

# CORDIC Fixed Point Simulation

---

Copyright (c) 2012 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to [youngwlim@hotmail.com](mailto:youngwlim@hotmail.com).

This document was produced by using OpenOffice and Octave.

---

Based on the following site:

[drdobbs.com](http://drdobbs.com)

“Implementing CORDIC Algorithms”, P. Jarvis, Dr Dobb's, Oct, 1990

ANSI-C version of the above by P. Knoppers.

# Circular

```
void Circular (long x, long y, long z)
{
  int i;
  X = x;
  Y = y;
  Z = z;
  for (i = 0; i <= fractionBits; ++i)
  {
    x = X >> i;
    y = Y >> i;
    z = atan[i];
    X -= Delta (y, Z);
    Y += Delta (x, Z);
    Z -= Delta (z, Z);
  }
}
```

$$x \Rightarrow x'_{i+1} = (x'_i - y'_i \sigma_i 2^{-i})$$

$$y \Rightarrow y'_{i+1} = (x'_i \sigma_i 2^{-i} + y'_i)$$

$$z \Rightarrow \alpha_{i+1} = \alpha_i - \tan(\sigma_i 2^{-i})$$

$$x = X \gg i; \quad \Rightarrow \quad X \cdot 2^{-i}$$

$$y = Y \gg i; \quad \Rightarrow \quad Y \cdot 2^{-i}$$

$z = \text{atan}[i];$

$$X -= \text{Delta}(y, Z); \quad \Rightarrow \quad X = (X - Y \sigma_i 2^{-i})$$

$$Y += \text{Delta}(x, Z); \quad \Rightarrow \quad Y = (X \sigma_i 2^{-i} + Y)$$

$Z -= \text{Delta}(z, Z);$

# Delta

```
#define Delta(n, Z) (Z >= 0) ? (n) : -(n)
```

$$x \rightarrow x'_{i+1} = (x'_i - y'_i \sigma_i 2^{-i})$$

$$y \rightarrow y'_{i+1} = (x'_i \sigma_i 2^{-i} + y'_i)$$

$$z \rightarrow \alpha_{i+1} = \alpha_i - \tan(\sigma_i 2^{-i})$$

```
X -= Delta (y, Z);      (Z >= 0) X = X - y : X = X + y;
```

```
Y += Delta (x, Z);      (Z >= 0) Y = Y + x : Y = Y - x;
```

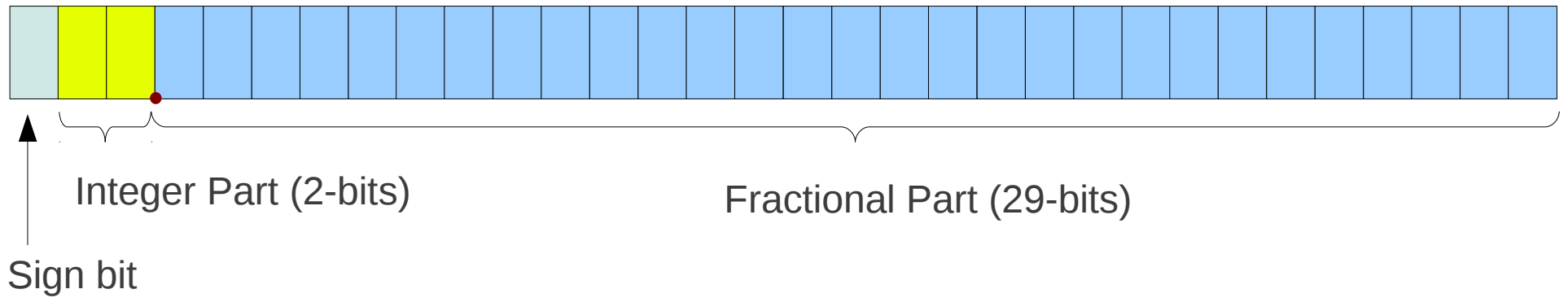
```
Z -= Delta (z, Z);      (Z >= 0) Z = Z - z : Z = Z + z;
```

$$y \rightarrow -y'_i \sigma_i 2^{-i}$$

$$x \rightarrow +x'_i \sigma_i 2^{-i}$$

$$z \rightarrow -\tan(\sigma_i 2^{-i})$$

# Fixed Point Format



# Computing ATAN constants instead of LUT (1)

Use power series to calculate the incremental angles

$$\alpha_i = \tan^{-1} 2^{-i}$$

$$\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} \dots \quad \text{for } x^2 \leq 1$$

$$\tan^{-1} x = x \quad \text{32-bit precision}$$

$$x^3/3 = 2^{-32} \quad x = \sqrt[3]{6 \cdot 2^{-11}}$$

$$\tan^{-1} 2^{-i} = 2^{-i} \quad \text{for } i \geq 11$$

$$2^{-in} / n = 2^{-i} \quad \text{for } 1 \leq i \leq 10$$

$$\frac{\pi}{4} \quad \text{for } i = 0$$

# Computing ATAN constants instead of LUT (2)

Use power series to calculate the incremental angles

$$\alpha_i = \tan^{-1} 2^{-i}$$

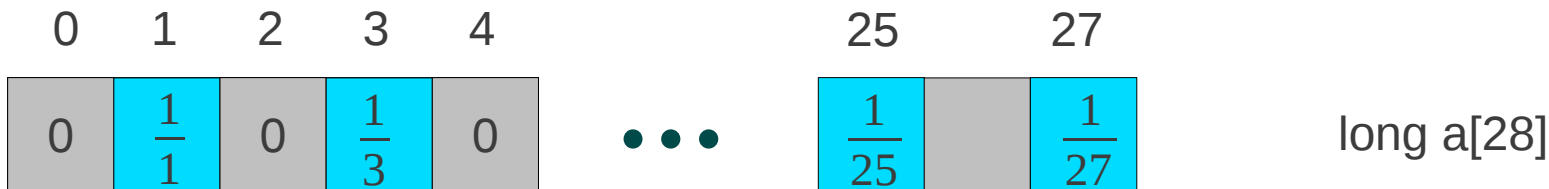
$$\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} \dots \quad \text{for } x^2 \leq 1$$

To compute the coefficients  $\frac{1}{k}$  (  $k = 3, 5, 7, \dots, 27$  )

## Reciprocal (k, n)

**k**: integer to be inversed,

**n**: precision for the desired fractional part



Restoring Division Alg



# Computing ATAN constants instead of LUT (3)

Use power series to calculate the incremental angles

$$\alpha_i = \tan^{-1} 2^{-i}$$

$$\tan^{-1} x = \frac{\pi}{2} - \left( x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} \dots \right) \quad \text{for } x^2 \geq 1$$

atan[fractionBits + 1]

## Poly2 (k, n)

Computes the power series for the specified number of terms for the specified power of two using **Horner's rule**

$$a_n x_n + a_{n-1} x_{n-1} + a_{n-2} x_{n-2} + \dots + a_1 x_1 + a_0$$

$$(\dots(((a_n x + a_{n-1}) x) + a_{n-2} x) + \dots + a_1) x + a_0$$

Restoring Division Alg

# Declarations

```
#define    fractionBits    29
#define    longBits       32
#define    One             (0100000000000L>>1)
#define    HalfPi         (014441766521L>>1)

long X0C, X0H, X0R;    /* seed for circular, hyperbolic, and square root */
long OneOverE, E;      /* the base of natural logarithms */
long HalfLnX0R;        /* constant used in simultaneous sqrt, ln computation */

static unsigned terms[11] = {0, 27, 14, 9, 7, 5, 4, 4, 3, 3, 3};
static long a[28];
static long atan[fractionBits + 1];
static long atanh[fractionBits + 1];

static long X;
static long Y;
static long Z;
```

$$2^{-in} / n = 2^{-i} \text{ for } 1 \leq i \leq 10$$

```
i= 1 → n= 27
i= 2 → n= 14
i= 3 → n=  9
i= 4 → n=  7
i= 5 → n=  5
i= 6 → n=  4
i= 7 → n=  4
i= 8 → n=  3
i= 9 → n=  3
i=10 → n=  3
```

# Preparing atan[30] array

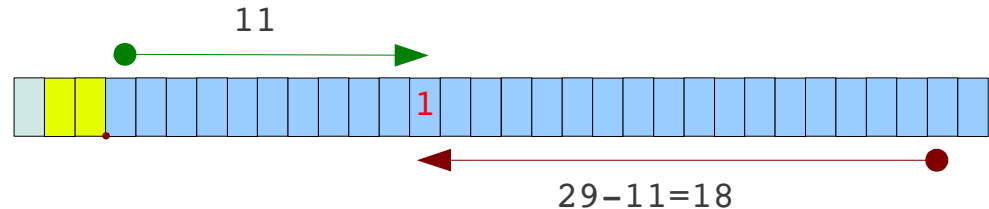
```
for (i = 0; i <= 13; ++i) {
    a[2*i] = 0;
    a[2*i+1] = Reciprocal (2*i+1, fractionBits);
}
atan[0] = HalfPi / 2;    /* atan(2^0)= pi / 4 */
for (i = 1; i <= 7; ++i)
    a[4*i-1] = -a[4*i-1];
for (i = 1; i <= 10; ++i)
    atan[i] = Poly2 (-i, terms[i]);
for (i = 11; i <= fractionBits; ++i)
    atan[i] = atanh[i] = 1L << (fractionBits - i);

printf ("\n\natan(2^-n)\n");
for (i = 0; i <= 10; ++i) {
    printf ("%2d ", i);
    WriteVarious (atan[i]);
}

r = 0;
for (i = 0; i <= fractionBits; ++i)
    r += atan[i];
printf ("radius of convergence");
WriteFraction (r);
```

$$\left\{ 0, \frac{1}{1}, 0, \frac{1}{3}, 0, \frac{1}{5}, \dots, 0, \frac{1}{27} \right\}$$

$$\frac{-1}{3}, \frac{-1}{7}, \frac{-1}{11}, \frac{-1}{15}, \frac{-1}{19}, \frac{-1}{23}, \frac{-1}{27}$$



# Making Power Series – Poly2( )

```
static unsigned terms[11] =  
    {0, 27, 14, 9, 7, 5, 4, 4, 3, 3, 3};
```

```
for (i = 1; i <= 10; ++i)  
    atan[i] = Poly2 (-i, terms[i]);
```

```
long Poly2 (int log, unsigned n)  
{  
    long r = 0;  
    int i;  
    for (i = n; i >= 0; --i)  
        r = (log < 0 ? r >> -log : r << log) + a[i];  
    return (r);  
}
```

```
atan[ 1] = Poly2 ( -1, 27);
```

```
atan[ 2] = Poly2 ( -2, 14);
```

```
atan[ 3] = Poly2 ( -3,  9);
```

```
atan[ 4] = Poly2 ( -4,  7);
```

```
atan[ 5] = Poly2 ( -5,  5);
```

```
atan[ 6] = Poly2 ( -6,  4);
```

```
atan[ 7] = Poly2 ( -7,  4);
```

```
atan[ 8] = Poly2 ( -8,  3);
```

```
atan[ 9] = Poly2 ( -9,  3);
```

```
atan[10] = Poly2 (-10,  3);
```

# WriteFraction(n)

```
void WriteFraction (long n)
{
    unsigned short i;
    unsigned short low;
    unsigned short digit;
    unsigned long k;
    putchar (n < 0 ? '-' : ' ');
    n = abs (n);
    putchar ((n >> fractionBits) + '0');
    putchar ('.');
    low = k = n << (longBits - fractionBits);
    /* align octal point at left */
    k >>= 4;
    /* shift to make room for a decimal digit */
    for (i = 1; i <= 8; ++i)
    {
        digit = (k *= 10L) >> (longBits - 4);
        low = (low & 0xf) * 10;
        k += ((unsigned long) (low >> 4)) -
            ((unsigned long) digit << (longBits - 4));
        putchar (digit + '0');
    }
}
```

$32 - 29 = 3$   
 $k = \text{frac part} * 2^3$

$k = \text{only frac part}$

$32 - 4 = 28$

# Computing ATAN constants instead of LUT

---

# Computing ATAN constants instead of LUT

---

# Computing ATAN constants instead of LUT

---



## References

- [1] <http://en.wikipedia.org/>
- [2] "Implementing CORDIC Algorithms", P. Jarvis, Dr Dobb's
- [3] ANSI-C version of [2] by P. Knoppers.