Cordic

September 3, 2015

Author: Jose M. Gomez

1 Introduction

The present document describes an implementation made of the cordic algorithm using the myhdl-numeric. The implementation uses shift an add approach to reduce the resources required.

2 CORDIC theory

The CORDIC theory is described in the paper by Meher et al.:

```
In [1]: from IPython.display import Latex
        import sympy as sp

        def equation(name, var):
            return Latex(r'\begin{equation} ' + name +
                         ' = ' + sp.latex(var) + r'\end{equation}')
```

The CORDIC algorithm provides a method to calculate rotations:

```
In [2]: sp.var('theta')

        R = sp.Matrix([[sp.cos(theta), -sp.sin(theta)], [sp.sin(theta), sp.cos(theta)]])

        equation('R', R)
```

Out[2]:

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{1}$$

Scaling R by $\frac{1}{cos(\theta)}$, the new pseudo-rotation matrix (R_c) becomes:

```
In [3]: R_c = (((1/sp.cos(theta))*R).applyfunc(sp.trigsimp))

        equation('R_c', R_c)
```

Out[3]:

$$R_c = \begin{bmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{bmatrix} \tag{2}$$

If the rotations are applied iteratively, with a succesive approximation for the angle, it is possible to get the desired rotation. For this, the $\alpha(i)$ is defined as:

In [4]:
```python
sp.var('i')
sp.var('n')

alpha = sp.Function('alpha')
alpha(i)

alpha_ieq = sp.atan(2**(-i))

equation(r'\alpha \left(i\right)', alpha_ieq)
```

Out[4]:

$$\alpha\left(i\right) = \operatorname{atan}\left(2^{-i}\right) \tag{3}$$

With these angles it is possible to calculate the rotation (ρ) value:

In [5]:
```python
sigma = sp.Function('sigma')

sigma(i)

rho = sp.Function('rho')
rho(n)

rho_neq = sp.summation(sigma(i)*alpha_ieq, (i, 0, n-1))

equation(r'\rho\left(n\right)', rho_neq)
```

Out[5]:

$$\rho\left(n\right) = \sum_{i=0}^{n-1} \sigma(i) \operatorname{atan}\left(2^{-i}\right) \tag{4}$$

Where $\sigma\left(i\right)$ changes its value between +1 and -1.

To ensure the convergence, the input angle must be between the converge range. This can be calculated when $\sigma\left(i\right)$ is 1 and n reaches ∞. It yields:

In [6]:
```python
rho_infty = rho_neq.replace(sigma(i), 1).subs(n, sp.oo).doit()

equation(r'\rho_{ \infty } = ' + sp.latex(rho_infty), rho_infty.n())
```

Out[6]:

$$\rho_{\infty} = \sum_{i=0}^{\infty} \operatorname{atan}\left(2^{-i}\right) = 1.74328662047234 \tag{5}$$

As a result, CORDIC can only be used between the range $-\rho_{\infty} \leq \theta \leq \rho_{\infty}$. So inside the first and the fourth quadrants.

The intermediate angles can be calculated using the formula $\omega\left(i+1\right) = \omega\left(i\right) - \sigma\left(i\right) \cdot \alpha\left(i\right)$. Where $\sigma\left(i\right)$ will be 1 if $\omega\left(i\right) \geq 0$ and -1 otherwise. As a result, the rotation matrix is transformed into:

In [7]:
```python
K = sp.Function('K')
K(i)

R_i = (K(i) * sigma(i) * R_c.applyfunc(lambda x: x.subs(theta, alpha_ieq)) - \
    sp.eye(2) * (K(i)*(sigma(i) - 1))).applyfunc(lambda x: x.collect(K(i)))


equation(r'R_i', R_i)
```

Out[7]:

$$R_i = \left[\begin{matrix}K{\left (i \right)} & - 2^{- i} K{\left (i \right)} \sigma{\left (i \right)}\\2^{- i} K{\left (i \right)} \sigma{\left (i \right)} & K{\left (i \right)}\end{matrix}\right] \tag{6}$$

Where K_i is:

```
In [8]: K_ieq = sp.cos(alpha_ieq)

        equation(r'K\left(i\right)', K_ieq)
```

Out[8]:

$$K\left(i\right) = \frac{1}{\sqrt{1 + 2^{- 2 i}}} \tag{7}$$

Applying the Rotation matrix properly yields:

```
In [9]: sp.var('K')

        R_pieq = ((K*sp.Product((1/K(i)*R_i).applyfunc(lambda x: x.ratsimp()), (i, 0, n))))

        equation('R_{\pi}', R_pieq)
```

Out[9]:

$$R_{\pi} = K \prod_{i=0}^{n} \left[\begin{matrix}1 & - 2^{- i} \sigma{\left (i \right)}\\2^{- i} \sigma{\left (i \right)} & 1\end{matrix}\right] \tag{8}$$

Where K is the product of the different $K\left(i\right)$:

```
In [10]: K_eq = sp.Product(K(i), (i, 0, n))

         K_res = K_eq.replace(K(i), K_ieq)

         equation(r'K = ' + sp.latex(K_eq), K_res)
```

Out[10]:

$$K = \prod_{i=0}^{n} K{\left (i \right)} = \prod_{i=0}^{n} \frac{1}{\sqrt{1 + 2^{- 2 i}}} \tag{9}$$

The K value can be precalculated, for example for n equal to 15, giving the result:

```
In [11]: equation('K', K_res.subs(n, 15).doit().n())
```

Out[11]:

$$K = 0.607252935103139 \tag{10}$$

To be sure that the four quadrants are covered. A first rotation can be included, that moves them to the oposite one:

```
In [12]: R_m = sp.Matrix([[-1, 0], [0, -1]])

         equation('R_{-1}', R_m)
```

Out[12]:

$$R_{-1} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \tag{11}$$

This rotation is applied if the θ is greatter than $\frac{\pi}{2}$ or less than $-\frac{\pi}{2}$. Of course, the rotation angle has also to be modified, to take into account it:

In [13]:
```
omega = sp.Function('omega')

omega(i)

omega_0 = theta - sigma(-1)*sp.pi

equation('\omega_0', omega_0)
```

Out[13]:

$$\omega_0 = \theta - \pi\sigma(-1) \tag{12}$$

In this case, $\sigma(-1)$ will be 1 if θ is greatter than $\frac{\pi}{2}$ or -1 if it is less than $-\frac{\pi}{2}$. The rotation is not applied if θ is already inside the first or fourth quadrant.

3 CORDIC implementation

The implementation myhdl-numeric library. This library is based on the fixed point library for vhdl. To make the calculations, the number of bits required are:

- The number of input bits (assuming it is the same as the output ones)
- The number of bits necessary for the operations (log2(bits))
- The guard bits to ensure a proper behaviour with the rounding

Also, the angle will be in binary format, so π is 1 and $-\pi$ is -1. As a result, the wrapping comming from angles comes naturally with this representation.

In [14]:
```
import myhdl as hdl
import math as m
import numpy as np
from enum import IntEnum

class Modes(IntEnum):
    rotation = 0
    vectoring = 1

def cordic(x, y, angle, mode=Modes.rotation, bits=16):
    """ Function to calculate rotations using the CORDIC algorithm. The inputs x, y and angle
    are assumed to be in double format, with values between [-1, 1[. The parameter bits indica
    the number of bits of the inputs taken into account."""

    GUARD_BITS = hdl.fixmath().guard_bits
    OFFSET_BITS = GUARD_BITS + m.ceil(m.log(bits, 2))
    BITS = bits + OFFSET_BITS
    QUARTER = hdl.sfixba(0.5, 2, -1)
    indexes = range(0, BITS)
    arctantab = [hdl.sfixba(m.atan(2.**-idx)/m.pi, 1, -BITS) for idx in indexes]
    COSCALE = hdl.sfixba(float(np.prod(1./np.sqrt(1+np.power(2.,(-2*np.array(indexes)))))), 2,
```

```
pTx = hdl.sfixba(float(x), 2, 2-BITS)
pTy = hdl.sfixba(float(y), 2, 2-BITS)
pTheta = hdl.sfixba(float(angle), 1, -BITS, hdl.fixmath(overflow=hdl.fixmath.overflows.wrap

# Get angle between -1/2 and 1/2 angles

if mode == Modes.rotation:
    if pTheta < -QUARTER:
        pTx = hdl.sfixba(-pTx, pTx)
        pTy = hdl.sfixba(-pTy, pTy)

        pTheta += (QUARTER << 1)
    elif pTheta >= QUARTER:
        pTx = hdl.sfixba(-pTx, pTx)
        pTy = hdl.sfixba(-pTy, pTy)

        pTheta -= (QUARTER << 1)
else:
    if pTy >= 0 and pTx < 0:
        pTx = hdl.sfixba(-pTx, pTx)
        pTy = hdl.sfixba(-pTy, pTy)

        pTheta += (QUARTER << 1)
    elif pTy < 0 and pTx < 0:
        pTx = hdl.sfixba(-pTx, pTx)
        pTy = hdl.sfixba(-pTy, pTy)

        pTheta -= (QUARTER << 1)

for (pCounter, atanval) in zip(indexes, arctantab):
    if ((pTheta < 0) and (mode == Modes.rotation)) or ((pTy >= 0) and (mode == Modes.vecto
        xtemp = pTx + (pTy >> pCounter)
        pTy = hdl.sfixba(pTy - (pTx >> pCounter), pTy)
        pTx = hdl.sfixba(xtemp, pTx)

        pTheta += atanval
    else:
        xtemp = pTx - (pTy >> pCounter)
        pTy = hdl.sfixba(pTy + (pTx >> pCounter), pTy)
        pTx = hdl.sfixba(xtemp, pTx)

        pTheta -= atanval

pCos = hdl.sfixba(pTx * COSCALE, 1, -int(bits))
pSin = hdl.sfixba(pTy * COSCALE, 1, -int(bits))

return (pCos, pSin, pTheta)
```

4 Test

We just execute the cordic algorithm, and compare the results with the ones comming from the sin and cos math functions:

In [15]:
```python
bits = 16

error = 0
for x in np.arange(-1., 1., 0.125):
    angle = hdl.sfixba(x, 2, -bits)
    result = cordic(1.0, 0.0, angle, bits=bits)
    print("angle: ", float(m.pi*float(angle)))
    print([float(val) for val in result])
    cos_val = hdl.sfixba(m.cos(m.pi*float(angle)), 1, -bits)
    sin_val = hdl.sfixba(m.sin(m.pi*float(angle)), 1, -bits)
    print(float(cos_val), float(sin_val))

    error += abs(float(cos_val-result[0])) + abs(float(sin_val-result[1]))

print("Total error: %s [LSB]" % (error*(2.**bits),))
```

```
angle:  -3.141592653589793
[-1.0, 0.0, 0.0]
-1.0 -1.52587890625e-05
angle:  -2.748893571891069
[-0.9238739013671875, -0.3826904296875, 0.0]
-0.9238739013671875 -0.3826904296875
angle:  -2.356194490192345
[-0.7071075439453125, -0.7071075439453125, 0.0]
-0.7071075439453125 -0.7071075439453125
angle:  -1.9634954084936207
[-0.3826904296875, -0.9238739013671875, 0.0]
-0.3826904296875 -0.9238739013671875
angle:  -1.5707963267948966
[0.0, -1.0, 0.0]
0.0 -1.0
angle:  -1.1780972450961724
[0.3826904296875, -0.9238739013671875, 0.0]
0.3826904296875 -0.9238739013671875
angle:  -0.7853981633974483
[0.7071075439453125, -0.7071075439453125, 0.0]
0.7071075439453125 -0.7071075439453125
angle:  -0.39269908169872414
[0.9238739013671875, -0.3826904296875, 0.0]
0.9238739013671875 -0.3826904296875
angle:  0.0
[0.9999847412109375, 0.0, 0.0]
0.9999847412109375 0.0
angle:  0.39269908169872414
[0.9238739013671875, 0.3826904296875, 0.0]
0.9238739013671875 0.3826904296875
angle:  0.7853981633974483
[0.7071075439453125, 0.7071075439453125, 0.0]
0.7071075439453125 0.7071075439453125
angle:  1.1780972450961724
[0.3826904296875, 0.9238739013671875, 0.0]
0.3826904296875 0.9238739013671875
angle:  1.5707963267948966
[0.0, 0.9999847412109375, 0.0]
```

```
0.0 0.9999847412109375
angle:  1.9634954084936207
[-0.3826904296875, 0.9238739013671875, 0.0]
-0.3826904296875 0.9238739013671875
angle:  2.356194490192345
[-0.7071075439453125, 0.7071075439453125, 0.0]
-0.7071075439453125 0.7071075439453125
angle:  2.748893571891069
[-0.9238739013671875, 0.3826904296875, 0.0]
-0.9238739013671875 0.3826904296875
Total error: 1.0 [LSB]
```

The error is 1LSB which comes from the $-\pi$ case.

An RTL version can be found in test_cordic_mod.py.

5 Conclusions

A CORDIC implementation has been made. The test shows an equivalent behaviour to the system trigonometric functions. The program uses the sfixba type which provides fixed point arithmetic behavior for python and myhdl.