**CE4518 – Analysis and Simulation of a Rotation-Mode CORDIC Processor**

**A picture containing refrigerator

Description automatically generated**

**Group Members:** Brian Bulfin (16157516), Rory Loughrey (16178041), Micheal O’Connor (16188713)

# **Background**

CORDIC (**C**o-**O**rdinate **R**otation by **DI**gital **C**omputer) is a low-complexity approach to calculating trigonometric, hyperbolic and some other functions using only addition and shift circuitry. CORDIC was conceived in 1956 by Jack E. Volder at the aeroelectronics department of Convair out of necessity to replace the analog resolver in its B-58 bomber's navigation computer. It is very suited to low-cost implementations which do not employ a hardware multiplier. CORDIC has been used for applications in diverse areas such as signal and image processing, communication systems, robotics and 3D graphics apart from general scientific and technical computation

CORDIC can be used to calculate a number of different functions. This project shows how to use CORDIC in rotation mode to calculate the sine and cosine of an angle, assuming that the desired angle is given in radians and represented in a fixed-point format. Consider an angle θ between the angles -90 ≤ θ ≤ 90. By search for it using a successively smaller set of angle steps Δai. Now, once the cosine and sine of θ have been calculated, the cosine and sine of (θ+Δai).

θ = 0° => cos(θ) = 1, sin(θ) = 0

cos(θ+Δai) = cos(θ) cos(Δai) - sin(θ) sin(Δai)

sin(θ+Δai) = cos(θ) cos(Δai) + sin(θ) sin(Δai)

An “undershoot” update is then done when the angle θ is too small, so the angle is updates by adding current angle increment Δai. Sine and cosine are then adjusted appropriately using update formulas. Similarly, the cosine and sine of (θ-Δai) can be found, and the appropriate update can be applied when the current angle is too big giving an overshoot problem.

θ = 0° => cos(θ) = 1, sin(θ) = 0

cos(θ-Δai) = cos(θ) cos(Δai) + sin(θ) sin(Δai)

sin(θ-Δai) = - cos(θ) cos(Δai) + sin(θ) sin(Δai)

However, there is still a problem with lots of arithmetic on each step. It is here that it is considered dividing out the cosine from the equation.

<=

<= cos(Δai)

Now, let tan(Δai) be something simple like 2-I, an i-bit right-shift.

<=

Next, the cosine term must be considered:

tan(Δai) = = =>  cos(Δai) = sin(Δai)

cos(Δai) =

cos2(Δai) = 1 - cos2(Δai)

=> cos2(Δai) =

<=

If n cycles are performed, then:

= …

…

From here there is no expensive per-cycle multiplications left, just bit-shifts and additions.

# **Implementation**

## CORDIC processor in C

The language chosen to implement the CORDIC processor was C. The first step was declaring all the required variables and assigning values when necessary. Every value was scaled up by multiplying it by 2^16 so only integer arithmetic was used. Pre-computed angles were included to reduce calculations. We chose the n cycle approach, so cosine started at K and sine was set to 0. The user was asked to enter an angle which was checked to ensure it was within the appropriate range.

A for loop is used for the calculations which has 14 iterations. The first step is updating the old cosine and sine variables with the most recent cosine and sine values. If the current angle is less than the target angle then the computed angle associated with the current count is added to the current angle. Cosine is itself minus the old sine value shifted to the right the amount of the count bits. Sine is itself plus the old cosine value shifted to the right the amount of the count bits. If the current angle is greater than the target angle then the computed angle is subtracted instead, Cosine is itself plus the old sine value right shifted i bits and sine is itself minus the old cosine value right shifted i bits. The values are printed to the console in decimal form for the sake of legibility.

## CORDIC processor in Verilog

In this project three modules were used to implement and test the CORDIC processor. LUT.v functioned as a look up table, containing all the values of the pre-calculated angles in radians multiplied by 2^16 as binary numbers. For cordic.v the inputs were the target angle, an initialise signal and the clock. The outputs are cosine and sine. The other variables needed were wires and registers. An always block that is sensitive to change in the init signal was used to initialize or reset the variables to their starting values.

Another always block sensitive to the rising edge of the clock was used to perform the calculations in iterations. The init signal and done signal must not have been high for it to have continued. The operations that were then performed were very similar to the C code. The old cosine and sine variables were set the current cosine and sine values. If the current angle was less than the target then the angle based on the count got from the look up tables was added. Cosine was itself minus the old sine right shifted by the count. Sine was itself plus the old cosine right shifted by the count. If the current angle was greater than the target then the angle from the look up tables was subtracted. Cosine was itself plus the old sine right shifted by the count and sine was itself minus the old cosine right shifted by the count. The current angle, cosine value and sine value were displayed every iteration. If the look up table value was 1 then the done signal was set high preventing further calculation. The count was incremented by one each time. Non-blocking assignment was used so all the computations happened in parallel.

The testbench just needed three registers, the target angle, the clock and the init signal and then two wires for cosine and sine. It was connected to the cordic module. All the registers were initialised and the file for viewing the waveforms created. Every ten seconds the clock inverted its value. A target value was selected. After five seconds the init signal was set to 0 changing all variables to their initial state and starting the calculations. After 400 seconds the init signal was set to 1 stopping the calculations if they were not done and resetting the values to their starting point. The target angle was changed by inverting it and then 20 seconds later the init signal was set to 0 again starting the process. After 400 seconds it finished.

# **Results**

A picture containing screen, monitor, clock

Description automatically generated

***Figure 1:*** *Waveform representations of most important CORDIC signals. Angles are represented in hexadecimal.*

Figure 1 illustrates the operation of the CORDIC Verilog module and displays the functionality of the INIT and DONE signals. The performance of our implementation was assessed by analysing its accuracy. First, the C program was used to find the sine and cosine of every possible angle from -π/2 to π/2, which was possible due to the 2.16 fixed point representation used. Each output was then compared to the outputs generated by the standard C math sin and cos functions to find the maximum errors and their associated angles. The program was run for 17 iterations at first, as this is the maximum possible given the 2.16 representation since Δa17 = 2-16, the smallest allowed by 2.16 fixed point.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Cos** | **Sin** | **Combined** |
| **Error** | -0.000173 | -0.000168 | 0.000263 |
| **Angle** | 1.555954 | 0.014847 | -0.014847 |

***Table 1:*** *Max errors and associated angles of the CORDIC C implementation for i < 17.*

Table 1 shows that the maximum errors for both sine and cosine occur near angles of 0 and π/2 respectively, where the outputs of the functions approach 0. At this point, the number of iterations was reduced until the max error was minimised, which occurred at 8 iterations.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Cos** | **Sin** | **Combined** |
| **Error** | 0.000073 | 0.000068 | 0.000131 |
| **Angle** | 1.486771 | 0.084030 | 1.486771 |

***Table 2:*** *Max errors and associated angles of the CORDIC C implementation for i < 8.*

The results presented in table 2 show that the max error dropped significantly by reducing the number of iterations. Once these errors and angles were calculated, the accuracy of the Verilog implementation could be found.

|  |  |  |  |
| --- | --- | --- | --- |
| **Cos(1.486771)** | | **Sin(0.08403)** | |
| **Actual** | 0000010101**0**1111100 | **Actual** | 0000010101**0**1111101 |
| **CORDIC** | 0000010101**1**0000001 | **CORDIC** | 0000010101**1**0000001 |

***Table 3:*** *Binary representations of worst-case cos and sin angles at 8 CORDIC iterations.*

Table 3 demonstrates the implementation is reliably accurate to at least 10 bits, or 8 decimal places. Due to the fixed number of iterations the results can never be guaranteed to be accurate. Even if the correct angle, sine, and cosine are found after a certain number of iterations the algorithm will persist and sacrifice accuracy as a result.

# **Conclusions**

The speed, efficiency, and simplicity of implementation make the rotation-mode CORDIC algorithm a good choice for low-complexity systems such as microcontrollers and FPGA’s without expensive hardware multipliers. However, the fundamental accuracy limitation enforced by the fixed number of iterations means that this algorithm should not be used for high-precision applications, especially as the previously mentioned limitations become less relevant as hardware continues to become cheaper.