



16 décembre 2019  
M. Arzel

# TAF SEH UEC 2

## Travaux Pratiques de conception d'architectures numériques

Opérateur CORDIC : de l'algorithme à la puce



**IMT Atlantique**  
Bretagne-Pays de la Loire  
École Mines-Télécom

# Table des matières



<b>1</b>	<b>Problème à traiter</b>	<b>3</b>
1.1	Intégration de la fonction Sinus à l'aide de l'opérateur CORDIC . . . . .	4
1.2	Votre mission . . . . .	6
1.2.1	Fournir un travail préparatoire (prévoir plus d'une heure!) . . . . .	6
1.2.1.1	Prendre en main l'opérateur Cordic . . . . .	6
1.2.1.2	Etablir l'architecture de la fonction sinus . . . . .	6
1.2.2	Décrire l'architecture en VHDL . . . . .	6
1.2.3	Tester l'architecture . . . . .	6
1.2.4	Résumer les performances d'intégration . . . . .	6
<b>2</b>	<b>Un article de référence sur le CORDIC</b>	<b>7</b>

# Chapitre 1

## Problème à traiter



## 1.1 Intégration de la fonction Sinus à l'aide de l'opérateur CORDIC

**Objectifs** Maîtriser la conception et la réalisation d'architectures numériques en intégrant des fonctions mathématiques avancées.

**Cahier des charges** Développer en VHDL un module de calcul du sinus pour une précision de sortie de  $10^{-6}$ . L'architecture proposée devra soit consommer le moins de ressources FPGA soit atteindre le débit maximal sur FPGA. Dans tous les cas, votre module sinus échangera avec des modules en amont et en aval selon un protocole de *hand-shake*. Chaque interface d'un composant est constitué de 2 signaux de contrôle (*\_rts* et *\_cts*) et du bus de données. L'émetteur et le récepteur se mettent d'accord pour un transfert de données en série, avec un seul échantillon par cycle d'horloge. Un élément qui est prêt à accepter des données sur une interface d'entrée positionnera le signal *\_cts* correspondant à '1'. Un élément qui a des données à transmettre sur une interface de sortie positionne le signal *\_rts* correspondant à '1'. Lorsque, sur un front montant d'horloge, les deux signaux d'une interface valent '1', on considère que les deux acteurs sont d'accord pour transmettre, et on décide que la transmission est valide. On peut alors passer à l'échantillon suivant.

**Pré-requis** En annexe est fourni un article décrivant l'opérateur Coordinate Rotation Digital Computer (CORDIC) permettant une implantation efficace de nombreuses fonctions trigonométriques. Le principe élémentaire est celui de la rotation d'un vecteur  $V$  de coordonnées  $(x, y)$  d'un angle  $\phi$ . Si  $\phi$  peut s'exprimer sous la forme  $\sum_{i=0}^{n-1} d_i \phi_i$  où  $d_i \in \{-1, +1\}$  et  $\{\phi_i\}_{i \in [0, n-1]}$  est une base d'angles de plus en plus petits lorsque  $i$  croît, alors la rotation par  $\phi$  se décompose en  $n$  rotations élémentaires de plus en plus précises. Les coordonnées  $(x', y')$  du vecteur résultat  $V'$  sont alors

$$x' = x \cos(\phi) - y \sin(\phi) \quad (1.1)$$

$$y' = x \sin(\phi) + y \cos(\phi) \quad (1.2)$$

donc aussi

$$x' = \cos(\phi)(x - y \tan(\phi)) \quad (1.3)$$

$$y' = \cos(\phi)(y + x \tan(\phi)) \quad (1.4)$$

soit de manière récursive pour tout  $i \in [1, n-1]$

$$x_i = \cos(\phi_i)(x_{i-1} - y_{i-1} \tan(d_i \phi_i)) \quad (1.5)$$

$$y_i = \cos(\phi_i)(y_{i-1} + x_{i-1} \tan(d_i \phi_i)) \quad (1.6)$$

et

$$x_0 = x \quad (1.7)$$

$$y_0 = y \quad (1.8)$$

Pour profiter au mieux du traitement numérique en base binaire, la base  $\{\phi_i\}$  est choisie telle que

$$\phi_i = \tan^{-1}(2^{-i}) \quad (1.9)$$

ce qui permet de transformer les équations (1.5) et (1.6) en

$$x_i = \cos(\tan^{-1}(2^{-i}))(x_{i-1} - d_i y_{i-1} 2^{-i}) \quad (1.10)$$

$$y_i = \cos(\tan^{-1}(2^{-i}))(y_{i-1} + d_i x_{i-1} 2^{-i}) \quad (1.11)$$

Les opérations requises sont alors *in fine* des successions d'additions et de décalage à droite et une mise à l'échelle finale par  $\prod_{i=0}^{n-1} \cos(\tan^{-1}(2^{-i}))$ . Tout cela suppose que l'on

fait le bon choix de valeur pour chaque  $d_i$ , ce qui peut être intégré à l'algorithme comme un procédé dichotomique de recherche d'angle (cf. section 1.2.4). Si  $\phi$  ne peut pas s'exprimer sous la forme  $\sum_{i=0}^{n-1} d_i \tan^{-1}(2^{-i})$ , on prendra alors l'approximation la plus proche sous cette forme et l'on déroulera la récursion précédente. Par exemple,

$$56^\circ \approx 45.00000^\circ + 26.56505^\circ - 14.03624^\circ - 7.12502^\circ \\ + 3.57633^\circ + 1.78991^\circ + 0.89517^\circ - 0.44761^\circ \quad (1.12)$$

$$\approx \tan^{-1}(2^0) + \tan^{-1}(2^{-1}) - \tan^{-1}(2^{-2}) - \tan^{-1}(2^{-3}) \\ + \tan^{-1}(2^{-4}) + \tan^{-1}(2^{-5}) + \tan^{-1}(2^{-6}) - \tan^{-1}(2^{-7}) \quad (1.13)$$

En jouant avec cet opérateur, il est possible d'obtenir de manière récursive, outre les coordonnées issues de rotations, les résultats (approchés) des fonctions trigonométriques, ceux de leurs inverses et ceux des fonctions trigonométriques hyperboliques.

Une implantation Matlab/Octave pour tester le sinus en CORDIC : *cordic\_sin.m*

```

1 function [res] = cordic_sin(val,n)
2 %on suppose que val est compris entre -Pi/2 et Pi/2
3 x=1;
4 y=0;
5 a=1;
6 %quantification sur n bits apres la virgule
7 z= round((val)*2^n)/2^n;
8 for k=0:n-1
9     d = sign(z);
10    xp=x - y*d*2^(-k);
11    yp=y + x*d*2^(-k);
12    x=xp;
13    y=yp;
14    z=z-d*atan(2^(-k));
15    a = a / sqrt(1+2^(-2*k));
16 end;
17
18 %facteur d'echelle pour VHDL
19 a= round(a*2^n);
20 %le facteur d'echelle est reajuste en virgule fixe
21 res = a/2^n*(y);

```

**Question préliminaire** Quelle alternative peut-on envisager à l'utilisation du CORDIC ? Les précisions seraient-elles finalement comparables ?

## 1.2 Votre mission

### 1.2.1 Fournir un travail préparatoire (prévoir plus d'une heure !)

#### 1.2.1.1 Prendre en main l'opérateur Cordic

Pour ce faire, vous devez

- produire une courbe de référence de l'opérateur Cordic appliqué au sinus ;
- produire l'environnement de test pour votre implantation VHDL, ce qui suppose :
  - élaborer un testbench VHDL qui lira des fichiers produits via Matlab/Octave, et donc
  - produire des fichiers d'entrée et sortie pour valider votre description VHDL ultérieure du Cordic.

#### 1.2.1.2 Etablir l'architecture de la fonction sinus

Attention, il existe des questions élémentaires à vous poser :

- quels opérateurs ?
- quelles largeurs pour les données ?
- des limitations sur les entrées ? si oui, des solutions ?
- le chemin critique ?
- *Pipe-line or not pipe-line ?*
- des paramètres de généricité ?
- etc.

### 1.2.2 Décrire l'architecture en VHDL

Réalisez une implantation générique en VHDL, en isolant clairement les unités de contrôle, opérative et de communication avec les interfaces, comme l'architecture préalable l'aura décrit. Ce que vous ne pouvez pas dessiner avec des "boîtes" vous ne pourrez pas décrire en VHDL.

### 1.2.3 Tester l'architecture

Validez par plusieurs simulations en réutilisant le testbench que votre travail préparatoire a fourni. Ces tests permettront de s'assurer que votre architecture est fonctionnelle. Ne négligez pas cette partie !

### 1.2.4 Résumer les performances d'intégration

Supposez que votre architecture est intégrée sur une carte de type Digilent Nexys Video (FPGA de type Artix 7 A7-200), et faites le bilan des ressources requises et de la fréquence maximale de fonctionnement. Votre solution correspond-elle à l'objectif que vous aviez choisi ?

## **Chapitre 2**

### **Un article de référence sur le COR-DIC**





# A survey of CORDIC algorithms for FPGA based computers

Ray Andraka  
Andraka Consulting Group, Inc  
16 Arcadia Drive  
North Kingstown, RI 02852  
401/884-7930 FAX 401/884-7950  
email:randraka@ids.net

## 1. ABSTRACT

**The current trend back toward hardware intensive signal processing has uncovered a relative lack of understanding of hardware signal processing architectures. Many hardware efficient algorithms exist, but these are generally not well known due to the dominance of software systems over the past quarter century. Among these algorithms is a set of shift-add algorithms collectively known as CORDIC for computing a wide range of functions including certain trigonometric, hyperbolic, linear and logarithmic functions. While there are numerous articles covering various aspects of CORDIC algorithms, very few survey more than one or two, and even fewer concentrate on implementation in FPGAs. This paper attempts to survey commonly used functions that may be accomplished using a CORDIC architecture, explain how the algorithms work, and explore implementation specific to FPGAs.**

### 1.1 Keywords

CORDIC, sine, cosine, vector magnitude, polar conversion

## 2. INTRODUCTION

The digital signal processing landscape has long been dominated by microprocessors with enhancements such as single cycle multiply-accumulate instructions and special addressing modes. While these processors are low cost and offer extreme flexibility, they are often not fast enough for truly demanding DSP tasks. The advent of reconfigurable logic computers permits the higher speeds of dedicated hardware solutions at costs that are competitive with the traditional software approach. Unfortunately, algorithms optimized for these microprocessor based systems do not usually map well into hardware. While hardware-efficient solutions often exist, the dominance of the software systems has kept those solutions out of the spotlight. Among these hardware-efficient algorithms is a class of iterative solutions for trigonometric and other

transcendental functions that use only shifts and adds to perform. The trigonometric functions are based on vector rotations, while other functions such as square root are implemented using an incremental expression of the desired function. The trigonometric algorithm is called CORDIC, an acronym for COordinate Rotation Digital Computer. The incremental functions are performed with a very simple extension to the hardware architecture, and while not CORDIC in the strict sense, are often included because of the close similarity. The CORDIC algorithms generally produce one additional bit of accuracy for each iteration.

The trigonometric CORDIC algorithms were originally developed as a digital solution for real-time navigation problems. The original work is credited to Jack Volder [4,9]. Extensions to the CORDIC theory based on work by John Walther[1] and others provide solutions to a broader class of functions. The CORDIC algorithm has found its way into diverse applications including the 8087 math coprocessor[7], the HP-35 calculator, radar signal processors[3] and robotics. CORDIC rotation has also been proposed for computing Discrete Fourier[4], Discrete Cosine[4], Discrete Hartley[10] and Chirp-Z [9] transforms, filtering[4], Singular Value Decomposition[14], and solving linear systems[1].

This paper attempts to survey the existing CORDIC and CORDIC-like algorithms with an eye toward implementation in Field Programmable Gate Arrays (FPGAs). First a brief description of the theory behind the algorithm and the derivation of several functions is presented. Then the theory is extended to the so-called unified CORDIC algorithms, after which implementation of FPGA CORDIC processors is discussed.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

FPGA 98 Monterey CA USA

Copyright 1998 ACM 0-89791-978-5/98/01..\$5.00

## 3. CORDIC THEORY: AN ALGORITHM FOR VECTOR ROTATION

All of the trigonometric functions can be computed or derived from functions using vector rotations, as will be discussed in the following sections. Vector rotation can also be used for polar to rectangular and rectangular to polar conversions, for vector magnitude, and as a building block in certain transforms such as the DFT and DCT. The CORDIC algorithm provides an iterative method of performing vector rotations by arbitrary angles using only shifts and adds. The algorithm, credited to Volder[4], is derived from the general (Givens) rotation transform:

$$x' = x \cos \phi - y \sin \phi$$

$$y' = y \cos \phi + x \sin \phi$$

which rotates a vector in a Cartesian plane by the angle  $\phi$ . These can be rearranged so that:

$$x' = \cos \phi \cdot [x - y \tan \phi]$$

$$y' = \cos \phi \cdot [y + x \tan \phi]$$

So far, nothing is simplified. However, if the rotation angles are restricted so that  $\tan(\phi) = \pm 2^{-i}$ , the multiplication by the tangent term is reduced to simple shift operation. Arbitrary angles of rotation are obtainable by performing a series of successively smaller elementary rotations. If the decision at each iteration,  $i$ , is which direction to rotate rather than whether or not to rotate, then the  $\cos(\delta_i)$  term becomes a constant (because  $\cos(\delta_i) = \cos(-\delta_i)$ ). The iterative rotation can now be expressed as:

$$x_{i+1} = K_i [x_i - y_i \cdot d_i \cdot 2^{-i}]$$

$$y_{i+1} = K_i [y_i + x_i \cdot d_i \cdot 2^{-i}]$$

where:

$$K_i = \cos(\tan^{-1} 2^{-i}) = 1/\sqrt{1+2^{-2i}}$$

$$d_i = \pm 1$$

Removing the scale constant from the iterative equations yields a shift-add algorithm for vector rotation. The product of the  $K_i$ 's can be applied elsewhere in the system or treated as part of a system processing gain. That product approaches 0.6073 as the number of iterations goes to infinity. Therefore, the rotation algorithm has a gain,  $A_n$ , of approximately 1.647. The exact gain depends on the number of iterations, and obeys the relation

$$A_n = \prod_n \sqrt{1+2^{-2i}}$$

The angle of a composite rotation is uniquely defined by the sequence of the directions of the elementary rotations. That sequence can be represented by a decision vector. The set of all possible decision vectors is an angular measurement

system based on binary arctangents. Conversions between this angular system and any other can be accomplished using a look-up. A better conversion method uses an additional adder-subtractor that accumulates the elementary rotation angles at each iteration. The elementary angles can be expressed in any convenient angular unit. Those angular values are supplied by a small lookup table (one entry per iteration) or are hardwired, depending on the implementation. The angle accumulator adds a third difference equation to the CORDIC algorithm:

$$z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-i})$$

Obviously, in cases where the angle is useful in the arctangent base, this extra element is not needed.

The CORDIC rotator is normally operated in one of two modes. The first, called rotation by Volder[4], rotates the input vector by a specified angle (given as an argument). The second mode, called vectoring, rotates the input vector to the  $x$  axis while recording the angle required to make that rotation.

In rotation mode, the angle accumulator is initialized with the desired rotation angle. The rotation decision at each iteration is made to diminish the magnitude of the residual angle in the angle accumulator. The decision at each iteration is therefore based on the sign of the residual angle after each step. Naturally, if the input angle is already expressed in the binary arctangent base, the angle accumulator may be eliminated. For rotation mode, the CORDIC equations are:

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}$$

$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$

$$z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-i})$$

where

$$d_i = -1 \text{ if } z_i < 0, +1 \text{ otherwise}$$

which provides the following result:

$$x_n = A_n [x_0 \cos z_0 - y_0 \sin z_0]$$

$$y_n = A_n [y_0 \cos z_0 + x_0 \sin z_0]$$

$$z_n = 0$$

$$A_n = \prod_n \sqrt{1+2^{-2i}}$$

In the vectoring mode, the CORDIC rotator rotates the input vector through whatever angle is necessary to align the result vector with the  $x$  axis. The result of the vectoring operation is a rotation angle and the scaled magnitude of the original vector (the  $x$  component of the result). The vectoring function works by seeking to minimize the  $y$  component of the residual vector at each rotation. The sign

of the residual y component is used to determine which direction to rotate next. If the angle accumulator is initialized with zero, it will contain the traversed angle at the end of the iterations. In vectoring mode, the CORDIC equations are:

$$\begin{aligned}x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\z_{i+1} &= z_i - d_i \cdot \tan^{-1}\left(2^{-i}\right)\end{aligned}$$

where

$$d_i = +1 \text{ if } y_i < 0, -1 \text{ otherwise.}$$

Then:

$$\begin{aligned}x_n &= A_n \sqrt{x_0^2 + y_0^2} \\y_n &= 0 \\z_n &= z_0 + \tan^{-1}\left(\frac{y_0}{x_0}\right) \\A_n &= \prod_n \sqrt{1 + 2^{-2i}}\end{aligned}$$

The CORDIC rotation and vectoring algorithms as stated are limited to rotation angles between  $-\pi/2$  and  $\pi/2$ . This limitation is due to the use of  $2^0$  for the tangent in the first iteration. For composite rotation angles larger than  $\pi/2$ , an additional rotation is required. Volder[4] describes an initial rotation  $\pm\pi/2$ . This gives the correction iteration:

$$\begin{aligned}x' &= -d \cdot y \\y' &= d \cdot x \\z' &= z + d \cdot \frac{\pi}{2}\end{aligned}$$

where  $d = +1$  if  $y < 0$ ,  $-1$  otherwise.

There is no growth for this initial rotation. Alternatively, an initial rotation of either  $\pi$  or  $0$  can be made, avoiding the reassignment of the x and y components to the rotator elements. Again, there is no growth due to the initial rotation:

$$\begin{aligned}x' &= d \cdot x \\y' &= d \cdot y \\z' &= z \text{ if } d = 1, \text{ or } z - \pi \text{ if } d = -1 \\d &= -1 \text{ if } x < 0, +1 \text{ otherwise.}\end{aligned}$$

Both reduction forms assume a modulo  $2\pi$  representation of the input angle. The style of first reduction is more consistent with the succeeding rotations, while the second reduction may be more convenient when wiring is restricted, as is often the case with FPGAs.

The CORDIC rotator described is usable to compute several trigonometric functions directly and others indirectly. Judicious choice of initial values and modes permits direct computation of sine, cosine, arctangent, vector magnitude and transformations between polar and Cartesian coordinates.

### 3.1 Sine and Cosine

The rotational mode CORDIC operation can simultaneously compute the sine and cosine of the input angle. Setting the y component of the input vector to zero reduces the rotation mode result to:

$$\begin{aligned}x_n &= A_n \cdot x_0 \cos z_0 \\y_n &= A_n \cdot x_0 \sin z_0\end{aligned}$$

By setting  $x_0$  equal to  $1/A_n$ , the rotation produces the unscaled sine and cosine of the angle argument,  $z_0$ . Very often, the sine and cosine values modulate a magnitude value. Using other techniques (e.g., a look up table) requires a pair of multipliers to obtain the modulation. The CORDIC technique performs the multiply as part of the rotation operation, and therefore eliminates the need for a pair of explicit multipliers. The output of the CORDIC rotator is scaled by the rotator gain. If the gain is not acceptable, a single multiply by the reciprocal of the gain constant placed before the CORDIC rotator will yield unscaled results. It is worth noting that the hardware complexity of the CORDIC rotator is approximately equivalent to that of a single multiplier with the same word size.

### 3.2 Polar to Rectangular Transformation

A logical extension to the sine and cosine computer is a polar to Cartesian coordinate transformer. The transformation from polar to Cartesian space is defined by:

$$\begin{aligned}x &= r \cos \theta \\y &= r \sin \theta\end{aligned}$$

As pointed out above, the multiplication by the magnitude comes for free using the CORDIC rotator. The transformation is accomplished by selecting the rotation mode with  $x_0$ =polar magnitude,  $z_0$ =polar phase, and  $y_0$ =0. The vector result represents the polar input transformed to Cartesian space. The transform has a gain equal to the rotator gain, which needs to be accounted for somewhere in the system. If the gain is unacceptable, the polar magnitude may be multiplied by the reciprocal of the rotator gain before it is presented to the CORDIC rotator.

### 3.3 General vector rotation

The rotation mode CORDIC rotator is also useful for performing general vector rotations, as are often encountered in motion correction and control systems. For general rotation, the 2 dimensional input vector is presented to the rotator inputs. The rotator rotates the vector through

the desired angle. The output is scaled by the CORDIC rotator gain, which must be accounted for elsewhere in the system. If the scaling is unacceptable, a pair of constant multipliers is required to compensate for the gain. CORDIC rotators may be cascaded in a tree architecture for general rotation in n-dimensions. Some optimization of multidimensional rotation is possible to permit computational savings over the general n-dimensioned case, as reported by Hsiao et al. [4]

### 3.4 Arctangent

The arctangent,  $\theta = \text{Atan}(y/x)$ , is directly computed using the vectoring mode CORDIC rotator if the angle accumulator is initialized with zero. The argument must be provided as a ratio expressed as a vector (x, y). Presenting the argument as a ratio has the advantage of being able to represent infinity (by setting  $x=0$ ). Since the arctangent result is taken from the angle accumulator, the CORDIC rotator growth does not affect the result.

$$z_n = z_0 + \tan^{-1}\left(\frac{y_0}{x_0}\right)$$

### 3.5 Vector Magnitude

The vectoring mode CORDIC rotator produces the magnitude of the input vector as a byproduct of computing the arctangent. After the vectoring mode rotation, the vector is aligned with the x axis. The magnitude of the vector is therefore the same as the x component of the rotated vector. This result is apparent in the result equations for the vector mode rotator:

$$x_n = A_n \sqrt{x_0^2 + y_0^2}$$

The magnitude result is scaled by the processor gain, which needs to be accounted for elsewhere in the system. This implementation of vector magnitude has a hardware complexity of roughly one multiplier of the same width. The CORDIC implementation represents a significant hardware savings over an equivalent Pythagorean processor. The accuracy of the magnitude result improves by 2 bits for each iteration performed.

### 3.6 Cartesian to Polar transformation

The Cartesian to Polar transformation consists of finding the magnitude ( $r = \sqrt{x^2 + y^2}$ ) and phase angle ( $\phi = \text{atan}[y/x]$ ) of the input vector, (x, y). The reader will immediately recognize that both functions are provided simultaneously by the vectoring mode CORDIC rotator. The magnitude of the result will be scaled by the CORDIC rotator gain, and should be accounted for elsewhere in the system. If the gain is unacceptable, it can be corrected by multiplying the resulting magnitude by the reciprocal of the gain constant.

### 3.7 Inverse CORDIC functions

In most cases, if a function can be generated by a CORDIC style computer, its inverse can also be computed. Unless

the inverse is calculable by changing the mode of the rotator, its computation normally involves comparing the output to a target value. The CORDIC inverse is illustrated by the Arcsine function.

### 3.8 Arcsine and Arccosine

The Arcsine can be computed by starting with a unit vector on the positive x axis, then rotating it so that its y component is equal to the input argument. The arcsine is then the angle subtended to cause the y component of the rotated vector to match the argument. The decision function in this case is the result of a comparison between the input value and the y component of the rotated vector at each iteration:

$$\begin{aligned}x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\z_{i+1} &= z_i - d_i \cdot \tan^{-1}\left(2^{-i}\right)\end{aligned}$$

where

$$d_i = +1 \text{ if } y_i < c, -1 \text{ otherwise, and}$$

$c$  = input argument.

Rotation produces the following result:

$$\begin{aligned}x_n &= \sqrt{(A_n \cdot x_0)^2 - c^2} \\y_n &= c \\z_n &= z_0 + \arcsin\left(\frac{c}{A_n \cdot x_0}\right) \\A_n &= \prod_n \sqrt{1 + 2^{-2i}}\end{aligned}$$

The arcsine function as stated above returns correct angles for inputs  $-1 < c/A_n x_0 < 1$ , although the accuracy suffers as the input approaches  $\pm 1$  (the error increases rapidly for inputs larger than about 0.98). This loss of accuracy is due to the gain of the rotator. For angles near the y axis, the rotator gain causes the rotated vector to be shorter than the reference (input), so the decisions are made improperly. The gain problems can be corrected using a “double iteration algorithm”[9] at the cost of an increase in complexity.

The Arccosine computation is similar, except the difference between the x component and the input is used as the decision function. Without modification, the arccosine algorithm works only for inputs less than  $1/A_n$ , making the double iteration algorithm a necessity. The Arccosine could also be computed by using the arcsine function and subtracting  $\pi/2$  from the result, followed by an angular reduction if the result is in the fourth quadrant.

### 3.9 Extension to Linear functions

A simple modification to the CORDIC equation permits the computation of linear functions:

$$\begin{aligned}x_{i+1} &= x_i - 0 \cdot y_i \cdot d_i \cdot 2^{-i} = x_i \\y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\z_{i+1} &= z_i - d_i \cdot (2^{-i})\end{aligned}$$

For rotation mode ( $d_i = -1$  if  $z_i < 0$ ,  $+1$  otherwise) the linear rotation produces:

$$\begin{aligned}x_n &= x_0 \\y_n &= y_0 + x_0 z_0 \\z_n &= 0\end{aligned}$$

This operation is similar to the shift-add implementation of a multiplier, and as multipliers go is not an optimal solution. The multiplication is handy in applications where a CORDIC structure is already available. The vectoring mode ( $d_i = +1$  if  $y_i < 0$ ,  $-1$  otherwise) is more interesting, as it provides a method for evaluating ratios:

$$\begin{aligned}x_n &= x_0 \\y_n &= 0 \\z_n &= z_0 - y_0/x_0\end{aligned}$$

The rotations in the linear coordinate system have a unity gain, so no scaling corrections are required.

### 3.10 Extension to Hyperbolic Functions

The close relationship between the trigonometric and hyperbolic functions suggests the same architecture can be used to compute the hyperbolic functions. While, there is early mention of using the CORDIC structure for hyperbolic coordinate transforms [4], the first description of the algorithm is that by Walther [1]. The CORDIC equations for hyperbolic rotations are derived using the same manipulations as those used to derive the rotation in the circular coordinate system. For rotation mode these are:

$$\begin{aligned}x_{i+1} &= x_i + y_i \cdot d_i \cdot 2^{-i} \\y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\z_{i+1} &= z_i - d_i \cdot \tanh^{-1}(2^{-i})\end{aligned}$$

where

$$d_i = -1 \text{ if } z_i < 0, +1 \text{ otherwise.}$$

Then:

$$\begin{aligned}x_n &= A_n [x_0 \cosh z_0 + y_0 \sinh z_0] \\y_n &= A_n [y_0 \cosh z_0 + x_0 \sinh z_0] \\z_n &= 0 \\A_n &= \prod_n \sqrt{1 - 2^{-2i}} \approx 0.80\end{aligned}$$

In vectoring mode ( $d_i = +1$  if  $y_i < 0$ ,  $-1$  otherwise) the rotation produces:

$$\begin{aligned}x_n &= A_n \sqrt{x_0^2 - y_0^2} \\y_n &= 0 \\z_n &= z_0 + \tanh^{-1}\left(\frac{y_0}{x_0}\right) \\A_n &= \prod_n \sqrt{1 - 2^{-2i}}\end{aligned}$$

The elemental rotations in the hyperbolic coordinate system do not converge. However, it can be shown[1] that convergence is achieved if certain iterations (1=4, 13, 40,..., k, 3k+1,...) are repeated.

The hyperbolic equivalents of all the functions discussed for the circular coordinate system can be computed in a similar fashion. Additionally, as Walther[1] points out, the following functions can be derived from the CORDIC functions:

$$\begin{aligned}\tan \alpha &= \sinh \alpha / \cosh \alpha \\ \tanh \alpha &= \sinh \alpha / \cosh \alpha \\ \exp \alpha &= \sinh \alpha + \cosh \alpha \\ \ln \alpha &= 2 \tanh^{-1}[y/x] \text{ where } x = \alpha + 1 \text{ and } y = \alpha - 1 \\ (\alpha)^{1/2} &= (x^2 - y^2)^{1/2} \text{ where } x = \alpha + 1/4 \text{ and } y = \alpha - 1/4\end{aligned}$$

It is worth noting the similarities between the CORDIC equations for circular, linear, and hyperbolic systems. The selection of coordinate system can be made by introducing a mode variable that takes on values 1, 0, or -1 for circular, linear and hyperbolic systems respectively. The unified [1] CORDIC iteration equations are then:

$$\begin{aligned}x_{i+1} &= x_i - m \cdot y_i \cdot d_i \cdot 2^{-i} \\y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\z_{i+1} &= z_i - d_i \cdot e_i\end{aligned}$$

where  $e_i$  is the elementary angle of rotation for iteration  $i$  in the selected coordinate system. Specifically,  $e_i = \tan^{-1}(2^{-i})$  for  $m=1$ ,  $e_i = 2^{-i}$  for  $m=0$ , and  $e_i = \tanh^{-1}(2^{-i})$  for  $m=-1$ . This unification, due to Walther, permits the design of a general purpose CORDIC processor.

### 3.11 Short cuts

For fixed angle rotations, as are encountered in such places as fast Fourier Transforms (FFTs), the arctangent base representation of the angle can be pre-computed and applied directly to the CORDIC rotator. This hardwiring of a fixed angle(s) eliminates the need for the angle accumulator, which reduces the circuit complexity by about 25 percent. If the constraints on the decision variable are relaxed to allow that variable to take on values of {-1,0,1} instead of just {-1,1}, the number of iterations can also be reduced. Iterations for which the decision variable is zero pass the data unrotated, and can thus be eliminated. This modification causes the gain to become a function of the rotated angle, so it is only useful if the rotation angle is fixed. Hu and Naganathan[10] propose a method of pre-computing the recoded angles for the ternary decision variable. This technique can significantly reduce the complexity of on-line CORDIC processors used for fixed angle rotations.

## 4. IMPLEMENTATION IN AN FPGA

There are a number of ways to implement a CORDIC processor. The ideal architecture depends on the speed versus area tradeoffs in the intended application. First we will examine an iterative architecture that is a direct translation from the CORDIC equations. From there, we will look at a minimum hardware solution and a maximum performance solution.

### 4.1 Iterative CORDIC Processors

An iterative CORDIC architecture can be obtained simply by duplicating each of the three difference equations in hardware as shown in Figure 1. The decision function,  $d_i$ , is driven by the sign of the  $y$  or  $z$  register depending on whether it is operated in rotation or vectoring mode. In operation, the initial values are loaded via multiplexers into the  $x$ ,  $y$  and  $z$  registers. Then on each of the next  $n$  clock cycles, the values from the registers are passed through the shifters and adder-subtractors and the results placed back in the registers. The shifters are modified on each iteration to cause the desired shift for the iteration. Likewise, the ROM address is incremented on each iteration so that the appropriate elementary angle value is presented to the  $z$  adder-subtractor. On the last iteration, the results are read directly from the adder-subtractors. Obviously, a simple state machine is required keep track of the current iteration, and to select the degree of shift and ROM address for each iteration.

The design depicted in Figure 1 uses word-wide data paths (called bit-parallel design). The bit-parallel variable shift shifters do not map well to FPGA architectures because of the high fan-in required. If implemented, those shifters will typically require several layers of logic (i.e., the signal will need to pass through a number of FPGA cells). The result is a slow design that uses a large number of logic cells.

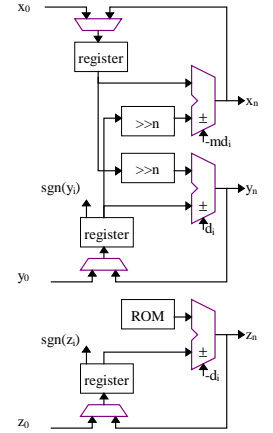
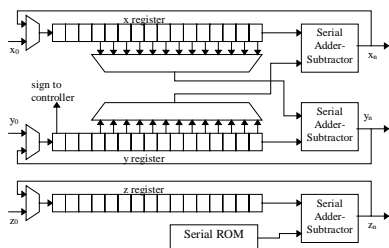


Figure 1. Iterative CORDIC structure

A considerably more compact design is possible using bit serial arithmetic. The simplified interconnect and logic in a bit serial design allows it to work at a much higher clock rate than the equivalent bit parallel design. Of course, the design also needs to be clocked  $w$  times for each iteration ( $w$  is the width of the data word). The bit serial design consists of three bit serial adder-subtractors, three shift registers and a serial Read Only Memory (ROM). Each shift register has a length equal to the word width. There is also some gating or multiplexers to select taps off the shift registers for the right shifted cross terms (shifting is accomplished using bit delays in bit serial systems). The bit serial CORDIC architecture is shown in Figure 2. In this design,  $w$  clocks are required for each of the  $n$  iterations, where  $w$  is precision of the adders. In operation, the load multiplexers on the left are opened for  $w$  clock periods to initialize the  $x$ ,  $y$  and  $z$  registers (these registers could also be parallel loaded to initialize). Once loaded, the data is shifted right through the serial adder-subtractors and returned to the left end of the register. Each iteration requires  $w$  clocks to return the result to the register. At the beginning of each iteration, the control state machine reads the sign of the  $y$  (or  $z$ ) register and sets the add/subtract controls accordingly. The appropriate tap off the register for the cross terms is also selected at the beginning of each iteration. During the  $n$ th iteration, the results can be read from the outputs of the serial adders while the next initialization data is shifted into the registers.



**Figure 2 Bit serial iterative CORDIC**

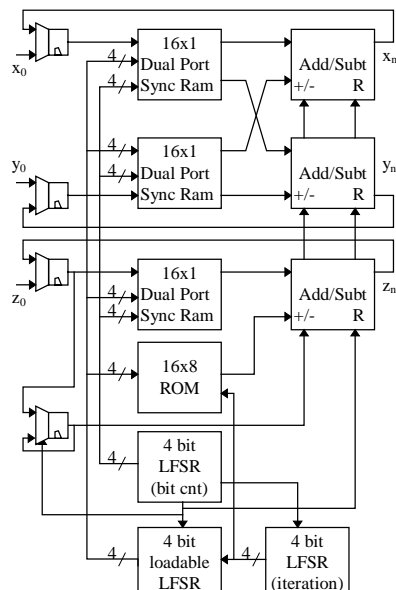
The simplicity of the bit serial design is apparent from figure 2. Even in this case, the wiring of the shift tap multiplexers can present problems in some FPGAs (this is one place where tri-state long lines can come in handy). Even so, the interconnect is minimal and the logic between registers is simple. This combination permits bit clock rates near the maximum toggle frequency of the FPGA. The possibility of using extreme bit clock frequencies makes up for the large number of clock cycles required to complete each rotation.

Now, if the design is in a Xilinx 4000E series part, the shift registers can be implemented in the CLB RAM[2]. The RAM emulates a shift register by incrementing the read/write address after each access. The dual port capability of the CLB RAM provides the capability to read two locations in the 16x1 RAM simultaneously [9]. By properly sequencing the second address, the effect of the shift tap multiplexer is achieved without a physical multiplexer. The result is the shift register and multiplexer for word lengths up to 16 bits are implemented in a single CLB (plus 8 CLBs for the 2 address sequencers and iteration counter, which are shared by the three shifters). The serial ROM also uses the CLB for data storage. One CLB is required for every two iterations. The 16 bit, 8 iteration CORDIC processor shown in Figure 3 uses only 21 CLBs, and will run at bit rates up to about 90 MHz (mainly limited by the RAM write cycle). This translates to about a 1.5 $\mu$ s processing time, which is only about three and a half times longer than the best one could expect from the much larger bit parallel iterative solution.

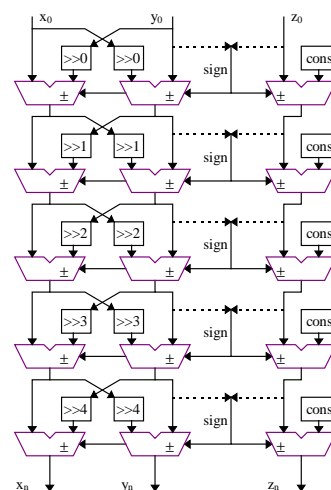
## 4.2 On-Line CORDIC Processors

The CORDIC processors discussed so far are iterative, which means the processor has to perform iterations at  $n$  times the data rate. The iteration process can unrolled[18] so that each of  $n$  processing elements always performs the same iteration. An unrolled CORDIC processor is shown in Figure 4. Unrolling the processor results in two significant simplifications. First the shifters are each a fixed shift, which means that they can be implemented in the wiring. Second, the lookup values for the angle accumulator are

distributed as constants to each adder in the angle accumulator chain. Those constants can be hardwired instead of requiring storage space. The entire CORDIC processor is reduced to an array of interconnected adder-subtractors. The need for registers is also eliminated, making the unrolled processor strictly combinatorial. The delay through the resulting circuit would be substantial, but the processing time is reduced from that required by the iterative circuit (if by nothing else than the set-up and hold times of the register). Most times, especially in an FPGA, it does not make sense to use such a large combinatorial circuit. The unrolled processor is easily pipelined by inserting registers between the adder-subtractors. In the case of most FPGA architectures there are already registers present in each logic cell, so the addition of the pipeline registers has no hardware cost.

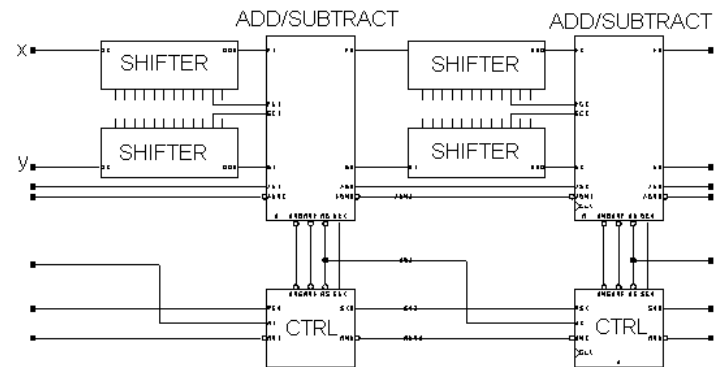


**Figure 3 Iterative bit serial design for Xilinx 4000E series FPGA uses 21 CLBs**



**Figure 4 Unrolled CORDIC processor**

The unrolled processor can also be converted to a bit serial design. Each adder subtractor is replaced by a serial adder-subtractor, separated by  $w$  bit shift registers. The shift registers are necessary to extract the sign of the  $y$  or  $z$  element before the first bits (lsbs) reach the next adder-subtractors. The right shifted cross terms are taken from fixed taps in the shift registers. Some method of sign extension for the shifted terms is required too. Figure 5



**Figure 5 two iterations of bit serial CORDIC pipeline in Atmel/NSC FPGA**

shows two iterations of a bit serial CORDIC processor implemented in an Atmel 6005 or NSC Clay31 FPGA. Notice the cross term is taken from different taps off the shift register at each iteration. This particular processor is used to compute vector magnitude. Since this is a vector mode process and the result angle is not required, there is no need for an angle accumulator. Figure 6 shows the detail of the adder-subtractor for that design. The adder subtractor in this case includes logic to extend the sign of the shifted cross term and to reset the adder subtractor between words. The entire 7 iteration design occupies approximately 20% of the FPGA and runs at bit rates up to 125 Mhz [3].

Higher performance requires either multiple bit serial processors running in parallel, or an unrolled parallel pipeline. Until recently, FPGAs did not have the required combination of logic and routing resource to build a parallel processor. This barrier is mostly due to the large amount of cross routing required between the  $x$  and  $y$  registers at each pipeline stage. Additionally, the performance diminishes as the word width is increased because of the carry propagation times across the adders. The Xilinx 4000E series has sufficient routing to realize a reasonably compact parallel CORDIC pipeline. Its dedicated carry logic provides acceptable performance for the adders. Figure 7 shows a 14 bit, 5 iteration pipelined CORDIC processor that fits comfortably in half of a 4013E. That design, used for polar to Cartesian coordinate transformations in a radar target generator, runs at 52 MHz (clock rate and data rate) in an XC4013E-2.

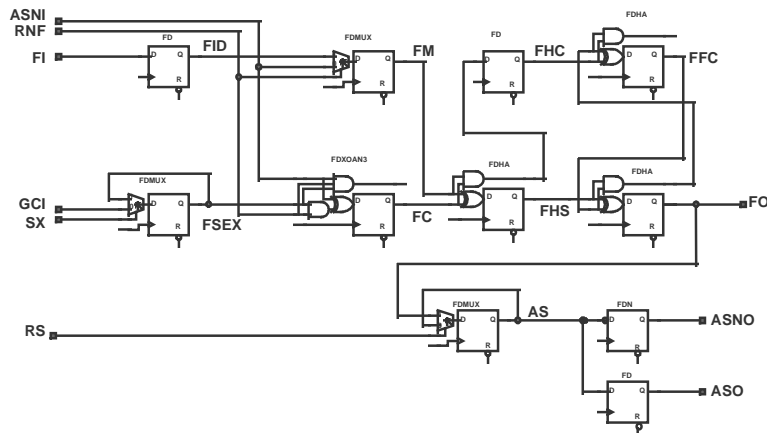


Figure 6 detail of pipelined bit serial adder-subtractor in Atmel/NSC FPGA

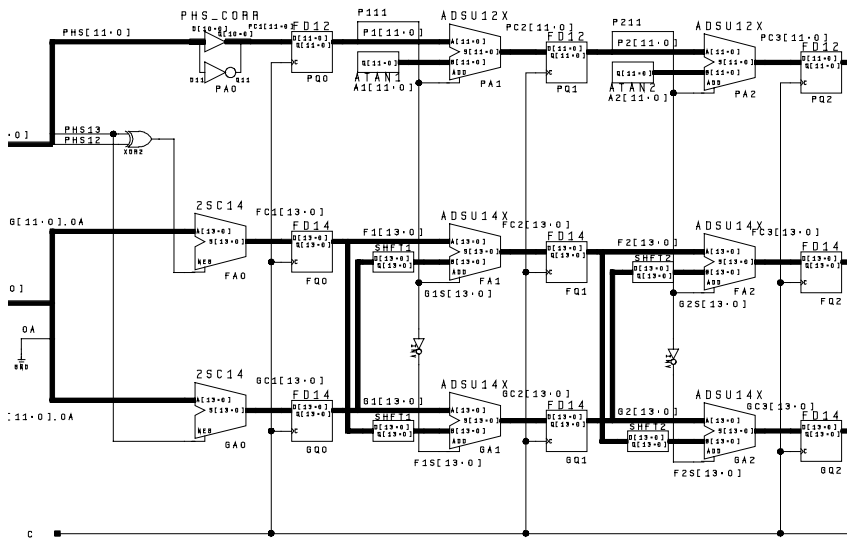


Figure 7 section of parallel pipelined CORDIC can run at over 50 Megasamples per second in a Xilinx XC4013E-2

## 5. CONCLUSIONS

The CORDIC algorithms presented in this paper are well known in the research and super-computing circles. It is, however, my experience that the majority of today's hardware DSP designs are being done by engineers with little or no background in hardware efficient DSP algorithms. The new DSP designers must become familiar with these algorithms and the techniques for implementing them in FPGAs in order to remain competitive. The CORDIC algorithm is a powerful tool in the DSP toolbox. This paper shows that tool is available for use in FPGA based computing machines, which are the likely basis for the next generation DSP systems.

## 6. REFERENCES

- [1] Ahmed, H. M., Delosme, J.M., and Morf, M., "Highly Concurrent Computing Structure for Matrix Arithmetic and Signal Processing," IEEE Comput. Mag., Vol. 15, 1982, pp. 65-82.
- [2] Alfke, P., "Efficient Shift Registers, LFSR Counters, and Long Pseudo Random Sequence Generators," Xilinx application note, August, 1995.
- [3] Andracka, R. J., "Building a High Performance Bit-Serial Processor in an FPGA," Proceedings of Design SuperCon '96, Jan 1996, pp.5.1 - 5.21
- [4] Deprettere, E., Dewilde, P., and Udo, R., "Pipelined CORDIC Architecture for Fast VLSI Filtering and Array Processing," Proc. ICASSP'84, 1984, pp. 41.A.6.1-41.A.6.4
- [5] Despain, A.M., "Fourier Transform Computations Using CORDIC Iterations," IEEE Transactions on Computers, Vol.23, 1974, pp. 993-1001.
- [6] Duh, W.J., and Wu, J.L., "Implementing the Discrete Cosine Transform by Using CORDIC Techniques," Proceedings the International Symposium on VLSI Technology, Systems and Applications, Taipei, Taiwan, 1989, pp. 281-285
- [7] Duprat, J. and Muller, J.M., "The CORDIC Algorithm: New Results for Fast VLSI Implementation," IEEE Transactions on Computers, Vol. 42, pp. 168-178, 1993.

- [8] Hsiao, S.F. and Delosme, J.M., "The CORDIC Householder Algorithm," Proceedings of the 10th Symposium on Computer Arithmetic, pp. 256-263, 1991.
- [9] Hu, Y.H., and Naganathan, S., "A Novel Implementation of Chirp Z-Transformation Using a CORDIC Processor," IEEE Transactions on ASSP, Vol. 38, pp. 352-354, 1990.
- [10] Hu, Y.H., and Naganathan, S., "An Angle Recoding Method for CORDIC Algorithm Implementation", IEEE Transactions on Computers, Vol. 42, pp. 99-102, January 1993
- [11] Knapp, S. K., "XC4000E Edge triggered and Dual Port RAM Capability," Xilinx application note, August 11, 1995
- [12] Marchesi, M., Orlandi, G., and Piazza, F., "Systolic Circuit for Fast Hartley Transform," Proceedings - IEEE International Symposium on Circuits and Systems, Espoo, Finland, June 1988, pp. 2685-2688
- [13] Mazenc, C., Merrheim, X., and Muller, J.M., "Computing Functions Arccos and Arcsin Using CORDIC," IEEE Transactions on Computers, Vol. 42, pp. 118-122, 1993.
- [14] Sibul, L.H. and Fogelsanger, A.L., "Application of Coordinate Rotation Algorithm to Singular Value Decomposition," IEEE Int. Symp. Circuits and Systems, pp. 821-824, 1984.
- [15] Volder, J., "Binary computation algorithms for coordinate rotation and function generation," Convair Report IAR-1 148 Aeroelectronics Group, June 1956.
- [16] Volder, J., "The CORDIC Trigonometric Computing Technique," IRE Trans. Electronic Computing, Vol EC-8, pp.330-334 Sept 1959.
- [17] Walther, J.S., "A unified algorithm for elementary functions," Spring Joint Computer Conf., 1971, proc., pp. 379-385.
- [18] Wang, S. and Piuri, V., "A Unified View of CORDIC Processor Design", Application Specific Processors, Edited by Earl E.Swartzlander, Jr., Ch. 5, pp. 121-160, Kluwer Academic Press, November 1996.