Chapter 24

CORDIC Algorithms and Architectures

Herbert Dawid
Synopsys, Inc.
DSP Solutions Group
Kaiserstr. 100
D-52134 Herzogenrath
Germany
email: dawid@synopsys.com

Heinrich Meyr
Aachen University of Technology
Integrated Systems in Signal Processing
ISS –611 810–
D–52056 Aachen
Germany
email: meyr@ert.rwth-aachen.de

Abstract

Digital signal processing (DSP) algorithms exhibit an increasing need for the efficient implementation of complex arithmetic operations. The computation of trigonometric functions, coordinate transformations or rotations of complex valued phasors is almost naturally involved with modern DSP algorithms. Popular application examples are algorithms used in digital communication technology and in adaptive signal processing. While in digital communications, the straightforward evaluation of the cited functions is important, numerous matrix based adaptive signal processing algorithms require the solution of systems of linear equations, QR factorization or the computation of eigenvalues, eigenvectors or singular values. All these tasks can be efficiently implemented using processing elements performing vector rotations. The COordinate Rotation DIgital Computer algorithm (CORDIC) offers the opportunity to calculate all the desired functions in a rather simple and elegant way.

The CORDIC algorithm was first introduced by Volder [1] for the computation of trigonometric functions, multiplication, division and datatype conversion, and later on generalized to hyperbolic functions by Walther [2]. 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 (sometimes called *microrotations*) but with variable rotation direction. Due to the simplicity of the involved operations the CORDIC algorithm is very well

suited for VLSI implementation. However, the CORDIC iteration is not a perfect rotation which would involve multiplications with sine and cosine. The rotated vector is also scaled making a *scale factor correction* necessary.

We first give an introduction into the CORDIC algorithm. Following we discuss methods for scale factor correction and accuracy issues with respect to a fixed wordlength implementation. In the second part of the chapter different architectural realizations for the CORDIC are presented for different applications:

1. Programmable CORDIC processing element
2. High throughput CORDIC processing element
3. CORDIC Architectures for Vector Rotation
4. CORDIC Architectures using redundant number systems

24.1 The CORDIC Algorithm

In this section we first present the basic CORDIC iteration before discussing the full algorithm which consists of a sequence of these iterations.

In the most general form one CORDIC iteration can be written as [2, 3]

$$\begin{aligned} x_{i+1} &= x_i - m \cdot \mu_i \cdot y_i \cdot \delta_{m,i} \\ y_{i+1} &= y_i + \mu_i \cdot x_i \cdot \delta_{m,i} \\ z_{i+1} &= z_i - \mu_i \cdot \alpha_{m,i} \end{aligned} \tag{1}$$

Although it may not be immediately obvious this basic CORDIC iteration describes a rotation (together with a scaling) of an intermediate plane vector $v_i = (x_i, y_i)^T$ to $v_{i+1} = (x_{i+1}, y_{i+1})^T$. The third iteration variable z_i keeps track of the rotation angle $\alpha_{m,i}$. The variable $m \in \{1, 0, -1\}$ specifies a circular, linear or hyperbolic coordinate system, respectively. The rotation direction is steered by the variable $\mu_i \in \{1, -1\}$. Trajectories for the vectors v_i for successive CORDIC iterations are shown in Fig. 24.1 for $m = 1$, in Fig. 24.2 for $m = 0$, and in Fig. 24.3 for $m = -1$, respectively. In order to avoid multiplications $\delta_{m,i}$ is defined to be

$$\begin{aligned} \delta_{m,i} &= d^{-s_{m,i}} && ; d: \text{ Radix of employed number system, } s_{m,i}: \text{integer number} \\ &= 2^{-s_{m,i}} && ; \text{ Radix 2 number system} \end{aligned} \tag{2}$$

For obvious reasons we restrict consideration to radix 2 number systems below: $\delta_{m,i} = 2^{-s_{m,i}}$. It will be shown later that the *shift sequence* $s_{m,i}$ is generally a nondecreasing integer sequence. Hence, a CORDIC iteration can be implemented using only shift and add/subtract operations.

The first two equations of the system of equations given in Eq. (1) can be written as a matrix-vector product

$$v_{i+1} = \begin{pmatrix} 1 & -m \cdot \mu_i \cdot \delta_{m,i} \\ \mu_i \cdot \delta_{m,i} & 1 \end{pmatrix} \cdot v_i = \boldsymbol{C}_{m,i} \cdot v_i \tag{3}$$

In order to verify that the matrix-vector product in Eq. (3) describes indeed a vector rotation and to quantify the involved scaling we consider now a general normalized

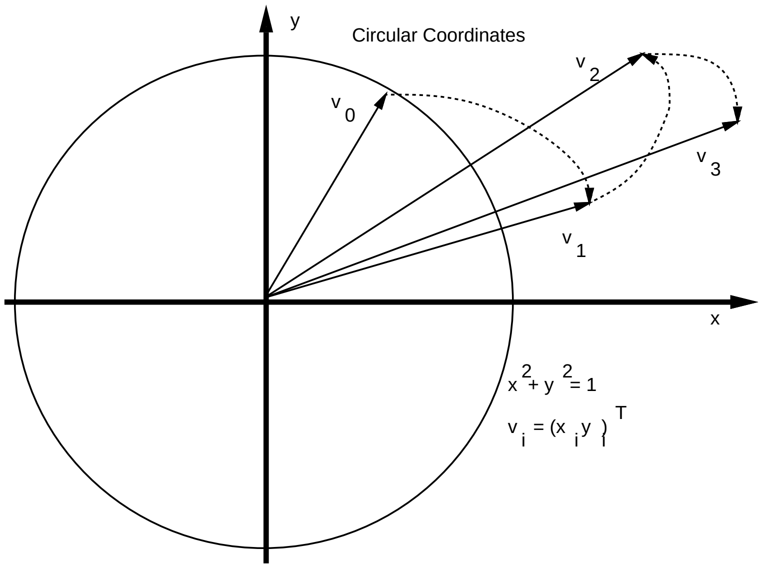

Figure 24.1 Rotation trajectory for the circular coordinate system ($m = 1$).

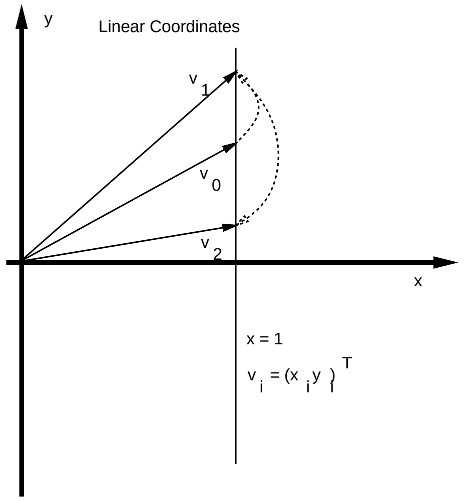

Figure 24.2 Rotation trajectory for the linear coordinate system ($m = 0$).

plane rotation matrix for the three coordinate systems. For $m = 1, 0, -1$ and an angle $\mu_i \cdot \alpha_{m,i}$ with μ_i determining the rotation direction and $\alpha_{m,i}$ representing an unsigned angle, this matrix is given by

$$\boldsymbol{R}_{m,i} = \begin{pmatrix} \cos(\sqrt{m} \cdot \alpha_{m,i}) & -\mu_i \cdot \sqrt{m} \cdot \sin(\sqrt{m} \cdot \alpha_{m,i}) \\ \frac{\mu_i}{\sqrt{m}} \cdot \sin(\sqrt{m} \cdot \alpha_{m,i}) & \cos(\sqrt{m} \cdot \alpha_{m,i}) \end{pmatrix} \qquad (4)$$

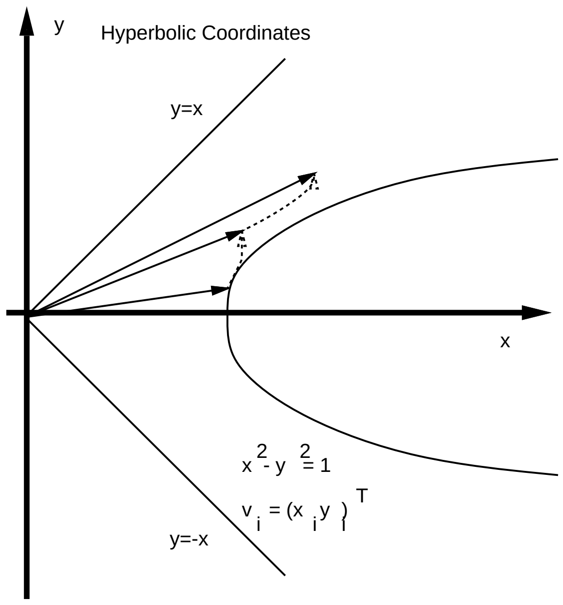

Figure 24.3 Rotation trajectory for the hyperbolic coordinate system ($m = -1$).

which can be easily verified by setting m to $1, 0, -1$, repectively, and using (for the hyperbolic coordinate system) the identities: $\sinh(z) = -isin(iz)$, $\cosh(z) = \cos(iz)$, and $tanh(z) = -i \cdot tan(iz)$ with $i = \sqrt{-1}$. The norm of a vector $(x, y)^T$ in these coordinate systems is defined as $\sqrt{x^2 + m \cdot y^2}$. Correspondingly, the norm preserving rotation trajectory is a circle defined by $x^2 + y^2 = 1$ in the circular coordinate system, while in the hyperbolic coordinate system the "rotation" trajectory is a hyperbolic function defined by $x^2 - y^2 = 1$ and in the linear coordinate system the trajectory is a simple line $x = 1$. Hence, the common meaning of a rotation holds only for the circular coordinate system. Clearly

$$\frac{1}{\cos(\sqrt{m} \cdot \alpha_{m,i})} \cdot \boldsymbol{R}_{m,i} = \begin{pmatrix} 1 & -\mu_i \cdot \sqrt{m} \cdot \tan(\sqrt{m} \cdot \alpha_{m,i}) \\ \frac{\mu_i}{\sqrt{m}} \cdot \tan(\sqrt{m} \cdot \alpha_{m,i}) & 1 \end{pmatrix}$$

holds, hence

$$\frac{1}{\cos(\sqrt{m} \cdot \alpha_{m,i})} \cdot \boldsymbol{R}_{m,i} = \boldsymbol{C}_{m,i} \quad \text{holds for} \quad \delta_{m,i} = \frac{1}{\sqrt{m}} \cdot \tan(\sqrt{m} \cdot \alpha_{m,i})$$

This proves that $\boldsymbol{C}_{m,i}$ is an unnormalized rotation matrix for $m \in \{-1, 1\}$ due to the scaling factor $\frac{1}{\cos(\sqrt{m} \cdot \alpha_{m,i})}$. $\boldsymbol{C}_{m,i}$ describes a rotation with scaling rather than a pure rotation. For $m = 0$, $\boldsymbol{R}_{m,i} = \boldsymbol{C}_{m,i}$ holds, hence $\boldsymbol{C}_{m,i}$ is a normalized rotation matrix in this case and no scaling is involved. The scale factor is given by

$$\begin{aligned} K_{m,i} &= \frac{1}{\cos(\sqrt{m} \cdot \alpha_{m,i})} = \frac{\sqrt{\cos^2(\sqrt{m} \cdot \alpha_{m,i}) + \sin^2(\sqrt{m} \cdot \alpha_{m,i})}}{\cos(\sqrt{m} \cdot \alpha_{m,i})} \\ &= \sqrt{1 + \tan^2(\sqrt{m} \cdot \alpha_{m,i})} \end{aligned} \tag{5}$$

For n successive iterations, obviously

$$v_n = \prod_{i=0}^{n-1} \boldsymbol{C}_{m,i} \cdot v_0 = \prod_{i=0}^{n-1} K_{m,i} \cdot \prod_{i=0}^{n-1} \boldsymbol{R}_{m,i} \cdot v_0$$

holds, i.e. a rotation by an angle

$$\theta = \sum_{i=0}^{n-1} \mu_i \cdot \alpha_{m,i}$$

is performed with an overall scaling factor of

$$K_m(n) = \prod_{i=0}^{n-1} K_{m,i} \tag{6}$$

The third iteration component z_i simply keeps track of the overall rotation angle accumulated during successive microrotations

$$z_{i+1} = z_i - \mu_i \cdot \alpha_{m,i}$$

After n iterations

$$z_n = z_0 - \sum_{i=0}^{n-1} \mu_i \cdot \alpha_{m,i}$$

holds, hence z_n is equal to the difference of the start value z_0 and the total accumulated rotation angle.

In Fig. 24.4, the structure of a processing element implementing one CORDIC iteration is shown. All internal variables are represented by a fixed number of digits, including the precalculated angle $\alpha_{m,i}$ which is taken from a register. Due to the limited wordlength some rounding or truncation following the shifts $2^{-s_{m,i}}$ is necessary. The adders/subtractors are steered with $-m\mu_i$, μ_i and $-\mu_i$, respectively.

A rotation by any (within some convergence range) desired rotation angle A_0 can be achieved by defining a converging sequence of n single rotations. The CORDIC algorithm is formulated given

1. a shift sequence $s_{m,i}$ defining an angle sequence

$$\alpha_{m,i} = \frac{1}{\sqrt{m}} \cdot \tan^{-1}(\sqrt{m} \cdot 2^{-s_{m,i}}) \quad \text{with } i \in \{0, \ldots, n-1\} \tag{7}$$

   which guarantees convergence. Shift sequences will be discussed in section 24.1.3.

2. a control scheme generating a sign sequence μ_i with $i \in \{0, \ldots, n-1\}$ which steers the direction of the rotations in this iteration sequence and guarantees convergence.

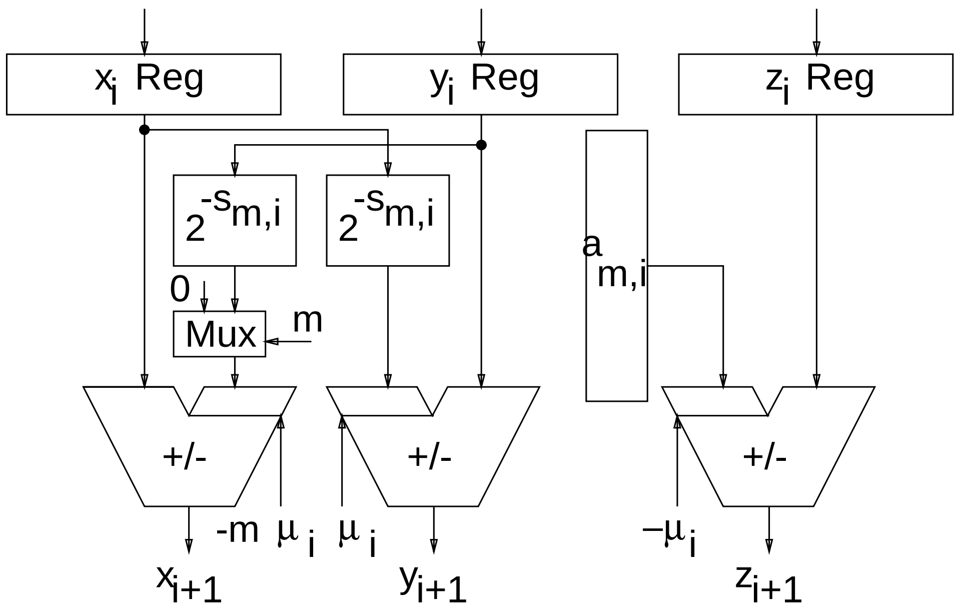

Figure 24.4 Basic structure of a processing element for one CORDIC iteration.

In order to explain the control schemes used for the CORDIC algorithm the angle A_i is introduced specifying the remaining rotation angle after rotation i. The direction of the following rotation has to be chosen such that the absolute value of the remaining angle eventually becomes smaller during successive iterations [2]

$$|A_{i+1}| = ||A_i| - \alpha_{m,i}| \tag{8}$$

Two control schemes fulfilling Eq. (8) are known for the CORDIC algorithm, the *rotation mode* and the *vectoring mode.*

24.1.1 Rotation Mode

In rotation mode the desired rotation angle $A_0 = \theta$ is given for an input vector $(x, y)^T$. We set $x_0 = x$, $y_0 = y$, and $z_0 = \theta = A_0$. After n iterations

$$z_n = \theta - \sum_{i=0}^{n-1} \mu_i \cdot \alpha_{m,i}$$

If $z_n = 0$ holds, then $\theta = \sum_{i=0}^{n-1} \mu_i \cdot \alpha_{m,i}$, i.e. the total accumulated rotation angle is equal to θ. In order to drive z_n to zero, $\mu_i = \text{sign}(z_i)$ is used leading to

$$\begin{aligned} x_{i+1} &= x_i - m \cdot \text{sign}(z_i) \cdot y_i \cdot 2^{-s_{m,i}} \\ y_{i+1} &= y_i + \text{sign}(z_i) \cdot x_i \cdot 2^{-s_{m,i}} \\ z_{i+1} &= z_i - \text{sign}(z_i) \cdot \alpha_{m,i} \end{aligned} \tag{9}$$

Obviously, for $z_0 = A_0$ and $z_i = A_i$

$$z_{i+1} = z_i - \text{sign}(z_i) \cdot \alpha_{m,i}$$

$$\begin{aligned}
&\text{hence} && \\
\operatorname{sign}(z_i)\cdot z_{i+1} &= \operatorname{sign}(z_i)\cdot z_i - \alpha_{m,i} \\
&= |z_i| - \alpha_{m,i} \\
&\text{Taking absolute values} && \\
|z_{i+1}| &= ||z_i| - \alpha_{m,i}| \\
\text{follows, satisfying Eq. (8) for } z_i &= A_i \qquad (10)
\end{aligned}$$

The finally computed scaled rotated vector is given by $(x_n, y_n)^T$. In Fig. 24.5, the trajectory for the rotation mode in the circular coordinate system is shown. It becomes clear that the vector is iteratively rotated towards the desired final position. The scaling involved with the successive iterations is also shown in Fig. 24.5.

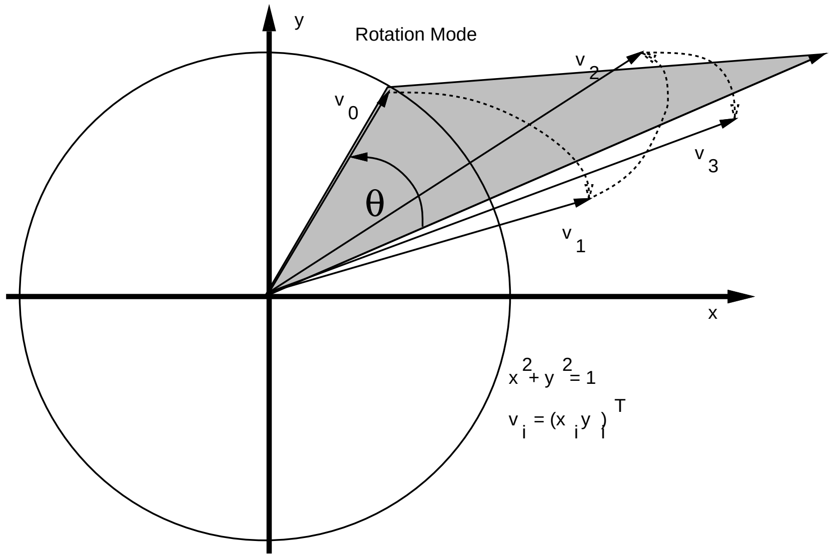

Figure 24.5 Rotation trajectory for the rotation mode in the circular coordinate system.

24.1.2 Vectoring Mode

In vectoring mode the objective is to rotate the given input vector $(x, y)^T$ with magnitude $\sqrt{x^2 + m \cdot y^2}$ and angle $\phi = -A_0 = \frac{1}{\sqrt{m}} \cdot \tan^{-1}(\sqrt{m} \cdot \frac{y}{x})$ towards the x–axis. We set $x_0 = x$, $y_0 = y$, and $z_0 = 0$. The control scheme is such that during the n iterations y_n is driven to zero: $\mu_i = -\operatorname{sign}(x_i) \cdot \operatorname{sign}(y_i)$. Depending on the sign of x_0 the vector is then rotated towards the positive ($x_0 \geq 0$) or negative ($x_0 < 0$) x–axis. If $y_n = 0$ holds, z_n contains the negative total accumulated rotation angle after n iterations which is equal to ϕ

$$z_n = \phi = -\sum_{i=0}^{n-1} \mu_i \cdot \alpha_{m,i}$$

and x_n contains the scaled and eventually (for $x_0 < 0$) signed magnitude of the input vector as shown in Fig. 24.6. The CORDIC iteration driving the y_i variable

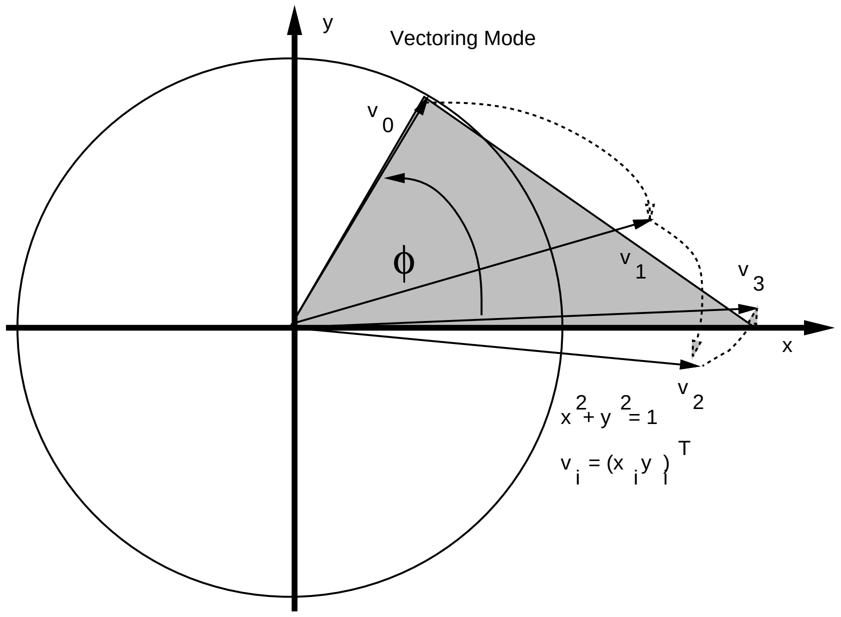

Figure 24.6 Rotation trajectory for the vectoring mode in the circular coordinate system.

to zero is given by

$$\begin{aligned} x_{i+1} &= x_i + m \cdot \operatorname{sign}(x_i) \cdot \operatorname{sign}(y_i) \cdot y_i \cdot 2^{-s_{m,i}} \\ y_{i+1} &= y_i - \operatorname{sign}(x_i) \cdot \operatorname{sign}(y_i) \cdot x_i \cdot 2^{-s_{m,i}} \\ z_{i+1} &= z_i + \operatorname{sign}(x_i) \cdot \operatorname{sign}(y_i) \cdot \alpha_{m,i} \end{aligned} \tag{11}$$

with $x_n = K_m(n) \cdot \operatorname{sign}(x_0) \cdot \sqrt{x^2 + m \cdot y^2}$. Obviously, the remaining rotation angle after iteration i is given by $A_i = -\frac{1}{\sqrt{m}} \cdot \tan^{-1}(\sqrt{m} \cdot \frac{y_i}{x_i})$. Clearly, $\operatorname{sign}(A_i) = -\operatorname{sign}(x_i) \cdot \operatorname{sign}(y_i)$ holds. With $z_0 = 0$ and $A_0 = -\phi$, $A_i = z_i - \phi$ holds. Using $\mu_i = -\operatorname{sign}(A_i)$

$$\begin{aligned} A_{i+1} &= z_{i+1} - \phi \\ &= z_i + \operatorname{sign}(x_i) \cdot \operatorname{sign}(y_i) \cdot \alpha_{m,i} - \phi \quad \text{using Eq. (11)} \\ &= z_i - \operatorname{sign}(A_i) \cdot \alpha_{m,i} - \phi \\ &= A_i - \operatorname{sign}(A_i) \cdot \alpha_{m,i} \end{aligned}$$

holds. Eq. (8) is again satisfied as was shown in Eq. (10).

24.1.3 Shift Sequences and Convergence Issues

Given the two iteration control schemes, shift sequences $s_{m,i}$ have to be introduced which guarantee convergence.

First, the question arises how to define convergence in this case. Since there are only n fixed rotation angles with variable sign, the desired rotation angle A_0 can only be approximated resulting in an angle approximation error $\Delta\phi$ [4]

$$\Delta\phi = A_0 - \sum_{i=0}^{n-1} \mu_i \cdot \alpha_{m,i}$$

$\Delta\phi$ does not include errors due to finite quantization of the rotation angles $\alpha_{m,i}$. In rotation mode, since $z_0 = A_0$, $\Delta\phi = z_n = A_n$ holds, i.e. z_n can not be made exactly equal to zero. In vectoring mode the angle approximation error is given by the remaining angle of the vector $(x_n, y_n)^T$

$$\Delta\phi = A_n = \frac{1}{\sqrt{m}} \cdot \tan^{-1}(\sqrt{m} \cdot \frac{y_n}{x_n}) = \begin{cases} \tan^{-1}(\frac{y_n}{x_n}) & \text{m} = 1 \\ \frac{y_n}{x_n} = \frac{y_n}{x_0} & \text{m} = 0 \\ \tanh^{-1}(\frac{y_n}{x_n}) & \text{m} = \text{-1} \end{cases}$$

Hence, convergence can only be defined taking into account the angle approximation error. Two convergence criteria can be derived (for a detailed discussion see [1, 2]):
1.) First, the chosen set of rotation angles has to satisfy

$$\alpha_{m,i} - \sum_{j=i+1}^{n-1} \alpha_{m,j} \leq \alpha_{m,n-1} \tag{12}$$

The reason for this requirement can be sketched as follows: if at any iteration stage i the current remaining rotation angle A_i is zero, it will be changed by $\pm\alpha_{m,i}$ in the next iteration. Then, the sum of rotation angles for the remaining iterations $\sum_{j=i+1}^{n-1} \alpha_{m,j}$ has to be large enough to bring the remaining angle after the last iteration A_n to zero within the accuracy given by $\alpha_{m,n-1}$.
2.) Second, the given rotation angle A_0 must not exceed the convergence range of the iteration which is given by the sum of all rotation angles plus the final angle

$$|A_0| \leq \sum_{i=0}^{n-1} \alpha_{m,i} + \alpha_{m,n-1}$$

Walther [2] has proposed shift sequences for each of the three coordinate systems for radix 2 implementation. It was shown by Walther [2] that for $m = -1$ the convergence criterion Eq. (12) is not satisfied for $\alpha_{-1,i} = \tanh^{-1}(2^{-i})$, but it is satisfied if the integers $(4, 13, 40, \ldots, k, 3k+1, \ldots)$ are repeated in the shift sequence. For any implementation, the angle sequence $\alpha_{m,i}$ resulting from the chosen shift sequence can be calculated in advance, quantized according to a chosen quantization scheme and retrieved from storage during execution of the CORDIC algorithm as shown in Fig. 24.4.

24.1.4 CORDIC Operation Modes and Functions

Using the CORDIC algorithm and the shift sequences stated above, a number of different functions can be calculated in rotation mode and vectoring mode as shown in Table 24.2.

coordinate system	shift sequence	convergence	scale factor
m	$s_{m,i}$	$\|A_0\|$	$K_m(n \to \infty)$
1	$0, 1, 2, 3, 4, \ldots, i, \ldots$	~ 1.74	~ 1.64676
0	$1, 2, 3, 4, 5. \ldots, i+1, \ldots$	1.0	1.0
-1	$1, 2, 3, 4, 4, 5, \ldots$	~ 1.13	~ 0.82816

Table 24.1 CORDIC shift sequences.

m	mode	initialization	output
1	rotation	$x_0 = x$	$x_n = K_1(n) \cdot (x \cos\theta - y \sin\theta)$
		$y_0 = y$	$y_n = K_1(n) \cdot (y \cos\theta + x \sin\theta)$
		$z_0 = \theta$	$z_n = 0$
		$x_0 = \frac{1}{K_1(n)}$	$x_n = \cos\theta$
		$y_0 = 0$	$y_n = \sin\theta$
		$z_0 = \theta$	$z_n = 0$
1	vectoring	$x_0 = x$	$x_n = K_1(n) \cdot \text{sign}(x_0) \cdot \sqrt{x^2 + y^2}$
		$y_0 = y$	$y_n = 0$
		$z_0 = \theta$	$z_n = \theta + \tan^{-1}(\frac{y}{x})$
0	rotation	$x_0 = x$	$x_n = x$
		$y_0 = y$	$y_n = y + x \cdot z$
		$z_0 = z$	$z_n = 0$
0	vectoring	$x_0 = x$	$x_n = x$
		$y_0 = y$	$y_n = 0$
		$z_0 = z$	$z_n = z + \frac{y}{x}$
- 1	rotation	$x_0 = x$	$x_n = K_{-1}(n) \cdot (x \cosh\theta + y \sinh\theta)$
		$y_0 = y$	$y_n = K_{-1}(n) \cdot (y \cosh\theta + x \sinh\theta)$
		$z_0 = \theta$	$z_n = 0$
		$x_0 = \frac{1}{K_{-1}(n)}$	$x_n = \cosh\theta$
		$y_0 = 0$	$y_n = \sinh\theta$
		$z_0 = \theta$	$z_n = 0$
- 1	vectoring	$x_0 = x$	$x_n = K_{-1}(n) \cdot \text{sign}(x_0) \cdot \sqrt{x^2 - y^2}$
		$y_0 = y$	$y_n = 0$
		$z_0 = \theta$	$z_n = \theta + \tanh^{-1}(\frac{y}{x})$

Table 24.2 Functions calculated by the CORDIC algorithm.

In addition, the following functions can be calculated from the immediate CORDIC outputs

$$\tan z = \frac{\sin z}{\cos z}$$

$$\begin{aligned}
\tanh z &= \frac{\sinh z}{\cosh z} \\
\exp z &= \sinh z + \cosh z \\
\ln z &= 2\tanh^{-1}(\frac{y}{x}) \quad \text{with } x = z+1 \text{ and } y = z-1 \\
\sqrt{z} &= \sqrt{x^2 - y^2} \quad \text{with } x = z + \frac{1}{4} \text{ and } y = z - \frac{1}{4}
\end{aligned}$$

24.2 Computational Accuracy

It was already mentioned that due to the n fixed rotation angles a given rotation angle A_0 can only be approximated, resulting in an angle approximation error $\Delta\phi \leq \alpha_{m,n-1}$. Even if all other error sources are neglected, the accuracy of the outputs of the nth iteration is hence principally limited by the magnitude of the last rotation angle $\alpha_{m,n-1}$. For large n, approximately $s_{m,n-1}$ accurate digits of the result are obtained since $s_{m,n-1}$ specifies the last right shift of the shift sequence. As a first guess the number of iterations should hence be chosen such that $s_{m,n-1} = W$ for a desired output accuracy of W bits. This leads to $n = W + 1$ iterations if the shift sequence given in Table 24.1 for $m = 1$ is used.

A second error source is given by the finite precision of the involved variables which has to be taken into account for fixed point as well as floating point implementations. The CORDIC algorithm as stated so far is suited for fixed point number representations. Some facts concerning the extension to floating point numbers will be presented in section 24.2.4. The format of the internal CORDIC variables is shown in Fig. 24.7. The internal wordlength is given by the wordlength W of the

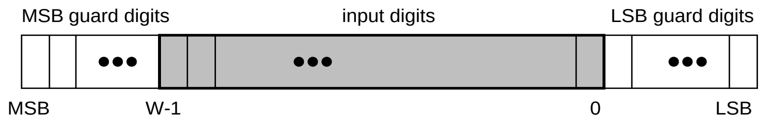

Figure 24.7 Format of the internal CORDIC variables.

fixed point input, enhanced by G guard bits on the most significant bit (MSB) and C guard bits at the least significant bit (LSB) side[1]. The MSB guard bits are necessary since for $m = 1$ the scale factor is larger than one, and since range extensions are introduced by the CORDIC rotation as obvious from the rotation trajectories given in Fig. 24.1, Fig. 24.2 and Fig. 24.3, respectively[2]. The successive right shifts employed in the CORDIC algorithm for the x_i and y_i variables, together with the limited number of bits in the internal fixed point datapath, require careful analysis of the involved computational accuracy and the inevitable rounding errors[3]. LSB

[1]In the CORDIC rotation mode and vectoring mode the iteration variables z_i and y_i are driven to zero during the n iterations. This fact can be exploited by either increasing the resolution for these variables during the iterations or by reducing their wordlength. Detailed descriptions for these techniques can be found in [5, 6] for the rotation mode and in [7] for the vectoring mode.

[2]For $m = 1$ two guard bits are necessary at the MSB side. First, the scale factor is larger than one: $K_1(n) \sim 1.64676$. Second a range extension by the factor $\sqrt{2}$ occurs for the x_n and y_n components.

[3]In order to avoid any rounding the number of LSB guard digits has to be equal to the sum of all elements in the shift sequence, which is of course not economically feasible.

guard digits are necessary in order to provide the desired output accuracy as will be discussed in section 24.2.3.

24.2.1 Number Representation

Below, we assume that the intermediate CORDIC iteration variables x_i and y_i are quantized according to a normalized (fractional) fixed point two's complement number representation. The value V of a number n with N binary digits $(n_{N-1}, \ldots, n_0)$ is given by

$$V = (-n_{N-1} + \sum_{j=0}^{N-2} n_j \cdot 2^{-(N-1)+j}) \cdot F \quad \text{with } F = 1 \tag{13}$$

This convenient notation can be changed to a two's complement integer representation simply by using $F = 2^{N-1}$ or to any other fixed point representation in a similar way, hence it does not pose any restriction on the input quantization of the CORDIC.
For $m = 1$ the different common formats for rotation angles have to be taken into account. If the angle is given in radians the format given in Eq. (13) may be chosen. F has to be adapted to $F = 2$ if the input range is $(-\pi/2, \pi/2) = (-1.57, 1.57)$. However, in many applications for the circular coordinate system it is desirable to represent the angle in fractions of π and to include the wrap around property which holds for the restricted range of possible angles $(-\pi, +\pi)$. If $F = \pi$ is chosen, the well known wrap around property of two's complement numbers ensures that all angles undergoing any addition/subtraction stay in the allowed range. This format was proposed by Daggett [8] and is sometimes referred to as "Daggett angle representation". The input angle range is often limited to $(-\pi/2, \pi/2)$ which guarantees convergence. The total domain of convergence can be easily expanded by including some "pre-rotations" for input vectors in the ranges $(\pi/2, \pi)$ and $(-\pi, -\pi/2)$. If the Daggett angle format is used it is very easy to determine the quadrant of a given rotation angle A_0 since only the upper two bits have to be inspected. Pre-rotations by $\pm\pi$ or $\pm\pi/2$ are very easy to implement. More sophisticated proposals for expanding the range of convergence of the CORDIC algorithm for all coordinate systems were given in [9].

24.2.2 Angle Approximation Error

It was shown already that the last rotation angle $\alpha_{m,n-1}$ determines the accuracy achievable by the CORDIC rotation. A straightforward conclusion is to increase the number of iterations n in order to improve accuracy since $\alpha_{m,n-1} = \frac{1}{\sqrt{m}} \cdot \tan^{-1}(\sqrt{m} \cdot 2^{-s_{m,n-1}})$ with $s_{m,i}$ being a nondecreasing integer shift sequence. However, the finite word length available for representing the intermediate variables poses some restrictions. If the angle is quantized according to Eq. (13) the value of the least significant digit is given by $2^{-W-C+1} \cdot F$. Therefore, $\alpha_{m,n-1} \geq 2^{-W-C+1} \cdot F$ must hold in order to represent this value. Additionally the rounding error, which increases with the number of iterations, has to be taken into account.

24.2.3 Rounding Error

Rounding is preferred vs. truncation in CORDIC implementations since the truncation of two's complement numbers creates a positive offset which is highly undesirable. Additionally, the maximum error due to rounding is only half as large as the error involved with truncation. The effort for the rounding procedure can be kept very small since the eventual addition of a binary one at the LSB position can be incorporated with the additions/subtractions which are necessary anyway.

Analysis of the rounding error for the z_i variable is easy since no shifts occur and the rounding error is just due to the quantization of the rotation angles. Hence, the rounding error is bounded by the accumulation of the absolute values of the rounding errors for the single quantized angles $\alpha_{m,i}$.

In contrast the propagation of initial rounding errors for x_i and y_i through the multiple iteration stages of the algorithm and the scaling involved with the single iterations make an analytical treatment very difficult. However, a thorough mathematical analysis taking into account this error propagation is given in [4] and extended in [10]. Due to limited space we present only a simplified treatment as can be found in [2]. Here the assumption is made that the maximum rounding errors associated with every iteration accumulate to an overall maximum rounding error for n iterations. While for z_i this is a valid assumption it is a coarse simplification for x_i and y_i. As shown in Fig. 24.7, $W + G + C$ bits are used for the internal variables with C additional fractional guard digits. Using the format given in Eq. (13) the maximum accumulated rounding error for n iterations is given by $e(n) = n \cdot F \cdot 2^{-(W+C-1)-1}$. If W accurate fractional digits of the result are to be obtained the resulting output resolution is $2^{-(W-1)} \cdot F$. Therefore, if C is chosen such that $e(n) \leq F \cdot 2^{-W}$, the rounding error can be considered to be of minor impact. From $n \cdot F \cdot 2^{-(W+C)} < F \cdot 2^{-W}$ it follows that $C \geq \log_2(n)$. Hence, at least $C = \lceil \log_2(n) \rceil$ additional fractional digits have to be provided.

24.2.4 Floating Point Implementations

The shift–and-add structure of the CORDIC algorithm is well suited for a fixed point implementation. The use of expensive floating point re–normalizations and floating point additions and subtractions in the CORDIC iterations does not lead to any advantage since the accuracy is still limited to the accuracy imposed by the fixed wordlength additions and subtractions which also have to be implemented in floating point adders/subtractors. Therefore, floating point CORDIC implementations usually contain an initial floating to fixed conversion. We consider here only the case $m = 1$ (detailed discussions of the floating point CORDIC can be found in [7, 11, 12, 13]). Below, it is assumed that the input floating point format contains a normalized signed mantissa m and an exponent e. Hence, the mantissa is a two's complement number[4] quantized according to Eq. (13) with $F = 1$. The components of the floating point input vector v are given by $v = (x, y)^T = (m_x \cdot 2^{e_x}, m_y \cdot 2^{e_y})^T$. The input conversion includes an alignment of the normalized signed floating point mantissas according to their exponents. Two approaches are known for the floating point CORDIC:

1. The CORDIC algorithm is started using the same shift sequences, rotation angles and number of iterations as for the fixed point CORDIC. We consider a floating point implementation of the vectoring mode.
   The CORDIC vectoring mode iteration written for floating point values x_i

[4] If the mantissa is given in sign–magnitude format it can be easily converted to a two's complement representation.

and y_i and $m = 1$ is given by

$$\begin{aligned} m_{x,i+1} \cdot 2^{e_x} &= m_{x,i} \cdot 2^{e_x} + \mu_i \cdot m_{y,i} \cdot 2^{e_y} \cdot 2^{-s_{m,i}} \\ m_{y,i+1} \cdot 2^{e_y} &= m_{y,i} \cdot 2^{e_y} - \mu_i \cdot m_{x,i} \cdot 2^{e_x} \cdot 2^{-s_{m,i}} \end{aligned} \tag{14}$$

Depending on the difference of the exponents $E = e_x - e_y$ two different approaches are used. If $E < 0$ we divide both equations 14 by 2^{e_y}

$$\begin{aligned} m_{x,i+1} \cdot 2^{e_x - e_y} &= m_{x,i} \cdot 2^{e_x - e_y} + \mu_i \cdot m_{y,i} \cdot 2^{-s_{m,i}} \\ m_{y,i+1} &= m_{y,i} - \mu_i \cdot m_{x,i} \cdot 2^{e_x - e_y} \cdot 2^{-s_{m,i}} \end{aligned} \tag{15}$$

Hence, we can simply set $y_0 = m_{y,0}$ and $x_0 = m_{x,0} \cdot 2^{e_x - e_y}$ and then perform the usual CORDIC iteration for the resulting fixed point two's complement inputs x_0 and y_0. Of course, some accuracy for the x_i variable is lost due to the right shift $2^{e_x - e_y}$. For $E \geq 0$ we could proceed completely accordingly and divide the equations by 2^{e_x}. Then the initial conversion represents just an alignment of the two floating point inputs according to their exponents. This approach was proposed in [11].

Alternatively, if $E \geq 0$ the two equations (14) are divided by 2^{e_x} and 2^{e_y}, respectively, as described already in [2]

$$\begin{aligned} m_{x,i+1} &= m_{x,i} + m \cdot \mu_i \cdot m_{y,i} \cdot 2^{-(s_{m,i} + E)} \\ m_{y,i+1} &= m_{y,i} - \mu_i \cdot m_{x,i} \cdot 2^{-(s_{m,i} - E)} \end{aligned} \tag{16}$$

Then, the usual CORDIC iteration is performed with two's complement fixed point inputs, but starting with an advanced iteration k: $x_k = m_{x,0}$ and $y_k = m_{y,0}$. Usually, only right shifts occur in the CORDIC algorithm. A sequence of left shifts would lead to an exploding number of digits in the internal representation since all significant digits have to be taken into account on the MSB side in order to avoid overflows. Therefore, the iteration k with $s_{m,k} = E$ is taken as the starting iteration. With $s_{m,k} = E$ all actually applied shifts remain right shifts for the n iterations: $i \in \{k, \ldots, k+n-1\}$. Hence, the index k is chosen such that optimum use is made of the inherent fixed point resolution of the y_i variable whose value is driven to zero. Unfortunately, the varying index of the start iteration leads to a variable scale factor as will be discussed later.

The CORDIC angle sequence $\alpha_{m,i}$ has also to be represented in a fixed point format in order to be used in a fixed point implementation. If the algorithm is always started with iteration $i = 0$ the angle sequence can be quantized such that optimum use is made of the range given by the fixed wordlength. If we start with an advanced iteration with variable index k the angle sequence has to be represented such that for the starting angle $\alpha_{m,k}$ no leading zeroes occur. Consequently, the angle sequence has to be stored with an increased resolution (i.e. with an increased number of bits) and to be shifted according to the value of k in order to provide the full resolution for the following n iterations.

So far we discussed the first approach to floating point CORDIC only for the vectoring mode. For the rotation mode similar results can be obtained (c.f. [2]). To summarize, several drawbacks are involved for $E \geq 0$:

(a) The scale factor is depending on the starting iteration k

$$K_m(n,k) = \prod_{j=k}^{n-1+k} K_{m,j} \tag{17}$$

Therefore the inverse scale factor as necessary for final scale factor correction has e.g. to be calculated in parallel to the iterations or storage has to be provided for a number of precalculated different inverse scale factors.

(b) The accuracy of the stored angle sequence has to be such that sufficient resolution is given for all possible values of k. Hence, the number of digits necessary for representing the angle sequence becomes quite large.

(c) If the algorithm is started with an advanced iteration k with $s_{m,k} = E$, the resulting right shifts given by $s_{m,i} + E$ for the x_i variable lead to increased rounding errors for a given fixed wordlength.

Nevertheless, this approach was proposed for a number of applications (c.f. [11, 14, 15]).

2. For full IEEE 754 floating point accuracy a floating point CORDIC algorithm was derived in [12, 13]. Depending on the difference of the exponents of the components of the input vector and on the desired angle resolution an optimized shift sequence is selected here from a set of predefined shift sequences. For a detailed discussion the reader is referred to [12, 13].

Following the fixed point CORDIC iterations the output numbers are re–converted to floating–point format. The whole approach with input–output conversions and internal fixed point datapath is called block floating–point [15, 12, 13].

24.3 Scale Factor Correction

At first glance, the vector scaling introduced by the CORDIC algorithm does not seem to pose a significant problem. However, the correction of a possibly variable scale factor for the output vector generally requires two divisions or at least two multiplications with the corresponding reciprocal values. Using a fixed–point number representation a multiplication can be realized by W shift and add operations where W denotes the wordlength. Now, the CORDIC algorithm itself requires on the order of W iterations in order to generate a fixed–point result with W bits accuracy as discussed in section 24.2. Therefore, correction of a variable scale factor requires an effort comparable to the whole CORDIC algorithm itself. Fortunately, the restriction of the possible values for μ_i to $(-1, 1)$ ($\mu_i \neq 0$) leads to a constant scale factor $K_m(n)$ for each of the three coordinate systems m and a fixed number of iterations n as given in Eq. (6).

A constant scale factor which can be interpreted as a fixed (hence not data dependent) gain can be tolerated in many digital signal processing applications[5]. Hence it should be carefully investigated whether it is necessary to compensate for the scaling at all.

[5]The drawback is that a certain unused headroom is introduced for the output values since the scale factor is not a power of two.

If scale factor correction can not be avoided, two possibilities are known: performing a constant factor multiplication with $\frac{1}{K_m(n)}$ or extending the CORDIC iteration in a way that the resulting inverse of the scale factor takes a value such that the multiplication can be performed using a few simple shift and add operations.

24.3.1 Constant Factor Multiplication

Since $\frac{1}{K_m(n)}$ can be computed in advance the well known *multiplier recoding* methods [16] can be applied. The effort for a constant factor multiplication is dependent on the number of nonzero digits necessary to represent the constant factor, resulting in a corresponding number of shift and add operations[6]. Hence, the goal is to reduce the number of nonzero digits in $\frac{1}{K_m(n)}$ by introducing a *canonical signed digit* [17] representation with digits $s_j \in \{-1, 0, 1\}$ and recoding the resulting number

$$\frac{1}{K_m(n)} = \sum_{j=0}^{W-1} s_j 2^{-j}$$

On the average, the number of nonzero digits can be reduced to $\frac{W}{3}$ [16], hence the effort for the constant multiplication should be counted as approximately one third the effort for a general purpose multiplication only.

24.3.2 Extended CORDIC Iterations

By extending the sequence of CORDIC iterations the inverse of the scale factor may eventually become a "simple" number (i.e. a power of two, the sum of powers of two or the sum of terms like (1 ± 2^{-j}), so that the scale factor correction can be implemented using a few simple shift and add operations. The important fact is that when changing the CORDIC sequence, convergence still has to be guaranteed and the shift sequence as well as the set of elementary angles $\alpha_{m,i}$ both change. Four approaches are known for extending the CORDIC algorithm:

1. Repeated Iterations

Single iterations may be repeated without destroying the convergence properties of the CORDIC algorithm [18] which is obvious from Eq. (12). Hence, a set of repeated iterations can be defined which leads to a simple scale factor. However, using this simple scheme the number of additional iterations is quite large reducing the overall savings due to the simple scale factor correction[19].

2. Normalization Steps

In [20] the inverse of the scale factor is described as

$$\frac{1}{K_m(n)} = \prod_{i=0}^{n-1} (1 - m \cdot \gamma_{m,i} \cdot 2^{-s_{m,i}})$$

with $\gamma_{m,i} \in \{0, 1\}$. The single factors in this product can be implemented by introducing normalization steps into the CORDIC sequence

$$\begin{aligned} x_{i+1,norm} &= x_{i+1} - m \cdot x_{i+1} \cdot \gamma_{m,i} \cdot 2^{-s_{m,i}} \\ y_{i+1,norm} &= y_{i+1} - m \cdot y_{i+1} \cdot \gamma_{m,i} \cdot 2^{-s_{m,i}} \end{aligned} \qquad (18)$$

[6]In parallel multiplier architectures the shifts are hardwired, while in a serial multiplier the multiplication is realized by a number of successive shift and add operations.

The important fact is that these normalization steps can be implemented with essentially the same hardware as the usual iterations since the same shifts $s_{m,i}$ are required and steered adders/subtractors are necessary anyway. No change in the convergence properties takes place since the normalization steps are pure scaling operations and do not involve any rotation.

3. Double shift iterations

A different way to achieve a simple scale factor is to modify the sequence of elementary rotation angles by introducing *double shift iterations* as proposed in [21]

$$\begin{aligned} x_{i+1} &= x_i - m \cdot \mu_i \cdot y_i \cdot 2^{-s_{m,i}} - m \cdot \mu_i \cdot y_i \cdot \eta_{m,i} \cdot 2^{-s'_{m,i}} \\ y_{i+1} &= y_i + \mu_i \cdot x_i \cdot 2^{-s_{m,i}} + \mu_i \cdot x_i \cdot \eta_{m,i} \cdot 2^{-s'_{m,i}} \end{aligned} \qquad (19)$$

where $\eta_{m,i} \in \{-1, 0, 1\}$ and $s'_{m,i} > s_{m,i}$. For $\eta_{m,i} = 0$ the usual iteration equations are obtained. The set of elementary rotation angles is now given by

$$\alpha_{m,i} = \frac{1}{\sqrt{m}} \cdot \tan^{-1}(\sqrt{m} \cdot (2^{-s_{m,i}} + \eta_{m,i} \cdot 2^{-s'_{m,i}}))$$

The problem of finding shift sequences $s'_{m,i}$ and $s_{m,i}$ which guarantee convergence, lead to a simple scale factor and simultaneously represent minimum extra hardware effort was solved in [22, 15, 23].

4. Compensated CORDIC Iteration

A third solution leading to a simple scale factor was proposed in [19] based on [24]

$$\begin{aligned} x_{i+1} &= x_i - m \cdot \mu_i \cdot y_i \cdot 2^{-s_{m,i}} + x_i \cdot \eta_{m,i} \cdot 2^{-s_{m,i}} \\ y_{i+1} &= y_i + \mu_i \cdot x_i \cdot 2^{-s_{m,i}} + y_i \cdot \eta_{m,i} \cdot 2^{-s_{m,i}} \end{aligned}$$

The advantage is that the complete subexpressions $x_i \cdot 2^{-s_{m,i}}$ and $y_i \cdot 2^{-s_{m,i}}$ occur twice in the iteration equations and hence need to be calculated only once. The set of elementary angles is here described by

$$\alpha_{m,i} = \frac{1}{\sqrt{m}} \cdot \tan^{-1}(\frac{\sqrt{m}}{2^{s_{m,i}} + \eta_{m,i}})$$

A comparison of the schemes 1.-4. in terms of hardware efficiency is outside the scope of this chapter. It should be mentioned, however, that the impact of the extra operations depends on the given application, the desired accuracy and the given wordlength. Additionally, recursive CORDIC architectures pose different constraints on the implementation of the extended iterations than pipelined unfolded architectures. A comparison of the schemes for a recursive implementation with an output accuracy of 16 bits can be found in [19].

24.4 CORDIC Architectures

In this section several CORDIC architectures are presented. We start with the dependence graph for the CORDIC which shows the operational flow in the algorithm. Note that we restrict ourselves to the conventional CORDIC iteration scheme. The dependence graph for extended CORDIC iterations can be easily derived based on the results. The nodes in the dependence graph represent operations (here: steered additions/subtractions and shifts $2^{-s_{m,i}}$) and the arcs represent the flow of intermediate variables. Note that the dependence graph does not include any timing information, it is just a graphical representation of the algorithmic flow. The dependence graph is transformed into a signal flow graph by introducing a suitable projection and a time axis (c.f. [25]). The timed signal flow graph represents a register–transfer level (RTL) architecture. Recursive and pipelined architectures will be derived from the CORDIC dependence graph in the following.

The dependence graph for a merged implementation of rotation mode and vectoring mode is shown in Fig. 24.8. The only difference for the two CORDIC modes is the way the control flags are generated for steering the adders/subtractors. The signs of all three intermediate variables are fed into a control unit which generates the control flags for the steered adders/subtractors given the used coordinate system m and a flag indicating which mode is to be applied.

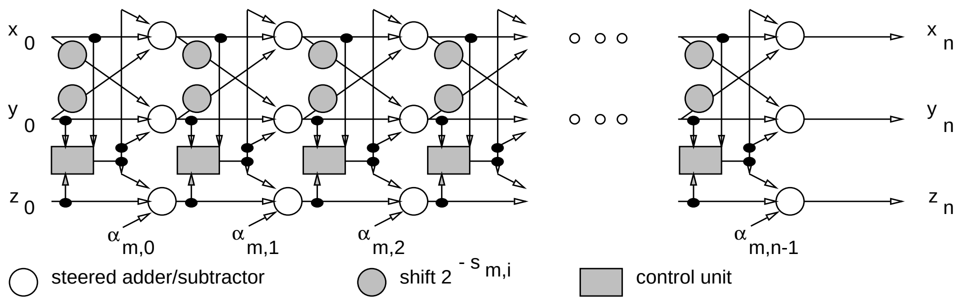

Figure 24.8 CORDIC dependence graph for rotation mode and vectoring mode.

In a one–to–one projection of the dependence graph every node is implemented by a dedicated unit in the resulting signal flow graph. In Fig. 24.9, the signal flow graph for this projection is shown together with the timing for the cascaded additions/subtractions (the fixed shifts are assumed to be hard–wired, hence they do not represent any propagation delay). Besides having a purely combinatorial implementation, pipeline registers can be introduced between successive stages as indicated in Fig. 24.9.

In the following we characterize three different CORDIC architectures by their clock period T_{Clock}, throughput in rotations per second and latency in clock cycles. The delay for calculating the rotation direction μ_i is neglected due to the simplicity of this operation, as well as flipflop setup and hold times. As shown in Fig. 24.9 every addition/subtraction involves a carry propagation from least significant bit (LSB) to most significant bit (MSB) if conventional number systems are used. The length of this ripple path is a function of the wordlength W, e.g. $T_{Add} \sim W$ holds for a Carry–Ripple addition. The sign of the calculated sum or difference is known only after computation of the MSB. Therefore, the clock period for the unfolded architecture without pipelining is given by $n \cdot T_{Add}$ as shown in Fig. 24.9. The throughput is equal to $\frac{1}{n \cdot T_{add}}$ rotations/s. The pipelined version has a latency of n clock cycles and a clock period $T_{Clock} = T_{add}$. The throughput is $\frac{1}{T_{add}}$ rotations/s.

It is obvious that the dependence graph in Fig. 24.8 can alternatively be projected in horizontal direction onto a recursive signal flow graph. Here, the

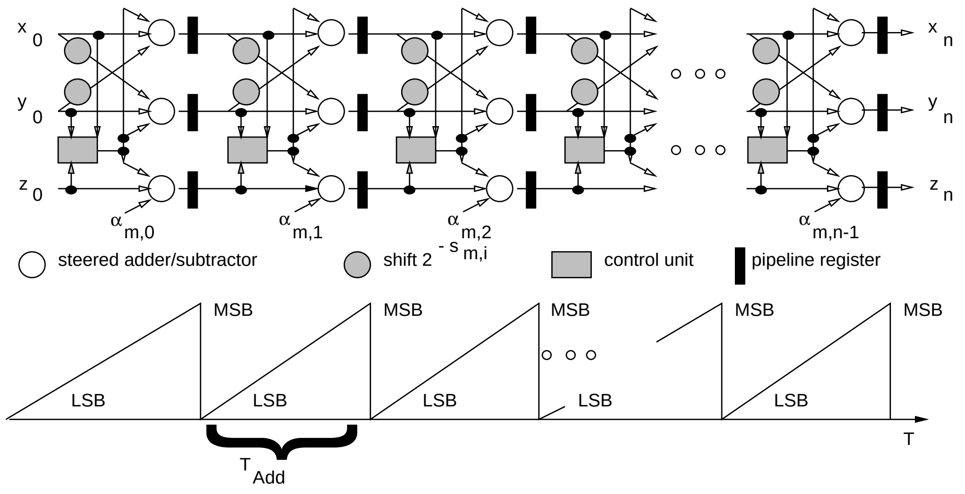

Figure 24.9 Unfolded (pipelined) CORDIC signal flow graph.

successive operations are implemented sequentially on a recursive shared processing element as shown in Fig. 24.10.

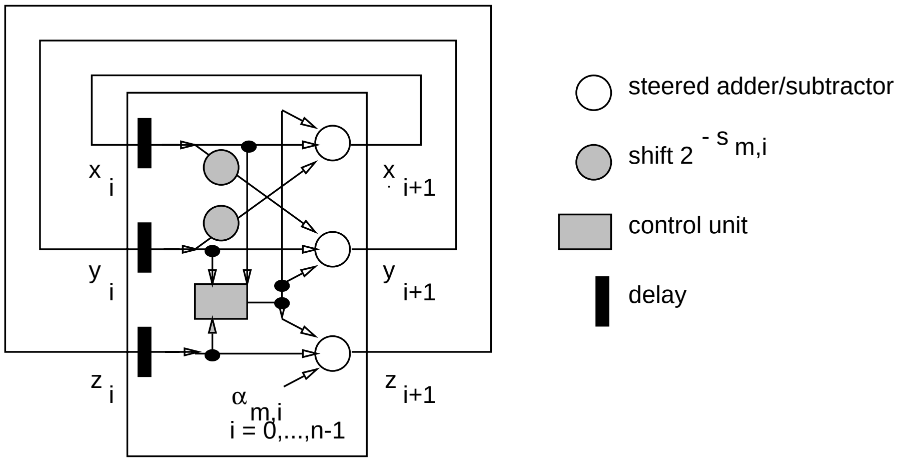

Figure 24.10 Folded (recursive) CORDIC signal flow graph.

Note that due to the necessity to implement a number of different shifts according to the chosen shift sequence, variable shifters (i.e. so called barrel shifters) have to be used in the recursive processing element. The propagation delay associated with the variable shifters is comparable to the adders, hence the clock period is given by $T_{Clock} = T_{Add} + T_{Shift}$. The total latency for n recursive iterations is given by n clock cycles and the throughput is given by $\frac{1}{n \cdot (T_{Add}+T_{Shift})}$ since new input data can be processed only every n clock cycles.

The properties of the three architectures are summarized in Table 24.3.

Architecture	Clock period	$\frac{\text{Throughput}}{\text{rotations/s}}$	$\frac{\text{Latency}}{\text{cycles}}$	Area
unfolded	$n \cdot T_{add}$	$\frac{1}{n \cdot T_{add}}$	1	$3n$add, 1reg
unfolded pipelined	T_{add}	$\frac{1}{T_{add}}$	n	$3n$add, $3n$regs
folded recursive	$T_{add} + T_{shift}$	$\frac{1}{n \cdot (T_{add} + T_{Shift})}$	n	3add, $3 + n$regs 2shifters

Table 24.3 Architectural properties for three CORDIC architectures.

24.4.1 Programmable CORDIC processing element

The variety of functions calculated by the CORDIC algorithm leads to the idea of proposing a programmable CORDIC processing element (PE) for digital signal processing applications, e.g. as an extension to existing arithmetic units in digital signal processors (DSPs) [18, 19]. The folded sequential architecture presented in Fig. 24.10 is the most attractive architecture for a CORDIC PE due to its low complexity. In this section, we give an overview of the structure and features of such a PE.

Standard DSPs contain MAC (Multiply–Accumulate) units which enable single cycle parallel multiply–accumulate operations. Functions can be evaluated using table–lookup methods or using iterative algorithms (e.g. Newton–Raphson iterations [17]) which can efficiently be executed using the standard MAC unit. Since the MAC unit performs single cycle multiplication and addition the multiplication realized with the linear CORDIC mode ($m = 0$) can not compete due to the sign–directed, sequential nature of the CORDIC algorithm which requires a number of clock cycles for multiplication. In contrast all functions calculated in the circular and hyperbolic modes compare favorably to the respective implementations on DSPs as shown in [19]. Therefore, a CORDIC PE extension for $m = 1$ and $m = -1$ to standard DSPs seems to be the most attractive possibility. It is desirable that the scale factor correction takes place inside the CORDIC unit since otherwise additional multiplications or divisions are necessary in order to correct for the scaling. As was already pointed out in section 24.3, several methods for scale factor correction are known. As an example, a CORDIC PE using the double shift iteration method is shown in Fig. 24.11. Here, the basic CORDIC iteration structure as shown in Fig. 24.4 was enhanced in order to facilitate the double shift iterations. The double shift iterations Eq. (19) with $s'_{m,i} \neq 0$ are implemented in two successive clock cycles[7]. For iterations with $s'_{m,i} = 0$ the shifters are steered to shift by $s_{m,i}$ and the part of the datapath drawn in dashed lines is not used. For double shift iterations with $s'_{m,i} \neq 0$ the result obtained after the first clock cycle is registered in the dashed registers and multiplexed into the datapath during the

[7]Alternatively, an architecture executing one double shift iteration per clock cycle is possible. However, this requires additional hardware. If the number of iterations with $s'_{m,i} \neq 0$ is small, the utilization of the additional hardware is poor.

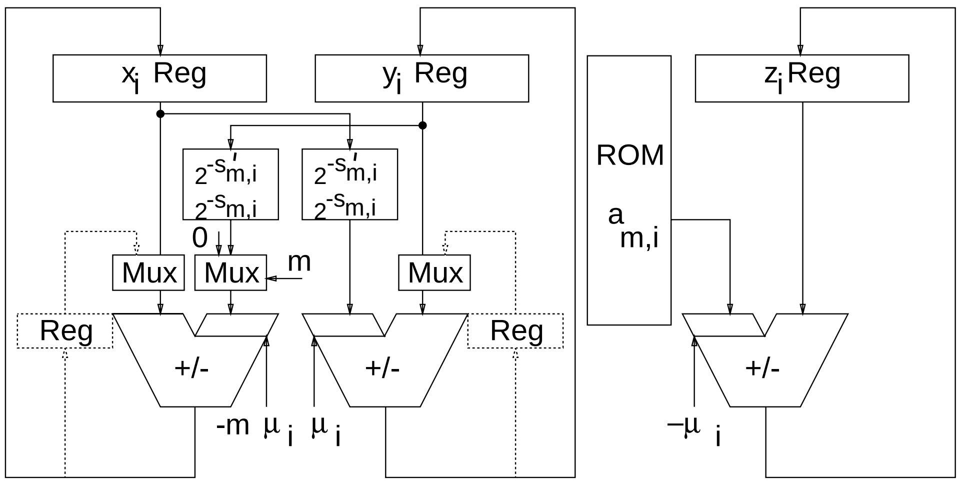

Figure 24.11 Programmable CORDIC processing element.

second clock cycle while the other registers are stalled. During this second clock cycle the shifters are steered to shift by $s'_{m,i}$ and the final result is obtained as given in Eq. (19).

The sets of elementary rotation angles to be used can be stored in a ROM as shown in Fig. 24.11 or in register files (arrays of registers), if the number of angles is reasonably small. A control unit steers the operations in the datapath according to the chosen coordinate system m and the chosen shift sequence. In addition, the CORDIC unit may be enhanced by a floating to fixed and fixed to floating conversion if the floating point data format is used. Implementations of programmable recursive CORDIC processing units were reported in [26, 19, 20].

24.4.2 Pipelined CORDIC architectures

In contrast to a universal CORDIC processing element the dominating motivation for a pipelined architecture is a high throughput constraint. Additionally, it is advantageous if relatively long streams of data have to be processed in the same manner since it takes a number of clock cycles to fill the pipeline and also to flush the pipeline if e.g. the control flow changes (a different function is to be calculated). Although pipeline registers are usually inserted in between the single CORDIC iterations as shown in Fig. 24.9 they can principally be placed everywhere since the unfolded algorithm is purely feedforward. A formalism to introduce pipelining is given by the well known cut–set retiming method [27, 25].
The main advantage of pipelined CORDIC architectures compared to recursive implementations is the possibility to implement hard–wired shifts rather than area and time consuming barrel shifters. However, the shifts can be hard–wired only for a single fixed shift sequence. Nevertheless, a small number of different shifts can be implemented using multiplexors which are still much faster and less area consuming than barrel shifters as necessary for the folded recursive architecture.
A similar consideration holds for the rotation angles. If only a single shift sequence is implemented the angles can be hard–wired into the adders/subtractors. A small

number of alternative rotation angles per stage can be implemented using a small combinatorial logic steering the selection of a particular rotation angle. ROMs or register files as necessary for the recursive CORDIC architecture are not necessary. Implementations of pipelined CORDICs are described in [21, 22, 15].

24.4.3 CORDIC Architectures for Vector Rotation

It was already noted that the CORDIC implementation of multiplication and division ($m = 0$) is not competitive. We further restrict consideration here to the circular mode $m = 1$ since much more applications exist than for the hyperbolic mode ($m = -1$).

Traditionally, vector rotations are realized as shown by the dependence graph given in Fig. 24.12. The sine and cosine values are generated by some table-lookup method (or another function evaluation approach) and the multiplications and additions are implemented using the corresponding arithmetic units as shown in Fig. 24.12. Below, we consider high throughput applications with one rotation per clock cycle, and low throughput applications, where several clock cycles are available per rotation.
High throughput applications: For high throughput applications, a one–to–one mapping of the dependence graph in Fig. 24.12 to a possibly pipelined signal flow graph is used. While only requiring a few multiplications and additions, the main drawback of this approach is the necessity to provide the sine and cosine values. A table-lookup may be implemented using ROMs or combinatorial logic. Since one ROM access is necessary per rotation, the throughput is limited by the access time of the ROMs. The throughput can not be increased beyond that point by pipelining. If even higher throughputs are needed, the ROMs have e.g. to be doubled and accessed alternatingly every other clock cycle. If on the other hand combinatorial logic is used for calculation of the sine and cosine values, pipelining is possible in principle. However, the cost for the pipelining can be very high due to the low regularity of the combinatorial logic which typically leads to a very high pipeline register count. As shown in section 24.4.2, it is easily and efficiently possible to

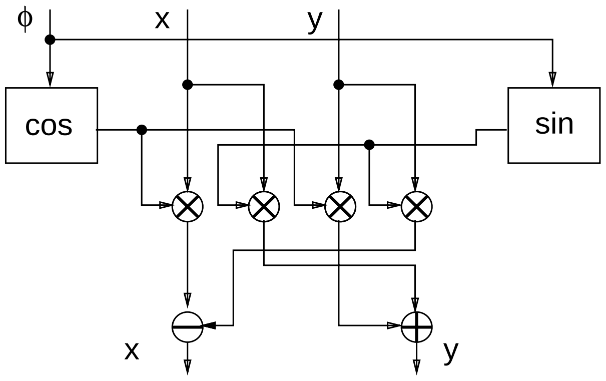

Figure 24.12 Dependence graph for the classical vector rotation.

pipeline the CORDIC in rotation mode. The resulting architectures provide very high throughput vector rotations. Additionally, the effort for a CORDIC pipeline

grows only linearly with the wordlength W and the number of stages n, hence about quadratically with the wordlength if $n = W + 1$ is used. In contrast the effort to implement the sine and cosine tables as necessary for the classical method grows exponentially with the required angle resolution or wordlength. Hence there is a distinct advantage in terms of throughput and implementation complexity for the CORDIC at least for relatively large wordlengths. Due to the n pipelining stages in the CORDIC the classical solution can be advantageous in terms of latency.
Low throughput applications:: A single resource shared multiplier and adder is sufficient to implement the classical method in several clock cycles as given for low throughput applications. However, at least one table shared for sine and cosine calculation is still necessary, occupying in the order of $(2^W - 1) * W$ bits of memory for a required wordlength of W bits for the sine and cosine values and the angle Φ[8]. In contrast, a folded sequential CORDIC architecture can be implemented using three adders, two barrel shifters and three registers as shown in Table 24.3. If $n = W + 1$ iterations are used, the storage for the n rotation angles amounts to $(W + 1) * W$ bits only. Therefore, the CORDIC algorithm is highly competetive in terms of area consumption for low throughput applications.

The CORDIC vectoring mode can be used for fast and efficient computation of magnitude and phase of a given input vector. In many cases, only the phase of a given input vector is required, which can of course be implemented using a table–lookup solution. However, the same drawbacks as already mentioned for the sine and cosine tables hold in terms of area consumption and throughput, hence the CORDIC vectoring mode represents an attractive alternative.

Other interesting examples for dedicated CORDIC architectures include sine and cosine calculation [6, 28], Fourier Transform processing [24], Chirp Z–transform implementation [29] and adaptive lattice filtering [30].

24.5 CORDIC Architectures using Redundant Number Systems

In conventional number systems, every addition or subtraction involves a carry propagation. Independent of the adder architecture the delay of the resulting carry ripple path is always a function of the wordlength. Redundant number systems offer the opportunity to implement carry-free or limited carry propagation addition and subtraction with a small delay independent of the used wordlength. Therefore they are very attractive for VLSI implementation. Redundant number systems have been in use for a long time e.g. in advanced parallel multiplier architectures (Booth, Carry–Save array and Wallace tree multipliers [16]). However, redundant number systems offer implementation advantages for many applications containing cascaded arithmetic computations. Recent applications for dedicated VLSI architectures employing redundant number systems include finite impulse response filter (FIR) architectures [31], cryptography [32] and the CORDIC algorithm. Since the CORDIC algorithm consists of a sequence of additions/subtractions the use of redundant number systems seems to be highly attractive. The main obstacle is given by the sign directed nature of the CORDIC algorithm. As will be shown below, the calculation of the sign of a redundant number is quite complicated in absolute contrast to conventional number systems where only the most significant bit has to be

[8]Symmetry of the sine and cosine functions can be exploited in order to reduce the table input wordlength by two or three bits.

inspected. Nevertheless, several approaches were derived recently for the CORDIC algorithm. A brief overview of the basic ideas is given.

24.5.1 Redundant Number Systems

A unified description for redundant number systems was given by Parhami [33] who defined *Generalized Signed Digit* (GSD) number systems. A GSD number system contains the digit set $\{-\alpha, -\alpha{+}1, \ldots, \beta{-}1, \beta\}$ with $\alpha, \beta \geq 0$, and $\alpha{+}\beta{+}1 > r$ with r being the radix of the number system. Every suitable definition of α and β leads to a different redundant number system. The value X of a W digit integer GSD number is given by:

$$X = \sum_{k=0}^{W-1} r^k x_k \quad , \quad x_k \in \{-\alpha, -\alpha+1, \ldots, \beta-1, \beta\} \tag{20}$$

An important subclass are number systems with $\alpha + \beta = r$, which are called "minimal redundant", since $\alpha + \beta = r - 1$ corresponds already to a conventional number system. The well known *Carry–Save* (CS) number system is defined by $\alpha = 0, \beta = 2, r = 2$. CS numbers are very attractive for VLSI implementation since the basic building block for arithmetic operations is a simple full adder[9].

In order to represent the CS digits two bits are necessary which are called c_i and s_i. The two vectors C and S given by c_i and s_i can be considered to be two two's complement numbers (or binary numbers, if only unsigned values occur). All rules for two's complement arithmetic (e.g. sign extension) apply to the C and the S number.

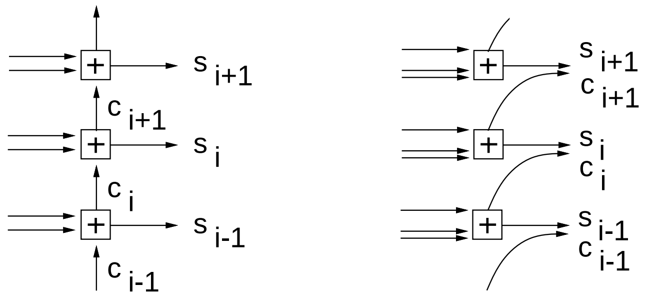

Figure 24.13 Carry ripple addition (left hand side) and 3–2 Carry Save addition (right hand side).

An important advantage of CS numbers is the very simple and fast implementation of the addition operation. In Fig. 24.13, a two's complement carry ripple addition and a CS addition is shown. Both architectures consist of W full adders for a wordlength of W digits. The carry ripple adder exhibits a delay corresponding to W full adder carry propagations while the delay of the CS adder is equal to a single

[9]With $\alpha = 1, \beta = 1, r = 2$ the well known *Binary Signed Digit* (BSD) number system results. BSD operations can be implemented using the same basic structures as for CS operations. The full adders used for CS implementation are replaced with "Generalized Full Adders" [32] which are full adders with in part inverting inputs and outputs.

full adder propagation delay and independent of the wordlength. The CS adder is called 3–2 adder since 3 input bits are compressed to 2 output bits for every digit position. This adder can be used to add a CS number represented by two input bits for every digit position and a usual two's complement number. Addition of two CS numbers is implemented using a 4–2 adder as shown in Fig. 24.14. CS subtraction is implemented by negation of the two two's complement numbers C and S in the minuend and addition as for two's complement numbers. It is well known that due to the redundant number representation *pseudo overflows* [34, 35] can occur. A correction of these pseudo overflows can be implemented using a modified full adder cell in the most significant digit (MSD) position. For a detailed explanation the reader is referred to [34, 35].

Conversion from CS to two's complement numbers is achieved using a so called Vector–Merging adder (VMA) [35]. This is a conventional adder adding the C and the S part of the CS number and generating a two's complement number. Since this conversion is very time consuming compared to the fast CS additions it is highly desirable to concatenate as many CS additions as possible before converting to two's complement representation.

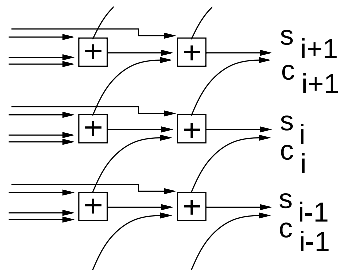

Figure 24.14 Addition of two Carry Save numbers (4–2 Carry Save addition).

The CORDIC algorithm consists of a sequence of additions/subtractions and sign calculations. In Fig. 24.15, three possibilities for an addition followed by a sign calculation are shown. On the left hand side a ripple adder with sign calculation is depicted. Determination of the sign of a CS number can be solved by converting the CS number to two's complement representation and taking the sign from the MSD, which is shown in the middle of Fig. 24.15. For this conversion a conventional adder with some kind of carry propagation from least significant digit (LSD) to MSD is necessary. Alternatively, the sign of a CS number can be determined starting with the MSD. If the C and the S number have the same sign, this sign represents the sign of the CS number. Otherwise succssive significant digits have to be inspected. The number of digits which have to be inspected until a definite decision on the sign is possible is dependent on the difference in magnitude of the C and the S number. The corresponding circuit structure is shown on the right hand side of Fig. 24.15. Since in the worst case all digits have to be inspected a combinatorial path exists from MSD to LSD.

To summarize, a LSD first and a MSD first solution exists for sign calculation for CS numbers. Both solutions lead to a ripple path whose length is dependent

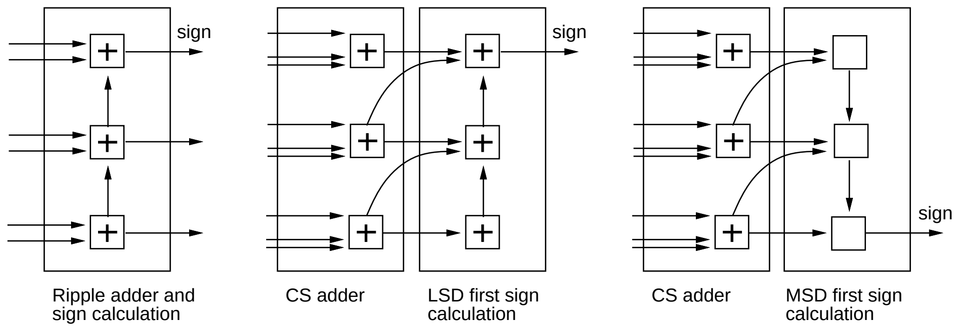
sign
sign
sign
Ripple adder and sign calculation
CS adder
LSD first sign calculation
CS adder
MSD first sign calculation

Figure 24.15 Addition and sign calculation using a ripple adder (left hand side), CS adder and LSD first (middle) as well as MSD first (right hand side) sign calculation.

on the wordlength. Addition and sign calculation using the two's complement representation requires less delay and less area. Therefore it seems that redundant arithmetic can not be applied advantageously to the CORDIC iteration.

24.5.2 Redundant CORDIC Architectures

In order to overcome the problem to determine the sign of the intermediate variables in CORDIC for redundant number systems several authors proposed techniques based on an estimation of the sign of the redundant intermediate results from a number of MSDs using a particular selection function ([7, 36, 28, 5]) for the circular coordinate system. If a small number of MSDs is used for sign estimation, the selection function can be implemented with a very small combinatorial delay. It was shown in the last section that in some cases it is possible to even *determine* the sign from a number of MSDs but in other cases not. The proposed algorithms differ in the treatment of the case that the sign can not be determined.

In [7] a redundant method for the vectoring mode is described. It is proposed not to perform the subsequent microrotation at all if the sign and hence the rotation direction can not be determined from the given number of MSDs. This is equivalent to expanding the range for the sign sequence μ_i from $\mu_i \in \{-1, 1\}$ to $\mu_i \in \{-1, 0, 1\}$. It is proved in [7] that convergence is still guaranteed. Recall that the total scale factor is given by the product of the scale factors involved with the single iterations. With $\mu_i \in \{-1, 0, 1\}$ the scale factor is variable

$$K_m(n) = \prod_{i=0,\mu_i \neq 0}^{n-1} K_{m,i} \tag{21}$$

The variable scale factor 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 for the rotation mode ([35] –[5]) describe techniques to avoid a variable scale factor. As proven in [6] the position of the significant digits of z_i in rotation mode changes by one digit per iteration since the magnitude of z_i decreases during the CORDIC iterations. In all following figures this is taken into account by a left shift of one digit for the intermediate variables z_i following each iteration. Then, the MSDs can always be taken for sign estimation.

Using the *double rotation method* proposed in [6, 5] every iteration or (micro–)rotation is represented by two subrotations. A negative, positive or non–rotation is implemented by combining two negative, positive or a positive and a negative subrotation, respectively. The sign estimation is based on the three MSDs of the redundant intermediate variable z_i. The range for μ_i is still given by $\{-1, 0, 1\}$. Since nevertheless exactly two subrotations are performed per iteration the scale factor is constant. An increase of about 50 percent in the arithmetic complexity of the single iterations has to be taken into account due to the double rotations. In the

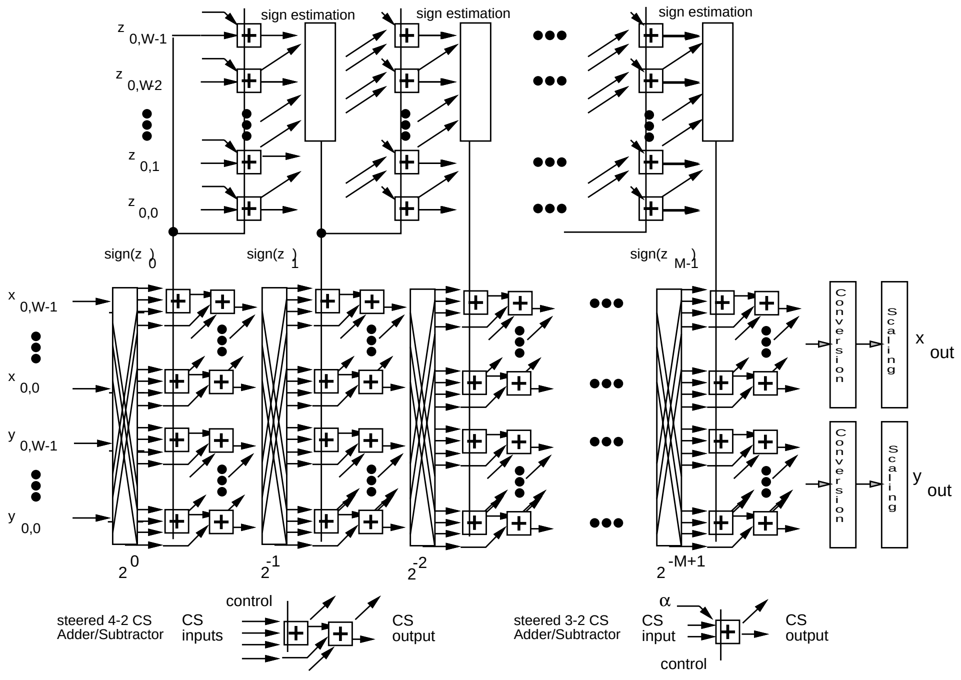

Figure 24.16 Parallel Architecture for the CORDIC Rotation Mode with sign estimation and $M > n$ iterations.

correcting iteration method presented in [6, 5] the case $\mu_i = 0$ is not allowed. Even if the sign can not be determined from the MSDs, a rotation is performed. Correcting iterations have to be introduced every m iterations. A worst case number of $m + 2$ MSDs (c.f. [6]) have to be inspected for sign estimation in the single iterations. In Fig. 24.16 an implementation for the rotation mode with sign estimation is shown with an input wordlength W and a number of M iterations with $M > n$ and n being the usual number of iterations. This method was extended to the vectoring mode in [37]. In [38], 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 [38] 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 a constant scale factor redundant implementation without additional or branching iterations was presented in [39, 40, 41], the *Differential CORDIC Algorithm* (DCORDIC). It was already mentioned (see eq. 8) that the rotation direction in CORDIC is chosen always such that the remaining rotation angle $|A_{i+1}| = ||A_i| - \alpha_{m,i}|$ eventually decreases. This equation can directly be implemented for the z_i variable in rotation mode and the y_i variable in vectoring mode. The important observation is that using redundant number systems an MSD first implementation of the involved operations addition/subtraction and absolute value calculation is possible without any kind of word–level carry propagation. Hence, successive iterations run concurrently with only a small propagation delay. As an example, the algorithm for the DCORDIC rotation mode is stated below. Since only absolute values are considered in Eq. (8)

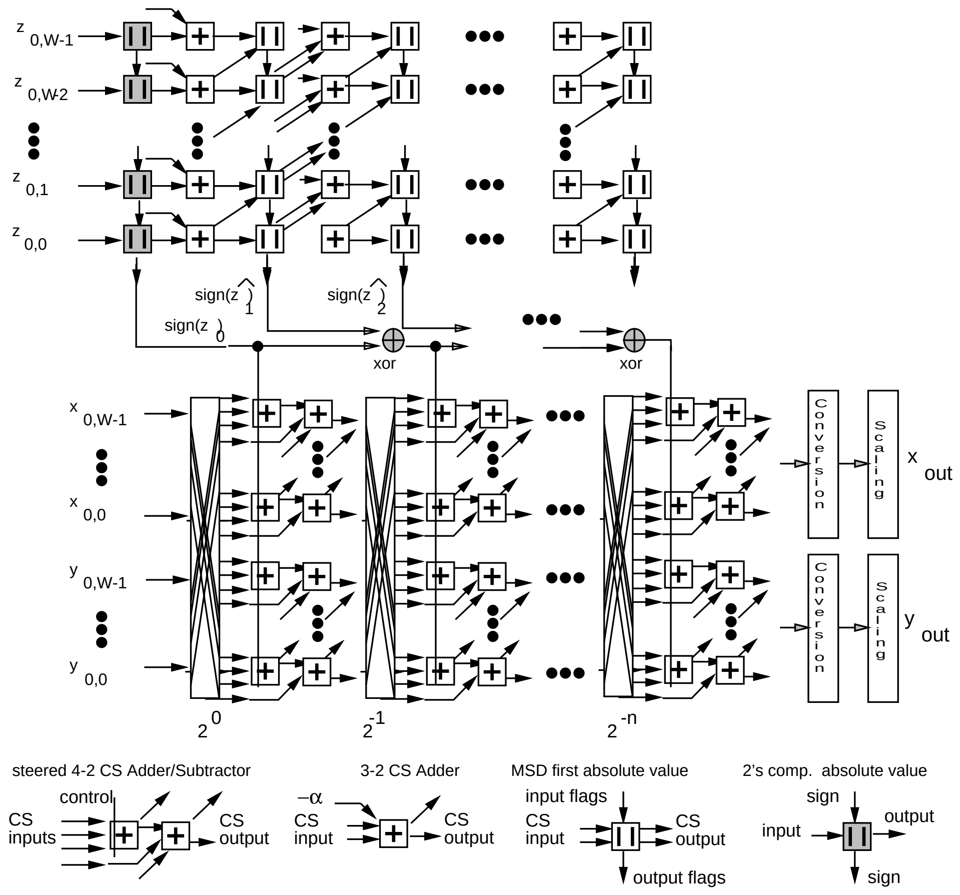

Figure 24.17 Parallel Architecture for the DCORDIC Rotation Mode with n iterations.

the iteration variable is called $\hat{z}_i$ with $|\hat{z}_i| = |z_i|$.

$$\begin{aligned} |\hat{z}_{i+1}| &= ||\hat{z}_i| - \alpha_i| \\ \text{sign}(z_{i+1}) &= \text{sign}(z_i) \cdot \text{sign}(\hat{z}_{i+1}) \\ x_{i+1} &= x_i - \text{sign}(z_i) \cdot y_i \cdot 2^{-i} \\ y_{i+1} &= y_i + \text{sign}(z_i) \cdot x_i \cdot 2^{-i} \end{aligned} \tag{22}$$

As shown in Eq. (22) the sign of the iteration variable z_i is achieved by differential decoding the sign of $\hat{z}_i$ given the initial sign sign(z_0). A negative sign of $\hat{z}_i$ corresponds to a sign change for z_i. The iteration equations for x_i and y_i are equal to the usual algorithm. In order to obtain sign($\hat{z_1}$) a single initial ripple propagation from MSD to LSD has to be taken into account for the MSD first absolute value calculation. The successive signs are calculated with a small bit–level propagation delay. The resulting parallel architecture for the DCORDIC rotation mode is shown in Fig. 24.17. Compared to the sign estimation approaches a clear advantage is given by the fact that no additional iterations are required for the DCORDIC.

24.5.3 Recent Developments

Below, some recent research results for the CORDIC algorithm are briefly mentioned which can not be treated in detail due to lack of space.
In [42] it is proposed to reduce the number of CORDIC iterations by replacing the second half of the iterations with a final multiplication. Further low latency CORDIC algorithms were derived in [43, 44] for parallel implementation of the rotation mode and for word–serial recursive implementations of both rotation mode and vectoring mode in [45]. The computation of $\sin^{-1}$ and $\cos^{-1}$ using CORDIC was proposed in [46, 47].
Recently, a family of generalized multi–dimensional CORDIC algorithms, so called *Householder* CORDIC algorithms, was derived in [48, 49]. Here, a modified iteration leads to scaling factors which are rational functions instead of square roots of rational functions as in conventional CORDIC. This attractive feature can be exploited for the derivation of new architectures [48, 49].

REFERENCES

[1] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Computers*, vol. EC–8, no. 3, pp. 330–34, September 1959.

[2] J. S. Walther, "A unified algorithm for elementary functions," in *AFIPS Spring Joint Computer Conference*, vol. 38, pp. 379–85, 1971.

[3] Y. H. Hu, "CORDIC–based VLSI Architectures for Digital Signal Processing," *IEEE Signal Processing Magazine*, pp. 16–35, July 1992.

[4] Y. H. Hu, "The Quantization Effects of the CORDIC Algorithm," *IEEE Transactions on Signal Processing*, vol. 40, pp. 834–844, July 1992.

[5] 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.

[6] 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. 989–95, September 1991.

[7] 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. 725–40, June 1990.

[8] D. H. Daggett, "Decimal–Binary Conversions in CORDIC," *IEEE Trans. on Electronic Computers*, vol. EC–8, no. 3, pp. 335–39, September 1959.

[9] X. Hu, R. Harber, and S. C. Bass, "Expanding the Range of Convergence of the CORDIC Algorithm," vol. 40, pp. 13–20, 1991.

[10] X. Hu and S. C. Bass, "A neglected Error Source in the CORDIC Algorithm," in *Proceedings IEEE ISCAS'93*, pp. 766–769, 1993.

[11] J. R. Cavallaro and F. T. Luk, "Floating point CORDIC for matrix computations," in *IEEE International Conference on Computer Design*, pp. 40–42, 1988.

[12] G. J. Hekstra and E. F. Deprettere, "Floating–Point CORDIC," *technical report: ET/NT 93.15, Delft University*, 1992.

[13] G. J. Hekstra and E. F. Deprettere, "Floating–Point CORDIC," in *Proc. 11th Symp. Computer Arithmetic*, (Windsor, Ontario), pp. 130–137, June 1993.

[14] J. R. Cavallaro and F. T. Luk, "CORDIC Arithmetic for a SVD processor," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 271–90, 1988.

[15] A. A. de Lange, A. J. van der Hoeven, E. F. Deprettere, and J. Bu, "An optimal floating-point pipeline Cmos CORDIC Processor," in *IEEE ISCAS'88*, pp. 2043–47, 1988.

[16] K. Hwang, *Computer Arithmetic: Principles, Architectures, and Design.* New York: John Wiley & Sons, 1979.

[17] N. R. Scott, *Computer number systems and arithmetic.* Englewood Cliffs: Prentice Hall, 1988.

[18] H. M. Ahmed, *Signal processing algorithms and architectures.* 1981. Ph.D. Thesis, Dept. Elec. Eng., Stanford (CA).

[19] R. Mehling and R. Meyer, "CORDIC–AU, a suitable supplementary Unit to a General–Purpose Signal Processor," *AEÜ*, vol. 43, no. 6, pp. 394–97, 1989.

[20] G. L. Haviland and A. A. Tuszynski, "A CORDIC arithmetic Processor chip," *IEEE Transactions on Computers*, vol. C–29, no. 2, pp. 68–79, Feb. 1980.

[21] E. F. Deprettere, P. Dewilde, and R. Udo, "Pipelined CORDIC architectures for fast VLSI filtering and array processing," in *Proceedings IEEE ICASSP*, pp. 41 A6.1 – 41 A6.4, March 1984.

[22] J. Bu, E. F. Deprettere, and F. du Lange, "On the optimization of pipelined silicon CORDIC Algorithm," in *Proceedings EUSIPCO 88*, pp. 1227–30, 1988.

[23] G. Schmidt, D. Timmermann, J. F. Böhme, and H. Hahn, "Parameter optimization of the CORDIC Algorithm and implementation in a CMOS chip," in *Proc. EUSIPCO'86*, pp. 1219–22, 1986.

[24] A. M. Despain, "Fourier Transform Computers using CORDIC Iterations," *IEEE Transactions on Computers*, vol. C–23, pp. 993–1001, Oct. 1974.

[25] S. Y. Kung, *VLSI Array Processors.* Englewood Cliffs: Prentice–Hall, 1988.

[26] D. Timmermann, H. Hahn, B. J. Hosticka, and G. Schmidt, "A programmable CORDIC Chip for Digital Signal Processing Applications," *IEEE Transactions on Solid–State Circuits*, vol. 26, no. 9, pp. 1317–1321, 1991.

[27] C. E. Leiserson, F. Rose, and J. Saxe, "Optimizing synchronous circuitry for retiming," in *Proc. of the 3rd Caltech Conf. on VLSI*, (Pasadena), pp. 87–116, March 1983.

[28] N. Takagi, T. Asada, and S. Yajima, "A hardware algorithm for computing sine and cosine using redundant binary representation," *Trans. IEICE Japan (in japanese)*, vol. J69–D, no. 6, pp. 841–47, 1986.

[29] Y. H. Hu and S. Naganathan, "Efficient Implementation of the Chirp Z–Transform using a CORDIC Processor," *IEEE Transactions on Signal Processing*, vol. 38, pp. 352–354, Feb. 1990.

[30] Y. H. Hu and S. Liao, "CALF: A CORDIC adaptive lattice filter," *IEEE Transactions on Signal Processing*, vol. 40, pp. 990–993, April 1992.

[31] T. Noll et al, "A Pipelined 330 MHz Multiplier," *IEEE Journal Solid State Circuits*, vol. SC–21, pp. 411–16, 1986.

[32] A. Vandemeulebroecke, E. Vanzieleghem, T. Denayer, and P. G. A. Jespers, "A new carry–free division algorithm and its application to a single–chip 1024–b RSA processor," *IEEE Journal Solid State Circuits*, vol. 25, no. 3, pp. 748–65, 1990.

[33] B. Parhami, “Generalized signed-digit number systems: A unifying framework for redundant number representations,” *IEEE Trans. on Computers*, vol. 39, no. 1, pp. 89–98, 1990.

[34] T. Noll, “Carry-save arithmetic for high-speed digital signal processing,” in *IEEE ISCAS'90*, vol. 2, pp. 982–86, 1990.

[35] T. Noll, “Carry–Save Architectures for High–Speed Digital Signal Processing,” *Journal of VLSI Signal Processing*, vol. 3, no. 1/2, pp. 121–140, June 1991.

[36] M. D. Ercegovac and T. Lang, “Implementation of fast angle calculation and rotation using on-line CORDIC,” in *IEEE ISCAS'88*, pp. 2703–06, 1988.

[37] J. Lee and T. Lang, “Constant–Factor Redundant CORDIC for Angle Calculation and Rotation,” *IEEE Trans. Computers*, vol. 41, pp. 1016–1035, August 1992.

[38] J. Duprat and J.-M. Muller, “The CORDIC Algorithm: New Results for Fast VLSI Implementation,” *IEEE Transactions on Computers*, vol. 42, no. 2, pp. 168–178, 1993.

[39] H. Dawid and H. Meyr, “The Differential CORDIC Algorithm: Constant Scale Factor Redundant Implementation without correcting Iterations,” *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 307–318, March 1996.

[40] H. Dawid and H. Meyr, “Very high speed CORDIC Implementation: Algorithm Transformation and novel Carry–Save Architecture,” in *Proceedings of the European Signal Processing Conference EUSIPCO '92*, (Brussels), pp. 358–372, Elsevier Science Publications, August 1992.

[41] H. Dawid and H. Meyr, “High speed bit–level pipelined Architectures for redundant CORDIC implementation,” in *Proceedings of the Int. Conf. on Application Specific Array Processors*, (Oakland), pp. 358–372, IEEE Computer Society Press, August 1992.

[42] D. Timmermann, H. Hahn, and B. J. Hosticka, “Modified CORDIC algorithm with reduced iterations,” *Electronics Letters*, vol. 25, no. 15, pp. 950–951, 1989.

[43] D. Timmermann and I. Sundsbo, “Area and Latency efficient CORDIC Architectures,” in *Proc. ISCAS'92*, pp. 1093–1096, 1992.

[44] D. Timmermann, H. Hahn, and B. Hosticka, “Low Latency Time CORDIC Algorithms,” *IEEE Trans. Computers*, vol. 41, no. 8, pp. 1010–1015, 1992.

[45] J. Villalba and T. Lang, “Low Latency Word Serial CORDIC,” in *Proceedings IEEE Conf. Application specific Systems, Architectures and Processors ASAP*, (Zurich), pp. 124–131, July 1997.

[46] C. Mazenc, X. Merrheim, and J. M. Muller, “Computing functions $\cos^{-1}$ and $\sin^{-1}$ using Cordic,” *IEEE Trans. Computers*, vol. 42, no. 1, pp. 118–122, 1993.

[47] T. Lang and E. Antelo, “CORDIC–based Computation of ArcCos and ArcSin,” in *Proceedings IEEE Conf. Application specific Systems, Architectures and Processors ASAP*, (Zurich), pp. 132–143, July 1997.

[48] S.F.Hsiao and J.M.Delosme, "Householder CORDIC algorithms," *IEEE Transactions on Computers*, vol. C–44, no. 8, pp. 990–1001, Aug. 1995.

[49] S.F.Hsiao and J.M.Delosme, "Parallel Singular Value Decomposition of Complex Matrices usijng Multi–dimensional CORDIC Algorithms," *IEEE Transactions on Signal Processing*, vol. 44, no. 3, pp. 685–697, March 1996.

[50] J. Lee and T. Lang, "On-line CORDIC for generalized singular value decomposition," in *SPIE Vol. 1058 High Speed Computing II*, pp. 235–47, 1989.

[51] 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," in *Proceedings IEEE ISCAS'88*, pp. 581–84, 1988.

[52] R. Künemund, H. Söldner, S. Wohlleben, and T. Noll, "CORDIC Processor with Carry-Save Architecture," in *Proc. ESSCIRC'90*, pp. 193–96, 1990.

[53] H. X. Lin and H. J. Sips, "On-line CORDIC Algorithms," *IEEE Trans. Computers*, no. 8, pp. 1038–52, August 1990.

[54] H. Yoshimura, T. Nakanishi, and H. Yamauchi, "A 50 MHz CMOS geometrical mapping processor," *IEEE Transactions on Circuits and Systems*, vol. 36, no. 10, pp. 1360–63, 1989.

LIST OF FIGURES

LIST OF TABLES