

# Sistemas Digitales para las Comunicaciones

## Trabajo Práctico Final

*Sistema de comunicaciones OFDM*

MSE 4ta cohorte.

Integrantes:  
Martin Menendez, Eric Pernia, Santiago Germino.

<b>Objetivo</b>	<b>2</b>
<b>Plataforma utilizada</b>	<b>2</b>
<b>Diseño e implementación</b>	<b>3</b>
<b>Resultados</b>	<b>4</b>
<b>Funcionamiento</b>	<b>4</b>
Microprocesador	4
Lógica programable	5
<b>Bloques utilizados</b>	<b>6</b>
AXI Stream Width Converter (IP de Xilinx)	6
Adaptador 8 a 2 (desarrollo propio)	6
Adaptador 4 a 8 (desarrollo propio)	7
FIFO AXI Stream (IP de Xilinx)	7
Convolutional Encoder (IP de Xilinx)	7
Mapping 16 PSK (desarrollo propio)	8
Complemento de fase (desarrollo propio)	10
IFFT (IP FFT de Xilinx)	10
FFT (IP de Xilinx)	11
De-Mapping 16 PSK (desarrollo propio)	11
Declaración del componente cordic_iter	12
Array de valores de la arctan()	13
Cálculo del ángulo, con corrección del cuadrante de la arctan()	14
Instanciación de N bloques cordic_iter	15
Conversión de valores dentro de cierta porción de ángulo a binario de 4 bits	15
Viterbi (IP de Xilinx)	17
<b>Utilidades</b>	<b>18</b>
Script en python para envío y recepción de datos	18
<b>Interfaz Alternativa de PC a Lógica Programable</b>	<b>19</b>
Verificación	22

# Objetivo

Implementar en FPGA un sistema básico de comunicaciones OFDM.

El diagrama propuesto (Figura 1) fue diseñado para el entorno de desarrollo “ISE” de Xilinx y una FPGA Spartan-3E.

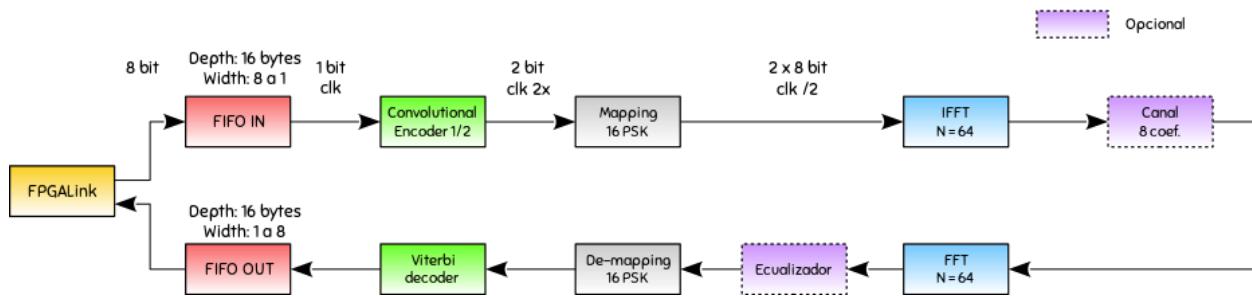


Figura 1. diagrama propuesto para el trabajo práctico.

## Plataforma utilizada

Debido a que Xilinx ya no soporta el entorno ISE, desde 2018 el posgrado migró sus herramientas de práctica al nuevo entorno de desarrollo de la empresa denominado “Vivado” y a un sistema heterogéneo de microprocesador + FPGA utilizando un kit de desarrollo Arty Z7-10 de Digilent Inc.

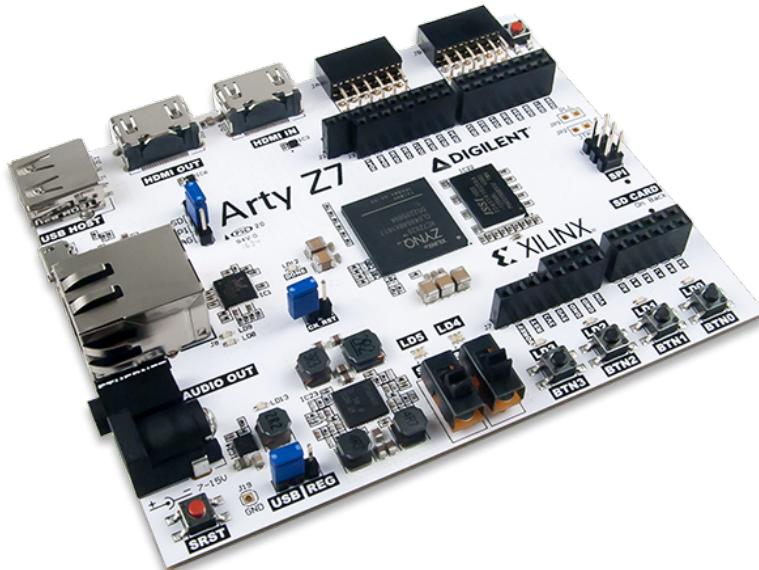


Figura 2. Kit de desarrollo Arty Z7-10 de Digilent Inc.

Este nuevo kit integra un microprocesador dual-core Cortex-A9 a 650 Mhz y una FPGA de la serie 7 de Xilinx con 17.600 LUTs, 35.200 Flip-Flops y 270 KB de Block RAM entre otras características.

## Diseño e implementación

Se diseñó un sistema OFDM utilizando como guía el diagrama propuesto y teniendo en cuenta las particularidades de la nueva plataforma y entorno de desarrollo:

1. El micro debe programarse e iniciarse.
2. El micro se comunica con la FPGA mediante un bus AXI Stream de 32 bits.
3. Los bloques IP (intellectual property de Xilinx) utilizan distinto número o configuración de bits en la entrada y/o salida.

Debido al punto 2 y que otras IP del entorno “Vivado” actualmente utilizan un bus AXI, el diseño del sistema OFDM se realizó previendo la interconexión de todos los bloques del sistema también mediante un bus AXI. El hecho de utilizar interfaces AXIs permitió realizar el proyecto con un solo dominio de reloj. En cuanto a los bits de entrada y salida, se trabajó utilizando los tamaños de datos y configuraciones posibles en las IP de dicho entorno.

En la Figura 3 se observa la interfaz del bus AXI que comunica al microprocesador o “processing system” con la lógica programable.

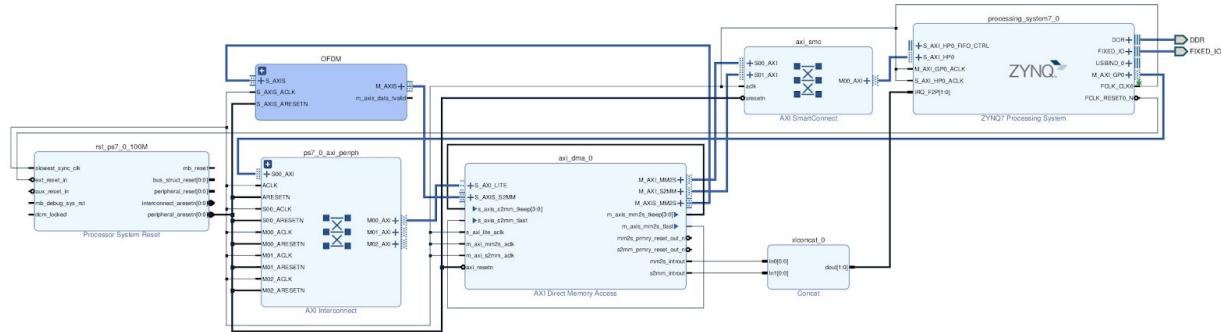
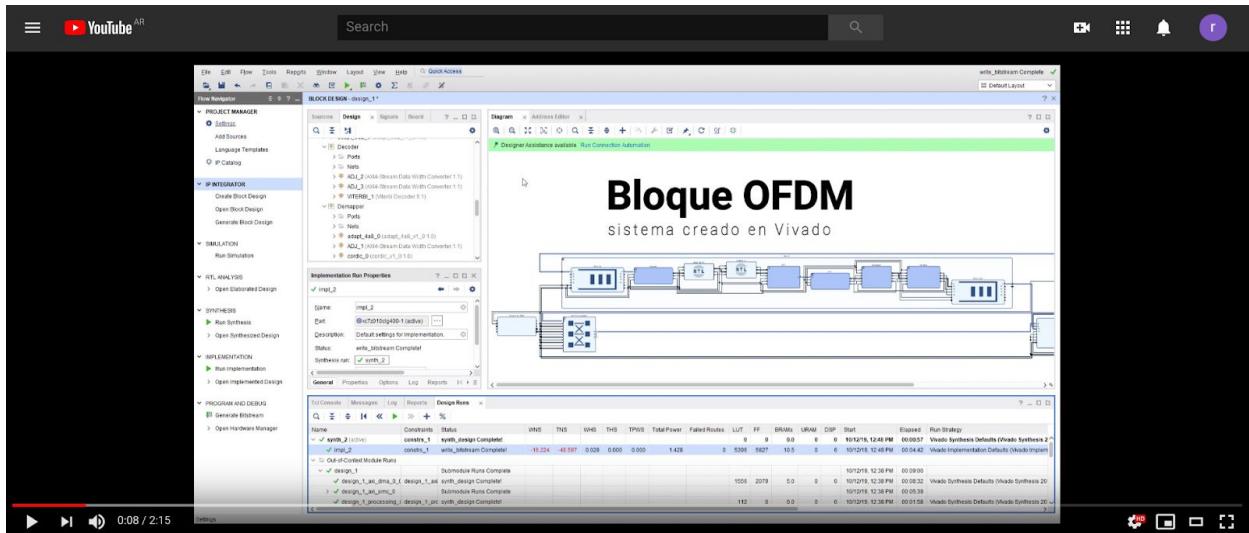


Figura 3. Interfaz entre la lógica programable y el “processing system”

El sistema desarrollado se encuentra en el bloque de lógica programable denominado “OFDM”. Como se comentó, este bloque está conectado al “processing system” mediante un bus AXI Stream de 32 bit de datos.

# Resultados

El funcionamiento del proyecto se ilustra a modo de resumen en un video informativo, dando una rápida descripción de los bloques conectados, el panorama general de conexiones, el uso del SDK y el script corriendo correctamente con la FPGA conectada por Ethernet. El video fue subido a Youtube para una fácil visualización: <https://www.youtube.com/watch?v=f85f6mEjRII>



Video ilustrativo del proyecto en Youtube

## Funcionamiento

La explicación sobre el funcionamiento se divide en dos dominios: microprocesador y lógica programable.

### Microprocesador

En el entorno “Vivado SDK” se indica que se desea un proyecto con soporte de FreeRTOS 10 y lwIP. Dado que el microprocesador es doble nucleo. se especifica que sólo se usará el primer núcleo. El microprocesador utiliza una interfaz UART para enviar mensajes de debug a través de un conversor integrado UART/USB hacia la PC y un puerto Ethernet manejado por la librería lwIP. Los datos enviados por la PC son recibidos por el puerto Ethernet del kit y re-enviados por el bus AXI hacia la lógica programable. Los datos enviados por la lógica programable son re-enviados por el puerto Ethernet hacia la PC. Para realizar el envío y recepción de datos se programó un script simple en Python. En la Figura 4 se observa un diagrama de los elementos que intervienen en la comunicación.

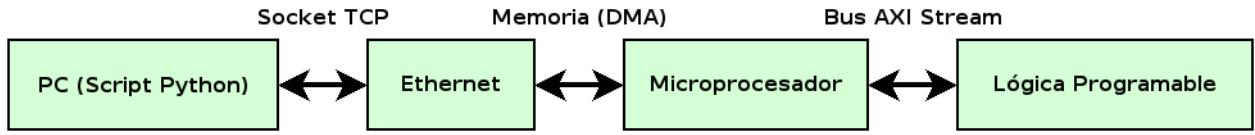


Figura 4. Comunicación entre PC y Lógica programable.

## Lógica programable

Se configura mediante la creación de un archivo bitstream con la configuración de la lógica programable a través del entorno “Vivado”. El sistema “OFDM” implementado en la lógica programable está conformado por los siguientes bloques:

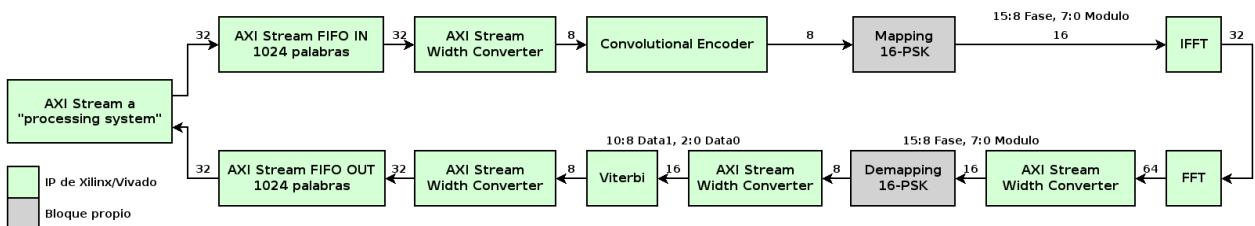


Figura 5. Elementos del sistema OFDM

Los datos enviados por el microprocesador a través del bus AXI son acumulados en un AXI Stream FIFO de 1024 palabras de 32 bit. A su vez, cada palabra se corta de a 8 bit para alimentar al Convolutional Encoder cuya finalidad es agregar redundancia a la información para protegerla. El resultado, también de 8 bit, es enviado a Mapping 16-PSK el cual calcula módulo y fase del símbolo a transmitir.

En este punto es importante aclarar que según la documentación de la IP de FFT (*Fast Fourier Transform*) de Xilinx, para lograr una IFFT (*Inverse Fast Fourier Transform*) es necesario complementar la fase ingresada, lo que se logra realizando una operación NOT sobre la fase. Hecho eso, la información de módulo y fase (de 8 bits cada una) es concatenada en 16 bit e ingresada al bloque IFFT. En este punto ya se dispone de un transmisor con mínima funcionalidad.

La salida de 32 bit del transmisor se conecta con la entrada del receptor en el cual hay un bloque FFT. El resultado, de 64 bit, se corta de a 16 bit (8 bit de fase y 8 bit de módulo) para alimentar al Demapping 16-PSK. Finalmente, los 8 bits de salida del Viterbi son acumulados en una FIFO de salida en palabras de 32 bits y enviados al microprocesador a través del bus AXI.

# Bloques utilizados

## *AXI Stream Width Converter (IP de Xilinx)*

- Función: si bien todos los bloques utilizan un bus AXI Stream para comunicarse, la cantidad de bits en cada uno es variable. Los bloques “AXI Stream Width Converter” cumplen la función de adaptar esa diferencia mediante una FIFO.
- Entrada: depende de lo que se busca adaptar, mínimo 8 bits.
- Salida: depende de lo que se busca adaptar, mínimo es 8 bits.
- Documentación:  
[https://www.xilinx.com/products/intellectual-property/axi4-stream\\_interconnect.html](https://www.xilinx.com/products/intellectual-property/axi4-stream_interconnect.html)

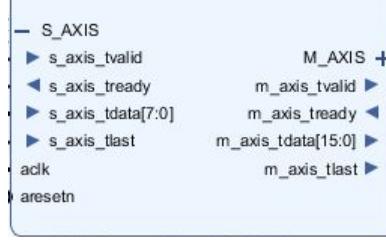


Figura 6. Bloque de conversión de tamaños

## *Adaptador 8 a 2 (desarrollo propio)*

- Función: recibe una señal de 8 bits y envía señales de 2 bits en cada ciclo de clock.
- Entrada: señal de 8 bits.
- Salida: señal de 2 bits.



Figura 7. Bloque de conversión de 8 a 2 bits.

## *Adaptador 4 a 8 (desarrollo propio)*

- Función: recibe una señal de 4 bits y acumula en dos ciclos de clock hasta enviar una salida de 8 bits cada dos ciclos de clock.
- Entrada: señal de 4 bits.
- Salida: señal de 8 bits.

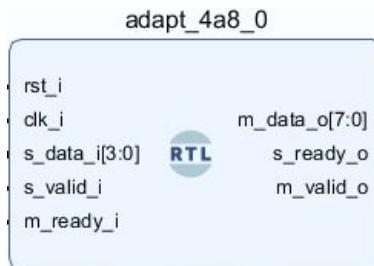


Figura 8. Bloque de conversión de 4 a 8 bits.

## *FIFO AXI Stream (IP de Xilinx)*

- Función: actúan de buffers de entrada y salida ya que el microprocesador de la placa utiliza 32 bits y las primeras y últimas etapas del sistema utilizan 8 bits. Los mismos almacenan hasta 1024 datos que se consideró suficiente para el desarrollo del proyecto.
- Entrada: 32 bits (FIFO in) | 8 bits (FIFO out)
- Salida: 8 bits (FIFO in) | 32 bits (FIFO out)
- Documentación: [https://www.xilinx.com/products/intellectual-property/fifo\\_generator.html](https://www.xilinx.com/products/intellectual-property/fifo_generator.html)

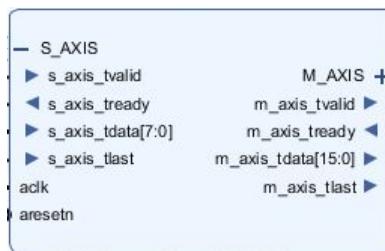


Figura 9. Bloque de conversión de tamaños

## *Convolutional Encoder (IP de Xilinx)*

- Función: codificar las tramas de datos con código octal 171 y 133 con encoder rate de  $\frac{2}{3}$ . El código 171 es 01111001 en binario, lo cual implica que para procesar el dato deberá considerar el

7mo, 6to, 5to, 4to, 2do y 1er bit (indicados con “1”). El código 133 es 01011011 en binario, lo cual implica que para procesar el dato deberá considerar el 7mo, 5to, 4to, 2do y 1er bit (indicados en “1”).

- Entrada: dato de 8 bits.
- Salida: dato de 8 bits.
- Documentación:

[https://www.xilinx.com/products/intellectual-property/convolutional\\_encoder.html](https://www.xilinx.com/products/intellectual-property/convolutional_encoder.html)

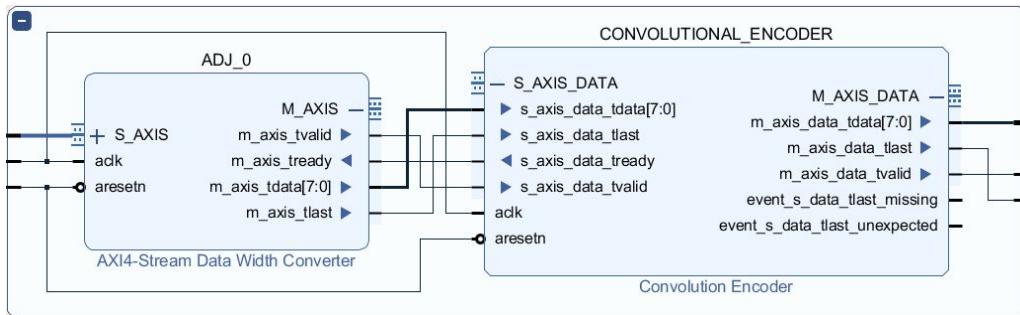


Figura 10. Bloque codificador convolucional

## Mapping 16 PSK (desarrollo propio)

- Función: codificar los símbolos de 8 bits recibidos a la entrada. La codificación se realiza con una tabla precalculada que mapea el valor del símbolo de entrada en coordenadas en un círculo unitario de fase equidistante. Para poder recibir 8 bits de entrada posee una etapa previa que consiste de un adaptador de 8 a 2 bits, donde en un ciclo de clock recibe el dato y demora cuatro ciclos de reloj en transmitir toda la información.
- Entrada: dato de 8 bits.
- Salida: módulo de 8 bits y fase de 8 bits.
- Documentación: adjunta en este trabajo.

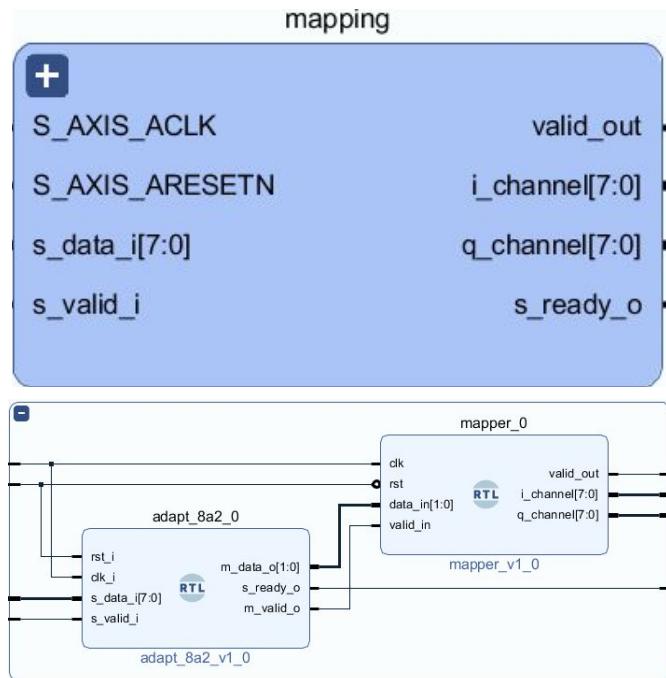


Figura 11. Bloque mapping 16 PSK

Entrada	Salida, ángulo (I,Q)	
	Canal I	Canal Q
0000	126	25
0001	106	71
0010	71	106
0011	25	126
0100	-25	126
0101	-71	106
0110	-106	71
0111	-126	25
1000	-126	-25
1001	-106	-71
1010	-71	-106
1011	-25	-126
1100	25	-126
1101	71	-106
1110	106	-71
1111	126	-25

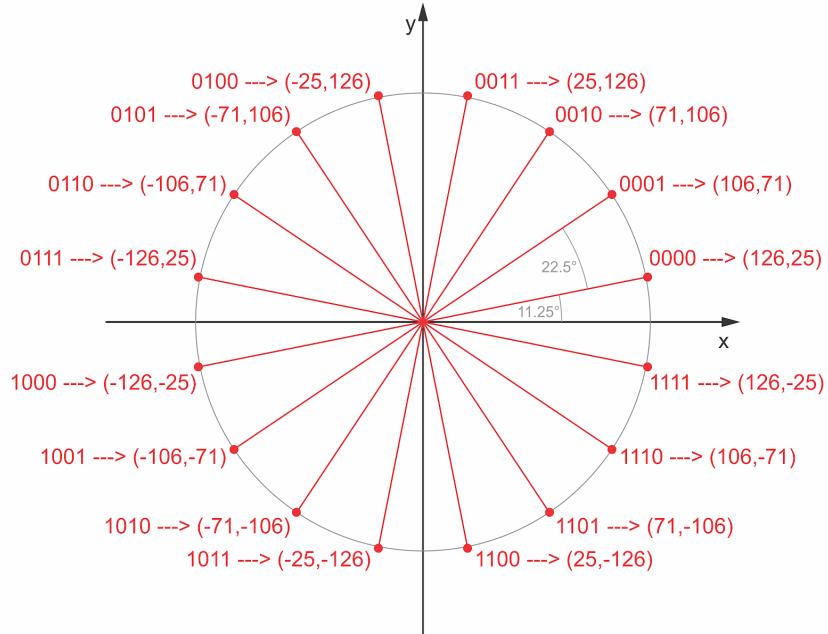


Figura 12. Símbolos de entrada y resultado de mapear el mismo con las tablas precalculadas para I y Q, lo que da como resultado un punto en alguna de las 16 fases equidistantes del círculo unitario.

## Complemento de fase (desarrollo propio)

- Función: para que el bloque FFT funcione como IFFT la documentación de Xilinx indica que debe enviarse el la fase invertida. Entonces se diseñó un módulo que mantiene igual la parte alta del dato (módulo) e invierte los valores de la parte baja del dato (fase).
- Entrada: módulo de 8 bits, fase de 8 bits
- Salida: módulo y fase invertidas concatenados en dato de 16 bits

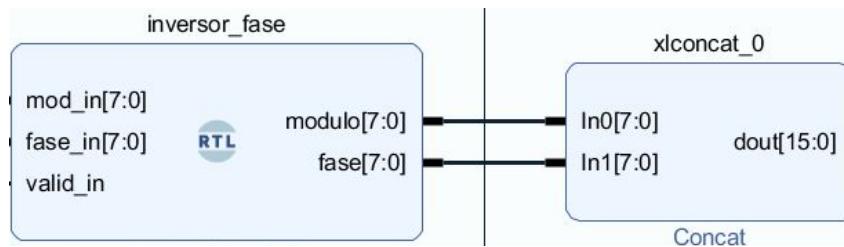


Figura 13. Bloque complemento de fase

## IFFT (IP FFT de Xilinx)

- Función: recibe el dato de 16 bits donde la parte alta es el módulo y la parte baja la fase invertida, haciendo que el bloque de FFT actúe como una IFFT. Con un largo de datos de 64 bits y un solo canal con Radix-2 lite burst I/O. Los datos serán en punto flotante sin escala y truncados. No es necesario conectar la configuración.
- Entrada: dato de 16 bits con módulo y fase invertida
- Salida: dato de 32 bits con la inversa de la transformada ya aplicada
- Documentación:  
[https://www.xilinx.com/support/documentation/ip\\_documentation/xfft\\_ds260.pdf](https://www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf)

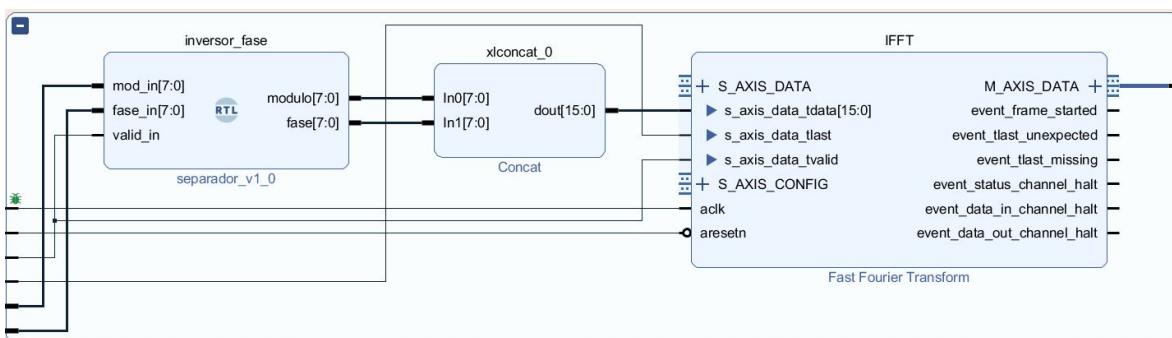


Figura 14. Bloque IFFT

## FFT (IP de Xilinx)

- Función: aplicar la FFT sobre los datos recibidos. Con un largo de datos de 64 bits y dos canales con Radix-2 lite burst I/O. Los datos serán en punto flotante sin escala y truncados. La entrada de configuración no fue especificada en ninguno de los dos bloques ya que por defecto al sintetizar esas entradas se conectan a masa y el bloque actúa con su configuración normal (FFT)
- Entrada: dato de 32 bits que previamente fueron generados por el bloque de IFFT
- Salida: dato de 64 bits donde se anulan los efectos de la IFFT
- Documentación: <https://www.xilinx.com/products/intellectual-property/fft.html>

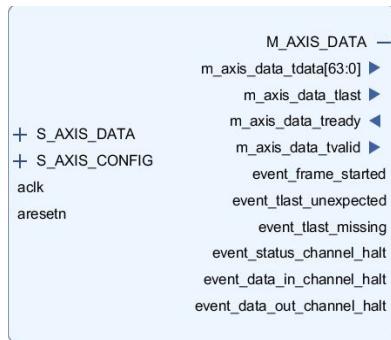
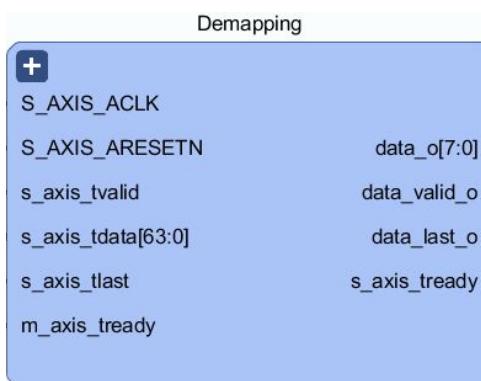


Figura 15. Bloque FFT

## De-Mapping 16 PSK (desarrollo propio)

- Función: recibir el punto en un círculo unitario de 16 fases equidistantes y devolver el símbolo correspondiente.
- Entrada: módulo de 8 bits y fase de 8 bits
- Salida: dato de 8 bits
- Documentación: adjunta en este trabajo.



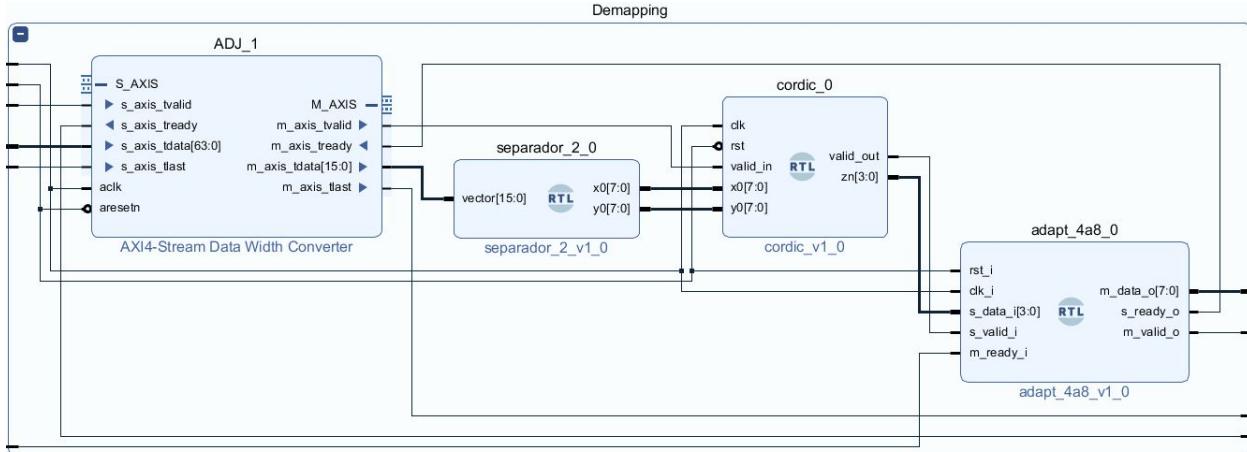


Figura 16. Bloque demapping

El bloque CORDIC, implementado en el archivo “Cordic.vhd”, tiene dos parámetros generic:

- 1) N: ancho en bits del bus de datos de las señales. Valor por defecto  $N=16$ .
- 2) Iter: cantidad instancias de taps instantiados de *cordic\_iter*. Valor por defecto  $Iter=8$ .

El código del bloque se puede dividir en las siguientes partes:

- Declaración de la entidad *cordic*.
- Declaración del componente *cordic\_iter*.
- Declaración de señales de conexión.
- Array de valores de la *arctan()*.
- Declaración de registros de entrada y salida.
- Dos procesos para el control de los registros de entrada y salida.
- Cálculo del ángulo, con corrección del cuadrante de la *arctan()*.
- Instanciación de varios bloques *cordic\_iter* condicional con If generate según el parámetro Iter.
- Conversión de valores dentro de cierta porción de ángulo a binario de 4 bits.

A continuación se desarrollará una explicación de las partes más relevantes:

### Declaración del componente *cordic\_iter*

Se declara utilizando los valores generic por defecto,  $N=16$  y  $SHIFT=1$ .

Este componente se encuentra desarrollado en el archivo “*cordic\_iter.vhd*” contiene la lógica de un tap del algoritmo CORDIC. Su estructura se observa en la Figura 17.

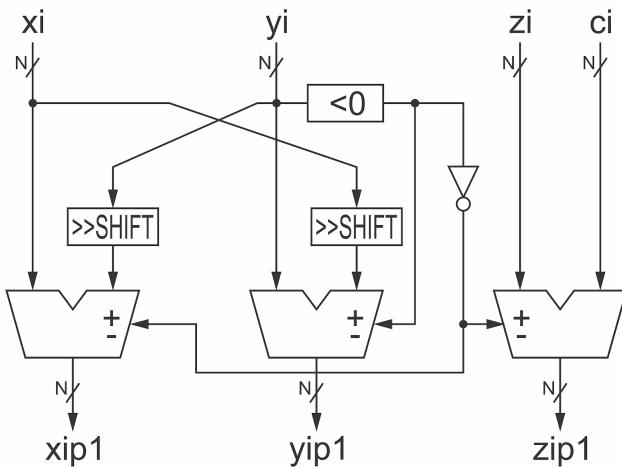


Figura 17. Estructura de un tap del algoritmo CORDIC.

Contiene 3 Sumadores/Restadores, 2 Shift registers SHIFT veces, 1 comparador y 1 negador.

El valor de  $ci$  conviene del array de  $\text{arctan}()$  (Figura 16).

Este bloque tiene dos parámetros generic:

- 1) N: ancho en bits del bus de datos de  $xi$ ,  $yi$ ,  $zi$ ,  $ci$ ,  $xip1$ ,  $yip1$  y  $zip1$ . Valor por defecto  $N=16$ .
- 2) SHIFT: cantidad de bits que se mueve el dato a derecha. Valor por defecto  $SHIFT=1$ .

### Array de valores de la $\text{arctan}()$

En la Figura 18 se observan los valores de la tabla precalculada para  $\text{arctan}()$  según el Nro. de Taps.

LUT de $\text{arctan}()$	
N (Nº taps)	$\text{arctan}(N)$
0	100
1	59
2	31
3	15
4	7
5	3
6	1
7	0

Figura 18. Tabla precalculada de  $\text{arctan}$  según N.

## Cálculo del ángulo, con corrección del cuadrante de la arctan()

En la Figura 19 se muestran las correcciones de ángulo para ingresar en el CORDIC según el cuadrante en el que se encuentre el ángulo definido por los valores de  $x$  e  $y$ . Por ejemplo, si el ángulo está en el segundo cuadrante, es decir, los valores de entrada son  $x\_reg < 0$  y  $y\_reg > 0$  se pasa el ángulo al primer cuadrante utilizando como ángulo de salida (el que se inyecta en el CORDIC para el cálculo) los valores formados por  $(x0in, y0in) = (y\_reg, -x\_reg)$  y un  $offset = 201$  ( $201$  representa  $+90^\circ$ ) a sumar luego de obtener el ángulo de salida  $z$ .

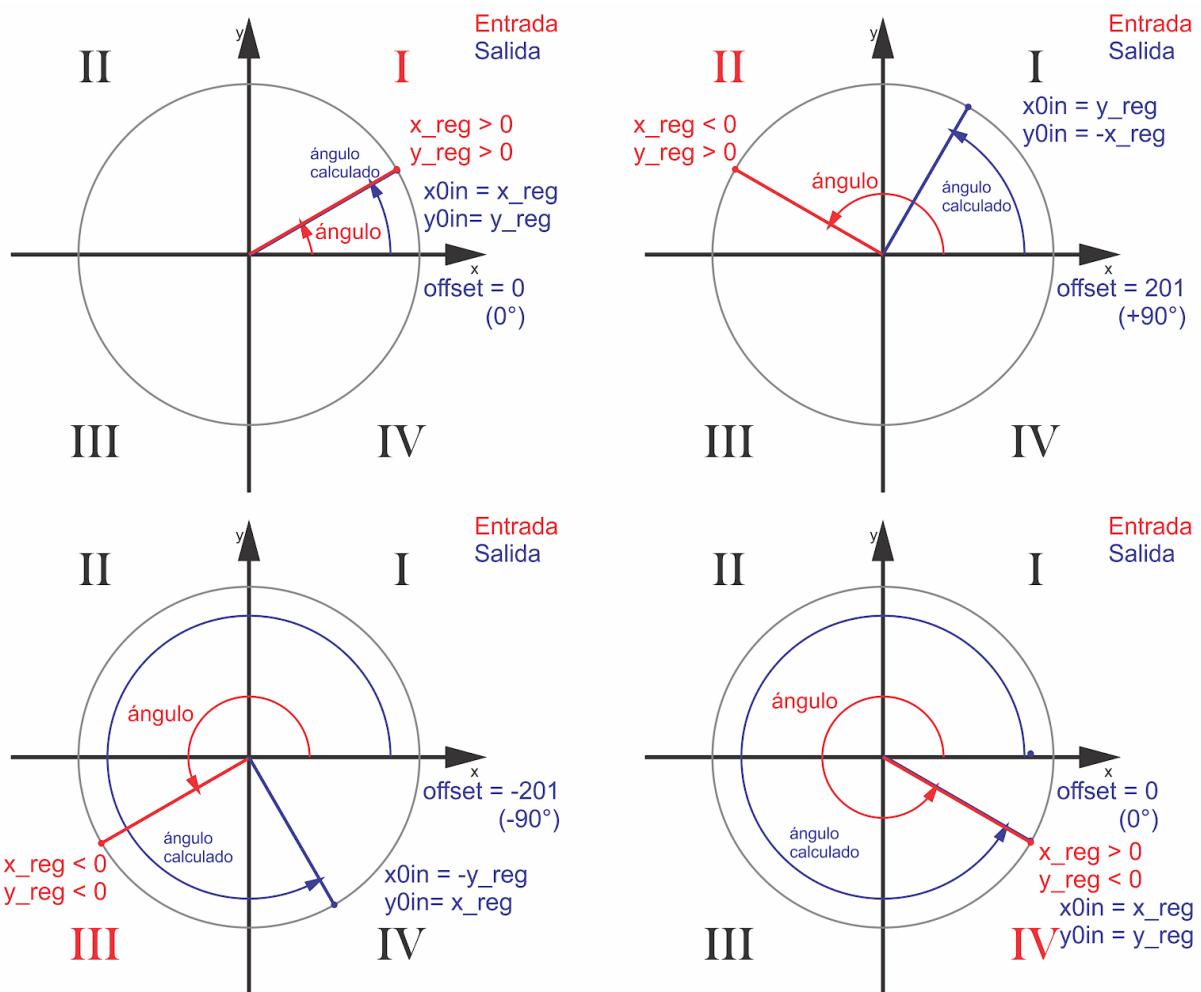


Figura 19. Correcciones de ángulos según el cuadrante.

En la Figura 20 se exponen los circuitos equivalentes al código VHDL encargado la corrección de los ángulos.

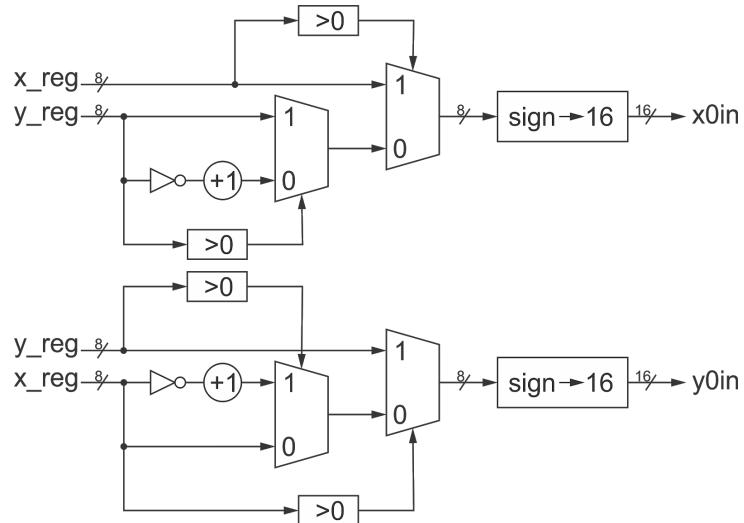


Figura 20. Circuitos de corrección de ángulos según el cuadrante.

En la Figura 21 se exponen las partes que componen el circuito y sus funcionalidades.

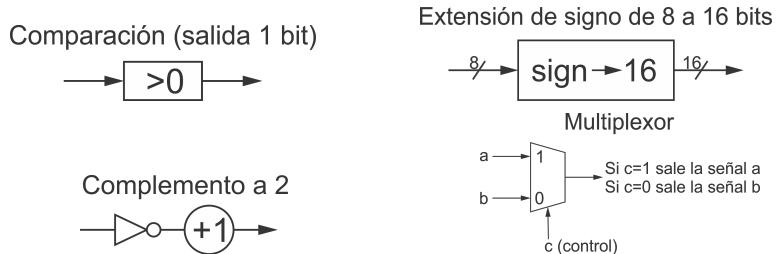


Figura 21. Partes que componen el circuito de la Figura 20.

### Instanciación de N bloques *cordic\_iter*

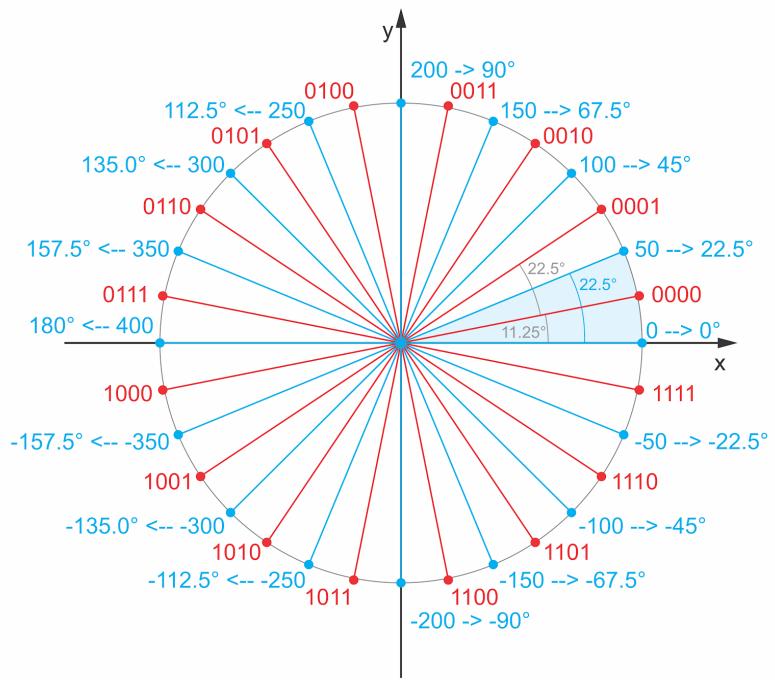
La instanciación es automática con If generate condicional según el parámetro Iter. Instancia Iter=8 taps del “*cordic\_iter*” cambiando las conexiones de entrada en el tap inicial, conectando todos los taps intermedios y cambiando las conexiones de salida del tap final.

### Conversión de valores dentro de cierta porción de ángulo a binario de 4 bits

Utilizando como entrada el índice (*index*) calculado mediante la salida del CORDIC (*znout*) y sumando el offset de corrección de *arctan()* se utiliza la siguiente tabla para mapear de 16 porciones de ángulos a valores binarios de 4 bits (salida) como se muestra en la Figura 22.

Entrada indice	Salida (4 bits)
index = znout + offset	
0 <= index < 50	0000
50 <= index < 100	0001
100 <= index < 150	0010
150 <= index < 200	0011
200 <= index < 250	0100
250 <= index < 300	0101
300 <= index < 350	0110
350 <= index < 400	0111
-50 < index < 0	1111
-100 < index < -50	1110
-150 < index < -100	1101
-200 < index < -150	1100
-250 < index < -200	1011
-300 < index < -250	1010
-350 < index < -300	1001
-400 < index < -350	1000

a)



b)

Figura 22. a) Tabla de conversión de index a salida de 4 bits (demapping final).

Nótese que 0 corresponde a  $0^\circ$  mientras que 50 corresponde a  $22.5^\circ$  y así continúa linealmente.

## Viterbi (IP de Xilinx)

- Función: decodificar la trama de datos con el mismo código de codificación que se utilizó en el codificador convolucional (171 y 133 en octal) para reconstruir la señal original. Se solicitó una licencia de prueba del viterbi privativo de Xilinx que dura 120 días, para tener plena compatibilidad con el encoder que también ofrece Xilinx en su catálogo de IP.
- Entrada: dato de 16 bits
- Salida: dato de 8 bits
- Documentación: [https://www.xilinx.com/products/intellectual-property/viterbi\\_decoder.html](https://www.xilinx.com/products/intellectual-property/viterbi_decoder.html)

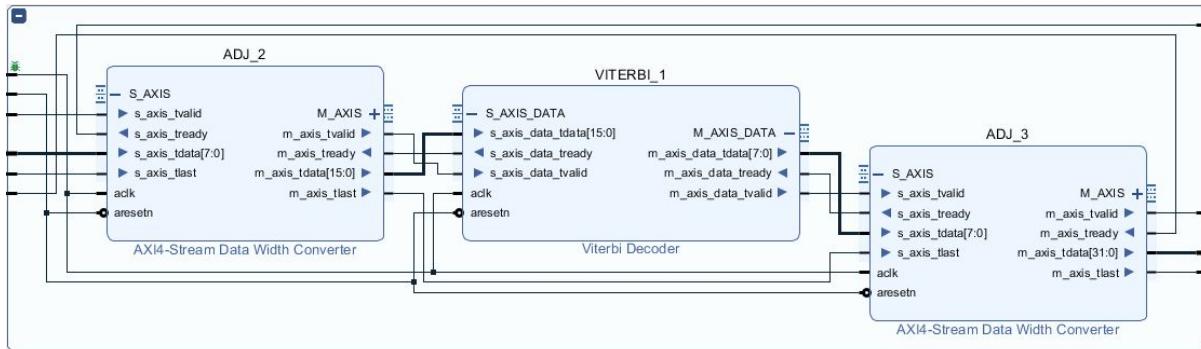


Figura 23. Bloque decodificador viterbi

# Utilidades

*Script en python para envío y recepción de datos*

```
import numpy as np
import socket
import time as t
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('192.168.1.10', 7))

data = np.random.randint(0,16,16)

print(data)
start = t.time()
s.send(data.tobytes())
end = t.time() + 0.0001
dataLen = len(data.tobytes())
dataRate = dataLen / (end - start) * 8
print('Bytes sent: {0!s} @{1!s}'.format(dataLen,dataRate))

start = t.time()
recvData = s.recv(len(data.tobytes()))
end = t.time() + 0.0001
dataLen = len(recvData)
dataRate = dataLen / (end - start) * 8
print('Bytes received: {0!s} @{1!s}'.format(dataLen,dataRate))

rxData = np.frombuffer(recvData, dtype=int)
print(rxData)
```

# Interfaz Alternativa de PC a Lógica Programable

Como interfaz alternativa para comprobar el funcionamiento se desarrolló esquema de la Figura 24.

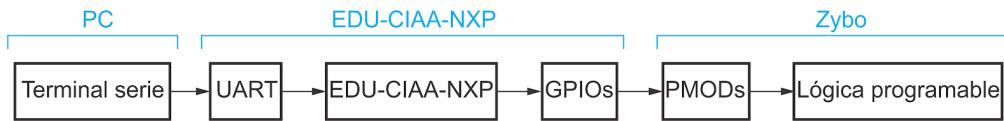


Figura 24. Esquema conceptual.

Se conecta la EDU-CIAA por USB a la PC para ver una terminal serie (u opcionalmente se observa con los LEDG = ON si lo que transmitido es recibido correctamente). Luego de GPIOs de la EDU-CIAA a los PMODs de la FPGA Zybo. En la Figura 25 se expone una fotografía del conexionado, se utilizan puertos de 8 pines (1 byte) para la entrada y salida de datos.

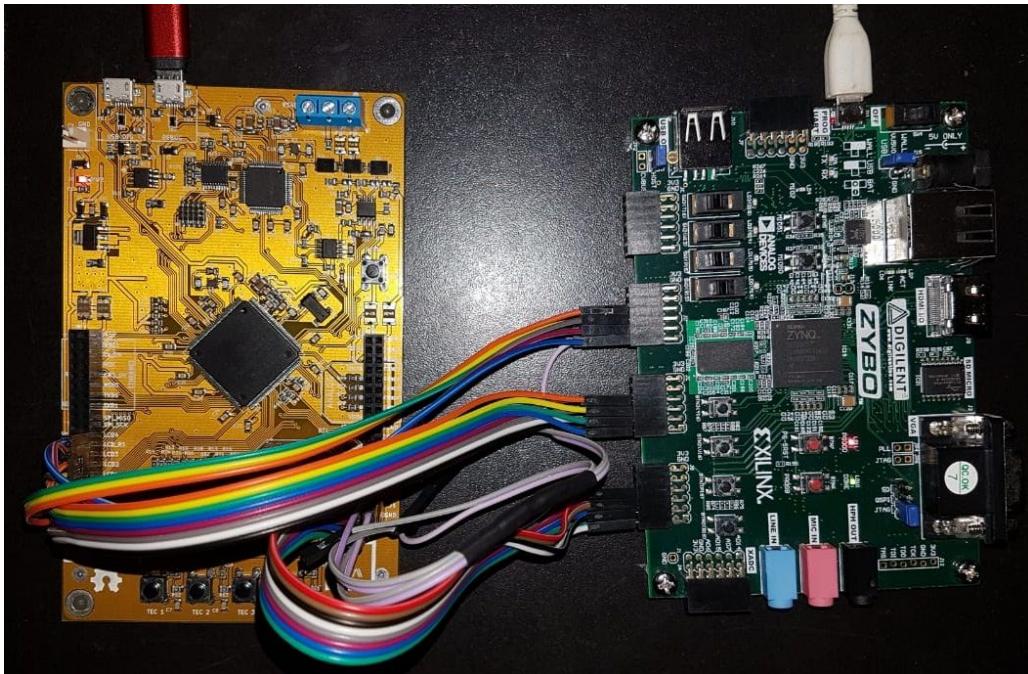


Figura 25. Conexión entre EDU-CIAA-NXP y Zybo.

Los pines de la Zybo elegidos se conectan directamente a la Lógica Programable (PS) saltando la parte del Sistema de Procesamiento (PS) completamente. De esta forma puede programarse directo desde Vivado sin usar el SDK, generando bitstream y se descarga desde allí a la placa Zybo.

Mediante TEC1 se inicia el envío de datos y con TEC3 la recepción de datos. En la Figura 27 se puede observar los datos enviados en la terminal:

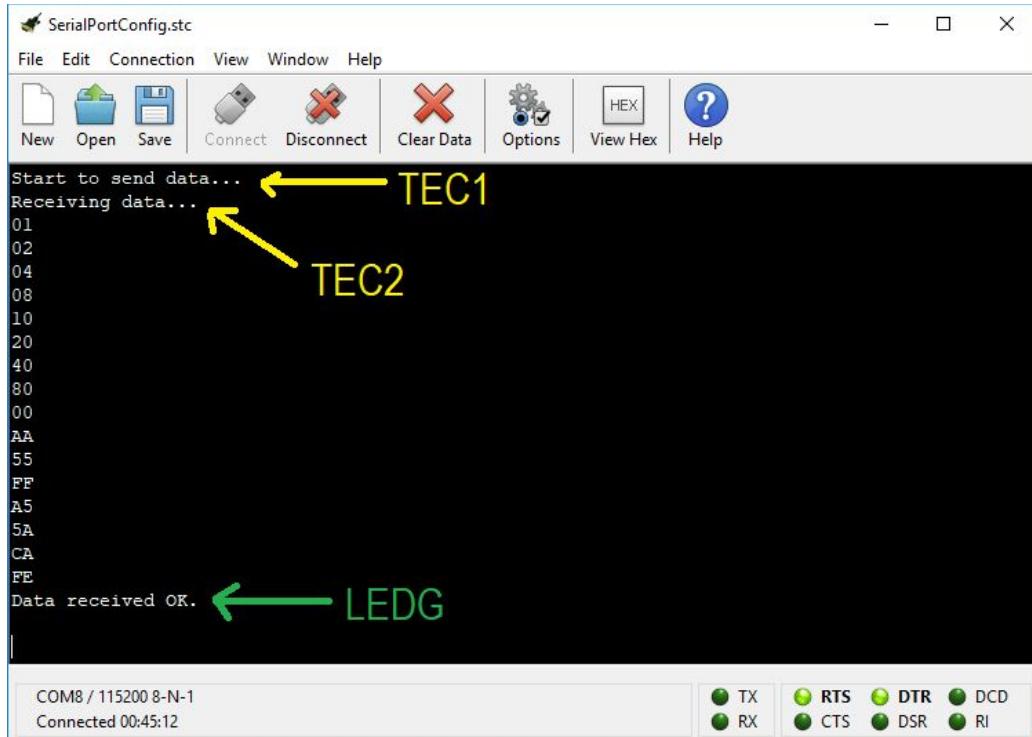


Figura 26. Terminal serie.

Se implementó un programa en C que genera una interfaz AXI Stream por software en la EDU-CIAA usando GPIOs. En la figura 27 se muestra un diagrama de esta interfaz.

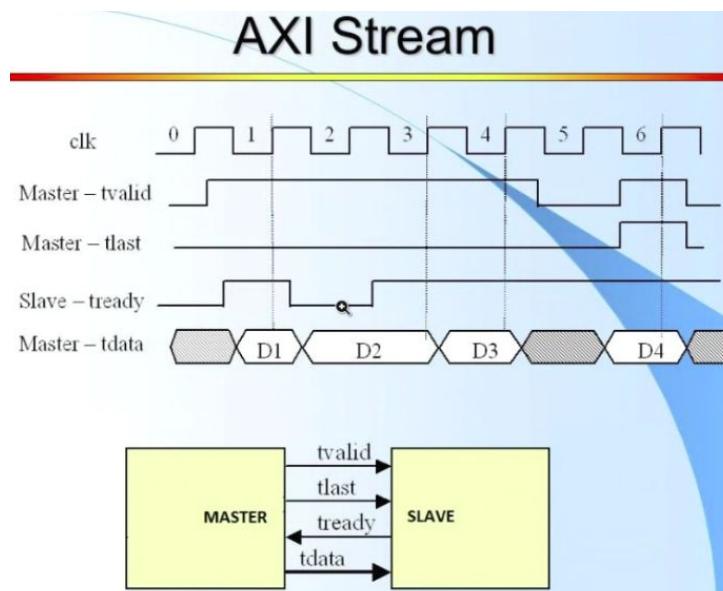


Figura 27. Interfaz AXI Stream.

La EDU-CIAA implementa ambos, el master para enviar datos hacia la Zybo y el slave para recibir lo que retorna la Zybo. En la Figura 28 se muestra un esquema de lo mismo y el pinout utilizado.

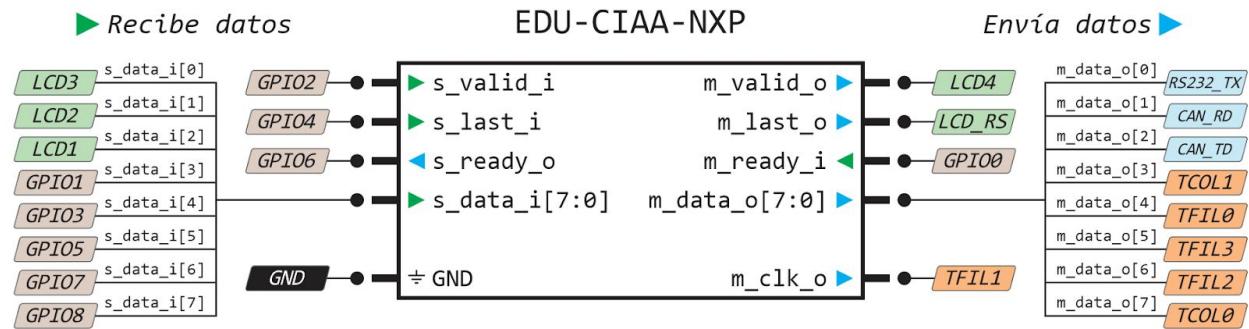


Figura 28. EDU-CIAA-NXP interfaz AXI Stream.

En la figura 29 se muestra el pinout del de la Zybo y donde se conecta con la EDU-CIAA-NXP.

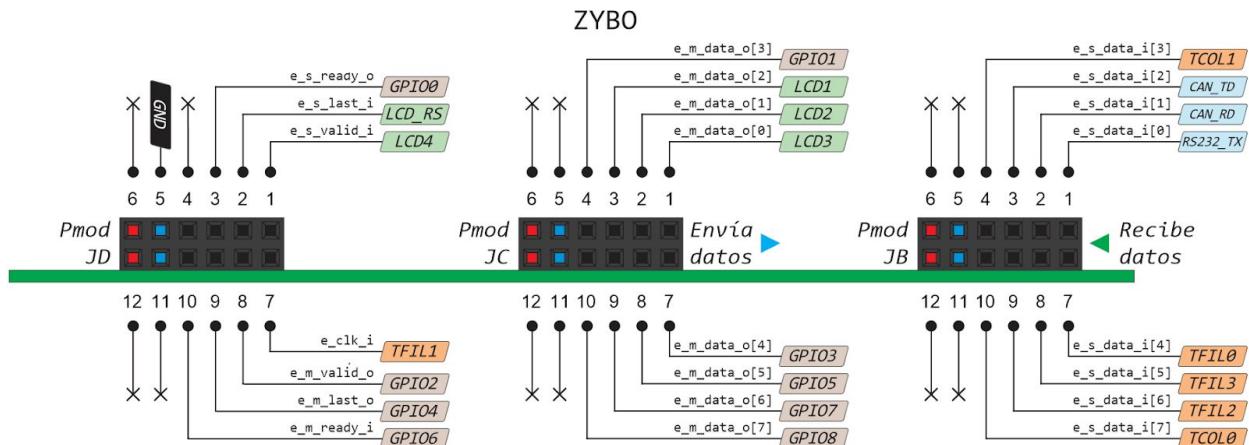


Figura 29. Conexión de PMODs de Zybo a EDU-CIAA-NXP.

La mayoría de los bloques utilizados en el el proyecto de Vivado implementan esta misma interfaz como se muestra en la Figura 30.

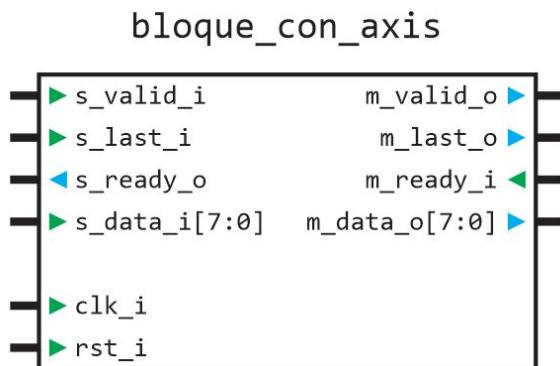


Figura 30. Bloque con interfaz AXI Stream.

## Verificación

Para verificar el funcionamiento del programa que corre en la EDU-CIAA-NXP se conectan todos los pines *master* a todos los *slave* manteniendo el orden como se observa en la fotografía de la Figura31.

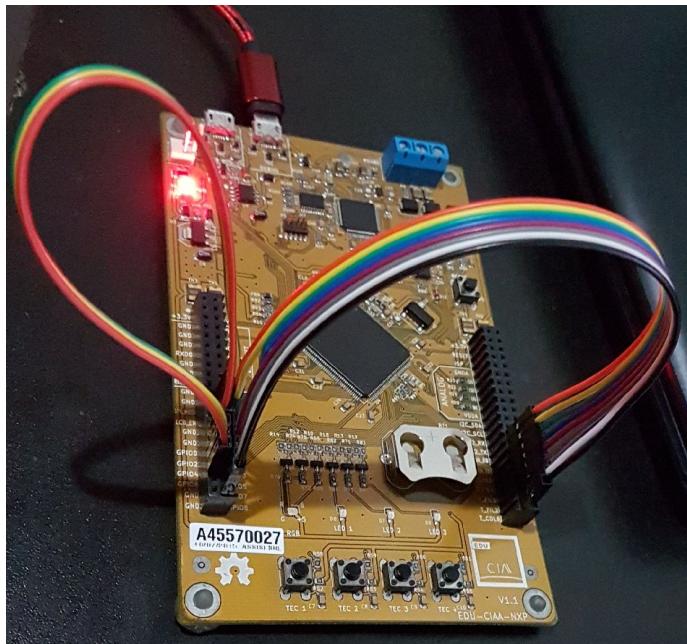


Figura 31. EDU-CIAA-NXP: conexión en bucle.

Al usar solo la FPGA el programa tarda muchísimo menos en sintetizar, implementar y generar el bitstream acelerando mucho las pruebas.

El programa de la EDU-CIAA-NXP se puede descargar:

[https://github.com/epernia/firmware\\_v3/tree/master/examples/c/sapi/gpio/axis\\_interface](https://github.com/epernia/firmware_v3/tree/master/examples/c/sapi/gpio/axis_interface)

En el siguiente enlace se puede descargar un proyecto de Vivado para utilizar esto:

<https://drive.google.com/open?id=1xNueIOUYKrrYduR6wbBnxn7EHuIClhdm>

Para portarlo a otra placa de FPGA de Xilinx básicamente se debe cambiar en el proyecto la *Board* y luego el archivo de *constraints* (el \*.xdc).

En la Figura 32 se expone el *block design* utilizado para las pruebas.

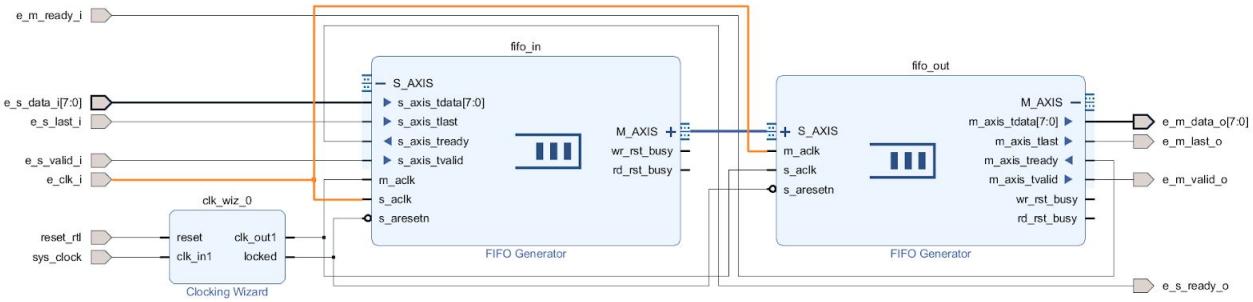


Figura 32. Block Design en en Vivado sin PS.

Entre FIFO IN y FIFO OUT se deben colocar todos los bloques del sistema OFDM.

Cabe destacar que el pin de clock no puede ser uno cualquiera dado que hay ciertos pines que puede implementar como entrada de clocks a los bloques, el *e\_clk\_i* es el pin *U14* de la FPGA que soporta modo *SRCC* como se observa en *constraint*:

```
#set_property -dict { PACKAGE_PIN U14    IOSTANDARD LVCMS33 } [get_ports {  
e_clk_i }]; #IO_L11P_T1_SRCC_34 Sch=JD3_P
```

Nótese que en el comentario se ve que soporta modo *SRCC*.