Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño de un electrocardiograma portátil en la FPGA Zynq-7000

Autor: Pedro Gutiérrez Lora
Tutor: Fernando Muñoz Chavero
Hipólito Guzmán Miranda

Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño de un electrocardiograma portátil en la FPGA Zynq-7000

Autor:
Pedro Gutiérrez Lora

Tutor:
Fernando Muñoz Chavero
Profesor titular

Hipólito Guzmán Miranda
Profesor Contratado Doctor

Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo Fin de Grado: Diseño de un electrocardiograma portátil en la FPGA Zynq-7000

Autor: Pedro Gutiérrez Lora

Tutor: Fernando Muñoz Chavero

Hipólito Guzmán Miranda

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis maestros

Resumen

La biotecnología ha empezado a formar parte de nuestras vidas casi sin darnos cuenta. Es un hecho que hoy día existen todo tipo de dispositivos capaces de recopilar información biológica del individuo que lo utiliza, y que pueden ser empleados para numerosos fines: en el ámbito médico, cuando se trata de monitorizar las constantes vitales de un paciente; en el ámbito cotidiano, para tener un control de las pulsaciones cuando se practica deporte; incluso en el ámbito de la tecnología, el campo de la bioseguridad, donde se identifica a una persona por su huella dactilar o su voz.

El corazón supondría una de las múltiples ramificaciones que surgen de la biotecnología y la biología en general. Como tal, es un órgano profundamente estudiado y la información que se tiene del mismo es abundante. Sin embargo, cuando se habla de obtener el perfil cardíaco de una persona para, por ejemplo, estudiar una posible enfermedad, la situación se complica. Si bien es cierto que existen dispositivos que pueden ser adquiridos incluso a nivel particular, no pueden ofrecer el nivel de información que se necesita para determinar el origen y la causa de dicha afección cardíaca.

Por otro lado, existe una serie de instrumentación médica empleada en hospitales para tal fin, siendo algunas de sus desventajas el que están destinados para un determinado fin y su funcionamiento ya no puede ser modificado para incluir alguna mejora o la imposibilidad de trasladar un histórico del seguimiento cardíaco de un paciente en caso de que cambiase a otro centro clínico.

Con este trabajo se pretende dar un paso más en el acercamiento entre el usuario y la biotecnología, mediante el diseño de electrocardiograma portátil empleando para ello una FPGA Zynq-7000. Este diseño abarcará desde la recogida de los impulsos eléctricos que provienen del corazón, su posterior conversión al dominio digital y procesado, y la transferencia de estos al sistema operativo que podrá ser ejecutado gracias al procesador que integra la Zynq-7000.

Abstract

Biotechnology is now an important part of our lives. It's a fact that there are hundreds of devices capable of gathering biological information about a person and that this information can be used to serve multiple areas: the medical field, to monitor the vital signs of a patient; everyday life, to check the pulse rate when practicing sports; even biosecurity, where it is possible to identify a subject by their fingerprints or voice.

The heart is one of the many branches of biotechnology and biology. It has been deeply studied and the available information about it is plentiful. However, things get complicated when it comes to creating a cardiac profile to diagnose heart disease. Although some devices can be bought from authorized retailers for that matter, they can't provide enough information to determine the origin and the cause of such a heart condition.

On the other hand, hospitals have medical equipment that can solve the problem of not having enough information but there are disadvantages as well. The medical equipment is made to perform one specific task and can't be modified in case improvements are required. Also, if someone were to move town, having their medical records transferred to their new doctor's office wouldn't be as easy as that person might think.

The purpose of this project is to overcome such difficulties by bringing both the user and biotechnology one step closer. This will be achieved with the design of a portable electrocardiography (EKG) using the Zynq-7000. It will start with the sampling of the electrical impulses of the heart, then it will be converted to digital signals and finally, this data will be transferred to the operating system that will be running thanks to the integrated processor.

Índice

Resumen **ix**

Abstract **xi**

Índice **xii**

Índice de Tablas **xiv**

Índice de Figuras **xvi**

Notación **xviii**

1 Introducción **1**

2 Datos de entrada al sistema **4**
2.1 Origen de los impulsos eléctricos *4*
2.2 Ondas ECG *5*
2.3 Colocación de los electrodos *5*
2.4 Derivaciones *6*

3 Conversión de la señal a digital **9**
3.1 Zynq-7000 *9*
3.1.1 Características 10
3.2 XADC *12*
3.2.1 Características 12
3.2.2 Requisitos para la disposición de pines 12
3.2.3 Funciones de transferencia del convertidor 13
3.2.4 Formas de acceder al XADC 14
3.2.5 Temporización 15
3.2.6 Registros del XADC 16
3.2.7 Modos de operación 18
3.3 ZedBoard *19*
3.3.1 Características 20
3.3.2 Modos de arranque 21
3.3.3 Conector XADC 23

4 Recepción de los datos convertidos **26**

4.1 Xillybus *26*
4.1.1 Características 26
4.1.2 Flujos de datos 27

5 Desarrollo del sistema **28**
5.1 Preparación de la tarjeta SD *28*
5.1.1 Instalando la imagen de Xillinux 28
5.1.2 Modificando devicetree.dtb 30
5.1.3 Generando xillydemo.bit inicial 31
5.2 Preparación del XADC *32*
5.2.1 Configuraciones en la pestaña "Basic" 32
5.2.2 Configuraciones en la pestaña "ADC Setup" 33
5.2.3 Configuraciones en la pestaña "Alarms" 34
5.2.4 Configuraciones en la pestaña "Channel Sequencer" 34
5.2.5 Configuraciones en la pestaña "Summary" 34
5.2.6 Creación del diseño top-level para el XADC 35
5.3 Preparación de Xillybus *38*
5.3.1 Dispositivo xillybus_datastream 40
5.3.2 Dispositivo xillybus_controlstream 41
5.3.3 Generación e inserción del nuevo core 41
5.3.4 Generación de la FIFO para el flujo de datos 42
5.3.5 Implementación de la FIFO y el XADC al top de Xillybus 43
5.3.6 Entradas analógicas en el fichero de restricciones 44

6 Resolución de errores y pruebas **45**
6.1 Pruebas y verificación de resultados *45*
6.1.1 XADC 45
6.1.2 FIFO 48
6.1.3 Xillybus 48
6.1.4 Sistema completo 50
6.2 Resolución de errores *52*
6.2.1 Valores que provienen de un canal inválido del XADC 52
6.2.2 Alta latencia en la actualización de los datos de la FIFO 54

7 Conclusiones y propuestas de mejora **55**

Anexo A: Código del sistema **57**
top_xadc.vhd *57*
xillybus.vhd *59*
xillybus.xdc *64*
tb_top_xadc.vhd *69*
envia_caract.vhd *70*
fifo.vhd *72*
tb_fifo.vhd *74*
leerBinario.py *76*
conversion.m *77*
leerHexadecimal.m *78*

Referencias **79**

ÍNDICE DE TABLAS

Tabla 3–1. Secuencia seguida en el modo por defecto 19
Tabla 3–2. Relación jumpers, boot_mode y MIO para ZedBoard Rev. D 21
Tabla 5–1. Opciones a seleccionar en la pestaña "ADC Setup" del asistente del XADC 34
Tabla 5–2. Pines de las entradas analógicas en la Zynq 44

ÍNDICE DE FIGURAS

Figura 2-1. Onda ECG [1] 5

Figura 2-2. Posición de los electrodos periféricos y precordiales respectivamente [3] 5

Figura 3-1. Configuraciones recomendadas por el fabricante. Izquierda: referencia externa; derecha: referencia interna del chip. 13

Figura 3-2. Función de transferencia unipolar 13

Figura 3-3. Función de transferencia bipolar 14

Figura 3-4. Puertos XADC 15

Figura 3-5. Registros de configuración 17

Figura 3-6. Configuración de jumpers para el modo de arranque con tarjeta SD 22

Figura 3-7. Configuración de jumpers para el modo de arranque con JTAG 23

Figura 3-8. Filtro anti-aliasing para las entradas del XADC 24

Figura 3-9. Pines del conector XADC 24

Figura 4-1. Flujo de datos con FIFO en dirección FPGA-Host 27

Figura 4-2. Flujo de datos con FIFO en dirección Host-FPGA 27

Figura 5-1. Interfaz del programa USB Image Tool 29

Figura 5-2. Asistente de configuración del XADC 32

Figura 5-3. Wrapper del XADC generado por el asistente 35

Figura 5-4. Pantalla inicial de creación de un nuevo core de la IP Core Factory de Xillybus 39

Figura 5-5. Pantalla que muestra el listado de dispositivos para el core generado 40

Figura 5-6. Pantalla de creación de un nuevo dispositivo en la factoría de Xillybus 40

Figura 5-7. Error que se muestra cuando no encuentra xillybus_core.ngc 42

Figura 5-8. Asistente de configuración de la FIFO 42

Figura 6-1. Ventana de simulación del XADC de Vivado 46

Figura 6-2. Monitorización del XADC desde la opción Hardware Manager 47

Figura 6-3. Ventana de simulación de la FIFO en Vivado 48

Figura 6-4. Esquema de *loopback* entre una FIFO y Xillybus 49

Figura 6-5. Diagrama de estados de envia_caract.vhd 50

Figura 6-6. Resultado de ejecutar el comando hexdump para ver los datos de la conversión del XADC 51

Figura 6-7. Resultado de guardar los resultados de la conversión en un fichero, formatearlo con Python y representarlo con Matlab. 52

Figura 6-8. Diagrama de estados de counter.vhd 53

Notación

FPGA	Field Programmable Gate Array
XADC	Xilinx Analog-Digital Converter
IP Core	Intellectual Property Core
FIFO	First In First Out
ECG	Electrocardiograma
AXI	Advanced eXtensible Interface
ARM	Advanced RISC Machine
AMBA	Advanced Microcontroller Bus Architecture
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
DRP	Dynamic Reconfiguration Port
JTAG	Joint Test Action Group
PS	Processing System
PL	Programmable Logic

1 INTRODUCCIÓN

Estando rodeados de dispositivos tecnológicos, se puede llegar a pensar que se ha llegado al límite y que no es posible una mayor integración de la tecnología en la sociedad. Lo cierto es que aún falta bastante para que esto sea del todo cierto, y aunque avanza a pasos agigantados, ciertos aspectos aún siguen manteniéndose como hasta ahora.

En relación a la biotecnología, la salud y el bienestar de la persona, ha habido un auge de los mismo en estos últimos años. Incluso los sistemas operativos móviles empiezan a incluir aplicaciones dedicadas a ello.

Con este trabajo se pretende investigar en la posibilidad de diseñar un electrocardiograma portátil aprovechando la potencia de procesamiento de una FPGA, en concreto, la Zynq-7000. Además, es posible ejecutar un sistema operativo ya que la misma cuenta con las especificaciones técnicas necesarias para hacerlo, lo que abrirá un sinfín de posibilidades acerca de qué hacer con los datos que se obtienen del usuario.

El objetivo del proyecto es, por tanto, comprobar hasta que punto es posible desarrollar un sistema completo, desde que se recibe la señal hasta que se convierte y procesa y está lista para ser enviada al sistema operativo. Para ello se intentará aprovechar al máximo los funcionalidades y características que incluye la FPGA sin recurrir a componentes externos. Con el fin de facilitar el desarrollo y que todo el tiempo se pueda enforcar en el sistema exclusivo necesario para el electrocardiograma, se empleará una placa de evaluación que integra la FPGA mencionada con ciertos añadidos que resultarán de utilidad para el trabajo.

La forma en que se organizará esta memoria será:

- Breve contacto con las señales que provienen del corazón. Cómo son, que información se necesita para obtener el electrocardiograma…
- Descripción de los dispositivos que se emplearán para recoger estas señales y convertirlas al dominio digital (FPGA, convertidor, placa de evaluación).
- Proceso detallado del desarrollo del sistema, desde cómo configurar el convertidor, pasando por la instalación del sistema operativo, hasta conseguir tener el sistema completo y funcional listo para ser usado.
- Pruebas llevadas a cabo y resolución de algunos de los errores que han supuesto una mayor dificultad de cara al avance del proyecto. De esta forma se pretende mostrar la forma de proeceder ante posibles errores que pudieran surgir y qué tipo de pruebas hacer para solucionarlos.

2 DATOS DE ENTRADA AL SISTEMA

La primera etapa del diseño del electrocardiograma portátil es la de la adquisicón de las señales que provienen del corazón. Pero para poder hacer esto adecuadamente, así como el posterior muestreo de dichas señales, es necesario conocer la naturaleza de las mismas.

2.1 Origen de los impulsos eléctricos

El lugar de donde proceden estos impulsos eléctricos es el nodo sinoauricular, estructura situada en la aurícula derecha del corazón. Es ahí donde se originan los impulsos eléctricos que se propagan a las aurículas, provocando su contracción. Los impulsos entonces alcanzan el nodo auriculoventricular, desde donde se irradia a los ventrículos y hace que se contraigan.

En este proceso intervienen los fenómenos de despolarización y repolarización de las células del miocardio, y son estos fenómenos los que un electrocardiograma registra. Mediante la colocación de electrodos en puntos específicos del cuerpo humano, es posible recoger las diferentes variaciones de potencial eléctrico que se producen en el corazón y representarlas en una gráfica tiempo-voltaje. Analizando la gráfica resultante es posible detectar un comportamiento anómalo del órgano principal del aparato circulatorio.

2.2 Ondas ECG

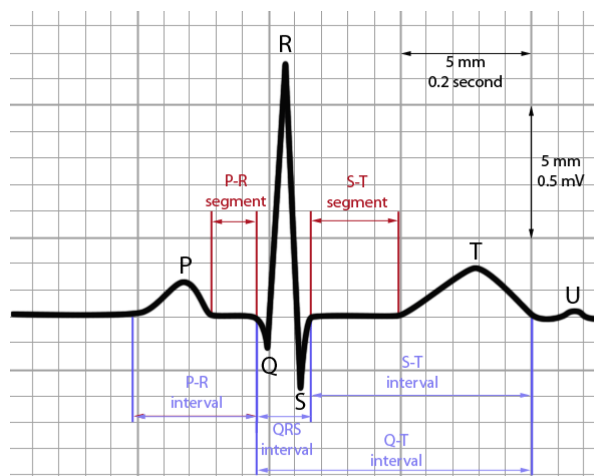

Figura 2-1. Onda ECG [1]

En la Figura 2-1 se observa la forma típica de un electrocardiograma. Está formado por un conjunto de intervalos que contienen ondas y segmentos, siendo estos últimos líneas que unen el final de una onda con otra [2].

- **Onda P**: despolarización auricular.
- **Segmento PR**.
- **Onda QRS** (también conocido como complejo QRS): despolarización ventricular. Aquí también ocurre la repolarización auricular, aunque no es apreciable en el ECG.
- **Segmento ST**.
- **Onda T**: repolarización ventricular.
- **Onda U**: no es apreciable en un ECG real.

2.3 Colocación de los electrodos

Para obtener el electrocardiograma hay que disponer una serie de electrodos sobre la superficie del cuerpo, proceso totalmente indoloro y sin ningún tipo de riesgo, que recogerán información sobre los distintos planos eléctricos del corazón.

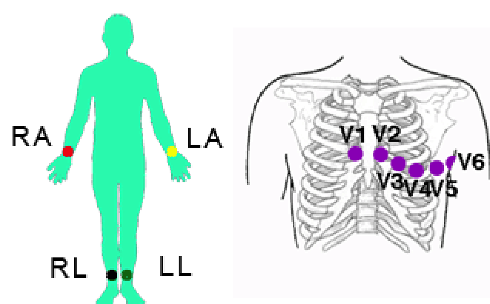

Figura 2-2. Posición de los electrodos periféricos y precordiales respectivamente [3]

Por lo general se suelen utilizar un total de 10 electrodos. Se distingue entre dos tipos:

- **Electrodos periféricos**: son un total de 4 electrodos que se colocan en el brazo derecho, brazo izquierdo, pierna derecha (se suele usar como potencial de referencia) y pierna izquierda.
- **Electrodos precordiales**: se corresponden con los 6 electrodos restantes y debe colocarse en el área precordial.

2.4 Derivaciones

Con estos 10 electrodos se obtienen las 12 derivaciones típicas que son representadas en el electrocardiagrama y que permiten tener una "imagen" del corazón desde diferentes puntos de vista o planos. Una derivación se define como la diferencia de tensión entre un electrodo y otro. Hay dos formas de clasificar las derivaciones y no son excluyentes entre sí:

- **Derivaciones según el potencial medido**:
    - **Derivaciones monopolares**: miden el potencial absoluto en un punto (esto es, la diferencia entre el potencial en ese punto y la referencia).
    - **Derivaciones bipolares**: miden el potencial entre dos puntos (siendo ninguno de ellos la referencia).

- **Derivaciones según la posición del electrodo**:
    - **Derivaciones del plano frontal**: son aquellas obtenidas a partir de los electrodos situados en las extremidades.
    - **Derivaciones precordiales**: se obtienen a partir de los electrodos de la zona precordial.

Aunque es cierto que, en casos especiales, se pueden colocar los electrodos en otras posiciones y obtener derivaciones diferentes. A continuación, se listan las 12 derivaciones comunes:

- I
    - Tipo de derivación: bipolar y de plano frontal.
    - Se corresponde con la diferencia de potencial entre el brazo derecho y el brazo izquierdo.

- II
    - Tipo de derivación: bipolar y de plano frontal.
    - Se corresponde con la diferencia de potencial entre el brazo derecho y la pierna izquierda.

- III
    - Tipo de derivación: bipolar y de plano frontal.
    - Se corresponde con la diferencia de potencial entre el brazo izquierdo y la pierna izquierda.

- aVR
    - Tipo de derivación: monopolar (aumentada) y de plano frontal.

    - Se corresponde con el potencial absoluto del brazo derecho.

- aVL
    - Tipo de derivación: monopolar (aumentada) y de plano frontal.
    - Se corresponde con el potencial absoluto del brazo izquierdo.

- aVF
    - Tipo de derivación: monopolar (aumentada) y de plano frontal.
    - Se corresponde con el potencial absoluto de la pierna izquierda.

- V1
    - Tipo de derivación: monopolar y precordial.

- V2
    - Tipo de derivación: monopolar y precordial.

- V3
    - Tipo de derivación: monopolar y precordial.

- V4
    - Tipo de derivación: monopolar y precordial.

- V5
    - Tipo de derivación: monopolar y precordial.

- V6
    - Tipo de derivación: monopolar y precordial.

La amplitud de estas señales es realmente baja (como se puede observar en Figura 2-1), por lo que sería conveniente amplificarlas antes de pasar a la conversión a digital de las mismas.

3 CONVERSIÓN DE LA SEÑAL A DIGITAL

Para poder analizar la información referente a los impulsos eléctricos del corazón se realiza una conversión a digital. Esto se puede conseguir con un convertidor analógico-digital cuyas características sean suficientes para procesar las señales de entrada, así que aprovechando que la Zynq-7000 incluye un convertidor con infinidad de opciones de configuración, se opta por utilizar este y así reducir al mínimo la cantidad de componentes externos a la FPGA necesarios para llevar a cabo el próposito del proyecto.

Este capítulo está dedicado al estudio de las características de la FPGA, además de profundizar en el convertidor XADC. Por último, se analiza la ZedBoard, una placa de evaluación que integra la Zynq-7000 y la cual ha sido usada para realizar las pruebas de este proyecto.

3.1 Zynq-7000

El dispositivo encargado de procesar los datos que provienen de la conversión a digital pertenece a la familia de Xilinx All Programmable SoC (AP SoC), la Zynq-7000. La característica principal de los AP SoC de Xilinx es que combinan la programabilidad software de un procesador con la programabilidad hardware de una FPGA, dando como resultado un dispositivo con un gran rendimiento, flexibilidad y escalabilidad. Para ello, integran un sistema de procesamiento (PS) el cual emplea un procesador ARM Cortex-A9 MPCore de doble núcleo con una lógica programable (PL) cuyas características varían según los diferentes modelos de la familia Zynq-7000 (aunque todas derivan de la tecnología de FPGAs de Xilinx 7 Series).

Las aplicaciones que puede tener este dispositivo son infinitas, por nombrar alguna de ellas: equipos de visión nocturna, comunicaciones radio o impresoras multifunción. Concretando para este proyecto donde se pretende analizar una gran cantidad de información en tiempo real, este dispositivo es uno de los más adecuados por su alta capacidad de procesamiento. Además, la posibilidad de contar con un sistema de procesamiento que permite recoger los datos de la lógica programable y presentarlos al usuario de forma transparente lo hace ideal para este propósito.

Una descripción detallada y en profundidad de la familia Zynq-7000 se puede encontrar en [4], aunque en el siguiente apartado se recogen algunas de las características más interesantes.

3.1.1 Características

3.1.1.1 Sistema de procesamiento (PS)

Unidad de procesamiento de aplicación (APU)

- Dual ARM Cortex-A9 MPCore.
    - Cuenta con memorias caché L1 de 32 KB para instrucciones y 32 KB para datos.
    - Memoria caché L2 compartida de 512 KB.
    - Temporizadores privados y temporizadores *watchdog* entre otros.
- Registros de control a nivel de sistema (SLCR).
    - Usados para controlar el compartamiento del PS.
- Snoop Control Unit (SCU).
    - Permite mantener la coherencia entre las diferentes memorias caché.
- SRAM (OCM) de 256 KB.
- DMA.
    - Cuenta con cuatro canales para PS y otros cuatro para PL.
- Controlador general de interrupciones (GIC).
- Temporizadores/contadores triples.

Interfaces de memoria

- DDR.
    - Permite transferir datos tanto en la transición de nivel bajo a alto como de alto a bajo de la señal de reloj.
    - Soporta DDR3, DDR3L, DDR2 y LPDDR-2.
    - Cuenta con un planificador de transacciones para optimizar el ancho de banda de datos y la latencia.
- Quad-SPI.
    - Soporte para lectura x1 y x2.
    - Soporte de señal de protección contra escritura.
    - El máximo reloj que admite en *master mode* es de 100 MHz.
    - Soporte para accesos a ROM a través de la interfaz AXI.
    - Static Memory Controller (SMC). Permite que tanto memorias NAND, NOR o SRAM paralelas puedan ser dispositivos primarios de arranque (*primary boot device*).

Periféricos E/S

- Hasta 54 pines de entrada/salida de propósito general.
    - Posibilidad de programar interrupciones usando estos pines.
- Dos controladores para Gigabit Ethernet.
    - Cuentan con la posibilidad de encender de forma remota el dispositivo (Wake on LAN).
    - Cumplen el estándar IEEE 802.3-2008 así como IEEE 1588 revision 2.0.
- Dos controladores para USB.
    - Compatible con la especificación OTG.
    - Funcionan como USB 2.0 de alta velocidad.

- Dos controladores SD/SDIO.
  - Permite la opción de arranque desde tarjeta SD.
  - Tiene su propio DMA.
- Dos controladores SPI (maestro o esclavo).
  - 4-wire SPI: MOSI, MISO, SCLK y SS.
  - Permite transmisión de datos full-duplex.
  - Cuenta con modo maestro (master mode) y modo esclavo (slave mode).
- Dos controladores CAN.
  - Conforme a los estándares ISO 11898 -1, CAN 2.0A, y CAN 2.0B.
  - Soporta tasa de datos hasta 1 Mb/s.
  - *Sleep mode*. Se despierta automáticamente.
- Dos controladores UART.
  - 6, 7 u 8 bits de datos.
- Dos controladores I2C.
  - Soporte para FIFOs de 16 bytes.
  - Cumple con las especificaciones de la versión 2 de I2C.
  - Cuenta con modo maestro (master mode) y modo esclavo (slave mode y slave monitor mode).
- PS MIO
  - Están divididos en dos dominios de tensión. Dentro de cada uno de estos dominios, cada MIO se puede programar de forma independiente.
  - Cuenta con 2 bancos de tensión de E/S: banco 0 (pines del 0 al 15) y banco 1 (pines del 16 al 53).
  - El nivel de tensión se puede programar según el banco (1.8 y 2.5/3.3V).

3.1.1.2 Lógica programable (PL)

Bloques de lógica configurable (CLB)

- Look-up tables (LUT) de 6 entradas.
- Capacidad de memoria dentro de la LUT.
- Sumadores en cascada.
- Funcionalidad para registros y registros de desplazamiento.

RAM de 36 Kb

- Hasta 72 bits de ancho.
- Configurable como RAM dual de 18 Kb.
- Circuitería de corrección de errores.

Procesamiento digiral de señal: DSP48E1.

- Multiplicadores/acumuladores de alta resolución (48 bits).
- Pipeline opcional.
- ALU opcional.

E/S configurables.

- Capacidades de desacoplo de alta frecuencia para una mejor integridad de la señal.
- E/S de alto alcance soportan desde 1.2 a 3.3V.

Transceptores gigabit de bajo consumo.

- Diferentes tasas de transferencia según el modelo concreto dentro de la familia Zynq-7000. Va desde los 6.25 Gb/s hasta los 12.5 Gb/s.
- Modo de bajo consumo.

Convertidor análogico-digital (XADC).

- Conversión de 12 bits a un máximo de 1 MSPS.
- Hasta 17 entradas análogicas configuradas por el usuario.
- Posibilidad de utilizar una referencia de tensión interna o externa.
- Acceso continuo por medio de JTAG a las medidas del ADC.

Bloques de interfaz integrados para diseños con PCI Express.

3.2 XADC

El componente fundamental en el diseño del electrocardiograma será el convertidor analógico-digital que integra la Zynq-7000. Recibirá las señales analógicas como datos de entrada y devolverá la conversión en digital a su salida.

En esta sección se detallarán diversos aspectos del XADC a tener en cuenta de cara al desarrollo del proyecto: características, formas de configuración, registros con los que cuenta, etc. En [5] existe un manual muy completo sobre el XADC en caso de necesitar información más específica sobre alguno de los apartados.

3.2.1 Características

- 12 bits de salida.
- Hasta 1 MSPS de tasa de muestreo.
- Puede acceder hasta 17 canales (1 dedicado y 16 auxiliares) externos de entrada analógica.
- Admite varios modos a la entrada (unipolar, bipolar…).
- Incluye algunos sensores que permiten medir la tensión de alimentación o la temperatura interna del chip.
- 2 ADCs (ADC A y ADC B).

3.2.2 Requisitos para la disposición de pines

Se debe cumplir con una disposición básica de pines o *pinout* (consultar Figura 3-1) para el correcto funcionamiento del convertidor, y para ello se recomiendan dos posibles configuraciones:

1. Usar una referencia externa de 1.25 V.

2. Usar una referencia proporcionada por el propio chip (cuando se requiere monitorización básica de la temperatura o la tensión de alimentación, es buena opción).

Independientemente de la configuración escogida, necesitará además una alimentación de 1.8 V en el pin V_{ccAUX}.

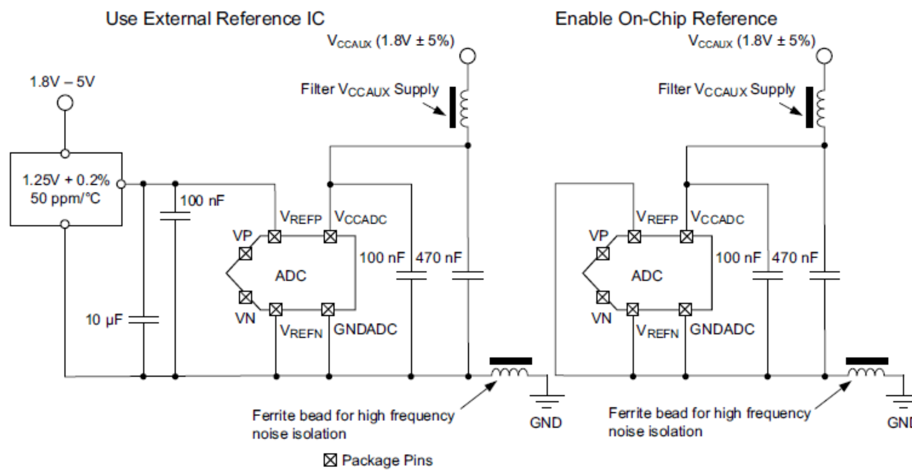

Figura 3-1. Configuraciones recomendadas por el fabricante. Izquierda: referencia externa; derecha: referencia interna del chip.

Cuenta con un canal analógico de entrada diferencial dedicado, V_P y V_N (si no se usan conectar a GND) y 16 canales auxiliares (que pueden ser usados como puertos E/S de propósito general).

Se puede superar V_{CCADC} o ir por debajo de GNDADC en no más de 100 mV o podría causar severos daños al XADC. Para limitar la corriente a 1.0 mA, se recomienda poner una resistencia de al menos 100 Ω en serie con la entrada analógica.

3.2.3 Funciones de transferencia del convertidor

El XADC permite ser configurado en dos modos de funcionamiento según la entrada: unipolar o bipolar.

- **Unipolar**: el rango nominal de entrada permitido va de 0 a 1 V. Producirá el código hexadecimal `000h` cuando en la entrada haya presente 0 V, y para el otro extremo del rango (1 V) devolverá `FFFh`.

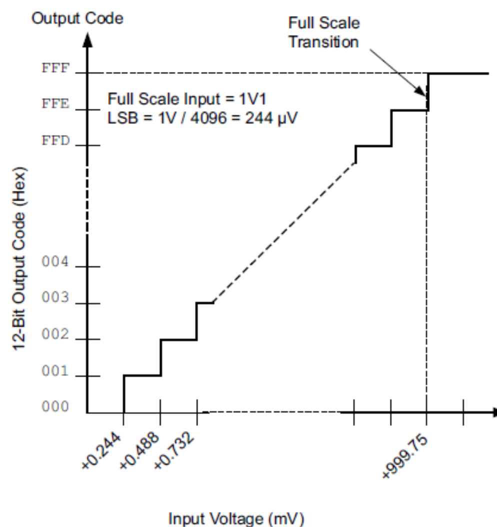

Figura 3-2. Función de transferencia unipolar

- **Bipolar**: permite la entrada de señales diferenciales y bipolares. En este caso la salida estará en complemento a dos (con 12 bits, en complemento a dos, se podrán obtener resultados comprendidos entre -2048 y 2047) para indicar el signo de la señal V_P relativa a V_N.

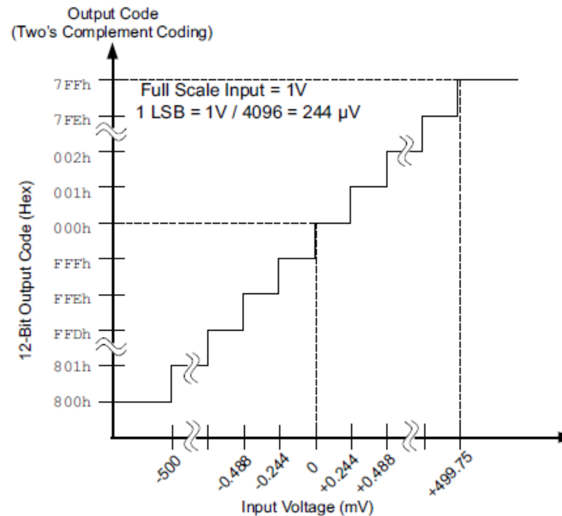

Figura 3-3. Función de transferencia bipolar

En ambas funciones de transferencia se observa que 1 LSB equivale a 244 µV, esto quiere decir que, si se producen cambios menores a 1 LSB, el convertidor no será capaz de apreciarlo.

Es importante resaltar que los ADCs siempre producen como resultado de la conversión 16 bits, siendo los 12 bits más significativos el valor deseada. Los 4 bits restantes se pueden usar para minimizar efectos de cuantización o mejorar la resolución.

3.2.4 Formas de acceder al XADC

3.2.4.1 PS-XADC

La conexión directa entre XADC y PS se hace a través de JTAG. Siempre se encuentra disponible para el PS. Es una interfaz con una tasa de datos baja, pensada para monitorizar los diferentes sensores internos con los que cuenta la Zynq-7000.

3.2.4.2 Periférico AXI

En un SoC normalmente coexisten diferentes elementos o dispositivos: la CPU, memorias, DSP, controlador DRAM, ADC/ACD, USB, etc. La forma de comunicar todos estos dispositivos entre ellos es usando buses. Se conocen diferentes tecnologías de buses para este propósito, siendo uno de ellos AXI.

AXI es parte de ARM AMBA, una familia de buses para controladores. Xilinx implementa la versión del protocolo AXI4, ver [6]. Al dar soporte a este estándar de comunicación como lo es AXI, se abre las puertas al uso de muchas IP que soportan este protocolo, ya no solo específicos de Xilinx sino de ARM en general. La mayoría de IP Cores disponen de un puerto para comunicación usando AXI, como ocurre con el XADC.

3.2.4.3 IP Core

La gestión en este caso se hace a través del **Dynamic Reconfiguration Port** (DRP). El IP Core es instanciado en el diseño y es accesible desde PL.

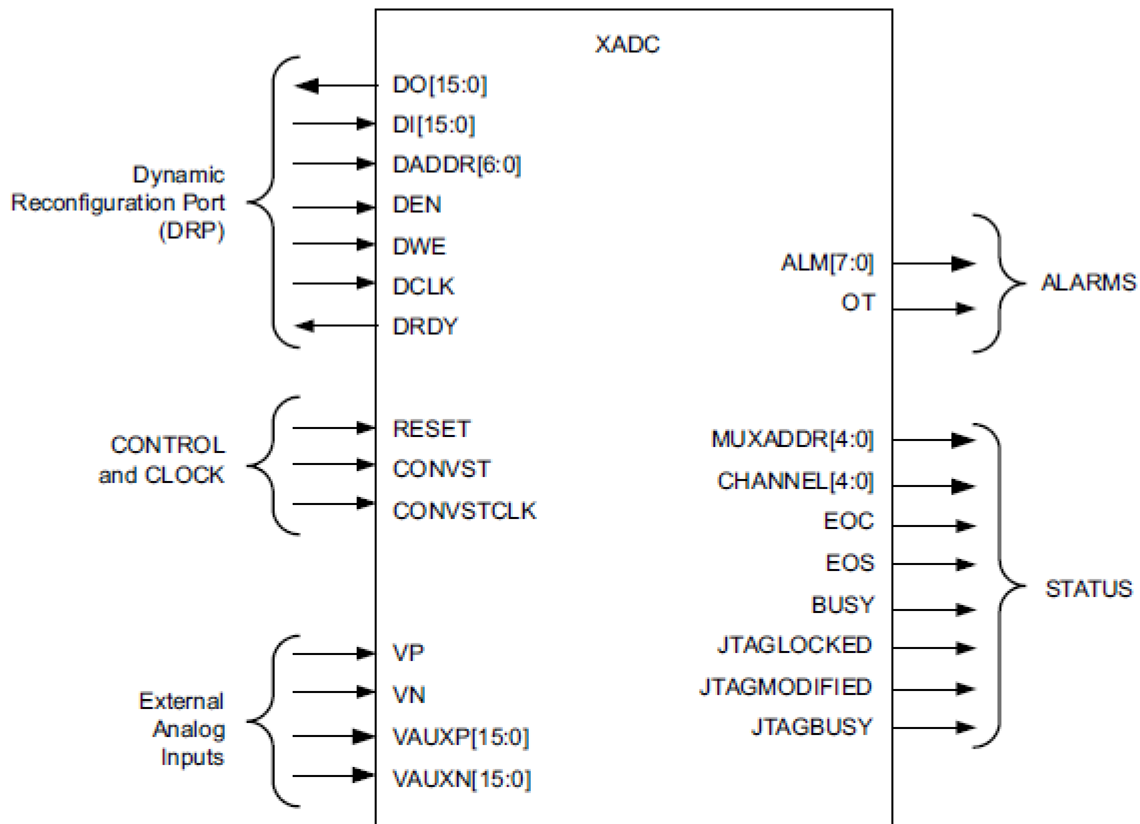

Figura 3-4. Puertos XADC

La manipulación de los diferentes registros del XADC se puede hacer a través de estos 7 puertos.

- **DO[15:0]**. Salida. Este bus de 16 bits es usado por el XADC para colocar los datos que hubiera en un registro que ha sido previamente leído.
- **DI[15:0]**. Entrada. En este bus de 16 bits se colocarán los datos que se desea escribir en cualquiera de los registros (que tengan permiso de escritura).
- **DADDR[6:0]**. Entrada. Es usado tanto si se lee como si se escribe un registro para indicar la dirección del mismo.
- **DEN**. Entrada. Se activa cuando hay una dirección válida en DADDR, para iniciar la lectura de un registro.
- **DWE**. Entrada. Se activa cuando una dirección válida de un registro en DADDR y un dato en DI para realizar la operación de escritura.
- **DCLK**. Entrada. Es la señal de reloj DRP (debe ser proporcionada).
- **DRDY**. Salida. Se activará siempre que haya un dato válido en DO.

3.2.5 Temporización

El XADC está sincronizado con la señal de reloj DRP denominadada DCLK. Este reloj debe ser

proporcionado de forma externa. A partir del DCLK, se obtiene un reloj de menor frecuencia que los ADC usarán internamente, ADCCLK. Este último debe estar entre 1 MHz y 26 MHz para que el XADC funcione.

Existen 2 modos de operación temporal:

- **Muestreo continuo**: automáticamente comienza la siguiente conversión al final del ciclo de conversión actual.
- **Muestreo por evento**: se debe iniciar la siguiente conversión después de que termine el ciclo actual de conversión usando las entradas CONVST o CONVSTCLK.

El proceso de conversión analógico-digital tiene una fase de adquisición y una de conversión.

- **Fase de adquisición**: el ADC adquiere el voltaje del canal de entrada seleccionado. El tiempo que dura esta fase depende de la impedancia del canal de entrada, pues esta fase consiste básicamente en cargar un condensador en el ADC. El XADC permite que se produzca la adquisición del siguiente canal mientras se está aún convirtiendo el canal actual.
- **Fase de conversión**: esta fase dura un total de 22 ADCCLK. Cuando el modo de secuenciador automático de canales está activo, el XADC cuenta con una salida, CHANNEL[4:0], que indica el canal que se está convirtiendo.

3.2.6 Registros del XADC

Todos los registros del XADC son accesibles a través del DRP (Dynamic reconfiguration port) o el JTAG TAP. El DRP permite acceder a un total de 128 registros (DADDR[6:0]) de 16 bits cada uno: las 64 primeras direcciones (desde 00h a 3Fh) son de solo lectura y contienen información de los datos de medida del ADC. Se conocen como registros de estado o *status registers*. De la dirección 40h a la 7Fh se encuentran los registros de control (que se pueden leer y escribir).

Las direcciones más interesantes de los registros de estado son:

- **03h**: resultado de la conversión A/D de VP / VN
- **De la 10h a la 1Fh**: resultado de la conversión A/D de los 16 canales auxiliares
- **09h** y **31h**: offset de ADC A y ADC B, respectivamente
- **0Ah** y **32h**: ganancia de ADC A y ADC B, respectivamente

3.2.6.1 Coeficiente de calibración

El XADC puede calibrar digitalmente cualquier error de offset y ganancia, generando un coeficiente de corrección para ambos.

- Los coeficientes de offset son los 12 bits más significativos y están expresados en LSB (es decir, si hay almacenado un 8, serán 8 LSBs). Usan complemento a dos.
- Los coeficientes de ganancia son 7 bits, siendo el bit más significativo un indicador de signo, y están en la parte menos significativa de los 16 bits.
    - Si el bit 7 es un 1: factor de corrección positivo.
    - Si el bit 7 es un 0: factor de corrección negativo.

Con 6 bits para indicar la magnitud y siendo el valor máximo posible 63 (`3Fh`), se pueden corregir errores en el rango ±0.1% x 63, básicamente 6.3%.

3.2.6.2 Registros de configuración

En cuanto a los 32 registros de control accesibles desde el DRP en el rango de direcciones `40h` a `5Fh`, los 3 primeros registros configuran el modo de operación de XADC. Se conocen como Config Reg #0 (`40h`), Config Reg #1 (`41h`) y Config Reg #2 (`42h`).

DI15	DI14	DI13	DI12	DI11	DI10	DI9	DI8	DI7	DI6	DI5	DI4	DI3	DI2	DI1	DI0	
CAVG	0	AVG1	AVG0	MUX	B$\overline{U}$	E$\overline{C}$	ACQ	0	0	0	CH4	CH3	CH2	CH1	CH0	Config Reg #0 DADDR[6:0] = `40h`
SEQ3	SEQ2	SEQ1	SEQ0	ALM6 (Note1)	ALM5 (Note1)	ALM4 (Note1)	ALM3	CAL3	CAL2	CAL1	CAL0	ALM2	ALM1	ALM0	OT	Config Reg #1 DADDR[6:0] = `41h`
CD7	CD6	CD5	CD4	CD3	CD2	CD1	CD0	0	0	PD1	PD0	0	0	0	0	Config Reg #2 DADDR[6:0] = `42h`

X17030-051616

Figura 3-5. Registros de configuración

Registro de configuración 0

- **Bit DI0 a DI4**: CH4 a CH0. Se utilizan cuando el XADC está en modo canal único o multiplexador externo y permiten seleccionar el canal de entrada.
- **Bit DI8**: ACQ. Aumenta el tiempo de establecimiento en las entradas analógicas externas en 6 ciclos de ADCCLK.
- **Bit DI9**: EC. Permite seleccionar el modo de muestreo. Un 1 activa el modo de muestro por evento, mientras que un 0 pone al ADC en modo de muestro continuo.
- **Bit DI10**: B$\overline{U}$. Se utiliza en modo canal único para seleccionar el modo de operación de las señales de entrada al ADC. Un 1 supone el modo bipolar, un 0 se corresponde con el modo unipolar.
- **Bit DI11**: MUX. Si este bit vale 1 se activará el modo de multiplexador externo.
- **Bit DI12 a DI13**: AVG0 a AVG1. Indican la cantidad de muestras para calcular el promedio en los canales seleccionados (se utilizan tanto para el modo de canal único como para el modo de secuenciador automático de canales).
- **Bit DI15**: CAVG. Sirve para desactivar el promediado a la hora de calcular los coeficientes de calibración. Por defecto está activo (esto equivale a un 0). Para desactivar, es necesario poner el bit CAVG a 1. El promediado está prefijado a 16 muestras.

Registro de configuración 1

- **Bit DI0**: OT. Se utiliza para desactivar la señal de altas temperaturas. Se desactiva poniendo este bit a 1.
- **Bit DI1 a DI3, DI8**: ALM0 a ALM3. Se utilizan para desactivar la salida de las diferentes alarmas individualmente. De forma respectiva, colocando un 1 en el bit correspondiente se desactivarían las alarmas de temperatura, V_{CCINT}, V_{CCAUX} y V_{CCBRAM}.
- **Bit DI4 a DI7**: CAL0 a CAL3. Se utilizan para activar los coeficientes de calibración del ADC y las medidas de los sensores del chip. Un 1 lo activa, mientras que un 0 resuelta en su desactivación.
- **Bit DI9 a DI11**: ALM4 a ALM6. Se utilizan para desactivar la salida de las diferentes alarmas individualmente. De forma respectiva, colocando un 1 en el bit correspondiente se desactivarían las alarmas de V_{CCPINT}, V_{CCPAUX} y V_{CCO_DDR}.
- **Bit DI12 a DI15**: SEQ0 a SEQ3. Activan la función del secuenciador de canales. La asignación de bits se verá de forma detallada en los modos de operación del convertidor.

Registro de configuración 2

- **Bit DI4 a DI5**: PD0 a PD1. Se utilizan para apagar el XADC. Si ambos bits valen 1, el bloque del XADC se apaga por completo. Si PD1 es 1 y PD0 es 0, solo apagará el ADC B.
- **Bit DI8 a DI15**: CD0 a CD7. Permiten escoger por cuánto se dividirá el reloj DRP (DCLK), división que dará lugar al reloj ADC (ADCCLK).

Además de estos, existen una serie de registros de configuración específicos para cuando el XADC se encuentra en modo de secuenciador automático de canales:

- **Selección de canal** (direcciones `48h` y `49h`): estos registros activan (1) y desactivan (0) un canal en el secuenciador. El bit 11 del registro `48h` activa o desactiva las entradas analógicas dedicadas. Los bits del 0 al 15 en el registro `49h` activan o desactivan los auxiliares.
- **Promediado** (`4Ah` y `4Bh`): activan o desactivan el promediado. (Misma asignación de bits que en el caso anterior).
- **Modo de la entrada analógica** (`4Ch` y `4Dh`): un 0 es unipolar y un 1 es bipolar. (Misma asignación de bits que en el caso anterior).
- **Tiempo de establecimiento** (`4Eh` y `4Fh`): por defecto, el tiempo de establecimiento para un canal externo en modo muestreo continuo es 4 ciclos de ADCCLK. Se puede extender a 10 poniendo un 1 en la posición correspondiente al canal deseado. Misma asignación de bits que en casos anteriores. (El tiempo de establecimiento es el tiempo de adquisición adicional después del final de una conversión). NOTA: Para el modo de canal único esto se modifica a través del bit ACQ en el Config Reg #0.

3.2.7 Modos de operación

El XADC cuenta con 3 modos de funcionamiento principales: el modo de canal único, modo de secuenciador automático de canal y modo de multiplexador externo. Cada uno de estos modos requiere de una configuración concreta en diferentes bits de los registros de configuración.

- **Modo de canal único** (*Single channel mode*): cuando los bits de secuenciación SEQ3:SEQ0 del Config Reg #1 tienen el valor `0011b`, se activa este modo. En este caso se tiene en cuenta lo que haya en los bits CH4:CH0 del Config Reg #0. También se tiene en cuenta lo que haya en el bit $B\bar{U}$ en el mismo registro de configuración que CH4:CH0. Cuando el bit $B\bar{U}$ es 1, el ADC está en modo bipolar; pero si vale 0 entonces es unipolar.
- **Modo de secuenciador automático de canal** (*Automatic channel sequencer mode*): usado para monitorizar más de un canal. El secuenciador automáticamente selecciona el siguiente canal a convertir. Se utilizan los registros de las direcciones `48h` a `4Fh` para su configuración. Este modo a su vez tiene distintos modos de secuenciación configurables a través de los bits SEQ3:SEQ0 del Config Reg #1:
    - **Modo por defecto** (0000 ó 11xx): en este modo el XADC monitoriza todos los sensores internos (temperatura, tensión, etc.) y almacena la conversión en los registros de estado. Lo hace en un orden especifico (consultar Tabla 3–1).
    - **Single Pass Mode** (0001): solo realiza la secuencia una vez (la indicada en los registros `48h` y `49h`) y cuando termina pasa del modo secuenciador automático de canal al de canal único, convirtiendo únicamente lo que haya en el canal seleccionado en el Config Reg #0 con los bits CH4:CH0. Al pasar al modo de canal único, se escribe en SEQ3-SEQ0: 0011 (desactiva el secuenciador). Se puede volver a activar escribiendo de nuevo los bits 0001 en SEQ3-SEQ0.

- **Modo de secuencia continua** (0010): funciona igual que el anterior solo que la secuencia se reinicia una vez termina y continua así indefinidamente mientras está activo el modo.
- **Modo de muestreo simultáneo** (01xx): automáticamente secuencia a través de 8 pares de entradas auxiliares (útil cuando se quiere preservar la relación de fase entre dos señales). Los canales V_{AUX0} hasta V_{AUX7} se asocian al ADC A y los V_{AUX8} hasta V_{AUX15} se asocian con ADC B. Un canal A (entiéndase como canal A a cualquiera de los canales del 0 al 7) y un canal B (cualquier canal del 8 al 15) son siempre muestreados y convertidos al mismo tiempo. Es posible tener un canal A en unipolar y su pareja en canal B como bipolar pero el tiempo de establecimiento debe ser el mismo para el par de canales. Si se configura un canal A con 10 ADCCLK, entonces su pareja B también lo tendrá. La secuencia se hará según el registro `49h` (solo sirven los 8 primeros bits pues, al estar emparejados el 0 con el 8, el 1 con el 9, etc. no son necesarios el resto).
- **Modo de ADC independiente** (10xx): en este caso los 2 ADC tienen diferentes funciones.
  - ADC A: se pone en "modo monitorización" (en este caso las alarmas están activas), muy parecido al modo por defecto.
  - ADC B: solo se puede usar para las entradas analógicas externas (el dedicado y los auxiliares).

  Este modo libera el segundo ADC para uso propio mientras se monitoriza la FPGA. Para la secuencia en este caso solo se tiene en cuenta el bit 11 (V_P/V_N) del registro `48h` y todos los del `49h` (V_{AUX}).

- **Modo de multiplexador externo** (*External multiplexer mode*): Se activa poniendo a 1 el bit MUX del Config Reg #0. Un bus de direcciones de salida permite controlar el multiplezador externo, MUXADDR[4:0] (el cual refleja el canal que se está adquiriendo en ese preciso instante y cambia de valor tan pronto como el XADC entra en la fase de adquisición). La salida del multiplexador se puede conectar tanto a V_P/V_N como a cualquier V_{AUX}. Habrá que especificar a cuál se conecta en los bits CH4:CH0. Es posible evitar usar todos los canales auxiliares de la FPGA y usarlos a través de un multiplexador. (Se puede usar el muestro simultáneo).

Tabla 3–1. Secuencia seguida en el modo por defecto

Orden	Canal (Dirección)
1	Calibración (`08h`)
2	V_{CCPINT} (`0Dh`)
3	V_{CCPAUX} (`0Eh`)
4	V_{CCO_DDR} (`0Fh`)
5	Temperatura (`0Fh`)
6	V_{CCINT} (`01h`)
7	V_{CCAUX} (`02h`)
8	V_{CCBRAM} (`06h`)

3.3 ZedBoard

Para realizar las pruebas de las partes del proyecto que requieren del uso de la Zynq, se ha empleado una placa

de evaluación que cuenta con una gran cantidad de periféricos y expansiones, la ZedBoard. Esto permite directamente efectuar las simulaciones en un entorno físico y real sin necesidad de hacer el conexionado manual de la FPGA con el resto de periféricos que se quisieran utilizar.

El procedimiento básico para utilizar la placa consiste en conectarla a la alimentación y programar la FPGA. Para ello existen diferentes opciones, siendo una tarjeta SD la elegida en este caso. Con esto se consiguen dos cosas: por un lado, en la tarjeta irá el fichero bitstream que permitirá programar la FPGA, y por otro tendrá montada una imagen de Xillinux que será el sistema operativo con el que se realizan las pruebas de recepción de datos.

3.3.1 Características

Xilinx Zynq-7000 (Modelo XC7Z020)

- QSPI Flash como configuración primaria.
- JTAG en cascada o tarjeta SD como opciones de configuración auxiliar.

Memoria

- 512 MB DDR3 (128 x 32).
- 256 Mb QSPI Flash.

Interfaces

- USB-JTAG.
- 10/100/1G Ethernet.
- USB OTG 2.0.
- Tarjeta SD.
- USB-UART.
- 5 cabeceras Pmod (1 PS, 4 PL).
- LPC FMC (FPGA Mezzanine Card).
- Una cabecera AMS (Agile Mixed Signaling).
- Dos botones de reset (1 PS, 1 PL).
- 7 pulsadores (2 PS, 5 PL).
- 8 interruptures (PL).
- 9 LEDs (1 PS, 8 PL).
- LED para indicar el estado “Terminado” (PL).

Osciladores

- 33.333 MHz (PS).
- 100 MHz (PL).

Audio/Gráficos

- Salida HDMI.

- VGA.
- Pantalla OLED de 128x32.
- Audio: Line-in, Line-Out, auriculares y micrófono.

Alimentación

- Interruptor de apagado/encendido.
- 12V @ 5A.

3.3.2 Modos de arranque

Es posible configurar la ZedBoard para que, según la disposición de una serie de jumpers, el modo de arranque sea: JTAG, Quad-SPI o tarjeta SD. Los dos modos de mayor utilidad en este proyecto serán el de arranque desde la tarjeta SD y el de JTAG, por lo que se incluye un pequeño esquema acerca de cómo posicionar los jumpers para arrancar en uno de estos dos modos.

Tabla 3–2. Relación jumpers, boot_mode y MIO para ZedBoard Rev. D

Jumper	Boot_mode	MIO
JP8	Boot_mode(1)	MIO(3)
JP9	Boot_mode(2)	MIO(4)
JP10	Boot_mode(0)	MIO(5)
JP7	Boot_mode(3)	MIO(2)
JP11	Boot_mode(4)	MIO(6)

3.3.2.1 Tarjeta SD

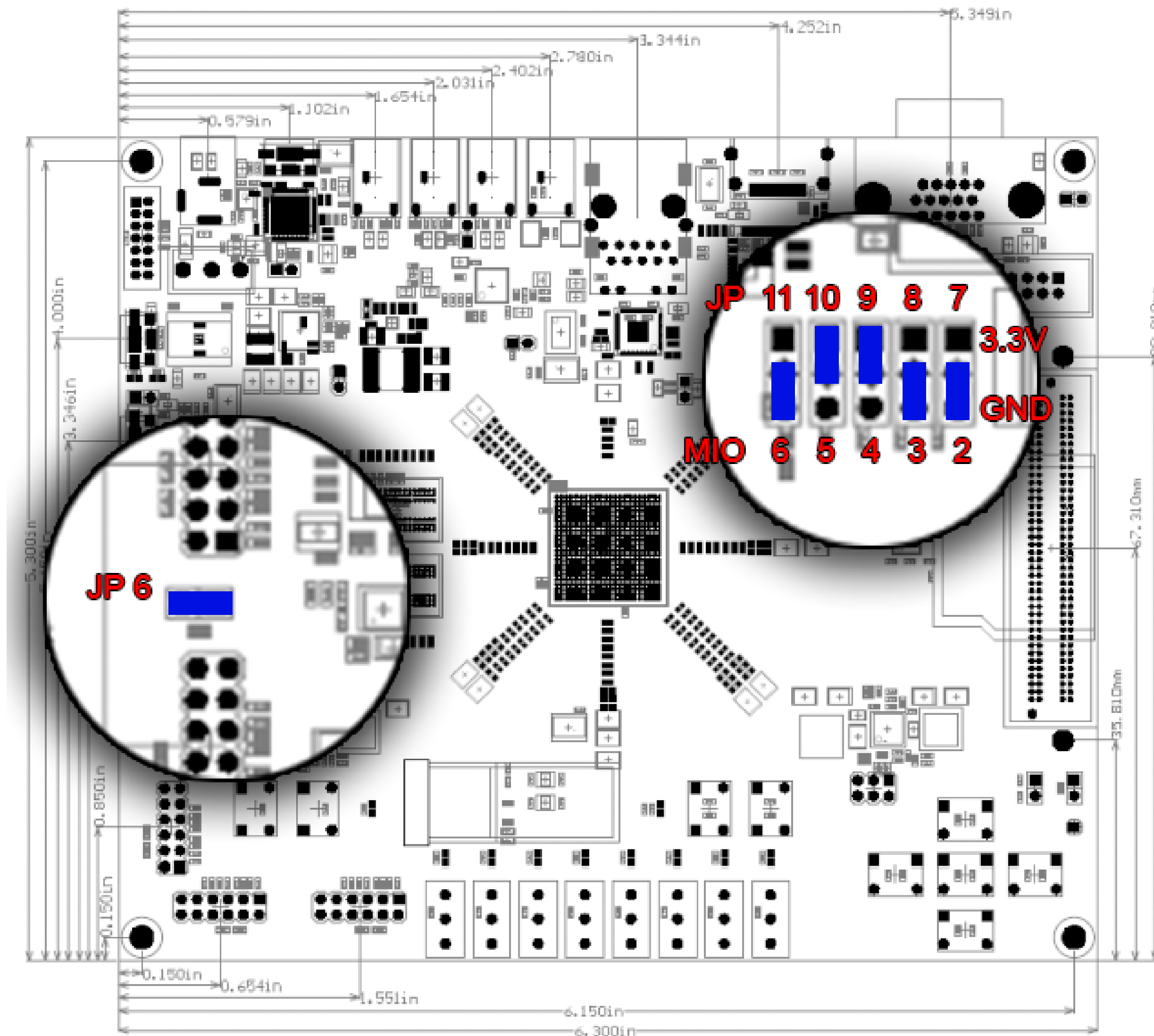

Figura 3-6. Configuración de jumpers para el modo de arranque con tarjeta SD

Se usará este modo cuando se quiera cargar un sistema operativo además del fichero bitstream. El modo de arranque desde tarjeta SD es el que se corresponde con los jumpers JP9 y JP10 conectados a 3.3V y el resto a GND. Además, hay que tener en cuenta que para este modo el JP6 debe estar cortocicuitado, de esta forma la FPGA podrá ver la tarjeta SD.

3.3.2.2 JTAG

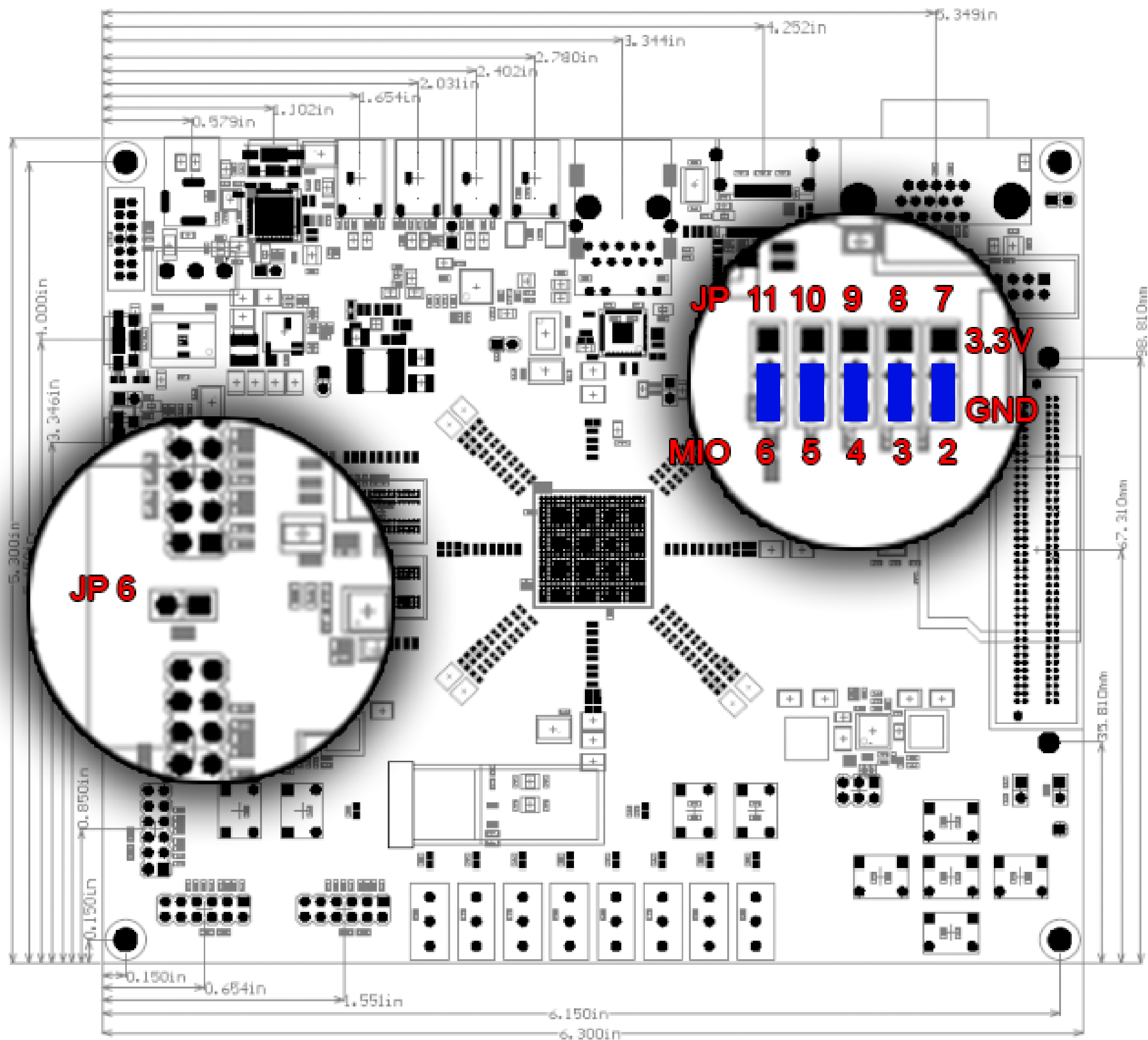

Figura 3-7. Configuración de jumpers para el modo de arranque con JTAG

Este modo se usa para programar la FPGA sin necesidad de usar ningún sistema operativo. Para ello habrá que ajustar los jumpers del 7 al 11 a GND, y dejar abierto el JP6. De esta forma, mediante un USB conectado al puerto PROG se cargará el fichero bitstream y el LED DONE se encenderá cuando haya terminado de programarla.

3.3.3 Conector XADC

La única forma de acceder al XADC de la Zynq-7000 en la ZedBoard es a través del conector o cabecera que se incluye en la placa. Cuenta con un total de 20 pines. Es posible probar el XADC conectando esta cabecera de la ZedBoard directamente a la Xilinx AMS Evaluation Card.

Figura 3-8. Filtro anti-aliasing para las entradas del XADC

De los 17 canales de entrada analógicos con los que cuenta la Zynq-7000 (1 dedicado y 16 auxiliares), solo 3 de ellos están disponibles para usar en la ZedBoard. Estos son V_P/V_N, V_{AUX0} y V_{AUX8}. Estas entradas cuentan con filtros anti-aliasing [7].

Figura 3-9. Pines del conector XADC

Una configuración básica del conector XADC para probar el convertidor implicaría utilizar únicamente los pares V_P/V_N y AGND. La razón de esto es, en primer lugar, V_P/V_N es el único canal dedicado para funcionar como entrada analógica, y como tal, este pin siempre será accesible y estará disponible para usar por el XADC sin necesidad de instanciarlo en el diseño. Por otro lado, a pesar de las configuraciones recomendadas en la sección 3.2.2, no será necesario alimentar al XADC con 1.8 V ni proporcionar una referencia de 1.25 V pues la ZedBoard se encarga de hacer esto internamente (debido a la forma en que ha sido diseñada). Por tanto, los pines que se observan de AV_{DD} así como V_{REF} en la Figura 3-9, son de salida (para su uso con otras tarjetas compatibles como se comentaba al comienzo de esta misma sección). El fabricante numera los pines de arriba abajo y de derecha a izquierda, siendo V_N el número 1 y V_P el número 2. La fila superior se corresponde con los pines impares y la inferior con los pares.

4 RECEPCIÓN DE LOS DATOS CONVERTIDOS

Uno de los principales atractivos de la Zynq-7000 es la posibilidad de usar lógica programable y un sistema operativo en un mismo dispositivo. No existe limitación alguna en el sistema que deba o pueda usar la Zynq-7000, por lo que existen diferentes opciones a escoger. Por ejemplo, la placa de evaluación ZedBoard incluye una tarjeta SD con una imagen preinstalada de Linux. También existen otras opciones como PetaLinux [8] o en el caso de un RTOS, VxWorks [9]. Pero para el propósito de este proyecto, el sistema operativo elegido es Xillinux.

Xillinux es es una distribución de Linux, más concretamente basado en Ubuntu LTS 12.04, para ZedBoard. Este sistema operativo incluye un controlador que forma parte del conjunto de Xillybus y, por tanto, está preparado para funcionar de forma óptima haciendo uso del mismo. Xillybus engloba un módulo FPGA de propiedad intelectual (FPGA IP Core) así como el controlador anteriormente mencionado. Es, consecuentemente, Xillybus el encargado de hacer posible la comunicación entre FPGA y sistema operativo.

4.1 Xillybus

4.1.1 Características

- Compatible con FPGAs de:
    - Xilinx (Virtex-5T, Spartan-6T, Virtex-6T, todas las FPGA Series 7, todas las FPGAs Ultrascale y Ultrascale+).
    - Altera (Cyclone, Arria y HardCopy).
- PCI Express, AXI3 y AXI4.
- Ancho de banda de hasta 3.5 GB/s simultáneamente en las dos direcciones (Host-FPGA y FPGA-Host).
- Sistemas operativos soportados: Linux (versiones superiores a 2.6.36) y Microsoft Windows 7, 8 y 10 (versiones de 32 y 64 bits).
- La descarga de Xillybus incluye el driver necesario tanto para Linux como para Windows.
- Aplicaciones típicas: adquisición de datos, interfaz con hardware, periféricos personalizados, verificación lógica a nivel hardware o coprocesamiento, entre otros [10].

- Consumo de lógica: 110 LUTs/flujo de datos + 2500 LUTs (estimado para Xilinx) + un pequeño número de memorias RAM.

4.1.2 Flujos de datos

El objetivo de Xillybus es hacer posible la comunicación entre la lógica programable (PL) y el sistema de procesamiento (PS). Para ello, actuará de intermediario en el proceso y se establecerán dos flujos de comunicación: uno en la dirección FPGA-Host y otro para Host-FPGA. La interacción con Xillybus no tendrá por qué ser directa, existiendo la posibilidad de utilizar FIFOs que serán leídas/escritas cuando sea conveniente.

4.1.2.1 Flujo FPGA-Host

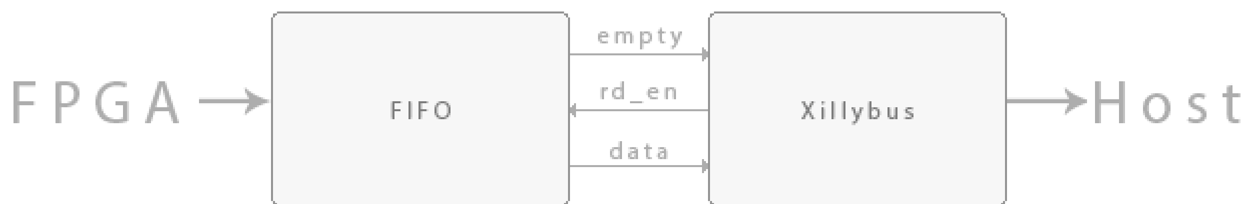

Figura 4-1. Flujo de datos con FIFO en dirección FPGA-Host

En este caso la aplicación de la lógica programable desea enviar información al host. Para conseguir esto, tendrá que escribir toda la información en la FIFO que aparece en la Figura 4-1 para que Xillybus, mediante las señales “empty”, “rd_en” y “data” pueda vaciar el contenido de la misma. Esto libera al programador de la FPGA de la preocupación acerca de cuándo Xillybus estará disponible para recibir esos datos, ya que este se encargará de ir leyendo la FIFO cuando no tenga otras tareas que atender, mientras que por el otro lado el programa de la lógica programable va rellenándola con información. Se consigue así la independencia entre la aplicación FPGA y el host, cada uno escribe o lee en el momento que pueda hacerlo sin depender del otro.

4.1.2.2 Flujo Host-FPGA

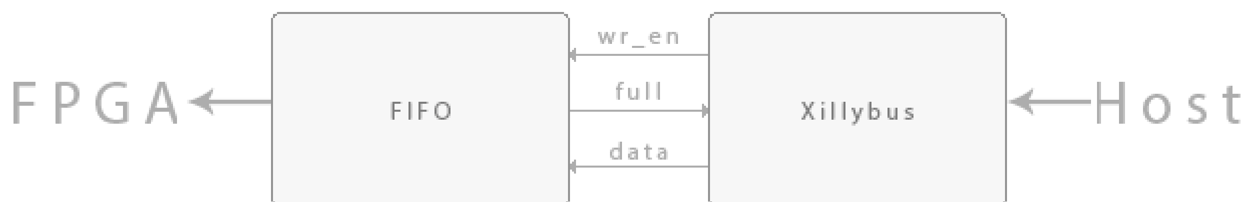

Figura 4-2. Flujo de datos con FIFO en dirección Host-FPGA

La situación opuesta ocurre cuando el host desea comunicar algo a la aplicación FPGA. El funcionamiento es análogo a la sección anterior, solo que intervienen señales diferentes entre la comunicación de Xillybus y la FIFO. En este caso se usarán “wr_en”, “full” y “data”. Cuando Xillybus tiene algo que escribir a la FIFO, pone los datos en “data” y activa el bit “wr_en” para hacer efectiva la escritura. Si el bit “full” está activo, Xillybus no podrá escribir en la FIFO por estar llena.

5 DESARROLLO DEL SISTEMA

Este capítulo explicará paso a paso cómo poner en marcha la ZedBoard de forma que esté lista para realizar la conversión de analógico a digital y llevar estos datos al sistema operativo instalado en una tarjeta SD. Se empezará explicando como montar la imagen de Xillinux, las modificaciones pertinentes al mismo, para luego proceder con la configuración personalizada de Xillybus, el XADC y finalmente la generación del fichero bit que programará toda la lógica de la FPGA.

5.1 Preparación de la tarjeta SD

5.1.1 Instalando la imagen de Xillinux

Para montar la imagen del sistema operativo en la tarjeta SD y probar su funcionamiento se necesita [11]:

- **Tarjeta SD**. Entre el material que incluye la ZedBoard, hay una tarjeta SD de 4 GB que sirve perfectamente. Cualquier otra tarjeta SD de 4 GB o más es válida también, siempre que se encuentre formateada en FAT32.
- **La imagen de Xillinux 1.3**. Esta puede ser descargada desde la web oficial en http://xillybus.com/downloads/xillinux-1.3.img.gz
- **Vivado 2014.4 o versiones posteriores**. Concretamente para el desarrollo de este proyecto se ha utilizado Vivado 2016.4. Es necesario una licencia para utilizarlo. Servirá para generar el fichero bit con toda la lógica, entre otros.
- **Algún programa para montar la imagen en la SD**. Se recomienda USB Image Tool (Windows).
- **El kit de ficheros para la partición de arranque de la placa que se use**. En este caso, como se emplea la ZedBoard, el enlace es el siguiente http://xillybus.com/downloads/xillinux-eval-zedboard-2.0a.zip. Hay otros enlaces para otras placas diferentes a la ZedBoard, disponible en la página oficial de Xillybus.
- **Un compilador de ficheros DTB**.

El primer paso será descargar la imagen de Xillinux y arrancar USB Image Tool (habiendo previamente

insertado la tarjeta SD en el lector de tarjetas del ordenador).

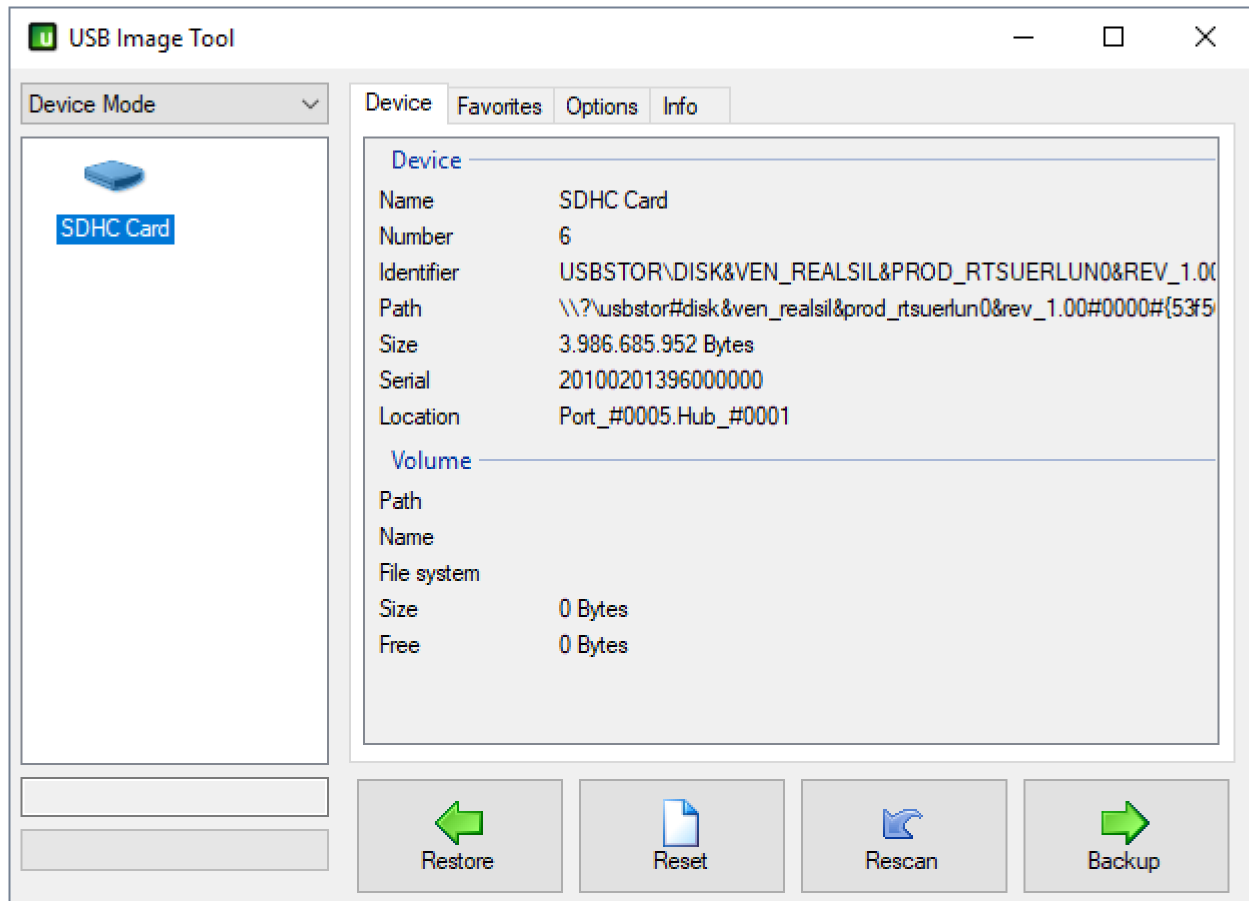

Figura 5-1. Interfaz del programa USB Image Tool

Teniendo seleccionado "**Device Mode**" en el desplegable que aparece en la esquina superior izquierda, seleccionar del listado de dispositivos conectado el correspondiente con la tarjeta SD. Lo siguiente será pulsar sobre el botón "**Restore**", lo que abrirá una ventana para seleccionar la imagen de Xillinux descargada anteriormente (xillinux-1.3.img.gz). Si no se mostrase el fichero, poner como tipo de archivos en la ventana de selección de ficheros "Compressed (gzip) image files". La imagen empezará a copiarse a la SD, proceso que tomará unos minutos. Al finalizar, retirar la tarjeta SD de forma segura.

Esto habrá creado dos particiones en la tarjeta SD: la primera, a la que no se podrá acceder (Windows o bien no la muestra, o si lo hace pedirá formatearla por estar en un formato desconocido. A pesar de que diga esto, nunca pulsar en formatear) y la segunda, que será la partición de arranque, de tamaño muy reducido en la que se deberán colocar 4 ficheros para que funcione correctamente al ponerla en la ZedBoard.

Estos cuatro ficheros son:

- **boot.bin**: el bootloader inicial. Contiene una serie de inicializaciones necesarias para poner en marcha el procesador y difiere según la placa usada.
- **devicetree.dtb**: contiene información hardware para el kernel de Linux.
- **uImage**: binario del kernel de Linux.
- **xillydemo.bit**: el fichero para programar la lógica de la FPGA.

Los dos primeros se incluyen en el kit de ficheros para la partición de arranque (concretamente en el directorio /bootfiles), el tercero es el único archivo que habrá al abrir la tarjeta SD tras haber montado la imagen de

Xillinux (esto quiere decir que no hace falta buscarlo pues ya estará ahí) y el cuarto habrá que generarlo usando Vivado.

El fichero boot.bin no requiere de ningún tipo de modificación, por lo que puede copiarse ya junto al uImage existente en la partición de arranque. Sin embargo, devicetree.dtb sí requiere alterar su contenido antes de copiar a la SD y xillydemo.bit necesita ser generado.

5.1.2 Modificando devicetree.dtb

Un árbol de dispositivos o *device tree*, es un tipo de estructura de datos que se usa para describir hardware. Contiene una serie de nodos con sus propiedades, las cuales son un conjunto clave-valor. Un nodo, además de propiedades, puede tener nodos hijo. Este fichero es utilizado por el kernel de Linux para configurar el hardware y algunos periféricos.

El devicetree.dtb que incluye por defecto el kit, cuenta con una entrada para el XADC que crea un conflicto a la hora de realizar la comunicación entre FPGA-Host usando Xillybus, así que hay que editarlo y eliminar dicha entrada. Pero este fichero no puede ser modificado directamente pues está en binario, y por tanto es necesario el uso de un compilador.

En Linux/Ubuntu (no tiene que ser Xillinux, puede ser en otro equipo), ejecutar el siguiente comando en el terminal:

```
sudo apt install device-tree-compiler
```

Esto descargará un compilador que nos permitirá convertir el fichero .dtb en un .dts, el cual sí puede ser modificado en cualquier editor de texto, y una vez editado se usará el mismo programa para compilar el .dts en un nuevo .dtb.

La forma de usar el compilador es muy sencilla, el comando básico tiene esta estructura:

```
dtc -I <formato_entrada> -O <formato_salida> -o <fichero_salida>
<fichero_entrada>
```

Situar el devicetree.dtb que incluye el kit por defecto (inicialmente se encuentra en la carpeta /bootfiles del ZIP descargado de la web de Xillybus) en el directorio que se desee y ejecutar la siguiente línea:

```
dtc -I dtb -O dts -o devicetree.dts devicetree.dtb
```

Esto generará un fichero devicetree.dts que se procederá a abrir en un editor de texto para buscar el siguiente fragmento de código:

```
ps7-xadc@f8007100 {
    clocks = <0x3 0xc>;
    compatible = "xlnx,ps7-xadc-1.00.a";
    interrupt-parent = <0x2>;
    interrupts = <0x0 0x7 0x4>;
    reg = <0xf8007100 0x20>;
};
```

Una vez encontrado, eliminar por completo y guardar los cambios. Con esto, solo falta volver a compilar el .dts en .dtb, ejecutando la siguiente línea en el terminal:

```
dtc -I dts -O dtb -o devicetree.dtb devicetree.dts
```

Esto sustituirá el devicetree.dtb que hubiese en el directorio con el nuevo. Ahora sí estaría listo para ser copiado a la partición de arranque de la tarjeta SD junto con el fichero uImage que había inicialmente y el boot.bin que se copió en la sección anterior.

5.1.3 Generando xillydemo.bit inicial

El kit de ficheros para la partición de arranque es un ZIP que contiene una carpeta principal llamada **xillinux-eval-zedboard-2.0a**. Se recomienda copiar esta carpeta completa al espacio de trabajo (workspace) habitual utilizado para Vivado y renombrarla a algo más simple como "xillydemo" (el nombre es totalmente libre).

El siguiente paso es abrir Vivado y correr un script Tcl que se encargará de configurar todo el proyecto. Dentro de la carpeta principal que se ha movido anteriormente al espacio de trabajo existen, entre otras, dos carpetas denominadas "verilog" y "vhdl". Cada una contiene sus ficheros y scripts Tcl específicos, aunque ambas implementan una funcionalidad idéntica, se deja al usuario la opción de escoger el lenguaje con el que se sienta más cómodo. Para este proyecto se trabajará con VHDL, por lo que en la pantalla principal de Vivado buscar en el menú de opciones **Tools > Run Tcl Script**… y navegar hasta la carpeta de trabajo y luego seleccionar el fichero **xillydemo-vivado.tcl** dentro del directorio **/vhdl**. Vivado comenzará a configurar el proyecto y a crear los directorios pertinentes.

Una vez Vivado termina con el proceso de configuración del proyecto, automáticamente lo abrirá en una nueva ventana.

Antes de proceder a generar el fichero bit, hay que realizar unos pequeños cambios en el top del proyecto (xillydemo.vhd) por estar utilizando Vivado. Por tanto, al abrir xillydemo.vhd hay que buscar y eliminar estas 3 líneas que se encuentra al comienzo del fichero en la lista de los puertos de la entidad:

```
PS_CLK : IN std_logic;
PS_PORB : IN std_logic;
PS_SRSTB : IN std_logic;
```

Después, en ese mismo fichero buscar las siguientes 3 líneas y descomentarlas:

```
-- signal PS_CLK :  std_logic;
-- signal PS_PORB :  std_logic;
-- signal PS_SRSTB :  std_logic;
```

Por ahora no es necesario realizar ningún cambio más ya que lo que se pretende en esta sección es generar un xillydemo.bit base con el que poder arrancar la placa y comprobar que funciona correctamente. Así que lo próximo será buscar en el menú de la izquierda **Program and Debug > Generate Bitstream**. Tras aceptar todas las ventanas que pudieran aparecer, Vivado comenzará a generar el fichero bit y cuando termine se encontrará dentro de la carpeta de trabajo en **/vhdl/vivado/xillydemo.runs/impl_1/xillydemo.bit**. Es importante que el fichero .bit que se copie a la tarjeta SD tenga el nombre xillydemo.bit, de lo contrario no arrancará. (Se recomienda realizar la prueba de Xillybus en la sección 6.1.3.1, antes de continuar a la preparación del XADC).

5.2 Preparación del XADC

Para poder hacer uso del XADC es necesario instanciarlo en el diseño. La forma más sencilla de hacer esto es a través de un asistente que incluye Vivado para la mayoría de sus IP Core. Para empezar, será necesario tener abierto el mismo proyecto de la sección anterior e ir al apartado **Project Manager > IP Catalog**. Esto abrirá un listado de IP Cores disponibles para utilizar. En la caja de búsqueda dentro del catálogo de IP Cores, escribir "xadc", de esta forma se reducirá el número de elementos del listado (de hecho, debería aparecer únicamente el deseado). Seleccionar **XADC Wizard**.

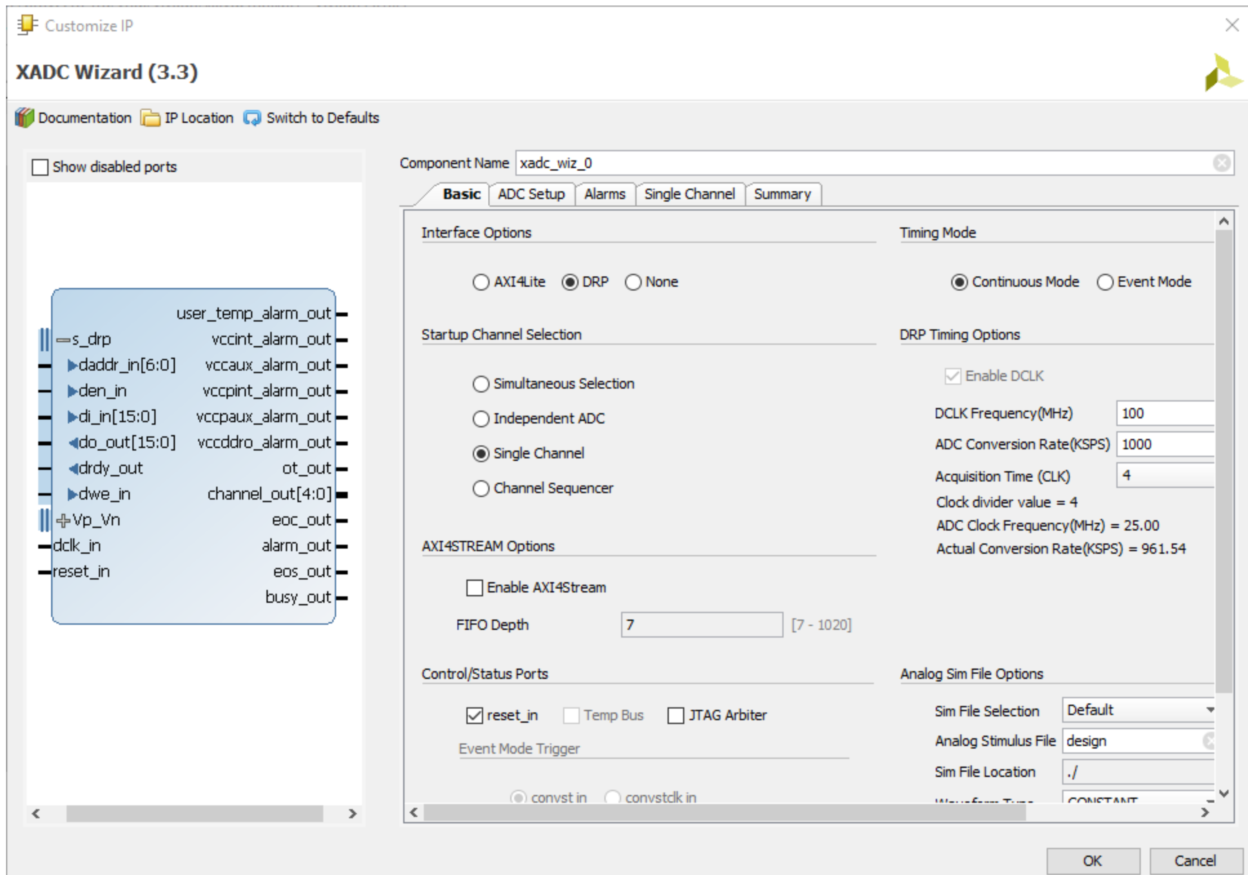

Figura 5-2. Asistente de configuración del XADC

El asistente de configuración del XADC se abrirá, organizando las diferentes opciones en 5 pestañas: **Basic**, **ADC Setup**, **Alarms**, **Single Channel/Channel Sequencer** y **Summary**.

5.2.1 Configuraciones en la pestaña "Basic"

- **Interface Options**. Es la forma en que se interactua con el XADC, habilitando una serie de puertos según la opción elegida. En este caso, se usará **DRP**. Al escoger la opción de DRP, se está usando únicamente la primitiva del XADC, así que no se utilizarán LUTs. Para ver de forma más detallada el consumo de recursos según los parámetros elegidos, se puede consultar [12, p. 12].
- **Startup Channel Selection**. Permite configruar el modo de operación del XADC. Puesto que se pretende muestrear los 3 canales disponibles en la ZedBoard de forma continua, se selecciona la opción **Channel Sequencer**.
- **AXI4STREAM Options**. No se hará uso de esta opción.

- **Control/Status Ports**. Habilita puertos con el fin de controlar o informar del estado del XADC. La señal "**reset_in**" viene activada por defecto. Se recomienda activar también **JTAG Arbiter**, y es que, aunque no se hará uso de las mismas en el proyecto final, pueden ser de utilidad para la fase de pruebas. Esta última opción habilitará 3 señales de salida:
  - **JTAGLOCKED**: cuando esta señal se activa (pasa a nivel alto), no se puede hacer ningún acceso de lectura o escritura a través por DRP, quedando bloqueado hasta que el usuario del JTAG deje de hacer uso del DRP.
  - **JTAGBUSY**: se activa durante alguna operación que se está llevando a cabo por JTAG a través del DRP.
  - **JTAGMODIFIED**: siempre que haya alguna escritura por parte de JTAG a algún registro DRP, esta señal se activará justo después de dicha escritura y no vovlerá a nivel alto hasta que no se vuelva a hacer una lectura/escritura por DRP (es una forma de notificar al usuario que ha podido haber un cambio en la configuración).
- **Timing Mode**. Esta opción permite elegir el modo de operación temporal. Se seleccionará la opción **Continuos Mode** (de esta forma la conversión se realizará de forma automática).
- **DRP Timing Options**. Para este apartado se presentan una serie de opciones:
  - **DCLK Frequency (MHz)**: será la frecuencia a la que irá la señal de reloj DRP. Debe ser proporcionada una señal de la frecuencia aquí indicada. En este caso se usará la señal de **100** MHz que ya proporciona Xillybus.
  - **ADC Conversion Rate (KSPS)**: con esta opción hay que tener especial cuidado. Recordando de la sección 3.2.5, existen dos relojes en juego: DCLK (el que se ha ajustado en la opción anterior) y el ADCCLK que es el reloj que utiliza internamente el XADC para funcionar. Este último es de una frecuencia inferior al DCLK pues se divide por un factor que se puede observar en **Clock divider value** (se encuetra justo debajo de la opción siguiente, Acquisition Time (CLK)), y este factor variará en función del ratio de conversión que se especifique. Para una aplicación de corazón como la que se trata en este proyecto, no se requiere un alto ratio de muestreo pues la frecuencia con las que trabaja el corazón son relativamente bajas. El problema es que según se indica en [5], el XADC será incapaz de funcionar para un ADCCLK inferior a 1 MHz o superior a 26 MHz, imposibilitando poner un ratio de conversión inferior a 40 KSPS (que resulta en un ADCCLK de 1.03 MHz). Aunque es posible escribir un ratio inferior y el asistente no pondrá ningún impedimento al generar la configuración correspondiente para el XADC, lo cierto es que a la hora de ponerlo en funcionamiento, el XADC no responderá. Por tanto, el valor que habrá que poner en este campo será de **40** KSPS.
  - **Acquisiton time (CLK)**: al seleccionar la opción de Channel Sequencer en Startup Channel Selection, esta opción queda deshabilitada.
- **Analog Sim File Options**. Relativas a la simulación (por software, usando Vivado) del XADC.
  - **Sim File Selection**: Relative Path txt.
  - **Sim File Location**: Colocar la ruta absoluta a un directorio del equipo. Preferiblemente dentro del directorio de trabajo. Para este ejemplo se ha usado la ruta D:\WorkspaceVHDL\proyecto_tfg\vhdl\vivado\xillydemo.srcs\sources_1\ip\xadc_wiz_0 Crear en ese directorio un fichero vacío llamado "design.txt".

5.2.2 Configuraciones en la pestaña "ADC Setup"

- **Sequencer Mode**. Puesto que se desea recorrer la secuencia de canales de forma continua, seleccionar **Continuos**.
- **Channel Averaging**. El XADC permite realizar un promediado de las muestras que toma. Por

defecto está desactivado. Para el proyecto se activará y se utilizará un promediado de **16** muestras. De esta forma los resultados de la conversión serán algo más fiables.

- **ADC Calibration**. ADC Offset and Gain Calibration.
- **Supply Sensor Calibration**. Sensor Offset and Gain Calibration.
- Marcar la opción **Enable CALIBRATION Averaging**.
- **External Multiplexer Setup**. Puesto que no se va a usar ningún multiplexador, dejar estas opciones por defecto (desactivadas).
- **Power Down Options**. Dejar ambos desactivados.

5.2.3 Configuraciones en la pestaña "Alarms"

Esta es la pestaña dedicada a las diferentes alarmas que se activan en caso superar unos ciertos límites (que se pueden especificar) de los distintos sensores internos. No es algo de interés para este trabajo, por lo que conviene desmarcar todas las alarmas que aquí aparecen.

5.2.4 Configuraciones en la pestaña "Channel Sequencer"

Al haber seleccionado el modo de secuenciador automático de canal, es necesario seleccionar qué canales se quieren muestrear. A continuación, se muestra una tabla con las opciones que se deberían marcar.

Tabla 5–1. Opciones a seleccionar en la pestaña "ADC Setup" del asistente del XADC

Jumper	Channel Enable	Average Enable	Bipolar	Acquisition Time
VP/VN	✔	✔	✔	
vauxp0/vauxn0	✔	✔	✔	
vauxp8/vauxn8	✔	✔	✔	

El significado de cada una de las opciones seleccionadas es:

- **Channel Enable**: selecciona ese canal para el muestreo.
- **Average Enable**: activa el promediado en ese canal.
- **Bipolar**: si se activa, la señal de entrada se interpretará como una señal bipolar.
- **Acquisition Time**: en este caso no aplica, por lo que no se selecciona para ninguna.

Dejar las opciones del resto de canales desactivadas.

5.2.5 Configuraciones en la pestaña "Summary"

Esta última pestaña no es de configuración como tal, sino que muestra un pequeño resumen de algunas opciones elegidas a lo largo de todas las pestañas. Comprobar que aparezca lo siguiente:

- Interface Selected: DRP.
- XADC operating mode: channel_sequencer.

- AXI4Stream Interface: false.
- Timing Mode: continuous.
- DCLK Freq(MHz): 100.
- Sequencer Mode: Continuous.
- Channel Averaging: 16.
- Enable External Mux: false.

Si todo es correcto, pulsar en OK y el asistente comenzará a generar un bloque del XADC que vendrá preconfigurado con las opciones escogidas.

5.2.6 Creación del diseño top-level para el XADC

Tras haber ejecutado el asistente de configuración, en el módulo "Sources" de Vivado habrá aparecido un nuevo recurso relacionado con el XADC recién generado. Al pulsar sobre el símbolo + y desplegar la jerarquía del IP Core, aparecerá el fichero **xadc_wiz_0.vhd**. Esto no es más que el *wrapper* o envoltorio del bloque, a partir del cual se creará el top-level del XADC.

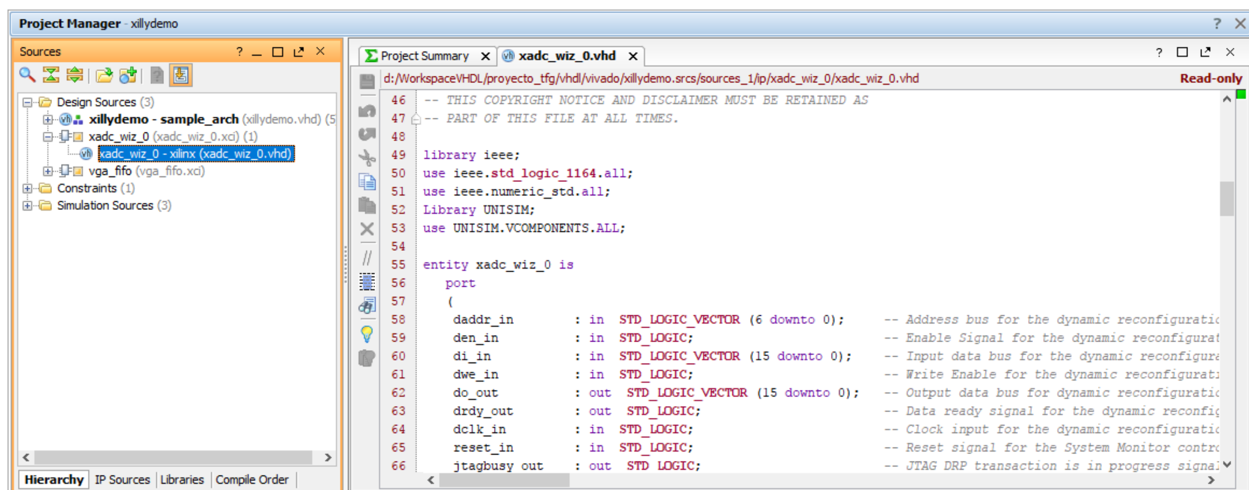

Figura 5-3. Wrapper del XADC generado por el asistente

Echando un vistazo al wrapper, en la arquitectura se observa como están inicializados todos los registros de configuración (*INIT_xx*, donde *xx* es la dirección del registro), del 40 al 5F, con los valores acordes a lo que se escogió en el asistente.

En base a este fichero automáticamente generado, se creará el diseño top-level. Para ello, en **Project Manager > Add Sources** se elegirá la opción **Add or create design sources**. Al pulsar en **Next**, aparece una ventana que permite o bien importar un fichero o crearlo de cero. Pulsar sobre **Create File** y rellenar con los siguientes datos:

- **File type**: VHDL
- **File name**: top_xadc
- **File location**: <Local to Project>

Al terminar, se presentará una ventana que permite definir los puertos de entrada y salida para la entidad del

fichero que se acaba de crear. Se hará durante la edición del fichero, así que pulsar OK.

Esto habrá generado un **top_xadc.vhd** en la lista del módulo "Sources". Ahora hay que editar el fichero para añadir como componente el wrapper del XADC y hacer las conexiones pertinentes para conseguir ponerlo en funcionamiento.

El componente a importar será la entidad de xadc_wiz_0.vhd, de forma que habrá que añadir el siguiente bloque de código a la arquitectura del top:

```
component xadc_wiz_0 is
   port
   (
    daddr_in        : in  STD_LOGIC_VECTOR (6 downto 0);
    den_in          : in  STD_LOGIC;
    di_in           : in  STD_LOGIC_VECTOR (15 downto 0);
    dwe_in          : in  STD_LOGIC;
    do_out          : out  STD_LOGIC_VECTOR (15 downto 0);
    drdy_out        : out  STD_LOGIC;
    dclk_in         : in  STD_LOGIC;
    reset_in        : in  STD_LOGIC;
    jtagbusy_out    : out  STD_LOGIC;
    jtaglocked_out  : out  STD_LOGIC;
    jtagmodified_out : out  STD_LOGIC;
    vauxp0          : in  STD_LOGIC;
    vauxn0          : in  STD_LOGIC;
    vauxp8          : in  STD_LOGIC;
    vauxn8          : in  STD_LOGIC;
    busy_out        : out  STD_LOGIC;
    channel_out     : out  STD_LOGIC_VECTOR (4 downto 0);
    eoc_out         : out  STD_LOGIC;
    eos_out         : out  STD_LOGIC;
    alarm_out       : out STD_LOGIC;
    vp_in           : in  STD_LOGIC;
    vn_in           : in  STD_LOGIC
);
end component;
```

El top del XADC es el que va a estar "en contacto" con la entidad de Xillybus, hay que definir las entradas y salidas teniendo esto en cuenta. Tras colocar el componente, lo siguiente será definir las entradas y salidas de la entidad:

- **clk**: Entrada. Serán los 100 MHz que se definieron en el asistente (DCLK).
- **reset**: Entrada. Esta señal se encargará de hacer un reset al bloque del XADC.
- **vp_in**: Entrada. Señal analógica positiva del canal dedicado.
- **vn_in**: Entrada. Señal analógica negativa del canal dedicado.
- **vauxp0**: Entrada. Señal analógica positiva del canal auxiliar 0.
- **vauxn0**: Entrada. Señal analógica negativa del canal auxiliar 0.
- **vauxp8**: Entrada. Señal analógica positiva del canal auxiliar 8.
- **vauxn8**: Entrada. Señal analógica negativa del canal auxiliar 8.
- **jtagLMB**: Salida. Será un bus de 3 bits, donde el bit más signifivativo será la señal JTAGLOCKED, el bit central será JTAGMODIFIED y el bit menos significativo JTAGBUSY. Como tal, no tendrá

utilidad en el funcionamiento del proyecto, sino que se usará como señal de control en pruebas.

- **dout**: Salida. Se corresponden a los 16 bits de la conversión del XADC.
- **drdy**: Salida. Esta señal pasa a nivel alto cuando hay un valor válido en el bus de salida de datos. Se utilizará como write enable de la FIFO a la que se escribirán los datos.
- **channel**: Salida. Bus de 5 bits que indica el canal cuya conversión se acaba de efectuar. Como tal, no tendrá utilidad en el funcionamiento del proyecto, sino que se usará como señal de control en pruebas.

Así que los puertos de la entidad de top_xadc quedarían así:

```
entity top_xadc is
    port
    (
        clk     : IN std_logic;
        reset   : IN std_logic;
        vp_in   : IN  STD_LOGIC;
        vn_in   : IN  STD_LOGIC;
        vauxp0  : IN  STD_LOGIC;
        vauxn0  : IN  STD_LOGIC;
        vauxp8  : IN  STD_LOGIC;
        vauxn8  : IN  STD_LOGIC;
        jtagLMB : OUT std_logic_vector(2 downto 0);
        dout    : OUT std_logic_vector(15 downto 0);
        drdy    : OUT std_logic;
        channel : OUT std_logic_vector(4 downto 0)
    );
end top_xadc;
```

Lo último que quedaría para acabar con el top es mapear los puertos del componente importado con los de la entidad, además de hacer las conexiones oportunas para poner en funcionamiento el XADC.

Se van a aprovechar las salidas del componente xadc_wiz_0, **eoc_out** y **channel_out**, para llevar a cabo la lectura de los registros donde se guardan los resultados de la conversión de los distintos canales. Aunque el XADC está constantemente conviritendo (pues así se ha configurado, en modo continuo), los resultados de la conversión no los saca por el puerto de salida a menos que se soliciten. Para ello, hay que hacer uso de las señales **den_in** y **daddr_in**. En daddr_in se coloca la dirección del registro que se desea leer, y una vez la dirección está ahí colocada se activa den_in para hacer la petición de lectura. Tras esto, el XADC coloca lo que haya en el registro en **do_out** y cuando los datos están estables activa la señal **drdy_out**.

Es aquí donde las señales eoc_out y channel_out entran en juego. La primera se activa justo cuando se ha terminado de realizar la conversión de un canal, y en channel_out se indica siempre el canal que se está convirtiendo. De esta forma, se puede emplear channel_out como daddr_in y eoc_out como den_in. Por otro lado, Xillybus invierte el orden de los bytes, así que la salida debe poner primero el byte menos significativo y el úlimo el más significativo (son 2 bytes en total).

```
aux_daddr <= "00" & aux_channel;
dout <= aux_dout(7 downto 0) & aux_dout(15 downto 8);
channel <= aux_channel;
```

Se han creado una serie de señales auxiliares:

```
signal aux_dout    : std_logic_vector(15 downto 0);
signal aux_channel : std_logic_vector(4 downto 0);
signal aux_eoc     : std_logic;
signal aux_daddr   : std_logic_vector(6 downto 0);
```

Por tanto, al realizar el mapeo de xadc_wiz_0, este quedará:

```
xadc : xadc_wiz_0
port map (
    daddr_in          => aux_daddr,
    den_in            => aux_eoc,
    di_in             => (others => '0'),
    dwe_in            => '0',
    do_out            => aux_dout,
    drdy_out          => drdy,
    dclk_in           => clk,
    reset_in          => reset,
    jtagbusy_out      => jtagLMB(0),
    jtaglocked_out    => jtagLMB(2),
    jtagmodified_out  => jtagLMB(1),
    vauxp0            => vauxp0,
    vauxn0            => vauxn0,
    vauxp8            => vauxp8,
    vauxn8            => vauxn8,
    busy_out          => open,
    channel_out       => aux_channel,
    eoc_out           => aux_eoc,
    eos_out           => open,
    alarm_out         => open,
    vp_in             => vp_in,
    vn_in             => vn_in
);
```

El código completo de top_xadc.vhd se puede encontrar en los anexos.

5.3 Preparación de Xillybus

Tal como se dijo en la primera sección de este capítulo, el kit empleado para la partición de arranque incluye una serie de funcionalidades por defecto que no son de ninguna utilidad para este proyecto, y por tanto el fichero bit que se generó solo era útil para comprobar el funcionamiento inicial del sistema, pero habría que generar uno nuevo con los cambios oportunos.

Para hacer esto, Xillybus en su web oficial ofrece una factoría que permite personalizar el IP Core de Xillybus y descargarlo. Desde la dirección http://xillybus.com/ipfactory/ es posible, previo registro gratuito, generar un nuevo core listo para usar en el proyecto [13].

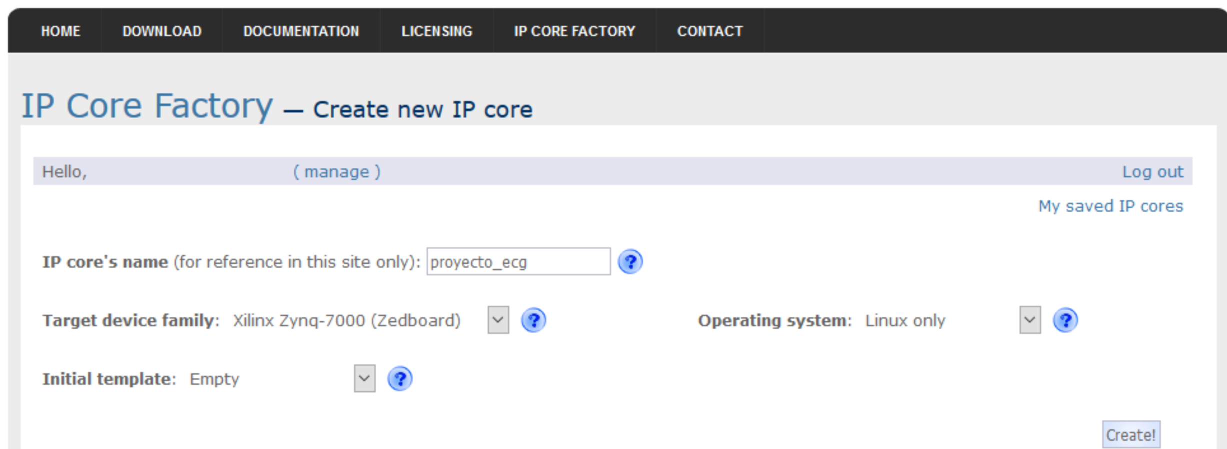

Figura 5-4. Pantalla inicial de creación de un nuevo core de la IP Core Factory de Xillybus

Pulsar sobre "**Add a new core**". En esta primera pantalla se pedirán datos básicos como el nombre que se le quiere dar al IP Core, la FPGA/tarjeta que se va a utilizar, etc. Se ha rellenado con los siguientes datos:

- **IP core's name**: proyecto_ecg (No tiene ningún impacto sobre el proyecto, así que es algo totalmente libre).
- **Target device family**: Xilinx Zynq-7000 (Zedboard).
- **Operating system**: Linux only.
- **Initial template**: Empty (Puesto que no se necesita nada de lo que incluye la demo por defecto).

Al pulsar sobre "**Create**" se creará el nuevo core, pero estará vacío. Lo siguiente será añadir dispositivos al core que serán los enlaces entre las FIFOs en la lógica y Xillinux (señales a las que conectaremos la FIFO y que Xillinux usará para comunicarse con estas).

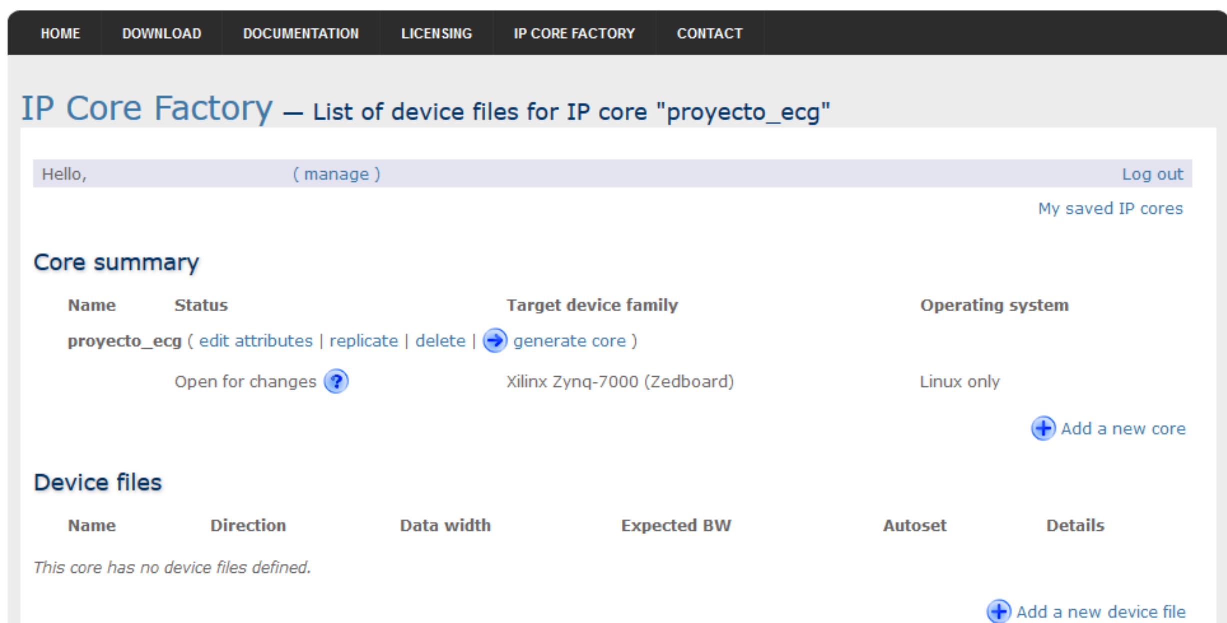

Figura 5-5. Pantalla que muestra el listado de dispositivos para el core generado

Cuando se pulsa en el enlace “**Add a new device file**”, aparece una nueva pantalla que permite configurar el dispositivo que se va a añadir. Las opciones disponibles son:

- **Device file’s name**: el nombre que se escoja aquí será el que aparecerá cuando se acceda desde Xillynus al directorio /dev/. Es el “programa” con el que se interactură para leer los datos de la FIFO en el sistema operativo.
- **Direction**: indica el sentido del flujo de datos.
- **Use**: dependiendo del tipo de aplicación que se escoja en este apartado, aparecerán o desaparecerán algunas opciones extra para configurar.
- **Data width**: sirve para indicar el ancho de los datos (de la FIFO, en definitiva).
- **Expected bandwidth**: el ancho de banda que se espera tener para este dispositvo en MB/s.
- **Autoset internals**: esta opción aparece marcada por defecto y realiza una configuración automática de algunos parámetros internos (flujo de datos asíncrono o síncrono, número de buffers y el tamaño de los mismos). En caso de querer configurarlos manualmente, bastará con desmarcarlo.
- **Buffering**: la opción de buffering solo se muestra para algunos casos de la opción “Use”. Permite indicar un tiempo aproximado de buffering.

Figura 5-6. Pantalla de creación de un nuevo dispositivo en la factoría de Xillybus

Aunque para el objeto de este trabajo no será necesario establecer un flujo en la dirección Host-FPGA, la web te obliga a crear dispositivos con flujo de datos en ambas direcciones (o uno bidireccional), por lo que se crearán dos dispositivos:

- **xillybus_datastream**: flujo de datos en dirección FPGA-Host.
- **xillybus_controlstream**: flujo de datos en dirección Host-FPGA.

5.3.1 Dispositivo xillybus_datastream

Los datos del dispositivo son:

- **Device file's name**: xillybus_datastream.
- **Direction**: Upstream (FPGA to host).
- **Use**: General purpose (se podría llegar a pensar que Data acquisition/playback podría ser una mejor opción para este caso, pero lo cierto es que esa opción añade un buffering y hace que los datos de la FIFO no se actualicen del todo hasta que no se deja de leer y se vuelve a iniciar la lectura).
- **Data width**: 16 bits.
- **Expected bandwidth**: 0.1 MB/s. Se ha configurado el XADC para que tenga un ratio de muestreo de 40 KSPS, y las muestras serán de 16 bits cada una. 40 KSPS * 2 B/sample = 80 KB/s, es decir 0.08 MB/s. Aunque Xillybus recomienda no poner más de lo que se tendrá porque podría afectar al ancho de banda de otros flujos de datos, puesto que no se va a hacer uso de ningún otro, no habrá problema en poner algo más de lo que realmente se usará.
- **Autoset internals**: activada.

5.3.2 Dispositivo xillybus_controlstream

Los datos del dispositivo son:

- **Device file's name**: xillybus_controlstream.
- **Direction**: Downstream (host to FPGA).
- **Use**: General purpose.
- **Data width**: 8 bits.
- **Expected bandwidth**: 0.01 MB/s.
- **Autoset internals**: activada.

5.3.3 Generación e inserción del nuevo core

Tras crear estos dos dispositivos, ya se puede generar el nuevo core desde la pantalla del listado de dispostivos (Figura 5-5), pulsar en "**generate core**". Xillybus tarda unos minutos en procesarlo, por lo que es normal que al principio no se pueda descargar inmediatamente.

Se descargará un ZIP con los ficheros necesarios a reemplazar en el directorio de trabajo actual del proyecto, además de un archivo de texto con unas instrucciones sobre cómo proceder para aplicar los cambios correctamente:

- En el directorio de trabajo del proyecto, dentro de /vhdl/src, hay varios archivos y entre ellos **xillybus.v** y **xillybus_core.v**. Estos dos ficheros deben ser reemplazados por los del mismo nombre incluidos en el ZIP del nuevo core descargado.
- En el directorio de trabajo del proyecto, dentro de /cores, hay un archivo llamado **xillybus_core.ngc**. Debe ser reemplazado por el fichero del mismo nombre incluido en el ZIP del nuevo core generado.
- Editar el top del proyecto, **xillybus.vhd**. La mayoría de funcionalidades del kit inicial ya no sirven, por tanto hay que eliminarlos y añadir los dos nuevos dispositivos que se han generado con el nuevo core. Se recomienda utilizar la plantilla "**template.vhd**" disponible en la carpeta "instantiation templates" del ZIP descargado para realizar las modificaciones en xillybus.vhd.

Se recomienda borrar el **xillybus_core.ngc** existente, tratar de generar el fichero bit y observar que Vivado da un error al intentar generarlo porque no encuentra el fichero faltante. Entonces, copiar el nuevo fichero y proceder a generar el fichero bit. De esta forma se puede asegurar que Vivado ha detectado los nuevos

cambios.

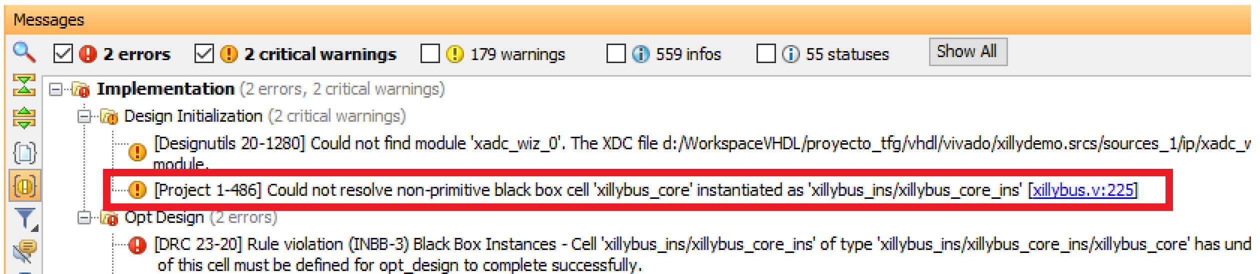

Figura 5-7. Error que se muestra cuando no encuentra xillybus_core.ngc

Tras realizar estos tres cambios, debería ser posible generar un fichero bit sin ningún tipo de complicación, aunque las conexiones entre el XADC y Xillybus aún no estarían listas pues falta el elemento de unión: la FIFO. Además, faltaría modificar la entidad del top de Xillybus y el fichero de restricciones para añadir las nuevas entradas analógicas, así como importar el top del XADC.

5.3.4 Generación de la FIFO para el flujo de datos

El proceso para generar la FIFO es muy parecido al seguido con el XADC, en el sentido de que hay que realizar una búsqueda entre la lista de IP Cores y viene acompañado de un asistente que permite generar el wrapper para la FIFO.

Esta vez si se escribe FIFO en el buscador del IP Catalog, aparecerán varios resultados coincidentes con la palabra FIFO. La que se busca es la que se encuentra dentro de la categoría **Memories & Storage Elements > FIFOs > FIFO Generator** [14].

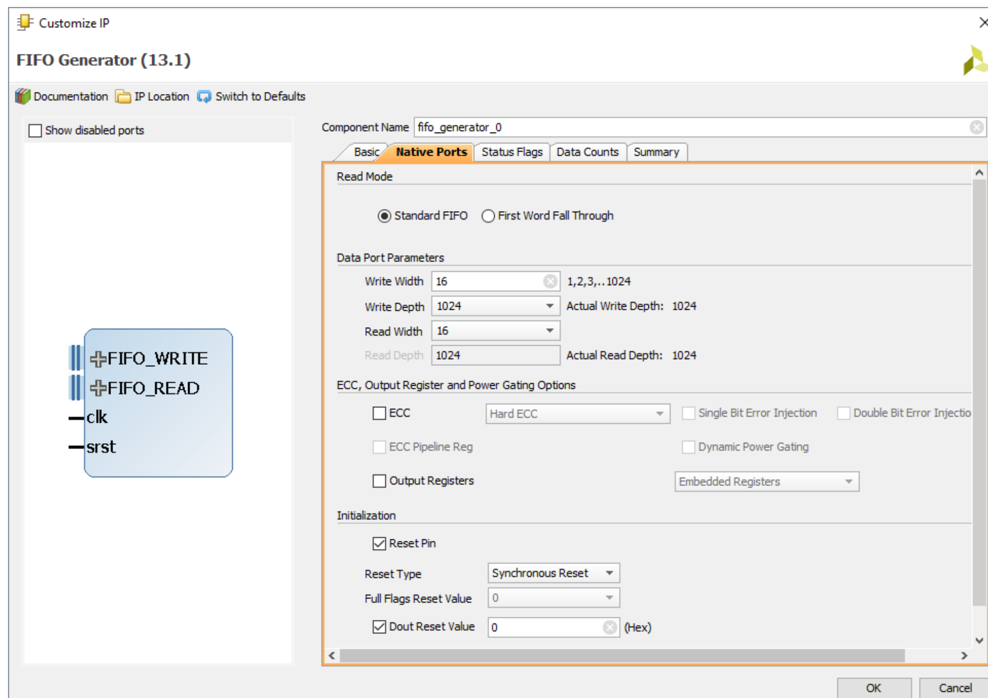

Figura 5-8. Asistente de configuración de la FIFO

Este asistente también organiza las opciones de configuración en diversas pestañas, pero en este caso solo resulta de interés una de ellas, el resto se dejará con los valores que trae por defecto.

La pestaña en cuestión es "**Native Ports**". En el apartado **Data Port Parameters**, aperecerá como ancho de escritura (**Write Width**) 18 bits por defecto. Habrá que cambiar este valor a **16**. Automáticamente se cambiará el ancho de lectura (Read Width) también a 16 (cambiarlo manualmente en caso de que no lo hiciera).

Es el único cambio que hay que realizar, pulsar sobre OK para generar la FIFO.

5.3.5 Implementación de la FIFO y el XADC al top de Xillybus

El último paso será añadir estos dos elementos a xillybus.vhd. Para ello habrá que importar la FIFO y el top del XADC como componentes.

Comentar que a la hora de mapear los puertos:

- XADC: los puertos "channel" y "jtagLMB" se han dejado a open pues solo se usarán en caso de necesitar realizar alguna comprobación sobre el funcionamiento del convertidor. El reset del XADC está a 0 todo el tiempo.
- FIFO: la entrada de datos de la FIFO será la salida de los datos del XADC, mientras que el wr_en de la FIFO es el drdy de salida del XADC. El resto de señales van conectadas con Xillybus:
    - **rd_en** se conecta con user_r_datastream_rden, de esta forma Xillybus gestiona la lectura de la FIFO.
    - **dout** se conecta con user_r_datastream_data, para que Xillybus pueda recibir los datos de la FIFO.
    - **empty** se conecta con user_r_datastream_empty, de forma que Xillybus no leerá la FIFO cuando esta señal este activa.
    - **full** se deja a "open".
    - **reset** se aplicará en función de user_r_datastream_open. Esta último la activa Xillybus cuando está leyendo la FIFO, en ese momento no se desea resetear la FIFO. Mientras que no se esté leyendo la FIFO, el contenido de la misma no es de interés, por lo que estará en reset. Se utilizará por tanto user_r_datastream_open negada.

Ambos componentes van conectados a la señal de reloj "**bus_clk**" que proporcióna Xillybus y que es de 100 MHz.

Faltaría por editar la propia entidad de Xillybus para admitir como entradas las señales analógicas a muestrear, por lo que se añaden junto a las existentes:

```
-- entradas para el XADC
vp_in    : IN  STD_LOGIC;
vn_in    : IN  STD_LOGIC;
vauxp0   : IN  STD_LOGIC;
vauxn0   : IN  STD_LOGIC;
vauxp8   : IN  STD_LOGIC;
vauxn8   : IN  STD_LOGIC
```

De esta forma se concluye con la modificación del top de Xillybus.

Se puede consultar el aspecto final y actualizado de xillybus.vhd en los anexos.

5.3.6 Entradas analógicas en el fichero de restricciones

Aún no se puede generar el fichero bit porque falta editar el fichero de restricciones o *contraints* [15], **xillybus.xdc**. Este archivo permite definir una serie de restricciones sobre las señales de reloj (creación de relojs virtuales) o la localización de los pines, por ejemplo.

Xillybus ya incluye sus propias restricciones, por lo que habrá que editar el archivo existente para especificar a qué pines se corresponden las nuevas entradas que se han añadido a la entidad de Xillybus (las entradas analógicas).

En [16], se menciona a qué pines físicos corresponden los distintos canales análogicos disponibles. Se recogen en la siguiente tabla:

Tabla 5–2. Pines de las entradas analógicas en la Zynq

Entrada analógica	Pin de la Zynq
VP	L11
VN	M12
VAUXP0	F16
VAUXN0	E16
VAUXP8	D16
VAUXN8	D17

Así que las restricciones referentes a los puertos quedarían como:

```
 set_property -dict "PACKAGE_PIN L11 IOSTANDARD LVCMOS33" [get_ports
"vp_in"]
 set_property -dict "PACKAGE_PIN M12 IOSTANDARD LVCMOS33" [get_ports
"vn_in"]

 set_property -dict "PACKAGE_PIN F16 IOSTANDARD LVCMOS33" [get_ports
"vauxp0"]
 set_property -dict "PACKAGE_PIN E16 IOSTANDARD LVCMOS33" [get_ports
"vauxn0"]

 set_property -dict "PACKAGE_PIN D16 IOSTANDARD LVCMOS33" [get_ports
"vauxp8"]
 set_property -dict "PACKAGE_PIN D17 IOSTANDARD LVCMOS33" [get_ports
"vauxn8"]
```

Por último, aunque el problema del XADC con el JTAG se solucionó al editar el devicetree.dtb que incluía Xillybus, no está de más desactivar el acceso por JTAG al XADC añadiendo esta línea a xillybus.xdc:

```
 set_property BITSTREAM.GENERAL.JTAG_XADC Disable [current_design]
```

Se puede consultar el código completo de xillybus.xdc en los anexos.

6 RESOLUCIÓN DE ERRORES Y PRUEBAS

El desarrollo del sistema no ha sido algo directo. Ha pasado por un largo proceso de pruebas con el fin de intentar solventar los principales errores que iban surgiendo durante la implementación de las distintas partes que conforman el proyecto al completo.

Lo que aquí se pretende es recoger algunos de los errores cuya resolución ha supuesto una cierta dificultad y la forma de proceder para solucionarlos. También se presentarán las herramientas qué se han utilizado para realizar estas pruebas.

6.1 Pruebas y verificación de resultados

Cada componente del proyecto ha sido simulado y probado de forma individual, de modo que se pueden detectar fallos de forma más precisa que lo que permitiría el conjunto de todos los componentes funcionando a la vez.

Una vez el funcionamiento propio de cada uno de estos componentes está asegurado, se prueba todo el sistema al completo en la ZedBoard.

6.1.1 XADC

6.1.1.1 Simulación software

Con Vivado es posible simular el funcionamiento del XADC. Para ello será necesario crear un archivo de simulación donde se especifiquen las distintas señales que se irán activando en múltiplos de ciclos de reloj. La forma de crear este archivo es igual que cuando se creó el top_xadc.vhd en la sección 5.2.6, solo que, a la hora de crearlo, en lugar de escoger la opción "Add or create design sources", se escogerá la opción "**Add or create simulation sources**" y el nombre del fichero será **tb_top_xadc.vhd**.

Este archivo VHDL tendrá la entidad vacía y en la arquitectura se importará el top_xadc, además de definir lo que dura un ciclo de reloj.

Contará con dos procesos, uno que se encargará de generar la señal de reloj, y el otro es el proceso de estímulos, donde se irán modificando las señales de entrada para ver qué se obtiene en las señales de salida. El código fuente del mismo se puede consultar en los anexos.

Por lo general, se podría simular el código con solo esto, pero en el caso del XADC hay un archivo más necesario para ello. Y es que las entradas del XADC no son digitales, sino que son analógicas. Por tanto, usa un fichero de entrada donde se especifica el valor de estas entradas en puntos concretos de tiempo. Este fichero se mencionaba durante la creación del XADC por medio del asistente de configuración en la sección 5.2.1. Si se configuró como ahí se indicaba, ahora es el momento de editar el fichero vacío que se creó. La estructura del fichero es la siguiente:

TIME	VP	VN	VAUXP[0]	VAUXN[0]	VAUXP[8]	VAUXN[8]
0	-0.25	0.25	-0.25	0.25	-0.25	0.25
67000	-0.1	0.2	-0.1	0.2	-0.1	0.2
100000	0.2	-0.1	0.2	-0.1	0.2	-0.1
134000	0.25	-0.25	0.25	-0.25	0.25	-0.25

La primera línea es obligatoria, pues indica en qué instante de tiempo ocurre y los valores de los distintos canales. En este caso, como solo se están monitorizando estos 3 canales, son los únicos que aparecen (pero también se podrían probar con sensores internos, por ejemplo, el de temperatura). El tiempo debe ir especificado en nanosegundos, la tensión en voltios y la temperatura en grados celsius. Puede tener tantas líneas como se desee.

Con estos dos ficheros ya se podría realizar la simulación. Para ello es necesario seleccionar el top_xadc.vhdl como Top del proyecto. Para ello, desde la lista de archivos fuente, hacer click derecho en top_xadc.vhdl y pulsar sobre "**Set as Top**". Hay que tener en cuenta que hay dejar el top del sistema como estaba (es decir, dejar xillybus.vhd como el top) antes de generar el fichero bit cuando se terminen de realizar las pruebas y simulaciones.

Ahora mismo no tiene ningún fichero de simulación asociado, algo que se hace desde la opción **Simulation > Simulation Settings**. Se abrirá una ventana, en la opción **Simulation top module name** seleccionar **tb_top_xadc**.

Por último, acceder a la opción **Simulation > Run Simulation > Run Behavioral Simulation**. Automáticamente cargará una nueva ventana con una cronología donde se podrán ver las señales definidas en el testbench del XADC. En caso de necesitar observar alguna otra señal que no se encuentra aquí, es posible añadirlas desde el menú que se encuentra a la izquierda.

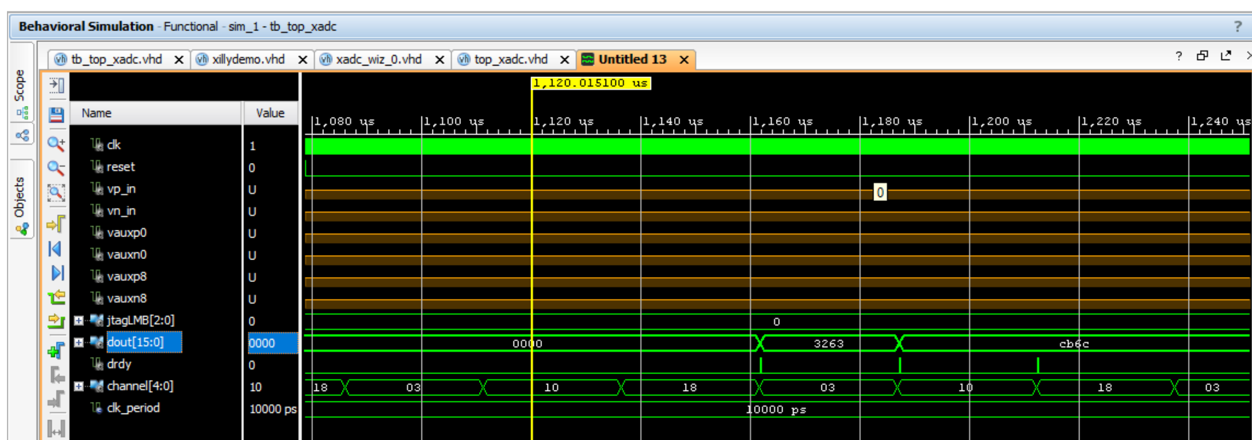

Figura 6-1. Ventana de simulación del XADC de Vivado

Tener en cuenta que, si se simula el XADC tal como se ha generado para el proyecto, está activado el promediado de 16 muestras, con lo cual los resultados tardarán en salir porque primero necesita 16 muestras de cada canal.

Las entradas analógicas no pueden visualizarse en el simulador, por eso aparecen como “Undefinied”.

Si el comportamiento que se muestra en el simulador es el esperado, se puede continuar con el siguiente paso, la simulación hardware. Que se hayan obtenido buenos resultados en la simulación software no indica que a la hora de ponerlo en funcionamiento en la realidad vaya a funcionar, pero ayuda a detectar cierto de fallos, como por ejemplo que se haya olvidado conectar alguna señal con otra y su salida no sea válida.

6.1.1.2 Prueba hardware

Existen varias formas de probar el XADC vía hardware, en este caso se opta por usar el Hardware Manager [17] que incluye Vivado. La simulación hardware se realiza por dos razones:

- Descartar un fallo físico en el XADC (comprobar que funciona correctamente).
- Comprobar que la configuración del mismo es la deseada.

Para monitorizar el XADC, el Hardware Manager accede al mismo por medio de JTAG, por lo que no es posible llevarlo a cabo dentro del mismo proyecto pues Xillinux impide que el XADC pueda ser accedido de esta forma. Se recomienda crear un nuevo proyecto donde lo único que hay que hacer es crear un nuevo XADC como se explicó en la sección 5.2, siguiendo los mismos pasos en el asistente de configuración. Una vez generado, lo único que habría que hacer es generar el fichero bit directamente.

De esta forma, y colocando la ZedBoard en modo de arranque por JTAG (ver sección 3.3.2.2), se procede a programar la FPGA con el fichero bit generado. Para ello hay que conectar el USB que incluye la ZedBoard al puerto PROG de la placa.

Ir a **Program and Debug > Hardware Manager**. Tras encender la placa y pulsar en la opción **Open target > Auto Connect**, automáticamente se conectará con la FPGA. Pulsar sobre “**Program device**” y utilizar el fichero bit recién generado.

En la lista de Hardware, dentro de localhost aparecerá la FPGA y dentro el XADC. Si se hace doble click sobre este, se abrirá el System Monitor [18] que permite monitorizar las diferentes entradas y sensores del convertidor. Así es posible comprobar que el XADC funciona correctamente y no está dañado.

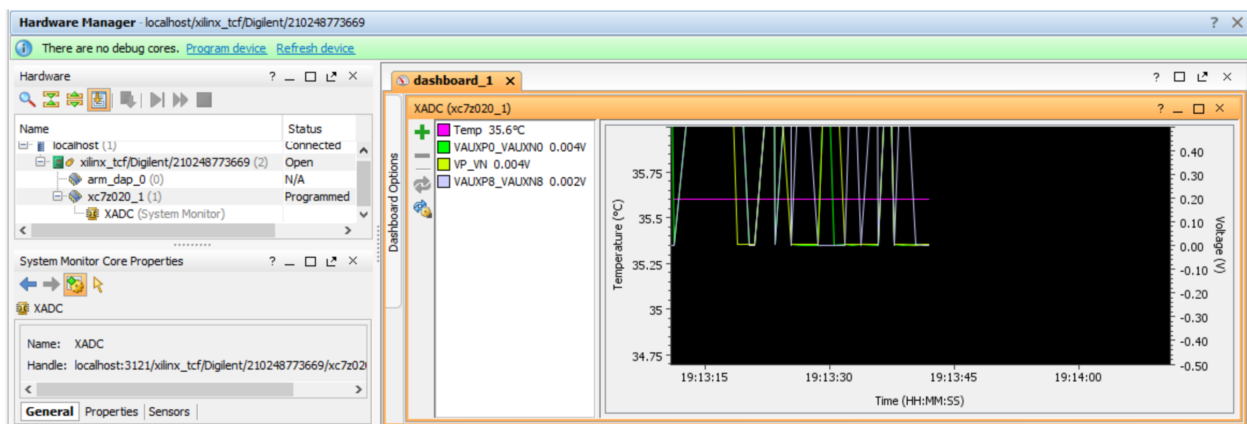

Figura 6-2. Monitorización del XADC desde la opción Hardware Manager

6.1.2 FIFO

6.1.2.1 Simulación software

Este es el componente menos problemático de todo el proyecto, por lo que su simulación podría ser algo opcional. Aunque para ser rigurosos, se ha creado un testbench también para la misma donde se hacen algunas pruebas escribiendo y leyendo (consultar anexos).

Mencionar que la FIFO se generó y directamente se incluyó en el top de Xillybus, con lo cual no tiene un código VHDL al que poder hacerle "Set as Top". La solución a esto es muy sencilla: crear un fifo.vhd donde la entidad sea idéntica al component fifo_generator_0. De esta forma ya se puede poner como top y pasarle su testbench.

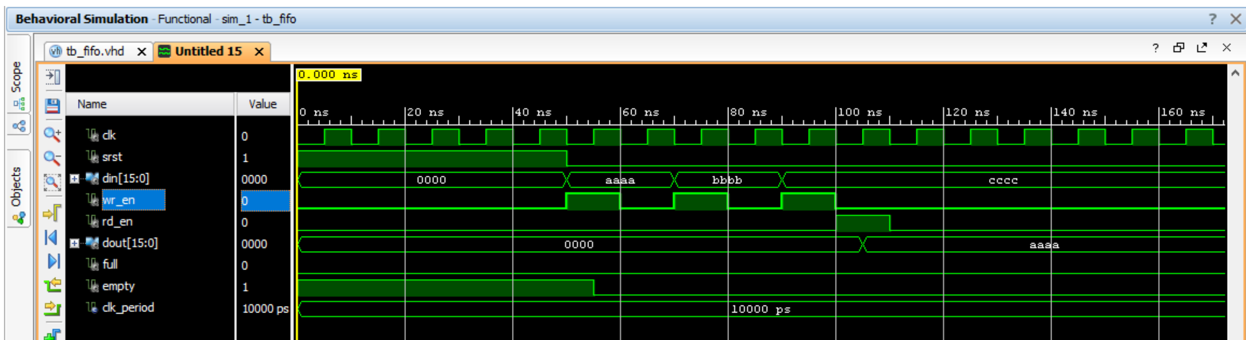

Figura 6-3. Ventana de simulación de la FIFO en Vivado

La simulación hardware de la misma no es necesaria, aunque si se quiere, se puede probar en conjunto con Xillybus en la siguiente sección.

6.1.3 Xillybus

6.1.3.1 Prueba hardware

El momento ideal para probar Xillybus es justo después de llevar a cabo la sección 5.1.3, porque en este caso se ha generado un fichero bit con toda la lógica que trae el kit a modo de demostración. Y este kit trae preconfiguradas 2 FIFOs, una de 8 bits y otra de 32, en modo *loopback*. Es decir, sus conexiones están hechas de tal forma que, al escribir en la FIFO, devolverá el mismo contenido que se ha escrito.

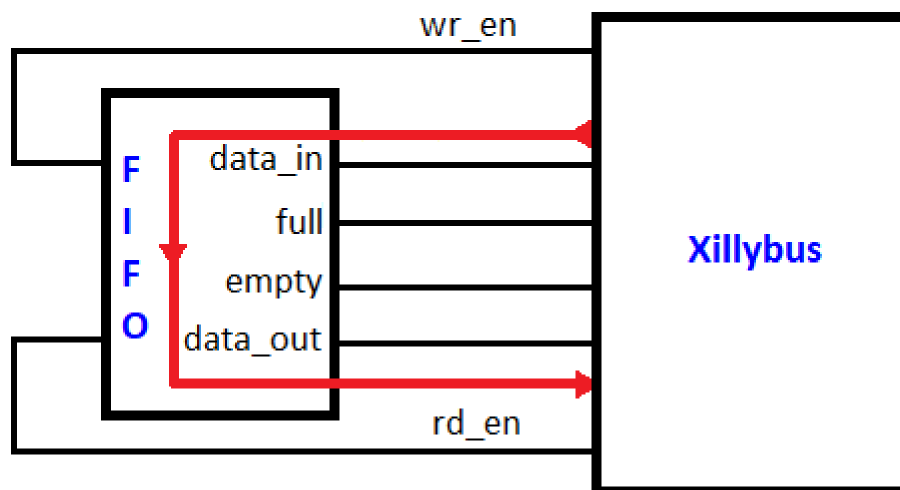

Figura 6-4. Esquema de *loopback* entre una FIFO y Xillybus

La forma de probar esto será con la ZedBoard en modo arranque tarjeta SD y todos los archivos puestos correctamente en la SD, arrancar Xillinux y abrir dos terminales [19]. Se usará la FIFO de 8 bits, de esta forma se pueden ver los caracteres cómodamente.

- En uno de ellos se abrirá el fichero que lee la FIFO. Para la de 8 bits la ruta es /dev/xillybus_read_8. Basta con hacer "**cat**" a ese fichero.

```
cat /dev/xillybus_read_8
```

- El otro será el de escritura, que será /dev/xillybus_write_8. En este caso, se escribirá a la FIFO:

```
cat > /dev/xillybus_write_8
```

Si se recibe por el primer terminal lo que escribió en el segundo, es que todo funciona correctamente.

Otra forma de probar esto si ya no se tiene el fichero bit que se generó al principio con el kit de demostración, es crear una máquina de estados simple que envíe dos caracteres (cada uno de un byte) a la FIFO de 16 que se configuró (de aquí lo que se mencionó anteriormente que la FIFO se podía simular en conjunto con Xillybus).

Solo habría que desconectar el top_xadc de la FIFO y poner en su lugar la máquina de estados que envía los caracteres.

Por hacerlo más interesante, se puede asociar la entrada de la máquina con uno de los interruptores que incorpora la ZedBoard, de forma que solo envíe una vez los dos caracteres cuando uno de ellos pase a la posición que se corresponde con un 1.

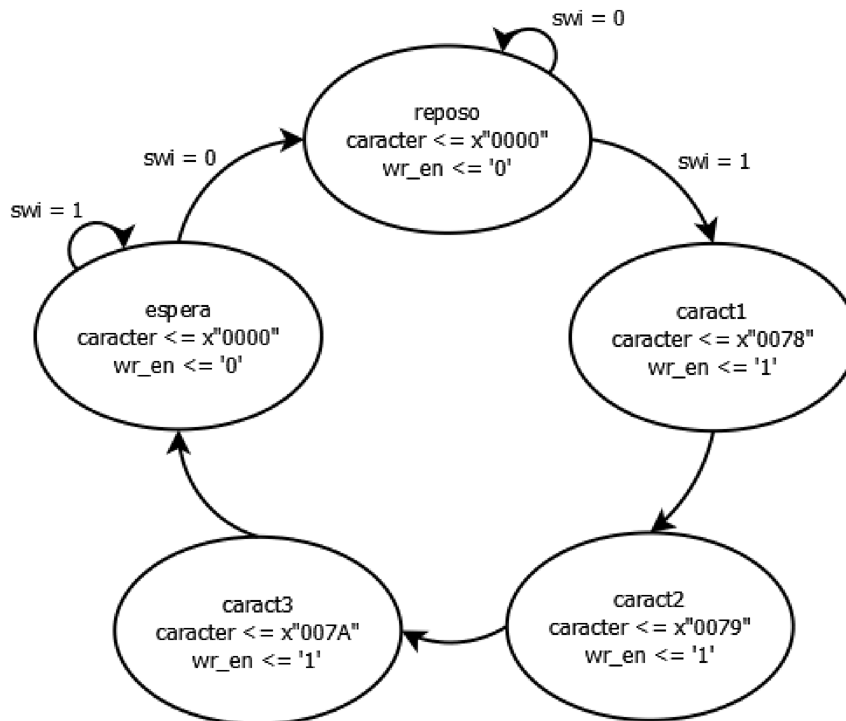

Figura 6-5. Diagrama de estados de envia_caract.vhd

Puesto que el top de Xillybus ya incluye entre sus entradas los interruptores, solo habría que asociarlo al mapear los puertos. Concretamente, el del SW0 se corresponde con PS_GPIO(11).

El código de envia_caract.vhd se puede consultar en los anexos.

6.1.4 Sistema completo

6.1.4.1 Prueba hardware

Para probar el sistema al completo es necesario haber generado el fichero bit del proyecto y tener la tarjeta SD cargada con los ficheros necesarios para arrancar la ZedBoard. También se recomienda tener un generador de señales para poder variar la tensión de entrada de los canales analógicos del XADC y así poder realizar pruebas.

Para una comprobación directa, bastaría con imprimir por pantalla los resultados que se están guardando en la FIFO a través del fichero /dev/xillybus_datastream. No tendría mucho sentido ejecutar el comando "cat" porque no serían caracteres imprimibles. Existe otro comando, "**hexdump**", que permite ver el contenido hexadecimal de la FIFO (funciona igual que "cat", solo que imprime el contenido binario sin traducir a caracteres):

```
hexdump -C /dev/xillybus_datastream
```

La opción -C, además de la representación hexadecimal, también lo imprime en ASCII.

```
root@localhost:~# hexdump -C /dev/xillybus_datastream
00000000  ff d5 ff b3 ff a1 ff c4  ff b1 ff a3 00 2f ff b5  |............./..|
00000010  ff a2 ff cc ff b0 ff a2  00 32 ff b7 ff a3 ff ec  |.........2......|
00000020  ff b0 ff a6 00 33 ff b5  ff 9f 00 0b ff b0 ff 9d  |.....3..........|
00000030  00 2b ff bd ff a0 ff ff  ff b3 ff 9e 00 0a ff b7  |.+..............|
00000040  ff a0 ff c4 ff b0 ff a2  00 21 ff b6 ff 9d ff c4  |.........!......|
00000050  ff b6 ff 9e 00 38 ff b8  ff a1 ff f8 ff b7 ff a6  |.....8..........|
00000060  00 20 ff b3 ff 9b ff c6  ff b7 ff a6 00 2a ff b9  |. ...........*..|
00000070  ff a0 ff ba ff b4 ff a3  00 2d ff b0 ff 9f ff b7  |.........-......|
00000080  ff ae ff 9b 00 56 ff b3  ff a3 ff d2 ff b4 ff a0  |.....V..........|
00000090  00 20 ff b3 ff 9d ff f3  ff af ff a0 00 0c ff b8  |. ..............|
000000a0  ff 9f 00 2f ff b5 ff 9a  00 64 ff bb ff 9d ff d7  |.../.....d......|
000000b0  ff b8 ff 9e ff e5 ff b2  ff a4 00 1b ff b8 ff a4  |................|
000000c0  00 1c ff b3 ff 9c ff c3  ff ba ff 99 00 42 ff bc  |.............B..|
000000d0  ff 9c ff de ff ba ff 9d  00 10 ff b7 ff a1 00 2d  |...............-|
000000e0  ff b8 ff a6 ff de ff bc  ff 9e 00 32 ff b8 ff a3  |...........2....|
000000f0  ff e9 ff bd ff a0 00 2b  ff b5 ff 9f ff e1 ff b8  |.......+........|
00000100  ff 9e 00 0b ff b7 ff 9e  00 35 ff bc ff 9f ff d8  |.........5......|
00000110  ff bc ff 9e 00 3e ff b8  ff 9f ff cb ff b5 ff 9f  |.....>..........|
00000120  00 23 ff b8 ff 9f ff f8  ff b7 ff a2 00 42 ff b8  |.#...........B..|
00000130  ff a3 00 06 ff b9 ff a1  ff e6 ff bb ff a0 ff ed  |................|
00000140  ff bb ff a6 00 1e ff ba  ff 9e 00 19 ff b7 ff 9f  |................|
00000150  ff bb ff bd ff 9f 00 25  ff bc ff 9e 00 15 ff b7  |.......%........|
```


Figura 6-6. Resultado de ejecutar el comando hexdump para ver los datos de la conversión del XADC

Así se consiguen ver los resultados, aunque irán pasando muy rápido. Si se quieren observar con detenimiento bastará con parar la ejecución (Ctrl + C) y volver a ejecutar si se quisiera continuar.

Otra forma de comprobar los valores de una manera algo más cómoda es guardando el contenido de la FIFO en un archivo (el cual crecerá rápidamente en tamaño) y siendo tratado posteriormente para representar los datos gráficamente.

Lo que aquí se propone es:

- Guardar el contenido directmante en un archivo con:

  ```
  cat /dev/xillybus_datastream > output.xadc
  ```

- Formatear ese fichero a algo que se pueda interpretar con, por ejemplo, Matlab. Para ello se usa Python.
- Recoger los datos formateado en Python y representarlo usando Matlab.

Lo que se obtiene al leer la FIFO es un fichero binario. Este fichero será procesado por un script en Python que irá leyendo de dos en dos bytes, se quedará solo con los 12 bits más significativos e irá convirtiendo esos bits a hexadecimal para producir un fichero llamado "output.format". En cada línea de este fichero habrá tres valores, el primero se corresponde con el canal dedicado V_P/V_N, el segundo con V_{AUX0} y el tercero con V_{AUX8}. Este fichero será interpretado por Matlab, de forma que guardará los valores de cada canal en su vector correspondiente y aplicará la conversión de hexadecimal a mV según la función de transferencia bipolar de la sección 3.2.3. Estos vectores luego podrán ser usados para ser representados gráficamente, pudiendo observar de forma gráfica cómo ha ido evolucionando la entrada del convertidor.

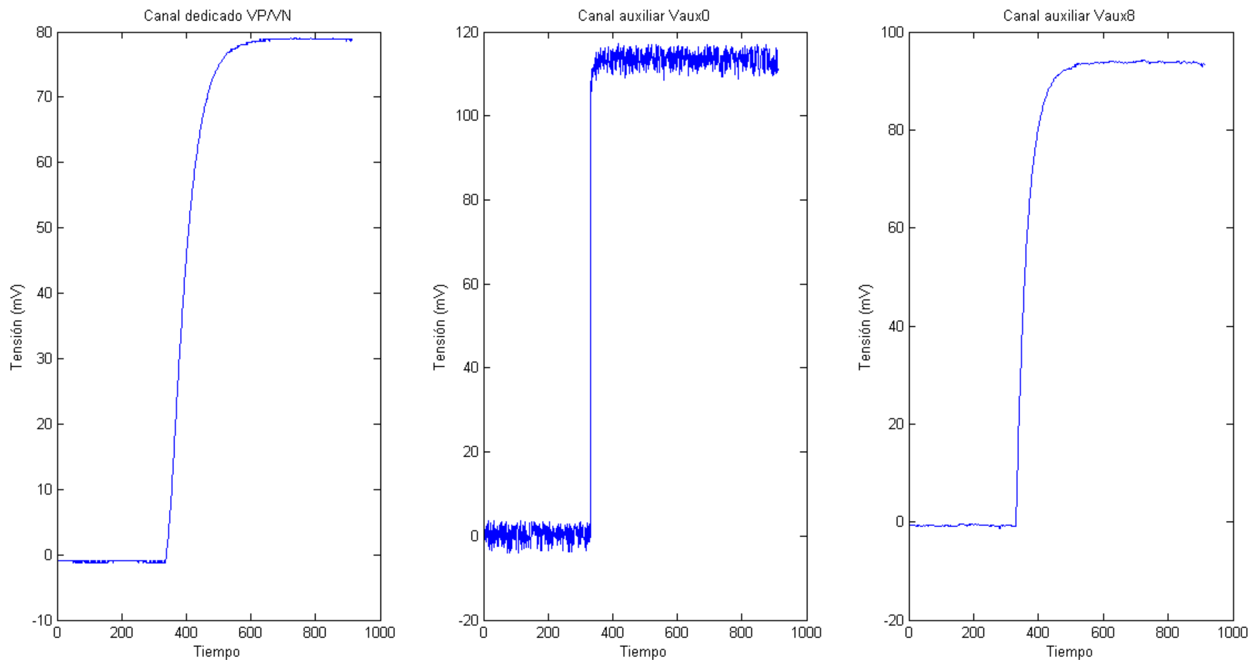

Figura 6-7. Resultado de guardar los resultados de la conversión en un fichero, formatearlo con Python y representarlo con Matlab.

Ambos scripts (Python y Matlab) se pueden consultar en los anexos.

6.2 Resolución de errores

6.2.1 Valores que provienen de un canal inválido del XADC

Este error tiene lugar realizando las pruebas finales de la sección anterior si se usa por defecto el devicetree.dtb que incluye el kit de Xillybus. A pesar de que todo se encuentra bien configurado, los valores que devuelve el XADC son erróneos pues provienen de un canal inválido. La causa de este fallo es no haber modificado el devicetree.dtb como se indica en la sección 5.1.2.

Detectar este error no es algo fácil, pues nada parece indicar que Xillinux pudiera interferir en el funcionamiento del XADC de alguna forma.

Para empezar, los valores que devolvía el XADC eran cercanos al máximo posible, `FFFh`. Si el XADC no se alimentaba con ninguna tensión, podrían parecer resultados coherentes pues, para el modo bipolar, valores cercanos a `FFFh` quiere decir que la tensión de alimentación es relativamente baja y negativa. Sin embargo, si se alimentaba, estos valores no cambiaban y continuaban cercanos a ese máximo. Con el fin de comprobar que los canales que se monitorizaban eran correctos, en lugar de escribir en la FIFO los valores que el XADC iba convirtiendo, se utilizó la salida “channel” (la que, durante el desarrollo del sistema, se comentó que solo tendría uso para pruebas, pero no en el proyecto final como tal), la cual indica qué canal se está convirtiendo en todo momento. El valor que devolvía era siempre `9h` que, consultando en [5, p. 45], significa canal inválido.

Había que ir examinando las diferentes señales involucradas en el proceso para comprobar que todo era correcto. Vivado incluye un IP Core, denominado Integrated Logic Analyzer [20], que permite debuggear la lógica por medio de su Hardware Manager. El problema es que hace uso de la conexión JTAG, algo que dejaba de funcionar cuando se usaba Xillinux.

Se optó por crear una máquina de estados que fuera contando pulsos y se usó en combinación con los LEDs de

la ZedBoard para ir mostrando los resultados de esta cuenta. Había que comprobar que el XADC estaba funcionando, así que se contabilizaron algunos pulsos como EOC, que se activa cuando se produce una conversión o DRDY, el cual se activa cuando hay un dato válido en el bus de salida. Ambos contadores funcionaban sin problema (se asignaron los bits más significativos a los LEDs de forma que se podía ver como se iban encendiendo). Esto sugería que, al menos, el XADC estaba funcionando, aunque de forma incorrecta.

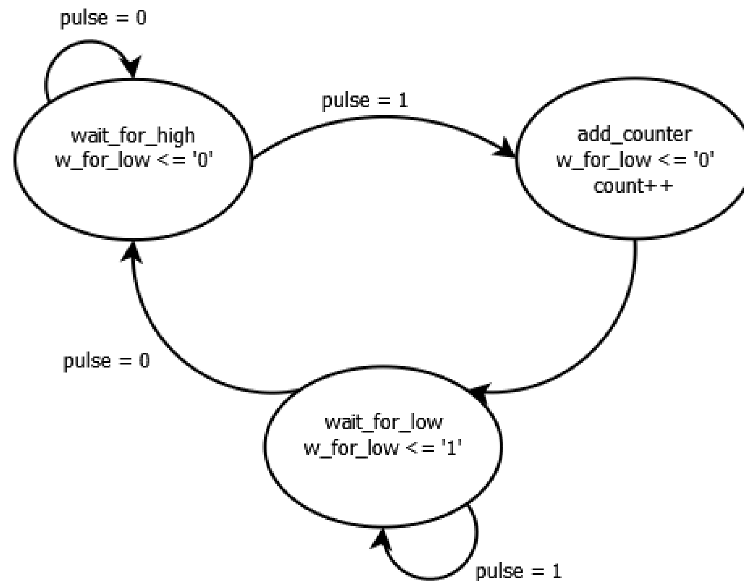

Figura 6-8. Diagrama de estados de counter.vhd

El XADC cuenta además con tres puertos de salida relacionado con JTAG (ver sección 5.2.1). Estos puertos se activan según qué acción esté realizando JTAG. Así que, por descartar que el problema no tuviese que ver con esto, se decidió contar el número de pulsos de la señal JTAGLOCKED. Idealmente, debía ser cero, pero lo cierto es que el contador devolvió un 1, con lo cual el XADC estaba siendo accedido de alguna forma a través de JTAG.

Puesto que el valor de la cuenta era solo 1, quería decir que solo era accedido en un momento puntual. Probablemente, al arranque del sistema. Lo siguiente era comprobar si esta señal se mantenía activa en 1, o solo se activaba durante un tiempo y luego volvía a nivel bajo. Para ello se añadió una pequeña funcionalidad al contador que, en caso de que la señal se quedará en nivel alto, activara una flag para avisar de ello (señal "w_for_low" en el diagrama de la Figura 6-8). Si JTAGLOCKED estaba siendo mantenido a nivel alto, era complementamente normal el funcionamiento anómalo del XADC. Pero lo cierto es que la señal, después de activar, volvía a nivel bajo.

Esto podía significar que, si Xillinux estaba accediendo al XADC por alguna razón, estaba reescribiendo los registros de configuración preinicializados por el asistente de configuración del XADC, dando lugar a un comportamiento no esperado del mismo.

Se comprobó que el XADC era accesible directamente desde Xillinux (acceso directo PS-XADC), con lo cual es probable que en el arranque del sistema se hiciera algún tipo de comprobación de todo el hardware y se produjera ese cambio en los registros anteriormente mencionado. Además, debía ser al arranque pues, momentáneamente la señal que controla el reset de la FIFO se puso a '0', de forma que no se reseteara cuando no se estaba leyendo y hacía un primer llenado correcto, y de ahí en adelante volvía a llenarse de datos incorrectos.

Se detecta que en devicetree.dtb existe un registro para el XADC, lo que permite el acceso directo desde Xillinux. Puesto que no era de ningún uso para este proyecto, se eliminó todo lo relacionado al XADC del devicetree, se recompiló y se volvieron a realizar las pruebas. Efectivamente, en el arranque Xillinux estaba modificando los registros del XADC y esto lo hacía porque estaba configurado como tal en el devicetree.dtb. Al eliminarlo de aquí, el XADC ya no es accesible directamente desde el PS, pero tampoco modifica los

registros de configuración, consiguiendo que se comporte de la manera deseada.

6.2.2 Alta latencia en la actualización de los datos de la FIFO

Tras solventar el error anterior, los datos se recibían correctamente, pero al leer la FIFO, si se variaba la tensión de entrada, los valores que devolvía la FIFO no parecían actualizarse. Había que dejar de leer (Ctrl + C) y volver a leer la FIFO para obtener datos actualizados.

Este comportamiento es debido a la configuración del dispositivo configruado para Xillybus. Al generar el IP Core en la web de Xillybus, si se le daba como uso al dispositivo "Data acquisition / playback", ocurre que los datos se almacenan en un buffer para evitar perder datos mientras no se leía la FIFO. En este caso, los datos que se pierdan mientras no se está leyendo no es de importancia, por eso en la sección 5.3.15.3.3 se escogió como uso "General purpose", de esta forma se evita este problema y se obtienen datos al poco tiempo de que se produzcan los cambios en la entrada.

7 CONCLUSIONES Y PROPUESTAS DE MEJORA

Se pretendía acercar los resultados de un electrocardiograma al usuario final haciendo uso de una FPGA con la capacidad de comunicarse con un sistema operativo. Después de llevar a cabo el proyecto, se ha visto que es completamente posible, si bien es cierto que presenta sus propias limitaciones.

La decisión de utilizar el menor número de componentes externos posibles, tratando de aprovechar lo que la Zynq-7000 (y la ZedBoard) tienen que ofrecer es lo que ha determinado que se optara por el convertidor analógico-digital que incluye la FPGA. Las facilidades que este ofrece son, principalmente, el hecho de ser un IP Core ampliamente probado por la comunidad de desarrolladores de Xilinx y las posibilidades de que se produzcan fallos críticos en el mismo son reducidas. Además, las opciones de configuración o personalización que ofrece a través del asistente de configuración son un gran añadido. El hecho de usar la ZedBoard ha facilitado aún más las cosas pues el XADC estaba listo para ser utilizado (no había que preocuparse por las alimentaciones, solo de facilitar las diferentes señales de entradas a convertir).

Por otro lado, Xillybus ha hecho posible la comunicación entre el XADC y el sistema operativo de forma tan simple como escribir en una FIFO. Aunque existen otros sistemas operativos que se podrían haber utilizado, habría supuesto tener que desarrollar los controladores y la lógica necesarios para que esta comunicación entre lógica programable y sistema de procesamiento fueran posible.

En cuanto a los impedimentos u obstáculos que puede presentar, uno de ellos, aunque de fácil solución, es la limitación que presenta la ZedBoard donde solo se pueden convertir 3 canales simultáneamente, mientras que para poder obtener un electrocardiograma típico debería ser capaz de monitorizar al menos 12 canales. La solución pasa por usar el modo de multiplexador externo, consiguiendo así acabar con el problema de los 3 canales.

La frecuencia de muestro sería otro de los incovenientes que presenta. Una tasa de muestreo de 40 KSPS es demasiado para señales que provienen del corazón, terminando con una cantidad enorme e innecesaria de muestras. Sin embargo, y si esto llegara a ser un problema, podría solucionarse desechando N-1 muestras por cada N muestras recibidas, de forma que en la FIFO se escribiesen menos muestras de las que el XADC produce. Esto es algo que se puede implementar en VHDL, como un bloque entre el XADC y la FIFO que lee Xillybus, y que no supondría un gran coste en cuanto a recursos.

Como propuesta de mejora o continuación del trabajo aquí realizado se propone:

- Desarrollar un amplificador que amplifique las señales cardíacas dentro de un rango aceptable para el XADC, y con unos electrodos, probar el sistema al completo (si no se usa un multiplexador externo,

bastaría con 4 electrodos para convertir las 3 derivaciones bipolares del plano frontal).

- Una aplicación en el sistema operativo que permita recoger los datos y los presente a través de una interfaz gráfica para un usuario, permitiendo acceso online (subiendo estos datos a un servidor web).
- Crear un filtro que ayude a eliminar las componentes de alta frecuencia y, en general, a mejorar la calidad de la señal recibida.
- Aprovechar el dispositivo de flujo de datos en dirección Host-FPGA que se incluyó en la creación del core de Xillybus pero al que no se le dio ningún uso para controlar de algún modo el funcionamiento de la lógica. Por ejemplo, cambiar el tipo de datos que se recibe, en lugar de la conversión del XADC, las pulsaciones detectadas.

ANEXO A: CÓDIGO DEL SISTEMA

En este anexo se recoge todo el código fuente usado en el proyecto y al que se ha hecho mención durante el desarrollo del mismo. Estarán organizados según el nombre que se le dio al fichero en el capítulo en el que aparece.

top_xadc.vhd

```
------------------------------------
-- Author: Pedro Gutierrez Lora
--
-- Create Date: 28.08.2017 13:24:52
-- Module Name: top_xadc - Behavioral
-- Project Name: Diseño de un electrocardiograma portatil
--
-- Additional Comments:
--
------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity top_xadc is
    port
    (
        clk     : IN std_logic;
        reset   : IN std_logic;
        vp_in   : IN  STD_LOGIC;
        vn_in   : IN  STD_LOGIC;
        vauxp0  : IN  STD_LOGIC;
        vauxn0  : IN  STD_LOGIC;
        vauxp8  : IN  STD_LOGIC;
        vauxn8  : IN  STD_LOGIC;
        jtagLMB : OUT std_logic_vector(2 downto 0);
        dout    : OUT std_logic_vector(15 downto 0);
        drdy    : OUT std_logic;
        channel : OUT std_logic_vector(4 downto 0)
    );
end top_xadc;

architecture Behavioral of top_xadc is
    component xadc_wiz_0 is
       port
       (
        daddr_in        : in  STD_LOGIC_VECTOR (6 downto 0);
```

```
        den_in          : in  STD_LOGIC;
        di_in           : in  STD_LOGIC_VECTOR (15 downto 0);
        dwe_in          : in  STD_LOGIC;
        do_out          : out  STD_LOGIC_VECTOR (15 downto 0);
        drdy_out        : out  STD_LOGIC;
        dclk_in         : in  STD_LOGIC;
        reset_in        : in  STD_LOGIC;
        jtagbusy_out    : out  STD_LOGIC;
        jtaglocked_out  : out  STD_LOGIC;
        jtagmodified_out : out  STD_LOGIC;
        vauxp0          : in  STD_LOGIC;
        vauxn0          : in  STD_LOGIC;
        vauxp8          : in  STD_LOGIC;
        vauxn8          : in  STD_LOGIC;
        busy_out        : out  STD_LOGIC;
        channel_out     : out  STD_LOGIC_VECTOR (4 downto 0);
        eoc_out         : out  STD_LOGIC;
        eos_out         : out  STD_LOGIC;
        alarm_out       : out STD_LOGIC;
        vp_in           : in  STD_LOGIC;
        vn_in           : in  STD_LOGIC
    );
    end component;

    -- señales auxiliares
    signal aux_dout    : std_logic_vector(15 downto 0);
    signal aux_channel : std_logic_vector(4 downto 0);
    signal aux_eoc     : std_logic;
    signal aux_daddr   : std_logic_vector(6 downto 0);
begin

    xadc : xadc_wiz_0
    port map (
        daddr_in         => aux_daddr,
        den_in           => aux_eoc,
        di_in            => (others => '0'),
        dwe_in           => '0',
        do_out           => aux_dout,
        drdy_out         => drdy,
        dclk_in          => clk,
        reset_in         => reset,
        jtagbusy_out     => jtagLMB(0),
        jtaglocked_out   => jtagLMB(2),
        jtagmodified_out => jtagLMB(1),
        vauxp0           => vauxp0,
        vauxn0           => vauxn0,
        vauxp8           => vauxp8,
        vauxn8           => vauxn8,
        busy_out         => open,
        channel_out      => aux_channel,
        eoc_out          => aux_eoc,
        eos_out          => open,
        alarm_out        => open,
        vp_in            => vp_in,
        vn_in            => vn_in
    );
```

```
    -- puesto que daddr es de 7 bits y channel es de 5,
    -- los 2 primeros bits serán siempre 0
    aux_daddr <= "00" & aux_channel;

    -- xillybus invierte el orden de los bytes, por eso
    -- hay que poner primero el byte menos significativo
    -- hasta el más significativo
    dout <= aux_dout(7 downto 0) & aux_dout(15 downto 8);

    -- se conecta la señal
    channel <= aux_channel;

end Behavioral;
```

xillybus.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity xillydemo is
  port (
    clk_100 : IN std_logic;
    otg_oc : IN std_logic;
    PS_GPIO : INOUT std_logic_vector(55 DOWNTO 0);
    GPIO_LED : OUT std_logic_vector(3 DOWNTO 0);
    vga4_blue : OUT std_logic_vector(3 DOWNTO 0);
    vga4_green : OUT std_logic_vector(3 DOWNTO 0);
    vga4_red : OUT std_logic_vector(3 DOWNTO 0);
    vga_hsync : OUT std_logic;
    vga_vsync : OUT std_logic;
    audio_mclk : OUT std_logic;
    audio_dac : OUT std_logic;
    audio_adc : IN std_logic;
    audio_bclk : IN std_logic;
    audio_lrclk : IN std_logic;
    smb_sclk : OUT std_logic;
    smb_sdata : INOUT std_logic;
    smbus_addr : OUT std_logic_vector(1 DOWNTO 0);

    -- entradas para el XADC
    vp_in   : IN  STD_LOGIC;
    vn_in   : IN  STD_LOGIC;
    vauxp0  : IN  STD_LOGIC;
    vauxn0  : IN  STD_LOGIC;
    vauxp8  : IN  STD_LOGIC;
    vauxn8  : IN  STD_LOGIC
    );
  end xillydemo;
```

```
 architecture sample_arch of xillydemo is
       component xillybus
           port (
             PS_CLK : IN std_logic;
             PS_PORB : IN std_logic;
             PS_SRSTB : IN std_logic;
             clk_100 : IN std_logic;
             otg_oc : IN std_logic;
             DDR_Addr : INOUT std_logic_vector(14 DOWNTO 0);
             DDR_BankAddr : INOUT std_logic_vector(2 DOWNTO 0);
             DDR_CAS_n : INOUT std_logic;
             DDR_CKE : INOUT std_logic;
             DDR_CS_n : INOUT std_logic;
             DDR_Clk : INOUT std_logic;
             DDR_Clk_n : INOUT std_logic;
             DDR_DM : INOUT std_logic_vector(3 DOWNTO 0);
             DDR_DQ : INOUT std_logic_vector(31 DOWNTO 0);
             DDR_DQS : INOUT std_logic_vector(3 DOWNTO 0);
             DDR_DQS_n : INOUT std_logic_vector(3 DOWNTO 0);
             DDR_DRSTB : INOUT std_logic;
             DDR_ODT : INOUT std_logic;
             DDR_RAS_n : INOUT std_logic;
             DDR_VRN : INOUT std_logic;
             DDR_VRP : INOUT std_logic;
             MIO : INOUT std_logic_vector(53 DOWNTO 0);
             PS_GPIO : INOUT std_logic_vector(55 DOWNTO 0);
             DDR_WEB : OUT std_logic;
             GPIO_LED : OUT std_logic_vector(3 DOWNTO 0);
             bus_clk : OUT std_logic;
             quiesce : OUT std_logic;
             vga4_blue : OUT std_logic_vector(3 DOWNTO 0);
             vga4_green : OUT std_logic_vector(3 DOWNTO 0);
             vga4_red : OUT std_logic_vector(3 DOWNTO 0);
             vga_hsync : OUT std_logic;
             vga_vsync : OUT std_logic;
             user_w_controlstream_wren : OUT std_logic;
             user_w_controlstream_full : IN std_logic;
             user_w_controlstream_data   :   OUT   std_logic_vector(7
DOWNTO 0);
             user_w_controlstream_open : OUT std_logic;
             user_r_datastream_rden : OUT std_logic;
             user_r_datastream_empty : IN std_logic;
             user_r_datastream_data  :  IN  std_logic_vector(15  DOWNTO
0);
             user_r_datastream_eof : IN std_logic;
             user_r_datastream_open : OUT std_logic;
             user_clk : OUT std_logic;
             user_wren : OUT std_logic;
             user_rden : OUT std_logic;
             user_wstrb : OUT std_logic_vector(3 DOWNTO 0);
             user_addr : OUT std_logic_vector(31 DOWNTO 0);
             user_rd_data : IN std_logic_vector(31 DOWNTO 0);
             user_wr_data : OUT std_logic_vector(31 DOWNTO 0);
             user_irq : IN std_logic);
      end component;
```

```
      component top_xadc is
          port (
              clk     : IN std_logic;
              reset   : IN std_logic;
              vp_in   : IN  STD_LOGIC;
              vn_in   : IN  STD_LOGIC;
              vauxp0  : IN  STD_LOGIC;
              vauxn0  : IN  STD_LOGIC;
              vauxp8  : IN  STD_LOGIC;
              vauxn8  : IN  STD_LOGIC;
              jtagLMB : OUT std_logic_vector(2 downto 0);
              dout    : OUT std_logic_vector(15 downto 0);
              drdy    : OUT std_logic;
              channel : OUT std_logic_vector(4 downto 0)
          );
      end component;

      component fifo is
          port (
              clk : IN STD_LOGIC;
              srst : IN STD_LOGIC;
              din : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
              wr_en : IN STD_LOGIC;
              rd_en : IN STD_LOGIC;
              dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
              full : OUT STD_LOGIC;
              empty : OUT STD_LOGIC
          );
      end component;

      signal bus_clk :  std_logic;
      signal quiesce : std_logic;

      signal user_w_controlstream_wren :  std_logic;
      signal user_w_controlstream_full :  std_logic;
      signal  user_w_controlstream_data  :    std_logic_vector(7  DOWNTO
0);
      signal user_w_controlstream_open :  std_logic;
      signal user_r_datastream_rden :  std_logic;
      signal user_r_datastream_empty :  std_logic;
      signal user_r_datastream_data :  std_logic_vector(15 DOWNTO 0);
      signal user_r_datastream_eof :  std_logic;
      signal user_r_datastream_open :  std_logic;
      signal user_clk :  std_logic;
      signal user_wren :  std_logic;
      signal user_rden :  std_logic;
      signal user_wstrb :  std_logic_vector(3 DOWNTO 0);
      signal user_addr :  std_logic_vector(31 DOWNTO 0);
      signal user_rd_data :  std_logic_vector(31 DOWNTO 0);
      signal user_wr_data :  std_logic_vector(31 DOWNTO 0);
      signal user_irq :  std_logic;

      -- Note that none of the ARM processor's direct connections to
pads is
      -- defined as I/O on this module. Normally, they should be
connected
```

```
      -- as toplevel ports here, but that confuses Vivado 2013.4 to
think that
      -- some of these ports are real I/Os, causing an implementation
failure.
      -- This detachment results in a lot of warnings during synthesis
and
      -- implementation, but has no practical significance, as these
pads are
      -- completely unrelated to the FPGA bitstream.

      signal PS_CLK :  std_logic;
      signal PS_PORB :  std_logic;
      signal PS_SRSTB :  std_logic;
      signal DDR_Addr : std_logic_vector(14 DOWNTO 0);
      signal DDR_BankAddr : std_logic_vector(2 DOWNTO 0);
      signal DDR_CAS_n : std_logic;
      signal DDR_CKE : std_logic;
      signal DDR_CS_n : std_logic;
      signal DDR_Clk : std_logic;
      signal DDR_Clk_n : std_logic;
      signal DDR_DM : std_logic_vector(3 DOWNTO 0);
      signal DDR_DQ : std_logic_vector(31 DOWNTO 0);
      signal DDR_DQS : std_logic_vector(3 DOWNTO 0);
      signal DDR_DQS_n : std_logic_vector(3 DOWNTO 0);
      signal DDR_DRSTB : std_logic;
      signal DDR_ODT : std_logic;
      signal DDR_RAS_n : std_logic;
      signal DDR_VRN : std_logic;
      signal DDR_VRP : std_logic;
      signal MIO : std_logic_vector(53 DOWNTO 0);
      signal DDR_WEB : std_logic;

      -- señal para FIFO
      signal reset_fifo : std_logic;

      -- señales auxiliares entre XADC y FIFO
      signal xadc_dout_fifo_din : std_logic_vector(15 downto 0);
      signal xadc_drdy_fifo_wren : std_logic;
 begin
      xillybus_ins : xillybus
      port map (
          -- Ports related to /dev/xillybus_controlstream
          -- CPU to FPGA signals:
          user_w_controlstream_wren => user_w_controlstream_wren,
          user_w_controlstream_full => user_w_controlstream_full,
          user_w_controlstream_data => user_w_controlstream_data,
          user_w_controlstream_open => user_w_controlstream_open,

          -- Ports related to /dev/xillybus_datastream
          -- FPGA to CPU signals:
          user_r_datastream_rden => user_r_datastream_rden,
          user_r_datastream_empty => user_r_datastream_empty,
          user_r_datastream_data => user_r_datastream_data,
          user_r_datastream_eof => user_r_datastream_eof,
          user_r_datastream_open => user_r_datastream_open,
```

```
        -- Ports related to Xillybus Lite
        user_clk => user_clk,
        user_wren => user_wren,
        user_rden => user_rden,
        user_wstrb => user_wstrb,
        user_addr => user_addr,
        user_rd_data => user_rd_data,
        user_wr_data => user_wr_data,
        user_irq => user_irq,

        -- General signals
        PS_CLK => PS_CLK,
        PS_PORB => PS_PORB,
        PS_SRSTB => PS_SRSTB,
        clk_100 => clk_100,
        otg_oc => otg_oc,
        DDR_Addr => DDR_Addr,
        DDR_BankAddr => DDR_BankAddr,
        DDR_CAS_n => DDR_CAS_n,
        DDR_CKE => DDR_CKE,
        DDR_CS_n => DDR_CS_n,
        DDR_Clk => DDR_Clk,
        DDR_Clk_n => DDR_Clk_n,
        DDR_DM => DDR_DM,
        DDR_DQ => DDR_DQ,
        DDR_DQS => DDR_DQS,
        DDR_DQS_n => DDR_DQS_n,
        DDR_DRSTB => DDR_DRSTB,
        DDR_ODT => DDR_ODT,
        DDR_RAS_n => DDR_RAS_n,
        DDR_VRN => DDR_VRN,
        DDR_VRP => DDR_VRP,
        MIO => MIO,
        PS_GPIO => PS_GPIO,
        DDR_WEB => DDR_WEB,
        GPIO_LED => GPIO_LED,
        bus_clk => bus_clk,
        quiesce => quiesce,
        vga4_blue => vga4_blue,
        vga4_green => vga4_green,
        vga4_red => vga4_red,
        vga_hsync => vga_hsync,
        vga_vsync => vga_vsync
    );

    xadc : top_xadc
        port map (
            clk     => bus_clk,
            reset   => '0',
            vp_in   => vp_in,
            vn_in   => vn_in,
            vauxp0  => vauxp0,
            vauxn0  => vauxn0,
            vauxp8  => vauxp8,
            vauxn8  => vauxn8,
            jtagLMB => open,
            dout    => xadc_dout_fifo_din,
```

```
            drdy    => xadc_drdy_fifo_wren,
            channel => open
        );

    fifo16 : fifo
        port map (
            clk => bus_clk,
            srst => reset_fifo,
            din => xadc_dout_fifo_din,
            wr_en => xadc_drdy_fifo_wren,
            rd_en => user_r_datastream_rden,
            dout => user_r_datastream_data,
            full => open,
            empty => user_r_datastream_empty
        );

        -- reseteamos la FIFO siempre que no se esté leyendo
        reset_fifo <= not(user_r_datastream_open);

        -- nunca se alcanza el end of file
        user_r_datastream_eof <= '0';

end sample_arch;
```

xillybus.xdc

```
create_clock -name gclk -period 10 [get_ports "clk_100"]
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets "clk_100"]

# Vivado constraints unrelated clocks. So set false paths.
set_false_path -from [get_clocks clk_fpga_1] -to [get_clocks
vga_clk_ins/*]
set_false_path -from [get_clocks vga_clk_ins/*] -to [get_clocks
clk_fpga_1]

# The VGA outputs are turned into an analog voltage by virtue of a
resistor
# network, so the flip flops driving these must sit in the IOBs to
minimize
# timing skew. The RTL code should handle this, but the constraint
below
# is there to fail if something goes wrong about this.
set_output_delay 5.5 [get_ports {vga*}]

set_property -dict "PACKAGE_PIN Y9 IOSTANDARD LVCMOS33" [get_ports
"clk_100"]
set_property -dict "PACKAGE_PIN T22 IOSTANDARD LVCMOS33" [get_ports
"GPIO_LED[0]"]
set_property -dict "PACKAGE_PIN T21 IOSTANDARD LVCMOS33" [get_ports
"GPIO_LED[1]"]
set_property -dict "PACKAGE_PIN U22 IOSTANDARD LVCMOS33" [get_ports
"GPIO_LED[2]"]
```

```
 set_property -dict "PACKAGE_PIN U21 IOSTANDARD LVCMOS33" [get_ports
"GPIO_LED[3]"]
 set_property -dict "PACKAGE_PIN Y21 IOSTANDARD LVCMOS33" [get_ports
"vga4_blue[0]"]
 set_property -dict "PACKAGE_PIN Y20 IOSTANDARD LVCMOS33" [get_ports
"vga4_blue[1]"]
 set_property -dict "PACKAGE_PIN AB20 IOSTANDARD LVCMOS33" [get_ports
"vga4_blue[2]"]
 set_property -dict "PACKAGE_PIN AB19 IOSTANDARD LVCMOS33" [get_ports
"vga4_blue[3]"]
 set_property -dict "PACKAGE_PIN AB22 IOSTANDARD LVCMOS33" [get_ports
"vga4_green[0]"]
 set_property -dict "PACKAGE_PIN AA22 IOSTANDARD LVCMOS33" [get_ports
"vga4_green[1]"]
 set_property -dict "PACKAGE_PIN AB21 IOSTANDARD LVCMOS33" [get_ports
"vga4_green[2]"]
 set_property -dict "PACKAGE_PIN AA21 IOSTANDARD LVCMOS33" [get_ports
"vga4_green[3]"]
 set_property -dict "PACKAGE_PIN V20 IOSTANDARD LVCMOS33" [get_ports
"vga4_red[0]"]
 set_property -dict "PACKAGE_PIN U20 IOSTANDARD LVCMOS33" [get_ports
"vga4_red[1]"]
 set_property -dict "PACKAGE_PIN V19 IOSTANDARD LVCMOS33" [get_ports
"vga4_red[2]"]
 set_property -dict "PACKAGE_PIN V18 IOSTANDARD LVCMOS33" [get_ports
"vga4_red[3]"]
 set_property -dict "PACKAGE_PIN Y19 IOSTANDARD LVCMOS33" [get_ports
"vga_vsync"]
 set_property -dict "PACKAGE_PIN AA19 IOSTANDARD LVCMOS33" [get_ports
"vga_hsync"]

 # IMPORTANT: Since four LEDs are taken by the Xillybus IP core, the
pin
 # placement doesn't match the one given by Digilent.

 # GPIO pin to reset the USB OTG PHY

 set_property -dict "PACKAGE_PIN G17 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[0]"]

 # On-board OLED

 set_property -dict "PACKAGE_PIN U11 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[1]"]
 set_property -dict "PACKAGE_PIN U12 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[2]"]
 set_property -dict "PACKAGE_PIN U9 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[3]"]
 set_property -dict "PACKAGE_PIN U10 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[4]"]
 set_property -dict "PACKAGE_PIN AB12 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[5]"]
 set_property -dict "PACKAGE_PIN AA12 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[6]"]

 # On-board LEDs. Note that only for LEDs are allocated, as opposed
to
```

```
 # Digilent's eight, and all placements that follow are shifted by four.
 # There was no other choice, as the tools don't allow unplaced PS GPIO pins.

 set_property -dict "PACKAGE_PIN V22 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[7]"]
 set_property -dict "PACKAGE_PIN W22 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[8]"]
 set_property -dict "PACKAGE_PIN U19 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[9]"]
 set_property -dict "PACKAGE_PIN U14 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[10]"]

 # On-board Slide Switches

 set_property -dict "PACKAGE_PIN F22 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[11]"]
 set_property -dict "PACKAGE_PIN G22 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[12]"]
 set_property -dict "PACKAGE_PIN H22 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[13]"]
 set_property -dict "PACKAGE_PIN F21 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[14]"]
 set_property -dict "PACKAGE_PIN H19 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[15]"]
 set_property -dict "PACKAGE_PIN H18 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[16]"]
 set_property -dict "PACKAGE_PIN H17 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[17]"]
 set_property -dict "PACKAGE_PIN M15 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[18]"]

 # On-board Left, Right, Up, Down, and Select Pushbuttons

 set_property -dict "PACKAGE_PIN N15 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[19]"]
 set_property -dict "PACKAGE_PIN R18 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[20]"]
 set_property -dict "PACKAGE_PIN T18 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[21]"]
 set_property -dict "PACKAGE_PIN R16 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[22]"]
 set_property -dict "PACKAGE_PIN P16 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[23]"]

 # Pmod JA

 set_property -dict "PACKAGE_PIN Y11 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[24]"]
 set_property -dict "PACKAGE_PIN AA11 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[25]"]
 set_property -dict "PACKAGE_PIN Y10 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[26]"]
 set_property -dict "PACKAGE_PIN AA9 IOSTANDARD LVCMOS33" [get_ports "PS_GPIO[27]"]
```

```
 set_property -dict "PACKAGE_PIN AB11 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[28]"]
 set_property -dict "PACKAGE_PIN AB10 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[29]"]
 set_property -dict "PACKAGE_PIN AB9 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[30]"]
 set_property -dict "PACKAGE_PIN AA8 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[31]"]

 # Pmod JB

 set_property -dict "PACKAGE_PIN W12 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[32]"]
 set_property -dict "PACKAGE_PIN W11 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[33]"]
 set_property -dict "PACKAGE_PIN V10 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[34]"]
 set_property -dict "PACKAGE_PIN W8 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[35]"]
 set_property -dict "PACKAGE_PIN V12 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[36]"]
 set_property -dict "PACKAGE_PIN W10 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[37]"]
 set_property -dict "PACKAGE_PIN V9 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[38]"]
 set_property -dict "PACKAGE_PIN V8 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[39]"]

 # Pmod JC

 set_property -dict "PACKAGE_PIN AB7 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[40]"]
 set_property -dict "PACKAGE_PIN AB6 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[41]"]
 set_property -dict "PACKAGE_PIN Y4 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[42]"]
 set_property -dict "PACKAGE_PIN AA4 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[43]"]
 set_property -dict "PACKAGE_PIN R6 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[44]"]
 set_property -dict "PACKAGE_PIN T6 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[45]"]
 set_property -dict "PACKAGE_PIN T4 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[46]"]
 set_property -dict "PACKAGE_PIN U4 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[47]"]

 # Pmod JD

 set_property -dict "PACKAGE_PIN V7 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[48]"]
 set_property -dict "PACKAGE_PIN W7 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[49]"]
 set_property -dict "PACKAGE_PIN V5 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[50]"]
 set_property -dict "PACKAGE_PIN V4 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[51]"]
```

```
 set_property -dict "PACKAGE_PIN W6 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[52]"]
 set_property -dict "PACKAGE_PIN W5 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[53]"]
 set_property -dict "PACKAGE_PIN U6 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[54]"]
 set_property -dict "PACKAGE_PIN U5 IOSTANDARD LVCMOS33" [get_ports
"PS_GPIO[55]"]

 # Pin for detecting USB OTG over-current condition

 set_property -dict "PACKAGE_PIN L16 IOSTANDARD LVCMOS33" [get_ports
"otg_oc"]

 # Pins connected to sound chip
 set_property -dict "PACKAGE_PIN AB1 IOSTANDARD LVCMOS33" [get_ports
"smbus_addr[0]"]
 set_property -dict "PACKAGE_PIN Y5 IOSTANDARD LVCMOS33" [get_ports
"smbus_addr[1]"]
 set_property -dict "PACKAGE_PIN AB4 IOSTANDARD LVCMOS33" [get_ports
"smb_sclk"]
 set_property -dict "PACKAGE_PIN AB5 IOSTANDARD LVCMOS33" [get_ports
"smb_sdata"]

 set_property -dict "PACKAGE_PIN Y8 IOSTANDARD LVCMOS33" [get_ports
"audio_dac"]
 set_property -dict "PACKAGE_PIN AA7 IOSTANDARD LVCMOS33" [get_ports
"audio_adc"]
 set_property -dict "PACKAGE_PIN AA6 IOSTANDARD LVCMOS33" [get_ports
"audio_bclk"]
 set_property -dict "PACKAGE_PIN Y6 IOSTANDARD LVCMOS33" [get_ports
"audio_lrclk"]
 set_property -dict "PACKAGE_PIN AB2 IOSTANDARD LVCMOS33" [get_ports
"audio_mclk"]

 # Pins para el XADC

 set_property -dict "PACKAGE_PIN L11 IOSTANDARD LVCMOS33" [get_ports
"vp_in"]
 set_property -dict "PACKAGE_PIN M12 IOSTANDARD LVCMOS33" [get_ports
"vn_in"]

 set_property -dict "PACKAGE_PIN F16 IOSTANDARD LVCMOS33" [get_ports
"vauxp0"]
 set_property -dict "PACKAGE_PIN E16 IOSTANDARD LVCMOS33" [get_ports
"vauxn0"]

 set_property -dict "PACKAGE_PIN D16 IOSTANDARD LVCMOS33" [get_ports
"vauxp8"]
 set_property -dict "PACKAGE_PIN D17 IOSTANDARD LVCMOS33" [get_ports
"vauxn8"]

 set_property BITSTREAM.GENERAL.JTAG_XADC Disable [current_design]
```

tb_top_xadc.vhd

```
------------------------------------
-- Author: Pedro Gutierrez Lora
--
-- Create Date: 28.08.2017 13:24:52
-- Module Name: tb_top_xadc - Behavioral
-- Project Name: Diseño de un electrocardiograma portatil
--
-- Additional Comments:
--
------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_top_xadc is
end tb_top_xadc;

architecture Behavioral of tb_top_xadc is
    component top_xadc is
        port (
            clk     : IN std_logic;
            reset   : IN std_logic;
            vp_in   : IN  STD_LOGIC;
            vn_in   : IN  STD_LOGIC;
            vauxp0  : IN  STD_LOGIC;
            vauxn0  : IN  STD_LOGIC;
            vauxp8  : IN  STD_LOGIC;
            vauxn8  : IN  STD_LOGIC;
            jtagLMB : OUT std_logic_vector(2 downto 0);
            dout    : OUT std_logic_vector(15 downto 0);
            drdy    : OUT std_logic;
            channel : OUT std_logic_vector(4 downto 0)
        );
    end component;

    -- señales auxiliares
    -- input
    signal clk     : std_logic := '0';
    signal reset   : std_logic := '0';
    signal vp_in   : STD_LOGIC;
    signal vn_in   : STD_LOGIC;
    signal vauxp0  : STD_LOGIC;
    signal vauxn0  : STD_LOGIC;
    signal vauxp8  : STD_LOGIC;
    signal vauxn8  : STD_LOGIC;

    -- output
    signal jtagLMB : std_logic_vector(2 downto 0);
    signal dout    : std_logic_vector(15 downto 0);
    signal drdy    : std_logic;
    signal channel : std_logic_vector(4 downto 0);

    -- Clock period definitions
```

```
    constant clk_period : time := 10 ns;
begin

    uut : top_xadc
        port map (
            clk     => clk,
            reset   => reset,
            vp_in   => vp_in,
            vn_in   => vn_in,
            vauxp0  => vauxp0,
            vauxn0  => vauxn0,
            vauxp8  => vauxp8,
            vauxn8  => vauxn8,
            jtagLMB => jtagLMB,
            dout    => dout,
            drdy    => drdy,
            channel => channel
        );

    -- Clock process definitions
    clk_process : process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        reset <= '1';
        wait for clk_period*5;

        reset <= '0';
        wait;

    end process;

end Behavioral;
```

envia_caract.vhd

```
------------------------------------
-- Author: Pedro Gutierrez Lora
--
-- Create Date: 28.08.2017 13:24:52
-- Module Name: envia_caract - Behavioral
-- Project Name: Diseño de un electrocardiograma portatil
--
-- Additional Comments:
```

```
--
------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity envia_caract is
    Port ( clk : in STD_LOGIC;
           srst : in STD_LOGIC;
           swi : in STD_LOGIC;
           swi2 : in STD_LOGIC;
           wr_en : out STD_LOGIC;
           caracter : out STD_LOGIC_VECTOR (15 downto 0));
end envia_caract;

architecture Behavioral of envia_caract is
type estado is (reposo,caract1,caract2,caract3,espera);
signal p_est,a_est: estado;
begin

fsm:process(a_est, swi)
begin
    p_est <= a_est;
    wr_en <= '0';
    caracter <= x"0000";

    case a_est is
        when reposo =>
            if (swi = '1') then
                p_est <= caract1;
            end if;

        when caract1 =>
            caracter <= x"0078";
            wr_en <= '1';
            p_est <= caract2;

        when caract2 =>
            caracter <= x"0079";
            wr_en <= '1';
            p_est <= caract3;

        when caract3 =>
            caracter <= x"007A";
            wr_en <= '1';
            p_est <= espera;

        when espera =>
            if (swi = '0' or swi = 'Z') then
                p_est <= reposo;
            end if;

    end case;
end process;

sinc:process(clk, srst)
```

```
begin
    if(rising_edge(clk)) then
        if (srst='1') then
            a_est <= reposo;
        else
            a_est <= p_est;
        end if;
    end if;
end process;
end Behavioral;
```

fifo.vhd

```
------------------------------------
-- Author: Pedro Gutierrez Lora
--
-- Create Date: 28.08.2017 13:24:52
-- Module Name: envia_caract - Behavioral
-- Project Name: Diseño de un electrocardiograma portatil
--
-- Additional Comments:
--
------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity top_xadc is
    port
    (
        clk     : IN std_logic;
        reset   : IN std_logic;
        vp_in   : IN  STD_LOGIC;
        vn_in   : IN  STD_LOGIC;
        vauxp0  : IN  STD_LOGIC;
        vauxn0  : IN  STD_LOGIC;
        vauxp8  : IN  STD_LOGIC;
        vauxn8  : IN  STD_LOGIC;
        jtagLMB : OUT std_logic_vector(2 downto 0);
        dout    : OUT std_logic_vector(15 downto 0);
        drdy    : OUT std_logic;
        channel : OUT std_logic_vector(4 downto 0)
    );
end top_xadc;

architecture Behavioral of top_xadc is
    component xadc_wiz_0 is
       port
       (
        daddr_in        : in  STD_LOGIC_VECTOR (6 downto 0);
        den_in          : in  STD_LOGIC;
```

```
        di_in            : in  STD_LOGIC_VECTOR (15 downto 0);
        dwe_in           : in  STD_LOGIC;
        do_out           : out  STD_LOGIC_VECTOR (15 downto 0);
        drdy_out         : out  STD_LOGIC;
        dclk_in          : in  STD_LOGIC;
        reset_in         : in  STD_LOGIC;
        jtagbusy_out     : out  STD_LOGIC;
        jtaglocked_out   : out  STD_LOGIC;
        jtagmodified_out : out  STD_LOGIC;
        vauxp0           : in  STD_LOGIC;
        vauxn0           : in  STD_LOGIC;
        vauxp8           : in  STD_LOGIC;
        vauxn8           : in  STD_LOGIC;
        busy_out         : out  STD_LOGIC;
        channel_out      : out  STD_LOGIC_VECTOR (4 downto 0);
        eoc_out          : out  STD_LOGIC;
        eos_out          : out  STD_LOGIC;
        alarm_out        : out STD_LOGIC;
        vp_in            : in  STD_LOGIC;
        vn_in            : in  STD_LOGIC
    );
    end component;

    -- señales auxiliares
    signal aux_dout    : std_logic_vector(15 downto 0);
    signal aux_channel : std_logic_vector(4 downto 0);
    signal aux_eoc     : std_logic;
    signal aux_daddr   : std_logic_vector(6 downto 0);
begin

    xadc : xadc_wiz_0
    port map (
        daddr_in          => aux_daddr,
        den_in            => aux_eoc,
        di_in             => (others => '0'),
        dwe_in            => '0',
        do_out            => aux_dout,
        drdy_out          => drdy,
        dclk_in           => clk,
        reset_in          => reset,
        jtagbusy_out      => jtagLMB(0),
        jtaglocked_out    => jtagLMB(2),
        jtagmodified_out => jtagLMB(1),
        vauxp0            => vauxp0,
        vauxn0            => vauxn0,
        vauxp8            => vauxp8,
        vauxn8            => vauxn8,
        busy_out          => open,
        channel_out       => aux_channel,
        eoc_out           => aux_eoc,
        eos_out           => open,
        alarm_out         => open,
        vp_in             => vp_in,
        vn_in             => vn_in
    );

    -- puesto que daddr es de 7 bits y channel es de 5,
```

```
    -- los 2 primeros bits serán siempre 0
    aux_daddr <= "00" & aux_channel;

    -- xillybus invierte el orden de los bytes, por eso
    -- hay que poner primero el byte menos significativo
    -- hasta el más significativo
    dout <= aux_dout(7 downto 0) & aux_dout(15 downto 8);

    -- se conecta la señal
    channel <= aux_channel;

end Behavioral;
```

tb_fifo.vhd

```
------------------------------------
-- Author: Pedro Gutierrez Lora
--
-- Create Date: 28.08.2017 13:24:52
-- Module Name: tb_fifo - Behavioral
-- Project Name: Diseño de un electrocardiograma portatil
--
-- Additional Comments:
--
------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY tb_fifo IS
END tb_fifo;

ARCHITECTURE behavior OF tb_fifo IS
    component fifo_generator_0 IS
        PORT (
            clk : IN STD_LOGIC;
            srst : IN STD_LOGIC;
            din : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
            wr_en : IN STD_LOGIC;
            rd_en : IN STD_LOGIC;
            dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
            full : OUT STD_LOGIC;
            empty : OUT STD_LOGIC
        );
    END component;

    -- señales auxiliares
    -- input
    signal clk    : STD_LOGIC := '0';
    signal srst   : STD_LOGIC := '0';
    signal din    : STD_LOGIC_VECTOR(15 DOWNTO 0) := x"0000";
```

```
    signal wr_en  : STD_LOGIC := '0';
    signal rd_en  : STD_LOGIC := '0';

    -- output
    signal dout   : STD_LOGIC_VECTOR(15 DOWNTO 0);
    signal full   : STD_LOGIC;
    signal empty  : STD_LOGIC;

    -- Clock period definitions
    constant clk_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut : fifo_generator_0
        PORT MAP (
            clk => clk,
            srst => srst,
            din => din,
            wr_en => wr_en,
            rd_en => rd_en,
            dout => dout,
            full => full,
            empty => empty
         );

    -- Clock process definitions
    clk_process : process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;


    -- Stimulus process
    stim_proc: process
    begin
        srst <= '1';
        wait for clk_period*5;

        srst <= '0';
        wr_en <= '1';
        din <= x"AAAA";
        wait for clk_period;

        wr_en <= '0';
        wait for clk_period;

        wr_en <= '1';
        din <= x"BBBB";
        wait for clk_period;

        wr_en <= '0';
        wait for clk_period;
```

```
        wr_en <= '1';
        din <= x"CCCC";
        wait for clk_period;

        wr_en <= '0';
        rd_en <= '1';
        wait for clk_period;

        rd_en <= '0';
        wait;
    end process;

END;
```

leerBinario.py

```
import os
import binascii

with open('output.xadc', mode='rb') as file:
    # si quisieramos leer el fichero al completo
    #fileContent = file.read()
    #print(fileContent)

    #obtenemos el tamanio del fichero en bytes
    size_file = os.path.getsize('output.xadc')
    #print(size_file)

    # asignamos los diferentes canales:
    vpvn = []
    vaux0 = []
    vaux8 = []

    # recorremos el fichero
    count_channel = 0
    count = 0
    while (count < size_file):
        # vamos a una posicion y leemos 2 bytes
        file.seek(count)
        couple_bytes = file.read(2)
        hex_data = binascii.hexlify(couple_bytes)
        text_string = hex_data.decode('utf-8')
        #print(text_string)

        if count_channel == 0:
            vpvn.append(text_string)
            count_channel = 1
        elif count_channel == 1:
            vaux0.append(text_string)
            count_channel = 2
        else:
            vaux8.append(text_string)
```

```
              count_channel = 0

          count=count+2

  #print vpvn
  #print vaux0
  #print vaux8

  # se escribe el nuevo fichero formateado en hexadecimal
  with open('output.format', mode='w') as file_wr:
      size_of_lists = [len(vpvn), len(vaux0), len(vaux8)]
      max_size = max(size_of_lists)
      aux = '0000'

      count_elements = 0
      while (count_elements < max_size):
          # esto es para que cuando se llegue al final del fichero
          # si el tamano de las muestras de cada canal no es igual,
          # se rellene con ceros
          try:
              vpvn_actual = vpvn[count_elements]
          except IndexError:
              vpvn_actual = aux

          try:
              vaux0_actual = vaux0[count_elements]
          except IndexError:
              vaux0_actual = aux

          try:
              vaux8_actual = vaux8[count_elements]
          except IndexError:
              vaux8_actual = aux

          file_wr.write("{}      {}      {}\n".format(vpvn_actual[0:3],
vaux0_actual[0:3], vaux8_actual[0:3]))
          count_elements = count_elements + 1
          if (count_elements == max_size):
              print('Se ha convertido el fichero')
```

conversion.m

```
function res = conversion(value)
    valor=hex2dec(value);
    if (valor <= 2047)
        res=0.244*valor;
    else
        res=(4095-valor+1)*(-0.244);
    end
end
```

leerHexadecimal.m

Depende de conversion.m

```
close all
clear all

raw = fileread('output.format');
lines = strsplit(raw, '\n');
lines(:,end) = [];

vpvn = zeros(1, length(lines));
vaux0 = zeros(1, length(lines));
vaux8 = zeros(1, length(lines));

for i=1:length(lines)
    cols = strsplit(lines{1,i}, ' ');
    vpvn(1,i) = conversion(deblank(cols(1,1)));
    vaux0(1,i) = conversion(deblank(cols(1,2)));
    vaux8(1,i) = conversion(deblank(cols(1,3)));
end

t=0:1:length(vpvn)-1;

subplot(1,3,1);
plot(t,vpvn);
title('Canal dedicado VP/VN');
xlabel('Tiempo');
ylabel('Tensión (mV)');

subplot(1,3,2);
plot(t,vaux0);
title('Canal auxiliar Vaux0');
xlabel('Tiempo');
ylabel('Tensión (mV)');

subplot(1,3,3);
plot(t,vaux8);
title('Canal auxiliar Vaux8');
xlabel('Tiempo');
ylabel('Tensión (mV)');
```

REFERENCIAS

[1] McGill Physiology Virtual Laboratory, «Cardiovascular Lab: Electrocardiogram: Introduction,» [En línea]. Available: http://www.medicine.mcgill.ca/physio/vlab/cardio/introECG.htm. [Último acceso: 29 Julio 2017].

[2] J. A. Tresguerres, Fisiología Humana, McGraw-Hill Interamericana de España S.L., 2010.

[3] My EKG, «Electrodos del Electrocardiograma,» [En línea]. Available: http://www.my-ekg.com/generalidades-ekg/electrodos-ekg.html. [Último acceso: 29 Julio 2017].

[4] Xilinx, «Zynq-7000 All Programmable SoC,» 27 Septiembre 2016. [En línea]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. [Último acceso: 10 Enero 2017].

[5] Xilinx, «7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter,» 27 Septiembre 2016. [En línea]. Available: https://www.xilinx.com/support/documentation/user_guides/ug480_7Series_XADC.pdf. [Último acceso: 28 Agosto 2017].

[6] Xilinx, «AXI Reference Guide,» 7 Marzo 2011. [En línea]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf. [Último acceso: 10 Agosto 2017].

[7] Xilinx, «Driving the Xilinx Analog-to-Digital Converter,» 24 Febrero 2015. [En línea]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp795-driving-xadc.pdf. [Último acceso: 2 Agosto 2017].

[8] Xilinx, «PetaLinux,» [En línea]. Available: http://www.wiki.xilinx.com/PetaLinux. [Último acceso: 30 Julio 2017].

[9] Wind River Systems, «VxWorks,» [En línea]. Available: https://www.windriver.com/products/vxworks/. [Último acceso: 30 Julio 2017].

[10] Xillybus, «IP core product brief,» 16 Febrero 2017. [En línea]. Available: http://xillybus.com/downloads/xillybus_product_brief.pdf. [Último acceso: 25 Julio 2017].

[11] Xillybus, «Getting started with the FPGA demo bundle for Xilinx,» [En línea]. Available: http://xillybus.com/downloads/doc/xillybus_getting_started_xilinx.pdf. [Último acceso: 20 Mayo 2017].

[12] Xilinx, «XADC Wizard v3.0,» 1 Abril 2015. [En línea]. Available: https://www.xilinx.com/support/documentation/ip_documentation/xadc_wiz/v3_0/pg091-xadc-wiz.pdf. [Último acceso: 29 Agosto 2017].

[13] Xillybus, «The guide to defining a custom Xillybus IP core,» [En línea]. Available: http://xillybus.com/downloads/doc/xillybus_custom_ip.pdf. [Último acceso: 20 Julio 2017].

[14] Xilinx, «FIFO Generator v13.1,» 5 Abril 2017. [En línea]. Available: https://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v13_1/pg057-fifo-generator.pdf. [Último acceso: 12 Abril 2017].

[15] Xilinx, «Vivado Design Suite User Guide, Using Constraints,» 4 Septiembre 2012. [En línea]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug903-vivado-using-constraints.pdf. [Último acceso: 22 Agosto 2017].

[16] Avnet, «ZedBoard (Zynq Evaluation and Development) Hardware User's Guide,» 27 Enero 2014. [En línea]. Available: http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf. [Último acceso: 20 Agosto 2017].

[17] Xilinx, «Vivado Design Suite User Guide, Programming and Debugging,» 30 Mayo 2014. [En línea]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug908-vivado-programming-debugging.pdf. [Último acceso: 30 Agosto 2017].

[18] Xilinx, «System Monitor and XADC,» [En línea]. Available: https://www.xilinx.com/products/technology/analog-mixed-signal.html. [Último acceso: 30 Agosto 2017].

[19] Xillybus, «Trying out the Xillybus interface,» [En línea]. Available: http://xillybus.com/doc/microblaze-xillybus-trivial. [Último acceso: 23 Julio 2017].

[20] Xilinx, «Integrated Logic Analyzer v6.1,» 6 Abril 2016. [En línea]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_1/pg172-ila.pdf. [Último acceso: 29 Agosto 2017].