Computing Functions $\cos^{-1}$ and $\sin^{-1}$ Using Cordic

Christophe Mazenc, Xavier Merrheim, and Jean-Michel Mullet

Abstract—After briefly recalling the main properties of the Cordic algorithm, we show that a slight modification of this algorithm enables the computation of the functions $\cos^{-1}$, $\sin^{-1}$, $\sqrt{1-t^2}$, $\cosh^{-1}$, $\sinh^{-1}$, and $\sqrt{1+t^2}$.

Index Terms—Computer arithmetic, Cordic, elementary functions.

I. INTRODUCTION

The Cordic algorithm was introduced in 1959 by Jack Volder [15]. This algorithm makes it possible to perform rotations (and therefore to compute sine, cosine, and $\tan^{-1}$ functions) and to multiply or divide numbers, using only shift-and-add elementary steps. In 1971, John Walther [16] generalized Volder’s algorithm in order to compute hyperbolic functions, logarithms, exponentials, and square roots. Cordic (or very similar algorithms) has been implemented in pocket calculators like Hewlett Packard’s HP 35 [4], and in arithmetic coprocessors like the Intel 8087. Several authors have proposed to use Cordic processors for signal processing applications (DFT or filtering [9]), for image processing [3], or for solving linear systems [1],[15].

Manuscript received April 19, 1991; revised March 2, 1992.

C. Mazenc and X. Merrheim are with Laboratoire LIP-IMAG, Ecole Normale Supérieure de Lyon, France.

J.-M. Muller is with CNRS, Laboratoire LIP-IMAG, Ecole Normale Supérieure de Lyon, France.

IEEE Log Number 9200214.

0018-9340/93$03.00 @ 1993 IEEE
In Walther’s version, Cordic consists of the following iteration:

\[
\begin{align*}
X_{n+1} &= X_n + d_n a_n y_n 2^{-s(n)} \\
y_{n+1} &= y_n + d_n e_n 2^{-s(n)} \\
\theta_{n+1} &= \theta_n + d_n e_n
\end{align*}
\]

where the results (i.e., the limit values of \(X_n, y_n\), and \(\theta_n\)) and the values of \(d_n\), \(a_n\), and \(e_n\) are presented in Fig. 1 and Table I. The constants \(e_n\) are precomputed and stored.

The function \(g\) is an artifact; in a practical implementation, the iterations are performed assuming \(u(n) = \text{const.}\), and in the hyperbolic mode (\(m = -1\)), the iterations 4, 13, 40, \(\cdots\), \(3k + 1\), are repeated. This repetition is necessary since the sequence \(e_n = \tanh^{-1} 2^{-n}\) does not satisfy the relation of Theorem 1 (see below), while the sequence \(e_n\) obtained from \(g_n\) by repeating the terms of indexes 4, 13, \(\cdots\), satisfies this relation. \(K'\) and \(K''\) are equal to

\[
\begin{align*}
K' &= \prod_{n=1}^{\infty} \cos(e_n) = \prod_{n=1}^{\infty} \sqrt{1 - 2^{-2s(n)}} \\
K'' &= 0.82815369062\cdots
\end{align*}
\]

Cordic is a very useful algorithm, since it allows computation of some of the most common mathematical functions [16]. For instance, it is obtained by adding \(\cosh\) and \(\sinh\) \(\ln\) \(x\) is obtained using the relation

\[
\ln (x) = 2 \tanh^{-1} \frac{|1 - x|}{|1 + x|}
\]

while \(\sqrt{x}\) is obtained by

\[
\sqrt{x} = \sqrt{\left(\frac{x + 1}{2}\right)^2 - \left(\frac{x - 1}{2}\right)^2}
\]

On-line implementations of Cordic have been proposed by several authors (see for instance [10], [11]). In order to explain our algorithm, let us clearly analyze Volder’s version of Cordic in the rotation mode.

First of all, we start from the following theorem (see [13] for proof):

**Theorem 1:** If \((\epsilon_n)\) is a decreasing sequence of positive real numbers such that \(\sum_{n=0}^{\infty} \epsilon_n < +\infty\), and if for any integer \(n\), \(\epsilon_n \leq \sum_{n=0}^{\infty} \epsilon_{n+1}\), then for any angle \(\theta\) \(\in \{ -\infty < \epsilon_n \ \text{e}, + \sum_{n=0}^{\infty} \epsilon_n \}\), the sequences \((\theta_n)\) and \((a_n)\) defined as

\[
\begin{align*}
\theta_0 &= 0 \\
ad_n &= \begin{cases} 
1 & \text{if } \theta_n \leq \theta \\
-1 & \text{if } \theta_n > \theta
\end{cases} \\
\theta_{n+1} &= \theta_n + d_n e_n
\end{align*}
\]

satisfy \(\lim_{n \to +\infty} \theta_n = \theta\).

The sequence \((\epsilon_n)\) is called a discrete basis, and the previous algorithm which gives \((\theta_n)\) and \((a_n)\) is called the bidirectional algorithm [13].

Now, let us assume that we want to perform a rotation of angle \(\theta\), i.e., to compute, from an initial two-dimensional vector \((x_0, y_0)\), a vector \((x_n, y_n)^t\) defined as

\[
\begin{align*}
(x_n) &= \left( \cos \theta \quad - \sin \theta \right) (x_0) \\
y_n &= \left( \sin \theta \quad \cos \theta \right) (y_0)
\end{align*}
\]

The basic idea of Cordic is to perform this rotation as a sequence of elementary rotations. Using the bidirectional algorithm and the

\[
\begin{align*}
\theta_0 &= 0, \quad x_0 = 1, \quad y_0 = 0 \\
d_n &= \begin{cases} 
1 & \text{if } \theta_n \leq \theta \\
-1 & \text{if } \theta_n > \theta
\end{cases} \\
\theta_{n+1} &= \theta_n + d_n e_n \\
\theta_0 &= 0, \quad x_0 = 1, \quad y_0 = 0 \\
d_n &= \begin{cases} 
1 & \text{if } \theta_n \leq \theta \\
-1 & \text{if } \theta_n > \theta
\end{cases} \\
\theta_{n+1} &= \theta_n + d_n e_n
\end{align*}
\]
Double Cordic iterations have
and
would give algorithms for
where
and the
provided we replace
obtained in step 
(vecmring mode)
if
sign 
In a very similar way, the algorithm
goes to 
now, a multiplication by this term reduces to an 

Assume 
by the terms
if

The algorithm for computing 
and 
therefore, we can compute 
the factor of the similarity becomes 
and 
and 

Obviously, the main drawback of these algorithms is the evaluation of 
tn. The iterative relation 

cannot be used since it involves a “true” multiplication. In order to avoid this drawback, we shall perform “double” Cordic iterations: we shall use the bidirectional algorithm with the discrete basis 

instead of the discrete basis 

. Therefore, at step 
the algorithm, we shall perform two similarities of angle 
Double Cordic iterations have already been used by Takagi, Asada, and Yajima [14], and by Delosme [6], [7] in quite a different context: the purpose of Takagi, Asada, and Yajima was to keep a constant scaling factor 
when quickly performing Cordic iterations using a redundant number system. The purpose of Delosme was to obtain simpler scaling factors. The main advantage of doubling iterations is that in step 
the factor of the similarity becomes 
now, a multiplication by this term reduces to an addition and a shift. Another advantage is that the convergence domain of the algorithm becomes larger: it gives a correct result for 

therefore, we can compute 
cos-1 
t and 
sin-1 t for any 
and 
The algorithm for computing 
cos-1 
t becomes

it gives: 
In a very similar way, the algorithm

gives 
In a very similar way, the algorithm

gives 

At step 
the algorithm 
cos-1 t, 
xn is equal to 
where 
has the same value as previously. Therefore, since 
goes to 
we
The exact value is 0.643501108793, 0.0862608838504, as follows:

\[
y_n \rightarrow K^2 \sqrt{1-t^2}.
\]

Thus, \( \cos^{-1} \beta \) may be avoided by performing another sequence of rotations in parallel with that of \( \cos^{-1} \alpha \), as follows:

\[
\begin{align*}
\theta_n &= 0, \ x_0 = 1, \ x_0' = 1/K^2, \ y_0 = 0, \ y_0' = 0, \ t_0 = t \\
d_n &= 1 \text{ if } x_n \geq t_n \text{ else } -1 \\
(x_{n+1}', y_{n+1}') &= \begin{pmatrix}
1 & -d_n 2^{-n} \\
0 & 1
\end{pmatrix}
\begin{pmatrix}
x_n \\
y_n
\end{pmatrix}

\text{then } x_n \text{ converges to } \sqrt{1+t^2}. \text{ As a matter of fact, this iteration may be viewed as a slight modification (change of initial values) of a classical square root iteration (see [12] for instance).}

IV. CONCLUSION

We have given here an extension of the Cordic algorithm which makes it possible to compute the functions \( \cos^{-1}, \sin^{-1}, \sqrt{1-t^2}, \sinh^{-1}, \cosh^{-1} \), and \( \sqrt{1+t^2} \). Our algorithms are suitable for VLSI implementation, and require only a slight modification of the original Cordic algorithm.
Interrupt Handling for Out-of-Order Execution Processors

H. C. Torng and Martin Day

Abstract—Processors with multiple functional units, including the superscalars, achieve significant performance enhancement through low-level execution concurrency. In such processors, multiple instructions are often issued and definitely executed concurrently and out-of-order. Consequently, interrupt and exception handling becomes a vexing problem. We identify factors that must be considered in evaluating the effectiveness of interrupt and exception handling schemes: latency, cost, and performance degradation. We then briefly enumerate proposals and implementations for interrupt and exception handling on out-of-order execution processors.

Next, we present an efficient hardware mechanism, the Instruction Window (IW), and a new approach, which allows for precise, responsive, and flexible interrupt and exception handling. The implementation of the IW is then discussed. The design of an S-cell IW has been carried out; it can work with a very short machine cycle and flexible interrupt and exception handling.

Finally, we present a comparison of all interrupt and exception handling schemes for out-of-order execution processors.

Index Terms—Interrupt handling, interrupt latency, low-level Concurrency, modified precise interrupt, out-of-order execution.

I. INTRODUCTION

Processors with multiple functional units issue and execute multiple instructions concurrently and possibly out-of-order; they enhance performance by extracting low-level concurrency from the instruction stream [1]–[3]. The CDC 6600, IBM 360/91, and the CRAY machines are forerunners of this class of processors; however, these processors issue at most one instruction per cycle. Due to advances in device technologies, recently announced RISC processors often issue and certainly execute multiple instructions concurrently. However, these processors have not been able to support interrupt and exception handling efficiently and with an acceptable latency.

In this paper, we address the interrupt handling problem, which has hampered the development of processors which execute and may even issue multiple instructions. We propose an efficient hardware mechanism, which supports an interrupt handling scheme with a flexible latency, set specifically for each type of interrupts requested.

The remaining sections are organized as follows: Section II presents a discussion of interrupts and exceptions. Factors for evaluating the effectiveness of interrupt handling schemes are presented. Existing proposals and implementations for interrupt handling on out-of-order execution processors are briefly reported in Section III.

Section IV presents the Instruction Window (IW), a simple and yet versatile hardware mechanism which supports efficient and flexible interrupt handling. Basic window operations are introduced in Section V. Section VI proposes an innovative interrupt handling scheme, which makes use of the IW. In Section VII, we discuss the implementation of the IW. Section VIII gives an evaluation of all interrupt handling schemes.

II. INTERRUPTS AND EXCEPTIONS

An important and indispensable feature of any processor is its ability to handle properly interrupts and exceptions. An I/O device, a sensor, or a timer may “interrupt” a processor to perform a specific task. An executing instruction may cause a page fault or an overflow/underflow; an “exception” thus results. Finally, one may place an instruction in an instruction stream to call for a “trap,” which initiates a pre-planned action. Presentations on interrupts, exceptions, and traps can be found from many sources, among them [4]–[8].

In this paper, we use the term interrupt to denote an interrupt, an exception or a trap. Our study does not treat the subject of interrupt detection; rather, we investigate how a processor responds to an interrupt request, once it has been received.

When an interrupt request is received, the processor must save its processor state, then load and execute an appropriate interrupt handler. Upon completion of the interrupt handling routine, the saved processor state is restored, and the interrupted process can then be restarted.

A processor state should contain enough and preferably only enough information so that the interrupted process can be restarted at the precise point where it was interrupted. To be able to resume an interrupted process, the processor state should consist of the contents of the general purpose registers, the program counter, the condition register, all index registers, and the relevant portion of the main memory.

The classical approach to identifying precisely the point where a process is interrupted is to save, among other vital items, the address of a specific instruction, say instruction $I_n$; when the processor state is saved, all instructions that precede instruction $I_n$ are saved. All instructions that precede instruction $I_n$ have been executed. Instruction $I_n$ and those that follow it have not. Instruction $I_n$ thus provides a precise interrupt point.

For processors, which execute instructions concurrently and possibly out-of-order, the identification of a precise interrupt point when an interrupt request is made may become very costly.

‘From now on, we will simply use interrupt to stand for interrupt and exception.'