Práctica 2
-
Algoritmo CORDIC

David Carrascal Acebron

University of Alcala, 28805, Alcala de Henares (Madrid), Spain.
`david.carrascal@uah.es`

Resumen Este informe presenta un caso práctico donde se abordará el flujo de trabajo habitual en la implementación de un algoritmo en una FPGA. El algoritmo que se implementará será **CO**ordinate **R**otation **DI**gital **C**omputer (CORDIC), el cual fue presentado por Jack E. Volder en 1959 como un método de cálculo en tiempo real de relaciones trigonométricas, en sistemas de navegación aérea [1]. Dicho algoritmo será estudiado de forma preliminar con Matlab para poder plantear sus posteriores implementaciones tanto en VHDL como en HLS.

Keywords: Algoritmo CORDIC · Coma fija y flotante · HLS-VHDL.

1. Introducción

CORDIC (**CO**ordinate **R**otation **DI**gital **C**omputer) es un algoritmo iterativo muy eficiente desde el punto de vista del hardware ya que utiliza rotaciones de vectores para calcular una amplia gama de funciones elementales. En su implementación más básica, solo se requiere de operaciones de suma y de desplazamientos de bits, de ahí que sea tan *hardware friendly* para las FPGAs y todo el hardware reconfigurable, dadas la notoria falta de multiplicadores en estos equipos [2].

Según se ha comentado en el inicio, el algoritmo fue presentado en 1959 por Jack E. Volver para calcular únicamente relaciones trigonométricas en sistemas de navegación de la época, pero no fue hasta el 1972 cuando J. S. Walther generalizó el algoritmo para su uso en el cálculo de multitud de funciones elementales, entre las cuales podemos indicar: multiplicación, división, relaciones trigonométricas, raíces cuadradas, etc [3]. Dichas operaciones elementales pueden ser seleccionadas en función del sistema de coordenadas en el que se trabaje. Entre los distintos sistemas de coordenadas destacan los siguientes realizar el cálculo de diferentes funciones elementales a partir de la relación entre ambos vectores.

- **Linear**, ver figura 1, nos permite calcular la multiplicación y división.

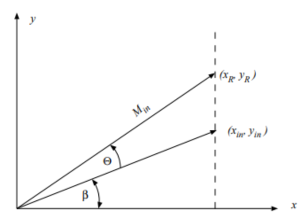


Figura 1. Algoritmo CORDIC empleando rotaciones en un sistema de coordenadas lineales [4].

- **Circular**, ver figura 2, nos permite calcular senos, cosenos, tangentes y arco-tangentes.

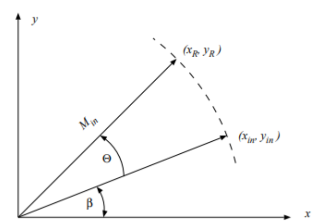


Figura 2. Algoritmo CORDIC empleando rotaciones en un sistema de coordenadas circulares [4].

- **Hiperbólico**, ver figura 3, nos permite calcular senos hiperbólicos, cosenos hiperbólicos, tangentes hiperbólicas, arco-tangentes hiperbólicos, raíces cuadradas y logaritmos.

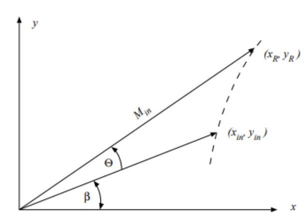


Figura 3. Algoritmo CORDIC empleando rotaciones en un sistema de coordenadas hiperbólicas [4].

Este algoritmo es habitualmente empleado en las FPGAs para resolver numerosas operaciones aritméticas o trigonométricas, debido a la limitación en los recursos internos que existe en estos equipos. Según se comentó anteriormente, principalmente por los escasos multiplicadores en los dispositivos, siendo idóneo encontrar una solución aproximada, que llegó con CORDIC, ofreciendo un equivalente únicamente con sumadores y desplazamientos de bits en un número determinado de iteraciones.

Dicho numero de operaciones, aproximarán más al resultado esperado de la operación en cuestión, pudiendo no llegar a converger, por lo que normalmente se estudia el sistema y se fija un número determinado de iteraciones para no malgastar tiempo de procesamiento en balde. Su funcionamiento se basa en la rotación de un vector bidimensional de entrada para obtener otro vector a la salida.

Cabe destacar que el algoritmo tiene **dos modos de funcionamiento** a la hora de rotar el vector de entrada:

- **Modo Rotación**, se define así, al modo en el cual tras introducir en la coordenada z inicial un ángulo de rotación, la coordenada z_n es forzada a cero. Se puede decir que este modo es aquel en el que un vector inicial es rotado un ángulo determinado, siendo ese ángulo un dato conocido a priori.

- **Modo Vectorización**, mediante este modo se proyecta un vector inicial sobre el eje x, haciendo por tanto cero la coordenada y final ($y_n = 0$), quedando en la z_n el ángulo del vector inicial, ya que inicialmente la componente z_i será nula, y sobre la componente x_n encontraremos el módulo del vector multiplicado por una constante que depende del número de iteraciones del algoritmo. Más adelante se desarrollarán las ecuaciones del algoritmo particularizadas para este modo de funcionamiento.

Siendo el **Modo Vectorización** el que se llevará a cabo por el autor de este informe en la realización de la práctica de acuerdo a la asignación de modos funcionamiento en la diapositiva 29 de la referencia [2].

Volviendo a las ecuaciones del algoritmo CORDIC, podemos expresar las coordenadas finales (x_2, y_2) se pueden expresar en función de las iniciales (x_1, y_1) y del ángulo a rotar (α):

$$\begin{aligned} x_2 &= x_1 cos(\alpha) - y_1 sen(\alpha) \\ y_2 &= x_1 sen(\alpha) + y_1 cos(\alpha) \end{aligned} \tag{1}$$

Indicado la ecuación anterior de una forma genérica, y para reducir el número de multiplicaciones, primeramente se extrae factor común al término coseno, creándose la expresión:

$$\begin{aligned} x_{i+1} &= cos(\alpha_i) \times (x_i - y_i tan(\alpha_i)) \\ y_{i+1} &= cos(\alpha_i) \times (x_i tan(\alpha_i) + y_i) \end{aligned} \tag{2}$$

Las multiplicaciones de tangentes, se pueden convertir en desplazamientos, si se eligen valores de ángulos que hagan que el valor de la tangente sea una **potencia de dos**. De esta forma, se consigue ahorrar recursos internos, al sustituir la multiplicación por un desplazamiento. Por tanto, se deberá conseguir que los valores de los ángulos sean de la forma:

$$tan(\alpha_i) = \sigma^i \times 2^{-i} \leftrightarrow \alpha_i = tan^{-1}(\sigma^i \times 2^{-i}) \tag{3}$$

Si despejamos las tangentes aceptando la condición de los ángulos en potencias de dos, podemos ver como ahora en la expresión de las nuevas coordenadas no aparecen expresiones trigonométricas a excepción de un coseno:

$$\begin{aligned} x_{i+1} &= cos(\alpha_i) \times (x_i - y_i \times \sigma^i \times 2^{-i}) \\ y_{i+1} &= cos(\alpha_i) \times (x_i \times \sigma^i \times 2^{-i} + y_i) \end{aligned} \tag{4}$$

En aras de suprimir este producto, se puede eliminar temporalmente este factor, y multiplicar en las coordenadas finales por el valor final de este factor, a modo de compensación. El valor final del término suprimido, puede ser calculado previamente al ser los ángulos (α_i) prefijados, ya que es el producto de todos los en cada micro- rotación. Por tanto, podemos definir nuestro factor de compensación final como:

$$K_m = \sum_{n-1}^{i=0} cos(\alpha_i) = \sum_{n-1}^{i=0} cos(tan^{-1}(\sigma^i \times 2^{-i})) \tag{5}$$

Por lo tanto, al sacar el coseno que estaba multiplicando en cada micro-rotación para ser multiplicado al final del algoritmo como una constante de compensación, podemos ver como las expresiones de las coordenadas en cada micro-rotación serán las siguientes:

$$\begin{aligned} x_{i+1} &= x_i - y_i \times \sigma^i \times 2^{-i} \\ y_{i+1} &= x_i \times \sigma^i \times 2^{-i} + y_i \end{aligned} \tag{6}$$

Para ir llevando la cuenta de las variaciones del ángulo, se introducirá una nueva variable llamada z, la cual la definiremos de la siguiente forma.

$$z_{i+1} = z_i - \alpha_i = z_i - tan^{-1}(\sigma^i \times 2^{-i}) \tag{7}$$

Una vez que hemos desarrollado todas las ecuaciones en las cuales de basa CORDIC, vamos a expresarlas en función del valor final, es decir después de n micro-rotaciones:

$$\begin{aligned} x_n &= K_m \times (x_i - m \times y_i \times \sigma^i \times 2^{-i}) \\ y_n &= K_m \times (x_i \times \sigma^i \times 2^{-i} + y_i) \\ z_n &= z_i - \alpha \end{aligned} \tag{8}$$

Donde recordemos que K_m es el denominado factor de escalado o compensación que lo podemos expresar como:

$$K_m = \sum_{n-1}^{i=0} (1 + m \times \sigma^{i^2} \times 2^{-2i})^{0,5} \qquad (9)$$

y donde se puede ver que se debe particularizar para el sistema de coordenadas con el cual se va a trabajar. En nuestro caso trabajaremos con el **sistema de coordenadas circulares**. Por tanto, y teniendo en cuenta que trabajaremos con el modo de vectorización podemos indicar los parámetros de las expresiones del algoritmo CORDIC finales son los siguientes.

- $m = 1$, al trabajar con el sistema de coordenadas circulares.
- i es la micro-rotación actual.
- σ^i, al trabajar en modo vectorización, $\sigma^i = -signo(y_i)$

Un aspecto muy importante para el correcto funcionamiento de CORDIC es el valor del ángulo a rotar. Éste debe estar entre $[\frac{-\pi}{2}, \frac{\pi}{2}]$, es decir, en el **primer o cuarto cuadrante**, para que el resultado converja. Esto no implica que el algoritmo no pueda trabajar para ángulos en el segundo o tercer cuadrante, solo habría que hace una corrección de cuadrante al inicio y al final del algoritmo CORDIC para que el ángulo inicial se vea desplazado o al primer cuadrante o al cuarto.

Obviamente el mover el vector de entrada al primer/cuarto cuadrante implica una variación en el valor de salida que habrá que corregir, por ello se habrá que realizar un post-procesamiento al vector de salida del módulo CORDIC.

Dichas correcciones sobre el cuadrante se pueden resumir de la siguiente forma. De forma matemática, si el vector se encuentra inicialmente en el segundo cuadrante, los cambios a realizar son:

$$\begin{aligned} x'_1 &= y_1 \\ y'_1 &= -x_1 \\ z'_1 &= z_1 + \frac{\pi}{2} \end{aligned} \qquad (10)$$

Mientras que si el vector de entrada se encuentra en el tercer cuadrante:

$$\begin{aligned} x'_1 &= -y_1 \\ y'_1 &= x_1 \\ z'_1 &= z_1 - \frac{\pi}{2} \end{aligned} \qquad (11)$$

Por último, solo se quiere dejar indicado en la introducción que en el modo vectorización terminaremos proyectando el vector sobre el eje x, por lo que tendremos que la componente y_n será 0 despues de n iteraciones, y que en x_n tendremos el modulo del vector multiplicado por la constante de compensación, al proyectar sobre el eje. Finalmente, según la definimos anteriormente, en la componente z encontraremos la suma de la fase del vector, teniendo finalmmente las siguientes expresiones a la salida.

$$\begin{aligned} x_n &= K_m \times \sqrt{(x_1^2 + y_1^2)} \\ y_n &= 0 \\ z_n &= z_1 + tan^{-1}(\frac{y_1}{x_1}) \end{aligned} \tag{12}$$

El resto de este informe estará estructurado de la siguiente manera. En la Sección 2 se indicará el estudio realizado en Matlab del algoritmo, con sus aspectos fundamentales y las implementaciones realizadas del mismo. En la Sección 3, se abordará la implementación en HLS del algoritmo y se comparará con los resultados de Matlab, así como posibles optimizaciones. En la Sección 4, se implementará el algoritmo en VHDL y se realizará una simulación del mismo. Por último, la Sección 5 finaliza el informe con algunas observaciones finales.

2. Simulación de CORDIC en Matlab

2.1. Coma flotante

A continuación se indica el código Matlab generado para la implementación del algoritmo de CORDIC para coma flotante. Dicha función se ha desarrollado en aras de comprobar la funcionalidad lógica del mismo y para hacer una aproximación sobre el número de iteraciones.

```
%% Practica de CORDIC - MATLAB - Coma flotante

% En mi caso tengo que trabajar con el modo de vectorizacion

function [x_fin,y_fin,z_fin] = cordic_float(x, y, z, n_iteraciones)

    % -- Vars aux --
    % Sistema de coordenadas circular
    m = 1;

    % Calculo previo de alpha
    angulos_totales = atan(2.^-(0:n_iteraciones-1));

    % Calculo previo de factor de compensación
    K = 1;
    for j=0:1:n_iteraciones
        K = K * (sqrt(1+2^(-2*j)));
```

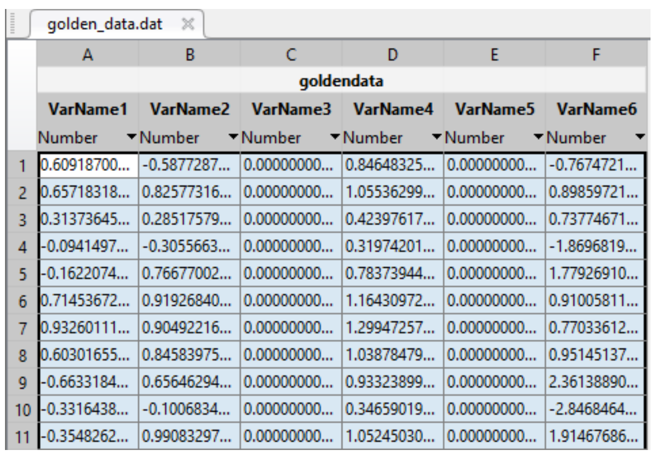
```
    end

    % Pre-procesado cuadrante
    if ( sign(x) < 0 )
        if ( sign(y) > 0 )
            % 2o cuadrante
            x_cor = y;
            y_cor = -1*x;
            z_cor = z + pi/2;
        else
            %3er cuadrante
            x_cor = -1*y;
            y_cor = x;
            z_cor = z - pi/2;
        end
    else
        x_cor = x;
        y_cor = y;
        z_cor = z;
    end

    % Main loop
    for i = (0:n_iteraciones-1)
        sigma = ((-1)*sign(y_cor));
        x_fin = x_cor - (m * sigma * 2^(-i) * y_cor);
        y_fin = y_cor + (sigma * 2^(-i) * x_cor);
        z_fin = z_cor - (sigma * angulos_totales(i + 1));

        % Shift
        x_cor = x_fin;
        y_cor = y_fin;
        z_cor = z_fin;

    end

    %Ajustamos con el factor de compesación
    y_fin = y_fin/K;
    x_fin = x_fin/K;
    z_fin = z_fin;
end
```

Para poder estudiar el analizar la viabilidad de la función desarrollada se tuvo que generar un *dataset* con los resultados esperados y ver como de lejos estábamos de estos. En esta primera prueba prueba se ha diseñado un dataset de forma manual según se nos ha indicado en clase, con una serie de valores conocidos y así poder comparar con los valores esperados. Dicho dataset para no ser muy pesado de generar, se ha empleado un pequeña función de Matlab que se lista a continuación.

```
%% Practica de CORDIC - MATLAB
function golden_dataset( len, filename)

    %Abrimos el fichero
    fid = fopen(filename,'w');


    for i = 1:1:len

        % El signo no puede caer en cero justo jaja
        signo = randi([-1,1],1,1);
        if signo == 0
            signo = 1;
        end

        x = signo*rand(1);
        signo = randi([-1,1],1,1);
        if signo == 0
            signo = 1;
        end

        y = signo*rand(1);
        z = 0;

        mod = sqrt(x^2+y^2);
        fase = atan(y/x);

        if(x < 0 && y > 0)
         fase = fase + pi;
        end

        if(x < 0 && y < 0)
         fase = fase - pi;
        end
```

```
        fprintf(fid,'%.16f\t%.16f\t%.16f\t%.16f\t
                    %.16f\t%.16f\n',x,y,z, mod,0,fase);
    end

    %Cerramos el fichero
    fclose(fid);


end
```

El dataset generado se ha llamado **golden_data.dat** y nos servirá en adelante para ver como de cerca estamos del valor esperado. En la figura 4, se puede apreciar como en la primeras tres columnas se encuentra el vector de entrada, y en las tres columnas de la derecha se encuentran los valores esperados una vez se ha ejecutado el algoritmo.



golden_data.dat

	A	B	C	D	E	F
	goldendata					
	VarName1	VarName2	VarName3	VarName4	VarName5	VarName6
	Number	Number	Number	Number	Number	Number
1	0.60918700...	-0.5877287...	0.00000000...	0.84648325...	0.00000000...	-0.7674721...
2	0.65718318...	0.82577316...	0.00000000...	1.05536299...	0.00000000...	0.89859721...
3	0.31373645...	0.28517579...	0.00000000...	0.42397617...	0.00000000...	0.73774671...
4	-0.0941497...	-0.3055663...	0.00000000...	0.31974201...	0.00000000...	-1.8696819...
5	-0.1622074...	0.76677002...	0.00000000...	0.78373944...	0.00000000...	1.77926910...
6	0.71453672...	0.91926840...	0.00000000...	1.16430972...	0.00000000...	0.91005811...
7	0.93260111...	0.90492216...	0.00000000...	1.29947257...	0.00000000...	0.77033612...
8	0.60301655...	0.84583975...	0.00000000...	1.03878479...	0.00000000...	0.95145137...
9	-0.6633184...	0.65646294...	0.00000000...	0.93323899...	0.00000000...	2.36138890...
10	-0.3316438...	-0.1006834...	0.00000000...	0.34659019...	0.00000000...	-2.8468464...
11	-0.3548262...	0.99083297...	0.00000000...	1.05245030...	0.00000000...	1.91467686...

Figura 4. Dataset generado para la verificación del algoritmo.

2.2. Coma fija

A continuación, se indica el código para el algoritmo de CORDIC en como fija. Se ha empleado 16bits de ancho y una 4QN. Para poder comparar posteriormente, se han ejecutado las pruebas para el numero total de entradas en el golden data para así poder comparar posteriormente.

```
%% Practica de CORDIC - MATLAB - Coma fija

% En mi caso tengo que trabajar con el modo de vectorizacion del algoritmo

function [x_fin,y_fin,z_fin] = cordic_fixed(x, y, z,
n_iteraciones, num_bit, redondeo)

    % Pre-procesado cuadrante
    if ( sign(x) < 0 )
        if ( sign(y) > 0 )
            x_cor = fi(y, 1, num_bit, num_bit - 2,
            'RoundingMethod', redondeo);
            y_cor = fi((-1)*x, 1, num_bit, num_bit - 2,
            'RoundingMethod', redondeo);
            z_cor = fi(z + pi/2, 1, num_bit, num_bit - 3,
            'RoundingMethod', redondeo);
        else
            x_cor = fi((-1)*y, 1, num_bit, num_bit - 2,
            'RoundingMethod', redondeo);
            y_cor = fi(x, 1, num_bit, num_bit - 2,
            'RoundingMethod', redondeo);
            z_cor = fi(z - pi/2, 1, num_bit, num_bit - 3,
            'RoundingMethod', redondeo);
        end
    else
        x_cor = fi(x, 1, num_bit, num_bit - 2, 'RoundingMethod', redondeo);
        y_cor = fi(y, 1, num_bit, num_bit - 2, 'RoundingMethod', redondeo);
        z_cor = fi(z, 1, num_bit, num_bit - 3, 'RoundingMethod', redondeo);
    end



    m = fi(1, 0, 1, 0);
    K_m_t = 1;
    angulos_totales = fi(atan(2.^-(0:n_iteraciones-1)), 1, num_bit,
    num_bit - 3, 'RoundingMethod', redondeo);

    % Main loop :)
    for i = (0:n_iteraciones-1)

```

```
        sigma = fi((-1)*sign(y_cor), 1, 2, 0);
        fi(2^(-i), 1, num_bit, num_bit - 2, 'RoundingMethod', redondeo);

        x_fin = fi(x_cor - (m * sigma * fi(2^(-i), 1, num_bit, num_bit - 2,
        'RoundingMethod', redondeo) * y_cor),1, num_bit, num_bit - 2,
        'RoundingMethod', redondeo);
        y_fin = fi(y_cor + (sigma * fi(2^(-i), 1, num_bit, num_bit - 2,
        'RoundingMethod', redondeo) * x_cor), 1, num_bit, num_bit - 2,
        'RoundingMethod', redondeo);
        z_fin = fi(z_cor - (sigma * angulos_totales(i + 1)), 1, num_bit,
        num_bit - 3, 'RoundingMethod', redondeo);


        potencia=fi(2^(-2*i), 1, num_bit, num_bit - 2,
        'RoundingMethod', redondeo);

        K_m = fi(sqrt(1+(m* (sigma^2) *potencia)), 1, num_bit,
        num_bit - 2, 'RoundingMethod', redondeo);
        K_m_t = fi(K_m_t * K_m, 1, num_bit, num_bit - 2,
        'RoundingMethod', redondeo);

        x_cor = x_fin;
        y_cor = y_fin;
        z_cor = z_fin;

    end

    % Ajustamos con el factor de compesación
    x_fin = fi((1/K_m_t) * x_fin,1, num_bit, num_bit - 2,
    'RoundingMethod', redondeo);
    y_fin = fi((1/K_m_t) * y_fin,1, num_bit, num_bit - 2,
    'RoundingMethod', redondeo);
    z_fin = z_fin;

end
```

2.3. Resultados `n_iteraciones` y en función de bit de entrada

Para realizar este apartado se ha diseñado un script[1], el cual no se añade en la memoria por su extensión pero se deja un link al repositorio de github donde

[1] https://github.com/davidcawork/CORDIC/tree/main/src

se encuentra. Dicho script probará el algoritmo CORDIC con coma fija y coma flotante, para obtener de ellos el error cuadrático y máximo cometido por las diferentes combinaciones de **número de bits** y **número de iteraciones** del algoritmo. A continuación se indican todos los resultados obtenidos.

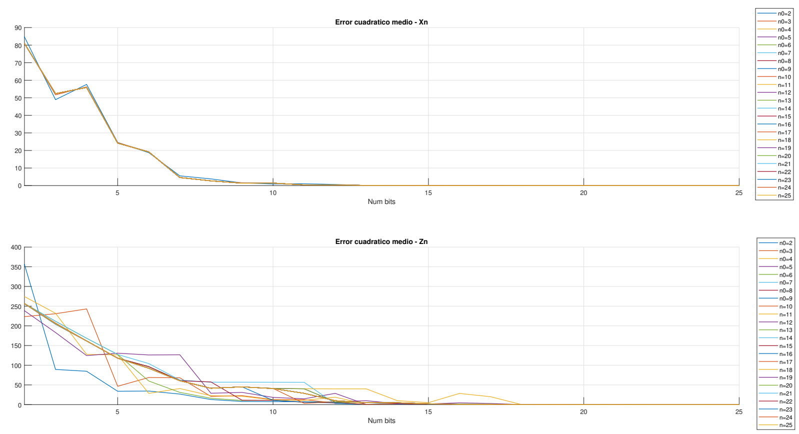


Figura 5. Estudio de la variación de las iteraciones respecto al número de bits del algoritmo con el error cuadrático medio.

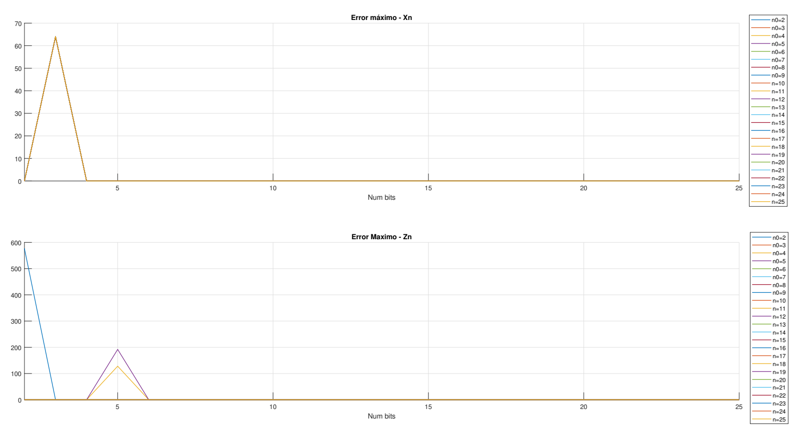


Figura 6. Estudio de la variación de las iteraciones respecto al número de bits del algoritmo con el error máximo.

Como se puede apreciar, a medida que aumentamos el número de bits, aumenta la precisión del algoritmo CORDIC así como el número de iteraciones. Analizando las muestras, se ha llegado a la decisión de emplear como **número suficiente de bits 20**, y **número de iteraciones 18**, ya que se considera de acuerdo se preguntó al profesor que el error cometido es lo suficientemente pequeño. Para una mejor visualización de la solución adoptada, se indica en la figura 7 y 8, la configuración elegida.

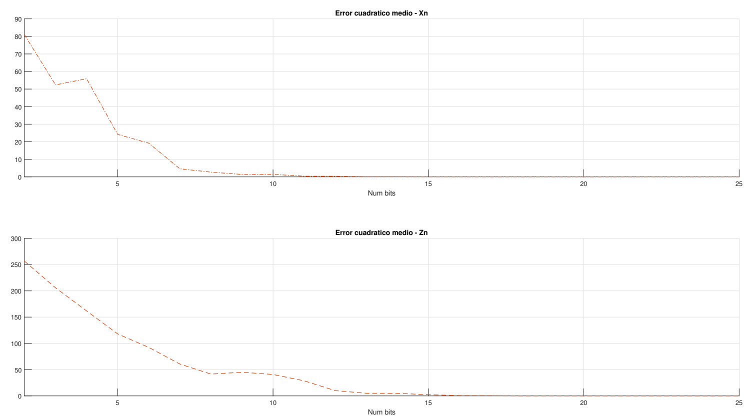


Figura 7. Estudio del error cuadrático medio para $n = 18$ iteraciones.

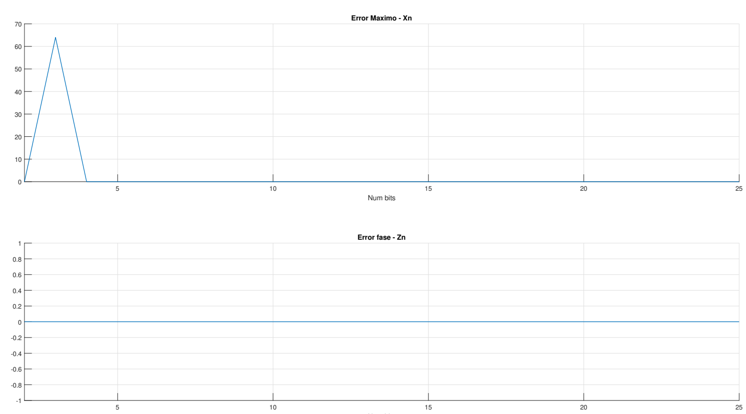


Figura 8. Estudio del error máximo para $n = 18$ iteraciones.

3. Simulación de CORDIC en HLS

En esta sección se debatirá y se mostrará como se ha implementado el algoritmo de CORDIC en su versión secuenciada, ya que hay que desarrollarlo en C, aunque para poder utilizar la librería de coma fija emplearemos C++. En aras de comprobar su funcionamiento, se simulará y se compararán con los datos obtenidos anteriormente. Para la implementación del sistema se han utilizado los siguientes ficheros:

3.1. Fichero de cabecera cordic_float.h

A continuación se indica el fichero de cabecera para la simulación en HLS:

```
#include <ap_fixed.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define PI 3.14159265358979323846

const int Tam_total = 18;
const int K = 20;
const int m = 1;
typedef ap_fixed<Tam_total, 2, AP_RND_CONV, AP_SAT> modulo;
typedef ap_fixed<Tam_total, 3, AP_RND_CONV,AP_SAT> fase;
const modulo PI2 = PI/2;

const fase angulos[20]={0.785398163397448,0.463647609000806,
                       0.244978663126864,0.124354994546761,
                           0.0624188099959574,0.0312398334302683,
                           0.0156237286204768,0.00781234106010111,
                           0.00390623013196697,0.00195312251647882,
                           0.000976562189559320,
                           0.000488281211194898,0.000244140620149362,
                           0.000122070311893670,
                           6.10351561742088e-05,3.05175781155261e-05,
                           1.52587890613158e-05,
                           7.62939453110197e-06,3.81469726560650e-06,
                           1.90734863281019e-06};

void Cordic(modulo x, modulo y, fase z, modulo * x_out,
modulo * y_out, fase * z_out);
```

3.2. Fichero de simulación `testbench.cpp`

A continuación, se indica el fichero de simulación del algoritmo CORDIC, todos los datos se loggean a un fichero para su posterior análisis.

```
#include "cordic_float.h"
#include <math.h>
#include <iostream>
#include <string>

int main(){
        /*variables aux*/
        modulo x_fin=0,y_fin=0;
        modulo aux;
        fase z_fin=0;
        FILE * fd_read, *fd_write;
        char datos2file[50];
        double x_file, y_file, z_file;


        fd_read = fopen("golden_data.dat","r+");
        if (fd_read==NULL){
                printf ("[ERROR] File cannot be openned\n");
                exit (1);
        }

        fd_write = fopen("dataout_hls.dat","w+");
        if (fd_write == NULL)
        {
                printf("[ERROR] Cannot create the file\n");
                exit(1);
        }

        // main loop
        while( fscanf(fd_read, "%lf\t%lf\t%lf\n", &x_file,
        &y_file, &z_file) != EOF )
        {
                Cordic(x_file, y_file, z_file, &x_fin, &y_fin, &z_fin);
                printf("%.10f\t %.10f\n\r",double((x_fin)),
                double((z_fin)));
                sprintf(datos2file,"%.10f\t%.10f\n\r",
                double(x_fin),double(z_fin));
```

```
                fwrite(datos2file, sizeof(char), sizeof(datos2file),
                fd_write);
        }

        // close files
        fclose(fd_read);
        fclose(fd_write);

}
```

3.3. Fichero de fuente `cordic_float.cpp`

A continuación, se indica el fichero de fuente del algoritmo CORDIC, como se indicó al principio de la sección se ha utilizado una arquitectura secuencial.

```
#include "cordic_float.h"

void Cordic(modulo x, modulo y, fase z, modulo * x_out,
            modulo * y_out, fase * z_out){

        /* vars aux*/
        modulo x_prev = 0, y_prev = 0, x_i = 0, y_i = 0,
        x_less = 0, y_less = 0;
        fase z_prev = 0, z_i = 0;
        modulo K_m_t = 1;
        double raiz = 0;
        int signo = 0;

        /* Pre-procesado*/
        if( x < 0 && y > 0 )
        {
                /*2o*/
                x_prev = y;
                y_prev = (-1)*x;
                z_prev = z + PI2;
        }
        else if( x < 0 && y < 0 )
        {
                /*3er*/
                x_prev = (-1)*y;
```

```
                y_prev = x;
                z_prev = z - PI2;
        }
        else
        {
                /*1 - 4 quad*/
                x_prev = x;
                y_prev = y;
                z_prev = z;
        }

    cordic:for (int i = 0; i < K; i++)
    {

                if(y_prev < 0)
                        signo = 1;
                else if (y_prev > 0)
                        signo = -1;
                else
                        signo = 0;


                x_less=x_prev>>(i);
                y_less=y_prev>>(i);
                x_i = x_prev - ((signo)*y_less);
                y_i = y_prev + ((signo)*x_less);
                z_i = z_prev - ((signo)*angulos[i]);
                x_prev = x_i;
                y_prev = y_i;
                z_prev = z_i;

                raiz = sqrt(double(1 + (m * (signo^2) * 2^(-2*i))));
                K_m_t = K_m_t * (modulo)raiz;
    }

    /*Factor de compensación de K*/
    *x_out = x_i/K_m_t;
    *y_out = y_i/K_m_t;
    *z_out = z_i;
}
```

3.4. Diferencias HLS vs Matlab

Como se ha indicado se han sacado todos los resultados a un fichero para poder comparar con los resultados vistos en Matlab. De esta forma podemos ver como de buena o mala es la implementación en HLS respecto la realizada en Matlab, tanto en como fija como en como flotante.

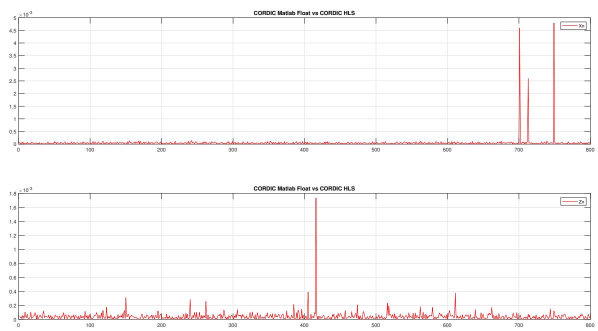


Figura 9. Estudio de las diferencias entre la implementación de Matlab en coma flotante y HLS.

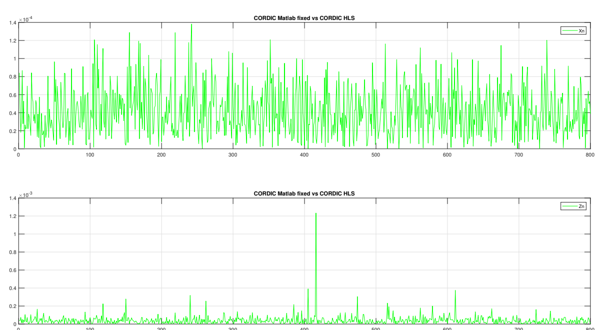


Figura 10. Estudio de las diferencias entre la implementación de Matlab en coma fija y HLS.

Como se puede las diferencias son mínimas teniendo diferencias del orden de 10^{-4}. Por lo que se da por valida la implementación de HLS.

3.5. Directivas

Una vez hemos comprobado el funcionamiento de nuestro algoritmo, vamos a introducir directivas para optimizar la ejecución del mismo.

Sin directivas En prime lugar, vamos a comprobar las características del algoritmo sin establecer ninguna directiva para optimizar su ejecución. Además podemos comprobar los recursos consumidos:

Latency

Unroll

Pipelined

4. Implementación de CORDIC en VHDL

No ha dado tiempo a completar esta parte de la memoria.. Se adjuntan los ficheros asociados en la entrega.

5. Conclusiones

En este informe se ha presentado el algoritmo CORDIC, se han visto las distintas implementaciones del algoritmo, empezando en la fase de análisis con Matlab, una de optimización pasando por HLS hasta llegar al diseño en hardware con VHDL. Los errores obtenidos entre las distintas plataformas se ha visto que pueden variar ligeramente, aproximadamente entorno a la milésima o décima de esta. Por lo tanto, se puede afirmar que los resultados obtenidos son satisfactorios según lo comentado en clase, donde además se ha puesto de manifiesto todas las etapas de desarrollo de un proyecto enfocado a dispositivos hardware reconfigurable.

Referencias

1. J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on electronic computers*, no. 3, pp. 330–334, 1959.
2. I. B. Muñoz, "Practica 2: Algoritmo cordic v1.3," *Diseño Electrónico para Comunicaciones*, 2021.
3. J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, spring joint computer conference*, 1971, pp. 379–385.
4. M. D. Ercegovac and T. Lang, "Chapter 11 - cordic algorithm and implementations," in *Digital Arithmetic*, ser. The Morgan Kaufmann Series in Computer Architecture and Design, M. D. Ercegovac and T. Lang, Eds. San Francisco: Morgan Kaufmann, 2004, pp. 608–648. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558607989500130

Synthesis Report for 'Cordic'

General Information

Date:	Mon Dec 20 23:52:05 2021
Version:	2017.4 (Build 2086221 on Fri Dec 15 21:13:3
Project:	Cordic_hls
Solution:	solution1
Product family:	zynq
Target device:	xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.12	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
940	940	940	940	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	2	0	4181
FIFO	-	-	-	-
Instance	-	0	3080	3333
Memory	0	-	15	5
Multiplexer	-	-	-	449
Register	-	-	1015	-
Total	0	4	4110	7968
Available	280	220	106400	53200
Utilization (%)	0	1	3	14

Detail

Figura 11. Estudio de recursos sin directivas en HLS.

Synthesis Report for 'Cordic'

General Information

Date:	Mon Dec 20 23:07:35 2021
Version:	2017.4 (Build 2086221 on Fri Dec 15 21:13:3
Project:	Cordic_hls
Solution:	solution1
Product family:	zynq
Target device:	xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.12	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
940	940	940	940	none

Detail

Instance

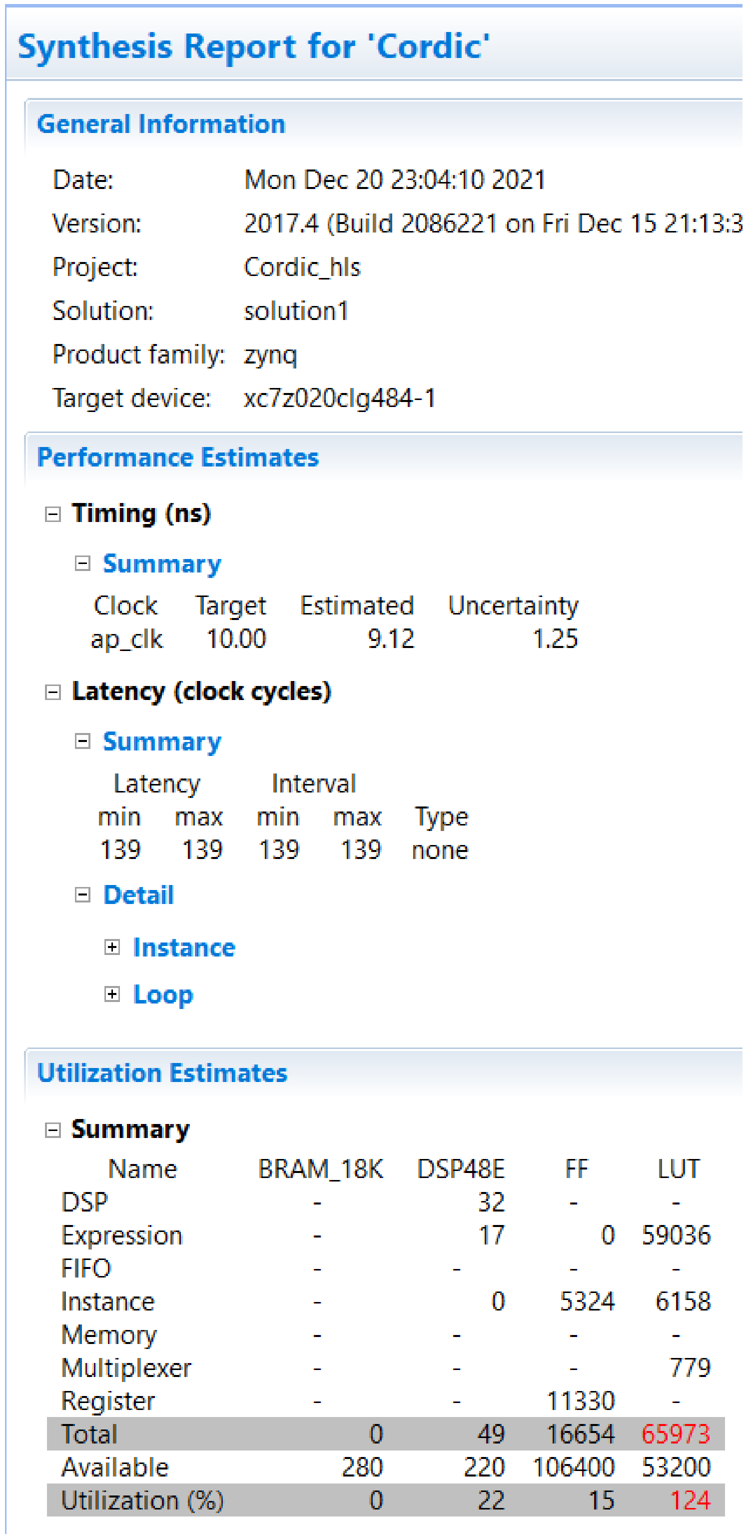
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	2	0	4181
FIFO	-	-	-	-
Instance	-	0	3080	3333
Memory	0	-	15	5
Multiplexer	-	-	-	449
Register	-	-	1015	-
Total	0	4	4110	7968
Available	280	220	106400	53200
Utilization (%)	0	1	3	14

Figura 12. Estudio de recursos con la directiva latency en HLS.



Synthesis Report for 'Cordic'

General Information

Date:	Mon Dec 20 23:04:10 2021
Version:	2017.4 (Build 2086221 on Fri Dec 15 21:13:3
Project:	Cordic_hls
Solution:	solution1
Product family:	zynq
Target device:	xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.12	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
139	139	139	139	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	32	-	-
Expression	-	17	0	59036
FIFO	-	-	-	-
Instance	-	0	5324	6158
Memory	-	-	-	-
Multiplexer	-	-	-	779
Register	-	-	11330	-
Total	0	49	16654	65973
Available	280	220	106400	53200
Utilization (%)	0	22	15	124

Figura 13. Estudio de recursos con la directiva unroll en HLS.

Synthesis Report for 'Cordic'

General Information

Date:	Mon Dec 20 23:47:17 2021
Version:	2017.4 (Build 2086221 on Fri Dec 15 21:13:3
Project:	Cordic_hls
Solution:	solution1
Product family:	zynq
Target device:	xc7z020clg484-1

Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	14.54	1.25

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
103	103	103	103	none

⊟ Detail

⊞ Instance

⊞ Loop

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	2	0	4197
FIFO	-	-	-	-
Instance	-	0	3080	3333
Memory	0	-	15	5
Multiplexer	-	-	-	324
Register	0	-	1341	320
Total	0	4	4436	8179
Available	280	220	106400	53200
Utilization (%)	0	1	4	15

⊟ Detail

Figura 14. Estudio de recursos con la directiva pipelined en HLS.