ACELERACIÓN DE SDR MEDIANTE EL USO DE RFNOC

KEVIN JOHAN OVIEDO RUEDA
JORGE ANDRÉS SANTOS RUEDA

CPS | RESEARCH GROUP

Universidad Industrial de Santander
Facultad de Ingenierías Fisicomecánicas
Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones
Bucaramanga
2018

ACELERACIÓN DE SDR MEDIANTE EL USO DE RFNOC

KEVIN JOHAN OVIEDO RUEDA
JORGE ANDRÉS SANTOS RUEDA

Trabajo de grado presentado como requisito para optar al título de

Ingeniero Electrónico

Director:
Óscar Mauricio Reyes Torres
Doctor en Ingeniería Electrónica

Codirector:
Carlos Andrés Angulo Julio
Magíster en Ingeniería Electrónica

Universidad Industrial de Santander
Facultad de Ingenierías Fisicomecánicas
Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones
Bucaramanga
2018

Dedicado a

A mi mamá, mis abuelos, mis tías, mi tio y mis primos, por su apoyo y cariño en todo momento

Kevin Johan Oviedo Rueda.

Dedicado a

mis padres, Jorge y Cristina, por todas sus enseñanzas, consejos y apoyo; por siempre estar ahí cuando más lo he necesitado; por haberme formado y brindado amor incondicional.

mi hermana María Paula, mi compañera de sueños, por apoyarme siempre y empujarme a mejorar como persona y profesional.

a mis abuelos Jorge y Myriam, primas Ju y Sil, tios políticos Kelly y Sash y no políticos Juanca y Luz por su apoyo, amor incondicional y por demostrarme que somos el mejor "equipa".

mi novia, Alejandra Aranda, por todo su cariño, compañía y ser mi soporte en momentos difíciles.

mi guía espiritual, Olga Lu, por darme tranquilidad en los momentos más complicados.

A todos mis compañeros y personas que formaron parte de mi desarrollo profesional y personal.

Jorge Andrés Santos Rueda.

CONTENIDO

LISTA DE FIGURAS

Pág.

LISTA DE CUADROS

Pág.

LISTA DE ANEXOS

Pág.

RESUMEN

TÍTULO: Aceleración de SDR mediante el uso de RFNoC *

AUTORES: Kevin Johan Oviedo Rueda **
Jorge Andrés Santos Rueda **

PALABRAS CLAVE: Aceleración de Software, FFT, FPGA, GNU-Radio, HDL, RFNoC, Software Defined Radio, USRP.

DESCRIPCIÓN:

Los sistemas de radio definido por software como los USRP serie X300 o E300 contienen un FPGA de altas prestaciones, que de fábrica, se utiliza para el procesamiento básico de señales en procesos como *Digital Down Conversion* y *Digital Up Conversion*, los cuales no utilizan la totalidad de los recursos del FPGA. Los recursos restantes podrían aprovecharse para ejecutar algoritmos que normalmente se llevan a cabo en el procesador del servidor, especialmente aquellos que se beneficien de la paralelización que los FPGAs ofrecen. En este trabajo se propuso RFNoC como arquitectura integradora que permite el procesamiento heterogéneo en SDR; para sustentar esta propuesta, primero se buscó realizar una verificación del entorno utilizando la transformada rápida de Fourier como prueba piloto, documentando la arquitectura y el proceso de implementación. Después se realizaron pruebas que verificaron el correcto funcionamiento del bloque; estas pruebas consistieron en el análisis de propiedades y medición de exactitud con respecto a su versión teórica. Finalmente se compararon resultados obtenidos del bloque RFNoC con respecto al bloque FFT de GNU-Radio los cuales permitieron indicar las situaciones donde el FPGA o el servidor destacan. Estos resultados ofrecen a los desarrolladores de SDR un método para aprovechar el potencial del FPGA en determinados momentos y podría expandir el uso de SDR a nuevas áreas de aplicación, cumpliendo la función de diversos dispositivos de propósito específico a los que podría reemplazar.

*Trabajo de investigación.

** Facultad de Ingenierías Fisicomecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Director: Óscar Mauricio Reyes Torres. Doctor en Ingeniería Electrónica.

ABSTRACT

TITLE: SDR Acceleration by using RFNoC *

AUTHORS: Kevin Johan Oviedo Rueda **
Jorge Andrés Santos Rueda **

KEYWORDS: FFT, FPGA, GNU-Radio, HDL, RFNoC, Software-acceleration, Software Defined Radio, USRP.

DESCRIPTION:

Software-defined radio systems such as the USRP series X300 or E300 contain a high-performance FPGA used by default for basic processing of signals in processes such as *Digital Down Conversion* and *Digital Up Conversion*, which do not use all the FPGA resources. The remaining resources could be used to execute algorithms that are normally carried out in the server processor, especially those that benefit from the parallelization that FPGAs offer. In this thesis, RFNoC was proposed as an integrating architecture that allows heterogeneous processing in SDR. To support this proposal, first it was sought to verify the environment using the fast Fouier transform as a pilot test, documenting the architecture features and the implementation process. After that, tests were carried out to verify the correct behavior of the block. It were consisted of properties analysis and accuracy measurement with respect to its theoretical version. Finally, results obtained from the RFNoC block were compared with respect to the GNU-Radio FFT block, which allowed to indicate where the FPGA or the server procesor stand out. These results offer SDR developers a method to harness the potential of the FPGA at certain times and could expand the use of SDR to new application areas, fulfilling the function of various specific purpose devices that it could replace.

---

*Bachelor Thesis

** Facultad de Ingenierías Fisicomecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Director: Óscar Mauricio Reyes Torres. Doctor en Ingeniería Electrónica.

INTRODUCCIÓN

Actualmente, la tecnología conocida como *Software Defined Radio* (*SDR*) ha tomado mayor importancia en el mundo de las comunicaciones[1]. Esta tecnología tiene como fin acercar lo máximo posible el procesamiento digital de la señal a la antena, convirtiendo los problemas que hasta ahora habían recaído en hardware (moduladores, amplificadores, filtros, etc.) en problemas tipo software, es decir, las señales son procesadas en el dominio digital en lugar del dominio analógico como en los radios convencionales[2].

Aunque el término *SDR* sugiere requisitos establecidos únicamente por software, es necesario considerar el hardware adicional utilizado para convertir señales del dominio analógico al digital. Esto se consigue mediante el uso de conversores analógico digital (*ADC*) y conversores digitales descendentes (*DDC*)[3]. El *ADC* es un convertidor que toma la señal analógica recibida por la antena y la separa en muestras, obteniendo una versión discreta de la misma; el *DDC* se encarga de eliminar la portadora de la señal tomada desde la antena, obteniendo una señal banda base que corresponde con la frecuencia original de la envolvente compleja[4]. Estos dispositivos se encuentran embebidos por lo general en periféricos como el "Universal Software Radio Peripheral" (*USRP*).

Para la realización de este trabajo de investigación se hizo uso del *USRP* de tercera generación X310 cuya arquitectura permite muestrear señales hasta un máximo de $200MSPS$, con la posibilidad de sincronización con un GPS/Multi-GNSS Disciplined

[1] HICKLING, Ronald. New technology facilitates true software-defined radio. RF Design Magazine. 2005

[2] MONTERO, Juan. Implementación de un sistema de comunicaciones basado en Software Radio. Universidad Autónoma de Madrid. 2014

[3] DHAR, Rahul; GEORGE, Gesly; MALANI, Amit; STEENKISTE, Peter. Supporting integrated MAC and PHY software development for the USRP SDR. Networking Technologies for Software Defined Radio Networks, 2006. SDR'06.1 st IEEE Workshop on. p 68-77.

[4] REYMUND, Thomas. Software Defined Radio with Graphical User Interface. Universidad Tecnológica de Viena. 2007

Oscillator para mayor precisión[5]. Contiene un *FPGA* Kintex-7 XC7K410T que es un dispositivo programable de alto desempeño con múltiples elementos digitales. En el Cuadro 1 se muestra el nombre y la cantidad de cada elemento utilizable en el FPGA. Los dispositivos USRP Ofrecen compatibilidad con diversas interfaces de *SDR* como *GNU-Radio*, C++, Amarisoft LTE 100 y OpenBTS[6].

Cuadro 1: Recursos FPGA Kintex-7 XC7K410T

Recurso	Elemento	Cantidad
Recursos lógicos	Slices	63,550
	Celdas lógicas	406,720
	CLB Flip-Flops	508,400
Recursos de memoria	Máxima RAM distribuida (Kb)	5,663
	Bloques RAM/FIFO w/ECC (36 Kb cada uno)	795
	Memoria total en bloques RAM (Kb)	28,620
Recursos de reloj	CMTs (1 MMCM + 1 PLL)	10
Recursos de E/S	Número máximo de E/S con salida simple	500
	Número máximo de E/S con pares diferenciales	240
Recursos de IP integrados	DSP48 Slices	1,540
	PCIe Gen2	1
	Mezclador de señales analógicas(AMS)	1
	Configuración AES/ bloques HMAC	1
	Transceptores GTX	16
Grados de velocidad	Uso comercial(C)	-1, -2
	Uso extendido (E)	-2L, -3
	Uso industrial	-1, -2, -2L

GNU-Radio es un software de desarrollo libre el cual provee bloques de procesamiento de señal para sistemas *SDR*, utilizado con hardware como el *USRP*, o sin hardware en un ambiente de simulación. *GNU-Radio* utiliza el procesador de propósito general (*GPP*) del computador o dispositivo donde esté instalado para realizar el tratamiento de señales, limitando su rendimiento y velocidad, siendo comple-

[5]ETTUS KNOWLEDGE BASE CONTRIBUTORS. GPSDO Selection Guide. En: Ettus Knowledge Base [En línea]. (2017) [Consultado 10 Dic. 2017]. Disponible en: `https://kb.ettus.com/GPSDO_Selection_Guide`

[6]TRUONG, Nguyen; SUH, Young-Joo; YU, Chansu. Latency analysis in GNU radio/USRP-based software radio platforms. Military Communications Conference, MILCOM 2013-2013 IEEE. p 305-310.

tamente independiente del hardware utilizado para obtener y convertir las señales[7].

Lo anterior podría indicar que en sistemas *SDR* existen mejores beneficios al invertir para mejorar el *GPP* del computador o dispositivo que aloja el software, que invertir para mejorar el hardware que obtiene y convierte las señales. Con RFNoC (*Radio Frequency Network on Chip*) es posible distribuir el procesamiento de las señales en las dos plataformas de hardware (*GPP* y *FPGA*) de manera simple y según lo más adecuado para la aplicación. Esto convierte al *USRP* X310 en algo más que un adaptador para el procesamiento de señales de *SDR*, lo convierte en una plataforma de procesamiento misma[8].

De esta forma, se pretende realizar una implementación con la arquitectura *RFNoC* y de esta manera contrastar resultados con los bloques tradicionales de *GNU-Radio*. Para lo cual se recurre inicialmente a una búsqueda de aplicaciones realizadas con la herramienta, donde sea posible hacer una revisión interna del funcionamiento. Posteriormente, se realiza una condensación de la información ofrecida por estas aplicaciones, documentando los pasos necesarios para realizar cualquier aplicación. La tercera etapa del proyecto consiste en implementar la transformada rápida de Fourier (FFT) utilizando un *Intellectual Property Core* (*IP-CORE*) diseñado por Xilinx, que sirva como ejemplo para verificar lo documentado. En la siguiente etapa, se agregan a la documentación todas las consideraciones adicionales tenidas en cuenta en la construcción del bloque.

En la quinta etapa se verifica el funcionamiento del bloque, midiendo su exactitud a través de una comparación con la transformada discreta de Fourier (DFT) y también verificando propiedades como linealidad, desplazamiento temporal y la respuesta al impulso. Para finalizar, se realiza una comparación con el bloque *FFT* de *GNU-Radio*, con pruebas como tiempo de procesamiento y latencia.

[7]MARWANTO, Arief; SARIJARI, Mohd; FISAL, Norsheila. Experimental study of OFDM implementation utilizing GNU Radio and USRP-SDR. Communications (MICC), 2009 IEEE 9th Malaysia International Conference on. p 132-135.

[8]BRAUN, Martin; PENDLUM, Jonathon; ETTUS, Matt. RFNoC: RF network-on-chip. Proceedings of the GNU Radio Conference. 2016

Se seleccionó la FFT como bloque de pruebas por la facilidad de verificación del correcto funcionamiento, debido a la similitud de implementación con la Transformada Inversa de Fourier (IFFT). Esto permite la conexión de estos bloques en cascada como método de prueba, siendo posible obtener la señal de entrada en la salida del diagrama de bloques.

Esta metodología no sólo ofrece resultados de funcionamiento, rendimiento y comparación entre las dos arquitecturas, sino que también, simplifica el procedimiento para futuros análisis e implementaciones; permitiendo extender la utilidad de los USRP más allá de la posibilidad de ser una plataforma adicional de procesamiento.

El trabajo se organiza por capítulos de la siguiente manera: en el capítulo uno se desarrollan las generalidades del proyecto, y en los anexos puede encontrarse información adicional muy útil que complementa este capítulo, incluyendo partes y niveles de desarrollo. En el capítulo dos se muestra la definición, implementación, síntesis y simulación de la *FFT*. En el capítulo tres se presenta una guía paso a paso para realizar una implementación básica de RFNoC y en el cuatro, cinco y seis, los resultados de las pruebas realizadas, las conclusiones y recomendaciones.

1. GENERALIDADES

1.1 RFNOC

RFNoC es un entorno de trabajo que implementa una arquitectura, del mismo nombre, que permite el procesamiento heterogéneo de datos; es decir, habilita el uso de varios tipos de procesadores para ejecutar aplicaciones de manera eficiente, asignando la carga de trabajo apropiada en el procesador adecuado. Los algoritmos de procesamiento de señales contienen módulos conocidos como bloques RFNoC que proporcionan una interfaz para encapsular cualquier bloque con *Hardware Description Language* (*HDL*)[9], los cuales son totalmente independientes de RFNoC y se pueden diseñar con cualquier herramienta que admita interfaces de flujo *AXI-STREAM*, incluidas VHDL, Verilog y Xilinx Vivado HLS[10]. A lo largo del documento se hará referencia a RFNoC como un entorno de trabajo, una arquitectura o un bloque dado el contexto.

1.2 PROTOCOLO AXI

AXI hace parte de la familia de protocolos *ARM AMBA*, los cuales son un estándar abierto de conexión de bloques funcionales en *System-on-Chip* (*SoC*), estos facilitan el desarrollo de diseños de procesadores con gran número de controladores y periféricos debido a que permite la reutilización de *IP-CORES* proporcionado por cualquier grupo de diseño en cualquier proyecto. La reutilización de *IP-CORES* es muy importante para reducir el tiempo y costo de diseños *SoC*[11]. El estándar utilizado dentro de la arquitectura de RFNoC es AXI4-STREAM (también llamado *AXI-STREAM*).

[9] PENDLUM, Jonathon. Rfnoc deep dive: Fpga side [En línea]. (2014) [Consultado 10 Dic. 2017]. Disponible en: https://www.ettus.com/content/files/RFNoC_Wireless_at_VT_FPGA.pdf

[10] ETTUS KNOWLEDGE BASE CONTRIBUTORS. RFNoC. En: Ettus Knowledge Base [En línea]. (2017) [Consultado 23 Abr. 2018]. Disponible en: https://www.ettus.com/content/files/RFNoC_Wireless_at_VT_FPGA.pdf

[11] ARM. AMBA Protocol. En: arm Developer [En línea]. (2010) [Consultado 10 Dic. 2017]. Disponible en: https://developer.arm.com/products/architecture/amba-protocol

1.2.1 AXI-STREAM Es usado como interfaz estándar para comunicar componentes donde se vayan a transferir datos. Esta interfaz puede ser usada para conectar un maestro que proporciona los datos, con uno o múltiples esclavos que los reciben. El protocolo admite múltiples flujos de datos usando el mismo conjunto de cables compartidos, que permiten construir una interconexión única para realizar operaciones de aumento y reducción de tamaño de datos[12].

El protocolo utiliza algunas señales estándar que no son obligatorias en su totalidad. La utilización de las señales opcionales depende exclusivamente de la aplicación. Estas señales se definen en el Cuadro 2. Donde n representa la cantidad de bytes; i, d, y u son valores seleccionados por el usuario que dependen de la aplicación.

Cuadro 2: Señales del protocolo *AXI-STREAM*

Señal	Definición
ACLK	Es la señal del reloj global, todas las señales son muestreadas en el flanco ascendente.
ARESETn	Es la señal de reinicio global, es un reinicio activo en alto
TVALID	Indica que el maestro está listo para enviar datos válidos
TREADY	Indica que el esclavo está listo para recibir los datos, la transferencia de un paquete ocurre cuando tanto la señal de *TVALID* como de *TREADY* estan activas.
TDATA[(8n-1):0]	Es la señal principal que es usada para proveer los datos que están pasando a través de la interfaz, la longitud debe ser un número entero de bytes
TSTRB[(n-1):0]	(Opcional) Indica si el contenido del byte de *TDATA* se procesa como un byte de datos o un byte de posición

[12]ARM. AMBA 4 AXI4-Stream Protocol. En: arm Developer [En línea]. (2010) [Consultado 10 Dic. 2017]. Disponible en: https://developer.arm.com/docs/ihi0051/latest/amba-axi4-stream-protocol-specification-v10

Cuadro 2: (Continuación)

Señal	Definición
TKEEP[(n-1):0]	(Opcional) Indica si el contenido del byte de *TDATA* es procesado como parte de una cadena de datos. Los bytes cuyos *TKEEP* no sean validados se consideran como bytes nulos y pueden ser removidos de la cadena de datos
TLAST	Indica el final de la cadena de datos
TID[(i-1):0]	(Opcional) Es el identificador de la cadena datos.
TDEST[(d-1):0]	(Opcional) Provee la información de ruteo para la cadena de datos.
TUSER[(u-1):0]	(Opcional) Es información definida por el usuario que puede ser transmitida de manera simultánea con la cadena de datos.

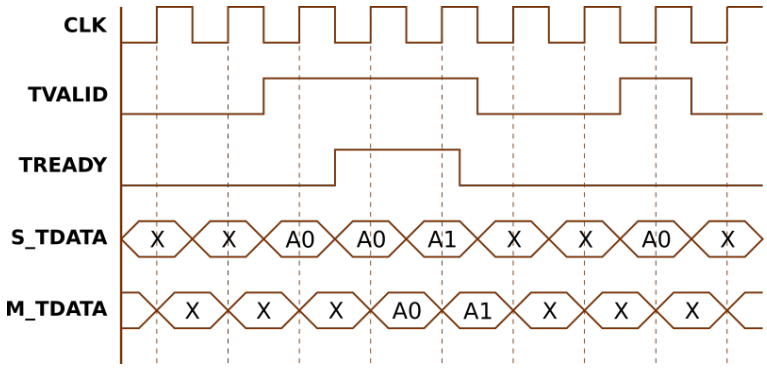
1.2.2 Transferencia de paquetes de datos. El proceso de transferencia de paquetes de datos entre el maestro y los esclavos está controlado por las señales *TVALID*, *TREADY* Y *TLAST* definidas en el Cuadro 2. Es importante mantener controlado y sincronizado el flujo de datos para que no existan pérdidas de información, mensajes incompletos o ciclos de reloj innecesarios.

Proceso de envío de paquetes. El envío de paquetes de datos es controlado a través de un flujo bidireccional entre el maestro y el esclavo mediante las señales *TVALID* y *TREADY*; ambas señales deben estar activas para que exista transferencia de datos. Sin embargo, este proceso tiene algunas restricciones, por ejemplo, el maestro no puede desactivar la señal *TVALID* después de su activación hasta que el esclavo active la señal *TREADY*. Por el contrario el esclavo puede activar y desactivar la señal *TREADY* cuando así se requiera. También es importante mencionar que el maestro no puede esperar a que el esclavo active la señal *TREADY* para activar *TVALID*, la señal debe activarse en el momento que la información esté

lista para ser enviada, por el contrario el esclavo puede esperar a la activación de *TVALID* para activar *TREADY*[13].

Activación de la señal TVALID antes que TREADY. Cuando el maestro activa *TVALID*, es decir, la muestra está lista para ser enviada, se debe esperar a la activación de *TREADY*, y luego de eso, se envía la muestra de *S_TDADA* a *M_TDATA*, es decir, de la entrada a la salida. Por otro lado, el envío del paquete se detiene debido a que *TREADY* se desactiva, por esta razón la muestra A0 del segundo paquete no es enviada.

Figura 1: TVALID antes que TREADY

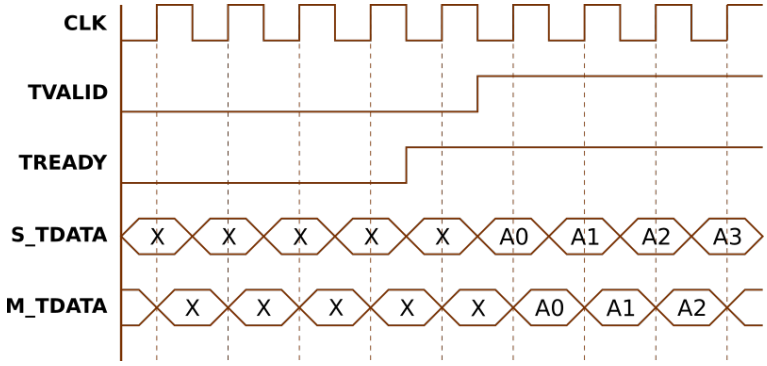


Activación de la señal TREADY antes que TVALID. En la Figura 2 se puede observar el caso donde el esclavo activa *TREADY* primero, es decir la muestra está lista para ser recibida. Aunque este no sea el caso, el valor de *TREADY* podría cambiar siempre y cuando *TVALID* no esté activo aún. Las muestras se envían una vez el maestro activa *TVALID*.

Activación de las señales TREADY y TVALID en el mismo ciclo de reloj. En la Figura 3 se puede observar el caso donde tanto el maestro como el esclavo han activado *TVALID* y *TREADY* en el mismo ciclo de reloj, y estas se mantienen activas solo durante ese ciclo. La transferencia del paquete ocurre en el flanco donde ambas

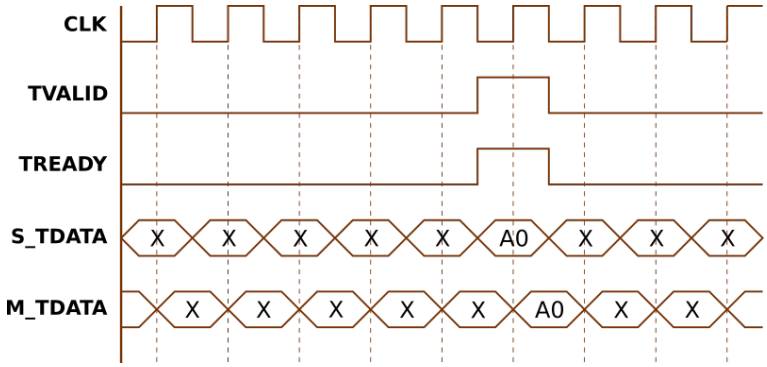
[13]XILINX INC. AXI Reference Guide. En: Xilinx [En línea]. (2011) [Consultado 10 Dic. 2017]. Disponible en: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

Figura 2: TREADY antes que TVALID



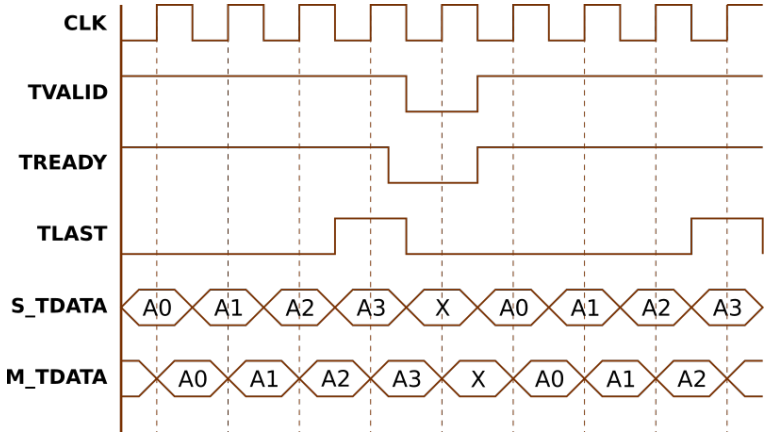
señales están activas, momento donde la muestra *A0* es enviada; sin embargo, el resto de muestras del paquete no se envía hasta que ambas señales se activen nuevamente.

Figura 3: TREADY Y TVALID en el mismo ciclo de reloj



1.2.3 Final de transferencia de un paquete. El final de transferencia de un paquete ocurre cuando *TLAST* es activado. *TLAST* se activa en la última muestra del paquete enviado, como se evidencia en la Figura 4. En este caso, tanto la señal *TVALID* como *TREADY* se desactivan por un ciclo de reloj, por lo que el envío de un nuevo paquete se retrasa un ciclo de reloj.

Figura 4: Final de transferencia del paquete



1.3 PAQUETES CHDR

Algunos dispositivos, como los *USRP X310* y en general todos los *USRP* de tercera generación utilizan un protocolo de transporte de datos llamado *Compressed Header* (*CHDR*), el cual comprime toda la información del paquete de datos enviado (excepto el *timestamp*) en 64 bits. Esto simplifica el envío de datos evitando el uso de complejas máquinas de estado para separar y obtener las características del paquete, lo que mejora el *throughput* y reduce el tiempo de síntesis de una imagen del *FPGA*. La forma general de la palabra con el protocolo *CHDR* se puede ver en el Cuadro 3

Cuadro 3: Forma general palabra paquetes CHDR

Dirección (Bytes)	Longitud (Bytes)	Componente
0	8	*CHDR*
8	8	*Timestamp* (opcional)
8/16	-	Datos

Si no se provee *timestamp*, los datos se asignan desde la dirección 8. El *timestamp* se cuenta por *ticks* y representa el orden en el que las muestras son enviadas. El cabecero *CHDR* de la palabra dentro del protocolo tiene una estructura que se puede ver en el Cuadro 4.

Cuadro 4: Estructura código CHDR

Dirección	Componente	Descripción
0:15	ID de destino	Espacio de 32 bits donde se provee un identificador del paquete de datos desde su origen hasta su destino.
15:31	ID de origen	
32:47	Tamaño	Proporciona la longitud del paquete de datos enviado (en bytes)
48:59	Secuencia de 12 dígitos	Contador de *FRAMES*
60	Error o fin del envío de datos	El bit en alto finaliza la transferencia de datos.
61	Timestamp	El bit en alto indica que el paquete contiene *TIMESTAMP*.
62:63	Tipo de paquete	Especifica el tipo de paquete: datos, control de flujo o comandos.

El Cuadro 5 muestra la tabla de verdad de los tipos de paquete.

Cuadro 5: Tipos de paquete

Bit 63	Bit 62	Bit 61	Bit 60	Tipo de paquete
0	0	x	0	Datos
0	0	x	1	Datos (Final de la cadena)
0	1	x	0	Control de flujo
1	0	x	0	Paquete de comandos
1	1	x	0	Respuesta de comandos
1	1	x	1	Error de respuesta de comandos

2. IMPLEMENTACIÓN FFT

La transformada de Fourier es una herramienta importante en muchas aplicaciones que requieren un procesamiento digital de señales tales como el tratamiento de imágenes, de audio, y telecomunicaciones. Por lo tanto, estas aplicaciones se ven afectadas por la complejidad y los recursos computaciones que requiere la transformada. La computación directa de la transformada discreta de Fourier DFT, dada por la ecuación 2.1, requiere N^2 operaciones siendo N el tamaño de X_k y x_n. El algoritmo de la transformada rápida de Fourier FFT, expuesto originalmente por Cooley y Tukey[14], abrió una nueva área en el procesamiento digital de señales al reducir el orden de complejidad a $N \log_2 N$.

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn} \qquad k = 0, \ldots, N-1 \tag{2.1}$$

$$W_N = e^{-i2\pi/N} \tag{2.2}$$

Desde Cooley-Tukey se han desarrollado múltiples algoritmos para la FFT, siendo los más utilizados en la práctica los algoritmos FHT, Radix-2, Radix-4 y Radix-split por su estructura simple que utiliza unidades de procesamiento básicas llamadas *butterflies* [15]. La mayoría de estos algoritmos se han implementado en GPP[16], en procesadores de señales digitales[17] y en circuitos integrados diseñados únicamente para procesamiento de la FFT[18]. Los *FPGA* han crecido en capacidad, mejorado en

[14]COOLEY, James; TUKEY, John. An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation, 1965. p 297-301.

[15]UZUN, Isa; AMIRA, Abbes. Towards a general framework for an FPGA-based FFT coprocessor. Seventh International Symposium on Signal Processing and Its Applications, 2003. Proceedings. p 617-620.

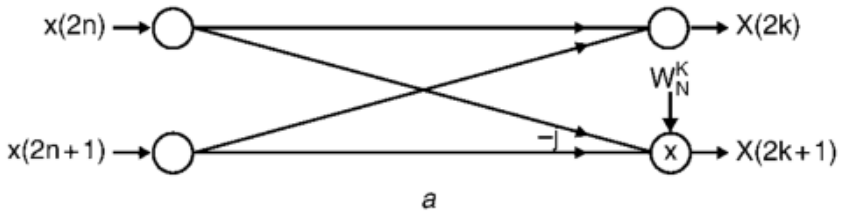
[16]FRIGO, Matteo; JOHNSON, Steven. The design and implementation of FFTW3. Proceedings of the IEEE, 2005, vol. 93, no 2, p. 216-231.

[17]TEXAS INSTRUMENTS. FFT Implementation on the TMS320VC5505, TMS320C5505, and TMS320C5515 DSPs [En línea]. (2013) [Consultado 20 Abr. 2018]. Disponible en: `http://www.ti.com/litv/pdf/sprabb6b`

[18]BAAS, Bevan M. A low-power, high-performance, 1024-point FFT processor. IEEE Journal of Solid-State Circuits, 1999, vol. 34, no 3, p. 380-387.

rendimiento y reducido su costo, por lo tanto, se han convertido en una solución viable a tareas computacionales de alto rendimiento como la FFT [19].

Figura 5: Radix-2 *Butterfly*



El algoritmo Radix-2 utiliza la descomposición más simple de la FFT que computa una DFT de 2 puntos (ver Figura 5). Para el algoritmo de decimación en frecuencia, la DFT se descompone en componentes de frecuencia pares (2.3) e impares (2.4). Esto permite utilizar el *butterfly* del algoritmo Radix-2 para computar una FFT de cualquier tamaño utilizando la unidad básica de procesamiento (Figura 6).

$$X_{2k} = \sum_{n=0}^{N/2-1} (x_n + x_{n+N/2}) W_{N/2}^{kn} \tag{2.3}$$

$$X_{2k+1} = \sum_{n=0}^{N/2-1} (x_n - x_{n+N/2}) W_{N/2}^{n} W_{N/2}^{kn} \tag{2.4}$$

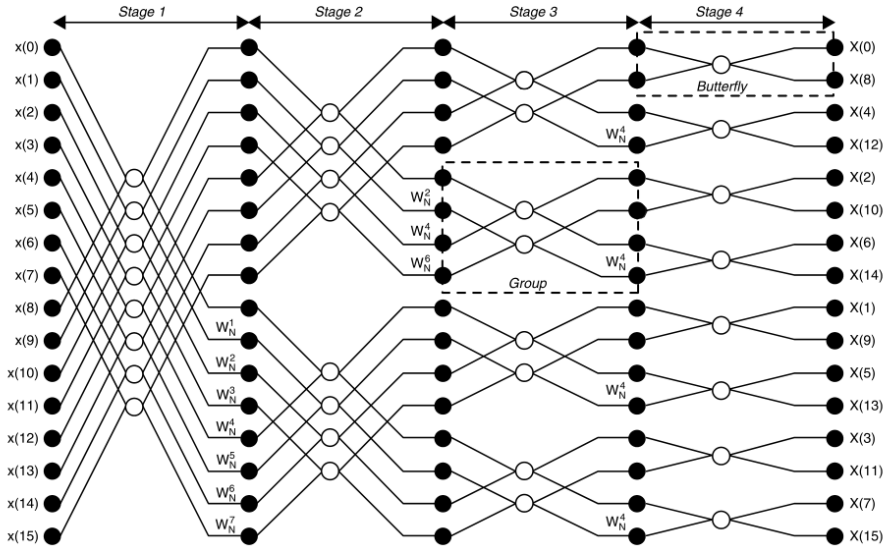
2.1 XILINX FFT IP-CORE

El *IP-CORE* LogiCORE FFT de Xilinx implementa el algoritmo Cooley-Tukey para calcular la DFT utilizando las descomposiciones Radix-4 y Radix-2 para su computación en cuatro arquitecturas disponibles. Este *IP-CORE* presenta características esenciales para su uso en la arquitectura RFNoC, específicamente el soporte de la interfaz *AXI-STREAM* y de aritmética en punto fijo[20].

[19] UZUN, Isa Servan; AMIRA, Abbes; BOURIDANE, Ahmed. FPGA implementations of fast Fourier transforms for real-time signal and image processing. IEE Proceedings-Vision, Image and Signal Processing, 2005, vol. 152, no 3, p. 283-296.

[20] XILINX INC. Fast Fourier Transform v9.0 LogiCORE IP Product Guide. En: Xilinx [En línea]. (2015) [Consultado 23 Abr. 2018]. Disponible en: `http://www.ti.com/litv/pdf/sprabb6b`

Figura 6: Diagrama de una FFT Radix-2 de 16 puntos



2.1.1 Configuración La creación y caracterización del archivo de configuración del *IP-CORE* de Xilinx se realiza en Vivado Design Suite mediante una interfaz gráfica. El *IP-CORE* ofrece una variedad de opciones de configuración e implementación, las cuales se definen en los Cuadros 6 y 7.

Cuadro 6: Opciones de configuración FFT

Configuración	Valor
Canales	1
N	2048
Arquitectura	*Pipelined, Streaming I/O*
N configurable en ejecución	Si

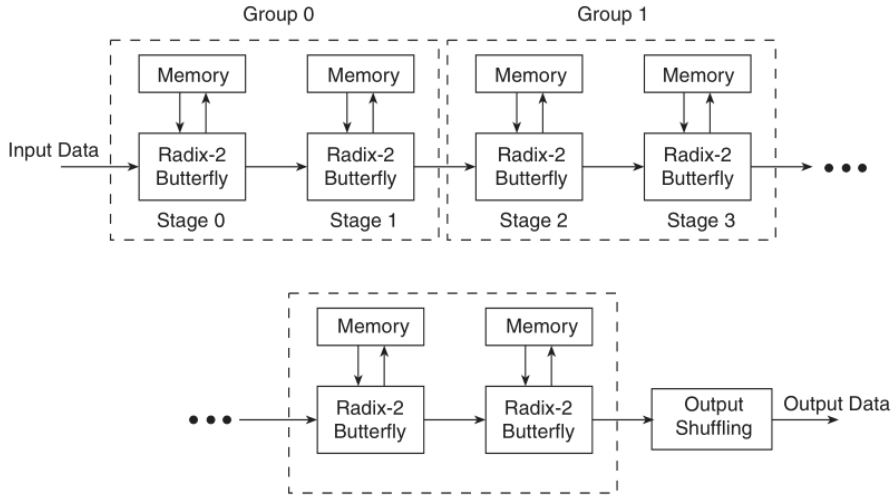
El tamaño de un paquete *CHDR* esta limitado por la máxima unidad de transmisión o MTU que podría contener una trama Ethernet del estándar IEEE 802.3 [21] o una Jumbo Ethernet[22], según las características de la tarjeta de red. Dado el tamaño en

[21] CHRISTENSEN, Ken, et al. IEEE 802.3 az: the road to energy efficient ethernet. IEEE Communications Magazine, 2010, vol. 48, no 11.

[22] ETHERNET ALLIANCE. Ethernet Jumbo Frames. [En línea]. (2009) [Consultado 23 Abr. 2018]. Disponible en: http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf

bytes de una muestra (ver Anexo F), el máximo N posible es de 2048 [23]. La arquitectura *Pipelined Streaming I/O* segmenta el cauce de múltiples unidades Radix-2 para ofrecer un procesamiento continuo de datos. Esta arquitectura presenta el mayor *throughput* a costa del mayor consumo de recursos[24].

Figura 7: Pipelined Streaming I/O



El *IP-CORE* permite cambiar el tamaño de la transformada, en ejecución, a valores menores al definido anteriormente, con un mínimo definido por la arquitectura de 16 y siempre potencias de 2, utilizando más recursos lógicos. Esto facilita el uso de diferentes tamaños sin la necesidad de volver a sintetizarlo.

Puesto que el tamaño de la parte real e imaginaria de las muestras siempre debe ser de 16 bits (ver Anexo F), los datos de salida se deben escalar para no perder información, y redondear para mayor precision[25]. La señal ARESETn permite un control directo sobre el estado del *IP-CORE*, por lo cual la señal ACLKEN no es necesaria.

[23] XILINX INC. Fast Fourier Transform v9.0 LogiCORE IP Product Guide. En: Xilinx [En línea]. (2015) [Consultado 23 Abr. 2018]. Disponible en: `http://www.ti.com/litv/pdf/sprabb6b`
[24] Ibid
[25] Ibid

Cuadro 7: Opciones de implementacion FFT

Implementación	Valor
Formato de datos	Punto fijo
Opción de escalamiento	Escalado
Modo de redondeo	Redondeo convergente
Tamaño de palabra de entrada	16 bits
Tamaño de palabra de W_N^{kn}	16 bits
ARESETn	Yes
ACLKEN	No
Orden de salida	Orden natural
Inserción cíclica de prefijo	No
XK_INDEX	No
OVFLO	No
Sistema de regulación de flujo	No en tiempo real

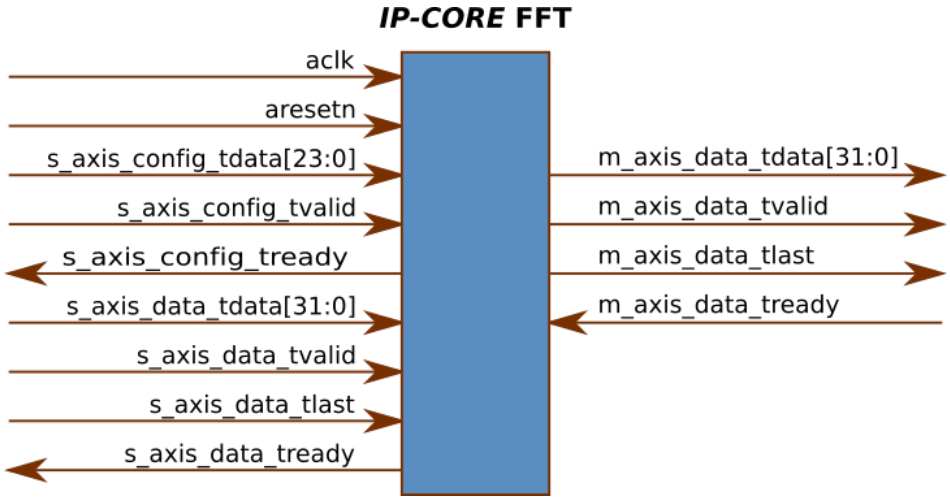
Dado que la arquitectura utiliza Radix-2 y una decimación en frecuencia, la salida de datos se presenta por defecto en un orden de bits invertidos. Reorganizar la salida a su orden natural implica agregar una latencia igual a N ciclos de reloj ademas de recursos de memoria. Una salida en orden natural ofrece autonomía al *IP-CORE* en el procesamiento de la DFT.

Teniendo en cuenta el aumento de un 1 bit a la salida de cada etapa de *radix-2*, un escalamiento es necesario para evitar desbordamiento [26]. Con el escalamiento adecuado y un orden de salida natural, las señales XK_INDEX (indice de la muestra) y OVFLOW (indicador de *overflow*) no son necesarias. Un sistema de regulación en tiempo real implica que el flujo de datos *AXI-STREAM* no posee interrupciones y, el maestro y esclavo poseen la capacidad de enviar y recibir a la misma tasa que el *IP-CORE*, en caso contrario se perderían datos procesados[27]. El sistema no en tiempo real da flexibilidad al diseño del maestro y esclavo, y asegura la transmisión de todos datos.

[26] Ibid
[27] Ibid

2.1.2 Interfaz El *IP-CORE* presenta dos canales *AXI-STREAM* de entrada y uno de salida junto a las señales de reloj, reinicio y eventos.

Figura 8: Interfaces del circuito FFT *IP-CORE*



- **aclk**: Señal de reloj general, todas las señales de entrada, salida e internas funcionan bajo este reloj.
- **aresetn**: Señal de reinicio síncrona, activa en bajo, que lleva todos los puertos de salida, contadores internos y variables de estado a sus valores iniciales. Todos los procesos son reinicializados.
- **s_axis_config**: Canal de configuración *AXI-STREAM* que lleva el tamaño de la transformada, su escalamiento y dirección.
- **s_axis_data**: Canal de datos de entrada *AXI-STREAM* con muestras reales e imaginarias.
- **m_axis_data**: Canal de datos de salida *AXI-STREAM* con muestras reales e imaginarias.

2.1.3 Operación Los canales de datos de entrada y salida funcionan bajo el protocolo *AXI-STREAM* y el formato de datos es el siguiente:

Los segmentos IMAG y REAL son los valores imaginarios y reales de la muestra respectivamente, en punto fijo y complemento dos. Las posiciones de la parte real

Cuadro 8: Formato de datos FFT *IP-CORE*

Dirección (Bits)	Longitud (Bits)	Componente
0	16	IMAG
16	16	REAL

e imaginaria de la muestra difieren al de la estructura utilizada por RFNoC como se muestra en la Cuadro 8

El canal de configuración trabaja bajo el protocolo *AXI-STREAM* y posee la estructura del Cuadro 9

Cuadro 9: Bus de configuración FFT *IP-CORE*

Dirección (Bits)	Longitud (Bits)	Componente
0	5	LOG2N
5	3	Reservado
8	1	FWD_INV
9	12	SCA

El segmento LOG2N contiene el logaritmo en base dos del tamaño de la transformada que puede tomar un valor desde cuatro hasta once. El segmento FWD_INV indica la dirección de la transformada, alto para una transformada directa.

SCA contiene el escalamiento de la FFT, donde el *IP-CORE* divide por un factor de 1, 2, 4 u 8 cada grupo (ver Figura 7) de Radix-2, según cada par de bits. El escalamiento del primer grupo se encuentra en los dos bits menos significativos. El *IP-CORE* escala la salida según la ecuación:

$$s = \prod_{i=0}^{\log_2 N-1} 2^{b_i} \tag{2.5}$$

Donde 2^{b_i} es la escala en el grupo i.

Por ejemplo, para un N igual a 256 el siguiente SCA:

$$SCA = 11\ 11\ 10\ 10\ 10\ 10 = 682 \tag{2.6}$$

$$s = 2^2 \cdot 2^2 \cdot 2^2 \cdot 2^2 = 256 \tag{2.7}$$

El escalamiento para grupos que pertenecen a N superiores se ignoran. Cuando el tamaño de la transformada no es una potencia de 4, el último grupo solo contiene una etapa Radix-2 y por lo tanto la escala del grupo solo puede ser 1 o 2.

Para un escalamiento igual a 512 para un N igual a 512, que no es potencia de 4:

$$SCA = 11\ 01\ 10\ 10\ 10\ 10 \tag{2.8}$$

$$s = 2^1 \cdot 2^2 \cdot 2^2 \cdot 2^2 \cdot 2^2 = 512 \tag{2.9}$$

2.1.4 Recursos y rendimiento El generador del *IP-CORE* en Vivado provee el rendimiento de la transformada para la configuración dada en función de N, mostrado en el Cuadro 10.

Cuadro 10: Rendimiento de *IP-CORE* FFT

N	**Ciclos**	**Latencia (μs)**
16	145	0.725
32	192	0.960
64	310	1.550
128	501	2.505
256	907	4.535
512	1674	8.370
1024	3231	16.155
2048	6302	31.510

La síntesis del *IP-CORE* arroja la cantidad de recursos utilizados en el Cuadro 11.

Cuadro 11: Recursos utilizados *IP-CORE* FFT

Tipo de recurso	Usados	Util %
Slice LUTs	2792	1.10
Slice Registers	5073	1.00
F7 Muxes	44	0.03
F8 Muxes	22	0.03
Block RAM Tile	7	0.88
DSPs	48	3.12

2.2 Circuito de desplazamiento

Para una mejor visualización de la FFT, su salida se reorganiza de tal forma que la componente de frecuencia cero se encuentra en el centro del espectro. Su uso debe ser opcional por la penalización de latencia (y recursos) que conlleva.

2.2.1 Diseño La salida de la transformada tiene un orden natural con paquetes de salida iguales al tamaño de la transformada, delimitados por la señal *AXI-STREAM TLAST*. El desplazamiento se puede lograr guardando todo el paquete en una memoria de acceso aleatorio (RAM) en el orden adecuado, es decir, guardar las primeras $N/2$ muestras en la segunda mitad de la RAM y la últimas en la primera, luego leer la RAM en orden para enviar los datos. Este método tiene una latencia igual a N ciclos de reloj.

Sin embargo, es posible utilizar la señal de *TLAST* para disminuir esta latencia. Guardando la primera mitad de la transformada en una RAM, luego enviando directamente la segunda mitad al esclavo para luego, enviar los datos guardados en la RAM y guardar en otra los datos recibidos del siguiente paquete, simultáneamente, delimitando correctamente el paquete desplazado con la señal de *TLAST*.

Este método posee una latencia de $N/2$ ciclos de reloj y se basa en la multiplexación de tres señales que provienen de dos RAM y del maestro, utilizando el protocolo *AXI-STREAM* para la sincronización de lectura y escritura en las RAM y el paso directo de las muestras.

Figura 9: Diagrama de tiempos circuito de desplazamiento mayor latencia

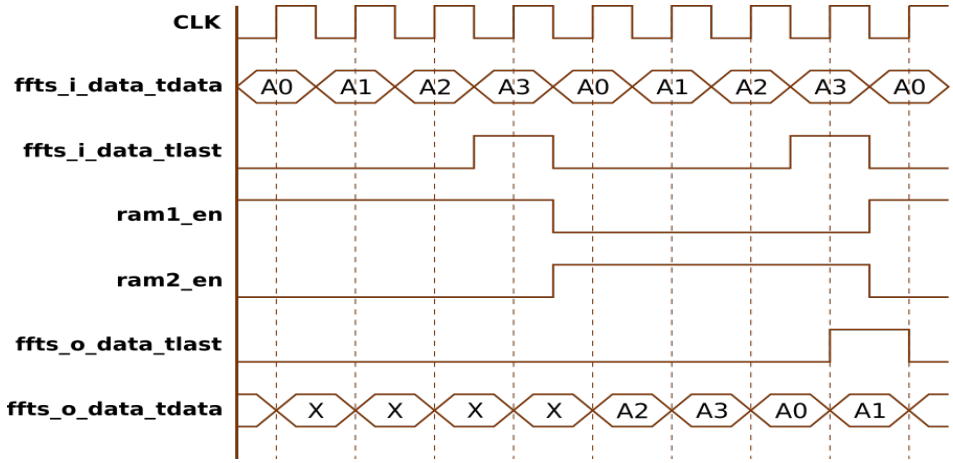
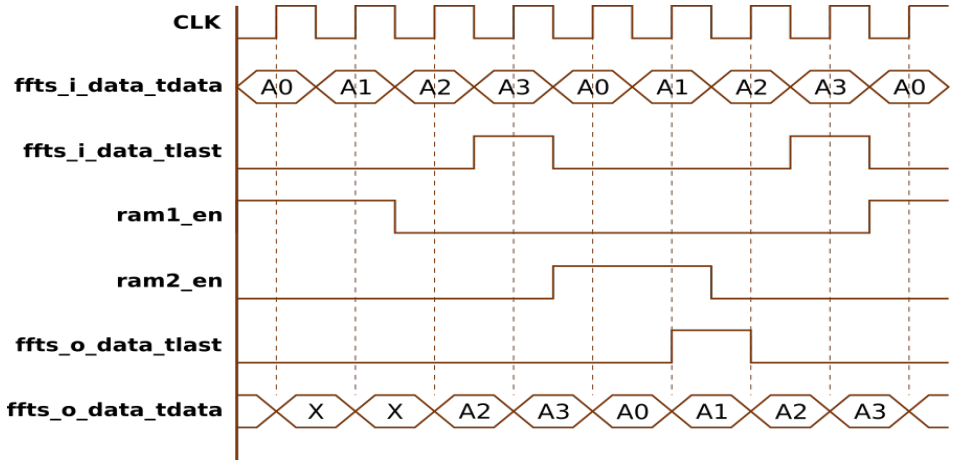
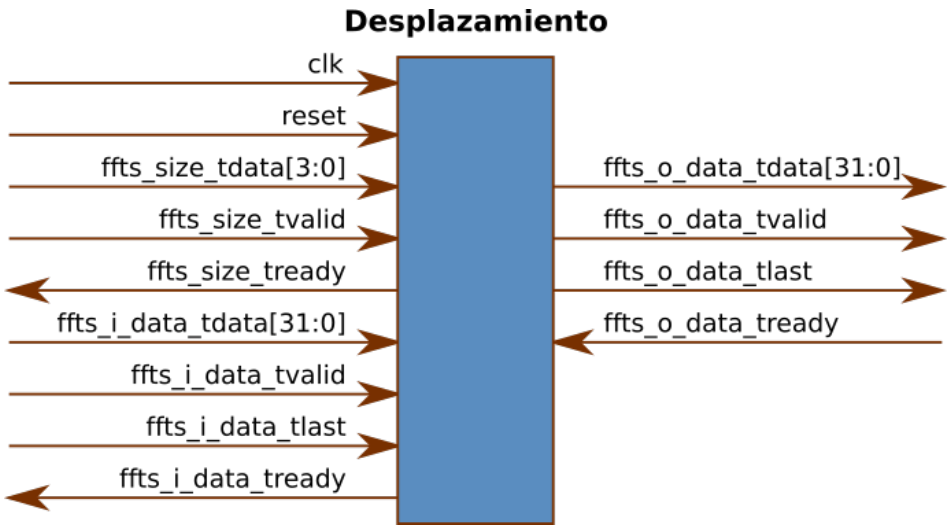


Figura 10: Diagrama de tiempos circuito de desplazamiento menor latencia



2.2.2 Interfaz Al igual que el *IP-CORE* FFT, el desplazamiento posee dos canales de entrada y uno de salida todos bajo el protocolo *AXI-STREAM*, además de las señales de reloj y reinicio.

Figura 11: Interfaces del circuito de desplazamiento



- **clk**: Señal de reloj general, todas las señales de entrada, salida e internas funcionan bajo este reloj.
- **reset**: Señal de reinicio síncrona, activa en alto, esta detiene todos los procesos y lleva todas las señales a su valor de reinicio.
- **ffts_i_size**: Canal *AXI-STREAM* que contiene el tamaño de la transformada en la misma posición que en el FFT *IP-CORE*.
- **ffts_i_data**: Canal *AXI-STREAM* de datos de entrada con muestras reales e imaginarias con la misma estructura que el canal de datos de entrada en el FFT *IP-CORE*.
- **ffts_o_data**: Canal *AXI-STREAM* de datos de salida con muestras reales e imaginarias con la misma estructura que el canal de datos de salida en el FFT *IP-CORE*.

2.2.3 Recursos y rendimiento Los recursos de *FPGA* que consume el desplazamiento se muestran en el Cuadro 12.

Cuadro 12: Recursos usados por el circuito de desplazamiento

Tipo de recurso	Usados	Util %
Slice LUTs	145	0.06
Slice Registers	75	0.01
Block RAM Tile	8	1.01
Bonded IOB	78	15.60
BUFGCTRL	1	3.13

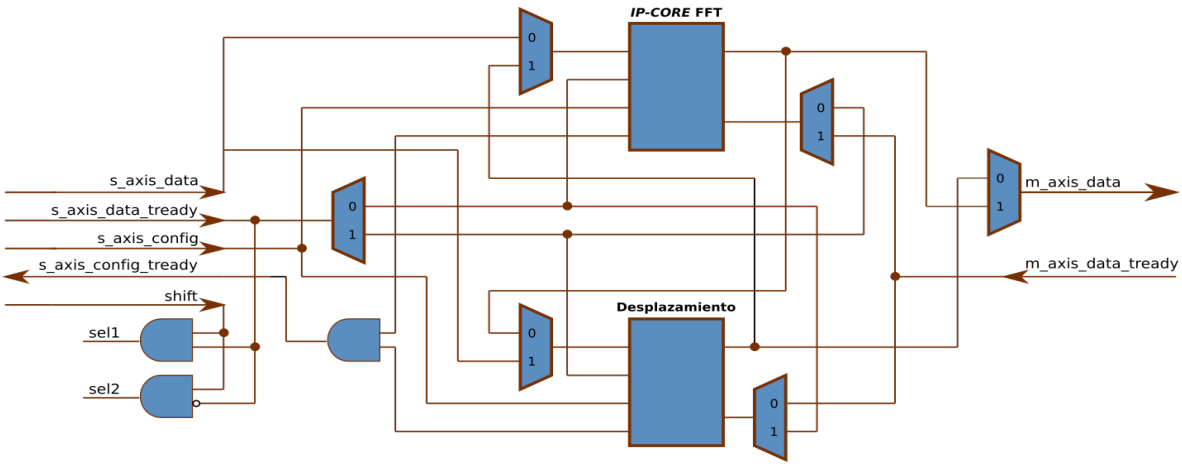
El rendimiento, como ya se mencionó antes, es proporcional al tamaño de la transformada. La latencia del circuito es de $N/2$ ciclos de reloj.

2.3 Circuito FFT

El circuito final une el *IP-CORE* FFT y el desplazamiento de tal forma que compute correctamente la transformada inversa o directa, y que el desplazamiento sea opcional.

Para esto es necesario multiplexar las entradas del FFT *IP-CORE*, del desplazamiento y la salida del circuito FFT, en función de las señales de configuración como se muestra en la Figura 12.

Figura 12: Diseño final FFT con desplazamiento



El circuito genera las configuraciones mostradas en las Figuras 13, 14 y 15.

Figura 13: Configuración interna de FFT directa con desplazamiento

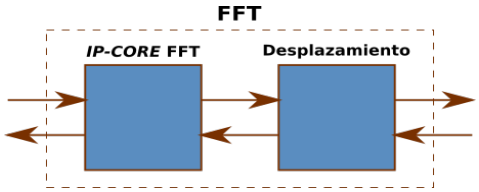


Figura 14: Configuración interna de FFT inversa con desplazamiento

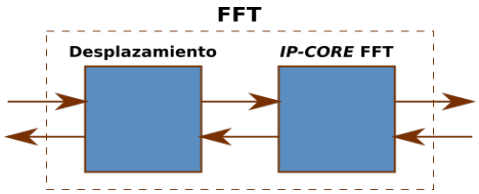
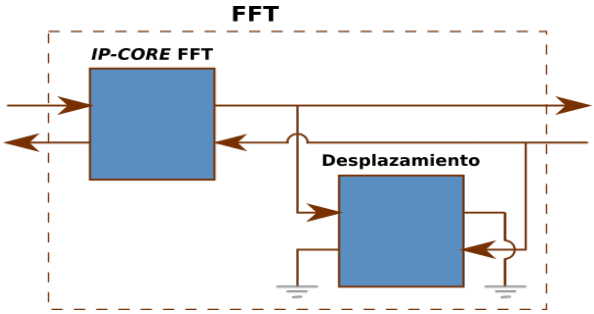


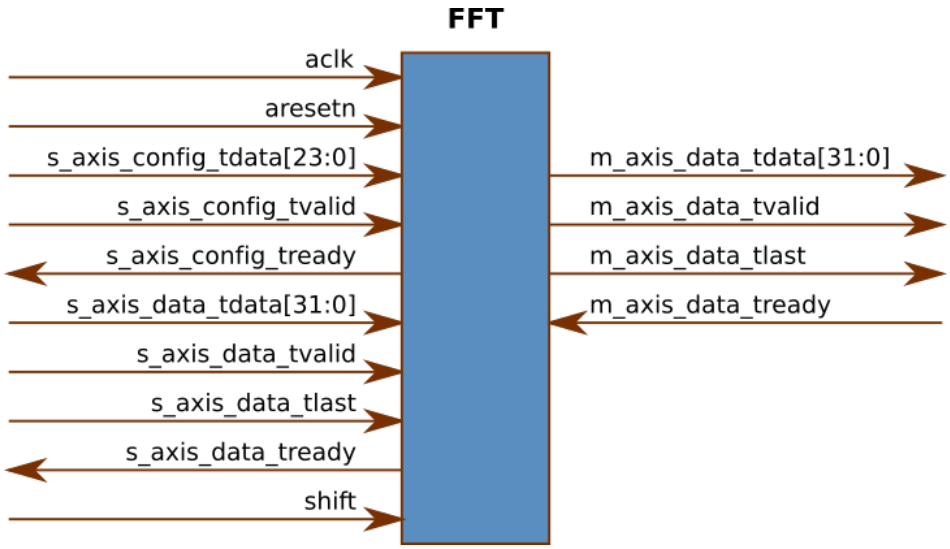
Figura 15: Configuracion interna de FFT directa o inversa sin desplazamiento



2.3.1 Interfaz El circuito FFT posee las mismas entradas y salidas que el FFT *IP-CORE*, con la adición de una señal que habilita el desplazamiento.

- **aclk**: Señal de reloj general, todas las señales de entrada, salida e internas funcionan bajo este reloj.
- **aresetn**: Señal de reinicio síncrona, activa en bajo, que lleva todos los puertos de salida, contadores internos y variables de estado a sus valores iniciales. Todos los procesos son reinicializados.
- **s_axis_config**: Canal de configuración *AXI-STREAM* que lleva el tamaño de la transformada, su escalamiento y dirección. Igual al canal de FFT *IP-CORE*.

Figura 16: Interfaces del circuito FFT



- **s_axis_data**: Canal de datos de entrada *AXI-STREAM* con muestras reales e imaginarias.
- **m_axis_data**: Canal de datos de salida *AXI-STREAM* con muestras reales e imaginarias.
- **shift**: Bit de habilitación de desplazamiento.

2.4 Bloque RFNoC FFT

El proceso de integración conlleva una serie de pasos que se tratarán en el capítulo 3. En esta sección se especificarán los detalles de integración para el funcionamiento del bloque RFNoC FFT.

2.4.1 Nivel FPGA Con la plantilla base para FPGA de RFNoC es necesario declarar el circuito FFT y conectar sus puertos a las interfaces de *axi wrapper* y *noc shell* correctas. Los puertos de datos de entrada y salida se conectan directamente a las interfaces de salida y entrada correspondientes del *axi wrapper*. El canal de configuración del circuito FFT debe ser *AXI-STREAM* y por tanto se utiliza el registro 128, la información proveniente del usuario en este registro pasa a través del *axi wrapper* donde se convierte a *AXI-STREAM* y sale del *axi wrapper* por el canal de

configuración dedicado que se conecta al del circuito FFT. Por ultimo, el bit de configuración de desplazamiento para el cual se utiliza un *setting register* con el registro 130.

2.4.2 Nivel UHD La integración a UHD corresponde a la definición de los registros, y de los argumentos que los modifican. Se definen las direcciones 128 y 130 para el canal de configuración y el bit de desplazamiento, respectivamente. Y también se especifican los argumentos del bloque a nivel de UHD que son el bit de desplazamiento, el bit de dirección, el escalamiento sin codificar y el tamaño de la transformada, que es igual al tamaño del paquete. Por último, se define el tamaño del paquete para el puerto de entrada y de salida igual al tamaño de la transformada.

```
...
<registers>
  <setreg>
    <name>AXI_CONF_BUS</name>
    <address>128</address>
  </setreg>
  <setreg>
    <name>AXI_CONF_BUS_TLAST</name>
    <address>129</address>
  </setreg>
  <setreg>
    <name>SHIFT</name>
    <address>130</address>
  </setreg>
</registers>

<args>
  <arg>
    <name>spp</name>
        ...
    <action>SR_WRITE("AXI_CONF_BUS", ADD(1398016, LOG2($spp)))</action>
  </arg>
  <arg>
    <name>inv</name>
    ...
```

```
      <action>SR_WRITE(.^XI_CONF_BUS", ADD(ADD(1397760, MULT($inv, 256)),
          LOG2($spp)))</action>
    </arg>
    <arg>
      <name>sca</name>
      ...
      <action>SR_WRITE("AXI_CONF_BUS", ADD(ADD($sca, MULT($inv, 256)),
          LOG2($spp)))</action>
    </arg>
    <arg>
      <name>shift</name>
          ...
      <action>SR_WRITE("SHIFT", $shift)</action>
    </arg>
  </args>

  <ports>
    <sink>
      <name>in </name>
      <type>sc16</type>
      <vlen>$spp</vlen>
      <pkt_size>%vlen </pkt_size>
    </sink>
    <source>
      <name>out </name>
      <type>sc16</type>
      <vlen>$spp</vlen>
      <pkt_size>%vlen </pkt_size>
    </source>
  </ports>
```

2.4.3 Nivel GNU-Radio A nivel de GNU-Radio se definen las interfaces con los argumentos del bloque a nivel de UHD. Y a su vez se especifican los parámetros que van a esas interfaces, estos son el tamaño de la transformada, el control del desplazamiento y dirección y la escala. También se declara el tamaño del vector a nivel de GNU-Radio igual al tamaño de la transformada.

```
  <make>FFTMod.fft(
   ...
self.$(id).set_arg("spp", $spp)
self.$(id).set_arg("inv", $inv)
self.$(id).set_arg("sca", $sca)
self.$(id).set_arg("shift", $shift)
  </make>

  <callback>set_arg("spp", $spp)</callback>
  <callback>set_arg("inv", $inv)</callback>
  <callback>set_arg("sca", $sca)</callback>
  <callback>set_arg("shift", $shift)</callback>

  <param>
    <name>FFT Size</name>
    <key>spp</key>
    <type>int</type>
    <option>
     ...
    </option>
  </param>

  <param>
    <name>Direction</name>
    <key>inv</key>
    <type>int</type>
    <option>
     ...
    </option>
  </param>

  <param>
    <name>Scaling</name>
    <key>sca</key>
    <value>1397760</value>
    <type>int</type>
  </param>
  ...
```

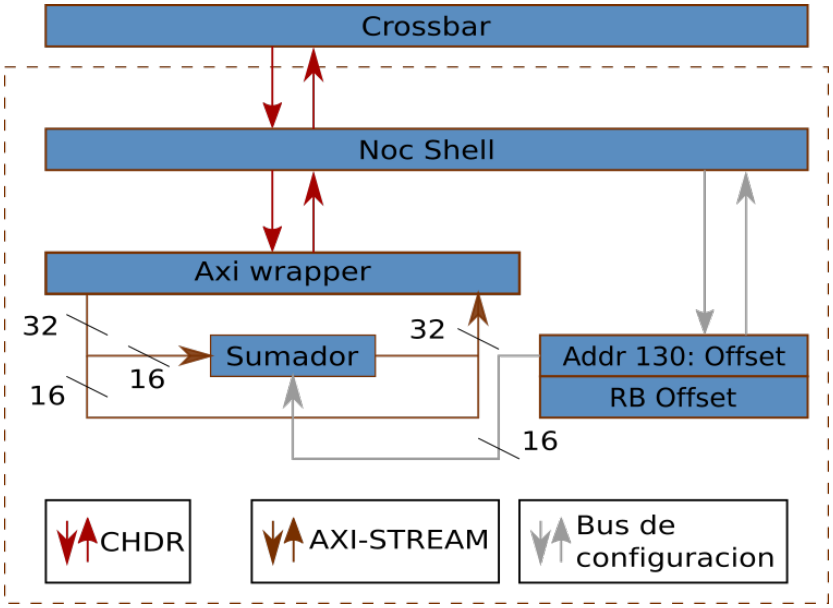
```
  <param>
    <name>Shift</name>
    <key>shift</key>
    <value>0</value>
    <type>int</type>
  </param>
  ...

  <!--RFNoC basic block configuration -->
  ...

  <sink>
    <name>in</name>
    <type>$type</type>
    <vlen>$spp</vlen>
    <domain>rfnoc</domain>
  </sink>

  <source>
    <name>out</name>
    <type>$type</type>
    <vlen>$spp</vlen>
    <domain>rfnoc</domain>
  </source>
</block>
```

3. GUÍA

En este capítulo se explica el proceso de creación de bloques RFNoC mediante un ejemplo cuyo funcionamiento consiste en sumar la señal de entrada con una señal real de *offset* controlada por el usuario. Si no se está familiarizado con la creación de bloques RFNoC, se recomienda revisar los Anexos F, G, H e I.

A lo largo del capítulo se resaltará con negrilla los espacios que el usuario debe editar, o elementos importantes. El esquema general a nivel *FPGA* de este ejemplo se puede ver en la Figura 17.

Figura 17: Esquema general nivel FPGA



3.1 CREACIÓN DEL MÓDULO Y BLOQUE

1. Primero se crea un módulo *OOT* vacío, para este caso con nombre *tutorial*. Esto se puede realizar ingresando el siguiente comando en la terminal:

```
$ cd ~/rfnoc/src/ && rfnocmodtool newmod tutorial
```

Si el proceso de creación del modulo fue exitoso, se debe recibir el siguiente mensaje:

```
Creating out-of-tree module in ./rfnoc-tutorial... Done.
Use 'rfnocmodtool add' to add a new block to this currently empty
    module.
```

2. Después, se agrega un bloque al módulo, en este caso con nombre *offset*.

```
$ cd ~/rfnoc/src/rfnoc-tutorial/ && rfnocmodtool add offset
```

3. Una vez ejecutado el comando, se deben rellenar los espacios como se muestra a continuación:

```
RFNoC module name identified: tutorial
Block/code identifier: offset
Enter valid argument list, including default arguments:
Add Python QA code? [y/N] N
Add C++ QA code? [y/N] N
Block NoC ID (Hexadecimal):
Random NoC ID generated: 7404F38FCDFB03AF
Skip Block Controllers Generation? [UHD block ctrl files] [y/N] y
Skip Block interface files Generation? [GRC block ctrl files] [y/N] y
```

Como no se asignó *Noc ID*, uno es generado aleatoriamente. Después los siguientes archivos son generados:

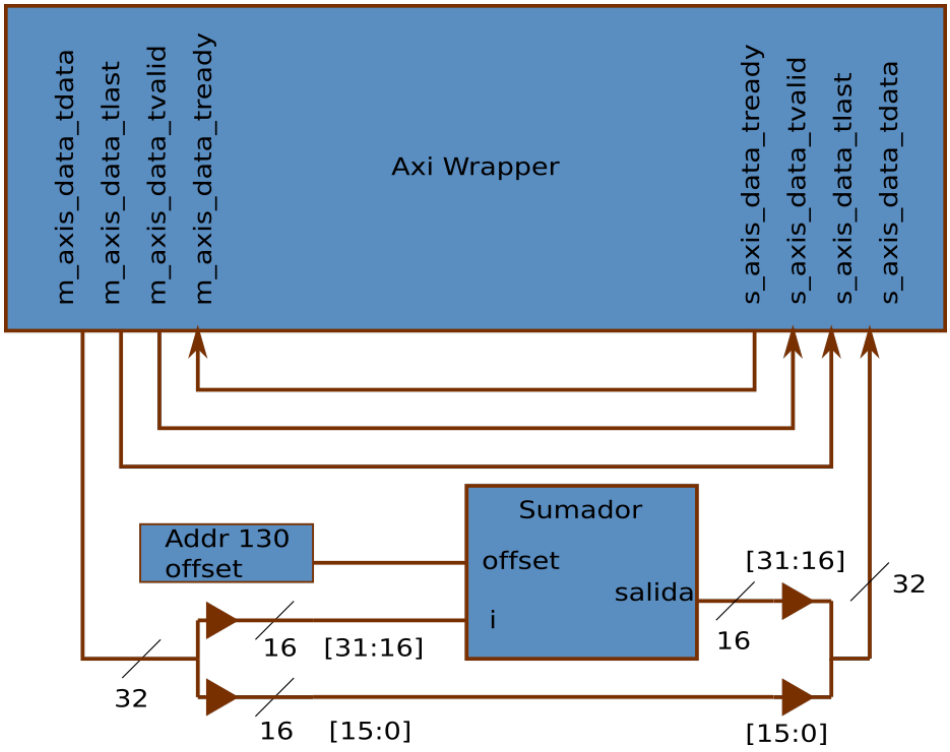
```
Editing swig/tutorial_swig.i...
Adding file 'grc/tutorial_offset.xml'...
Adding file 'rfnoc/blocks/offset.xml'...
Adding file 'rfnoc/fpga-src/noc_block_offset.v'...
rfnoc/testbenches/noc_block_offset_tb folder created
Adding file 'rfnoc/testbenches/noc_block_offset_tb/noc_block_offset_tb.sv'...
Adding file 'rfnoc/testbenches/noc_block_offset_tb/Makefile'...
Adding file 'rfnoc/testbenches/noc_block_offset_tb/CMakeLists.txt'...
```

Las direcciones de los archivos en negrilla corresponden con las plantillas que se editarán en las secciones posteriores. Se puede encontrar información detallada de esta parte de la guía, en el Anexo F.

3.2 Plantilla de verilog y simulación

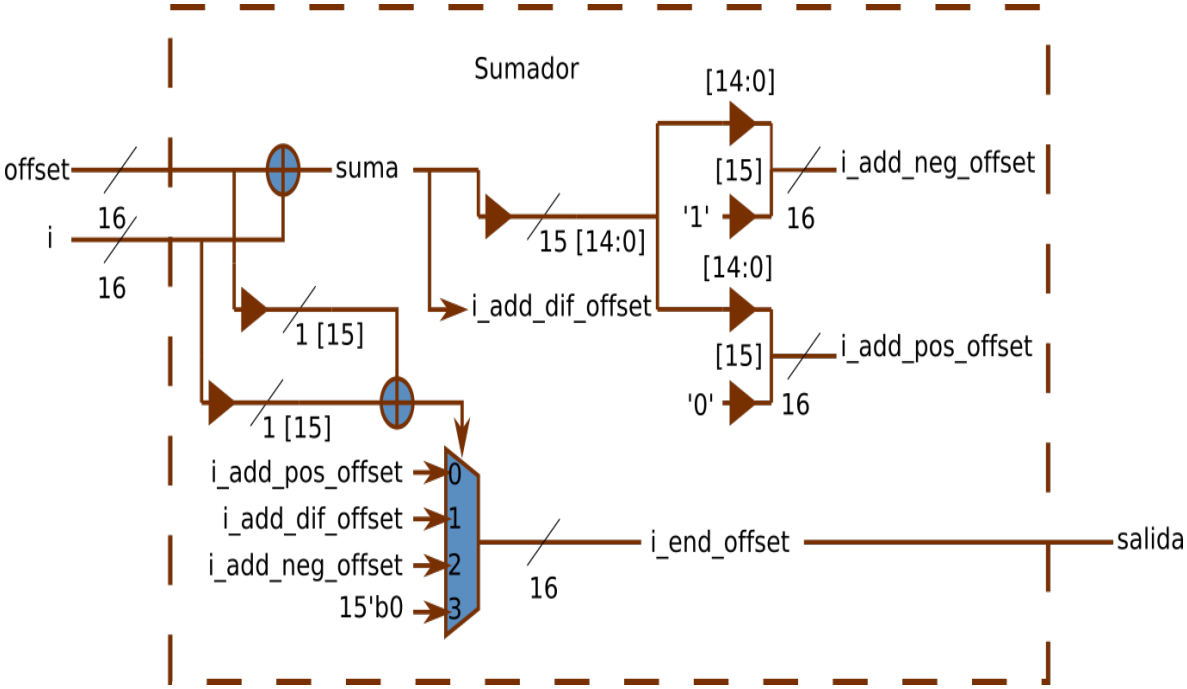
1. Antes de empezar a editar la plantilla es necesario realizar el diseño en diagrama de bloques con el fin de simplificar el proceso de depuración. El diseño para este caso se puede ver en las Figuras 18 y 19.

Figura 18: Diagrama del bloque *offset*



2. A continuación se debe acceder a la plantilla correspondiente a la descripción en verilog del bloque a implementar en el FPGA, esto es posible ejecutando el siguiente comando:

Figura 19: Diagrama del bloque sumador




```
$ gedit ~/rfnoc/src/rfnoc-tutorial/rfnoc/fpga-src/noc_block_offset.v
```

3. Después se edita la plantilla según el diseño previo.

```
...
// NoC Shell registers 0 - 127,
// User register address space starts at 128
localparam SR_USER_REG_BASE = 130;
...
localparam [7:0] SR_OFFSET = SR_USER_REG_BASE;
localparam [7:0] SR_TEST_REG_1 = SR_USER_REG_BASE + 8'd1;
wire [15:0] offset;
setting_reg #(
  .my_addr(SR_OFFSET), .awidth(8), .width(16))
sr_offset (
  .clk(ce_clk), .rst(ce_rst),
  .strobe(set_stb), .addr(set_addr), .in(set_data), .out(offset), .
```

```
      changed());

  wire [31:0] test_reg_1;
  setting_reg #(
    .my_addr(SR_TEST_REG_1), .awidth(8), .width(32))
  sr_test_reg_1 (
    .clk(ce_clk), .rst(ce_rst),
    .strobe(set_stb), .addr(set_addr), .in(set_data), .out(test_reg_1)
        , .changed());

  // Readback registers
  // rb_stb set to 1'b1 on NoC Shell
  always @(posedge ce_clk) begin
    case(rb_addr)
      8'd0 : rb_data <= {48'd0,
          offset}; 48 bits en cero + 16 bits de offset = 64 bits
      8'd1 : rb_data <= {32'd0, test_reg_1};
      default : rb_data <= 64'h0BADC0DE0BADC0DE;
    endcase
  end

  wire [15:0] i = m_axis_data_tdata[31:16]; //en fase
  wire [15:0] q = m_axis_data_tdata[15:0];  //en cuadratura
  reg [15:0] i_end_offset;
  wire [31:0] add_offset;
  wire [15:0] i_add_neg_offset, i_add_dif_offset, i_add_pos_offset;
  wire [15:0] suma = i + offset;
  wire [1:0] sel = i[15]+offset[15];

  assign i_add_neg_offset = {1'b1, suma[14:0] };
  assign i_add_dif_offset = suma;
  assign i_add_pos_offset = {1'b0, suma[14:0] };

  always @(sel, i_add_neg_offset, i_add_dif_offset, i_add_pos_offset)
    case(sel)
      2'd0: i_end_offset = i_add_pos_offset;
      2'd1: i_end_offset = i_add_dif_offset;
      2'd2: i_end_offset = i_add_neg_offset;
```

```
        default: i_end_offset = 16'b0;
      endcase

    assign add_offset = {i_end_offset,q};

    /* Simple Loopback */
    assign m_axis_data_tready = s_axis_data_tready;
    assign s_axis_data_tvalid = m_axis_data_tvalid;
    assign s_axis_data_tlast  = m_axis_data_tlast;
    assign s_axis_data_tdata  = add_offset;
endmodule
```

Se ha resaltado en negrilla los elementos que el usuario debe editar. Se estableció la dirección del registro offset en 130 pero es posible establecer cualquier dirección de registro de usuario que no incluya las direcciones reservadas para los registros de configuración 128 y 129.

4. A continuación se simula el bloque para verificar el funcionamiento y evitar errores en niveles superiores. Es importante revisar el Anexo G donde se explica detalladamente la estructura de la simulación. Para simular, primero se abre la plantilla de simulación, que puede realizarse mediante el comando:

```
$ gedit ~/rfnoc/src/rfnoc-tutorial/rfnoc/testbenches/
    noc_block_offset_tb/noc_block_offset_tb.sv
```

5. Después se edita para incluir el bloque y los estímulos a simular, tal como se ve a continuación:

```
    ...
    /////////////////////////////////////////////////////
    ** Verification
    ////////////////////////////////////////////////////*/
    initial begin : tb_main
      string s;
      logic [31:0] random_word;
      logic [63:0] readback;
```

```
logic [15:0] offset;
...
/*********************************************************
** Test 4 -- Write / readback user registers
*********************************************************/
`TEST_CASE_START("Write / readback user registers");
random_word = $random();
tb_streamer.write_user_reg(sid_noc_block_offset, noc_block_offset.
    SR_OFFSET, random_word[15:0]);
tb_streamer.read_user_reg(sid_noc_block_offset, 0, readback);
$sformat(s, "User register 0 incorrect readback! Expected: %0d,
    Actual %0d", readback[15:0], random_word[15:0]);
`ASSERT_ERROR(readback[15:0] == random_word[15:0], s);
...
/*********************************************************
** Test 5 -- Test sequence
*********************************************************/
// offset's user code is a loopback, so we should receive
// back exactly what we send
`TEST_CASE_START("Test sequence");
offset=2;
tb_streamer.write_user_reg(sid_noc_block_offset, noc_block_offset.SR_OFFSET,
    offset);
fork
  begin
    cvita_payload_t send_payload;
    for (int i = 0; i < SPP/2; i++) begin
      send_payload.push_back({i[15:0], 16'd0, i[15:0], 16'd0 });
    end
    tb_streamer.send(send_payload);
  end
  begin
    cvita_payload_t recv_payload;
    cvita_metadata_t md;
    logic [63:0] expected_value;
    logic [15:0] suma;
    tb_streamer.recv(recv_payload,md);
    for (int i = 0; i < SPP/2; i++) begin
```

```
            if (offset[15]+i[15] == 0) begin
                expected_value = {1'd0, i[14:0], 16'd0, 1'd0, i[14:0], 16'd0} + {1'd0,
                    offset[14:0], 16'd0, 1'd0, offset[14:0], 16'd0 };
            end
            else if (offset[15]+i[15] == 2) begin
                expected_value = {1'd1, i[14:0], 16'd0, 1'd1, i[14:0], 16'd0} +{1'd0,
                   offset[14:0], 16'd0, 1'd0, offset[14:0], 16'd0 };
            end
            else if (offset[15]+i[15] == 1) begin
                suma = i[15:0] + offset[15:0];
                expected_value = {suma[15:0], 16'd0, suma[15:0], 16'd0};
            end
            $display("El valor esperado es: %b", expected_value);
            $display("El valor obtenido es: %b", recv_payload[i]);
            $sformat(s, "Incorrect value received! Expected: %0d, Received: %0d",
                expected_value, recv_payload[i]);
            `ASSERT_ERROR(recv_payload[i] == expected_value, s);
        end
    end
join
`TEST_CASE_DONE(1);
`TEST_BENCH_DONE;
```

Aquí, se compara el valor esperado (que se construye realizando la suma dentro de la simulación) con el valor obtenido (la respuesta del entorno ante cierta configuración).

6. Una vez modificada la plantilla, se deben ejecutar los siguientes comandos:

```
$ cd ~/rfnoc/src/rfnoc-tutorial/
$ mkdir build
$ cd build
$ cmake ../
```

Estos comandos crean la carpeta *build* y los archivos internos. Si no se presentan errores, se deben mostrar los siguientes mensajes al final de la ejecución.

```
-- Configuring done
-- Generating done
-- Build files have been written to: ~/rfnoc/src/rfnoc-tutorial/build
```

7. Después se ejecuta el siguiente comando:

```
$ make test_tb
```

   Este modifica los archivos necesarios y establece las rutas correctas de las herramientas de simulación, donde se deben mostrar los siguientes mensajes después de su ejecución.

```
Scanning dependencies of target test_tb
Built target test_tb
```

8. Finalmente, se ejecuta la simulación mediante el comando:

```
$ sudo make noc_block_offset_tb
```

   Si no hay errores en la simulación, significa que no hay errores de sintaxis ni de funcionalidad y es posible empezar a editar las plantillas de niveles superiores.

   Los reportes de la simulación se almacenan en la dirección *~/rfnoc/src/rfnoc-Tutorial/rfnoc/testbenches/noc_block_offset_tb* con el nombre *xsim.log*.

   Se puede encontrar información detallada, en el Anexo G

3.3 Archivo XML nivel UHD

1. Primero se debe acceder a la plantilla del archivo *XML*, esto puede ser realizado ejecutando el siguiente comando:

```
$ gedit ~/rfnoc/src/rfnoc-tutorial/rfnoc/blocks/offset.xml
```

   Esta se edita según los parámetros establecidos en la plantilla de verilog, como se muestra.

```
...
  <registers>
    <setreg>
      <name>OFFSET</name>
      <address>130</address>
    </setreg>
  </registers>

 <args>
    <arg>
      <name>offset</name>
      <type>int</type>
      <value>1</value>
      <check>GE($offset, -32765) AND LE($offset, 32767)</check>
      <check_message>El offset debe estar entre [-32765, 32767]</check_message>
      <action>SR_WRITE("OFFSET", $offset)</action>
    </arg>
  </args>

  <ports>
    <sink>
      <name>in</name>
      <type>sc16</type>
    </sink>
    <source>
      <name>out</name>
      <type>sc16</type>
    </source>
  </ports>

</nocblock>
```

Se puede encontrar información detallada en el Anexo H.

3.4 Archivo XML nivel GNU-Radio

1. Se accede a la plantilla del archivo *XML* nivel *GNU-Radio*, esto es posible ejecutando el siguiente comando:

```
$ gedit ~/rfnoc/src/rfnoc-tutorial/grc/tutorial_offset.xml
```

2. Esta plantilla se edita según los parámetros establecidos en la plantilla de verilog, y en la plantilla *XML* nivel *UHD* como se muestra.

```
...
            $block_index,
            $device_index
)
self.$(id).set_arg("offset", $offset)
  </make>
  <callback>set_arg("offset", $offset)</callback>

  <param>
    <name>Valor de offset</name>
    <key>offset</key>
    <value>1.0</value>
    <type>real</type>
  </param>

...

  <sink>
    <name>in</name>
    <type>complex</type>
    <vlen>$grvlen</vlen>
    <domain>rfnoc</domain>
  </sink>

  <source>
    <name>out</name>
    <type>complex</type>
    <vlen>$grvlen</vlen>
    <domain>rfnoc</domain>
```

```
    </source>
</block>
```

Se puede encontrar información detallada en el Anexo I.

3.5 CREAR LA IMAGEN DEL FPGA

1. Primero se debe ingresar a la carpeta donde está localizado el constructor de la imagen del *FPGA*, por ejemplo mediante el comando:

```
$ cd ~/rfnoc/src/uhd-fpga/usrp3/tools/scripts
```

2. Una vez aquí, se debe ejecutar el siguiente comando para abrir la interfaz del constructor de la imagen.

```
$ sudo python3 uhd_image_builder_gui.py
```

3. Dentro de la interfaz se debe realizar lo siguiente:

   - Pulsar el botón *Add OOT Blocks* y seleccionar la carpeta del módulo que se desee añadir, que en este caso corresponde con la carpeta *rfnoc-tutorial*.
   - En el cuadro *Select build target* seleccionar *X310_RFNOC_HG*.
   - En el cuadro *List of blocks available* desplegar la opción *OOT Blocks for X300 devices* y seleccionar el bloque que se desee agregar, en este caso*offset*.
   - Pulsar en las opciones *Fill with FIFOs* y *Clean IP*.

   Lo anterior debe generar en el espacio *uhd_image_builder command* un comando como el siguiente:

```
./uhd_image_builder.py offset -I ~/rfnoc/src/rfnoc-tutorial/rfnoc -c
    --fill-with-fifos -t X310_RFNOC_HG -d X310
```

4. Finalmente se presiona el botón *Generate .bit file* que inicia el proceso de síntesis de la imagen. Para ver el estado del proceso de síntesis se debe revisar la terminal donde se ejecutó el comando al abrir la interfaz. Este proceso puede tardar alrededor de tres horas. Una vez finalizado el proceso, la imagen se

guarda en la dirección */rfnoc/src/uhd-fpga/usrp3/top/x300/build* con el nombre *usrp_x310_fpga_RFNOC_HG.bit*

Se puede encontrar información detallada en el Anexo F.

3.6 CARGAR LA IMAGEN AL FPGA E INSTALAR

1. Se debe conectar el *USRP x310* y configurar la red cableada.

2. Después ingresar el siguiente comando en la terminal.

```
$ uhd_usrp_probe
```

   Esto muestra las características del *USRP* conectado, incluida la dirección IP.

3. Para cargar la imagen se debe ingresar el siguiente comando:

```
$ uhd_image_loader –args "type=x300, addr=[Direccion IP]" –fpga–path ~/
    rfnoc/src/uhd–fpga/usrp3/top/x300/build/usrp_x310_fpga_RFNOC_HG.bit
```

4. Se debe reiniciar el *USRP* después de cargada la imagen. Finalmente se ingresan los comandos:

```
$ cd ~/rfnoc/src/rfnoc–tutorial/build
$ make install
```

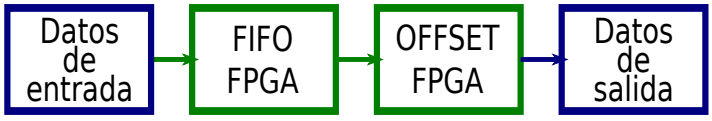
   Estos comandos realizan la instalación del bloque en *GNU-Radio* y permiten su utilización en la interfaz.

3.7 PRUEBA DEL BLOQUE EN GNU-RADIO

1. Una vez se ha reiniciado el USRP es posible hacer uso del bloque *offset* para lo cual se debe iniciar el programa *GNU-Radio companion* y dentro de la interfaz se realiza el diagrama de bloques:

   La prueba se realiza utilizando como datos de entrada la señal de la ecuación 3.1 y Figura 23, la cual es una exponencial compleja discreta con periodo $M = 20$,

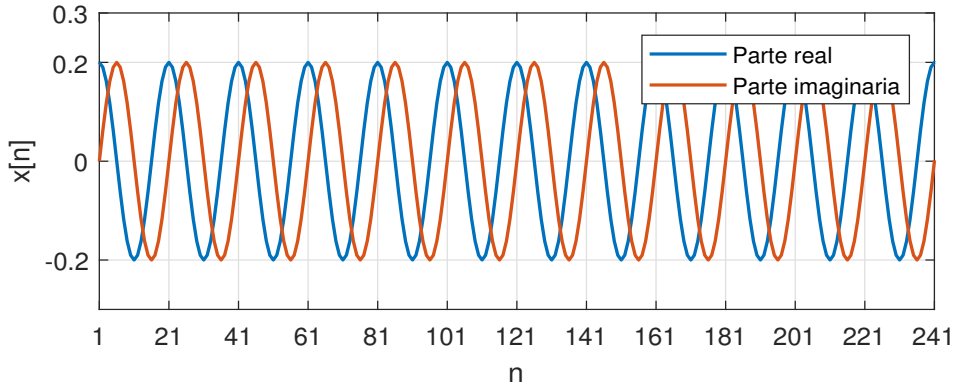
Figura 20: Diagrama de bloques prueba *offset*



que es una versión discreta de una exponencial compleja continua con frecuencia $f = 100k[Hz]$, muestreada a una frecuencia de muestreo $f_s = 2M[samp/s]$.

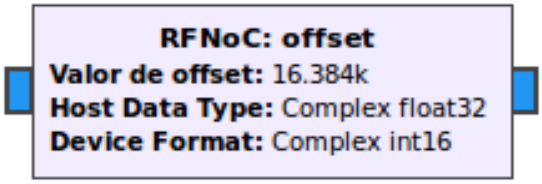
$$x[n] = 0{,}2exp(2\pi \times 0{,}05jn) \qquad n = 0, 1, \ldots, 241 \tag{3.1}$$

Figura 21: Datos de entrada prueba *offset*



2. El bloque se configura en la interfaz de GNU-Radio, según la Figura 22.

Figura 22: Configuración bloque *offset* en GNU-Radio companion

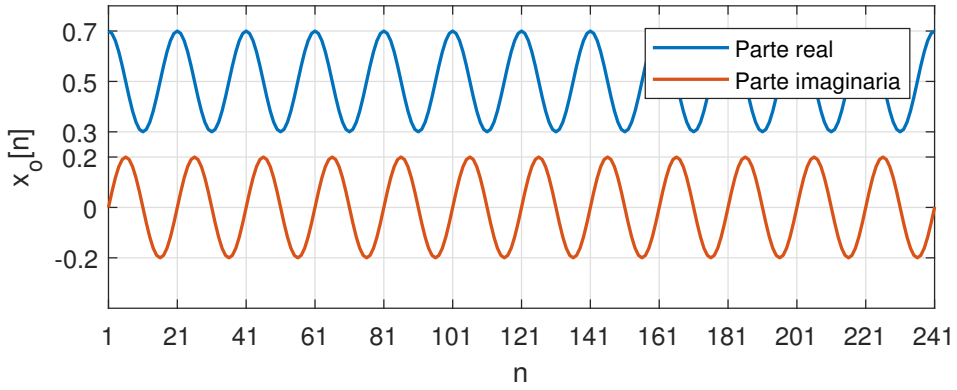


3. Se establece un valor de offset de $16{,}384k$ que corresponde con un valor en flotante de $16{,}384k \times 2^{-15} = 0{,}5$ (ver Anexo F). Esto significa que se espera obtener una señal equivalente a la de la ecuación 3.2.

$$x_o[n] = 0{,}2cos(2\pi \times 0{,}05n) + 0{,}5 + j0{,}2sen(2\pi \times 0{,}05n) \qquad n = 0, 1, \ldots, 241 \tag{3.2}$$

Graficando los datos de salida se obtiene:

Figura 23: Datos de salida prueba *offset*



De la Figura 23 se puede comprobar que la señal se ve afectada por el valor de offset establecido en el bloque de la Figura 22.

4. PRUEBAS

En este capítulo se presentará la descripción de las pruebas realizadas y el análisis de los resultados para la verificación del correcto funcionamiento del bloque RFNoC FFT desarrollado en este proyecto, su consumo de recursos de *FPGA*, y la exploración de características de la arquitectura RFNoC, su comportamiento en escenarios específicos y el contraste con la FFT en *GPP* de *GNU-Radio*.

Las pruebas presentadas en este capítulo se realizaron, a menos que se indique lo contrario, bajo las siguientes condiciones:

- *USRP* X310 utilizando la interfaz ethernet 1Gigabit, con daugtherboards CBX y SBX.
- Computador A, de mesa, con un Intel Pentium 4 GPP 3.2GHz $\times$ 2 de 32 bits, 3.4 GiB de RAM, con una tarjeta Broadcom NetXtreme BCM5751 Gigabit Ethernet y sistema operativo Ubuntu 16.04 LTS.
- Computador B, portátil, con un Intel Core i5-2450M GPP 2.5GHz $\times$ 4 de 64 bits, 3.4 GiB de RAM, con una tarjeta Qualcomm Atheros AR8151 Gigabit Ethernet y sistema operativo Ubuntu 16.04 LTS.
- UHD-FPGA: rfnod-devel branch commit 0cac477
- UHD: rfnoc-devel branch, commit 1908672
- gr-ettus: master branch, commit 814a7de
- *GNU-Radio* 3.7.111
- Datos fuente en un formato punto fijo SC16

4.1 RECURSOS

Para esta prueba se sintetizaron cuatro imágenes RFNoC utilizando la estrategia de síntesis definida por la arquitectura. La configuración base, no contiene bloques RFNoC diferentes a los que trae por defecto una imagen (Radio, DDC, DUC, etc), las imágenes FFT, FIFO y FFT&FIFO contienen: un bloque RFNoC con la FFT di-

señada en el Capítulo 2, el bloque FIFO de ettus y ambos bloques respectivamente, obteniendo los recursos utilizados por estas imágenes en el Cuadro 13.

Cuadro 13: Porcentajes de utilización

Recurso	Base	FFT	FIFO	FFT&FIFO
Slice LUTs	36,87	39,79	38,30	41,48
Slice Registers	20,08	21,86	20,83	22,74
Block RAM Tile	38,93	42,33	40,64	43,14
DSPs	4,61	7,73	4,61	7,73

De lo anterior, se observa que el porcentaje de uso de cada bloque RFNoC respecto a la configuración base, no es constante cuando se tienen configuraciones diferentes. Sin embargo, si los recursos utilizados por la configuración base, se comparan con las imágenes que contienen los bloques FIFO y FFT de forma individual, se encuentra que el uso de estos bloques no es mayor al 2 % y 4 % respectivamente. La mayor restricción de recursos se presenta en los bloque de RAM dedicados, con un 61,07 % libre y la menor es de módulos DSP con un 95,49 % libre. La FFT utiliza un 3,12 % de módulos DSP, y teniendo en cuenta su complejidad, se encuentra una gran cantidad de espacio para el desarrollo de bloques que puedan aprovechar este tipo de recursos.

4.2 VERIFICACIÓN

La verificación del funcionamiento del bloque FFT con arquitectura RFNoC se realizó confirmando el cumplimiento de las propiedades de linealidad y desplazamiento temporal, además de la respuesta de un impulso a la entrada. Estas pruebas son suficientes para este fin, según Ergün[27]. Adicionalmente, se verificó la exactitud del bloque, comparando resultados obtenidos con los de la DFT [28].

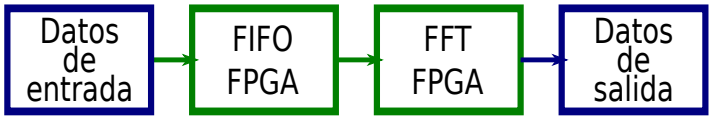
Las pruebas de verificación se realizaron con el flujograma de la Figura 24, utilizando datos tipo *sc16*. Todos los datos de prueba de entrada, a menos que se indique lo

[27] ERGÜN, Funda. Testing multivariate linear functions: Overcoming the generator bottleneck. In Proc. Twenty-Seventh Ann. ACM Symp. 1995, Junio. p. 407–416

[28] HSW, Hwei; MEHRA, Raj; VELASCO, Federico. Análisis de Fourier (1987)

contrario, poseen una amplitud igual al rango máximo positivo de un dato *sc16* que es igual a $2^{15} - 1$ ó 32767 y un tamaño de transformada N igual a 256. Esta amplitud impar genera errores esperados en la comparación muestra a muestra realizada en las siguientes tres pruebas de validación. Estos errores provienen del escalamiento por una potencia de dos y el tipo de redondeo utilizado por cada implementación, que se verán con mas detalle en la sección 4.2.4.

Figura 24: Flujograma verificación



4.2.1 Linealidad El objetivo de esta prueba es verificar si la relación de (4.5) se cumple. Donde $x_1[n]$ y $x_2[n]$ son las exponenciales complejas de (4.1) y (4.2) cuya parte real e imaginaria se ve en las Figuras 25 y 26; a y b son constantes con valor $0,7$ y $0,3$ respectivamente.

Figura 25: Señal $x_1[n]$ dominio del tiempo

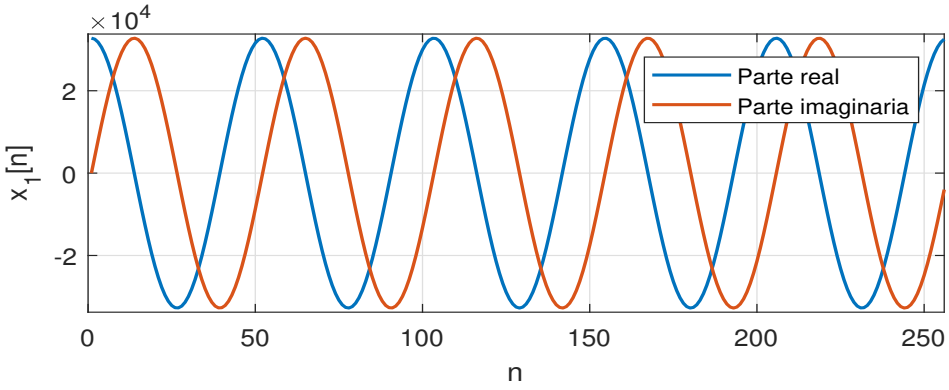
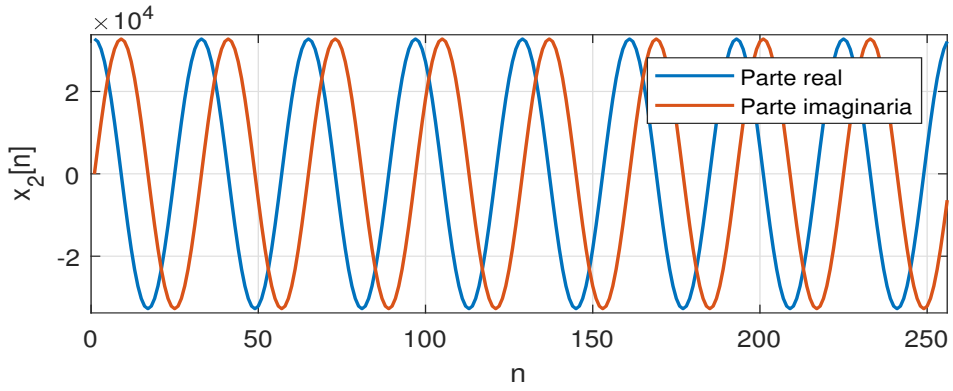


Figura 26: Señal $x_2[n]$ dominio del tiempo



$$x_1[n] = exp(\frac{10\pi jn}{256}) \qquad n = 0, 1, \ldots, 255 \tag{4.1}$$

$$x_2[n] = exp(\frac{16\pi jn}{256}) \qquad n = 0, 1, \ldots, 255 \tag{4.2}$$

$$X_3[k] = FFT(ax_1[n] + bx_2[n]) \qquad k = 0, 1, \ldots, 255 \tag{4.3}$$

$$X_4[k] = aFFT(x_1[n]) + bFFT(x_2[n]) \qquad k = 0, 1, \ldots, 255 \tag{4.4}$$

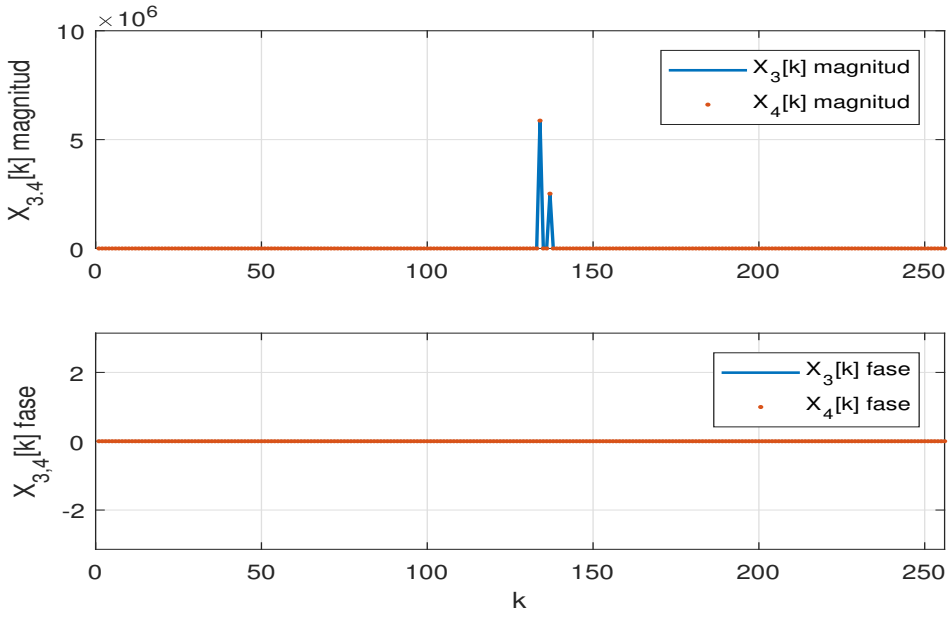
$$X_3[k] = X_4[k] \tag{4.5}$$

Los resultados obtenidos en magnitud y fase de $X_3[k]$ y $X_4[k]$ se pueden ver en la Figura 27.

La Figura 27 confirma la propiedad de linealidad, debido a la similitud en las respuestas de magnitud y fase de $X_3[k]$ y $X_4[k]$ con un porcentaje de error muestra a muestra relativo medio del 0 %.

4.2.2 Desplazamiento temporal El objetivo de esta prueba es verificar si la relación (4.7) se cumple. Donde $X[k]$ se obtiene de (4.6), m es el desplazamiento en el tiempo, j la unidad imaginaria, y N el tamaño de la FFT. En este caso, $x[n-m]$ es un impulso desplazado como se puede ver en la Figura 28, N y m son constantes con valor de 256 y 15 respectivamente.

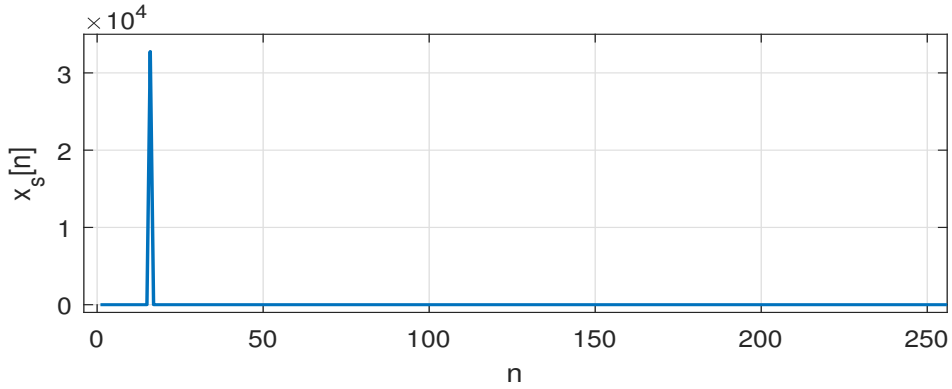
Figura 27: Magnitud y fase de $X_3[k]$ y $X_4[k]$



$$X[k] = FFT(x[n]) \tag{4.6}$$

$$FFT(x[n-m]) = X[k]exp(-j\frac{2\pi}{N}km) \tag{4.7}$$

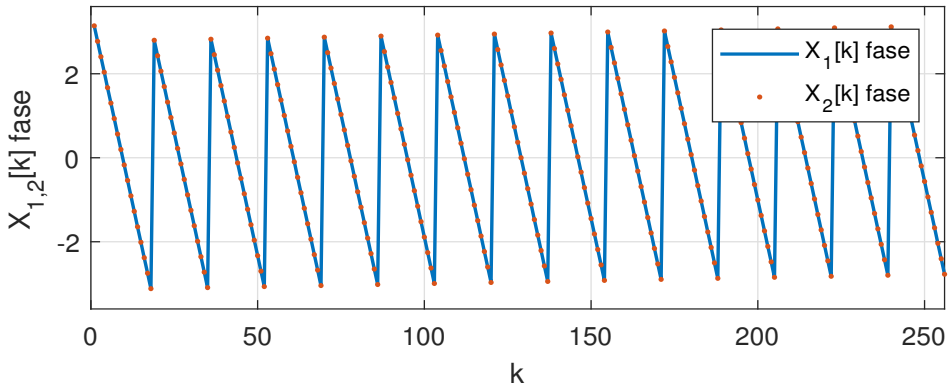
Figura 28: Señal desplazada en el tiempo



Los resultados obtenidos en fase de la transformada de la señal desplazada y la

transformada de la señal sin desplazar ajustada, se pueden ver en la Figura 29. Se obvian los resultados de magnitud, debido a su irrelevancia en esta prueba y a su valor constante.

Figura 29: Fase de las señales $FFT(x[n-m])$ y $X[k]exp(-j\frac{2\pi}{N}km)$



La Figura 29 confirma la propiedad de desplazamiento debido a la similitud en la respuesta de fase de ambas señales, con un porcentaje de error relativo muestra a muestra medio de 0.36 %. Este error relativamente grande respecto a la prueba anterior, se da por el paso de una representación real-imaginaria a una de magnitud-fase, siendo la operación de cambio a fase, realizada en MATLAB, la más sensible.

4.2.3 Respuesta al impulso El objetivo de esta prueba es verificar si la relación (4.8) se cumple, donde c es una constante igual a la amplitud del impulso que posee un valor de 32767 y N igual a 256. La función $c\delta[n]$ se puede ver en la Figura 30.

$$FFT(c\delta[n]) = \frac{c}{N} \tag{4.8}$$

Aplicando la FFT a la señal de impulso, se obtiene la respuesta que se puede ver en la Figura 31.

De esta forma se obtiene el valor de $\frac{c}{N}$ igual 128, que multiplicado por el valor de N resulta en un c igual a 32768, esta pequeña diferencia proviene del escalamiento par

Figura 30: Señal de impulso utilizada

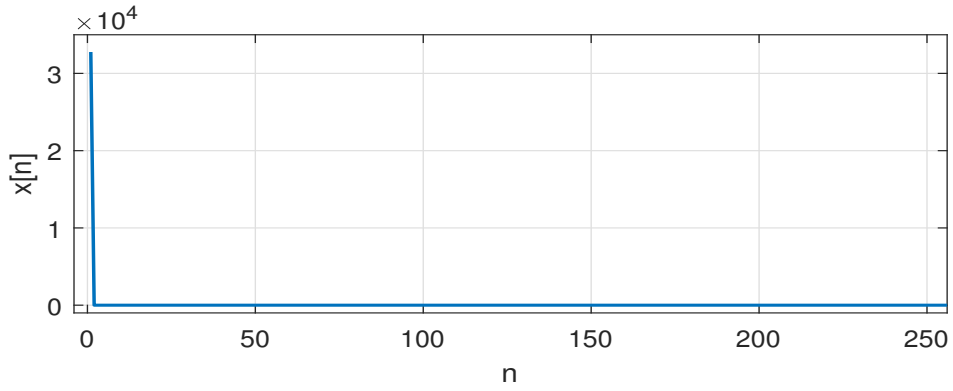
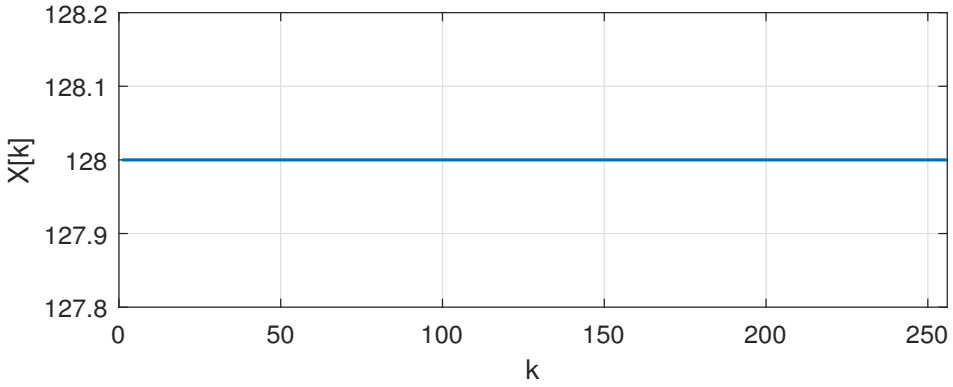


Figura 31: Respuesta al impulso magnitud



aplicado a un valor impar y su redondeo. Este resultado verifica que la respuesta al impulso corresponde con el valor esperado, con un error relativo de $0,003\,\%$.

4.2.4 Exactitud El objetivo de esta prueba es calcular la exactitud del bloque FFT con arquitectura RFNoC comparándolo con la DFT mediante el cálculo de la media de la señal de error relativo separada en parte real e imaginaria con diferentes valores de N. En la Figura 32 se observa la entrada para $N = 256$, que corresponde con una señal de impulso desplazada una muestra; y en la Figura 33 la parte real e imaginaria de la función de error para el mismo N.

Si se calcula la media a la parte real e imaginaria de la señal de error relativo y se

Figura 32: Señal de entrada para $N = 256$

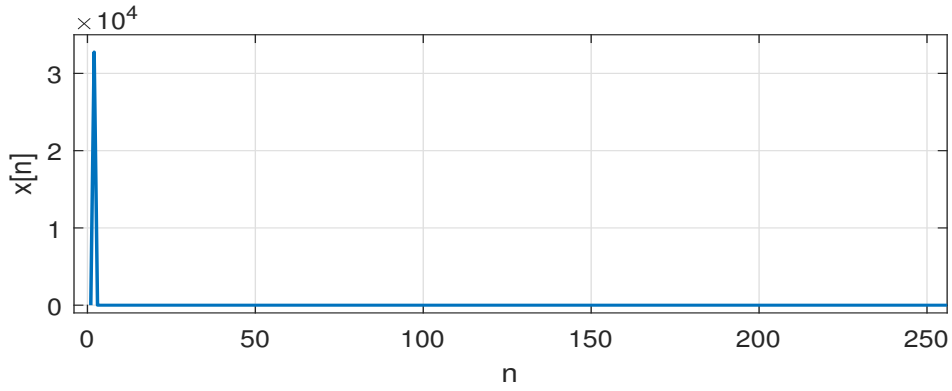
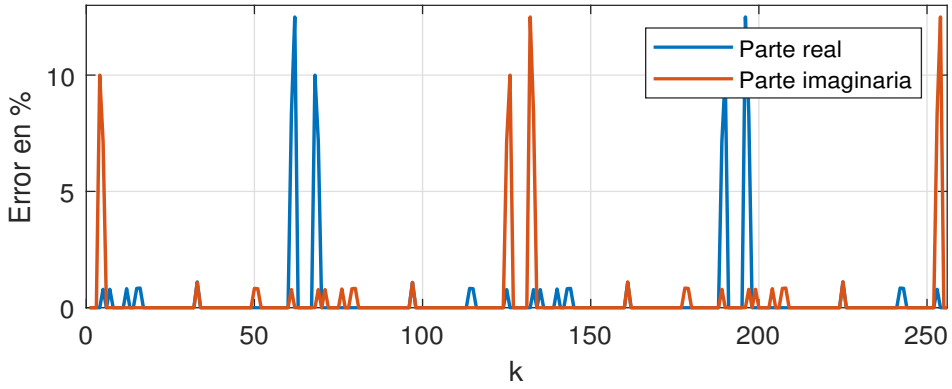


Figura 33: Señal de error relativo para $N = 256$

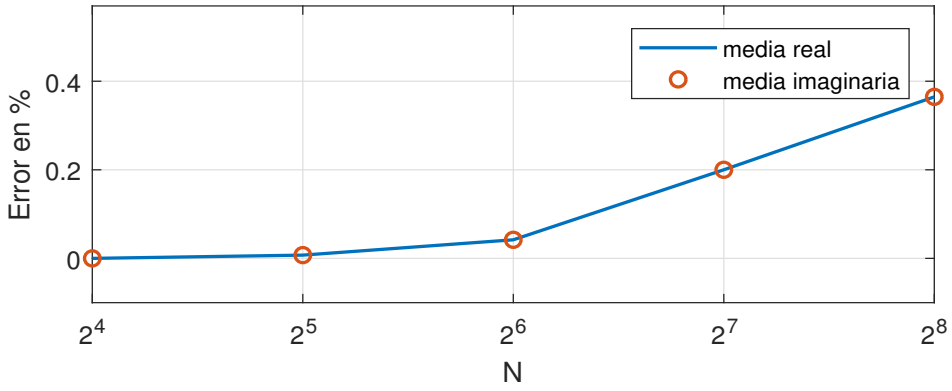


hace el mismo procedimiento para distintos valores de N, se obtienen los resultados de la Figura 34.

Lo anterior muestra una relación directa entre el error relativo y el valor de N, con un error mínimo de $0\,\%$ para $N = 16$ y máximo de $0,3647\,\%$ para $N = 256$. Esta relación directa se debe al escalamiento aplicado, que es igual a N, por lo tanto al aumentar N disminuye en la misma proporción el rango de la señal, aumentando el error por precision y redondeo. Teniendo en cuenta la diferencia de redondeo entre las dos implementaciones, convergente para la FFT [29] y mitad hacia arriba para la DFT, son

[29] XILINX INC. Fast Fourier Transform v9.0 LogiCORE IP Product Guide. En: Xilinx [En línea]. (2015) [Consultado 23 Abr. 2018]. Disponible en: `http://www.ti.com/litv/pdf/sprabb6b`

Figura 34: Media señal de error relativo diferentes N



esperados los picos de error alrededor de cero, mostrados en la Figura 33.

4.3 DESEMPEÑO

En esta sección se realiza una serie de pruebas que exploran las características de la arquitectura RFNoC utilizando el bloque *FFT* desarrollado con arquitectura RFNoC, juntos con los bloques genéricos, y el de las librerías de *GNU-Radio*, en aspectos como tiempo de procesamiento y latencia bajo diferentes escenarios. Estos escenarios se definen por los dispositivos por los que fluye los datos a ser procesados.

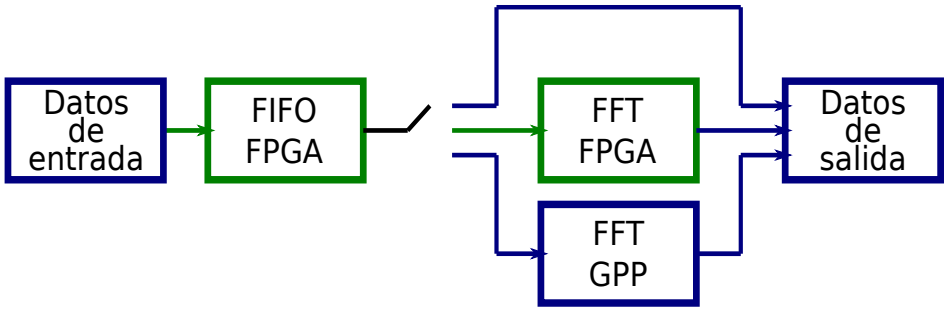
Estos dispositivos pueden ser:

- **Servidor:** El computador al que esta conectado el USRP.
- **FPGA:** Unidad de procesamiento del USRP.
- **RX:** Conjunto de recepción RF y ADC.
- **TX:** Conjunto de transmisión RF y DAC.

4.3.1 Servidor-FPGA-Servidor

Throughput con número definido de muestras El objetivo de esta prueba es comparar el tiempo que toma procesar 2^{16} muestras al bloque *FFT* y *FIFO* de RFNoC, y al bloque *FFT* de *GNU-Radio*; variando el tamaño del vector o paquete de datos N y la frecuencia de muestreo F_s en *GNU-Radio* para cada prueba. Para esto se diseñó el flujograma mostrado en la Figura 35 que opera de la siguiente forma: el paquete de datos de prueba se lee desde un archivo, luego se envía al *USRP* donde pasa a través de un bloque *FIFO* para luego volver al servidor, o para ser procesado ya sea en el bloque *FFT* de RFNoC o *FFT* de *GNU-Radio* y volver al servidor.

Figura 35: Flujograma Servidor-FPGA-Servidor



La toma de tiempos se realiza editando el código fuente del flujograma, utilizando la librería *time* de Python y el *API* de *GNU-Radio* para el criterio de parada. Cada muestra de tiempo se toma del promedio de veinte pruebas individuales con los mismos parámetros. Los resultados de tiempo para los tres bloques a diferentes frecuencias de muestreo se pueden ver en las Figuras 36, 37, 38, y 39

Bajo las tres configuraciones no se observa alguna diferencia significativa en el rendimiento; todas poseen, dentro del rango de error, tiempos de procesamiento similares. Las diferencias puntuales entre pruebas provienen de las variaciones en la tasa de transferencia inherentes del protocolo Ethernet. Sin embargo, se observa una tendencia decreciente en el tiempo de procesamiento a medida que aumenta el tamaño de la transformada, y una tendencia decreciente en el tiempo de procesamiento al aumentar la frecuencia de muestreo. A partir de esta similitud en tiempo de procesamiento entre los tres escenarios, uno de control y dos de prueba, se

Figura 36: Tiempo de procesamiento para $F_s = 2 \times 10^4$ con diferentes N

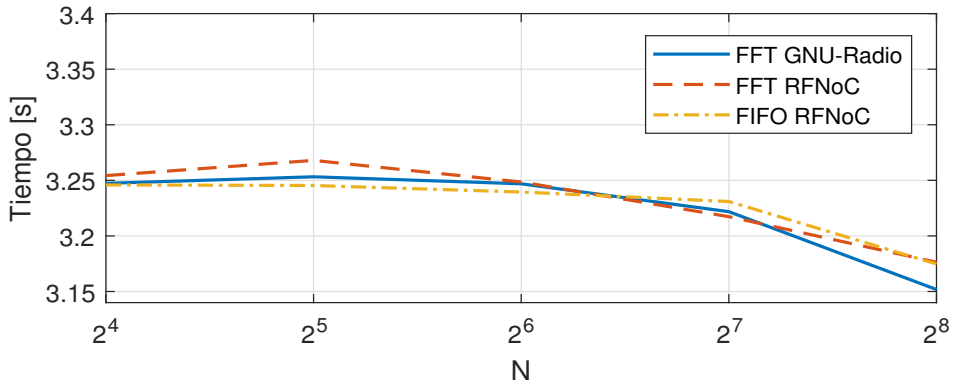


Figura 37: Tiempo de procesamiento para $F_s = 2 \times 10^5$ con diferentes N

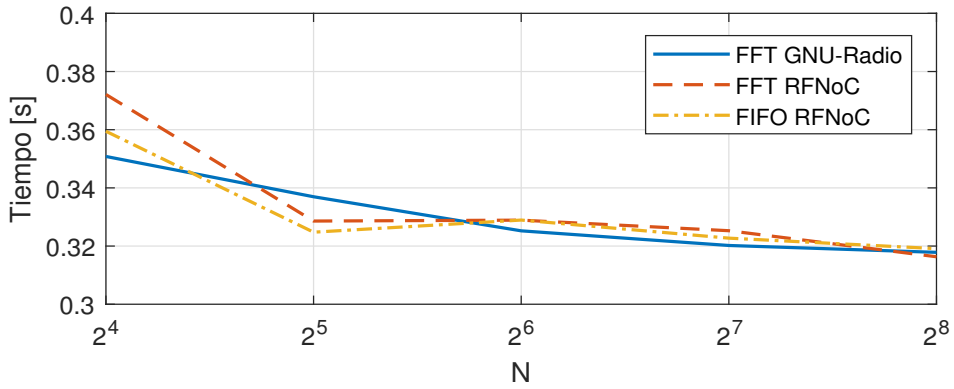


Figura 38: Tiempo de procesamiento para $F_s = 2 \times 10^6$ con diferentes N

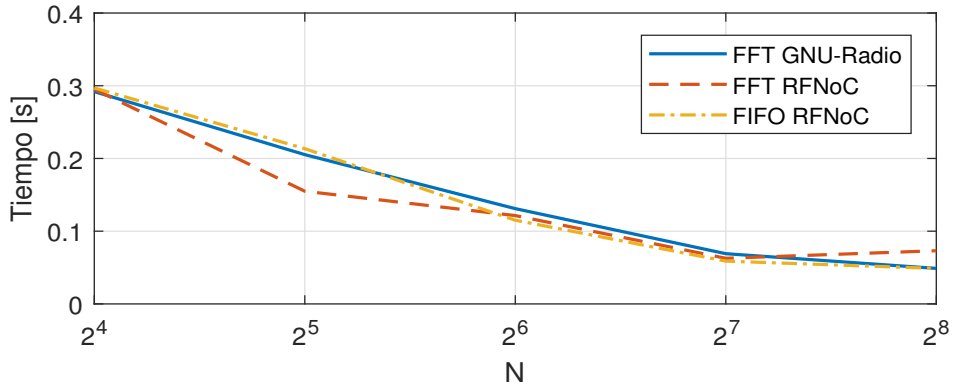
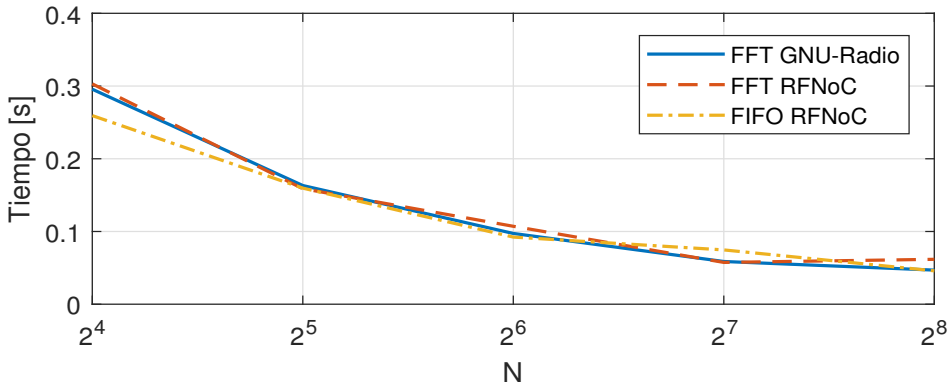


Figura 39: Tiempo de procesamiento para $F_s = 2 \times 10^7$ con diferentes N



encuentra este tiempo no esta ligado a cuál bloque procesa los datos. El procesamiento de cada bloque es mucho más rápido (Cuadro 10 y Frigo[30]) respecto al de transferencia por lo tanto la diferencia en conjunto seria despreciable. Se presenta estancamiento de rendimiento a los 2 MS/s, cuando el limite teórico se debería dar alrededor de los 25 MS/s que ofrece la interfaz 1 Gigabit Ethernet[31], se plantean las pruebas de las dos siguientes secciones.

Tasa de transferencia Ethernet Esta prueba pretende encontrar la tasa de transferencia en la interfaz Ethernet para la máxima frecuencia de muestreo teórica de 25 MS/s en función de seis tamaños de paquete equidistantes respecto al máximo valor posible.

Figura 40: Flujograma en prueba de tasa de transferencia Ethernet

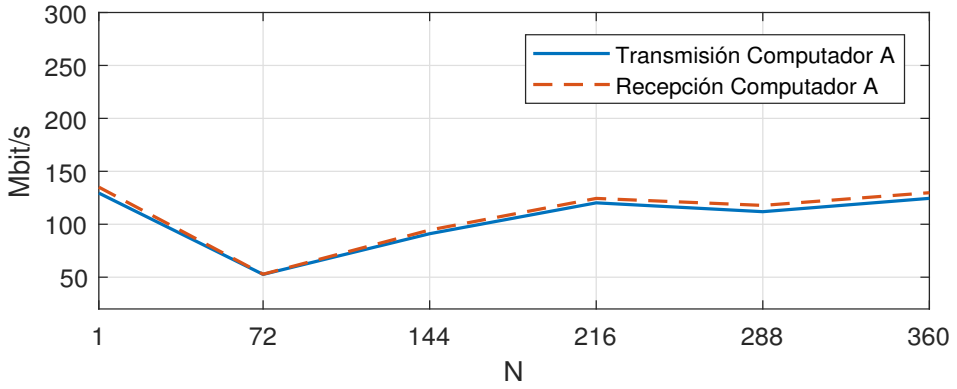
Con un flujo de datos continuo hacia y desde el *USRP* como el mostrado en la Figura

[30] FRIGO, Matteo; JOHNSON, Steven. FFT Benchmark Results. [En línea]. (2017) [Consultado 23 Abr. 2018]. Disponible en: `http://www.fftw.org/speed/`

[31] ETTUS KNOWLEDGE BASE CONTRIBUTORS. About USRP Bandwidths and Sampling Rates. En: Ettus Knowledge Base [En línea]. (2016) [Consultado 25 Abr. 2018]. Disponible en: `https://kb.ettus.com/index.php?title=About_USRP_Bandwidths_and_Sampling_Rates&oldid=2544`

40, y con la herramienta *VNSTAT*, que permite medir datos estadísticos en una interfaz Ethernet sobre un tiempo determinado, se obtuvo la tasa de transferencia promedio en transmisión y recepción durante un lapso de veinte segundos con diez pruebas individuales para cada N, se obtuvieron los resultados mostrados en la Figura 41.

Figura 41: Promedio velocidad transmisión-recepción

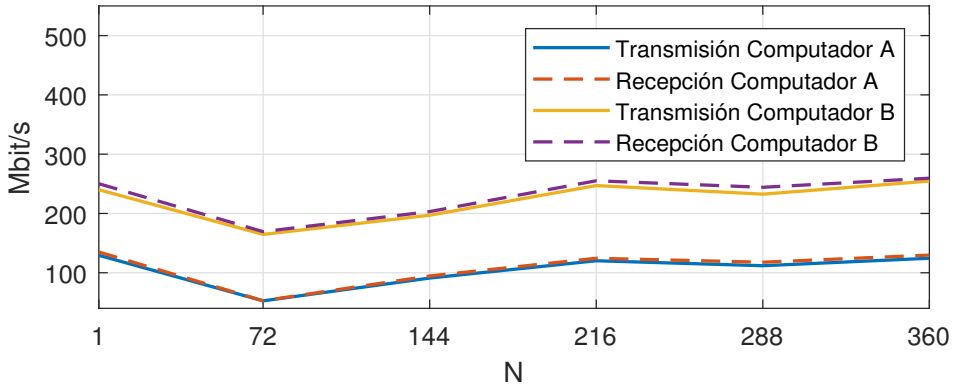


Se observa una relación entre la tasa de transferencia y el tamaño del paquete en todas las muestras, excepto en la que posee un tamaño de paquete igual a uno. Esta excepción se debe a que un tamaño de paquete igual a una muestra, en *GNU-Radio*, se considera un flujo continuo de datos llamado *Stream* y RFNoC selecciona el tamaño máximo de paquete para la transferencia por Ethernet. La relación proporcional proviene, en parte, de la razón entre la longitud del cabecero CHDR y la de los datos del paquete, que disminuye el *Throughput*.

Tasa de transferencia Ethernet para un servidor diferente Se realizó la prueba anterior, bajo las mismas condiciones, cambiando solo el servidor al computador B. Se obtuvieron los datos mostrados en la Figura 42 donde se comparan con los resultados de la sección 4.3.1.

Los datos indican un aumento en la tasa de transferencia de transmisión y recepción en el computador B respecto al computador A. Por lo tanto, el hardware del servidor afecta en gran medida el rendimiento de una implementación en RFNoC, limitando

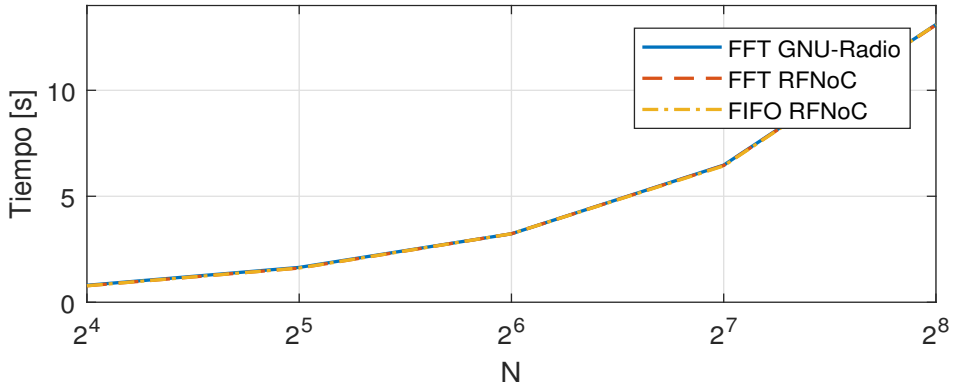
Figura 42: Promedio velocidad transmisión-recepción Computadores A y B



el *Throughput*.

Throughput con número definido de paquetes Esta prueba pretende complementar los resultados obtenidos en la prueba de Throughput con número definido de muestras. La prueba se realiza bajo los mismos parámetros y permite comparar el tiempo de procesamiento de 2^{10} paquetes. Cada muestra dentro de las gráficas es el promedio de diez pruebas individuales para cada N y cada gráfica tiene asociada una frecuencia de muestreo. Los resultados de tiempo para los tres bloques a diferentes frecuencias de muestreo se pueden ver en las Figuras 43, 44, 45, y 46.

Figura 43: Tiempo de procesamiento para $F_s = 2 \times 10^4$ con diferentes N



A diferencia de las pruebas anteriores, se puede observar un aumento en el tiem-

Figura 44: Tiempo de procesamiento para $F_s = 2 \times 10^5$ con diferentes N

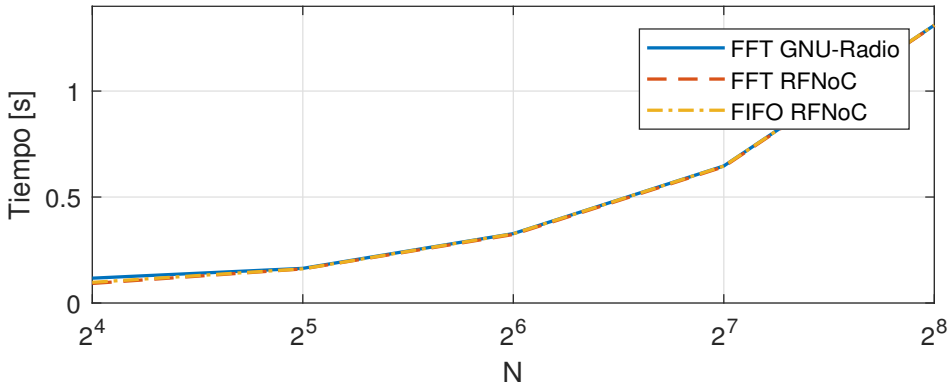


Figura 45: Tiempo de procesamiento para $F_s = 2 \times 10^6$ con diferentes N

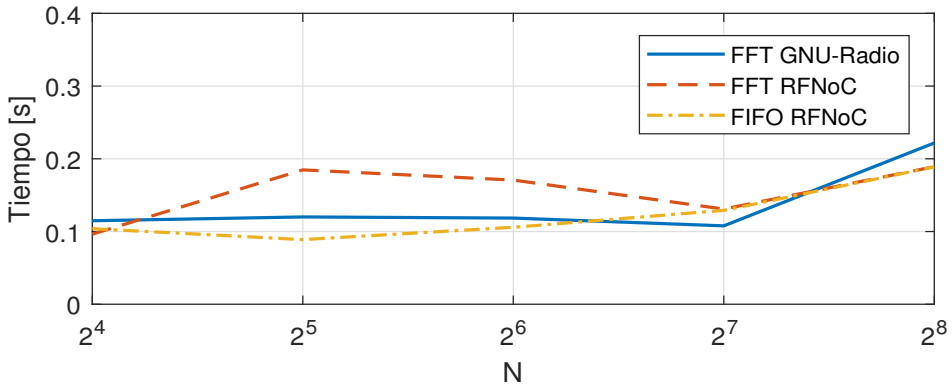
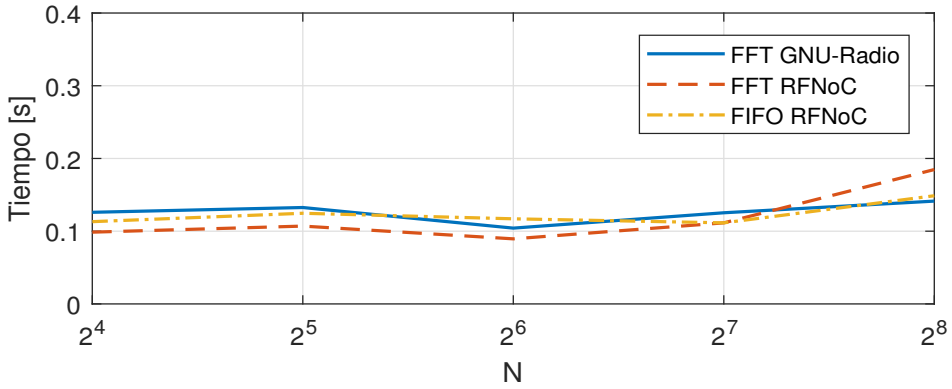


Figura 46: Tiempo de procesamiento para $F_s = 2 \times 10^7$ con diferentes N

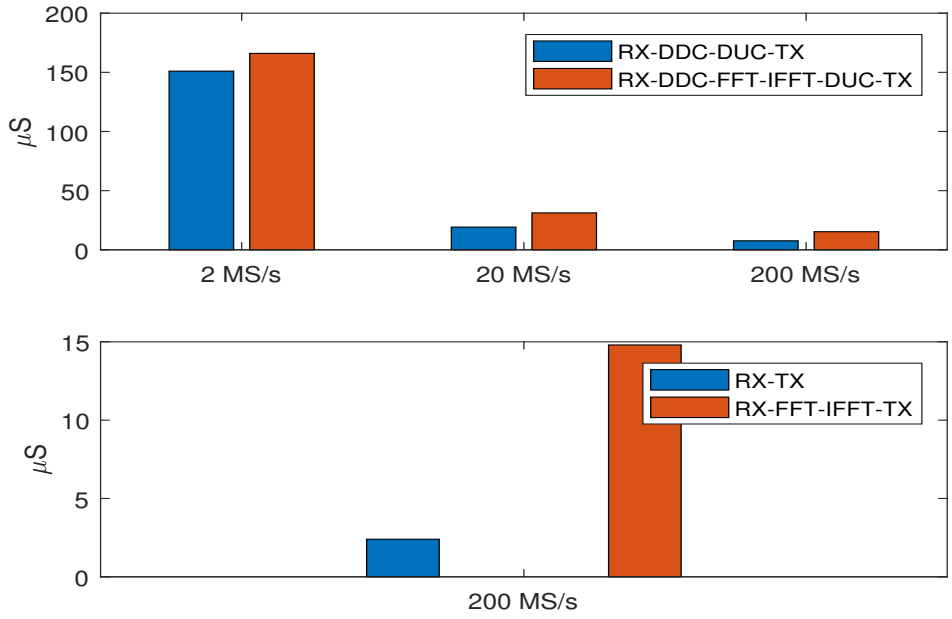


po de procesamiento a medida que aumenta el parámetro N, esto es esperado y se debe a que un N más grande implica mayor cantidad de muestras procesadas por cada paquete. Sin embargo, se observa que con una frecuencia de muestreo igual o superior a 2×10^6, el tiempo de procesamiento se mantiene. Esto concuerda con los resultados obtenidos en la Figura 40 donde se obtiene un rango de tasa de transferencia transmitida-recibida para el computador A de 50-120 Mbit/s, que considerando 32 muestras por paquete se obtiene un rango de frecuencia de muestreo de $1{,}5 \times 10^6$-$3{,}75 \times 10^6$, por lo que aumentar la frecuencia de muestreo más allá de este rango no supone ventaja alguna.

4.3.2 RX-FPGA-TX Utilizando RFNoC es posible realizar todo el procesamiento digital de señales en el FPGA eliminando el flujo de datos hacia y desde el servidor, siendo este utilizado solo para la configuración del USRP (Ver anexo F). Esta prueba mide la latencia de procesamiento de diferentes configuraciones entre la recepción y transmisión. La señal recibida por el USRP es una señal cuadrada 1 kHz con ciclo de trabajo de 10 %, modulada en amplitud y portadora de 450 MHz.

Con una resolución de 0,1 μs en el osciloscopio, la diferencia temporal entre el flanco de subida pulso de entrada y de salida es medida veinte veces, y su promedio, para diferentes configuraciones, se muestra en la Figura 47. La FFT e IFFT tienen deshabilitado el desplazamiento y computan una transformada de 256 muestras. Los bloques DDC y DUC diezman la frecuencia de procesamiento del FPGA de 200MS/s, y son ignorados en el escenario RX-TX, y en el RX-FFT-IFFT-TX que funcionan a la frecuencia máxima.

Figura 47: Latencia diferentes configuraciones

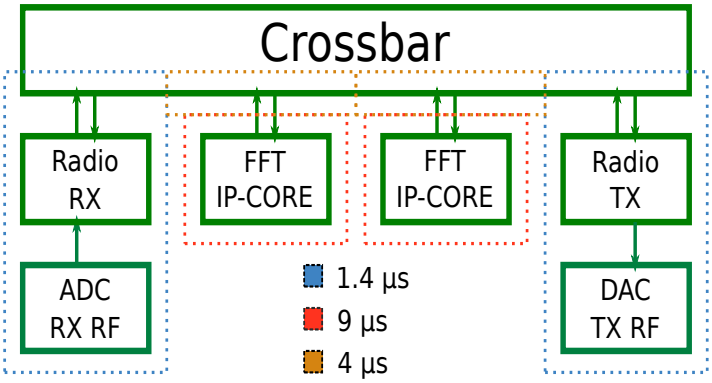


Se observa que la latencia disminuye al aumentar la frecuencia de procesamiento, dado que el aumento de frecuencia implica menos tiempo en realizar un mismo proceso. Una aproximación de la latencia RX-FFT-IFFT-TX se puede obtener a partir de la latencia experimental RX-TX, el retardo de comunicación entre bloques RFNoC y la latencia de la transformada en sí, obteniendo una latencia aproximada de 14,4 μs, como se muestra en la Figura 48.

Según Sunderlin[32], la latencia experimental entre dos bloques RFNoC es de aproximadamente 2 μs. Esta latencia representa el retardo que agrega el *noc shell*, *axi wrapper* y *crossbar* a una señal desde que entra a un bloque RFNoC hasta la entrada del siguiente bloque en el flujo, que de acuerdo a lo mostrado en el Cuadro 10 la latencia mínima de la transformada es de 4,5 μs. La diferencia entre la latencia aproximada y la medida es mínima, confirmando la rapidez de la transformada en este escenario.

[32]SUNDERLIN, Josh. Measured Latency Introduced by RFNoC Architecture. [En línea]. (2017) [Consultado 23 Abr. 2018]. Disponible en: `https://www.gnuradio.org/wp-content/uploads/2017/12/Jarriel-Cook-Measured-Latency-Introduced-by-RFNoC-Architecture.pdf`

Figura 48: Latencia de una configuración RX-FFT-IFFT-TX



5. CONCLUSIONES

En este documento se exploró el entorno de trabajo RFNoC como una posibilidad para la aceleración de SDR, específicamente dispositivos *USRP* de tercera generación. Para esto se desarrolló un bloque RFNoC que computa la FFT, dada su importancia en el procesamiento digital de señales y como prueba de concepto. Se documentó el entorno de trabajo RFNoC en el capítulo 1 y en los anexos; se describió la implementación de la transformada rápida de Fourier en *FPGA* en el capítulo 2 y se realizó una recopilación de los pasos para el desarrollo de un bloque RFNoC en el capítulo 3. Por ultimo se realizaron pruebas para verificar el funcionamiento de la FFT en el bloque RFNoC, pruebas de comparación con la FFT de GNU-Radio y pruebas de desempeño de la arquitectura RFNoC en el capítulo 4.

- A partir de la documentación condensada en los capítulos 1 y 3, y en los anexos G, H e I, donde se describe a detalle la arquitectura RFNoC, se pudo evidenciar que el procedimiento de acople de los tres niveles de desarrollo se reduce a la modificación de tres plantillas cuya complejidad es independiente de la implementación a realizar. Esto hace posible enfocar el esfuerzo de cualquier diseño de bloques RFNoC al desarrollo de HDL.

- En la sección 4.3 se realizaron pruebas con la arquitectura RFNoC utilizando diferentes flujogramas de la herramienta gráfica *GNU-Radio Companion*, donde el procesamiento tiene lugar de manera simultanea tanto en el *GPP* del computador que aloja el software, como en el *FPGA* del *USRP*. Esto verificó la posibilidad de procesamiento heterogéneo que ofrece RFNoC y la completa integración con la herramienta gráfica.

- En los capítulos 2 y 3 se realizó la implementación de los bloques *FFT* y un sumador de *offset*. Adicionalmente, el entorno contiene bloques ejemplo, como generadores de señales, multiplicadores, bloques *FIFO*, etc. Por esto, se puede

concluir que es posible implementar diseños con diferentes niveles de complejidad, cantidad de recursos y características.

- De acuerdo con la tendencias de la Figura 41 se concluye que existe una relación directa entre el tamaño del paquete *CHDR* y el *throughput* de la interfaz ethernet; así, para alcanzar el mayor rendimiento, es necesario utilizar el tamaño de paquete mas cercano al MTU.

- En la sección 4.3.1, en el apartado donde se calcula el *throughput*, a partir de las Figuras 36, 37, 38, y 39, y complementando con la sección 4.3.1, se evidencia que aunque el procesamiento ocurra únicamente en el *FPGA*, el rendimiento de cualquier proceso RFNoC cuyo flujo de datos incluya el computador (servidor-*FPGA*-servidor, servidor-*FPGA*-TX o RX-*FPGA*-servidor), estará limitado por la interfaz ethernet y el *hardware* del computador. Sin embargo, la configuración RX-*FPGA*-servidor no se ve tan afectada por esta limitante, debido a que aunque la transferencia de datos enviados desde el *FPGA* al servidor si se limite, el procesamiento en el *FPGA* no lo hace, resultado ventajoso, en un caso que se requiera realizar un procesamiento de alto rendimiento en el *FPGA*, y luego reducir la cantidad de datos para su envío al servidor.

- La configuración RX-*FPGA*-TX no tiene el limitante del *throughput* de la interfaz ethernet, por lo que puede alcanzar el máximo *throughput* de 200 MS/s del USRP. A partir de esto, se realizaron pruebas con diferentes frecuencias de procesamiento, encontrando una relación directa con la latencia.

6. RECOMENDACIONES

- El proyecto muestra una serie de resultados basados en diversas configuraciones con los que se busca mostrar los escenarios donde el procesamiento en el FPGA de un USRP X310 puede resultar más ventajoso que en un GPP. Sin embargo, estos resultados se ven limitados por la tasa de transferencia de la conexión Ethernet, con estandar 1Gigabit Ethernet, y por las características del servidor utilizado. Es necesario realizar nuevas pruebas que complementen este trabajo, con un servidor de mejores características que cuente con una tarjeta de red que permita utilizar el estándar 10 Gigabit Ethernet.

- Puede ser de interés realizar pruebas análogas a las realizadas en el proyecto utilizando USRP E3xx, los cuales cuentan con un sistema embebido que les permite comunicarse con el FPGA de forma directa, pero con una CPU que depende del USRP por lo que extender el procesamiento puede resultar ventajoso.

- Un punto de partida para un futuro proyecto puede consistir en explorar los alcances de la arquitectura y verificar si es posible expandir su utilidad a nuevas áreas de aplicación, de forma tal que un USRP pueda cumplir la función de dispositivos de propósito especifico.

BIBLIOGRAFÍA

- ARM. AMBA Protocol. En: arm Developer [En línea]. (2010) [Consultado 10 Dic. 2017]. Disponible en: https://developer.arm.com/products/architecture/ amba - protocol

- ARM. AMBA 4 AXI4-Stream Protocol. En: arm Developer [En línea]. (2010) [Consultado 10 Dic. 2017]. Disponible en: https://developer.arm.com/docs/ihi0051 / latest /amba-axi4-stream-protocol-specification-v10

- BAAS, Bevan M. A low-power, high-performance, 1024-point FFT processor. IEEE Journal of Solid-State Circuits, 1999, vol. 34, no 3, p. 380-387.

- BRAUN, Martin; PENDLUM, Jonathon; ETTUS, Matt. RFNoC: RF network-on-chip. Proceedings of the GNU Radio Conference. 2016.

- CHRISTENSEN, Ken, et al. IEEE 802.3 az: the road to energy efficient ethernet. IEEE Communications Magazine, 2010, vol. 48, no 11.

- COOLEY, James; TUKEY, John. An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation, 1965. p 297-301.

- DHAR, Rahul; GEORGE, Gesly; MALANI, Amit; STEENKISTE, Peter. Supporting integrated MAC and PHY software development for the USRP SDR. Networking Technologies for Software Defined Radio Networks, 2006. SDR'06.1 st IEEE Workshop on. p 68-77.

- ERGÜN, Funda. Testing multivariate linear functions: Overcoming the generator bottleneck. In Proc. Twenty-Seventh Ann. ACM Symp. 1995, Junio. p. 407–416

- ETHERNET ALLIANCE. Ethernet Jumbo Frames. [En línea]. (2009) [Consultado

23 Abr. 2018]. Disponible en: http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf

- ETTUS KNOWLEDGE BASE CONTRIBUTORS. About USRP Bandwidths and Sampling Rates. En: Ettus Knowledge Base [En línea]. (2016) [Consultado 25 Abr. 2018]. Disponible en: https://kb.ettus.com/index.php?title=About_USRP_Bandwidths_and_Sampling_Rates&oldid=2544

- ETTUS KNOWLEDGE BASE CONTRIBUTORS. GPSDO Selection Guide. En: Ettus Knowledge Base [En línea]. (2017) [Consultado 10 Dic. 2017]. Disponible en: https://kb.ettus.com/GPSDO_Selection_Guide

- ETTUS KNOWLEDGE BASE CONTRIBUTORS. RFNoC. En: Ettus Knowledge Base [En línea]. (2017) [Consultado 23 Abr. 2018]. Disponible en: https://www.ettus.com/content/files/RFNoC_Wireless_at_VT_FPGA.pdf

- FRIGO, Matteo; JOHNSON, Steven. FFT Benchmark Results. [En línea]. (2017) [Consultado 23 Abr. 2018]. Disponible en: http://www.fftw.org/speed/

- FRIGO, Matteo; JOHNSON, Steven. The design and implementation of FFTW3. Proceedings of the IEEE, 2005, vol. 93, no 2, p. 216-231.

- HICKLING, Ronald. New technology facilitates true software-defined radio. RF Design Magazine. 2005.

- HSW, Hwei; MEHRA, Raj; VELASCO, Federico. Análisis de Fourier (1987).

- MONTERO, Juan. Implementación de un sistema de comunicaciones basado en Software Radio. Universidad Autónoma de Madrid. 2014.

- MARWANTO, Arief; SARIJARI, Mohd; FISAL, Norsheila. Experimental study of OFDM implementation utilizing GNU Radio and USRP-SDR. Communications (MICC), 2009 IEEE 9th Malaysia International Conference on. p 132-135.

- PENDLUM, Jonathon. Rfnoc deep dive: Fpga side [En línea]. (2014) [Consultado 10 Dic. 2017]. Disponible en: https://www.ettus.com/content/files/RFNoC_Wireless_at_VT_FPGA.pdf

- REYMUND, Thomas. Software Defined Radio with Graphical User Interface. Universidad Tecnológica de Viena. 2007

- SUNDERLIN, Josh. Measured Latency Introduced by RFNoC Architecture. [En línea]. (2017) [Consultado 23 Abr. 2018]. Disponible en: https://www. gnuradio.org /wp-content/ uploads/ 2017 /12/Jarriel- Cook -Measured- Latency -Introduced-by-RFNoC-Architecture.pdf

- TEXAS INSTRUMENTS. FFT Implementation on the TMS320VC5505 , TMS320C5505, and TMS320C5515 DSPs [En línea]. (2013) [Consultado 20 Abr. 2018]. Disponible en: http://www.ti.com/litv/pdf/sprabb6b

- TRUONG, Nguyen; SUH, Young-Joo; YU, Chansu. Latency analysis in GNU radio/USRP-based software radio platforms. Military Communications Conference, MILCOM 2013-2013 IEEE. p 305-310.

- UZUN, Isa; AMIRA, Abbes. Towards a general framework for an FPGA-based FFT coprocessor. Seventh International Symposium on Signal Processing and Its Applications, 2003. Proceedings. p 617-620.

- UZUN, Isa Servan; AMIRA, Abbes; BOURIDANE, Ahmed. FPGA implementations of fast Fourier transforms for real-time signal and image processing. IEE Proceedings-Vision, Image and Signal Processing, 2005, vol. 152, no 3, p. 283-296.

- XILINX INC. AXI Reference Guide. En: Xilinx [En línea]. (2011) [Consultado 10 Dic. 2017]. Disponible en: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

- XILINX INC. Fast Fourier Transform v9.0 LogiCORE IP Product Guide. En: Xilinx [En línea]. (2015) [Consultado 23 Abr. 2018]. Disponible en: http://www.ti.com/litv/pdf/sprabb6b

ANEXOS

A. Recomendaciones de Software

- **Sistema operativo:** Es necesario utilizar un sistema operativo basado en Linux debido a que la mayoría de comandos usados durante el proceso de construcción del bloque son propios del mismo. Ubuntu con versiones *"Long Term Support"* (LTS) se considera una buena alternativa, debido a que cuenta con un mayor soporte, además suele ser más estable y presentar menos errores que las versiones simples. Otro factor es la facilidad para descargar e instalar el entorno mediante un gestor conocido como *PyBOMBS* que trabaja de mejor manera en este sistema operativo.

- **Versión de Vivado Design Suite:** Se recomienda utilizar la versión 2015.4 debido a que esta ofrece por defecto compatibilidad con UHD en dispositivos USRP X3XX para versiones de UHD menores a la 3.11, a partir de esta versión es necesario utilizar Vivado Design Suite 2017.4 o superiores. Para poder utilizar otra versión se debe modificar el código del archivo bash localizado en */src/uhd-fpga/usrp3/top/x300/setupenv.sh/* sustituyendo el valor de la variable *"VIVADO_VER"* por el valor de la versión seleccionada.

- **Licencia de Vivado Design Suite:** En caso de seleccionar la versión 2015.4 de Vivado como sintetizador, se debe adquirir ya sea la licencia "Design Edition" o "System Edition" debido a que el proceso requiere de "High-Level Synthesis" (HLS) que solo está disponible con cualquiera de las dos licencias.

B. Recomendaciones de Hardware

- **Memoria RAM:** Es necesario tener un mínimo de 8 GB de memoria RAM en el computador donde se vaya a realizar la síntesis de la imagen del FPGA. Este requerimiento se debe a que Vivado utiliza muchos recursos en este proceso. Sin embargo, existe una alternativa y es incrementar el tamaño de la memoria virtual o espacio de intercambio (swap). En Linux esto se puede hacer a través del terminal modificando el valor de una variable llamada “swappiness” mediante el comando:

```
$ sudo sysctl -w vm.swappiness=valor
```

  Para conocer el valor actual de esta variable se debe ingresar el comando:

```
$ sudo cat/proc/sys/vm/swappiness
```

- **Ethernet:** Es obligatorio tener una tarjeta Ethernet 1 Gigabit o 10 Gigabit en el computador o no se reconocerá el USRP X310.

C. Establecimiento RFNoC

La forma más limpia de instalar el entorno RFNoC es mediante el uso del asistente *PyBOMBS* * utilizando en orden los comandos del siguiente Cuadro.

Comandos necesarios para instalar RFNoC

Comando	Descripcion
pybombs recipes add gr-recipes git+https://github.com/gnuradio/gr-recipes.git	Añade recetas adicionales de desarrollo y uso de *GNU-Radio*
pybombs recipes add ettus git+https://github.com/EttusResearch/ettus-pybombs.git	Añade recetas adicionales ubicados en los repositorios de Ettus Research
pybombs prefix init ∼/rfnoc -R rfnoc -a rfnoc	Crea el directorio de rfnoc dentro de la carpeta de usuario (∼), descarga el codigo fuente, las aplicaciones y las dependencias de los repositorios, y los enlaza.

PyBOMBS Puede ser instalado mediante los comandos del siguiente Cuadro.

Comandos necesarios para instalar PyBOMBS

Comando	Descripcion
sudo apt-get install git	Instala GIT
sudo apt-get install python-setuptools python-dev python-pip build-essential	Instala PIP y otras dependencias de Python
sudo pip install git+https://github.com/gnuradio/pybombs.git	Instala la última versión de PyBOMBS del repositorio de GIT.

Una carpeta llamada rfnoc es creada, una vez la instalación se realice correctamente. Adicionalmente, es necesario ejecutar un archivo .bash que se encarga de agregar las direcciones importantes de todas las aplicaciones; para esto, se sugiere

*Es un asistente útil para instalar paquetes, diseñado originalmente para usuarios de *GNU-Radio*

realizar lo siguiente [*]: se debe abrir el archivo .bashrc, por ejemplo mediante el comando *sudo gedit ∼/.bashrc* y después ingresar en una línea vacía el comando *source ∼/rfnoc/setup_env.sh*. No es necesario usar el editor gedit, aunque puede ser instalado mediante el comando:*sudo apt-get install gedit*.

Es posible agregar las direcciones sin modificar el archivo .bashrc, ingresando *source ∼/rfnoc/setup_env.sh* en cada terminal donde se vaya a utilizar alguna herramienta del entorno.

[*]Probado en Ubuntu 16.04 LTS.

D. Composición RFNoC

Una vez configurado e instalado el entorno, es útil conocer su composición para tener una idea más específica del alcance y una idea general de la utilidad de cada parte.

GNU-Radio Es un software que permite al usuario realizar el trabajo de dispositivos analógicos comunes en bloques digitales simples con una o varias entradas y salidas donde es posible modificar los parámetros de cada bloque. Aunque su principal propósito es obtener señales analógicas a través de ciertos dispositivos, también puede ser utilizando en un ambiente de simulación con entradas definidas por fuentes digitales desde el computador.

Ofrece la utilización de modulos *Out of tree*(*OOT*) los cuales son componentes del Software que no están propiamente en él y por tanto no modifican el código fuente, esto posibilita el compartir proyectos entre múltiples usuarios. La arquitectura RFNoC esta basada en estos bloques [**?**].

Su código fuente se encuentra en *∼/rfnoc/src/gnuradio/* y dentro de esta se resalta la carpeta *gr-utils*, que contiene utilidades para el trabajo con *GNU-Radio* como la lectura y escritura de muestras guardas en archivos con herramientas de software matemático como MATLAB u Octave.

gr-ettus Es un módulo *OOT* que permite utilizar bloques RFNoC dentro de la interfaz, de la misma forma que los bloques tradicionales pero sin modificar ni alterar directamente el código de *GNU-Radio*. La carpeta de gr-ettus puede encontrarse en la dirección *∼/rfnoc/src/gr-ettus/* y presenta las siguientes carpetas de importancia que contienen:

- **examples** Ejemplos de flujogramas de GRC que utilizan bloques RFNoC de gr-ettus.
- **grc** Archivos de configuración *XML* a nivel de *GNU-Radio* de los bloques RFNoC de gr-ettus.

UHD Como se mencionó, la herramienta RFNoC habilita el procesamiento de datos en el *FPGA*, sin embargo para que esto ocurra primero debe existir comunicación directamente entre el software y el dispositivo *USRP*. Por lo tanto para poder utilizar la herramienta es necesario instalar el controlador o *driver* el cual es conocido como *USRP Hardware Driver* (UHD).

Este *driver* proporciona el control necesario para transportar muestras de las señales del usuario hacia y desde el hardware *USRP*, así como controlar diversos parámetros (por ejemplo, frecuencia de muestreo, frecuencia central, ganancias, etc.) del radio. Una vez instalado *UHD* se puede acceder a la dirección *∼/rfnoc/src/uhd/*, en donde se resaltan las siguientes carpetas por contener:

- **tools** Herramientas autónomas, que permiten depurar el dispositivo *USRP*, como la interpretación de paquetes *CHDR*, y la carga de una imagen al *FPGA* a través del puerto JTAG.
- **host/utils** Código fuente de herramientas para la configuración y prueba de dispositivos *USRP*, tales como su configuración y detección por el sistema, pruebas RF y de latencia, entre otras.
- **host/examples** Código fuente de ejemplos de uso del *USRP*.
- **host/include/uhd/rfnoc/blocks** Archivos de configuración *XML* a nivel de *UHD* de los bloques RFNoC de gr-ettus.
- **host/include/uhd/rfnoc/** Archivos fuente de cabecera de los bloque RFNoC de gr-ettus.
- **host/lib/rfnoc** Código fuente de los bloques RFNoC de gr-ettus.

UHD-FPGA Es un repositorio que contiene el código fuente en lenguaje de diseño de hardware (HDL) de *FPGA* para *USRP*, además de las herramientas necesarias para la creación , o síntesis, del archivo llamado imagen que define el comportamiento del *FPGA* y su simulación. Estas fuentes corresponden a los controladores de RF, de periféricos, comunicación con el *host*, RFNoC, entre otros. El repositorio se puede encontrar en *∼/rfnoc/src/uhd-fpga/*, dentro de la cual se destacan por su contenido las siguientes carpetas:

- **usrp3** Fuentes y herramientas de los dispositivos USRP de tercera generación.
- **usrp3/lib/rfnoc** Fuentes HDL para bloques RFNoC, ademas de los archivos fuente HDL y de simulación de los bloques RFNoC de gr-ettus.
- **usrp3/lib/sim/rfnoc** Archivos fuente de cabecera para simulación de bloques RFNoC.
- **usrp3/tools/make** Archivos de configuración que especifican variables de entorno y parámetros de síntesis y simulación.
- **usrp3/tools/scripts** Ejecutables para la simulación, síntesis, implementación y generación de reportes de una imagen RFNoC.
- **usrp3/tools/top/x300** Archivos de configuración y de fuente de la imagen RFNoC, ademas de las carpetas de salida de la síntesis.

E. Herramientas RFNoC

Herramienta rfnocmodtool La herramienta rfnocmodtool de *UHD* permite crear plantillas de módulos y bloques RFNoC mediante diferentes comandos y opciones. La estructura general para el uso de la herramienta es:

```
$ rfnocmodtool <comandos> [opciones]
```

La lista de comandos posibles se puede ver en el siguiente Cuadro.

Lista de comandos rfnocmodtool

Comando	Alias	Descripcion
disable	dis	Desactiva un bloque existente
info	getinfo, inf	Obtiene información acerca del modulo
remove	rm, del	Elimina un bloque de un modulo
makexml	mx	Crea el archivo *Extensible Markup Language (XML)* para la unión del bloque con *GNU-Radio*
add	insert	Agrega un bloque a un módulo existente.
newmod	nm, create	Permite crear un nuevo módulo *OOT*

Para obtener las opciones de cada comando se debe ejecutar:

```
$ rfnocmodtool help <comandos>
```

Se puede crear un nuevo módulo *OOT* escribiendo en la terminal.

```
$ rfnocmodtool newmod [Nombre del modulo]
```

Aunque la herramienta permite al usuario crear módulos en cualquier dirección, es necesario por coherencia con el entorno crearlos dentro de la carpeta *∼/rfnoc/src/*. El módulo está contenido en una carpeta llamada *rfnoc-[Nombre del módulo]* que debe presentar los siguientes archivos y carpetas si no se presentaron errores durante su creación.

```
apps cmake CMakeLists.txt docs examples grc include lib MANIFEST.
    md python README.md rfnoc swig
```

Para la adición de bloques a un módulo *OOT* es necesario abrir la carpeta del módulo y después ejecutar:

```
$ rfnocmodtool add [Nombre del bloque]
```

Donde se deberán ingresar los siguientes parámetros:

- **NoC ID:** Es un número hexadecimal de 16 dígitos (64 bits) que sirve como identificación del bloque tanto en hardware como en software. Es importante que al momento de ingresar el NoC ID se escriban todos los 16 dígitos o de lo contrario no se podrá encontrar el bloque cuando se vaya a sintetizar. En caso de que no se proporcione un NoC ID se tomará un valor aleatorio.
- **Generación de controladores de bloque:** Ofrecen un control adicional en lenguaje C ++ que el usuario puede agregar en el nivel *UHD*. El uso de estos controladores depende de la complejidad del bloque y generalmente son necesarios únicamente en casos donde la configuración de registros requiera gran cantidad de cálculos, imposibles de realizar en el archivo *XML* proporcionado por *UHD* o agregado mediante el lenguaje de script Noc-Script. En caso de que el usuario no proporcione un valor, las plantillas de los archivos C ++ se generan por defecto debido a que si no se utilizan simplemente no se cargan.
- **Interfaz de bloque:** Permite agregar más funciones específicas al diseño del bloque en el nivel *GNU-Radio*, imposibles de realizar en el archivo *XML* o agregado mediante el lenguaje Noc-Script. En caso de que el usuario no proporcione un valor, el archivo de la interfaz se generará por defecto debido a que si no se utiliza simplemente no se carga.

Una vez que un bloque haya sido añadido satisfactoriamente a un módulo, se crean todas las plantillas a nivel *FPGA*, *UHD* y *GNU-Radio* necesarias para la configuración y simulación del bloque RFNoC. Estas plantillas se encuentran en las direccio-

nes del siguiente Cuadro.

Localización plantillas RFNoC

Nivel de desarrollo	Dirección
Nivel *FPGA*	*rfnoc-[Nombre del módulo]/rfnoc/fpga-src/noc_block_[Nombre del bloque].v*
Simulación	*rfnoc-[Nombre del módulo]/rfnoc/testbenches/noc_block_[Nombre del bloque]_tb/noc_block_[Nombre del bloque]_tb.sv*
Nivel UHD	*rfnoc-[Nombre del módulo]/rfnoc/blocks/[Nombre del bloque].xml*
Nivel *GNU-Radio*	*rfnoc-[Nombre del módulo]/grc/[Nombre del módulo]_[Nombre del bloque].xml*

Herramienta uhd_image_builder Utilizar la herramienta puede ser complicado por la cantidad de opciones y parámetros posibles, por lo que no es recomendable utilizarla adicionando estos parámetros manualmente; sin embargo es posible obtener la lista y estructura del comando, ejecutando:

```
$ ~/rfnoc/src/uhd-fpga/usrp3/tools/scripts/uhd_image_builder.py --help
```

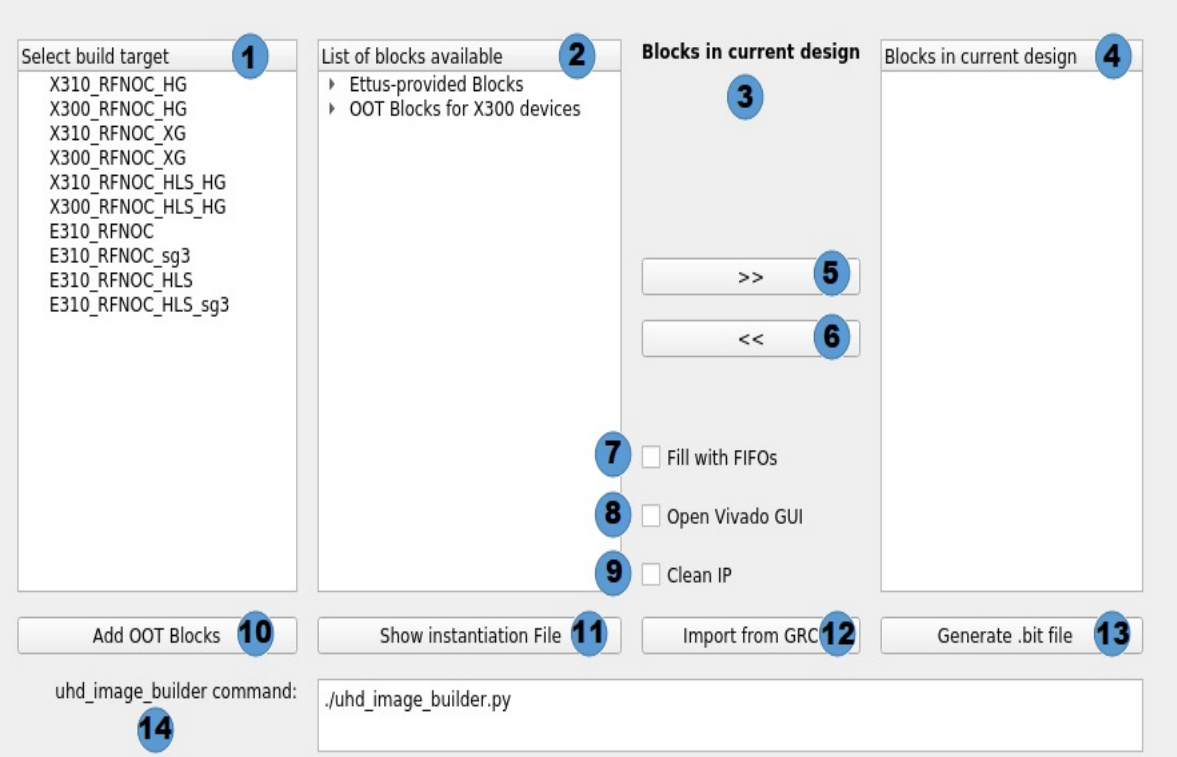
Una forma interactiva de obtener los parámetros de configuración, es a través de una interfaz a la que se puede acceder en la dirección */rfnoc/src/uhd-fpga/usrp3/tools/scripts/* mediante el comando:

```
$ sudo python3 uhd_image_builder_gui
```

En la siguiente Figura se puede ver la interfaz con etiquetas en los elementos importantes.

- **1. Select build target:** En este panel se listan todas las opciones de dispositivos disponibles. La lista puede variar dependiendo del tipo de repositorio del *FPGA* que el usuario está utilizando. El significado de cada parámetro se puede ver a continuación en el siguiente Cuadro:

Esquema general desarrollo *FPGA*



Lista de opciones para cada USRP

Parámetro	Significado
HG	1GigE en SFP+ Port0, 10Gig en SFP+ Port1
XG	10GigE en ambos puertos SFP+
HLS	La opción High Level Synthesis de Vivado está activa
sgX	Grado de velocidad (sólo para USRP E300)

- **2. List of blocks available:** En este panel se listan los bloques disponibles que pueden ser incluidos dentro de un determinado diseño. Esta lista separa los bloques RFNoC creados por Ettus Research y los módulos OOT añadidos por el usuario. El título de la lista cambia dinámicamente cada vez que se seleccione

un dispositivo diferente, lo cual implica que es necesario añadir módulos OOT para cada dispositivo de manera independiente. Lo anterior se explica mejor en el numeral 10.

- **3. Blocks in current design:** Esta sección brinda información acerca del número máximo de bloques para determinado USRP y la cantidad de estos bloques que ya han sido añadidos.
- **4. Blocks in current design:** Este panel es ocupado añadiendo elementos de la lista de bloques disponibles. Todos los bloques incluidos en este panel serán compilados en la imagen del *FPGA*
- **5. Add button (>>):** Manualmente añade el bloque desde el panel central al diseño.
- **6. Remove button (<<):** Elimina bloques del diseño actual.
- **7. Fill with FIFOs:** Activando esta opción, se rellenará cada espacio disponible no especificado con bloques FIFO. El número de bloques FIFO que serán agregados depende de la cantidad máxima de bloques posibles explicado en el numeral 3.
- **8. Open Vivado GUI:** Activar esta opción abrirá la interfaz de Vivado durante la construcción de la imagen del *FPGA*, permitiendo al usuario guardar un proyecto de Vivado completo desde la interfaz para su desarrollo.
- **9. Clean IP:** Limpia el *IP-CORE* antes de una nueva construcción (recompilando todos los IP).
- **10. Add OOT blocks:** Manualmente añade los modulos *OOT* generados por la herramienta rfnocmodtool apuntando al archivo Makefile.srcs localizado en el directorio *∼/rfnoc/rfnoc-[Nombre del módulo]/fpga-src/*. Para esto, se debe seleccionar la carpeta del módulo; y en consecuencia, el bloque aparecerá dentro del panel explicado en el numeral 2.
- **11. Show Instantiation File:** Mediante el uso de esta interfaz, se auto genera el archivo utilizado por Vivado para construir la imagen del *FPGA*. Haciendo clic en este botón, es posible editar este archivo.
- **12. Import from GRC:** Si el usuario tiene un flujograma de *GNU-Radio* con bloques RFNoC, esta aplicación puede leer esos bloques y llenar los espacios en el diseño actual con estos.

- **13. Generate .bit file:** Inicia el proceso de construcción de imagen.
- **14. uhd_image_builder command:** Esta linea de código es dinámicamente construida mientras el usuario selecciona las diferentes opciones dentro de la interfaz. Es recomendable guardar esta línea de código para usar la próxima vez que se vaya a construir la imagen del *FPGA* y evitar seleccionar todas las opciones nuevamente.

Herramienta uhd_image_loader Tiene dos únicos parámetros que deben ser agregados al código de la herramienta escribiendo el argumento antecedido de "–".

- **args:** Brinda información del *USRP*, tanto la serie del dispositivo, como su dirección IP.
- **fpga-path:** Se debe establecer la ruta donde la imagen compilada ha sido guardada.

El comando completo sería el siguiente:

```
$ uhd_image_loader --args "type=[tipo de USRP], addr= [Direccion IP]" --
    fpga-path [direccion de la imagen]
```

Una vez una imagen ha sido cargada satisfactoriamente en el *USRP*, este se debe reiniciar. Para comprobar la lista de bloques con la que el *USRP* ha sido cargado, se puede usar el comando:

```
$ uhd_usrp_probe
```

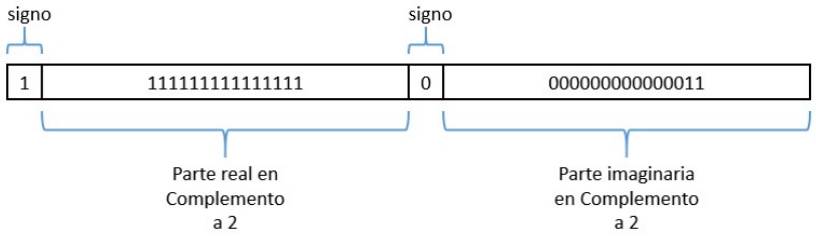
F. Estructura de los datos

La arquitectura RFNoC utiliza un formato de datos denominado Q1.15, donde el primer número representa el bit de signo, y el segundo la longitud en bits de cada muestra. El rango de valores que pueden tomar las muestras con este formato es $[-1, 1 - 2^{-15}]$ con una resolución de 2^{-15} y son representadas en complemento a dos.

Dentro del *FPGA*, las muestras viajan por medio de la señal *TDATA*, que es un bus de 32 bits distribuido de la siguiente forma: los bits más significativos (direcciones 31-16) corresponden con la parte real de la muestra , y los menos significativos (direcciones 15-0) con la parte imaginaria.

En la siguiente Figura se puede ver un ejemplo de una muestra con valor $-1 + 3j$ en el *FPGA*, donde j representa la componente imaginaria.

Ejemplo valor de muestra en *FPGA*



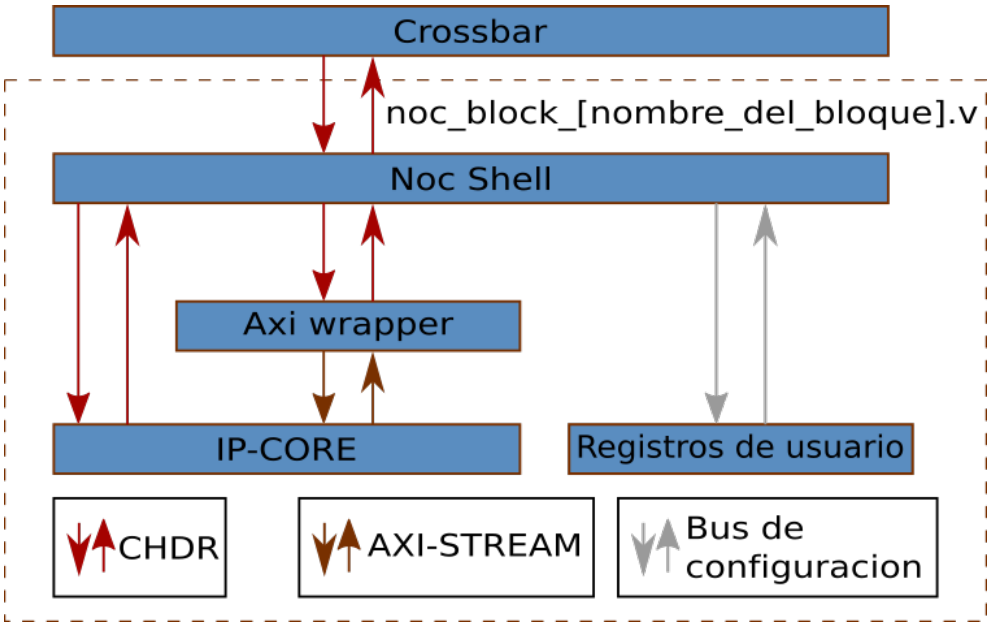
Sin embargo, este valor no representa el valor en flotante de la muestra en *GNU-Radio*. Para calcularlo se debe multiplicar el valor de la muestra por la resolución. En este caso, el valor de la muestra en flotante en *GNU-Radio* sería $-0{,}00003051757 + 0{,}00009155273j$

G. Nivel de desarrollo *FPGA*

En este anexo se explica lo referente al bloque RFNoC a nivel de *FPGA*, es decir a la plantilla de verilog, el *IP-CORE* o bloque funcional y los archivos de simulación. Se puede acceder a estos archivos a través de la dirección proporcionada en el anexo F donde se explica el uso de la herramienta rfnocmodtool en el espacio *Nivel FPGA*.

Configuración de la plantilla de verilog Esta plantilla se puede subdividir en tres partes importantes que el usuario debe modificar según los requerimientos de la aplicación del bloque. En la siguiente Figura se muestra un esquema general de este nivel donde se pueden apreciar estas tres partes.

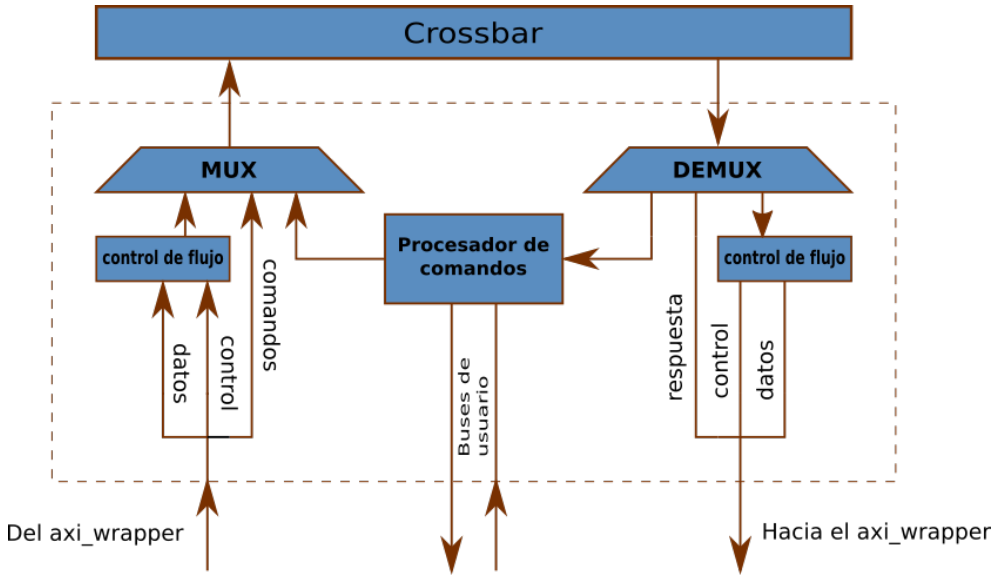
Esquema general desarrollo *FPGA*



Noc Shell Es una etapa de flujo bidireccional a la que se someten los datos enviados desde el computador o los datos ya procesados dentro del *FPGA*. En la siguiente Figura se ve un esquema general y su código puede ser encontrado en la dirección:
∼/rfnoc/src/uhd-fpga/usrp3/lib/rfnoc. Tiene tres funciones principales: multiplexación/-demultiplexación, control de flujo de datos y configuración de las señales de registro

que se explican a continuación.

Esquema general *Noc Shell*

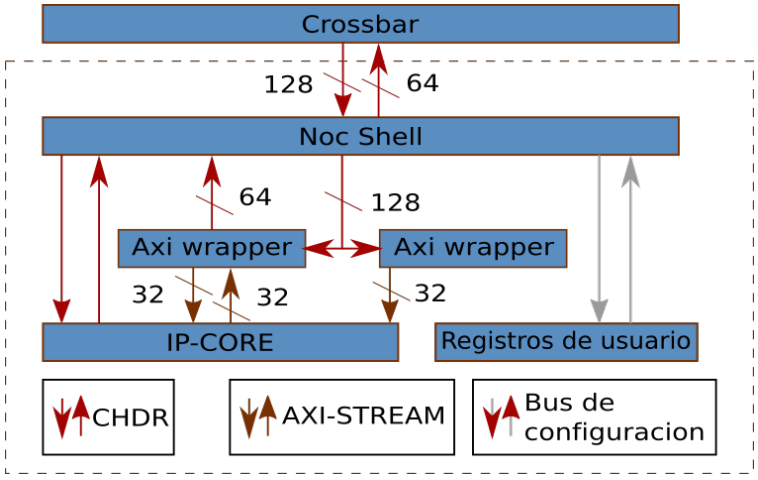


- **Multiplexación/demultiplexación:** Debido a que los datos se envían y reciben con el protocolo *CHDR*, (ver sección 1.3) es necesario separar los datos según el tipo de paquete, el comando especificado, la longitud, etc. La ventaja de utilizar este protocolo es que fácilmente se puede interpretar los paquetes de datos sin necesidad de complejas máquinas de estado. El cabecero *CHDR* puede ser automático o proporcionado por el usuario.
- **Control de flujo de datos:** Dentro del código del cabecero *CHDR* existen dos bits para determinar el tipo de paquete enviado, uno de esos tipos es el control de flujo de datos que controla la transmisión de datos. Estas señales de control se conectan con las señales de control del protocolo *AXI-STREAM* que simplifica el procesamiento en el *FPGA*. Esta es la razón por la cual los *USRP* de tercera generación utilizan este protocolo.
- **Configuración de las señales de registro:** Estas señales se encargan de establecer la interacción entre los valores guardados y los valores leídos en las direcciones establecidas por el usuario. Tienen comunicación directa con el código del usuario. En el apartado de *axi wrapper* se explica más sobre esto.

Número de puertos de entrada y salida Es posible utilizar más de un bus de datos en la entrada y en la salida. Para hacerlo, se debe modificar el valor de los parámetros *INPUT_PORTS* y *OUTPUT_PORTS* agregándolos a la definición del bloque; lo cual, en lugar de agregar nuevas señales, modifica el tamaño de los buses de entrada y salida a razón de 64 bits por cada puerto extra. Esto significa que se requiere instanciar un bloque *axi wrapper* adicional por cada par de puertos extra, y una concatenación o separación de las salidas o entradas de cada *axi wrapper* por cada puerto de entrada o salida adicional.

En la siguiente Figura se puede ver un ejemplo con dos puertos de entrada y un puerto de salida. El bus que comunica al *crossbar* con *noc shell* establece los dos canales de 64 bits de manera serial. *Noc shell* demultiplexa este bus convirtiendolo en uno de 128 bits; despues, se separa en grupos de 64 bits que se conectan a un *axi wrapper*. Debido a que sólo se presenta una salida, sólo es necesario el uso de un *axi wrapper* para enviar los datos de vuelta, sin embargo en la plantilla se debe definir la salida del otro *axi wrapper* aunque no se conecte.

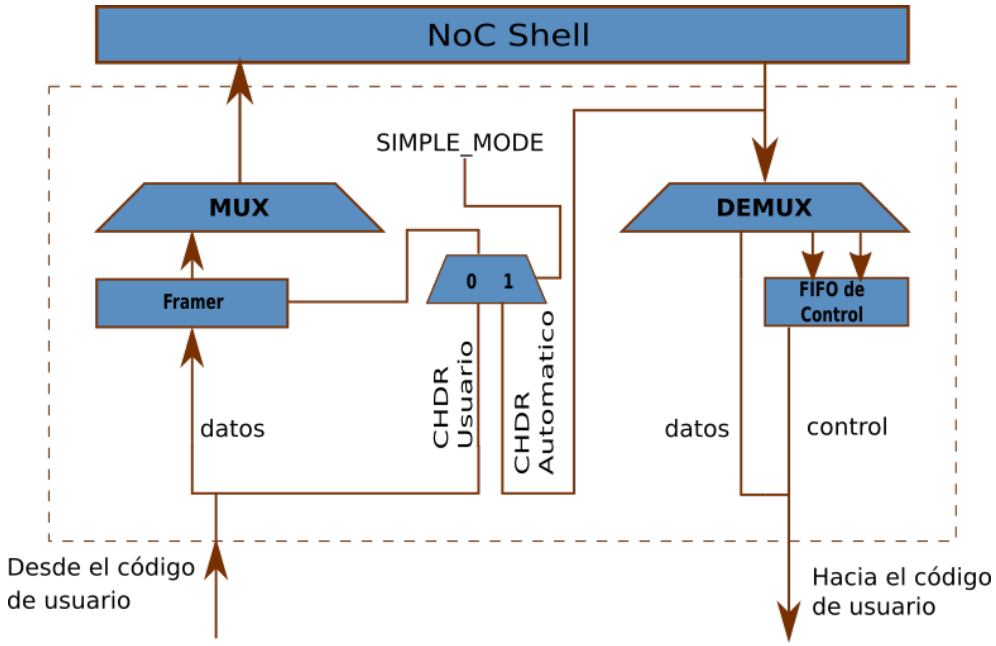
Ejemplo 2 puertos de entrada 1 puerto de salida



Axi Wrapper En esta etapa, todas las señales del protocolo *AXI-STREAM* son agregadas (sección 1.2). El cabecero *CHDR* es interpretado, estableciendo las caracteristicas de la señal de salida denominada *TDATA*. En la siguiente Figura se ve

un esquemático de esta etapa.

Esquema general *axi wrapper*



El código de *axi wrapper* puede ser encontrado en la dirección: *~/rfnoc/src/uhd-fpga/usrp3/lib/rfnoc*

Buses de configuración El usuario tiene derecho a utilizar 128 registros de 32 bits cada uno desde la dirección 128 a 256 para la interfaz *AXI-STREAM* o almacenar datos. A diferencia de otras señales, la señal de configuración no se puede asignar directamente como entrada en el *axi wrapper*. Para asignar registros al bus de configuración se deben modificar los parámetros *SR_AXI_CONFIG_BASE* y *NUM_AXI_CONFIG_BUS*. El primer parámetro establece desde donde se empiezan a tomar las direcciones de registro para el bus de configuración; por otro lado, el segundo parámetro establece la cantidad de registros que conforman el bus de configuración; es decir, se reservan los registros con direcciones desde *SR_AXI_CONFIG_BASE* hasta *SR_AXI_CONFIG_BASE + 2*NUM_AXI_CONFIG_BUS - 1* siendo posible tomar tantos registros como sea necesario dentro de los 128 posibles, los valores por defecto de estos parámetros son 129 y 1 respectivamente.

Es importante tener en cuenta al momento de asignar una señal, que la longitud total del bus de configuración será la suma de las longitudes individuales de cada registro, es decir *32*NUM_AXI_CONFIG_BUS*.

Para modificar estos parámetros se debe agregar el nombre del parámetro junto con el valor en la definición de *axi wrapper* en la plantilla. Por defecto el único parámetro definido es *SIMPLE MODE*.

Parámetro SIMPLE MODE Permite al usuario utilizar su propio cabecero *CHDR*. Si la señal está en alto se utiliza un cabecero automático pero si la señal está en bajo se puede ingresar un cabecero propio. Para ingresar un cabecero propio, se debe hacer uso de la señal *TUSER*. La señal *TUSER* es una señal auxiliar del *axi wrapper* con una longitud de 128 bits donde los 64 bits menos significativos deben ser usados para *timestamp* obligatoriamente y los 64 restantes para el cabecero; esto quiere decir que es posible modificar la longitud de los paquetes de salida estableciendo su longitud entre las direcciones 96 y 111 de la señal *TUSER*.

Longitudes de paquetes Las longitudes de los paquetes de entrada y de salida están establecidas por los valores proporcionados en el código *CHDR* entre las direcciones 32 a 47. Sin embargo, es posible modificarlas poniendo en alto los parámetros *RESIZE_INPUT_PACKET* y *RESIZE_OUTPUT_PACKET* para los paquetes de entrada y salida respectivamente. Se debe tener cuidado al modificar estas longitudes, debido a que se pueden producir algunos errores si ambas no son iguales.

Código de usuario Hace referencia a lo que el usuario debe realizar que no está por defecto en la plantilla y se debe instanciar o agregar, para que el bloque pueda tener alguna utilidad. Para construir correctamente un bloque es necesario tener conocimientos en descripción de hardware y un manejo correcto del lenguaje verilog; sin embargo, es posible diseñar *IP-CORE* en cualquier otro lenguaje que permita Vivado y luego instanciar estos. Para poder incluir bloques dentro de la plantilla es necesario modificar el archivo *Makefile.srcs* localizado en la carpeta de la plantilla

debido a que este debe contener las direcciones de todas las fuentes a nivel *FPGA*, incluyendo la plantilla misma. Para poder utilizar un bloque que no tenga el protocolo *AXI-STREAM* integrado existe la posibilidad de incluir bloques *FIFO* a la entrada y la salida para sincronizarlo.

Es importante mencionar que aunque es posible construir los *IP-CORES* en otro lenguaje, la plantilla creada automáticamente por la herramienta de *UHD* está escrita en verilog haciendo de su uso obligatorio.

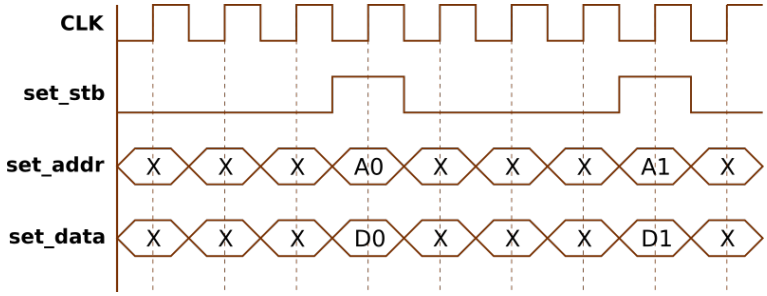
Configuración de registros de usuario Estos registros permiten almacenar valores ingresados o definidos por el usuario y modifican de alguna manera el comportamiento del *IP-CORE* o bloque funcional; sin embargo, es necesario definirlos y configurarlos en la plantilla de verilog. Existen dos tipos diferentes de registros: los que solo sirven para almacenar el valor; y los que además de almacenar, también contienen señales del protocolo *AXI-STREAM*. El tipo de registro recomendado para cada aplicación puede variar siendo los que cuentan con el protocolo necesarios en aplicaciones donde el valor de los registros deba cambiar en sincronía con los datos. El nombre para poder instanciar cada uno es *setting_reg* y *axi_setting_reg* respectivamente.

En la siguiente Figura se puede observar el funcionamiento de las señales de los registros de usuario. Cuando la señal *set_stb* tiene un flanco de subida, los datos de entrada en *set_data* se guardan en la dirección *set_addr* hasta el siguiente flanco de subida.

Es importante aclarar que no es necesario definir los registros destinados a la configuración del bloque funcional o *IP-CORE* que estén incluidos en la señal de configuración del protocolo *AXI-STREAM*.

A continuación se muestra un ejemplo, donde se configura *registro_1*. Primero se establecen los parámetros dirección de registro y tamaño, con valores *SR_REGISTRO_1* y *32* respectivamente y después se asigna la salida a una señal creada previamente.

Señales registros de usuario



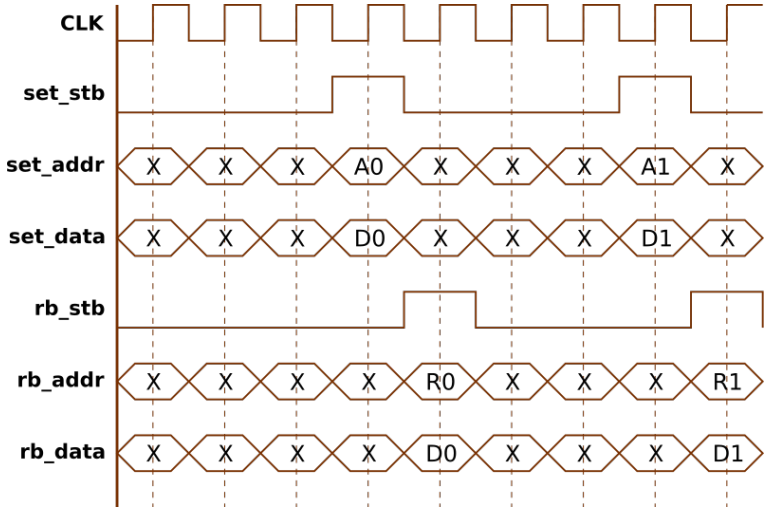

```
localparam [7:0] SR_REGISTRO_1 = SR_USER_REG_BASE;
wire [31:0] registro_1;
setting_reg #(
  .my_addr(SR_REGISTRO_1), .awidth(8), .width(32))
sr_registro_1 (
  .clk(ce_clk), .rst(ce_rst),
  .strobe(set_stb), .addr(set_addr), .in(set_data), .out(registro_1), .
      changed());
```

Configuración de registros de readback Además de almacenar datos, es posible enviar los datos de vuelta hacia el usuario mediante las señales *rb_data*, *rb_stb* y *rb_addr*. Para hacerlo, debe ocurrir un flanco de subida en la señal *rb_stb* por lo menos un ciclo de reloj después de guardar el valor de un registro, es decir después del flanco de subida de la señal *set_stb*. En ese caso el registro con dirección *rb_addr* será enviado a través de la señal *rb_data*. Lo anterior se puede ver mejor en la siguiente Figura.

No es necesario que el valor de *set_addr* y *rb_addr* sea el mismo para enviar de vuelta el valor de un registro guardado, la razón es que la dirección de *readback* depende de la opción asignada a dicho registro en un multiplexor y no a la dirección como tal del registro.

Lo anterior, se evidencia mejor a continuación.

Señales registros de usuario y readback

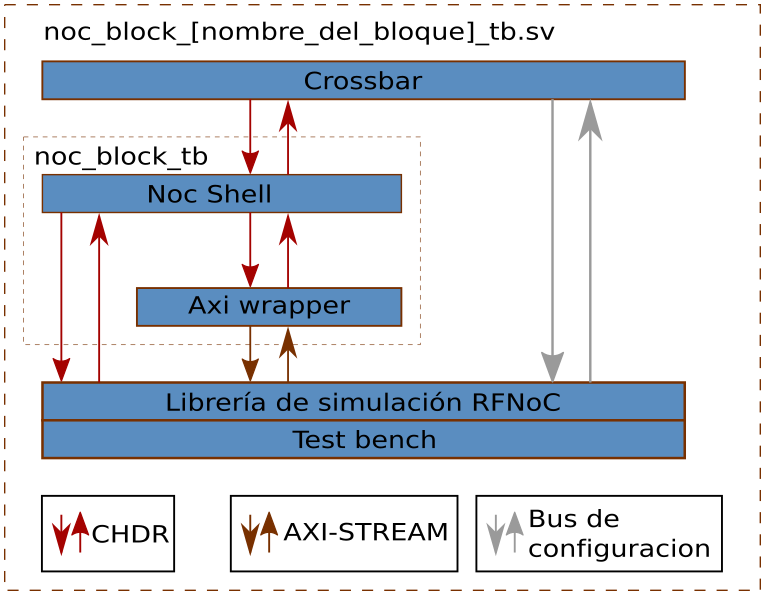



```
always @(posedge ce_clk) begin
  case(rb_addr)
    8'd0 : rb_data <= {32'd0, registro_1};
    8'd1 : rb_data <= {32'd0, registro_2};
    default : rb_data <= 64'h0BADC0DE0BADC0DE;
  endcase
end
```

Simulación Es importante crear bancos de pruebas para cualquier bloque RFNoC antes de realizar el proceso de compilación para una detección temprana de fallas o errores. A continuación, en la siguiente Figura se muestra la arquitectura general de banco de pruebas creada por la herramienta rfnocmodtool. Esta arquitectura permite a los usuarios probar bloques personalizados como si estuvieran integrados en la arquitectura RFNoC misma.

Cuando se realiza una prueba se ejecutan cinco casos distintos, donde cada uno tiene una función importante. Los casos 1-3 se introducen por defecto dentro de la plantilla y no requieren modificación, mientras que los casos 4 y 5 necesitan de código adicional para ser establecidos.

Arquitectura del banco de pruebas



- **Caso 1:** Verifica el correcto reinicio del bloque NoC.
- **Caso 2:** Verifica que el NoC ID ingresado sea el correcto.
- **Caso 3:** Verifica la conección del banco de pruebas con la plantilla de verilog para recibir y enviar datos.
- **Caso 4:** Verifica que los registros de lectura y escritura puedan almacenar o enviar datos respectivamente.
- **Caso 5:** Permite escrbir valores dentro de los registros de escritura e interactuar con lo descrito en la plantilla de verilog según lo requiera el usuario.

Guardar datos en registros: Esto se realiza de la siguiente forma: primero se debe definir el valor a guardar, por ejemplo:

```
registro_1 = 5;
```

Después, usando el objeto *tb_streamer* accediendo a la función *write_user_reg*, se debe asignar la dirección y el valor a guardar, en este caso:

```
tb_streamer.write_user_reg(sid_noc_block_[nombre del bloque], noc_block_[
    nombre del bloque].SR_REGISTRO_1, registro_1);
```

De esta forma se escribe el valor de *registro_1* en la dirección *SR_REGISTRO_1*

Enviar muestras: Para realizar esto, es importante conocer la estructura de los datos (ver Anexo F).

Primero, se crea un objeto *send_payload* de la clase *cvita_payload_t*; después se hace un llamado a la función *push_back* del objeto, y se asigna el valor de la muestra; finalmente, mediante el objeto *tb_streamer* y la función *send* se envía la muestra. Es importante mencionar que esta función envía datos en grupos de 64 bits, que corresponde con dos muestras.

En el siguiente ejemplo se envían dos muestras complejas con valor idéntico, 1 + j.

```
cvita_payload_t send_payload;
send_payload.push_back({15'd0,1'd1, 15'd0, 1'd1,15'd0,1'd1, 15'd0, 1'd1})
    ;
tb_streamer.send(send_payload);
```

Recibir muestras: Se deben crear dos objetos, *recv_payload* de la clase *cvita_payload_t* y *md* de la clase *cvita_metadata_t* y luego usar la función *recv* del objeto *tb_streamer*. La muestras recibidas se almacenan en *recv_payload*.

En el siguiente ejemplo se verifica si el valor obtenido corresponde con el valor esperado, suponiendo que el valor esperado es 1 + j,

```
logic [63:0] expected_value;
cvita_payload_t recv_payload;
cvita_metadata_t md;
tb_streamer.recv(recv_payload,md);
expected_value = {15'd0,1'd1, 15'd0, 1'd1,15'd0,1'd1, 15'd0, 1'd1};
$display("El valor esperado es: %b", expected_value);
$display("El valor obtenido es: %b", recv_payload[1]);
```

```
‘ASSERT_ERROR(recv_payload[1] == expected_value, s);
```

Ejecución de la simulación: Para ejecutar el banco de pruebas, primero se debe generar los ficheros de compilación para el entorno. Esto se realiza ingresando a la carpeta del modulo y escribiendo el comando:

```
$ mkdir build && cd build && cmake ../
```

El segundo paso es modificar los archivos necesarios y establecer la dirección correcta de las herramientas de simulación mediante el comando:

```
$ make test_tb
```

Este paso sólo se debe realizar una vez por cada módulo *OOT*, no es necesario realizarlo cada vez que un bloque sea agregado al módulo. Finalmente se ejecuta el banco de pruebas mediante el comando:

```
$ make noc_block_[nombre del bloque]_tb
```

H. Nivel de desarrollo UHD

En el nivel *UHD* se presenta todo lo necesario para que el software interprete las órdenes de creación de registros, asignación de argumentos, definición de variables, tipos de entrada y salida, y las envié al *USRP* y en consecuencia al *FPGA*. Es importante que el desarrollo del nivel *UHD* se realice una vez se haya diseñado, descrito, simulado y depurado el bloque a nivel de *FPGA*, debido a que cualquier modificación en ese nivel puede generar un cambio importante en este. Se puede acceder a estos archivos a través de la dirección proporcionada en el anexo F donde se explica el uso de la herramienta rfnocmodtool en el espacio *Nivel UHD*.

Este nivel está diseñado para entender toda la información de la declaración del bloque en un archivo con lenguaje de etiquetas *XML*, aunque a menudo es necesario integrar una lógica simple adicional con el lenguaje de *scripting* llamado *Noc-Script* o incluir controladores de bloque en lenguaje C++. A diferencia del bloque a nivel *FPGA*, no es necesario sintetizar o compilar el archivo *XML* por lo que ningún software adicional es requerido. Una vez modificado el archivo XML, los cambios realizados se importan automáticamente a *UHD* cada vez que se ejecute una aplicación.

Configuración del archivo XML El lenguaje de etiquetas *XML*, define un conjunto de reglas para codificar documentos en un formato enfatizado en la simplicidad y la generalidad. No es necesario conocer a profundidad el lenguaje para ser capaz de modificar la plantilla y configurar todo lo necesario en este nivel; solo es importante conocer el significado de cada etiqueta y su función dentro del bloque.

Estructura del lenguaje La estructura del lenguaje es muy simple, cada etiqueta debe tener una apertura y un cierre, y dentro de la etiqueta pueden existir atributos o más etiquetas. Los atributos son una manera de incorporar características o propiedades a las etiquetas y constan de dos partes: la propiedad del elemento y el valor de la propiedad, que siempre va entre comillas dobles o simples. Por ejemplo en la siguiente línea tomada directamente de la plantilla *<id revision="0">121212121233A</id>* la etiqueta *id* posee un atributo llamado *revision* con valor de cero, adicionalmente la plantilla posee también un valor que corres-

ponde con el número hexadecimal. No se debe modificar esta línea debido a que contiene el *ID* del bloque y la variable *revision* verifica este valor cada vez que se ejecuta el código.

Definición de registros Definir registros se hace obligatorio si se configuró algún registro en la plantilla de verilog. Cada registro se define dentro de la etiqueta <*setreg*> o <*readback*> dependiendo del tipo, y el conjunto de todas estas etiquetas a su vez están contenidas dentro de otra denominada <*registers*>. Existen dos etiquetas importantes dentro de la definición de registros, <*name*> y <*address*>. El valor de la etiqueta <*name*> debe asignarse al nombre de algún registro definido en la plantilla a nivel de *FPGA*, y el valor de la etiqueta <*address*> a la dirección establecida de ese registro. No se debe olvidar definir los registros de configuración los cuales tienen que ser nombrados como: *AXIS_CONFIG_BUS* y *AXIS_CONFIG_BUS_TLAST* cuya dirección depende de los parámetros del *axi wrapper* (ver anexo G).

A continuación, un ejemplo con *REGISTRO_1* asignado a la dirección 130.

```
<registers>
  <setreg>
    <name>REGISTRO_1</name>
    <address>130</address>
  </setreg>
</registers>
```

Definición de argumentos El número de argumentos necesarios depende de la cantidad de parámetros seleccionados para el bloque en *GNU-Radio*. Los argumentos, son las variables dentro del archivo *XML*. Cada uno está definido dentro de una etiqueta llamada <*arg*> y el conjunto total de argumentos están contenidos en una etiqueta llamada <*args*>. Posee muchas etiquetas importantes obligatorias u opcionales tales como: <*name*> <*type*> <*value*> <*check*> <*check_message*> y <*action*>.

- **name:** de manera similar a los registros, esta etiqueta también es usada para asignarle un nombre a la etiqueta de nivel superior, en este caso a los argumentos. Es posible acceder al valor de un argumento escribiendo el nombre, precedido del signo “$”.
- **type**: esta etiqueta es utilizada para definir el tipo de variable del argumento como *int* (entero), *string* (cadena de caracteres) etc.
- **value:** establece el valor por defecto con el que se cargará el argumento de manera inicial. Este valor debe concordar con el tipo de variable del argumento.
- **check:** permite introducir una condición dentro del argumento. Requiere de código adicional en *Noc-Script*.
- **check message:** es un mensaje que se muestra cada vez que no se cumpla la condición de *check*.
- **action:** se ejecuta esta acción en caso de que la condición de check se haya cumplido. Requiere de código adicional en Noc-Script y generalmente se usa para asignar valores a los registros. Esto se realiza mediante la función *SR_WRITE(”param_1”, $param_2)*, donde *param_1* corresponde al nombre del registro, y *param_2* al valor entero que se le va a asignar al registro.

A continuación se define el argumento *registro_1* de tipo *double* con valor por defecto *1.0*. Se verifica si el valor del argumento está entre -1 y 1; si no, un mensaje es mostrado; si se cumple, el valor del argumento es guardado en *REGISTRO_1*.

```
<args>
   <arg>
      <name>registro_1</name>
      <type>double</type>
      <value>1.0</value>
      <check>GE(registro_1, −1,0)ANDLE(registro_1, 1.0)</check>
      <check_message>registro_1 debe estar entre [-1, 1]</check_message>
      <action>SR_WRITE(REGISTRO_1”, $registro_1)</action>
   </arg>
 </args>
```

Definición de los puertos Los puertos establecen el tipo, la forma y el tamaño de los datos enviados y recibidos por el *USRP*, estos se definen dentro de la etiqueta <*ports*> donde pueden existir puertos de entrada y puertos de salida. La etiqueta de los puertos de entrada es llamada <*sink*> y la de salida <*source*>. Por defecto, se asigna una sub etiqueta con la creación del bloque, <*name*>. Sin embargo es útil agregar la etiqueta <*type*> donde se asigna la cantidad de bits reales e imaginarios que componen una muestra con opciones como sc8, sc16, etc.

Es necesario también establecer la cantidad de muestras por paquete, de no hacerlo se toma por defecto una muestra por paquete. Para hacerlo, se debe agregar la sub etiqueta <*pkt_size*> y asignar el valor del paquete. Opcionalmente, si se requieren datos enviados a través de un vector, es posible establecer su longitud mediante la sub etiqueta <*vlen*>. Se debe agregar el mismo número de entradas y salidas establecido a nivel *FPGA*.

El siguiente es un ejemplo de dos puertos de entrada y uno de salida, de tipo *sc16* con un valor de muestra por paquete asignado a través del argumento *$spp*.

```
<ports>
  <sink>
    <name>entrada_1</name>
    <type>sc16</type>
    <pkt_size>$spp</pkt_size>
  </sink>
  <sink>
    <name>entrada_2</name>
    <type>sc16</type>
    <pkt_size>$spp</pkt_size>
  </sink>
  <source>
    <name>salida_1</name>
    <type>sc16</type>
    <pkt_size>$spp</pkt_size>
  </source>
</ports>
```

Lenguaje *Noc Script* *Noc Script* permite el uso de operaciones matemáticas simples dentro del archivo *XML*. En el siguiente Cuadro se muestra la estructura de cada función junto con su definición.

Lista de funciones Noc Script

Función	Descripción
$ADD(x, y)$	Retorna el resultado de $x + y$
$MULT(x, y)$	Retorna el resultado de $x \times y$
$DIV(x, y)$	Retorna el resultado de $x \div y$
$LE(x, y)$	Retorna verdadero si $x \leq y$
$GE(x, y)$	Retorna verdadero si $x \geq y$
$LT(x, y)$	Retorna verdadero si $x < y$
$GT(x, y)$	Retorna verdadero si $x > y$
$IROUND(x)$	Retorna el valor redondeado de x
$IS\_PWR\_OF\_2(x)$	Retorna verdadero si x es potencia de 2
$LOG2(x)$	Retorna redondeado hacia abajo el valor de $\log_2 x$
$MODULO(x, y)$	Retorna el resultado de x % y
$EQUAL(x, y)$	Retorna verdadero si $x = y$
$SHIFT\_RIGHT(x, y)$	Retorna x desplazado y posiciones a la derecha
$SHIFT\_LEFT(x, y)$	Retorna x desplazado y posiciones a la izquierda
$BITWISE\_AND(x, y)$	Retorna el resultado bit a bit de x & y
$BITWISE\_OR(x, y)$	Retorna el resultado bit a bit de $x \| y$
$BITWISE\_XOR(x, y)$	Retorna el resultado bit a bit de x *XOR* y
$XOR(x, y)$	Retorna el resultado de x *XOR* y
$NOT(x)$	Retorna el valor negado de x
$TRUE()$	Siempre retorna un valor verdadero
$FALSE()$	Siempre retorna un valor falso
$IF(x, y)$	Evalúa x, si es verdadero ejecuta y. Retorna verdadero si x es verdadero
$IF\_ELSE(x, y, z)$	Evalúa x, si es verdadero ejecuta y, si no ejecuta z

(Continuación)

Función	Descripción
$SLEEP(x)$	Espera por x segundos. Las fracciones son permitidas con precisión de milisegundos

I. Nivel de desarrollo *GNU-Radio*

El bloque RFNoC a nivel *GNU-Radio* se encarga de definir los parámetros del bloque dentro de la interfaz, a través de un archivo *XML* cuya dirección se puede ver en el anexo F donde se explica el uso de la herramienta rfnocmodtool en el espacio *Nivel GNU-Radio*. Aunque el formato y el lenguaje son los mismos que los del nivel de desarrollo UHD, el tipo de etiquetas usadas es diferente; además, se incluye una sección para asignar el valor de los argumentos establecidos a nivel *UHD*.

Argumentos Los argumentos deben ser asignados utilizando una función que se define dentro de la etiqueta $<$*make* $>$. Esta etiqueta es una porción de un código de *python*, por lo tanto es importante la sangría. La función se debe localizar en una linea única con la siguiente estructura: *self.$(id).set_arg("argu", $argu)* donde "argu" representa el nombre del argumento definido a nivel *UHD*. Adicionalmente, es necesario realizar un llamado de dicha función dentro del archivo *XML*, lo cual generalmente tiene lugar una linea después del cierre de la etiqueta $<$*make* $>$ o antes de la definición de los parametros. Esto se debe, a que el archivo *XML* se interpreta de manera secuencial; y por lo tanto, se hace necesario primero establecer los registro donde se va a almacenar el valor del parámetro, antes de poder asignar un espacio al parámetro dentro del bloque en *GNU-Radio*. Este llamado, se realiza mediante la siguiente linea de código: $<$*callback*$>$*set_arg("argu", $argu)*$</$*callback*$>$.

A continuación, se puede ver la asignación del argumento *registro_1* con el valor del parametro *parametro_1*

```
  <make>
  ...

          $block_index,
          $device_index
)
self.$(id).set_arg("registro_1",  $parametro_1)
  </make>
  <callback>set_arg("registro_1", $parametro_1)</callback>
```

Parámetros Los parámetros son opciones que el usuario modifica desde la interfaz de *GNU-Radio*, que alteran el funcionamiento del bloque dentro de un diagrama de flujo. Se definen dentro de la etiqueta $<param>$ y puede contener las sub etiquetas $<name>$, $<key>$, $<value>$,$<type>$y $<option>$.

- **Name:** En este espacio se asigna el titulo del parámetro dentro del bloque en *GNU-Radio*; es decir, la etiqueta que se ubica antes del cuadro de texto donde se establece el valor en la interfaz.
- **Key:** Es la variable donde se guarda el valor ingresado en el cuadro de texto del bloque en *GNU-Radio*.
- **Value:** Es el valor por defecto del cuadro de texto.
- **Type:** Indica el tipo de variable que se ingresa dentro del cuadro de texto.
- **Option:** Esta etiqueta es opcional y sirve para proporcionar una lista de posibles valores para un parametro.

En el siguiente ejemplo, se crea un cuadro de texto con nombre *Parametro 1* cuyo valor es almacenado en la variable *parametro_1* seleccionando entre las opciones *opcion_1* y *opcion_2*.

```
<param>
  <name>Parametro 1</name>
  <key>parametro_1</key>
  <type>enum</type>
  <option>
    <name>Opcion 1</name>
    <key>opcion_1</key>
  </option>
  <option>
    <name>Opcion 2</name>
    <key>opcion_2</key>
  </option>
</param>
```

Entradas y salidas del bloque Las etiquetas $<$*sink* $>$ y $<$*source* $>$ permiten identificar las entradas y salidas del bloque respectivamente. Incluyen una etiqueta $<$*domain* $>$ con valor *rfnoc* que no se debe modificar pues indica que el bloque pertenece a este dominio. En la etiqueta $<$*type* $>$ se debe seleccionar el tipo de datos que permitirá el bloque, por ejemplo: complejo, vector, real, etc. Se debe agregar el mismo número de entradas y salidas establecido en niveles inferiores y es necesario definir cada entrada o salida en una etiqueta distinta.

El siguiente ejemplo, presenta dos entradas *in_1* e *in_2* y una única salida *out* de tipo complejo.

```
  <sink>
    <name>in_1</name>
    <type>complex</type>
    <domain>rfnoc</domain>
  </sink>
  <sink>
    <name>in_2</name>
    <type>complex</type>
    <domain>rfnoc</domain>
  </sink>
  <source>
    <name>out</name>
    <type>complex</type>
    <domain>rfnoc</domain>
  </source>
</block>
```

J. Habilitación y uso del escenario RX-FPGA-TX

En este anexo se explica el procedimiento utilizado en este trabajo para la habilitación y uso de un escenario RX-FPGA-TX. Puesto que este escenario no es soportado por defecto por RFNoC, la modificación del código fuente de UHD y grettus es necesaria. El proceso se desarrolla de forma mas especifica en *, del cual se deriva esta guía.

1. **Deshabilitación de *Timestamp*:** Un *Timestamp* o marca de tiempo es agregado al paquete CHDR durante la recepción para que el servidor pueda sincronizar múltiples señales de entrada. El servidor también agrega un *Timestamp* durante la transmisión para definir un tiempo especifico de emisión. Esto es un problema al utilizar la configuracion RX-FPGA-TX, puesto que los datos siempre estarían en el pasado y el transmisor los rechazaría, teniendo en cuenta que el transmisor y receptor utilizan el mismo reloj.

   Primero se define el registro 158 del radio en UHD que permite el habilitar o deshabilitar el *timestamp* del bloque de Radio RFNoC, agregando la linea en negrilla al archivo *∼/rfnoc/src/uhd/host/lib/rfnoc/radio_ctrl_impl.hpp*:

```
...
        static const uint32_t RX_CTRL_CLEAR_CMDS = 157;
        static const uint32_t RX_CTRL_OUTPUT_FORMAT= 158;
        static const uint32_t MISC_OUTS = 160;
...
```

   Luego se escribe un cero en este registro para deshabilitar el *Timestamp* agregando la linea en negrilla a *∼/rfnoc/src/uhd/host/lib/rfnoc/radio_ctrl_impl.cpp*;

```
...
void radio_ctrl_impl::issue_stream_cmd(const uhd::stream_cmd_t &
    stream_cmd, const size_t chan)
{
        ...
     //issue the stream command
```

*Nick Foster, "Stupid RFNoC Tricks: Loopback", Disponible en línea: https://corvid.io/2017/04/22/stupid-rfnoc-tricks-loopback/

```
    const uint64_t ticks = (stream_cmd.stream_now)? 0 : stream_cmd.
        time_spec.to_ticks(get_rate());
    sr_write(regs::RX_CTRL_OUTPUT_FORMAT, boost::uint32_t(0), chan);
    sr_write(regs::RX_CTRL_CMD, cmd_word, chan);
    sr_write(regs::RX_CTRL_TIME_HI, uint32_t(ticks >> 32), chan);
    sr_write(regs::RX_CTRL_TIME_LO, uint32_t(ticks >> 0),  chan); //
        latches the command
}
...
```

Por ultimo se compila de nuevo UHD y se instala la librería ejecutando el siguiente comando en la carpeta *~/rfnoc/src/uhd/host/build*:

```
make install
```

2. **Habilitar el *Streamer*** Por defecto, RFNoC necesita que un flujo de datos entre o salga del servidor para que este controle el USRP y habilite el flujo de datos. En la configuración RX-FPGA-TX es necesario agregar manualmente este control de flujo. Para ello se agregan las siguientes definiciones en negrilla a *~/rfnoc/gr-ettus/include/ettus/rfnoc_radio.h* y a *~/rfnoc/gr-ettus/lib/rfnoc_radio_impl.h* sin el modificador *virtual*

```
        ...
        virtual void set_tx_antenna(const std::string &ant, const
            size_t chan=0) = 0;
        virtual void set_rx_antenna(const std::string &ant, const
            size_t chan=0) = 0;
        virtual void set_tx_streamer(bool active, const size_t port) = 0;
        virtual void set_rx_streamer(bool active, const size_t port) = 0;
        virtual void issue_stream_cmd(const uhd::stream_cmd_t &cmd, const size_t
            chan=0) = 0;
        virtual void set_tx_dc_offset(bool enable, const size_t chan
            =0) = 0;
        ...
```

```
        ...
        void set_tx_antenna(const std::string &ant, const size_t chan
            =0);
```

```
        void set_rx_antenna(const std::string &ant, const size_t chan
            =0);
        void set_tx_streamer(bool active, const size_t port);
        void set_rx_streamer(bool active, const size_t port);
        void issue_stream_cmd(const uhd::stream_cmd_t &cmd, const size_t chan=0);
        void set_tx_dc_offset(bool enable, const size_t chan=0);
        ...
```

Luego se implementan en *~/rfnoc/gr-ettus/lib/rfnoc_radio_impl.cc*:

```
        ...
    void rfnoc_radio_impl::set_rx_antenna(const std::string &ant,
        const size_t chan)
    {
      _radio_ctrl->set_rx_antenna(ant, chan);
    }

    void rfnoc_radio_impl::set_tx_streamer(bool active, const size_t port)
    {
    _radio_ctrl->set_tx_streamer(active, port);
    }

    void rfnoc_radio_impl::set_rx_streamer(bool active, const size_t port)
    {
    _radio_ctrl->set_rx_streamer(active, port);
    }

    void rfnoc_radio_impl::issue_stream_cmd(const uhd::stream_cmd_t &cmd, const
        size_t chan)
    {
    _radio_ctrl->issue_stream_cmd(cmd, chan);
    }

    // FIXME everything down from here needs to be mapped on to the
        block API
        ...
```

Por ultimo se compila de nuevo gr-ettus ejecutando el siguiente comando en la carpeta *~/rfnoc/src/gr-ettus/build*:

```
make install
```

3. **Utilizar comandos de control en flujograma** Una vez agregadas las interfaces para gr-ettus, se modifica el código de Python de un flujograma que tenga la configuración RX-FPGA-TX, agregando las siguientes lineas en negrilla después de la instaciación del bloque de alto nivel:

```
...
def main(top_block_cls=[Nombre del bloque de alto nivel], options=None
    ):
     ...
     tb = top_block_cls()
     tb.[ID bloque radio RX].set_rx_streamer(True, 0)
     stream_cmd =
         uhd.stream_cmd_t(uhd.stream_cmd_t.STREAM_MODE_START_CONTINUOUS)
     tb.[ID bloque radio RX].issue_stream_cmd(stream_cmd)
     tb.start()
          ...
```

Si después de ejecutar el flujograma se obtiene un error indicando la ausencia de alguno de los objetos anteriores, es necesario recompilar UHD y gr-ettus, eliminando la carpeta build y ejecutando los siguientes comandos en la carpeta raíz de cada uno:

```
mkdir build && cd build
cmake ../
make install
```

K. Errores encontrados

Durante el desarrollo del proyecto se presentaron errores en múltiples situaciones, a continuación se hace una lista del problema generado y la solución planteada.

- **Error:** El gestor de licencias de Vivado,no reconoce la dirección MAC del computador. **Solución:** Se debe cambiar el nombre de la interfaz ethernet a eth0. Para esto se debe editar el archivo */etc/udev/rules.d/10-networks.rules* añadiendo

```
SUBSYSTEM==”net”, ACTION==”add”, ATTR{address}==”Mac_address”, NAME=”
    eth0””
```

  Donde *Mac_address* es la direccion MAC de la tarjeta de ethernet.

- **Error:** Algunos archivos dentro de la carpeta base de rfnoc no contaban con los permisos necesarios para sintetizar una imagen. **Solución:** Es necesario dar permisos de lectura y escritura a todos los archivos dentro de la carpeta rfnoc.

```
$ sudo chmod -r 777 ~/rfnoc
```

- **Error:** Conflicto al instalar *GNU-Radio* del paquete de RFNoC. **Solución:** .Si ya se instalo una version de *GNU-Radio*, con el gestor de paquetes por ejemplo, es probable que existan conflictos entre ambas versiones, se recomienda desinstalar y limpiar cualquier versión previamente instalada.

- **Error:** No se ejecuta uhd_image_builder_gui.py. **Solución:** Esta herramienta requiere de una libreria de Qt solo disponible en python 3 o nuevas versiones, se recomienda ejecutar el siguiente comando.

```
$ python3 ./uhd_image_builder_gui.py
```

- **Error:** Un poco antes de finalizar el proceso de sintesis de una imagen, un error es retornado. **Solución:** Esto ocurre debido a que según el proceso de instalación de Vivado o RFNoC, se requieren permisos de administrador para la escritura de las imagenes.

- **Error:** Error al sintetizar usando dash por defecto en el terminal. **Solución:** En algunas distribuciones de linux como ubuntu, dash está definido por defecto, para cambiarlo a bash se debe.

```
$ sudo dpkg-reconfigure dash
$ ll /bin/sh
```

  Cuando el primer comando se ejecuta, se debe seleccionar No y después ejecutar el segundo comando.

- **Error:** No es posible abrir *GNU-Radio companion* pero antes funcionaba bien. **Solución:** Una actualización del sistema puede corromper paquetes importantes dentro del entorno como python, boost, cmake, entre otros. Se sugiere actualizar el entorno con pyboms, aunque esto sobreescribe los archivos editados por el usuario.

- **Error:** Error ”GET”no descriptivo en *GNU-Radio companion*. **Solución:** Una posible causa es el llamado de una variable dentro de el archivo XML antes de su definición. Los archivos XML se ejecutan de manera secuencial.

L. Observaciones adicionales

- Para configurar todas las direcciones de rfnoc se debe: abrir el archivo .bashrc, por ejemplo mediante el comando:

```
sudo gedit ~/.bashrc
```

  Y después agregar al final del archivo la linea:

```
source ~/rfnoc/setup_env.sh
```

  Es posible agregar las direcciones sin modificar el archivo .bashrc, ingresando el ultimo comando en cada terminal donde se vaya a utilizar alguna herramienta del entorno

- Es necesario configurar los *SOCKET BUFFER* y *MTU*. En el caso de los primeros se debe acceder al archivo */etc/sysctl.conf* y agregar lo siguiente:

```
net.core.rmem_max=33554432
net.core.wmem_max=33554432
```

  Despues se debe reiniciar el sistema o ingresar en una terminal:

```
$ sudo sysctl -w net.core.rmem_max=33554432
$ sudo sysctl -w net.core.wmem_max=33554432
```

  Para configurar el MTU se debe ingresar lo siguiente en la terminal

```
$ sudo ifconfig <interface> mtu 9000 # Para 10 Gigabit Ethernet
$ sudo ifconfig <interface> mtu 1500 # Para 1 Gigabit Ethernet
```

- En GNU-Radio companion existe un identificador de errores llamado *parser errors*, en el menú *Help*. Esta opción permite obtener los errores de configuración de los archivos XML de grc. Es necesario tener especial cuidado con la sangría dentro de la etiqueta <*make*> (ver Anexo I).

- Dentro del flujo de datos, sólo es posible el envío al USRP una vez, es decir, no se pueden añadir bloques de RFNoC a un flujograma de GNU-Radio companion de manera no secuencial o un error es generado.

- Algunas veces, cuando se han ejecutado distintos flujogramas de *GNU-Radio companion*, el *USRP* deja de funcionar registrando un error en la terminal, para solucionarlo se debe reiniciar el *USRP*.

- Cuando se desean utilizar módulos de Verilog (IP-CORE, archivos fuente, etc.) en la plantilla del nivel de FPGA de RFNoC, es necesario agregar las fuentes a la carpeta *fpga-src* del módulo y modificar el archivo *Makefileśrcs* de la siguiente forma

```
$(addprefix SOURCES_PATH, \
noc_block_[Nombre del bloque].v \
[Nombre del modulo 1 en Verilog].v \
[Nombre del modulo 2 en VHDL].vhdl \
[Nombre del IP-CORE].xci \
) \
```

También el archivo *Makefile* en la carpeta *testbenches/noc_block_[Nombre del bloque]_tb* para su simulación

```
...
DESIGN_SRCS += $(abspath \
../../fpga-src/[Nombre del modulo 1 en Verilog].v \
../../fpga-src/[Nombre del modulo 2 en VHDL].vhdl \
../../fpga-src/[Nombre del IP-CORE].xci \
)
...
```