

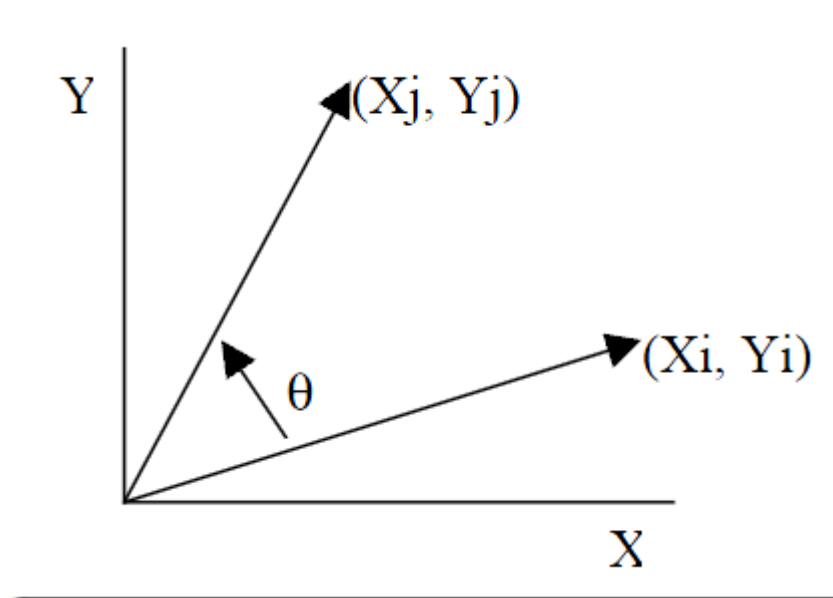
cordic算法的verilog实现及modelsim仿真

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.Volder 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：

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

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：

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

公式 (2) 提取 $\cos \theta$ ，可修改为：

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

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：

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

所有迭代角度累加值等于最终需要的旋转角度 θ ：

$$\sum_{n=0}^{\infty} S_n \theta_n = \theta \quad (5)$$

这里 S_n 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。

$$S_n = \{-1; +1\} \quad (6)$$

可以得到如下公式：

$$\tan \theta_n = S_n 2^{-n} \quad (7)$$

结合公式（3）和（7），得到公式（8）：

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

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：

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

第二步计算所有的余弦值并相乘，这个值K称为增益系数。

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

由于K值是常量，我们可以先忽略它。

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

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

or as

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

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。

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

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。

$$S_n = \begin{cases} -1 & \text{if } Z_n < 0 \\ +1 & \text{if } Z_n \geq 0 \end{cases} \quad (15)$$

通过公式（5）和（15），将未旋转的角度变为0。

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

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：

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。

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

$$Y_i = 0$$

$$Z_i = \theta$$

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

1.2 向量模式

向量模式下，有两种特例：

$$\begin{aligned}
 1) \quad & X_i = X \\
 & Y_i = Y \\
 & Z_i = 0 \\
 & [X_j, Y_j, Z_j] = \left[P\sqrt{X_i^2 + Y_i^2}, 0, \arctan\left(\frac{Y_i}{X_i}\right) \right] \\
 \\
 2) \quad & X_i = 1 \\
 & Y_i = a \\
 & Z_i = 0 \\
 & [X_j, Y_j, Z_j] = \left[P\sqrt{1 + a^2}, 0, \arctan(a) \right]
 \end{aligned}$$

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：



```

1 function v = cordic(beta,n)
2 % This function computes v = [cos(beta), sin(beta)] (beta in radians)
3 % using n iterations. Increasing n will increase the precision.
4
5 if beta < -pi/2 || beta > pi/2
6     if beta < 0
7         v = cordic(beta + pi, n);
8     else
9         v = cordic(beta - pi, n);
10    end
11    v = -v; % flip the sign for second or third quadrant
12 %    return
13 end
14
15 % Initialization of tables of constants used by CORDIC
16 % need a table of arctangents of negative powers of two, in radians:
17 % angles = atan(2.^-(0:27));
18 angles = [ ...
19     0.78539816339745    0.46364760900081    0.24497866312686    0.12435499454676
20     ...
21     0.06241880999596    0.03123983343027    0.01562372862048    0.00781234106010
22     ...
23     0.00390623013197    0.00195312251648    0.00097656218956    0.00048828121119
24     ...
25     0.00024414062015    0.00012207031189    0.00006103515617    0.00003051757812
26     ...
27     0.00001525878906    0.00000762939453    0.00000381469727    0.00000190734863
28     ...
29     0.00000095367432    0.00000047683716    0.00000023841858    0.00000011920929
30     ...
31     0.00000005960464    0.00000002980232    0.00000001490116    0.00000000745058
32 ];
33 % and a table of products of reciprocal lengths of vectors [1, 2^-2j]:
34 Kvalues = [ ...
35     0.70710678118655    0.63245553203368    0.61357199107790    0.60883391251775
36     ...
37     0.60764825625617    0.60735177014130    0.60727764409353    0.60725911229889
38     ...
39     0.60725447933256    0.60725332108988    0.60725303152913    0.60725295913894
40     ...
41     0.60725294104140    0.60725293651701    0.60725293538591    0.60725293510314
42     ...
43     0.60725293503245    0.60725293501477    0.60725293501035    0.60725293500925
44     ...
45     0.60725293500897    0.60725293500890    0.60725293500889    0.60725293500888
46 ];
47 Kn = Kvalues(min(n, length(Kvalues)));
48
49 % Initialize loop variables:
50 v = [1;0]; % start with 2-vector cosine and sine of zero
51 poweroftwo = 1;
52 angle = angles(1);
53
54 % Iterations
55 for j = 0:n-1;
56     if beta < 0
57         sigma = -1;
58     else
59         sigma = 1;
60     end
61     v = v + sigma * poweroftwo * [sin(angle); cos(angle)];
62     poweroftwo = poweroftwo / 2;
63     angle = angles(2);
64 end
65 v = v / Kn;
66 end

```

```

47     end
48     factor = sigma * poweroftwo;
49     R = [1, -factor; factor, 1];
50     v = R * v; % 2-by-2 matrix multiply
51     beta = beta - sigma * angle; % update the remaining angle
52     poweroftwo = poweroftwo / 2;
53     % update the angle from table, or eventually by just dividing by
54     % two, (a=arctan(a), a is small enough)
55     if j+2 > length(angles)
56         angle = angle / 2;
57     else
58         angle = angles(j+2);
59     end
60 end
61
62 % Adjust length of output vector to be [cos(beta), sin(beta)]:
63 v = v * Kn;
64 return
65 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度用16位二进制表示 2^{16} ，每一度为 $2^{16}/360$ 。



```

1 //360°--2^16, phase_in = 16bits (input [15:0] phase_in)
2 //1°--2^16/360
3 `define rot0 16'h2000 //45
4 `define rot1 16'h12e4 //26.5651
5 `define rot2 16'h09fb //14.0362
6 `define rot3 16'h0511 //7.1250
7 `define rot4 16'h028b //3.5763
8 `define rot5 16'h0145 //1.7899
9 `define rot6 16'h00a3 //0.8952
10 `define rot7 16'h0051 //0.4476
11 `define rot8 16'h0028 //0.2238
12 `define rot9 16'h0014 //0.1119
13 `define rot10 16'h000a //0.0560
14 `define rot11 16'h0005 //0.0280
15 `define rot12 16'h0003 //0.0140
16 `define rot13 16'h0001 //0.0070
17 `define rot14 16'h0001 //0.0035
18 `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。



```
1 parameter PIPELINE = 16;
2 //parameter K = 16'h4dba;//k=0.607253*2^15
3 parameter K = 16'h9b74;//gain k=0.607253*2^16,9b74,n means the number pipeline
4 //pipeline 16-level //maybe overflow,matlab result not overflow
5 //MSB is signed bit,transform the sin and cos according to phase_in[15:14]
6 reg [16:0] x0=0,y0=0,z0=0;
7 reg [16:0] x1=0,y1=0,z1=0;
8 reg [16:0] x2=0,y2=0,z2=0;
9 reg [16:0] x3=0,y3=0,z3=0;
10 reg [16:0] x4=0,y4=0,z4=0;
11 reg [16:0] x5=0,y5=0,z5=0;
12 reg [16:0] x6=0,y6=0,z6=0;
13 reg [16:0] x7=0,y7=0,z7=0;
14 reg [16:0] x8=0,y8=0,z8=0;
15 reg [16:0] x9=0,y9=0,z9=0;
16 reg [16:0] x10=0,y10=0,z10=0;
17 reg [16:0] x11=0,y11=0,z11=0;
18 reg [16:0] x12=0,y12=0,z12=0;
19 reg [16:0] x13=0,y13=0,z13=0;
20 reg [16:0] x14=0,y14=0,z14=0;
21 reg [16:0] x15=0,y15=0,z15=0;
22 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。



```
1 reg [1:0] quadrant [PIPELINE:0];
2 integer i;
3 initial
4 begin
5     for(i=0;i<=PIPELINE;i=i+1)
6         quadrant[i] = 2'b0;
7 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 $[0^\circ, 90^\circ]$ 。此外，完成x0，y0和z0的初始化，并增加一位符号位。




```

1 //phase_in[15:14] determines which quadrant the angle is.
2 //00 means first;01 means second;00 means third;00 means fourth
3 //initialization: x0 = K,y0 = 0,z0 = phase_in,then the last result(x16,y16) =
  (cos(phase_in),sin(phase_in))
4 always @ (posedge clk)//stage 0,not pipeline
5 begin
6     x0 <= {1'b0,K}; //add one signed bit,0 means positive
7     y0 <= 17'b0;
8     z0 <= {3'b0,phase_in[13:0]}; //control the phase_in to the range[0-Pi/2]
9 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： $x(n+1) \leq x(n) + \{n\{y(n)[16]\}, y(n)[16:n]\}$ ， 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。



```

1 always @ (posedge clk)//stage 1
2 begin
3     if(z0[16])//the diff is negative so clockwise
4     begin
5         x1 <= x0 + y0;
6         y1 <= x0 - y0;
7         z1 <= z0 + `rot0;
8     end
9     else
10    begin
11        x1 <= x0 - y0;//x1 <= x0;
12        y1 <= x0 + y0;//y1 <= x0;
13        z1 <= z0 - `rot0;//reversal 45
14    end
15 end
16
17 always @ (posedge clk)//stage 2
18 begin
19     if(z1[16])//the diff is negative so clockwise
20     begin
21         x2 <= x1 + {y1[16],y1[16:1]};
22         y2 <= y1 - {x1[16],x1[16:1]};
23         z2 <= z1 + `rot1;//clockwise 26
24     end
25     else
26     begin
27         x2 <= x1 - {y1[16],y1[16:1]};
28         y2 <= y1 + {x1[16],x1[16:1]};
29         z2 <= z1 - `rot1;//anti-clockwise 26
30     end
31 end
32
33 always @ (posedge clk)//stage 3
34 begin
35     if(z2[16])//the diff is negative so clockwise
36     begin
37         x3 <= x2 + {{2{y2[16]}},y2[16:2]}; //right shift n bits,divide 2^n
38         y3 <= y2 - {{2{x2[16]}},x2[16:2]}; //left adds n bits of MSB,in first
quadrant x or y are positive,MSB =0 ? ?
39         z3 <= z2 + `rot2;//clockwise 14    //difference of positive and
negative number and no round(4,5)
40     end
41     else
42     begin
43         x3 <= x2 - {{2{y2[16]}},y2[16:2]};
44         y3 <= y2 + {{2{x2[16]}},x2[16:2]};
45         z3 <= z2 - `rot2;//anti-clockwise 14
46     end
47 end
48
49 always @ (posedge clk)//stage 4
50 begin
51     if(z3[16])
52     begin
53         x4 <= x3 + {{3{y3[16]}},y3[16:3]};
54         y4 <= y3 - {{3{x3[16]}},x3[16:3]};
55         z4 <= z3 + `rot3;//clockwise 7
56     end
57     else

```

```

58     begin
59         x4 <= x3 - {{3{y3[16]}}},y3[16:3]];
60         y4 <= y3 + {{3{x3[16]}}},x3[16:3]];
61         z4 <= z3 - `rot3;//anti-clockwise 7
62     end
63 end
64
65 always @ (posedge clk)//stage 5
66 begin
67     if(z4[16])
68     begin
69         x5 <= x4 + {{4{y4[16]}}},y4[16:4]];
70         y5 <= y4 - {{4{x4[16]}}},x4[16:4]];
71         z5 <= z4 + `rot4;//clockwise 3
72     end
73     else
74     begin
75         x5 <= x4 - {{4{y4[16]}}},y4[16:4]];
76         y5 <= y4 + {{4{x4[16]}}},x4[16:4]];
77         z5 <= z4 - `rot4;//anti-clockwise 3
78     end
79 end
80
81 always @ (posedge clk)//STAGE 6
82 begin
83     if(z5[16])
84     begin
85         x6 <= x5 + {{5{y5[16]}}},y5[16:5]];
86         y6 <= y5 - {{5{x5[16]}}},x5[16:5]];
87         z6 <= z5 + `rot5;//clockwise 1
88     end
89     else
90     begin
91         x6 <= x5 - {{5{y5[16]}}},y5[16:5]];
92         y6 <= y5 + {{5{x5[16]}}},x5[16:5]];
93         z6 <= z5 - `rot5;//anti-clockwise 1
94     end
95 end
96
97 always @ (posedge clk)//stage 7
98 begin
99     if(z6[16])
100     begin
101         x7 <= x6 + {{6{y6[16]}}},y6[16:6]];
102         y7 <= y6 - {{6{x6[16]}}},x6[16:6]];
103         z7 <= z6 + `rot6;
104     end
105     else
106     begin
107         x7 <= x6 - {{6{y6[16]}}},y6[16:6]];
108         y7 <= y6 + {{6{x6[16]}}},x6[16:6]];
109         z7 <= z6 - `rot6;
110     end
111 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。



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



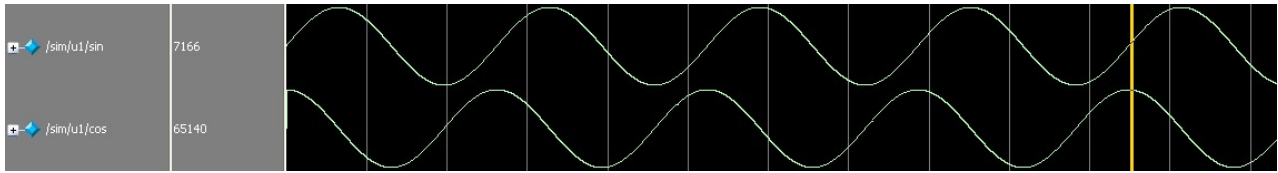
最后，根据寄存的高2位角度输入值，利用三角函数关系，得出最后的结果，其中负数进行了补码操作。



```
1 //alter register, according to quadrant[16] to transform the result to the
right result
2 always @ (posedge clk) begin
3   eps <= z15;
4   case(quadrant[16]) //or 15
5     2'b00:begin //if the phase is in first quadrant,the sin(X)=sin(A),cos(X)=cos(A)
6       cos <= x16;
7       sin <= y16;
8     end
9     2'b01:begin //if the phase is in second quadrant,the
sin(X)=sin(A+90)=cosA,cos(X)=cos(A+90)=-sinA
10      cos <= ~(y16) + 1'b1;//-sin
11      sin <= x16;//cos
12    end
13    2'b10:begin //if the phase is in third quadrant,the sin(X)=sin(A+180)=-
sinA,cos(X)=cos(A+180)=-cosA
14      cos <= ~(x16) + 1'b1;//-cos
15      sin <= ~(y16) + 1'b1;//-sin
16    end
17    2'b11:begin //if the phase is in forth quadrant,the sin(X)=sin(A+270)=-
cosA,cos(X)=cos(A+270)=sinA
18      cos <= y16;//sin
19      sin <= ~(x16) + 1'b1;//-cos
20    end
21  endcase
22 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： $x(n+1) \leq x(n) + \{ \{n\{y(n)[16]\}\}, y(n)[16:n] \}$ ，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：



```

1 `timescale 1 ns/100 ps
2 //360°--2^16,phase_in = 16bits (input [15:0] phase_in)
3 //1°--2^16/360
4 `define rot0 16'h2000 //45
5 `define rot1 16'h12e4 //26.5651
6 `define rot2 16'h09fb //14.0362
7 `define rot3 16'h0511 //7.1250
8 `define rot4 16'h028b //3.5763
9 `define rot5 16'h0145 //1.7899
10 `define rot6 16'h00a3 //0.8952
11 `define rot7 16'h0051 //0.4476
12 `define rot8 16'h0028 //0.2238
13 `define rot9 16'h0014 //0.1119
14 `define rot10 16'h000a //0.0560
15 `define rot11 16'h0005 //0.0280
16 `define rot12 16'h0003 //0.0140
17 `define rot13 16'h0001 //0.0070
18 `define rot14 16'h0001 //0.0035
19 `define rot15 16'h0000 //0.0018
20
21 module cordic(
22 output reg signed [16:0] sin,cos,eps,
23 input [15:0] phase_in,
24 input clk
25 );
26 parameter PIPELINE = 16;
27 //parameter K = 16'h4dba;//k=0.607253*2^15
28 parameter K = 17'h09b74;//k=0.607253*2^16,9b74,
29 //pipeline 16-level //maybe overflow,matlab result not overflow
30 //MSB is signed bit,transform the sin and cos according to phase_in[15:14]
31 reg signed [16:0] x0=0,y0=0,z0=0;
32 reg signed [16:0] x1=0,y1=0,z1=0;
33 reg signed [16:0] x2=0,y2=0,z2=0;
34 reg signed [16:0] x3=0,y3=0,z3=0;
35 reg signed [16:0] x4=0,y4=0,z4=0;
36 reg signed [16:0] x5=0,y5=0,z5=0;
37 reg signed [16:0] x6=0,y6=0,z6=0;
38 reg signed [16:0] x7=0,y7=0,z7=0;
39 reg signed [16:0] x8=0,y8=0,z8=0;
40 reg signed [16:0] x9=0,y9=0,z9=0;
41 reg signed [16:0] x10=0,y10=0,z10=0;
42 reg signed [16:0] x11=0,y11=0,z11=0;
43 reg signed [16:0] x12=0,y12=0,z12=0;
44 reg signed [16:0] x13=0,y13=0,z13=0;
45 reg signed [16:0] x14=0,y14=0,z14=0;
46 reg signed [16:0] x15=0,y15=0,z15=0;
47 reg signed [16:0] x16=0,y16=0,z16=0;
48
49 reg [1:0] quadrant [PIPELINE:0];
50 integer i;
51 initial
52 begin
53     for(i=0;i<=PIPELINE;i=i+1)
54         quadrant[i] = 2'b0;
55 end
56
57 //phase_in[15:14] determines which quadrant the angle is.
58 //00 means first;01 means second;00 means third;00 means fourth
59 //initialization: x0 = K,y0 = 0,z0 = phase_in,then the last result(x16,y16) =

```

```

(cos(phase_in),sin(phase_in))
60 always @ (posedge clk)//stage 0,not pipeline
61 begin
62     x0 <= K; //add one signed bit,0 means positive
63     y0 <= 17'd0;
64     z0 <= {3'b0,phase_in[13:0]}; //control the phase_in to the range[0-Pi/2]
65 end
66 //pipeline
67 //z0[16] = 0,positive
68 always @ (posedge clk)//stage 1
69 begin
70     if(z0[16])//the diff is negative so clockwise
71     begin
72         x1 <= x0 + y0;
73         y1 <= y0 - x0;
74         z1 <= z0 + `rot0;
75     end
76     else
77     begin
78         x1 <= x0 - y0;//x1 <= x0;
79         y1 <= y0 + x0;//y1 <= x0;
80         z1 <= z0 - `rot0;//reversal 45
81     end
82 end
83
84 always @ (posedge clk)//stage 2
85 begin
86     if(z1[16])//the diff is negative so clockwise
87     begin
88         x2 <= x1 + (y1 >>> 1);
89         y2 <= y1 - (x1 >>> 1);
90         z2 <= z1 + `rot1;//clockwise 26
91     end
92     else
93     begin
94         x2 <= x1 - (y1 >>> 1);
95         y2 <= y1 + (x1 >>> 1);
96         z2 <= z1 - `rot1;//anti-clockwise 26
97     end
98 end
99
100 always @ (posedge clk)//stage 3
101 begin
102     if(z2[16])//the diff is negative so clockwise
103     begin
104         x3 <= x2 + (y2 >>> 2); //right shift n bits,divide 2^n,signed
extension,Arithmetic shift right
105         y3 <= y2 - (x2 >>> 2); //left adds n bits of MSB,in first quadrant x
or y are positive,MSB =0 ? ?
106         z3 <= z2 + `rot2;//clockwise 14    //difference of positive and
negative number and no round(4,5)
107     end
108     else
109     begin
110         x3 <= x2 - (y2 >>> 2); //Arithmetic shift right
111         y3 <= y2 + (x2 >>> 2);
112         z3 <= z2 - `rot2;//anti-clockwise 14
113     end
114 end

```

```

115
116 always @ (posedge clk)//stage 4
117 begin
118     if(z3[16])
119         begin
120             x4 <= x3 + (y3 >>> 3);
121             y4 <= y3 - (x3 >>> 3);
122             z4 <= z3 + `rot3;//clockwise 7
123         end
124     else
125         begin
126             x4 <= x3 - (y3 >>> 3);
127             y4 <= y3 + (x3 >>> 3);
128             z4 <= z3 - `rot3;//anti-clockwise 7
129         end
130 end
131
132 always @ (posedge clk)//stage 5
133 begin
134     if(z4[16])
135         begin
136             x5 <= x4 + (y4 >>> 4);
137             y5 <= y4 - (x4 >>> 4);
138             z5 <= z4 + `rot4;//clockwise 3
139         end
140     else
141         begin
142             x5 <= x4 - (y4 >>> 4);
143             y5 <= y4 + (x4 >>> 4);
144             z5 <= z4 - `rot4;//anti-clockwise 3
145         end
146 end
147
148 always @ (posedge clk)//STAGE 6
149 begin
150     if(z5[16])
151         begin
152             x6 <= x5 + (y5 >>> 5);
153             y6 <= y5 - (x5 >>> 5);
154             z6 <= z5 + `rot5;//clockwise 1
155         end
156     else
157         begin
158             x6 <= x5 - (y5 >>> 5);
159             y6 <= y5 + (x5 >>> 5);
160             z6 <= z5 - `rot5;//anti-clockwise 1
161         end
162 end
163
164 always @ (posedge clk)//stage 7
165 begin
166     if(z6[16])
167         begin
168             x7 <= x6 + (y6 >>> 6);
169             y7 <= y6 - (x6 >>> 6);
170             z7 <= z6 + `rot6;
171         end
172     else
173         begin

```



```

174         x7 <= x6 - (y6 >>> 6);
175         y7 <= y6 + (x6 >>> 6);
176         z7 <= z6 - `rot6;
177     end
178 end
179
180 always @ (posedge clk)//stage 8
181 begin
182     if(z7[16])
183     begin
184         x8 <= x7 + (y7 >>> 7);
185         y8 <= y7 - (x7 >>> 7);
186         z8 <= z7 + `rot7;
187     end
188     else
189     begin
190         x8 <= x7 - (y7 >>> 7);
191         y8 <= y7 + (x7 >>> 7);
192         z8 <= z7 - `rot7;
193     end
194 end
195
196 always @ (posedge clk)//stage 9
197 begin
198     if(z8[16])
199     begin
200         x9 <= x8 + (y8 >>> 8);
201         y9 <= y8 - (x8 >>> 8);
202         z9 <= z8 + `rot8;
203     end
204     else
205     begin
206         x9 <= x8 - (y8 >>> 8);
207         y9 <= y8 + (x8 >>> 8);
208         z9 <= z8 - `rot8;
209     end
210 end
211
212 always @ (posedge clk)//stage 10
213 begin
214     if(z9[16])
215     begin
216         x10 <= x9 + (y9 >>> 9);
217         y10 <= y9 - (x9 >>> 9);
218         z10 <= z9 + `rot9;
219     end
220     else
221     begin
222         x10 <= x9 - (y9 >>> 9);
223         y10 <= y9 + (x9 >>> 9);
224         z10 <= z9 - `rot9;
225     end
226 end
227
228 always @ (posedge clk)//stage 11
229 begin
230     if(z10[16])
231     begin
232         x11 <= x10 + (y10 >>> 10);

```

```

233     y11 <= y10 - (x10 >>> 10);
234     z11 <= z10 + `rot10;//clockwise 3
235 end
236 else
237 begin
238     x11 <= x10 - (y10 >>> 10);
239     y11 <= y10 + (x10 >>> 10);
240     z11 <= z10 - `rot10;//anti-clockwise 3
241 end
242 end
243
244 always @ (posedge clk)//STAGE 12
245 begin
246     if(z11[16])
247     begin
248         x12 <= x11 + (y11 >>> 11);
249         y12 <= y11 - (x11 >>> 11);
250         z12 <= z11 + `rot11;//clockwise 1
251     end
252     else
253     begin
254         x12 <= x11 - (y11 >>> 11);
255         y12 <= y11 + (x11 >>> 11);
256         z12 <= z11 - `rot11;//anti-clockwise 1
257     end
258 end
259
260 always @ (posedge clk)//stage 13
261 begin
262     if(z12[16])
263     begin
264         x13 <= x12 + (y12 >>> 12);
265         y13 <= y12 - (x12 >>> 12);
266         z13 <= z12 + `rot12;
267     end
268     else
269     begin
270         x13 <= x12 - (y12 >>> 12);
271         y13 <= y12 + (x12 >>> 12);
272         z13 <= z12 - `rot12;
273     end
274 end
275
276 always @ (posedge clk)//stage 14
277 begin
278     if(z13[16])
279     begin
280         x14 <= x13 + (y13 >>> 13);
281         y14 <= y13 - (x13 >>> 13);
282         z14 <= z13 + `rot13;
283     end
284     else
285     begin
286         x14 <= x13 - (y13 >>> 13);
287         y14 <= y13 + (x13 >>> 13);
288         z14 <= z13 - `rot13;
289     end
290 end
291

```

```

292 always @ (posedge clk)//stage 15
293 begin
294     if(z14[16])
295     begin
296         x15 <= x14 + (y14 >>> 14);
297         y15 <= y14 - (x14 >>> 14);
298         z15 <= z14 + `rot14;
299     end
300     else
301     begin
302         x15 <= x14 - (y14 >>> 14);
303         y15 <= y14 + (x14 >>> 14);
304         z15 <= z14 - `rot14;
305     end
306 end
307
308 always @ (posedge clk)//stage 16
309 begin
310     if(z15[16])
311     begin
312         x16 <= x15 + (y15 >>> 15);
313         y16 <= y15 - (x15 >>> 15);
314         z16 <= z15 + `rot15;
315     end
316     else
317     begin
318         x16 <= x15 - (y15 >>> 15);
319         y16 <= y15 + (x15 >>> 15);
320         z16 <= z15 - `rot15;
321     end
322 end
323 //according to the pipeline,register phase_in[15:14]
324 always @ (posedge clk)
325 begin
326     quadrant[0] <= phase_in[15:14];
327     quadrant[1] <= quadrant[0];
328     quadrant[2] <= quadrant[1];
329     quadrant[3] <= quadrant[2];
330     quadrant[4] <= quadrant[3];
331     quadrant[5] <= quadrant[4];
332     quadrant[6] <= quadrant[5];
333     quadrant[7] <= quadrant[6];
334     quadrant[8] <= quadrant[7];
335     quadrant[9] <= quadrant[8];
336     quadrant[10] <= quadrant[9];
337     quadrant[11] <= quadrant[10];
338     quadrant[12] <= quadrant[11];
339     quadrant[13] <= quadrant[12];
340     quadrant[14] <= quadrant[13];
341     quadrant[15] <= quadrant[14];
342     quadrant[16] <= quadrant[15];
343 end
344 //alter register, according to quadrant[16] to transform the result to the
right result
345 always @ (posedge clk) begin
346     eps <= z15;
347     case(quadrant[16]) //or 15
348     2'b00:begin //if the phase is in first quadrant,the
sin(X)=sin(A),cos(X)=cos(A)

```

```

349         cos <= x16;
350         sin <= y16;
351     end
352 2'b01:begin //if the phase is in second quadrant,the
sin(X)=sin(A+90)=cosA,cos(X)=cos(A+90)=-sinA
353         cos <= ~(y16) + 1'b1;//-sin
354         sin <= x16;//cos
355     end
356 2'b10:begin //if the phase is in third quadrant,the sin(X)=sin(A+180)=-
sinA,cos(X)=cos(A+180)=-cosA
357         cos <= ~(x16) + 1'b1;//-cos
358         sin <= ~(y16) + 1'b1;//-sin
359     end
360 2'b11:begin //if the phase is in forth quadrant,the sin(X)=sin(A+270)=-
cosA,cos(X)=cos(A+270)=sinA
361         cos <= y16;//sin
362         sin <= ~(x16) + 1'b1;//-cos
363     end
364 endcase
365 end
366
367 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
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。

