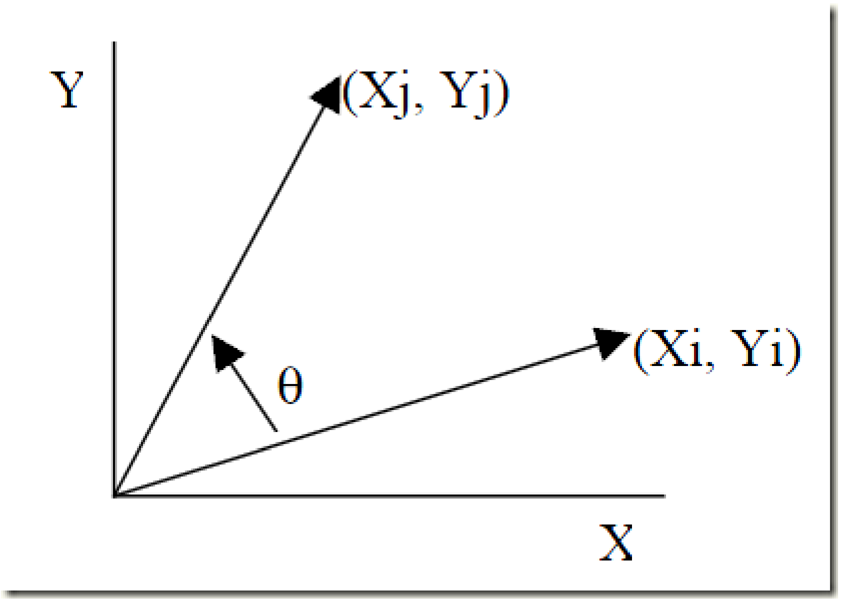
cordic算法的verilog实现及modelsim仿真

cnblogs.com/aikimi7/p/3929592.html

1. 算法介绍

CORDIC（Coordinate Rotation Digital Computer）算法即坐标旋转数字计算方法，是J.D.Volder1于1959年首次提出，主要用于三角函数、双曲线、指数、对数的计算。该算法通过基本的**加和移位运算代替乘法运算**，使得矢量的旋转和定向的计算不再需要三角函数、乘法、开方、反三角、指数等函数，计算向量长度并能把直角坐标系转换为极坐标系。**因为Cordic 算法只用了移位和加法**，很容易用纯硬件来实现，非常适合FPGA实现。

CORDIC算法完成坐标或向量的平面旋转（下图以逆时针旋转为例）。



旋转后，可得如下向量：

$$\begin{bmatrix} X_j \\ Y_j \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \end{bmatrix} \tag{1}$$

旋转的角度θ经过多次旋转得到的（步步逼近，接近二分查找法），每次旋转一小角度。单步旋转定义如下公式：

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \begin{bmatrix} \cos\theta_n & -\sin\theta_n \\ \sin\theta_n & \cos\theta_n \end{bmatrix} \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{2}$$

公式（2）提取cosθ，可修改为：

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \cos\theta_n \begin{bmatrix} 1 & -\tan\theta_n \\ \tan\theta_n & 1 \end{bmatrix} \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{3}$$

修改后的公式把乘法次数从4次改为3次，剩下的乘法运算可以通过选择每次旋转的角度去除，将每一步的正切值选为2的指数（二分查找法），除以2的指数可以通过右移操作完成（verilog）。

每次旋转的角度可以表示为：

$$\theta_n = \arctan\left(\frac{1}{2^n}\right) \tag{4}$$

所有迭代角度累加值等于最终需要的旋转角度θ：

$$\sum_{n=0}^{\infty} S_n \theta_n = \theta \tag{5}$$

这里Sn为1或者-1，根据旋转方向确定（后面有确定方法，公式（15））,顺时针为-1，逆时针为1。

$$S_n = \{-1;+1\} \tag{6}$$

可以得到如下公式：

$$\tan\theta_n = S_n 2^{-n} \tag{7}$$

结合公式（3）和（7），得到公式（8）：

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \cos\theta_n \begin{bmatrix} 1 & -S_n 2^{-n} \\ S_n 2^{-n} & 1 \end{bmatrix} \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{8}$$

到这里，除了余弦值这个系数，算法只要通过简单的移位和加法操作完成。而这个系数可以通过预先计算最终值消掉。首先重新重写这个系数如下：

$$\cos\theta_n = \cos\left(\arctan\left(\frac{1}{2^n}\right)\right) \tag{9}$$

第二步计算所有的余弦值并相乘，这个值K称为增益系数。

$$K = \frac{1}{P} = \prod_{n=0}^{\infty} \cos\left(\arctan\left(\frac{1}{2^n}\right)\right) \approx 0.607253 \tag{10}$$

由于K值是常量，我们可以先忽略它。

$$\begin{cases} X_j = K(X_i\cos\theta - Y_i\sin\theta) \\ Y_j = K(Y_i\cos\theta + X_i\sin\theta) \end{cases} \tag{11}$$

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & -S_n 2^{-n} \\ S_n 2^{-n} & 1 \end{bmatrix} \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{12}$$

or as

$$\begin{cases} X_{n+1} = X_n - S_n 2^{-2n} Y_n \\ Y_{n+1} = Y_n + S_n 2^{-2n} X_n \end{cases} \tag{13}$$

到这里我们发现，算法只剩下移位和加减法，这就非常适合硬件实现了，为硬件快速计算三角函数提供了一种新的算法。在进行迭代运算时，需要引入一个新的变量Z，表示需要旋转的角度θ中还没有旋转的角度。

$$Z_{n+1} = \theta - \sum_{i=0}^{n} \theta_i \tag{14}$$

这里，我们可以把前面提到确定旋转方向的方法介绍了，就是通过这个变量Z的符号确定。

$$S_n = \begin{cases} -1 & if\ Z_n < 0 \\ +1 & if\ Z_n \geq 0 \end{cases} \tag{15}$$

通过公式（5）和（15），将未旋转的角度变为0。

$$Z_{n+1} = Z_n - S_n\, arctan(2^{-n}) \tag{16}$$

一个类编程风格的结构如下，反正切值是预先计算好的。

```
For n=0 to [inf]
    If (Z(n) >= 0) then
        X(n + 1) := X(n) – (Yn/2^n);
        Y(n + 1) := Y(n) + (Xn/2^n);
        Z(n + 1) := Z(n) – atan(1/2^n);
    Else
        X(n + 1) := X(n) + (Yn/2^n);
        Y(n + 1) := Y(n) – (Xn/2^n);
        Z(n + 1) := Z(n) + atan(1/2^n);
    End if;
End for;
```

1.1 旋转模式

旋转模式下，CORDIC算法驱动Z变为0，结合公式（13）和（16），算法的核心计算如下：

$$[X_j, Y_j, Z_j] = [P(X_i \cos(Z_i) - Y_i \sin(Z_i)),\ P(Y_i \cos(Z_i) + X_i \sin(Z_i)),\ 0]$$

一种特殊情况是，另初始值如下：

$$X_i = \frac{1}{P} = K \approx 0.60725$$

$$Y_i = 0$$

$$Z_i = \theta$$

$$[X_j, Y_j, Z_j] = [\cos\theta,\ \sin\theta,\ 0]$$

因此，旋转模式下CORDIC算法可以计算一个输入角度的正弦值和余弦值。

1.2 向量模式

向量模式下，有两种特例：

1) $X_i = X$

$Y_i = Y$

$Z_i = 0$

$$[X_j, Y_j, Z_j] = \left[P\sqrt{X_i^2 + Y_i^2},\ 0,\ \arctan\left(\frac{Y_i}{X_i}\right)\right]$$

2) $X_i = 1$

$Y_i = a$

$Z_i = 0$

$$[X_j, Y_j, Z_j] = \left[P\sqrt{1 + a^2},\ 0,\ \arctan(a)\right]$$

因此，向量模式下CORDIC算法可以用来计算输入向量的模和反正切，也能开方计算，并可以将直角坐标转换为极坐标。

算法介绍：http://en.wikipedia.org/wiki/Cordic ，http://blog.csdn.net/liyuanbhu/article/details/8458769

2. matlab实现

根据算法原理，利用维基百科中给的程序，在matlab中跑了一遍，对算法有了一定程度的了解。

程序如下：

```
function v = cordic(beta,n)
% This function computes v = [cos(beta), sin(beta)] (beta in radians)
% using n iterations. Increasing n will increase the precision.

if beta < -pi/2 || beta > pi/2
    if beta < 0
        v = cordic(beta + pi, n);
    else
        v = cordic(beta - pi, n);
    end
    v = -v; % flip the sign for second or third quadrant
%     return
end

% Initialization of tables of constants used by CORDIC
% need a table of arctangents of negative powers of two, in radians:
% angles = atan(2.^-(0:27));
angles =  [  ...
    0.78539816339745   0.46364760900081   0.24497866312686   0.12435499454676 ...
    0.06241880999596   0.03123983343027   0.01562372862048   0.00781234106010 ...
    0.00390623013197   0.00195312251648   0.00097656218956   0.00048828121119 ...
    0.00024414062015   0.00012207031189   0.00006103515617   0.00003051757812 ...
    0.00001525878906   0.00000762939453   0.00000381469727   0.00000190734863 ...
    0.00000095367432   0.00000047683716   0.00000023841858   0.00000011920929 ...
    0.00000005960464   0.00000002980232   0.00000001490116   0.00000000745058 ];
% and a table of products of reciprocal lengths of vectors [1, 2^-2j]:
Kvalues = [ ...
    0.70710678118655   0.63245553203368   0.61357199107790   0.60883391251775 ...
    0.60764825625617   0.60735177014130   0.60727764409353   0.60725911229889 ...
    0.60725447933256   0.60725332108988   0.60725303152913   0.60725295913894 ...
    0.60725294104140   0.60725293651701   0.60725293538591   0.60725293510314 ...
    0.60725293503245   0.60725293501477   0.60725293501035   0.60725293500925 ...
    0.60725293500897   0.60725293500890   0.60725293500889   0.60725293500888 ];
Kn = Kvalues(min(n, length(Kvalues)));

% Initialize loop variables:
v = [1;0]; % start with 2-vector cosine and sine of zero
poweroftwo = 1;
angle = angles(1);

% Iterations
for j = 0:n-1;
    if beta < 0
        sigma = -1;
    else
        sigma = 1;
```

```
    end
    factor = sigma * poweroftwo;
    R = [1, -factor; factor, 1];
    v = R * v; % 2-by-2 matrix multiply
    beta = beta - sigma * angle; % update the remaining angle
    poweroftwo = poweroftwo / 2;
    % update the angle from table, or eventually by just dividing by
    % two,(a=arctan(a),a is small enough)
    if j+2 > length(angles)
        angle = angle / 2;
    else
        angle = angles(j+2);
    end
end

% Adjust length of output vector to be [cos(beta), sin(beta)]:
v = v * Kn;
return
end
```

3. 硬件实现

实现主要参考了相关作者的代码，然后对其进行了修改，最终实现了16级的流水线，设计完成旋转模式下正弦值和余弦值的计算。

http://www.cnblogs.com/qiweiwang/archive/2010/07/28/1787021.html，http://www.amobbs.com/forum.php?mod=viewthread&tid=5513050&highlight=cordic

下面分段介绍下各部分代码：

首先是角度的表示，进行了宏定义，360读用16位二进制表示2^16，每一度为2^16/360。

```
//360°--2^16,phase_in = 16bits (input [15:0] phase_in)
//1°--2^16/360
`define rot0  16'h2000    //45
`define rot1  16'h12e4    //26.5651
`define rot2  16'h09fb    //14.0362
`define rot3  16'h0511    //7.1250
`define rot4  16'h028b    //3.5763
`define rot5  16'h0145    //1.7899
`define rot6  16'h00a3    //0.8952
`define rot7  16'h0051    //0.4476
`define rot8  16'h0028    //0.2238
`define rot9  16'h0014    //0.1119
`define rot10 16'h000a    //0.0560
`define rot11 16'h0005    //0.0280
`define rot12 16'h0003    //0.0140
`define rot13 16'h0001    //0.0070
`define rot14 16'h0001    //0.0035
`define rot15 16'h0000    //0.0018
```

然后是流水线级数定义、增益放大倍数以及中间结果位宽定义。流水线级数16，为了满足精度要求，有文献指出流水线级数必须大于等于角度位宽16（针对正弦余弦计算的CORDIC算法优化及其FPGA实现）。增益放大2^16，为了避免溢出状况中间结果（x，y，z）定义为17为，最高位作为符号位判断，1为负数，0为正数。

```
parameter PIPELINE = 16;
//parameter K = 16'h4dba;//k=0.607253*2^15
parameter K = 16'h9b74;//gian k=0.607253*2^16,9b74,n means the number pipeline
//pipeline 16-level    //maybe overflow,matlab result not overflow
//MSB is signed bit,transform the sin and cos according to phase_in[15:14]
reg [16:0] x0=0,y0=0,z0=0;
reg [16:0] x1=0,y1=0,z1=0;
reg [16:0] x2=0,y2=0,z2=0;
reg [16:0] x3=0,y3=0,z3=0;
reg [16:0] x4=0,y4=0,z4=0;
reg [16:0] x5=0,y5=0,z5=0;
reg [16:0] x6=0,y6=0,z6=0;
reg [16:0] x7=0,y7=0,z7=0;
reg [16:0] x8=0,y8=0,z8=0;
reg [16:0] x9=0,y9=0,z9=0;
reg [16:0] x10=0,y10=0,z10=0;
reg [16:0] x11=0,y11=0,z11=0;
reg [16:0] x12=0,y12=0,z12=0;
reg [16:0] x13=0,y13=0,z13=0;
reg [16:0] x14=0,y14=0,z14=0;
reg [16:0] x15=0,y15=0,z15=0;
reg [16:0] x16=0,y16=0,z16=0;
```

还需要定义memory型寄存器数组并初始化为0，用于寄存输入角度高2位的值。

```
reg [1:0] quadrant [PIPELINE:0];
integer i;
initial
begin
    for(i=0;i<=PIPELINE;i=i+1)
    quadrant[i] = 2'b0;
end
```

接着，是对输入角度象限处理，将角度都转换到第一象限，方便处理。输入角度值最高两位赋值0，即转移到第一象限[0°,90°]。此外，完成x0，y0和z0的初始化，并增加一位符号位。

```
//phase_in[15:14] determines which quadrant the angle is.
//00 means first;01 means second;00 means third;00 means fourth
//initialization: x0 = K,y0 = 0,z0 = phase_in,then the last result(x16,y16) = (cos(phase_in),sin(phase_in))
always @ (posedge clk)//stage 0,not pipeline
begin
    x0 <= {1'b0,K}; //add one signed bit,0 means positive
    y0 <= 17'b0;
    z0 <= {3'b0,phase_in[13:0]};//control the phase_in to the range[0-Pi/2]
end
```

接下来根据剩余待旋转角度z的符号位进行16次迭代处理，即完成16级流水线处理。迭代公式：x(n+1） <= x(n) + {{n{y(n)[16]}},y(n)[16:n]}，n为移位个数。右移之后高位补位，这里补位有一些不理解。移位可能存在负数，且没有四舍五入。按理说第一象限不存在负数，但后续仿真汇总确实有负数出现，但仿真结果良好。

```
always @ (posedge clk)//stage 1
begin
  if(z0[16])//the diff is negative so clockwise
  begin
      x1 <= x0 + y0;
      y1 <= x0 - y0;
      z1 <= z0 + `rot0;
  end
  else
  begin
      x1 <= x0 - y0;//x1 <= x0;
      y1 <= x0 + y0;//y1 <= x0;
      z1 <= z0 - `rot0;//reversal 45
  end
end

always @ (posedge clk)//stage 2
begin
    if(z1[16])//the diff is negative so clockwise
    begin
        x2 <= x1 + {y1[16],y1[16:1]};
        y2 <= y1 - {x1[16],x1[16:1]};
        z2 <= z1 + `rot1;//clockwise 26
    end
    else
    begin
        x2 <= x1 - {y1[16],y1[16:1]};
        y2 <= y1 + {x1[16],x1[16:1]};
        z2 <= z1 - `rot1;//anti-clockwise 26
    end
end

always @ (posedge clk)//stage 3
begin
    if(z2[16])//the diff is negative so clockwise
    begin
        x3 <= x2 + {{2{y2[16]}},y2[16:2]}; //right shift n bits,divide 2^n
        y3 <= y2 - {{2{x2[16]}},x2[16:2]}; //left adds n bits of MSB,in first quadrant x or y are positive,MSB =0 ? ?
        z3 <= z2 + `rot2;//clockwise 14    //difference of positive and negtive number and no round(4,5)
    end
    else
    begin
        x3 <= x2 - {{2{y2[16]}},y2[16:2]};
        y3 <= y2 + {{2{x2[16]}},x2[16:2]};
        z3 <= z2 - `rot2;//anti-clockwise 14
    end
end

always @ (posedge clk)//stage 4
begin
    if(z3[16])
    begin
        x4 <= x3 + {{3{y3[16]}},y3[16:3]};
        y4 <= y3 - {{3{x3[16]}},x3[16:3]};
        z4 <= z3 + `rot3;//clockwise 7
    end
    else
```

```
    begin
        x4 <= x3 - {{3{y3[16]}},y3[16:3]};
        y4 <= y3 + {{3{x3[16]}},x3[16:3]};
        z4 <= z3 - `rot3;//anti-clockwise 7
    end
end

always @ (posedge clk)//stage 5
begin
    if(z4[16])
    begin
        x5 <= x4 + {{4{y4[16]}},y4[16:4]};
        y5 <= y4 - {{4{x4[16]}},x4[16:4]};
        z5 <= z4 + `rot4;//clockwise 3
    end
    else
    begin
        x5 <= x4 - {{4{y4[16]}},y4[16:4]};
        y5 <= y4 + {{4{x4[16]}},x4[16:4]};
        z5 <= z4 - `rot4;//anti-clockwise 3
    end
end

always @ (posedge clk)//STAGE 6
begin
    if(z5[16])
    begin
        x6 <= x5 + {{5{y5[16]}},y5[16:5]};
        y6 <= y5 - {{5{x5[16]}},x5[16:5]};
        z6 <= z5 + `rot5;//clockwise 1
    end
    else
    begin
        x6 <= x5 - {{5{y5[16]}},y5[16:5]};
        y6 <= y5 + {{5{x5[16]}},x5[16:5]};
        z6 <= z5 - `rot5;//anti-clockwise 1
    end
end

always @ (posedge clk)//stage 7
begin
    if(z6[16])
    begin
        x7 <= x6 + {{6{y6[16]}},y6[16:6]};
        y7 <= y6 - {{6{x6[16]}},x6[16:6]};
        z7 <= z6 + `rot6;
    end
    else
    begin
        x7 <= x6 - {{6{y6[16]}},y6[16:6]};
        y7 <= y6 + {{6{x6[16]}},x6[16:6]};
        z7 <= z6 - `rot6;
    end
end
```

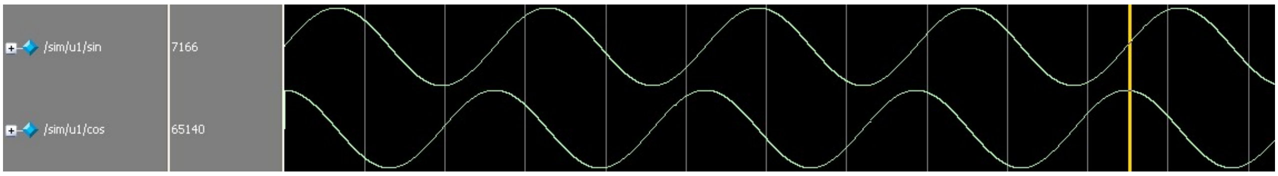
由于进行了象限的转换，最终流水结果需要根据象限进行转换为正确的值。这里寄存17次高2位角度输入值，配合流水线结果用于象限判断，并完成转换。

```
//according to the pipeline,register phase_in[15:14]
always @ (posedge clk)
begin
  quadrant[0] <= phase_in[15:14];
  quadrant[1] <= quadrant[0];
  quadrant[2] <= quadrant[1];
  quadrant[3] <= quadrant[2];
  quadrant[4] <= quadrant[3];
  quadrant[5] <= quadrant[4];
  quadrant[6] <= quadrant[5];
  quadrant[7] <= quadrant[6];
  quadrant[8] <= quadrant[7];
  quadrant[9] <= quadrant[8];
  quadrant[10] <= quadrant[9];
  quadrant[11] <= quadrant[10];
  quadrant[12] <= quadrant[11];
  quadrant[13] <= quadrant[12];
  quadrant[14] <= quadrant[13];
  quadrant[15] <= quadrant[14];
  quadrant[16] <= quadrant[15];
end
```

最后，根据寄存的高2位角度输入值，利用三角函数关系，得出最后的结果，其中负数进行了补码操作。

```
//alter register, according to quadrant[16] to transform the result to the right result
always @ (posedge clk) begin
eps <= z15;
case(quadrant[16]) //or 15
2'b00:begin //if the phase is in first quadrant,the sin(X)=sin(A),cos(X)=cos(A)
        cos <= x16;
        sin <= y16;
        end
2'b01:begin //if the phase is in second quadrant,the sin(X)=sin(A+90)=cosA,cos(X)=cos(A+90)=-sinA
        cos <= ~(y16) + 1'b1;//-sin
        sin <= x16;//cos
        end
2'b10:begin //if the phase is in third quadrant,the sin(X)=sin(A+180)=-sinA,cos(X)=cos(A+180)=-cosA
        cos <= ~(x16) + 1'b1;//-cos
        sin <= ~(y16) + 1'b1;//-sin
        end
2'b11:begin //if the phase is in forth quadrant,the sin(X)=sin(A+270)=-cosA,cos(X)=cos(A+270)=sinA
        cos <= y16;//sin
        sin <= ~(x16) + 1'b1;//-cos
        end
endcase
end
```

4. Modelsim仿真结果



仿真结果应该还是挺理想的。后续需要完成的工作：1.上述红色出现的问题的解决；2.应用cordic算法，完成如FFT的算法。

后记：

在3中，迭代公式：x(n+1） <= x(n) + {{n{y(n)[16]}},y(n)[16:n]}，上述右移操作都是手动完成：首先最高位增加1位符号位（1为负，0为正），然后手动添加n位符号位（最高位）补齐，即实际上需要完成的算术右移（>>>）。本设计定义的reg为无符号型，在定义时手动添加最高位为符号位。verilog-1995中只有integer为有符号型，reg和wire都是无符号型，只能手动添加扩展位实现有符号运算。而在verilog-2001中reg和wire可以通过保留字signed定义为有符号型。另外，涉及有符号和无符号型的移位操作等可参考下面的文章。

verilog有符号数详解：http://www.cnblogs.com/LJWJL/p/3481995.html，

Verilog-2001新特性及代码实现：http://www.asic-world.com/verilog/verilog2k.html，

逻辑移位与算术移位区别：
http://www.cnblogs.com/yuphone/archive/2010/09/21/1832217.html，
http://blog.sina.com.cn/s/blog_65311d330100ij9n.html

原来的算法实现针对Verilog-1995中reg和wire没有有符号型，也没有verilog-2001中的算术移位而实现的。根据verilog-2001新特性，引入有符号型reg和算术右移，同样实现了前文的结果。代码如下：

```
`timescale 1 ns/100 ps
//360?°--2^16,phase_in = 16bits (input [15:0] phase_in)
//1?°--2^16/360
`define rot0  16'h2000    //45
`define rot1  16'h12e4    //26.5651
`define rot2  16'h09fb    //14.0362
`define rot3  16'h0511    //7.1250
`define rot4  16'h028b    //3.5763
`define rot5  16'h0145    //1.7899
`define rot6  16'h00a3    //0.8952
`define rot7  16'h0051    //0.4476
`define rot8  16'h0028    //0.2238
`define rot9  16'h0014    //0.1119
`define rot10 16'h000a    //0.0560
`define rot11 16'h0005    //0.0280
`define rot12 16'h0003    //0.0140
`define rot13 16'h0001    //0.0070
`define rot14 16'h0001    //0.0035
`define rot15 16'h0000    //0.0018

module cordic(
output reg signed [16:0] sin,cos,eps,
input [15:0] phase_in,
input clk
);
parameter PIPELINE = 16;
//parameter K = 16'h4dba;//k=0.607253*2^15
parameter K = 17'h09b74;//gian k=0.607253*2^16,9b74,
//pipeline 16-level    //maybe overflow,matlab result not overflow
//MSB is signed bit,transform the sin and cos according to phase_in[15:14]
reg signed [16:0] x0=0,y0=0,z0=0;
reg signed [16:0] x1=0,y1=0,z1=0;
reg signed [16:0] x2=0,y2=0,z2=0;
reg signed [16:0] x3=0,y3=0,z3=0;
reg signed [16:0] x4=0,y4=0,z4=0;
reg signed [16:0] x5=0,y5=0,z5=0;
reg signed [16:0] x6=0,y6=0,z6=0;
reg signed [16:0] x7=0,y7=0,z7=0;
reg signed [16:0] x8=0,y8=0,z8=0;
reg signed [16:0] x9=0,y9=0,z9=0;
reg signed [16:0] x10=0,y10=0,z10=0;
reg signed [16:0] x11=0,y11=0,z11=0;
reg signed [16:0] x12=0,y12=0,z12=0;
reg signed [16:0] x13=0,y13=0,z13=0;
reg signed [16:0] x14=0,y14=0,z14=0;
reg signed [16:0] x15=0,y15=0,z15=0;
reg signed [16:0] x16=0,y16=0,z16=0;

reg [1:0] quadrant [PIPELINE:0];
integer i;
initial
begin
    for(i=0;i<=PIPELINE;i=i+1)
    quadrant[i] = 2'b0;
end

//phase_in[15:14] determines which quadrant the angle is.
//00 means first;01 means second;00 means third;00 means fourth
//initialization: x0 = K,y0 = 0,z0 = phase_in,then the last result(x16,y16) =
```

```
(cos(phase_in),sin(phase_in))
always @ (posedge clk)//stage 0,not pipeline
begin
    x0 <= K; //add one signed bit,0 means positive
    y0 <= 17'd0;
    z0 <= {3'b0,phase_in[13:0]};//control the phase_in to the range[0-Pi/2]
end
//pipeline
//z0[16] = 0,positive
always @ (posedge clk)//stage 1
begin
  if(z0[16])//the diff is negative so clockwise
  begin
      x1 <= x0 + y0;
      y1 <= y0 - x0;
      z1 <= z0 + `rot0;
  end
  else
  begin
      x1 <= x0 - y0;//x1 <= x0;
      y1 <= y0 + x0;//y1 <= x0;
      z1 <= z0 - `rot0;//reversal 45
  end
end

always @ (posedge clk)//stage 2
begin
    if(z1[16])//the diff is negative so clockwise
    begin
        x2 <= x1 + (y1 >>> 1);
        y2 <= y1 - (x1 >>> 1);
        z2 <= z1 + `rot1;//clockwise 26
    end
    else
    begin
        x2 <= x1 - (y1 >>> 1);
        y2 <= y1 + (x1 >>> 1);
        z2 <= z1 - `rot1;//anti-clockwise 26
    end
end

always @ (posedge clk)//stage 3
begin
    if(z2[16])//the diff is negative so clockwise
    begin
        x3 <= x2 + (y2 >>> 2); //right shift n bits,divide 2^n,signed extension,Arithmetic shift right
        y3 <= y2 - (x2 >>> 2); //left adds n bits of MSB,in first quadrant x or y are positive,MSB =0 ? ?
        z3 <= z2 + `rot2;//clockwise 14    //difference of positive and negtive number and no round(4,5)
    end
    else
    begin
        x3 <= x2 - (y2 >>> 2); //Arithmetic shift right
        y3 <= y2 + (x2 >>> 2);
        z3 <= z2 - `rot2;//anti-clockwise 14
    end
end
```

```

always @ (posedge clk)//stage 4
begin
    if(z3[16])
    begin
        x4 <= x3 + (y3 >>> 3);
        y4 <= y3 - (x3 >>> 3);
        z4 <= z3 + `rot3;//clockwise 7
    end
    else
    begin
        x4 <= x3 - (y3 >>> 3);
        y4 <= y3 + (x3 >>> 3);
        z4 <= z3 - `rot3;//anti-clockwise 7
    end
end

always @ (posedge clk)//stage 5
begin
    if(z4[16])
    begin
        x5 <= x4 + (y4 >>> 4);
        y5 <= y4 - (x4 >>> 4);
        z5 <= z4 + `rot4;//clockwise 3
    end
    else
    begin
        x5 <= x4 - (y4 >>> 4);
        y5 <= y4 + (x4 >>> 4);
        z5 <= z4 - `rot4;//anti-clockwise 3
    end
end

always @ (posedge clk)//STAGE 6
begin
    if(z5[16])
    begin
        x6 <= x5 + (y5 >>> 5);
        y6 <= y5 - (x5 >>> 5);
        z6 <= z5 + `rot5;//clockwise 1
    end
    else
    begin
        x6 <= x5 - (y5 >>> 5);
        y6 <= y5 + (x5 >>> 5);
        z6 <= z5 - `rot5;//anti-clockwise 1
    end
end

always @ (posedge clk)//stage 7
begin
    if(z6[16])
    begin
        x7 <= x6 + (y6 >>> 6);
        y7 <= y6 - (x6 >>> 6);
        z7 <= z6 + `rot6;
    end
    else
    begin
```

```
        x7 <= x6 - (y6 >>> 6);
        y7 <= y6 + (x6 >>> 6);
        z7 <= z6 - `rot6;
    end
end

always @ (posedge clk)//stage 8
begin
    if(z7[16])
    begin
        x8 <= x7 + (y7 >>> 7);
        y8 <= y7 - (x7 >>> 7);
        z8 <= z7 + `rot7;
    end
    else
    begin
        x8 <= x7 - (y7 >>> 7);
        y8 <= y7 + (x7 >>> 7);
        z8 <= z7 - `rot7;
    end
end

always @ (posedge clk)//stage 9
begin
    if(z8[16])
    begin
        x9 <= x8 + (y8 >>> 8);
        y9 <= y8 - (x8 >>> 8);
        z9 <= z8 + `rot8;
    end
    else
    begin
        x9 <= x8 - (y8 >>> 8);
        y9 <= y8 + (x8 >>> 8);
        z9 <= z8 - `rot8;
    end
end

always @ (posedge clk)//stage 10
begin
    if(z9[16])
    begin
        x10 <= x9 + (y9 >>> 9);
        y10 <= y9 - (x9 >>> 9);
        z10 <= z9 + `rot9;
    end
    else
    begin
        x10 <= x9 - (y9 >>> 9);
        y10 <= y9 + (x9 >>> 9);
        z10 <= z9 - `rot9;
    end
end

always @ (posedge clk)//stage 11
begin
    if(z10[16])
    begin
        x11 <= x10 + (y10 >>> 10);
```

```
        y11 <= y10 - (x10 >>> 10);
        z11 <= z10 + `rot10;//clockwise 3
    end
    else
    begin
        x11 <= x10 - (y10 >>> 10);
        y11 <= y10 + (x10 >>> 10);
        z11 <= z10 - `rot10;//anti-clockwise 3
    end
end

always @ (posedge clk)//STAGE 12
begin
    if(z11[16])
    begin
        x12 <= x11 + (y11 >>> 11);
        y12 <= y11 - (x11 >>> 11);
        z12 <= z11 + `rot11;//clockwise 1
    end
    else
    begin
        x12 <= x11 - (y11 >>> 11);
        y12 <= y11 + (x11 >>> 11);
        z12 <= z11 - `rot11;//anti-clockwise 1
    end
end

always @ (posedge clk)//stage 13
begin
    if(z12[16])
    begin
        x13 <= x12 + (y12 >>> 12);
        y13 <= y12 - (x12 >>> 12);
        z13 <= z12 + `rot12;
    end
    else
    begin
        x13 <= x12 - (y12 >>> 12);
        y13 <= y12 + (x12 >>> 12);
        z13 <= z12 - `rot12;
    end
end

always @ (posedge clk)//stage 14
begin
    if(z13[16])
    begin
        x14 <= x13 + (y13 >>> 13);
        y14 <= y13 - (x13 >>> 13);
        z14 <= z13 + `rot13;
    end
    else
    begin
        x14 <= x13 - (y13 >>> 13);
        y14 <= y13 + (x13 >>> 13);
        z14 <= z13 - `rot13;
    end
end

```

```
always @ (posedge clk)//stage 15
begin
    if(z14[16])
    begin
        x15 <= x14 + (y14 >>> 14);
        y15 <= y14 - (x14 >>> 14);
        z15 <= z14 + `rot14;
    end
    else
    begin
        x15 <= x14 - (y14 >>> 14);
        y15 <= y14 + (x14 >>> 14);
        z15 <= z14 - `rot14;
    end
end

always @ (posedge clk)//stage 16
begin
    if(z15[16])
    begin
        x16 <= x15 + (y15 >>> 15);
        y16 <= y15 - (x15 >>> 15);
        z16 <= z15 + `rot15;
    end
    else
    begin
        x16 <= x15 - (y15 >>> 15);
        y16 <= y15 + (x15 >>> 15);
        z16 <= z15 - `rot15;
    end
end
//according to the pipeline,register phase_in[15:14]
always @ (posedge clk)
begin
  quadrant[0] <= phase_in[15:14];
  quadrant[1] <= quadrant[0];
  quadrant[2] <= quadrant[1];
  quadrant[3] <= quadrant[2];
  quadrant[4] <= quadrant[3];
  quadrant[5] <= quadrant[4];
  quadrant[6] <= quadrant[5];
  quadrant[7] <= quadrant[6];
  quadrant[8] <= quadrant[7];
  quadrant[9] <= quadrant[8];
  quadrant[10] <= quadrant[9];
  quadrant[11] <= quadrant[10];
  quadrant[12] <= quadrant[11];
  quadrant[13] <= quadrant[12];
  quadrant[14] <= quadrant[13];
  quadrant[15] <= quadrant[14];
  quadrant[16] <= quadrant[15];
end
//alter register, according to quadrant[16] to transform the result to the right result
always @ (posedge clk) begin
eps <= z15;
case(quadrant[16]) //or 15
2'b00:begin //if the phase is in first quadrant,the sin(X)=sin(A),cos(X)=cos(A)
```

```
        cos <= x16;
        sin <= y16;
        end
2'b01:begin //if the phase is in second quadrant,the sin(X)=sin(A+90)=cosA,cos(X)=cos(A+90)=-sinA
        cos <= ~(y16) + 1'b1;//-sin
        sin <= x16;//cos
        end
2'b10:begin //if the phase is in third quadrant,the sin(X)=sin(A+180)=-sinA,cos(X)=cos(A+180)=-cosA
        cos <= ~(x16) + 1'b1;//-cos
        sin <= ~(y16) + 1'b1;//-sin
        end
2'b11:begin //if the phase is in forth quadrant,the sin(X)=sin(A+270)=-cosA,cos(X)=cos(A+270)=sinA
        cos <= y16;//sin
        sin <= ~(x16) + 1'b1;//-cos
        end
endcase
end

endmodule
```

另外，代码中可以适当优化下：1.流水线操作时，定义的中间寄存器在定义是可以选择memory型，且可以单独建立module或者task进行封装迭代过程； 2. 最后对高2位角度寄存时，可以利用for语句选择移位寄存器实现，如下所示。

```
always @ (posedge clk,negedge rst_n)
begin
    if(!rst_n)
        for(i=0;i<=PIPELINE;i=i+1)
            quadrant[i]<=2'b00;
    else
        if(ena)
            begin
                for(i=0;i<PIPELINE;i=i+1)
                    quadrant[i+1]<=quadrant[i];
                quadrant[0]<=phase_in[7:6];
            end
end
```

疑问：有一点比较奇怪的是，转移到第一象限后，x和y不该存在负数的情况，但是现在确实有，这一点比较费解，所以将算术右移改为逻辑右移，在函数极值时存在错误。