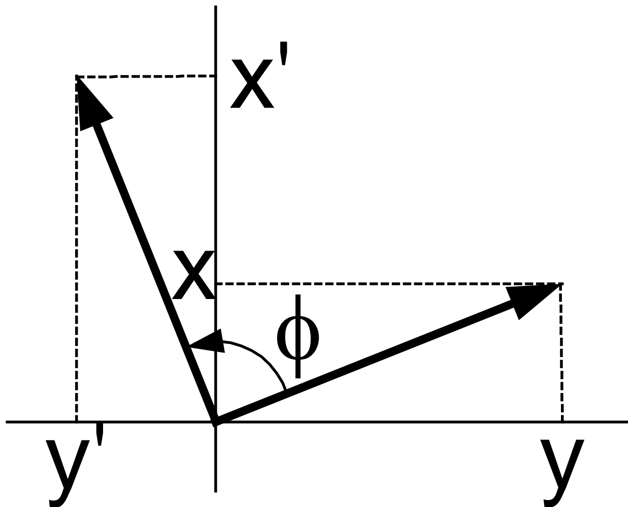
CORDIC Algorithm
COordinate Rotation DIgital Computer

- Example: vector rotation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

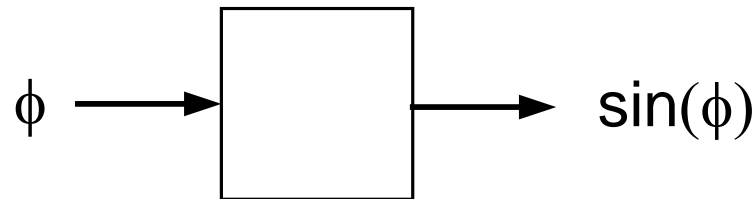


Implementation cost:

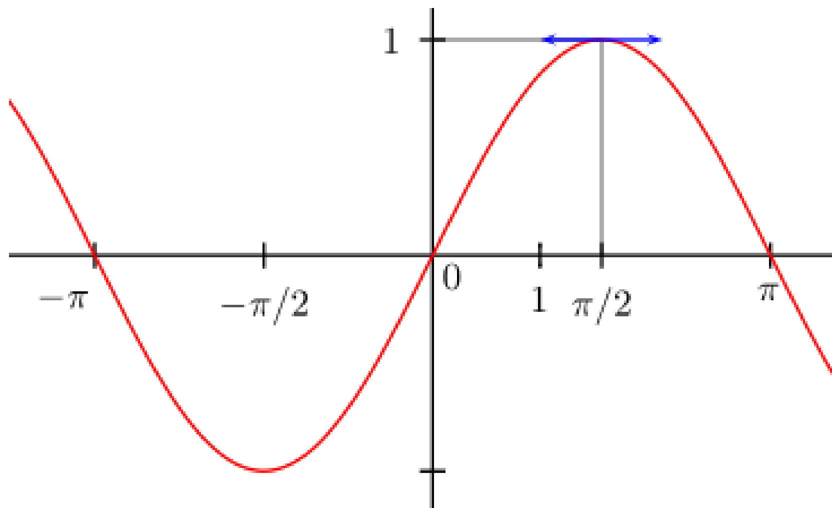
- 2 multiplications
- 2 addition or subtraction
- 2 sin() and cos()

How do we compute sin(ϕ) or cos(ϕ)?

- Easy, but expensive (for high accuracy):
  Direct table look-up:



- Exploit symmetry of the sine function:



$\sin(\phi)=\sin(\pi - \phi)$

$\sin(-\phi)=-\sin(\phi)$

also:

$\sin(\phi)=\cos(\pi/2 - \phi)$

Only one quadrant needed

Direct Table Look-up (Example)

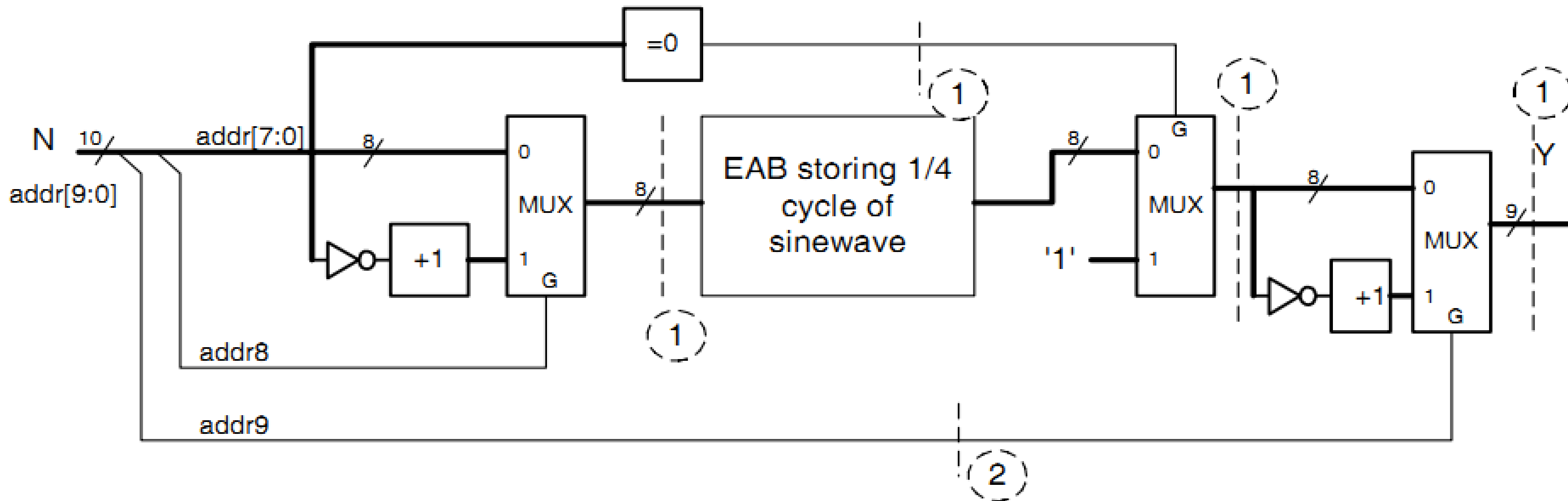
- Example: Use embedded block RAM (EAB) in 256 x 8 bit configuration to store ¼ cycle of a sine table such that:
  - Mem[K] = 255 * sin (π * K / 512) for K = 0 to 255.
  - Generate the other quadrants by manipulating the address and negating the ROM/RAM values.
  - The rule to generate the EAB address '**reflection**' and amplitude negation are:-

addr9	addr8	Address to EAB	Negation
0	0	addr[7:0]	No
0	1	256 – addr[7:0]	No
1	0	addr[7:0]	Yes
1	1	256 – addr[7:0]	Yes

from Peter Cheung, Imperial College London

Direct Table Look-up (Example)

- This works except for N=256 and 768 when addr[7:0] = 0.
- Therefore, detect this condition and force output to either +255 or –255.
- Improve speed by inserting pipeline registers at dotted lines.
- Numbers in circle indicate number of pipeline register stages.



from Peter Cheung, Imperial College London

Direct Table Look-up (Example)

- Trade-off between memory and the logic for multiplexing and complement computation
- Sine tables are provided in several DSPs
- Note: a BRAM on our Spartan-6 can hold 2K x 9 bit (considering the example, extra logic is not required)
- For higher precision: two level table look-up:

$$\sin(\alpha + \beta) = \sin(\alpha) \cos(\beta) + \cos(\alpha) \sin(\beta)$$

(trick: coarse table for α and fine table for β)

Requires two multiplications and one addition

How is it done in Software? (MATHLIB.C)

```
float sin(arg)
float arg;
{   /* Coefficients are #3370 from Hart & Cheney (18.80D). */
    static float p[5] = {.1357884e8, -.49429081e7, .440103053e6, -.138472724e5, .145968841e3 };
    static float q[5] = {.864455865e7, .408179225e6, .946309610e4, .132653491e3, 1.0 };
    static float y,ysq;
    static long quad,e;
    float F_poly();
    quad = (arg >= 0.0) ? 0 : (arg = -arg, 2);
    arg *= .63661977;       /* 2 / pi */
    y = arg - (e = arg);
    quad=(e + quad) % 4;
    if (quad & 1) y = 1.0 - y;
    if(quad & 2) y = -y;

    ysq = y * y;
    return y * F_poly(ysq,p,4)/F_poly(ysq,q,4);
}

float F_poly(x, p, n)  /* evaluate nth deg. polynomial, n >= 1 */
float x,p[];
{   static float result;  static int i;
    for (result = p[i = n]; i; ) result = p[--i] + x * result;
    return result; }
```

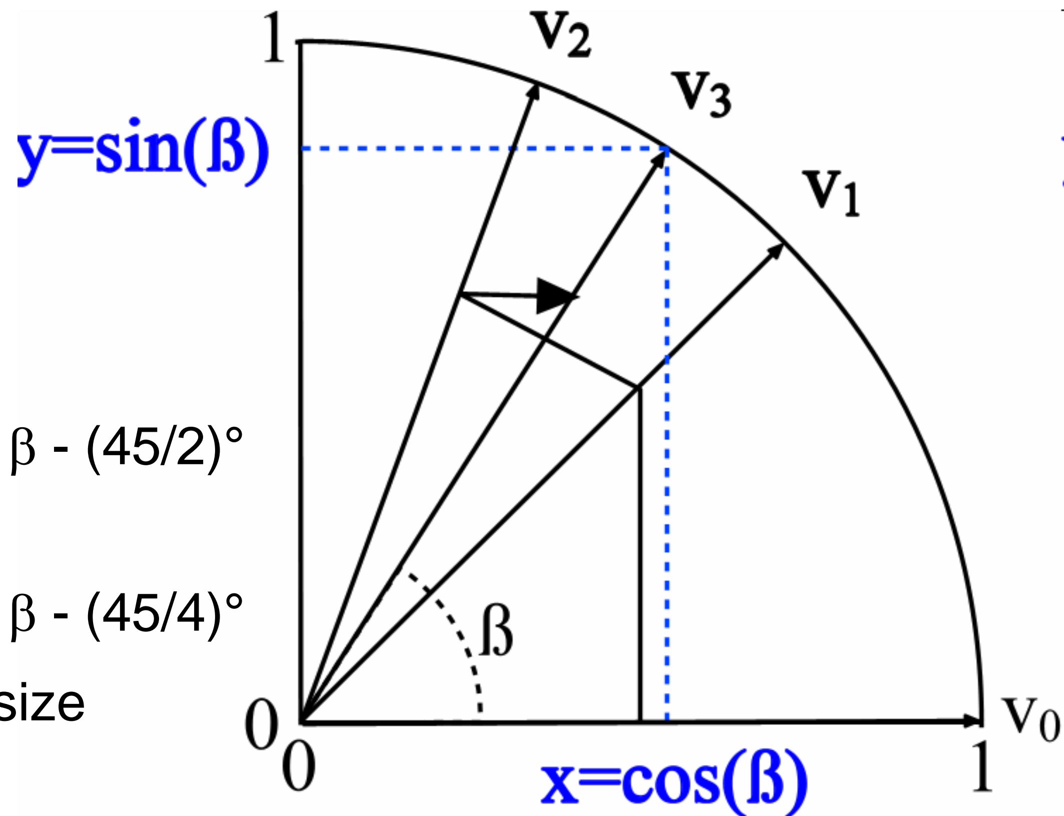
The coefficents are from an article of 1968 by John Hart

CORDIC Algorithm
COordinate Rotation DIgital Computer

- Method for elementary function evaluation (e.g., $\sin(z)$, $\cos(z)$, $\tan^{-1}(y)$)
- The modern CORDIC algorithm was first described in 1959 by Jack E. Volder. It was developed to replace the analog resolver in the B-58 bomber's navigation computer. (from Wikipedia)
- Used in Intel 80x87 coprocessor and Intel 80486
- Commonly used for FPGAs
- Complexity Comparable to Division

CORDIC Algorithm: Key Ideas

- Rather than computing $\sin(\phi)$ directly, we iteratively rotate β towards ϕ
- Ideal search within first quadrant:
  - Step 1: set $\beta = 45°$
  - Step 2: if $\phi >= \beta$ then
    $\beta = \beta + (45/2)°$ else $\beta = \beta - (45/2)°$
  - Step 3: if $\phi >= \beta$ then
    $\beta = \beta + (45/4)°$ else $\beta = \beta - (45/4)°$
  - Continue by halving step size of β in each iteration
- Search is stable if b[i]>b[i+1]>=b[i]/2



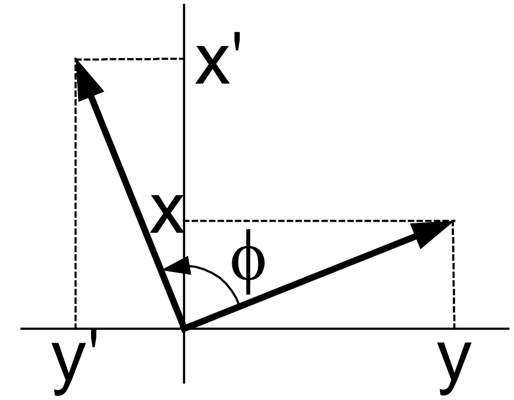
from Wikipedia

CORDIC Algorithm: Back to our Vector Rotation Example

- Example: vector rotation:

$$x' = x\ cos(\phi) - y\ sin(\phi)$$

$$y' = y\ cos(\phi) + x\ sin(\phi)$$



- Rewrite as: (note $sin(\phi)/cos(\phi) = tan(\phi)$)

$$x' = cos(\phi)\ [\ x - y\ tan(\phi)\ ]$$

$$y' = cos(\phi)\ [\ y + x\ tan(\phi)\ ]$$

- Trick: allow only iterative rotations so that $tan(\beta) = \underline{\underline{\pm 2^{-i}}}$

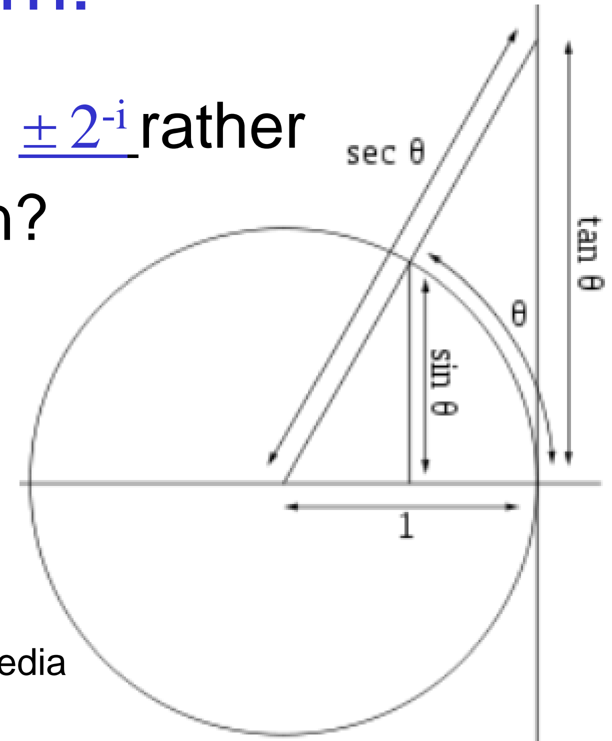
$$x_{i+1} = cos(\ tan^{-1}(\pm 2^{-i})\ ) \cdot [\ x_i - y_i \cdot \underline{d_i \cdot 2^{-i}}\ ]$$

$$y_{i+1} = cos(\ tan^{-1}(\pm 2^{-i})\ ) \cdot [\ y_i + x_i \cdot \underline{d_i \cdot 2^{-i}}\ ]$$

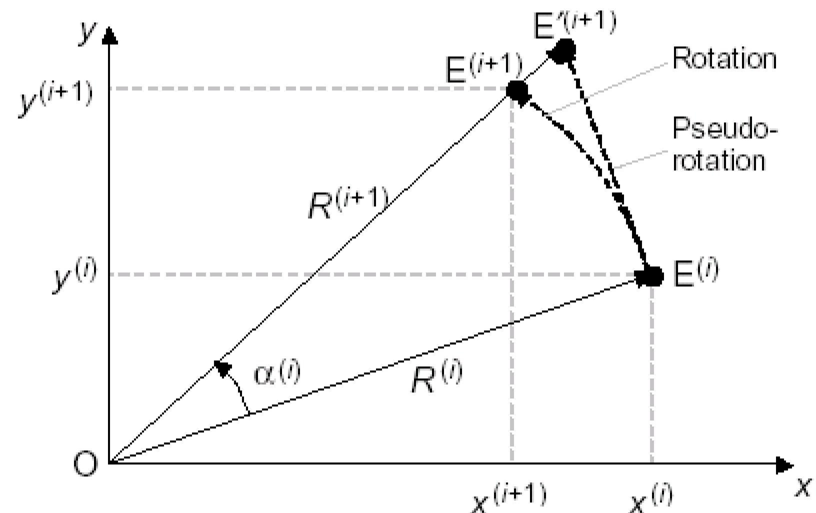
With the rotate direction $d_i = \pm 1$

CORDIC Algorithm:

- What does it mean to rotate $tan(\beta) = \pm 2^{-i}$ rather then rotating $\pm \beta \cdot 2^{-i}$ in each iteration?
- For smaller angles, it converges against the same:
- Important:
  $\pm 2^{-i}$ is a simple shift!
  (can in some cases be directly implemented within the routing)
- Rotation by an arbitrary angle is difficult, so we perform psuedorotations



from Wikipedia

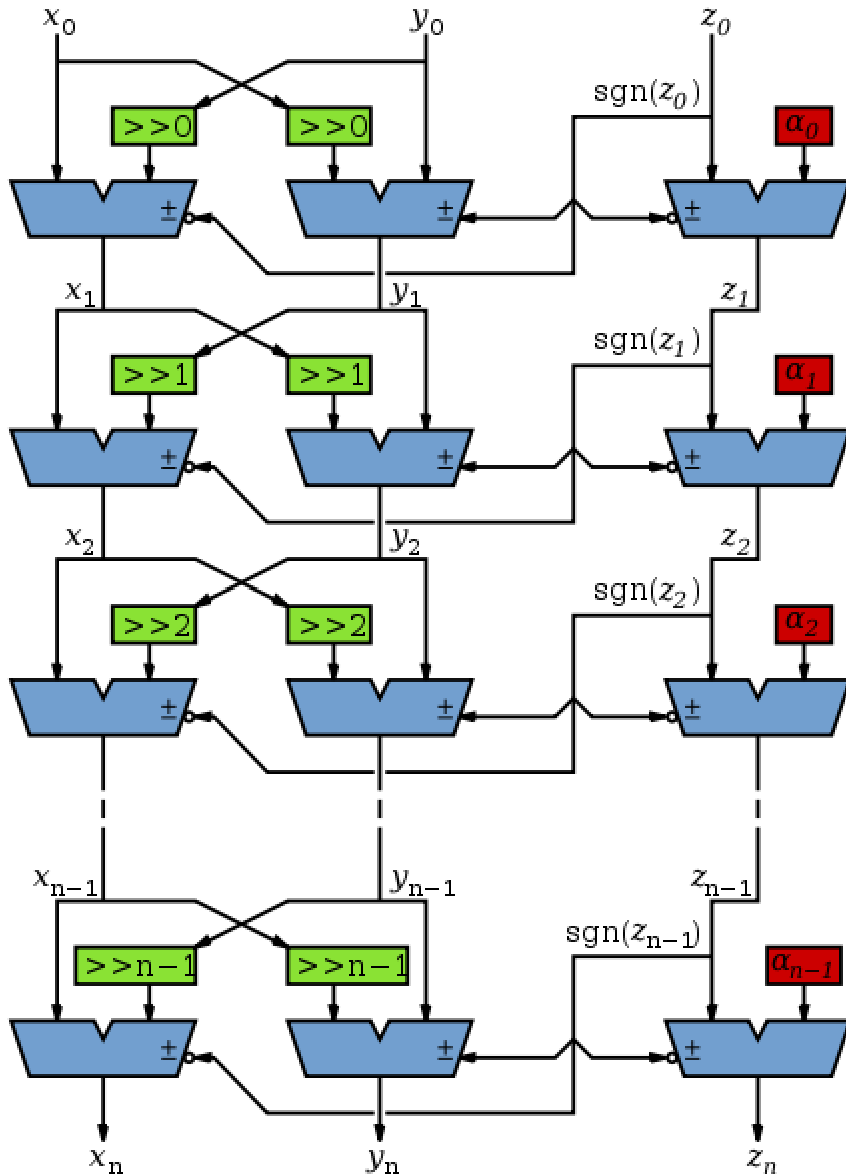


CORDIC Rotation Angles

i	2^{-i}	arctan(2^{-i})*360/2π		45* 2^{-i}
0	1	45*		
1	0.5	26.56505118*	4.065051177*	22.5
2	0.25	14.03624347	0.753717879	11.25
3	0.125	7.125016349	0.106894615	5.625
4	0.0625	3.576334375	0.013826201	2.8125
5	0.03125	1.789910608	0.001743421	1.40625
6	0.015625	0.89517371	0.000218406	0.703125
7	0.0078125	0.447614171	2.73158E-05	0.3515625
8	0.00390625	0.2238105	3.41494E-06	0.17578125
9	0.001953125	0.111905677	4.26882E-07	0.087890625
10	0.000976563	0.055952892	5.33607E-08	0.043945313
11	0.000488281	0.027976453	6.6701E-09	0.021972656
12	0.000244141	0.013988227	8.33763E-10	0.010986328
13	0.00012207	0.006994114	1.0422E-10	0.005493164
14	6.10352E-05	0.003497057	1.30276E-11	0.002746582
15	3.05176E-05	0.001748528	1.62844E-12	0.001373291
16	1.52588E-05	0.000874264	2.03555E-13	0.000686646
17	7.62939E-06	0.000437132	2.54444E-14	0.000343323
18	3.8147E-06	0.000218566	3.18056E-15	0.000171661
19	1.90735E-06	0.000109283	3.97577E-16	8.58307E-05
20	9.53674E-07	5.46415E-05	4.96903E-17	4.29153E-05

* 4.06505=26.565-45/2

CORDIC Hardware (Bit-parallel, unrolled)



http://en.wikibooks.org/wiki/Digital_Circuits/CORDIC

$$x_{i+1} = K_i \cdot [\, x_i - y_i \cdot d_i \cdot 2^{-i} \,]$$

$$y_{i+1} = K_i \cdot [\, y_i + x_i \cdot d_i \cdot 2^{-i} \,]$$

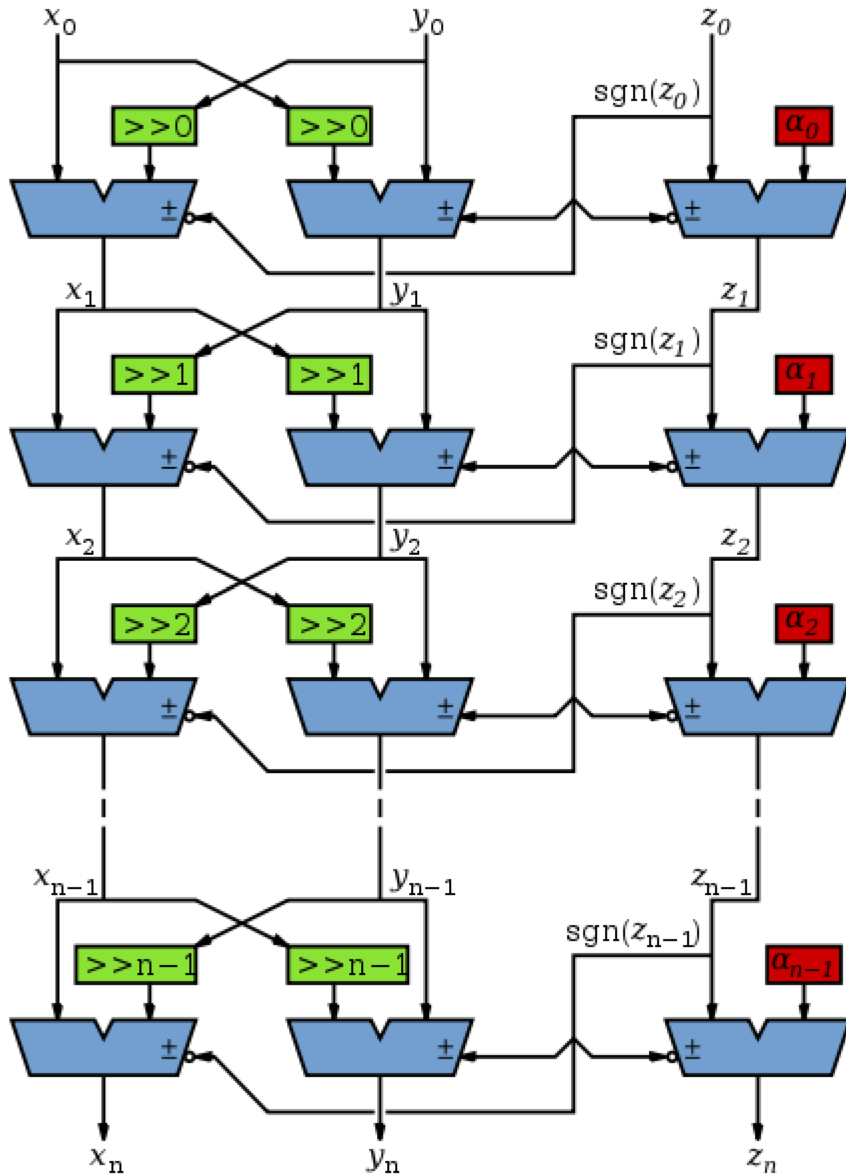
$$z_{i+1} = z_i - d_i \cdot arctan(2^{-i})$$

$arctan(2^{-i})$ values are precomputed (a_0, a_1, ...) can be scaled to binary number range, e.g. 2π=256

d_i = -1 *if* $z_i < 0$, +1 otherwise; to evaluate $z_i < 0$, we simply use the z_i sign bit (MSB).

We will soon come to K_i.

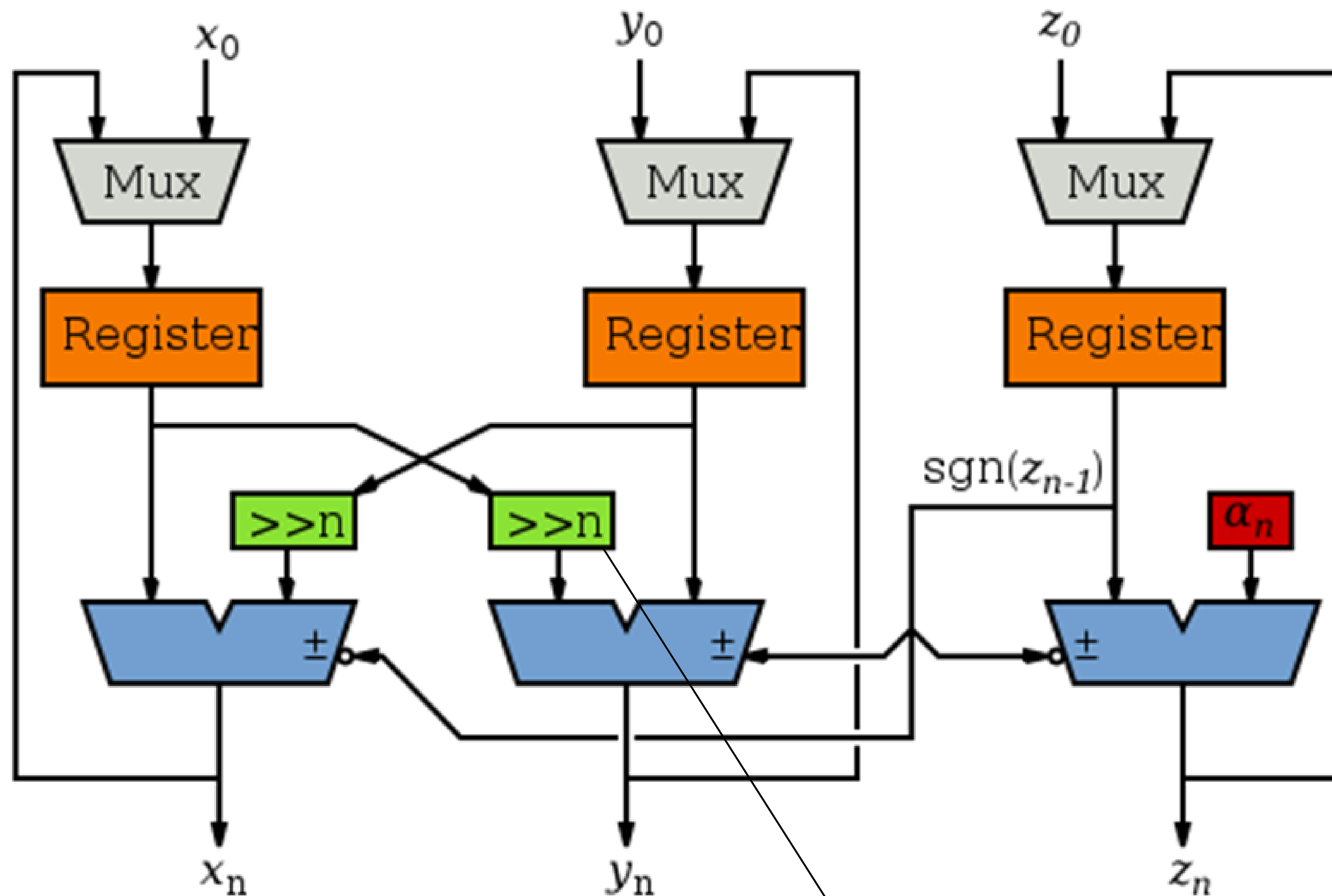
CORDIC Hardware (Bit-parallel, unrolled)



http://en.wikibooks.org/wiki/Digital_Circuits/CORDIC

- Implementation cost: three ADD/SUB ALUs per iteration.
- Shift operations: hardwired
- rotate angles (a_i) are fit into the logic
- Typically with pipeline register after each iteration (results in very high throughput)
- Improvement of the angle resolution by almost one signal bit iteration.

CORDIC Hardware (Bit-parallel, iterative)



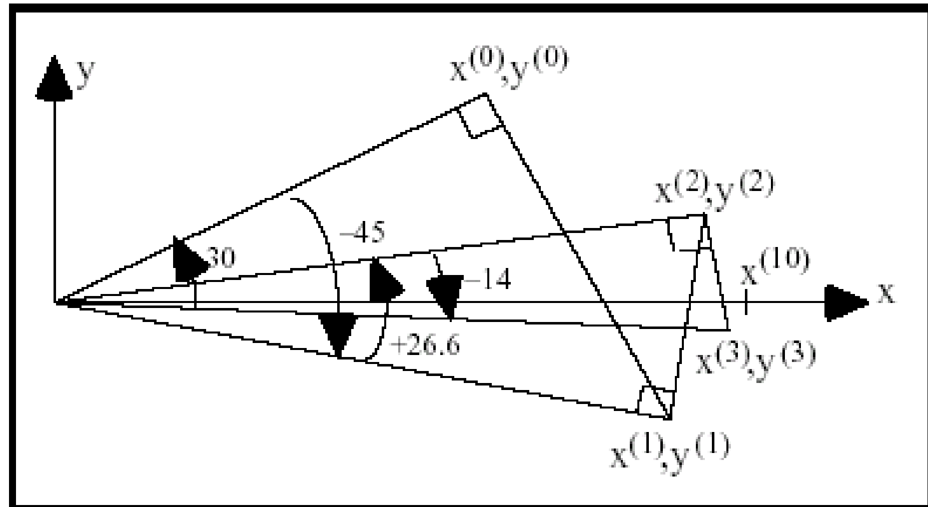
http://en.wikibooks.org/wiki/Digital_Circuits/CORDIC

- Lower throughput (n times less)
- The barrel-shifter is variable and costs logic
- a_n is stored in a small register file

Basic CORDIC Iteration

Choosing the signs of the rotation angles in order to force *z* to 0

i	$z^{(i)}$	$-$	$\alpha^{(i)}$	$=$	$z^{(i+1)}$
0	+30.0	–	45.0	=	–15.0
1	–15.0	+	26.6	=	+11.6
2	+11.6	–	14.0	=	–2.4
3	–2.4	+	7.1	=	+4.7
4	+4.7	–	3.6	=	+1.1
5	+1.1	–	1.8	=	–0.7
6	–0.7	+	0.9	=	+0.2
7	+0.2	–	0.4	=	–0.2
8	–0.2	+	0.2	=	+0.0
9	+0.0	–	0.1	=	–0.1



The first three of 10 pseudo-rotations leading from $(x^{(0)}, y^{(0)})$ to $(x^{(10)}, 0)$ in rotating by +30°.

CORDIC Algorithm: Gain

- Cordic uses iterative rotations in steps of $tan(\beta) = \pm 2^{-i}$

$$x_{i+1} = cos(\ tan^{-1}(\pm 2^{-i})\ ) \cdot [\ x_i - y_i \cdot d_i \cdot 2^{-i}\ ]$$

$$y_{i+1} = cos(\ tan^{-1}(\pm 2^{-i})\ ) \cdot [\ y_i + x_i \cdot d_i \cdot 2^{-i}\ ]$$

- How we deal with $cos(\ tan^{-1}(\pm 2^{-i})\ )$?
- The cosine is symmetric:

$$cos(\ tan^{-1}(2^{-i})\ ) = cos(\ tan^{-1}(-2^{-i})\ )$$

- $cos(\ tan^{-1}(2^{-i})\ )$ is the gain K_i of an iteration

$$K_i = \cos(\arctan(2^{-i})) = \frac{1}{\sqrt{1+2^{-2i}}}$$

- We can compute K offline for all n iterations: $K = \prod_n K_i$
- The gain approaches 0.6037, if n goes to infinity

CORDIC Algorithm: Gain

- In order to compensate the gain, we have to scale the result with the reciprocal value of the gain:

$$A = \prod_{n} \sqrt{1 + 2^{-2i}}$$

- We can compute A offline for all n iterations
- A approaches 1.647, if n goes to infinity
- No overhead if combined with other scaleing values

Rotation Angle Limits

- Rotation/Vector Algorithms Limited to ±90°
- Due to Use of $\alpha = \tan(2^0)$ for First Iteration
- Several Ways to Extend Range

Can use trig identities to covert the problem to one that is within the domain of convergence

One Way is to Use Additional Rotation for Angles Outside Range

This Rotation is Initial ±90° Rotation

$$x' = -d \cdot y$$

$$y' = d \cdot x$$

$$z' = z + d \cdot \left(\pi/2 \right)$$

$$d_i = \begin{cases} +1, & y < 0 \\ -1, & \text{otherwise} \end{cases}$$

CORDIC Uses

OPERATION	MODE	INITIALIZE	DIRECTION
Sine, Cosine	Rotation	$x=1/A_n$, $y=0$, $z=\alpha$	Reduce z to Zero
Polar to Rect.	Rotation	$x=(1/A_n)X_{mag}$, $y=0$, $z=X_{phase}$	Reduce z to Zero
General Rotation	Rotation	$x=(1/A_n)x_0$, $y=(1/A_n)y_0$, $z=\alpha$	Reduce z to Zero
Arctangent	Vector	$x=(1/A_n)x_0$, $y=(1/A_n)y_0$, $z=0$	Reduce y to Zero
Vector Magnitude	Vector	$x=(1/A_n)x_0$, $y=(1/A_n)y_0$, $z=0$	Reduce y to Zero
Rect. to Polar	Vector	$x=(1/A_n)x_0$, $y=(1/A_n)y_0$, $z=0$	Reduce y to Zero
Arcsine, Arccosine	Vector	$x=(1/A_n)$, $y=0$, $arg=\sin\alpha$ or $\cos\alpha$	Reduce y to Value in *arg* Register

- Can Use CORDIC For Others Also:
  - Linear Functions
  - Hyperbolic Functions
  - Square Rooting
  - Logarithms, Exponentials

CORDIC – Rotation/Vector Modes

- Rotation Mode:

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}$$
$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$
$$z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-i})$$

$$x_n = A_n \left[x_0 \cos z_0 - y_0 \sin z_0 \right]$$
$$y_n = A_n \left[y_0 \cos z_0 + x_0 \sin z_0 \right]$$
$$z_n = 0$$
$$A_n = \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}}$$
$$d_i = \begin{cases} -1, & z_i < 0 \\ +1, & \text{otherwise} \end{cases}$$

- Vector Mode:

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}$$
$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$
$$z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-i})$$

$$x_n = A_n \sqrt{x_0^{\ 2} + y_0^{\ 2}}$$
$$y_n = 0$$
$$z_n = z_0 + \tan{-1}\left(y_0 / x_0 \right)$$
$$A_n = \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}}$$
$$d_i = \begin{cases} +1, & y_i < 0 \\ -1, & \text{otherwise} \end{cases}$$