Progettazione e sintesi di circuiti digitali

Lecture 15
CORDIC algorithm
CORDIC Algorithm

• CORDIC (COordinate Rotation DIgital Computer) is an iterative algorithm developed to compute trigonometric functions using only shift and add/subtract operations

• Based on 2D coordinate rotation equation

• Simple extensions allow to compute square root and hyperbolic functions

• Used in early math-coprocessors, radar signal processing, robotics, 3D graphics, communication systems
CORDIC: coordinate rotation

• Simple rotation in 2D:
  – vector \((x_A, y_A)\) is rotated of an angle \(\theta\) to \((x_R, y_R)\)

\[
\begin{align*}
\theta & = \cos \theta - y_A \sin \theta \\
y_R &= x_A \sin \theta + y_A \cos \theta
\end{align*}
\]

2D rotation equations:

\begin{align*}
x_R &= x_A \cos \theta - y_A \sin \theta \\
y_R &= x_A \sin \theta + y_A \cos \theta
\end{align*}

in matrix form:

\[
\begin{bmatrix}
x_R \\
y_R
\end{bmatrix} = R(\theta) \begin{bmatrix} x_A \\
y_A
\end{bmatrix}
\]

where:

\[
R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\
\sin \theta & \cos \theta \end{bmatrix}
\]
CORDIC: coordinate rotation

• When a rotation $R(\theta)$ is applied to $(x_A=1, y_A=0)$, then the resulting vector is:

\[
\begin{bmatrix}
    x_R \\
    y_R
\end{bmatrix} = R(\theta) \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \cos \theta \\ \sin \theta \end{bmatrix}
\]

• Then, rotating vector $(1,0)$ by $\theta$ gives $\cos \theta$ and $\sin \theta$

• How to compute $(x_R, y_R)$ using only non costly operations (like shifts and adds)?
CORDIC: coordinate rotation

• Basic idea: decompose rotation $R(\theta)$ into a sequence of rotations $R(\alpha_k)$ such that that:

\[
\theta = \sum_{k=0}^{N-1} \alpha_k, \quad \Rightarrow \quad R(\theta) = \prod_{k=0}^{N-1} R(\alpha_k)
\]

• Then, vector rotation can be performed iteratively, using pre-computed values of $\cos \alpha_k$ and $\sin \alpha_k$:

\[
x_{k+1} = x_k \cos \alpha_k - y_k \sin \alpha_k
\]
\[
y_{k+1} = x_k \sin \alpha_k + y_k \cos \alpha_k
\]

• However, we still need multiplications. How to avoid them?
CORDIC: rotation decomposition

• Further decompose $R(\alpha_k)$ into a scaling operation and a rotation-extension:

$$x_{k+1} = \cos \alpha_k \left( x_k - y_k \tan \alpha_k \right)$$
$$y_{k+1} = \cos \alpha_k \left( x_k \tan \alpha_k + y_k \right)$$

• choose elementary angles $\alpha_k$ such that:

$$\alpha_k = \tan^{-1}(2^{-k}) \quad \Rightarrow \quad \tan \alpha_k = 2^{-k}$$

$\alpha_0 = 45^\circ$, $\alpha_1 = 26.6^\circ$, $\alpha_2 = 14^\circ$, $\alpha_3 = 7.1^\circ$, ...
CORDIC: rotation-extension

• Then, the rotation-extension operation:

\[
C(\alpha_k) = \frac{1}{\cos \alpha_k} \quad R(\alpha_k) = \begin{bmatrix}
1 & -d_k 2^{-k} \\
\left| d_k 2^{-k} & 1 \right|
\end{bmatrix}
\]

with \( d_k = \pm 1 \) (-1 is for negative angles as we defined only positive \( \alpha_k \)), results in a new set of iterative update equations that do not require multiplications:

\[
\tilde{x}_{k+1} = (\tilde{x}_k - \tilde{y}_k d_k \tan \alpha_k) = (\tilde{x}_k - \tilde{y}_k d_k 2^{-k})
\]

\[
\tilde{y}_{k+1} = (\tilde{x}_k d_k \tan \alpha_k + \tilde{y}_k) = (\tilde{x}_k d_k 2^{-k} + \tilde{y}_k)
\]

• However, the result is not a pure rotation
CORDIC: modulus scaling factor

- Since $C(\alpha_k)$ is a rotation scaled by a factor $1/\cos \alpha_k < 1$, the modulus of the rotated vector increases:

$$M_{k+1} = M_k \frac{1}{\cos \alpha_k}$$

- If $M_N$ is the vector modulus after $N$ rotation-extensions, then:

$$M_N = M_0 \prod_{k=0}^{N-1} \frac{1}{\cos \alpha_k} = M_0 \prod_{k=0}^{N-1} \left(1 + 2^{-2k}\right)^{1/2}$$
CORDIC: modulus scaling factor

• The modulus scaling factor asymptotic value is:

\[ K = \lim_{{N \to \infty}} \prod_{{k=0}}^{{N-1}} (1 + 2^{-2k})^{1/2} \approx 1.6468\ldots \]

• Only few iterations are required for the scaling factor to get very close to K:

<table>
<thead>
<tr>
<th>N</th>
<th>(K_N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.4142</td>
</tr>
<tr>
<td>2</td>
<td>1.5811</td>
</tr>
<tr>
<td>3</td>
<td>1.6298</td>
</tr>
<tr>
<td>4</td>
<td>1.6425</td>
</tr>
<tr>
<td>5</td>
<td>1.6457</td>
</tr>
<tr>
<td>6</td>
<td>1.6465</td>
</tr>
<tr>
<td>9</td>
<td>1.6468</td>
</tr>
</tbody>
</table>
CORDIC: modulus scaling factor

- Example of rotation-extension of an angle $\theta$
CORDIC algorithm: rotation mode

• The CORDIC rotation-extension operator $C(\alpha_k)$ can be operated in two modes, giving two variants of the CORDIC algorithm

• CORDIC algorithm – rotation mode:
  
  \[ x_{k+1} = x_k - y_k d_k 2^{-k} \]
  
  \[ y_{k+1} = x_k d_k 2^{-k} + y_k \]
  
  \[ z_{k+1} = z_k - d_k \alpha_k \]
  
  \[ d_k = -1 \text{ if } z_k < 0, \quad +1 \text{ otherwise} \]

  1. load $\theta$ in $z_0$ and initial vector coordinates in $(x_0, y_0)$
  2. compute $(x_{k+1}, y_{k+1}, z_{k+1})$
  3. if $z_{k+1} = 0 \text{ (or } z_{k+1} < \varepsilon \text{) stop, else go to 2}$
CORDIC algorithm: rotation mode

• At the end of the iterative algorithm we get:

\[ x_N = K_N (x_0 \cos \theta - y_0 \sin \theta) \]
\[ y_N = K_N (x_0 \sin \theta + y_0 \cos \theta) \]
\[ z_N = 0 \text{ (or } z_N < \varepsilon) \]

• Example: starting with the initial condition

\[ x_0 = 1/K, \ y_0 = 0, \ z_0 = \theta \]

given that \( K_N \approx K \), we get:

\[ x_N = \cos \theta \]
\[ y_N = \sin \theta \]
CORDIC algorithm: rotation mode

• Since:
  \[ \theta_{\text{MAX}} = \sum_{k=0}^{\infty} \alpha_k = \sum_{k=0}^{\infty} \tan^{-1}(2^{-k}) = 99.88^\circ \]

• when \( \theta \geq \theta_{\text{MAX}} \) or \( \theta \leq -\theta_{\text{MAX}} \), a correction to the algorithm is needed

• Then, we add a preliminary iteration such that, if \( \theta \geq 90^\circ \):
  \[
  z'_0 = \phi = \theta - 90^\circ \\
  x'_0 = -y_0 \\
  y'_0 = x_0 \\
  x_N = K_N(x'_0 \cos \phi - y'_0 \sin \phi) = K_N(x_0 \cos \theta - y_0 \sin \theta) \\
  y_N = K_N(x'_0 \sin \phi + y'_0 \cos \phi) = K_N(x_0 \sin \theta + y_0 \cos \theta)
  \]
  
  where \( \cos \phi = \sin \theta \) and \( \sin \phi = -\cos \theta \)

• If \( \theta \leq -90^\circ \), then:
  \[
  z'_0 = \varphi = \theta + 90^\circ, \\
  x'_0 = y_0, \\
  y'_0 = -x_0
  \]
CORDIC algorithm: rotation mode

CORDIC algorithm precision:
• For M-bit precision in trigonometric functions, M iterations are needed
  – can be demonstrated based on the fact that, for large enough values of k:
    \[ \tan^{-1}(2^{-k}) \approx 2^{-k}, \text{ with } \tan^{-1}(2^{-k}) \text{ always less than } 2^{-k} \]
  – then, if k > M, the further change in z is less than the LSB
CORDIC algorithm: rotation mode

- In rotation mode, the algorithm can be used to:
  - rotate vector \((v_x, v_y)\) of an angle \(\theta\)
    \[
    \Rightarrow \text{load } \theta \text{ in } z_0, \frac{v_x}{K} \text{ in } x_0, \frac{v_y}{K} \text{ in } y_0
    \]
  - compute sine and cosine of an angle \(\theta\)
    \[
    \Rightarrow \text{load } \theta \text{ in } z_0, \frac{1}{K} \text{ in } x_0, 0 \text{ in } y_0
    \]
  - convert from polar \((r, \theta)\) to cartesian \((x, y)\) coordinates
    \[
    \Rightarrow \text{load } \theta \text{ in } z_0, \frac{r}{K} \text{ in } x_0, 0 \text{ in } y_0
    \]
CORDIC algorithm: vectoring mode

• In vectoring mode, the algorithm target is to reduce to zero the $y_k$ coordinate

• CORDIC algorithm – vectoring mode:

\[
\begin{align*}
x_{k+1} &= x_k - y_k d_k 2^{-k} \\
y_{k+1} &= x_k d_k 2^{-k} + y_k \\
z_{k+1} &= z_k - d_k \alpha_k \\
d_k &= -1 \text{ if } y_k > 0, \quad +1 \text{ otherwise}
\end{align*}
\]

1. load initial vector coordinates in $(x_0, y_0)$ and 0 in $z_0$
2. compute $(x_{k+1}, y_{k+1}, z_{k+1})$
3. if $y_{k+1} = 0$ (or $y_{k+1} < \varepsilon$) stop, else go to 2
CORDIC algorithm: vectoring mode

- If $z_0=0$, then $z_N$ contains the angle $\theta$ of the initial vector w.r.t. the $x$-axis:

\[
z_N = z_0 + \theta = z_0 + \tan^{-1}\left(\frac{y_0}{x_0}\right)
\]

- The modulus of the final vector is equal to that of the initial one scaled by $K$:

\[
x_N = K \sqrt{x_0^2 + y_0^2}
\]

\[(x_N = K \cdot M_0, y_N = 0)\]
CORDIC algorithm: vectoring mode

- In vectoring mode, the algorithm can be used to:
  - compute magnitude and angle of a vector \((v_x, v_y)\) (equivalent to cartesian to polar coordinate conversion)
    \[ \Rightarrow \text{load } 0 \text{ in } z_0, \frac{v_x}{K} \text{ in } x_0, \frac{v_y}{K} \text{ in } y_0 \]
  - compute the arctangent function:
    \[ \Rightarrow \text{load } 0 \text{ in } z_0; \text{ the result is } z_N = \tan^{-1}(\frac{y_0}{x_0}) \]
  - compute the arcsine and arccosine functions:
    \[ \Rightarrow \text{load } 0 \text{ in } z_0, \frac{1}{K} \text{ in } x_0, w \text{ in } y_0; \text{ iterate the equations using the modified decision rule:} \]
    \[ d_k = -1 \text{ if } y_k > w, +1 \text{ otherwise;} \]
    \[ \text{the result is } z_N = \sin^{-1}(w) \]
    \[ \Rightarrow \text{similar for arccosine, just load } w \text{ in } x_0, \frac{1}{K} \text{ in } y_0 \text{ and decide rotation direction based on whether } x_k > w \text{ or not} \]
CORDIC algorithm extensions

• Extension to linear functions
  – use $\alpha_k=2^{-k}$ instead of $\alpha_k=\tan^{-1}(2^{-k})$
  – yields $x_0 \cdot y_0$ (rotation mode) or $y_0/x_0$ (vectoring mode)

• Extension to hyperbolic functions
  – use $\alpha_k=\tanh^{-1}(2^{-k})$
  – yields $\cosh(z_0)$ and $\sinh(z_0)$ in rotation mode, $\tanh^{-1}(y_0/x_0)$ in vectoring mode
  – must repeat iterations $k=4, 13, 40, ..., j, 3j+1, ...$ to achieve convergence
CORDIC processor implementation examples

- ALU (datapath) to implement CORDIC equations
  - requires adders, shifters and multiplexers
- A ROM to store the angles $\alpha_k$
- A control unit to manage the iterations based on the decision rule ($d_k=\pm 1$)
CORDIC processor implementation examples

- Datapath computing one iteration per clock cycle
  - N clock cycles required to complete one full operation
  - 3 adders/subtractors, 3 registers, 2 barrel shifters
CORDIC processor implementation examples

- Datapath implementing $n$ iterations in cascade
  - $3 \cdot n$adders/subtractors
  - $3 \cdot n$ registers (optional)
  - fixed shifters (come for free)

- Registers allow pipelining
  - high throughput (one result per clock cycle)
  - $T_{CLK} \approx t_{pCQ} + t_{ADD} + t_{su}$
  - latency: $n$ clock cycles