CHAPTER 25

CORDIC ARCHITECTURES FOR FPGA COMPUTING

Chris Dick
Advanced Systems Technology Group
DSP Division of Xilinx, Inc.

Because field-programmable gate arrays (FPGAs) are often used for realizing complex mathematical calculations, the FPGA designer is in need of a set of math libraries to support such implementations. The literature is rich with algorithmic options for evaluating the type of math functions (e.g., sine, cosine, sinh, cosh, arctangent, atan2, logarithms) that are typically found in a math library for general-purpose and DSP processors. The enormous flexibility of the FPGA coupled with the vast suite of algorithmic options for computing math functions can make the development of an FPGA math library a challenging task.

Common approaches to evaluating math functions include polynomial approximation-based techniques [13] and Newton-style iterations [13], to name a couple. One of the most useful and flexible approaches available to the hardware designer for developing high-performance computing hardware is the CORDIC (COordinate Rotation DIgital Computer) algorithm.

CORDIC is unparalleled in its ability to encapsulate a diversity of math functions in one basic set of iterations. It can be viewed as the Swiss Army Knife, so to speak, of arithmetic—that is, a single hardware architecture, with very minimal control overhead, having the ability to compute sine, cosine, cosh, sinh, atan2, square root, and polar-to-rectangular and rectangular-to-polar conversions, to name only a few functions.

It is in coordinate transformations that the algorithm comes into its own. In both, multi-operand input and multi-element output vectors are involved. There are a plethora of alternatives for realizing, say, division in an FPGA, and most of the CORDIC alternatives provide good hardware efficiency. However, the algorithm remains unrivaled when it comes to processing multi-element I/O vectors, as is the case when converting from Cartesian to polar coordinates or vice versa. CORDIC falls into the class of shift-and-add algorithms—it is a multiplierless method dominated by additions. FPGAs are very efficient at realizing arbitrary precision adders, and so the CORDIC algorithm is in many ways a natural fit for course-grained FPGA architectures such as the Xilinx Virtex-4 family of devices [41].

This chapter begins with a brief tutorial overview of the CORDIC algorithm. Because most hardware realizations of CORDIC employ fixed-point arithmetic, design considerations for quantizing the datapath and selecting a suitable number of iterations are provided. Approaches for architecting FPGA CORDIC processors are then presented. Various options are discussed that highlight the use of FPGA features such as embedded multipliers, embedded multiply–accumulator (MACC) tiles, and logic fabric to deliver hardware realizations that provide various trade-offs between throughput, latency, logic fabric utilization, and numerical accuracy. A brief overview of the System Generator [38] design flow used to produce our implementations is also provided. Design considerations for producing very high throughput (450–500 MHz) implementations in Virtex-4 [41] devices are presented as well.

25.1 CORDIC ALGORITHM

The CORDIC algorithm was first published by Volder [35] in 1959 as a technique for efficiently implementing the trigonometric functions required for real-time aircraft navigation. Since first being published, the method has been extensively analyzed and extended to the point where a very rich set of functions is accessible from the one basic set of equations. The algorithm is dominated by bit shifts and additions and so was an ideal match for early-generation computing technology in which multiplication and division were expensive in terms of computation time and physical resources. Volder essentially presented iterative techniques for performing translations between Cartesian and polar coordinate systems (*vectoring mode*), and a method for realizing a plane rotation (*rotation mode*) using a series of arithmetic shifts and adds.

Since its publication, the CORDIC algorithm has been applied to many different applications and has been used as the cornerstone of the arithmetic engine in many VLSI signal-processing implementations [34]. It has been used extensively for computing various types of transforms, including the fast Fourier transform (FFT) [10,11], the discrete cosine transform [4], and the discrete Hartley transform [3]. And it has found widespread use in realizing various classes of digital filters, including Kalman filters [31], adaptive lattice structures [21], and adaptive nulling [30]. A large body of work has been published on CORDIC-based approaches for implementing various types of linear algebra operations, including singular value decomposition (SVD) [1], Given's rotations [30], and QRD-RLS (recursive least squares) filtering [14].

A brief tutorial style treatment of the basic algorithm is provided here; its FPGA implementation will be discussed in subsequent sections.

25.1.1 Rotation Mode

The CORDIC algorithm has two basic modes: vectoring and rotation. These can be applied in several coordinate systems, including circular, hyperbolic, and linear, to compute various functions such as atan2, sine, cosine, and even division. We begin our treatment by considering the problem of constructing an efficient method to realize a plane rotation of the vector (x_s, y_s) through an angle θ to produce a vector (x_f, y_f), as shown in Figure 25.1.

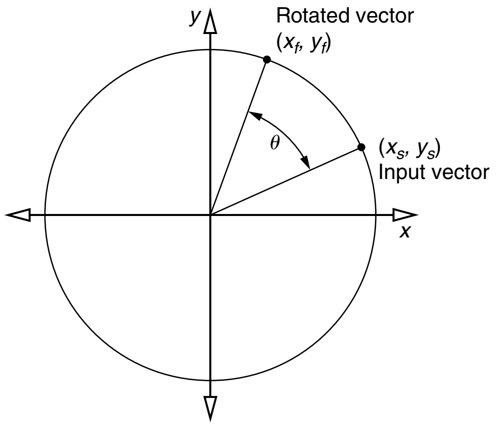


FIGURE 25.1 ■ Plane rotation of the vector (x_s, y_s) through an angle θ.

The rotation is formally captured in matrix form by equation 25.1.

$$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_s \\ y_s \end{bmatrix} = ROT(\theta) \begin{bmatrix} x_s \\ y_s \end{bmatrix} \tag{25.1}$$

which can be expanded to the set of equations in equation 25.2.

$$\begin{aligned} x_f &= x_s \cos\theta - y_s \sin\theta \\ y_f &= x_s \sin\theta + y_s \cos\theta \end{aligned} \tag{25.2}$$

The development of a simplified approach for producing rotation through the angle θ begins by considering it not as one lumped operation but as the result of a series of smaller rotations, or *micro-rotations*, through the set of angles α_i where

$$\theta = \sum_{i=0}^{\infty} \alpha_i \tag{25.3}$$

The rotation can now be cast as a product of smaller rotations, or

$$ROT(\theta) = \prod_{i}^{\infty} ROT(\alpha_i) \tag{25.4}$$

If these values α_i are carefully chosen, we can provide a very efficient computation structure. Equation 25.2 can be modified to reflect a micro-rotation $ROT(\alpha_i)$, leading to equation 25.5.

$$\begin{aligned} x_{i+1} &= x_i \cos\alpha_i - y_i \sin\alpha_i \\ y_{i+1} &= x_i \sin\alpha_i + y_i \cos\alpha_i \end{aligned} \tag{25.5}$$

where $(x_0, y_0) = (x_s, y_s)$. Factoring permits the equations to be expressed as

$$\begin{aligned} x_{i+1} &= \cos\alpha_i (x_i - y_i \tan\alpha_i) \\ y_{i+1} &= \cos\alpha_i (y_i + x_i \tan\alpha_i) \end{aligned} \tag{25.6}$$

which positions the iterative update as the product of two procedures: a scaling by the $\cos\alpha_i$ term and a similarity transformation, or scaled rotation.

The next significant step that leads to an algorithm that lends itself to an efficient hardware realization is to place restrictions on the values that α_i can take. If

$$\alpha_i = \tan^{-1}\left(\sigma_i 2^{-i}\right) \tag{25.7}$$

where $\sigma_i \in \{-1, +1\}$, then equation 25.6 can be written as

$$\begin{aligned} x_{i+1} &= \cos\alpha_i \left(x_i - \sigma_i y_i 2^{-i}\right) \\ y_{i+1} &= \cos\alpha_i \left(y_i + \sigma_i x_i 2^{-i}\right) \end{aligned} \tag{25.8}$$

The purpose of σ_i will be explained shortly.

With the exception of the scaling term, these equations can be implemented using only additions, subtractions, and shifts. In the set of equations that are typically presented as the CORDIC iterations, and following the lead of Volder [35], the scaling term is usually excluded from the defining equations to produce the modified set of equations

$$\begin{aligned} x_{i+1} &= x_i - \sigma_i y_i 2^{-i} \\ y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \end{aligned} \tag{25.9}$$

To determine the value of these σ_i we introduce a new variable, z (the *angle* variable). The recurrence on z is defined by equation 25.10.

$$z_{i+1} = z_i - \sigma_i \tan^{-1}\left(2^{-i}\right) \tag{25.10}$$

If the z variable is initialized with the desired angle of rotation θ—that is, z_0—it can be driven to 0 by conditionally adding or subtracting terms of the form $\tan^{-1}\left(2^{-i}\right)$ from the state variable z. The conditioning is captured by the term σ_i as a test on the sign of the current state of the angle variable z_i—that is,

$$\sigma_i = \begin{cases} 1 & \text{if } z_i \geq 0 \\ -1 & \text{if } z_i < 0 \end{cases} \tag{25.11}$$

Driving z to 0 is actually an iterative process for decomposing θ into a weighted linear combination of terms of the form $\tan^{-1}\left(2^{-i}\right)$. As z goes to 0, the vector (x_0, y_0) experiences a sequence of micro-rotation extensions that in the limit $n \to \infty$ converge to the coordinates $\left(x_f, y_f\right)$.

The complete algorithm is summarized in equation 25.12.

$$\begin{aligned}
i &= 0 \\
x_0 &= x_s \\
y_0 &= y_s \\
z_0 &= \theta \\
x_{i+1} &= x_i - \sigma_i y_i 2^{-i} \\
y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \\
z_{i+1} &= z_i - \sigma_i \tan^{-1}\left(2^{-i}\right) \\
\sigma_i &= \begin{cases} 1 & \text{if } z_i \geq 0 \\ -1 & \text{if } z_i < 0 \end{cases}
\end{aligned} \tag{25.12}$$

which is easily realized in hardware because of the simple nature of the arithmetic required. The only complex function is the $\tan^{-1}$, which can be precomputed and stored in a memory.

Because of the manner in which the updates are directed, this mode of the CORDIC algorithm is sometimes referred to as the *z-reduction mode*. Figure 25.2 shows the signal flow graph for the algorithm. Observe the butterfly-style architecture in the cross-addition update.

25.1.2 Scaling Considerations

Because the scaling term $\cos\alpha_i$ has not been carried over into equation 25.12, the input vector (x_0, y_0) not only undergoes a rotation but also experiences scaling or growth by a factor $1/\cos\alpha_i$ at each iteration. That is,

$$\begin{aligned}
R_{i+1} &= K_{c,i} R_i = \frac{1}{\cos\alpha_i} R_i = \left(1 + \sigma_i^2 2^{-2i}\right)^{1/2} R_i \\
&= \left(1 + 2^{-2i}\right)^{1/2} R_i
\end{aligned} \tag{25.13}$$

where $R_i = |x_i + jy_i|$ designates the modulus of the vector at iteration i, and the subscript c associates the scaling constant with the *circular* coordinate system.

Figure 25.3 illustrates the growth process at each of the intermediate CORDIC iterations as (x_0, y_0), which is translated to its final location (x_f, y_f). For an infinite number of iterations the scaling factor is

$$K_c = \prod_{i=0}^{\infty} \left(1 + 2^{-2i}\right)^{1/2} \approx 1.6468 \tag{25.14}$$

It should also be noted that, since $\sigma_i \in \{-1, +1\}$, the scaling term is a constant that is independent of the angle of rotation.

As captured by equation 25.4, the angle of rotation θ is decomposed into an infinite number of elemental angles α_i, which implies that an infinite number of iterations is theoretically required. In practice, a finite number of iterations, n, is selected to make the system realizable in software or hardware. Application of n iterations translates (x_0, y_0) to (x_n, y_n) rather than to (x_f, y_f)

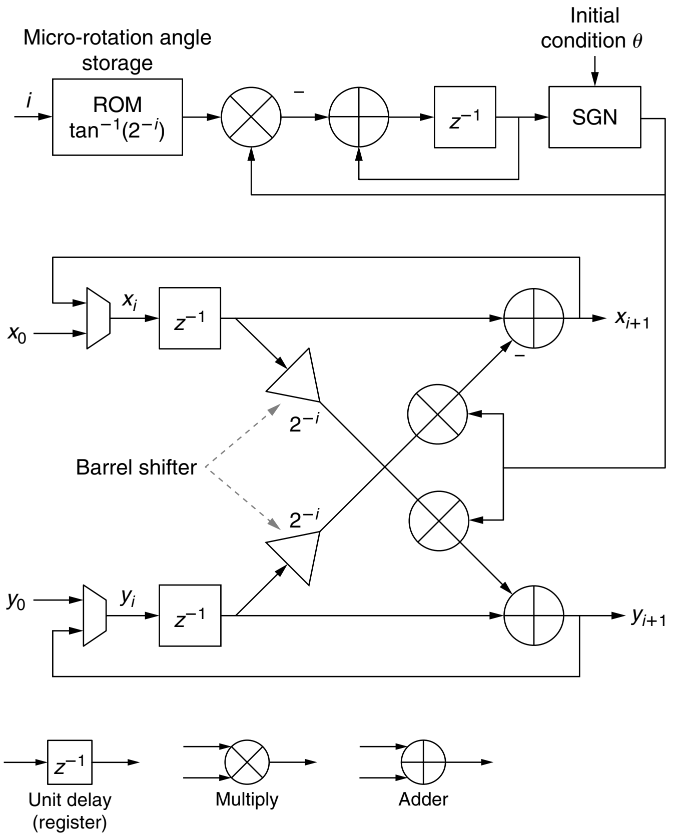


FIGURE 25.2 ■ A signal flow graph for CORDIC vector rotation.

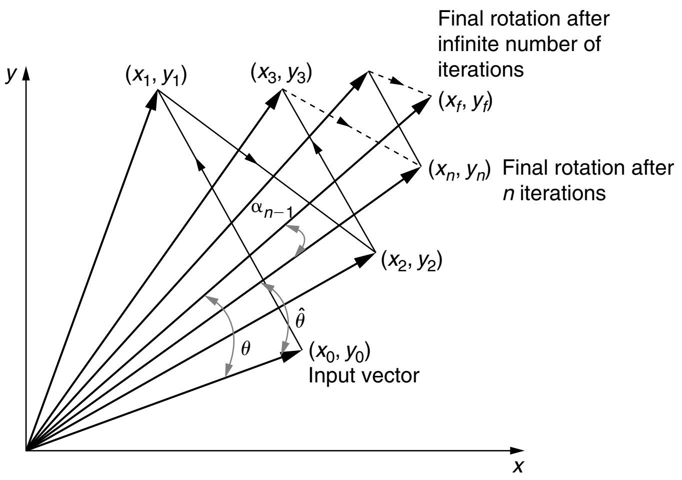

y
(x_1, y_1)
(x_3, y_3)
Final rotation after infinite number of iterations
(x_f, y_f)
(x_n, y_n) Final rotation after n iterations
α_{n-1}
(x_2, y_2)
$\hat{\theta}$
θ
(x_0, y_0) Input vector
x

FIGURE 25.3 ■ Each iteration of a CORDIC rotation introduces vector growth by a factor of $\frac{1}{\cos \alpha_i} = \left(1 + \sigma_i^2 2^{-2i}\right)^{1/2}$.

as shown in Figure 25.3. The rotation error $\left|\arg\left(x_f + jy_f\right) - \arg\left(x_n + jy_n\right)\right|$ has an upper bound of α_{n-1}, which is the smallest term in the weighted linear expansion of θ.

For an *infinite-precision* arithmetic implementation of the system of equations, each iteration contributes one additional effective fractional bit to the result. Most hardware implementations of the CORDIC algorithm are realized using fixed-point arithmetic, and, as will be discussed soon, the relationship between the number of effective output binary result digits is very different from that of a floating-point realization of the algorithm.

25.1.3 Vectoring Mode

The CORDIC vectoring mode is most commonly used for implementing a conversion from a rectangular to a polar coordinate system. In contrast to rotation mode, where Z is driven to 0, in the vectoring mode the initial vector (x_0, y_0) is rotated until the y component is driven to 0. The modification to the basic algorithm required to accomplish this goal is to direct the iterations using the sign of y_i. As the y variable is reduced, the corresponding angle of rotation is accumulated in the z register. The complete vectoring algorithm is captured by equation 25.15.

$$
\begin{aligned}
i &= 0 \\
x_0 &= x_s \\
y_0 &= y_s \\
z_0 &= 0 \\
x_{i+1} &= x_i - \sigma_i y_i 2^{-i} \\
y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \\
z_{i+1} &= z_i - \sigma_i \tan^{-1}\left(2^{-i}\right) \\
\sigma_i &= \begin{cases} 1 & \text{if } y_i < 0 \\ -1 & \text{if } y_i \geq 0 \end{cases}
\end{aligned}
\tag{25.15}
$$

This CORDIC mode is commonly referred to as *y-reduction* mode.

Figure 25.4 shows the results of a CORDIC vector mode simulation for $\arg(x_s + jy_s) = 7\pi/8$ and $|x_s + jy_s| = 1$. The top plot (a) shows the true angle of the input vector (solid line) overlaid with $\arg(x_i + jy_i), i = 1, \ldots, 16$. We note the oscillatory behavior of (x_i, y_i) about the true value of the angle. Overdamped or underdamped behavior will be produced depending on the system initial conditions. The lower plot (b) shows, for this case of initial conditions, how rapidly the algorithm can converge toward the correct solution. In fact, for many practical applications, a *short* CORDIC (small number of iterations) produces acceptable performance.

For example, in a 16-QAM (quadrature amplitude modulation) carrier recovery circuit [29] employing a Costas Loop [23], a 5-iteration CORDIC usually provides adequate performance [12].

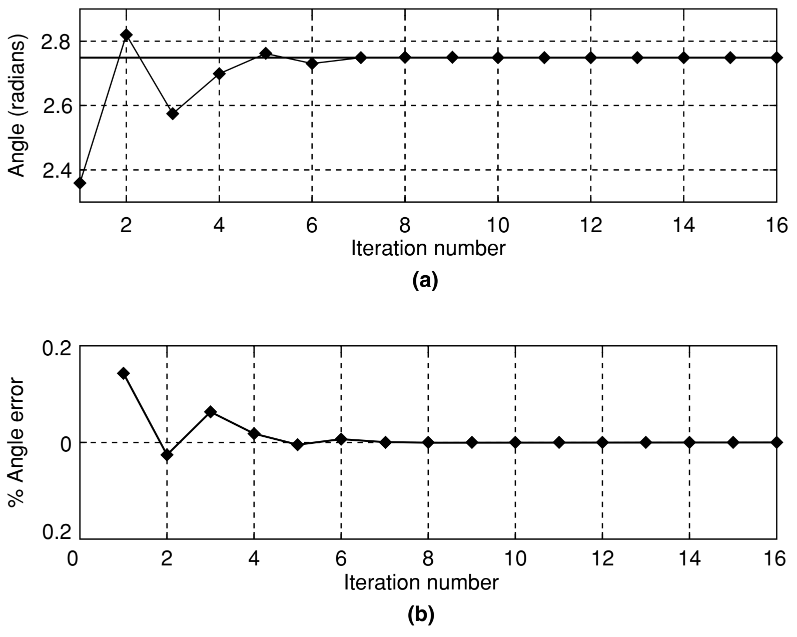


FIGURE 25.4 ■ Convergence of CORDIC vectoring. The top plot (a) shows the true angle of the input vector $\arg(x_s + jy_s)$ (solid line) overlaid with $\arg(x_i + jy_i)$, $i = 1,\ldots,16$. The bottom plot (b) is the percentage angle error as a function of the iteration number.

25.1.4 Multiple Coordinate Systems and a Unified Description

Alternative versions of the CORDIC engine can be defined under the circular, hyperbolic, and linear coordinate systems [13]. These use a computation similar to that of the basic CORDIC algorithm, but can provide additional functions. It is possible to capture the vectoring and rotation modes of the CORDIC algorithm in all three coordinate systems using a single set of unified equations. To do this a new variable, m, is introduced to identify the coordinate system so that

$$m = \begin{cases} +1 & \text{circular coordinates} \\ 0 & \text{linear coordinates} \\ -1 & \text{hyperbolic coordinates} \end{cases} \tag{25.16}$$

The unified micro-rotation is

$$\begin{aligned} x_{i+1} &= x_i - m\sigma_i y_i 2^{-i} \\ y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \\ z_{i+1} &= \begin{cases} z_i - \sigma_i \tan^{-1}\left(2^{-i}\right) & \text{if } m = 1 \\ z_i - \sigma_i \tan\text{h}^{-1}\left(2^{-i}\right) & \text{if } m = -1 \\ z_i - \sigma_i\left(2^{-i}\right) & \text{if } m = 0 \end{cases} \end{aligned} \tag{25.17}$$

The scaling factor is $K_{m,i} = \left(1 + m2^{-2i}\right)^{1/2}$.

TABLE 25.1 ■ Functions computed by a CORDIC processor for the circular ($m = 1$), hyperbolic ($m = -1$), and linear ($m = 0$) coordinate systems

Coordinate system	Rotation/vectoring	Initialization	Result vector
1	Rotation	$x_0 = x_s$	$x_n = K_{1,n} \cdot (x_s \cos\theta - y_s \sin\theta)$
		$y_0 = y_s$	$y_n = K_{1,n} \cdot (y_s \cos\theta + x_s \sin\theta)$
		$z_0 = \theta$	$z_n = 0$
		$x_0 = 1/K_{1,n}$	$x_n = \cos\theta$
		$y_0 = 0$	$y_n = \sin\theta$
		$z_0 = \theta$	$z_n = 0$
1	Vectoring	$x_0 = x_s$	$x_n = K_{1,n} \cdot \operatorname{sgn}(x_0) \cdot \left(\sqrt{x^2 + y^2}\right)$
		$y_0 = y_s$	$y_n = 0$
		$z_0 = \theta$	$z_n = \theta + \tan^{-1}(y_s/x_s)$
0	Rotation	$x_0 = x_s$	$x_n = x_s$
		$y_0 = y_s$	$y_n = y_s + x_s y_s$
		$z_0 = z_s$	$z_n = 0$
0	Vectoring	$x_0 = x_s$	$x_n = x_s$
		$y_0 = y_s$	$y_n = 0$
		$z_0 = z_s$	$z_n = z_s + y_s/x_s$
-1	Rotation	$x_0 = x_s$	$x_n = K_{-1,n} \cdot (x_s \cosh\theta + y_s \sinh\theta)$
		$y_0 = y_s$	$y_n = K_{-1,n} \cdot (y_s \cosh\theta + x_s \sinh\theta)$
		$z_0 = \theta$	$z_n = 0$
		$x_0 = 1/K_{-1,n}$	$x_n = \cosh\theta$
		$y_0 = 0$	$y_n = \sinh\theta$
		$z_0 = \theta$	$z_n = 0$
-1	Vectoring	$x_0 = x_s$	$x_n = K_{-1,n} \cdot \operatorname{sgn}(x_0) \cdot \left(\sqrt{x^2 - y^2}\right)$
		$y_0 = y_s$	$y_n = 0$
		$z_0 = \theta$	$z_n = \theta + \tanh^{-1}(y_s/x_s)$

TABLE 25.2 ■ CORDIC shift sequences, ranges of covergence, and scale factor bound for circular, linear, and hyperbolic coordinate systems

Coordinate system	Shift sequence	Convergence	Scale factor
m	$s_{m,i}$	θ_{MAX}	K_m ($n \longrightarrow \infty$)
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	≈ 0.83816

* For $m = -1$, the following iterations are repeated: $\{4, 13, 40, 121, \ldots, k, 3k + 1, \ldots\}$.

Operating the two modes in the three coordinate systems, in combination with suitable initialization of the algorithm variables, generates a rich set of functions, shown in Table 25.1. Table 25.2 summarizes the shift sequences, maximum angle of convergence θ_{MAX} (elaborated on in a later section), and scaling function for the three coordinate systems. Note that each system requires slightly different shift sequences (the sequence of i values).

25.1.5 Computational Accuracy

One of the first design requirements for the fixed-point arithmetic implementation of a CORDIC processor is to define the numerical precision requirements of the datapath. This includes defining the numeric representation for the input operands and the processing engine internal registers, in addition to the number of micro-rotations that will be required to achieve a specified numerical quality of result. To guide this process it is useful to have an appreciation for the sources of computation noise in CORDIC arithmetic. While CORDIC processing can be realized with floating-point arithmetic [2,7], we will restrict our discussion to fixed-point arithmetic implementations, as they are the most commonly used numeric type employed in FPGA realizations.

Two primary noise sources are to be considered. One is associated with the weighted and finite linear combination of elemental angles that are used to represent the desired angle of rotation θ; the second source is associated with the rounding of the datapath variables x, y, and z. These noise sources are referred to as the *angle approximation* and the *rounding error*, respectively.

Angle approximation error

In this discussion we assume that all finite-precision quantities are represented using fixed-point 2's complement arithmetic, so the value F of a normalized number u represented using m binary digits $(u_{m-1}u_{u-2}\ldots u_0)$ is

$$F = -u_{m-1} + \sum_{j=0}^{m-2} u_j \cdot 2^{-m+j+1} \tag{25.18}$$

As will be presented next, there is a requirement in the CORDIC algorithm to accommodate bit growth in both the integer and fractional fields of the x and y variables. To accommodate this, the data format is enhanced with an additional G_I and G_F integer and fractional guard bits, respectively, so that a number with $B_I + G_I$ and $B_F + G_F$ bits allocated to the integer and fractional fields s and r, respectively $\left(s_{B_I+G_I-1}s_{B_I+G_I-2}\ldots s_0 r_{B_F+G_F-1} r_{B_F+G_F-2}\ldots r_0\right)$, is expressed as

$$F = -r_{B_I+G_I-1} \cdot 2^{B_I+G_I-1} + \sum_{j=0}^{B_I+G_I-2} s_j \cdot 2^j + \sum_{j=0}^{B_F+G_F-1} r_j \cdot 2^{-(B_F+G_F)+j} \tag{25.19}$$

Figure 25.5 illustrates the extended data format. The integer guard bits are necessary to accommodate the vector growth experienced when operating in circular coordinates. The fractional guard bits are required to support the word growth that occurs in the fractional field of the x and y registers due to the successive arithmetic shift-right operations employed in the iterative updates. It is assumed that the input samples are represented as normalized $(1 \cdot B_F)$ quantities.

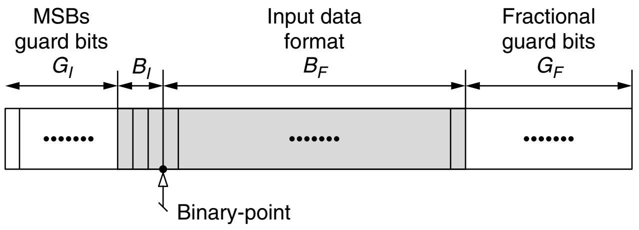


FIGURE 25.5 ■ The fractional fixed-point data format used for internal storage in the quantized CORDIC algorithm.

There are n fixed rotation angles $\alpha_{m,i}$ employed to approximate a desired angle of rotation θ. Neglecting all other error sources, the accuracy of the calculation is governed by the nth and final rotation, which limits the angle approximation error to $\alpha_{m,n-1}$. Because $\alpha_{m,n-1} = \frac{1}{\sqrt{m}} \tan^{-1}\left(\sqrt{m} \cdot 2^{-s_{m,n-1}}\right)$, the angle approximation error can be made arbitrarily small by increasing the number of micro-rotations n. Of course, the number of bits allocated to represent the elemental angles $\alpha_{m,i}$ needs to be sufficient to support the smallest angle $\alpha_{m,n-1}$. The number representation defined in equation 25.19 results in a least significant digit weighting of $2^{-(B_F+G_F)}$. Therefore, $\alpha_{m,n-1} \geq 2^{-(B_F+G_F)}$ must hold in order to represent $\alpha_{m,n-1}$. Approximately $n+1$ iterations are required to generate B_F significant fractional bits.

Datapath rounding error

As discussed earlier, most FPGA realizations of CORDIC processors employ fixed-point arithmetic. The update of the x, y, and z state variables according to equation 25.12 produces a dynamic range expansion, which is ideally supported by precisions that accommodate the worst-case bit growth. The number of additional guard bits beyond the original precision of the input operands can be very large, and carrying these additional bits in the datapath is generally impractical. For example, in the circular mode of operation the number of additional fractional bits required to support a full-precision calculation is determined by the sum of the shift sequence $s_{m,i}$.

If the input operands are presented as a 16.15 value (a 16-bit field width with 15 fractional bits) and 16 micro-rotations are performed, the bit growth for the fractional component of the datapath is $\sum_{i=0}^{15} i = 120$ bits. Thus, the total number of fractional bits required for a full-precision calculation is $120+15=135$. While FPGAs certainly provide the capability to support arbitrary precision arithmetic, it would be highly unusual to construct a CORDIC processor with such a wide datapath. In fact, the error in the CORDIC result vector can be maintained to a desired value using far few fractional guard bits, as discussed next.

Rather than by accommodating the bit growth implied in the algorithm, the dynamic range expansion is better handled by rounding the newly computed state variables. Control over wordlength can be achieved using unbiased rounding, simple truncation, or other techniques [26]. True rounding, while the preferred approach because of the smaller error introduced when compared to truncation, can be the most area consuming because a second addition is potentially required. In some cases, the cost of rounding can be significantly reduced by exploiting the carry-in port of the adders used in the implementation. Truncation is obviously the simplest approach, requiring only the extraction of a bit field from the full-precision value, but it introduces an undesirable positive bias in the final result and an error component that is twice the magnitude of unbiased rounding. Nevertheless, truncation arithmetic is the option most frequently employed in FPGA CORDIC datapath design.

A simple approach to understanding the quantization effects of the CORDIC algorithm was first presented by Walther [36]. A very complete analysis was later published by Hu [16], with further work reported by Park and Cho [28] and Hu and Bass [17].

For many practical applications Walther's method produces acceptable results, and this is the approach we will use to design the FPGA implementations. A brief summary of the method is presented here.

Analysis of the rounding error for the z variable is straightforward because there are no data shifts involved in the state update, as there are with the x and y variables. The rounding error is simply due to the quantization of the rotation angles. The upper bound on the error is then the accumulation of the absolute values of the rounding errors for the quantized angles $\alpha_{m,i}$.

Datapath pruning and its associated quantization effects for the x and y variables is certainly a more challenging analysis than that for the angle variable because the scaling term involved in the cross-addition update. Nevertheless, several extensive treatments have been published. The effects of error propagation in the algorithm were reported by Hu in a Cray Research publication [5] and later extended by Hu and Bass [17]. Walther's treatment takes a slightly simplified approach and assumes that the maximum rounding error for n iterations is the sum of the absolute value of the maximum rounding error associated with each micro-rotation and the subsequent quantization that is performed to control word growth.

The format for the CORDIC variables was shown in Figure 25.5. $B = B_I + B_F + G_F + G_I$ bits are used to for internal storage, with $B_F + G_F$ of these bits assigned to the fractional component of the representation. The maximum error for one iteration is therefore of magnitude $2^{-(B_F+G_F)}$. In the simplified analysis, the rounding error $e(n)$ in the final result, and after all n iterations, is simply n times this quantity, which is $e(n) = n2^{-(B_F+G_F)}$. If B_F accurate fractional bits are required in the result word, the required resolution is $2^{-(B_F-1)}$. If B_F is selected such that $e(n) \leq 2^{-B_F}$, the datapath quantization can effectively be ignored. This implies that $n2^{-(B_F+G_F)} \leq 2^{-B_F}$, which requires $B_F \geq \log_2(n)$. Therefore, $G_F = \lceil \log_2(n) \rceil$ fractional guard bits are required to produce a result that has an accuracy of B_F fractional bits. This simplified treatment of the computation noise is a reasonable approximation that can help guide the definition of the datapath width required to meet a specified numerical fidelity.

Figure 25.6 shows the results of a simulation using different data representations for the x, y, and z variables of a CORDIC vectoring algorithm in circular

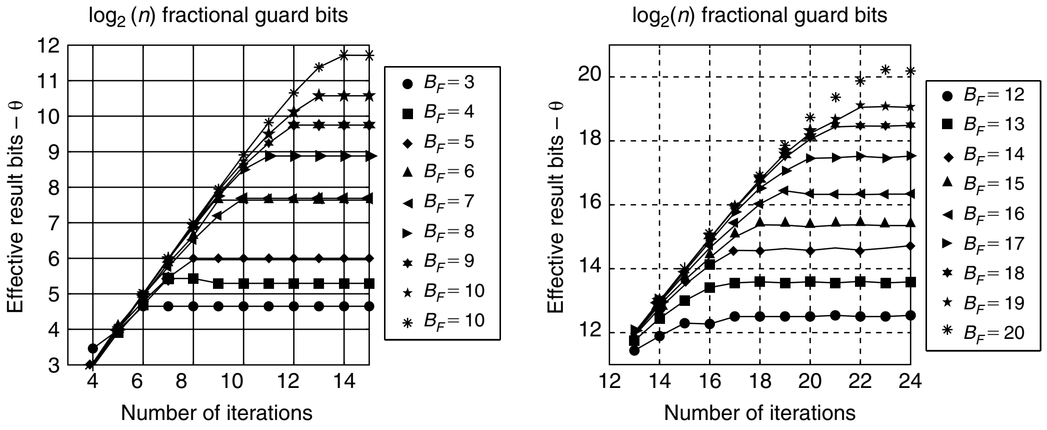


FIGURE 25.6 ■ The effiective number of result bites for a CORDIC vector processor (circular coordinates). The number of fractional guard bites is $G_F = \lceil log_2(n) \rceil$.

coordinates. Unit modulus complex vectors with random angles were generated and projected onto the CORDIC input sample (x_0, y_0). Each sample point in the plot represents the maximum absolute error of the angle estimate resulting from 4000 trials. We note that in all of the simulations the effective number of fractional output bits is matched to the number of fractional bits in the input operand.

The simplified treatment of the rounding noise generated in the update equations is certainly pessimistic and produces a requirement on the number of guard bits that is biased slightly higher than what might typically be required.

Selecting $G_F = \lceil \log_2(n) \rceil$ is certainly a safe, if not a slightly overengineered, choice. In the context of an FPGA realization, an additional bit of precision carried by the variables has almost negligible impact on the area and maximum operating clock frequency of the design.

An additional observation from the plots in Figure 25.6 is that the production of B_F effective output digits requires more iterations than the $B_F + 1$ iterations required for a full floating-point implementation—an additional three iterations are, in general, necessary. The implication of this is that two additional bits must be allocated to represent the elemental angles to provide the angle resolution implied by the adjusted iteration count.

Defining the number of guard bits G_I is very straightforward based on the number of integer bits B_I in the input operands, the coordinate system to be employed (e.g., circular, hyperbolic, or linear), and the mode (vectoring or rotation). For example, if the input data is in standard 2's complement format and bounded by ± 1, then $B_I = 1$. This means that the l^2 norm of the input (x_0, y_0) is $\sqrt{2}$. For the CORDIC vectoring mode, the range extension introduced by the iterations is approximately $K_1 \approx 1.6468$ for any reasonable number of iterations. The maximum that the final value of the x register can assume is approximately $\sqrt{2} \cdot 1.6468 \approx 2.3289$, which requires that $G_I = 2$.

TABLE 25.3 ■ Number of rotations and required CORDIC processor datapath format required to achieve a desired number of effective output bits

Number of effective fractional result bits	Micro-rotations: n	Internal storage data format: x and y	Internal storage data format: z
8	10	(15.12)	(15.14)
12	15	(19.16)	(19.18)
16	19	(24.21)	(24.23)
24	27	(32.29)	(32.31)

Based on this approach, a reasonable procedure for selecting the number of CORDIC micro-rotations and a suitable quantization for the x, y, and z variables, given the effective number of fractional bits required in the output, is the following:

1. Define the number of iterations as $n = B_F + 3$.
2. Select the field width for the x and y variables as $2 + B_I + B_F + \log_2(n)$ for the vectoring mode in circular coordinates—$B_F + \log_2(n)$ of these bits are of course allocated to the fractional component of the register.
3. Select the fractional precision of the angle register z to be $B_F + \log_2(n) + 2$, while maintaining 1 bit for the integer portion of the register.
4. Apply similar reasoning to select n and G_I for the other coordinate systems and modes.

Based on this approach, Table 25.3 shows the number of micro-rotations n and the internal data storage format corresponding to 8, 12, 16, 24, and 32 effective fractional result bits. The notation $(p \cdot q)$ indicates a bit field width of p bits, with q of these bits allocated to the fractional component of the value.

25.2 ARCHITECTURAL DESIGN

There are many hardware architecture options to evaluate when considering FPGA CORDIC datapath implementation. A particular choice is determined by the design specifications of numerical accuracy, throughput, and latency. At the highest level are key architectural decisions on whether a folded [27] or fully parallel [27] pipelined (or nonpipelined) architecture is to be used. At a lower, technology-specific level, FPGA features associated with a particular FPGA family are also a factor in the decision process. For example, later-generation FPGAs such as the Virtex-4 family [41] include an array of arithmetic units called the XtremeDSP Slice [43] (referred to as the DSP48 in the remainder of the chapter).

As discussed later, a CORDIC implementation can be realized that is mostly based on the DSP48 embedded tile. Thus, with this particular family of devices the designer has a choice of producing an implementation that is completely logic slice based [40] or biased toward the use of DSP48 elements. The process that guides such decisions is elaborated in the next section.

25.3 FPGA IMPLEMENTATION OF CORDIC PROCESSORS

One of the elegant properties of FPGA computing is the ability to construct a compute engine closely tailored to the problem specifications, including processing throughput, latency, and numerical accuracy. Consider, for example, the throughput requirement. At one end of the architecture spectrum, and when modest processing rates are involved, a fully folded [27] implementation, where the same logic is used for all iterations (folding factor $= n$), is one option. In this case, new operands are delivered, and a new result vector is produced, every n clock cycles. This choice of implementation results in the smallest FPGA footprint at the expense of processing rate. If a high-throughput unit is required, a fully parallel, or completely unfolded implementation (folding factor = 1) that allocates a complete hardware PE to each iteration is appropriate. This will of course result in the largest area, but provides the highest compute rate.

25.3.1 Convergence

One of the design considerations for the CORDIC engine is the region of convergence that needs to be supported by the implementation, as the basic form of the algorithm does not converge for all input coordinates. For the rotation mode, the CORDIC algorithm converges provided that the absolute value of the rotation angle is no larger than $\theta_{MAX} \approx 1.7433$ radians, or approximately 99.88°.

In many applications we need to support input arguments that span all four quadrants of the complex plane—that is, a so-called *full-range* CORDIC. Much published work addresses this requirement [8, 19, 25], and many elegant extensions to the basic set of CORDIC iterations have been produced. Some of them introduce additional iterations and, while maintaining the basic shift-and-add property of the algorithm, result in a significant time or area penalty.

The most straightforward approach for handling the convergence issue in FPGA hardware is to first note that the natural range of convergence extends beyond the angle $\pi/2$. That is, the basic set of equations converges over the interval $[-\pi/2, \pi/2]$. To extend the implementation to converge over $[-\pi, \pi]$, we can simply detect when the input angle extends beyond the first quadrant, map that angle to either the first or fourth quadrants, and make a post-micro-rotation correction to account for the input angle mapping. This architecture is illustrated in Figure 25.7.

The input mapping is particularly simple. Referring to Figure 25.7, if x_0 is negative, the quadrants must be changed by applying $a \pm \pi/2$ ($\pm 90°$) rotation. Whether it is a positive or negative rotation is determined by the sign of y_0. To compensate for the input mapping, an angle rotation is conditionally applied to the micro-rotation engine result z'_n to produce the final output value z_n. Details

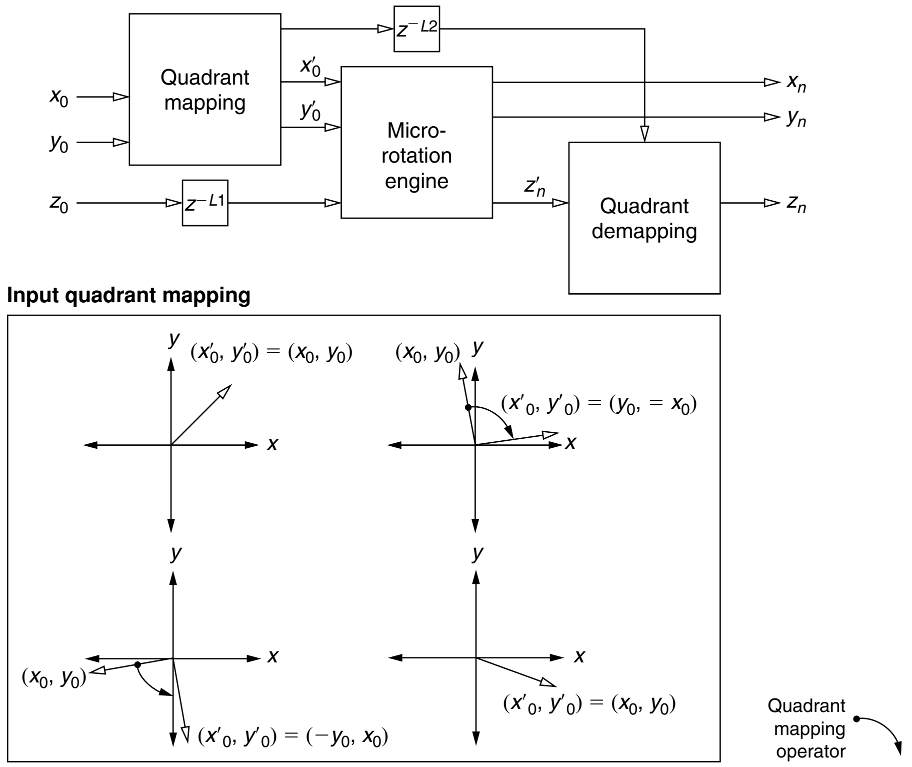


FIGURE 25.7 ■ A full-range CORDIC processor showing input quadrant mapping, micro-rotation engine, and quadrant correction.

of the course angle rotator and matching quadrant correction circuit are shown in Figure 25.8. The area cost for an FPGA implementation of the circuits is modest [40].

25.3.2 Folded CORDIC

The folded CORDIC architecture allocates a single PE to service all of the required micro-rotations. At one architectural extreme a bit-serial implementation employing a single 3-2 full adder, with appropriate control circuitry and state storage, can address all of the required updates for x, y, and z. However, our treatment employs a word-oriented architecture that associates unique functional units (FU) with each of the x, y, and z processing engines, as shown in Figure 25.9.

Multiple mapping options are available when projecting the dependency graph onto an FPGA architecture. In the Xilinx Virtex-4 family [41], one option for supporting the adder/subtractor FUs is to utilize the logic fabric and realize these modules at the cost of one lookup table (LUT) per result digit. So for example, the addition of two 16-bit operands to generate a 17-bit sum requires 17 LUTs. An alternative is to use the 48-bit adder in the DSP48 tile.

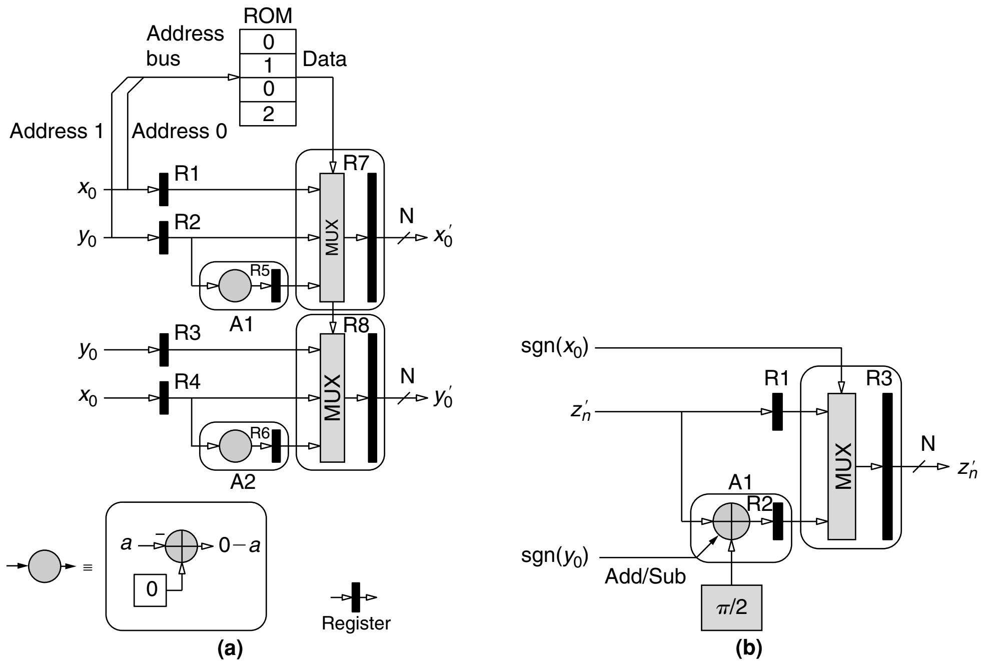


FIGURE 25.8 ■ A course angle rotator preceding a micro-rotation engine for a full-range CORDIC processor (a). A post-micro-rotation quadrant correction circuit (b).

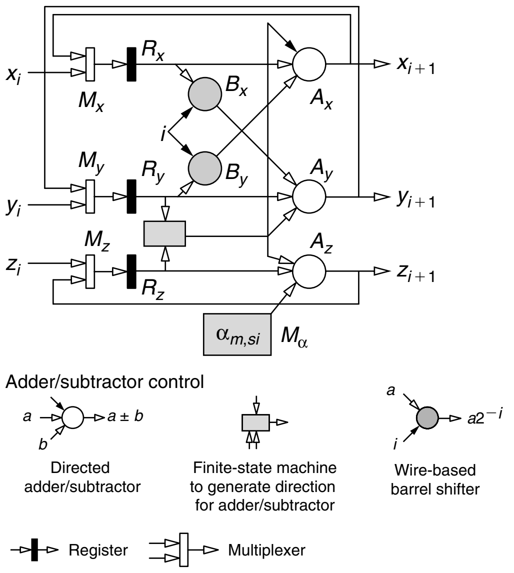


FIGURE 25.9 ■ A folded CORDIC architecture with separate functional units for each of the x, y, and z updates. Only the micro-rotation engine is shown.

There are also several mapping options for the barrel shifter: It can be realized in the logic fabric, with the multiplier in the DSP48 tile, or, for that matter, using an embedded multiplier in any FPGA family that supports this architectural component (e.g., Virtex-II Pro [39] or Spartan-3E [37]).

Consider a fabric-only implementation of a vectoring CORDIC algorithm in circular coordinates. In this case all of the FUs are implemented directly in the logic fabric. The FPGA area, A_F, can be expressed as

$$A_F = 3 \cdot a_{add} + 2 \cdot a_{barrel} + 3 \cdot a_{mux} + a_{LUT} + a_Q + a_{Q^{-1}} \tag{25.20}$$

where a_{add}, a_{barrel}, a_{mux}, a_{LUT}, a_Q, and $a_{Q^{-1}}$ correspond to the area of an adder, barrel shifter, input multiplexer, elementary angle LUT, quadrant input mapper, and output mapper circuits, respectively. The FPGA logic fabric is designed to efficiently support the implementation of arbitrary-precision high-speed adder/subtractors. Each configurable logic block (CLB) [41] includes dedicated circuitry that provides fast carry resolution, with the LUT itself producing the half-sum.

The component that can be costly in terms of area is the barrel shifter. The barrel shifter area cost can be much more significant than the aggregate cost of the adder/subtractors used for updating the x, y, and z variables. For example, in a design that supplies 16 effective result digits, the 2 barrel shifters occupy an aggregate area of 226 LUTs while the adders occupy 74 LUTs in total. Here, the barrel shifters have a footprint approximately three times that of the adders.

The barrel shifter area can be reduced if a multiplier-based barrel shifter is used rather than a purely logic fabric–based implementation. FPGA families such as Spartan-3E [37], Virtex-II Pro [39], and Virtex-4 [40] include an array of embedded multipliers, which are useful for realizing arithmetic shifts. The multiplier accepts 18-bit precision operands and produces a 36-bit result. When used as a barrel shifter, one port of the multiplier is supplied with the input operand that is to experience the arithmetic shift, while the second port accepts the shift value 2^i, where i is the iteration index. In a typical hardware implementation the iteration index rather than the exponentiated value is usually available in the control plane that coordinates the operation of the circuit. The exponentiation can be done via a small LUT implemented using distributed memory [40]. Multiple multiplier primitives can be combined with an adder to form a barrel shifter that can support a wider datapath. For the previous example, multiplier realization of the barrel shifter results in an FPGA footprint that is less than half that of an entirely fabric-based implementation.

The folded CORDIC architecture is a recursive graph, which means that deep pipelining cannot be employed to reduce the critical path. The structure can accept a new set of operands, and produces a result every n clock cycles.

25.3.3 Parallel Linear Array

When throughput is the overriding design consideration, a fully parallel pipelined CORDIC realization is the preferred architecture. With this approach

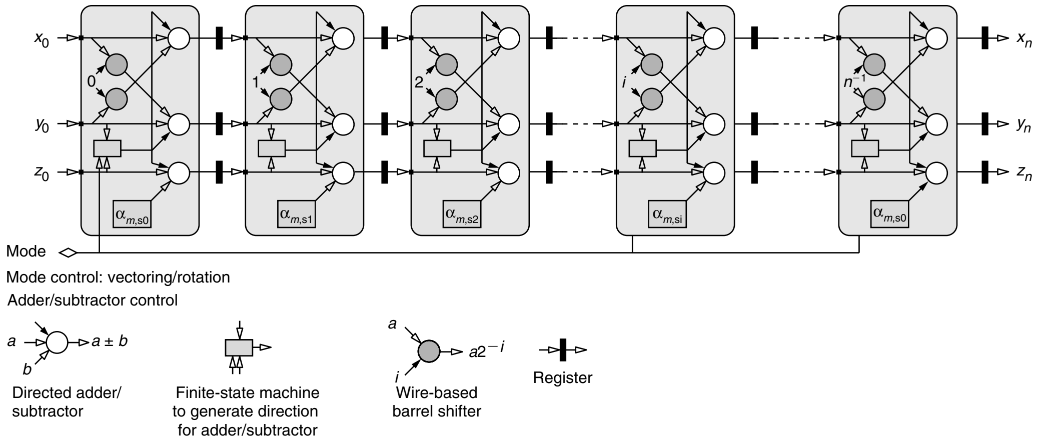


FIGURE 25.10 ■ A programmable parallel pipelined CORDIC array. In a completely unfolded implementation, the barrel shifters are realized as FPGA routing and so consume no resources other than interconnect.

the CORDIC algorithm is completely unrolled and each operation is projected onto a unique hardware resource, as shown in Figure 25.10.

One interesting effect of the unrolling is that the data shifts required in the cross-addition update can be realized as wiring between successive CORDIC processing elements (PEs). Unlike the folded architecture, where either LUTs or embedded multipliers are consumed to realize the barrel shifter, no resources other than interconnect are required to implement the shift in the linear array architecture. The only functional units required for each PE with this approach are three adder/subtractors and a small amount of logic to implement the control circuit that steers the add/subtract FUs. The micro-rotation angle for each PE is encoded as a constant supplied on one arm of the adder/subtractor that performs the angle update—no LUT resources are required for this. Note in Figure 25.10 that the sign bit of the y and z variables is supplied to the control circuit that is local to each processing engine. This permits the architecture to operate in the y- or z-reduction configuration under the control of the Mode input control signal, and thus support vectoring or rotation, respectively.

Figure 25.11(a) shows a comparison of the area functions for the parallel and folded architectures. The folded implementation is entirely fabric based. As expected, the area of the parallel design exhibits modest exponential growth and, for an effective number of result digits greater than 15, occupies more than three times the area of the folded architecture. For the case of 24 effective result digits, the parallel design is larger by a factor of approximately 5. Figure 25.11(b) contrasts the throughput of the two architectures. Naturally, the parallel design has a constant throughput of one CORDIC operation per second for a normalized clock rate of 1, while the throughput for the folded design falls off as the inverse of the number of iterations.

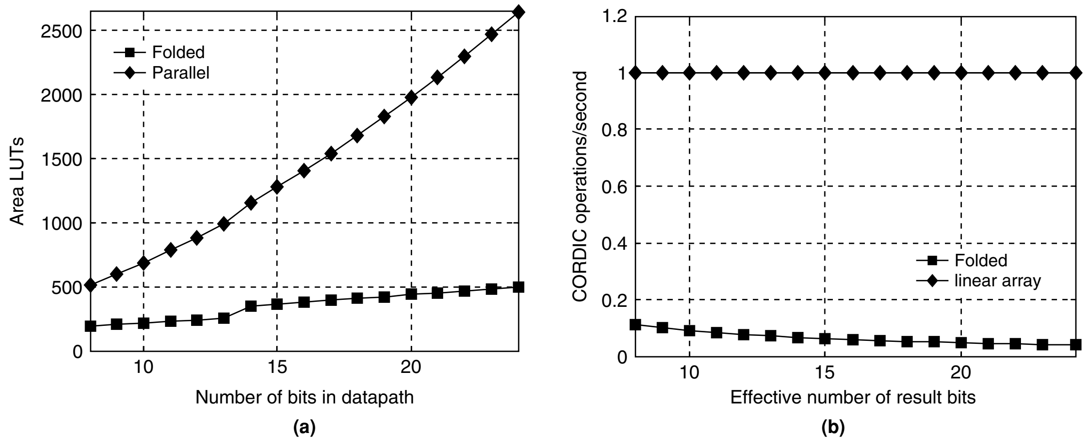
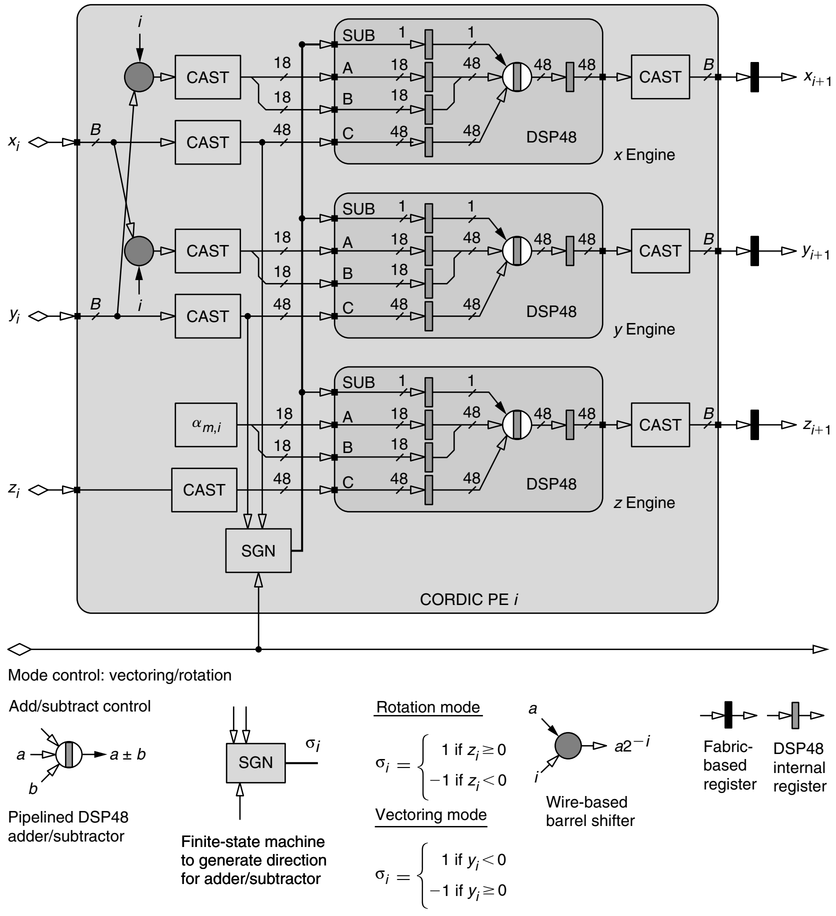


FIGURE 25.11 ■ (a) Comparison of the FPGA resource requirements for folded and linear array CORDIC architectures—circular coordinates. (b) Throughput in rotations/vectoring operations per second for the two architectures. A normalized clock rate of 1 is assumed.

The parallel design has a performance advantage of approximately an order of magnitude for the number of effective result bits great than 10. In an FPGA implementation the advantage is significantly more than this because of the higher clock frequency that can be supported by the linear array compared to the folded processor. With its heavy pipelining, the linear array typically achieves an operating frequency approximately twice that of the folded architecture, so for high-precision calculations—for example, on the order of 24 effective fractional bits or greater—the parallel implementation has a throughput advantage of approximately 50, which is delivered in a footprint that is only five times that of the folded design.

The add/subtract FUs can be realized using the logic fabric or the 48-bit adder that is resident in each DSP48 tile in the Virtex-4 class of FPGAs. The DSP48 [42] is a dynamically configurable embedded processing block that supports over 40 different op-codes, optimized for signal-processing tasks. The logic fabric approach tends to result in an implementation that operates at a lower clock frequency than a fully pipelined version based on the DSP48. The DSP48-based implementations can operate at very high clock frequencies—in the region of 500 MHz in the fastest "–12" speed-grade parts [40]. However, for a datapath precision of up to 36 bits, three DSP48 tiles are required for each CORDIC iteration (see Figures 25.12 and 25.13). For scenarios where throughput is the overarching requirement, these resource requirements are acceptable.

A potential downside to the use of the DSP48 in this application is that the multiplier colocated with the high-precision adder is not available for use by another function if the adder is used by the CORDIC PE. This is because the input and output ports of the block are occupied supporting the addition/subtraction and there is no I/O available to access other functions (such as the multiplier) in the tile.



x_i
y_i
z_i
B
i
CAST
$\alpha_{m,i}$
SUB
A
B
C
18
48
1
DSP48
x Engine
y Engine
z Engine
x_{i+1}
y_{i+1}
z_{i+1}
SGN
CORDIC PE i
Mode control: vectoring/rotation
Add/subtract control
a
b
$a \pm b$
Pipelined DSP48 adder/subtractor
σ_i
Finite-state machine to generate direction for adder/subtractor
Rotation mode
$\sigma_i = \begin{cases} 1 \text{ if } z_i \geq 0 \\ -1 \text{ if } z_i < 0 \end{cases}$
Vectoring mode
$\sigma_i = \begin{cases} 1 \text{ if } y_i < 0 \\ -1 \text{ if } y_i \geq 0 \end{cases}$
$a2^{-i}$
Wire-based barrel shifter
Fabric-based register
DSP48 internal register

FIGURE 25.12 ■ Processing element i of a Virtex-4 DSP48-based CORDIC processor.

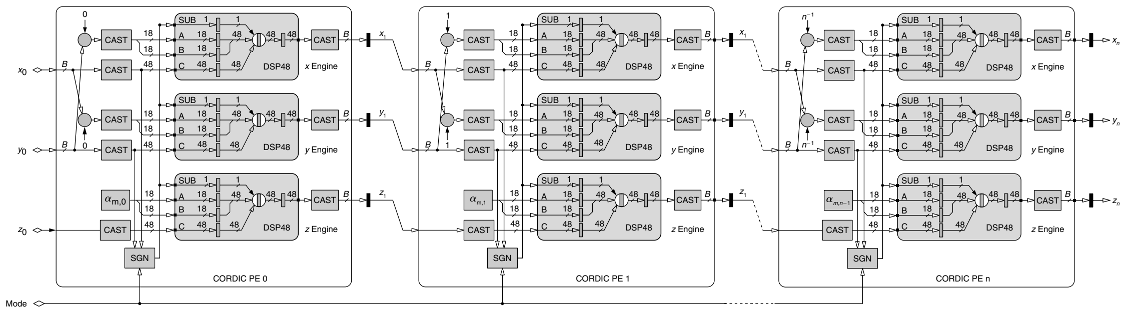


FIGURE 25.13 ■ A programmable parallel pipelined CORDIC array based almost entirely on the Virtex-4 DSP48 embedded tile. Each DSP48 has three levels of pipelining. Additional fabric-based registers are included to pipeline the routing between DSP48 tiles.

25.3.4 Scaling Compensation

As highlighted earlier, the rotation mode of the CORDIC algorithm produces a rotation extension (i.e., it increases or decreases the distance of the point from the origin) rather than a pure rotation. The growth associated with circular and hyperbolic coordinate systems is approximately $K_{1,n} \approx 1.6468$ and $K_{-1,n} \approx 0.8382$, respectively. In some applications this growth can be tolerated, and there is no need to perform any compensation. For example, if the vectoring mode is used to map the output vector of a discrete Fourier transform (DFT) from Cartesian to polar coordinates in order to compute a magnitude spectrum, the CORDIC scaling may not be an issue because all terms are similarly scaled. If the CORDIC output is to be further processed, there might be an opportunity to absorb the CORDIC scale factor in the postprocessing circuit. Continuing with the DFT example, if the magnitude spectrum is to be compared with a threshold in order to make a decision about a particular spectral bin, the CORDIC scaling can be absorbed into the threshold value.

If the scaling cannot be tolerated, several scaling compensation techniques are possible. Some approaches employ modified iterations [20, 32, 33] while others exploit alternatives such as online arithmetic [6]. Some methods merge scaling iterations with the basic CORDIC iterations [15], which result in either an area penalty or a time penalty if the basic CORDIC hardware is to be used for both the fundamental updates and the scaling iterations. It is also possible to employ a modified set of elemental angles [9].

The problem of scaling compensation has been examined by many researchers, and many creative and elegant results have been produced; however, the most direct way to accommodate the problem in an FPGA is to employ its embedded multipliers. The architecture of a programmable and scale-compensated CORDIC engine is shown in Figure 25.14. The `Mode` control signal defines if a vectoring or rotation operation is to be performed. It essentially controls if the iteration update is guided by the sign of the y or z variable for vectoring or rotation, respectively. The `Coordinate_System` signal selects the coordinate system for the processor: circular, hyperbolic, or linear. This control line selects the page in memory where the elemental angles are stored: $\tan^{-1}(2^{-i})$, $i = 0, \ldots, n-1$ for circular; $\tanh^{-1}(2^{-i})$, $i = 1, \ldots, n$ for hyperbolic; and (2^{-i}), $i = 0, \ldots, n-1$ for linear. `Coordinate_System` also indexes a small memory located in FPGA distributed memory that stores the values $1/K_{m,n}$ for use by the scaling compensation multiplier $M1$. Naturally, the precision of these constants should be commensurate with the number of effective result bits.

25.4 SUMMARY

This chapter provided an overview of the CORDIC algorithm and its implementation in current-generation FPGAs such as the Xilinx Virtex-4 family. The basic set of CORDIC equations was first reviewed, and the utility of this simple shift-and-add-type algorithm was highlighted by the many functions that can be accessed through it. We also highlighted the fact that, while there are many options for architecting math functions in hardware, the CORDIC approach

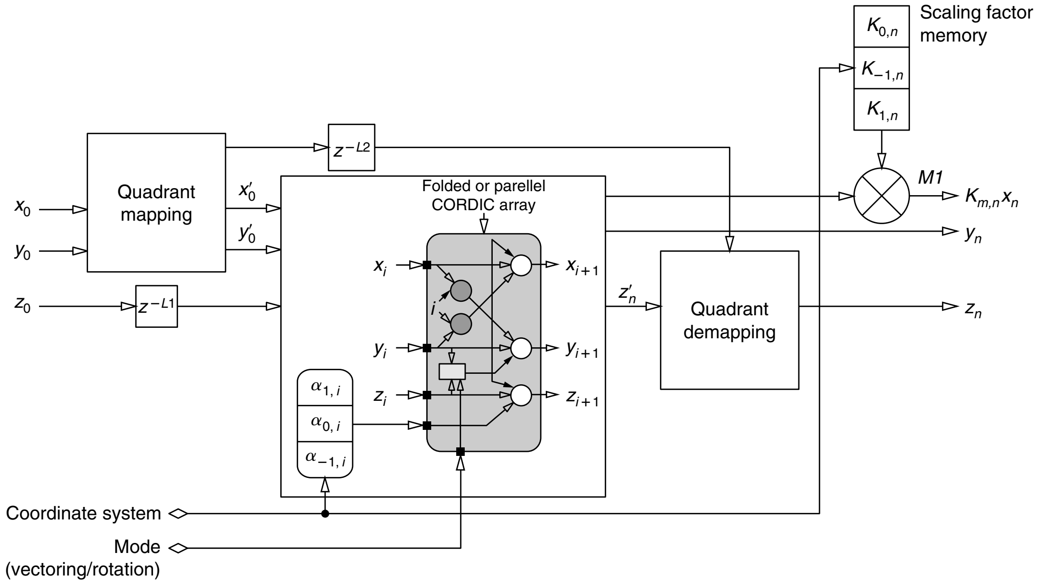


FIGURE 25.14 ■ A programmable CORDIC processor with multiplier-based scaling compensation.

comes into its own when multi-element input and output vectors are involved. The functional requirements of the angle and cross-addition updates make it an excellent match for FPGAs because of the utility and efficiency with which these devices realize addition and subtraction.

Most hardware realizations of the CORDIC algorithm employ fixed-point arithmetic, and this is certainly true of nearly all FPGA implementations. We showed that it is therefore important to understand the effects of quantizing the datapath. While this analysis can be complex [16], for most applications the simplified approach first described by Walther [36] is suitable for most cases and provides excellent results.

The FPGA implementation of a CORDIC processor would appear to be straightforward. However, FPGA-embedded functions such as multipliers and the DSP48 provide opportunities for architectural innovation and for design trade-offs that satisfy design requirements. For example, embedded multipliers can be exchanged for logic fabric with the implementation of the barrel shifter. The wide 48-bit adder in the DSP48 can be used almost as the sole arithmetic building block of a complete fully parallel CORDIC array.

References

[1] J. R. Cavallaro, F. T. Luk. CORDIC arithmetic for an SVD processor. *Journal of Parallel and Distributed Computing* 5, 1988.

[2] J. R. Cavallaro, F. T. Luk. Floating-point CORDIC for matrix computations. *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1988.

[3] L. W. Chang, S. W. Lee. Systolic arrays for the discrete Hartley transform. *IEEE Transactions on Signal Processing* 29(11), November 1991.

[4] W. H. Chen, C. H. Smith, S. C. Fralick. A fast computational algorithm for the discrete cosine Transform. *IEEE Transactions on Communications* C-25, September 1977.

[5] Cray Research. *Cray XD1 Supercomputer*, *http://www.cray.com/products/xd1/index.html*.

[6] H. Dawid, H. Meyer. The differential CORDIC algorithm: Constant scale factor redundant implementation without correcting iterations. *IEEE Transactions on Computers* 45(3), March 1996.

[7] A. A. J. de Lange, A. J. van der Hoeven, E. F. Deprettere, J. Bu. An optimal floating-point pipeline CMOS CORDIC processor. *IEEE Symposium on Circuits and Systems*, June 1988.

[8] J. M. Delsme. VLSI implementation of rotations in pseudo-Euclidean spaces. *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* 2, 1983.

[9] E. Deprettere, P. Dewilde, R. Udo. Pipelined CORDIC architectures for fast VLSI filtering and array processing. *Proceedings of the ICASSP'84*, 1984.

[10] A. M. Despain. Very fast Fourier transform algorithms for hardware implementation. *IEEE Transactions on Computers* C-28, May 1979.

[11] A. M. Despain. Fourier transform computers using CORDIC iterations. *IEEE Transactions on Computers* 23, October 1974.

[12] C. Dick, F. Harris, M. Rice. FPGA implementation of carrier phase synchronization for QAM demodulators. *Journal of VLSI Signal Processing, Special Issue on Field-Programmable Logic* (R. Woods, R. Tessier, eds.), Kluwer Academic, January 2004.

[13] D. Ercegovac, T. Lang. *Digital Arithmetic*, Morgan Kaufmann, 2004.

[14] B. Haller, J. Gotze, J. Cavallaro. Efficient implementation of rotation operations for high-performance QRD-RLS filtering. *Proceedings of the International Conference on Application-Specific Systems, Arthictectures and Processors*, July 1997.

[15] G. H. Haviland, A. A. Tuszinsky. A CORDIC arithmetic processor chip. *IEEE Transactions on Computers* c-29(2), February 1980.

[16] Y. H. Hu. The quantization effects of the CORDIC algorithm. *IEEE Transactions on Signal Processing* 40, July 1992.

[17] X. Hu, S. C. Bass. A neglected error source in the CORDIC algorithm. *IEEE International Symposium on Circuits and Systems* 1, May 1993.

[18] X. Hu, S. C. Bass. A neglected error source in the CORDIC algorithm. *Proceedings of the IEEE ISCAS*, 1993.

[19] X. Hu, R. G. Garber, S. C. Bass. Expanding the range of convergence of the CORDIC algorithm. *IEEE Transactions on Computers* 40(1), January 1991.

[20] J. Lee. Constant-factor redundant CORDIC for angle calculation and rotation. *IEEE Transactions on Computers* 41(8), August 1992.

[21] Y. H. Liao, H. E. Liao. CALF: A CORDIC adaptive lattice filter. *IEEE Transactions on Signal Processing* 40(4), April 1992.

[22] Mathworks, The, *http://www.mathworks.com/*.

[23] U. Mengali, A. N. D'Andrea. *Synchronization Techniques for Digital Receivers*, Plenum Press, 1997.

[24] J. Mia, K. K. Parhi, E. F. Deprettere. Pipelined implementation of CORDIC-based QRD-MVDR adaptive beamforming. *IEEE Fourth International Conference on Signal Processing*, October 1998.

[25] J. M. Muller. Discrete basis and computation of elementary functions. *IEEE Transactions on Computers* C-34(9), September 1985.

[26] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.
[27] K. K. Parhi. *VLSI Digital Signal Processing Systems Design and Implementation*, John Wiley, 1999.
[28] S. Y. Park, N. I. Cho. Fixed-point error analysis of CORDIC processor based on the Variance Propagation Formula. *IEEE Transactions on Circuits and Systems* 51(3), March 2004.
[29] J. G. Proakis, M. Salehi. *Communication Systems Engineering*, Prentice-Hall, 1994.
[30] C. M. Rader. VLSI systolic arrays for adaptive nulling. *IEEE Signal Processing Magazine* 13(4), July 1996.
[31] T. Y. Sung, Y. H. Hu. Parallel VLSI implementation of Kalman filter. *IEEE Transactions on Aerospace and Electronic Systems* AES 23(2), March 1987.
[32] N. Takagi. Redundant CORDIC methods with a constant scale factor for sine and cosine computation. *IEEE Transactions on Computers* 40(9), September 1991.
[33] D. H. Timmerman, B. J. Hosticka, B. Rix. A new addition scheme and fast scaling factor compensation methods for CORDIC algorithms. *Integration, the VLSI Journal* (11), 1991.
[34] D. H. Timmerman, B. J. Hosticka, G. Schmidt. A programmable CORDIC chip for digital signal processing applications. *IEEE Journal of Solid-State Circuits* 26(9), September 1991.
[35] J. E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers* 3, September 1959.
[36] J. S. Walther. A unified algorithm for the elementary functions. *AFIPS Spring Joint Computer Conference* 38, 1971.
[37] Xilinx Inc. Spartan-3E Datasheet, *http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category= /Data+Sheets/FPGA+Device+Families/Spartan-3E*.
[38] Xilinx Inc. System Generator for DSP, *http://www.xilinx.com/ise/optional_prod/system_generator.htm*.
[39] Xilinx Inc. Virtex-II Pro Datasheet, *http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?category=Publications/FPGA+Device+Families/Virtex-II+Pro&iLanguageID=1*.
[40] Xilinx Inc. Virtex-4 Datasheet, *http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?sGlobalNavPick=&sSecondaryNavPick=&category=-1210771&iLanguage ID=1*.
[41] Xilinx Inc. Virtex-4 Multi-Platform FPGA, *http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm*.
[42] Xilinx Inc. XtremeDSP Design Considerations Guide, *http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/xtremedsp.htm*.
[43] Xilinx Inc. XtremeDSP Slice, *http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/xtremedsp.htm*.