



MICROARQUITECTURAS Y SOFTCORES  
CARRERA DE ESPECIALIZACIÓN EN SISTEMAS EMBEBIDOS

---

**IP core CORDIC**

---

Alumno:

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

18 de junio de 2025

# Índice

<b>1</b>	<b>Introducción</b>	<b>2</b>
<b>2</b>	<b>Implementación módulo CORDIC</b>	<b>2</b>
<b>3</b>	<b>Diseño IP Core</b>	<b>2</b>
<b>4</b>	<b>Implementación del sistema base</b>	<b>3</b>
<b>5</b>	<b>Síntesis e implementación</b>	<b>3</b>
<b>6</b>	<b>Generación de aplicación en SDK</b>	<b>4</b>
<b>7</b>	<b>Validación de funcionamiento</b>	<b>6</b>
<b>8</b>	<b>Conclusión</b>	<b>7</b>
<b>9</b>	<b>Anexo A</b>	<b>8</b>
9.1	Introducción . . . . .	8
9.2	CORDIC . . . . .	8
9.3	Implementación . . . . .	9
9.4	Simulaciones . . . . .	11
9.5	Síntesis e Implementación . . . . .	13
9.6	Conclusión . . . . .	15
9.7	Anexo . . . . .	16

# 1 Introducción

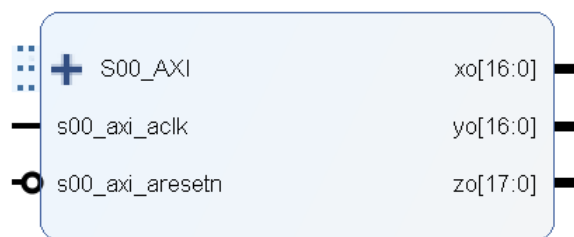
A lo largo de este trabajo práctico se desarrollará e implementará un componente de hardware digital descrito en lenguaje VHDL (IP Core), que será simulado e integrado en un kit FPGA. Este componente corresponde a la implementación de una arquitectura CORDIC iterativa. Formará parte de un sistema base de procesamiento que incluye un microprocesador Cortex-A9, con el cual se establecerá la conectividad entre el PS (*Processing System*) y la PL (*Programmable Logic*). El core recibirá datos provenientes del procesador, y su funcionamiento será controlado mediante un programa desarrollado en lenguaje C.

## 2 Implementación módulo CORDIC

Para una explicación detallada del algoritmo CORDIC, incluyendo su desarrollo matemático, formulación recursiva, simulaciones, implementación en hardware y testeo utilizando un VIO en la FPGA consultar el Anexo A. En dicho anexo se incorpora el trabajo práctico previamente realizado en la materia Circuitos Lógicos Programables, donde se abordó en profundidad el funcionamiento del algoritmo y su implementación como IP Core en una FPGA.

## 3 Diseño IP Core

Para el diseño del *IP core* CORDIC se utilizó una única interfaz AXI *slave*, a través de la cual el microprocesador se comunica con el *IP core* mediante cuatro registros. Los registros 0, 1 y 2 se emplean para cargar los valores de entrada:  $x_i$ ,  $y_i$  y  $z_i$ , que representan las coordenadas iniciales y el ángulo de rotación deseado. El cuarto registro utiliza únicamente el primer bit, que actúa como señal de *start* para iniciar la operación de rotación. A continuación, en la figura 3.1 se muestra un diagrama del *IP core* implementado.



**Figura 3.1:** Diagrama en bloque IP core.

En cuanto a las salidas, se agregaron tres señales que corresponden a  $x_o$ ,  $y_o$  y  $z_o$ , las cuales representan las coordenadas finales luego de la rotación. En este diseño, dichas señales no se comunican con el microprocesador, sino que están pensadas para conectarse a algún puerto de salida. En este trabajo, específicamente, se utilizaron como entradas a un módulo VIO para su visualización y validación.

Además, se incorporó un parámetro que define la cantidad de bits a utilizar en el CORDIC. Este valor está directamente relacionado, debido al funcionamiento del algoritmo, tanto con la cantidad de iteraciones como con la precisión de las entradas y salidas.

Una vez finalizado el diagrama, se realizó la síntesis del diseño con el objetivo de verificar su correcta implementación.

## 4 Implementación del sistema base

Para la verificación del *IP core*, se implementó un sistema basado en el procesador ARM Cortex-A9, destinado a ser ejecutado en la placa Arty Z7-10, utilizando la herramienta *Vivado*. A través de esta aplicación se diseñó el sistema de hardware y se generó el el SDK, donde se desarrolló una aplicación de ejemplo para verificar la funcionalidad del hardware, tanto del *IP core* como la comunicación con el microprocesador.

En primer lugar, utilizando el *IP Integrator* de la herramienta *Vivado*, se agregó el bloque del sistema de procesamiento ZYNQ, donde se habilitaron los periféricos necesarios. Luego, se incorporaron el módulo VIO y el *IP core* del CORDIC, los cuales fueron conectados parcialmente de forma manual y parcialmente utilizando las funciones automáticas de la herramienta. Estas conexiones generaron automáticamente los bloques de *system reset* e interconexión AXI, obteniendo como resultado el diseño final que se observa en la figura 4.1.

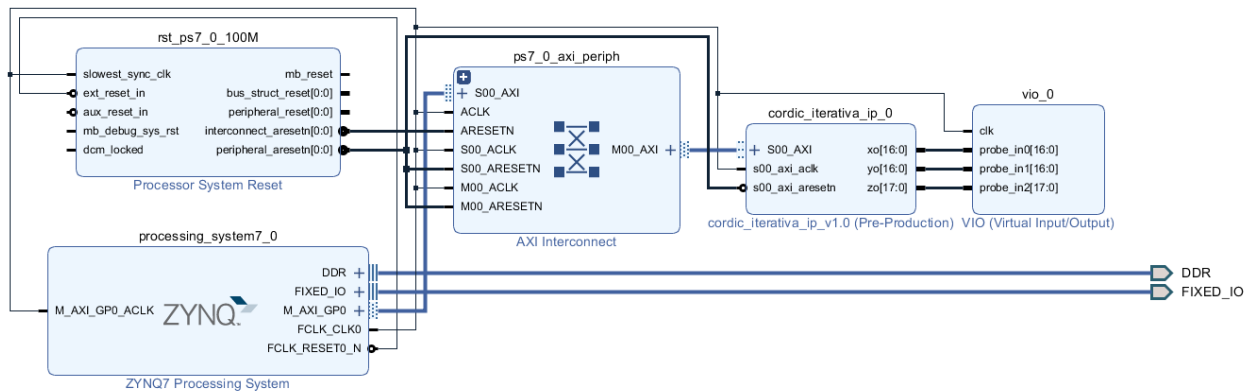


Figura 4.1: Diagrama en bloque del sistema general.

## 5 Síntesis e implementación

Una vez finalizadas el diseño, se prosiguió a realizar la síntesis y la implementación del circuito donde a continuación, en las figuras 5.1 y 5.2, se muestran los esquemas correspondientes como la tabla de recursos utilizados.

Name	Slice LUTs (17600)	Slice Registers (35200)	Bonded IOPADs (130)
▼ <b>N</b> system_wrapper	6.16%	4.10%	100.00%
▣ dbg_hub (dbg_hub_CV)	0.00%	0.00%	0.00%
▼ <b>I</b> system_i (system)	6.16%	4.10%	0.00%
> <b>I</b> cordic_iterativa_ip_0 (system_cordic_iterativa_ip_0_0)	1.55%	0.83%	0.00%
> <b>I</b> processing_system7_0 (system_processing_system7_0_0)	0.64%	0.00%	0.00%
> <b>I</b> ps7_0_axi_periph (system_ps7_0_axi_periph_0)	2.39%	1.70%	0.00%
> <b>I</b> rst_ps7_0_100M (system_rst_ps7_0_100M_0)	0.11%	0.11%	0.00%
> <b>I</b> vio_0 (system_vio_0_0)	1.48%	1.46%	0.00%

Figura 5.1: Tabla de recursos de la síntesis.

Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	LUT as Memory (6000)	LUT Flip Flop Pairs (17600)	Bonded IOPADs (130)	BUFGCTRL (32)	BSCAN2 (4)
▼ <b>N</b> system_wrapper	7.67%	5.67%	13.34%	7.19%	1.40%	4.83%	100.00%	6.25%	25.00%
▼ <b>I</b> dbg_hub (dbg_hub)	2.63%	2.05%	5.25%	2.49%	0.40%	1.75%	0.00%	3.13%	25.00%
> <b>I</b> inst (xsdbm_v3_0_0_xsdbm)	2.63%	2.05%	5.25%	2.49%	0.40%	1.75%	0.00%	3.13%	25.00%
▼ <b>I</b> system_i (system)	5.04%	3.62%	8.32%	4.70%	1.00%	3.02%	0.00%	3.13%	0.00%
> <b>I</b> cordic_iterativa_ip_0 (system_cordic_iterativa_ip_0_0)	1.55%	0.83%	2.20%	1.55%	0.00%	0.81%	0.00%	0.00%	0.00%
> <b>I</b> processing_system7_0 (system_processing_system7_0_0)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	3.13%	0.00%
> <b>I</b> ps7_0_axi_periph (system_ps7_0_axi_periph_0)	1.92%	1.24%	2.98%	1.59%	0.98%	0.98%	0.00%	0.00%	0.00%
> <b>I</b> rst_ps7_0_100M (system_rst_ps7_0_100M_0)	0.09%	0.09%	0.25%	0.09%	0.02%	0.08%	0.00%	0.00%	0.00%
> <b>I</b> vio_0 (system_vio_0_0)	1.48%	1.46%	3.30%	1.48%	0.00%	1.15%	0.00%	0.00%	0.00%

Figura 5.2: Tabla de recursos de la implementación

## 6 Generación de aplicación en SDK

Una vez finalizadas las etapas de síntesis e implementación, se procedió a generar el *bitstream* y a exportar el hardware junto con dicho archivo, lo que permitió la generación del entorno SDK. A continuación, se desarrolló el código en lenguaje C para verificar el funcionamiento completo del sistema.

Para la aplicación en C, se codificaron manualmente distintos valores de prueba con el fin de evaluar el funcionamiento del sistema. Se observaron las salidas generadas para verificar si los resultados eran correctos en cada caso. A continuación, se muestra parte de un ejemplo del código en C utilizado durante una de las ejecuciones de prueba.

```
#include "xparameters.h"
#include "xil_io.h"
#include "cordic_iterativa_ip.h"

//=====

int main (void)
{
```

```

int i,j;

xil_printf("-- Start of the Program --\r\n");

while (1)
{
    CORDIC_ITERATIVA_IP_mWriteReg(XPAR_CORDIC_ITERATIVA_IP_0_S00_AXI_BASEADDR,
        CORDIC_ITERATIVA_IP_S00_AXI_SLV_REG3_OFFSET, 0);
    CORDIC_ITERATIVA_IP_mWriteReg(XPAR_CORDIC_ITERATIVA_IP_0_S00_AXI_BASEADDR,
        CORDIC_ITERATIVA_IP_S00_AXI_SLV_REG0_OFFSET, 824);
    CORDIC_ITERATIVA_IP_mWriteReg(XPAR_CORDIC_ITERATIVA_IP_0_S00_AXI_BASEADDR,
        CORDIC_ITERATIVA_IP_S00_AXI_SLV_REG1_OFFSET, 1427);
    CORDIC_ITERATIVA_IP_mWriteReg(XPAR_CORDIC_ITERATIVA_IP_0_S00_AXI_BASEADDR,
        CORDIC_ITERATIVA_IP_S00_AXI_SLV_REG2_OFFSET, 21845);
    CORDIC_ITERATIVA_IP_mWriteReg(XPAR_CORDIC_ITERATIVA_IP_0_S00_AXI_BASEADDR,
        CORDIC_ITERATIVA_IP_S00_AXI_SLV_REG3_OFFSET, 1);
    for(j=0; j<10; j++) {
        for (i=0; i<9999999; i++);
    }

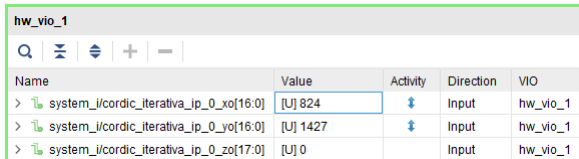
    CORDIC_ITERATIVA_IP_mWriteReg(XPAR_CORDIC_ITERATIVA_IP_0_S00_AXI_BASEADDR,
        CORDIC_ITERATIVA_IP_S00_AXI_SLV_REG3_OFFSET, 0);
    CORDIC_ITERATIVA_IP_mWriteReg(XPAR_CORDIC_ITERATIVA_IP_0_S00_AXI_BASEADDR,
        CORDIC_ITERATIVA_IP_S00_AXI_SLV_REG0_OFFSET, 1000);
    CORDIC_ITERATIVA_IP_mWriteReg(XPAR_CORDIC_ITERATIVA_IP_0_S00_AXI_BASEADDR,
        CORDIC_ITERATIVA_IP_S00_AXI_SLV_REG1_OFFSET, 0);
    CORDIC_ITERATIVA_IP_mWriteReg(XPAR_CORDIC_ITERATIVA_IP_0_S00_AXI_BASEADDR,
        CORDIC_ITERATIVA_IP_S00_AXI_SLV_REG2_OFFSET, 32768);
    CORDIC_ITERATIVA_IP_mWriteReg(XPAR_CORDIC_ITERATIVA_IP_0_S00_AXI_BASEADDR,
        CORDIC_ITERATIVA_IP_S00_AXI_SLV_REG3_OFFSET, 1);
    for(j=0; j<10; j++) {
        for (i=0; i<9999999; i++);
    }
}
}

```

Como se puede observar en el código, la secuencia principal consiste en setear en 0 el bit de *start* del *IP core*, luego cargar los valores de entrada correspondientes, y finalmente setear en 1 dicho bit para iniciar la operación. Dado que se trata de un *IP core* CORDIC iterativo, se requieren 16 ciclos de reloj para completar la conversión. En este caso, se implementa un bucle anidado que actúa como retardo. Si bien este retardo es excesivo en términos de tiempo real, se utiliza intencionalmente para facilitar la visualización de los cambios de salida en una escala de tiempo perceptible para el usuario. En este caso particular, dado que las salidas están conectadas a un VIO, los valores obtenidos deben visualizarse desde la interfaz de Vivado.

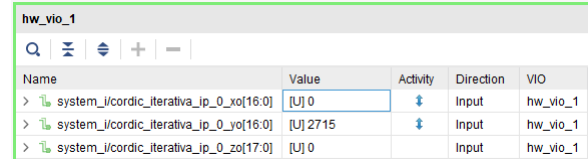
## 7 Validación de funcionamiento

Una vez implementada la aplicación para verificar el funcionamiento del sistema, se procedió a realizar las pruebas correspondientes para validar tanto el comportamiento del *IP core* como la comunicación con el microprocesador. En las Figuras 7.1, 7.2 y 7.3 se muestran algunas de las simulaciones realizadas.



Name	Value	Activity	Direction	VIO
> system_i/cordic_iterativa_ip_0_xo[16:0]	[U] 824	↓	Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_yo[16:0]	[U] 1427	↓	Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_zo[17:0]	[U] 0		Input	hw_vio_1


(a) Rotación 60°.



Name	Value	Activity	Direction	VIO
> system_i/cordic_iterativa_ip_0_xo[16:0]	[U] 0	↓	Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_yo[16:0]	[U] 2715	↓	Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_zo[17:0]	[U] 0		Input	hw_vio_1

(b) Rotación 30°.

**Figura 7.1:** Prueba de rotación de 60° y 30°.



Name	Value	Activity	Direction	VIO
> system_i/cordic_iterativa_ip_0_xo[16:0]	[U] 1165	↓	Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_yo[16:0]	[U] 1164	↓	Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_zo[17:0]	[U] 0		Input	hw_vio_1

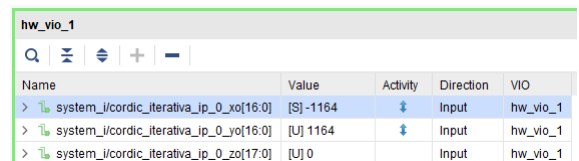
(a) Rotación 45°.



Name	Value	Activity	Direction	VIO
> system_i/cordic_iterativa_ip_0_xo[16:0]	[U] 0		Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_yo[16:0]	[U] 1649		Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_zo[17:0]	[U] 0		Input	hw_vio_1

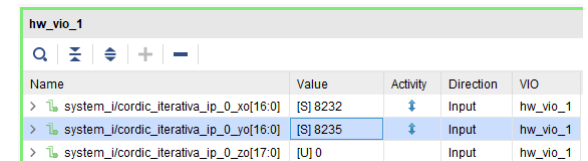
(b) Rotación 90°.

**Figura 7.2:** Prueba de rotación de 45° y 90°.



Name	Value	Activity	Direction	VIO
> system_i/cordic_iterativa_ip_0_xo[16:0]	[S] -1164	↓	Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_yo[16:0]	[U] 1164	↓	Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_zo[17:0]	[U] 0		Input	hw_vio_1

(a) Rotación 135°.



Name	Value	Activity	Direction	VIO
> system_i/cordic_iterativa_ip_0_xo[16:0]	[S] 8232	↓	Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_yo[16:0]	[S] 8235	↓	Input	hw_vio_1
> system_i/cordic_iterativa_ip_0_zo[17:0]	[U] 0		Input	hw_vio_1

(b) Rotación 180°.

**Figura 7.3:** Prueba de rotación de 135° y 180°.

Las pruebas llevadas a cabo corresponden a las mismas realizadas en el trabajo práctico final de la materia Circuitos Lógicos Programables. Para más detalles sobre las simulaciones y sus resultados consultar a la sección correspondiente del Anexo A. A continuación, se presenta un resumen de los resultados obtenidos.

En la figura 7.1 se muestra la rotación de un vector que inicialmente tiene un valor de 1000 en  $x_i$  y 0 en  $y_i$ , es decir, está alineado con el eje  $x$ . Primero se le aplica una rotación de 60°, y posteriormente se lo rota 30° adicionales. Como resultado, se obtiene un vector alineado con el eje  $y$ , lo cual es coherente con una rotación total de 90°.

En la figura 7.2 se parte nuevamente de un vector con valores iniciales de 1000 en  $x_i$  y 0 en  $y_i$ , es decir, alineado con el eje  $x$ . Primero se aplica una rotación de 45°, lo que genera un vector con componentes iguales en  $x$  e  $y$ . Luego, al mismo vector se le aplica una rotación adicional de 90°, obteniendo como resultado un vector alineado con el eje  $y$ , tal como se espera.

En la figura 7.3 se muestra, en primer lugar, la rotación de un vector con valores iniciales de 1000 en  $x_i$  y 0 en  $y_i$  aplicando un ángulo de  $135^\circ$ . El resultado es un vector ubicado en el segundo cuadrante, con componentes  $x$  e  $y$  iguales en magnitud, pero con  $x$  negativo. A continuación, se rota un segundo vector con valores iniciales de  $-5000$  tanto en  $x_i$  como en  $y_i$  aplicando una rotación de  $180^\circ$ , obteniendo como resultado un vector con ambos componentes positivos, ubicado en el primer cuadrante, lo cual es coherente con lo esperado.

En todas las simulaciones se observa el escalamiento propio del algoritmo CORDIC, y la conversión completa se obtiene tras 16 ciclos de reloj. Además, en todos los casos la salida  $Z_o$  converge a cero o a un valor muy próximo.

## 8 Conclusión

A partir del desarrollo realizado en este trabajo, se concluye que se logró implementar y verificar exitosamente un IP core basado en el algoritmo CORDIC, integrándolo con un sistema basado en un procesador ARM Cortex-A9 sobre la placa Arty-Z7-010.

Asimismo, se profundizó en el diseño e integración de hardware y software, utilizando Vivado para la síntesis y generación del bitstream, y desarrollando una aplicación en C para validar la comunicación y funcionalidad del IP core. Este proceso permitió comprender mejor la interacción entre el hardware programable y el software embebido, así como las consideraciones de temporización y sincronización propias de este tipo de sistemas.



## 9 Anexo A

### 9.1 Introducción

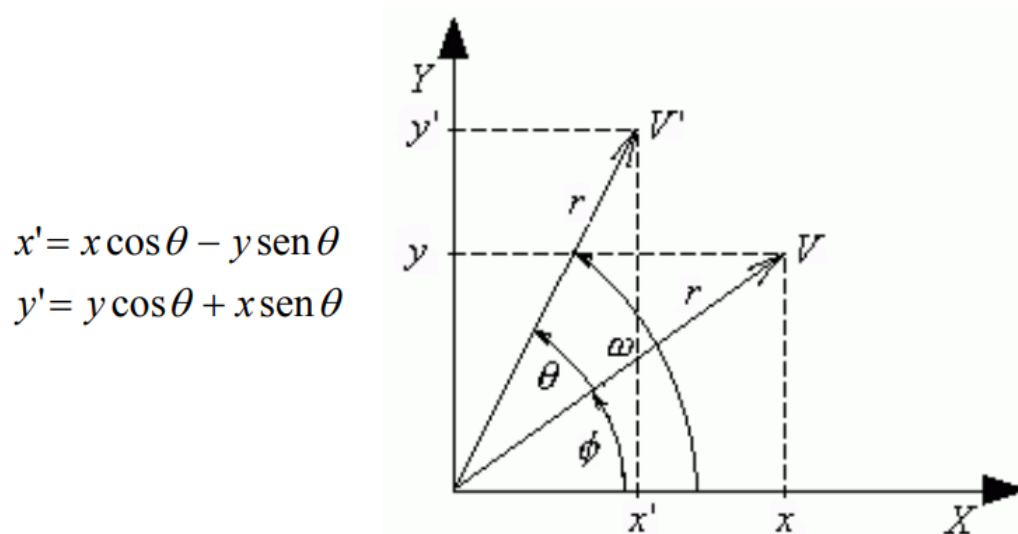
A lo largo de este trabajo práctico se desarrollará, simulará e implementará una arquitectura CORDIC iterativa mediante la descripción de un circuito combinacional y secuencial, aplicando el lenguaje de descripción de hardware VHDL.

### 9.2 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



**Figura 9.1:** 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  $\tan \theta = 2^{-i}$ , lo que convierte la multiplicación por la

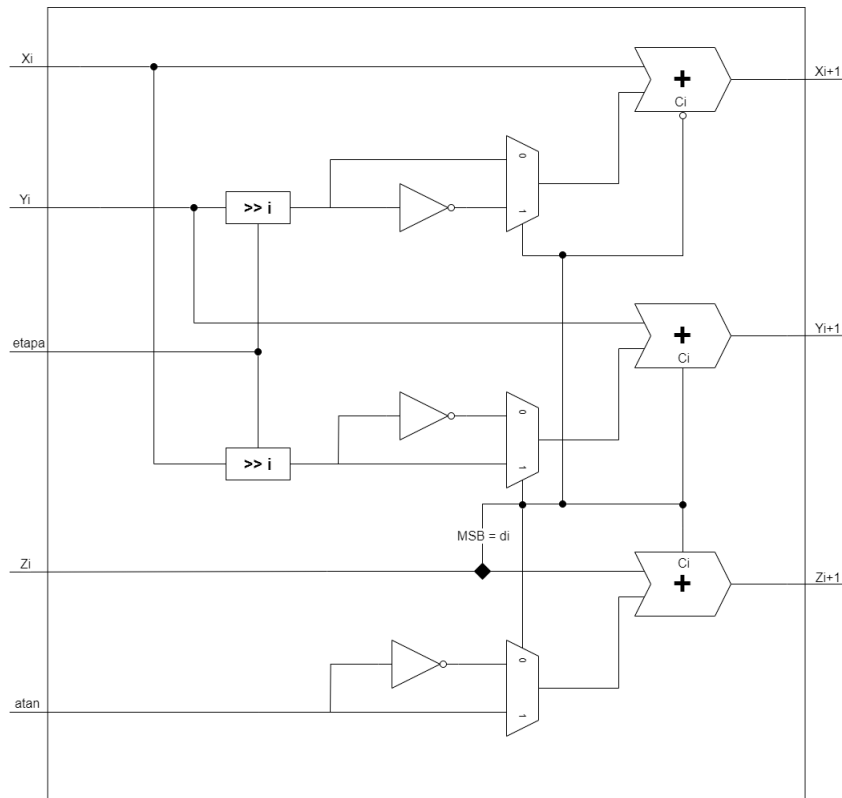
tangente en una simple operación de desplazamiento de bits. De este modo, se obtienen las siguientes ecuaciones

$$\begin{aligned}x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\z_{i+1} &= z_i - d_i \cdot \operatorname{tg}^{-1}(2^{-i})\end{aligned}$$

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.

### 9.3 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.



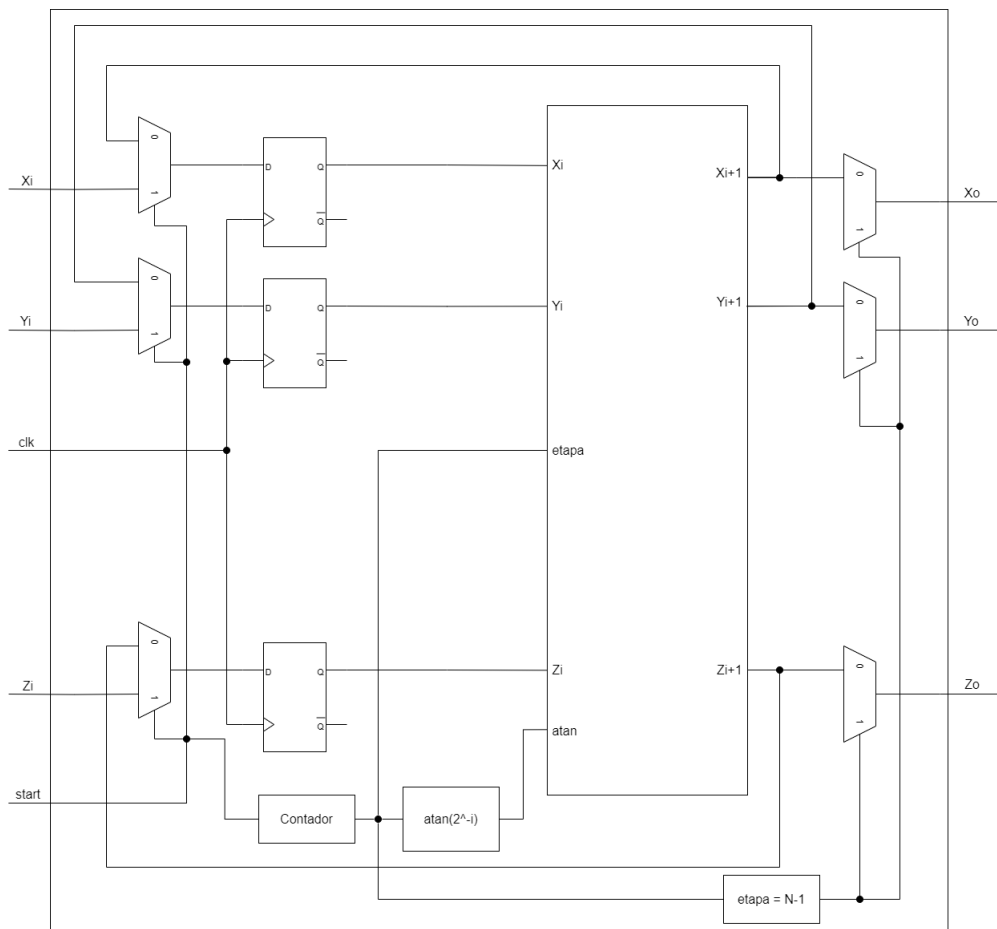
**Figura 9.2:** Diagrama implementación CORDIC.

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 9.3:** 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  $90^\circ$  y  $-90^\circ$ , por lo tanto si desea poder rotar  $360^\circ$  se debe agregar una etapa de pre procesamiento. Para ello, se implementó un circuito capaz de detectar en que cuadrante se encontraba el ángulo de rotación y en caso de estar entre  $90^\circ$  y  $180^\circ$  o entre  $-90^\circ$  y  $-180^\circ$ , 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  $180^\circ$ . De esta forma se vuelve a obtener una operación en la cual el CORDIC únicamente tiene que rotar entre  $90^\circ$  y  $-90^\circ$ .

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.

## 9.4 Simulaciones

Una vez implementados los circuitos, se procedió a realizar las simulaciones correspondientes para verificar el funcionamiento del algoritmo. En las figuras 9.4, 9.5 y 9.6 se muestran algunas de las simulaciones realizadas.

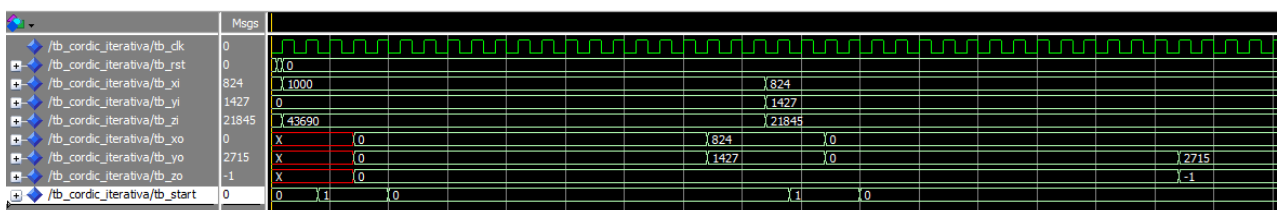
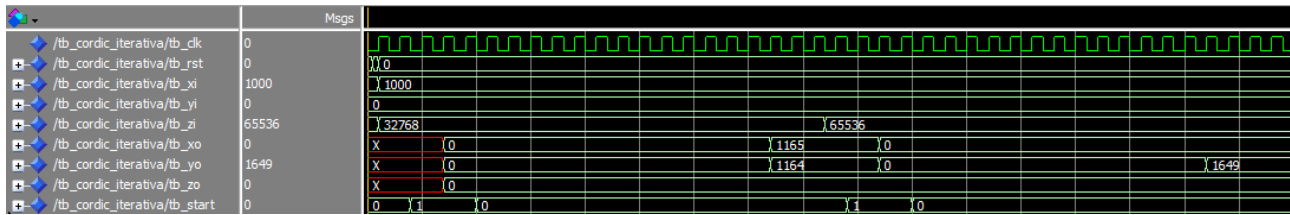
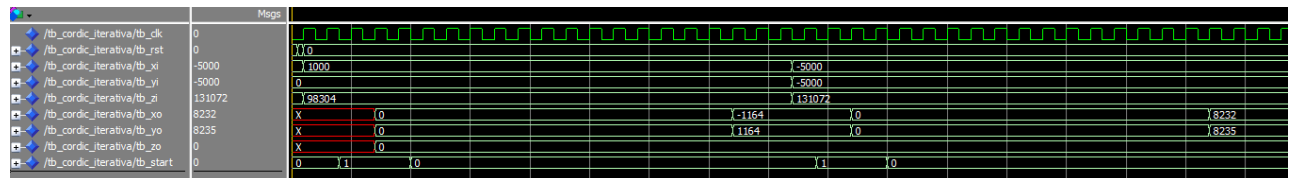


Figura 9.4: Simulación de rotación de  $60^\circ$  y  $30^\circ$ .



**Figura 9.5:** Simulación de rotación de 45° y 90°.



**Figura 9.6:** 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 9.4 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 9.5, 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.

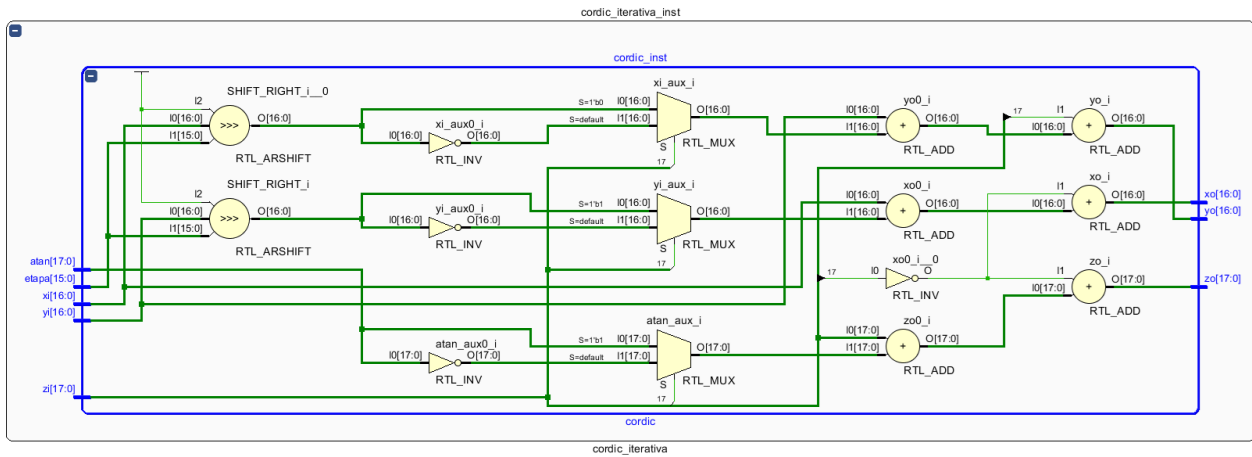
Por ultimo, en la 9.5, se parte de un vector con componente únicamente en el eje x, al cual se lo rota 135°, obteniendo como resultado final un vector con componentes x e y iguales en magnitud, pero con el componente x negativo. Esto es lógico, ya que es equivalente a rotar primero 90° y luego 45°, ubicando 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.

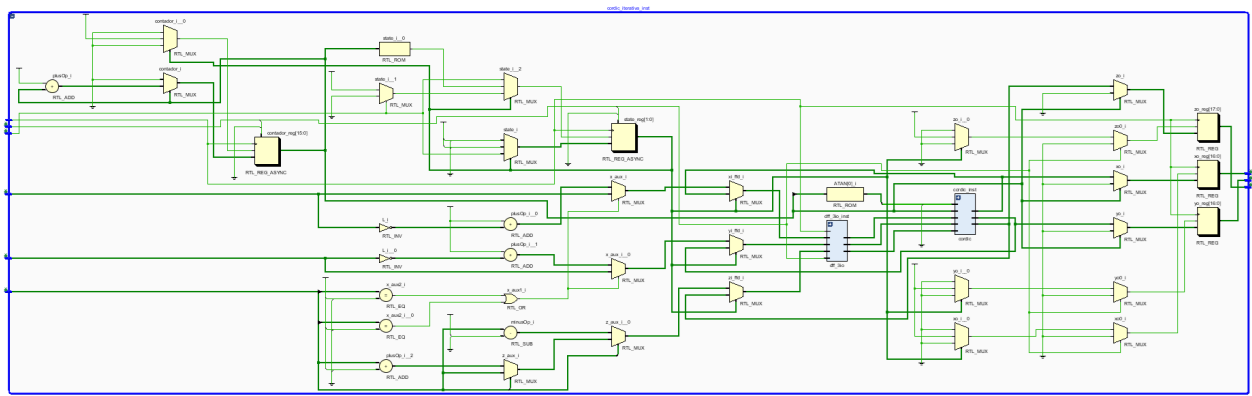
Además, se ejecutaron otras simulaciones con valores en los que el resultado no es tan evidente. Para verificar la precisión de los datos obtenidos, se desarrolló un script en Python, incluido en el Anexo de este informe, que implementa el algoritmo de forma iterativa, permitiendo comparar los resultados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

## 9.5 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 9.7:** Esquemático unidad lógica CORDIC.



**Figura 9.8:** Esquemático arquitectura CORDIC iterativa.

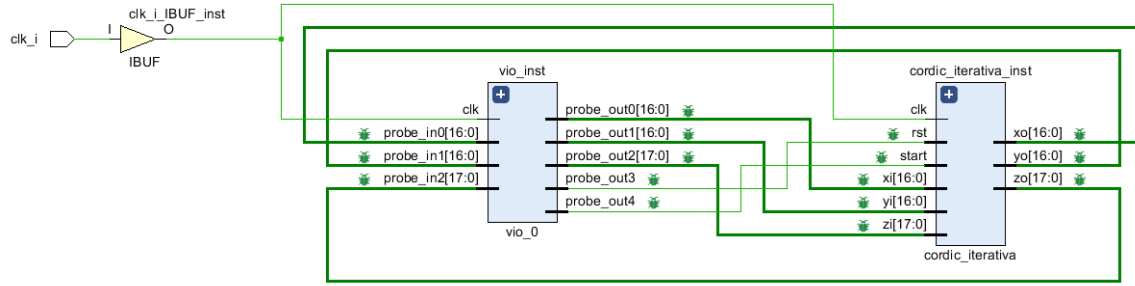


Figura 9.9: Esquemático VIO.

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)	BUFCTRL (32)	BSCANE2 (4)
<b>N</b> cordic_VIO	6.14%	4.52%	10.41%	6.00%	0.40%	3.78%	1.00%	6.25%	25.00%
> <b>I</b> cordic_iterativa_inst (c...	1.37%	0.35%	1.68%	1.37%	0.00%	0.45%	0.00%	0.00%	0.00%
> <b>I</b> dbg_hub (dbg_hub)	2.63%	2.05%	5.14%	2.49%	0.40%	1.69%	0.00%	3.13%	25.00%
> <b>I</b> vio_inst (vio_0)	2.14%	2.12%	4.11%	2.14%	0.00%	1.59%	0.00%	0.00%	0.00%

Figura 9.10: 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 9.11: 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 9.12: 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 9.13:** Prueba de rotación de 135° y 180° en la FPGA.

## 9.6 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 ciertas 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.



## 9.7 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)}")
```