Circuitos Lógicos Programables
Carrera de Especialización en Sistemas Embebidos

CORDIC

Alumno:

Nombre	Teléfono	Mail
Meoli, Lucas Pablo	+54 9 11 23194140	meolilucas@gmail

11 de octubre de 2024

Índice

1 Introducción

A lo largo de este trabajo práctico se desarrollará, simulará e implementará una arquitectura CORDIC iterativa mediante la descripción de un circuito combinacional y secuencial, aplicando el lenguaje de descripción de hardware VHDL.

2 CORDIC

CORDIC (COordinate Rotation DIgital Computer) es un algoritmo iterativo que se utiliza para realizar cálculos matemáticos de funciones trigonométricas mediante la rotación de vectores. Fue desarrollado en 1959 por Jack E. Volder para simplificar la implementación de funciones en sistemas electrónicos, eliminando la necesidad de multiplicadores costosos en hardware.

El algoritmo se basa en una serie de rotaciones vectoriales o escalamientos mediante ángulos predefinidos, utilizando operaciones de suma, resta y desplazamientos de bits, lo que permite su implementación eficiente en hardware.

El algoritmo original, tal como fue propuesto en su forma inicial, describe la rotación de un vector en un plano bidimensional cartesiano. Su operación se basa en la fórmula general de rotación de vectores, que se presenta a continuación

$$x' = x\cos\theta - y\,\mathrm{sen}\,\theta$$
$$y' = y\cos\theta + x\,\mathrm{sen}\,\theta$$

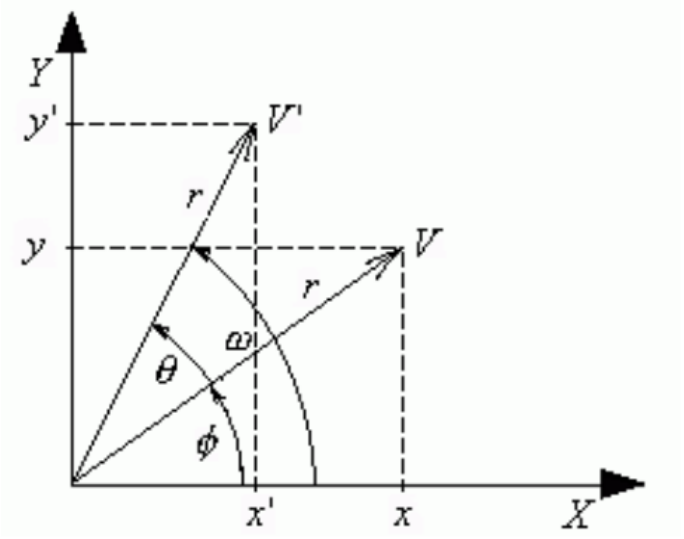


Figura 2.1: Ecuaciones rotación.

A partir del desarrollo matemático, y considerando rotaciones con ángulos de rotación en dirección tangente, se puede derivar un algoritmo recursivo que, tras un cierto número de iteraciones, permite aproximar el vector rotado con un error predefinido. Esto se logra restringiendo los ángulos de rotación de manera que $tg\theta = 2^{-i}$, lo que convierte la multiplicación por la tangente en una simple operación de desplazamiento de bits. De este modo, se obtienen las siguientes ecuaciones

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

Estas ecuaciones describen el algoritmo CORDIC y, como se mencionó anteriormente, dan lugar a un algoritmo recursivo en el que solo se utilizan operaciones de suma, resta y desplazamiento de bits.

3 Implementación

En primer lugar se prosiguió a realizar la implementación principal del algoritmo de CORDIC, es decir la unidad aritmética lógica básica. Para ello se implementó el siguiente circuito.

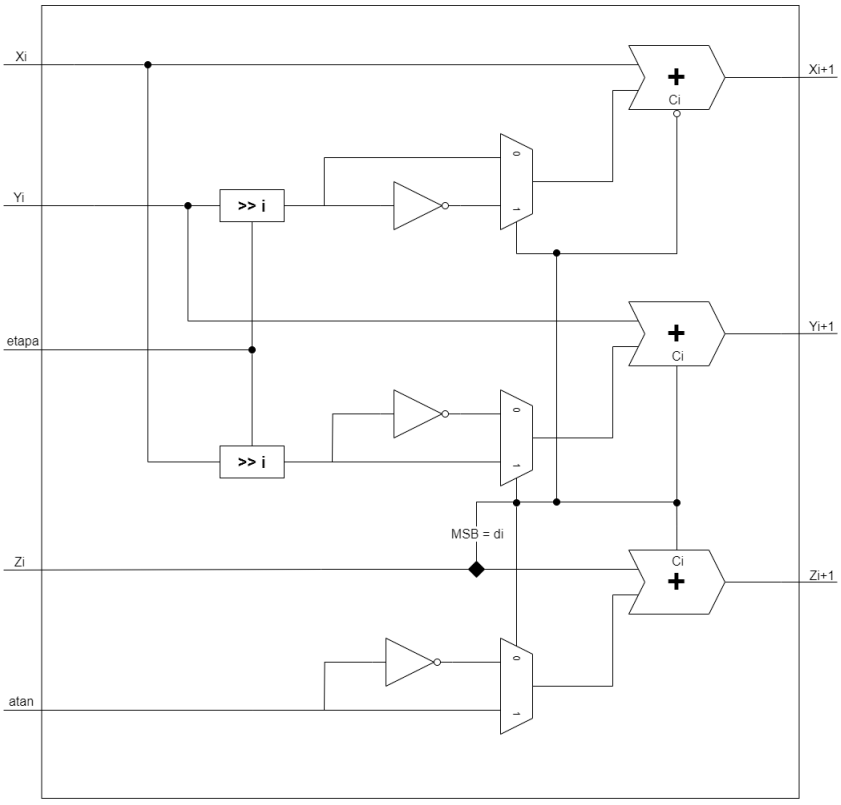


Figura 3.1: Diagrama implementación CORDIC.

En primer lugar, para calcular el valor de la coordenada X_{i+1}, se implementó un circuito que suma el valor de la entrada X_i con la entrada Y_i desplazada una cantidad de veces igual

al número de la etapa en la que se encuentra, conservando el signo. Además, el valor de Y_i puede estar invertido o no, dependiendo del signo del ángulo de entrada. El mismo circuito se implementó para obtener la coordenada Y_{i+1}, con la diferencia de que se invierten los roles en el sumador, y la inversión de la coordenada Y_i es opuesta a la de X_i; es decir, si una se invierte, la otra no.

En segundo lugar, se implementó un circuito capaz de calcular el valor del ángulo Z_{i+1}, el cual depende de la suma entre la entrada Z_i y la señal *atan*, que corresponde al arco tangente del ángulo asociado a la etapa. Esta señal puede estar invertida o no, dependiendo del signo del ángulo. De esta forma, se obtiene el bloque principal para implementar el algoritmo de CORDIC.

Posteriormente, se decidió implementar una arquitectura iterativa con el propósito de reducir la cantidad de bloques de hardware necesarios. Para esta implementación, se procedió a diseñar el siguiente circuito, como se muestra en el diagrama de bloques.

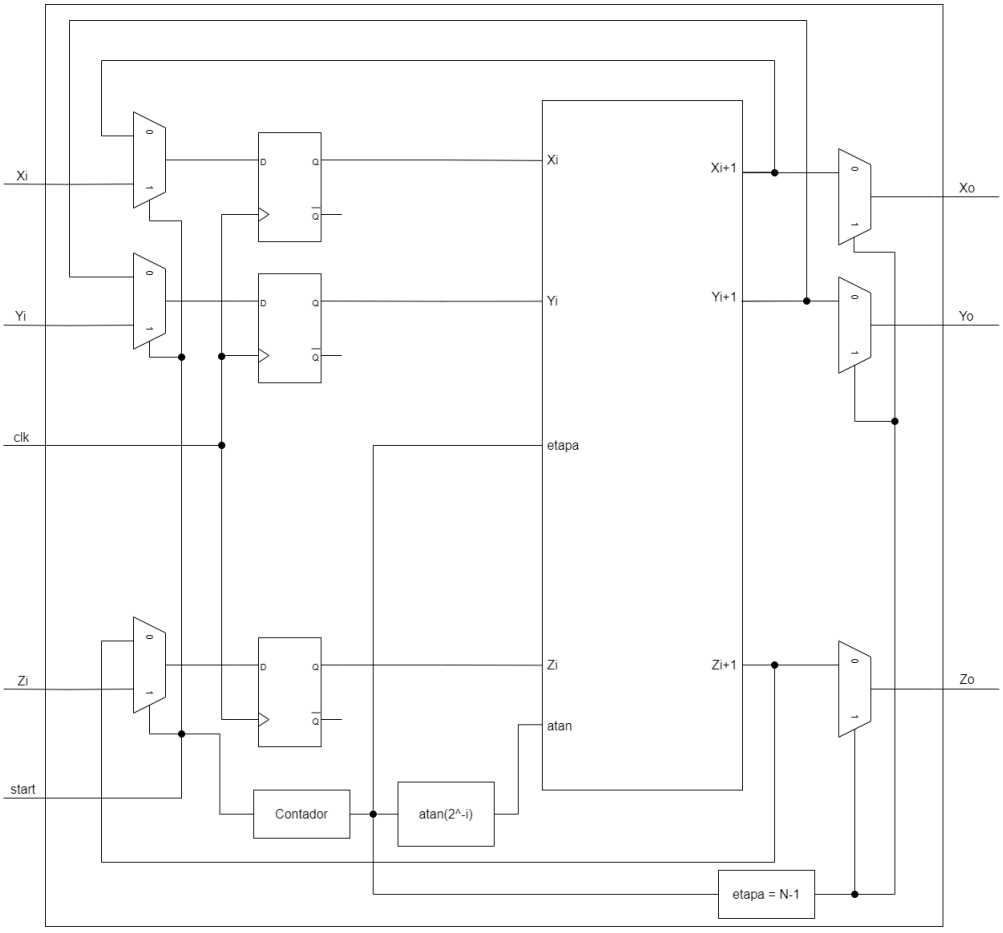


Figura 3.2: Diagrama arquitectura iterativa.

Para la implementación de la arquitectura iterativa, se utilizó el mismo bloque CORDIC explicado anteriormente. La idea principal consistió en conectar las entradas X_i, Y_i y Z_i del bloque a un multiplexor controlado por una señal de start. La salida del multiplexor está conectada a la entrada del bloque CORDIC, de modo que este multiplexor proporciona una realimentación

cuando la señal start está en estado bajo. En estado alto, la entrada del multiplexor proviene directamente de las entradas. Sin embargo, la implementación final no siguió exactamente este enfoque, ya que se diseñó una máquina de estados. A partir de cada flanco ascendente de la señal start, la máquina se activa y pasa por los estados IDLE, COUNTING y DONE. De este modo, una vez iniciada la rotación, la máquina entra en el estado COUNTING y permanece en él hasta que se complete la rotación. Al finalizar el procesamiento, un nuevo flanco ascendente en la señal start dará comienzo a una nueva rotación.

Por otra parte, entre el multiplexor y el bloque CORDIC se encuentra un flip-flop que actualiza las entradas del CORDIC después de cada ciclo de reloj. Además, se implementó un bloque contador que controla tanto las iteraciones como la cantidad de posiciones a desplazar en las entradas. Una vez que el contador alcanza el valor prefijado de iteraciones (15 en este caso, dado que se utilizan 16 bits de precisión), activa un multiplexor que habilita la salida del sistema. Este contador también se utiliza para seleccionar los valores correspondientes de la tabla predefinida de arco tangentes.

Por último, se prosiguió a calcular el precordic, el cual se trata de un procesamiento de las señales de entrada. Esto se debe implementar ya que el algoritmo de CORDIC solo funciona aproximadamente para rotar ángulos entre 90° y -90°, por lo tanto si desea poder rotar 360° se debe agregar una etapa de pre procesamiento. Para ello, se implemento un circuito capaz de detectar en que cuadrante se encontraba el ángulo de rotación y en caso de estar entre 90° y 180° o entre -90° y -180°, el precordic se encarga de complementar los valores de la coordenadas x e y, es decir transformarlas en $-x$ y $-y$. Luego, dependiendo entre que valores se encuentre el ángulo, el precordic se encarga de restarle o de sumarle 180°. De esta forma se vuelve a obtener una operación en la cual el CORDIC únicamente tiene que rotar entre 90° y -90°.

En esta implementación del algoritmo CORDIC no se aplicó la corrección correspondiente al factor de escalamiento. Como resultado, el vector obtenido tras la rotación será aproximadamente 1.647 veces mayor que el valor teórico esperado, debido a la acumulación del factor de escala inherente al algoritmo.

4 Simulaciones

Una vez implementados los circuitos, se procedió a realizar las simulaciones correspondientes para verificar el funcionamiento del algoritmo. En las figuras 4.1, 4.2 y 4.3 se muestran algunas de las simulaciones realizadas.

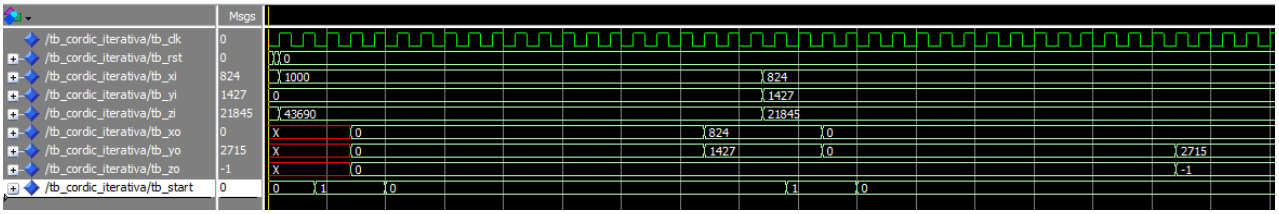


Figura 4.1: Simulación de rotación de 60° y 30°.

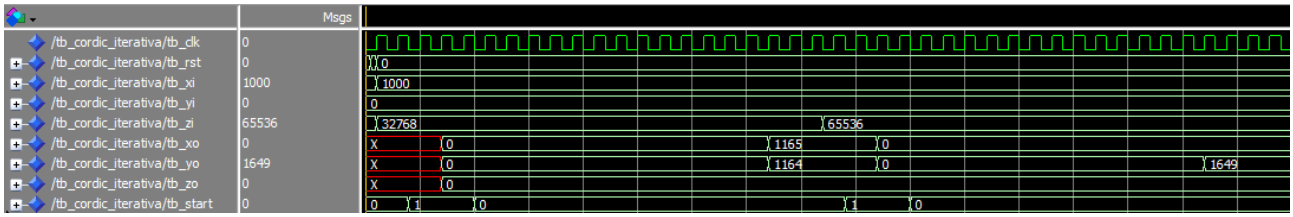


Figura 4.2: Simulación de rotación de 45° y 90°.

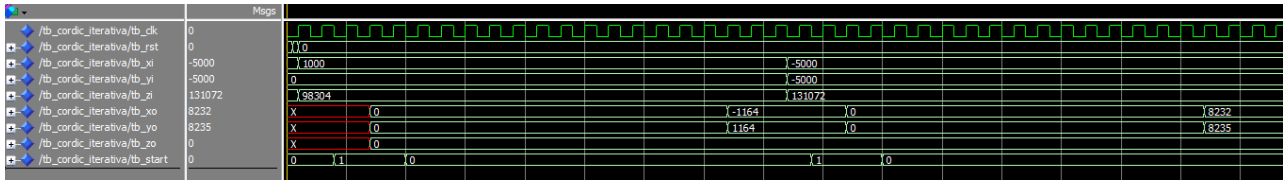


Figura 4.3: Simulación de rotación de 135° y 180°.

En primer lugar, se debe mencionar que tanto los valores de las entradas como los de las salidas están expresados en números decimales. Por este motivo, dado que la representación de los ángulos está expresada en palabras de 18 bits, el número 131072 corresponde a 180°, el número 65536 corresponde a 90°, y así sucesivamente.

En la figura 4.1 se puede observar cómo, partiendo de un vector con componente únicamente en el eje x, se lo rota 60°. Posteriormente, el vector resultante se rota 30° adicionales, obteniendo como resultado final un vector con componente únicamente en el eje y, lo cual es esperado dado que se ha realizado una rotación total de 90°.

En la figura 4.2, en primer lugar se rota un vector 45° que inicialmente tiene solo componente en el eje x, obteniendo un vector con componentes iguales en los ejes x e y. Posteriormente, se rota el mismo vector 90°, obteniendo un vector con únicamente componente en el eje y.

Por ultimo, en la 4.2, se parte de un vector con componente únicamente en el eje x, al cual se lo rota 135°, obteniendo como resultado final un vector con componentes x e y iguales en magnitud, pero con el componente x negativo. Esto es lógico, ya que es equivalente a rotar primero 90° y luego 45°, ubicando el vector en el segundo cuadrante. Posteriormente, se rota otro vector 180°, el cual tiene ambos componentes negativos y de igual magnitud, obteniendo como resultado un vector con ambos componentes positivos y de igual valor. Esto es coherente, ya que el vector se ha desplazado del tercer cuadrante al primer cuadrante.

En todas las simulaciones mencionadas anteriormente, se puede observar que en todos los casos el vector experimenta un escalamiento, como se mencionó previamente. Además, siempre transcurren 16 ciclos de reloj hasta obtener el resultado final y en todos los casos, al obtener el resultado final, la salida Z_o es 0 o un valor cercano a 0.

Además, se ejecutaron otras simulaciones con valores en los que el resultado no es tan evidente. Para verificar la precisión de los datos obtenidos, se desarrolló un script en Python, incluido en el Anexo de este informe, que implementa el algoritmo de forma iterativa, permitiendo comparar los resultados del script con los de las simulaciones.

5 Síntesis e Implementación

Una vez finalizadas las simulaciones de los circuitos descriptos mediante el lenguaje VHDL, se prosiguió a utilizar la herramienta *Vivado* para implementar el diseño de los circuitos combinacionales y secuenciales sobre el dispositivo Arty-Z7-10. A partir de ello, se obtuvieron los esquemáticos de los circuitos y la tabla de los recursos utilizados.

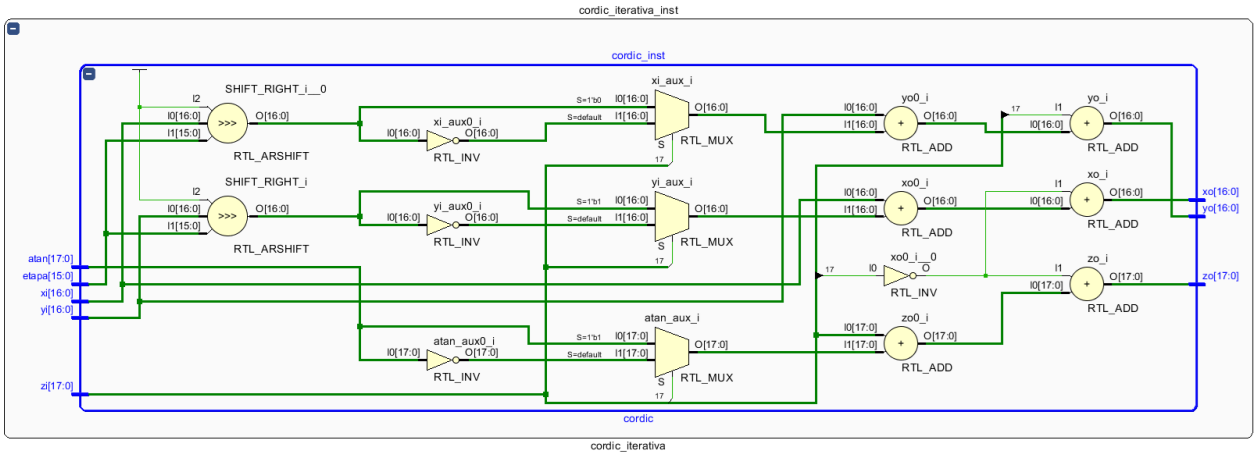


Figura 5.1: Esquemático unidad lógica CORDIC.

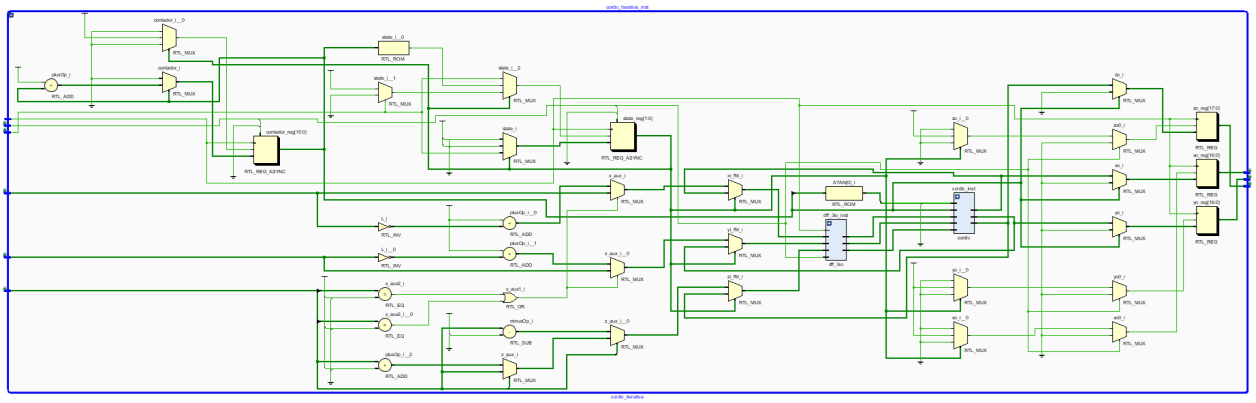


Figura 5.2: Esquemático arquitectura CORDIC iterativa.

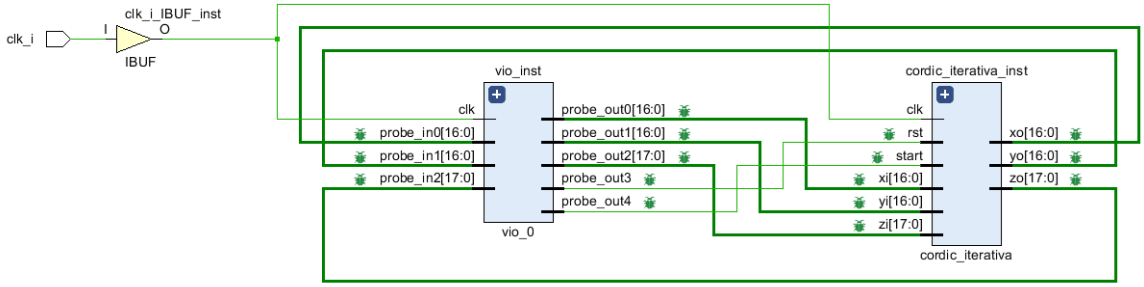


Figura 5.3: Esquemático VIO.

Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	LUT as Memory (6000)	LUT Flip Flop Pairs (17600)	Bonded IOB (100)	BUFGCTRL (32)	BSCANE2 (4)
cordic_VIO	6.14%	4.52%	10.41%	6.00%	0.40%	3.78%	1.00%	6.25%	25.00%
cordic_iterativa_inst (c...	1.37%	0.35%	1.68%	1.37%	0.00%	0.45%	0.00%	0.00%	0.00%
dbg_hub (dbg_hub)	2.63%	2.05%	5.14%	2.49%	0.40%	1.69%	0.00%	3.13%	25.00%
vio_inst (vio_0)	2.14%	2.12%	4.11%	2.14%	0.00%	1.59%	0.00%	0.00%	0.00%

Figura 5.4: Tabla de recursos utilizados.

Una vez concluida la etapa de síntesis e implementación, se procedió a generar el bitstream y a realizar diversas pruebas con la FPGA conectada al servidor, donde se testearon distintos casos. A continuación, en las figuras , se pueden observar algunas de las simulaciones, las cuales coinciden con las realizadas previamente.

Name	Value	Activity	Direction	VIO
rst	[B] 0		Output	hw_vio_1
start	[B] 1		Output	hw_vio_1
xi[16:0]	[U] 1000		Output	hw_vio_1
yi[16:0]	[U] 0		Output	hw_vio_1
zi[17:0]	[U] 43690		Output	hw_vio_1
input_x_vio[16:0]	[S] 824		Input	hw_vio_1
input_y_vio[16:0]	[S] 1427		Input	hw_vio_1
input_z_vio[17:0]	[U] 0		Input	hw_vio_1

(a) Rotación 60°.

Name	Value	Activity	Direction	VIO
rst	[B] 0		Output	hw_vio_1
start	[B] 1		Output	hw_vio_1
xi[16:0]	[U] 824		Output	hw_vio_1
yi[16:0]	[U] 1427		Output	hw_vio_1
zi[17:0]	[U] 21845		Output	hw_vio_1
input_x_vio[16:0]	[S] 0		Input	hw_vio_1
input_y_vio[16:0]	[S] 2715		Input	hw_vio_1
input_z_vio[17:0]	[S] -1		Input	hw_vio_1

(b) Rotación 30°.

Figura 5.5: Prueba de rotación de 60° y 30° en la FPGA.

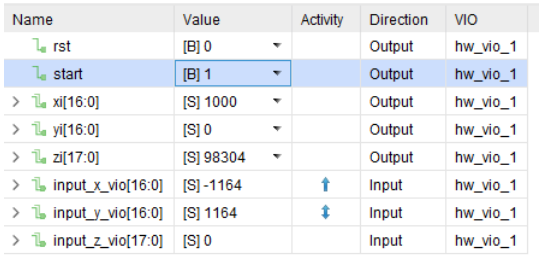
Name	Value	Activity	Direction	VIO
rst	[B] 0		Output	hw_vio_1
start	[B] 1		Output	hw_vio_1
xi[16:0]	[S] 1000		Output	hw_vio_1
yi[16:0]	[S] 0		Output	hw_vio_1
zi[17:0]	[S] 32768		Output	hw_vio_1
input_x_vio[16:0]	[S] 1165		Input	hw_vio_1
input_y_vio[16:0]	[S] 1164		Input	hw_vio_1
input_z_vio[17:0]	[S] 0		Input	hw_vio_1

(a) Rotación 45°.

Name	Value	Activity	Direction	VIO
rst	[B] 0		Output	hw_vio_1
start	[B] 1		Output	hw_vio_1
xi[16:0]	[S] 1000		Output	hw_vio_1
yi[16:0]	[S] 0		Output	hw_vio_1
zi[17:0]	[S] 65536		Output	hw_vio_1
input_x_vio[16:0]	[S] 0		Input	hw_vio_1
input_y_vio[16:0]	[S] 1649		Input	hw_vio_1
input_z_vio[17:0]	[S] 0		Input	hw_vio_1

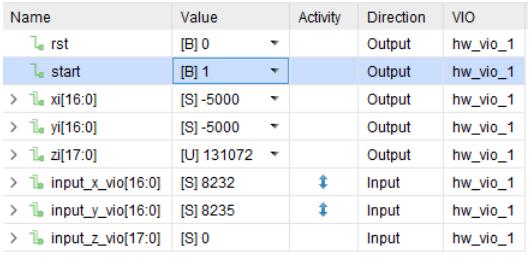
(b) Rotación 90°.

Figura 5.6: Prueba de rotación de 45° y 90° en la FPGA.



Name	Value	Activity	Direction	VIO
rst	[B] 0		Output	hw_vio_1
start	[B] 1		Output	hw_vio_1
xi[16:0]	[S] 1000		Output	hw_vio_1
yi[16:0]	[S] 0		Output	hw_vio_1
zi[17:0]	[S] 98304		Output	hw_vio_1
input_x_vio[16:0]	[S] -1164	↑	Input	hw_vio_1
input_y_vio[16:0]	[S] 1164	↕	Input	hw_vio_1
input_z_vio[17:0]	[S] 0		Input	hw_vio_1

(a) Rotación 135°.



Name	Value	Activity	Direction	VIO
rst	[B] 0		Output	hw_vio_1
start	[B] 1		Output	hw_vio_1
xi[16:0]	[S] -5000		Output	hw_vio_1
yi[16:0]	[S] -5000		Output	hw_vio_1
zi[17:0]	[U] 131072		Output	hw_vio_1
input_x_vio[16:0]	[S] 8232	↕	Input	hw_vio_1
input_y_vio[16:0]	[S] 8235	↕	Input	hw_vio_1
input_z_vio[17:0]	[S] 0		Input	hw_vio_1

(b) Rotación 180°.

Figura 5.7: Prueba de rotación de 135° y 180° en la FPGA.

6 Conclusión

A partir de lo realizado, se puede concluir que se logró cumplir el objetivo de este trabajo, ya que se implementó una arquitectura capaz de ejecutar el algoritmo CORDIC.

Además, se profundizó en la comprensión y familiarización con el lenguaje de descripción de hardware VHDL, lo cual fue facilitado por los diversos errores cometidos antes de alcanzar el código final y su correcto funcionamiento. También se logró entender las ventajas y desventajas de la arquitectura iterativa para la implementación del algoritmo CORDIC.

Una de las principales ventajas de la arquitectura iterativa es su bajo consumo de recursos, dado que siempre opera de forma iterativa. Sin embargo, esto implica esperar varios ciclos de reloj para obtener el resultado final, lo que puede ser contraproducente en ciertos aplicaciones. Por lo tanto, se concluye que esta arquitectura es ideal en situaciones donde se requiere una utilización mínima de recursos. Sin embargo, si se necesita mayor velocidad, será necesario recurrir a otra arquitectura, como la arquitectura pipeline.

7 Anexo

Script de python para calculo de rotaciones

```
    import numpy as np

def cordic_angles(num_iterations):
    return np.arctan(2.0 ** -np.arange(num_iterations))

def cordic_rotation(x_in, y_in, z_in, num_iterations):
    x = x_in
    y = y_in
    z = z_in
    angles = cordic_angles(num_iterations)

    for i in range(num_iterations):
        if z < 0:
            x_new = x + (y >> i)
            y_new = y - (x >> i)
            z_new = z + angles[i]
        else:
            x_new = x - (y >> i)
            y_new = y + (x >> i)
            z_new = z - angles[i]

        x, y, z = x_new, y_new, z_new

    return x, y, z

# Entradas
x_in = 19000
y_in = 9500
z_in_degrees = 60
z_in = np.deg2rad(z_in_degrees)
num_iterations = 16  # Precisión de 16 bits

# Ejecutar el algoritmo CORDIC
x_out, y_out, z_out = cordic_rotation(x_in, y_in, z_in, num_iterations)

# Mostrar los resultados
print(f"Zin radianes: {z_in}")
print(f"Xout: {x_out}")
print(f"Yout: {y_out}")
print(f"Zout (en grados): {np.rad2deg(z_out)}")
```