_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

# DESARROLLO DE UN SISTEMA DE CAPTURA DE DATOS Y PROGNOSIS PARA ESTABLECER AVISOS DE MANTENIMIENTO EN MAQUINAS CNCs BASADO EN FACTORY 4.0

**Juan José Albiach Sánchez**

**Tutor: José María Grima Palop**

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2015-16

Valencia, 1 de julio de 2016

# Agradecimientos

En primer lugar agradecer este proyecto a mi profesor tutor José María Grima, por haberme ayudado y dedicado su tiempo durante la realización de este proyecto de fin de grado, además de por haber sido un excelente profesor durante mis años como estudiante. También recordar a todos los profesores de la escuela que han contribuido con su esfuerzo y cercanía a formarnos.

Querría dedicar este proyecto a mis padres, por su cariño y apoyo por aguantar todas esas discusiones, momentos de estrés y desesperación estoicamente. A mis amigos, que supieron escucharme cuando lo necesité y corregirme cuando me equivocaba. A todos aquellos que hoy en día no se encuentran físicamente con nosotros pero los cuales nos protegen y cuidan desde allá donde estén.

Por último y no menos importante quiero dedicar este proyecto a ella, la voz serena y cauta que me guía en esta oscuridad.

# 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 de un diseño y el que por general más tiempo requiere y el lugar más propenso a cometer errores. Para paliar de forma primaria este problema, se hará uso de un código C que sea capaz de acceder a la memoria virtual completa de la placa. Después de esto se mapearan las direcciones pertinentes en RAM, para su posterior inclusión en un fichero .txt el cual guarde de forma provisional los datos recogidos por las memorias del diseño.

Para demostrar el funcionamiento del sistema se desarrollará un sistema de mensajería basado en colas con el cual extraeremos los datos del .txt para en un proyecto futuro en el cual se realizará un entono grafico para interpretar los mismos así como su persistencia en una base de datos.

Hay que agradecer a Ford S.L por la adquisición de la Z-turnBoard así como por el software proveniente de Xilinx adquirido.

# Resum

Aquest treball se centra en l'codisseny HW / SW d'un PSoC (Programmable System on Chip) format per un sistema de processament (PS) i lògica programable (PL). El codisseny es compon de diversos passos, com l'elecció d'un sistema operatiu per al SoC, la programació de la FPGA i el més important, la definició de la interfície de comunicació entre tots dos.

Habitualment la interfície de comunicació entre software i hardware és el punt més delicat d'un disseny i el que per general més temps requereix i el lloc més propens a cometre errors. Per pal·liar de forma primària aquest problema, es farà ús d'un codi C que siga capaç d'accedir a la memòria virtual completa de la placa. Després d'això es mapejaran les adreçes pertinents en RAM, per a la seu posterior inclusió en un fitxer .txt el qual guarde de manera provisional les dades recollides per les memòries del disseny.

Per demostrar el funcionament del sistema es desenvoluparà un sistema de missatgeria basat en cues amb el qual extraurem les dades del .txt per a un projecte futur, realitzar un entorn gràfic per interpretar els mateixos així com la seua persistència en una base de dades.

Cal agraïr a Ford S.L per l'adquisició de la Z-turnBoard així com pel software provinent de Xilinx adquirit.

# Abstract

This work focuses on the HW/SW co-design of a PSoC, composed of a processing system (PS) and a programable 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 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, we will make us of a C code to be able to Access the entire virtual memory of the board. After this, only the relevant addresses will be mapped into RAM, for subsequent inclusion in a .txt file which temporarily store the data collected by the memories of design.

To demonstrate the operation of the system, a system based messaging queues which will extract data from .txt will be developed. This data will be included in a future project in which a graphic intone will be made to interpret them. After this, this data will be written into a database.

# Índice

# Índice de Tablas

# Índice de Figuras

# Notación

| | |
|---|---|
| ADC | Analogic-to-Digital Converter |
| AMBA | Advanced Microcontroller Bus Architecture |
| 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 |
| FFT | Fast Fourier Transform |
| 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 |
| I2C | 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 |
| 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 |
| SGMII | Serial Gigabit Media Independent Interface |
| SoC | System on chip |
| SPI | Serial Peripheral Interface |
| SPLD | Simple Programmable Logic Device |
| SRAM | StaticRandom Access Memory |
| SSTL | Stub Series Terminated Logic |
| 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

## 1.1 Introducción

Ford España S.L lleva desarrollando desde hace años un proyecto cuyo concepto gira en torno al término Factory 4.0 o factoría inteligente. Dicho proyecto engloba diversos campos en los cuales se busca realizar un mantenimiento predictivo de las máquinas/estaciones.

En este proyecto se desarrollará uno de los campos buscados por la empresa, el cual es la prognosis de las máquinas de la línea de montaje mediante un hardware desarrollado.

## 1.2 Objetivos

El principal objetivo de este proyecto será mostrar a Ford España que es posible obtener una solución a su problema, notablemente más barata que las posibles soluciones que han ofrecido una serie de proveedores. Además de este objetivo también se mostrará a la empresa que esta solución es totalmente compatible con el proyecto Factory 4.0 que se está desarrollando en la planta.

Se realizará un codiseño HW/SW en un módulo Zynq-7000 AP SoC fabricado por Xilinx. Para ello será realizada una revisión del sistema utilizado así como 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 módulo que realice una captura mediante un XADC a los datos del cual, en segunda instancia, se les realizará una FFT (Fast Fourier Transform), dentro de la misma implementación hardware, cuya salida será enviada en tiempo real a la parte software, la cual estructurará los datos recibidos de una forma óptima para en un futuro proyecto almacenar dichos datos en una base de datos sobre la cual ,se aplicaran algoritmos de aprendizaje basados en redes neuronales para conocer el estado y/o probabilidades de rotura/fallo de las máquinas donde sea instalado dicho modulo.

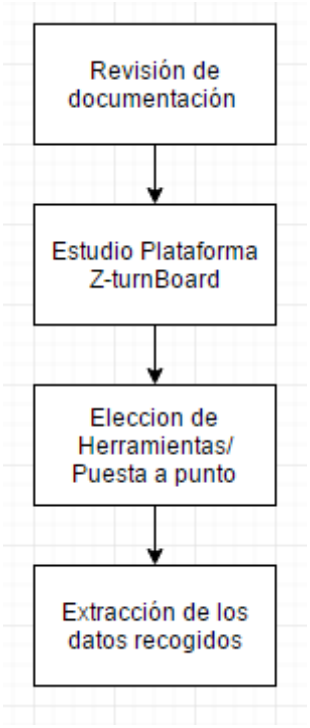
Tanto el diseño como la aplicación descrita serán integrados en una maquina CNC, en concreto en la operación OP35A, la cual se considera dentro de la planta de motores una operación critica debido a su complejidad. Dicha operación es la encargada de realizar la camisa de bloque (camisa de cilindro), la cual a su vez se encuentra en la línea de Mecanizado.

## 1.3 Metodología de Trabajo

El trabajo constará de las partes que se citan a continuación:

- Se comenzará 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 estudiará específicamente la plataforma Z-TurnBoard, incluyendo acerca de la misma documentación, datasheets y herramientas apropiadas para el trabajo con la misma.

- 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 creara un sistema para extraer los datos recogidos de la propia memoria y almacenarlos en un fichero temporal de tipo txt. Accediendo a este fichero se conseguirá extraer todos los datos recogidos por el diseño implementado.



## 2.1 Xilinx Zynq-7000 All Programmable SoC

### 2.1.1 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 MP Core™ 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.
- Diagnostico e imagen médica.
- Video 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 las7z030/7z035/7z045/7z100). [1]

## 2.1.2 Diagrama de Bloques

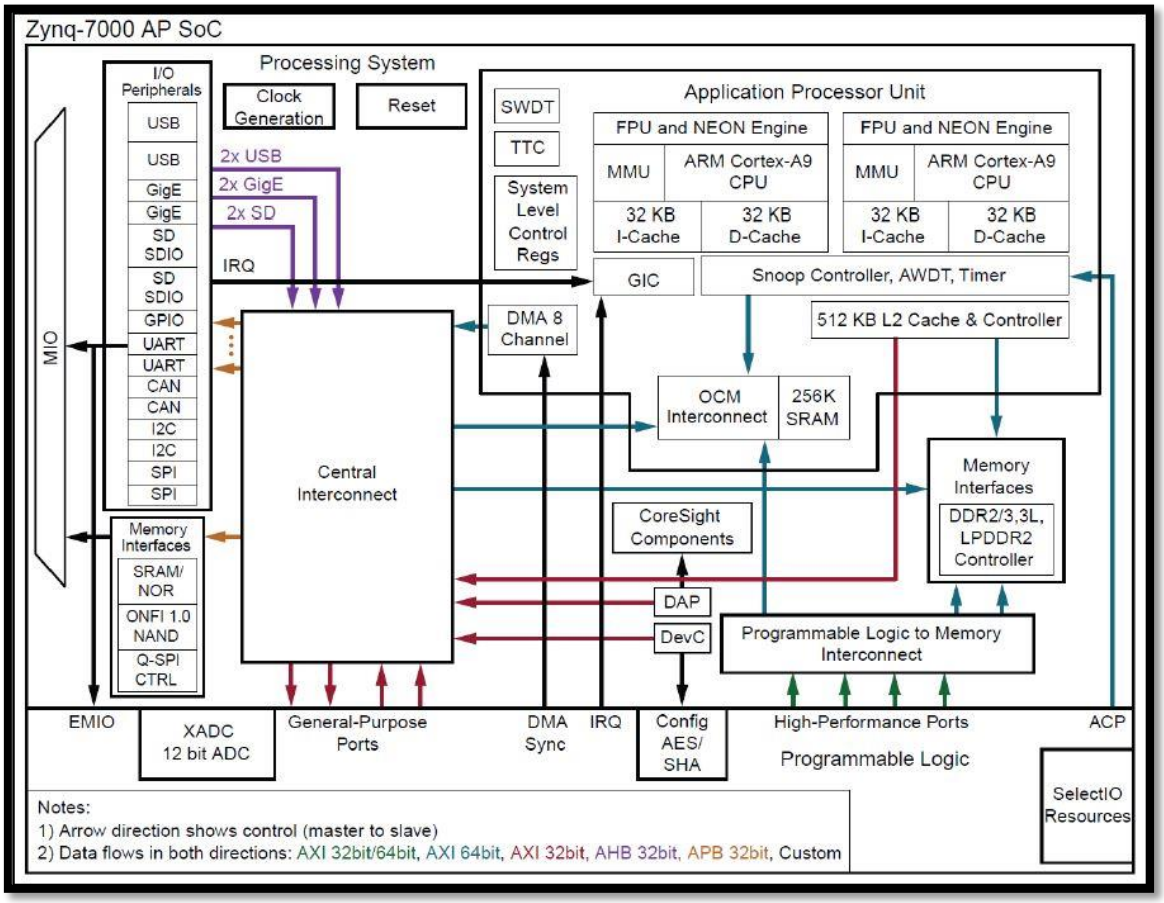



FIGURA 1 - DIAGRAMA DE BLOQUES ZYNQ-7000 AP SOC

## 2.1.3 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 timer triple.

### Cachés

- 32 KB de caché de nivel 1 asociativa de 4 vias (independiente por cada CPU).
- 512 KB de caché de nivel 2 asociativa de 8 vias (compartida por las CPUs).

- Soporte de 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 de 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 hasta12 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.

**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.

**PCIExpress**

- 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 contínuo por JTAG.

### 2.1.4 Dispositivos de la familia

Existen 7 dispositivos de la familia Zynq-7000, con características muy parecidas en el PS, pero con bastantes diferencias en la PL ya que cada uno usa un modelo diferente de FPGA de Xilinx. Se recogen en la siguiente tabla: [2]

| 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 Simple / Double en coma flotante para cada procesador | | | | | | |
| | Frecuencia Maxima | 667 MHz; 766 MHz; 866 MHz | | | 667 MHz; 800MHz; 1 GHz | | | 667 MHz<br>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 estatica | 2x Quad-SPI, NAND, NOR | | | | | | |
| | Canales DMA | 8 (4 dedicas a la PL) | | | | | | |
| | Perifericos | 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x32b GPIO | | | | | | |
| | Perifericos con DMA | 2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO | | | | | | |
| | Seguridad[1] | Autentificacion RSA, AES, y SHA-256-bit cifrado y autentificacion Secure Boot | | | | | | |
| **Interfaces entre PS y PL (Interfaces primarias e interrupciones solo)** | | 2x AXI 32b Maestro 2x AXI 32-bit Esclavo<br>AXI 64-bit ACP<br>Memoria 4x AXI 64-bit/32bit<br>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 logica programable (Puertas ASIC aproximadas[2]) | 28K (~430K) | 74K (~1,1M) | 85K (~1,3M) | 125K(~1,9) | 275K(~4,1M) | 350K(~5,2M) | 444K(~6,6M) |
| | LUTs | 17600 | 42600 | 53200 | 78600 | 171900 | 218600 | 277400 |
| | Biestables | 35200 | 92400 | 106400 | 157200 | 343800 | 437200 | 554800 |
| | RAM de bloque (nº de bloques 36 Kb) | 240 KB(60) | 380 KB (95) | 569 KB (140) | 1060 KB (265) | 2000 KB (500) | 2180 KB (545) | 3020 KB (755) |
| | DSP Slices | 80 | 160 | 220 | 400 | 900 | 900 | 2020 |
| | Rendimiento de pico de los DSP (FIR simetrico) | 100 GMAC's | 200 GMAC`s | 276 GMAC's | 593 GMAC's | 1334 GMAC's | 1334 GMAC's | 2622 GMAC's |
| | PCI Express | - | Gen2 x4 | - | Gen2 x4 | Gen2 x8 | Gen2 x8 | Gen2 x8 |
| | XADC | 2x 1 MSPS ADC's, 12 bits con hasta 17 entradas diferenciales | | | | | | |
| | Seguridad | AES y SHA 256b para código de arranque y configuración, encriptado y autentificación de la PL | | | | | | |

TABLA 1 - MODELOS DE LA FAMILIA ZYNQ-7000

## 2.2 Z-turnBoard™

### 2.2.1 Introducción

La Z-turnBoard es una SBC (Single Board Computer) de bajo coste y alto rendimiento basada en los Xilinx Zynq-7010 o Zynq-7020 APSoC, los cuales combinan un procesador ARM® Cortex™-A9 MPCore™ de doble núcleo con una Artix-7 FPGA de Xilinx de 85000 celdas lógicas. Es una plataforma ideal para desarrollar por la gran cantidad de periféricos que contiene y sus posibilidades de expansión. [3]

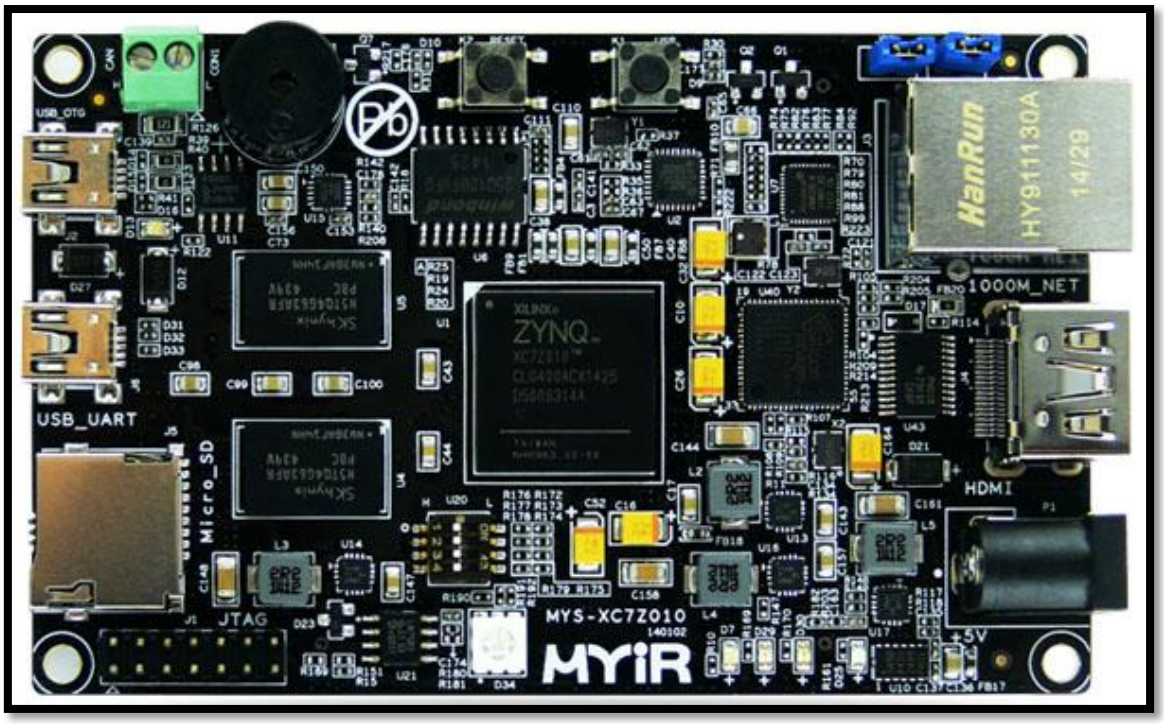



FIGURA 2 – Z-TURNBOARD TOP VIEW

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

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

Para terminar con su introducción sus características son las que se muestran a continuación:

- Xilinx®XC7Z020-1CLG400C Zynq AP SoC:

    - Configuración principal: tarjeta SD.
    - Opciones de configuración auxiliares: QSPI Flash, JTAG y USB UART.
- Memoria:
    - 1 GB DDR3 (2 x 512 MB, 32-bit).
    - 16 MB QSPI Flash.
- Interfaces:
    - 10/100/1000 Ethernet.
    - USB OTG 2.0.
    - Tarjeta SD.
    - USB 2.0 FS puente USB-UART.
    - 1 Bus CAN.
    - 1 LPC FMC (Low Pin Count FPGA).
    - 2 botones de reset (1 PS, 1 PL).
    - Switch Selector de 4 canales.
    - 5 LEDs (3 LEDs usuario, 1 LED indicador de energía, 1 RGB LED).
    - 1 Buzzer.
    - Acelerómetro de 3 ejes sobre la misma placa.
    - Sensor de temperatura.
- Imagen y audio
    - Salida HDMI.
- Alimentación:
    - Regulador AC/DC 5V @ 2ª.
- Software
    - ISE WebPACK Design Suite.
    - Vivado Desing Suite.
    - Vivado HLS.
    - Xilinx SDK.

Es necesario citar, que junto a la compra de esta placa, se adquirio la I/O cape de la misma. Esta I/O cape proporciona al módulo citado anteriormente más interfaces, como son los que se citan a continuación:

- XADC doble de 12 bits.
- 3 Conectores PMOD compatibles con Digilent.
- 1 Conector para Cámara.
- 1 Conector para un LCD táctil.
- Una serie de Pines para I/O de la misma.

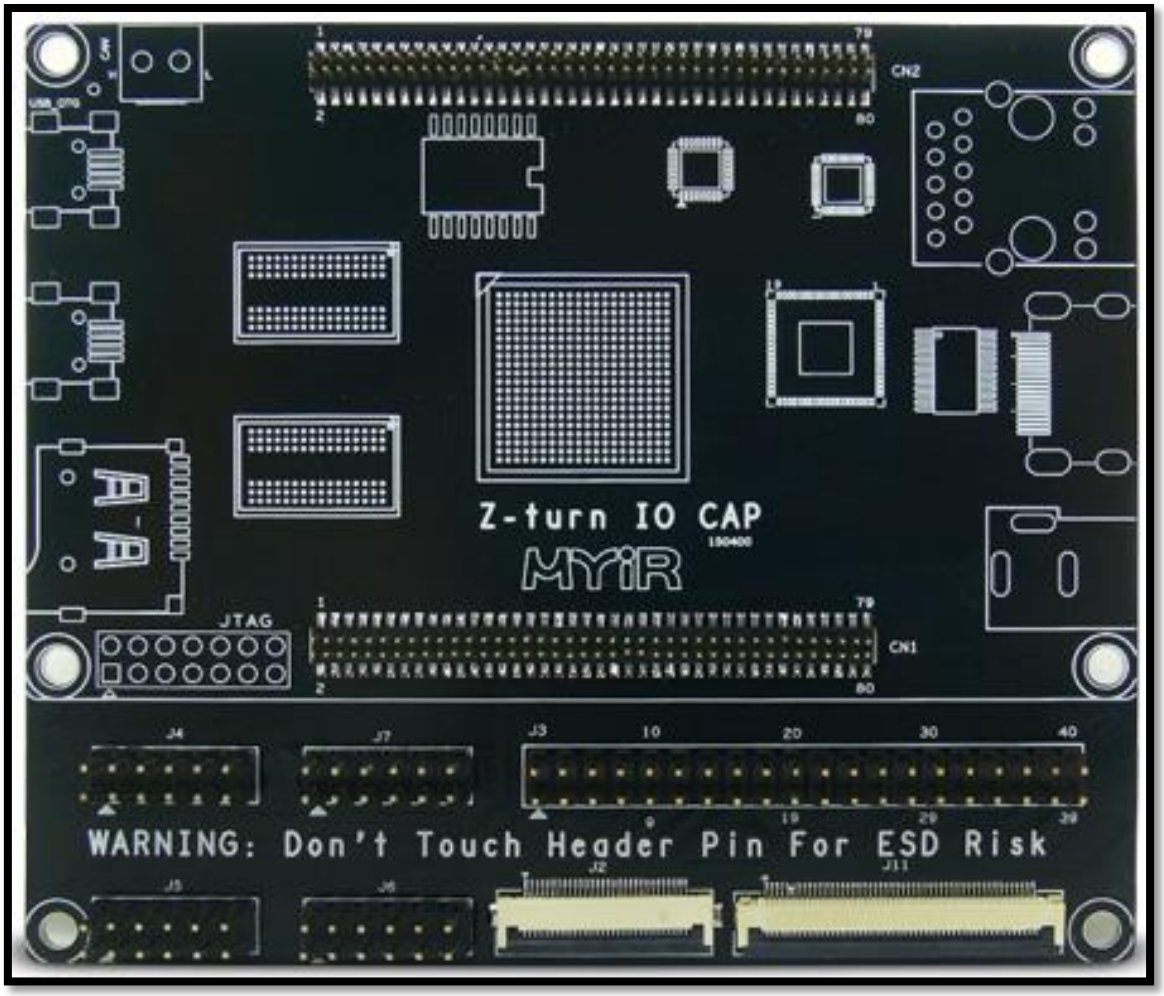



FIGURA 3 - ZTURN I/O CAPE

## 2.2.2 Características y Componentes

### 2.2.2.1 Memoria DDR3

La Z-turnBoard incluye dos módulos DDR3 de 512 Mb con una longitud de palabra de 32 bits, que crean un total de 1Gb de memoria.

Las herramientas de Xilinx permiten además, entre otras cosas, el entrenamiento de DRAM a través de Xilinx Platform Studio o del IP Editor de Vivado. Esto se consigue modificando 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 de la misma.

### 2.2.2.2 QSPI Flash

Memoria Flash NOR serie quad-SPI de 16 Mb. Esta se puede usar tanto para inicializar PS como para configurar la PL (bitstream).

### 2.2.2.3 Tarjeta SD

La tarjeta SD poder ser utilizada de dos formas: como memoria externa no volátil y como sistema de booteo para arrancar el Zynq. El conector de dicha tarjeta de memoria, es un conector estándar de 9 pines TE 2041021-1, el cual a su vez se encuentra conectado con el periférico SD/SDIO del PS del Zynq para controlar la comunicación con la SD.

Es importante citar que para utilizar dicha tarjeta SD como sistema de booteo, los jumpers J1 y J2 deberán encontrase en modo On y Off respectivamente.

### 2.2.2.4 USB OTG

La Z-turnBoard implementa un interfaz USB OTG como PHY usando un chip común para todas las placas de la misma familia, el TI TUSB1210 Standalone USB Transceiver Chip. El chip soporta velocidades de hasta 480 Mb/s y opera a un voltaje de 1,8 V.

Es necesario citar que este USB no alimenta la placa, sin embargo si se modifican los jumpers de forma que la placa funcione en modo HOST u OTG, esta proporcionara 5 V.

### 2.2.2.5 Puente USB a UART

Esta función la proporciona un dispositivo el cual es genérico para toda la gama de la familia Zynq 7000 AP SoC. El dispositivo en cuestión es Cypress CY7C64225 USB-to-UART Bridge, el cual se conecta al periférico UART del PS. El conector externo de la Z-turnBoard 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 a su vez proporciona drivers para Virtual COM Port (VCP) de acceso gratuito, que permite que el UART le aparezca al ordenador que esté conectado a la placa como un puerto COM así puede usarse software para acceder a la misma del tipo Tera Term, Putty, HyperTerm, etc…

### 2.2.2.7 HDMI

El transmisor TPD12S016 HDMI de Texas Instruments proporciona la interfaz de video digital. Se trata de un transmisor a 225 MHz compatible con HDMI 1.4 que soporta 1080p60 con modo de color de 16 bits, YCbCr, 4:2:2.

Texas Instruments ofrece ciertos drivers de Linux y diseños de referencia que muestran como interactuar con este dispositivo.

### 2.2.2.8 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 dispositivos que pueda ser reseteado. La Z-turnBoard, al igual que otras placas de la misma familia, 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 Z-turnBoard son reseteados por esta señal.

- **PS Reset o User 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.

### 2.2.2.9 I/O de usuario

La Z-turnBoard dispone de los siguientes elementos en cuanto a I/O de usuario se refiere:

- Acelerómetro de 3 ejes montado sobre la misma placa.
- Sensor de temperatura digital y watchdog termal STLM75 con rangos de entre -55ºC y 125ºC con ±0.5ºC de precisión.
- Switch selector de 4 canales de entrada.
- 5 LEDs entre los cuales se encuentran: 3 LEDs de usuario, 1 LEDs indicativo de alimentación y 1 RGB LED.
- 1 Buzzer TMB12A05.

### 2.2.2.10 10/100/1000 Ethernet

La Z-turnBoard implementa un puerto Ethernet 10/100/1000 usando un Atheros AR8035, conectado al Zynq mediante la interfaz estandarizada RGMII. Opera a 1,8 V e incorpora dos LEDs de estado que indican tráfico y estado valido del enlace.

### 2.2.2.11 Conectores Digilent Pmod

Si se hace uso de una I/O cape, la Z-turnBoard tiene 3 conectores PMOD compatibles con los proporcionados por Digilent. Estos son de ángulo recto, machos y de 0,1" que incluyen ocho I/O más dos señales de tierra más dos señales de 3,3 V como se muestra en la figura 8. Todos y cada uno de los PMOD están conectados a la PL.

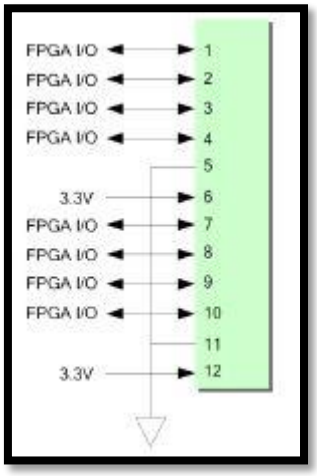



FIGURA 4 - CONEXIONES PMOD (FTE: DIGILENT)

### 2.2.2.12 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 la realización de un reset los pines de modo son leídos para determinar que dispositivo será el de arranque primario.

Los pines encargados son: JP1 y JP2, que permiten cambiar los modos incluyendo el uso de JTAG en cascada. Más información de sus combinaciones se puede encontrar en [3].

Para profundizar en las etapas de arranque, se proporciona más información en el capítulo 3 “Boot and configuration” de [4].

### 2.2.2.13 JTAG

La Z-turnBoard proporciona un conector tradicional de JTAG para ser usado con los cables de Xilinx Platform y los cables JTAG HS1 de Digilent.

## 2.3 Revisión de las 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:

- **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: [4] [16]

- **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 será usado Vivado Design Suite, en un capitulo posterior se explicará con más detalle el entorno y sus herramientas de desarrollo.

# 3 Preparación del Sistema

## 3.1 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, 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.

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.

## 3.2 Selección del sistema operativo

### 3.2.1 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.[6]

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: [7]

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

Aparte de estas herramientas, las cuales estarían orientadas al desarrollador, la propia distribución de Linux también incluye una serie de paquetes, algunos se citan a continuación:

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

Petalinux está disponible completamente gratuito y sin ninguna restricción de en cuanto a licencias se refiere. Si se desea más información acerca de este SO se puede acudir a: [8], [9], [10], [11].

### 3.2.2 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. [12]

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. [13]

Arch está bajo la licencia GNU GPLv2 (software libre) y además está disponible de manera gratuita. La información sobre su instalación en la Z-turnBoard (ZedBoard, Zybo, MicroZed) se puede encontrar en [14].

### 3.2.3 Xillinux

Xillinux es una distribución que incluye Linux y un kit de código FPGA para las plataformas ZedBoard, ZyBo, SocKit board y MicroZed (Para la Z-turnBoard se puede gastar cualquiera de los anteriores, ya que se trata de una placa que derivaría de las mismas). 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. [15]

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 [16].

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.

### 3.2.4 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.
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.

### 3.2.5 Tabla Comparativa

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

TABLA 2 - COMPARACIÓN DE SO LINUX

Para este trabajo la distribución de Linux elegida ha sido Xillinux, debido a que su uso es relativamente fácil y a que incluye Xillibus, que aunque no se gaste en este desarrollo en un primer momento, hará que en desarrollos/mejoras futuras de la aplicación el codiseño entre PS y PL sea mucho más sencillo y menos propenso a la contención de errores. Gracias a estos cores de Xillibus, el desarrollador podrá centrarse más en perfeccionar la aplicación sin perder tanto tiempo en la comunicación entre ambas partes.

## 3.3 Herramientas de desarrollo hardware

Una vez preparada la parte software es necesario elegir y poner a punto las herramientas de la parte hardware. Al ser un núcleo fabricado por Xilinx, lo más normal será usar herramientas que proporcione el propio fabricante del dispositivo. Hasta hace bien poco la herramienta usada por excelencia para este tipo de proyecto era Xilinx ISE, la cual actualmente se encuentra obsoleta. Dicho esto, se hará uso del paquete Vivado proporcionado por Xilinx y en el cual se incluyen: Vivado HLS (High Level Synthesis), Xilinx SDK y Vivado Design Suite. De estos tres últimos, después de haber realizado numerosas pruebas y después de haber revisado todos y cada uno de los manuales disponibles por parte del fabricante, el escogido será Vivado Design Suite en la versión 2015.3.

### 3.3.1 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, como se ha citado anteriormente, es posible descargarlo desde la página oficial de Xilinx [16] en todas sus versiones (desde la 2011 hasta la actual 2016.3). El proceso de instalación será como si de otro programa en Windows se tratase, guiado por un instalador y muy sencillo. Una vez el programa ha sido instalado, simplemente será necesario hacer doble click sobre su acceso directo y elegir la opción de nuevo proyecto. Es necesario citar, que no directamente se pasará a la pantalla que se muestra a continuación, sino que el usuario será guiado por una serie de pantallas en las cuales deberá citar el tipo de chip que pretende utilizar en el desarrollo, así como si este precisa de bloques IPs o contantes xdc propias o generadas a partir de otro programa del fabricante. Dicho esto y habiendo pasado el usuario por todos los pasos que el configurador del proyecto requiere, se llegará a la pantalla que sigue a continuación, de la cual se explicarán las posibilidades más interesantes:

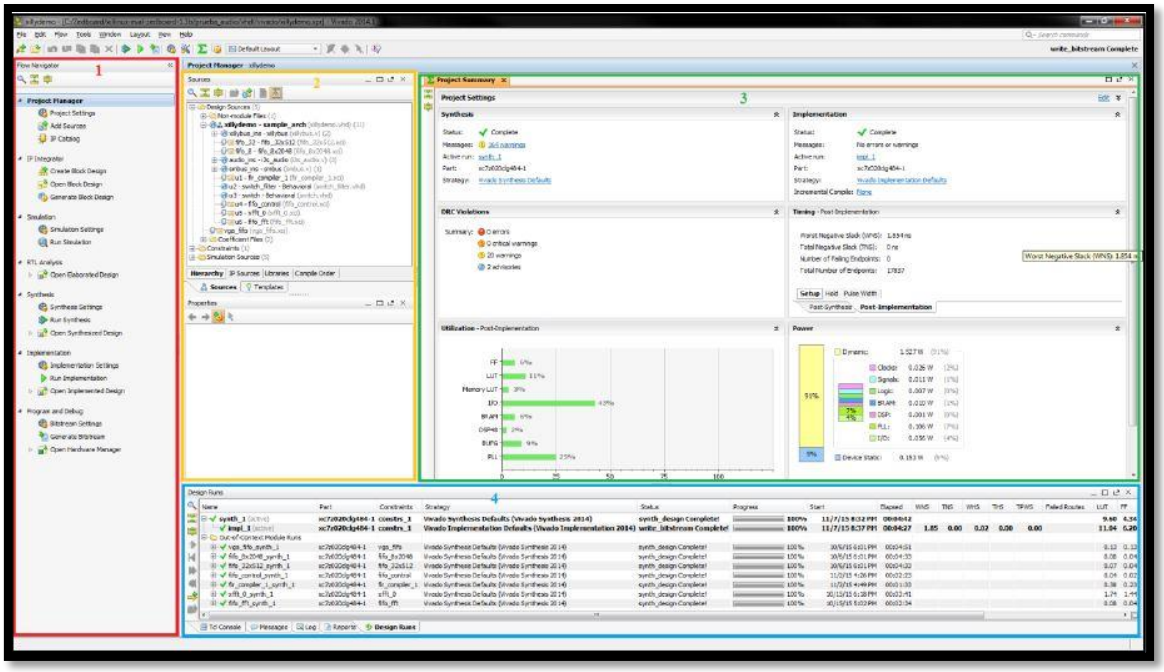


FIGURA 5 - ENTORNO DESARROLLO VIVADO

1. **Flow Navigator:** Proporciona acceso a los comandos y herramientas para llevar la aplicación desde la generación de un diagrama de bloques hasta su síntesis, implementación y generación del bitstream correspondiente. Como apartados con más relevancia se pueden citar:
    a. **IP Catalog:** Aquí se muestran todas y cada una de las IPs del catálogo de Xilinx ordenadas por su función o procedencia. Para añadirlas al proyecto bastará con seleccionarlas o arrastrarlas dentro del workspace. Este proceso es llevado a cabo mediante un asistente en el cual también se configuran las funcionalidades del IP en función del uso que se va a hacer del mismo.
    b. **Open Elaborated Design:** genera un esquemático en el que se muestran todos los bloques programados y las correspondientes interconexiones entre los mismos. Aparecerían las señales y sus propiedades como pueden ser el ancho o el tipo de palabra. Esta vista es considerada una de las mejores herramientas para depurar dentro de Vivado Design Suite.
    c. **Run Synthesis:** lleva a cabo la síntesis del diseño.
    d. **Run Implementation:** tras realizar la síntesis, aquí se implementa el diseño.
    e. **Generate Bitstream:** si se hace click en esta opción, se crea el bitstream. Es posible hacer click directamente sobre esta opción y automáticamente se realizaran todos los pasos anteriores a este. Si el diseño es el correcto y cumple con los requisitos, el bitstream se creará satisfactoriamente.

       Se podría decir que este paso es el más importante de cara a la realización de nuestro proyecto, ya que el bitstream generado, será cargado mediante comandos de Linux directamente sobre la FPGA de nuestro sistema embebido adquiriendo este la funcionalidad programada desde Vivado.

2. **Data Windows Area:** Esta sección mostrara información relacionada con los ficheros fuente y datos, como podría ser:
    a. **Sources:** serán mostrados todos los ficheros de forma jerárquica, ya sean ficheros fuente, IPs usadas, librerías y el orden de compilación. Desde la pestaña *IP Sources* es posible observar las distintas plantillas como las de instanciar IPs o los testbench
    b. **Properties:** simplemente mostraría información acerca del objeto seleccionado dentro del workspace del usuario.
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 propio proyecto implementado.
4. **Results Window Area:** Muestra el estado y el resultado de los comandos ejecutados en el programa por el usuario. Se encuentra dividida en varias pestañas, las cuales son explicadas brevemente a continuación:
    a. **Tcl Console:** permite introducir comandos Tcl y ver el resultado de comandos anteriores asi como los ejecutados por Vivado. Su funcionamiento sería similar a 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. **Design Runs:** ver y gestionar todos los procesos ejecutados, con vista detallada del último proceso de cada IP.

# 4 Descripción de la solución

## 4.1 Introducción

Una vez ha sido elegido y preparado todo el entorno de desarrollo y las herramientas que deberán ser utilizadas, es hora de diseñar el sistema. En primer lugar será necesario conocer en más profundidad el sistema operativo (SO) que va a ser empleado, pues de él depende gran parte del diseño y de las futuras mejoras que se nombraran en el apartado 7 de esta memoria.

## 4.2 Xillinux

Xillinux es una distribución que se encuentra basada en Ubuntu 12.04 para los dispositivos de la familia Zynq-7000, pensada como una plataforma para el desarrollo rápido de software con FPGA. Soporta ZedBoard, MicroZed y Zybo, además de los derivados de los mismos, como es el caso que se muestra en esta memoria con la Z-turnBoard.

Está pensada 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 mediante programas que simulen un terminal remoto (PuTTy, TeraTerm, etc...). Incluye parte de lógica hardware como por ejemplo el adaptador HDMI. 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.

Lo que verdaderamente hace interesante a Xillinux es la inclusión de Xillybus, que consiste en un IP core para la propia FPGA y de un driver de Linux para la integración entre lógica programable del dispositivo y las aplicaciones de espacio de usuario. Acerca de este IP Core profundizaremos más en el apartado 7 ya que queda pendiente entre el desarrollador y la empresa en cuestión un desarrollo posterior a este proyecto, por lo tanto será considerado como una futura mejora del proyecto. [15]


FIGURA 6 - ENTORNO GRAFICO XILLINUX

## 4.3 DMA

### 4.3.1 Introducción

AXI DMA es un IP Core proporcionado por Xilinx que permite realizar transferencias de datos entre memoria, a través de una interfaz AXI4 Memory Mapped, y periféricos con interfaz AXI4-Stream.

Uno de los motivos del empleo del DMA para realizar un movimiento de datos es evitar la sobrecarga del procesador en dicho proceso.

### 4.3.2 IP Core

El DMA de Xilinx proporciona diversos modos de funcionamiento, algunos de ellos son: Micro DMA, multicanal, transferencias en dos dimensiones, etc. Sin embargo, en este trabajo se centrará la atención el empleo del modo Direct Register Mode (Simple DMA).

Este modo proporciona una configuración para realizar transferencias simples de DMA por MM2S y S2MM las cuales hacen un uso menor de los recursos de la FPGA.

La figura 12 muestra la estructura interna del IP del DMA. En este apartado se llevara a cabo una introducción al DMA y su funcionamiento. Para más información consúltese [16].

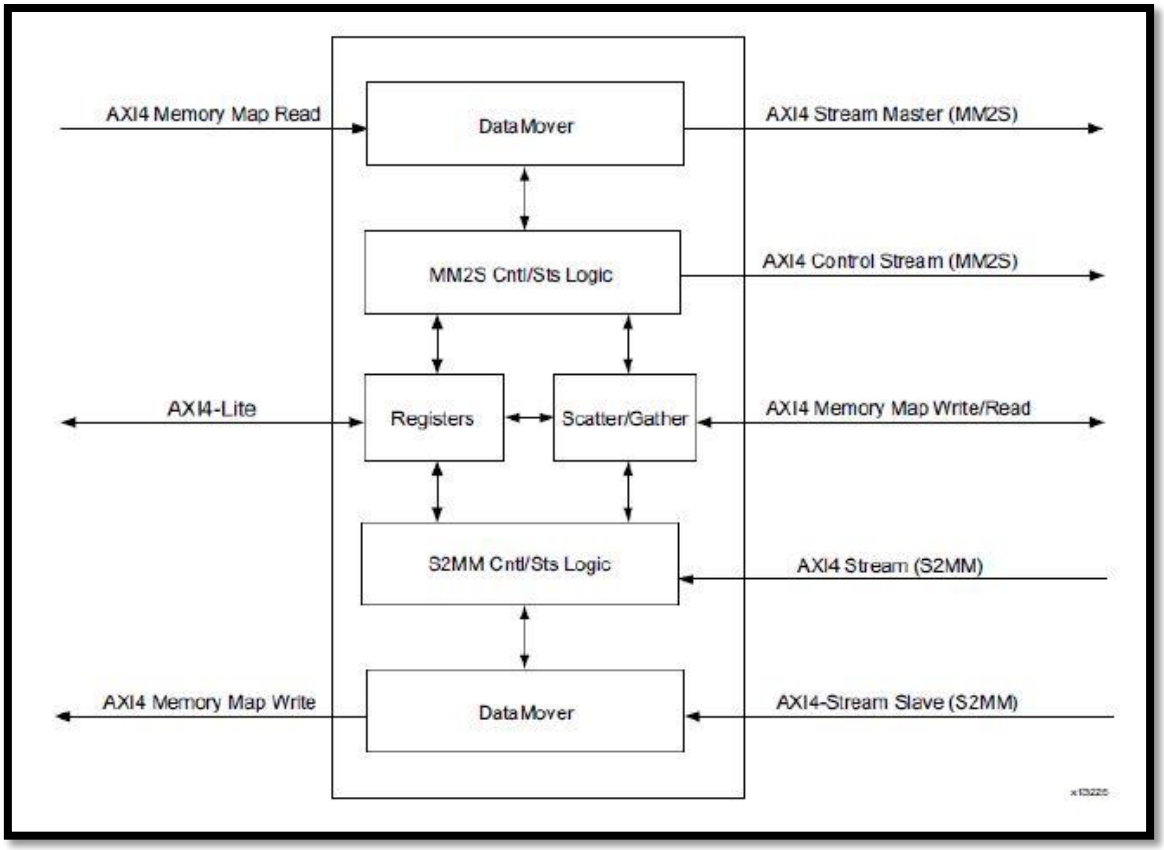



FIGURA 7 - ESQUEMA MODULO DMA

Para una mejor comprensión del DMA, se definirán dos tipos de canales que se identificarán mediante los acrónimos MM2S y S2MM. Estos canales estarán asociados a los conceptos de transmisión y recepción como se explica a continuación:

- **MM2S**: Canal de transmisión donde los datos procedentes de la memoria (a través de una interfaz mapeada en memoria) se transmiten como un stream de datos hacia la PL.
- **S2MM**: Canal de recepción donde los datos procedentes de la PL en forma de stream se transmiten a memoria a través de una interfaz mapeada en memoria. Tal como se muestra en la Figura 2-14, algunas de las interfaces están formadas por varios canales.

Estas interfaces son:

- **Interfaz MM2S:** Formada por un canal de datos y un canal de control, ambos AXI4-Stream.
  - **AXI4-Stream Master:** Canal por el que se transmiten los paquetes de datos procedentes de la memoria, en formato stream.
  - **AXI4 Control Stream:** Canal por el que se envía información de control y datos de usuario al cliente DMA en la PL.
- **Interfaz S2MM**: Formada por un canal de datos y un canal de control, ambos AXI4-Stream.

    - **AXI4-Stream Slave:** Canal de por el que se recibe un stream de datos procedente de la PL que se desea enviar a memoria.
    - **AXI4 Stream (Control):** Canal de tipo stream por el que se recibe información de control e información de usuario procedentes del cliente DMA.
- **Interfaz AXI-Lite:** Canal destinado a la configuración de los registros del DMA.
- **Interfaz AXI Scatter/Gather:** Encargada de la configuración de registros específicos del modo de funcionamiento Scatter/Gather.
- **Interfaces AXI4 Memory Map:** Canales de lectura y escritura destinados a la comunicación entre el DMA y la memoria.

## 4.4 FFT (Fast Fourier Transform)

FFT es la abreviatura usual (del inglés Fast Fourier Transform) de un eficiente algoritmo que permite calcular la transformada de Fourier discreta (DFT) y su inversa. La FFT es de gran importancia en una amplia variedad de aplicaciones, desde el tratamiento digital de señales y filtrado digital en general a la resolución de ecuaciones en derivadas parciales o los algoritmos de multiplicación rápida de grandes enteros. El algoritmo pone algunas limitaciones en la señal y en el espectro resultante. Por ejemplo: la señal de la que se tomaron muestras y que se va a transformar debe consistir de un número de muestras igual a una potencia de dos. La mayoría de los analizadores TRF permiten la transformación de 512, 1024, 2048 o 4096 muestras. El rango de frecuencias cubierto por el análisis TRF depende de la cantidad de muestras recogidas y de la proporción de muestreo.

Uno de los algoritmos aritméticos más ampliamente utilizados es la transformada rápida de Fourier, un medio eficaz de ejecutar un cálculo matemático básico y de frecuente empleo. La transformada rápida de Fourier es de importancia fundamental en el análisis matemático y ha sido objeto de numerosos estudios. La aparición de un algoritmo eficaz para esta operación fue una piedra angular en la historia de la informática.

Las aplicaciones de la transformada rápida de Fourier son múltiples. Es la base de muchas operaciones fundamentales del procesamiento de señales, donde tiene amplia utilización.

En este proyecto en concreto, la FFT será implementada mediante un diseño jerárquico implementado en vivado, la cual constará de una serie de bloques interconectados que definirán el funcionamiento del propio Core top (jerarquía). Se describirá más a fondo este bloque jerárquico en el apartado 6, en el cual se describe el desarrollo de la aplicación final que ha sido implementada.

## 4.5 Accelerator Coherency Port (ACP)

El Accelerator Coherency Port (ACP) es una interfaz esclava de 64 bits de bus de datos conectada directamente a la SCU que proporciona accesos a la memoria con coherencia de caché.
En los diseños de SoCs basados en FPGA es común tener que realizar transferencias de datos desde el PS hacia la PL, para ello se suele utilizar el DMA. Al ser el procesador quien modifica

los datos a transferir, es posible que los datos existentes en la memoria caché no se correspondan con los que hay almacenados en la memoria del dispositivo, luego los datos no son coherentes. Es aquí donde cobra importancia el uso del ACP para asegurar la coherencia de datos entre ambas memorias.
Además de la ventaja de tener accesos coherentes a la memoria, el uso del ACP mejora el rendimiento del sistema y la energía consumida disminuye. Esto es debido a que al mantener la coherencia de caché, no es necesario realizar limpiados de memoria (flush), siendo más eficiente.
En la figura 13 se muestra la conexión entre la SCU, las memorias cache y de dispositivo y el puerto ACP.

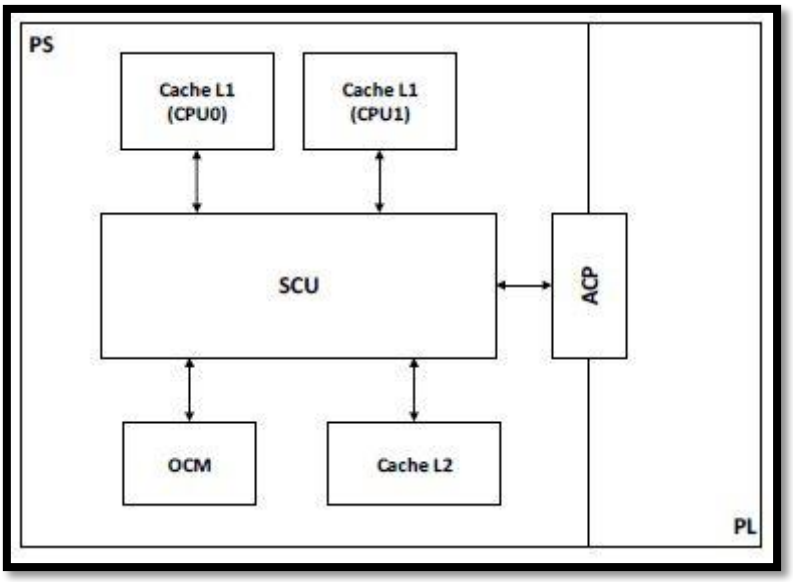



FIGURA 8 - INTERCONEXIÓN ACP PS/PL

El procedimiento del ACP para realizar una lectura de datos de la memoria de forma coherente sigue los siguientes pasos. Para ello es necesario que las siguientes señales de la interfaz AXI tengan los valores: **ARUSER[0]** = 1 y **ARCACHE[1]** = 1.

1. Comprobar las memorias caché de nivel 1 del procesador.
2. Si los datos que se desean leer no se encuentran en dichas memorias, se comprueba la memoria caché de nivel 2.
3. Si tampoco están los datos en esta memoria caché se realiza una lectura de la memoria del dispositivo. Para el proceso de escritura en memoria de forma coherente, también es necesario **ARUSER[0]** = 1 y **ARCACHE[1]** = 1. El procedimiento es el siguiente:
    1. Si los datos se encuentran en la memoria caché del procesador, primero se invalidan dichos datos.
    2. Una vez invalidados los datos (o en el caso de que no existiesen) se realiza la escritura.

.

# 5 Desarrollo de la Aplicación Final

## 5.1 Instalación del Sistema Operativo

El único material necesario para esta acción será una tarjeta SD, la cual se recomienda que sea de la marca Sandisk ya que otras marcas suelen reportar errores para este tipo de trabajos. En esta tarjeta bastara con copiar los archivos necesarios para el arranque de Xillinux. Los archivos pueden ser encontrados en su página web (http://xillybus.com/xillinux) y son dos:

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

En este punto la placa ya estaría preparada para arrancar con sus sistema operativo, aunque faltarían una serie de paquetes que el propio usuario deberá descargar editando el interfaz de red de su placa y usando el comando **apt-get**. En esta memoria no se va a entrar en profundidad en como escoger todos los paquetes adecuados, ya que lo que busca describir la memoria en cuestión es el proceso de resolución de un problema planteado por un cliente.

## 5.2 Desarrollo Hardware

Una vez ya ha sido instalado el Sistema operativo, se migrara todo el trabajo a la plataforma Windows, donde se hará uso de la herramienta para desarrolladores proporcionada por Xilinx, Vivado Design Suite. En esta herramienta, trabajaremos colocando bloques los cuales, algunos tienen funciones ya definidas y otros sus funciones pueden ser personalizadas por el usuario, y cableando dichos bloques entre ellos para conseguir una funcionalidad final acorde con el problema a paliar.

En el caso de esto proyecto, la intención es realizar un “analizador de espectros”. Este sistema, mediante los módulos que se citan a continuación, se encargará de recibir datos por el XADC los cuales son capturados por medio de un acelerómetro. Estos datos se dejaran en un región de memoria por si el cliente precisa de ellos para realizar análisis, pero también estos datos pasaran al siguiente modulo, el cual se encargara de realizar una FFT (Fast Fourier Transform) a dichos datos. Con esta FFT se conseguirá ver las vibraciones que ocurren en la maquina debido a los diferentes procesos por los que pasa (fresado, movimiento de paletas, adiamantado) sabiendo así el posible estado de la herramienta que se está empleando y si esta precisa de un cambio o no. Este posible cambio de herramienta se verá reflejado en el espectro de vibraciones, ya que de un espectro conocido, se pasará a un espectro que no tiene nada que ver con el original, apareciendo en este nuevos armónicos así como posibles productos de intermodulación.

### 5.2.1 Zynq 7000

Este bloque representará el sistema central de procesamiento, el procesador. Todas las operaciones y todos los módulos que se añadan al proyecto deberán tener una correlación con el mismo.

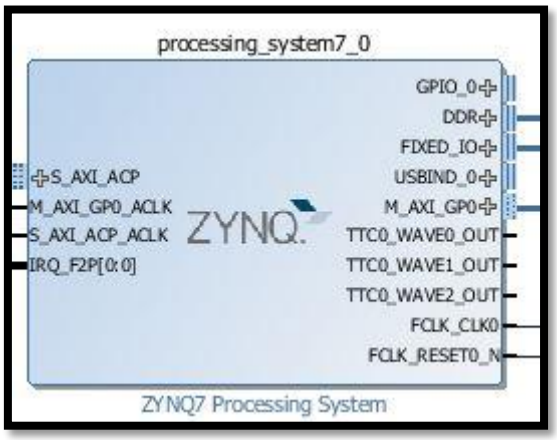



FIGURA 9 - BLOQUE ZYNQ

Realizando doble el click sobre el modulo se accede a la pantalla que se muestra en la imagen siguiente, su diseño de bloques.

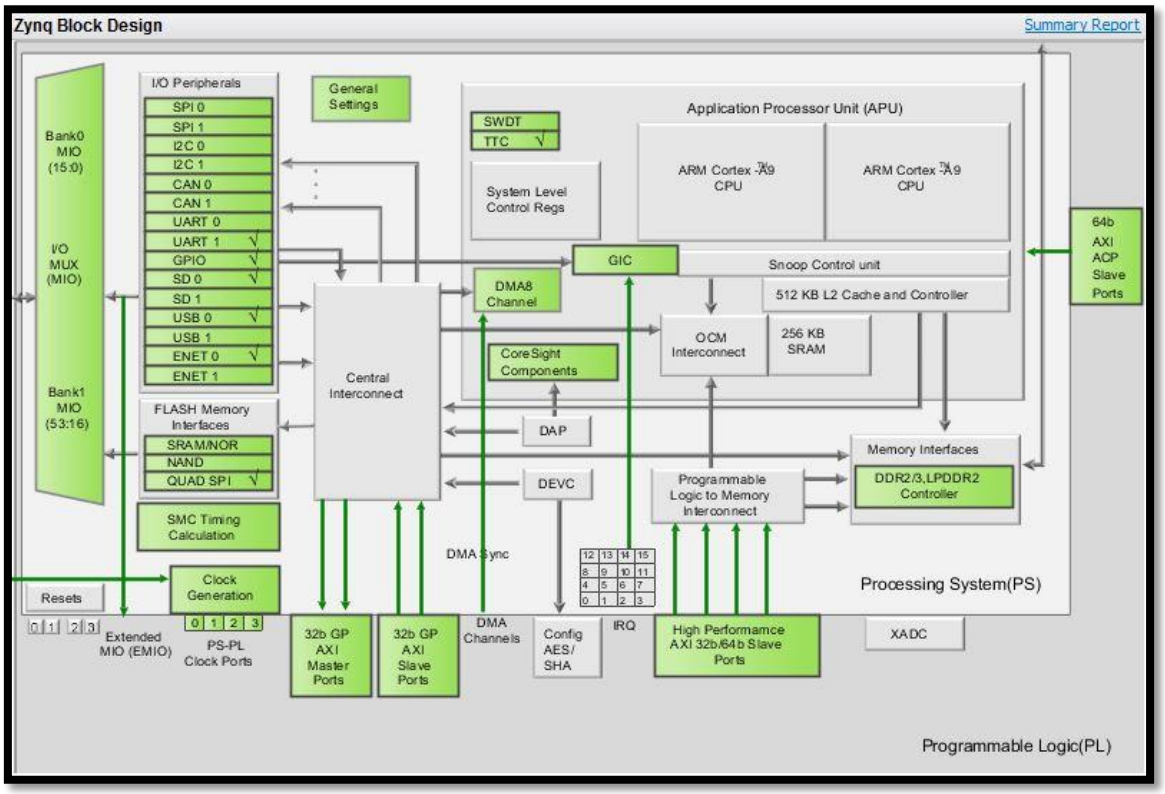



FIGURA 10 - DIAGRAMA DE BLOQUES ZYNQ

Como se observa en este diseño de bloques, todas las partes que se encuentran marcadas en verde tienen conexión directa con el procesador.

Este procesador es completamente editable a partir de su diseño de bloques y todos los cambios que se realicen en el mismo se verán reflejados en su bloque original (imagen anterior a esta). Como se observa no es necesario tener activas todas las funciones que el procesador ofrece, por lo menos para este diseño en concreto. Como funciones más importantes a citar a partir de las que se encuentran activas en la imagen podemos nombrar:

- **I/O**: Será necesaria tener activa toda la capa de I/O que ofrece el sistema, a partir de la I/O cape, por si se busca introducir por las mismas alguna señal o por otra parte por si se busca extraer un cierto dato a través de estos pines.
- **UART1**: Será estrictamente necesario definir un USB tipo UART el cual configuraremos con una velocidad de 115200 Baudios, una longitud de 8 bits y el bit de paridad a cero. Mediante esta definición de USB se conseguirá realizar comunicaciones con la unidad central de procesamiento (CPU) ya sea mediante un simulador de terminal o por la propia consola de Linux.
- **USB0**: Este dispositivo definirá un USB normal y corriente por si existiese una necesidad futura de conectar un ratón o un teclado a la placa en cuestión para utilizarla como si de un pc se tratase.
- **ENET0**: Como la placa únicamente dispone de un puerto Ethernet no habrá necesidad de definir ninguno más. Este puerto es incluido para dotar a la propia placa de conexión a la red o por si se necesita acceder a la misma haciendo uso del protocolo SSH.
- **Clock Generation**: Se utilizará para crear una serie de relojes con los que trabajaran a la par tanto memorias como procesador o como terceros bloques.
- **Memory Interfaces (DDR2/3)**: Incluido debido a que tendremos la necesidad de emplear bloques de RAM en el sistema para almacenar datos.
- **64b AXI ACP Slave Port**: Como será necesario realizar comunicaciones de datos entre PL y PS este puerto se encargará de que dichas comunicaciones dispongan de una coherencia en cuanto a datos se refiere.

Estos serían algunos de los interfaces más importantes en cuanto al bloque Zynq se refiere. Es necesario citar que este bloque es imprescindible, pues si no estuviese no sería posible el diseño por falta de comunicación y de coherencia entre PS y PL.

### 5.2.2 Clck Wizard

Partiendo del propio reloj interno del Zynq (velocidad del procesador) conectándolo a este bloque conseguiremos una gran variedad de señales de reloj definidas por el usuario. En este proyecto en concreto se hacen uso de dos señales de reloj: 75 Mhz y 150Mhz. Prácticamente todo el sistema se regirá por la señal de reloj de 75 Mhz excepto el bloque de procesamiento de la FFT el cual funcionará el doble de rápido 150 Mhz. El motivo por el cual el desarrollador ha escogido esta forma de diseñar ha sido porque es necesario que la FFT este procesada una vez lleguen nuevos datos. Si la relación de los relojes hubiese sido al contrario, el dato

capturado habría llegado al bloque FFT el doble de rápido que esta puede procesar, haciendo que el sistema saturase a la entrada de dicho bloque e impidiendo que el dato sea mostrado en tiempo real en una aplicación futura.

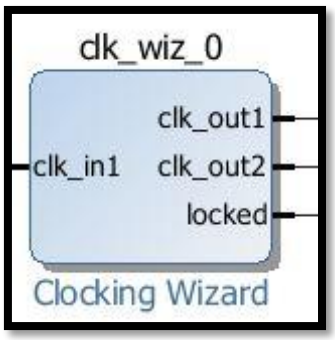



FIGURA 11 - BLOQUE CLK

La entrada del reloj será de 667.66 MHz y sus dos salidas serán de clk_1 75 Mhz y clk_2 150 Mhz.

### 5.2.3 Axi Interconnect

El bloque Axi interconnect, como su propio nombre indica, se trata de un bloque de interconexión del sistema. Este bloque será empleado para interconectar los distintos elementos del diseño. En concreto conectará los **puertos master** y los **puertos slave** con sus respectivos **puertos master** y **puertos slave** de otros bloques sincronizando a su vez sus velocidades de reloj respectivas.

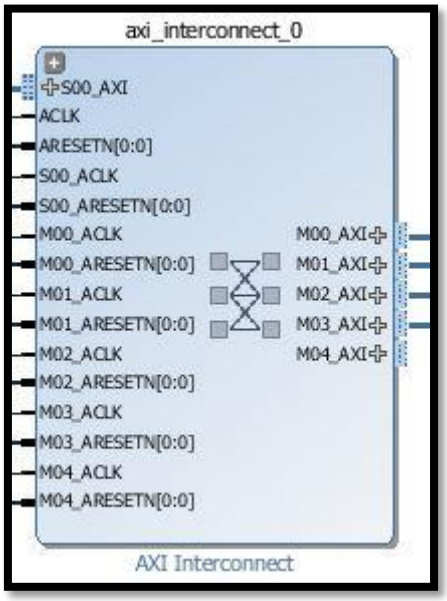



FIGURA 12 - BLOQUE AXI INTERCONNECT

### 5.2.4 System Processor Reset

Al igual que ocurre con el bloque de reloj, este módulo se alimentará de un reset central el cual viene dado por el procesador. A partir de este reset base se obtendrán dos sub-resets los cuales serán utilizados para: uno de ellos resetear el módulo de interconexión que se muestra arriba y el otro para resetear cualquier otro modulo periférico, ya sean módulos dentro de la jerarquía u **otros módulos totalmente independientes del axi interconnect** central del diseño.

### 5.2.5 Jerarquía FFT

Llegados a este punto se describe uno de los bloques más importantes del sistema, el bloque de procesamiento de la FFT. Para crear este bloque ha sido utilizado un diseño tipo jerárquico, en el cual se ha descrito como un bloque superior formado de otros bloques menores en su interior (esto se entenderá mejor al ver las imágenes posteriores).

El modulo diseñado por el desarrollador con sus entradas y sus salidas es el que se muestra a continuación:

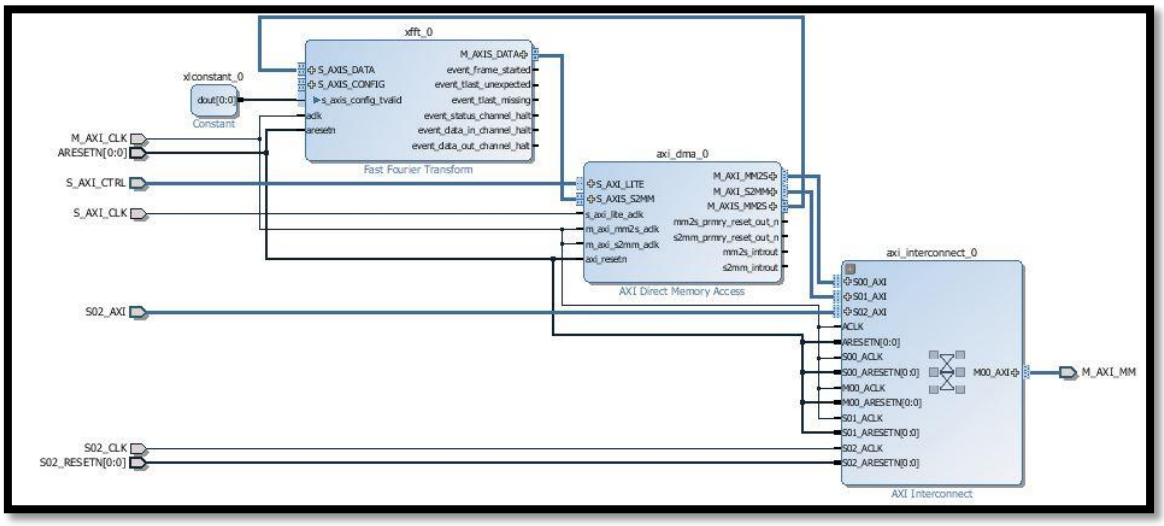



FIGURA 13 - JERARQUÍA FFT

Estos 4 bloques formarían el diseño jerárquico del bloque de procesamiento. Se pueden observar distintas entradas de reloj así como distintas entradas de reset, pero lo que realmente es importante recalcar es la funcionalidad de cada uno de los 4 bloques que se muestran:

- **Bloque Constant:** Como su nombre indica se trata de una constante. El bloque FFT necesitaba la definición de una serie de constantes a su entrada, las cuales al no ser controladas de forma externa por un usuario se fijan mediante la utilización de este bloque constante.
- **Bloque Fast Fourier Transform:** Se trata de uno de los bloques más importantes del sistema. En este se llevará a cabo todo el procesado a partir del dato en bruto que se reciba a su entrada. Este bloque permite una configuración completa en cuanto a FFT

se refiere. Permite realizar FFTs de distintos tamaños así como la posibilidad de no utilizar un único canal sino hasta 4 canales simultáneos. Además de esto permite modificar el modo en que se transferirán los datos a posteriori pudiendo elegir transferencias tipo FIFO o transferencias tipo stream.

Para este diseño en concreto se ha utilizado una configuración de una FFT de **1024 puntos** a **1** solo **canal**. El porqué de esto es debido a que no es necesaria una mayor resolución para tratar de ver las vibraciones que le importan al cliente. El motivo de elegir un solo canal, es debido a que el sistema, tal y como se encuentra en la actualidad, solo dispone de una entrada de datos (XADC) por lo tanto definir un segundo canal sin tener un segundo módulo de captura sería igual a ralentizar el programa. Como último aspecto se ha elegido una transferencia tipo **stream** para que el dato sea escrito o mostrado lo más rápidamente posible.

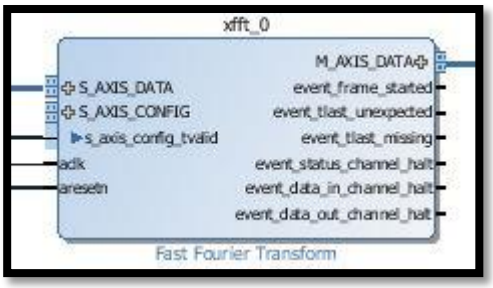



FIGURA 14 - BLOQUE FFT

- **Bloque DMA:** El bloque DMA sería el segundo bloque importante del sistema. En las direcciones que se asignen a este bloque, las cuales pueden ser modificadas al antojo del desarrollador en su totalidad (direccionamiento y tamaño), se encontrara el dato procesado por el bloque anterior (FFT). Estas DMA deberán ser accedidas empleando métodos tales como la programación de código en C por parte del desarrollador desde programas exteriores a Vivado para poder disponer de los datos requeridos. Esto se explicará en mayor profundidad en apartados posteriores.

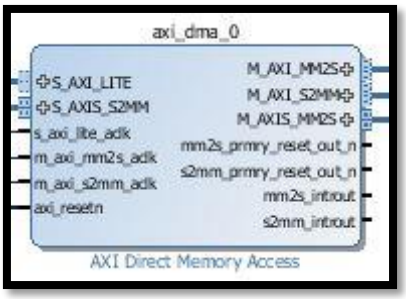



FIGURA 15 - BLOQUE AXI DMA

Para este diseño se ha elegido la dirección de inicio para el DMA de la FFT 0x4040_0000 con una profundidad (longitud) de 64 Kb llegando a la dirección 0x4040_FFFF.

- **Bloque Axi Interconnect:** Tiene la misma función que el principal solo que dentro de la jerarquía, interconectar.

## 5.2.6 Jerarquía XADC

El segundo diseño importante del módulo global es la jerarquía XADC la cual será la encargada de la captura de los datos que provendrán del acelerómetro adosado a la máquina. Su funcionalidad será básicamente recoger datos del exterior para nutrir a los módulos que se encuentran posterior a él.

El modulo creado por el desarrollador se muestra a continuación junto con sus entradas, salidas y futuras mejoras ya adosadas:

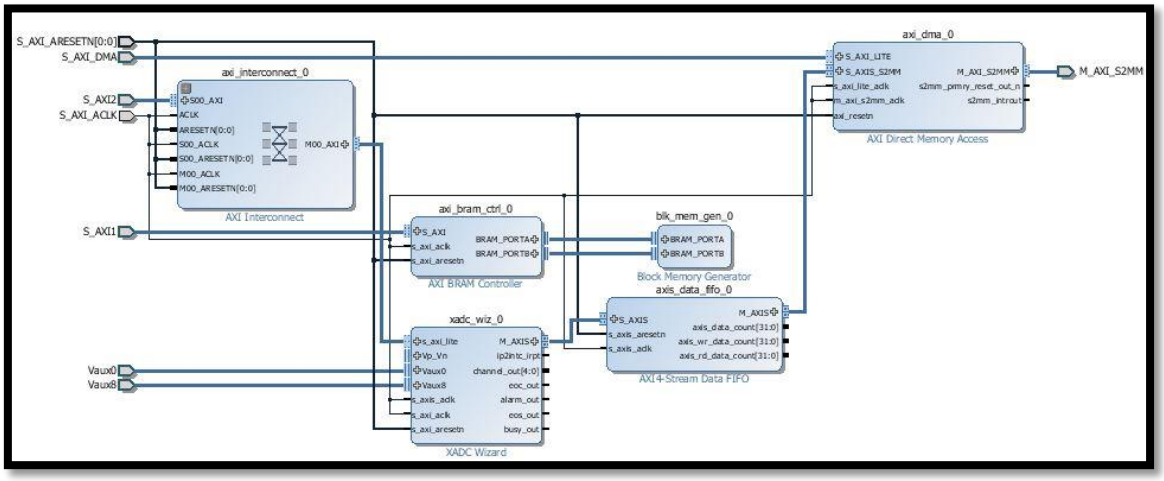



FIGURA 16 - JERARQUÍA XADC

Los 6 bloques que se ven conformarían el sistema de captura diseñado. De ellos podemos citar las siguientes características:

- **Bloque XADC_Wizard**: Como su nombre indica se trata de un configurador de XADCs. Al igual que el bloque FFT permite configurar en su totalidad cómo será la captura. Además de todo esto, es capaz de mandar alarmas al usuario, siempre que estén activas en su configuración. Estas alarmas son tratadas como flags por el resto del diseño y pueden ser enviadas al exterior, para el usuario, mediante cualquier tipo de GPIO.
- **Bloque AXI4 Stream Data FIFO**: El bloque XADC dispone de una configuración para una transferencia tipo FIFO al bloque posterior, pero por su simplicidad y sus pocas opciones de configuración, se ha optado por la inclusión de una FIFO en modo stream para alimentar al dispositivo DMA que se encuentra posterior a la misma.
- **Bloque AXI DMA**: Tiene la misma función que ha sido citada en el DMA del módulo FFT. A su funcionalidad igualmente se entrara en mayor profundidad en apartados posteriores.
- **Pines Vaux0 y Vaux8**: Estos pines son los que provendrán del acelerómetro y donde se conectará de forma exterior el mismo. En la configuración del XADC Wizard estos pines se han asignado al pin 1 y al pin 2 respectivamente del XADC exterior (Ver anexo Pines XADC). Los pines de entrada del XADC de esta placa únicamente soportan tensión y en

un rango en concreto, por lo que la entrada que se le debería ofrecer a este XADC debería ser de **0V a 1.8V** para un funcionamiento correcto.

- **Axi Interconnect**: La función de este bloque sería similar a los dos interconnect citados anteriormente, interconectar.

A parte de todos los bloques citados anteriormente, aparecen dos bloques más que serán empleados para futuras mejoras concertadas entre la empresa y el desarrollador. Estos bloques se tratan de memorias RAM, las cuales jugaran en un futuro un papel muy destacado en cuanto al diseño se refiere.

### 5.2.7 Diseño Global

Después de describir todos y cada uno de los bloques, el diseño global quedará como se muestra a continuación:

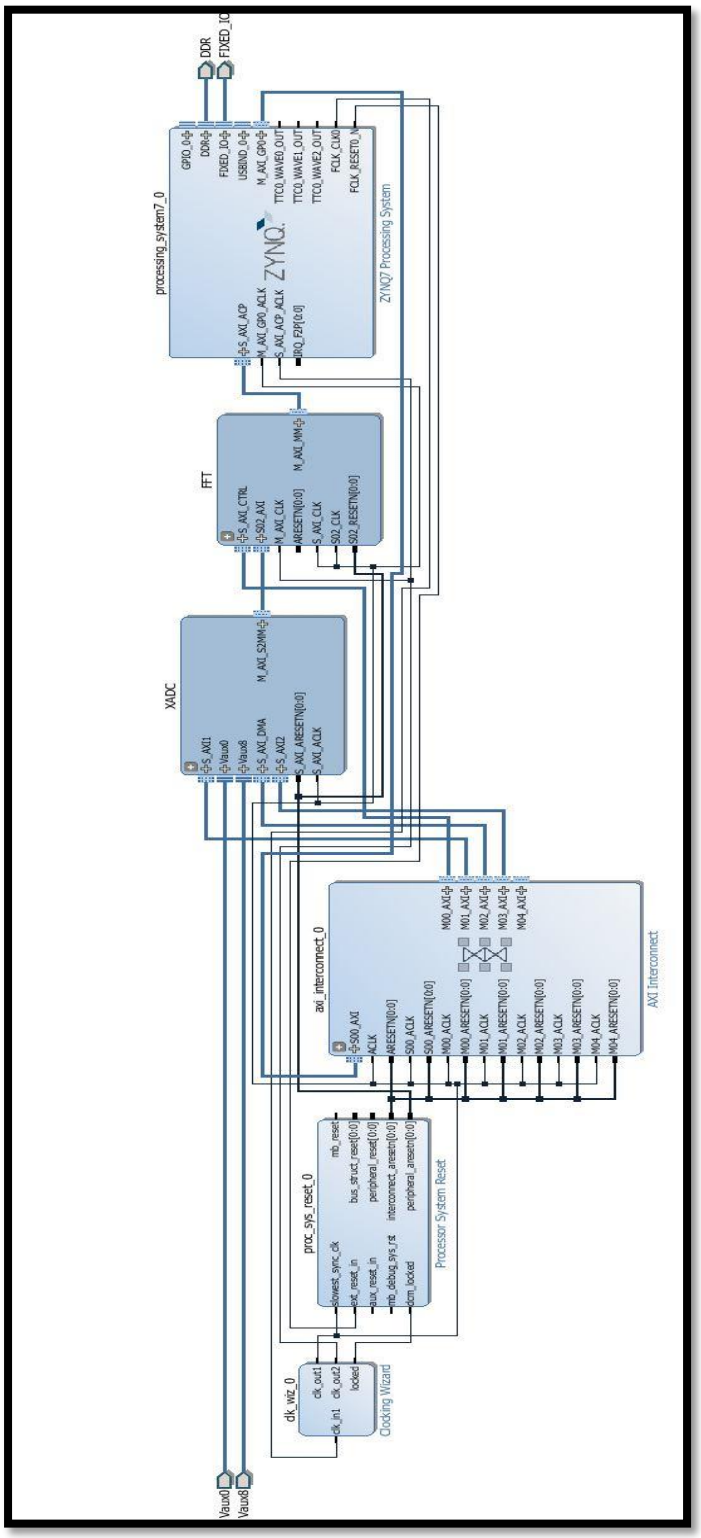



FIGURA 17 - DISEÑO GLOBAL

Si se observa el diseño en profundidad se puede ver que aparecen todos los bloques explicados anteriormente.

Como paso posterior a este diseño será necesario generar un bitstream el cual será cargado desde Linux directamente sobre la FPGA. Previo a realizar dicho bitstream, es necesario que

Vivado realice una síntesis y una implementación del diseño. Ambas se generan si todo esta correcto automáticamente tras ejecutar la orden de generar el bitstream.

Después del compilado del diagrama de bloques, si todo el proceso ha sido satisfactorio, Vivado abre dos vistas: una vista del mapeado del dispositivo y una vista del pin planner del mismo. En la vista del mapeado se podrá ver como ha quedado el diseño realizado sobre la FPGA. En la vista del pin planner ser verán todos y cada uno de los pines que han sido utilizados y su colocación. En esta vista se deberá realizar dos inclusiones y son las referidas a los dos pines que fueron escogidos como entradas del XADC. Para colocarlos dentro del mapa bastara con asignarles dos pines que se encuentren libres. En este caso el desarrollador ha escogido los pines: B20 y C20 para Vaux0 y A20 y B19 para Vaux8.

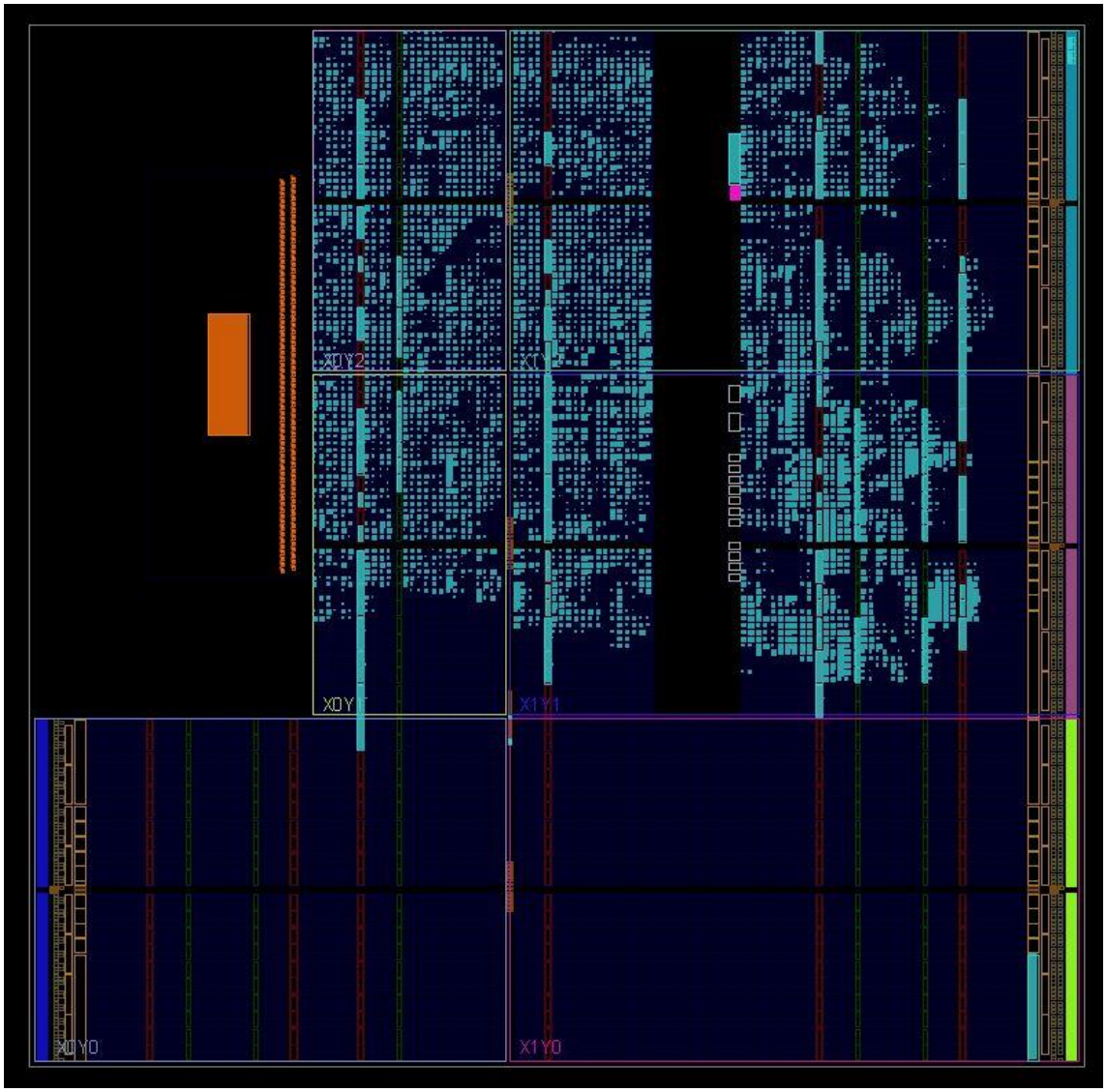



FIGURA 18 - FPGA MAPEADA

FIGURA 19 - PIN PLANNER

## 5.3 Desarrollo Software

Como parte posterior al desarrollo en hardware, estando este terminado por completo, es necesario realizar una programación software. Esta programación consistirá en el acceso a las memorias que se han citado anteriormente (**DMA**) para extraer los datos recogidos por la parte software.

### 5.3.1 Acceso a las Zonas de Memoria

Muchos controladores, sobre todo los de dispositivos por bloques, manejan el acceso directo a memoria o DMA. Para explicar el funcionamiento del DMA, es necesario conocer como ocurren las lecturas de disco cuando no se usa el mismo DMA. En primer lugar el controlador lee el bloque de la unidad en serie, bit a bit, hasta que todo el bloque está en el buffer interno del controlador. A continuación, el controlador calcula la suma de la verificación para comprobar que no ocurrieron errores de lectura, y luego causa una interrupción. Cuando el sistema operativo comienza a ejecutarse, puede leer el bloque del disco del buffer del controlador byte por byte o palabra por palabra, ejecutando un ciclo, leyéndose en cada iteración un byte o una palabra de un registro del controlador y almacenándose en la memoria.

Naturalmente, un ciclo de CPU programado para leer los bytes del controlador uno por uno desperdicia tiempo de CPU.

Por todo lo citado anteriormente, surge el DMA, el cual fue inventado para liberar a la CPU de este trabajo de bajo nivel. Cuando se usa el DMA, la CPU proporciona al controlador dos elementos de información, además de la dirección en disco del bloque: la dirección de memoria donde debe colocarse el bloque y el número de bytes que deben transferirse, como se muestra en la figura siguiente.

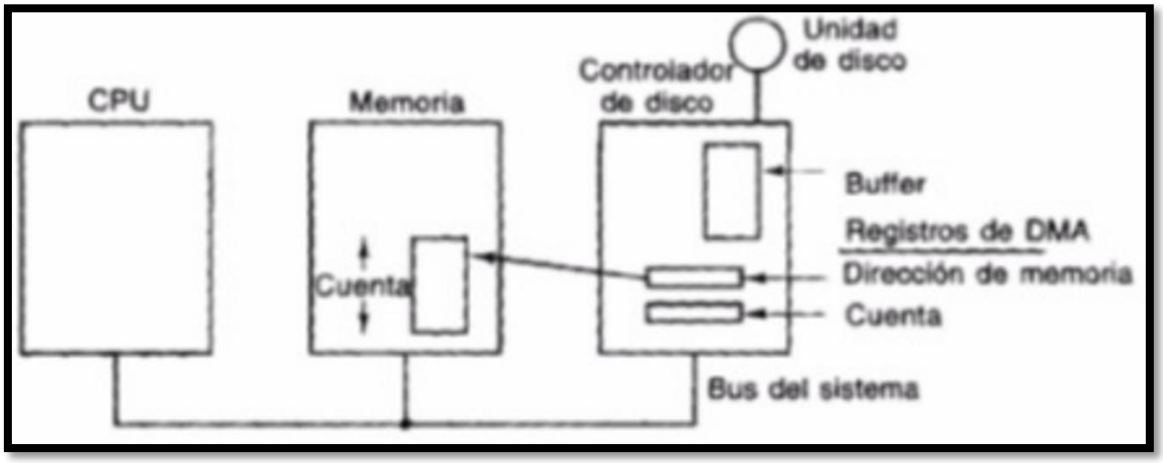



FIGURA 20 – ACCESOS A MEMORIA

Una vez que el controlador ha leído todo el bloque del dispositivo, lo ha colocado en su buffer y ha calculado la suma de verificación, copia el primer byte o palabra en la memoria principal en la dirección especificada por la dirección de memoria del DMA. Luego, el controlador incrementa la dirección de DMA y decrementa la cuenta de DMA en el número de bytes que se acaban de transferir. Este proceso se repite hasta que la cuenta de DMA es cero, y en ese momento el controlador causa una interrupción. Cuando el sistema operativo inicia, no tiene que copiar el bloque en la memoria; ya está ahí.

### 5.3.1.1 Desarrollo del Programa de Acceso

El programa de acceso a dichas memorias ha sido descrito mediante un código C el cual puede ser encontrado en los anexos que se citan al final de esta memoria (Ver anexo acceso a memorias). Básicamente en este código se describen una serie de funciones entre las cuales destacan:

- **Función FATAL**: Esta función se lanzará siempre que ocurra un fallo en el acceso a memoria o en la apertura de los dispositivos correspondientes, ya sea porque la dirección a la cual se quiere acceder no existe o por el contrario el dispositivo que se busca tampoco existe.
- **Función MAIN:** Sera la función principal descrita en este código. Dentro de la misma se ejecutaran el resto de funciones necesarias para realizar un acceso correcto a las memorias.
- **Funciones WRITE_SAMPLES y READ_SAMPLES:** Como su nombre indica, serán las encargadas respectivamente de leer y escribir la muestras recogidas por las memorias en el archivo .txt, el cual deberá ser tratado posteriormente.

- **Función INITIALIZE_AXI_DMA:** Esta función será la encargada de inicializar con su valor correspondiente todos los registros de memoria con los cuales se trabajará. Es la función más importante, se encarga de gestionar todo el proceso de acceso a memorias.
- **Función LOAD_MM2S y LOAD_S2MM:** Ambas funciones se encargaran de la salida y de la entrada respectivamente del dispositivo DMA, controlando su dirección y su anchura de bus para realizar una correcta transmisión/recepción de los datos.

Si se resume de forma breve el código descrito, en el cual se hace uso de todas las funciones anteriores, tendría una funcionalidad tal y como se describe a continuación.

En primer lugar se abrirá el dispositivo DMA el cual esta descrito por un dispositivo de Linux. Si esta parte falla se lanzara la función error, en cambio sí es satisfactoria, se lanzará la función **initialize_axi_dma.** A posteriori entrarán las funciones **load_s2mm** y **load_mm2s**, las cuales tomaran la función descrita anteriormente. Una vez ha sido realizado todo lo anterior y ha resultado satisfactorio, se lanzaran las funciones **write_samples** o **read_samples** según sea conveniente (**read_samples** para DMA del **FFT** y **write_samples/read_samples** para el DMA de XADC).

#### 5.3.1.2 Modificación del kernel de Linux

Para conseguir que todo dicho acceso a memoria funcione, es necesario modificar tanto el kernel del sistema Linux como su booteo.

En primer lugar, si se habla del sistema de booteo, es necesario modificar el devicetree que posee la placa por defecto en el cual se cita, como su nombre indica, el árbol de dispositivos que la placa iniciará una vez sea conectada. En dicho devicetree será necesario incluir una definición de las memorias DMA, tal y como se cita en el anexo, ya que estas no se encuentran en el árbol de dispositivos por defecto que posee el sistema de booteo de fábrica. Estos DMA serán definidos como “generic-uio”, es decir, entradas/salidas genéricas de usuario. Junto con esta definición serán incluidas tanto su dirección de inicio como su longitud en memoria, y también una etiqueta (en este caso serán **uio0** y **uio1**) todo ello se observa en el anexo citado anteriormente.

En segundo lugar, será necesario modificar el kernel de la placa para permitir que las nuevas definiciones incluidas en el devicetree surtan efecto. Esto se conseguirá accediendo al directorio /usr/src desde un terminal remoto (**PuTTy**). Tras acceder a este directorio se deberá entrar en la carpeta kernels y sobre el kernel correspondiente del sistema ejecutar la orden: **make menuconfig**. Si todo funciona correctamente, se abrirá una interfaz gráfica tipo BIOS en la cual se accederá al apartado **Device Drivers**. Una vez dentro de device drivers entraremos al submenú **Userspace I/O Drivers** y dentro del mismo pondremos en modo activo el built-in.

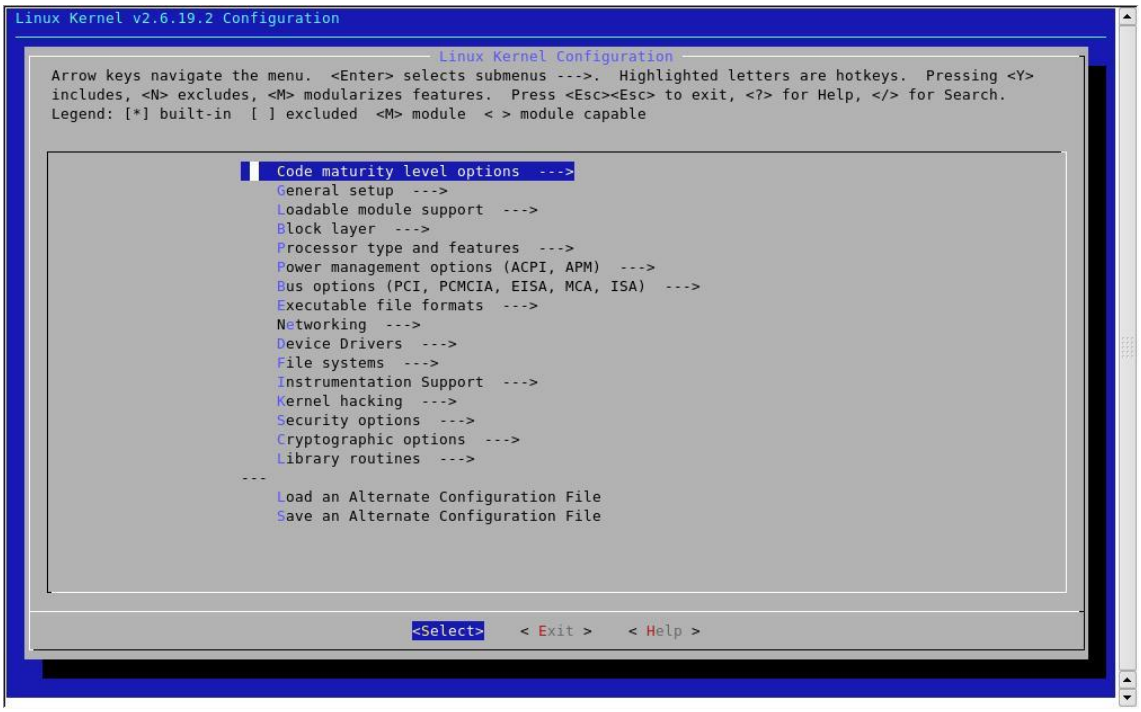



FIGURA 21- MENÚ KERNEL

Tras realizar estos pasos y para comprobar que han sido ejecutados de forma correcta, ejecutaremos la orden **$ grep UIO .config** en el terminal de Linux, la cual deberá arrojar las siguientes líneas si todo el proceso ha resultado en éxito:

CONFIG_UIO=y

CONFIG_UIO-PDRV=y

CONFIG_UIO_PDRV_GENIRQ=Y

Después de comprobar que todo ha sido realizado correctamente se realizara un reboot de la placa. Tras este podremos acceder al directorio **/dev** dentro del cual encontraremos los dos dispositivos DMA definidos por **uio0** y **uio1** a los cuales accederá el programa en C.

## 5.4 Desarrollo Shell Script

Después de todo el desarrollo que ha sido citado anteriormente era necesario el desarrollo de un Shell script de Linux el cual fuese capaz de modificar la secuencia del boot del dispositivo hardware a antojo del desarrollador.

### 5.4.1 Descripción del Shell Script

Como se puede apreciar en su nombre, el término Shell Script proviene de los dos términos que lo componen:

- **Shell**: Se trata del término empleado en informática para referirse a un intérprete de comandos, el cual consistiría en una interfaz de usuario tradicional de un sistema operativo basado en Unix. Mediante las instrucciones que se aportarían mediante el intérprete, el usuario es capaz de comunicarse con el núcleo y ejecutar dichas órdenes que le permiten controlar el funcionamiento de la computadora.
- **Script**: Se trata de un archivo de órdenes. Es un programa relativamente sencillo el cual se almacena en un fichero de texto plano.

### 5.4.2 Programación del Shell Script mediante Bash

El Shell Script propuesto por el desarrollador, consiste en una serie de órdenes que se ejecutarán una vez el sistema hardware sea iniciado y siempre tras la lectura del kernel hecha por el mismo, de la manera contraria el Script será leído pero no ejecutado correctamente llevando el sistema a un iniciado de forma casual (por defecto).

En este Script serán escritas dos órdenes importantes dentro del sistema propuesto. Estas órdenes serán descritas mediante un fichero de texto plano generado mediante un editor de textos (**nano** en este caso). Después de realizar la escritura del fichero, será necesario también dotar al mismo de permisos de ejecución e introducirlo en la estructura de ejecución que posee el sistema hardware. Las órdenes a ejecutar por el script serán:

- **Carga de la FPGA**: En dicha función el sistema cargara el archivo .bit que genera Vivado Design Suite en el dispositivo FPGA dotando al sistema hardware de su funcionalidad más importante.
- **Carga del programa encargado de acceder a memorias**: Después y siempre después de haber cargado la FPGA, se procederá con esta orden. La misma se encargara del acceso a cada uno de los DMA que posee el programa creado, mapeando dichas direcciones de RAM a una serie de direcciones virtuales que permitan acceder al dato almacenado.

Conocidas las órdenes a ejecutar y su prioridad, el script será descrito mediante una estructura tipo C separando la carga de FPGA y la carga de acceso en dos funciones distintas, las cuales verán su ejecución controlada dentro de una tercera función principal main ( Ver anexo Shell Script).

Con este Shell script se conseguirá paliar uno de los problemas más importantes que a día de hoy posee el hardware: si se pierde la alimentación en el mismo o este sufre una caída de tensión, el aparato al volver a arrancar, realizará el arranque predeterminado el cual le ha sido dotado en fabrica, perdiendo así por completo su funcionalidad.

## 5.5 Montaje del diseño final

Como se ha citado en los objetivos al principio de esta memoria, el sistema desarrollado será colocado en el armario de los reguladores de los motores de la operación OP35A. A la placa le será conectado por el XADC (respetando los pines) un acelerómetro que está siendo usado por

el equipo de predictivo y se dejará un mes capturando datos. El montaje del dispositivo se muestra a continuación en varias imágenes:

FIGURA 22 - MONTAJE ARMARIO 1

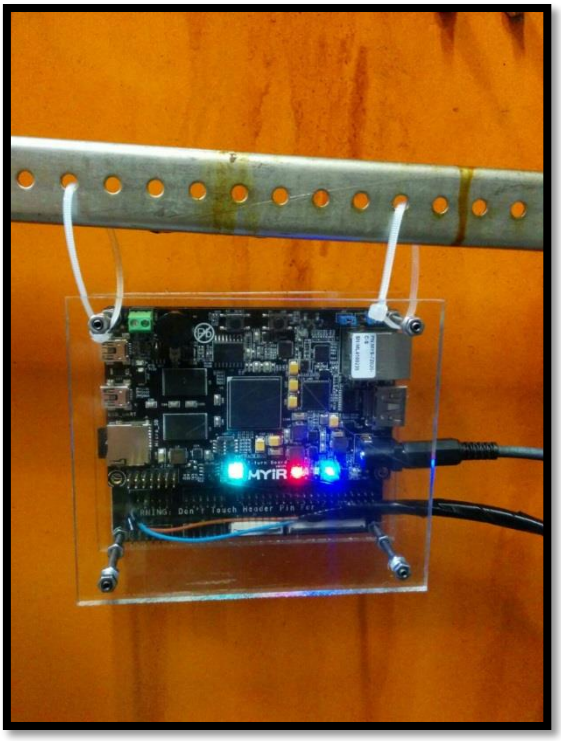



FIGURA 23- MONTAJE ARMARIO 2

FIGURA 24 - MONTAJE ARMARIO 3

## 5.6 Resultados

A continuación se muestran resultados reales tomados en máquina para demostrar el correcto funcionamiento del sistema entre las fechas de 27/06/2016 y 30/06/2016.

La máquina en cuestión, sufrió una avería a fecha de 28/06/2016 reflejándose esta en los resultados arrojados por el sistema de captura

Las gráficas que se muestran, han sido analizadas por ingenieros mecánicos de Ford España S.L, los cuales se han encargado de una correcta representación de las mismas en cuanto a escalas, valores y ejes se refiere, llegando a las siguientes conclusiones:

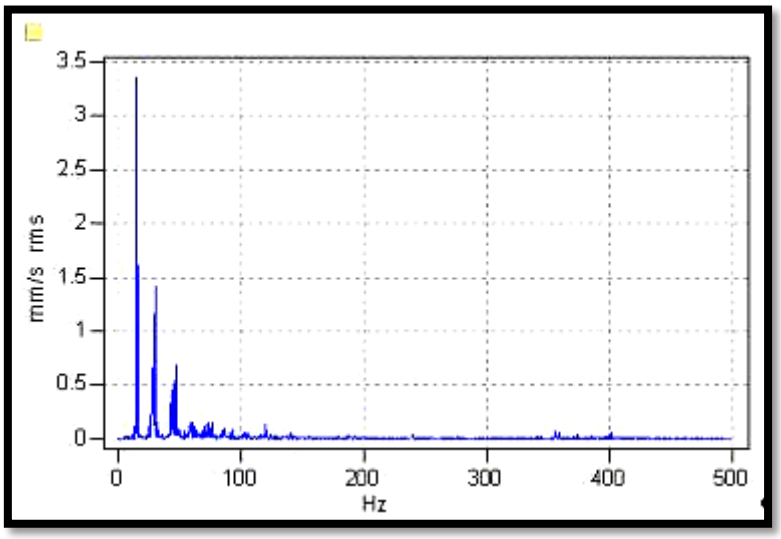



FIGURA 25 - MAQUINA EN ESTADO NORMAL

En esta figura se observa un espectro de vibraciones característico de la no existencia de fallos.

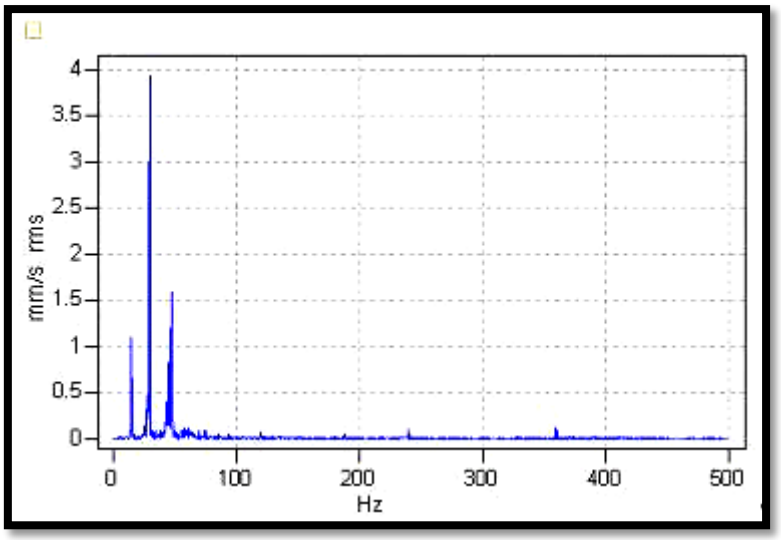



FIGURA 26 - MAQUINA FALLANDO

Este segundo espectro, a razón de ver de los ingenieros especializados en mecatrónica de Ford, se evidencian problemas de desalineación causados por defectos en los porta escobillas. Los niveles de vibración de alrededor de 4 mm/s se encontrarían dentro de los límites admisibles según la normativa interna. Pese a que estos límites se encuentran dentro de la normativa interna, la maquina no estaría realizando correctamente su función en uno de los ejes y por consecuencia, la camisa del bloque de motor tendría unas posibilidades superiores de salir de la misma operación con algún defecto de fabricación.

# 6 Pliego de Condiciones

## 6.1 Requisitos de Hardware

Como requisitos de Hardware se encuentran los siguientes:

- Ordenador Portatil Inspiron 15 Serie 7000
  - Intel Core i7 6ª generación
  - Memoria RAM 16 GB
  - Disco Duro 1 TB
  - SSD 128 GB
  - Windows 10
- Tarjeta de desarrollo Z-turnBoard (o de su misma familia)
  - Zynq-7Z020C-CLG400 o superior
  - FPGA 85000 Logic Cells o superior
- I/O Cape Z-turnBoard
- Cable Ethernet
- Cable USB to UART (Proporcionado por el fabricante).
- Cable de Alimentación (Proporcionado por el fabricante).

## 6.2 Requisitos Software

- Xilinx Developer Pack
  - Vivado Design Suite 2015.3
  - Xilinx SDK
  - Vivado HLS
- Putty o TeraTerm
- WinSCP
- Adobe Reader
- Editor de texto Notepad++
- Conexión a Internet

## 6.3 Presupuesto

En este apartado se realizará el desglose del presupuesto estimado para la realización de este proyecto. En él se tendrá en cuenta la mano de obra necesaria, equipos informáticos, licencias de programas y materiales entre otros gastos.

**Mano de Obra**

En este apartado se incluirán los gastos debidos a la mano de obra necesaria para realizar el proyecto. El tiempo necesario de desarrollo del proyecto ha sido de 3 meses (90 días)para la parte de diseño del sistema, que a una media de 6 horas diarias hacen un total de 540 horas. Por otra parte, la redacción del libro ha supuesto una duración de 30 días a una media de 3 horas diarias lo que hacen un total de 90 horas.

Para los costes de mano de obra se tomará como referencia el contrato que ha sido asignado al trabajador por parte de la empresa contratante. En este se refleja que el trabajador deberá

ganar 420 € al mes, con lo que, si dividimos esta cifra entre días del mes y hora trabajada, el trabajador contratado cobrará una media de 2,33€/hora.

| Tarea | Horas | Salario (Euros/Hora) | Total |
|---|---|---|---|
| Diseño del Sistema | 540 | 2.333 | 1258,2 € |
| Redacción del libro | 90 | 2.33 | 209,7 € |
| **TOTAL** | | | 1467,9 € |

TABLA 3 - MANO DE OBRA

**Recursos Hardware**

En este apartado se incluirá el equipo utilizado para el desarrollo del proyecto, así como la tarjeta de desarrollo y materiales utilizados en las pruebas realizadas del sistema que ha sido diseñado.

| Material | Precio | Amortización | Tiempo de uso | Subtotal |
|---|---|---|---|---|
| Ordenador Portátil Dell | 1250 € | 3 años | 4 meses | 173,61€ |
| Z-turnBoard (incluye cables) | 100 € | - | 4 meses | 100,00€ |
| I/O Cape | 35 € | - | 4 meses | 35,00€ |
| **TOTAL** | | | | 308,61€ |

TABLA 4 - RECURSOS HARDWARE

**Recursos Software**

Algunos de los programas empleados requieren de licencias para el uso completo de todas las herramientas del programa o de algunos componentes en concreto. Los gastos asociados a las licencias de los programas se muestran a continuación(los programas que no aparecen en la tabla es debido a que son libres/gratuitos):

| Programa | Precio | Amortización | Tiempo de uso | Subtotal |
|---|---|---|---|---|
| Vivado Design Suite | 2647 € | 2 años | 5 meses | 551.45 € |
| **TOTAL** | | | | 551.45 € |

TABLA 5 - RECURSOS SOFTWARE

**Impresión y encuadernación de los libros**

En este apartado se incluirán todos los materiales necesarios para la redacción, impresión y encuadernación de los libros:

| Concepto | Precio |
|---|---|
| Impresión | 30,00€ |
| Encuadernación | 60,00€ |
| **TOTAL** | 90,00€ |

TABLA 6 - RECURSOS IMPRESIÓN/ENCUADERNACIÓN

**Coste Total**

| Concepto | Precio |
|---|---|
| Mano de Obra | 1467,90€ |
| Recursos Hardware | 308,61€ |
| Recursos Software | 551,40€ |
| Impresión/Encuadernación | 90,00€ |
| **TOTAL** | 2417,91 € |

TABLA 7 - RECURSOS TOTALES

# 7 Trabajos Futuros y Conclusiones

## 7.1 Conclusiones

Es necesario citar que el punto más importante de este trabajo y de cualquier otro que se realice, es la documentación y la preparación previa del mismo. En este caso la revisión de esta documentación 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 sospesando las ventajas e inconvenientes de cada uno ha hecho que se llegue a encontrar la solución más adecuada, en este caso, Xillinux.

Esta distribución, la cual está acompañada de la solución Xillybus hará que la comunicación entre FPGA y procesador sea muy intuitiva y fácil de implementar en una mejora futura que se va a realizar del mismo por parte del desarrollador.

Para darle uso al codiseño se ha llevado a cabo el desarrollo de una aplicación la cual gira entorno a una FFT. Esta aplicación ha sido diseñada con la intención de realizar un mantenimiento predictivo y una prognosis de las máquinas de la línea, se trata de un módulo capaz de capturar las distintas vibraciones que aparecen durante el funcionamiento de las mismas maquinas. Ha sido un desarrollo que combinado muchas capas de trabajo, desde la electrónica hasta la programación de módulos capaces de acceder a las memorias del Zynq para extraer los datos necesarios, pasando por el procesamiento de señales, Linux y otras herramientas de trabajo. Esto ha permitido desarrollar una visión amplia de varios campos de las telecomunicaciones, centrándose en mayor medida en el terreno de la electrónica, 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 como sistema de captura y además es posible mantener el dispositivo conectado a una red, se obtiene un sistema muy interesante para temas relacionados con IoT, un mercado que crecerá a pasos agigantados en los próximos años. En definitiva se trata de un dispositivo que agrupa tantas tecnologías en su interior que sus posibilidades son prácticamente ilimitadas.

## 7.2 Trabajos Futuros

Los caminos de mejora futuros se pueden ver desde dos puntos de vista distintos. Si se quiere continuar una aplicación 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, los cuales han sido ya hablados con la empresa contratante:

- Si se enfoca al codiseño HW/SW siguiendo la aplicación desarrollada para la industria:
    - Utilizar el módulo de Xillybus que proporciona Xillinux (el cual no ha sido utilizado en este proyecto). Con este módulo facilitaríamos en gran medida las comunicaciones entre FPGA y procesador ya que trataría a las memorias como

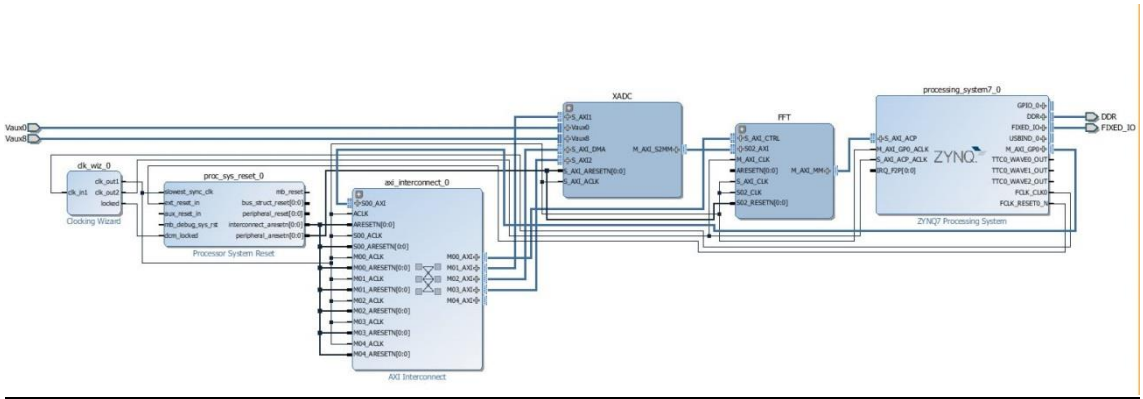
memorias FIFO y podríamos acceder a ellas fácilmente usando comandos proporcionados por las mismas desde por ejemplo un fichero C.

- Aumentar el número de canales de captura mediante extensiones PMOD. Esto se conseguiría conectando ADCs a cada una de las conexiones PMOD. Con esto se conseguiría capturar datos de hasta 4 máquinas distintas arreglo a las entradas de las que dispone la I/O Cape de la Z-turnBoard.
- Aumentar el número de canales de captura implica aumentar el número de canales de procesado del bloque de la FFT. Aumentar a 4 canales el procesado de datos mediante la FFT modificando el IP Core correspondiente del Diseño.
- Añadir al diseño en Vivado, Cores SPI para poder en un futuro adosar a la Z-turnBoard un disco duro externo el cual pueda almacenar programas o los mismos datos capturados por la placa.

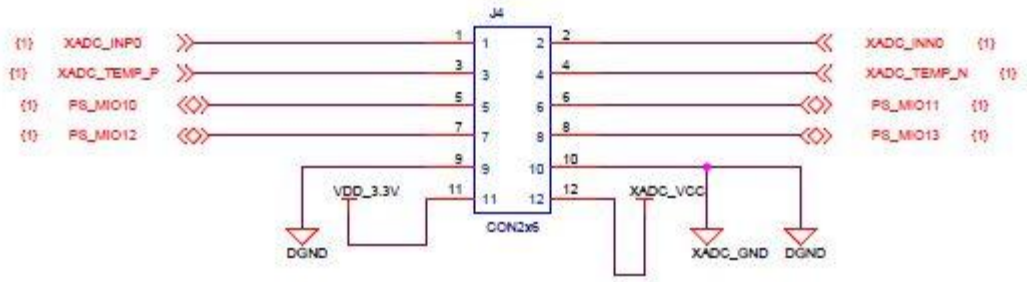
- Si se enfoca el diseño desde un punto de vista en cuanto Software:
  - Incluir ZeroMQ en nuestro diseño ya sea programado en Python o Labview. ZeroMQ se trata de un sistema de mensajería basado en colas con el cual se podrían comunicar fácilmente los datos que se encuentren en el entorno Linux con cualquier SO que difiera del mismo para una posterior inclusión de los mismos en un entorno grafico o en una base de datos.
  - Sustituir el programa C actual que realiza el acceso a memoria por el mismo tipo de programa pero esta vez programado en Python y usando el sistema de mensajería ZeroMQ programado dentro del mismo.
  - Realizar un entorno gráfico, para representar los datos que han sido extraídos del sistema embebido mediante el sistema de mensajería, con alguna herramienta/programa tipo Labview.
  - Realizar una API para una posible consulta acerca del estado de la maquina online mediante un PC o mediante un terminal móvil.

**Diagramas/Esquemáticos:**

- **Diagrama de bloques Global:**




- **Diagrama Pines XADC**




**Códigos:**

- **Código Acceso a Memorias C:**

  - **DMA FFT:**

```
//Programa que accedera a las posiciones de memoria virtual
mapeadas
//para mapearlas posteriormente en DDR
//Despues de esto extraeremos la informacion de dichas
posiciones
//para imprimirla por pantalla o almacenarla en base de datos
//o mostrar resultados en algun tipo de aplicacion grafica
(Labview o algun applet)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <fcntl.h>
#include <ctype.h>
#include <termios.h>
#include <sys/types.h>
#include <sys/mman.h>



void FATAL (void){
   printf("No se ejecutÃ³ correctamente. Revisar\n");
}

#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE-1)

#define BASE_ADDR 0x1000000
#define TX_BUFFER (BASE_ADDR + 0x001000)
#define RX_BUFFER (BASE_ADDR + 0x003000)

#define AXI_DMA_BASE_ADDR 0x40400000

#define MM2S_DMACR (AXI_DMA_BASE_ADDR + 0x00)  //CR = Control
Register
#define MM2S_DMASR (AXI_DMA_BASE_ADDR + 0x04)  //SR = Status
Register
#define MM2S_SA (AXI_DMA_BASE_ADDR + 0x18)     //SA = Start
Address
#define MM2S_LENGHT (AXI_DMA_BASE_ADDR + 0x28) //Longitud

#define S2MM_DMACR (AXI_DMA_BASE_ADDR + 0x30)  //CR = Control
Register
#define S2MM_DMASR (AXI_DMA_BASE_ADDR + 0x34)  //SR = Status
Register
#define S2MM_DA (AXI_DMA_BASE_ADDR + 0x48)     //DA =
Destination Adress
#define S2MM_LENGHT (AXI_DMA_BASE_ADDR + 0x58) //Longitud

int main(int argc, char * argv []){
   int fd;
   void *map_base, *virt_addr, *map_base_thread,
*virt_addr_thread;
   //unsigned long read_result, writeval;
   //off_t target;
   // int access_type
   fd = open("/dev/uio0", O_RDWR | O_SYNC); //Abrimos toda la
memoria fisica del Xilinx
   printf("%d",fd);
   if((fd = open("/dev/uio0",O_RDWR | O_SYNC)) == -1 ){
        FATAL();
   }
   printf ("/dev/uio0 opened successfully.\n");
   fflush (stdout);
```

```
   //Inicializamos el DMA
   initialize_axi_dma();


   //Cargamos s2mm y mm2s para transferencia
   load_s2mm();
   load_mm2s();

   //Escribimos el dato
   write_samples();
   //read_samples();

   close(fd);
   return 0;

}


void write_samples (void){
   int i = 0, k = 0;
   int fd;
   off_t sample_addr;
   unsigned char * buffer;
   unsigned char * myData;
   void *map_base;
   FILE *file;

   file = fopen("datos.txt","w");
   buffer = malloc (256*1024);
   for(i=0;i<65536*4;i++)
   {
              *(buffer + i) = *(myData + i);
              (file,*(myData + i));
   }
   fclose(file);

   map_base = mmap(NULL, 256*1024, PROT_READ | PROT_WRITE,
MAP_SHARED,fd,TX_BUFFER);
   memcpy (map_base,buffer,256*1024);


   // if(file == NULL){
        // printf("Error al abrir archivo\n");
   // }
   // else {
        // fprintf(file,)
   // }

   if (munmap (map_base,MAP_SIZE) == -1){
        FATAL();
   }
   free (buffer);
}
```

```
void read_samples (void){
   int i = 0, k = 0;
   int fd;
   void *map_base;
   off_t sample_addr;
   unsigned char * buffer;
   unsigned char * myData_out;
   unsigned long read_result=0;

   buffer = malloc (256*1024);
   map_base = mmap(NULL, 256*1024, PROT_READ | PROT_WRITE,
MAP_SHARED,fd,RX_BUFFER);
   memcpy (buffer , map_base, 256*1024);
   for (i=0;i<65536*4;i++){
        *(myData_out+i) = *(buffer+i);
   }

   if(munmap(map_base, MAP_SIZE) == -1){
        FATAL();
   }
   free (buffer);
}

void initialize_axi_dma(void){
   off_t sample_addr;
   unsigned long read_result = 0;
   void *map_base;
   int fd;
   unsigned long virt_addr = 0;

   //MM2S_DMASR

   sample_addr = MM2S_DMASR;
   map_base = mmap(0,MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, sample_addr & ~MAP_MASK);
   if (map_base == (void *)-1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long *) virt_addr) |= 0x10001;
   read_result = *((unsigned long *) virt_addr);

   printf("MM2S_DMASR set to: %lx\n",read_result);

   //MM2S_DMACR

   sample_addr = MM2S_DMACR;
   map_base = mmap(0,MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, sample_addr & ~MAP_MASK);
   printf("%lx",map_base);
   if (map_base == (void *) -1){
        FATAL();
   }
```

```
   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long *) virt_addr) |= 0x10001;
   read_result = *((unsigned long *) virt_addr);

   printf("MM2S_DMACR set to: %lx\n",read_result);

   //MM2S_DMASR

   sample_addr = MM2S_DMASR;
   map_base = mmap(0,MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, sample_addr & ~MAP_MASK);
   if (map_base == (void *)-1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long *) virt_addr) |= 0x10001;
   read_result = *((unsigned long *) virt_addr);

   printf("MM2S_DMASR set to: %lx\n",read_result);

   if(munmap(map_base,MAP_SIZE)== -1) {
        FATAL();
   }


   //S2MM_DMACR

   sample_addr = S2MM_DMACR;
   map_base = mmap (0,MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED,fd, sample_addr & ~MAP_MASK);
   if(map_base == (void* ) -1){
        FATAL();
   }

   virt_addr = map_base +(sample_addr & MAP_MASK);
   *((unsigned long*)virt_addr) |= 0x10001;
   read_result = *((unsigned long*)virt_addr);

   printf("S2MM_DMACR set to: %lx\n",read_result);

   if(munmap(map_base,MAP_SIZE) == -1) {
        FATAL();
   }

   //S2MM_DMASR

   sample_addr = S2MM_DMASR;
   map_base = mmap (0,MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED,fd, sample_addr & ~MAP_MASK);
   if(map_base == (void* ) -1){
        FATAL();
```

```
   }

   virt_addr = map_base +(sample_addr & MAP_MASK);
   *((unsigned long*)virt_addr) |= 0x10001;
   read_result = *((unsigned long*)virt_addr);

   printf("S2MM_DMASR set to: %lx\n",read_result);

   if(munmap(map_base,MAP_SIZE) == -1){
        FATAL();
   }
}

void load_mm2s (void) {
   off_t sample_addr;
   unsigned long read_result = 0;
   void *map_base;
   unsigned long virt_addr = 0;
   int fd;

   //MM2_SA
   sample_addr = MM2S_SA;
   map_base = mmap(0,MAP_SIZE,PROT_READ | PROT_WRITE,
MAP_SHARED,fd, sample_addr & ~MAP_MASK);
   if(map_base == (void*)-1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long*) virt_addr)= TX_BUFFER;
   read_result = *((unsigned long*) virt_addr);

   printf("MM2S_DA set to: %lx\n",read_result);

   if(munmap (map_base,MAP_SIZE) == -1){
        FATAL();
   }

   //MM2S_LENGHT

   sample_addr = MM2S_LENGHT;
   map_base = mmap(0,MAP_SIZE,PROT_READ | PROT_WRITE,
MAP_SHARED,fd, sample_addr & ~MAP_MASK);
   if(map_base == (void*)-1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long*) virt_addr)= 0x40000;
   read_result = *((unsigned long*) virt_addr);

   printf("MM2S_LENGHT set to: %lx\n",read_result);
```

```
   if(munmap (map_base,MAP_SIZE) == -1) {
        FATAL();
   }


}

void load_s2mm (void){
   off_t sample_addr;
   unsigned long read_result = 0;
   unsigned long virt_addr = 0;
   int fd;
   void *map_base;

   //S2MM_DA

   sample_addr = S2MM_DA;
   map_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, sample_addr & ~MAP_MASK);
   if(map_base == (void*) - 1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);
   *((unsigned long*) virt_addr) = RX_BUFFER;
   read_result = *((unsigned long*)virt_addr);

   printf("S2MM_DA set to: %lx\n",read_result);

   if(munmap(map_base,MAP_SIZE) == -1) {
        FATAL();
   }

   //S2MM_LENGHT

   sample_addr = S2MM_LENGHT;
   map_base = mmap(0,MAP_SIZE,PROT_READ | PROT_WRITE,
MAP_SHARED, fd, sample_addr & ~MAP_MASK);
   if(map_base == (void*)-1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long*) virt_addr)= 0x40000;
   read_result = *((unsigned long*) virt_addr);

   printf("S2MM_LENGHT set to: %lx\n",read_result);

   if(munmap (map_base,MAP_SIZE) == -1){
        FATAL();
   }
}
```

- **DMA XADC:**

```
//Programa que accedera a las posiciones de memoria virtual
mapeadas
//para mapearlas posteriormente en DDR
//Despues de esto extraeremos la informacion de dichas
posiciones
//para imprimirla por pantalla o almacenarla en base de datos
//o mostrar resultados en algun tipo de aplicacion grafica
(Labview o algun applet)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <fcntl.h>
#include <ctype.h>
#include <termios.h>
#include <sys/types.h>
#include <sys/mman.h>



void FATAL (void){
   printf("No se ejecutÃ³ correctamente. Revisar\n");
}

#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE-1)

#define BASE_ADDR 0x1000000
#define TX_BUFFER (BASE_ADDR + 0x001000)
#define RX_BUFFER (BASE_ADDR + 0x003000)

#define AXI_DMA_BASE_ADDR 0x40410000

#define MM2S_DMACR (AXI_DMA_BASE_ADDR + 0x00)  //CR = Control
Register
#define MM2S_DMASR (AXI_DMA_BASE_ADDR + 0x04)  //SR = Status
Register
#define MM2S_SA (AXI_DMA_BASE_ADDR + 0x18)     //SA = Start
Address
#define MM2S_LENGHT (AXI_DMA_BASE_ADDR + 0x28) //Longitud

#define S2MM_DMACR (AXI_DMA_BASE_ADDR + 0x30)  //CR = Control
Register
#define S2MM_DMASR (AXI_DMA_BASE_ADDR + 0x34)  //SR = Status
Register
#define S2MM_DA (AXI_DMA_BASE_ADDR + 0x48)     //DA =
Destination Adress
#define S2MM_LENGHT (AXI_DMA_BASE_ADDR + 0x58) //Longitud
```

```
int main(int argc, char * argv []){
   int fd;
   void *map_base, *virt_addr, *map_base_thread,
*virt_addr_thread;
   //unsigned long read_result, writeval;
   //off_t target;
   // int access_type
   fd = open("/dev/uio1", O_RDWR | O_SYNC); //Abrimos toda la
memoria fisica del Xilinx
   printf("%d",fd);
   if((fd = open("/dev/uio1",O_RDWR | O_SYNC)) == -1 ){
        FATAL();
   }
   printf ("/dev/uio1 opened successfully.\n");
   fflush (stdout);

   //Inicializamos el DMA
   initialize_axi_dma();


   //Cargamos s2mm y mm2s para transferencia
   load_s2mm();
   load_mm2s();

   //Escribimos el dato
   //write_samples();
   read_samples();

   close(fd);
   return 0;

}


void write_samples (void){
   int i = 0, k = 0;
   int fd;
   off_t sample_addr;
   unsigned char * buffer;
   unsigned char * myData;
   void *map_base;
   FILE *file;

   file = fopen("datos.txt","w");
   buffer = malloc (256*1024);
   for(i=0;i<65536*4;i++)
   {
              *(buffer + i) = *(myData + i);
              (file,*(myData + i));
   }
   fclose(file);

   map_base = mmap(NULL, 256*1024, PROT_READ | PROT_WRITE,
MAP_SHARED,fd,TX_BUFFER);
```

```
   memcpy (map_base,buffer,256*1024);


   // if(file == NULL){
        // printf("Error al abrir archivo\n");
   // }
   // else {
        // fprintf(file,)
   // }

   if (munmap (map_base,MAP_SIZE) == -1){
        FATAL();
   }
   free (buffer);
}

void read_samples (void){
   int i = 0, k = 0;
   int fd;
   void *map_base;
   off_t sample_addr;
   unsigned char * buffer;
   unsigned char * myData_out;
   unsigned long read_result=0;

   buffer = malloc (256*1024);
   map_base = mmap(NULL, 256*1024, PROT_READ | PROT_WRITE,
MAP_SHARED,fd,RX_BUFFER);
   memcpy (buffer , map_base, 256*1024);
   for (i=0;i<65536*4;i++){
        *(myData_out+i) = *(buffer+i);
   }

   if(munmap(map_base, MAP_SIZE) == -1){
        FATAL();
   }
   free (buffer);
}

void initialize_axi_dma(void){
   off_t sample_addr;
   unsigned long read_result = 0;
   void *map_base;
   int fd;
   unsigned long virt_addr = 0;

   //MM2S_DMASR

   sample_addr = MM2S_DMASR;
   map_base = mmap(0,MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, sample_addr & ~MAP_MASK);
   if (map_base == (void *)-1){
        FATAL();
   }
```

```
   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long *) virt_addr) |= 0x10001;
   read_result = *((unsigned long *) virt_addr);

   printf("MM2S_DMASR set to: %lx\n",read_result);

   //MM2S_DMACR

   sample_addr = MM2S_DMACR;
   map_base = mmap(0,MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, sample_addr & ~MAP_MASK);
   printf("%lx",map_base);
   if (map_base == (void *) -1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long *) virt_addr) |= 0x10001;
   read_result = *((unsigned long *) virt_addr);

   printf("MM2S_DMACR set to: %lx\n",read_result);

   //MM2S_DMASR

   sample_addr = MM2S_DMASR;
   map_base = mmap(0,MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, sample_addr & ~MAP_MASK);
   if (map_base == (void *)-1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long *) virt_addr) |= 0x10001;
   read_result = *((unsigned long *) virt_addr);

   printf("MM2S_DMASR set to: %lx\n",read_result);

   if(munmap(map_base,MAP_SIZE)== -1) {
        FATAL();
   }


   //S2MM_DMACR

   sample_addr = S2MM_DMACR;
   map_base = mmap (0,MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED,fd, sample_addr & ~MAP_MASK);
   if(map_base == (void* ) -1){
        FATAL();
   }

   virt_addr = map_base +(sample_addr & MAP_MASK);
```

```
   *((unsigned long*)virt_addr) |= 0x10001;
   read_result = *((unsigned long*)virt_addr);

   printf("S2MM_DMACR set to: %lx\n",read_result);

   if(munmap(map_base,MAP_SIZE) == -1) {
        FATAL();
   }

   //S2MM_DMASR

   sample_addr = S2MM_DMASR;
   map_base = mmap (0,MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED,fd, sample_addr & ~MAP_MASK);
   if(map_base == (void* ) -1){
        FATAL();
   }

   virt_addr = map_base +(sample_addr & MAP_MASK);
   *((unsigned long*)virt_addr) |= 0x10001;
   read_result = *((unsigned long*)virt_addr);

   printf("S2MM_DMASR set to: %lx\n",read_result);

   if(munmap(map_base,MAP_SIZE) == -1){
        FATAL();
   }
}

void load_mm2s (void) {
   off_t sample_addr;
   unsigned long read_result = 0;
   void *map_base;
   unsigned long virt_addr = 0;
   int fd;

   //MM2_SA
   sample_addr = MM2S_SA;
   map_base = mmap(0,MAP_SIZE,PROT_READ | PROT_WRITE,
MAP_SHARED,fd, sample_addr & ~MAP_MASK);
   if(map_base == (void*)-1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long*) virt_addr)= TX_BUFFER;
   read_result = *((unsigned long*) virt_addr);

   printf("MM2S_DA set to: %lx\n",read_result);

   if(munmap (map_base,MAP_SIZE) == -1){
        FATAL();
   }
```

```
   //MM2S_LENGHT

   sample_addr = MM2S_LENGHT;
   map_base = mmap(0,MAP_SIZE,PROT_READ | PROT_WRITE,
MAP_SHARED,fd, sample_addr & ~MAP_MASK);
   if(map_base == (void*)-1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);

   *((unsigned long*) virt_addr)= 0x40000;
   read_result = *((unsigned long*) virt_addr);

   printf("MM2S_LENGHT set to: %lx\n",read_result);

   if(munmap (map_base,MAP_SIZE) == -1) {
        FATAL();
   }


}

void load_s2mm (void){
   off_t sample_addr;
   unsigned long read_result = 0;
   unsigned long virt_addr = 0;
   int fd;
   void *map_base;

   //S2MM_DA

   sample_addr = S2MM_DA;
   map_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, sample_addr & ~MAP_MASK);
   if(map_base == (void*) - 1){
        FATAL();
   }

   virt_addr = map_base + (sample_addr & MAP_MASK);
   *((unsigned long*) virt_addr) = RX_BUFFER;
   read_result = *((unsigned long*)virt_addr);

   printf("S2MM_DA set to: %lx\n",read_result);

   if(munmap(map_base,MAP_SIZE) == -1) {
        FATAL();
   }

   //S2MM_LENGHT

   sample_addr = S2MM_LENGHT;
   map_base = mmap(0,MAP_SIZE,PROT_READ | PROT_WRITE,
MAP_SHARED, fd, sample_addr & ~MAP_MASK);
   if(map_base == (void*)-1){
```

```
            FATAL();
      }

      virt_addr = map_base + (sample_addr & MAP_MASK);

      *((unsigned long*) virt_addr)= 0x40000;
      read_result = *((unsigned long*) virt_addr);

      printf("S2MM_LENGHT set to: %lx\n",read_result);

      if(munmap (map_base,MAP_SIZE) == -1){
            FATAL();
      }
   }
```

- **Código Devicetree DMA:**

  Este será el código a añadir sobre el devicetree original que posee la placa.

```
axi_dma@40400000{
      compatible = "generic-uio";
      reg = <0x40400000 0x10000>;
};

axi_dma@40410000{
      compatible = "generic-uio";
      reg = <0x40410000 0x10000>;
```

- **Código Shell Script**

```
#Programa en el que lanzare despues del boot mis ejecutables de
forma
#automatica
#DeclaraciÃ³n de vriable

#!/bin/bash

aux=0


#En primer lugar funciones FPGA y Python

function fpga()
{
      echo "Inicializacion funcion FPGA"
      cat /home/desarrollo/FPGA.bit > /dev/xdevcfg
      return

}

function pub ()
{
```

```
      echo "Inicializacion del script C"
      #Es necesario colocar & para que se ejecute como hilo
      gcc /home/desarrollo/axidma &
      gcc /home/desarrollo/axidma_adc &
      return
}

main ()
{
      echo "Iniciando Programa Personal"
      #Iniciamos FPGA
      fpga
      pub
      return
}
#Si la FPGA se encuentra lanzada lanzar siguiente paso

main
exit 0
#Si la FPGA no esta lanzada relanzar FPGA y volver a comprobar

#Si todo va bien lanzar script de python

#Pasos siguientes
```

# Referencias


[1] Xilinx, «Zynq-7000 All Programmable SoC Technical Reference Manual (UG585),» 10 04 2016. [En línea]. Available: www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. [Último acceso: 13 06 2016].

[2] 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 06 2016].

[3] 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 06 2016].

[4] 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 06 2015].

[5] Xilinx, «Xilinx Wiki Linux,» [En línea]. Available: http://www.wiki.xilinx.com/Linux. [Último acceso: 23 06 2016].

[6] Xilinx, «Xilinx Wiki -Petalinux,» [En línea]. Available: http://www.wiki.xilinx.com/PetaLinux. [Último acceso: 23 06 2016].

[7] Xilinx, «Petalinux Tools,» [En línea]. Available: http://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html. [Último acceso: 23 06 2016].

[8] 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 06 2016].

[9] 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 06 2016].

[10] 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 06 2016].

[11] 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 06 2016].

[12] A. L. ARM, «Arch Linux ARM,» [En línea]. Available: http://archlinuxarm.org/. [Último acceso: 23 06 2016].

[13] A. Linux, «ArchWiki,» [En línea]. Available: https://wiki.archlinux.org/. [Último acceso: 23 06 2016].

[14] A. L. ARM, «ZedBoard | Arch Linux ARM,» [En línea]. Available: http://archlinuxarm.org/platforms/armv7/xilinx/zedboard. [Último acceso: 23 06 2016].

[15] Xillybus, «Xillinux: A Linux distribution for Zedboard, ZyBo, MicroZed and SocKit | xillybus.com,» [En línea]. Available: http://xillybus.com/xillinux. [Último acceso: 23 06 2016].

[16] Xilinx, «Downloads,» [En línea]. Available: http://www.xilinx.com/support/download.html. [Último acceso: 26 06 2016].