can save some silicon area on a circuit.

The similarity between the range reduction and the multiplication leads us to another idea: in order to accelerate the first reduction, one can perform a Booth recoding [Booth 1951], or merely a modified booth recoding [Hwang 1979], of x. This would give a signed digit (with digits -1, 0 and 1) representation of x with at least half of the digits equal to zero. Then the number of terms to be added during the first reduction would be halved.

4 Accuracy of MRR with floating-point inputs

As pointed out in section 2.2, if the input value x is a m-mantissa bit radix-2 floating-point number, and if the terms m_i's of the first reduction and the terms kC of the second reduction are represented with q fractional bits then the *absolute* error on the reduced argument is bounded by $(m+1)2^{-q-1}$. This makes it possible to evaluate sines and cosines of huge floating-point numbers with a good *absolute* accuracy. However, in floating-point, one is more concerned with the *relative* accuracy: what could be done if the result of MRR is zero or a number close to zero? This would indicate that the exact reduced argument is very small, but the computed value would only have a few (maybe not at all) significant digits. In the sequel of this paper, we deal with that problem.

4.1 How could MRR results close to zero be avoided ?

As in section 2.2, the input number x is the radix-2 floating-point number:

$$x = 0.x_1x_2x_3\ldots x_m \times 2^{exponent}$$

Therefore, x is equal to 2^eM, where:

$$\begin{cases} M &= x_1x_2\ldots x_m \\ e &= exponent - m \end{cases}$$

M is an integer. Assume that the result of the range reduction is very small, say less than a very small real number ϵ. This means that there exists an integer k such that:

$$|x - kC| \leq \epsilon$$

This implies:

$$\left|\frac{2^eM}{k} - C\right| \leq \frac{\epsilon}{k} \tag{4}$$

This means that $\frac{2^eM}{k}$ is a very good *rational approximation* of C. To see to what extent relation (4) is likely to happen, we can compare this approximation to the sequence of the "best" possible rational approximations of C, i.e. the sequence of its *continued fraction approximations*. Let us call (P_i/Q_i) the sequence of the continued fraction approximations of C. For instance if $C = \pi$ then:

$$\frac{P_0}{Q_0} = 3 \quad \frac{P_1}{Q_1} = \frac{22}{7} \quad \frac{P_2}{Q_2} = \frac{333}{106} \; \frac{P_3}{Q_3} = \frac{355}{113} \quad \frac{P_4}{Q_4} = \frac{103993}{33102}$$

Let us define $\mu(k)$ as the integer such that:

$$Q_{\mu(k)-1} < k \leq Q_{\mu(k)}$$

Since (from the theory of continued fractions), for each rational number $\frac{n}{d}$ such that $d \leq Q_i$, we have:

$$\left|\frac{n}{d} - C\right| \geq \left|\frac{P_i}{Q_i} - C\right|$$

we deduce, from (4):

$$\left|\frac{P_{\mu(k)}}{Q_{\mu(k)}} - C\right| \leq \frac{\epsilon}{k}$$

Therefore:

$$\left|\frac{P_{\mu(k)}}{Q_{\mu(k)}} - C\right| Q_{\mu(k)-1} \leq \epsilon \tag{5}$$

So, if one wants to get a relative error ϵ_r for input values lower than x_{max}, the way to proceed is the following one:

- evaluate $k_{max} = \left\lfloor \frac{x_{max}}{C} \right\rceil$
- compute $\epsilon = \min_{k \leq k_{max}} \left|\frac{P_{\mu(k)}}{Q_{\mu(k)}} - C\right| Q_{\mu(k)-1}$
- make sure that the number q of fractional digits used for the reduction satisfies $\frac{(m+1)2^{-q-1}}{\epsilon} \leq \epsilon_r$

5 Conclusion

We have proposed a fast and accurate algorithm to perform the range reduction for computing the elementary functions. This algorithm can be implemented using a slightly modified multiplier, so that range reduction and multiplication can be performed using the same hardware, which can save some silicon area on a circuit. Using this algorithm, accurately and quickly computing the sine of a number such as 10^{200} becomes possible.

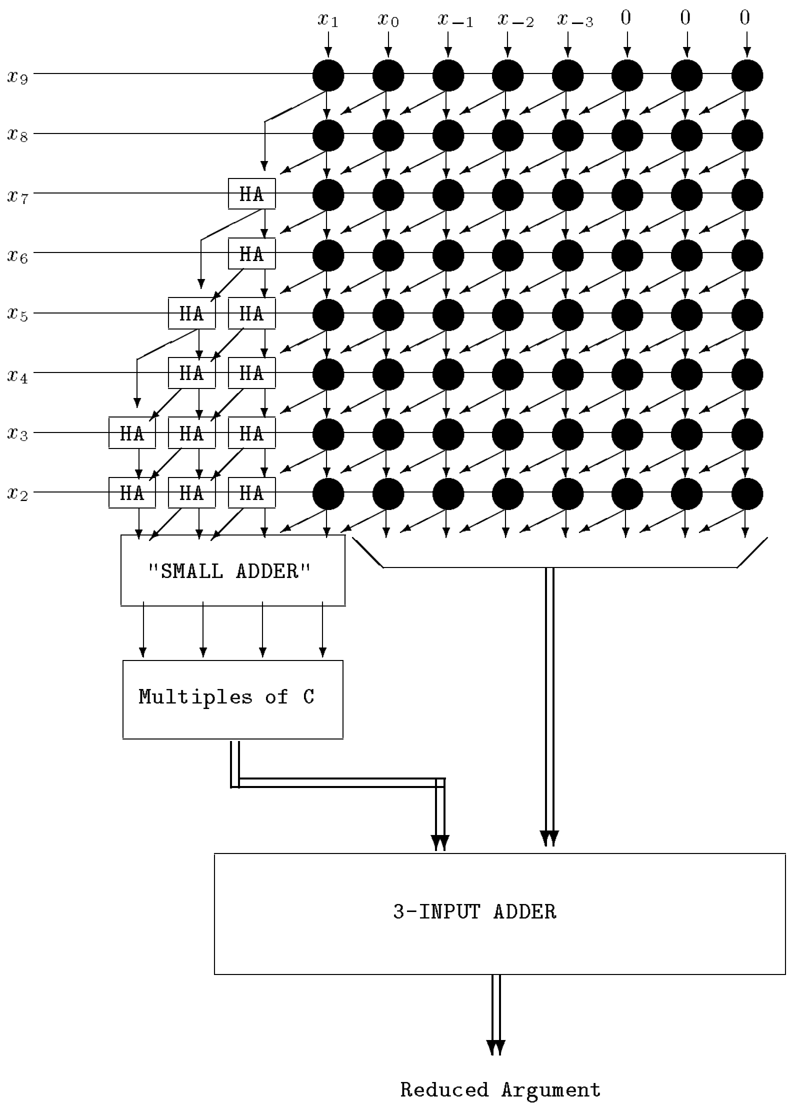

Fig. 2. A cellular array for range reduction

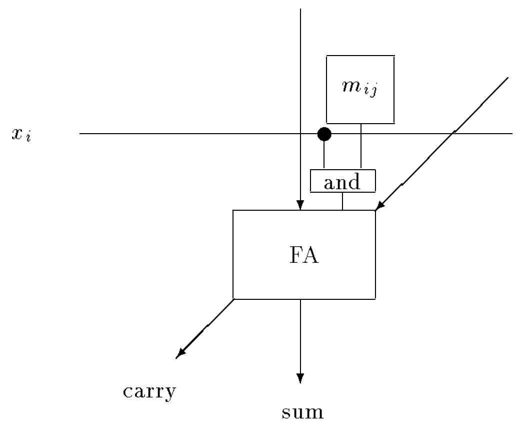

Fig. 3. A black cell of the cellular array

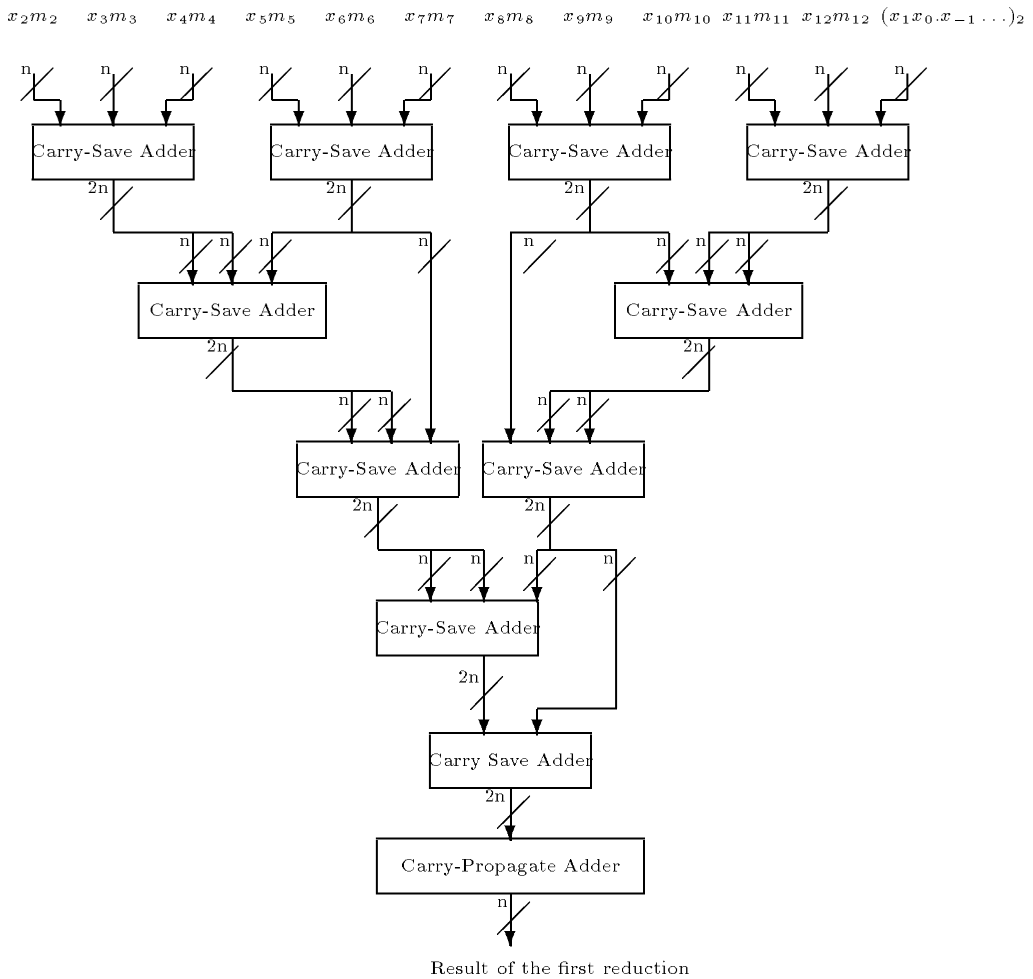
x_2m_2 x_3m_3 x_4m_4 x_5m_5 x_6m_6 x_7m_7 x_8m_8 x_9m_9 $x_{10}m_{10}$ $x_{11}m_{11}$ $x_{12}m_{12}$ $(x_1x_0.x_{-1}\ldots)_2$
n
2n
Carry-Save Adder
Carry Save Adder
Carry-Propagate Adder
Result of the first reduction

Fig. 4. A logarithmic-time range-reduction tree

References

[Asada et al. 1987] T. Asada, N. Takagi, and S. Yajima. A hardware algorithm for computing sine and cosine using redundant binary representation. *Systems and computers in Japan*, 18(8), 1987.

[Booth 1951] A.D. Booth. A signed binary multiplication technique. *Quarterly journal of mechanics and applied mathematics*, 4(2):236–240, 1951. Reprinted in E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[Braun 1963] E.L. Braun. *Digital computer design*. New York academic, 1963.

[Cody and Waite 1980] W. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall Inc, 1980.

[Dadda 1965] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34:349–356, March 1965. Reprinted in E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[Daumas et al. 1994] M. Daumas, C. Mazenc, X. Merrheim, and J.M. Muller. Fast and Accurate Range Reduction for the Computation of Elementary Functions. In *14th IMACS World Congress on Computational and Applied Mathematics*, Atlanta, Georgia, 1994.

[Ercegovac 1973] M.D. Ercegovac. Radix 16 evaluation of certain elementary functions. *IEEE Transactions on Computers*, C-22(6), June 1973. Reprinted in E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[Harata et al. 1987] Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, and N. Takagi. A high-speed multiplier using a redundant binary adder tree. *IEEE journal of solid-state circuits*, SC-22(1):28–34, February 1987. Reprinted in E.E. Swartzlander, Computer Arithmetic, Vol. 2, IEEE Computer Society Press Tutorial, 1990.

[Hwang 1979] K. Hwang. *Computer Arithmetic Principles, Architecture and design*. Wiley & Sons Inc, 1979.

[DeLugish 1970] B. De Lugish. *A Class of Algorithms for Automatic Evaluation of Functions and Computations in a Digital Computer*. PhD thesis, Dept. of Computer Science, University of Illinois, Urbana, 1970.

[Nakamura 1986] S. Nakamura. Algorithms for iterative array multiplication. *IEEE Transactions on Computers*, C-35(8), August 1986.

[Remes 1934] E. Remes. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Acad. Sci. Paris*, 198, 1934.

[Takagi et al. 1985] N. Takagi, H. Yasukura, and S. Yajima. High speed multiplication algorithm with a redundant binary addition tree. *IEEE Transactions on Computers*, C-34(9), September 1985.

[Volder 1959] J. Volder. The cordic computing technique. *IRE Transactions on Electronic Computers*, 1959. Reprinted in E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[Wallace 1964] C.S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, pages 14–17, February 1964. Reprinted in E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[Walther 1971] J. Walther. A unified algorithm for elementary functions. In *Joint Computer Conference Proceedings*, 1971. Reprinted in

E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.