CORDIC dan sistem Shifting

You're right to point that out! It may seem at first glance that there are multiplications in the CORDIC algorithm, but in fact, **all multiplications in CORDIC are replaced by simple shift and add/subtract operations**. This is the key feature that makes CORDIC so efficient for hardware implementations like FPGAs. Let's break it down to clarify exactly how the algorithm works without multipliers.

Understanding the CORDIC Rotation Equations

In the CORDIC algorithm, the main idea is to rotate a vector by a sequence of angles that are powers of two (e.g., $\arctan(2^{-i})$ for each iteration i). This lets us approximate the rotation operation using only shifts and adds/subtracts.

The CORDIC update equations for each iteration i are:

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

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

where d_i is the direction of rotation (either +1 or -1) depending on whether we need to rotate clockwise or counterclockwise to reduce the remaining angle.

Replacing Multiplication with Shift Operations

In these equations:

- **Multiplication by 2^{-i}** is actually a right shift by i bits.
- For example, $y_i \cdot 2^{-i}$ can be computed as $y_i >> i$ (right shift by i bits).

Thus, the iterative update equations become:

$$x_{i+1} = x_i - d_i \cdot (y_i >> i)$$

$$y_{i+1} = y_i + d_i \cdot (x_i >> i)$$

Example of CORDIC Iteration without Multipliers

Let's go through an example with actual values to illustrate this. Suppose we want to calculate $\sin(\theta)$ and $\cos(\theta)$ for an angle θ using CORDIC:

1. **Initialize**: Set up the initial values:
   - $x_0 = K$, where $K \approx 0.607252$ (in fixed-point representation, this could be stored as a constant).
   - $y_0 = 0$.
   - $z_0 = \theta$ (the target angle we want to approximate).

2. **Iteration Loop**: For each iteration i, perform the following:
    - **Determine the rotation direction d_i**:
        - If z_i (current angle accumulator) is positive, set $d_i = +1$; otherwise, $d_i = -1$.
    - **Update x, y, and z using shifts**:
        - $x_{i+1} = x_i - d_i \cdot (y_i >> i)$
        - $y_{i+1} = y_i + d_i \cdot (x_i >> i)$
        - $z_{i+1} = z_i - d_i \cdot \text{atan}(2^{-i})$
            - The $\text{atan}(2^{-i})$ values are constants that can be precomputed and stored in a ROM or LUT.

    By the end of these iterations, x will approximate $\cos(\theta)$ and y will approximate $\sin(\theta)$.

Why No Multipliers Are Needed

The key point here is that **all instances of "multiplying by a power of two" are implemented as shifts**. Shifting is much simpler than multiplication in hardware:

Why No Multipliers Are Needed

The key point here is that **all instances of "multiplying by a power of two" are implemented as shifts**. Shifting is much simpler than multiplication in hardware:

- **Bit Shifting**: A shift operation $y >> i$ is implemented as wiring in digital logic, which is very fast and uses minimal resources.
- **Adding/Subtracting**: The algorithm only requires additions and subtractions based on the sign d_i, which is efficiently implemented in FPGA hardware.

Scaling Factor K

The vector length changes slightly as each rotation angle (by powers of two) is applied. To counteract this, the final x and y values need to be scaled by a constant factor K, which is the product of all the cosines of the rotation angles used:

$$K = \prod_{i=0}^{n-1} \cos(\text{atan}(2^{-i})) \approx 0.607252$$

This scaling factor K is a single constant multiplier applied at the end (if using fixed-point scaling

from the beginning). In FPGA implementations, this scaling factor can be precomputed and applied just once to achieve the final results for sine and cosine, and it may even be combined into the initialization values to avoid additional multipliers.

Summary

In short, **the CORDIC algorithm avoids multipliers by using only shifts and additions/subtractions**:

- The iterative rotations by angles that are powers of two enable each rotation to be represented as a shift-and-add operation.
- Only bit shifts and additions/subtractions are used within each iteration.
- A single scaling constant K can be pre-applied, eliminating the need for an additional multiplier.

This efficiency makes CORDIC ideal for FPGA implementations of trigonometric functions. Let me know if you'd like more examples, optimizations, or further clarification on any part of this!

Q8.8 dan Kenapa pake Q8.8 lebih mudah untuk shifting

Great question! In fixed-point representation, right bit shifts work similarly to integer right shifts, but you have to be mindful of how you interpret the results to maintain the precision and scaling of the fixed-point values.

Fixed-Point Basics

In fixed-point representation:

- We have a binary point dividing the integer and fractional parts of a number.
- For example, in a **Q8.8** fixed-point format, we use 8 bits for the integer part and 8 bits for the fractional part.
- A value of **Q8.8** fixed-point representation of 3.25 would look like `00000011.01000000` in binary, where `00000011` represents the integer 3, and `01000000` represents the fraction 0.25.

Right Shift in Fixed-Point Arithmetic

When you right-shift a fixed-point number, you're effectively dividing the number by 2 for each shift (just like with integers). However, for fixed-point numbers, this affects the fractional part as well.

Right Shift in Fixed-Point Arithmetic

When you right-shift a fixed-point number, you're effectively dividing the number by 2 for each shift (just like with integers). However, for fixed-point numbers, this affects the fractional part as well.

For example:

1. **Q8.8 Fixed-Point**:
   - Consider `3.25` in Q8.8 format, which is represented as `00000011.01000000` in binary.
   - A right shift by 1 would make it `00000001.10100000`, which represents `1.625` (i.e., $3.25/2$).
2. **Q16.16 Fixed-Point**:
   - In a Q16.16 format, shifting the number right by 1 divides the entire value (integer and fractional) by 2.
   - If we have `32768.0` (represented as `0000000000000001.0000000000000000` in Q16.16), shifting it right by 1 gives `16384.0`, which is half the value.