

Introduction

- On a current FPGA there are *many* multipliers and adders available. However for various communications techniques, and matrix algorithms which require trigonometry, square root etc,

How would you perform these computations on an FPGA?

Perhaps look-up tables, iterative techniques (or even come up with algorithms to try and circumvent the trigonometry!)

- In this section the CORDIC algorithm is introduced; this is a “shift and add” algorithm that allows calculation of many various trigonometric functions, such as:
 - $\sqrt{x^2 + y^2}$
 - $\cos \theta$, $\tan \theta$, $\sin \theta$
 - and other functions including divide and logarithmic functions.



Notes:

For more detail and background on the CORDIC algorithm the following material may be useful:

[1] R. Andraka. A survey of CORDIC algorithms for FPGA based computers. www.andraka.com/cordic.htm

[2] The CORDIC Algorithms. www.ee.byu.edu/ee/class/ee621/Lectures/L22.PDF

[3] CORDIC Tutorial. <http://my.execpc.com/~geezer/embed/cordic.htm>

[4] M. J. Irwin. Computer Arithmetic. <http://www.cse.psu.edu/~cg575/lectures/cse575-cordic.pdf>

This is nothing “new” in the CORDIC technique. In fact it dates back to 1957 in a paper by an author J. Volder. In the 1950s shift and adds on large physical computers was the limit of technology so CORDIC was of real interest. Also in 1970s with the advent of handheld calculators from Hewlett Packard and other companies, many had an internal CORDIC unit to calculate all of the trigonometric functions (those who recall this time, will remember that taking the tangent of an angle, had a delay of sometimes up to a second while the calculator calculated the result!).

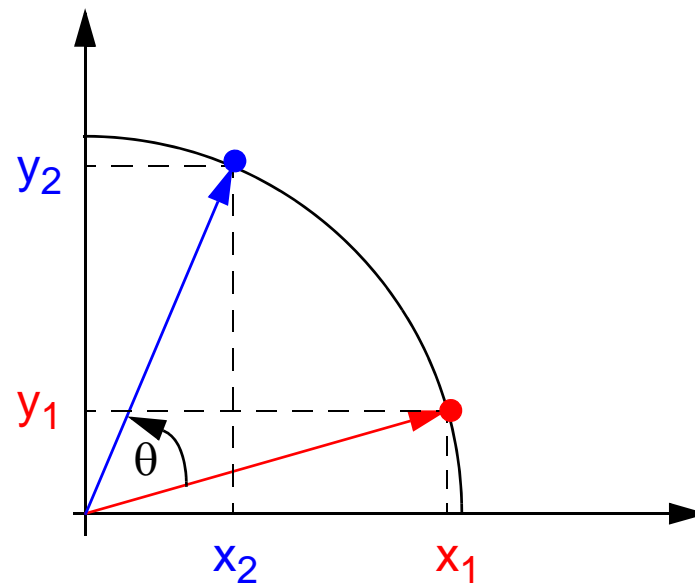
In the 1980s CORDIC was less relevant given the advent of high speed multipliers and general purpose processors with plenty of memory available. However now in the 2000's for FPGAs, CORDIC is definitely a candidate technique for the calculation of trigonometric functions in DSP applications such as MIMO, beamforming and other adaptive systems.

Cartesian Coordinate Plane Rotations

- The standard method of rotating a point $((x_1, y_1))$ by θ degrees in the xy plane to a point (x_2, y_2) is given by the well known equations:

$$x_2 = x_1 \cos \theta - y_1 \sin \theta$$

$$y_2 = x_1 \sin \theta + y_1 \cos \theta$$



- This is variously known as a plane rotation, a vector rotation, or in linear (matrix) algebra, a Givens Rotation.

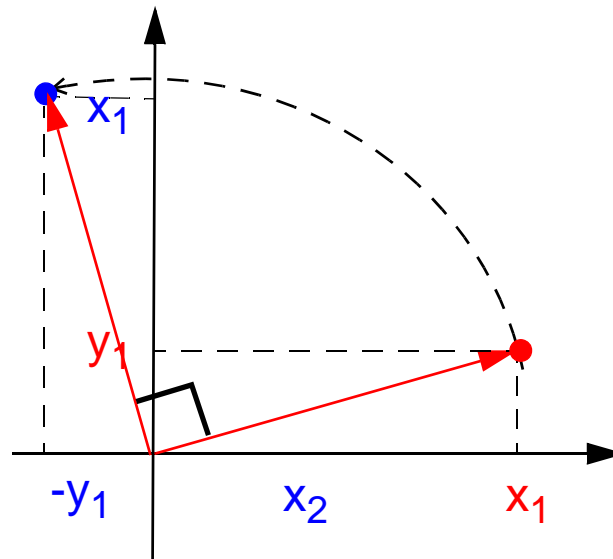
Notes:

This can also be written in a matrix vector form as:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

So for example a 90° phase shift would be:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} -y_1 \\ x_1 \end{bmatrix}$$



Pseudo-Rotations

- By taking out the factor $\cos \theta$ term we can rewrite the equations as:

$$x_2 = x_1 \cos \theta - y_1 \sin \theta = \cos \theta (x_1 - y_1 \tan \theta)$$

$$y_2 = x_1 \sin \theta + y_1 \cos \theta = \cos \theta (y_1 + x_1 \tan \theta)$$

- If we now drop the $\cos \theta$ term then we have a ***pseudo-rotation***:

$$\hat{x}_2 = \cancel{\cos \theta} (x_1 - y_1 \tan \theta) = x_1 - y_1 \tan \theta$$

$$\hat{y}_2 = \cancel{\cos \theta} (y_1 + x_1 \tan \theta) = y_1 + x_1 \tan \theta$$

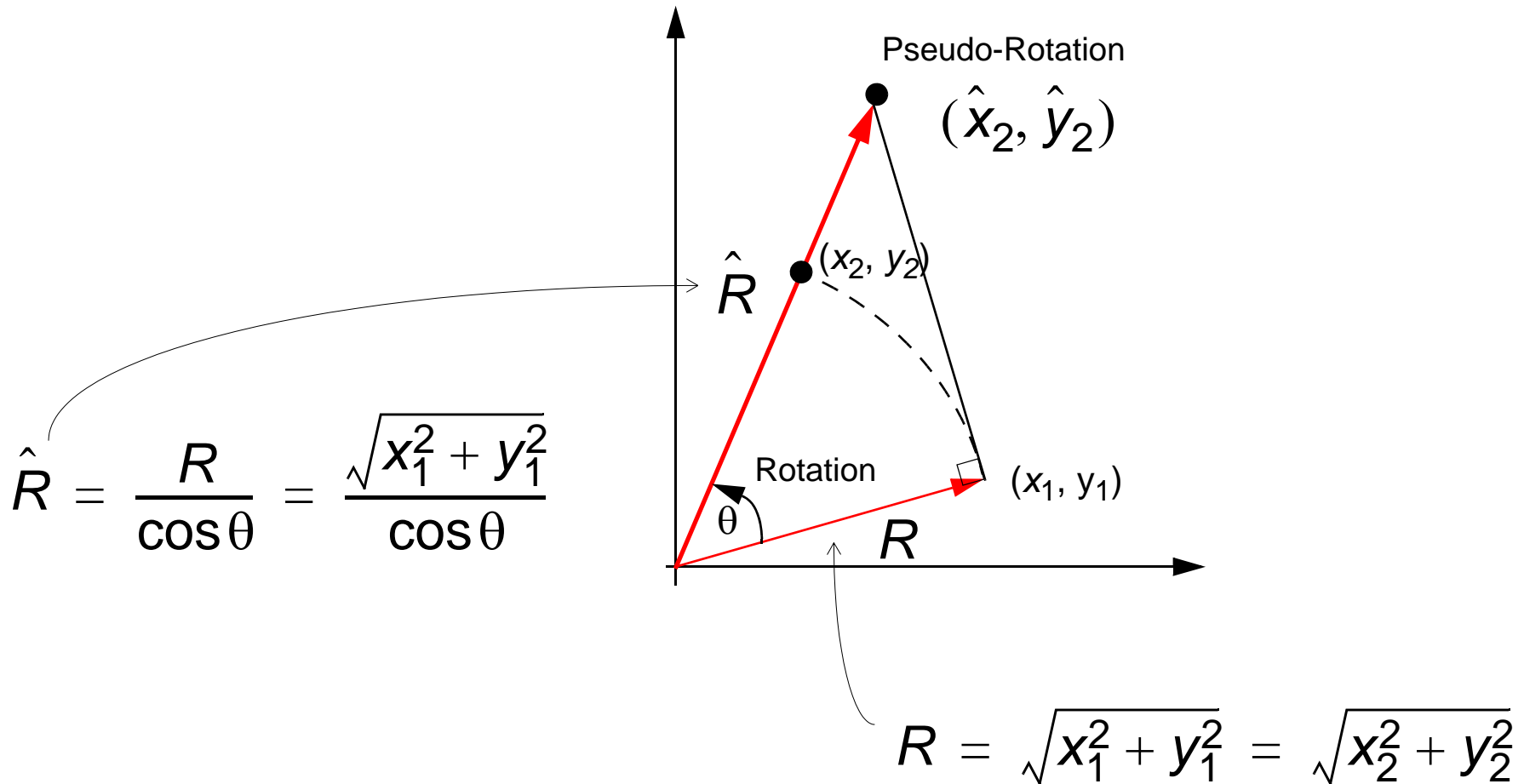
ie. the angle of rotation is correct but the x and y values are scaled by $\cos^{-1} \theta$ (i.e. both become larger than before as $\cos^{-1} \theta > 1$).

- Note that we have **NO** mathematical justification for dropping the $\cos \theta$ term, however later we note it can make the computation of plane rotations more amenable to simple operations.



Notes:

In the xy plane this is:



Therefore the magnitude of the vector R increases by a factor of $1/\cos \theta$ after the pseudo-rotation.

So the resulting point is at the right angle, but the vector is the wrong magnitude.

The CORDIC Method

- The key to the CORDIC method is to only (pseudo-) rotate by angles of θ where $\tan\theta^i = 2^{-i}$. Therefore in the equation:

$$\hat{x}_2 = x_1 - y_1 \tan\theta = x_1 - y_1 2^{-i}$$

$$\hat{y}_2 = y_1 + x_1 \tan\theta = y_1 + x_1 2^{-i}$$

- The table below shows the rotation angles (to 9 decimal places) that can be used for each iteration (i) of the CORDIC algorithm:

i	θ^i (Degrees)	$\tan\theta^i = 2^{-i}$
0	45.0	1
1	26.555051177...	0.5
2	14.036243467...	0.25
3	7.125016348...	0.125
4	3.576334374...	0.0625

Notes:

At this stage we alter the transform to become an iterative algorithm. We restrict the angles that we are able to rotate by, such that to rotate by an arbitrary θ requires a series of successively smaller rotations at each iteration i . The rotation angles obey the law: $\tan\theta^{(i)} = 2^{-i}$. Obeying this law causes the **multiplication by the tangent term to become a shift**.

The first few iterations then take the form:

1st iteration: rotate by 45° ; 2nd iteration: rotate by 26.6° ; 3rd iteration: rotate by 14° etc.

The direction with each rotation takes obviously affects the accumulative angle that is rotated. Arbitrary angles can be rotated in the range $-99.7 \leq \theta \leq 99.7$. The sum of all angles obeying the law $\tan\theta^i = 2^{-i}$ is 99.7. For angles outside this range trigonometric identities can be used to convert the desired angle into one within the range. The number of bits resolution in the angle is of course relevant to the final accuracy.

i	tan θ	Angle, θ	cos θ
1	1	45.000000000	0.707106781
2	0.5	26.5650511771	0.894427191
3	0.25	14.0362434679	0.9701425
4	0.125	7.1250163489	0.992277877
5	0.0625	3.5763343750	0.998052578
6	0.03125	1.7899106082	0.999512076
7	0.015625	0.8951737102	0.999877952
8	0.0078125	0.4476141709	0.999969484
9	0.00390625	0.2238105004	0.999992371
10	0.001953125	0.1119056771	0.999998093
11	0.000976563	0.0559528919	0.999999523
12	0.000488281	0.0279764526	0.999999881
13	0.000244141	0.0139882271	0.99999997

$$\cos 45 \times \cos 26.5 \times \cos 14.03 \times \cos 7.125 \dots \times \cos 0.0139 = 0.607252941$$

$1/0607252941 = 1.6467602$. Therefore after 13 rotations to scale the pseudo-rotation back requires a multiply by 1.64676024187.. The number of bits resolution in the angles will be significant to the accuracy of the final rotation.

Angle Accumulator

- The pseudo rotation shown earlier can now be expressed for each iteration as:

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$

$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$

- At this stage we introduce a 3rd equation called the Angle Accumulator which is used to keep track of the accumulative angle rotated at each iteration:

$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)} \quad (\text{Angle Accumulator})$$

where $d_i = +/- 1$

- The symbol d_i is a decision operator and is used to decide which direction to rotate.



Notes:

At this stage we can now express the equations for each iteration as:

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$

$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$

where d_i the decision operator is used to give the rotation a clockwise or anticlockwise direction. The conditions of d_i depend on the **mode** of operation which shall be discussed shortly.

Also, at this stage we introduce a third equation called the **Angle Accumulator** which is used to keep track of the accumulative angle rotated at each iteration:

$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)}$$

The three equations now represent the CORDIC algorithm for rotations in a Circular Coordinate System. We shall see later that there are other coordinate systems that can be used with the CORDIC method to calculate a greater range of functions.

Shift-Add Algorithm

- Hence, the original algorithm has now been reduced to an iterative **shift-add** algorithm for pseudo-rotations of a vector:

$$x^{(i+1)} = (x^{(i)} - d_i(2^{-i}y^{(i)}))$$

$$y^{(i+1)} = (y^{(i)} + d_i(2^{-i}x^{(i)}))$$

$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)}$$

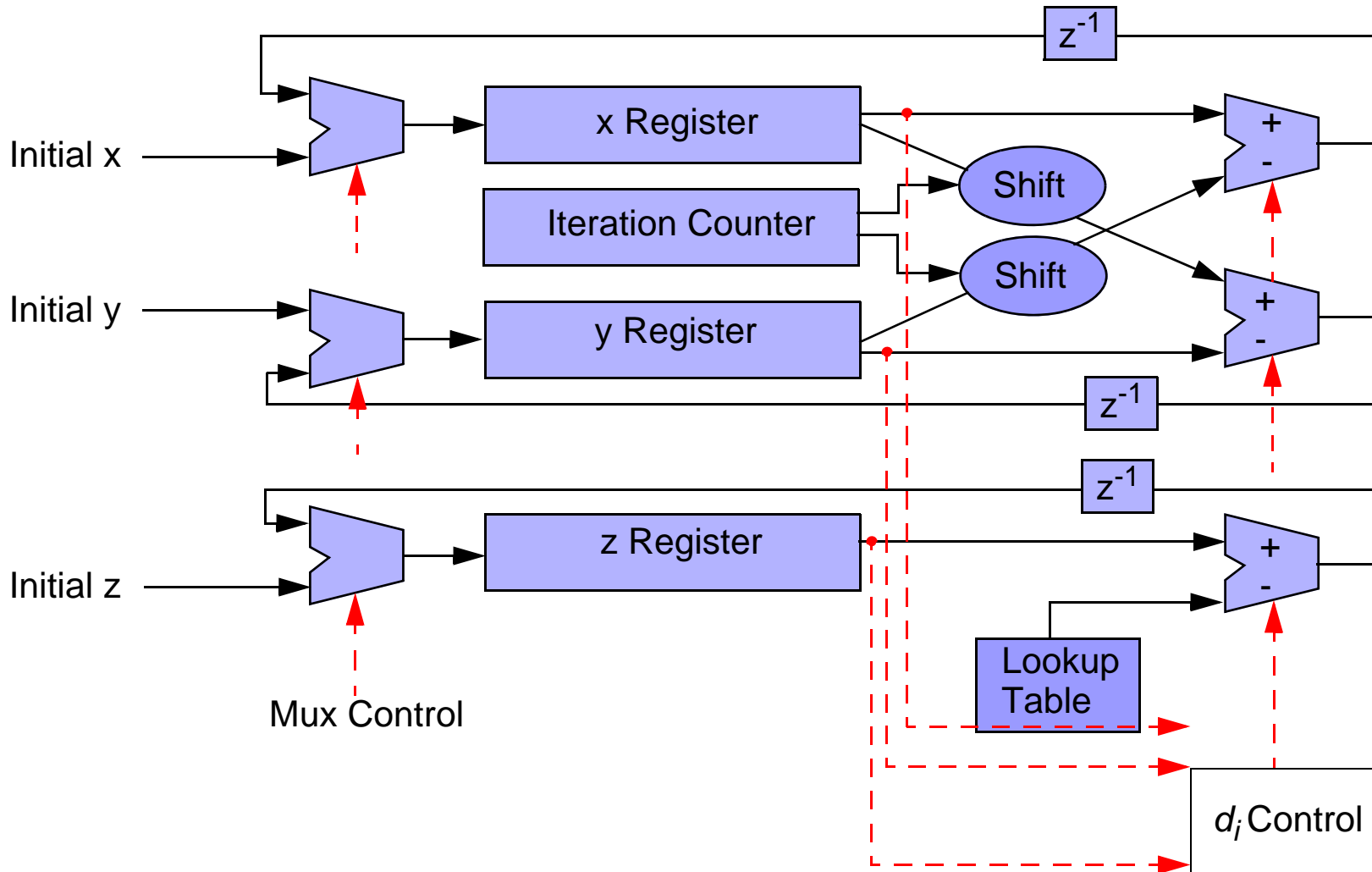
- Thus each iteration requires:
 - 2 shifts**
 - 1 table lookup** ($\theta^{(i)}$ values)
 - 3 additions**



Notes:

Here, the reason for removing the $\cos \theta$ term earlier becomes clear. With this term removed, the transform is reduced to an iterative shift-add algorithm for pseudo-rotations.

CORDIC hardware:



The Scaling Factor

- The Scaling Factor is a by-product of the pseudo-rotations.
- When simplifying the algorithm to allow pseudo-rotations the $\cos\theta$ term was omitted.
- Thus outputs $x^{(n)}$, $y^{(n)}$ are scaled by a factor K_n where:

$$K_n = \prod_n 1 / (\cos \theta^{(i)}) = \prod_n (\sqrt{1 + 2^{(-2i)}})$$

- However if the number of iterations are known then the **Scaling Factor** K_n can be precomputed.
- Also, $1/K_n$ can be precomputed and used to calculate the true values of $x^{(n)}$ and $y^{(n)}$.



Notes:

To simplify the Givens rotation we removed the $\cos\theta$ term to allow us to perform pseudo-rotations. However, this simplification has a **side-effect**. The output values $x^{(n)}$ and $y^{(n)}$ are scaled by a factor K_n known as the Scaling Factor where:

$$K_n = \prod_n 1/(\cos\theta^{(i)}) = \prod_n \left(\sqrt{1 + \tan^2\theta^{(i)}} \right) = \prod_n \left(\sqrt{1 + 2^{(-2i)}} \right)$$

$$K_n \rightarrow 1.6476 \text{ as } n \rightarrow \infty$$

$$1/K_n \rightarrow 0.6073 \text{ as } n \rightarrow \infty$$

n = number of iterations

However, if we know the number of iterations that will be performed then we can precompute the value of $1/K_n$ and correct the final values of $x^{(n)}$ and $y^{(n)}$ by multiplying them by this value.

Rotation Mode

- The CORDIC method is operated in one of two modes;
- The mode of operation dictates the condition for the control operator d_i ;
- In Rotation Mode choose: $d_i = \text{sign}(z^{(i)}) \Rightarrow z^{(i)} \rightarrow 0$;
- After n iterations we have:

$$x^{(n)} = K_n(x^{(0)} \cos z^{(0)} - y^{(0)} \sin z^{(0)})$$

$$y^{(n)} = K_n(y^{(0)} \cos z^{(0)} + x^{(0)} \sin z^{(0)})$$

$$z^{(n)} = 0$$

- Can compute $\cos z^{(0)}$ and $\sin z^{(0)}$ by starting with $x^{(0)} = 1/K_n$ and $y^{(0)} = 0$

Notes:

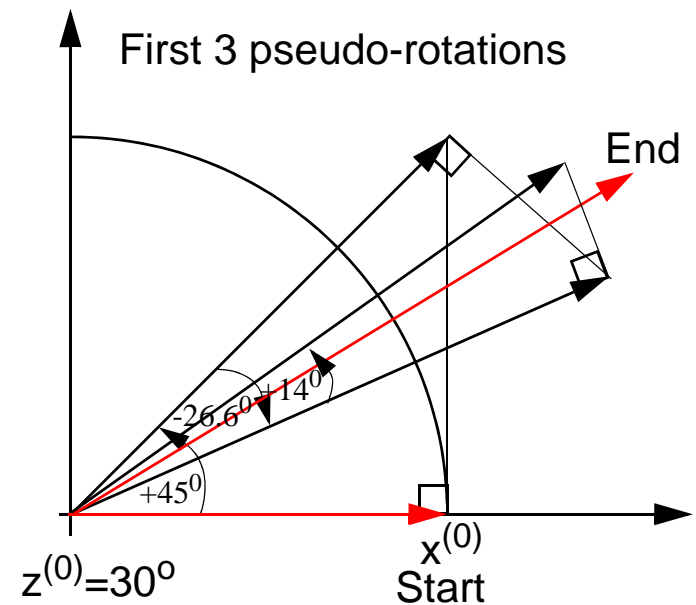
In Rotation Mode the decision operator d_i obeys the condition:

$$d_i = \text{sign}(z^{(i)})$$

Thus, we input $x^{(0)}$ and $z^{(0)}$ ($y^{(0)} = 0$) and then drive $z^{(0)}$ towards 0.

Example: calculate $\sin z^{(0)}$, $\cos z^{(0)}$ where $z^{(0)} = 30^\circ$

i	d_i	$\theta^{(i)}$	$z^{(i)}$	$y^{(i)}$	$x^{(i)}$
0	+1	45	+30	0	0.6073
1	-1	26.6	-15	0.6073	0.6073
2	+1	14	+11.6	0.3036	0.9109
3	-1	7.1	-2.4	0.5313	0.8350
4	+1	3.6	+4.7	0.4270	0.9014
5	+1	1.8	+1.1	0.4833	0.8747
6	-1	0.9	-0.7	0.5106	0.8596
7	+1	0.4	+0.2	0.4972	0.8676
8	-1	0.2	-0.2	0.5040	0.8637
9	+1	0.1	+0	0.5006	0.8657



Vectoring Mode

- In Vectoring Mode choose: $d_i = -\text{sign}(x^{(i)}y^{(i)}) \Rightarrow y^{(i)} \rightarrow 0$
- After n iterations we have:

$$x^{(n)} = K_n \left(\sqrt{(x^{(0)})^2 + (y^{(0)})^2} \right)$$

$$y^{(n)} = 0$$

$$z^{(n)} = z^{(0)} + \tan^{-1} \left(\frac{y^{(0)}}{x^{(0)}} \right)$$

- Can compute $\tan^{-1} y^{(0)}$ by setting $x^{(0)} = 1$ and $z^{(0)} = 0$



Notes:

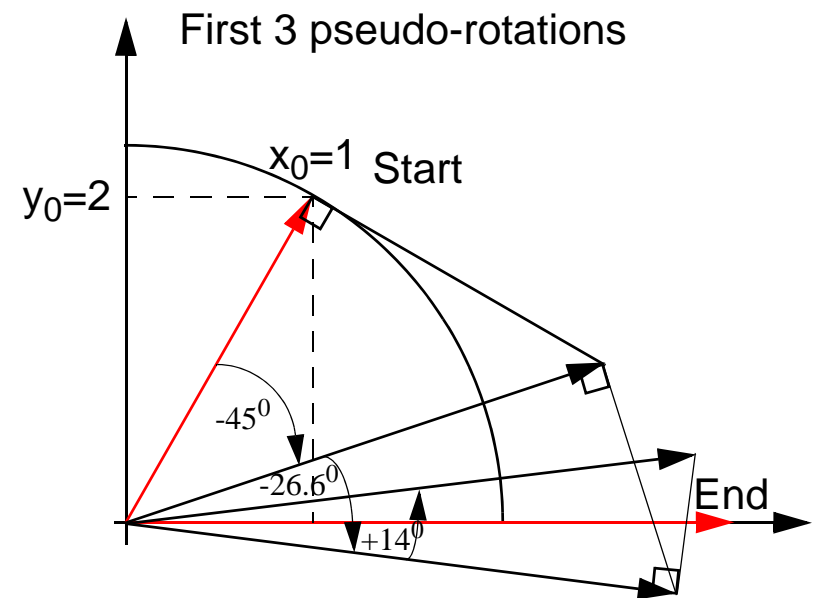
In Vectoring Mode the decision operator d_i obeys the condition:

$$d_i = -\text{sign}(x^{(i)}y^{(i)})$$

Thus, we input $x^{(0)}$ and $y^{(0)}$ ($z^{(0)} = 0$) and then drive $y^{(0)}$ towards 0.

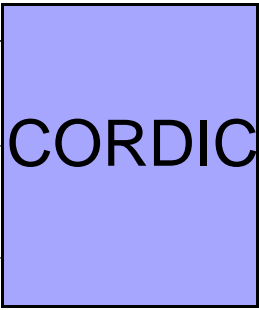
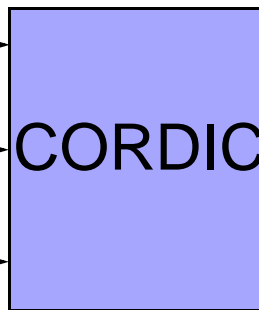
Example: calculate $\tan^{-1}(y^{(0)}/x^{(0)})$ where $y^{(0)} = 2$ and $x^{(0)} = 1$

i	$z^{(i)}$	θ^i	$y^{(i)}$
0	0	45	2
1	45	26.6	1
2	71.6	14	-0.5
3	57.6	7.1	0.375
4	64.7	3.6	-0.078
5	61.1	1.8	0.151
6	62.9	0.9	0.039
7	63.8	0.4	-0.019
8	63.4	0.2	0.009



Circular Coordinate System

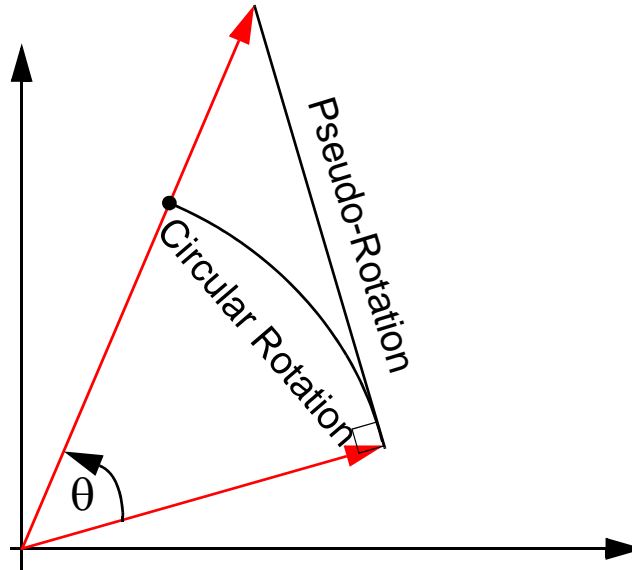
- So far only pseudo-rotations in a Circular Coordinate System have been considered.
- Thus, the following functions can be computed:

Coordinate System	Rotation Mode $z^{(i)} \rightarrow 0; d_i = \text{sign}(z^{(i)})$	Vectoring Mode $y^{(i)} \rightarrow 0; d_i = -\text{sign}(x^{(i)}y^{(i)})$
Circular	 <p> $x \rightarrow$ CORDIC $\rightarrow K(x.\cos z - y.\sin z)$ $y \rightarrow$ CORDIC $\rightarrow K(y.\cos z + x.\sin z)$ $z \rightarrow$ CORDIC $\rightarrow 0$ </p> <p>For $\cos z$ & $\sin z$, set $x = 1/K, y = 0$</p>	 <p> $x \rightarrow$ CORDIC $\rightarrow K(x^2+y^2)^{1/2}$ $y \rightarrow$ CORDIC $\rightarrow 0$ $z \rightarrow$ CORDIC $\rightarrow z + \tan^{-1}(y/x)$ </p> <p>For $\tan^{-1} y$, set $x = 1, z = 0$</p>

- However, more functions can be computed if we use other coordinate systems.

Notes:

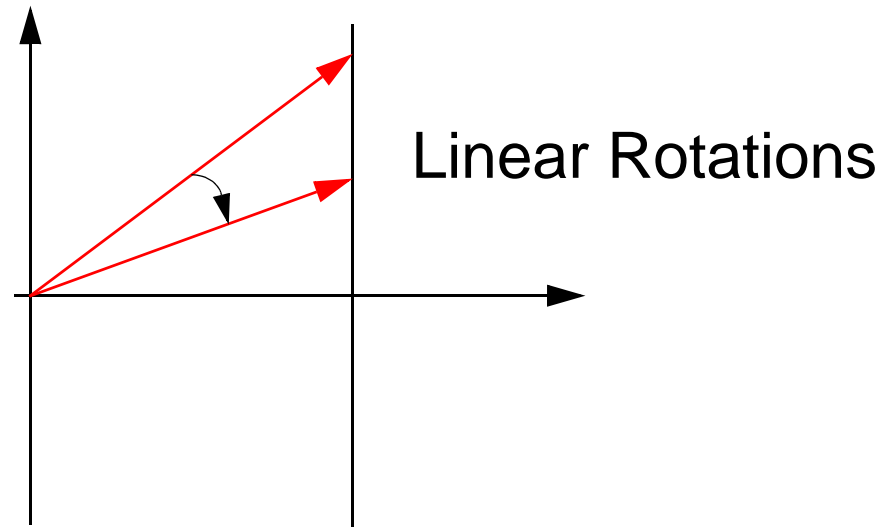
With rotations in a Circular Coordinate System we are limited to the number of functions that can be calculated.



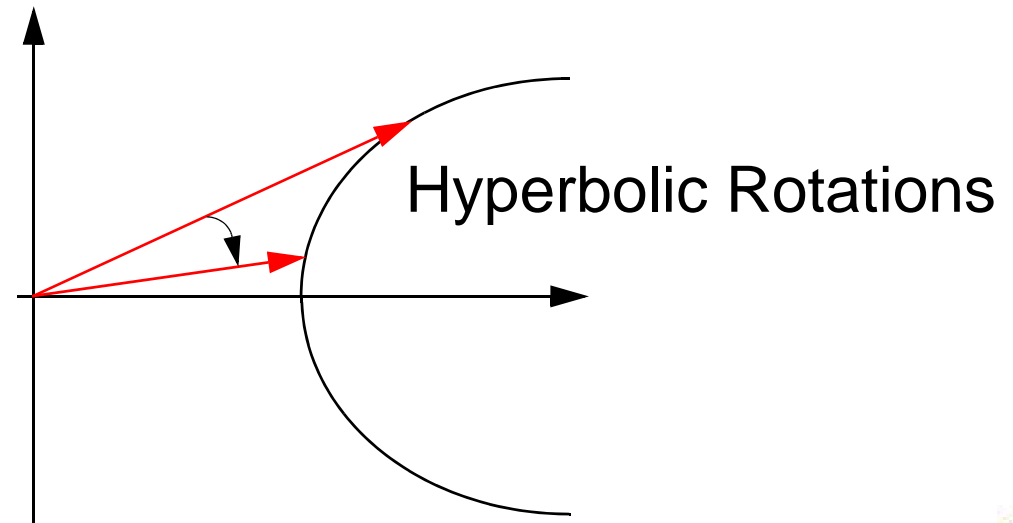
However, we shall see that by considering rotations in other coordinate systems we can calculate more functions directly, such as multiplications and divides, which allow us to calculate even more functions indirectly.

Other Coordinate Systems

- Linear Coordinate System



- Hyperbolic Coordinate System



Notes:

The advantage of using other coordinate systems with the CORDIC algorithm is that it allows more functions to be calculated. The drawback is that the system becomes more complex. The set of rotation angles used for the Circular Coordinate System are no longer valid when using the CORDIC algorithm with a Linear or Hyperbolic system. Hence, two other sets of angles are used for rotations made in these systems.

We shall see shortly that the CORDIC equations can be generalised for the 3 coordinate systems and that this involves the introduction of two new variables to the equations. One of these new variables ($e^{(i)}$) represents the set of angles used to represent rotations in the appropriate coordinate system.

Generalised CORDIC Equations

- With the addition of two other Coordinate Systems the CORDIC equations can now be generalised to accommodate all three systems:

$$x^{(i+1)} = (x^{(i)} - \mu d_i (2^{-i} y^{(i)}))$$

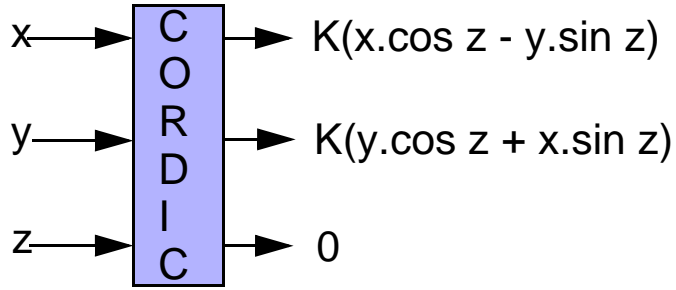
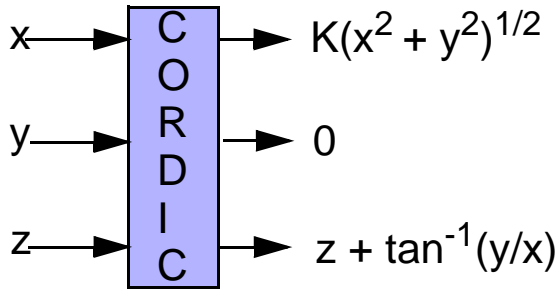
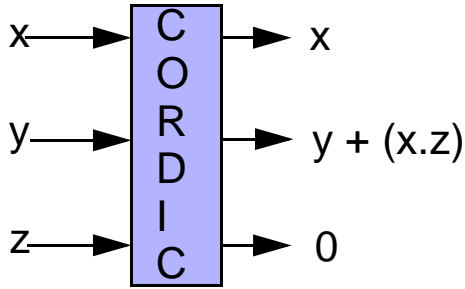
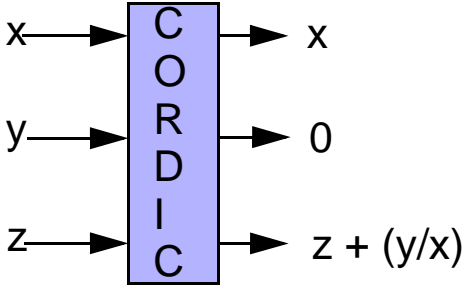
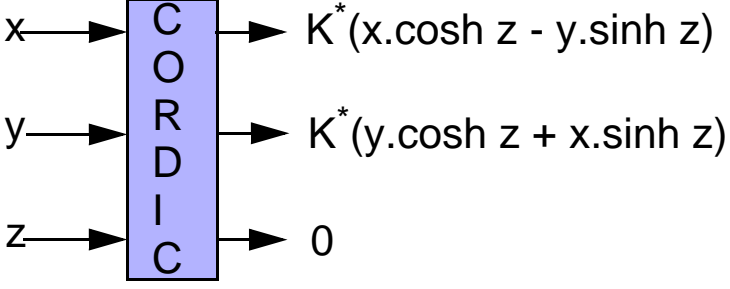
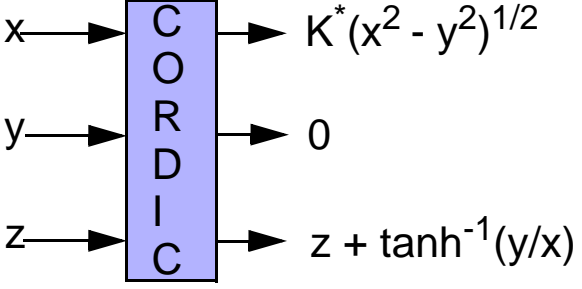
$$y^{(i+1)} = (y^{(i)} + d_i (2^{-i} x^{(i)}))$$

$$z^{(i+1)} = z^{(i)} - d_i e^{(i)}$$

- Circular Rotations: $\mu = 1, e^{(i)} = \tan^{-1} 2^{-i}$
- Linear Rotations: $\mu = 0, e^{(i)} = 2^{-i}$
- Hyperbolic Rotations: $\mu = -1, e^{(i)} = \tanh^{-1} 2^{-i}$

Notes:

Summary of CORDIC Functions

	Rotation Mode: $d_i = \text{sign}(z^{(i)}); z^{(i)} \rightarrow 0$	Vectoring Mode: $d_i = -\text{sign}(x^{(i)}y^{(i)}); y^{(i)} \rightarrow 0$
Circular $\mu = 1$ $e^{(i)} = \tan^{-1}2^{-i}$	 <p>For $\cos z$ & $\sin z$, set $x = 1/K, y = 0$</p>	 <p>For $\tan^{-1} y$, set $x = 1, z = 0$</p>
Linear $\mu = 0$ $e^{(i)} = 2^{-i}$	 <p>For multiplication, set $y = 0$</p>	 <p>For division, set $z = 0$</p>
Hyperbolic $\mu = -1$ $e^{(i)} = \tanh^{-1}2^{-i}$	 <p>For $\cosh z$ & $\sinh z$, set $x = 1/K^*, y = 0$</p>	 <p>For $\tanh^{-1}y$, set $x = 1, z = 0$</p>



Notes:

When using the CORDIC algorithm for Hyperbolic rotations the scaling factor K is different from the one used for Circular rotations.

The Hyperbolic scaling factor is denoted K^* and is calculated using the equation:

$$K^*_n = \prod_n \left(\sqrt{1 - 2^{(-2i)}} \right)$$

$$K^*_n \rightarrow 0.82816 \text{ as } n \rightarrow \infty$$

$$1/K^*_n \rightarrow 1.20750 \text{ as } n \rightarrow \infty$$

n = number of iterations

Other Functions

- Although the CORDIC algorithms can only compute a limited number of functions directly, there are many more that can be achieved indirectly:

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

$$\tanh z = \frac{\sinh z}{\cosh z}$$

$$\ln w = 2 \tanh^{-1} \left| \frac{w-1}{w+1} \right|$$

$$e^z = \sinh z + \cosh z$$

$$w^t = e^{t \ln w}$$

$$\tan^{-1} w = \tan^{-1} \frac{\sqrt{1-w^2}}{w}$$

$$\sin^{-1} w = \tan^{-1} \frac{w}{\sqrt{1-w^2}}$$

$$\cosh^{-1} w = \ln(w + \sqrt{1-w^2})$$

$$\sinh^{-1} w = \ln(w + \sqrt{1+w^2})$$

$$\sqrt{w} = \sqrt{(w+1/4)^2 - (w-1/4)^2}$$

Notes:

There are many more functions that can be computed indirectly using the CORDIC algorithm.

Example: calculate $\tan z$

First, directly calculate $\cos z$ and $\sin z$ using the CORDIC system in Circular Rotation Mode.

Second, feed these values back into the system to divide the latter by the former using Linear Vectoring Mode, which will yield $\tan z$.

Precision & Convergence

- For k bits of precision in trigonometric functions, k iterations are required.
- Convergence is guaranteed for Circular & Linear CORDIC using angles in range $-99.7 \leq z \leq 99.7$:
 - for angles outside this range use standard trig identities.
- Elemental rotations using Hyperbolic CORDIC do not converge:
 - convergence is achieved if certain iterations are repeated;
 - $i = 4, 13, 40, \dots, k, 3k+1, \dots$



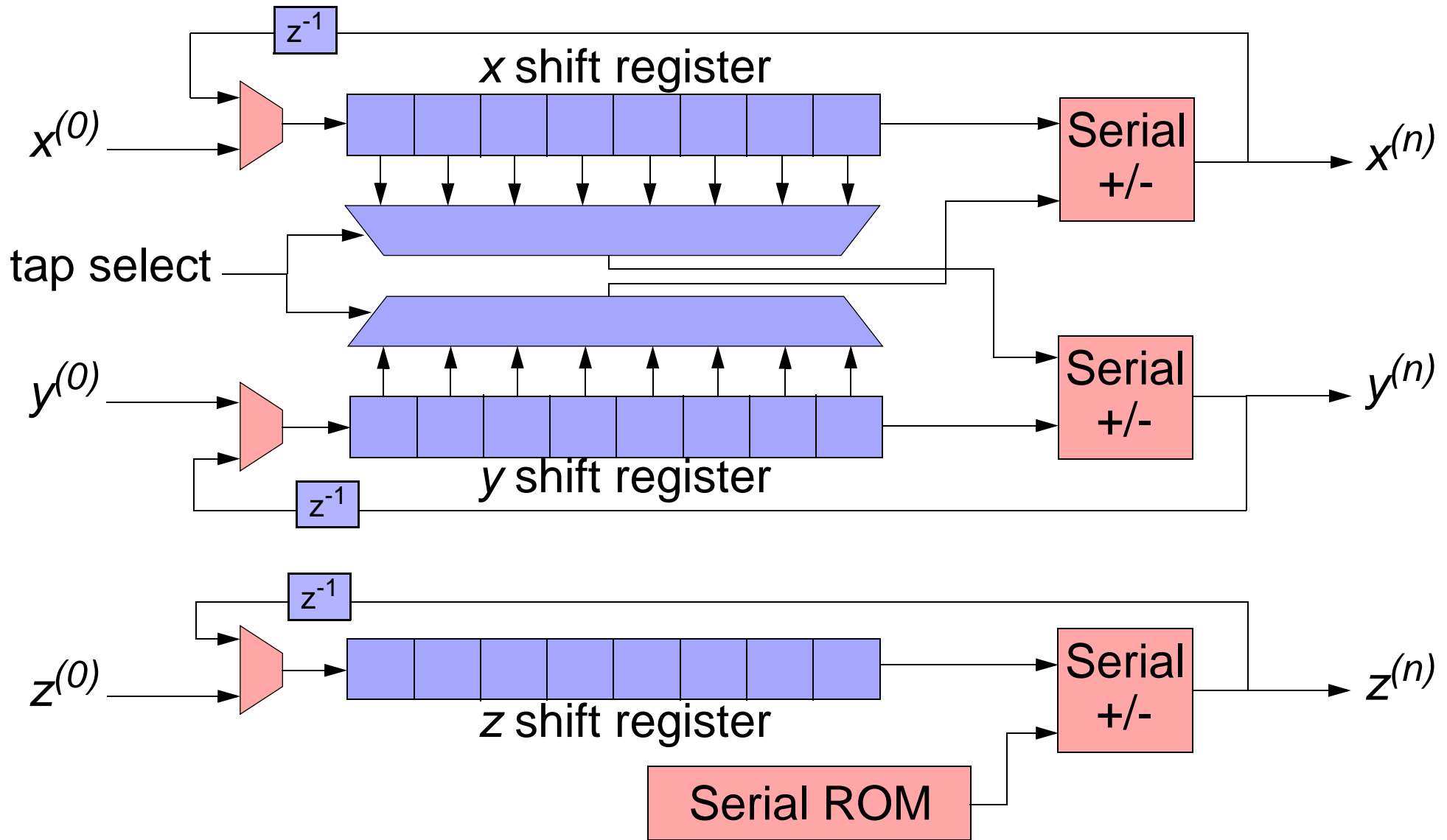
Notes:

- The ideal CORDIC architecture depends on **speed** vs **area** tradeoffs in the intended application.
- A direct translation of the CORDIC equations is an iterative bit-parallel design, however:
 - bit-parallel variable shift shifters do not map well into FPGAs;
 - require several FPGA cells resulting in large, slow design.
- We shall consider an iterative bit-serial solution to illustrate:
 - a minimum area architecture;
 - one implementation of variable shift shifters.



Notes:

Iterative Bit-Serial Design

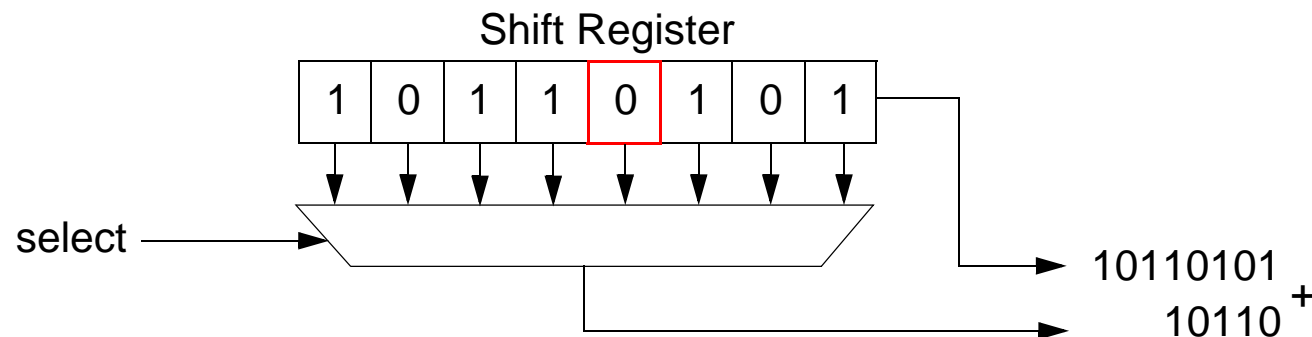


Notes:

This design consists of 3 bit serial adder/subtractors, 3 shift registers and a serial ROM (to hold the rotation angles). Also, 2 multiplexers are required to implement the variable shift shifters. Each shift register in this design must have a length equal to the word width. Consequently the design has to be clocked w times for each iteration (where $w = \text{word width}$).

The design operates by first loading the initial values $x^{(0)}$, $y^{(0)}$ and $z^{(0)}$ into the respective shift registers. Then, the data is shifted right through the serial adder/subtractors and returned to the left end of the shift registers. The variable shift shifters are implemented using 2 multiplexers. At the beginning of each iteration both multiplexers are set to read an appropriate tap from the shift registers. The data from each multiplexer is then passed onto the appropriate adder/subtractor. Also, the sign of the x , y and z registers must be read at the beginning of each iteration so that the adder/subtractors may be set to the correct operation depending on the operating mode. During the final iteration the results can be read directly from the adder/subtractors.

The diagram below illustrates how a shift register and multiplexer are used to implement the variable shift shifter. In this case a 2^{-3} shift has been applied to the data stored in the shift register. Obviously the select line of the multiplexer is set at the beginning of each iteration and will control the size of the shift required.



Using CORDIC for Vector Magnitude

- In this section, the accuracy of the CORDIC algorithm when computing the magnitude of a vector shall be discussed. In particular, the following issues will be presented:
 - How to use CORDIC to compute $\sqrt{x^2 + y^2}$.
 - The error involved in this computation.
 - Choosing the correct parameters to give a desired accuracy.
 - How does this design compare to a Direct approach?



Notes:

For more detail and background on the CORDIC algorithm the following material may be useful:

[1] R. Andraka. A survey of CORDIC algorithms for FPGA based computers. www.andraka.com/cordic.htm

[2] The CORDIC Algorithms. www.ee.byu.edu/ee/class/ee621/Lectures/L22.PDF

[3] CORDIC Tutorial. <http://my.execpc.com/~geezer/embed/cordic.htm>

[4] M. J. Irwin. Computer Arithmetic. <http://www.cse.psu.edu/~cg575/lectures/cse575-cordic.pdf>

Application

- Why use CORDIC to compute the magnitude of a vector?
- The QR-algorithm is used in many adaptive DSP applications.
- Part of this algorithm requires performing a Givens rotation:

$$x_{new} = x \cos \theta - y \sin \theta$$

$$y_{new} = x \sin \theta + y \cos \theta$$

- To perform this requires $\cos \theta$ and $\sin \theta$ which can be found using:

$$\cos \theta = \frac{x}{\sqrt{x^2 + y^2}} \quad \sin \theta = \frac{y}{\sqrt{x^2 + y^2}}$$

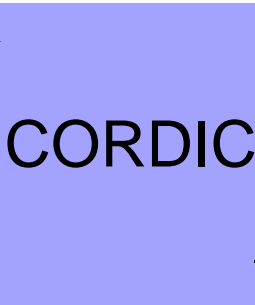
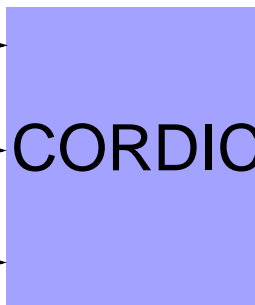
- Hence, this is one application where the magnitude of a vector is required.



Notes:

Computing The Magnitude Of A Vector

- To compute the magnitude of a vector, a Circular coordinate system must be used with Vectoring mode:

Coordinate System	Rotation Mode $z^{(i)} \rightarrow 0; d_i = \text{sign}(z^{(i)})$	Vectoring Mode $y^{(i)} \rightarrow 0; d_i = -\text{sign}(x^{(i)}y^{(i)})$
Circular	 <p> $x \rightarrow$ $K(x.\cos z - y.\sin z)$ $y \rightarrow$ CORDIC $\rightarrow K(y.\cos z + x.\sin z)$ $z \rightarrow$ 0 </p> <p>For $\cos z$ & $\sin z$, set $x = 1/K, y = 0$</p>	 <p> $x \rightarrow$ $K(x^2+y^2)^{1/2}$ $y \rightarrow$ CORDIC $\rightarrow 0$ $z \rightarrow$ $z + \tan^{-1}(y/x)$ </p> <p>For $\tan^{-1} z$, set $x = 1, z = 0$</p>

- The 'K' value is the scaling factor, which can be removed by multiplying the result by 1/K.



Notes:

To compute the magnitude of a vector using CORDIC, circular rotations must be used with Vectoring mode. The x output will then generate the following result:

$$x = K\sqrt{x^2 + y^2}$$

Clearly the scaling factor K must be removed. This can be achieved by multiplying the x output by $1/K$. The value of K is dependent on the number of iterations which is known before hand and thus can be precomputed according to:

$$K(n) = \prod_{i=0}^{n-1} k(i) = \prod_{i=0}^{n-1} \sqrt{1 + 2^{(-2i)}}$$

Simplifying The Equations

- The Generalised CORDIC equations are:

$$x^{(i+1)} = (x^{(i)} - \mu d_i(2^{-i} y^{(i)}))$$

$$y^{(i+1)} = (y^{(i)} + d_i(2^{-i} x^{(i)}))$$

$$z^{(i+1)} = z^{(i)} - d_i e^{(i)}$$

- which, for vector magnitude calculations can be simplified to:

$$x^{(i+1)} = (x^{(i)} - d_i(2^{-i} y^{(i)}))$$

$$y^{(i+1)} = (y^{(i)} + d_i(2^{-i} x^{(i)}))$$

- The angle accumulator can be neglected when computing the magnitude of a vector. Also, $\mu = 1$ for a Circular coordinate system.

Notes:

The full set of Generalised CORDIC equations are:

$$x^{(i+1)} = (x^{(i)} - \mu d_i (2^{-i} y^{(i)}))$$

$$y^{(i+1)} = (y^{(i)} + d_i (2^{-i} x^{(i)}))$$

$$z^{(i+1)} = z^{(i)} - d_i e^{(i)}$$

where,

- Circular Rotations: $\mu = 1$, $e^{(i)} = \tan^{-1} 2^{-i}$
- Linear Rotations: $\mu = 0$, $e^{(i)} = 2^{-i}$
- Hyperbolic Rotations: $\mu = -1$, $e^{(i)} = \tanh^{-1} 2^{-i}$

When using Vectoring mode, $d_i = -\text{sign}(x^{(i)} y^{(i)})$. The x equation is the one that will generate the magnitude of the vector and it is only dependent on the y equation. Hence, the z equation (angle accumulator) can be ignored. This leaves only:

$$x^{(i+1)} = (x^{(i)} - d_i (2^{-i} y^{(i)}))$$

$$y^{(i+1)} = (y^{(i)} + d_i (2^{-i} x^{(i)}))$$