The Differential CORDIC Algorithm: Constant Scale Factor Redundant Implementation without Correcting Iterations

Herbert Dawid, *Student Member, IEEE*, and Heinrich Meyr, *Fellow, IEEE*

Abstract—The CORDIC algorithm is a well-known iterative method for the efficient computation of vector rotations, and trigonometric and hyperbolic functions. Basically, CORDIC performs a vector rotation which is not a perfect rotation, since the vector is also scaled by a constant factor. This scaling has to be compensated for following the CORDIC iteration.

Since CORDIC implementations using conventional number systems are relatively slow, current research has focused on solutions employing redundant number systems which make a much faster implementation possible. The problem with these methods is that either the scale factor becomes variable, making additional operations necessary to compensate for the scaling, or additional iterations are necessary compared to the original algorithm.

In contrast we developed transformations of the usual CORDIC algorithm which result in a constant scale factor redundant implementation without additional operations. The resulting "Differential CORDIC Algorithm" (DCORDIC) makes use of on–line (most significant digit first redundant) computation. We derive parallel architectures for the radix–2 redundant number systems and present some implementation results based on logic synthesis of VHDL descriptions produced by a DCORDIC VHDL generator. We finally prove that, due to the lack of additional operations, DCORDIC compares favorably with the previously known redundant methods in terms of latency and computational complexity.

Index Terms—CORDIC, carry save, signed digit, redundant number systems, radix-2, VLSI architecture, computer arithmetic.

1 Introduction

Applications of modern digital signal processing systems exhibit an increasing need for the efficient implementation of complex arithmetic operations. In adaptive signal processing many algorithms require the solution of systems of linear equations, or the computation of eigenvalues, eigenvectors or singular values. Additionally, algorithms used in communication technology require the computation of trigonometric functions, coordinate transformations, vector rotations, or hyperbolic rotations. The CORDIC algorithm offers the opportunity to calculate the desired functions in a rather simple and elegant way.

The CORDIC algorithm was first introduced by Volder [5] for the computation of trigonometric functions, multiplication, division and datatype conversion, and later on generalized to hyperbolic functions by Walther [6]. Two basic CORDIC modes are known leading to the computation of different functions, the rotation mode and the vectoring mode.

For both modes the algorithm can be realized as an iterative sequence of additions/subtractions and shift operations which are rotations by a fixed rotation angle (microrotations), but with variable rotation direction. Due to the simplicity of the involved operations, the CORDIC is very well suited for VLSI realization ([7], [8], [9], [10], [11], [12], [13]). However, the CORDIC iteration is not a perfect rotation, which would involve multiplications with sine and cosine. The rotated vector is also scaled, which has to be corrected by a final multiplication with a precomputed constant.

The computation time and the achievable throughput of CORDIC processors using conventional arithmetic are determined by the carry propagation involved with the additions/subtractions, since the direction of the CORDIC microrotation is steered by the sign of the previous iteration results. This sign is not known prior to the computation of the MSB (most significant bit). The use of redundant arithmetic is well known to speed up additions/subtractions, because carry-free or limited carry-propagation operation becomes possible. However, the application of redundant arithmetic for the CORDIC is not straightforward, because a complete word level carry–propagation is still required in order to determine the sign of a redundant number (this also holds for generalized signed digit number systems as described in [23]).

In order to overcome this problem, several authors proposed techniques for estimating the sign of the redundant intermediate results from a number of MSDs (most significant digits) ([17], [21], [24], [22]).

If the sign and therefore the rotation direction cannot be estimated reliably from the MSDs, no microrotation is performed at all. However, the scaling factor involved with the CORDIC algorithm depends on the actually performed rotations. Therefore here, the scaling factor is variable, and

• *The authors are with the Institute for Integrated Systems in Signal Processing, Aachen University of Technology (RWTH), IEE 611810, D-52056, Aachen, Germany. E-mail: dawid@ert/rwth-aachen.de.*

Manuscript received Jan. 25, 1994; revised Feb. 3, 1995.
For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number C96006.

has to be calculated in parallel to the usual CORDIC iteration. Additionally, a division by the variable scaling factor has to be implemented following the CORDIC iteration.

A number of recent publications dealing with constant scale factor redundant (CFR) CORDIC implementations of the rotation mode ([14], [15], [16], [17], [18], [19], [20], [21], [22]) describe sign estimation techniques, where every iteration is really performed, in order to overcome this problem. This method was extended to the vectoring mode in [25]. However, either a considerable increase (about 50 percent) in the complexity of the iterations (*double rotation method* [16]) or a 50 percent increase in the number of iterations (*correcting iteration method* [16], [19], [15], [14]) occurs.

In [26], a different CFR algorithm is proposed for the rotation mode. Using this "branching CORDIC," two iterations are performed in parallel if the sign can not be estimated reliably, each assuming one of the possible choices for rotation direction. It is shown in [26] that at most two parallel branches can occur. However, this is equivalent to an almost twofold effort in terms of implementation complexity of the CORDIC rotation engine.

In contrast to the abovementioned approaches we develop transformations of the usual CORDIC iteration resulting in a constant scale factor redundant implementation without additional or branching iterations. Additionally, there is only a very moderate increase in the complexity of the iterations. It will be shown that our "Differential CORDIC (DCORDIC)" method compares favorably to the sign estimation methods.

In this paper, we consider radix–2 redundant number systems which are very attractive for VLSI realization. We derive parallel architectures for the rotation as well as the vectoring mode. Finally we discuss latency, complexity and throughput of these architectures in comparison to the known redundant methods and give some implementation examples.

2 Review of the CORDIC Algorithm

In this section, we briefly describe the conventional CORDIC algorithm for rotation and vectoring mode. For a detailed discussion the reader is referred to Walther [6] and Volder [5]. The CORDIC algorithm performs the rotation of a given vector (x_0, y_0) with magnitude M and phase P by means of a sequence of rotations with fixed angles $\alpha_i = \arctan(2^{-i})$, but variable rotation direction, which are called microrotations. The rotated intermediate vector is represented by the iteration components x_i and y_i. The third iteration component p_i keeps track of the accumulated overall rotation angle. The rotated vector is scaled by $\frac{1}{|\cos \alpha_i|}$ during a microrotation.

By suitable control of the direction of successive microrotations, one iteration component is forced to zero and a number of different arithmetic operations can be computed.

Given N bits wordlength of the input vector (x_0, y_0) and the input phase p_0, then $N + 1$ iterations lead to results of sufficient accuracy [6]. It is important to note that the output vector (x_{N+1}, y_{N+1}) is scaled by a constant factor $K = \prod_{i=0}^{N} \frac{1}{|\cos(\alpha_i)|}$ due to the scaling associated with the $N + 1$ microrotations. It is therefore necessary to introduce a multiplication by $\frac{1}{K}$ as a final scaling operation.

2.1 Rotation Mode

In rotation mode, the rotation of an input vector (x_0, y_0) by a given angle p_0 is calculated. This is achieved by initializing the p component with p_0, and then forcing the p component iteratively to zero. Hence the rotation directions are steered by the p component sign ($i \in (0, \ldots, N)$):

$$\begin{aligned} x_{i+1} &= x_i - \text{sign}(p_i) \cdot y_i \cdot 2^{-i} \\ y_{i+1} &= y_i + \text{sign}(p_i) \cdot x_i \cdot 2^{-i} \\ p_{i+1} &= p_i - \text{sign}(p_i) \cdot \alpha_i \end{aligned} \tag{1}$$

with

$$\text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

The finally computed scaled rotated vector is given by (x_{N+1}, y_{N+1}).

2.2 Vectoring Mode

In vectoring mode, magnitude M and phase P of the input vector are calculated (note that $p_0 = 0$ holds here). The y component is forced to zero, i.e., the input vector (x_0, y_0) is rotated towards the x–axis. Then, x_{N+1} contains the scaled and eventually signed magnitude M, and the overall accumulated rotation angle p_{N+1} is a measure for the phase P. Hence, the CORDIC iteration for the vectoring mode is controlled by the sign of the x- and the y-component ($i \in (0, \ldots, N)$).

$$\begin{aligned} x_{i+1} &= x_i + \text{sign}(y_i) \cdot \text{sign}(x_i) \cdot y_i \cdot 2^{-i} \\ y_{i+1} &= y_i - \text{sign}(y_i) \cdot \text{sign}(x_i) \cdot x_i \cdot 2^{-i} \\ p_{i+1} &= p_i + \text{sign}(y_i) \cdot \text{sign}(x_i) \cdot \alpha_i \end{aligned} \tag{2}$$

If $x_0 \geq 0$, then $M = \frac{1}{K} \cdot x_{N+1}$ and $P = p_{N+1}$ hold. If $x_0 < 0$, then $M = -\frac{1}{K} \cdot x_{N+1}$ and $P = \pi + p_{N+1}$ hold.

3 The Differential CORDIC Algorithm

In this section we derive transformations of the original CORDIC algorithm resulting in a different formulation, which is shown to be equivalent to the usual CORDIC in terms of accuracy and convergence. However, the resulting differential CORDIC (DCORDIC) algorithms for the rotation and the vectoring mode exhibit a structure which can be exploited to derive very fast and efficient redundant implementations as presented in the following sections.

Since the transformed algorithms work on new temporary variables, given by absolute values rather than the usual iteration variables, we introduce the following terminology using a generic variable g, which can be either of the CORDIC variables x, y, or p:

$$\hat{g}_{i+1} = \text{sign}(g_i) \cdot g_{i+1} \tag{3}$$

with the following obvious properties:

$$|\hat{g}_{i+1}| = |g_{i+1}| \tag{4}$$

and

$$\text{sign}(\hat{g}_{i+1}) = \text{sign}(g_i) \cdot \text{sign}(g_{i+1}) \tag{5}$$

Using the fact that $\frac{1}{\text{sign}(g)} = \text{sign}(g)$ we conclude:

$$\text{sign}(g_{i+1}) = \text{sign}(g_i) \cdot \text{sign}(\hat{g}_{i+1}) \tag{6}$$

This means, the sign of the original iteration variables can be derived recursively from the sign of the corresponding new variables in a simple way. In fact, the signs of $\hat{g}_i$ can be considered to be the differentially encoded signs of g_i, which are differentially decoded using (6). For this reason we call the novel CORDIC algorithm which will be derived below the "Differential CORDIC Algorithm (DCORDIC)."

3.1 DCORDIC Rotation Mode

In CORDIC rotation mode the p component is forced to zero, which steers the x and y iterations. The equation $p_{i+1} = p_i - \text{sign}(p_i) \cdot \alpha_i$ (see (1)) can be interpreted in the following way: From the two alternatives for rotation direction, the one which leads to a smaller *absolute value* of the updated p–component is always chosen. Hence, the idea is to transform the conventional iteration into an iteration involving only the absolute value of the new $\hat{p}_i$ variable.

THEOREM 1 (Rotation Mode Recurrence for $|\hat{p}_i|$): *For the absolute value of the $\hat{p}_i$ variable*

$$|\hat{p}_{i+1}| = \left| |\hat{p}_i| - \alpha_i \right|$$

holds for $i \in (0, \ldots, N)$.

Hence, an iteration involving only subtractions and absolute value computations is achieved.

PROOF. The original recurrence

$$p_{i+1} = p_i - \text{sign}(p_i) \cdot \alpha_i$$

can be transformed into

$$\begin{aligned} \text{sign}(p_i) \cdot p_{i+1} &= \text{sign}(p_i) \cdot p_i - \alpha_i \\ \hat{p}_{i+1} &= |p_i| - \alpha_i \text{ using (3)} \\ &= |\hat{p}_i| - \alpha_i \text{ using (4)} \end{aligned}$$

Taking absolute values $|\hat{p}_{i+1}| = |p_{i+1}| = \left| |\hat{p}_i| - \alpha_i \right|$ using (4) holds. □

It is apparent that for x_i and y_i the actual sign of p_i is still needed, while only the absolute value $|\hat{p}_i|$ appears in the transformed iteration. We already proved in (6) that $\text{sign}(p_{i+1}) = \text{sign}(p_i) \cdot \text{sign}(\hat{p}_{i+1})$. Hence, the actual sign of p_{i+1} can recursively be calculated from the given initial sign of p_0 and the sign of $\hat{p}_{i+1}$. It will be shown in Section 4.5 that this can be interpreted as a kind of "differential decoding" involved naturally with DCORDIC.

Hence, the following algorithm results for the DCORDIC rotation mode:

Algorithm [DCORDIC rotation mode]
[Input]
x_0, y_0: components of the input vector
p_0: rotation angle
[Output]
x_{N+1}, y_{N+1}: scaled rotated vector
[Algorithm]
$\hat{p}_0 = |p_0| \Rightarrow |\hat{p}_0| = \hat{p}_0$
for $i = 0$ to N do
begin
steering variable: $|\hat{p}_{i+1}| = \left| |\hat{p}_i| - \alpha_i \right|$
sign calculation: $\text{sign}(p_{i+1}) = \text{sign}(p_i) \cdot \text{sign}(\hat{p}_{i+1})$
dependent variables: $x_{i+1} = x_i - \text{sign}(p_i) \cdot y_i \cdot 2^{-i}$
$y_{i+1} = y_i + \text{sign}(p_i) \cdot x_i \cdot 2^{-i}$
end

3.2 DCORDIC Vectoring Mode

In CORDIC vectoring mode the y–component is forced to zero. From the two alternatives for rotation direction the one which leads to a smaller *absolute value* of the updated y–component is always chosen. Again, the idea is to derive an iteration directly operating on absolute values of the new $\hat{y}_i$ variable.

THEOREM 2 (Vectoring Mode Recurrence for $|\hat{y}_i|$): *For the absolute value of the $\hat{y}_i$ variable*

$$|\hat{y}_{i+1}| = \left| |\hat{y}_i| - |x_i| \cdot 2^{-i} \right|$$

holds for $i \in (0, \ldots, N)$.

PROOF. The recurrence

$$y_{i+1} = y_i - \text{sign}(y_i) \cdot \text{sign}(x_i) \cdot x_i \cdot 2^{-i}$$

transforms into

$$\begin{aligned} \text{sign}(y_i) \cdot y_{i+1} &= \text{sign}(y_i) \cdot y_i - \text{sign}(x_i) \cdot x_i \cdot 2^{-i} \\ \hat{y}_{i+1} &= |y_i| - |x_i| \cdot 2^{-i} \text{ using (3)} \\ &= |\hat{y}_i| - |x_i| \cdot 2^{-i} \text{ using (4)} \end{aligned}$$

Hence, $|\hat{y}_{i+1}| = |y_{i+1}| = \left| |\hat{y}_i| - |x_i| \cdot 2^{-i} \right|$ using (4) holds. □

The sign of y_i can be calculated as

$$\text{sign}(y_{i+1}) = \text{sign}(y_i) \cdot \text{sign}(\hat{y}_{i+1})$$

as was proven in (6).

For the x component, we can achieve a simplified iteration as well. Recall the original recurrence (2)

$$x_{i+1} = x_i + \text{sign}(y_i) \cdot \text{sign}(x_i) \cdot y_i \cdot 2^{-i}$$

and therefore

$$\begin{aligned} \text{sign}(x_i) \cdot x_{i+1} &= \text{sign}(x_i) \cdot x_i + \text{sign}(y_i) \cdot y_i \cdot 2^{-i} \\ \hat{x}_{i+1} &= |x_i| + |y_i| \cdot 2^{-i} \text{ using (3)} \\ &= |\hat{x}_i| + |\hat{y}_i| \cdot 2^{-i} \text{ using (4)} \end{aligned}$$

holds.

The right hand side of this equation is always positive, indicating that:

$$\hat{x}_{i+1} = \begin{cases} +x_{i+1} & \text{if } x_0 \geq 0 \quad i \in (0, \ldots, N) \\ -x_{i+1} & \text{if } x_0 < 0 \quad i \in (0, \ldots, N) \end{cases}$$

which can simply be proved by using (6). Hence, we can always use $\text{sign}(x_{i+1}) = \text{sign}(x_0)$ for the p component itera-

tion. However, if $x_0 < 0$, a still simpler method is to rotate the initial vector (x_0, y_0) by the angle π (simply negating the x and y component) and adding or subtracting π from the CORDIC rotation angle. Then $x_0 \geq 0$ always holds, and therefore $x_i \geq 0$ also holds. Hence, $x_i = \hat{x}_i$ holds and $|\hat{x}_i|$ can be replaced by $\hat{x}_i$ in all preceding equations. Furthermore, $\text{sign}(x_i) = 1$ can be fixed for the p recurrence.

The following algorithm results for the DCORDIC vectoring mode for $x_0 > 0$:

Algorithm [DCORDIC vectoring mode]
[Input]
x_0, y_0: components of the input vector
$p_0 = 0$: input phase
[Output]
$\hat{x}_{N+1}$: scaled magnitude of the input vector
p_{N+1}: phase of the input vector
[Algorithm]

$$\hat{x}_0 = x_0$$
$$\hat{y}_0 = |y_0| \Rightarrow |\hat{y}_0| = \hat{y}_0$$
for $i = 0$ to N do
begin
steering variable: $|\hat{y}_{i+1}| = \left| |\hat{y}_i| - \hat{x}_i \cdot 2^{-i} \right|$
sign calculation: $\text{sign}(y_{i+1}) = \text{sign}(\hat{y}_{i+1}) \cdot \text{sign}(y_i)$
dependent variables: $\hat{x}_{i+1} = \hat{x}_i + |\hat{y}_i| \cdot 2^{-i}$
$p_{i+1} = p_i + \text{sign}(y_i) \cdot \alpha_i$
end

4 DCORDIC Implementation

Below, we will consider only the well known binary signed digit (BSD) and carry save (CS) radix–2 redundant number systems (see, e.g., [27]) since they are very well suited for VLSI implementation. For reasons which will become clear below, it is advantageous to employ BSD numbers rather than CS numbers here. We additionally use the well known (p, n) coding for the BSD numbers which was proven to be advantageous for implementation in [28].[1]

Since absolute value calculations are required for DCORDIC implementation, we first state an absolute value algorithm for BSD numbers. We then summarize some important properties of the DCORDIC algorithms. Finally, we derive the necessary pseudo overflow correction for BSD numbers and present architectures for the rotation and the vectoring mode. Here we make use of the well known concept of generalized full adders (GFAs, see, e.g., [29]) in order to jointly describe the architectures for the CS as well as the BSD case.

The first striking feature of the DCORDIC algorithms is the additional absolute value calculation: Generally, a digit parallel absolute value calculation (for redundant as well as nonredundant number systems) can be implemented by a sign calculation followed by a steered negation. However, due to the delay of the sign calculation which is the main problem for a redundant implementation of the original algorithm, nothing would be gained. The situation looks different if we employ on–line computation [30].[2]

For BSD numbers, absolute value calculations can be implemented in an on–line manner (i.e., in most significant digit (MSD) first mode) with an on–line delay of $\delta = 0$.[3]

4.1 BSD Absolute Value Computation

In this section we will state the known on–line algorithm for calculating the absolute value Y of a BSD number X with an on-line delay of $\delta = 0$. The algorithm is based on the simple fact that the sign of a BSD number is equal to the sign of the most significant nonzero digit. We also generate the sign of the input number as an additional output.

Algorithm [BSD Absolute Value]
F: an enumeration type with elements
(positive, negative, nodec (no decision))
[Input]
X: an arbitrarily coded BSD number
with digits $x_i \quad i \in (0, \ldots, N-1)$
(x_0 is MSD)
[Output]
Y: a positive BSD number with $Y = |X|$
and digits $y_i \quad i \in (0, \ldots, N-1)$
S: sign(X)
[Algorithm]

Step 1: $F_{-1} = \text{nodec}$
Step 2: for $j = 0$ to $N-1$ do
begin

$$y_j = \begin{cases} x_j & \text{if } F_j = \text{positive} \\ -x_j & \text{if } F_{j-1} = \text{negative} \\ x_j & \text{if } F_{j-1} = \text{nodec and } x_j \geq 0 \\ -x_j & \text{if } F_{j-1} = \text{nodec and } x_j = -1 \end{cases}$$

$$F_j = \begin{cases} F_{j-1} & \text{if } F_{j-1} = \text{positive or negative} \\ \text{positive} & \text{if } F_{j-1} = \text{nodec and } x_j = +1 \\ \text{negative} & \text{if } F_{j-1} = \text{nodec and } x_j = -1 \\ \text{nodec} & \text{if } F_{j-1} = \text{nodec and } x_j = 0 \end{cases}$$

end

$$\text{Step 3 } S \leftarrow \begin{cases} +1 & \text{if } F_{N-1} = \text{positive or nodec} \\ -1 & \text{if } F_{N-1} = \text{negative} \end{cases}$$

Starting with the MSD, the digits are inspected and as soon as a nonzero digit has been found a decision is met. For implementation, we have to take into account two flag bits for passing the three valued enumeration type F_j from higher to lower digits. It is obvious that the on-line delay for this operation is $\delta = 0$, since the output digits can be calculated immediately from the incoming digits and the input flag information. The necessary steered digit negation can be implemented for (p, n) coded numbers involving mainly a single EXOR gate delay. Together with the simple

1. Note that for the (p, n) coding, the value of a digit is given by $p - n$ with bits p and n. Therefore, negation can, e.g., simply be implemented by exchanging p and n.

2. The essence of on–line arithmetic is the use of redundant number systems together with most significant digit first operation.

3. In On–line computation, the on–line delay is defined to be δ if $(j + \delta)$ digits of the operands are required in order to generate the jth digit of the result.

combinational logic realizing the flag generation the combinational logic effort necessary to implement this operation is comparable to a single full adder cell, while the propagation delay is about half that of a full adder.

For the CS number system, the derivation of a MSD first absolute value computation can be found in Appendix A. It should be noted, however, that the main problem of CS absolute value computation is the necessity to consider a correction term of +2 for negation, since two twos complement negations take place. This correction has to be deferred and taken into account during the following operations, which leads to more complicated algorithms and architectures [4]. Therefore, although there are only slight general structural differences between CS and BSD arithmetic, BSD arithmetic is preferred throughout the rest of this paper.

4.2 Pseudo Overflow Correction and Computational Accuracy

It is well known that, using redundant number systems, pseudo overflows can occur. This means that even if the value of an addition result fits into the chosen wordlength, the redundant sum can overflow. In [14], an efficient pseudo overflow correction was presented for the CS number system. It should be noted that this overflow treatment is of great importance for the efficiency of the overall system, because an eventually increased critical path due to an overflow correction limits the speed of the whole application. We will here briefly describe a digit–local pseudo overflow correction for (p, n) coded BSD numbers, i.e., a correction which is implemented taking into account only the values of the MSD transfer digits and the MSD itself. This can only be achieved by limiting the actually usable range of a W digit number to $(-2^{W-1}; 2^{W-1})$. It is easy to show that otherwise, the pseudo overflow correction involves all digits and becomes relatively complicated.

In the following discussion we assume that the actual range of the addition result is consistent with the usable digit range. We add the two W digit BSD numbers $S = X + Y$ with $S = \sum_{i=0}^{W-1} s_i \cdot 2^i$ and $s_i \in (-1, 0, +1)$. Each of the numbers and the sum satisfy the range limitation imposed above. During the 4–2 addition, a positive MSD transfer digit T_p and a negative MSD transfer digit T_n occur which exceed the wordlength W. If $T_p - T_n = 0$, then obviously no pseudo overflow occurs. We hence have to consider the two remaining cases:

Case 1: If $T_p - T_n = 1$, a positive pseudo overflow occurs with value 2^W. Since the sum is known to fit into $(-2^{W-1}; 2^{W-1})$, the value r represented by the remaining digits has to fulfill

$$\begin{aligned} 2^W + r &\le 2^{W-1} \\ r &\le -2^{W-1} \end{aligned} \tag{7}$$

We will now draw a conclusion from this fact in terms of the value of the digit s_{W-1}, which is the addition result without overflow correction. Assume that $s_{W-1} \ge 0$. Then $r \ge \sum_{u=0}^{W-2} -2^i = -2^{W-1} + 1$ holds. Hence, even if all digits below s_{W-1} are equal to -1, the condition from (7) cannot be met if $s_{W-1} \ge 0$. It can obviously only be met if $s_{W-1} = -1$ holds. Therefore, the pseudo overflow can be corrected locally by adding the overflow value 2^W to s_{W-1}, hence setting $s_{W-1} = +1$.

Case 2: If $T_p - T_n = -1$, a negative pseudo overflow occurs with value -2^W. Since the sum is known to fit into $(-2^{W-1}; 2^{W-1})$, the value r given by the remaining digits has to fulfill:

$$\begin{aligned} -2^W + r &\ge -2^{W-1} \\ r &\ge 2^{W-1} \end{aligned} \tag{8}$$

With a proof very similar to the one given above, it can be shown that the condition from (8) can only be met if $s_{W-1} = +1$ holds. Therefore, the pseudo overflow can locally be corrected by adding the overflow value -2^W to s_{W-1}, hence setting $s_{W-1} = -1$.

For implementation, we can modify the most significant cell to:

If $x_{W-1} + y_{W-1} = \pm 2$ we set $s_{W-1} = \pm 1$ and do not generate a MSD transfer digit, because in order to satisfy the imposed range limitation, a positive/negative transfer has to come. If $x_{W-1} + y_{W-1} = \pm 1$ then we also do not generate a transfer out since if there is a transfer in it is ∓ 1. Finally, for $x_{W-1} + y_{W-1} = 0$ of course no transfer occurs. The implementation of this cell should not be more complex than the two GFAs usually necessary for 4–2 addition.

Another topic which is very important for realization is the achieved computational accuracy. For every CORDIC realization, several effects (e.g., quantization of the rotation angles α_i) limit computational accuracy. A detailed discussion of this topic is outside the scope of this paper, the reader is referred to the in depth treatment given in [32]. However, some aspects shall be briefly discussed. One important error source is the rounding or truncation error due to the shifts for the x and y variables. An application without rounding or truncation would require $N_{xy} = 2 \cdot N$ due to the maximum shift 2^{-N} (see (1) and (2)). It has been shown by several authors, that, if rounding to the nearest is employed, a number of $\log_2 N$ guard digits at the LSD side are sufficient to achieve an accuracy of ±LSD (e.g., [7]). It should be noted that, using radix–2 number systems, an additional guard digit is necessary due to the twofold maximum rounding error of the redundant representation. The internal wordlength has to be further enlarged at the MSD side by two digits [7], hence $N_{xy} = 2 + N + \log_2 N$ holds. Note that for the p wordlength, $N_p = N$ is usually sufficient.

4.3 Sign Calculation

It is obvious from the BSD absolute value algorithm, that the sign of the operand is still known only after processing the least significant digit (LSD), indicating a large latency. However, when implementing the DCORDIC algorithms, this latency occurs only once, not for every iteration. In order to explain this, it is easier to consider an already pipelined implementation as shown in Fig. 1. The left hand side shows a 3–2 addition and absolute value calculation as necessary for the $\hat{p}_i$ variable in DCORDIC rotation mode. Note

that the circuit is directly fed with the precomputed value $-\alpha_i$. The right hand side shows a 4–2 subtraction and absolute value calculation for $\hat{y}_i$ in DCORDIC vectoring mode. Note that since, as mentioned before, negation is simply implemented by exchanging the p and n bit, the negation of x_i is performed by wiring and GFAs are sufficient for implementing a fixed subtraction.

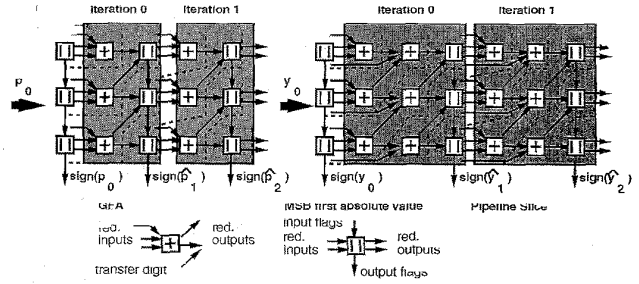


Fig. 1. Left-hand side: $\left|\hat{p}_{i+1}\right| = \left|\left|\hat{p}_i\right| + (-\alpha_i)\right|$ in rotation mode; right-hand side: $\left|\hat{y}_{i+1}\right| = \left|\left|\hat{y}_i\right| + (-\hat{x}_i)\cdot 2^{-i}\right|$ in vectoring mode.

For discussion below, we introduce the following terminology:

- **Iteration Delay I:** Delay or latency (in clock cycles) for one iteration. This delay is equal to the delay between the generation of the signs calculated during absolute value computation.
- **Sign Delay S:** Initial delay in clock cycles for sign calculation during the first iteration.

For reasons to be explained in Section 4.2, we also introduce the wordlength N_{xy} for the x and y variables and the N_p for the p variables, instead of the general wordlength N used before.

4.3.1 Rotation Mode: $\hat{p}_i$

For the pipelining as shown in Fig. 1, it is easy to see that the iteration delay for $\hat{p}_i$ is $I_p = 2$, since two cycles are necessary for every iteration (the generated output signs are separated by two pipeline stages).

The sign delay, however, is given by $N_p + 1$ cycles, if N_p is the involved wordlength, hence $S_p = N_p + 1$. As is clear from Fig. 1, the sign delay occurs only once in the beginning. Note that we have combinatorial paths (see Fig. 1) containing the GFA carry plus absolute value calculation, as well as the GFA sum path. Since the GFA sum path is usually much slower than the carry path, this path is assumed to be the critical path. Therefore the cycle time is assumed to be equivalent to the cycle time for a single pipelined GFA.

4.3.2 Vectoring Mode: $\hat{y}_i$

The situation is similar as for the rotation mode, besides the fact that here a 4–2 addition occurs instead of a 3–2 addition. Due to the MSD first computation, the iteration delay is now obviously $I_y = 3$ (see Fig. 1).

The sign delay is given by $N_{xy} + 2$ cycles here: $S_y = N_{xy} + 2$. As becomes clear from Fig. 1, the sign delay occurs again only once in the beginning. Since only the fast carry paths are involved in the critical path (see Fig. 1), we estimate the cycle time to be again equivalent to the cycle time for a single pipelined GFA.

4.4 Total Delay

In the last section we discussed the latency involved with the absolute value calculation for the iterations involving the steering DCORDIC variables. In order to determine the total latency, we also have to take into account the steered 4–2 additions/subtractions for x_i and y_i in rotation mode as well as the 4–2 additions for x_i and the steered 3–2 additions for p_i in vectoring mode. For the rotation mode, the iteration delay for the $\hat{p}_i$ variable was determined above to be $I_p = 2$ corresponding to two GFA delays. However, the delay for x_i and y_i is obviously given by two GFA delays for the 4–2 additions and the delay of a steered negation, which is assumed to be 0.5 GFA delays. Hence, here, the resulting total iteration delay I_r for the rotation mode is 2.5 GFA delays.

For the vectoring mode, the 4–2 additions for x_i (2 GFA delays) and the steered 3–2 additions for p_i (1.5 GFA delays) can be implemented within the three GFA delays necessary for the $\hat{y}_i$ variable ($I_y = 3$). Hence, here, the total vectoring mode iteration delay is $I_v = I_y = 3$.

Please recall that, as was proven during the DCORDIC derivation, the sign calculation involved with DCORDIC is exactly the same as for the conventional CORDIC. Hence, no additional or correcting iterations have to take place. For the total delay, we hence have to take into account only the usual number of iterations.

Since the output is produced in redundant form, we have to consider a redundant-to-binary conversion, which – if the outputs are produced in digit–parallel form as for x_{N+1} and y_{N+1} in rotation mode—is implemented using a fast conventional adder for the CS and BSD number system. For the vectoring mode, the output magnitude x_{N+1} is calculated in MSD form. Here, it is advantageous to apply the "on–the–fly conversion" derived in [33], which involves only a wordlength–independent and small amount of additional latency. We take into account either conversion scheme by introducing the delay term T_{conv}.

Additionally, we finally have to implement the scale factor correction. Please note that the scale factor correction can either be implemented by a fixed coefficient multiplier with fixed factor $\frac{1}{K}$ or by introducing additional rotations (e.g., [10], [7]) in order to make the scale factor a simple number (e.g., a power of two, which can be corrected by a simple shift). Since the additional rotations are implemented by combining two CORDIC iterations and since exactly the same signs—steering the dependent variable iterations—are determined using DCORDIC, this "additional iterations method" can obviously also be applied to DCORDIC. A detailed comparison of these methods is beyond the scope of this paper, it should be only mentioned that the choice of either a fixed coefficient multiplier or the "additional iterations method" is left to the designer. We account for either of these two methods by introducing the delay T_{scale}.

Hence, the total computation time for N iterations in rotation mode is given by:

$$\begin{aligned} T_{Rot} &= S_p + N \cdot I_r + T_{conv} + T_{scale} \\ &= N_p + 1 + 2.5 \cdot N + T_{conv} + T_{scale} \\ &= 3.5 \cdot N + 1 + T_{conv} + T_{scale} \end{aligned} \quad (9)$$

and for the vectoring mode by

$$\begin{aligned} T_{Vec} &= S_y + N \cdot I_v + T_{conv} + T_{scale} \\ &= N_{xy} + 2 + 3 \cdot N + T_{conv} + T_{scale} \\ &= N + \log_2 N + 5 + 3 \cdot N + T_{conv} + T_{scale} \\ &= 4 \cdot N + \log_2 N + 5 + T_{conv} + T_{scale} \end{aligned} \quad (10)$$

4.5 DCORDIC Architectures

The digit parallel implementation of the DCORDIC algorithm results in very fast parallel redundant architectures. For the steering variables, i.e., the p component in rotation mode and the y component in vectoring mode, there is *always* a 3–2 or a 4–2 subtraction followed by an absolute value computation. The dependent variables, i.e., x and y in rotation mode and x and p in vectoring mode, are updated using simple 4–2 additions and steered 4–2 additions/subtractions, respectively.

Note that in the following figures, the plus boxes represent GFAs and the minus boxes negations together with GFAs. The absolute value boxes represent the combinational logic for implementing CS or BSD absolute value computation. Since the basic structure of CS and BSD architectures is identical, the figures can be used as a reference for implementing both possibilities.

If the signs are defined to be the MSDs of two's complement numbers, hence with 0 indicating a positive number and 1 indicating a negative number, instead of the definition given in (1), the sign multiplication as given in (6) is replaced by a simple EXOR operation performing the differential decoding.

4.5.1 Rotation Mode

The parallel and pipelined architecture for the DCORDIC rotation mode is shown in Fig. 2. Pipeline slices are indicated there by dashed lines. Every signal crossing a slice is delayed by a pipeline register. The p component block with MSD first subtraction and absolute value computation is a bit-level systolic array with purely local communication. Due to the MSD first absolute value computation, the pipeline slices are diagonal. This also implies a preskewing triangle in order to correctly delay the successive input digits of the p component. Because of the sign delay S involved with the absolute value computation, the x and y inputs have to be delayed for S clock cycles (see Section 4.3). From sign($\hat{p}_1$) and the sign of the initial p component sign(p_0), the signs of the following p iteration components are differentially decoded. These signs steer the operations of the x and y component iteration. The steered bitparallel additions/subtractions performed in the x and y block are vertically pipelined. Every 4–2 adder is cut by two pipeline slices. Since a 4–2 adder is implemented by two cascaded stages of 3–2 adders (with pseudo overflow correction for the MSD), this corresponds to a delay of one 3–2 adder with pseudo overflow correction, which matches the delay of one 3–2 adder and one absolute value computation for the p component. If we consider additionally the delay of the number conversion and scale factor correction, it can easily be verified that the delay given in (9) holds.

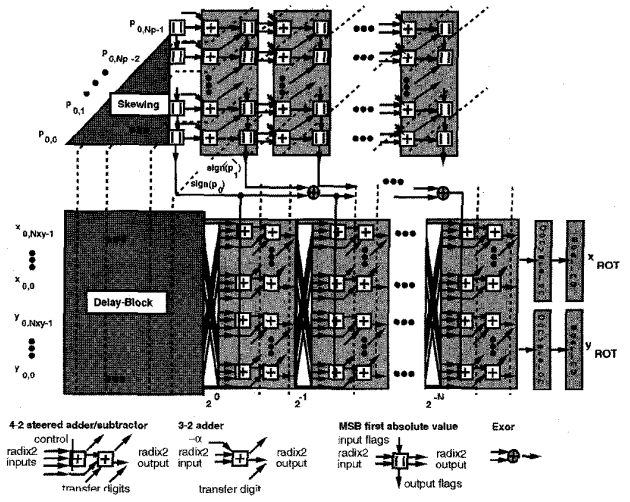


Fig. 2. Parallel architecture for the rotation mode.

4.5.2 Vectoring Mode

The parallel and pipelined architecture is shown in Fig. 3. The y component block consists of MSD first subtractions and absolute value computations. For the x component, in principle, the redundant addition $\hat{x}_{i+1} = \hat{x}_i + |\hat{y}_i| \cdot 2^{-i}$ could be implemented in digit parallel or MSD first mode. However, since $|\hat{y}_i|$ is calculated in MSD first mode, it is necessary to implement this addition in MSD first mode, too. From the absolute value computation, the differentially decoded signs of the y iteration component are computed, which are used to steer the operation for the p component. The p block consists of bitparallel steered 3–2 additions/subtractions.

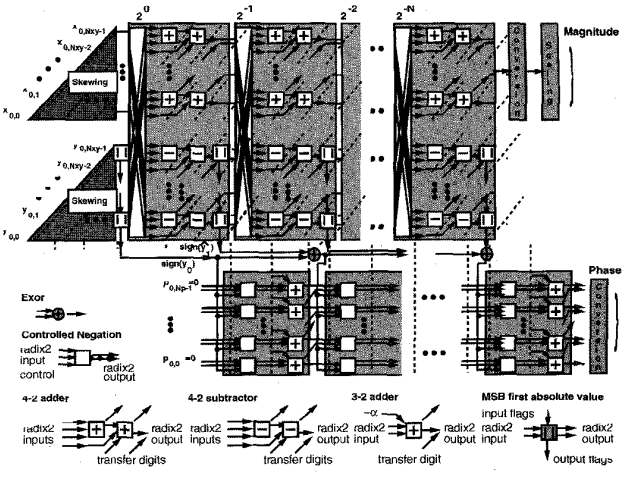


Fig. 3. Parallel architecture for the vectoring mode.

TABLE 1
SYNTHESIS RESULTS FOR DCORDIC ROTATION MODE (MAX CONDITIONS INCLUDING WIRE LOAD ESTIMATES, 1.0μ CMOS EUROPEAN SILICON STRUCTURES ES2)

Technology	Design	Number System	Clock	Wordlength	Standard Cell Area
1.0μCMOS	standard cell	Carry Save	62.5 MHz	12 bit	6.9 mm^2
1.0μCMOS	standard cell	Carry Save	62.5 MHz	16 bit	11.7 mm^2

Due to the MSD first operation for the x and y component, the corresponding pipeline slices are diagonal, as shown in Fig. 3. Again skewing triangles are required for the x and y inputs. It should be noted that due to the MSD first operation together with the chosen pipelining scheme, x and y digits shifted for 2^{-i} have to be delayed by i registers.

If p_0 is equal to zero (as is usual for vectoring mode), the p calculation starts after the sign delay S discussed in Section 4.3. It can easily be verified from Fig. 3 that the total delay given in (10) holds.

5 VHDL GENERATOR AND IMPLEMENTATION RESULTS

We developed a generator producing synthesizable VHDL descriptions for the DCORDIC architectures given arbitrary sets of algorithmic parameters. The Generator itself consists of generic VHDL models[4] and is part of the ComBox Library [1], a library for VLSI implementation of Communication Systems which is currently under development at our institute. The generic model parameters include all involved wordlengths (input, output, internal), the number of stages, the degree of pipelining to be applied, the chosen coding for the angle values α_i, and the internal number system (carry save or binary signed digit). The parameterized models produced by the generator can be synthesized using commercial logic synthesis tools. Using the optimized generator models and logic synthesis, it was possible to achieve further area improvements compared to a previous CORDIC processor design (100 MHz clock using typical conditions) we published in [2]. As also becomes clear from Table 1, the achieved throughput is independent of the wordlength due to the bit-level pipelined CS architecture. Note that here only every second of the pipeline slices shown in Fig. 2 was implemented.

6 DISCUSSION

A number of recent publications dealing with redundant CORDIC implementations of the rotation mode or sine and cosine computation which is a special case of the rotation mode ([14], [15], [16], [17], [18], [19], [20], [21], [22]) describe sign-estimation techniques to make an efficient and fast redundant CORDIC implementation possible.

Here, the sign of the steering iteration component (p_i in rotation and y_i in vectoring mode) is estimated from a given number of most significant digits (MSDs). However, since an exact sign computation involves all digits, such a sign estimation is subject to errors.

4. It should be noted here that the VHDL facilities for generic modeling, i.e., generics and generate statements, make it possible to write such generators.

In [21], a redundant signed binary digit set with values $\hat{s}\epsilon(-1, 0, +1)$ is employed for the sign estimate. Here, +1 and −1 denote that the sign can be exactly determined from the MSDs, while $\hat{s} = 0$ covers the case that no decision on the sign is possible. In this case, no rotation is performed, which leads to variability of the usually constant scale factor $\hat{K} = \prod_{i:\hat{s}\neq 0} \frac{1}{|\cos(\alpha_i)|}$. If it is not necessary to compensate for the scaling, this method is very efficient. Otherwise, a division by $1/\hat{K}$ has to be provided following the last CORDIC iteration.

In order to avoid additional area and delay for such an additional division, constant scale factor redundant (CFR) implementations were derived in [14], [15], [16], [17], [18], [19], [20], [21], [22]. Here, for the sign estimate only the values $\hat{s}\epsilon(-1, +1)$ are possible, since then every rotation is really performed and a constant scale factor is achieved. However, wrong choices of the sign estimate lead to rotation errors, which have to be corrected by repetition of some iterations. In [16], two methods are described. Using the *double correcting method*, there is an about twofold increase in the complexity of the first half of the iterations, and using the *correcting iteration method* the number of iterations which have to be repeated is a function of the number t of MSDs from which the sign is estimated. At least every $(t - 1)$th iteration has to be repeated in order to ensure convergence. Usually [15], [22], three MSDs are employed, which leads to 50% additional iterations for a given wordlength. The correcting iteration method was extended to the vectoring mode in [25] leading to the same number of additional iterations as for the redundant rotation mode. Below, we compare throughput, latency and area of the different redundant CORDIC methods. We consider an unfolded pipelined implementation as discussed in this paper.

Throughput: Since the resulting circuit is purely feed-forward, pipelining can be introduced with an arbitrary degree for every method. Hence, there should be no difference in terms of the resulting throughput.

In order to compare latency and area of DCORDIC to the sign estimation methods, we estimate the latency of the iteration stages given in [16] for the correcting iterations method and the double iterations method for the rotation mode, and for the method given in [25] for the vectoring mode. We assume that three MSDs are used for the sign estimation (as also reported in [22], [15]), which leads to 50% additional iterations, and we neglect the effect of the number conversion and scale factor correction which have to be implemented for either of these methods.

Latency: The latency in clock cycles for DCORDIC rotation mode (without the latency involved with the number conversion and scale factor correction) and vectoring mode

is obvious from (9) and (10). It was also shown there that the clock period for both cases is equal to the period of a single pipelined GFA stage.

For the rotation mode correcting iteration method, we assume for each iteration a delay of 2.5 GFAs, since obviously 2 GFAs are necessary for the redundant 4–2 additions of the x and y variables, and the delay of the steered negation which is also necessary is assumed to be equal to 0.5 GFA delays. The p path including the sign estimation is assumed to be at least as fast as this critical path, since there only 3–2 additions are necessary since the α_i values are given in conventional form. Since 1.5 N iterations are necessary, we have a total delay (excluding number conversion and scale factor correction) of $1.5 \cdot N \cdot 2.5 = 3.75 \cdot N$ GFA delays. For the double correcting method, we assume that a double iteration exhibits about twice the delay of a usual iteration which is given by 2.5 GFA delays. We also recognize the fact that the doubling of iterations is only necessary for the first half of the total number of iterations, and hence count $1.5 \cdot N \cdot 2.5$ GFA delays leading to the same delay as for the correcting iteration method.

For the vectoring mode [25], the situation is different. Since the sign estimation has to be performed for the y variable, we have a critical path consisting of a 4–2 steered addition/subtraction plus the additional sign estimation for three MSDs. Hence we assume the delay for each iteration to be equal to two GFA delays for the 4–2 addition, 0.5 GFA delays for the steered negation, and another GFA delay for the sign estimation, resulting in 3.5 GFA delays per iteration leading to a total of $1.5 \cdot N \cdot 3.5 = 5.25$ GFA delays.

Combinational Area: In order to compare the conbinational area necessary for implementation of the various methods, we assume that the additional area necessary for the absolute value computation and differential decoding in DCORDIC is compensated for by the fact that for the p component in rotation mode and for the x and y component in vectoring mode no steered negations are necessary. We also neglect the area necessary for the sign estimation, and hence have the same area per iteration for all methods discussed here. Hence, the total amount of area is proportional to the number of stages, which is shown in Table 2. As is obvious from Table 2, there are advantages in delay as well as area for the DCORDIC for both modes. Please note, however, that in particular the area can be considered only to be a rough estimate, since it is based on simply counting the number of stages. It mainly indicates the area for the combinational parts of the circuit.

Latches: Generally, the DCORDIC algorithms suffer from a larger latch consumption than the other redundant methods. The actually needed number of latches depends on the chosen pipelining degree. Using the generic DCORDIC VHDL models, the user can specify the pipelining degree as a simple generic model parameter. This allowed us to determine the optimum pipelining degree—for the used European Silicon Structures (ES2) 1.0μ standard cell technology—with regard to area–time (AT) efficiency. An implementation with one pipeline slice per 4–2 addition/subtraction turned out to be most efficient, corresponding to an implementation including only every second of the pipeline slices as indicated in Figs. 2 and 3. In the following we set $N_{xy} = N_p = N$ and $N + 1 \sim N$ for reasons of simplicity. Then, the number of latches required for one stage of the sign estimation methods is given by $6N$, since two latches are necessary for each of the N digits for the three iteration components. On the whole, we need $6N \cdot 1.5N = 9N^2$ latches for the $1.5N$ iterations.

For DCORDIC rotation mode, we need $N^2/4$ latches for the p preskewing triangle, N^2 latches for the x and y input delay. For the x and y iterations, we need $4N$ latches per iteration stage, hence a total of $4N^2$ latches. Finally, for the p iterations, we need five latches per two digits and iteration due to the chosen pipelining, resulting in $5/2N^2$ latches. This leads on the whole to $7.75N^2$ latches.

For DCORDIC vectoring mode, we need $2 \cdot N^2/4$ latches for the skewing triangles, and three latches per digit for every x or y iteration (this can be concluded from Fig. 3). Hence we need $6N^2$ latches for the total x and y component. For the p component we have on the average 1.5 pipeline slices per stage corresponding to three latches per digit and stage, resulting in a total of $3N^2$ latches. Finally, we have to take into account the latches involved with the shifts for the x and y component. For a shift i and the chosen pipelining we need 2 latches/digit $\cdot \frac{i}{2} \cdot (N - i)$ shifted digits $= i \cdot N - i^2$ latches, leading to a total of

$$N \cdot \sum_{i=0}^{N} i - \sum_{i=0}^{N} i^2 = \frac{N^2 \cdot (N+1)}{2} - \frac{N \cdot (N+1) \cdot (2N+1)}{6} = \frac{N^3 + 1}{6} \sim \frac{N^3}{6}$$

latches. Hence, for the whole DCORDIC vectoring mode, we need $9.5N^2 + \frac{N^3}{6}$ latches.

Combination of Rotation and Vectoring Mode: Although no joint implementation of rotation and vectoring mode was described in the literature, it seems that this should be relatively easily possible for the sign estimation methods. It has to be noted, that for DCORDIC, such a joint implementation is far from being straightforward, since the structure of the architectures (see Figs. 2 and 3) is rather different. However, if *all* operations for x, y, and p are implemented in MSD first mode, and some control logic is implemented additionally, a joint implementation should be possible. The resulting structure will rather be similar to the DCORDIC vectoring mode than the rotation mode.

Generalization: Although the focus of this paper is on the computation of trigonometric functions, it should be mentioned that the DCORDIC algorithms and architectures can be easily transferred to the hyperbolic (i.e., for computing hyperbolic functions) and linear (i.e., for computing division and multiplication) CORDIC operational modes. The transformations introduced here for the CORDIC iteration can be generally applied to a whole class of sign directed "shift and add" algorithms (e.g., [34]). The common feature of these algorithms is that one variable is recursively driven to a particular value, such that other variables yield the desired result. This technique is sometimes also called "iterative cotransformation" [35].

TABLE 2
COMPARISON OF REDUNDANT CORDIC METHODS

Method	Total Delay / GFA delays	Combinational Area / Iterations	Latches
double iteration [16]	$3.75 \cdot N$	$1.5 \cdot N$	$9 \cdot N^2$
correcting iteration [16]	$3.75 \cdot N$	$1.5 \cdot N$	$9 \cdot N^2$
DCORDIC rotation mode	$3.5 \cdot N + 1$	N	$7.75 \cdot N^2$
vectoring mode [25]	$5.25 \cdot N$	$1.5 \cdot N$	$9 \cdot N^2$
DCORDIC vectoring mode	$4 \cdot N + \log_2 N + 5$	N	$9.5 \cdot N^2 + N^3/6$

7 Summary and Conclusions

In this paper, we presented algorithm transformations for the CORDIC resulting in the novel "Differential CORDIC algorithm DCORDIC." DCORDIC is realized as a sequence of additions/subtractions and absolute value computations, which can be efficiently implemented using redundant number systems with MSD first operation. Both accuracy and numeric properties of the DCORDIC algorithm were shown to be exactly the same as for the conventional CORDIC algorithm. Therefore, a fast and efficient implementation with a constant scale factor becomes possible without additional correcting iterations as necessary for previously known redundant CORDIC methods. After presenting on–line algorithms for the absolute value computations involved with DCORDIC, we derived bit parallel architectures. Finally we presented a DCORDIC VHDL Generator together with several logic synthesis results, which prove that very fast and efficient implementations are possible. A rough comparison of DCORDIC with known redundant CORDIC methods indicates that DCORDIC is superior in terms of latency as well as area, while the same throughput can be achieved.

The transformations introduced here can also generally be applied to the class of "shift and add" algorithms [34] and algorithms employing the principle of iterative cotransformation [35].

Appendix: MSD First Carry Save Absolute Value Computation

It is well known that a CS-number can be regarded as two twos complement binary numbers, called C and S number. CS negation is hence implemented by inversion of all bits and a final addition of "+2."

In the following discussion, we consider an input CS number X consisting of the two twos complement numbers C and S. The output to be calculated is the output CS number $Y = |X|$. The number X is hence given by:

$$X = -(c_{W-1} + s_{W-1}) \cdot 2^{W-1} + \sum_{j=0}^{W-2} (c_j + s_j) \cdot 2^j$$

We assume that no overflows occur, i.e., the wordlength of X and Y is large enough to hold the actual value. When designing the MSD–first absolute value computation, we have to take into account the effect of actually adding C and S to produce the two's complement sum, i.e., the value of the X number.

We first summarize the basic principle of the algorithm.

Inversion will be done by bit inversion and addition of +2. This addition will be done in the next stage of the implementation (in this sense the algorithm is NOT a complete on–line absolute value algorithm). A decision given only the MSDs is possible only if C and S have the same sign. Hence, if $X_{W-1} = 00$ we decide positive and do nothing, if $X_{W-1} = 11$ we decide negative and complement both operands. Otherwise (for $X_{W-1} = 10$ or 01), we have to defer the decision until more digits are inspected. The problem is that even for an unknown decision, a valid output has to be produced, since we need an algorithm with a small on–line delay as was stated for BSD numbers in Section 4.1.

As long as the following digits are also 01 or 10 we leave the digits as is because their value will remain 1 both if we complement or not. If we now find for the first time a digit which is 00, we decide for a negative number since no carry occurs so that the most significant ones are not modified. The digit 00 is complemented to 11 and all following digits are also complemented. If otherwise, we find for the first time a digit which is 11 we decide for a positive number, since a carry is definitely generated at this level, and the most significant ones will become zeroes. This and all following digits are left unchanged.

However, we developed a different method which leads to a smaller combinational delay. Using the first algorithm stated above, the most significant ones are left unchanged, and a carry is enforced since the first output digit which does not have the value one is set to 11 anyway, resulting in a chain of most significant zeroes. Hence, we can alternatively already output the most significant zeroes: For the case (different signs: $c_{W-1} + s_{W-1} = 1$), we output $Y_{W-1} = 00$, assuming that a carry will propagate from lower bit levels up to the MSD level (hence assuming the value X is positive). Note that we later have to correct the error possibly introduced here. The value of the input number is then given by:

$$X = -2^{W-1} + \sum_{j=0}^{W-2} (c_j + s_j) \cdot 2^j$$

As long as $(c_j + s_j) = 1$ for $W - 2 \geq j > W - 2 - N$, we continue outputting $c_j = s_j = 0$, still assuming that a carry will propagate from lower bit levels up to this bit level. Note that $c_{W-2-N} + s_{W-2-N} = 2$ is sufficient to produce such a carry. The actual value for N additional bit levels accounted for in this way is:

$$V = -2^{W-1} + \sum_{i=0}^{N-1} 2^{W-2-i}$$
$$= -2^{W-N-1} \qquad (11)$$

while the outputted value is zero. If $c_{W-2-N} + s_{W-2-N} = 2$ is observed taking into account the $(N + 1)$st not MSD digit with a value of $2 \cdot 2^{W-2-N} = 2^{W-N-1}$, this obviously compensates for the value of V. Hence, it is sufficient to output $Y_{W-2-N} = 00$. Simultaneously, it is proved that the value of X is positive, because the sum of the upper digits taken already into account is zero and the remaining digits are all positive. Hence all remaining digits can be left unchanged.

If otherwise, taking into account the $(N + 1)$st not MSD digit, $c_{W-2-N} + s_{W-2-N} = 0$ is observed, the value of $V = -2^{W-N-1}$ can obviously not be compensated for by the remaining positive digit levels. Hence, the sum of the C and the S number is negative: $X < 0$. The value of the complete number is then obviously given by:

$$X = V + \sum_{j=0}^{W-2-N-1} (c_j + s_j) \cdot 2^j$$

However, we want to compute the absolute value, which is here given by $-X$:

$$\begin{aligned}
-X &= -V - \sum_{j=0}^{W-2-N-1} (c_j + s_j) \cdot 2^j \\
&= 2^{W-2-N+1} - \sum_{j=0}^{W-2-N-1} (c_j + s_j) \cdot 2^j \\
&= 2 \cdot (1 + \sum_{j=0}^{W-2-N-1} 2^j) \\
&\quad - \sum_{j=0}^{W-2-N-1} (c_j + s_j) \cdot 2^j \\
&= 1 + \sum_{j=0}^{W-2-N-1} 2^j \\
&\quad - \sum_{j=0}^{W-2-N-1} c_j \cdot 2^j \\
&\quad + 1 + \sum_{j=0}^{W-2-N-1} 2^j \\
&\quad - \sum_{j=0}^{W-2-N-1} s_j \cdot 2^j \\
&= 1 + \sum_{j=0}^{W-2-N-1} (1 - c_j) \cdot 2^j \\
&\quad + 1 + \sum_{j=0}^{W-2-N-1} (1 - s_j) \cdot 2^j \\
&= 1 + \sum_{j=0}^{W-2-N-1} \bar{c}_j \cdot 2^j \\
&\quad + 1 + \sum_{j=0}^{W-2-N-1} \bar{s}_j \cdot 2^j \qquad (12)
\end{aligned}$$

Note that $(1 - c_j)$ is equivalent to the inverted bit $\bar{c}_i$. This means that we output $c_{W-2-N} + s_{W-2-N} = 0$, the remaining bits of both the C and the S number have to be inverted, and the deferred correction of +2 has to be implemented. The resulting algorithm is stated as:

Algorithm [Optimized CS Absolute Value]
F: an enumeration type with elements
(positive, negative, nodec (no decision))
[Input]
X: a CS number
with digits $x_i \quad i \in (0, \ldots, N-1)$
(x_0 is MSD)
[Output]
Y: a positive CS number with $Y = |X|$
and digits $y_i \quad i \in (0, \ldots, N-1)$
S: sign(X)
[Algorithm]

Step 1: F_{-1} = nodec
Step 2: for $j = 0$ to $N - 1$ do
begin

$$y_j = \begin{cases} x_j & \text{if } F_j = \text{positive} \\ \text{not } x_j & \text{if } F_{j-1} = \text{negative} \\ 0 & \text{if } F_{j-1} = \text{nodec} \end{cases}$$

$$F_j = \begin{cases} F_{j-1} & \text{if } F_{j-1} = \text{positive or negative} \\ \text{positive} & \text{if } F_{j-1} = \text{nodec and } x_j = +2 \\ \text{negative} & \text{if } F_{j-1} = \text{nodec and } x_j = 0 \\ \text{nodec} & \text{if } F_{j-1} = \text{nodec and } x_j = +1 \end{cases}$$

end

$$\text{Step 3: } S \leftarrow \begin{cases} +1 & \text{if } F_{N-1} = \text{positive or nodec} \\ -1 & \text{if } F_{N-1} = \text{negative} \end{cases}$$

Although the derivation of this algorithm is much more complicated, it leads to superior implementations compared to the simple algorithm stated initially. Note that the algorithmic description of the simple algorithm is very similar to the one given for BSD absolute value calculation. If you compare this (cf Section 4.1) and the optimized algorithm, it becomes clear that the output function for y_j is much simpler here, leading to a very simple circuit [4].

ACKNOWLEDGMENTS

We wish to thank Tomas Lang and the anonymous reviewers for their efforts and highly valuable comments, which were of great help in improving the quality of this paper. We also wish to thank our students, Ralf Gaisbauer and Christian Lütkemeyer, for their important contributions to this work.

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under grant Me 653/13.

REFERENCES

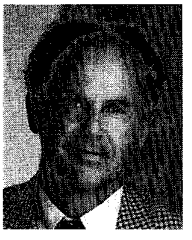
[1] H. Meyr, H. Dawid, O. Joeressen, and P. Zepter, "Design of High Speed Communication Systems," *Proc. ISCAS*, p. 2.134, IEEE, May 1994.

[2] H. Dawid and H. Meyr, "High Speed Bit-Level Pipelined Architectures for Redundant CORDIC Implementation," *Proc. Int'l Conf. Application Specific Array Processors*, pp. 358-372, Oakland, Calif., IEEE CS Press, Aug. 1992.

[3] H. Dawid and H. Meyr, "Very High Speed CORDIC Implementation: Algorithm Transformation and Novel Carry-Save Architecture," *Proc. European Signal Processing Conf. EUSIPCO '92*, pp. 358-372, Brussels, Elsevier Science Publications, Aug. 1992.
[4] H. Dawid and H. Meyr, "VLSI Implementation of the CORDIC Algorithm Using Redundant Arithmetic," *Proc. IEEE Int'l Symp. Circuits and Systems ISCAS*, pp. 1,089-1,092, San Diego, May 10-13, 1992.
[5] J.E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electronic Computing*, vol. 8, pp. 330-334, Sept. 1959.
[6] J.S. Walther, "A Unified Algorithm for Elementary Functions," *AFIPS Spring Joint Computer Conf.*, vol. 38, pp. 379-385, 1971.
[7] G. Schmidt, D. Timmermann, J.F. Böhme, and H. Hahn, "Parameter Optimization of the CORDIC Algorithm and Implementation in a CMOS Chip," *Proc. EUSIPCO '86*, pp. 1,219-1,222, 1986.
[8] J. Lee and T. Lang, "On-Line CORDIC for Generalized Singular Value Decomposition," *SPIE Vol. 1058 High Speed Computing II*, pp. 235-247, 1989.
[9] S. Note, J. van Meerbergen, F. Catthoor, and H. de Man, "Automated Synthesis of a High Speed CORDIC Algorithm with the Cathedral-III Compilation System," *Proc. IEEE ISCAS '88*, pp. 581-584, 1988.
[10] J. Bu, E.F. Deprettere, and F. du Lange, "On the Optimization of Pipelined Silicon CORDIC Algorithm," *Proc. EUSIPCO '88*, pp. 1,227-1,230, 1988.
[11] J.R. Cavallaro and F.T. Luk, "Floating Point CORDIC for Matrix Computations," *Proc. IEEE Int'l Conf. Computer Design*, pp. 40-42, 1988.
[12] J.R. Cacallaro and F.T. Luk, "CORDIC Arithmetic for a SVD Processor," *J. Parallel and Distributed Computing*, vol. 5, pp. 271-290, 1988.
[13] A.A. de Lange, A.J. van der Hoeven, E.F. Deprettere, and J. Bu, "An Optimal Floating-Point Pipeline CMOS CORDIC Processor," *Proc. IEEE ISCAS '88*, pp. 2,043-2,047, 1988.
[14] T. Noll, "Carry-Svae Architectures for High-Speed Digital Signal Processing," *J. VLSI Signal Processing*, vol. 3, pp. 121-140, June 1991.
[15] R. Künemund, H. Söldner, S. Wohlleben, and T. Noll, "CORDIC Processor with Carry-Save Architecture," *Proc. ESSCIRC '90*, pp. 193-196, 1990.
[16] N. Takagi, T. Asada, and S. Yajima, "Redundant CORDIC Methods with a Constant Scale Factor for Sine and Cosine Computation," *IEEE Trans. Computers*, vol. 40, no. 9, pp. 989-995, Sept. 1991.
[17] M.D. Ercegovac and T. Lang, "Redundant and On-Line CORDIC: Application to Matrix Triangularisation and SVD," *IEEE Trans. Computers*, vol. 38, no. 6, pp. 725-740, June 1990.
[18] H.X. Lin and H.J. Sips, "On-Line CORDIC Algorithms," *IEEE Trans. Computers*, vol. 38, no. 8, pp. 1,038-1,052, Aug. 1990.
[19] T. Noll, "Carry-Save Arithmetic for High-Speed Digital Signal Processing," *IEEE ISCAS '90*, vol. 2, pp. 982-986, 1990.
[20] H. Yoshimura, T. Nakanishi, and H. Yamauchi, "A 50 MHz CMOS Geometrical Mapping Processor," *IEEE Trans. Circuits and Systems*, vol. 36, no. 10, pp. 1,360-1,363, 1989.
[21] M.D. Ercegovac and T. Lang, "Implementation of Fast Angle Calculation and Rotation Using On-Line CORDIC," *Proc. IEEE ISCAS '88*, pp. 2,703-2,706, 1988.
[22] N. Takagi, T. Asada, and S. Yajima, "A Hardware Algorithm for Computing Sine and Cosine Using Redundant Binary Representation," *Systems and Computers in Japan*, vol. 18, no. 8, pp. 1-9, 1987.
[23] B. Parhami, "On the Implementation of Arithmetic Support Functions for Generalized Signed-Digit Number Systems," *IEEE Trans. Computers*, vol. 42, no. 3, pp. 379-384, Mar. 1993.
[24] N. Takagi, T. Asada, and S. Yajima, "A Hardware Algorithm for Computing Sine and Cosing Using Redundant Binary Representation," *Trans. IEICE Japan*, vol. J69-D, no. 6, pp. 841-847, 1986 (in Japanese).
[25] J. Lee and T. Lang, "Constant-Factor Redundant CORDIC for Angle Calculation and Rotation," *IEEE Trans. Computers*, vol. 41, no. 8, pp. 1,016-1,035, Aug. 1992.
[26] J. Duprat and J.-M. Muller, "The CORDIC Algorithm: New Results for Fast VLSI Implemenation," *IEEE Trans. Computers*, vol. 42, no. 2, pp. 168-178, Feb. 1993.
[27] B. Parhami, "Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations," *IEEE Trans. Computers*, vol. 39, no. 1, pp. 89-98, Jan. 1990.
[28] B. Parhami, "Carry-Free Addition of Recoded Binary Signed-Digit Numbers," *IEEE Trans. Computers*, vol. 38, pp. 1,470-1,476, 1988.
[29] A. Vandemeulebroecke, E. Vanzieleghem, T. Denayer, and P.G.A. Jespers, "A New Carry-Free Diversion Algorithm and Its Application to a Single-Chip 1024-b RSA Processor," *IEEE J. Solid State Circuits*, vol. 25, no. 3, pp. 748-765, 1990.
[30] M.D. Ercegovac, "On-Line Arithmetic: An Overview," *Real Time Signal Processing VII: Proc. SPIE*, vol. 495, pp. 86-93, 1984.
[31] T. Lang, private communication, Jan. 1994.
[32] Y.H. Hu, "The Quantization Effects of the CORDIC Algorithm," *IEEE Trans. Circuits and Systems*, vol. 40, no. 4, pp. 834-844, 1992.
[33] M.D. Ercegovac and T. Lang, "On-the-Fly Conversion of Redundant into Conventional Representations," *IEEE Trans. Computers*, vol. 36, pp. 895-897, 1987.
[34] W.H. Specker, "A Class of Algorithms for ln X, exp X, sin X, cos X, $\tan^{-1}$ X and $\cot^{-1}$ X," *IEEE Trans. Electronic Computers*, vol. 14, no. 1, pp. 85-86, 1965.
[35] N.R. Scott, *Computer Number Systems and Arithmetic*. Englewood Cliffs, N.J.: Prentice Hall, 1988.

Herbert Dawid (S'92) received the Dipl-Ing degree (summa cum laude) in electrical engineering from Aachen University of Technology (RWTH) in 1989. He is currently pursuing his PhD degree at the Institute for Integrated Systems in Signal Processing (IEE) at Aachen University, where he has headed the VLSI group since 1990.

His current research interests include computer arithmetic, digital signal processing, and, especially, digital receiver design, as well as VLSI architecture design, modeling, synthesis, and simulation.



Heinrich Meyr received his MS and PhD from the ETH in Zurich, Switzerland. He is currently engaged in a dual role in academia and industry. He is a professor of electrical engineering at the Aachen University of Technology (TWTH Aachen), where he heads and institute involved in the analysis and design of complex signal processing systems for communication applications. He has worked extensively for the past 20 years in the areas of communication theory, synchronization, and digital signal processing. His research has been applied to the design of many industrial products. He was a cofounder of CADIS GmbH (recently acquired by Synopsys, Mountain View, California), a company which commercialized the tool suite "COSSAP," which is extensively used in the communications industry worldwide. He is also a member of the Board of Directors of ASCOM Ltd., a large international communication concern with headquarters in Switzerland.

Dr. Meyr has published numerous IEEE papers and is author (with Dr. G. Ascheid) of the book *Synchronization in Digital Communication*, volume I (John Wiley & Sons, 1990). He is presently completing volume II, which deals primarily with a digital synchronization algorithm. He holds more than a dozen patents. He served as a vice president for international affairs of the IEEE Communications Society and is a fellow of the IEEE.