FPGAs *for* DSP 4

CORDIC 算法

Return
DSP*edia* Home

Return
DSP Notes Menu

THIS SLIDE IS BLANK

简介

- 目前的 FPGA 具有 *许多* 乘法器和加法器。然而各种各样的通信技术和矩阵算法则需要三角函数、平方根等的运算。

  如何在 FPGA 上执行这些运算？

  可以使用查找表，或是迭代法

- 本节介绍了 CORDIC 算法；这是一个“移位相加”算法，允许计算不同的三角函数，例如：
  - $\sqrt{x^2 + y^2}$
  - $\cos\theta$, $\tan\theta$, $\sin\theta$
  - 包括除法和对数函数在内的其它函数。

Top

Notes:

关于 CORDIC 算法的细节问题，可参见下面的材料：

[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

CORIDC 技术并不是什么新鲜的东西。事实上它可以追溯到 1957 年由 J. Volder 发表的一篇文章。在上个世纪五十年代，在大型实际的计算机中的实行移位相加受到了当时技术上的限制，所以使用 CORDIC 变得非常必要。到了七十年代，Hewlett Packard 和其他公司出产了手持计算器，许多计算器使用一个内部 CORDIC 单元来计算所有的三角函数（了解这件事的人们一定还记得，那时求一个角度的正切值需要延迟大约 1 秒中）。

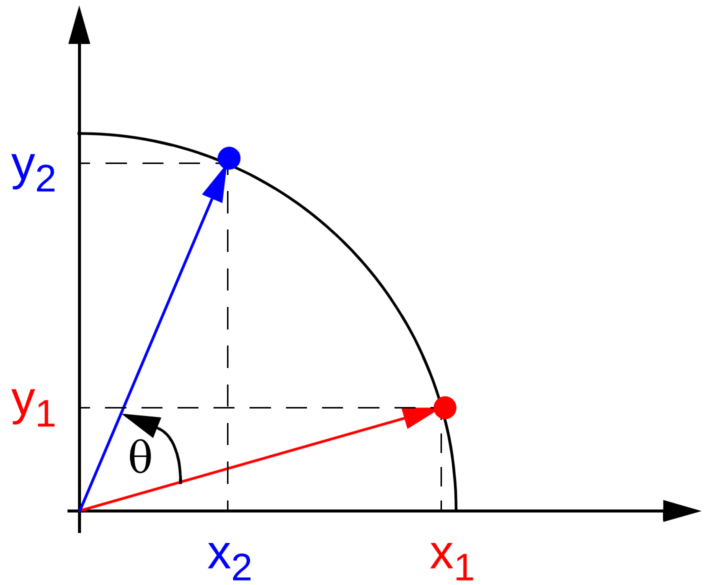
二十世纪八十年代，随着高速度乘法器与带有大存储量的通用处理器的出现，CORDIC 算法变得无关紧要了。然而在二十一世纪的今天，对于 FPGA 来说，CORDIC 一定是在 DSP 应用中（诸如 多输入多输出（MIMO），波束形成以及其他自适应系统）计算三角函数的备选技术。

笛卡尔坐标平面旋转

- 在 xy 坐标平面上将点 ((x_1, y_1) 旋转 θ 角度到点 (x_2, y_2) 的标准方法如下所示：

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

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



- 这被称为是平面旋转、向量旋转或者线性（矩阵）代数中的 Givens 旋转。

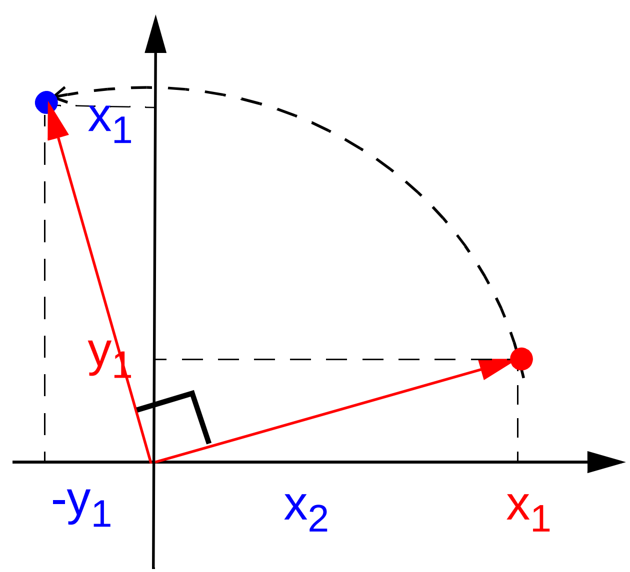
Notes:

上面的方程组同样可写成矩阵向量形式：

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

例如一个 90^o 相移为：

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



伪旋转

- 通过提出因数 cosθ，方程可写成下面的形式：

$$x_2 = x_1\cos\theta - y_1\sin\theta = \cos\theta(x_1 - y_1\tan\theta)$$
$$y_2 = x_1\sin\theta + y_1\cos\theta = \cos\theta(y_1 + x_1\tan\theta)$$

- 如果去除 cosθ 项，我们得到 ***伪旋转*** 方程式：

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

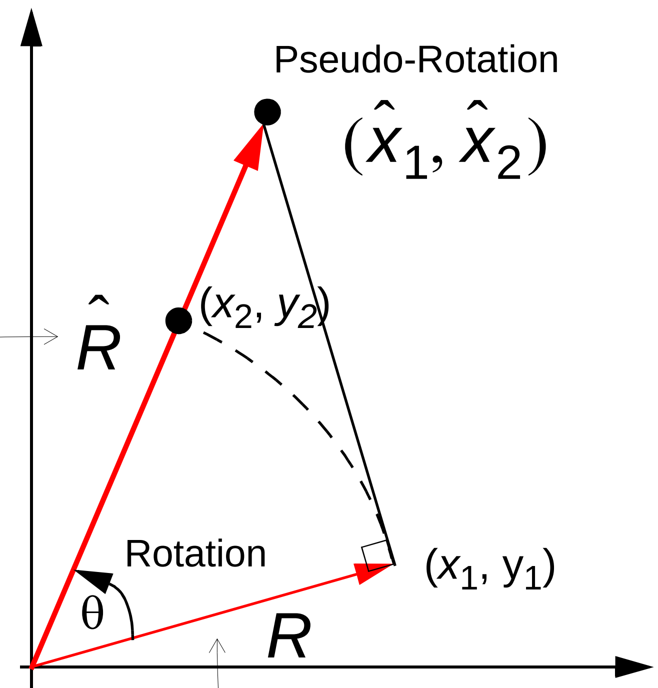
即旋转的角度是正确的，但是 x 与 y 的值增加 $\cos^{-1}\theta$ 倍（由于 $\cos^{-1}\theta > 1$，所以模值变大。

- 注意我们 ***并不能*** 通过适当的数学方法去除 cosθ 项，然而随后我们发现去除 cosθ 项可以简化坐标平面旋转的计算操作。

Notes:

在 xy 坐标平面中：

$$\hat{R} = \frac{R}{\cos\theta} = \frac{\sqrt{x_1^2 + y_1^2}}{\cos\theta}$$



$$R = \sqrt{x_1^2 + y_1^2} = \sqrt{x_2^2 + y_2^2}$$

因此经过伪旋转之后，向量 R 的模值将增加 $1/\cos\theta$ 倍。

向量旋转了正确的角度，但模值出现错误。

CORDIC 方法

- CORDIC 方法的核心是 (伪) 旋转角度 θ，其中 $\tan\theta^i = 2^{-i}$。故方程为：

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

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

- 下面的表格指出用于 CORDIC 算法中每个迭代 (i) 的旋转角度 (精确到 9 位小数):

由于i是整数，所以对应的角度值都是一一确定的，只能通过几个角度的加减组合来达到你所想要的角度值

注意有三个方面的变化：
1.角度累加（减）
2.坐标值累加（减）
3.向量的模（也就是长度的，相对于横纵坐标的）累加（减）

这三个累加的变化时不一样的，注意区别，角度的累加和长度的累加有一定的对应关系

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

Top

Notes:

在这里，我们把变换改成了迭代算法。我们将各种可能的旋转角度加以限制，使得对任意角度 θ 的旋转能够通过一系列连续小角度的旋转迭代 i 来完成。旋转角度遵循法则：$\tan\theta^{(i)} = 2^{-i}$，遵循这样的法则，乘以正切项变成了移位操作。

前几次迭代的形式为：
第 1 次迭代：旋转 45^o;　第 2 次迭代：旋转 26.6^o,　第 3 次迭代：旋转 14^o　等。

很显然，每次旋转的方向都影响到最终要旋转的累积角度。在 $-99.7 \le \theta \le 99.7$ 的范围内的任意角度都可以旋转。满足法则的所有角度的总和 $\tan\theta^{i} = 2^{-i}$ 为 99.7。对于该范围之外的角度，可使用三角恒等式转化成该范围内的角度。当然，角分辨率的数据位数与最终的精度有关。

i	tanθ	Angle, θ	cosθ
1	1	45.0000000000	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

这个系数是相对于连续旋转13（按同一个方向转）次得到的要相乘的系数

连续旋转13次，旋转的角度是99.7或者-99.7（假设每次都是角度相加或相减，也就是转的方向相同），每转一个角度a,对应的坐标值变为原来的1/cosa倍,连续旋转13次，那么最终的坐标的值就是原来的1/（cosa1*cosa2……）倍，也就是1.6467602

cos 45 x cos 26.5 x cos 14.03 x cos 7.125 ... x cos 0.0139 =　0.607252941

1/0607252941 = 1.6467602。因此，在 13 次旋转以后，为了标定伪旋转的幅度，要求乘以一个系数 1.64676024187。角分辨率的数据位数对最终的旋转精度非常关键。

角度累加器

- 前面所示的伪旋转现在可以表示为（对每次迭代）：

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

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

坐标值的累加

- 在这里我们引入第三个方程，被称为角度累加器，用来在每次迭代过程中追踪累加的旋转角度：

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

where d_i = +/- 1

角度的累加，与

- 符号 d_i 是一个判决算子，用于确定旋转的方向。

di的值由旋转的方向来决定，刚好tana是一个奇函数，tana=-tan(-a),cos(a)是一个偶函数，所以角度的累加，当为逆时针的时候，di为1，反时针的di为-1，而角度a无需变为-a,还是用tana,cosa就可以了

Top

Notes:

在这里，我们把每次迭代的方程表示为：

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

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

其中判决算子 d_i 决定旋转的方向是顺时针还是逆时针。d_i 的值取决于下面将要讨论的 **操作模式** 。

这里我们引入了名为 **角度累加器** 的第三个等式，用于追踪每次迭代中旋转的角度的叠加：

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

上述三个方程式为圆周坐标系中用于角度旋转的 CORDIC 算法的表达式。在本章的后续部分中我们还将看到 CORDIC 算法被用于其它的坐标系，通过使用这些坐标系可以执行更大范围的函数计算。

移位 - 加法算法

- 因此，原始的算法现在已经被减化为使用向量的伪旋转来表示的迭代 移位 - 相加 算法：

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

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

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

- 因此每个迭代需要：
  - 2 次移位

    2的-i次方，用移位实现，每右移n位就把原来的是乘以了一个2的-i次方了，每一次迭代x,y的坐标值分别作一次移位，共两次
  - 1 次查找表 ($\theta^{(i)}$ 值)

    每一次迭代，都会有一个相对固定的角度的累加，这个角度 分别是2的-i次方对应的角度值，用查找表实现，（因为一个i就对应了一个固定的角度值）
  - 3 次加法

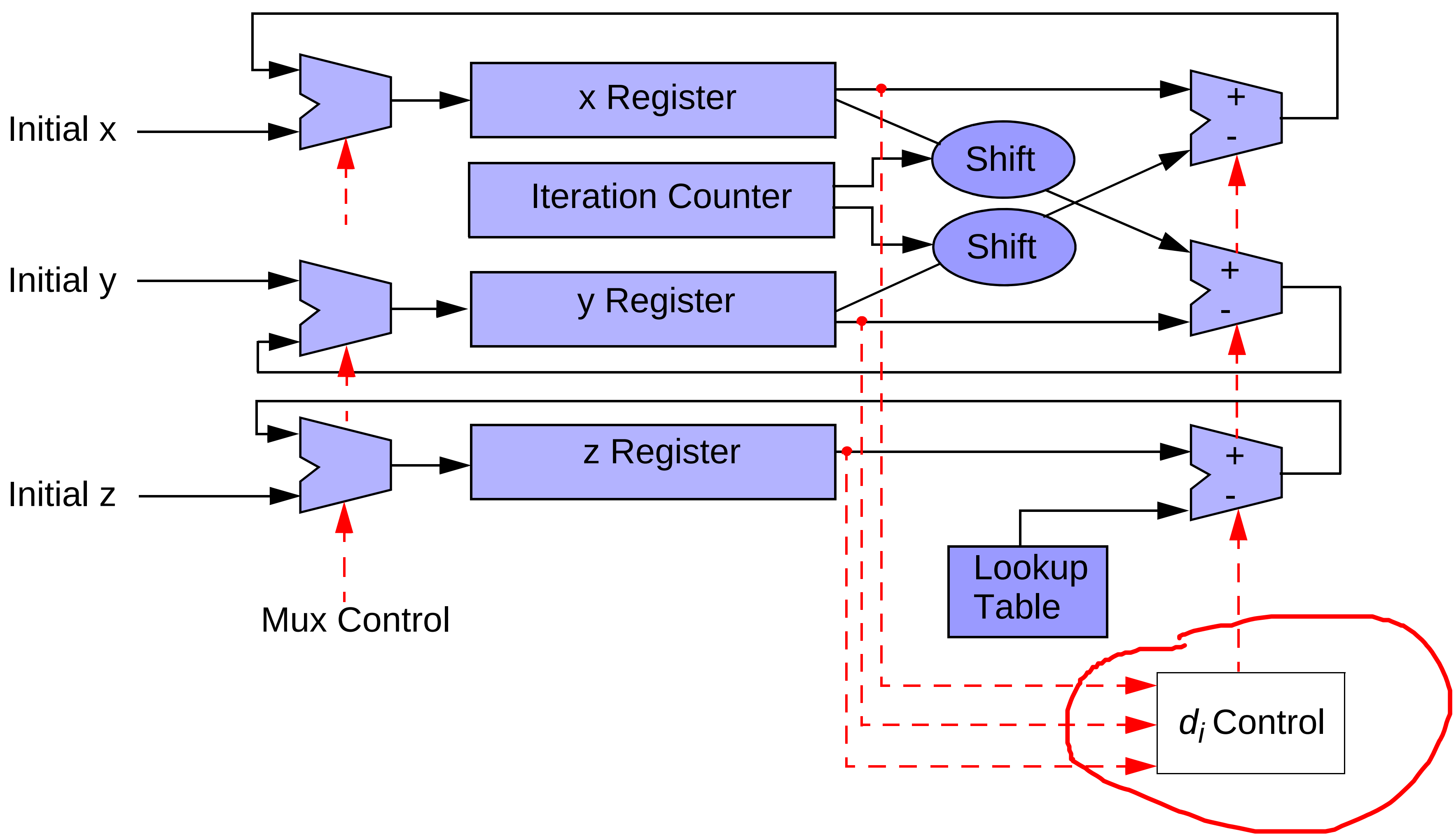
    x,y,z的累加，共三次

Top

Notes:

前面提到的去除 cos θ 项的原因是显而易见的。当将该项去除时，转换公式已经被简化为伪旋转的迭代移位相加计算。

CORDIC 硬件：



伸缩因子

n次旋转，不管旋转的方向，cosa都是正的，所以就直接相乘了，当旋转的角度和次数确定以后，kn的值也就确定了

- 伸缩因子是伪旋转的副产物。
- 当简化算法以允许伪旋转时，cosθ 项被忽略。
- 这样，输出 $x^{(n)}, y^{(n)}$ 被伸缩 K_n 倍，其中：

把kn和2的-i次方联系起来了

三角函数变换公式
1/cos^2(a)=1+tan^2(a)

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

- 然而如果迭代次数可知，则我们可以预先计算 伸缩因子 K_n。
- 同样，$1/K_n$ 也可被预先计算以得到 $x^{(n)}$ 和 $y^{(n)}$ 的真值。

Top

Notes:

为了简化 Givens 旋转，我们去除了 $\cos\theta$ 项以执行伪旋转。然而，该简化引发了**负面效应**。输出值 $x^{(n)}$ 和 $y^{(n)}$ 被乘以一个因子 K_n，该因子被称为伸缩因子，这里：

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

旋转的角度和次数不同，kn的值是不同的，但是是可以确定的

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

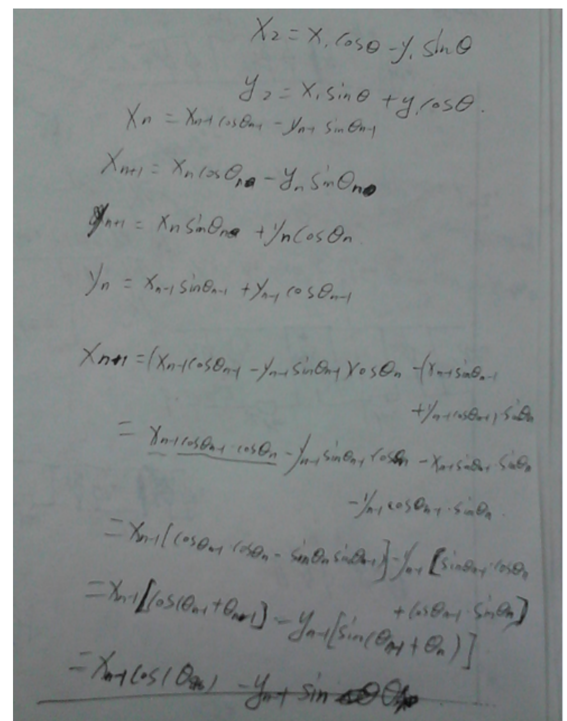
当依次旋转的角度增加，最大达到99.7度时，kn接近1.6476，i越大2的-i次方越小，旋转的角度也越小

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

$$n = \text{number of iterations}$$

然而，如果我们已知了将被执行的迭代次数，我们便可以预先计算出 $1/K_n$ 的值，并通过将 $1/K_n$ 与 $x^{(n)}$ 和 $y^{(n)}$ 相乘来校正 $x^{(n)}$ 和 $y^{(n)}$ 的最终值。

可以调整大小来看旋，这个是证明下面的旋转模式里这个式子正确性



$X_2 = X_1\cos\theta - y_1\sin\theta$

$y_2 = X_1\sin\theta + y_1\cos\theta.$

$X_n = X_{n-1}\cos\theta_{n-1} - y_{n-1}\sin\theta_{n-1}$

$X_{n+1} = X_n\cos\theta_n - y_n\sin\theta_n$

$y_{n+1} = X_n\sin\theta_n + y_n\cos\theta_n.$

$y_n = X_{n-1}\sin\theta_{n-1} + y_{n-1}\cos\theta_{n-1}$

$X_{n+1} = (X_{n-1}\cos\theta_{n-1} - y_{n-1}\sin\theta_{n-1})\cos\theta_n - (X_{n-1}\sin\theta_{n-1} + y_{n-1}\cos\theta_{n-1})\sin\theta_n$

$= X_{n-1}\cos\theta_{n-1}\cdot\cos\theta_n - y_{n-1}\sin\theta_{n-1}\cos\theta_n - X_{n-1}\sin\theta_{n-1}\sin\theta_n - y_{n-1}\cos\theta_{n-1}\cdot\sin\theta_n.$

$= X_{n-1}[\cos\theta_{n-1}\cos\theta_n - \sin\theta_n\sin\theta_{n-1}] - y_{n-1}[\sin\theta_{n-1}\cos\theta_n + \cos\theta_{n-1}\sin\theta_n]$

$= X_{n-1}[\cos(\theta_{n-1}+\theta_{n})] - y_{n-1}[\sin(\theta_{n-1}+\theta_n)]$

$= X_{n-1}\cos(\theta) - y_{n-1}\sin\theta$

旋转模式

z0表示的最终要旋转的角度之和，其中的角度有正有负，但最终的角度之和是确定的。
y0,x0标志没有旋转时候的x,y的坐标值

- CORDIC 方法有两种操作模式；
- 工作模式决定了控制算子 d_i 的条件；
- 在旋转模式中选择：$d_i = \mathrm{sign}(z^{(i)}) \Rightarrow z^{(i)} \rightarrow 0$;
- n 次迭代后我们得到：

把xn,yn,的表达式代入xn+1,yn+1里的表达式，再用三角函数里的和（差）公式来化简

1

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

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

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

最终的坐标值就只和初始的没旋转之前的坐标值和最终旋转要达到的角度值有关了

- 通过设置 $x^{(0)} = 1/K_n$ 和 $y^{(0)} = 0$ 可以计算 $\cos z^{(0)}$ 和 $\sin z^{(0)}$。

这个是自己预先要计算的，干好可以把kn相乘得到1，简化了式子，不同的角度kn的值是不同的

Notes:

Top

旋转模式中，判决算子 d_i 满足下面条件：

+30为逆时针的最后要累加的角度的值，设最终累加值为30，最后累加的值和还是30.

zi- θ i就得到新的累加的值了

如果初始相位为0的话，就最后一个累加值为30了，那就需要zi+ θ i了

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

i	tanθ	Angle, θ	cosθ
1	1	45.0000000000	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

$K_n \to 1.6476$ as $n \to \infty$

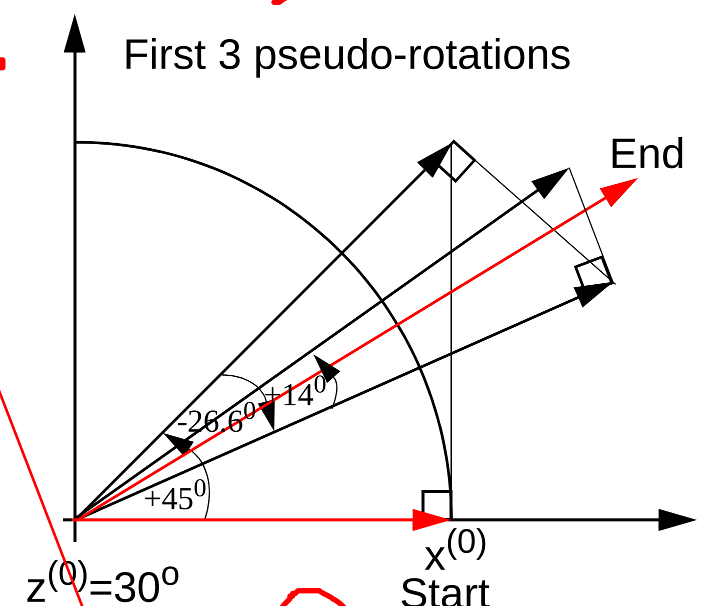
$1/K_n \to 0.6073$ as $n \to \infty$

因此，我们输入 $x^{(0)}$ 和 $z^{(0)}$ ($y^{(0)} = 0$)，然后通过迭代使 $z^{(0)}$ 取值趋近于 0。

例如：当 $z^{(0)} = 30^o$ 时，计算 $\sin z^{(0)}$, $\cos z^{(0)}$

2

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



旋转的次数不同的话，kn也不一样啊，？？但迭代次数足够大的时候，可以近似等于0.6……

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

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

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

xn接近0表示角度的累加是30了

1,2式对比，可以看出xn与角度之间的关系，太

向量模式

对1式进行极坐标转化即可得到

- 在向量模式中选择：$d_i = -\text{sign}(x^{(i)}y^{(i)}) \Rightarrow y^{(i)} \rightarrow 0$
- 经过 n 次迭代后：

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

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

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

- 通过设定 $x^{(0)} = 1$ 和 $z^{(0)} = 0$ 来计算 $\tan^{-1} y^{(0)}$。

Notes:

向量模式中，判决算子 d_i 满足下面条件：

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

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

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

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

因此，我们输入 $x^{(0)}$ 和 $y^{(0)}$ ($z^{(0)} = 0$)，并通过迭代使 $y^{(0)}$ 取值趋近于 0。

例如：当 $y^{(0)} = 2$ 并且 $x^{(0)} = 1$ 时，计算 $\tan^{-1}(y^{(0)}/x^{(0)})$

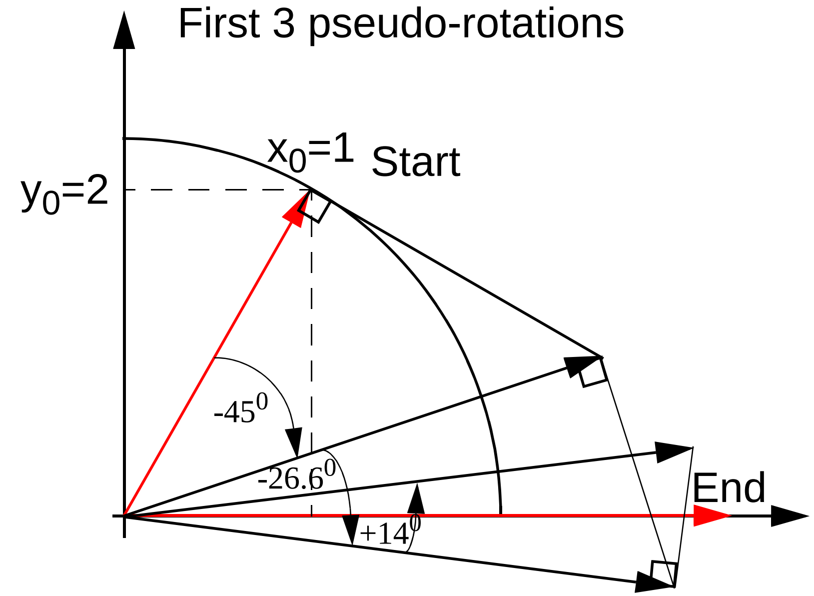
$$x^{(i+1)} = (x^{(i)} - d_i(2^{-i}y^{(i)}))$$

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

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

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

First 3 pseudo-rotations

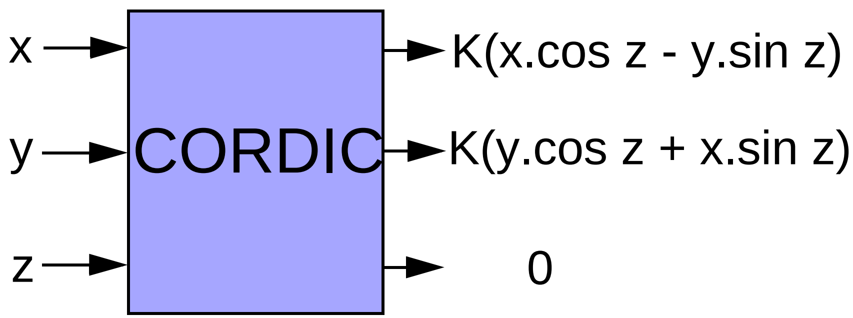
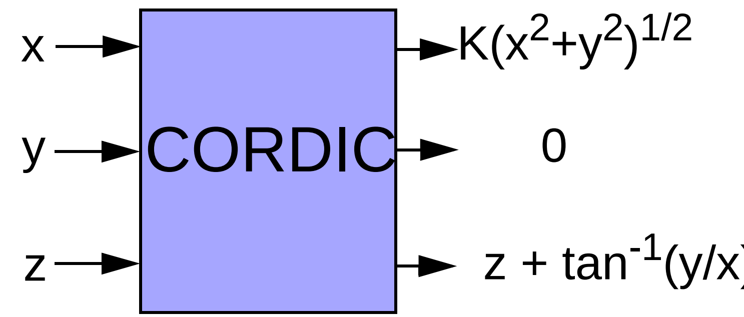


圆坐标系

- 目前我们仅涉及了圆坐标系中的伪旋转。

判决因子不一样，应该就是两种根本不同的方法，不仅仅只是坐标的变换

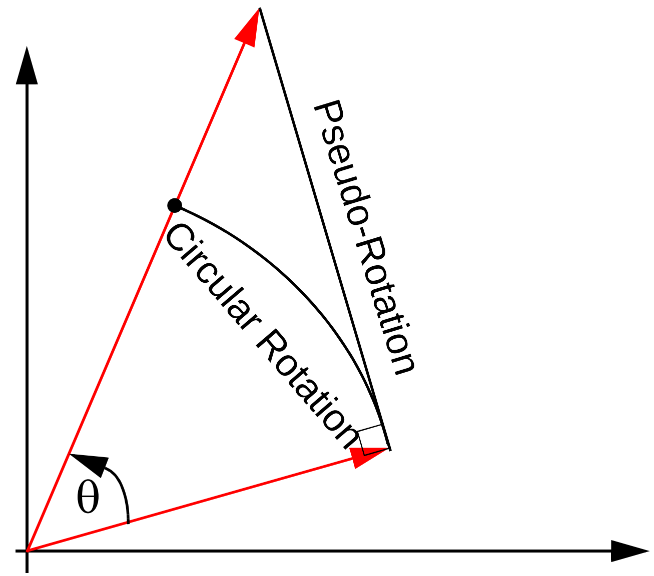
- 因此，下面的函数可被计算：

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	 For cos z & sin z, set x = 1/K, y = 0	 For $\tan^{-1} y$, set x = 1, z = 0

- 然而，如果使用其它的坐标系，可以计算更多的函数。

Notes:

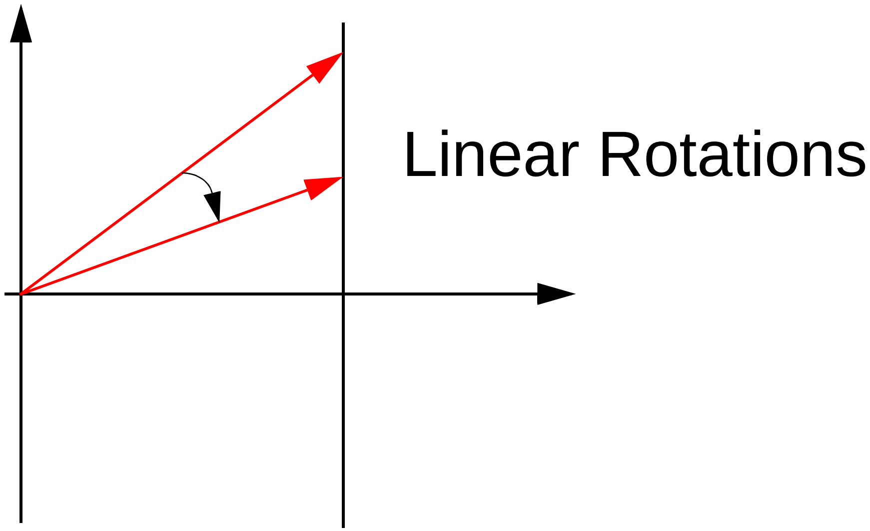
使用圆周坐标系中的旋转，可被计算的函数的数目受到了限制。



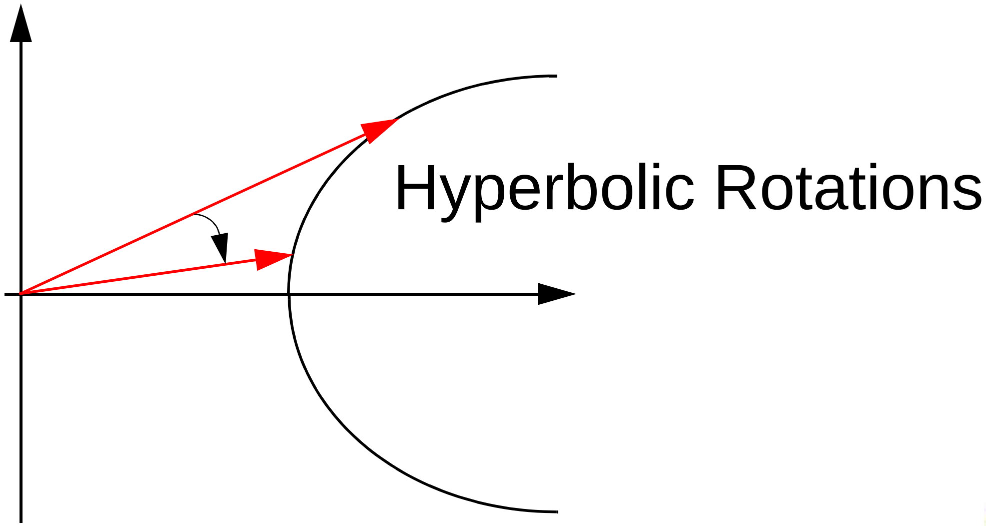
然而，我们将会看到，通过考虑其它坐标系中的旋转，我们可以直接计算更多的函数，如乘法和除法，进而间接计算更多的其它函数。

其它的坐标系

- 线性坐标系



- 双曲线坐标系



Top

Notes:

使用其它坐标系的 CORDIC 算法的优点是可以计算更多的函数，而缺点则是系统将变得更加复杂。当把 CORDIC 算法用于线性或双曲坐标系时，在圆周坐标系中的旋转角度集将不再有效。所以，这些系统应使用其它的两种旋转角度集。

我们会发现，可以推导出可在 3 个坐标系中表示 CORDIC 方程的通用公式。这意味着在方程式中引入两个新变量。其中一个新变量 ($e^{(i)}$) 代表了适当的坐标系中用于表示旋转的角度集。

当把 CORDIC 算法用于双曲旋转时，伸缩因子 K 与圆周旋转的因子有所不同。

双曲伸缩因子，记为 K^*，使用下列方程计算：

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

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

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

$$n = \text{number of iterations}$$

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

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

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

- 圆周旋转： $\mu = 1, e^{(i)} = \tan^{-1} 2^{-i}$
- 线性旋转： $\mu = 0, e^{(i)} = 2^{-i}$
- 双曲线旋转： $\mu = -1, e^{(i)} = \tanh^{-1} 2^{-i}$

通用的 CORDIC 方程

- CORDIC 方程可被归纳到包括其它两个坐标系在内的三个坐标系中：

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

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

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

- 圆周旋转： $\mu = 1, e^{(i)} = \tan^{-1} 2^{-i}$
- 线性旋转： $\mu = 0, e^{(i)} = 2^{-i}$
- 双曲线旋转： $\mu = -1, e^{(i)} = \tanh^{-1} 2^{-i}$

Notes:

Top

CORDIC 函数总结

划红线的是是可以求出来的函数值

	Rotation Mode: d_i=sign($z^{(i)}$); $z^{(i)} \rightarrow 0$	Vectoring Mode: d_i=-sign($x^{(i)}y^{(i)}$); $y^{(i)} \rightarrow 0$
Circular $\mu = 1$ $e^{(i)} = \tan^{-1}2^{-i}$	CORDIC: $x \rightarrow K(x.\cos z - y.\sin z)$ $y \rightarrow K(y.\cos z + x.\sin z)$ $z \rightarrow 0$ For cos z & sin z, set x = 1/K, y = 0	CORDIC: $x \rightarrow K(x^2 + y^2)^{1/2}$ $y \rightarrow 0$ $z \rightarrow z + \tan^{-1}(y/x)$ For $\tan^{-1}$ y, set x = 1, z = 0
Linear $\mu = 0$ $e^{(i)} = 2^{-i}$	CORDIC: $x \rightarrow x$ $y \rightarrow y + (x.z)$ $z \rightarrow 0$ For multiplication, set y = 0	CORDIC: $x \rightarrow x$ $y \rightarrow 0$ $z \rightarrow z + (y/x)$ For division, set z = 0
双曲线旋转坐标 Hyperbolic $\mu = -1$ $e^{(i)} = \tanh^{-1}2^{-i}$	CORDIC: $x \rightarrow K^*(x.\cosh z - y.\sinh z)$ $y \rightarrow K^*(y.\cosh z + x.\sinh z)$ $z \rightarrow 0$ For cosh z & sinh z, set x = 1/K^*, y = 0	CORDIC: $x \rightarrow K^*(x^2 - y^2)^{1/2}$ $y \rightarrow 0$ 这里的k和在旋转模式下的圆周坐标的k不一样 $z \rightarrow z + \tanh^{-1}(y/x)$ For $\tanh^{-1}$y, set x = 1, z = 0

Notes:

Top

当把 CORDIC 算法用于双曲旋转时，伸缩因子 K 与圆周旋转的因子有所不同。

双曲伸缩因子，记为 K^*，使用下列方程计算：

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

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

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

$$n = \text{number of iterations}$$

其它的函数

画红线的部分是需要间接利用之前的几个坐标系里的课求出的函数值求出的

- 尽管 CORDIC 算法仅能够直接计算少量的函数，但更多的函数可以通过间接的方法来获得：

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

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

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

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

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

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

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

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

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

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

Notes:

我们可以间接使用 CORDIC 算法来间接计算许多函数。

例如：计算 tan z

首先，在圆周旋转模式中使用 CORDIC 算法来直接计算 cos z 和 sin z。

其次，将这些值回馈到系统中，使用线性矢量模式，并用前者除以后者，将得到 tan z。

精度 & 收敛性

- 在三角函数中，*k* 位的精度要求 *k* 次迭代。?
- 使用 -99.7 ≤ *z* ≤ 99.7 范围内角度，圆周和线性 CORDIC 一定收敛
    - 对于该范围之外的角度，需要使用标准三角恒等式。
- 使用双曲 CORDIC 时，单元的旋转不一定收敛：
    - 如果重复某些迭代，CORDIC 将收敛；
    - $i = 4, 13, 40, \ldots\ldots, k, 3k+1, \ldots$

Notes:

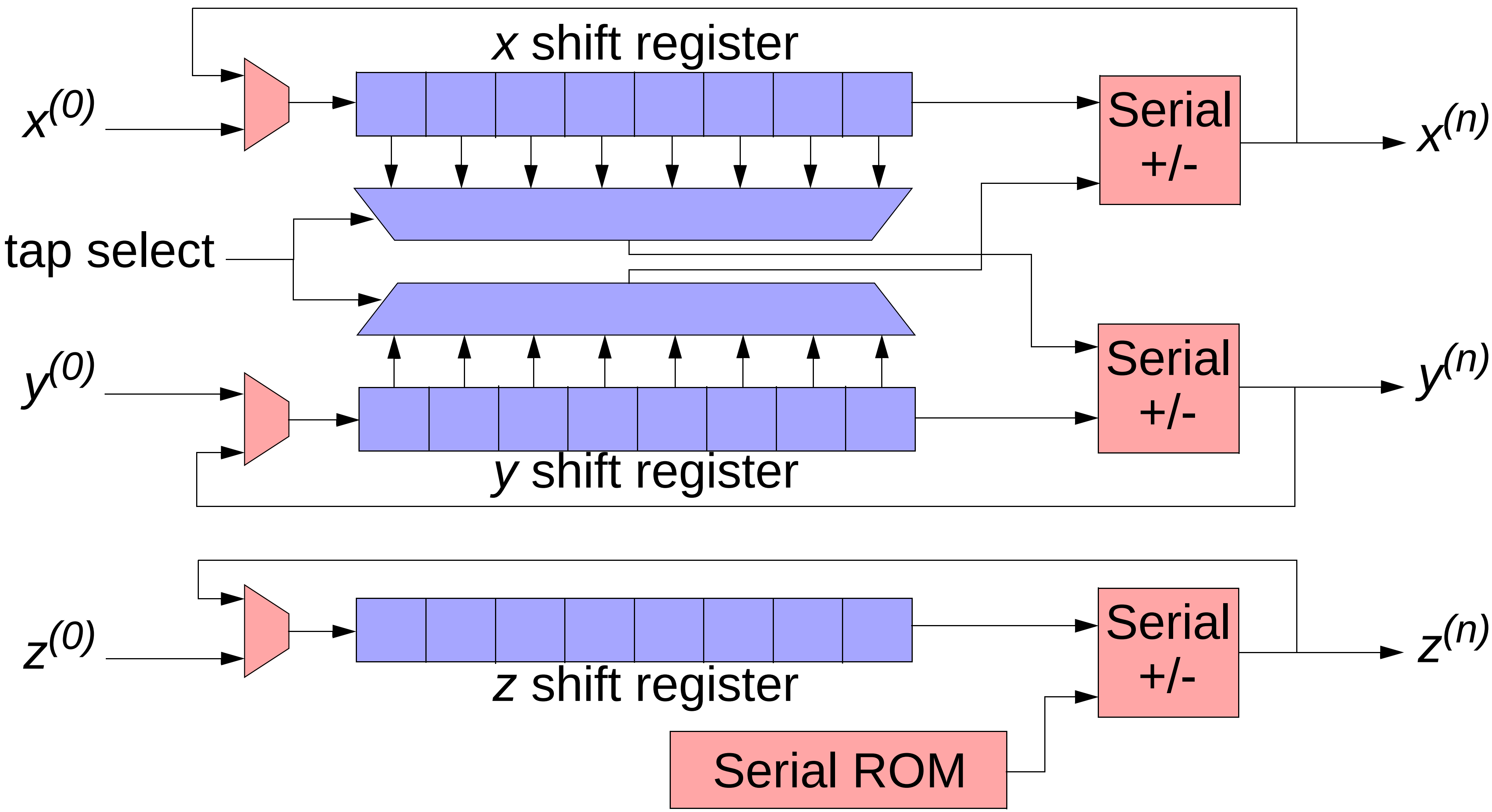
FPGA 实现

- 理想 CORDIC 架构取决于具体应用中 速率 与 面积 的权衡。
- 可以将 CORDIC 方程直接翻译成迭代型的位并行设计，然而：
  - 位并行变量移位器不能很好地映射到 FPGA 中；
  - 需要若干个 FPGA 单元。导致设计规模变大而设计时间变长。
- 我们将要考虑一个位 - 串行解决方案来说明：
  - 最小面积的架构；
  - 变量移位寄存器的实现。

变量移位寄存器！！！！！！！！！每次移位的次数是不一样的！！！！！！

Notes:

迭代型位 - 串行设计


x shift register
$x^{(0)}$
Serial
+/-
$x^{(n)}$
tap select
$y^{(0)}$
Serial
+/-
$y^{(n)}$
y shift register
$z^{(0)}$
Serial
+/-
$z^{(n)}$
z shift register
Serial ROM

Top

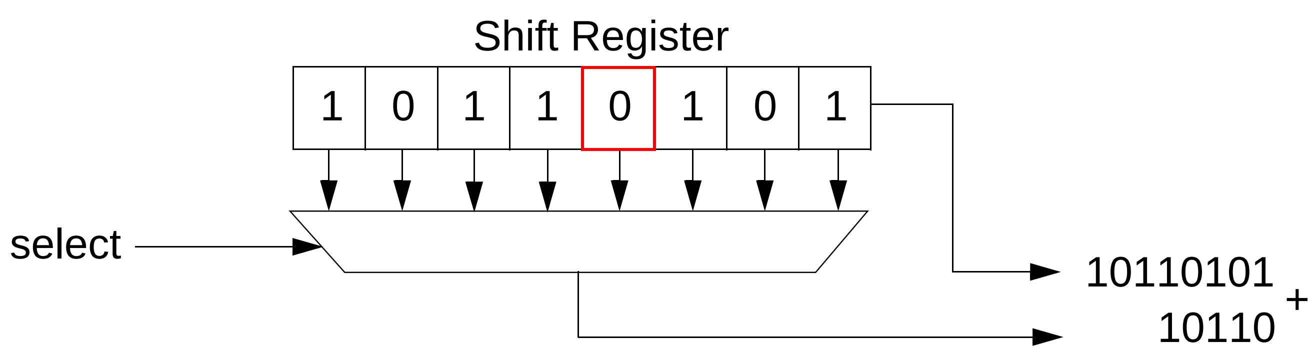
Notes:

该设计包括 3 个位串行加法器/减法器、3 个移位寄存器以及一个串行 ROM (存放旋转角度)。同时需要 2 个复用器以实现可变位移器。本设计中每个移位寄存器必须具有与字宽相等的长度。因此每次迭代都需要将该逻辑电路运行 w 次 (其中 w = 字宽度)。

?

首先通过将初值 $x^{(0)}$, $y^{(0)}$ 和 $z^{(0)}$ 装入相关的移位寄存器中来运行。因此，数据通过加法器 / 减法器右移并被返回到移位寄存器的左端。变量移位器通过 2 个复用器来实现。在每个迭代的初始阶段，两个复用器均被设置为从移位寄存器中读取合适的抽头数据。来自每个复用器的数据被传送到了合适的加法器 / 减法器。在每次迭代的开始，x, y 和 z 寄存器的符号被读出以便将加法器 / 加法器设置到正确的操作模式。在最后一次迭代过程中，结果可直接从加法器 / 减法器中读取。

下图说明了使用移位寄存器和复用器实现可变移位器。图中所示对储存在移位寄存器中的数据执行 2^{-3} 的移位操作。显然需要在每个迭代的初始就设定好多路复用器的选择线，这将能够控制需要移位的次数。



加一个选择器来控制移位的次数，移位的次数依次增加一次

使用 CORDIC 计算向量幅度

- 本节将讨论计算向量幅度时 CORDIC 算法的准确性。特别要说明以下几点：
    - 如何使用 CORDIC 来计算 $\sqrt{x^2+y^2}$。
    - 该计算过程中存在的误差。
    - 选择正确的参数来获得希望的精度。
    - 与直接方法相比，这种设计有何特色？

Top

Notes:

关于 CORDIC 算法的基础以及细节问题，可参见下面的材料 :

[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

应用

- 为什么使用 CORDIC 来计算向量幅度？
- 许多自适应 DSP 应用中使用了 QR- 算法。
- 该算法的一部分需要执行 Givens 旋转：

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

- 为了执行 Givens 旋转，需要计算 $\cos\theta$ 与 $\sin\theta$ 。可用下面方式求得：

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

- 因此这是一种需要计算向量幅度的应用例子。

Notes:

Top

向量幅度计算

- 为计算向量的幅度，必须同时使用圆周坐标系与向量模式：

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	x → CORDIC → $K(x.\cos z - y.\sin z)$ y → CORDIC → $K(y.\cos z + x.\sin z)$ z → CORDIC → 0 For cos z & sin z, set $x = 1/K$, $y = 0$	x → CORDIC → $K(x^2+y^2)^{1/2}$ y → CORDIC → 0 z → CORDIC → $z + \tan^{-1}(y/x)$ For $\tan^{-1} z$, set $x = 1$, $z = 0$

- ‘K’ 的值为标度因子，它可通过对结果乘 1/K 来去除。

Top

Notes:

为了使用 CORDIC 计算一个向量的幅度，必须要同时使用圆周旋转和向量模式。*x* 输出将产生如下的结果：

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

很明显标度因子 *K* 必须被去除。这可以通过 把 *x* 与 *1/K* 相乘来实现。*K* 值取决于迭代的次数。迭代次数是事先知道的，因此，可以根据下面的等式计算：

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

简化方程组

- CORDIC 方程组被归纳为：

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

- 向量的模值计算可简化为：

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

- 当计算向量模值时， 角度累加器可以被忽略。 而且对于圆周坐标系，$\mu = 1$。

结果和中间值和di判断都与角度累加器无关

Top

Notes:

通用 CORDIC 方程是：

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

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

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

其中，

- 圆周旋转：$\mu = 1, e^{(i)} = \tan^{-1} 2^{-i}$
- 线性旋转：$\mu = 0, e^{(i)} = 2^{-i}$
- 双曲线旋转：$\mu = -1, e^{(i)} = \tanh^{-1} 2^{-i}$

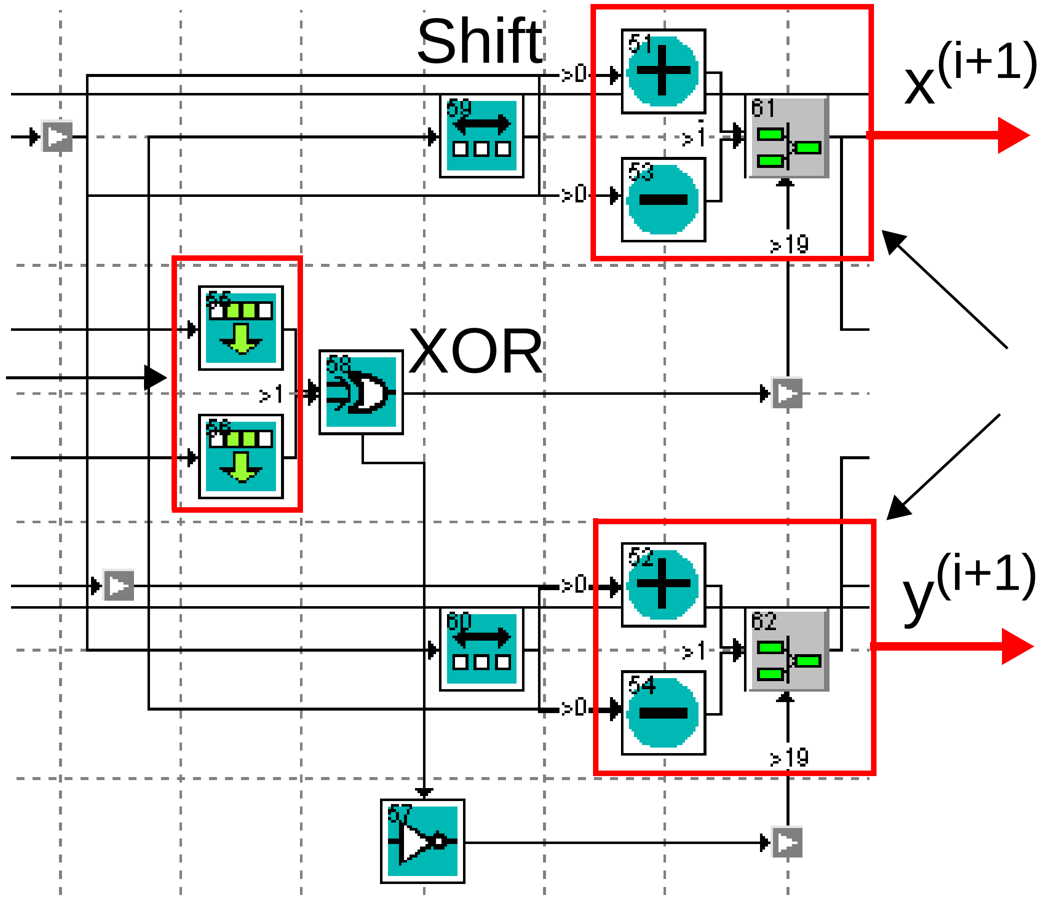
当使用向量模式时，$d_i = -sign(x^{(i)} y^{(i)})$。$x$ 方程用于计算向量的模值，并且 x 方程仅取决于 y 方程。因此，z 方程（角度累加器）可以被忽略。只余下：

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

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

所需硬件

- 实现 *单次迭代* 所需的硬件如下图所示：



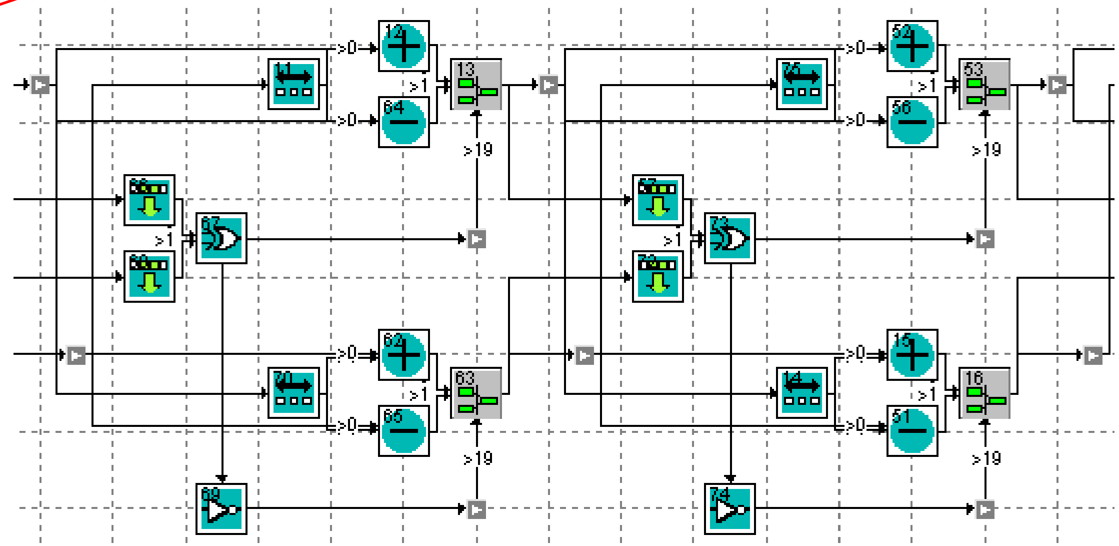
Notes:

还有一种可以不展开，直接就用一个元件电路实现（逻辑复用）

Top

实现 CORDIC 算法的一种方法是把它展开，并且每次迭代都在其自己的专用硬件中执行。下图所示为两次迭代。

逻辑复制



我们可以很容易地解释 XOR 门和 NOT 门的使用。判决算子根据 $d_i = -sign(x^{(i)}y^{(i)})$ 执行相应的操作。因此它有下列行为：

x	***y***	d_i
+ve	+ve	-ve
+ve	-ve	+ve
-ve	+ve	+ve
-ve	-ve	-ve

产生正确的 d_i 行为的基本要求是把 x 和 y 的最高有效位作为 XOR 门的输入。注意当 x 方程执行加法时，y 方程必定执行减法，反之亦然。因此，XOR 信号必须通过一个 NOT 门取反，从而达到正确控制加法器 / 减法器的目的。

多少个迭代？

- 一次单独的 CORDIC 迭代所需要的硬件是足够简单的。
- 然而，为了获得所希望的准度，需要回答以下两个问题：
  - 需要多少次迭代？
  - 需要多宽的数据路径？
- Yu Hen Hu（如注释所示）开发了一种算法，该算法可以解决 CORDIC 迭代中基于总量化误差 (OQE) 的问题。
- 一旦确定了 OQE，即可计算有效小数位的数目。

Top

Notes:

Y. H. Hu, “The Quantization Effects of the CORDIC Algorithm”, in IEEE Trans. On Signal Processing, Vol 40, No 4, April 1992

确定 OQE

- Yu Hen Hu 提出 OQE 由两种误差组成：
  - 近似误差 - CORDIC 旋转角度存在有限个基本角度量化所带来的量化误差。，我们所取的角度不断的逼近但还是有一个小小的没有真正接近的角度引起的，和迭代次数有关
  - 舍入误差 - 取决于实际实现中使用的有限精确度的代数运算。

计算的时候数据表示的位宽（用二进制表示）所引起的去查，由我们所取的值和小数的位数来决定的

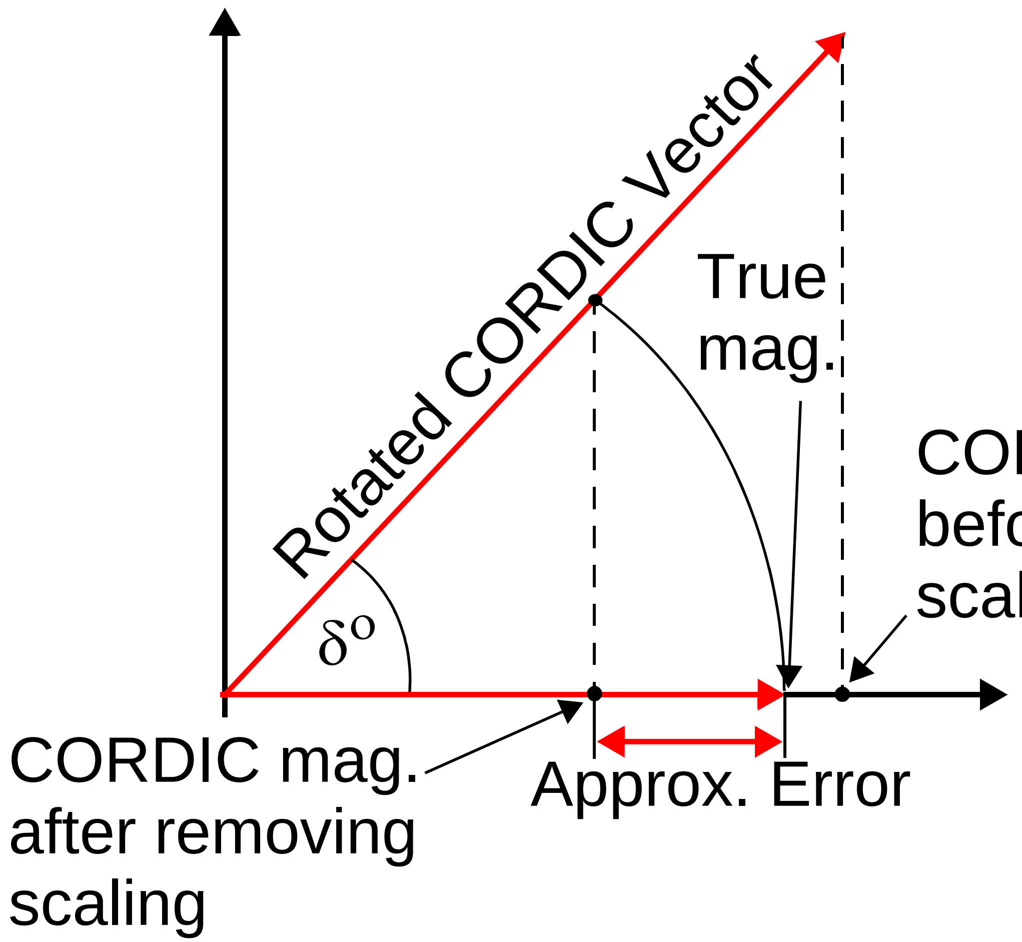
- 我们可根据下面的参数来定义上述两种误差：
  - 迭代次数 (n)。
  - 数据路径中的小数位的位数 (b)。
  - 最大向量的模值 ($|v(0)|$)。
- OQE 是上述误差的总和。

Notes:

Top

近似误差 (i)

- 使用向量模式时，我们的目标就是通过迭代使向量趋近 x 轴。
- 然而，有限次的旋转通常导致余下一个小角度 δ ，从而引起近似误差。



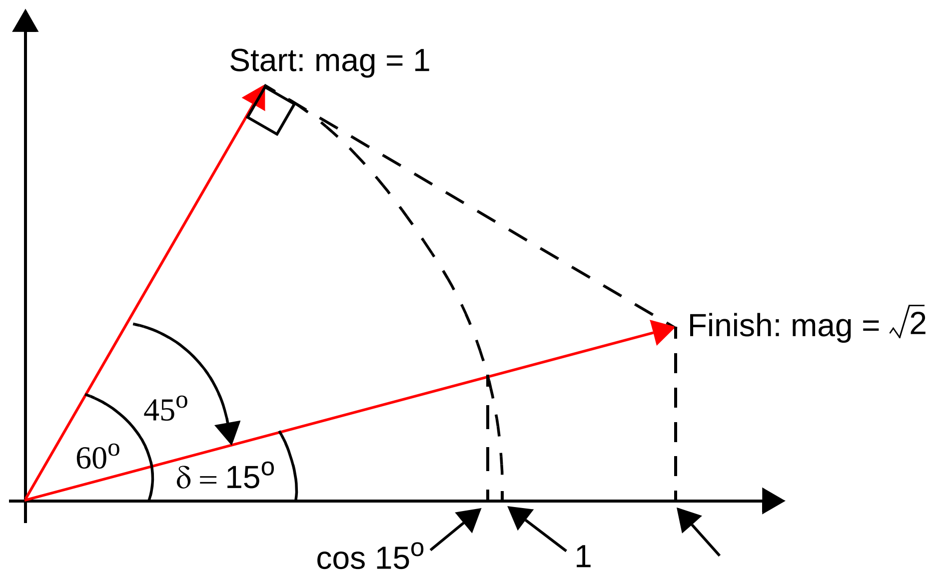
Top

Notes:

下图说明了只执行 1 次迭代的例子。模值为 1 的向量在开始时的初始角度 60^o。第一次迭代将向量旋转 45^o，从而导致了 $\delta = 15^o$ 的角度量化误差。对于一次迭代，其伸缩因子 K 等于：

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

因此，旋转向量的幅度现为 。x 方程给出的值为 $\sqrt{2}\cos 15^o$。用这个值除以伸缩因子 K，我们得到了真正的量化幅度 $\cos 15^o$。



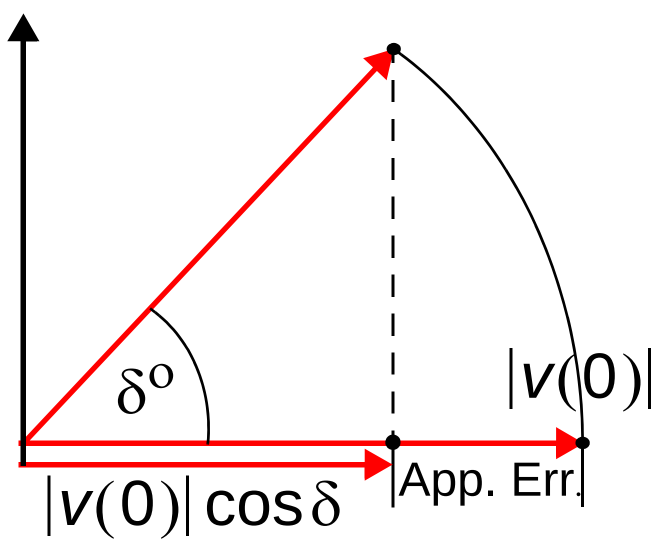
上图对应的 x 方程的输出如下所示。

$$x^{(i+1)} = (x^{(i)} - d_i(2^{-i}y^{(i)})) = \cos 60^o + \sin 60^o = \sqrt{2}\cos 15^o$$

Developed by: www.steepestascent.com

近似误差 (ii)

- 为了计算近似误差的上界，必须现找出 δ 的上界。
- Yu Hen Hu 提出：$\delta \le a(n-1) = \mathrm{atan}(2^{-n+1})$。
- 其中 $a(n-1)$ 为最终的旋转角度。 最后一次旋转（也就是迭代中离x轴最近的一次也是迭代中最后的一次角度）
- 观察下图，显然近似误差为：



如果角度为0的话，可以认为是没有近似误差的了

$$App.\ Err. = |v(0)| - |v(0)|\cos(\mathrm{atan}(2^{-n+1}))$$

Notes:

舍入误差

- Yu Hen Hu 推导出的舍入误差为：

$$Rounding\ Error = 2^{-b-0.5}\left[\frac{G(\mu,n)}{K_\mu(n)}+1\right]$$

- 其中，

$$G(\mu,n) = 1 + \sum_{j=1}^{n-1}\prod_{i=j}^{n-1} k_\mu(i)$$

$$K_\mu(n) = \prod_{i=0}^{n-1} k_\mu(i) = \prod_{i=0}^{n-1}\sqrt{1+\mu 2^{(-2i)}}$$

- 同样，b = 数据路径中的小数位个数，n = 迭代次数。

Top

Notes:

如需进一步了解舍入误差，请查看：

Y. H. Hu, “The Quantization Effects of the CORDIC Algorithm”, in IEEE Trans. On Signal Processing, Vol 40, No 4, April 1992

估算 d_{eff}

- 为了计算有效位的个数 d_{eff}, 必须首先计算 OQE。
- 前面已经提到了 OQE 的计算方法： 先估算deff，再计算得到b,n

,n

$$OQE = Approximation\ Error + Rounding\ Error$$

- 因此有效位的个数为：

$$d_{eff} = -(\log_2 OQE)$$

- 这种方法求得的 d_{eff} 值依赖于所选择的 b 和 n 的值。
- 然而，我们希望先指定 d_{eff}，然后求出 b 和 n 的值。
- 因此，Yu Hen Hu 采用的方法是通过取不同组的 b 和 n，将计算出的 d_{eff} 值编制成表。通过查表找到所需的 d_{eff}, 其对应的 b 和 n 即为可知。

Top

Notes:

使用下面的等式可求得有效小数位的个数：

$$d_{eff} = -(\log_2 OQE)$$

下表说明了一小部分 OQE 与 d_{eff} 值之间的联系。

OQE	d_{eff}
≤ 0.5	1
≤ 0.25	2
≤ 0.125	3
≤ 0.0625	4
≤ 0.03125	5
≤ 0.015625	6
.	.
.	.
.	.

并不是所有计算器都允许计算以 2 为底的对数运算。因此，可使用下面的运算来代替：

$$\log_2 OQE = \frac{\log_{10} OQE}{\log_{10} 2}$$

预测与仿真

- 使用 OQE 方程，我们可以计算出一个表，从而对一组 n 和 b，可以预测出 d_{eff}。
- 假如输入被限制在 ±0.5 的范围内，那么 $|v(0)| = \sqrt{0.5}$。
- 使用 $|v(0)|$ 的值，对 $3 \le n \le 9$ 和 $8 \le b \le 10$，我们计算出下列的表格。仿真值也同样在表格中列出。

	Predicted			Simulated		
n / b	**8**	**9**	**10**	**8**	**9**	**10**
3	5.09	5.31	5.43	5.32	5.71	5.80
4	6.03	6.59	6.98	6.22	6.88	7.34
5	6.28	7.13	7.88	6.52	7.56	8.27
6	6.21	7.17	8.10	6.55	7.11	8.12
7	6.06	7.05	8.04	6.42	7.29	8.29
8	5.92	6.91	7.91	6.33	7.29	8.31
9	5.78	6.78	7.78	6.00	7.00	8.00

Top

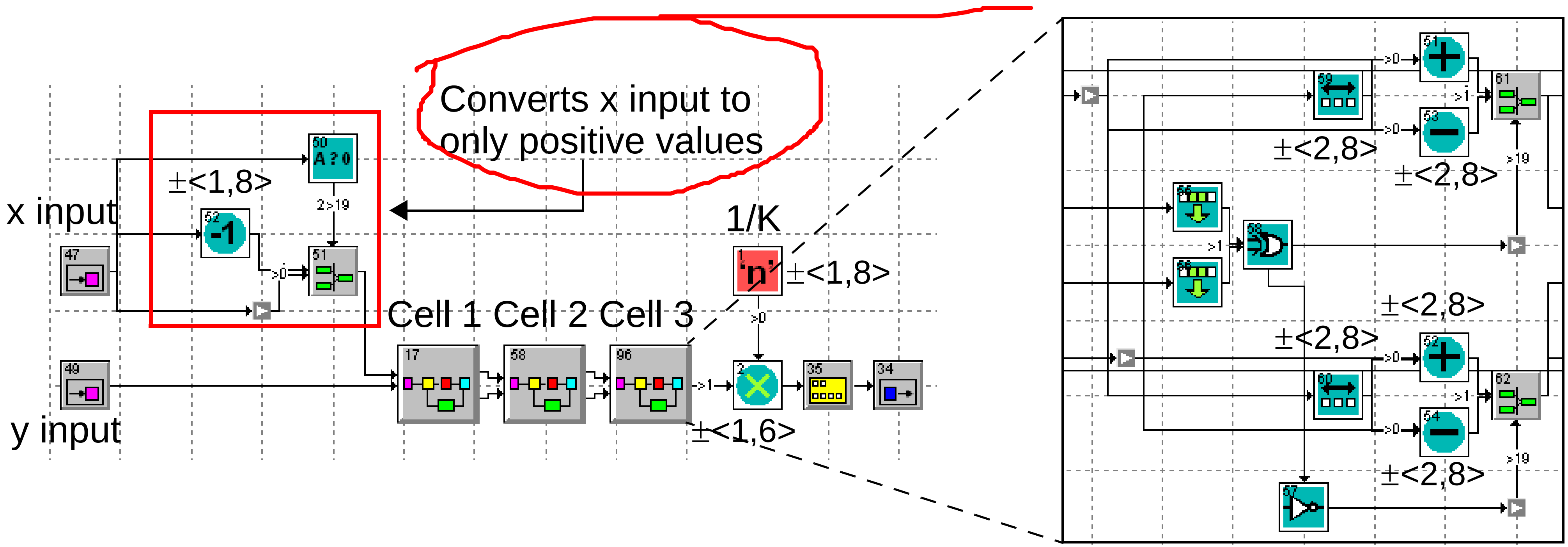
Notes:

给定一个 d_{eff}，d_{eff} 表可被用来找出 n 和 b 的值。比如说我们希望计算带有 6 个小数位精度的向量幅度。通过查下表，能够提供该精确度的最有效的结构是 $n = 4$, $b = 8$。当我们使用这组值设计 CORDIC 系统时，相对于浮点设计，最坏情况下的所产生的误差小于 2^{-6}。

	Predicted			Simulated		
n / b	**8**	**9**	**10**	**8**	**9**	**10**
3	5.09	5.31	5.43	5.32	5.71	5.80
4	**6.03**	6.59	6.98	**6.22**	6.88	7.34
5	6.28	7.13	7.88	6.52	7.56	8.27
6	6.21	7.17	8.10	6.55	7.11	8.12
7	6.06	7.05	8.04	6.42	7.29	8.29
8	5.92	6.91	7.91	6.33	7.29	8.31
9	5.78	6.78	7.78	6.00	7.00	8.00

CORDIC 整体设计

- 到目前为止，我们仅仅说明计算 1 次迭代所需的逻辑。
- 这里给出的完整设计的参数为 $n = 3$ 和 $b = 8$。



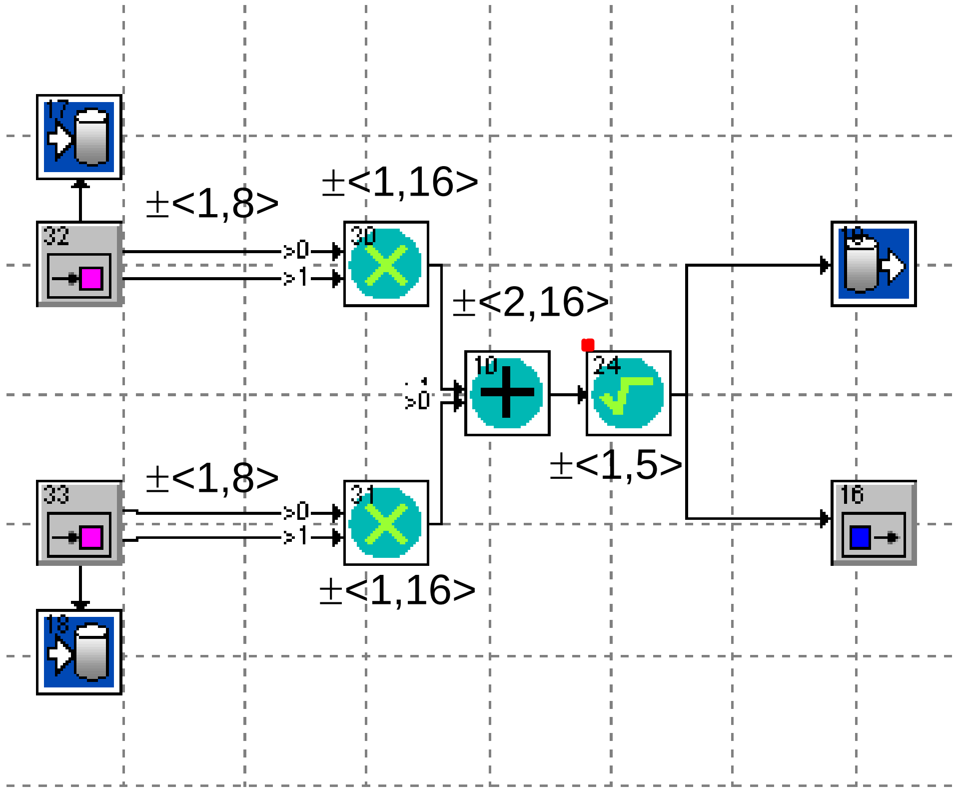
- 该设计具有 5 个有效小数位的准确度。注意乘法器的输出是 ±<1,6> 而不是 ±<1,5>。如果输出为 ±<1,5> 则有效小数位数目将下降到 4。

Notes:

Top

等效的直接设计

- 使用直接的方法所产生的等效的设计可被用于计算 $\sqrt{x^2+y^2}$ 。



- 问题是，两个设计哪个更高效？

Top

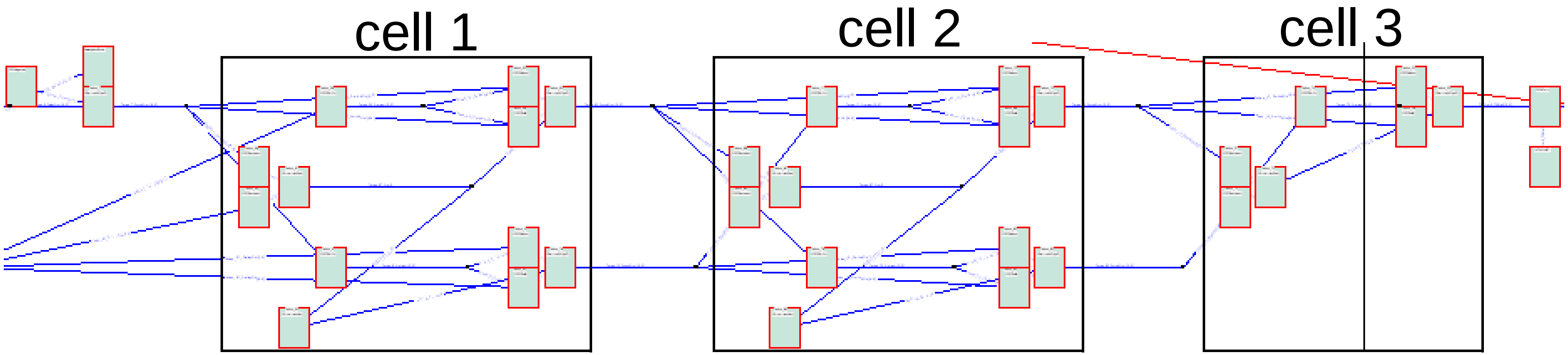
Notes:

注意，直接设计使用了乘法器和加法器的全精度。这意味着对于 ±<1,8> 输入数据，加法器的输出是完全正确的 - 即不存在量化误差。因为数据进入平方根单元是全精度的，这意味着，假如采用浮点解决方案并把浮点输出截断到相同的位数，那么该设计与浮点设计将具有完全相同的输出。故如果需要 5 个有效小数位，输出也只有 5 位。在 CORDIC 系统中，情况并不是这样，而是从 6 个小数位的输出中得到 5 个有效小数位。

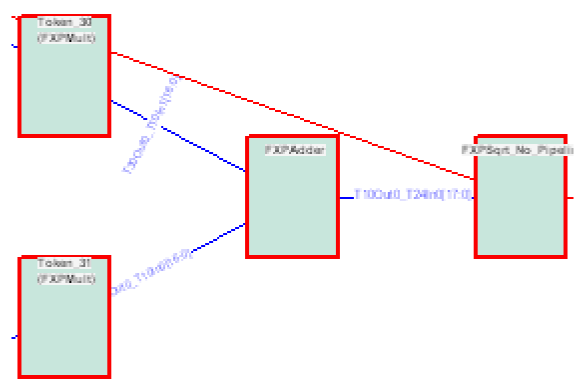
两个设计的比较 (i)

- 使用 Hardware Design Studio (HDS), 两个设计的 VHDL 将被自动生成。

- HDS CORDIC 原理图：



- HDS 直接方法的原理图：



Notes:

Top

使用 Hardware Design Studio (HDS), 两个设计的 VHDL 文件被自动生成。这些文件将被综合以比较设计的规格和传输量。原理图说明了使用端口与连接来组成设计的 VHDL 文件。

如何比较两个设计？(ii)

- 使用 Synplify Pro 将两种进行综合，目标器件是 Xilinx xc2v8000。综合结果如下：

	HDS CORDIC	HDS Direct	SysGen CORDIC
Slices	65	73	103
LUTs	99	125	172
18x18 Mults	0	2	0
Speed (MHz)	>45	>25	>30

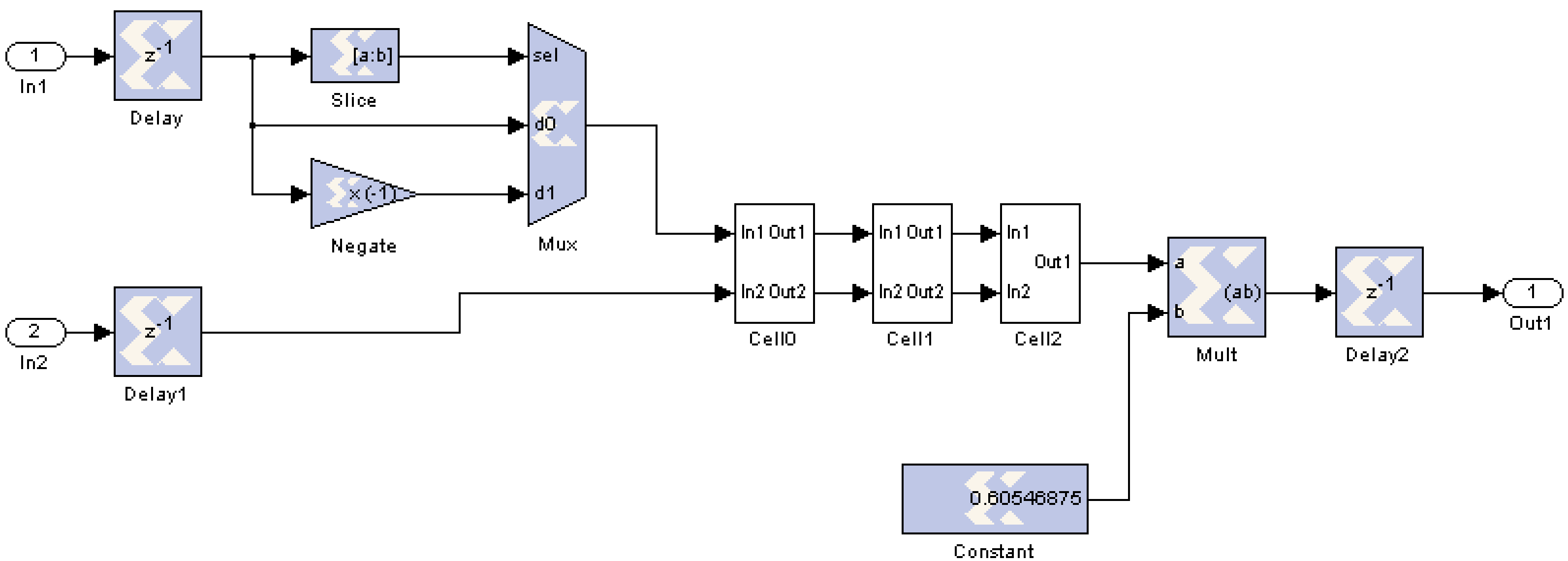
- 注意到同样可以使用 System Generator 创建一个等效的 CORDIC 设计，并使用 ISE/XST 对设计进行综合。结果同样列于上表。
- 这样的比较稍微有一点不公正，因为使用直接方法的平方根模块是其实际所需规模的两倍大。
- 对于平方根模块，即使 LUT 的数目被减半，直接方法所使用的总的逻辑电路仍然要多。

Top

Notes:

表面看来，CORDIC 系统似乎使用了比直接方法少得多的资源，并且具有更高的吞吐量。然而在直接方法中，LUT 主要是由平方根单元使用的，它使用的资源比其实际所需大了两倍。因此，在直接方法中，LUT 的使用数目实际上可以减半，来近似最小可能的设计。对于一个具有 63 个 LUT 的设计，它使用了 HDS CORDIC 系统中 LUT 的大约 60% ，它使用了 2 个 18x18 专用乘法器。如果乘法器使用 FPGA 上的逻辑电路来构成，那么直接方法所用的 LUT 的数目要比任何一个 CORDIC 系统大得多。

为了比较 HDS 和 System Generator, 我们使用 System Generator 创建了一个等效的 CORDIC 系统。该系统如下图所示：



通过观察综合结果我们发现 HDS 在实现 CORDIC 设计时效果更好。

总结

- 本章节介绍了 CORDIC 算法的理论。
- 我们使用 Givens 变换作为该算法的基础。
- 通过简化，我们将变换化简为一系列更小的伪旋转的连续迭代。
- 每个迭代都是由移位和加法组成。
- 作为伪旋转的结果，存在着一个负面效果，我们将其称之为伸缩因子。
- 操作模式和其它坐标系统的引入扩大了可计算函数的范围。
- 最后在 FPGA 上实现一个 CORDIC 内核。

Notes:

Top