

PRÁCTICA PROFESIONAL SUPERVISADA



TEMA:

Relevamiento, análisis y ejercicio de la arquitectura Ectus USRP B200

LUGAR DE REALIZACIÓN:

Laboratorio de Comunicaciones Digitales - FCEfYN - UNC.

COORDINADORA PS:

Ing. Rodriguez, Carmen.

SUPERVISOR:

Zerbini, Carlos Alberto

TUTOR:

Ing. Ayarde, Juan Martín.

INTEGRANTE:

Mayol, Adelina.

MATRICULA:

34.140.737

CARRERA:

Ingeniería en Computación.

ÍNDICE

1.INTRODUCCIÓN

2.DESARROLLO

2.1.Adaptación del entorno de software y herramientas de trabajo

2.1.1 Instalación del driver UHD

2.1.3 Instalación de Xilinx ISE

2.1.3 Instalación de librerías GnuRadio

2.2.Pruebas de comunicación con la placa Ettus B200

2.2.1 Prueba del camino de comunicación PC - plataforma ETTUS B200

2.2.2 Instalación y ensayo de una imagen de FPGA modificada

2.3.Exploración del área de comunicaciones inalámbricas en fpga

2.3.1.Relevamiento del camino de transmisión de los datos

2.3.1.1 Camino de transmisión de los datos

2.3.1.2 Protocolos utilizados

2.3.1.2.1 Protocolo AXI4-Stream

2.3.1.2.1 Protocolo RUN - STROBE

2.3.1.2.1 Protocolo CHDR

2.3.2.Simulación y análisis de módulos y protocolos.

2.3.2.1 descripción de protocolos

2.3.2.2 simulación del camino de transmisión

3.CONCLUSIÓN

4.BIBLIOGRAFÍA

1. INTRODUCCIÓN

El siguiente informe detalla las tareas y pasos realizados durante la Práctica Profesional Supervisada para relevar, analizar y ejercitar los bloques de la arquitectura que forman parte del camino de procesamiento en hardware reconfigurable de la placa UHD B200 del fabricante Ettus. Esta plataforma se utiliza en sistemas de Radio Definida por Software (Software Defined radio, SDR) para proyectos del Laboratorio. Los sistemas SDR son profusamente utilizados en la actualidad para desarrollar e implementar cadenas de transmisión y recepción en sistemas inalámbricos. Constan de tres etapas principales, como se observa en la Fig. 1:

- **Front-end de RF**, etapa analógica donde se realizan operaciones tales como amplificación, filtrado, y mezcla para posterior digitalización de señales en frecuencia intermedia (IF)
- **Procesamiento digital en FPGAs**, donde realiza una primera etapa de procesamiento de señales a alta velocidad, que consiste básicamente en extracción de canales (channelizer) a partir de las señales de IF capturadas, para el caso de recepción, y posterior envío hacia un procesador para su tratamiento mediante software
- **Procesamiento digital mediante software**, donde las señales digitalizadas son moduladas/demoduladas obteniendo la banda base, que es tratada adicionalmente en etapas tales como sincronismo en frecuencia y tiempo, codificación, etc..

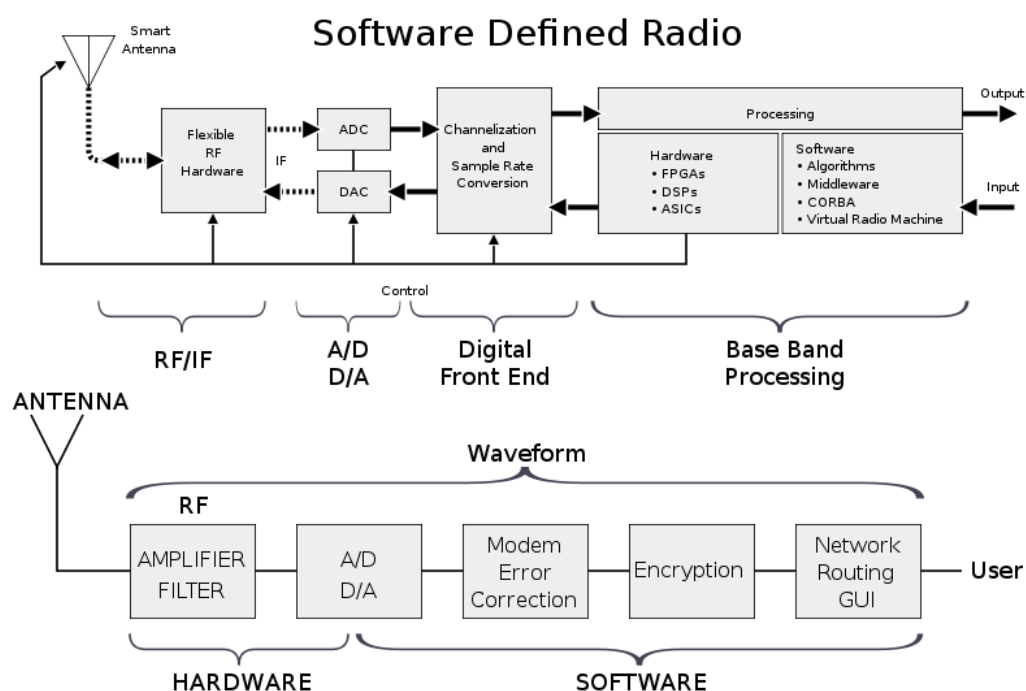


Figura 1: Arquitectura general de un sistema SDR

La plataforma ETTUS B200 utilizada implementa tanto el front-end de RF como el front-end digital enviando luego las muestras obtenidas a una PC mediante bus USB, como se observa en la Fig. 2. Si bien se cuenta con el código de descripción de hardware (HDL) correspondiente, para su efectivo uso en el desarrollo de arquitecturas de comunicaciones digitales y dado que la documentación disponible es muy escasa, esta arquitectura debe ser relevada y ejercitada.

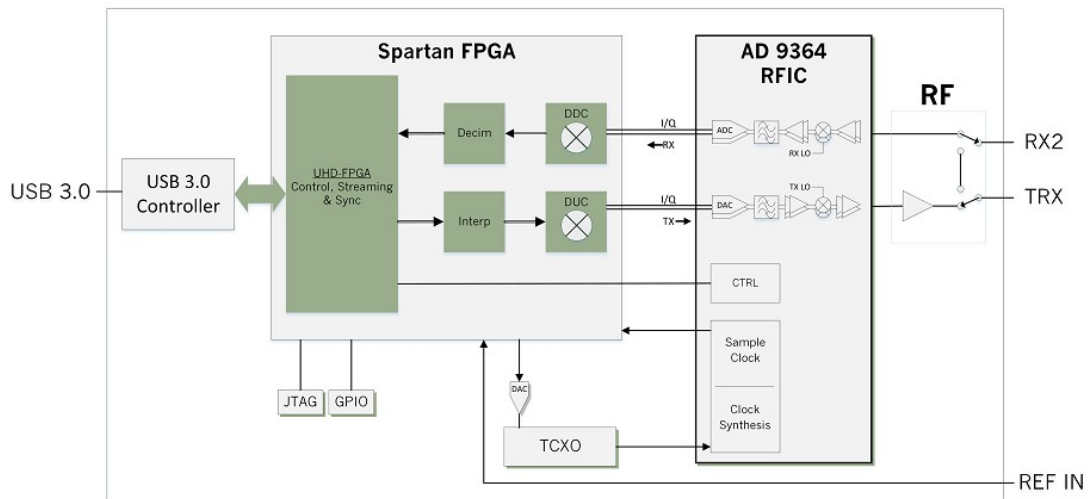


Figura 2: Cadena de procesamiento en la placa ETTUS B200

A fin de organizar las tareas necesarias, se establece el siguiente plan de trabajo:

- En primer término, se realiza una adaptación del entorno de software y de herramientas, tales como Xilinx ISE, driver *uhd*, gnuradio, y librerías accesorias de software.
- Luego, se ejecutan ensayos de comunicación con la placa, primero haciendo uso de imágenes de hardware predefinidas, y luego realizando modificaciones a estas imágenes a fin de comprobar su funcionamiento.
- Finalmente, se realiza un relevamiento detallado del camino de datos de transmisión, inspeccionando el código verilog y RTL de los módulos que realizan la parte relevante del procesamiento digital de señal (DSP). Como punto de particular interés, se identifican los protocolos de comunicación utilizados, tales como AXIS, CHDR y otros particulares del proveedor. Esto permitirá la incorporación a la arquitectura base de nuevos módulos desarrollados en el Laboratorio con fines específicos.

2. DESARROLLO

2.1. Adaptación del entorno de software y herramientas de trabajo

Para poder trabajar con la plataforma ETTUS B200, se realizó la instalación, configuración y adaptación de las herramientas, entre las cuales podemos mencionar como más relevantes:

- **Driver UHD (Universal Hardware Device):** controlador que permite la comunicación con la plataforma ETTUS B200. Asimismo, mediante la instalación de un módulo específico, permite la integración con la librería Gnuradio para procesamiento de las muestras obtenidas.
- **Xilinx ISE** (Integrated Development Environment): entorno utilizado para desarrollar, ensayar y sintetizar las imágenes de hardware reconfigurable, que serán luego transferidas a la FPGA Xilinx Spartan.
- **Gnuradio:** librería abierta de módulos software para procesamiento digital en sistemas SDR.

A continuación, se documentan los pasos a seguir para instalar estas herramientas sobre un SO Linux Ubuntu 18.04.2 LTS, 64bits.

Instalación del driver UHD [1]

a) Descarga e instalación de dependencias

```
$ sudo apt-get install libboost-all-dev libusb-1.0-0-dev python-makodxygen python-docutils cmake build-essential python bash build-essential
```

b) Selección, descarga e instalación del driver UHD a partir del repositorio provisto por el desarrollador Ettus.

Repositorio: [2] <https://github.com/EttusResearch/uhd>

Selección de branch: 3.10.3.0

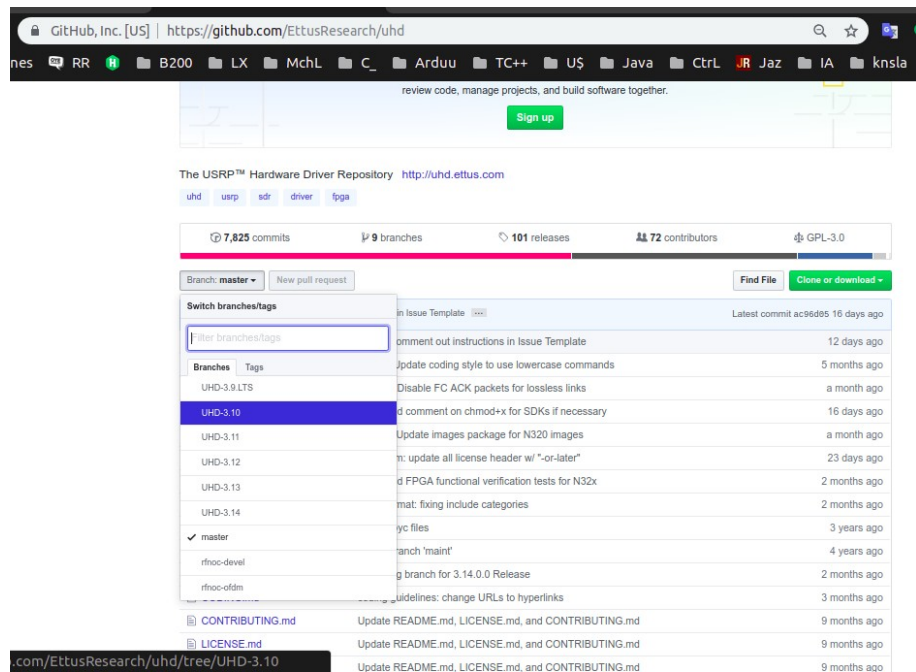


Figura 3: vista del repositorio UHD conteniendo el código fuente del driver

c) A fin de asegurar la correcta interacción entre driver y FPGA, el fabricante fuerza a controlar la compatibilidad entre distintas versiones del driver y sus respectivas versiones de imágenes para el FPGA, lo cual debe ser revisado cuidadosamente para evitar problemas posteriores en la ejecución. Por ello, se inspecciona el número de compatibilidad aceptado para FPGA, especificado en archivo `b200_impl.hpp` del código fuente del driver. En nuestro caso, el driver UHD acepta compatibilidad con código FPGA entre 5 y 14.

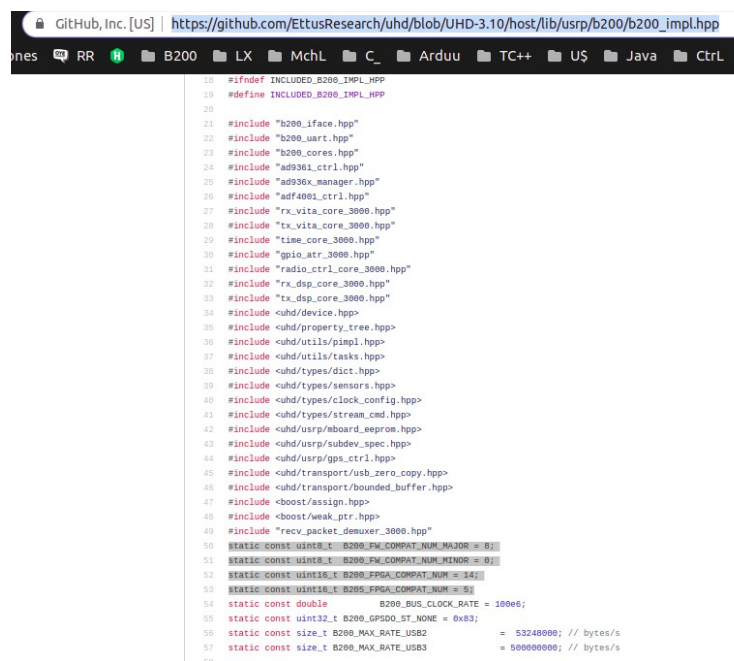


Figura 4: comprobación de compatibilidad del driver UHD

d) Acceso al código HDL de la imagen del FPGA, como se observa en la Fig. 5.

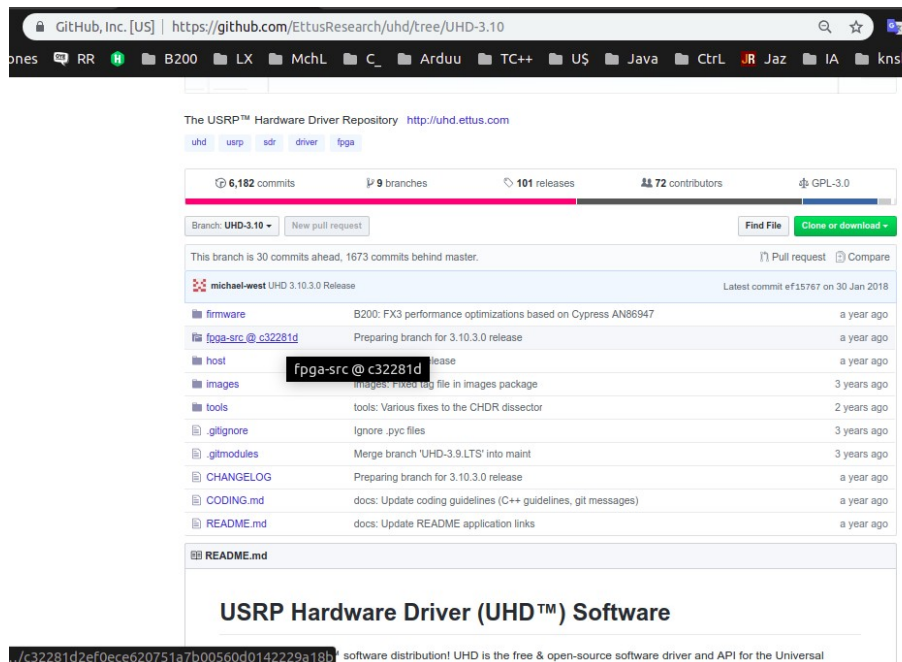


Figura 5: vista del repositorio conteniendo el código HDL para el FPGA

Para nuestro caso, se selecciona el branch “maint”, como se observa en la Fig. 6:

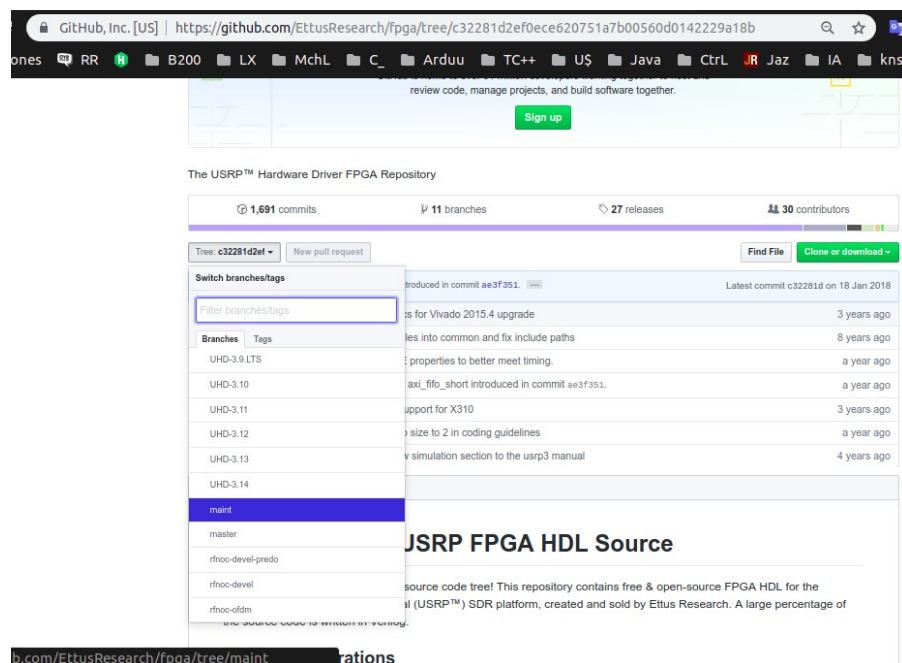
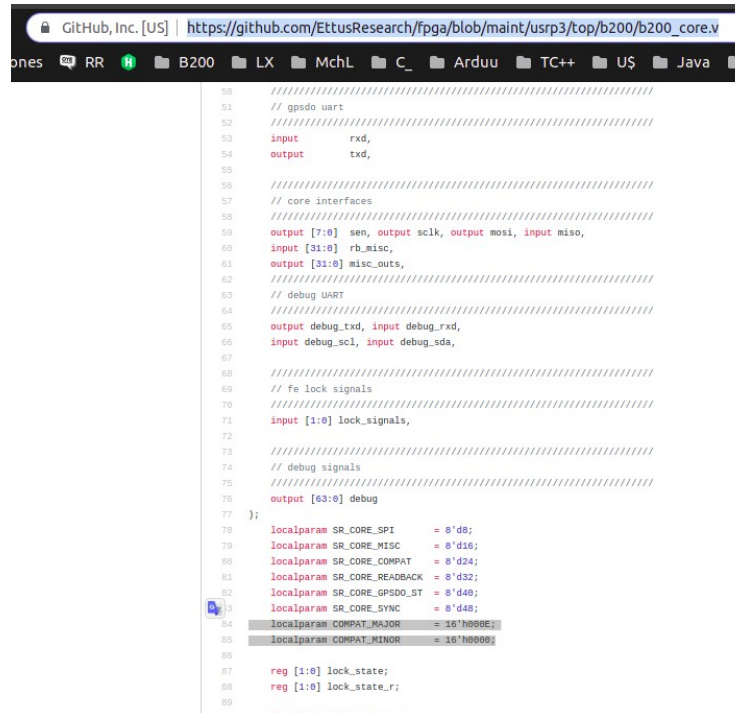


Figura 6: selección de branch de código HDL

A continuación se inspecciona de numero de compatibilidad de FPGA , descritos en archivo b200_core.v y que debe corresponder con los números presentes en el código del driver. En este caso, el código del FPGA presenta números de compatibilidad entre 0 y 14.

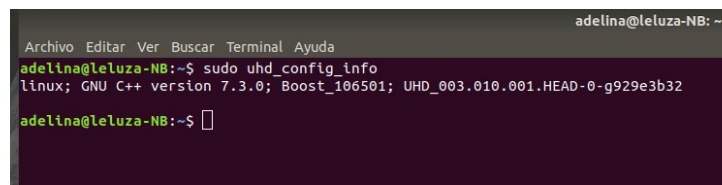


```
50 //////////////////////////////////////////////////
51 // gpsdo uart
52 //////////////////////////////////////////////////
53 input    rxd,
54 output   txd,
55
56 //////////////////////////////////////////////////
57 // core interfaces
58 //////////////////////////////////////////////////
59 output [7:0] sen, output sclk, output mosi, input miso,
60 input [31:0] rb_misc,
61 output [31:0] misc_outs,
62 //////////////////////////////////////////////////
63 // debug UART
64 //////////////////////////////////////////////////
65 output debug_txd, input debug_rxd,
66 input debug_scl, input debug_sda,
67
68 //////////////////////////////////////////////////
69 // fe lock signals
70 //////////////////////////////////////////////////
71 input [1:0] lock_signals,
72
73 //////////////////////////////////////////////////
74 // debug signals
75 //////////////////////////////////////////////////
76 output [63:0] debug
77
78 );
79 localparam SR_CORE_SPI      = 8'd8;
80 localparam SR_CORE_MISC    = 8'd16;
81 localparam SR_CORE_COMPAT   = 8'd24;
82 localparam SR_CORE_READBACK = 8'd32;
83 localparam SR_CORE_GPSDO_ST = 8'd40;
84 localparam SR_CORE_SYNC     = 8'd48;
85 localparam COMPAT_MAJOR     = 16'h800E;
86 localparam COMPAT_MINOR     = 16'h8000;
87
88 reg [1:0] lock_state;
89 reg [1:0] lock_state_r;
```

Figura 7: comprobación de números de compatibilidad para el código HDL

Con esto, los números de compatibilidad entre driver UHD y FPGA quedan corroborados.

Una vez bajado e instalado el driver, se puede verificar que la versión sea la pretendida haciendo uso del comando que muestra la versión de software como muestra la Fig. 8.



```
adelina@leluza-NB: ~
Archivo Editar Ver Buscar Terminal Ayuda
adelina@leluza-NB:~$ sudo uhd_config_info
linux; GNU C++ version 7.3.0; Boost_106501; UHD_003.010.001.HEAD-0-g929e3b32
adelina@leluza-NB:~$
```

Figura 8: comprobación de la versión del driver instalado

Una vez bajada una imagen sintetizada `usrp_b200_fpga_c.bin` a la FPGA e instalado el driver junto a sus herramientas asociadas, y bajo el supuesto de que sus números de compatibilidad sean los correctos, se comprueba el reconocimiento de la placa Ettus B200 por la computadora mediante el comando `uhd_usrp_probe`, tal como lo muestra la Fig. 9.

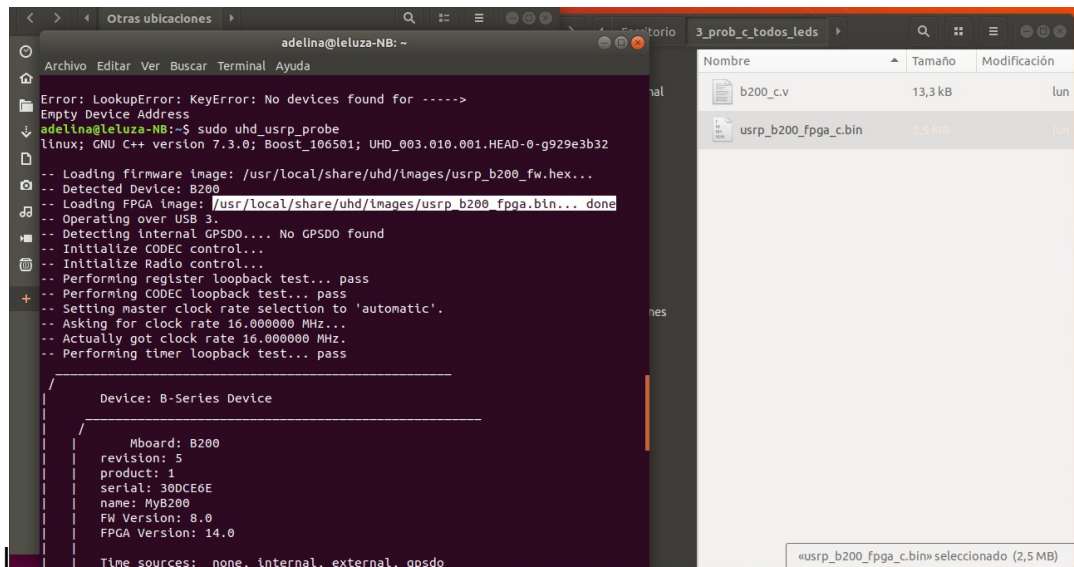


Figura 9: reconocimiento de la plataforma B200 mediante la herramientas del driver

e) Compilación de la imagen de hardware para la plataforma USRP B200
(fuente: https://files.ettus.com/manual/md_usrp3_build_instructions.html)

Para la compilación del código de hardware USRP es necesario un compilador de código HDL, como el que provee la herramienta Xilinx ISE. Se detalla a continuación la instalación de este entorno.

Instalación de Xilinx ISE

Para la instalación se siguieron los pasos de la guía <https://www.youtube.com/watch?v=meO-b6lb17Y>, seleccionando “opt” como directorio de instalación, y licencia *WebPack*.

Pasos para compilar:

1. Agregar `xtclsh` a la variable `PATH`.
2. Correr el script para 64bits.

```
$ source top/Xilinx/14.7/ISE_DS/settings64.sh (64-bit platform)
```

3. En la carpeta de FPGA previamente descargada, ubicación `/usrp3/top/b200`, ejecutar

```
$ make B200
```

entre varios otros archivos, la compilación arroja el archivo binario para cargar en FPGA:

```
FPGA/usrp3/top/b200/build/usrp_b200_fpga.bin
```

También es posible generar el archivo de proyecto para GUI de la herramienta ISE Xilinx, ejecutando:

```
$ make B200 PROJECT_ONLY=1
```

Instalación de librerías Gnuradio

Aclaración: para evitar problemas de incompatibilidad de versiones con el driver UHD, se debe instalar GnuRadio **después** del driver UHD.

La instalación de GnuRadio se puede realizar desde su código fuente o desde el repositorio de ubuntu. Dado que nuestro trabajo se concentrará en el aspecto de la plataforma, optamos en este caso por la última opción:

```
$ apt install gnuradio
```

Posteriormente, se verifica la instalación del software GnuRadio, utilizando la versión de UHD instalada previamente, como se observa en la Fig. 10.

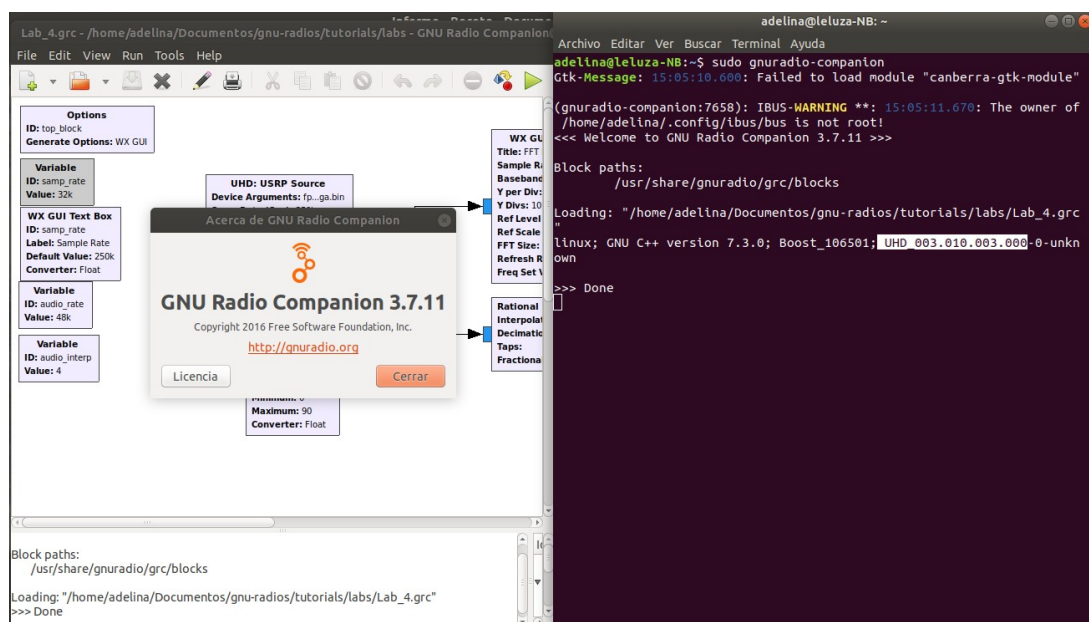


Figura 10: comprobación de la versión instalada de GnuRadio

2.2. Pruebas de comunicación con la placa Ettus B200

Prueba del camino de comunicación PC - plataforma ETTUS B200

En este ensayo, se comprueba la correcta comunicación entre la PC y los tres componentes principales de la plataforma:

- Microcontrolador FW3 de Cypress, que gestiona la comunicación USB 3.0 mediante firmware bajado desde la PC
- FPGA, que recibe la imagen de la PC, y que por defecto realiza tareas de canalización y down/up sampling
- Front-end de RF, que recibe comandos de configuración desde la PC

Como primer ensayo para comprobar la correcta comunicación, se ejecutó un programa receptor de fm, correspondiente al Lab 4 provisto por Ettus, y que se observa en la Fig. 11. (fuente: https://files.ettus.com/tutorials/labs/Lab_1-5.pdf).

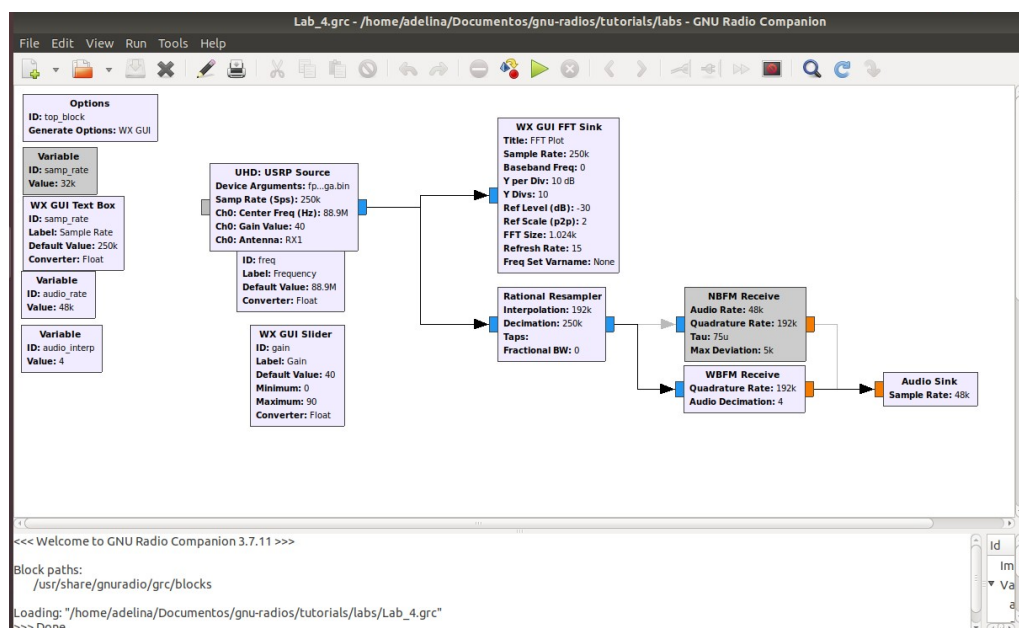


Figura 11: ejemplo de ensayo con un receptor de FM

El programa recibe una señal en banda base, la cual es procesada haciendo uso de un demodulador de FM fijado en el valor de 88.9 Mhz (estación de FM Pobre Johnny). De esta manera se conecta, el programa desde la pc, con ambos módulos de la placa, FW3 y FPGA, cargando sus respectivas imágenes: `usrp_b200_fw.hex` y `usrp_b200_fpga.bin`.

Estos modulos son representador por el bloque UHD: USRP Source. En la Fig. 12 se observa el cambio del valor del argumento del bloque *USRP Source* para que tome la imagen desde la ubicación donde la herramienta de síntesis genera el archivo binario para FPGA.

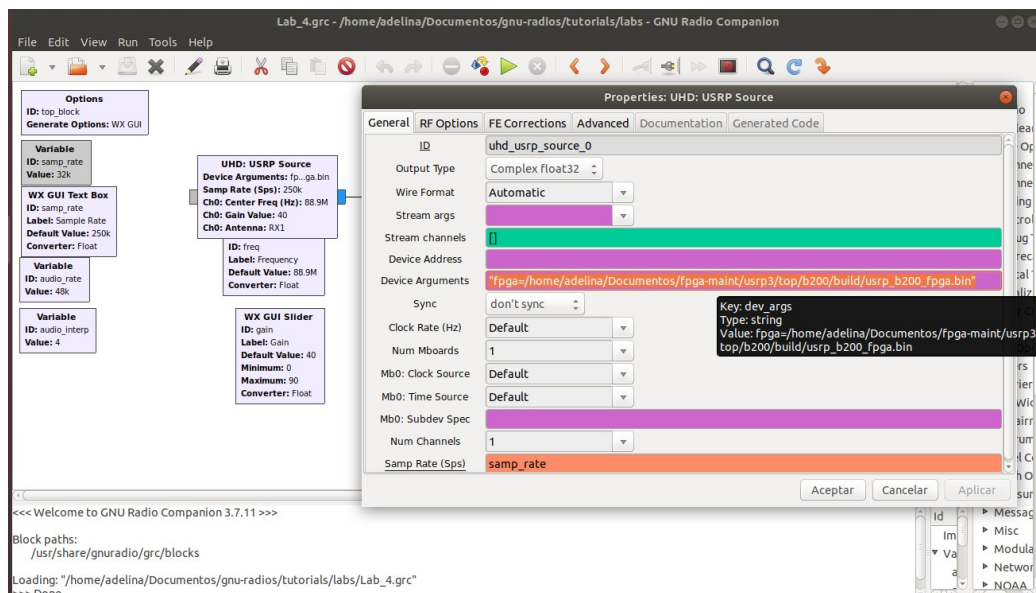


Figura 12: configuración de parámetros referidos a la ubicación de imagen para FPGA

También es posible evidenciar la carga de los archivos de ambos módulos ejecutando `uhd_usrp_probe`

`--args="fpga=/CarpetaFPGA/usrp3/top/b200/build/usrp_b200_fpga.bin"`, como muestra la Fig. 13.

```
adelina@leluza-NB: ~
Archivo Editar Ver Buscar Terminal Ayuda

TX Codec: A
Name: B200 TX dual DAC
Gain Elements: None

adelina@leluza-NB:~$ sudo uhd_usrp_probe --args="fpga=/home/adelina/Documents/f
pga-maint/usrp3/top/b200/build/usrp_b200_fpga.bin"
linux; GNU C++ version 7.3.0; Boost_106501; UHD_003.010.001.HEAD-0-g929e3b32

-- Detected Device: B200
-- Loading FPGA image: /home/adelina/Documents/fpga-maint/usrp3/top/b200/build/
usrp_b200_fpga.bin... done
-- Operating over USB 3.
-- Detecting internal GPSDO... No GPSDO found
-- Initialize CODEC control...
-- Initialize Radio control...
-- Performing register loopback test... pass
-- Performing CODEC loopback test... pass
-- Setting master clock rate selection to 'automatic'.
-- Asking for clock rate 16.000000 MHz...
-- Actually got clock rate 16.000000 MHz.
-- Performing timer loopback test... pass

Device: B-Series Device

Mboard: B200
revision: 5
product: 1
serial: 30DCE6E
name: MyB200
```

Figura 13: proceso de carga de imagen desde consola

Instalación y ensayo de una imagen de fpga modificada

En el presente punto, se pretende realizar una modificación a la imagen que se carga en la FPGA, llevando adelante todos los pasos necesarios para que esta nueva imagen se baje efectivamente y se ejecute sobre la plataforma. A fin de simplificar este primer paso, se

plantea una primera modificación que no involucre el camino de procesamiento de datos, por lo que se modifica la arquitectura con el fin de prender un led determinado de la placa.

En el sitio del fabricante se encuentra un esquemático común a la placa B210 y la placa B200 (fuente: <https://files.ettus.com/schematics/b200/>). En la Fig. 14 podemos ver que los leds del bloque front-end están etiquetados como “TX2”. Sin embargo, en la placa B200, se corresponden con los registros TX1.

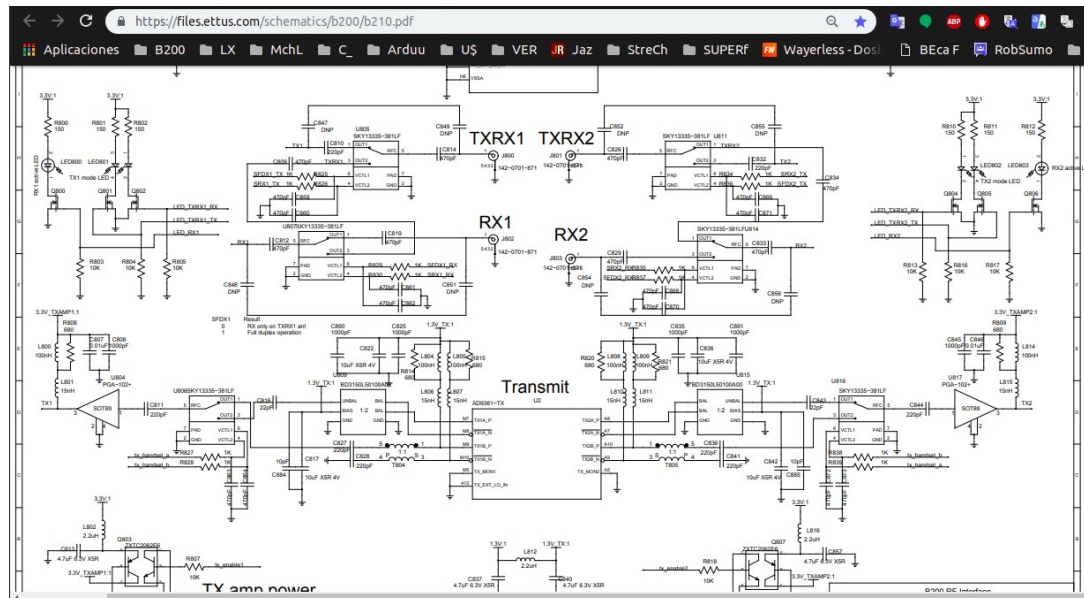


Figura 14: esquemático de las placas B200/B210, mostrando ubicación del LED a activar

Las modificaciones introducidas en el archivo `b200.v` para prender un led del front-end son (Fig. 15):

- 1) declaración de un cable, para redireccionar la señal led-on de TX-1.
- 2) colocación de un “1” en la salida hacia led-TX1 (se trabaja con lógica positiva por estar conectado con un transistor npn)

```

Abrir  *b200.v  Guardar
// Frontend assignments
// Most B2x0's have frontends swapped (radio0 to FE2), but some hardware revisions do not.
// The ATR pins are mapped from radio to frontend here based on the swap_atr_n bit.
wire swap_atr_n;
wire [7:0] radio0_gpio, radio1_gpio;
reg [7:0] fe0_gpio, fe1_gpio;
wire led_rx1_aux; // declarando la nueva salida

always @(posedge radio_clk) begin //Registers in the IOB
    fe0_gpio <= swap_atr_n ? radio1_gpio : radio0_gpio;
    fe1_gpio <= swap_atr_n ? radio0_gpio : radio1_gpio;
end
assign {tx_enable1, SFDX1_RX, SFDX1_TX, SRX1_RX, SRX1_TX, led_rx1_aux, LED_TXRX1_RX, LED_TXRX1_TX} = fe0_gpio; //redireccionando el led de encendido RX1 hacia la nueva salida declarada
assign {tx_enable2, SFDX2_RX, SFDX2_TX, SRX2_RX, SRX2_TX, LED_RX2, LED_TXRX2_RX, LED_TXRX2_TX} = fe1_gpio;

assign LED_RX1 = 1'b1; // modificacion introducida --> prendiendo led independiente

wire [31:0] misc_outs; reg [31:0] misc_outs_r;
always @(posedge bus_clk) misc_outs_r <= misc_outs; //register misc ios to ease routing to flop
wire spare_signal; // WAS codec_arst...now obsolete, this bit can be reused when UHD doesn't contain codec_arst code.

```

Figura 15: cambios realizados para activación de un LED

2.3.Exploración del área de comunicaciones inalámbricas en FPGA

2.3.1.Relevamiento del camino de transmisión de los datos

El camino general de datos se puede observar en la Fig. 16. Los datos a transmitir, ingresan a la FPGA por el puerto *GPIF_D* [31:0], desde el chip FW3, y sale hacia el módulo front-end, por el puerto *tx* [31:0].

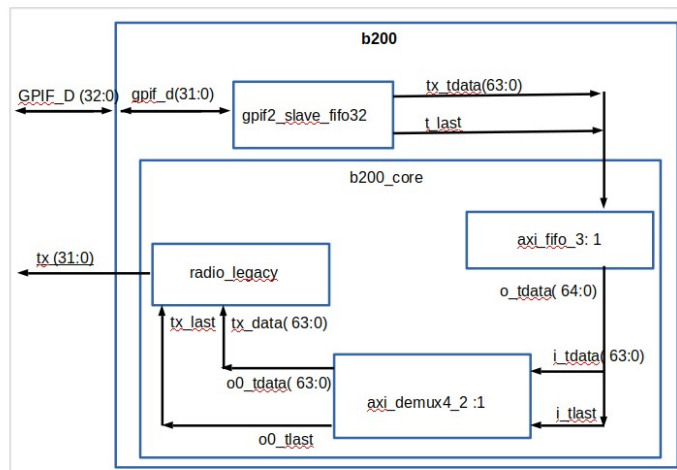


Figura 16: camino de datos en la arquitectura del FPGA

Los módulos que componen el camino de transmisión son:

- gpio2_slave_fifo32
 - axi_fifo_3
 - axi_demux_4
 - radio_legacy → contiene los componentes del DSP, además de las configuraciones y el control del hardware y clocks.
-
- axi_fifo → almacena la información en palabras de 64 bits, configuradas por CHDR.
 - tx_deframer → Reorganiza la información a enviar. Adjunta a cada muestra, su respectiva información de tiempo y encabezado del paquete al que pertenece.
 - tx_control → Devuelve las muestras “puras”, sin información del protocolo de transporte CHDR.
 - responder → Modulo opcional, representa el entorno para el funcionamiento de los demás módulos.
 - duc_chain → operaciones DSP.

Protocolos utilizados

La comunicación , tanto entre módulos, como con el exterior de la placa, es gestionada por 3 protocolos:

1. Protocolo AXI4-Stream, responsable de la comunicación entre módulos convencionales de FPGA.
2. Protocolo Run-Strobe, responsable de la transferencia del dato puro hacia el modulo de DSP.
3. Protocolo CHDR, encargado del aspecto de radio transporte.

Podríamos resumir, diciendo que el protocolo CHDR guarda relacion con la señal, mientras los protocolos AXI4-Stream y Run-Strobe, gestionan la comunicacion entre bloques.

Protocolo AXI4-Stream

AXI4-Stream es el protocolo encargado de transmitir por el canal TDATA, vectores de muestras, en una única dirección. El canal es de ancho ajustable, y gestionado por las señales de VALID, READY, y LAST, de la siguiente manera:

Cuando el módulo emisor o “master”, carga los datos en el canal, activa la señal de VALID. De esta manera, el módulo receptor o “slave”, es notificado de la existencia de datos a transmitir. A su vez, cuando éste se encuentra disponible para la recepción de datos, activa la señal de READY, notificando al emisor. La señal LAST, es activada por el emisor, cuando coloca el último dato en el canal. Por lo tanto, podemos concluir que *existirá transferencia, siempre que VALID y READY estén activos en simultaneo*.

Protocolo RUN - STROBE

El protocolo Run-Strobe, es ligeramente más simple. Únicamente maneja las señales de RUN para comunicar la existencia de datos al receptor (equivalente a VALID) y STROBE, para comunicar al emisor la disponibilidad para recibir (equivalente a READY).

De igual manera, podemos concluir que, existirá transferencia, siempre que RUN y STROBE estén activos en simultaneo.

Protocolo CHDR

Para dispositivos de radio, definidos por software, existen 2 protocolos relevantes: VRT (VITA Radio Transport) y CHDR (encabezado comprimido, protocolo específico de Ettus). Este ultimo es utilizado por B200 y otros dispositivos de 3era generacion. El protocolo CHDR, es el encargado de intercambio de muestras entre host y dispositivos. Es un protocolo trivial al protocolo VRT (defined by the VITA-49 standard.).

La razon por la que se utiliza un protocolo de transporte como VITA, es para interoperabilidad

entre equipos transeptores y lo que generan procesamiento, diferente proveedores, facilitando el la comunicacion entre estos. Es por ello, que salva aspectos como marcas de tiempo y paridad.

CHDR es un protocolo que “comprime” la información reflejada por el estándar VITA-49, es menos complejo, y organizado en palabras de longitud fija de 64 bits.

El protocolo contiene la información descriptiva del paquete completo, en la primer palabra, denominada “header”. Además, una segunda palabra opcional, con información sobre las marcas temporales, y el resto de palabras, con los datos relevantes.

Es compatible con arquitecturas Little and Big Endian.

Las marcas de tiempo, utilizan el formato de tiempo universal coordinado (UTC), el cual empieza a contar desde las 00:00:00 del 1/Ene/1970, representado como un entero sin signo de 32 bits.

SID o Stream ID, es un identificador de 32 bits, que asocia paquetes a flujos.

La Tabla 1 describe el significado de los bits que conforman la palabra de cabecera del paquete, mientras que la Tabla 2 detalla los tipos de paquetes según su código.

bit	Descripción
64:63	Tipo de paquete (descrito a continuación)
61	Tiene(1) o no (0) marcas de tiempo
60	Ultimo paquete de la ráfaga o Bandera de error
59:48	Número de secuencia (12 bits)
47:32	Longitud del paquete en Bytes (16 bits)
31:0	Stream ID (32 bits)

Tabla 1: encabezado del paquete CHDR

Bit 63	Bit 62	Bit 60	Packet Type
0	0	0	Data
0	0	1	Data (End-of-burst)
0	1	0	Flow Control
1	0	0	Command Packet
1	1	0	Command Response
1	1	1	Command Response (Error)

Tabla 2: Tipos de paquetes posibles

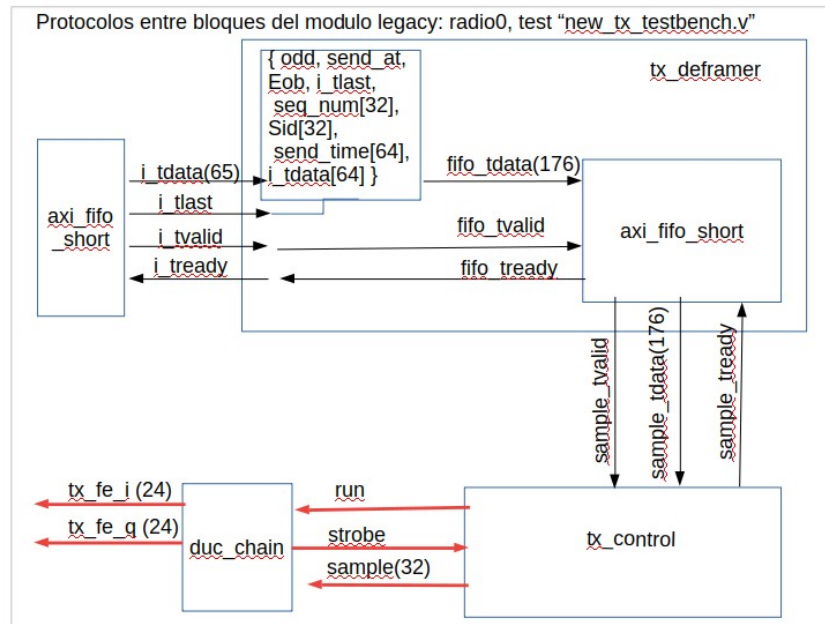


Figura 17: buses utilizados

La Fig. 17 expone la utilización de los protocolos AXI4-Stream y CHDR entre los primeros 4 bloques (flechas negras), y el protocolo Run-Strobe entre los últimos 2 bloques(flechas rojas).

2.3.2. Simulación y análisis de módulos y protocolos.

Análisis de los protocolos

Se ejecutó el test-bench `new_tx_tb.v` para analizar el comportamiento de los protocolos AXI4-Stream, CHDR y Run-Strobe, a través de los módulos `axi_fifo`, `deframer`, `control` y `duc_chain`. Esto permite introducirse mas en la arquitectura, poder analizar el camino y evidenciar el uso de los protocolo. Lo que no es posible, bajando nada mas la imagen a la placa.

El test-bench envía una ráfaga, compuesta por 2 paquetes que contienen 100 pares de muestras cada uno. Esto se realiza a partir de las tareas "send_burst" y "send packet".

Podemos apreciar la utilización del protocolo AXI4-Stream con tamaño de canal ajustable, que transporta los paquetes entre los módulos `axi-fifo/deframer`, y `deframer/control`.

La comunicación entre los módulos `control/duc_chain`, está regida por el protocolo Run-Strobe.

Los 2 paquetes que componen la ráfaga, cumplen con el transporte de 50 pares de muestras de información, con valores consecutivos, cada una; mas configuraciones por el protocolo CHDR.

Cada paquete, esta dividido en tantas palabras, como cantidad de pares de muestras, más una palabra de cabecera, y otra palabra con marcas de tiempo.

A continuacion, se describe la información contenida en la primer palabra del primer paquete enviado (primer marca celeste), correspondiente a la cabecera CHDR: `"0 2000 0068 deadbeef"` (hexa), e ilustrada en los resultados del simulador ISIM de la Fig. 18.

Resulta útil destacar que el ancho de la palabra son 64 bits, determinados por CHDR. Sin embargo, el ancho de canal de transmisión entre, la tarea `send_packet` y el modulo `axi_fifo`, es de 65 bits. Esto se debe, a que la señal de `LAST`, perteneciente al protocolo AXI4-Stream, esta adjuntada como bit mas significativo a la palabra, completándose así, los 65 bits correspondientes al ancho del canal.

La segunda palabra del primer paquete enviado (segunda marca celeste), representando las marcas temporales del paquete: `"0 0000 0000 0000 0100"` (hexa), se puede observar en la Fig. 18.

Nº Bit	SIGNIFICADO	VALOR
64 (MSB)	Señal "last" (pertenece al protocolo AXIS)	1'b0 (no es la última palabra)
63:62	Tipo de paquete	2'b00 (paquete de dato)
61	Contiene marcas de tiempo	1'b1 (sí)
60	Fin de ráfaga / bandera de error	1'b0 (no es fin de ráfaga)
59:48	Número de la secuencia de paquete	3'h000
47:32	Longitud del paquete en Bytes	4'h0068 (104 Bytes)
31:0	ID Stream	8'hDeadbeef (relleno)

Figura 18: lectura de palabras CHDR, cabecera y marcas temporales, del primer paquete.

En la figura 19, podemos apreciar la implementación del protocolo AXI4-Stream entre la tarea “send_packet” y los módulos `axi-fifo`, `axi-fifo/deframer`.

En la primera marca de tiempo en color celeste de la Fig.19, se puede ver que se activa la señal `VALID i_tvalid` y cargan los datos en el canal `i_tdata[64:0]`. Debido a que, la señal `READY i_tready` estaba activa desde antes, la transferencia hacia el módulo `axi-fifo`, por la tarea `send_packet` es iniciada.

De igual manera sucede, entre los módulos `axi-fifo` y `deframer`. Esto es representado por la segunda marca temporal celeste, en la Fig.19.

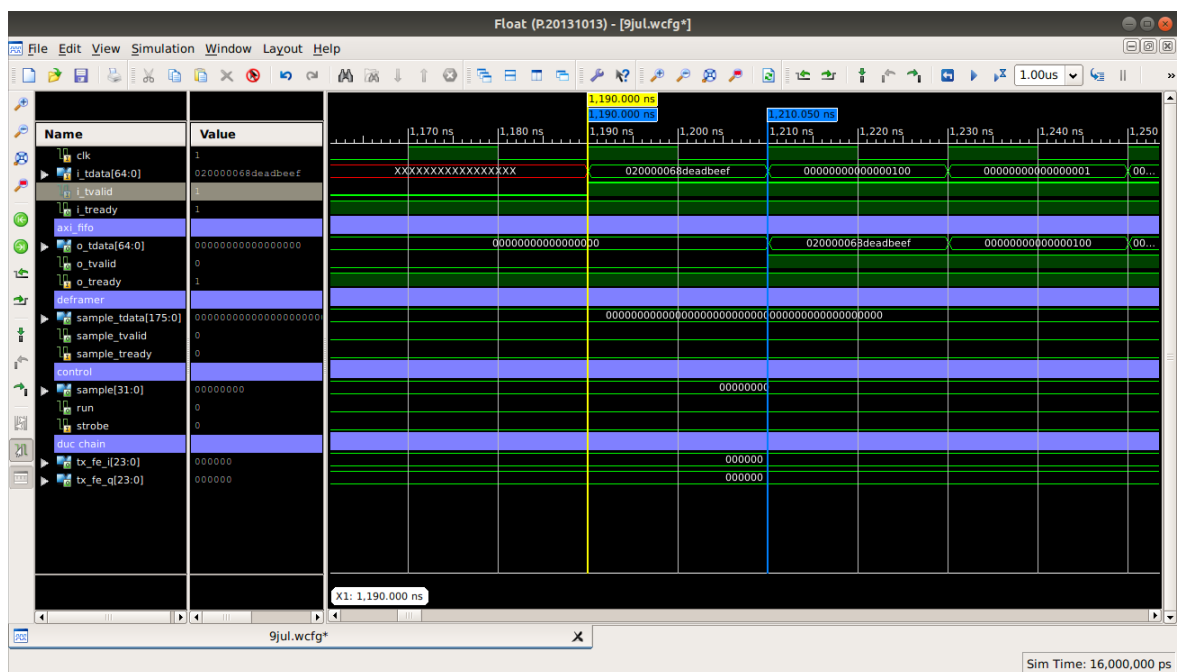


Figura 19: Inicio de transferencia por los módulos gestionados por el protocolo AXI4-Stream

La Fig.20, la primer marca celeste, denota una situación donde continúa la transmisión hacia el módulo `axi-fifo`, por parte de la tarea, debido a que esta, no responde al a la señal de `READY`.

Cabe aclarar que, la tarea “send_packet”, no es un módulo, y no está diseñada para responder como tal al protocolo AXI4-Stream. Es por ello que continúa cargando `i_tdata[64:0]`, a pesar de estar desactivada la señal `READY`.

Un tiempo más tarde, se vuelve a activar la señal `READY` (segunda marca celeste) del módulo `axi-fifo`, pero al no existir más palabras para transmitir por parte de la tarea, no se activa nuevamente la señal `VALID` de la tarea.

Esto evidencia una pérdida de muestras a partir del valor `00042004100420042` en adelante, pertenecientes al segundo paquete. Es decir, los pares de muestras de información ingresadas al modulo `axi-fifo`, van desde el valor `0000000000000001-00f500f500f500f6` hasta el valor `0001000000010001-003d003c003d003d`.

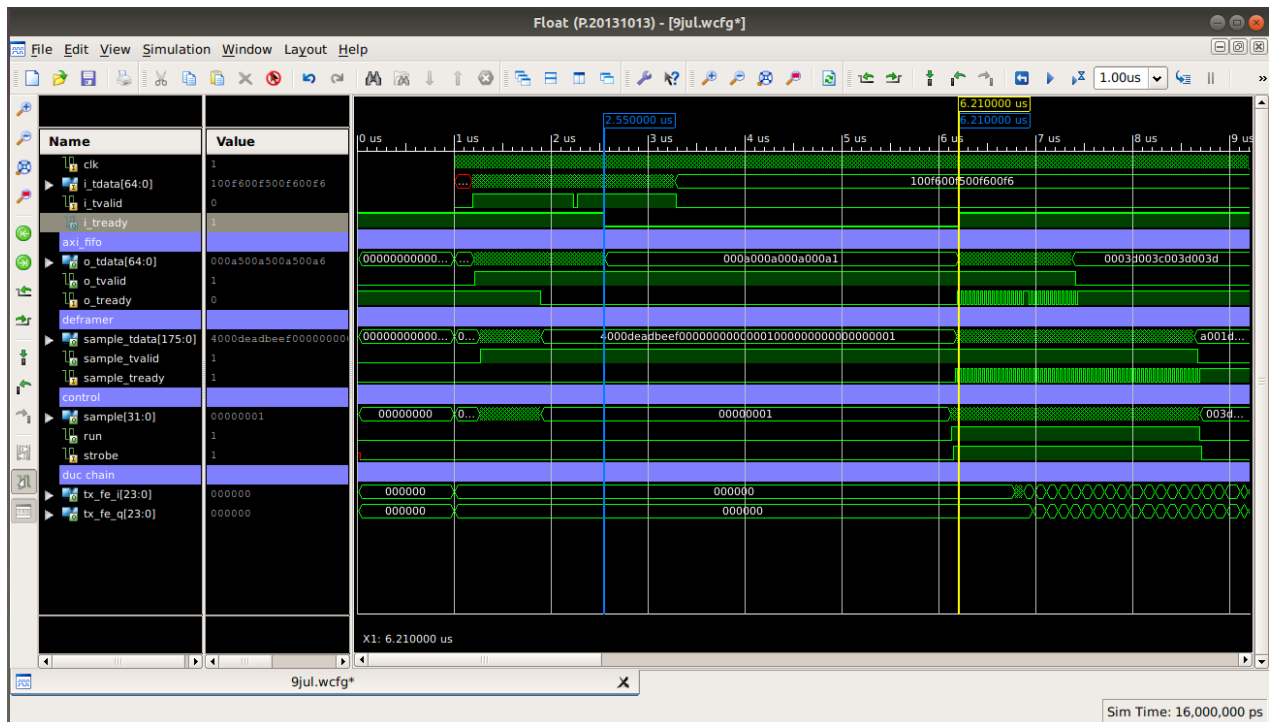


Figura 20: Muestras despreciadas

En la Fig. 21, se puede notar la interrupción de transferencia desde el módulo `axi_fifo`, hacia el módulo `deframer`, por desactivar la señal de `READY` `o_tready` (marca amarilla). La transferencia se reanuda, cuando se vuelve a activar dicha señal (marca celeste).

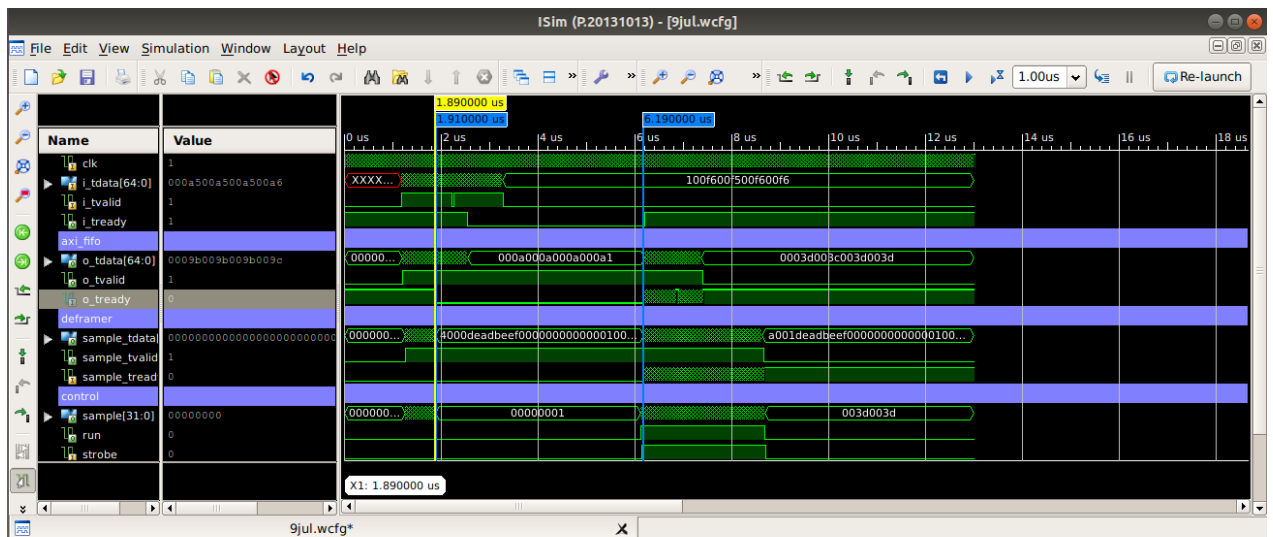


Figura 21: Señal `READY` desactivada, por módulo `deframer`.

Las señales de gestión de transferencia desde el módulo `deframer`, hacia `control` son: `sample_tvalid` y `sample_tready`. El canal tiene un ancho de palabra de 176 bits (`sample_tdata[175:0]`), que incluye la señal de `LAST`.

El modulo `deframer`, contiene una fifo de ancho 176. Esta es utilizada para almacenar y adjuntar a cada muestra de información, su encabezado y marcas temporales

correspondientes.

Esto lo podemos ver en la Figura 22, en el valor señalado por la primer marca celeste "5000 deadbeaf 00000000000000100 00f500f500f500f6"

Equivalentes a :

- "5" HEXA = 0101 BIN, representa la paridad, si contiene marcas temporales, si es el ultimo paquete d ella ráfaga y si es el ultimo valor enviado
- "000" el numero de secuencia
- "deadbeef", el SID
- "00000000000000100", las marcas temporales
- "00f500f500f500f6", las muestras de información

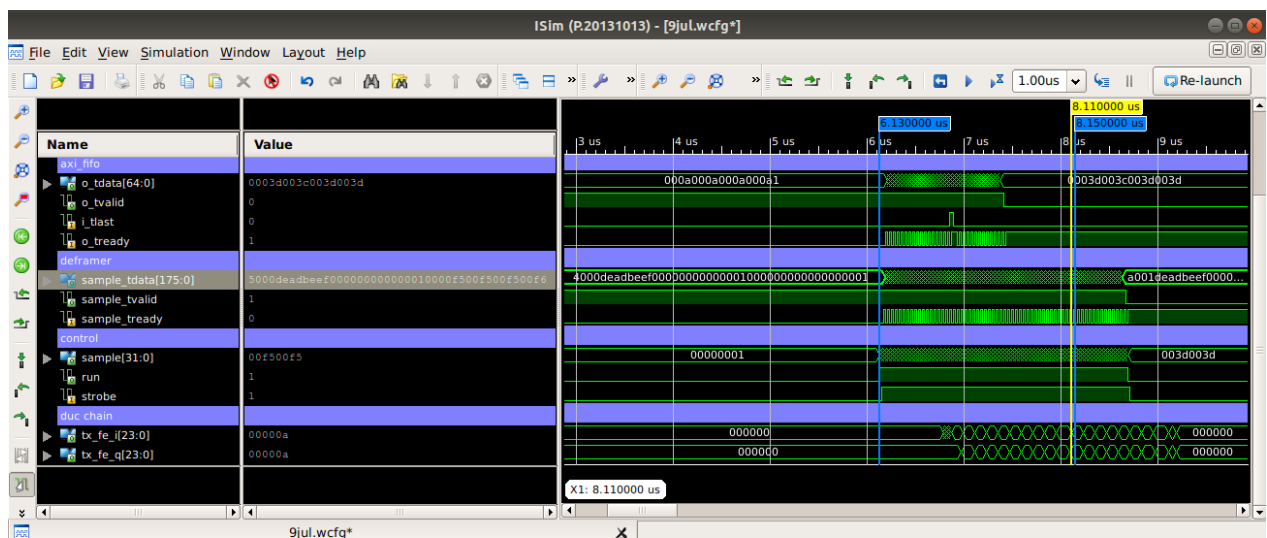


Figura 22: Discriminación de palabras entre el primero y segundo paquete.

Para comunicar desde el módulo `control`, hacia el módulo `duc_chain` encargado del DSP, se utiliza el protocolo Run-Strobe. La transferencia es simplemente gestionada por las señales `RUN` y `STROBE` únicamente. La transmisión cesa, un ciclo después de desactivarse la señal de `RUN`, procedente del modulo `control`.

La simpleza del protocolo de comunicación, es debido a que, a partir de esta etapa de la comunicación, el protocolo CHDR ya no se encuentra presente. Simplemente son transferidos los pares de muestras de información.

El modulo `duc_chain`, es el encargado de canalizar e interpolar las muestras. Estos procesos, se encargar de colocar la información, en el ancho de banda deseado y correspondiente con las especificaciones del Front-end. Se realiza un muestreo de la señal, a una mayor tasa de frecuencia, para adaptarse a las velocidades del Front-End, y poder transmitir. En el código, es posible ver esta implementación mediante el uso de filtros CIC y HB.

De la Fig. 23, podemos apreciar el inicio (marca amarilla) y final (marca celeste) de la transmisión de datos por el canal `sample[31:0]`, que transporta los pares de muestras hasta el modulo DSP. El dato a la salida del modulo DSP, es una secuencia, reflejo de la secuencia enviada inicialmente.

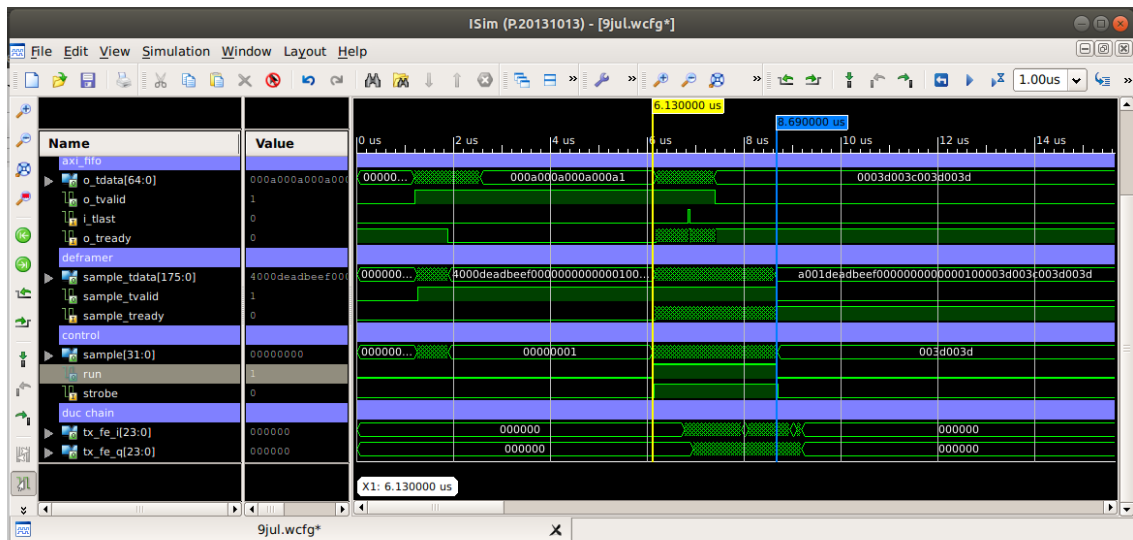


Figura 23: Ejecución protocolo Run – Strobe.

A continuación, se detallan las modificaciones agregadas en 2 archivos para la ejecución del test bench.

Modificaciones agregadas al archivo `new_tx_tb.v` :

- agregar al inicio del archivo la directiva ``define ISIM` para que se ejecute con simulador ISIM; de lo contrario, el proyecto no encuentra el simulador.
- el valor del dato se incrementa en pasos decimales de 20485, esto es, con la finalidad de obtener valores al final. De lo contrario, las muestras se pierden en el modulo DSP.

```
samp0 <= samp0 + 32'h0005_0005;
```

```
samp1 <= samp1 + 32'h0005_0005;
```

- al comienzo del test, se lanza 1 única ráfaga, compuesta por 2 paquetes, que trasladan 50 pares de muestras de datos cada uno. Por esto, las demás tareas son comentadas. Esto se realiza con la finalidad de reducir la complejidad del test bench, sin involucrar el manejo de errores, y lograr comprenderlo en detalle.

```
send_burst(2/*count*/,100/*len*/,32'hA000_0000/*start*/,64'h100/
*time*/,12'h000/*seqnum*/,1/*sendat*/, 32'hDEADBEEF/*sid*/);
```

Modificaciones agregadas al archivo `duc_chain.v`:

- Redefinición de placa, correspondiente a B200, y utilización de la nueva arquitectura

```
duc_chain
#(.BASE(SR_TX_DSP), .DSPNO(0), .WIDTH(24), .NEW_HB_INTERP(1), .
DEVICE("SPARTAN6")) duc_chain
```

- se realiza una asignación directa de 10 en el factor de escala, reemplazando al módulo de banco de registros `setting_reg` para mayor comodidad. En la practica, este registro es seteado por la pc, pero a fines del test, se seteo un valor en este momento.

```
// setting_reg #(.my_addr(BASE+1), .width(18)) sr_1
// (.clk(clk),.rst(rst),.strobe(set_stb),.addr(set_addr),
// .in(set_data),.out(scale_factor),.changed());
assign scale_factor = 18'd10;
```

Conclusión de la Simulación

El test realizado, permitió la exploraron del camino de transmision de datos, evidenciando un comportamiento correcto de los módulos, y de la interacción de los datos.

Esto permitió reconocer el lugar mas adecuado para una futura colocación de un modulo IPCore generador de símbolos, y los protocolos de interacción con los que trabajaría para la transmisión.

4.CONCLUSIÓN

Considerando el trabajo con una placa, que no era de desarrollo, y carecía de documentación oficial, se realizó adaptación del entorno y herramientas necesarias para trabajar con la placa Ettus B200.

Se logró el conocimiento de la arquitectura interna, y camino de transmisión de la placa Ettus B200. Además de sus 3 protocolos de comunicación utilizados: AXI4-Stream, CHDR y Run-Strobe.

Se identificó el lugar más apropiado para introducir un eventual IPCore en el camino de procesamiento, y de las interfaces necesarias para la interacción con el resto de la arquitectura.

Del relevamiento realizado, resulta documentación disponible, que hasta el día de hoy no existe. Facilitando la introducción la arquitectura de la placa Ettus B200. Para que sea más simple una futura inserción de un módulo dentro de esta, conociendo sus protocolos, para poder comunicar con módulos futuros.

5.BIBLIOGRAFÍA

- [1] Guia Driver UHD: https://files.ettus.com/manual/page_build_guide.html
- [2] Repositorio Driver UHD: <https://github.com/EttusResearch/uhd>
- [3] Descripcion protocolo VITA-49
<https://conferences.sigcomm.org/sigcomm/2013/papers/srif/p45.pdf>
- [4] Referencia Protocolo AXI4-Stream
https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf