Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación
Intensificación en Sistemas Electrónicos

Codiseño HW/SW en System-on-Chip programable de última generación

Autor: Jesús Fernández Manzano
Tutor: Hipólito Guzmán Miranda

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

Sevilla, 2016

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

Codiseño HW/SW en System-on-Chip programable de última generación

Autor:
Jesús Fernández Manzano

Tutor:
Hipólito Guzmán Miranda
Profesor Contratado Doctor

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado: Codiseño HW/SW en System-on-Chip programable de última generación

Autor: Jesús Fernández Manzano

Tutor: 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, 2016

El Secretario del Tribunal

Agradecimientos

Quisiera agradecer en primer lugar a mi tutor Hipólito Guzmán Miranda por darme la oportunidad de llevar a cabo este trabajo y por haberme atendido cuando ha hecho falta. Al resto de los profesores que me han enseñado tantas cosas desde el colegio hasta la universidad.

También quiero agradecer a mi familia, sin la cual no podría haber llegado hasta aquí. Padres, hermanos, tíos, primos y abuelos. A todos los amigos que he conocido en Sevilla y me han acompañado durante estos cuatro años de carrera.

Y por último, una mención a mis compañeros de clase, en especial al Tridente Electrónico y a Julio por ser con los que más tiempo, proyectos, prácticas y momentos he compartido, haciendo que toda la carrera sea más llevadera.

Resumen

Este trabajo se centra en el codiseño HW/SW de un PSoC (Programmable System-on-Chip) formado por un sistema de procesamiento (PS) y lógica programable (PL). El codiseño se compone de varios pasos, como la elección de un sistema operativo para el SoC, la programación de la FPGA y lo más importante, la definición de la interfaz de comunicación entre ambos.

Habitualmente, la interfaz de comunicación entre software y hardware es el punto más delicado en un diseño, el que más tiempo requiere y el lugar más a propenso a contener errores. Para solucionar este problema se va a hacer uso de Xillybus, que consiste en un conjunto de herramientas que hacen más transparente al desarrollador la comunicación HW/SW. Con esto se consigue una comunicación extremo a extremo, robusta y muy fácil de integrar, que permite tener preparado el PSoC para cualquier aplicación en poco tiempo.

Para demostrar el funcionamiento del sistema se van a desarrollar algunas aplicaciones que muestren varios usos del dispositivo, tales como el acceso desde la parte software a GPIOs conectados a la lógica programable, control en tiempo de ejecución de la FPGA y una comunicación en tiempo real HW/SW para mostrar un espectrograma del audio captado por el sistema.

Hay que agradecer al programa de universidades de Xilinx por la donación de la ZedBoard y el software Vivado.

Abstract

This work focuses on the HW / SW co-design of a PSoC, composed of a processing system (PS) and a programmable logic (PL). The co-design consists of several steps such as the choice of an operating system for the processing system, programming the FPGA and most importantly, the definition of the communication interface between them.

Usually, the communication interface between software and hardware is the most complicated and time-consuming point in the design, as well as the more prone to containing errors. To solve this problem, Xillybus is going to be used, a set of tools that make HW / SW communication transparent to the developer. With this, we achieve an end to end, robust and easy to integrate communication, which allows the PSoC to be prepared for any application in a short period of time.

In order to show how the system works, some applications will be developed. Each application represents different uses of the device, such as the access from the PS to the GPIOs connected to the programmable logic, controlling the FPGA in runtime and a real-time HW / SW communication in the form of a spectrogram.

Thanks to the Xilinx University Program for donating the ZedBoard and the Vivado software.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

AC	Alternating Current
ADC	Analogic-to-Digital Converter
AJAX	Asynchronous JavaScript And XML
AMBA	Advanced Microcontroller Bus Architecture
AMS	Agile Mixed Signaling
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BSP	Board Support Packages
CAN	Controller Area Network
CLB	Configurable Logic Blocks
CMOS	Complementary Metal–Oxide–Semiconductor
CPLD	Complex Programmable Logic Device
DC	Direct Current
DDR3	Double Data Rate type three
DMA	Direct Memory Access
DSP	Digital Signal Processor
ECC	Error Correction Code
EEPROM	Electrically Erasable Programmable Read-Only Memory
EHCI	Enhanced Host Controller Interface
EOF	End-Of-File
FFT	Fast Fourier Transform
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
GAL	Generic Array Logic
GCC	GNU Compiler Collection
GMII	Gigabit Media-Independent Interface
GNU	GNU's Not Unix
GPIO	General-Purpose Input/Output
GPL	General Public License
HDL	Hardware Description Language
HDMI	High Definition Multimedia Interface
HW	Hardware
I/O	Input/Output
I^2C	Inter-Integrated Circuit
I2S	Inter-IC Sound
IDE	Integrated Design Environment
IEEE	Institute of Electrical and Electronics Engineers

IoT	Internet of Things
IP	Intellectual Property
IP	Internet Protocol
JTAG	Joint Test Action Group
LED	Light-Emitting Diode
LPC FMC	Low Pin Count FPGA Mezzanine Card
LPDDR2	Low Power Double Data Rate type two
LUT	Look-Up Tables
LVCMOS	Low Voltage Complementary Metal-Oxide-Semiconductor
LVDS	Low-voltage differential signaling
MMC	MultiMediaCard
OLED	Organic Light-Emitting Diode
ONFI	Open NAND Flash Interface
OTP	One Time Programming
OTG	On-The-Go
PAL	Programmable Array Logic
PL	Programmable Logic
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PLL	Phased-Locked Loop
PS	Processing System
QEMU	Quick Emulator
QoS	Quality of Service
RAM	Random Access Memory
RGMII	Reduced Gigabit Media-Independent Interface
RTL	Register-Transfer Level
RTOS	Real-Time Operating System
SC	Secure Digital
SDIO	Secure Digital Input Output
SerDes	Serializer/Deserializer
SGMII	Serial Gigabit Media Independent Interface
SoC	System on chip
SPI	Serial Peripheral Interface
SPLD	Simple Programmable Logic Device
SRAM	Static Random Access Memory
SSTL	Stub Series Terminated Logic
STFT	Short-Time Fourier Transform
SW	Software
Tcl	Tool Command Language
UART	Universal Asynchronous Receiver-Transmitter
ULPI	UTMI Low Pin Interface
USB	Universal Serial Bus
UTMI	USB 2.0 Transceiver Macrocell Interface
VGA	Video Graphics Array

1 INTRODUCCIÓN

Every new beginning comes from some other beginning's end.

- Séneca -

1.1 Motivación

Tradicionalmente, los sistemas electrónicos se han dividido en varios grupos según su función. Por una parte están los circuitos específicamente diseñados para una aplicación, cuya función es siempre la misma (ASICs) y por otra están los circuitos diseñados sin tener en mente una aplicación en concreto en tiempo de fabricación, sino para ser programados posteriormente según las necesidades de uso.

Dejando a un lado los ASICs y centrándonos en los segundos, estos a su vez pueden dividirse en dos corrientes: los dispositivos lógicos programables (CPLDs, FPGAs…) y los microcontroladores. Los primeros están compuestos por bloques lógicos que pueden ser interconectados entre sí de diferentes formas para formar un circuito con una función, y los segundos que tienen un hardware definido y que la función es programada por software. Es decir, en unos programamos el hardware y en otros el software.

Cada aplicación tiene unos requisitos que, para ser cumplidos, necesita en concreto uno de estos tipos de sistemas electrónicos. Sin embargo, estos requisitos cada vez son mayores y a veces pueden parecer incompatibles entre sí, como el procesado en tiempo real, bajo tiempo de desarrollo o bajo coste. De la necesidad de cumplir estas restricciones surgen nuevos diseños híbridos, que combinan elementos hardware y software que se complementan entre sí.

Estos nuevos sistemas combinan las mejores prestaciones del hardware con la flexibilidad, sencillez y menor tiempo de desarrollo que tiene el software. La unión de estos dos elementos hace necesario que exista una interacción entre ellos, convirtiendo este punto en uno de los más complicados de plantear y que más tiempo y problemas conlleva. Para abordar este proceso de codiseño existen algunas herramientas como la que se utilizará en este trabajo que facilitan mucho esta labor.

1.2 Objetivos

El objetivo de este trabajo es realizar un codiseño HW/SW en un Zynq-7000 AP SoC de Xilinx. Para ello se hará una revisión del sistema utilizado así como de las herramientas comerciales disponibles. Se desarrollará una aplicación que sirva como ejemplo para ilustrar el codiseño.

La aplicación consistirá en una implementación hardware de un espectrograma, cuya salida se envía en tiempo real a la parte software y será mostrada en una aplicación web, también se implementan varios filtros de audio que modifican la entrada del espectrograma y permiten ver en la visualización los cambios que producen sobre la señal.

También habrá un ejemplo de cómo utilizar GPIOs que, conectados a la lógica programable, pueden usarse desde el entorno Linux.

1.3 Estructura del trabajo

El trabajo consta de cuatro partes:

- Comienza por una revisión y documentación del estado del arte en codiseño HW/SW y de las herramientas disponibles para ello.
- Después de estudiar el contexto, se estudia específicamente la plataforma ZedBoard™, incluyendo su documentación, datasheets y herramientas apropiadas para ella.
- Posteriormente se elegirán las herramientas a usar y se pondrá a punto el sistema y el entorno de desarrollo que se va a utilizar.
- Finalmente se creará la aplicación que muestre un ejemplo de cómo sería un codiseño HW/SW en esta plataforma.

2 ESTADO DEL ARTE

There is no reason anyone would want a computer in their home.

- Ken Olsen, 1977 -

2.1 Evolución de los sistemas de procesamiento

Los procesadores de silicio continúan siendo el corazón de estos sistemas desde que en los años 50 el transistor reemplazase a las válvulas de vacío en la construcción de los primeros ordenadores. Desde aquel momento los procesadores siguieron avanzando y haciéndose más pequeños para aumentar su escala de integración. La evolución desde el primer microprocesador Intel 4004 hasta los actuales ha sido imparable, siempre multiplicando el rendimiento de sus predecesores.

Su progreso ha estado ligado al hecho de que su uso principal era mover a unos ordenadores que cada vez requerían más potencia, por lo que su desarrollo ha consistido durante mucho tiempo en aumentar el número de transistores por chip, haciéndolos cada vez más pequeños y en elevar la frecuencia de reloj para ofrecer más operaciones por segundo. La frecuencia de reloj ha sido el indicador principal de rendimiento durante mucho tiempo, ha pasado de 700 kHz al rango actual de 2 - 4 GHz donde ya es difícil seguir aumentándola y se ha mantenido estable. En la tabla 1 aparece la evolución de los procesadores de PC de Intel.

Tras llegar a este límite en la frecuencia de reloj, se han usado otras formas de seguir aumentando el rendimiento con cambios en la arquitectura del microprocesador y sobre todo paralelizando. Añadiendo más núcleos en el mismo encapsulado se consigue una computación concurrente capaz de ejecutar muchas más tareas en el mismo tiempo.

En definitiva, los requerimientos de más capacidad de procesamiento de los PC han llevado a los microprocesadores a evolucionar por el camino del rendimiento. Sin embargo, existen otro tipo de dispositivos que hacen uso de microprocesadores y que cada vez están cobrando más importancia: los sistemas embebidos o empotrados.

Nombre	Año	Frecuencia de reloj	Nº de transistores	Tecnología de fabricación
Intel 4004	1971	108 KHz	2.300	10µ
Intel 8008	1972	500-800 KHz	3.500	10µ
Intel 8080	1974	2 MHz	4.500	6µ
Intel 8086	1978	5 MHz	29.000	3µ
Intel 286	1982	6 MHz	134.000	1,5µ
Intel 386	1985	16 MHz	275.000	1,5µ
Intel 486	1989	25 MHz	1.200.000	1µ
Intel Pentium	1993	66 MHz	3.100.000	0,8µ
Intel Pentium Pro	1995	200 MHz	5.500.000	0,6µ
Intel Pentium II	1997	300 MHz	7.500.000	0,25µ
Intel Pentium III	1999	500 MHz	9.500.000	0,18µ
Intel Pentium 4	2001	1.5 GHz	42.000.000	0,18µ
Intel Core 2 Duo	2006	2,93 GHz	291.000.000	65nm
Intel Core 2 Quad	2007	3 GHz	820.000.000	45nm
Intel Core i7	2008 – actualidad	2,2 – 3,8 GHz	> 1.000.000.000	32 – 14 nm

Tabla 1 – Evolución de los procesadores de Intel. [1]

Un sistema empotrado es un sistema electrónico diseñado específicamente para realizar determinadas funciones, generalmente formando parte de un sistema de mayor complejidad. Normalmente están ligados a requisitos de computación en tiempo real.

Al contrario que los PC, estos no requieren grandes prestaciones sino que buscan un consumo mínimo, bajo coste y un tamaño reducido. Esto influye en el desarrollo de sus microprocesadores, en los que se va a buscar estas mismas características. Así aparece otra rama de evolución de microprocesadores en los que se busca un diseño eficiente y de bajo costo, donde no se necesita un gran rendimiento. [2]

No solo el microprocesador cambia: para reducir el tamaño y precio del sistema final se intenta agrupar todos los módulos posibles en un solo chip formando un circuito integrado al que se le denomina System on Chip (SoC).

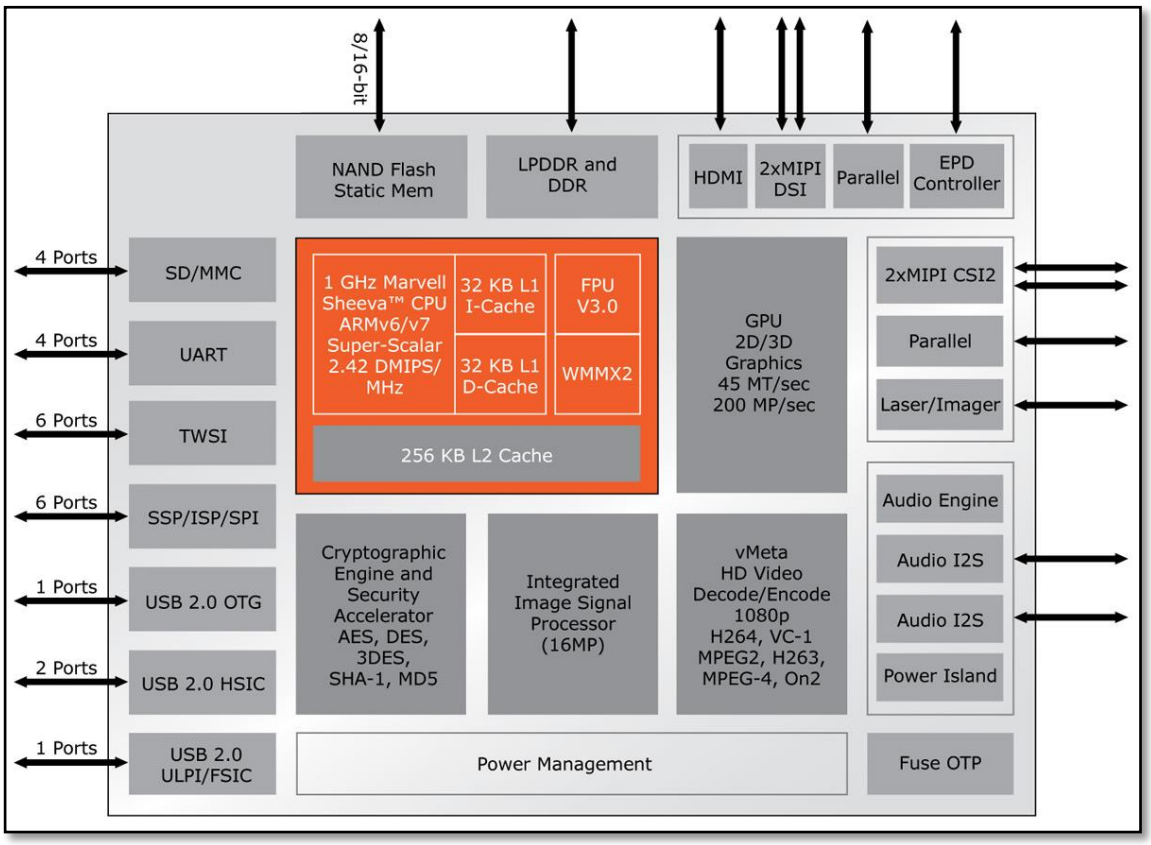

Figura 1 – Diagrama del SoC Marvell Armada 600. (Fuente: Marvell Technology Group Ltd.)

Estos SoC suelen contener:

- Un microprocesador, microcontrolador o un DSP.
- Bloques de memoria (RAM, EEPROM, flash…).
- Osciladores y PLLs.
- Interfaces de comunicación (USB, Ethernet, I2C…).
- Reguladores de voltaje y circuitos de gestión de alimentación.
- Otros periféricos necesarios.

Dada la gran cantidad de sistemas embebidos que hay en el mercado, los SoC se convierten en la arquitectura más usada en los sistemas electrónicos. Esto sin contar el gran incremento que van a experimentar con la llegada del Internet de las Cosas (IoT), en la que se estima que habrá 6.400 millones de objetos conectados en 2016 y hasta 20.800 millones en 2020. [3]

2.2 Evolución de la lógica programable

Los dispositivos de lógica programable (PLD) son circuitos integrados que contienen una estructura de elementos digitales cuya interconexión y funcionalidad pueden ser configuradas antes de ser usados en una aplicación, no tienen una función definida en tiempo de fabricación. Se pueden clasificar en los siguientes tipos:

- Dispositivos programables de baja capacidad (PLA, PAL y GAL) que reemplazan a pequeños circuitos.
- Dispositivos programables de alta capacidad (CPLD, FPGA) que reemplazan a sistemas completos y son los más importantes en la actualidad. [4]

Dentro de la primera categoría se agrupan:

- **Programmable Logic Array (PLA):** Contienen un conjunto de puertas AND programables conectadas a un conjunto de puertas OR también programables con las cuales se complementan para producir la salida deseada. Esta disposición permite que un gran número de funciones lógicas sean sintetizadas en forma de suma de productos o en producto de sumas[1].
- **Programmable Array Logic (PAL):** También contienen un conjunto de puertas AND programables, pero a diferencia de las anteriores, la matriz de puertas OR es fija. Con esta arquitectura solo se pueden implementar sumas de productos pero a cambio hace al dispositivo más rápido, pequeño y barato. Generalmente, las PAL se implementan con tecnología de proceso basada en fusibles y es, por tanto, programable una sola vez (OTP).
- **Generic Array Logic (GAL):** Son una evolución de las PAL, tienen las mismas propiedades lógicas que estas pero además pueden ser borradas y reprogramadas. Esto las hace muy útiles en la fase de prototipado de un diseño, ya que cualquier fallo en la lógica puede ser corregido reprogramándola. Para ello en vez de usar fusibles, utilizan una tecnología de proceso reprogramable como por ejemplo EEPROM (E^2CMOS).

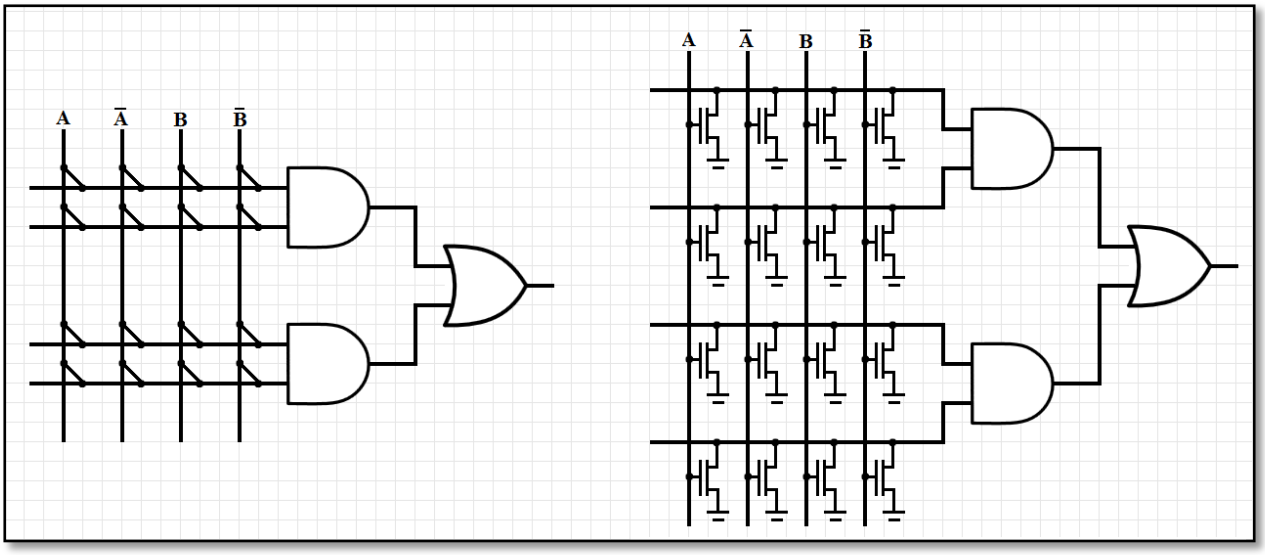

Figura 2 – Evolución de PAL a GAL

[1] Cualquier función de lógica combinacional puede expresarse en forma de suma de productos.

Todos estos dispositivos están obsoletos en la actualidad, los que se usan hoy en día son los de alta capacidad:

- **Complex Programmable Logic Device (CPLD):** Son dispositivos con una mayor escala de integración que los anteriores. Están formados por múltiples bloques SPLD conectados entre sí por una matriz de interconexiones programables (PIA, Programmable Interconnect Array). Tanto los bloques SPLD como la PIA se programan mediante software y pueden implementarse funciones más complejas conectando la salida de un SPLD a la entrada de otro. Por supuesto, estos dispositivos son reprogramables ya que utilizan tecnología EEPROM o SRAM.

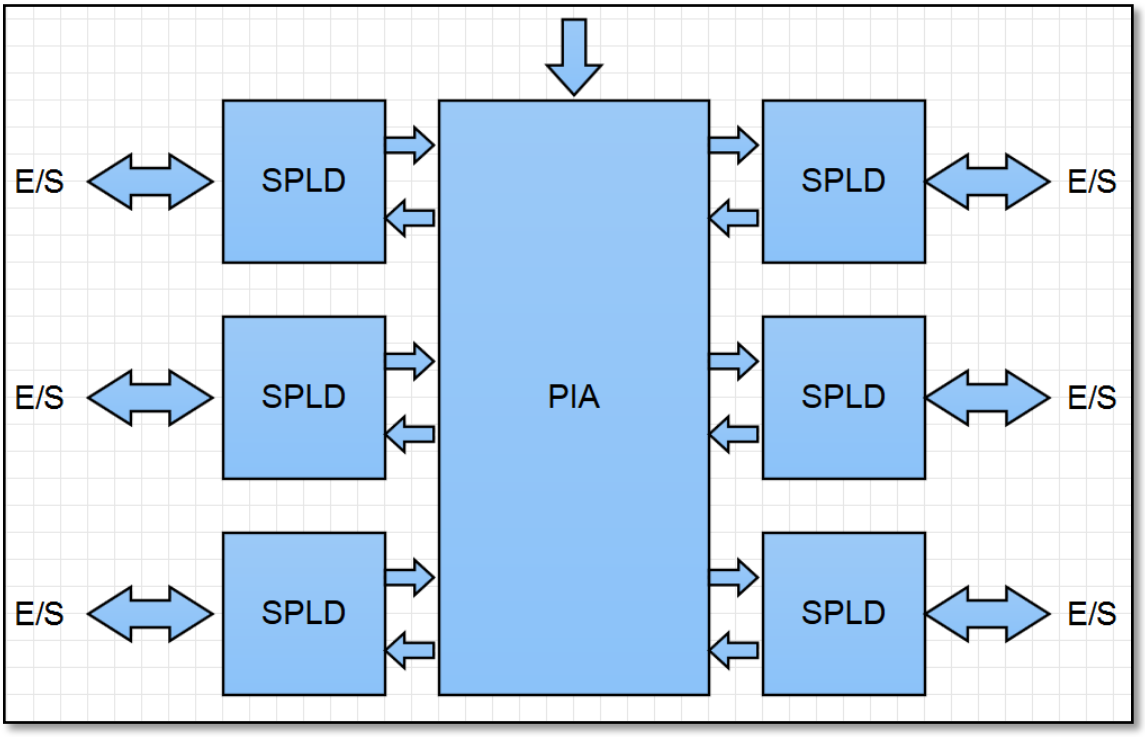

Figura 3 – Diagrama genérico de un CPLD

- **Field Programmable Gate Array (FPGA):** A diferencia de los anteriores, las FPGA no utilizan matrices PAL/PLA, sino que están compuestas por tres elementos básicos: el bloque lógico configurable (CLB), las interconexiones y los bloques de entrada/salida. Los bloques lógicos de la FPGA no son tan complejos como los bloques de un CPLD pero contienen muchos más, esto hace que tengan una densidad y un número de puertas equivalentes mucho mayor.

  Estos bloques lógicos están formados por una LUT (Look-Up Table) y una lógica asociada que emplea un biestable, lo que permite implementar lógica combinacional, lógica secuencial o una combinación de ambas. [5]

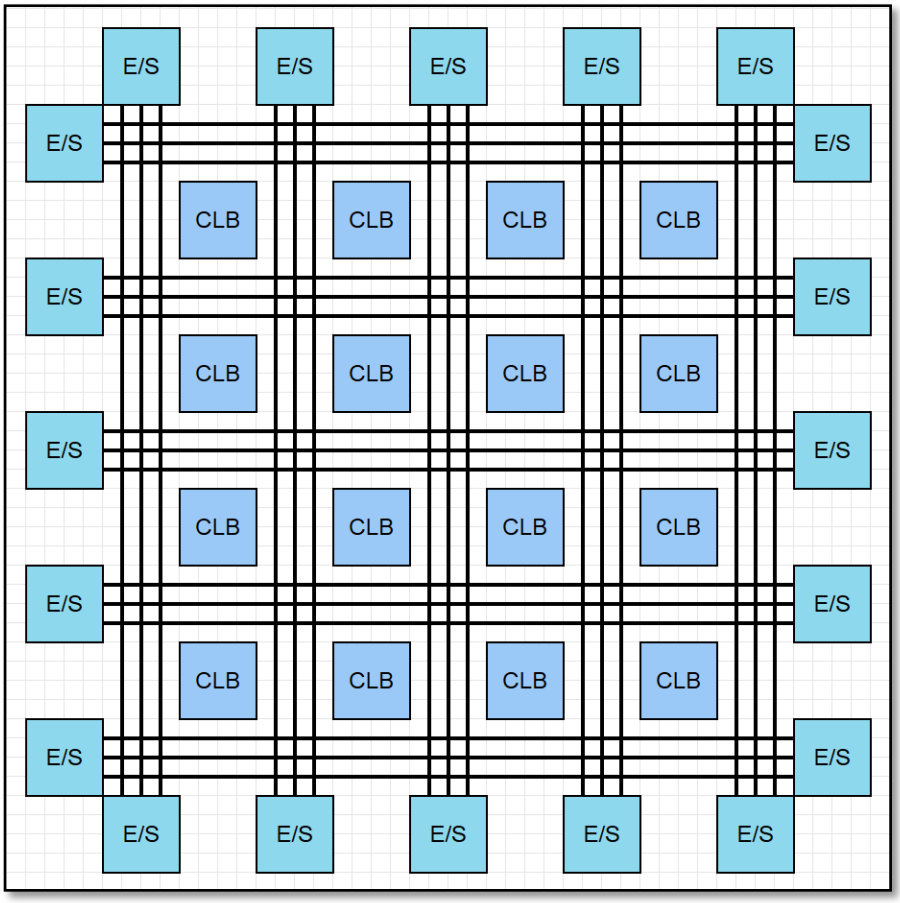

Figura 4 – Estructura de una FPGA

2.3 Marco actual en el codiseño HW/SW

Tras hablar en los capítulos previos de los SoC y de los PLD, este es el punto en el que se unen los dos conceptos para llegar al Programmable System on Chip (PSoC). Un PSoC está compuesto de un procesador, bloques configurables analógicos y digitales e interconexiones programables (Figura 6). Los bloques configurables marcan la principal diferencia de los PSoC respecto a otros microcontroladores.

El término PSoC fue utilizado por Cypress por primera vez en 2002 al comercializar su familia *PSoC 1*. En la actualidad comercializan hasta la familia *PSoC 5*, con algunos modelos como el *PSoC 4 BLE*, que incluyen un procesador de 32 bits, Bluetooth 4.1 Low Energy y actualizaciones de firmware Over-the-Air. Son ya sistemas muy completos que no solo incluyen el procesamiento sino también elementos como las comunicaciones con otros dispositivos.

Los PSoC son especialmente útiles en proyectos en los que se requiere manipular señales analógicas externas al microcontrolador. Las capacidades analógicas (y también digitales) son controladas por los bloques configurables, que a su vez son configurados a través de una serie de registros. Estos registros son inicializados al arranque usando la memoria flash incorporada en el PSoC.

Empotrar esta funcionalidad analógica en el microcontrolador tiene varias ventajas. La más directa es la reducción en espacio que se traduce en ahorro económico. También el sistema final es mucho más simple porque no hay que rutar señales de alimentación y tierra a varios circuitos integrados en la placa. Tampoco hay que rutar señales de un chip a otro ya que estas son manejadas por el microcontrolador.

Otro de los campos en los que se obtienen ventajas es el ruido, ya que el camino que tiene que recorrer una señal eléctrica dentro de un único circuito integrado es menor que el camino que tendría que recorrer externamente entre dos componentes. Un trayecto más corto hace que la antena que aparece sea más pequeña, ganando en inmunidad y emitiendo menos ruido. [6]

No menos importante es la ganancia en rendimiento, añadir una FPGA a un microprocesador aporta mucha más capacidad de procesamiento en paralelo y en tiempo real. Esta es una característica muy importante por ejemplo en sistemas embebidos, que necesitan medir, actuar y transmitir constantemente información del entorno.

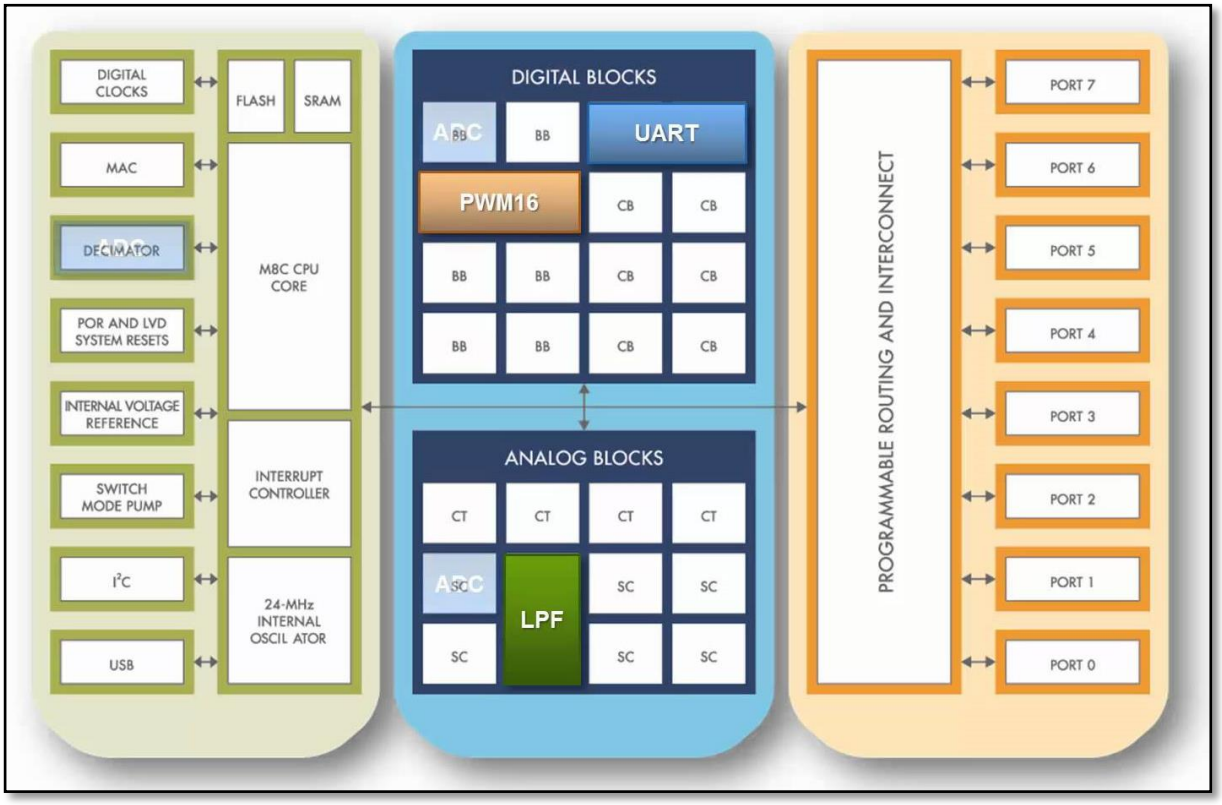

Figura 5 – Diagrama de bloques de un PSoC 1 de Cypress. (Fuente: Cypress Semiconductor)

Xilinx por su parte también ha entrado en este mercado con su familia Zynq®-7000 All Programmable SoC. Se basa también en combinar un SoC con lógica programable, pero sigue una arquitectura un poco diferente a los PSoC de Cypress. Están formados por un SoC con un procesador Dual-core ARM® Cortex™-A9 MPCore™ acompañado de una FPGA, en el próximo capítulo se entrará con más detalle en su arquitectura.

Altera es otra compañía que se ha decantado por productos híbridos SoC + FPGA. Tiene varias familias en este segmento cubriendo las gamas alta, media y baja, con las series *Stratix 10*, *Arria 10/V* y *Cyclone V* respectivamente.

Para este trabajo se va a usar la familia de Xilinx, concretamente la plataforma Zedboard™, que es un kit de desarrollo que tiene más elementos aparte del PSoC como una memoria RAM de 512 MB, GPIOs, conectividad HDMI, VGA, de audio, expansión por FMC y Pmod, etc. Todos estos componentes la hacen idónea para prototipado y desarrollos de pruebas de concepto.

2.3.1 Revisión de herramientas de desarrollo

Las herramientas disponibles en el mercado son propietarias de cada fabricante y están diseñadas para trabajar con sus respectivas plataformas. Todos proporcionan la misma herramienta, un entorno IDE (Integrated Design Environment) con el que es posible sintetizar hardware para sus dispositivos, depurar el código, usar IPs y tener soporte completo.

En el caso de Cypress Semiconductor, esta ofrece tres herramientas para sus desarrolladores: [7]

- **PSoC Creator:** permite programar los dispositivos por medio de una interfaz gráfica, en la que el usuario crea un esquemático del sistema que desea. Este esquemático se crea colocando e interconectando unos componentes predefinidos en el IDE representados por iconos que se unen entre sí. También proporciona herramientas para programar en Verilog, C o vía diagramas de máquinas de estado, así como para su depuración.

- **PSoC Designer:** tiene las mismas características que el anterior, pero este se usa para la familia PSoC 1, mientras que el anterior se usa para las familias PSoC 3, PSoC 4, PSoC 4 BLE, PRoC BLE and PSoC 5LP.

- **PSoC Programmer:** es un IDE que puede ser usado con cualquiera de los dos anteriores para programar cualquier familia de Cypress. Provee al usuario de una capa de acceso al hardware con APIs para diseñar aplicaciones específicas en lenguajes como C, C#, Perl y Python.

Xilinx también ofrece un conjunto de herramientas, entre las que destacan: [8]

- **ISE Design Suite:** es el IDE de Xilinx para diseño y síntesis de circuitos a través de HDL (Hardware Description Language), también permite sintetizar, ejecutar análisis de tiempo, ver diagramas RTL (Register-Transfer Level) y simular los diseños. Desde octubre de 2013 ha entrado en la fase de mantenimiento y no hay planeadas nuevas versiones, por lo que Xilinx recomienda usar Vivado Design Suite en su lugar para nuevos desarrollos.

- **Vivado Design Suite:** el sucesor de ISE, tiene todas sus funcionalidades y añade soporte para los nuevos dispositivos SoC. Además proporciona nuevas herramientas como High-Level Synthesis para programar los dispositivos en C y C++, o el IP integrator, para configurar e integrar IPs del catálogo de Xilinx. También hace uso de estándares como el AMBA® AXI4 interconnect, el Tool Command Language (Tcl) o el Synopsys Design Constraints.

Para este trabajo se usará Vivado Design Suite, en un capítulo posterior se explicará con más detalle el entorno y sus herramientas de desarrollo.

3 REVISIÓN DEL SISTEMA DE DESARROLLO

One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man.

- Elbert Hubbard -

3.1 Xilinx Zynq®-7000 All Programmable SoC

3.1.1 Introducción

La familia Zynq-7000 está basada en la arquitectura Xilinx All Programmable SoC (AP SoC). Estos productos integran un sistema de procesamiento (PS) ARM® Cortex™-A9 MPCore™ de doble núcleo y una lógica programable (PL) de Xilinx en un único dispositivo fabricado con tecnología de 28 nm de bajo consumo y alto rendimiento.

Los Zynq-7000 ofrecen la flexibilidad y escalabilidad de una FPGA mientras proporcionan el rendimiento, consumo y facilidad de uso normalmente asociados a un ASIC. La familia tiene un amplio rango de dispositivos, desde los pensados para aplicaciones de bajo coste hasta las de alto rendimiento. Todos estos dispositivos comparten el mismo PS, mientras que la PL y los recursos de I/O (entrada/salida) varían entre ellos. Como consecuencia, los Zynq-7000 pueden servir para un amplio rango de aplicaciones:

- Sistemas de asistencia a la conducción.
- Control de motores industriales, redes industriales y visión de máquinas.
- Radio LTE.
- Diagnóstico e imagen médica.
- Vídeo y cámaras.

La arquitectura Zynq-7000 permite la implementación de lógica propia en la PL y de software propio en el PS. La integración de estas dos partes permite niveles de rendimiento que otras soluciones con dos chips no pueden alcanzar debido al limitado ancho de banda de I/O, latencia y consumo.

También cuenta con un gran número de IP de Xilinx y drivers de Linux para los periféricos del PS y PL. Como el procesador está basado en ARM, también se dispone de otras herramientas de terceros e IPs en combinación con las propias de Xilinx. Por supuesto, el uso de un procesador habilita el soporte de sistemas operativos como Linux.

El PS y la PL están en dominios de alimentación separados, lo que permite al usuario el apagado de la PL si se requiere. Los procesadores del PS siempre arrancan primero, permitiendo una configuración de la PL desde el software. Esta configuración puede hacerse al arranque o en cualquier punto en el futuro. Además, la PL puede ser parcial o totalmente reconfigurada.

La PL deriva de las FPGA de la serie 7 de Xilinx (Artix®-7 para las 7z010/7z015/7z020 y Kintex®-7 para las 7z030/7z035/7z045/7z100). [9]

3.1.2 Diagrama de bloques

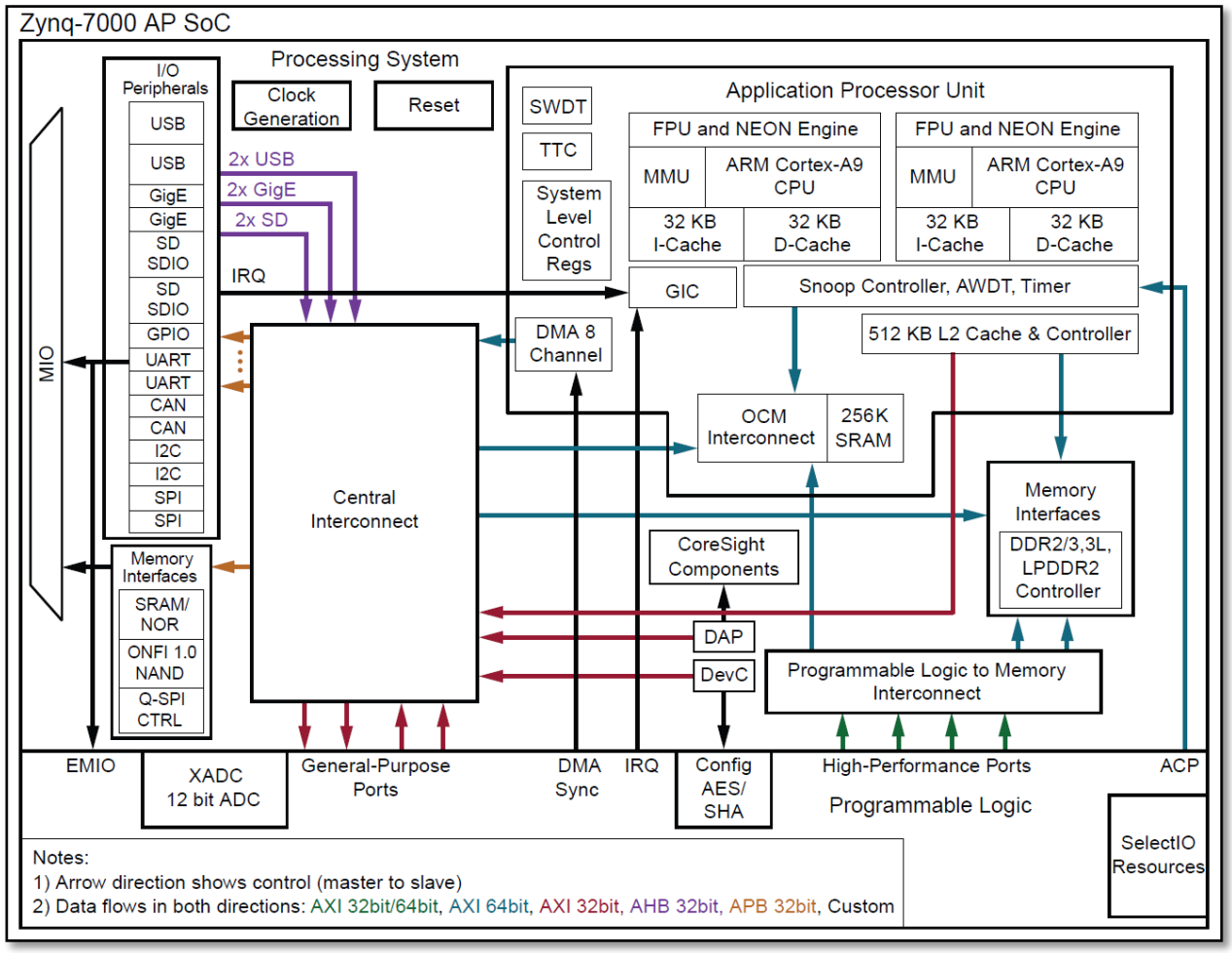

Figura 6 – Diagrama de bloques Zynq-7000 (Fuente: Xilinx)

3.1.3 Características

Sistema de procesamiento (PS)

ARM® Cortex™-A9 Based Application Processor Unit (APU)

- 2,5 DMIPS/MHz por CPU.
- Frecuencia de CPU hasta 1 GHz.
- Soporte de multiprocesador.
- Arquitectura ARMv7-A.
- Motor de procesamiento multimedia NEON™.
- Unidad de coma flotante vectorial (VFPU) de precisión simple y doble.
- CoreSight™ y Program Trace Macrocell (PTM) para debug.
- Tres watchdogs, un timer global y dos contadores de timer triple.

Cachés

- 32 KB de caché nivel 1 asociativa de 4 vías (independiente por cada CPU).
- 512 KB de caché nivel 2 asociativa de 8 vías (compartida por las CPUs).
- Soporte para paridad de byte.

Memoria On-Chip

- ROM de arranque.
- 256 KB RAM.
- Soporte para paridad de byte.

Interfaces para memoria externa

- Controlador de memoria dinámica multiprotocolo.
- Interfaces para memorias DDR3, DDR3L, DDR2 y LPDDR2 de 16 bits y 32 bits.
- Soporte para ECC en modo de 16 bits.
- 1 GB de espacio de direccionamiento.
- Bus de datos para SRAM de 8 bits de hasta 64 MB.
- Soporte para memoria flash NOR paralela.
- Soporte para memoria flash NAND ONFI1.0 (1 bit ECC).
- Memoria serie flash NOR 1 bit SPI, 2 bits SPI, 4 bits SPI y 8 bits SPI.

Controlador DMA de 8 canales

- Soporte para transacciones memoria a memoria, memoria a periférico, periférico a memoria y scatter-gather.

Periféricos de I/O e interfaces

- Dos periféricos Ethernet 10/100/1000 con soporte IEEE 802.3 y IEEE 15881 revisión 2.0.
- Interfaces GMII, RGMII y SGMII.
- Dos periféricos USB 2.0 OTG, cada uno soportando hasta 12 puntos finales.
- IP core USB 2.0.
- Soporta los modos OTG, high-speed, full-speed y low-speed.
- USB host que cumple la especificación EHCI de Intel.
- Interfaz física externa ULPI de 8 bits.
- Dos interfaces para bus CAN conforme a CAN 2.0-B.
- Las interfaces anteriores cumplen los estándares CAN 2.0-A, CAN 2.0-B e ISO 118981-1.
- Interfaz física externa.
- Dos controladores SD/SDIO 2.0/MMC3.31.
- Dos puertos SPI full-duplex.
- Dos UART de hasta 1Mb/s.
- Dos interfaces maestro y esclavo I2C.
- GPIO con cuatro bancos de 32 bits, de los cuales hasta 54 bits se pueden usar con el PS (un banco de 32 bits y otro de 22 bits) y hasta 64 bits (dos bancos de 32 bits) conectados con la PL.
- Hasta 54 I/O multiplexadas para asignaciones de pin en periféricos.

Interconexiones

- Conectividad de alto ancho de banda dentro del PS y entre PS y PL.
- Basado en ARM AMBA AXI.
- Soporte de QoS en maestros críticos para latencia y control del ancho de banda.

[1] IEEE 1588 define el PTP (Precision Time Protocol), se usa para sincronizar relojes a través de una red con mayor precisión que el NTP (Network Time Protocol)

Lógica programable (PL)

Bloques lógicos configurables

- 6-input Look-Up Tables (LUT).
- Biestables
- Sumadores en cascada.

RAM de bloque de 36 Kb

- Doble puerto.
- Hasta 72 bits de ancho.
- Configurable como dual de 18 Kb.

Bloques DSP

- Multiplicador 25 x 18 con signo de 48 bits.
- Presumador de 25 bits.

Bloques I/O programables

- Soporta LVCMOS, LVDS y SSTL.
- De 1,2 V a 3,3V.
- Retraso I/O programable y SerDes.

JTAG Boundary-Scan

- IEEE 1149.1.

PCI Express

- Compatible con PCI Express 2.1 en modo Endpoint y Root.
- Soporta velocidades Gen1 (2,5 Gb/s) y Gen2 (5 Gb/s).
- Soporta hasta 8 líneas.

Transceptores serie de bajo consumo

- Hasta 16 transmisores y receptores.
- Soporte tasas de hasta 12,5 Gb/s.

Convertidores analógico-digitales

- Dos XADCs de 12 bits.
- Hasta 17 canales de entrada configurables.
- Medición de voltaje y temperatura on-chip.
- Un millón de muestras por segundo.
- Acceso continuo por JTAG.

3.1.4 Dispositivos de la familia

Existen 7 dispositivos en la familia Zynq-7000, con características muy parecidas en el PS, pero con bastantes diferencia en la PL ya que cada uno usa un modelo diferente de FPGA de Xilinx. Se recogen en la siguiente tabla: [10]

Zynq-7000 All Programmable SoC								
	Nombre	**Z-7010**	**Z-7015**	**Z-7020**	**Z-7030**	**Z-7035**	**Z-7045**	**Z-7100**
Sistema de procesamiento (PS)	Procesador	Dual ARM® Cortex™-A9 MPCore™ con CoreSight™						
	Extensiones del procesador	NEON™ & Precisión Single / Double en coma flotante para cada procesador						
	Frecuencia máxima	667 MHz ; 766 MHz; 866 MHz			667 MHz; 800 MHz; 1 GHz			667 MHz 800 MHz
	Caché L1	32 KB instrucciones, 32 KB datos por procesador						
	Caché L2	512 KB						
	Memoria on-chip	256 KB						
	Soporte memoria externa	DDR3, DDR3L, DDR2, LPDDR2						
	Memoria externa estática	2x Quad-SPI, NAND, NOR						
	Canales DMA	8 (4 dedicados a la PL)						
	Periféricos	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO						
	Periféricos con DMA	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO						
	Seguridad[1]	Autenticación RSA, AES y SHA-256-bit cifrado y autenticación para Secure Boot						
Interfaces entre PS y PL (Interfaces primarias e interrupciones solo)		2x AXI 32b Maestro 2x AXI 32-bit Esclavo AXI 64-bit ACP				Memoria 4x AXI 64-bit/32-bit 16 interrupciones		
Lógica programable (PL)	PL equivalente de la serie 7 de Xilinx	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA
	Celdas de lógica programable (Puertas ASIC aproximadas)[2]	28K (~430K)	74K (~1,1M)	85K (~1,3M)	125K (~1,9M)	275K (~4,1M)	350K (~5,2M)	444K (~6,6M)
	LUTs	17.600	46.200	53.200	78.600	171.900	218.600	277.400
	Biestables	35.200	92.400	106.400	157.200	343.800	437.200	554.800
	RAM de bloque (nº de bloques de 36 Kb)	240 KB (60)	380 KB (95)	560 KB (140)	1.060 KB (265)	2.000 KB (500)	2.180 KB (545)	3.020 KB (755)
	DSP Slices	80	160	220	400	900	900	2.020
	Rendimiento de pico de los DSP (FIR simétrico)	100 GMACs	200 GMACs	276 GMACs	593 GMACs	1.334 GMACs	1.334 GMACs	2.622 GMACs
	PCI Express	-	Gen2 x4	-	Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	XADC	2x 1 MSPS ADCs, 12 bits con hasta 17 entradas diferenciales						
	Seguridad	AES y SHA 256b para código de arranque y configuración, encriptado y autenticación de la PL						

Tabla 2 – Modelos de la familia Zynq-7000

[1] La seguridad es compartida por el PS y la PL.
[2] El número de puertas ASIC equivalentes depende de la función implementada. Aquí se toma 1 celda lógica = ~15 puertas ASIC

3.2 ZedBoard™

3.2.1 Introducción

La ZedBoard (Zynq Evaluation and Development) es una placa basada en el AP SoC Zynq-7000 de Xilinx. Concretamente el modelo Z-7020 que combina un procesador ARM® Cortex™-A9 MPCore™ de doble núcleo con una Artix-7 FPGA de Xilinx de 85.000 celdas lógicas. Es una plataforma ideal para desarrollar por la gran cantidad de periféricos que contiene y sus posibilidades de expansión. [11]

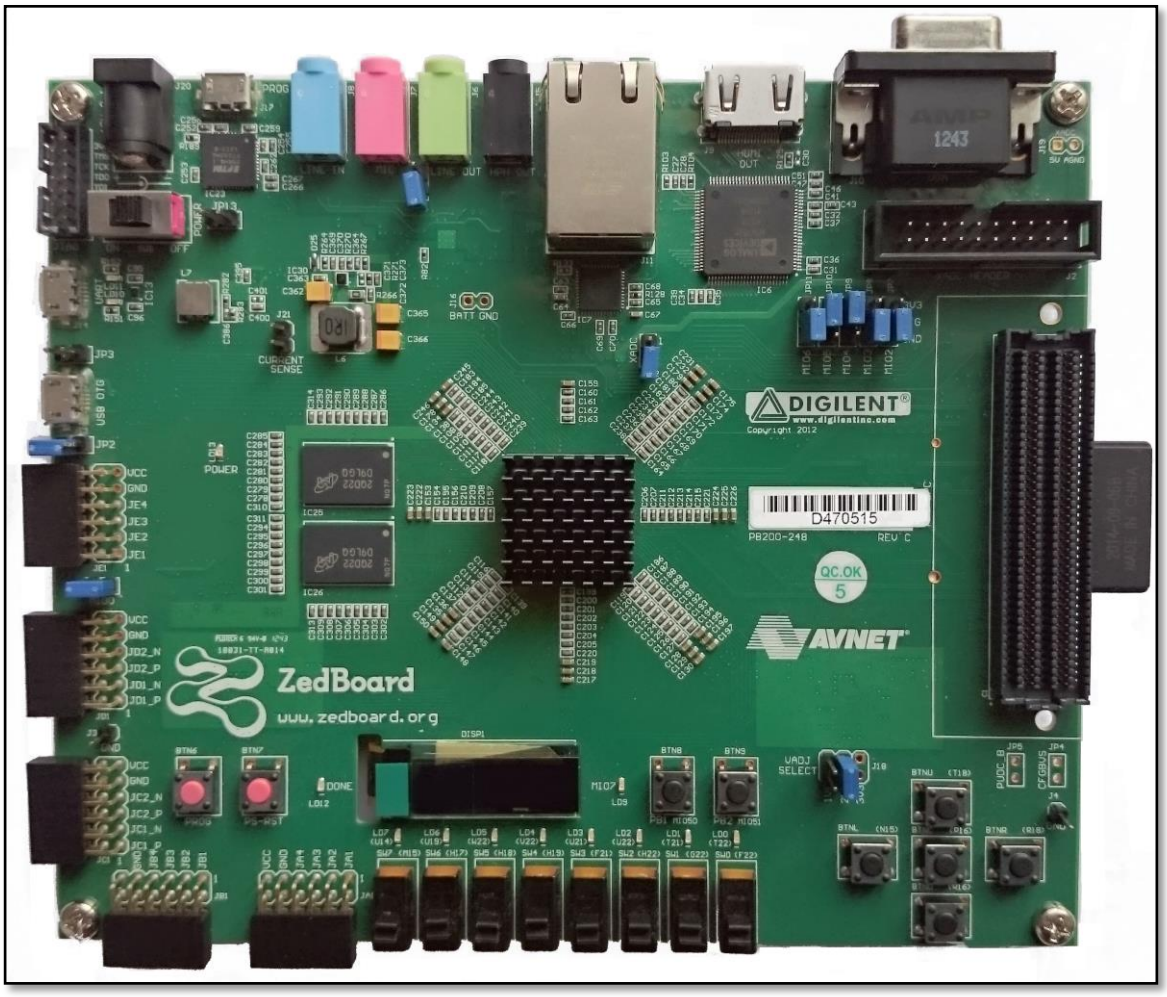

Figura 7 – ZedBoard

Es una plataforma, por tanto, con un gran número de aplicaciones, entre las que se pueden destacar:

- Procesamiento de vídeo.
- Control de motor.
- Aceleración de software.
- Desarrollo Linux/Android/RTOS.
- Procesamiento ARM embebido.
- Todas las aplicaciones del Zynq-7000.

Para terminar su introducción, estas son sus características:

- Xilinx® XC7Z020-1CLG484C Zynq-7000 AP SoC.
    - Configuración principal: QSPI Flash.
    - Opciones de configuración auxiliares: tarjeta SD y JTAG en serie.
- Memoria:
    - 512 MB DDR3 (128M x 32).
    - 256 Mb QSPI Flash.
- Interfaces:
    - USB-JTAG.
        - Acceso a la PL.
        - Pines para el PS conectados a través de Pmod.
    - 10/100/1000 Ethernet.
    - USB OTG 2.0.
    - Tarjeta SD.
    - USB 2.0 FS puente USB-UART.
    - 5 conectores compatibles con los Pmod de Digilent (1 PS, 4 PL).
    - 1 LPC FMC (Low Pin Count FPGA Mezzanine Card).
    - 1 conector AMS (Agile Mixed Signaling).
    - 2 botones de reset (1 PS, 1PL).
    - 7 pulsadores (2 PS, 5 PL).
    - 8 switches (PL).
    - 9 LEDs de usuario (1 PS, 8 PL).
    - DONE LED (PL).
- Osciladores on-board:
    - 33,333 MHz (PS)
    - 100 MHz (PL).
- Imagen/Audio:
    - Salida HDMI.
    - VGA (Color 12 bits).
    - Pantalla OLED 128x32.
    - Audio line-in, line-out, auriculares y micrófono.
- Alimentación:
    - Switch On/Off.
    - Regulador AC/DC 12V @ 5A.
- Software:
    - ISE WebPACK Design Software.
    - Licencia de ChipScope para XC7Z020.

3.2.2 Características y componentes

3.2.2.1 Memoria DDR3

La ZedBoard incluye dos módulos DDR3 de 128 Mb x 16 de Micron que crean un total de 512 MB con una interfaz de 32 bits. Al principio la ZedBoard usaba el modelo Micron MT41J128M16HA-15E:D, pero en 2012 alcanzó su fin de vida, así que está planeado que migre al modelo MT41K128M16JT-125. La interfaz soporta velocidades de hasta 533 MHz (1066Mb/s).

Las herramientas de Xilinx permiten además el entrenamiento de DRAM a través de Xilinx Platform Studio o del *IP editor* en Vivado. Esto consiste en modificar los parámetros *DQS to Clock Delay* y *Board Delay* que recalculan los tiempos de escritura y lectura de la memoria para obtener un mejor rendimiento.

3.2.2.2 QSPI Flash

Memoria Flash NOR serie quad-SPI de 256 Mb, modelo Spansion S25FL256S. Se puede usar tanto para inicializar el PS como para configurar la PL (bitstream). Spansion proporciona el Spansion Flash File System (FFS) para ser usado después del arranque del procesador. Soporta velocidades de hasta 104 MHz, para aprovechar los 100 MHz a los que funciona el Zynq. En el modo quad-SPI, esto se traduce en 400Mb/s.

3.2.2.3 Tarjeta SD

La tarjeta SD se puede usar para arrancar el Zynq y como memoria externa no volátil. Se conecta a través de un conector estándar de 9 pines TE 2041021-1, este a su vez se conecta con el periférico SD/SDIO del PS del Zynq para controlar la comunicación con la SD.

Nota: para usar la tarjeta SD, JP6 debe estar cerrado.

3.2.2.4 USB OTG

La ZedBoard implementa una de las dos interfaces USB OTG disponibles, como PHY se usa el TI TUSB1210 Standalone USB Transceiver Chip. El chip soporta velocidades de hasta 480 Mb/s y opera a un voltaje de 1,8V.

La interfaz USB se puede configurar para los modos Host, dispositivo y OTG cambiando el estado de varios jumpers. Este puerto USB no alimenta la placa, sin embargo la ZedBoard le proporciona 5V cuando está en modo Host u OTG.

3.2.2.5 Puente USB a UART

Esta función la proporciona el dispositivo Cypress CY7C64225 USB-to-UART Bridge, el cual se conecta al periférico UART del PS. El conector externo de la ZedBoard es un USB micro B, únicamente se implementa una conexión básica de transmisión y recepción, pero si se requiere control de flujo se puede añadir a través de un PL-Pmod.

Cypress proporciona drivers para Virtual COM Port (VCP) de acceso gratuito, que permite que el UART le aparezca al ordenador que esté conectado a la ZedBoard como un puerto COM, así puede usarse software como HyperTerm o Tera Term.

3.2.2.6 USB-JTAG

La ZedBoard también provee de funciones de JTAG basadas en el dispositivo USB High Speed JTAG Module, SMT1de Digilent. El conector usado es igualmente un USB micro B.

3.2.2.7 HDMI

El transmisor ADV7511 HDMI de Analog Devices proporciona la interfaz de video digital. Se trata de un transmisor a 225MHz compatible con HDM 1.4 y DVI 1.0 que soporta 1080p60 con modo de color de16 bits, YCbCr, 4:2:2.

También soporta S/PDIF y 8 canales de audio I2S, aunque la interfaz I2S no está conectada en la ZedBoard. El S/PDIF puede llevar audio comprimido incluyendo Dolby® Digital, DTS®, and THX®. Analog Devices ofrece drivers de Linux y diseños de referencia que muestran como interactuar con este dispositivo.

3.2.2.8 VGA

La ZedBoard también permite salida de video de 12 bits de color a través de un conector VGA (TE 4-1734682-2). Cada color se crea a partir de una red de resistencias desde cuatro pines de la PL.

3.2.2.9 Códec de Audio I2S

El códec de audio ADAU1761 de Analog Devices proporciona procesamiento digital de audio al Zynq. Permite grabar y reproducir a 48 KHz aunque soporta tasas de muestreo desde 8 a 96 KHz, además tiene control de volumen. El códec se puede configurar usando Analog Devices SigmaStudio™ para optimizar el audio en aplicaciones específicas, también ofrece filtros, algoritmos y mejoras. Analog Devices también tiene drivers para Linux de este dispositivo.

Las conexiones externas son cuatro conectores Jack de 3,5 mm (micrófono, línea de entrada, línea de salida y auriculares).

3.2.2.10 OLED

La ZedBoard contiene una pantalla de 30mm x 11,5mm x 1,45 mm, de 128x32 píxeles, matriz pasiva y monocromática. Todo a cargo del Inteltronic/Wisechip UG-2832HSWEG04 OLED Display.

3.2.2.11 Reloj

El PS del Zynq usa una fuente de reloj dedicada a 33,333 MHz con terminación de series (Fox 767-33.333333-12). La infraestructura del PS puede generar hasta cuatro relojes basados en PLL para la lógica programable. Otro oscilador de 100 MHz (Fox 767-100-136) proporciona la entrada de reloj de la PL.

3.2.2.12 Fuentes de reset

- **Power-on Reset:** el PS del Zynq soporta señales externas de power-on reset, este reset es el maestro de todo el chip. Esta señal resetea cada registro del dispositivo que pueda ser reseteado. La ZedBoard conduce esta señal a un comparador que mantiene la señal en el sistema hasta que todas las alimentaciones sean válidas. Muchos otros circuitos de la ZedBoard son reseteados también por esta señal.
- **Pulsador de programa (PROG):** este botón inicia la reconfiguración de la PL por el procesador.
- **PS reset:** este reset permite resetear toda la lógica del dispositivo sin alterar el entorno de depuración. Por ejemplo, los puntos de parada colocados por el usuario siguen establecidos después del reset. Debido a razones de seguridad, se borra toda la memoria contenida en el PS incluyendo la on-chip. La PL también se resetea pero los pines que seleccionan el modo de arranque no se vuelven a muestrear.

3.2.2.13 I/O de usuario

- **Pulsadores:** existen 7 pulsadores para ser usados como GPIOs, cinco para la PL y dos para el PS. Están conectados mediante pull-down, así que pulsarlos los conecta a alimentación.
- **DIP Switches:** la ZedBoard contiene ocho switches de entrada.
- **LEDs:** ocho LEDs en la PL y uno en el PS.

3.2.2.14 10/100/1000 Ethernet

La ZedBoard implementa un puerto Ethernet 10/100/1000 usando un Marvell 88E1518, conectado al Zynq mediante la interfaz estandarizada RGMII. Opera a 1,8V e incorpora además dos LEDs de estado que indican tráfico y estado válido del enlace.

3.2.2.15 Conector LPC FMC

Hay disponible una ranura LPC FMC que hace que la ZedBoard soporte una gran variedad de módulos. La ranura contiene 68 I/O, los cuales también pueden ser usados como 34 pares diferenciales. El voltaje aplicado varía entre 1,8V, 2,5V y 3,3V. La configuración de 3,3V podría ser la más peligrosa por lo que no está disponible con el hardware por defecto de la placa. Para llegar a los 3,3V hay que conectar los pads 3V3 en el J18.

Es importante, antes de conectar una tarjeta FMC asegurarse de que el voltaje correcto está seleccionado en el J18, si no podría resultar en daños a la tarjeta FMC y/o a la ZedBoard.

3.2.2.16 Conectores Digilent Pmod

La ZedBoard tiene cinco conectores Pmod (2x6) de Digilent. Estos son de ángulo recto, hembras y de 0,1" que incluyen ocho I/O más dos señales de tierra más dos señales de 3,3V como se muestra en la figura 8. Cuatro de los Pmod están conectados a la PL y el restante al PS.

Dos de los Pmods, JC1 y JD1 están alineados en una configuración dual y tienen sus I/O rutadas diferencialmente para soportar LVDS a 525 Mb/s.

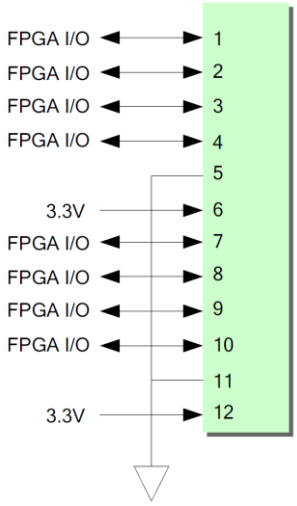

Figura 8 – Conexiones Pmod [11]

3.2.2.17 Conector AMS (Agile Mixed Signaling)

El XADC proporciona conectividad analógica para diseños de este tipo, incluyendo tarjetas AMS como la AMS Evaluation Card de Xilinx.

El conector analógico está conectado cerca del LPC FMC, tanto las I/O analógicas como digitales pueden ser conectadas fácilmente en una tarjeta. Esto permite al conector analógico ser conectado a la tarjeta FMC usando un cable de cinta o plano como se indica en la figura 9. El conector analógico también puede ser usado solo para la conexión de señales analógicas externas.

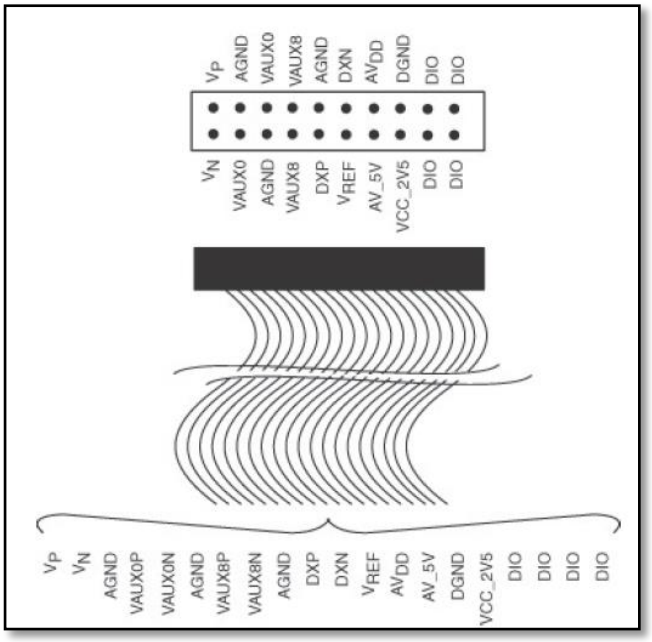

Figura 9 – Pin out del conector analógico [11]

3.2.2.18 Modos de arranque

Los dispositivos Zynq-7000 usan un proceso de arranque multietapa que soporta tanto arranque seguro como no seguro. El PS es el maestro del arranque y del proceso de configuración. Los modos posibles son: NOR, NAND, Quad-SPI, tarjeta SD y JTAG. Tras el reset, los pines de modo son leídos para determinar qué dispositivo será el de arranque primario.

Los pines encargados son los MIO[8:2], controlados a través de jumpers que permiten cambiar los modos incluyendo el uso de JTAG en cascada y el uso de PLL internos. Más información sobre sus combinaciones se puede encontrar en [11].

Para profundizar en las etapas de arranque, se proporciona más información en el capítulo 3 "Boot and configuration" de [12] y en [9].

3.2.2.19 JTAG

Aparte del USB-JTAG, la ZedBoard proporciona un conector tradicional de JTAG para ser usado con los cables de Xilinx Platform y los cables JTAG HS1de Digilent.

La ZedBoard automáticamente añade el FMC dentro de la cadena de JTAG cuando una tarjeta FMC es conectada a la placa.

4 PREPARACIÓN DEL SISTEMA

The Linux philosophy is "laugh in the face of danger".
Ooops. Wrong one. "Do it yourself". That's it.

- Linus Torvalds -

4.1 Introducción

Antes de comenzar el desarrollo de aplicaciones embebidas es necesario tomar ciertas decisiones respecto a la arquitectura del sistema. Quizás la más importante de ellas es decidir qué sistema operativo usar o no usar ninguno (bare-metal).

Un diseño bare-metal es aquel que no usa un sistema operativo, se suele usar cuando la aplicación no necesita muchas características que son proporcionadas por el sistema operativo. En estos diseños se trata directamente con el hardware, accediendo a los registros, moviendo y operando datos directamente. Todos los recursos como interrupciones, timers y I/O tienen que ser considerados por el código del programador. Hoy en día con las herramientas de síntesis de alto nivel, se puede programar directamente en C sin tener que bajar al lenguaje ensamblador lo que ahorra mucho tiempo y agiliza el proceso.

Este método funciona bien cuando el sistema es sencillo, tiene pocas tareas que gestionar o necesita un determinismo muy preciso. Un sistema operativo consume una pequeña cantidad de los recursos del procesador y tiende a ser menos determinista, pero con el incremento en la capacidad y velocidad de los procesadores embebidos, esta carga es casi insignificante en muchos sistemas.

Los sistemas operativos, como Linux[1], ofrecen un conjunto de facilidades que permite a los desarrolladores abstraerse del hardware y no tener la necesidad entender cada pequeño detalle de él. Entre estas ayudas está la priorización de tareas, gestión de la memoria, drivers, sistemas de archivos, comunicación entre procesos, balance de carga entre varios procesadores o seguridad. Muchas de estas tareas se ejecutan en segundo plano de forma transparente al desarrollador, lo que le permite centrarse en el software de la aplicación. De esta forma el sistema adquiere mucha más flexibilidad y es posible llevar a cabo diseños de mayor complejidad y más rápidamente.

Además Linux es un sistema operativo de código abierto, disponible en varias distribuciones y también puede ser compilado desde su repositorio. Gracias a esto, se pueden seleccionar los módulos que se quieren compilar y no es obligatorio construirlo entero, lo que viene muy bien en sistemas embebidos donde solo se necesitan funciones mínimas.

Como última elección están los RTOS, pensados para aplicaciones en tiempo real. Tienen un comportamiento más determinista debido a su planificador de tareas, que garantiza un patrón de ejecución predecible. Son sistemas operativos más ligeros pero con menos funciones que los de propósito general, es una elección a medio camino entre SO y bare-metal.

[1] En este trabajo se usa el término Linux tanto para referirse al sistema operativo formado por el software GNU junto al núcleo Linux (GNU/Linux), como para referirse únicamente al núcleo Linux. Las distribuciones Linux de escritorio normalmente contienen muchos componentes GNU, mientras que algunos sistemas embebidos contienen pocos o ningún paquete GNU porque están pensados para ser eficientes en espacio. Ambos casos serán englobados en "Linux".

Para este trabajo se va usar un sistema operativo Linux, por las posibilidades que ofrece para desarrollar cualquier tipo de aplicación con la mayor facilidad posible. En el siguiente subcapítulo se abordará la tarea de elegir el SO más indicado para el dispositivo.

4.2 Selección del sistema operativo

4.2.1 Xilinx Linux

Xilinx mantiene su propia imagen de Linux, basada en el kernel oficial, en la que ofrece soporte de las partes específicas de Xilinx que se encuentran en el kernel (drivers y BSPs). Está contenido en un servidor git público (https://github.com/xilinx) con todas las ventajas que ello conlleva: se tiene un control de versiones, la comunidad de desarrolladores puede enviar sus parches para seguir mejorándolo y se integra fácilmente con el entorno de trabajo.

En este servidor no solo está el kernel de Linux, sino que hay varios repositorios como el del u-boot, BSP, device tree, toolchain y otras herramientas. Para generar la imagen es necesario descargar estas herramientas de Xilinx y compilar la imagen para ARM, generar el u-boot, device tree y preparar el medio de arranque. [13]

Son los mismos pasos que para compilar un Linux genérico añadiendo los drivers de Xilinx (http://www.wiki.xilinx.com/Linux+Drivers).

Xilinx también ofrece unos Linux pre-construidos que se pueden usar en lugar de generar el kernel Linux y crear una imagen de arranque. Estas versiones se pueden encontrar en: http://www.wiki.xilinx.com/Zynq+Releases.

4.2.2 Petalinux

Petalinux es una distribución de Linux de Xilinx que incluye tanto el S.O. Linux como un entorno de desarrollo para los dispositivos de Xilinx. Es el entorno recomendado por Xilinx ya que está totalmente integrado, testeado y documentado, además al ser un producto de Xilinx cuenta con su gestión, soporte, y seguimiento de errores. [14]

Está basado en el Linux que mantiene Xilinx en su servidor git, pero además incluye otras herramientas que hacen más sencillo el desarrollo de aplicaciones en sus plataformas: [15]

- Interfaces de línea de comandos.
- Generadores y plantillas de desarrollo de aplicaciones, drivers y librerías.
- Generador de imágenes de arranque.
- Agentes de depuración y soporte para el System Debugger de Xilinx..
- Herramientas GCC.
- Simulador QEMU.

Aparte de estas herramientas para el desarrollador, la propia distribución Linux también incluye paquetes como:

- Bootloader.
- Kernel optimizado para la CPU.
- Librerías y aplicaciones Linux.
- Desarrollo de aplicaciones C y C++.
- Depuración.
- Soporte de hilos y FPU.
- Servidor web integrado para gestión remota y configuraciones de firmware.

Petalinux está disponible gratis y sin restricciones de licencias. Más información disponible en: [16] [17] [18] [19].

4.2.3 Arch Linux ARM

Esta distribución es un port de Arch Linux para procesadores ARM. Soporta conjuntos de instrucciones de ARMv5te, ARMv6, ARMv7 y ARMv8. Al depender de Arch Linux, que es una distribución importante dentro de los Linux de escritorio, cuenta siempre con las últimas actualizaciones y los paquetes más nuevos, además de con una gran comunidad de usuarios.

Hereda la filosofía de Arch, que es la de la simplicidad y centrada en el usuario. Está dirigida a usuarios experimentados en Linux, a los que da control completo y la responsabilidad sobre el sistema. La distribución sigue un ciclo *rolling-release*, que quiere decir que recibe pequeñas actualizaciones diariamente en vez de grandes actualizaciones cada cierto tiempo [20].

En palabras de los creadores "es ligera, flexible, simple y tiene como propósito ser muy parecida a UNIX. Su filosofía de diseño e implementación la hacen fácil de extender y moldear". Destaca especialmente por su wiki, una de las mejores documentadas del mundo Linux [21].

Arch está bajo la licencia GNU GPLv2 (software libre) y además está disponible de manera gratuita. La información sobre su instalación en la ZedBoard se puede encontrar en [22].

4.2.4 Xillinux

Xillinux es una distribución que incluye Linux y un kit de código FPGA para las plataformas ZedBoard, ZyBo, SocKit board y MicroZed. Está basada en Ubuntu 12.04 para ARM y puede hacer que la placa se comporte como un PC, ejecutando un entorno gráfico y conectando ratón y teclado.

La configuración e instalación es rápida y sencilla, está integrada con las herramientas de Xilinx (Vivado) y no requiere conocimientos de Linux ni FPGA. Al estar basada en Ubuntu cuenta con el soporte, comunidad y paquetes de la distribución de escritorio más extendida entre los usuarios de Linux [23].

Su principal ventaja es que incluye el IP core Xillybus. Consiste en un IP core para la FPGA y un driver para Linux (y Windows) que se encargan de la comunicación PS-PL, sin que el desarrollador deba preocuparse por el diseño de bajo nivel. Para Linux, el canal de comunicación con la FPGA se ve como un fichero más del sistema, por lo que se puede escribir y leer de él con las funciones típicas de ficheros. En el otro lado, la FPGA ve la comunicación como una FIFO, lo que escriba en ella llegará al PS y lo que lea será lo que el PS le ha enviado [24].

Xillybus se encarga del resto, abstrae la comunicación sobre PCIe y hace uso de la DMA del host para asegurar transferencias extremo a extremo, continuas y robustas.

Xillinux también incluye el device tree, u-boot y resto de herramientas necesarias para tener el sistema funcionando al instante. Además contiene una demo básica en la que hay implementada una comunicación entre FPGA y host para que el usuario compruebe desde el primer momento como funciona y tenga un ejemplo de cómo desarrollar su aplicación que haga uso de Xillybus.

La distribución de Linux y los drivers de Xillybus están bajo la licencia GPL, así que pueden ser usados y distribuidos como se quiera. El IP core de Xillybus sin embargo es gratis solo para propósitos académicos: uso en clases, laboratorios, proyectos de estudiantes y proyectos de investigación con escaso o ningún presupuesto, mientras que para uso comercial es necesario pagar por él [25].

4.2.5 Tabla comparativa

	Ventajas	Inconvenientes
Xilinx Linux	Soporte de Xilinx. Incluye kernel, BSPs, drivers, u-boot y resto de herramientas necesarias. Código disponible en git.	Compilación y construcción de todas las herramientas y de Linux a mano. Documentación mínima.
Arch Linux ARM	Soporte de la comunidad de Arch Linux. Distribución ligera. Muchos paquetes para la distribución, mantenidos y actualizados diariamente. Wiki extensa.	Distribución general para ARM, pero no centrada en Zynq-7000. Existe poca documentación para esta plataforma. Poco centrada en el codiseño.
PetaLinux	Soporte de Xilinx. Distribución de Linux ya compilada. SDK y multitud de herramientas para desarrollar para ella. Licencias comerciales también incluyen soporte dedicado de Xilinx. Gran cantidad de documentación.	Muy centrada en Linux, pero poco en el codiseño con la FPGA
Xillinux	Basada en Ubuntu 12.04: • Distribución estable y con una gran comunidad. • Cuenta con gran cantidad de repositorios con paquetes mantenidos y actualizados. Soporte de Ubuntu y Xillybus. Posibilidad de tener un escritorio gráfico con ratón y teclado. Rápida y fácil de configurar e instalar. Enfocada en el codiseño HW/SW con el IP core Xillybus, lo que acelera y facilita los diseños. Buena documentación específica para Zynq y ZedBoard.	Licencia de Xillybus gratuita para estudiantes e investigaciones de bajo presupuesto, pero licencia de pago para uso comercial

Tabla 3 – Comparación de S.O. Linux.

Para este trabajo se va a elegir Xillinux, por su facilidad de uso y por la inclusión de Xillybus que hará que el codiseño entre PS y PL sea mucho más sencillo y menos propenso a errores. Con esto el desarrollador puede centrarse en la aplicación en sí sin perder tanto tiempo en la comunicación entre ambas partes. En el capítulo 5 se explicará en mayor profundidad su funcionamiento.

4.3 Herramientas de desarrollo hardware

Una vez preparada la parte software, hay que poner a punto las herramientas para trabajar en la parte hardware. Lo más normal es usar las herramientas que proporciona el fabricante del dispositivo que se use, en este caso Xilinx. Hasta hace poco la herramienta principal de Xilinx era ISE, pero ha dejado de mantenerse en favor de Vivado Design Suite, así que esta será la usada en este trabajo. Por otra parte, el SO elegido recomienda usar en concreto la versión Vivado 2014.1[1], que será la finalmente escogida.

4.3.1 Vivado Design Suite

Vivado es la herramienta principal de Xilinx para desarrollo en FPGA. Proporciona una interfaz gráfica de usuario desde la que es posible llevar a cabo todo el diseño y la síntesis de circuitos. Todas sus herramientas y opciones están escritas en formato Tcl (Tool Command Language), lo que hace posible usar tanto la interfaz gráfica como el intérprete de comandos.

Como la base de datos está accesible por Tcl, los cambios en las restricciones, configuraciones u opciones en las herramientas se aplican en tiempo real, sin necesidad de forzar la reimplementación. Tiene muchas opciones que se pueden utilizar en cualquier etapa del diseño como definir restricciones de tiempo, explorar el catálogo de IP, simulaciones o restricciones físicas con técnicas de floorplanning. También ofrece realimentación del diseño en todo momento con la estimación de recursos usados, retraso de interconexión o consumo de energía.

Vivado puede descargarse desde la página de Xilinx [26] en todas sus versiones. El proceso de instalación es sencillo y guiado por el instalador. Una vez instalado solo hay que abrirlo e iniciar un nuevo proyecto, después aparece la pantalla principal del programa, de la que se explicarán ahora las posibilidades más interesantes:

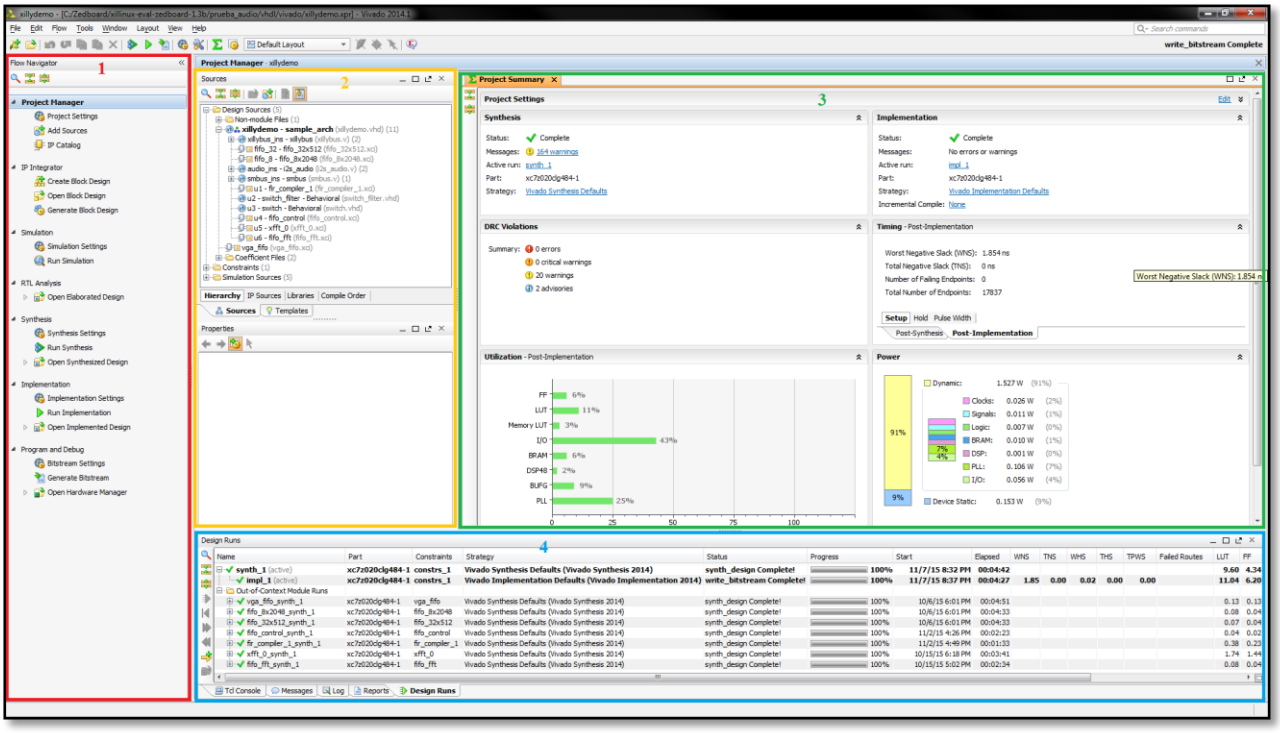

Figura 10 – Entorno de Vivado

[1] Recientemente la versión recomendada de Vivado ha cambiado de 2014.1 a 2014.4. Cualquier versión puede usarse pues los cambios son pequeños, aunque siempre es recomendable utilizar la más reciente.

1. **Flow Navigator:** Proporciona acceso a los comandos y herramientas para llevar la aplicación desde el diseño a la generación del bitstream. Los apartados más importantes son:
    a. **IP Catalog:** aquí se encuentran disponibles todas las IP del catálogo de Xilinx para añadirlas al proyecto. Este proceso se lleva a cabo mediante un asistente en el que también se configura la IP de acuerdo a las necesidades de uso.
    b. **Open Elaborated Design:** genera un esquemático en el que se muestran todos los bloques programados y sus interconexiones entre ellos. Aparecen las señales y sus propiedades, como el tipo o ancho de palabra. Es una herramienta muy útil para depurar.
    c. **Run Synthesis:** lleva a cabo la síntesis del diseño.
    d. **Run Implementation:** tras la síntesis, aquí se implementa el diseño.
    e. **Generate Bitstream:** como el nombre indica, haciendo click en esta opción se crea el bitstream. Se puede seleccionar esta opción directamente sin sintetizar ni implementar antes y automáticamente el programa hará esos dos pasos.
2. **Data Windows Area:** Esta sección muestra información relacionada con los ficheros fuente y de datos, como:
    a. **Sources:** aparecen de forma jerárquica todos los ficheros fuente, las IP usadas, librerías y el orden de compilación. Desde la pestaña *IP Sources* se pueden ver distintas plantillas como las de instanciar IPs o los testbench.
    b. **Properties:** muestra información del objeto seleccionado.
3. **Workspace:** Aquí aparecen ventanas con interfaz gráfica que requieren más espacio de pantalla como el editor de texto, los esquemáticos o el resumen del proyecto.
4. **Results Windows Area:** Muestra el estado y el resultado de comandos ejecutados en el programa, está dividida en varias pestañas que se explican a continuación:
    a. **Tcl Console:** permite introducir comandos Tcl y ver el resultado de comandos anteriores así como de los ejecutados por Vivado, funciona como un terminal de Linux.
    b. **Messages:** muestra todos los mensajes de información, warning y errores del proceso de diseño.
    c. **Log:** aquí aparecen los logs producidos durante los procesos de síntesis, implementación y simulación.
    d. **Reports:** acceso a los informes generados durante los procesos anteriores.
    e. **Designs Runs:** ver y gestionar todos los procesos ejecutados, con vista detallada del último proceso de cada IP.

5 DESCRIPCIÓN DE LA SOLUCIÓN

It's not that I'm so smart, it's just that I stay with problems longer.

- Albert Einstein -

5.1 Introducción

Una vez elegido y preparado todo el entorno de desarrollo y las herramientas a utilizar, es hora de diseñar el sistema. Para ello, primero es necesario conocer más en profundidad el S.O. que se va a utilizar, pues de él depende gran parte del diseño, debido al uso de Xillybus. Así que a continuación se va a describir cómo funciona Xillinux y el driver + IP core de Xillybus que incluye.

5.2 Xillinux

Xillinux es una distribución basada en Ubuntu 12.04 para los dispositivos Zynq-7000, pensada como una plataforma para el desarrollo rápido de software con FPGA. Soporta ZedBoard, MicroZed y Zybo.

Está preparada para ser usada con una configuración de ratón, teclado y pantalla como los Linux de escritorio, pero también es posible controlarla por línea de comandos por el puerto USB UART. Incluye parte de lógica hardware, por ejemplo el adaptador VGA. El resto de características destacables son las que comparte con Ubuntu en su versión 12.04:

- GCC 4.6.3.
- Soporte de actualizaciones durante 5 años.
- Gestor de paquetes para instalar con facilidad aplicaciones y herramientas.
- Distribución estable.
- Comunidad de usuarios muy grande.
- Mucha documentación disponible.

Pero lo que verdaderamente hace interesante a Xillinux es la inclusión de Xillybus, que consiste en un IP core para la FPGA y un driver de Linux para la integración entre la lógica programable del dispositivo y las aplicaciones de espacio de usuario que se ejecutan en procesadores ARM. Con esto solo se necesitan unas habilidades básicas de programación para diseñar una aplicación en la que FPGA y Linux trabajen juntos. [27]

Figura 11 – Entorno gráfico de Xillinux

5.3 Xillybus

5.3.1 Introducción

Xillybus es una solución para el transporte de datos entre FPGA y un host que ejecute Linux o Windows. Está disponible para sistemas embebidos y ordenadores que usen PCI express como medio de transporte de datos.

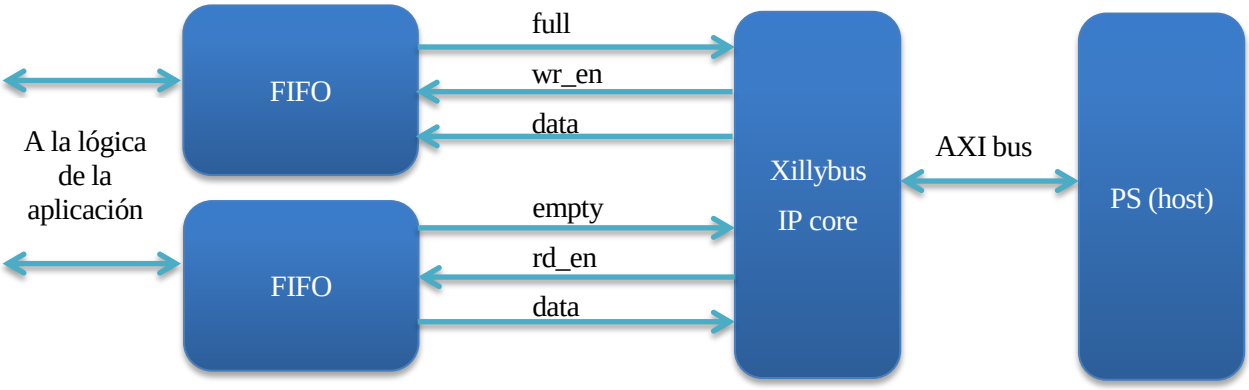

Figura 12 – Esquema de Xillybus (FPGA)

Su funcionamiento es el que se puede ver en la figura 11. El IP core de Xillybus se comunica con la lógica de la aplicación a través de una FIFO con las típicas señales de lectura/escritura en FIFO (data, read/write enable, full/empty). La FIFO tiene libertad en cuanto a anchura y profundidad de acuerdo a los requisitos de la aplicación.

Esta configuración libera al desarrollador de la necesidad de manejar el tráfico con el host. Sencillamente, Xillybus comprueba las señales empty/full e inicia la transferencia de datos hasta que la FIFO se llene o se vacíe, según la dirección del tráfico.

En el otro lado de la comunicación, Xillybus envía los datos utilizando el bus AXI, generando peticiones DMA en el bus del procesador.

La aplicación en el host interactúa con ficheros de dispositivo que se comportan como las tuberías de Linux. Se pueden abrir, leer y escribir como cualquier otro fichero, pero se parecen más a tuberías entre procesos o a flujos TCP/IP. La única diferencia para el programa ejecutándose en el host es que el otro extremo no es un proceso, sino una FIFO en la FPGA.

Figura 13 – Esquema de Xillybus (Linux)

El driver soporta cualquier configuración del IP core de Xillybus. Los flujos y sus atributos son autodetectados por el driver cuando se carga en el sistema operativo del host y se crean los ficheros del dispositivo asociados. Estos ficheros se encuentran en /dev/xillybus_nombre.

Al mismo tiempo, cuando el driver se carga, se reservan buffers DMA en la memoria del host y se informa a la FPGA de sus direcciones. Estos buffers y sus tamaños son independientes para cada flujo de datos. [27]

Al final esto son cosas internas, lo realmente importante para el desarrollador es que solo tiene que preocuparse por leer y escribir en ficheros en Linux y por leer y escribir en FIFOs en la FPGA.

5.3.2 Flujos síncronos vs asíncronos

Cada flujo de datos de Xillybus tiene uno de estos comportamientos. Los flujos asíncronos pueden comunicar datos entre el host y la FPGA sin la intervención del software del espacio de usuario mientras que el fichero de dispositivo esté abierto. Por otra parte, los flujos síncronos solo envían o reciben datos cuando el usuario ejecuta un write o un read sobre el fichero.

Los flujos asíncronos tienen un mejor rendimiento porque el flujo de datos continua en segundo plano incluso cuando la CPU no está atendiendo esa tarea y se encuentra con otra, mientras que los síncronos son más fáciles de manejar y son la opción preferida cuando se necesita una sincronización entre el programa del host y el de la FPGA.

Los flujos síncronos tienen también otra utilidad destacable, se pueden utilizar para escribir y leer posiciones determinadas de la FPGA (*seekable stream*) en una especie de interfaz de memoria. Usando funciones como fseek(), se puede mover la posición del fichero en la que leer o escribir. [28] La posición del flujo se presenta en la lógica de la FPGA en cables separados como una dirección, así que interactuar con memorias o registros es directo. [29]

5.3.2.1 Flujos FPGA a host

Los flujos asíncronos llenan el buffer de la DMA del host cada vez que sea posible, es decir, cuando el fichero es abierto, hay datos disponibles y hay espacio libre en los buffer de la DMA.

En flujos síncronos, el IP core de Xillybus no entregará datos de la FPGA al host hasta que este no los pida explícitamente leyendo el fichero. Los flujos síncronos se deben evitar en aplicaciones que requieran un gran ancho de banda, por dos razones:

- El flujo de datos se interrumpe cuando a la CPU no le toca ejecutar la aplicación, así que el canal se queda sin utilizar durante cierto periodo de tiempo.
- La FIFO que almacena los datos puede desbordarse durante este periodo en el que la aplicación no está extrayendo datos de ellas.

5.3.2.2 Flujos host a FPGA

En flujos asíncronos, las llamadas a funciones que escriben en el fichero para enviar datos a la FPGA retornan inmediatamente si los datos caben enteros en los buffers de la DMA. Estos datos son transmitidos después a la FPGA cuando una de estas condiciones ocurre:

- El buffer de la DMA se llena.
- El descriptor del fichero es explícitamente vaciado.
- El descriptor de fichero se cierra.
- Un temporizador expira, forzando un vaciado automático si no se ha escrito nada durante cierto espacio de tiempo, normalmente 10 ms.

Para flujos síncronos, la llamada a la función de escritura en el fichero no retornará hasta que los datos hayan llegado a las FIFO de la FPGA[1]. Debe ser evitado en aplicaciones que requieran gran ancho de banda por los mismos dos motivos mencionados anteriormente. [29]

5.3.3 IP core

El IP core de Xillybus está pensado para interactuar con la lógica de la aplicación mediante FIFOs o memorias síncronas. También se podría sustituir la FIFO por lógica que imite su comportamiento, aunque no es recomendable, al menos en los primeros diseños.

Hay que tener en cuenta varios aspectos antes de trabajar con el IP core: el reloj, el ancho de palabra y las señales que conforman su interfaz.

Todas las señales que entren y salgan del IP core deben estar sincronizadas con la señal bus_clk que proporciona el propio core. Este reloj lo genera el PCIe o el procesador y tiene una frecuencia que depende de la plataforma (100 MHz en Zynq), aunque también se puede configurar la frecuencia dentro de una lista de valores. [30]

Puede trabajar con anchos de palabra de 8,16 y 32 bits, cuanto mayor ancho más rendimiento en la transmisión de los datos. Esto es debido a que las palabras se transportan a través de caminos internos del IP core a la tasa bus_clk. En consecuencia, transportar una palabra de 8 bits ocupa el mismo intervalo de tiempo que transportar una de 32 bits, lo que lo hace 4 veces más lento. [31]

5.3.3.1 Descripción de las señales

Todas las señales que usa Xillybus siguen una cierta convención para nombrarlas que facilita su manejo. Cada señal tiene un nombre que se divide en cuatro partes separadas por el carácter '_': *user_w_devfile_func*.

1. El prefijo "user" es común a todas las señales.
2. Una bandera que puede ser 'w' o 'r' (write o read), que indica si el flujo es en dirección host-FPGA o FPGA-host respectivamente. Se usa el punto de vista del host.
3. "devfile" es el nombre que se le quiera dar a ese flujo de datos. Esta cadena también será el nombre del fichero de dispositivo que se crea en el host: /dev/xillybus_devfile.
4. "func" hace referencia a la función de esa señal (data, wren, full…).

Solo serán necesarias cuatro señales en caso de dirección host-FPGA y cinco en FPGA-host para establecer la comunicación con el IP core.

[1] Las funciones de I/O de más alto nivel, como fwrite(), añaden una capa más con su propio buffer. Por lo tanto, estas funciones pueden retornar antes que los datos hayan llegado a la FIFO.

5.3.3.2 Señales para transmisión host-FPGA

- **user_w_devfile_data:** señal de salida del core que contiene los datos que se quieren enviar. Tendrá ancho de 8 bits, 16 bits o 32 bits.
- **user_w_devfile_wren:** señal de salida del core, es un "write enable" para los datos. Está activa a la vez que hay datos válidos en la señal user_w_devfile_data.
- **user_w_devfile_full**: señal de entrada del core, indica que la FIFO está llena y no puede aceptar más datos. Esta señal se activa un ciclo de reloj después de último ciclo de escritura que la llenó, que es el comportamiento estándar de las FIFO. Mientras esté activa asegura que no se producen más ciclos de escritura.
- **user_w_devfile_open:** señal de salida del core que se activa ('1') cuando el correspondiente fichero del host se abre para escritura. Un uso típico de esta señal es resetear la FIFO u otra lógica entre sucesivas aperturas del fichero (usada como un reset activo a nivel bajo). Si el fichero es abierto en el host por múltiples procesos (como resultado de un fork), esta señal se mantiene activa hasta que todas las instancias se cierren.

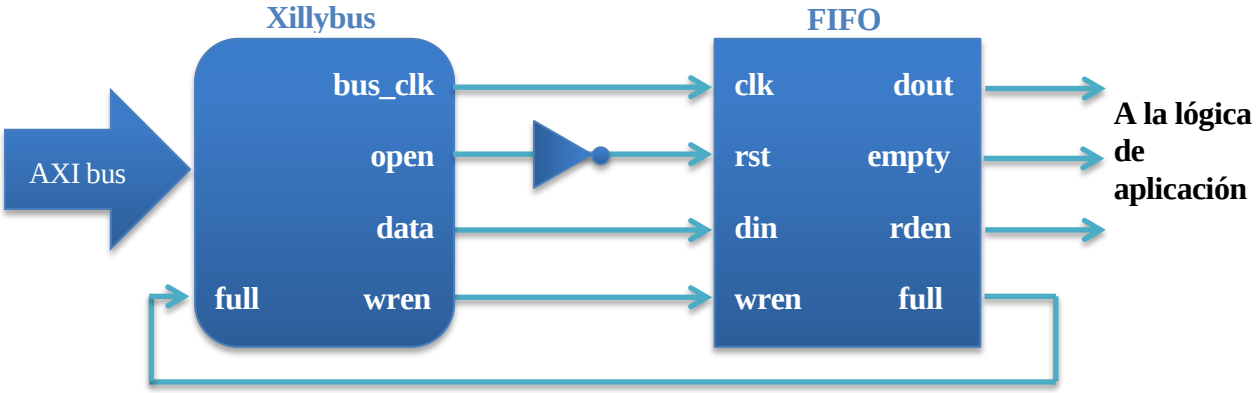

Figura 14 – Ejemplo de conexión en dirección Host-FPGA

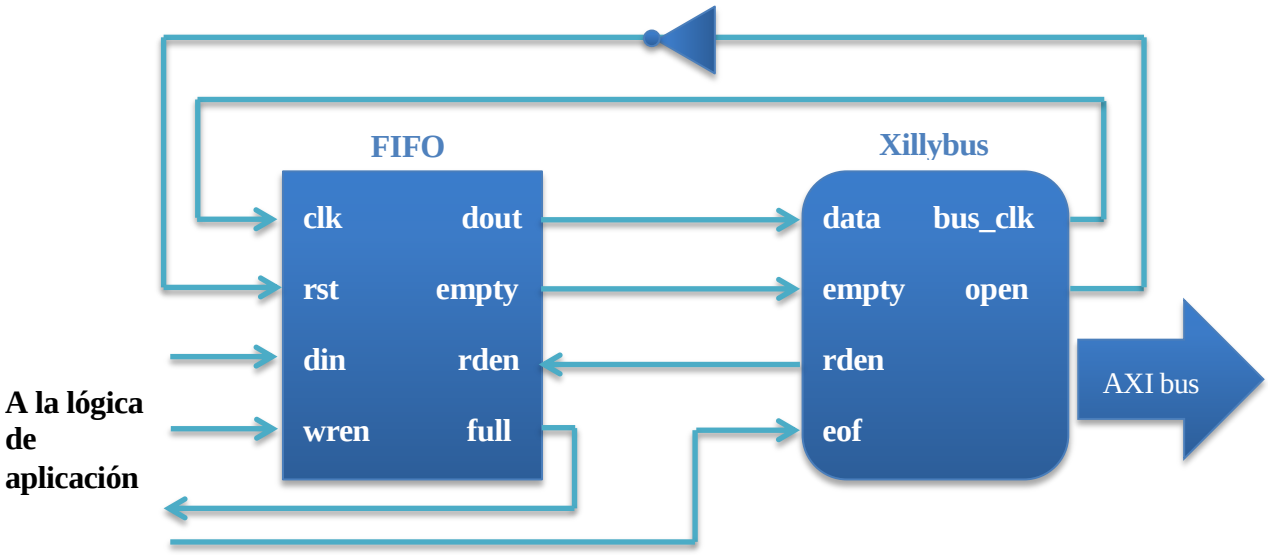

Figura 15 - Ejemplo de conexión en dirección FPGA-Host

5.3.3.3 Señales para transmisión FPGA-host

- **user_r_devfile_data:** entrada del core que contiene los datos durante el ciclo de lectura. Como se ha dicho ya, puede tener ancho de 8, 16 o 32 bits. Esta señal contendrá los datos un ciclo de reloj después de haber activado el "read enable", que es el comportamiento estándar de una FIFO.
- **user_r_devfile_rden:** salida del core que tiene la función "read enable". Cuando se activa, el core espera que haya datos válidos en user_r_devfile_data en el próximo ciclo de reloj.
- **user_r_devfile_empty:** entrada al core que informa no se pueden leer más datos porque la FIFO está vacía. La señal pasa de 0 a 1 un ciclo después de la última lectura. Cuando está activa asegura que no se producen más lecturas.
- **user_r_devfile_open:** salida del core que se activa cuando el fichero correspondiente del host es abierto para lectura. Esta señal suele usarse para resetear la FIFO u otra lógica entre sucesivas aberturas del fichero (usado como reset activo a nivel bajo). Si el fichero es abierto en el host por múltiples procesos (como resultado de un fork), esta señal se mantiene activa hasta que todas las instancias se cierren.
- **user_r_devfile_eof:** entrada del core que le indica que debe generar un evento end-of-file (fin de fichero). Funciona como la señal *empty*, con la diferencia que una vez que se activa, el core no intentará más lecturas hasta que el fichero en el host sea cerrado y reabierto. En la parte del host, la aplicación que estuviese leyendo el fichero recibe un EOF.

  Esta señal al igual que *empty* debe ser activada solo en un ciclo de reloj después de una lectura. También podría ponerse a '1' en cualquier ciclo de reloj si la señal *empty* ya está activada, así que una forma de asegurarse que se usa correctamente es tenerla combinada en una puerta AND junto a *empty*.

Anteriormente se mencionó que Xillybus permitía leer y escribir en determinadas posiciones imitando el comportamiento de una interfaz de memoria. Para ello se hace uso de dos señales más: *user_devfile_addr* y *user_devfile_addr_update*. En este trabajo no se usarán porque no se implementa ninguna interfaz de este tipo, pero su uso es también muy sencillo y se puede encontrar más información en [30].

5.3.4 Ancho de banda y latencia

Ancho de banda y latencia son dos parámetros importantes a la hora de diseñar sistemas mixtos host-FPGA. En cuanto a Xillybus, el ancho de banda depende de la capa de transporte inferior (PCIe y AXI), en el caso de la ZedBoard se usa AXI así que se explicará este.

El bus AMBA usa distintas señales para datos y direcciones, así que teóricamente todo el bus de datos puede ser aprovechado sin tener perdida por incluir las direcciones. La frecuencia de reloj es dada por la lógica de aplicación a elegir entre un rango de valores. Tomando una frecuencia de reloj relativamente alta de 150 MHz con un ancho del bus de datos de 64 bits, se obtienen 1,2 GB/s.

Sin embargo Xillybus no alcanza toda la velocidad posible, porque su filosofía se basa en la simplicidad más que en el rendimiento y esgrime como argumento que en todos los escenarios, un programa en la parte software está involucrado. Este software, incluso con los procesadores de hoy en día, difícilmente trabaja a tasas mayores de 150-200 MB/s.

El límite en Xillybus viene impuesto principalmente por el reloj que gobierna el IP core (bus_clk) y el ancho del camino que siguen los datos (32 bits). En el caso de la ZedBoard el límite teórico es de 100 MHz x 32 bits = 400 MB/s.

Aun así, la demanda de ancho de banda se ha incrementado así que en 2016 se liberarán nuevas versiones del IP core de Xillybus con una mayor capacidad. [32]

Para la latencia se han hecho pruebas que indican que el valor típico está en el rango de los 10-50µs desde que se ejecuta un write() hasta que los datos llegan a la FIFO en la FPGA, esto por supuesto depende del hardware. Existen dos causas que provocan un aumento de la latencia: incorrectas prácticas de programación y tiempo de dedicación de la CPU a la aplicación. Una buena explicación de estas prácticas se puede encontrar en [33].

5.4 Codiseño HW/SW

Tras haber entendido cómo funciona Xillybus, se puede comenzar a diseñar una aplicación. Para este trabajo se ha preparado una que muestre varias formas de interactuar entre hardware y software. Aspectos como la comunicación en tiempo real entre PL y PS, control de parámetros de la lógica de la FPGA desde Linux en tiempo de ejecución o acceso a los GPIOs de la PL desde software.

De entre la gran variedad de aplicaciones que se podrían desarrollar, se ha optado por una de audio. Xillybus proporciona una demo en la que aparecen ejemplos de cómo usar el IP core y un desarrollo que usa audio y VGA para mostrar el escritorio gráfico de Linux conectando la Zedboard a un monitor. Aprovechando ese desarrollo de audio se harán unas modificaciones para implementar la aplicación.

En la figura 15 se ve el camino que sigue el audio en Xillinux. Entra por la entrada de micrófono o línea IN de la Zedboard que están conectadas a la PL. De aquí se envía por Xillybus hasta el PS. De esta forma se puede acceder desde Linux al audio que se recoge del exterior. De igual manera, si desde Linux se genera sonido, se envía a la PL donde se encuentra la salida de auriculares o altavoces y se puede oír.

Figura 16 – Camino por defecto del audio

La aplicación a que se va a desarrollar va a recoger el audio de entrada antes de que llegue al PS, hará procesamiento de este en la FPGA y enviará los datos procesados a Linux para su representación, de la manera que se muestra en la siguiente figura:

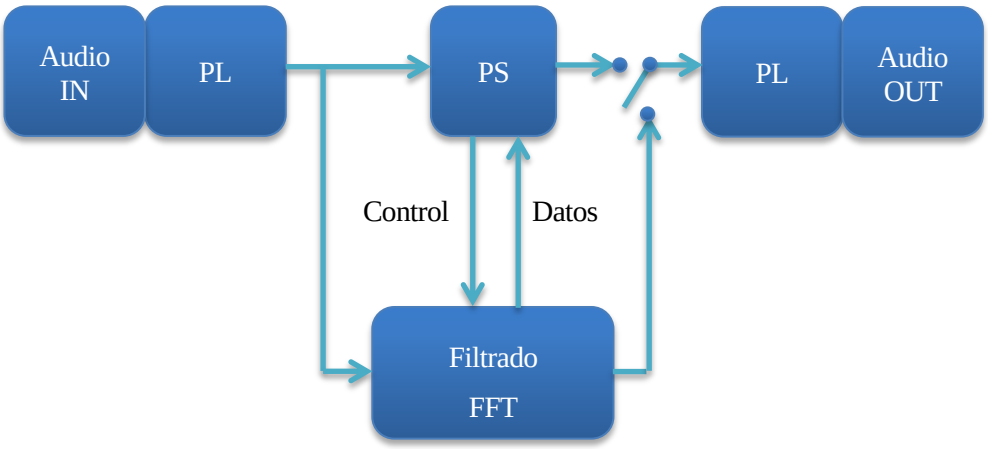

Figura 17 – Esquema de aplicación propuesta (PL)

El procesamiento va a consistir en el filtrado de la señal y el cálculo de su FFT. El resultado de la FFT se va a enviar a Linux para la representación del espectrograma de la señal. Esto va a requerir mandar al PS datos en tiempo real del audio que va entrando en el sistema. También desde el PS se controlará el filtrado implementado en la PL mediante señales de control.

Todo esto será el desarrollo en la lógica programable, ahora en la parte software hay que implementar el control de la FPGA y la recepción de los datos de la FFT. Para ello se va a programar una aplicación web que cumpla estos propósitos y además represente los datos en forma de espectrograma. El hecho de elegir una aplicación web en vez de otra local basada en C, Java o cualquier otro lenguaje, aporta la ventaja de comunicarse con la ZedBoard desde cualquier otro dispositivo conectado a Internet y de poder controlar y ver los resultados desde él.

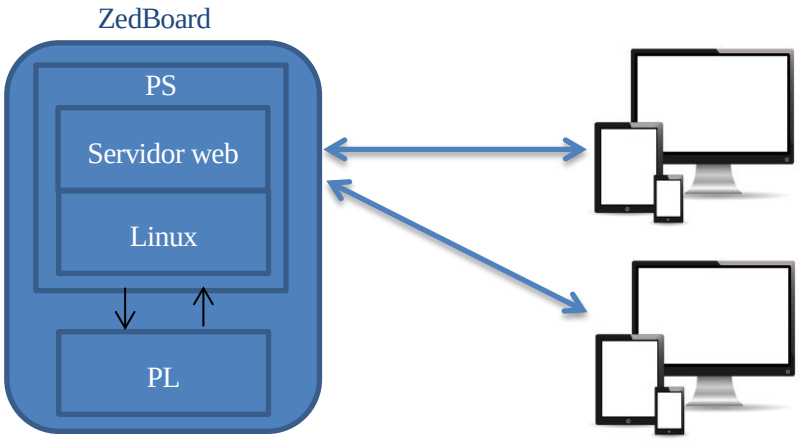

Figura 18 – Esquema de aplicación propuesta

Por último, se hará un ejemplo de acceso a los GPIOs y también se incluirán en la aplicación web para que puedan ser controlados remotamente desde cualquier dispositivo externo. Un esquema más detallado de la arquitectura y tecnologías usadas es el siguiente:

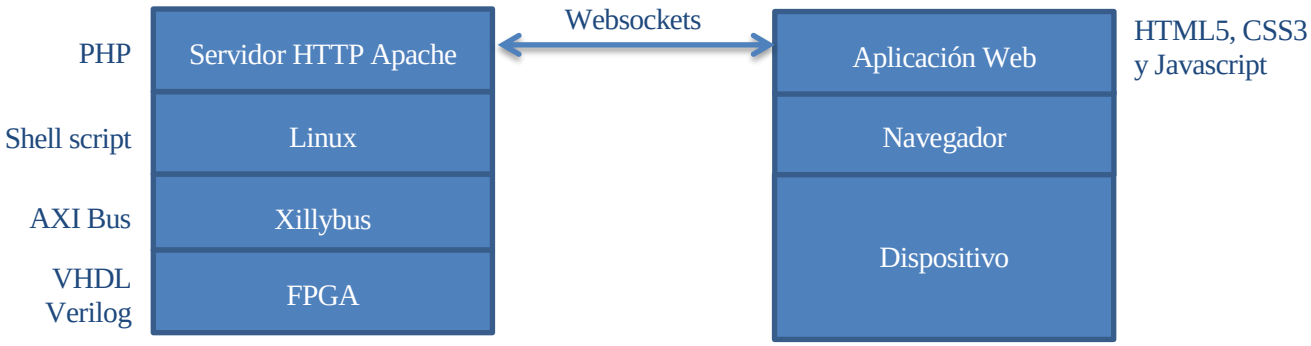

Figura 19 – Arquitectura y tecnologías usadas

5.4.1 Espectrograma

El ejemplo más importante que se va a desarrollar en este trabajo es el espectrograma. Por ello es conveniente explicar en qué consiste y cómo funciona. Un espectrograma es la representación del espectro de frecuencias de una señal respecto al tiempo. El análisis del espectrograma permite entender no sólo la distribución espectral de una señal sino también seguir su evolución a lo largo del tiempo.

Los espectrogramas tienen varias aplicaciones como la identificación de palabras, sismología, desarrollo de RF y microondas o procesadores de señales.

La forma más habitual de representación es mediante una gráfica en dos dimensiones. En el eje vertical se muestran las frecuencias y en el horizontal el tiempo; y se añade una tercera dimensión mediante la intensidad de color, que marca la amplitud de la señal en cada frecuencia.

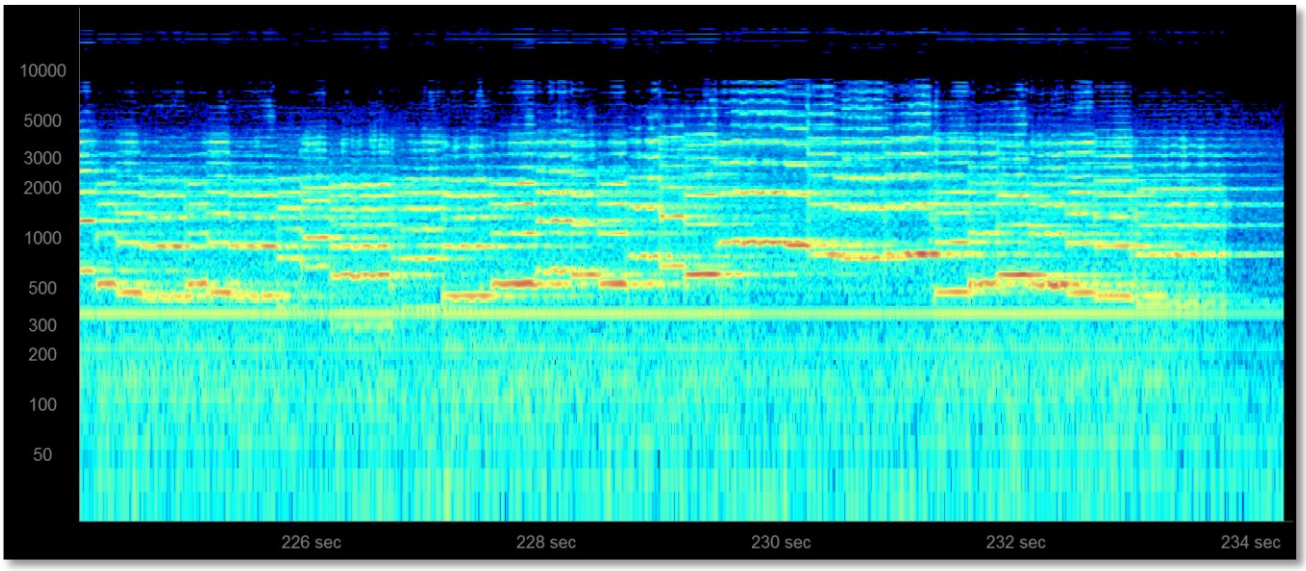

Figura 20 – Espectrograma de un violín

Un espectrograma se crea calculando la magnitud al cuadrado de la STFT (Short-Time Fourier Transform) de la señal descompuesta en varios trozos. La colocación consecutiva en el tiempo de estas STFT en cada trozo genera el espectrograma.

$$espectrograma(n, w) = |STFT(n, w)|^2$$

5.4.1.1 STFT (Short-Time Fourier Transform)

La STFT es una transformada de Fourier usada para calcular el espectro de una señal mientras esta cambia en el tiempo. El procedimiento consiste en dividir la señal en segmentos más cortos de igual longitud y después calcular la transformada de Fourier de cada uno de ellos:

$$STFT = X(n, w) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-j\omega n}$$

Donde x[n] es la señal de entrada y w[n] una función ventana que reduce los artefactos de aplicar la transformada a cada intervalo. Por lo tanto el proceso se reduce a tomar n muestras de la señal, calcular su transformada, recoger las siguientes n muestras, calcular su transformada y así sucesivamente. De cada transformada se calcula su módulo, se eleva al cuadrado y se representa, lo que hecho de forma continua genera el espectrograma.

Para cada transformada se usará el algoritmo de la FFT, que como cualquier otro algoritmo de cálculo de la transformada, genera N valores X(k) a partir de N muestras x(n). Al tratarse de audio, las muestras son valores reales y se tiene que los valores del conjunto X(k) muestran la siguiente simetría:

$$X(N-k) = X^*(k)$$

Siendo k=N/2, el punto medio del conjunto X(k). Por tanto, de cada N valores, la transformada genera N/2 + 1 valores diferentes, pudiéndose obviar la segunda mitad de ellos, de aquí también se puede deducir que la frecuencia más alta será f(N/2). Por último, el espaciado entre los puntos en frecuencia (bin) es constante Δf [34].

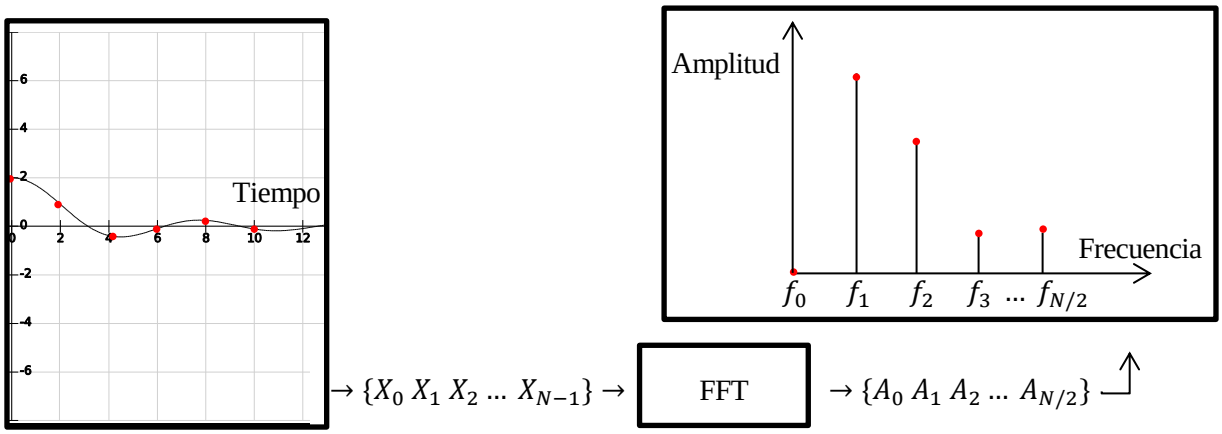

Figura 21 – Operación de la FFT

Particularizando en nuestro caso, la ZedBoard tiene un códec de audio que muestrea a 48 kHz, lo que limita la frecuencia máxima a:

$$f_{N/2} = \frac{f_s}{2} = 24\ kHz$$

El número de puntos N es un valor que se puede escoger teniendo en cuenta que a mayor número de muestras, más resolución en frecuencia y tamaño de ventana se tendrá, pero también habrá un mayor número de datos sobre lo que operar. Con un valor de 1024 será suficiente para esta aplicación, lo que da unos resultados de:

$$\Delta f = \frac{f_s}{N} = 46{,}875 Hz$$

$$T_w = \frac{1}{\Delta f} = 21{,}3\ ms$$

En resumen, la FPGA de la ZedBoard registrará 1024 muestras cada 21,3 ms, a este conjunto de muestras le calculará la FFT y el resultado será enviado a Linux para que sea representado en un espectrograma. Todo este proceso debe producirse en un tiempo inferior al de T_w, para que se pueda seguir recibiendo el siguiente conjunto de muestras en tiempo real sin que se produzcan retrasos.

6 DESARROLLO DE LA APLICACIÓN DE EJEMPLO

Ever tried. Ever failed. No matter. Try again.
Fail again. Fail better.

- Samuel Beckett -

6.1 Instalación de Xillinux

El único material necesario es una tarjeta SD, la cual se recomienda que sea de la marca Sandisk porque otras suelen reportar errores. En esta tarjeta se copiarán los archivos necesarios para el arranque de Xillinux. Los archivos pueden encontrarse en su página web (http://xillybus.com/xillinux) y son dos:

- La imagen de Xillinux lista para cargar en la SD.
- Kit de la partición de arranque.

Lo primero es descomprimir el kit de arranque, que tendrá un nombre con este formato xillinux-eval-board-XXX.zip. Contiene varios directorios en los que están los ficheros con los que se va a estar trabajando en Vivado, por ello es aconsejable descomprimirlo en el directorio donde se suela trabajar[1].

Para el desarrollo hardware se puede trabajar tanto en Verilog como en VHDL. En este trabajo se usará VHDL así que el primer paso es ir a vhdl/src/xillydemo.vhd y editar este fichero. Es necesario eliminar las tres siguientes líneas que están al principio:

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

Y descomentar estas líneas en la definición de la arquitectura:

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

Tras esto, se arranca Vivado, y sin abrir ningún proyecto se hace click en Pick Tools > Run Tcl Script y se elige el archivo xillydemo-vivado.tcl en el subdirectorio verilog/ o vhdl/, dependiendo del lenguaje que se vaya a usar. Esto desencadenará una serie de tareas y creará el proyecto en Vivado (aparecerán warnings, pero no errores).

A partir de aquí ya es posible editar cualquier fichero y se podría comenzar el desarrollo hardware, pero vamos a continuar con el proceso hasta arrancar la placa por primera vez.

Sin hacer ninguna modificación más puede pulsarse en *Generate bitstream*, que por sí solo ejecutará la síntesis e implementación. El bitstream generado puede encontrarse en vivado/xillydemo.runs/impl_1/.

[1] La ruta del directorio de trabajo no debe incluir espacios en blanco.

Con el bitstream ya están listos todos los ficheros que hay que incluir en la tarjeta SD. Ahora hay que descomprimir el segundo archivo descargado (xillinux-x.x.img.gz) y escribirlo en la SD.

Esta imagen contiene una tabla de particiones, una partición FAT donde se colocará el bitstream generado y dos archivos más, y otra partición ext4 con la raíz del sistema de ficheros de Linux. La segunda partición no es reconocida por los ordenadores que usan Windows, así que la SD puede aparecer con muy poco tamaño, unos 16MB.

El proceso es muy sencillo y existen varias formas de llevarlo a cabo. Desde Windows se necesita algún programa externo que cargue una imagen en una tarjeta SD, por ejemplo *Win32 Disk Imager* que está disponible para descargar desde varias páginas web. Sea cual sea el programa utilizado, solo hay que seleccionar como imagen la descargada de Xillinux y como medio de instalación la tarjeta SD.

Tras terminar el proceso, se extrae la tarjeta SD y se vuelve a insertar en el PC para que reconozca sus nuevas particiones. Al abrirla aparecerá un único archivo en ella llamado uImage. Junto a él será necesario copiar tres archivos más:

- El bitstream que se generó (xillidemo.bit) que se encuentra en la ruta ya mencionada.
- Los ficheros boot.bin y devicetree.dtb que están en el kit de la partición de arranque en el subdirectorio bootfiles/.

Así acaba el proceso de construcción de Xillinux, para comprobar que todo está bien la tarjeta SD debe contener estos cuatro archivos en la partición de arranque (la única que se muestra si se está usando Windows):

- **uImage**: que corresponde al binario del kernel de Linux.
- **boot.bin**: contiene las inicializaciones del procesador y el U-boot.
- **devicetree.dtb**: el device tree que contiene la información del hardware para que el kernel sepa que drivers debe cargar.
- **xillydemo.bit**: el bitstream generado en Vivado.

Ya se puede extraer la tarjeta SD e insertarla en la ZedBoard, solo hay que tener una cosa más en cuenta, la colocación de los jumpers de la placa, ya que uno de ellos cambia respecto a la configuración detallada en la guía de hardware de la ZedBoard [11]. En concreto, para Xillinux hay que colocar un jumper en JP2 para que los dispositivos USB conectados a la placa (ratón y teclado) reciban su alimentación de 5V. La colocación de todos los jumpers debe quedar como en la figura 18.

Por último, conectar los periféricos que se vayan a usar: se puede optar por una configuración de escritorio con monitor, ratón y teclado en la cual la placa arrancaría gráficamente y se puede usar como cualquier Linux de PC. Por otra parte, se puede conectar al PC por el puerto USB UART y controlar la ZedBoard por consola. Para este caso hay que configurar la transmisión por UART con la siguiente configuración: 115.200 baudios, 8 bits de datos, 1 bit de parada y sin control de flujo. Windows no tiene ningún programa para comunicación UART por USB pero hay muchos por Internet disponibles, el usado para este trabajo es TeraTerm.

Aquí finaliza el proceso para arrancar la placa por primera vez. Tras el primer arranque es recomendable preparar dos cosas más: ajustar el tamaño del sistema de archivos para aprovechar más la capacidad de la SD y permitir conexiones SSH.

Para lo primero hay una guía rápida en el capítulo 4.4.1 de [27] . Por ejemplo en una SD de 8 GB, aparece que solo existen 1.7 GB pero tras esta operación el espacio total llega a los 8 GB que deberían ser.

Y para SSH es tan sencillo como ejecutar el comando:

```
# apt-get install ssh-server
# passwd root
```

Hay que asignar alguna contraseña al usuario porque ssh no deja hacer inicio de sesión a nadie sin contraseña.

Este proceso no hay que realizarlo más veces, cuando se haga algún cambio en la FPGA desde Vivado, se genera un nuevo bitstream y se sustituye por el que exista en la tarjeta SD. Se inserta la tarjeta en la ZedBoard y ya están los cambios aplicados, las iteraciones son muy rápidas como se puede observar.

Si se quiere consultar una guía más detallada o resolver algún error que haya surgido en el proceso (no debería aparecer ninguno si se siguen correctamente los pasos indicados), se puede encontrar en [27] .

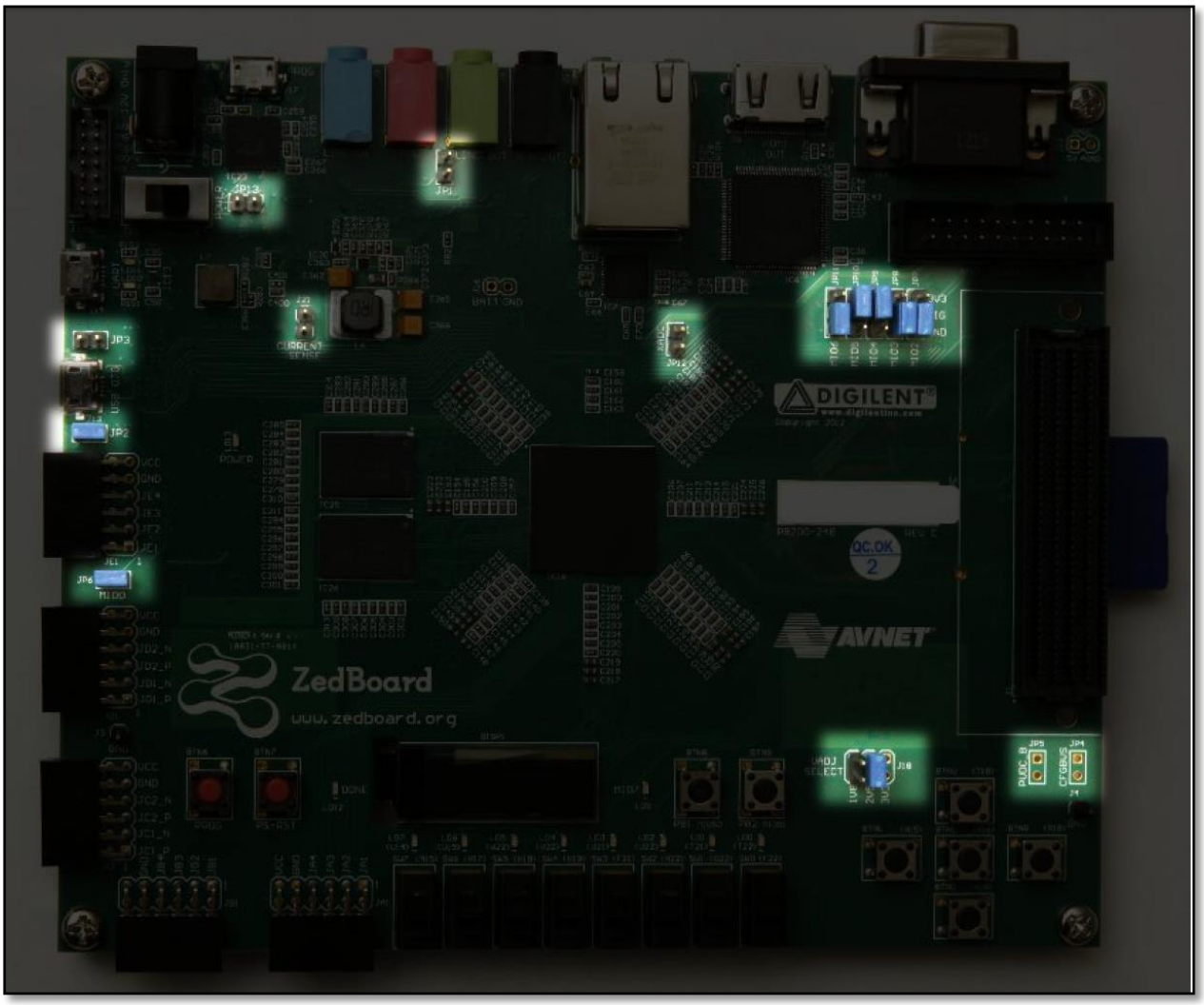

Figura 22 – Colocación de los jumpers en la ZedBoard [27]

6.2 Configuración del IP core

Como se ha descrito en el capítulo 5.4, van a existir dos flujos entre Linux y FPGA, uno para control y otro para datos. Hay que generar un IP core de Xillybus que esté preparado para manejar esos flujos, ajustando sus parámetros de rendimiento y recursos consumidos. Esto se hace a través de un asistente en una web de Xillybus, es un proceso rápido de unos minutos que se puede hacer siguiendo las indicaciones de la página, pero está bien conocer algunos detalles antes.

La página web encargada del proceso es http://xillybus.com/custom-ip-factory. En ella lo primero será registrarse para que te permita descargar los cores generados. Tras ello, se puede comenzar a crear el core personalizado. El proceso consta de tres pasos, el primero corresponde a la pantalla de la figura 23.

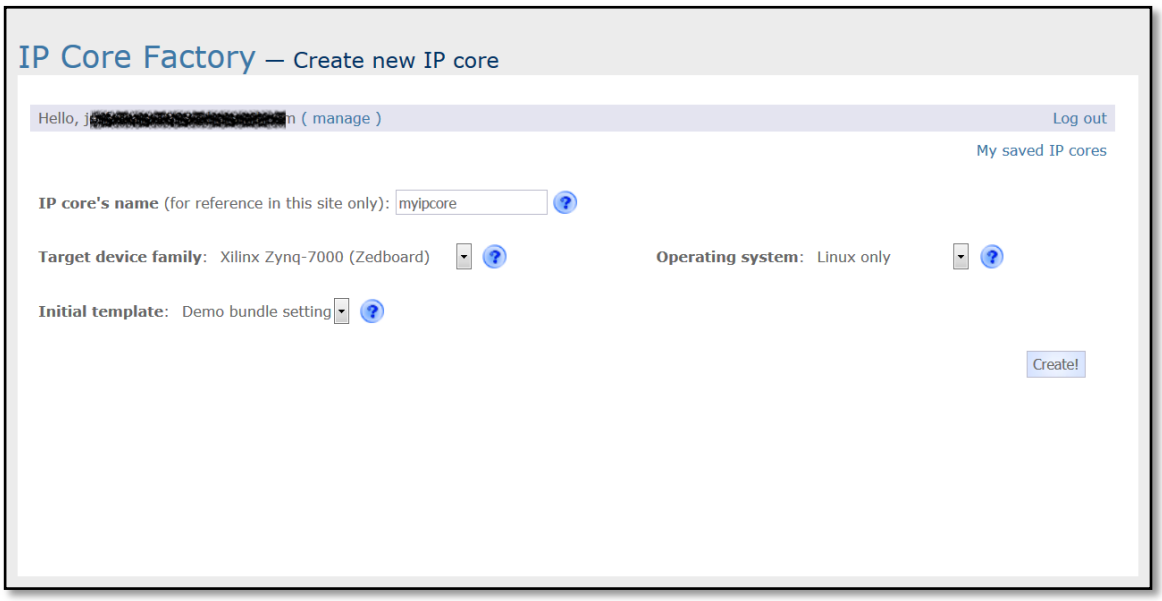

Figura 23 – Creación del IP core (1)

Es bastante explicativo por sí solo, hay que elegir como dispositivo Xilinx Zynq-7000 (ZedBoard), sistema operativo Linux y como plantilla inicial se puede escoger vacía o con la configuración de la demo que trae Xillinux. Cualquier opción es válida en este campo, pero la primera vez es interesante dejar incluida la configuración de la demo porque así se puede probar y entender mejor el funcionamiento de Xillybus, con los ejemplos que trae la propia distribución.

En la siguiente pantalla aparece un resumen del IP core, con los ficheros de dispositivo que contiene (flujos que maneja) y sus características. Si se eligió en el paso anterior la configuración de la demo aparecerán 7 ficheros, en cambio si se eligió vacío, en esta sección no habrá ninguno. En cualquier caso hay que crear los dos flujos que se van a usar, así que debajo de la página hay que pulsar *Add a new device file*.

Aparece una nueva sección que ahora te deja configurar los parámetros del flujo (Figura 24). Empezaremos por el flujo de control. El primer parámetro es el nombre que tendrá el fichero de dispositivo al que acceder desde Linux. Se ha escogido de nombre xillybus_switch porque una de las primeras funciones que iba a tener es hacer de conmutador. La dirección en este caso es *Downstream (host to FPGA)*. En el parámetro uso hay varias opciones para elegir según las características que tenga el flujo de datos, la opción seleccionada es importante y merece la pena entender bien en qué consiste

El uso que se seleccione aquí influirá en el tamaño del buffer que se cree y en el control de flujo cuando la opción *Autoset internals* esté seleccionada (esta opción se explicará un poco más adelante). Por ejemplo:

- Cuando se elige *data acquisition /playback* o *frame grabbing / video playback*, se crean unos buffers con un tamaño y control de flujo adecuados para facilitar un flujo continuo de datos.
- *Data Exchange with coprocessor* asume que se va a requerir un gran rendimiento, pero no necesariamente continuidad.
- Cuando se elige *address/data interface*, el fichero es *seekable* (se puede avanzar y retroceder en él con fseek()), lo que crea un flujo síncrono en ambas direcciones.

El uso que se la va a dar en este trabajo es el que requiere menos recursos, se enviará un byte de datos cada bastante tiempo cuando se quiera cambiar alguna configuración, así que se selecciona *General purpose*.

El ancho de bus de datos 8 bits, que es el más pequeño, ancho de banda esperado de 10 B/s y la opción *Autoset internals* marcada.

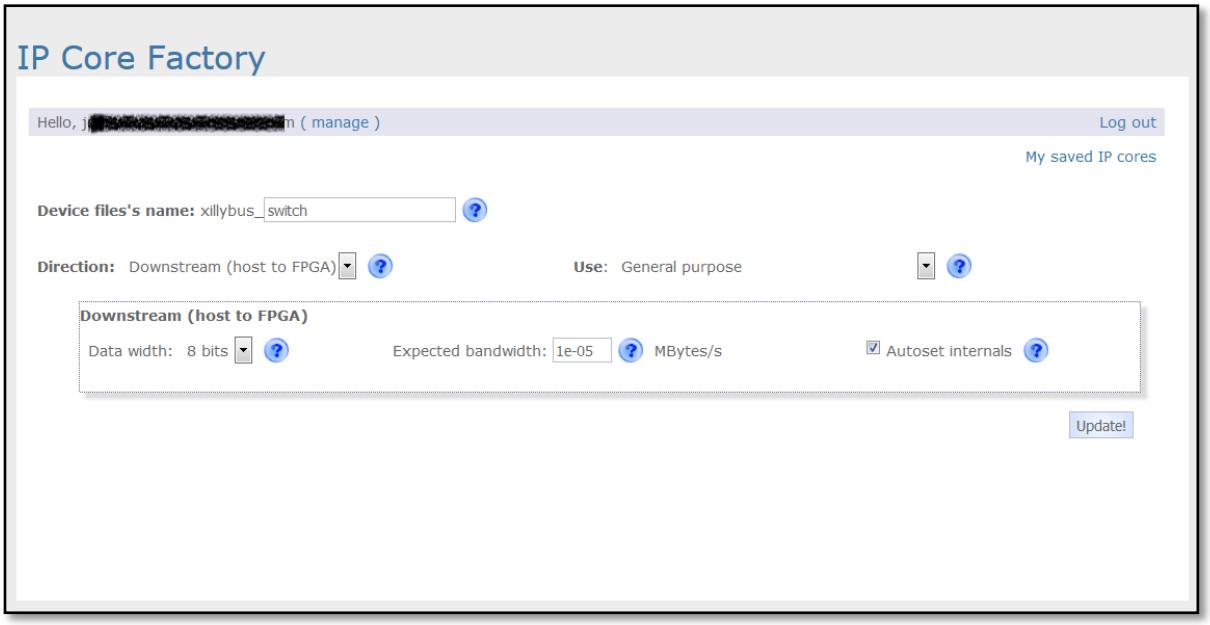

Figura 24 – Creación del IP core (2)

Esta es otra de las opciones importantes que merece una explicación con detalle. Cuando se selecciona, automáticamente decide qué tipo de flujo se va a necesitar (asíncrono o síncrono), número de buffers DMA, tamaño de cada buffer y aceleración DMA (este último solo se tiene en cuenta cuando la comunicación va sobre PCIe, en nuestro caso se usa el bus AXI, así que no aplica).

Estas decisiones son tomadas teniendo como referencia los parámetros indicados en los campos de uso, ancho de banda, dirección y ancho de datos, por eso es importante elegir en estos unos valores adecuados a la aplicación. Es recomendable dejar la opción activada, porque hacerlo manualmente puede llevar a efectos negativos en el rendimiento si no se entiende bien lo que se está haciendo. Si aun así es totalmente necesario hacerlo manualmente para ajustar algún detalle que no funciona, se puede encontrar más información de cómo afecta cada parámetro en [31].

Así se configura un flujo, el otro flujo necesario se genera de la misma forma, pero con los parámetros ajustados a él. En este caso la dirección debe ser *Upstream (FPGA to host)*, el uso se ha escogido *Data acquisition / playback*, porque se va a necesitar un flujo continuo que envíe los datos de la FFT en tiempo real. El ancho de los datos el máximo posible, 32 bits y el ancho de banda esperado hay que calcularlo para tener una cifra aproximada.

Para ello, se miran las características del códec de audio de la ZedBoard [11]. Este funciona con una tasa de muestreo de 48 KHz, lo que quiere decir que a esa tasa la FFT recibirá datos y sacará su salida. La salida es de 32 bits que es el máximo admitido por Xillybus, por lo que el ancho de banda será de unos $48\ KHz * 32\ bits = 1.536\ Kb/s = 192\ KB/s \approx 0{,}2\ MB/s$.

Como se ha elegido un uso para flujos en tiempo real, aparece un nuevo parámetro llamado *Buffering time* que es usado por *Autoset internals* para calcular el tamaño del buffer DMA que se debe reservar para el flujo. El total de memoria reservada para el flujo viene dado por el producto de este tiempo de buffer y el ancho de banda esperado. El tiempo de buffer por defecto para flujos que no son de tiempo real es de 10 ms, por elegir un valor mayor se ha escogido 50 ms, lo que reservara un buffer de $0{,}2\frac{MB}{s} * 50\ ms = 10\ KB$. Con este tamaño y sabiendo que cada muestra ocupa 32 bits, se podrían almacenar en el buffer 2.500 muestras, que a una tasa de 48 KHz corresponde a 52,5 ms de buffering. Útil si la CPU está ciertos milisegundos atendiendo a otra tarea y no puede recoger los datos que le llegan. Con este valor se cubre un tiempo de 52,5 ms que la CPU puede no estar atendiendo a este flujo sin que desborde el buffer. Si el valor no fuera suficiente y se perdiesen muestras, habría que aumentarlo.

Hay que resaltar que esta es la memoria aproximada que puedes esperar que el algoritmo automático reserve. Pero puede que no haya memoria suficiente disponible y reserve menos, así que es importante comprobarlo en el fichero README que se descarga junto al IP core al terminar de configurarlo. El flujo quedará configurado como en la figura 21.

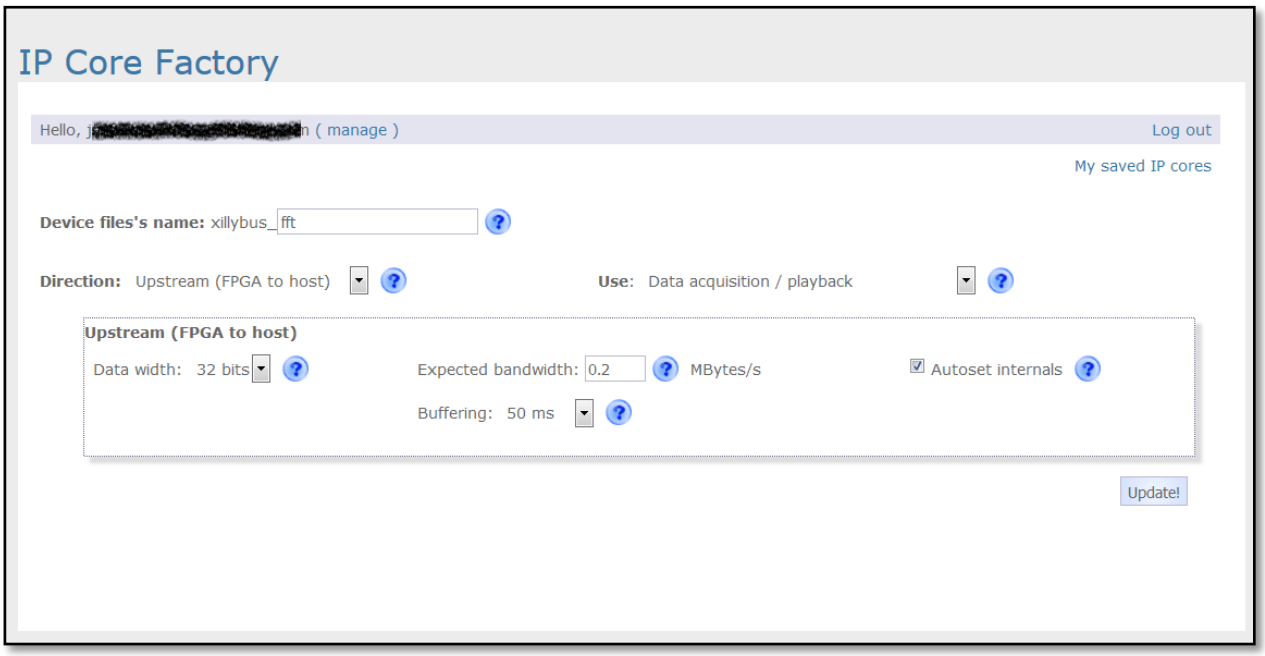

Figura 25 – Creación del IP core (3)

Una vez creados los dos ficheros de dispositivo, se pulsa en *generate core* y tras esperar uno o dos minutos el core estará listo para descargar. Viene comprimido en un zip junto a unos ficheros más que también serán necesarios. El primero a tener en cuenta es el README que contiene las instrucciones para incluir el nuevo core y la descripción de los flujos creados, es interesante mirar aquí las características que ha generado el *Autoset internals* como se ha comentado anteriormente.

Aparte del README, hay cuatro ficheros más que se deben usar:

- Los ficheros xillybus.v y xillybus_core.v tienen que reemplazar a los ya existentes que se encuentran en el subdirectorio vhdl/src/.
- El fichero xillybus_core.ngc también tiene que reemplazar al ya existente en el subdirectorio cores/.
- El fichero xillydemo.vhd que está dentro del directorio instantiation templates/ contiene la nueva arquitectura de la entidad xillydemo. Se abre este archivo, se copia la arquitectura y se sustituye la arquitectura del fichero xillydemo.vhd que está en el subdirectorio vhdl/src/. Este fue el fichero en el que se comentaron y descomentaron algunas líneas al principio del capítulo.

Con esas tres modificaciones ya está insertado el nuevo core en el proyecto de Vivado. Este es el proceso que hay que realizar cada vez que se añada un flujo de datos nuevo entre HW y SW.

6.3 Desarrollo hardware

Para empezar a diseñar la parte hardware, se va a partir de la Figura 16 para desgranarla y especificar con más detalle lo que se va a desarrollar. El diseño contendrá los siguientes bloques:

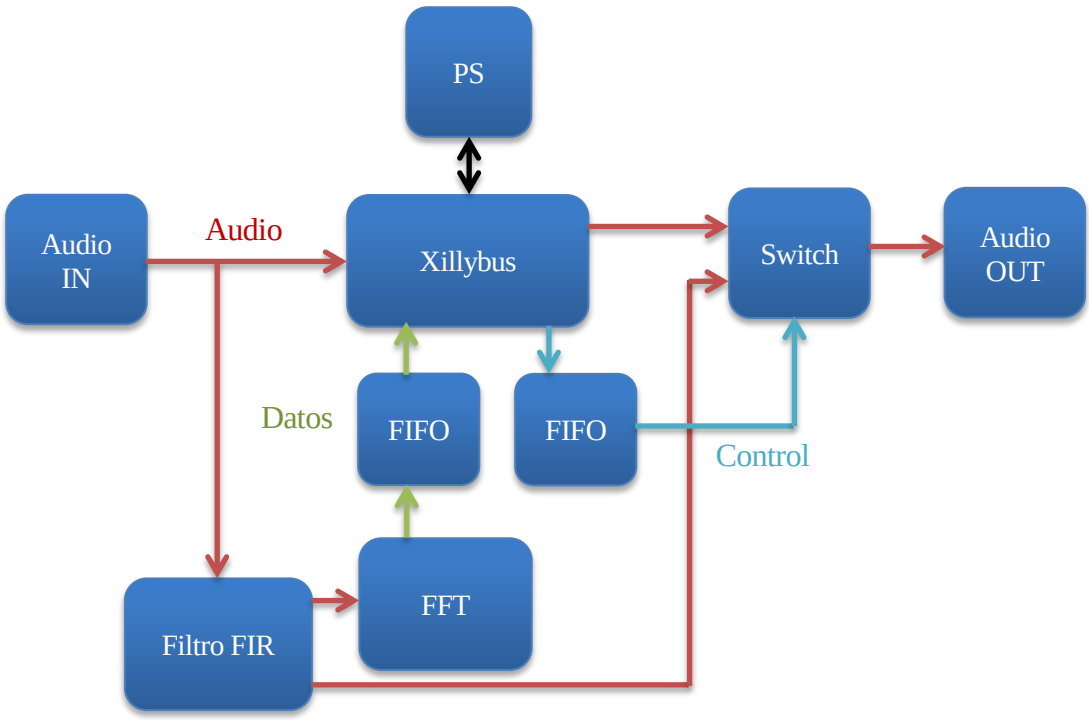

Figura 26 – Desarrollo hardware

1. **Bloque de audio**: Es el bloque encargado de recibir el audio del exterior y de enviarlo hacia afuera, viene dado por Xillybus y de él será necesario extraer el audio para enviarlo a otros bloques.
2. **Filtro FIR**: Recibe el audio y lo filtra, aplicando varios filtros diferentes que serán seleccionados a través de la señal de control recibida de Linux. El audio se envía tanto al bloque de FFT como al bloque switch que es el que elige qué audio se va a reproducir, si el tratado en hardware o el que produce Linux.
3. **FFT**: Recibe el audio original o filtrado por el bloque anterior y calcula su FFT. El resultado se enviará a Xillybus a través de su correspondiente FIFO.
4. **Switch**: Este bloque recibe el audio de Linux y el de hardware y conmuta entre ellos para dejar que se escuche uno u otro. Esto se controla a través de la señal de control que recibe de Xillybus. A su vez, en la señal de control también se elige el filtro que se quiere ejecutar en el bloque de Filtro FIR, así que este bloque le reenviará esa señal al filtro.
5. **FIFOs**: Intermediarias entre Xillybus y la lógica desarrollada.

El primer paso será analizar la configuración por defecto de la demo de Xillybus para ver el camino que sigue el audio y cómo poder extraerlo para usarlo. Una forma muy útil y rápida de hacerlo en vez de revisar el código fuente, es viendo el diseño que crea Vivado haciendo click en *Open Elaborated Design*.

Tras estudiar el esquemático que se genera, se llega a la conclusión de que el audio sigue el siguiente camino:

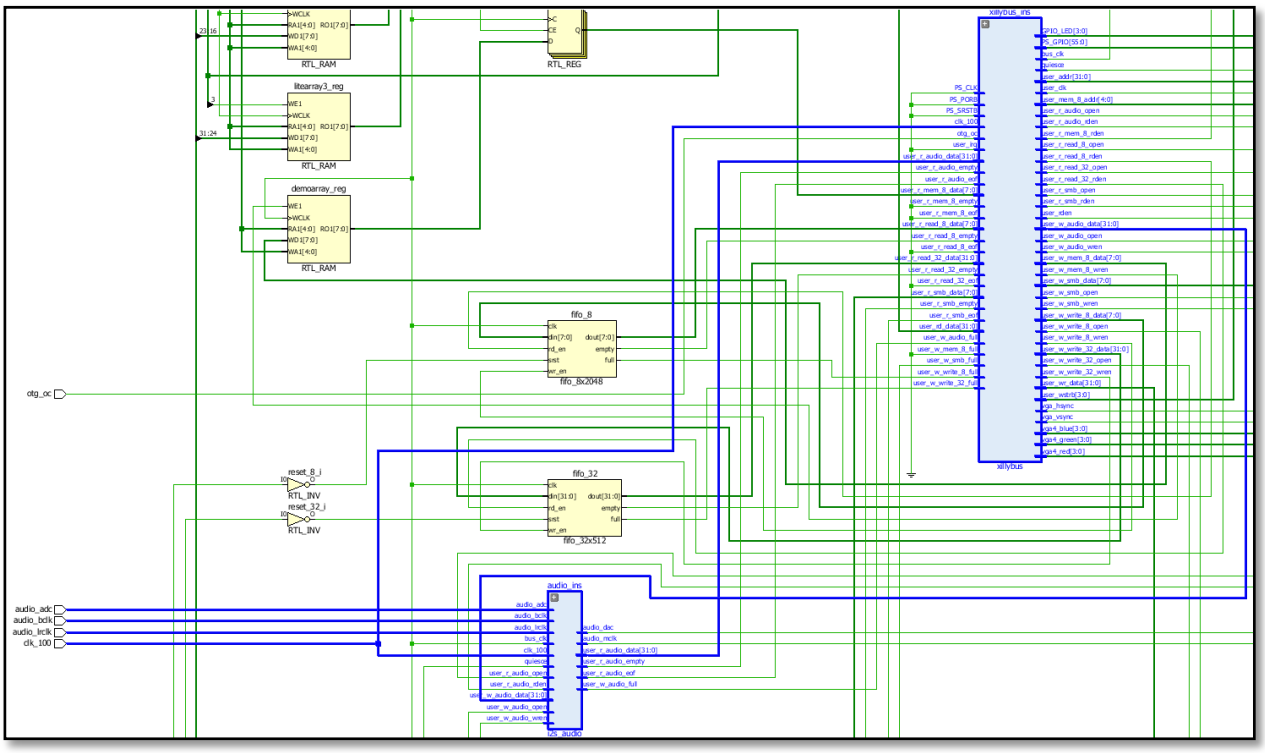

Figura 27 – Camino de audio por defecto

El audio llega al sistema a través de un bloque llamado i2s_audio y este lo envía al bloque xillybus. El proceso contrario pasa por el mismo camino pero en la otra dirección. Así que este bloque es el encargado tanto de la entrada como de la salida.

6.3.1 I2s_audio

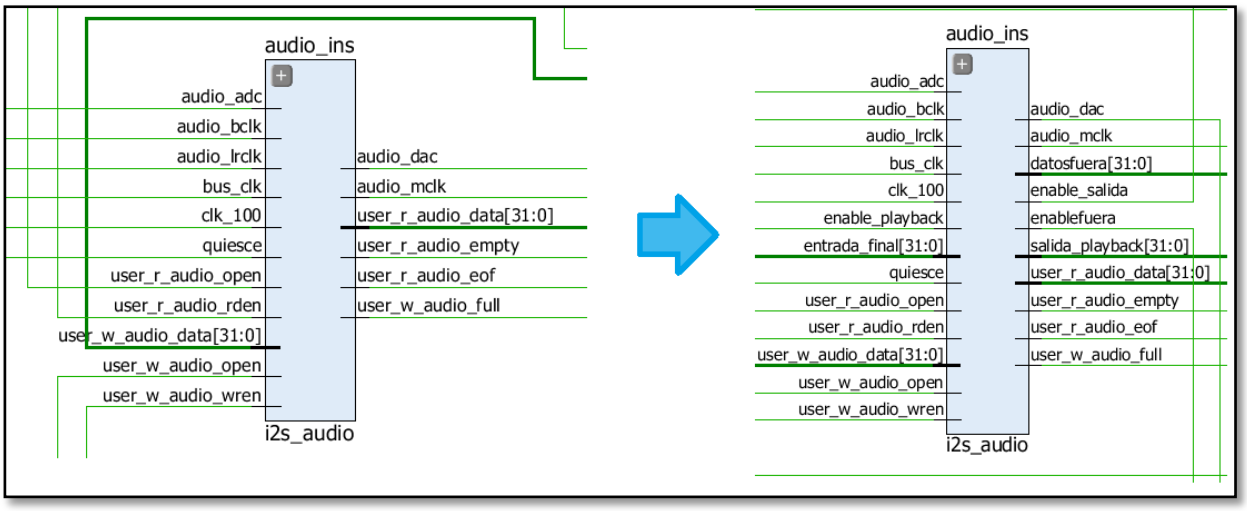

Figura 28 – Bloque i2s_audio

Este bloque recibe el audio que capta el códec de audio, el sonido viene codificado según el estándar I2S (Inter-IC Sound), que consta de tres líneas:

- Una línea de reloj de bit (audio_bclk).
- Una línea de reloj de palabra, o selección de palabra. Se usa para indicar si los datos que se están recibiendo pertenecen al canal izquierdo o al derecho (audio_lrclk).
- Una línea de datos multiplexados (audio_adc).

El funcionamiento completo del protocolo no es necesario conocerlo, pero si se quiere encontrar más información, está disponible en [35]. El propósito principal del bloque es recibir los bits por estas líneas serie, formar palabras de 32 bits y escribirlas en la FIFO que luego lee Xillybus. Cada palabra de 32 bits está formada por 16 bits del canal derecho y otros 16 del izquierdo.

Conocido esto, lo interesante es obtener el audio justo antes de que se escriba en la FIFO, que es cuando está formada la palabra de 32 bits. Para ello se crean dos variables nuevas de salida *datosfuera* y *enablefuera*, que como su nombre indica sacan el audio fuera del bloque.

En el otro sentido, por defecto el audio que proviene de Linux se escribe en la *playback_fifo* que está dentro de *i2s_audio*. La salida de esta FIFO es enviada a la salida de audio de la ZedBoard (*play_shreg*). Este camino hay que cambiarlo también para implementar el mecanismo de switch de la Figura 25: el audio de salida no va a ser siempre el que provenga de Linux sino que se podrá elegir entre este y el que pasa por el filtro.

El primer cambio es desconectar la salida de la *playback_fifo* del códec y llevarla fuera del bloque *i2s_audio*, donde posteriormente se conectará al bloque switch. Para ello se crea la señal *salida_playback* que se usará para este propósito. Después de este cambio la entrada del códec que recibe los datos que se quieren reproducir, está sin conexión. Aquí hay que conectar la salida del bloque switch, que es el que le enviará el audio que se quiere reproducir, esa es la función de la variable *entradafinal*, que internamente se conecta a la señal *play_shreg*.

Todos los cambios se entienden mejor viendo el diseño que genera Vivado y observando los cambios en el código que están en el fichero i2s_audio.v.

6.3.2 Filtro FIR

Para implementar el filtrado, se va a optar por un filtro FIR ya que Xilinx proporciona un IP core llamado FIR Compiler [36] que permite generar un filtro a partir de los parámetros de configuración dados. Se puede acceder a él desde Vivado en la opción IP Catalog → Digital Signal Processing → Filters → FIR Compiler.

Tras hacer click en él se inicia un asistente para configurar los parámetros del filtro. El primer parámetro que pide son los coeficientes del filtro, para obtenerlos primero hay que diseñar el filtro que se va a emplear. Para este trabajo se van a diseñar tres filtros, un paso bajo, un paso banda y un paso alto. Se cargarán los tres en el IP core y se seleccionará uno u otro desde Linux mediante la señal de control.

Para diseñarlos se ha hecho uso de la herramienta fdatool de Matlab. Cualquier filtro podría ser implementado teniendo en cuenta que el sonido audible está comprendido entre 20Hz y 20kHz. El filtro de paso bajo que se usará será de 0 a 3,5 kHz, el paso banda de 1 a 6 kHz y el paso alto de 5 kHz en adelante. Todos los filtros se han diseñado intentando llegar a un compromiso entre frecuencia de corte abrupta (que requiere un mayor número de coeficientes) y un orden del filtro lo más reducido posible para que no genere demasiados coeficientes, ya que se traduce en un mayor tiempo de procesamiento en la FPGA. Los detalles de los tres filtros se pueden consultar en las siguientes imágenes:

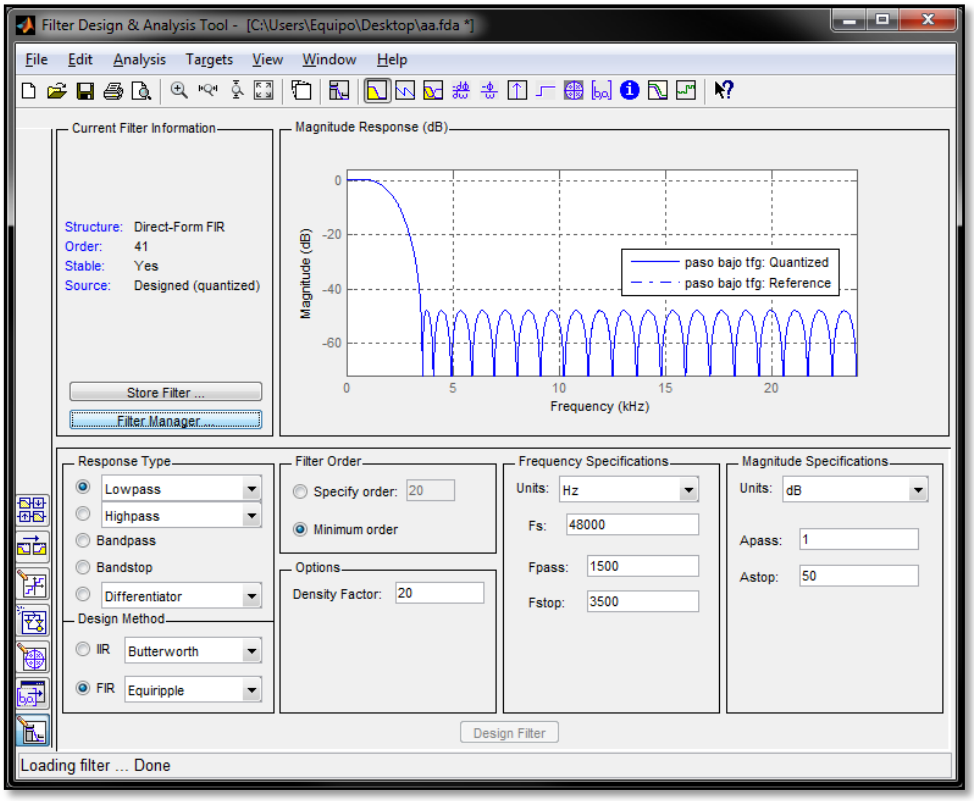

Figura 29 – Filtro paso bajo

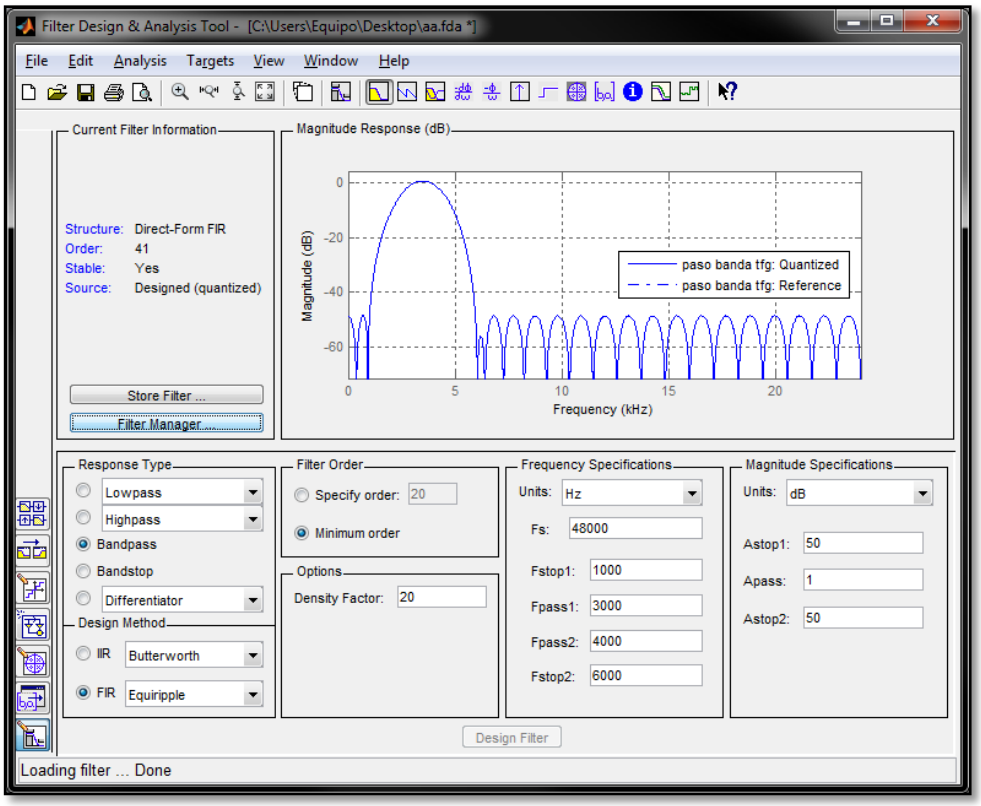

Figura 30 – Filtro paso banda

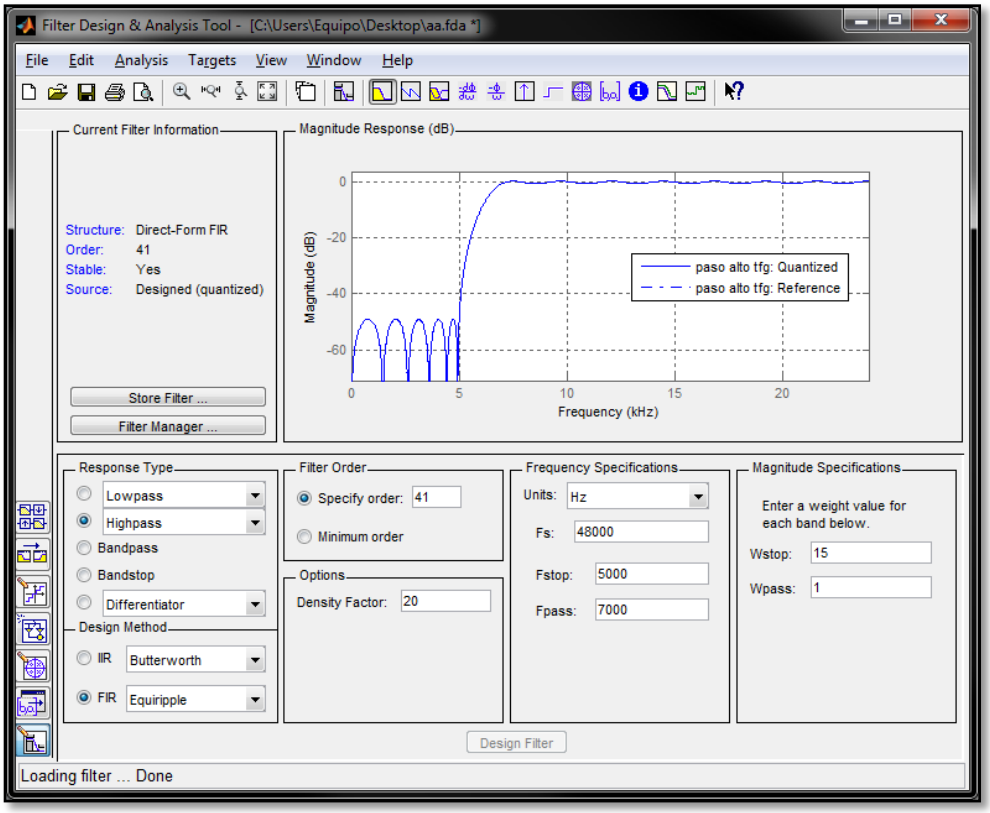

Figura 31 – Filtro paso alto

Tras diseñar cada uno de los filtros, se hace click en la opción *Targets* del menú superior y dentro de ella en *XILINX Coefficient (.COE) File*, así se generarán tres ficheros .coe con los coeficientes de cada filtro.

El siguiente paso es unir los coeficientes de los tres ficheros .coe en uno solo, esto se hace eligiendo uno de los .coe como base y copiando los coeficientes del resto de ficheros justo después del último coeficiente del fichero base. Es decir, si el primer coe acaba con estos coeficientes:

```
.
.
9f37,
a584,
d59b,
1494,
429b,
4bc4,
30f8;
```

Se cambia el último ';' por una ',' y se añaden a continuación los coeficientes de otro fichero:

```
.
.
9f37,
a584,
d59b,
1494,
429b,
4bc4,
30f8,
000a, <- Empiezan los nuevos coeficientes
1ac2,
45ee,
.
.
```

Con esto se consigue tener los tres filtros en un único fichero .coe, que será el usado en Vivado. Volvemos ahora al asistente del FIR compiler. En el campo *Select Source* se selecciona COE File y más abajo se indica la ruta al fichero .coe. En *Number of Coefficient Sets* se escribe un 3 ya que tenemos tres filtros. El resto de opciones de esa pestaña se dejan por defecto.

En la pestaña de *Channel Specification* se seleccionan estas opciones: *Chanel Sequence: Basic*, *Number of Chanels:* 1, *Number of Paths:* 2 (uno para el audio de canal izquierdo y otro para el canal derecho), *Select Format: Frequency Specification*, *Input Sampling Frequency:* 0.048 (48 kHz, que es la velocidad de muestreo del códec), *Clock Frequency:* 100 (100 MHz es el reloj que genera Xillybus).

En la pestaña *Implementation*: *Coefficient Type: Signed*, *Coeficient Width:* 16, *Input Data Type: Signed*, *Input Data Width:* 16 (cada canal es de 16 bits), *Input Data Fractional Bits:* 0, *Output Rounding Mode: Symmetric Rounding to Infinity* (opción con mejor relación precisión-recursos utilizados) y *Output Width:* 16.

El resto de pestañas se pueden dejar con la configuración por defecto, salvo la inclusión de la señal *ARESETn*, ya que siempre es bueno contar con un reset en cada bloque. Por último se hace click en *OK* para generar el core que creará un bloque con las siguientes señales:

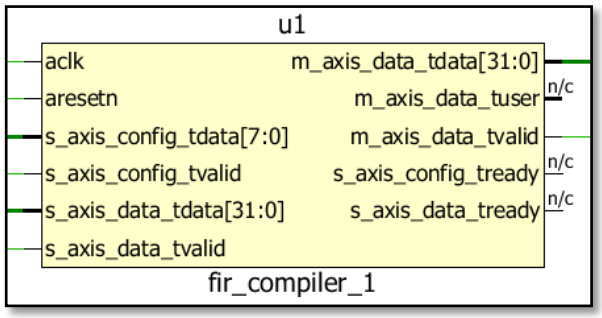

Figura 32 – Bloque del filtro

- **Aclk:** Reloj que se debe conectar a la señal *bus_clk*.
- **Aresetn:** Señal de reset, se conectará a la señal de control para resetear el sistema desde Linux.
- **S_axis_config_tdata[7:0]:** Señal de configuración, se usarán los bits 1:0 para seleccionar qué filtro de los 3 usar. Esta señal se conectara al siguiente bloque, que es el encargado de controlar al filtro.
- **S_axis_config_tvalid:** Señal de enable para la anterior.
- **S_axis_data_tdata[31:0]:** Entrada del audio, ancho de 32 bits de los que 16 son para cada canal. Se conecta a la señal *datosfuera* del bloque i2s audio.
- **S_axis_data_tvalid:** Enable de la señal anterior. Se conecta a la señal *enablefuera* del bloque i2s audio.
- **M_axis_data_tdata[31:0]:** Salida que contiene el audio filtrado.
- **M_axis_data_tvalid:** Enable de la señal anterior.

6.3.3 Switch filtro

Este es un bloque que no aparece en la figura 26, porque se podría incluir dentro de Filtro FIR. Su función es la de recibir la salida del filtro y el audio original y conmutar entre estos dos para cuando no se quiera utilizar el audio filtrado, sino el original. Su segunda función es la de controlar qué filtro aplicar de los tres que contiene el bloque anterior. Esto lo hace recibiendo la señal de control desde el bloque *Switch* y enviándosela al filtro a través de su entrada *s_axis_config_tdata*. Este es el bloque y sus señales:

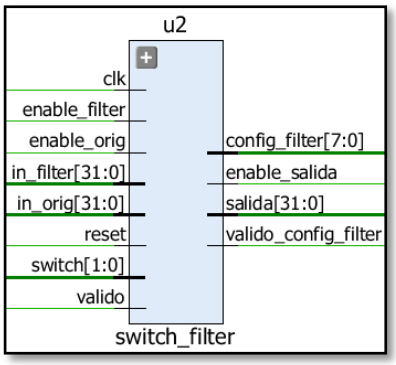

Figura 33 – Bloque switch filtro

- **Clk:** Reloj que se debe conectar a la señal *bus_clk*.
- **Reset:** Señal de reset, se conectará a la señal de control para resetear el sistema desde Linux.
- **In_filter[31:0]:** Audio procedente del filtro. Conectado a su salida *m_axis_data_tdata.*
- **Enable_filter:** Enable de la señal anterior. Conectado a *m_axis_data_tvalid* del filtro.
- **In_orig[31:0]:** Audio original sin filtrar. Se conecta a la señal *datosfuera* del bloque i2s audio.
- **Enable_orig:** Enable de la señal anterior. Se conecta a la señal *enablefuera* del bloque i2s audio.
- **Switch[1:0]:** Señal que indica cuál de los tres filtros elegir. Procede del bloque *Switch*.
- **Valido:** Enable de la señal anterior.
- **Salida[31:0]:** Salida del bloque, conmutará entre el audio original y el filtrado según se indique desde Linux a través de la señal *switch[1:0]*.
- **Enable_salida**: Corresponde al enable de la señal anterior.
- **Config_filter[7:0]:** Salida que indica al filtro qué filtro aplicar. Se conecta a la señal de este llamada *s_axis_config_tdata.*
- **Valido_config_filter:** Enable de la señal anterior. Se conecta a la señal *s_axis_config_tvalid* del filtro.

6.3.4 FFT

El siguiente bloque será el encargado de efectuar la FFT de la señal de audio. Al igual que el filtro FIR, Xilinx proporciona un IP core configurable para calcular la FFT [37], así que se usará este. Se encuentra en *IP Catalog → Digital Signal Processing → Transforms → FFTs → Fast Fourier Transform.*

Se desplegará un asistente de configuración, en el que se seleccionarán las opciones más apropiadas. En la primera pestaña *Configuration* se escogen las siguientes: *Number of Channels:* 1, *Transform Length:* 1024 (Con este número de muestras es suficiente para la precisión que se requiere) y *Architecture Choice: Radix-2, Burst I/O.* Se podría elegir cualquier otra arquitectura, pues no hay restricciones de tamaño o velocidad. El muestreo se produce a 48 kHz que comparado con el reloj de 100 MHz es muy lento, por lo que se ha escogido la arquitectura con menor rendimiento y menor consumo de recursos.

En la pestaña de *Implementation* se escogen las siguientes opciones: *Data Format: Fixed Point, Scaling*

Options: Block Floating Point[1], *Rounding Modes: Convergent Rounding, Input Data Width:* 16 (a la FFT solo le pasaremos los 16 bits de un canal, el otro no es necesario), *Phase Factor Width:* 16. Se marcan también las opciones de *ARESETn*, *XK_INDEX* (útil para depurar, acompaña a cada muestra en la salida con su número de índice) y por último *Throttle Scheme: Non Real Time* que tiene unas restricciones menores para la entrada de datos en el bloque, aunque *Real Time* también podría haber sido marcado.

La pestaña restante puede dejarse con la configuración por defecto. Tras concluir se generará un bloque con las siguientes señales:

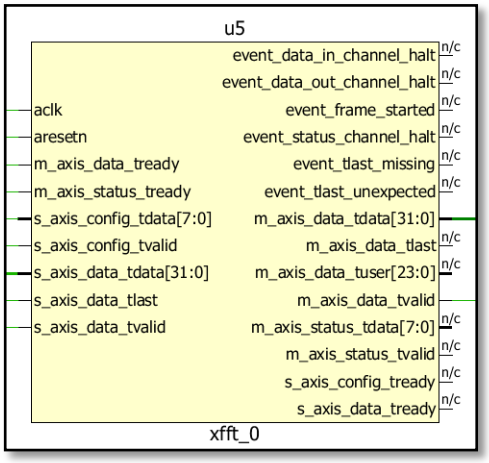

Figura 34 – Bloque FFT

- **Aclk:** Reloj que se debe conectar a la señal *bus_clk*.
- **Aresetn:** Señal de reset, se conectará a la señal de xillybus *user_r_fft_open* para resetear la FFT cuando no se está usando por el host.
- **S_axis_data_tdata[31:0]:** Entrada a la FFT, los 16 bits más altos corresponden a la parte imaginaria de la señal y los 16 más bajos a la parte real. El sonido solo tiene componente real, así que se conectarán únicamente los bits 15:0 a la salida del bloque anterior y los bits restantes a tierra.
- **S_axis_data_tvalid:** Enable para la señal anterior. Se conecta a *enable_salida* del bloque *switch_filter*.
- **S_axis_data_tlast:** Indica cuando llega la última muestra de la trama, se dejará a tierra pues no se usa en el cálculo sino para generar otros eventos en la salida.
- **S_axis_config_tdata[7:0]:** solo se usa el bit 0 para indicar si se quiere una transformada directa o inversa. Se conecta siempre a alimentación para indicar transformada directa.
- **S_axis_config_tvalid:** Enable para la señal anterior, también se deja conectado a '1'.
- **M_axis_data_tready:** Señal activada por la *fifo_fft*, cuando se activa indica que puede admitir más datos. Se conecta a la señal *full* negada de la *fifo_fft*.
- **M_axis_status_tready:** Cumple la misma función que la señal anterior pero para el canal de estado en vez de el de datos. Este canal no se usará así que se deja conectada a '1'.
- **M_axis_data_tdata[31:0]:** Salida de la FFT, los 16 bits más altos corresponden a la parte imaginaria de la señal y los 16 más bajos a la parte real. Se conecta a la entrada de la *fifo_fft*.
- **M_axis_data_tvalid:** Enable para la señal anterior. Se conecta al *wr_en* de la *fifo_fft*.

El resto de señales que no se han mencionado se pueden dejar sin conectar. Se usan para mostrar información del estado de la FFT.

[1] Con el *block floating point*, cada etapa aplica suficiente escalado automáticamente para mantener los números en rango y el escalado puede ser conocido por un exponente en la salida.

6.3.5 Switch

Este es uno de los bloques principales del sistema. Es el que recibe la señal de control de Linux y la distribuye a los demás bloques además de actuar sobre sí mismo. Esta señal es un vector de 8 bits, donde cada bit tiene la siguiente función:

- **Bit 0:** si vale 0, este bloque deja pasar el audio de Linux y si vale 1 deja pasar el audio que se ha procesado en hardware.
- **Bits [2:1]:** Estos dos bits se envían al bloque *switch_filter* que configura el filtro en uno de estos 4 modos:
  - **00:** Audio sin filtrar.
  - **01:** Filtro paso bajo.
  - **10:** Filtro paso banda.
  - **11:** Filtro paso alto.
- **Bit 3:** Se activa para hacer reset al resto de bloques, incluido este, lo que reinicia sus máquinas de estado o los reinicializa si son IP cores.
- **Bits [7:4]:** Sin uso.

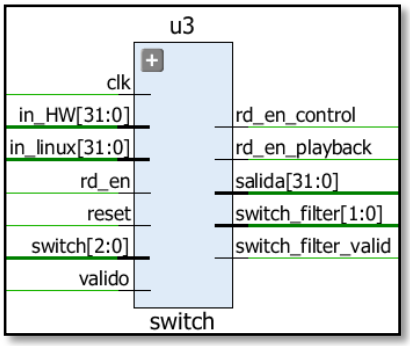

Figura 35 – Bloque switch

Este es el bloque con sus entradas y salidas:

- **Clk:** Reloj que se debe conectar a la señal *bus_clk*.
- **Reset:** Señal de reset, se conectará a la señal de control para resetear el sistema desde Linux.
- **In_HW[31:0]:** Audio procedente del hardware. Se conecta a la salida de *switch_filter*.
- **In_linux[31:0]:** Audio de Linux. Se conecta a la *salida_playback* del bloque *i2s_audio*.
- **Switch[2:0]:** Recibe la señal de control de Xillybus. Se conecta a la salida de la *FIFO_control*.
- **Valido:** Enable de la señal anterior. Se conecta a la señal *valid* de la *FIFO_control*.
- **Rd_en:** Señal que manda el códec cuando está listo para reproducir más datos, se usa internamente para activar la señal de salida *rd_en_playback* que hace que salga el audio de Linux guardado en la *fifo_playback*. Se conecta a la señal *enable_salida* del bloque *i2s_audio*.
- **Salida[31:0]:** Audio de salida que conmuta entre los dos de entrada. Se conecta a la señal *entrada_final* del bloque *i2s_audio.*
- **Switch_filter[1:0]:** Corresponde a los bits [2:1] de control y se envían al bloque *switch_filter*.
- **Switch_filter_valid:** Enable de la señal anterior.
- **Rd_en_playback**: Señal de *rd_en* que se envía a la *fifo_playback* cuando se quiere leer audio de Linux. Se conecta a la señal *enable_playback* del bloque *i2s_audio.*
- **Rd_en_control**: señal de *rd_en* que se envía a la *fifo_control* para leer la señal de control de Linux.

6.3.6 FIFO_control

FIFO usada para la comunicación de la señal de control con Xillybus. Tendrá un ancho de 8 bits y una profundidad de 32 palabras, pero este último parámetro no implica limitaciones puesto que se enviará un dato cada mucho tiempo. Sirve como ejemplo para mostrar cómo se conecta Xillybus a una FIFO, como se explicó en la figura 13:

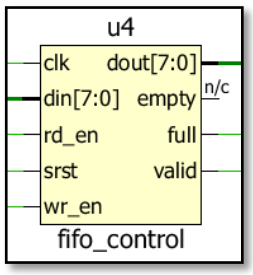

Figura 36 – FIFO_control

- **Clk:** Reloj que se debe conectar a la señal *bus_clk*.
- **Srst:** Señal de reset, se conecta la señal negada *user_w_switch_open* de Xillybus.
- **Din[7:0]:** Entada por la que se recibe la señal de control, se conecta a la señal *user_w_switch_data* de Xillybus.
- **Wr_en:** Enable de la señal anterior, se conecta a la señal *user_w_switch_wren* de Xillybus.
- **Rd_en:** Read_enable para leer la señal de control desde el bloque *switch*, se conecta a la señal *rd_en_control* de este bloque.
- **Dout[7:0]:** Salida de la FIFO, los bits [2:0] se conectan a la señal *switch* del bloque *switch*. El bit 3 se lleva a las entradas de reset de otros bloques.
- **Valid:** Enable de la señal anterior.
- **Full:** Indica cuando se llena la FIFO y no admite más datos, se conecta a la señal *user_w_switch_full* de Xillybus.

6.3.7 FIFO_FFT

La otra FIFO es la encargada de la comunicación de la FFT. Debe ser de ancho 32 bits para que quepa una muestra entera y de profundidad se ha escogido 2048 palabras, suficiente para almacenar dos transformadas de 1024 muestras. Su conexión es un ejemplo de lo explicado en la figura 14:

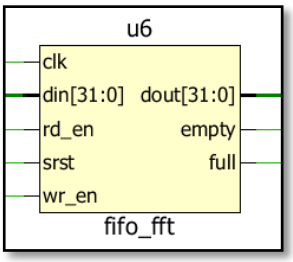

Figura 37 – FIFO_FFT

- **Clk:** Reloj que se debe conectar a la señal *bus_clk*.
- **Srst:** Señal de reset, se conecta la señal negada *user_r_fft_open* de Xillybus.
- **Din[31:0]:** Entada por la que se recibe el resultado de la FFT, se conecta a la señal *m_axis_data_tdata* de la FFT.
- **Wr_en:** Enable de la señal anterior, se conecta a la señal *m_axis_data_tvalid* de la FFT.

- **Rd_en:** Read_enable para leer un dato desde Xillybus, se conecta a la señal *user_r_fft_rden* de este.
- **Dout[31:0]:** Salida de la FIFO, se conecta a la señal *user_r_fft_data* de Xillybus.
- **Empty:** Señal que indica que la FIFO está vacía y no hay más datos para leer. Se conecta a la señal *user_r_fft_empty* de Xillybus
- **Full:** Indica cuando se llena la FIFO y no admite más datos, se conecta a la señal *m_axis_data_tready* negada de la FFT. De este modo si la FIFO se llena, la FFT no insertará más muestras en ella.

6.3.8 GPIOs

Con Xillybus es muy fácil controlar los GPIOs conectados a la FPGA desde Linux porque en el fichero de restricciones *vivado-essentials/xillydemo.xdc*, están por defecto todos los pines conectados al PS. En los comentarios está descrito a qué pin corresponde cada GPIO.

Para usarlos desde Linux solo es necesario conocer el número de GPIO de cada uno. Por ejemplo, si se quieren usar los cinco pulsadores que aparecen a continuación hay que recordar los números 19-23:

```
# 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]"]
```

Estos GPIOs aparecerán en Linux identificados con sus números correspondientes. El acceso a ellos se mostrará en el próximo capítulo. Si por el contrario se quisiesen usar en la FPGA y no en Linux, habría que hacer unas modificaciones al fichero que vienen detalladas en el capítulo 5.4 de [27].

6.4 Desarrollo software

La intención en la parte software es recibir los datos de la FFT, almacenarlos y ordenarlos para representar el espectrograma. Asimismo, este espectrograma debe estar disponible para verlo desde otro dispositivo, por lo que se montará en una aplicación web. No solo hay que recibir datos, también existe la comunicación en dirección host-FPGA, así que la aplicación también deberá ofrecer la posibilidad de enviar la señal de control a la FPGA.

El desarrollo debe contar entonces con un servidor web, en el que se van a programar con PHP las funciones de escritura y lectura en ficheros para la comunicación con la FPGA. Estos datos tienen que ser enviados a nuestra aplicación web que se está ejecutando en otro dispositivo. Hay dos tipos de datos: los que vienen de la FPGA, que tienen requisitos de tiempo real, y los que se envían a ella, que serán esporádicos y cuando se produzca algún evento.

Para satisfacer estas necesidades, se van a usar dos tipos de tecnologías de comunicación entre cliente y servidor: websockets para las comunicaciones en tiempo real y AJAX para las de control.

En el cliente se recibirán o enviarán estos datos y se representarán mediante tecnologías web como HTML5, CSS3 y Javascript. En este capítulo se describirá todo este desarrollo.

6.4.1 Instalación servidor HTTP Apache

A la hora de instalar un programa en la ZedBoard es cuando se aprecia una de las ventajas de Xillinux: estar basado en Ubuntu. Esto hace la instalación prácticamente trivial ya que el servidor Apache se encuentra en los repositorios y se puede instalar con un simple comando:

```
apt-get install apache2
```

Después se ejecutan dos configuraciones básicas:

- Al final del fichero */etc/apache2/apache2.conf* añadir la siguiente línea para evitar los warnings al arrancar:

  ```
  ServerName localhost
  ```

- Editar el fichero */etc/apache2/conf.d/charset* y descomentar la siguiente línea para evitar problemas con las tildes:

  ```
  AddDefaultCharset UTF-8
  ```

Y ya estaría listo para usar, para comprobar que funciona se abre el navegador de Xillinux y se ingresa en la dirección: http://localhost o http://127.0.0.1, en la que debería aparecer la página por defecto de Apache.

La ruta que usa el servidor para buscar los ficheros web es */var/www/*, aquí se deberán colocar todos los ficheros programados.

Por último hay que instalar php, igual de rápido con el comando: [1]

```
apt-get install php5
```

Con esto se tiene el soporte necesario para empezar el desarrollo, comenzando inicialmente por el desarrollo en el lado del servidor.

6.4.2 Desarrollo del servidor

La comunicación con la FPGA es muy simple, como ya se ha explicado consiste en trabajar con ficheros de texto. El punto que es más laborioso es la comunicación con el cliente mediante websocket y AJAX.

AJAX es una tecnología basada en Javascript que permite la actualización de páginas web de forma asíncrona intercambiando pequeñas cantidades de datos con el servidor en segundo plano. Esto permite cambiar una parte de la web sin necesidad de recargar la página entera.

Por otra parte, los websockets ofrecen una comunicación en tiempo real entre cliente y servidor, es una conexión que se mantiene siempre abierta hasta que se cierra de forma intencionada. Esto permite al servidor enviar datos a todos los clientes conectados a ese socket sin tener que estar procesando peticiones HTTP continuamente. Esto reduce mucho la latencia al eliminar toda la capa del protocolo HTTP que es demasiado pesada para aplicaciones de tiempo real.

Estas dos tecnologías son muy amplias y tienen mucho contenido detrás, pero como no son el objetivo de este trabajo, simplemente se ha hecho una descripción de ellas para entender en qué consisten y para qué se van a usar.

6.4.2.1 Lectura de la FFT

El paso más complicado es implementar un websocket en PHP, por ello se va a hacer uso de un script [38] que ya lo hace para centrarnos únicamente en la comunicación con la FPGA. Es un fichero que implementa un ejemplo de chat a través de websocket, nos quedaremos con todo menos con la funcionalidad del chat que será reemplazada por la lectura de la FFT.

La parte interesante del código, en la que se lee la FFT, aparece a continuación:

[1] En el momento que se instaló php su versión más reciente era la 5, recientemente se liberó la versión 7 con muchas mejoras en rendimiento

```
<?php
$fichero = fopen("/dev/xillybus_fft", "rb");
fread($fichero, 2048);
$grafica = "";

for ($i=511; $i >= 0; $i--) {
        $var = fread($fichero, 4);
        $real = two_complement(hexdec(bin2hex($var[1] . $var[0])));
        $img = two_complement(hexdec(bin2hex($var[3] . $var[2])));

        $result = pow($real,2) + pow($img,2);
        $grafica .= $result . " ";
}

$sended_text = mask($grafica);
send_message($sended_text); //send data
?>
```

Primero se abre el fichero en modo lectura con *fopen()*, luego se hace una lectura basura de 2048 bytes. Esta lectura se debe a que se ha implementado una FFT de 1024 puntos, como se explicó en el apartado 5.4.1.1. Las muestras tienen una simetría, por lo que en realidad hay 512 muestras útiles y luego otras 512 simétricas. Es decir, se pueden desechar las primeras 512 muestras y como cada muestra tiene 4 bytes, hacen un total de 2048 bytes que no sirven.

Una vez leídas estas muestras simétricas, comienza la lectura de las muestras que interesan, que se recogen en un bucle de 512 iteraciones. En cada iteración se leen 4 bytes: dos bytes de parte real y 2 de parte imaginaria.

Es importante conocer el orden en el que se leen esos 4 bytes, la FPGA envía los datos de 32 bits en paralelo, sin embargo la función *fread()* lee de byte en byte desde el menos significativo al más significativo, de ahí la forma en la que se forman las variables *$real* e *$img* a continuación.

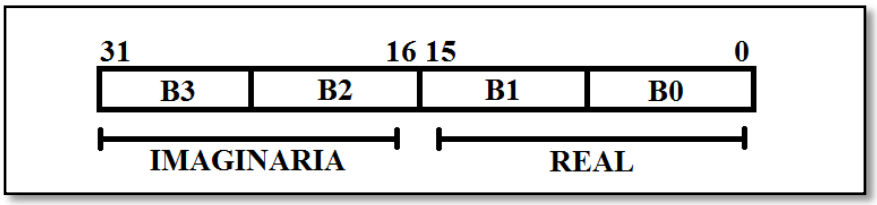

Figura 38 – Orden de lectura de los bytes

Por ejemplo, en la variable *$real* se concatenan los bytes 1 y 0 para formar la parte real completa y se convierte de binario (en complemento a dos) a decimal. Lo mismo se hace con la parte imaginaria y los bytes 2 y 3.

Ahora como se explicó en el capítulo 5.4.1, para formar el espectrograma se necesita el módulo al cuadrado de la transformada, esto se calcula en la siguiente línea elevando al cuadrado las partes reales e imaginarias y sumándolas. El resultado se almacena en la variable gráfica y se lee la siguiente muestra. Este proceso se repite 512 veces para recoger el conjunto de muestras en la variable gráfica, que finalmente es codificada como exige el protocolo websocket con la función *mask()* y se envía a la página web para ser representada.

Toda esta porción de código explicada está incluida en un bucle que se ejecuta constantemente recogiendo las muestras de 512 en 512 y enviándolas a la página web para su visualización.

6.4.2.2 Envío de la señal de control

El envío es mucho más sencillo porque se envía un byte de datos cada cierto tiempo. Por ello se hará a través de AJAX, cuando el cliente envíe algún dato nuevo se ejecutará el siguiente código PHP de forma asíncrona y enviará los datos recibidos por el cliente a la FPGA:

```
<?php
        $received_text = $_GET["filtro"];
        $fichero = fopen("/dev/xillybus_switch", "wb");
        fwrite($fichero, $received_text);
        fclose($fichero);
?>
```

Al fichero php se le envía desde la página web la variable filtro que contiene un número entero que representa cada uno de los filtros de la FPGA. Por ejemplo, si el cliente selecciona el filtro paso banda, se le enviará al fichero php el número 5, que en binario es 00000101. Si recordamos en el apartado 6.3.5 el significado de cada uno de estos 8 bits, se ve que se está seleccionando el audio hardware filtrado con paso banda.

Este número se almacena en la variable *$received_text*. Luego se procede a abrir el fichero en modo escritura, se escribe en él la variable y se cierra. Muy simple y rápido.

6.4.2.3 GPIOs

La lectura y escritura de GPIOs se hace como en cualquier otro Linux. Están accesibles en la ruta */sys/class/gpio/*, en la que aparecerán unas carpetas llamadas gpioXX donde XX es un número. Básicamente cuando se quiere trabajar con GPIOs primero hay que reservarlos, establecer la dirección en entrada o salida y empezar a usarlos. Una vez se ha terminado de usarlos hay que liberarlos para permitir que sean utilizados por cualquier otro proceso.

Para este trabajo se van a usar estos conjuntos de GPIOs: 8 switches, 4 LEDs de usuario y 5 pulsadores. Para identificar qué número corresponde a cada GPIO, se siguen los pasos descritos en el apartado 6.3.8 y se tienen estos:

- LEDs: 7-10.
- Switches: 11-18.
- Pulsadores: 19-23.

Existe un offset entre el número de GPIO indicado en el fichero .xdc y el número asignado por Linux y es 54 en Xillinux. Así que los números de GPIO serían:

- LEDs: 61-64.
- Switches: 65-72.
- Pulsadores: 73-77.

El primer paso es reservarlos o exportarlos, esto se hace con el siguiente comando. Donde XX debe ir desde 61 hasta 77:

```
echo XX > /sys/class/gpio/export
```

El siguiente paso es indicar si son de entrada o salida. Los switches y pulsadores serán de entrada y los LEDs son de salida. Para el caso de los de entrada:

```
echo in > /sys/class/gpio/gpioXX/direction
```

Y para los de salida dos comandos, uno para indicar la dirección y otro para dar permiso de escritura al servidor Apache, para que pueda escribir los datos que le lleguen desde el cliente:

```
echo out > /sys/class/gpio/gpioYY/direction
chmod o+w /sys/class/gpio/gpioYY/value
```

Con esto ya están todos los GPIOs preparados para ser usados, ahora desde un fichero PHP se escribirá o leerá de ellos según el caso. Para leer el estado de los de entrada y enviarlos al cliente se ejecuta el siguiente código:

```
<?php
for ($gpio = 65, $salida = ""; $gpio <= 77; $gpio++) {
        $fichero = fopen("/sys/class/gpio/gpio" . $gpio . "/value", "r");
        $salida .= fgets($fichero);
        $salida .= " ";
        fclose($fichero);
}

$sended_text = mask($salida);
send_message($sended_text); //send data
?>
```

Mediante un bucle for se recorren los GPIOs y para cada iteración se abre su respectivo fichero con *fopen()*, se lee su estado que será 1 o 0, se concatena en la variable *$salida* y se cierra el fichero con *fclose()*. Cuando se han recorrido todos, se codifica la salida con la función *mask()* como exige el protocolo websocket y se envía a la página web para ser visualizado. Este trozo de código está dentro de un bucle que se ejecuta constantemente para tener resultados del estado de los GPIOs en tiempo real.

Para el caso de los GPIOs de salida, el fichero PHP recibirá la orden del cliente de ponerlos a 1 o 0 y éste lo escribirá de la misma forma en cada GPIO:

```
<?php
$received_text = $_GET["leds"];
$valores = explode(",", substr($received_text, 0));

for ($gpio = 61; $gpio <= 64; $gpio++) {
        $fichero = fopen("/sys/class/gpio/gpio" . $gpio . "/value", "w");
        fwrite($fichero, $valores[64-$gpio]);
        fclose($fichero);
}
?>
```

Primero obtiene la variable *leds* que le ha enviado el cliente que tendrá como contenido una combinación de 1 y 0 separados por comas (1,0,1,0) que indica el estado en el que se quiere poner cada GPIO. En la siguiente línea se separan y obtienen estos valores. Y finalmente se escriben en cada fichero con las mismas funciones de otras veces.

Si se quieren liberar los GPIOs se hace igual que al reservarlos pero con la palabra *unexport*:

```
echo XX > /sys/class/gpio/unexport
```

6.4.3 Desarrollo del cliente

6.4.3.1 Espectrograma

El cliente ejecutara una página web que se comunicará con el servidor para el intercambio de datos. En la página se seleccionarán diversas opciones que se enviarán al servidor para formar la señal de control y también recibirá de él los datos del espectrograma para representarlos. La aplicación web está desarrollada con tecnologías web como HTML5, CSS3 y Javascript y usa el framework Bootstrap para que la página sea adaptable a cualquier dispositivo (PC, móvil, tablet, etc.).

Para la representación del espectrograma se ha hecho uso de una librería de espectrograma [39] que facilita el proceso evitando que haya que hacerlo desde cero, simplemente modificando el ejemplo para usarlo conjuntamente con websockets. Y por último, se ha utilizado una librería para gráficas interactivas [40].

De la unión de todas estas tecnologías se ha desarrollado una página web con el aspecto que aparece en la figura 39.

Al principio ofrece la posibilidad de elegir el audio de Linux y el de hardware. Más abajo aparecen los 3 filtros implementados en la FPGA que se pueden elegir haciendo click en alguno de ellos o no usar ninguno haciendo click en “sin filtro”. Más abajo aún, aparece el espectrograma que se está dibujando constantemente con los datos que está recibiendo.

No se va a explicar en detalle toda la página porque no forma parte del codiseño HW/SW en sí, lo que se sale del objetivo del trabajo. Pero se explicara brevemente su funcionamiento a continuación para tener conciencia de lo que ocurre por detrás.

Respecto a la señal de control, cada vez que se selecciona uno de los filtros, audio hardware o audio Linux, se lanza un evento que desencadena el envío de un número entero por AJAX al servidor PHP. Este número ya se explicó a lo que correspondía en binario y cómo se enviaba a la FPGA.

Para los datos de la FFT, hay que recordar que el servidor leía de 512 en 512 muestras y las enviaba al cliente a través de un websocket. Aquí en el cliente se recibe ese conjunto de 512 muestras, que corresponde a una columna del espectrograma. Esas muestras son números, que al recibirlos se escalan de 0 a 1 y se les asigna un color según su valor. El color de cada muestra es lo que se dibuja en pantalla, formando una de las columnas. Este proceso repetido por cada conjunto de 512 muestras hace que se vaya formando el espectrograma columna a columna en tiempo real. Los detalles del proceso se encuentran en el código, en el que se puede consultar todos los pasos.

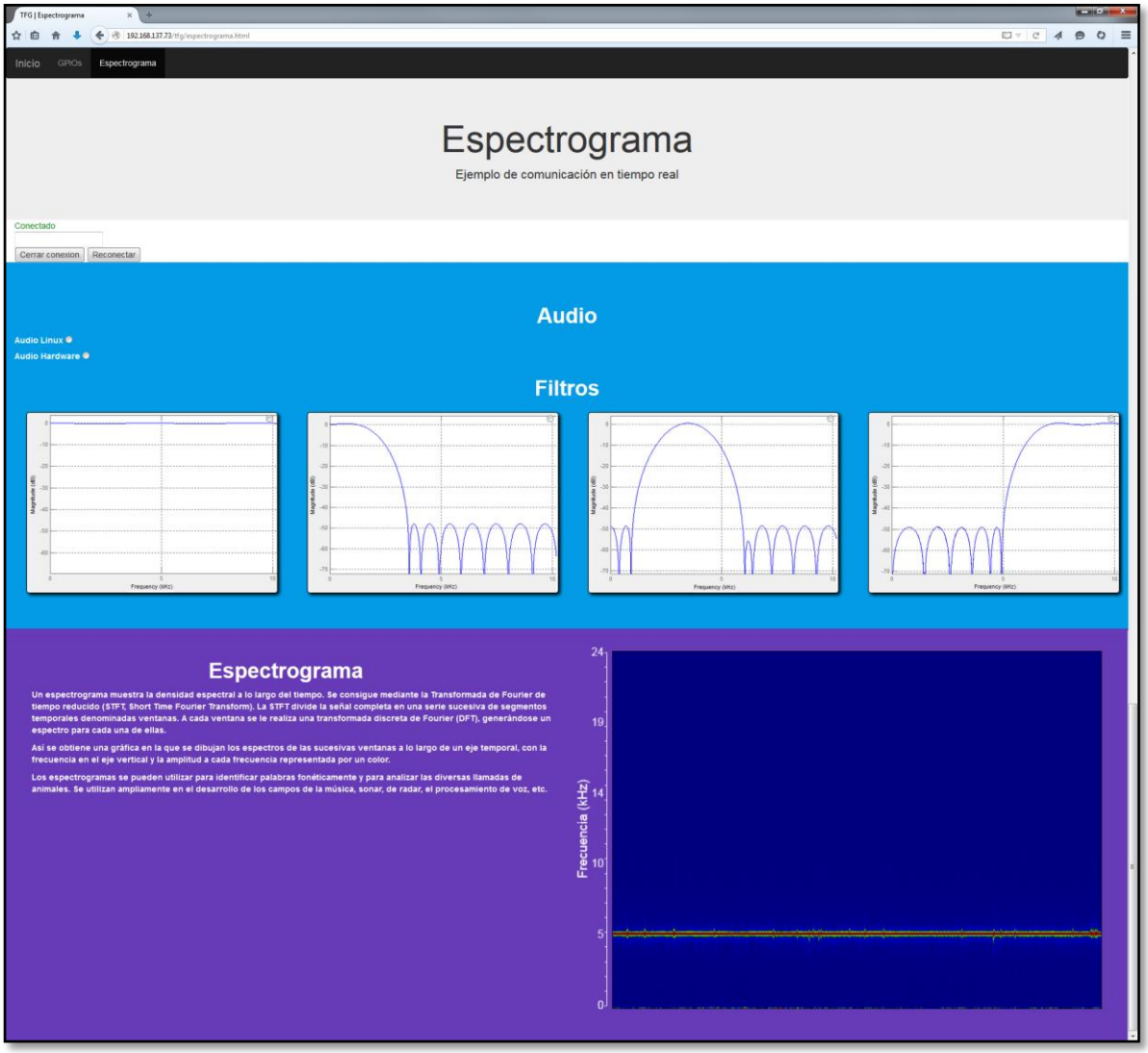

Figura 39 – Aplicación web espectrograma

6.4.3.2 Acceso a GPIOs

Para el control y monitorización de los GPIOs se ha creado otra página similar a la del espectrograma, también con AJAX y websocket para envío y recepción de datos. En ella aparecen, por una parte, los switches y los pulsadores que cambiarán su estado de OFF a ON cuando lo hagan en la placa y, por otra parte, los LEDs que permiten ser pulsados para encenderlos o apagarlos, lo que hace que se enciendan o apaguen también en la placa.

Al igual que con los filtros del apartado anterior, pulsar un LED hace saltar un evento que envía un 1 o 0 al servidor para indicar que se encienda o apague el LED físico correspondiente. La página tiene un aspecto similar:

Figura 40 – Aplicación web GPIOs

7 TRABAJOS FUTUROS Y CONCLUSIONES

The best way to predict the future is to invent it.

- Alan Kay -

7.1 Conclusiones

Se ha hecho un recorrido por la evolución de los procesadores y la lógica programable para llegar a entender el punto actual en el que estas dos tecnologías se han unido en un PSoC. Nos hemos centrado en un PSoC en concreto, el Zynq 7020 de Xilinx con su plataforma de desarrollo ZedBoard. Esta placa ha servido para ilustrar como llevar a cabo un codiseño HW/SW de una forma sencilla que elimine la mayoría de problemas asociados a comunicar aspectos hardware con software.

El punto más importante de este trabajo, y de cualquier otro que se realice, es la documentación y preparación previa. En este caso ha servido para conocer el sistema de desarrollo y lo más importante y que más ha marcado el devenir del trabajo, la elección del sistema operativo. El buscar entre varias alternativas sopesando las ventajas e inconvenientes de cada una ha hecho que se llegue a encontrar la solución más apropiada: Xillinux.

Esta distribución que está acompañada de la solución de Xillybus ha hecho posible que la comunicación entre FPGA y procesador sea muy intuitiva y fácil de implementar. Normalmente la comunicación entre ambas partes es el punto que más recursos y tiempo requiere y donde más errores se pueden acumular. Con la solución aportada, el mayor tiempo se ha dedicado a los desarrollos independientes en cada parte, dejando que el intercambio de datos entre ellas sea rápido y sencillo.

Para ejemplificar un codiseño se ha llevado a cabo una aplicación pretendiendo abarcar distintos tipos de comunicaciones: en tiempo real, cada cierto tiempo y mediante GPIOs. Ha sido un desarrollo que ha combinado muchas capas de trabajo, desde la electrónica hasta la programación de aplicaciones web, pasando por el procesamiento de señales, Linux y otras herramientas de trabajo. Lo que ha permitido desarrollar una visión amplia de varios campos de las telecomunicaciones y el logro de integrarlo todo en un trabajo.

Los PSoC abren una nueva ventana de aplicaciones para los sistemas embebidos, uniendo la capacidad de procesamiento en paralelo de la FPGA con la flexibilidad y posibilidades de Linux. Si se aprovecha como en este trabajo para añadir un servidor web y mantener el dispositivo conectado a Internet, se obtiene un sistema muy interesante para IoT, un mercado que crecerá a pasos agigantados en los próximos años. En definitiva, es un dispositivo que agrupa tantas tecnologías en su interior, que sus posibilidades son casi ilimitadas

7.2 Trabajos futuros

Los caminos de mejora futuros se pueden ver desde dos puntos de vista. Si se quiere continuar con la aplicación de ejemplo desarrollada o si se sigue por la vía del codiseño HW/SW en general. Según el camino elegido existen varios puntos que podrían mejorarse de cara a futuros desarrollos:

- Continuando el ejemplo desarrollado:
    - Una mejora de rendimiento consistiría en cambiar el servidor HTTP Apache y los scripts PHP por un demonio de Linux llamado *Websocketd* [41]. Esto eliminaría estas dos capas que son las que más consumos requieren.
    - En la parte FPGA, el IP core del filtro FIR permite una recarga dinámica de los coeficientes del filtro. Esto permitiría elegir cualquier filtro en tiempo de ejecución y no solo elegir entre los 3 prediseñados que se han incluido.
    - Incluir una función ventana cuando se calcula la STFT para obtener resultados más precisos eliminando los artefactos que se producen al cortar la señal en trozos.
    - Tamaño de la FFT configurable. Actualmente está fijo a 1024 puntos pero el IP core también permite cambiar este valor en tiempo de ejecución. También sería interesante poder cambiarlo desde Linux como el filtro.
    - Monitorización de los GPIOs mediante interrupciones o *polling* en Linux. Actualmente se hace una lectura constante del estado de los GPIOs, pero se podría hacer más eficiente haciendo que el kernel haga saltar una interrupción cuando haya un cambio en ellos para que nuestra aplicación lo atienda.
    - Tratamiento de errores. Uno de los objetivos del trabajo es mostrar la sencillez con la que se trabaja con Xillybus, por ello se ha mantenido el código lo más sencillo y claro posible, dejando a un lado qué hacer si ocurren errores al no poderse abrir un fichero, leer, escribir, etc.
- Enfocado en el codiseño HW/SW sin seguir con la aplicación desarrollada:
    - Actualizar a la nueva versión de Xillybus que saldrá en los próximos meses y que está enfocada en aumentar el ancho de banda máximo.
    - Un desarrollo usando las interfaces de memoria que proporciona Xillybus (*seekable stream*), ya que son muy útiles para escribir por ejemplo en registros de control de la FPGA.
    - Probar aplicaciones de coprocesamiento / aceleración hardware para aprovechar que la FPGA ejecuta ciertas operaciones mucho más rápido, con menor consumo de energía o de forma más eficiente. Es otra de las aplicaciones de Xillybus, no solo enviar datos de una parte a la otra sino que trabajen en paralelo en el mismo objetivo.
    - Elaborar un sistema para la depuración de Xillybus que permita encontrar fallos durante su uso. Es algo que no existe por defecto y que sería mucha ayuda.

REFERENCIAS

[1] C. Wilkes, «The evolution of a revolution,» 22 10 2007. [En línea]. Available: http://download.intel.com/pressroom/kits/IntelProcessorHistory.pdf. [Último acceso: 22 11 2015].

[2] R. González Carvajal y C. Luján, *Apuntes de Sistemas Embebidos,* Sevilla: Departamento de Ingeniería Electrónica, Escuela Superior de Ingeniería, Universidad de Sevilla, 2015.

[3] R. v. d. Meulen, «Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015,» Gartner, 10 2015. [En línea]. Available: http://www.gartner.com/newsroom/id/3165317. [Último acceso: 25 11 2015].

[4] F. Muñoz Chavero, *Apuntes Dispositivos Lógicos Programables,* Sevilla: Departamento de Ingeniería Electrónica, Escuela Superior de Ingeniería, Universidad de Sevilla, 2013.

[5] T. L. Floyd, «Software y lógica programable,» de *Fundamentos de sistemas digitales,* Madrid, Pearson Educación S.A., 2006.

[6] R. Ashby, «Why Use the Cypress PSoC?,» de *Designer's guide to the Cypress PSoC,* Burlington, MA : Elsevier Newnes, 2005.

[7] C. Semiconductor, «PSoC Software | Cypress,» [En línea]. Available: http://www.cypress.com/products/psoc-software. [Último acceso: 10 12 2015].

[8] Xilinx, «Hardware Zone,» [En línea]. Available: http://www.xilinx.com/products/design-tools/hardware-zone.html. [Último acceso: 10 12 2015].

[9] Xilinx, «Zynq-7000 All Programmable SoC Technical Reference Manual (UG585),» 10 04 2015. [En línea]. Available: www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. [Último acceso: 13 12 2015].

[10] Xilinx, «Zynq-7000 All Programmable SoC Overview,» 27 05 2015. [En línea]. Available: www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. [Último acceso: 13 12 2015].

[11] Avnet, «ZedBoard_HW_Users_Guide,» 27 01 2014. [En línea]. Available: http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf. [Último acceso: 17 12 2015].

[12] I. Xilinx, «Zynq-7000 All Programmable SoC Software Developers Guide (UG821),» 14 09 2015. [En línea]. Available: www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf. [Último acceso: 20 12 2015].

[13] Xilinx, «Xilinx Wiki Linux,» [En línea]. Available: http://www.wiki.xilinx.com/Linux. [Último acceso: 23 12 2015].

[14] Xilinx, «Xilinx Wiki - Petalinux,» [En línea]. Available: http://www.wiki.xilinx.com/PetaLinux. [Último acceso: 23 12 2015].

[15] Xilinx, «Petalinux Tools,» [En línea]. Available: http://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html. [Último acceso: 23 12 2015].

[16] I. Xilinx, «PetaLinux Tools Documentation: Workflow Tutorial (UG1156),» 25 11 2014. [En línea]. Available: http://www.xilinx.com/support/documentation/sw_manuals/petalinux2014_4/ug1156-petalinux-tools-workflow-tutorial.pdf. [Último acceso: 23 12 2015].

[17] I. Xilinx, «PetaLinux Tools Documentation: Reference Guide (UG1144),» 26 11 2014. [En línea]. Available: http://www.xilinx.com/support/documentation/sw_manuals/petalinux2014_4/ug1144-petalinux-tools-reference-guide.pdf. [Último acceso: 23 12 2015].

[18] I. Xilinx, «PetaLinux Tools Documentation: Command Line Reference Guide (UG1157),» 25 11 2014. [En línea]. Available: http://www.xilinx.com/support/documentation/sw_manuals/petalinux2014_4/ug1157-petalinux-tools-command-line-guide.pdf. [Último acceso: 23 12 2015].

[19] Xilinx, «PetaLinux Tools,» [En línea]. Available: http://www.xilinx.com/support/documentation-navigation/development-tools/software-development/petalinux-tools.html?resultsTablePreSelect=documenttype:SeeAll#documentation. [Último acceso: 23 12 2015].

[20] A. L. ARM, «Arch Linux ARM,» [En línea]. Available: http://archlinuxarm.org/. [Último acceso: 23 12 2015].

[21] A. Linux, «ArchWiki,» [En línea]. Available: https://wiki.archlinux.org/. [Último acceso: 23 12 2015].

[22] A. L. ARM, «ZedBoard | Arch Linux ARM,» [En línea]. Available: http://archlinuxarm.org/platforms/armv7/xilinx/zedboard. [Último acceso: 23 12 2015].

[23] Xillybus, «Xillinux: A Linux distribution for Zedboard, ZyBo, MicroZed and SocKit | xillybus.com,» [En línea]. Available: http://xillybus.com/xillinux. [Último acceso: 23 12 2015].

[24] Xillybus, «Principle of operation | xillybus.com,» [En línea]. Available: http://xillybus.com/doc/xilinx-pcie-principle-of-operation. [Último acceso: 23 12 2015].

[25] Xillybus, «Licensing,» [En línea]. Available: http://xillybus.com/licensing. [Último acceso: 23 12 2015].

[26] Xilinx, «Downloads,» [En línea]. Available: http://www.xilinx.com/support/download.html. [Último acceso: 26 12 2015].

[27] X. Ltd., «Getting started with Xillinux for Zynq-7000 EPP,» 13 2 2015. [En línea]. Available: http://xillybus.com/downloads/doc/xillybus_getting_started_zynq.pdf. [Último acceso: 29 12 2015].

[28] X. Ltd, «Getting started with Xillybus on a Linux host,» 30 09 2015. [En línea]. Available: http://xillybus.com/downloads/doc/xillybus_getting_started_linux.pdf. [Último acceso: 31 12 2015].

[29] X. Ltd, «Xillybus host application programming guide for Linux,» 11 07 2014. [En línea]. Available: http://xillybus.com/downloads/doc/xillybus_host_programming_guide_linux.pdf. [Último acceso: 31 12 2015].

[30] X. Ltd, «Xillybus FPGA designer's guide,» 14 02 2015. [En línea]. Available: http://xillybus.com/downloads/doc/xillybus_fpga_api.pdf. [Último acceso: 3 1 2016].

[31] X. Ltd, «The guide to defining a custom IP core,» 21 03 2014. [En línea]. Available: http://xillybus.com/downloads/doc/xillybus_custom_ip.pdf. [Último acceso: 03 01 2016].

[32] Xillybus, «Xillybus' data bandwidth,» [En línea]. Available: http://xillybus.com/doc/xillybus-bandwidth. [Último acceso: 10 01 2016].

[33] Xillybus, «Xillybus data latency,» [En línea]. Available: http://xillybus.com/doc/xillybus-latency. [Último acceso: 10 01 2016].

[34] Apuntes de Instrumentación Electrónica, Tema 11 - Analizadores de señal, Departamento de Ingeniería Electrónica, Universidad de Sevilla, 2011.

[35] P. Semiconductor, «I2S bus specification,» 14 07 1997. [En línea]. Available: http://web.archive.org/web/http://www.nxp.com/acrobat_download/various/I2SBUS.pdf. [Último acceso: 20 01 2016].

[36] I. Xilinx, «Xilinx PG149 LogiCORE IP FIR Compiler v7.1, Product Guide,» 05 03 2014. [En línea]. Available: http://www.xilinx.com/cgi-bin/docs/ipdoc?c=fir_compiler;v=v7_1;d=pg149-fir-compiler.pdf. [Último acceso: 20 01 2016].

[37] I. Xilinx, «Fast Fourier Transform v9.0 LogiCORE IP Product Guide (PG109),» 07 10 2015. [En línea]. Available: http://www.xilinx.com/cgi-bin/docs/ipdoc?c=xfft;v=v9_0;d=pg109-xfft.pdf. [Último acceso: 20 01 2016].

[38] sanwebe, «Chat-Using-WebSocket-and-PHP-Socket,» [En línea]. Available: https://github.com/sanwebe/Chat-Using-WebSocket-and-PHP-Socket. [Último acceso: 23 01 2016].

[39] S. Bleier, «spectrogram.js,» [En línea]. Available: https://github.com/sebleier/spectrogram.js. [Último acceso: 24 01 2016].

[40] R. Heyes, «rgraph | 2D/3D JavaScript charts - Free and Open Source,» [En línea]. Available: http://www.rgraph.net/. [Último acceso: 24 01 2016].

[41] websocketd, «WebSockets the Unix way,» [En línea]. Available: http://websocketd.com/. [Último acceso: 26 01 2016].