

# Índice general

<b>1. Introducción</b>	<b>3</b>
1.1. Motivación . . . . .	3
1.2. Protocolos disponibles para la transmisión de datos entre PC y FPGA . . . . .	7
1.3. Bus Serial Universal 2.0 . . . . .	11
1.3.1. Descripción general de un sistema USB . . . . .	12
1.3.2. Dispositivos que componen un sistema USB . . . . .	14
1.3.3. Paquetes USB . . . . .	15
1.3.4. Tipos de Transferencias . . . . .	17
1.3.5. Descriptores . . . . .	19
1.4. Objetivos . . . . .	20
1.4.1. Objetivo Principal . . . . .	20
1.4.2. Objetivos Particulares . . . . .	20
1.5. Estructura del Informe . . . . .	21
1.6. Sumario del capítulo . . . . .	21
<b>2. Interfaz USB</b>	<b>23</b>
2.1. Elección de la Interfaz . . . . .	23
2.2. El controlador FX2LP EZ-USB y su configuración . . . . .	25
2.2.1. Microcontrolador Cypress 8051 Mejorado . . . . .	26
2.2.2. Frecuencia de trabajo del sistema . . . . .	26
2.2.3. Memoria FIFO . . . . .	27
2.2.4. Modos de entrada y salida automáticos . . . . .	30
2.2.5. Encabezado y declaraciones importantes . . . . .	32
2.2.6. Descriptores USB . . . . .	33
2.3. Depuración y verificación de funcionamiento . . . . .	38
2.3.1. Biblioteca FX2LPSerial . . . . .	39
2.3.2. Testigos LED . . . . .	40
2.3.3. Prueba de envío y recepción de datos . . . . .	41
2.4. Sumario del capítulo . . . . .	42
<b>3. Desarrollo del sistema maestro para la comunicación entre la FPGA y la interfaz Cypress</b>	<b>43</b>
3.1. Elección de la FPGA . . . . .	43
3.2. Señales de comunicación de la Máquina de Estados Finitos . . . . .	45
3.2.1. Señales de comunicación entre la FPGA y el controlador FX2LP . . . . .	46
3.2.2. Comunicación interna del FPGA . . . . .	48

---

*ÍNDICE GENERAL*

3.3.	Diseño de la Máquina de Estados Finitos . . . . .	50
3.4.	Síntesis de la Máquina de Estados en VHDL . . . . .	52
3.5.	Verificación funcional de la síntesis . . . . .	55
3.6.	Placa de Interconexión . . . . .	58
3.7.	Sumario del capítulo . . . . .	59
<b>4.</b>	<b>Pruebas de funcionamiento y desempeño del sistema desarrollado</b>	<b>61</b>
4.1.	Implementación de un sistema genérico en FPGA . . . . .	61
4.1.1.	Declaración de la entidad . . . . .	62
4.1.2.	Instanciación de la MEF . . . . .	63
4.1.3.	Otros componenetes y señales de control . . . . .	64
4.2.	Pruebas de la comunicación entre el FPGA y la PC . . . . .	67
4.2.1.	Elección de la biblioteca libusb-1.0 . . . . .	67
4.3.	Resultados . . . . .	67
4.4.	Conclusiones . . . . .	67
4.5.	Trabajos Futuros . . . . .	67
<b>Apéndice A.</b>	<b>Códigos de descripción escritos en VHDL</b>	<b>73</b>
A.1.	Código de implementación de la MEF que controla la interfaz FX2LP . . . . .	73
A.2.	Código de validación de la MEF . . . . .	77
A.3.	Código de síntesis del sistema de prueba . . . . .	77

# Capítulo 1

## Introducción

El presente informe busca dar a conocer al lector las tareas y actividades desarrolladas por el autor, en el marco del Trabajo Final de la carrera Ingeniería Electrónica, dictada en la Facultad de Ingeniería de la Universidad Nacional de San Juan. El objetivo del trabajo es diseñar e implementar una interfaz para la transmisión de datos hacia una computadora personal (PC), adquiridos por sistemas desarrollados en arreglos de compuertas de campo programables (FPGA) para aplicaciones científicas, a través del Bus Serial Universal (USB). A lo largo de este documento, se comprenderá la problemática que se resuelve y la configuración, fundamentos y modo de uso del sistema propuesto.

En la sección 1.1 se presentan las motivaciones de este trabajo y se detalla la problemática a resolver. Luego, se detallan los objetivos que persigue este trabajo. Seguido a esto, se otorga un esquema que describe la solución planteada y se justifica el protocolo elegido. Finalmente, se repasan algunos conceptos importantes de la norma USB que luego se utilizan en el trabajo desarrollado.

### 1.1. Motivación

El grado de avance que han experimentado la electrónica y la tecnología en general, gracias a la industria de los semiconductores, permite que la producción científica pueda adquirir una gran cantidad de datos. Para llevar a cabo la producción del conocimiento, es necesario el relevamiento y registro de diferentes tipos de magnitudes físicas y/o químicas sobre el objeto o proceso a investigar. En muchas ocasiones, estas magnitudes resultan difíciles de observar y cuantificar, por lo que es conveniente transformar las variables a conocer en otras más sencillas de medir. Para este propósito, se utilizan transductores.

Se conoce como transductor a cualquier dispositivo que recibe estímulos energéticos de una condición, situación o fenómeno físico y/o químico y los convierte en una señal asociada y definida de otra forma de energía[1, 2]. En otras palabras, los transductores son conversores de energías[1, 2, 3]. Se denomina sensor a una clase particular de transductor que genera, como variable de salida, una señal eléctrica que está especialmente adaptada para ser ingresada en un circuito electrónico, o adecuada al sistema de medida que se utilice [4, 5, 6].

Las altas escalas de integración de circuitos alcanzadas en la actualidad posibilitan el diseño de sistemas sensoriales cada vez más complejos, en los cuales se logra agrupar miles de sensores en áreas reducidas, obteniendo medidas simultáneas y flujos crecientes de datos. Este trabajo se

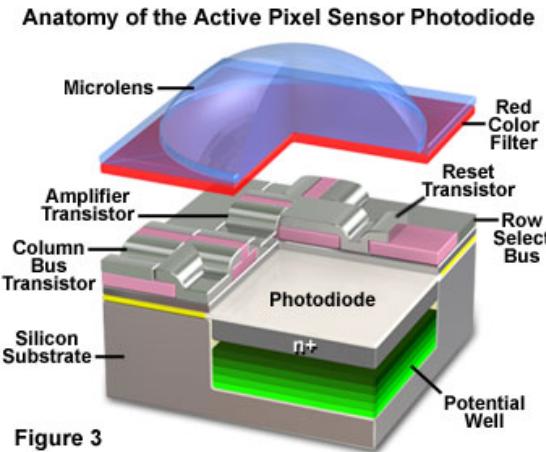


Figura 1.1: Esquema físico de un APS[8]

centrará en la transmisión de datos provenientes de sensores de imagen, uno de los desarrollos que se encuentra en boga.

Una imagen, desde un punto de vista digital, es un arreglo bidimensional de números, los cuales pueden ser exhibidos en una pantalla en forma de intensidad y colores de luz. Cada punto del arreglo que se muestra en pantalla se denomina pixel, acrónimo del inglés *PIcture EElement*, o elemento de imagen. Por esto, un sensor de imagen puede estar compuesto, bien por un arreglo bidimensional de sensores lumínicos (como la cámara de un teléfono celular), como por un transductor que es simultáneamente desplazado y medido, (método utilizado, entre otras, para la microscopía de fuerza atómica [7]), o por una combinación de ambos métodos. Por ejemplo, un scanner posee un arreglo lineal de transductores que son desplazados a través de la hoja para generar una imagen digital. En cualquiera de los casos, es de suma utilidad que la lectura de imágenes sea realizada en el menor tiempo posible, ya que cada imagen conlleva una cantidad no menor de datos.

Uno de los trabajos que más aportó al desarrollo de sensores de imágenes modernos, fue la introducción de los APS (*Active Pixel Sensor*, o sensor de píxeles activos) [9]. Este sensor integra en un proceso CMOS (acrónimo inglés de Metal-Óxido-Semiconductor Complementario, que es el método actualmente más económico para integrar transistores en una única pastilla de silicio), un fotodiodo, un transistor de reset (utilizado para controlar el tiempo de integración, es decir, de exposición a la luz) transistores de selección (utilizados para conectar un pixel determinado dentro del arreglo) y un amplificador seguidor de fuente en cada pixel[8]. El fotodiodo, previamente cargado, transduce la luz en una descarga eléctrica y el amplificador convierte la carga remanente en tensión para facilitar su lectura. La Figura 1.1 muestra el dibujo de un APS. Se observa el área sensible a la luz y los diferentes transistores que intervienen en su funcionamiento. Además se incorpora una micro-lente cuya función es la de enfocar los fotones sobre el área sensible y un filtro utilizado para identificar colores. En el caso de sensores monocromáticos, se omite la colocación del filtro de color durante la fabricación.

A partir del desarrollo de los APS, se fue perfeccionando el método hasta obtener circuitos integrados con mayor cantidad de píxeles y que pueden tener diversas aplicaciones. Por ejemplo, en los trabajos [10] y [11] se presentan sensores CMOS basados en la arquitectura MIMOSA (de *Minimum Ionizing particule MOS Active pixel Sensor*). Estos sensores se desarrollaron con el objetivo específico de detección de radiación ionizante.

También existen desarrollos de sensores de radiación a través de APS comerciales. Perez *et al.* identificaron eventos producidos por partículas alfa en campos de radiación mixtos mediante el procesamiento de imágenes adquiridas con sensores comerciales CMOS[12] y desarrollaron detectores de neutrones térmicos con sensores APS cubiertos con una capa de  $\text{Gd}_2\text{O}_3$ [13]. Galimberti *et al.* utilizaron un sensor de imágenes comercial para realizar un detector de gas Rn en el ambiente[14]. En otro trabajo, Hizawa, *et al.* fabricaron un sensor que adquiere imágenes midiendo el pH con cada uno de los píxeles[15], pudiendo observar de fenómenos químicos en tiempo real.

Como se mencionó antes, una imagen digital es un arreglo de datos. Esto quiere decir que un sensor de imágenes con  $n$  píxeles de largo y  $m$  de ancho, captura  $n \times m$  datos en cada lectura. A su vez, para digitalizar valores, un circuito debe poseer, al menos, un conversor analógico-digital (ADC) de  $x$  cantidad de bits, lo que implica que cada dato estará compuesto por  $x$  dígitos binarios, es decir, un volumen importante de datos por cada lectura. Como ejemplo, un sensor comercial VGA, en su configuración más básica, posee 640 líneas horizontales y 480 verticales, con una resolución de 8 bits por cada pixel, lo que otorga 2.457.600 bits por cada lectura del sensor.[16] Si además se incorpora la cantidad de imágenes que se toman en función del tiempo (cuadros por segundo o fps), nos otorga un flujo de datos para nada despreciable.

Desde el punto de vista de la electrónica digital, para poder adquirir y transmitir grandes volúmenes de datos, se requiere de circuitos que sean capaces de operar a altas frecuencias de conmutación. El diseño de dichos circuitos no es trivial, ya que cuando las longitudes de onda de las señales presentes son comparables con las dimensiones físicas de dichos circuitos, debe considerarse el uso de líneas de transmisión[17]. Esto implica que no se puede diseñar utilizando un criterio de uniformidad en los parámetros y exige un análisis mas detallado y preciso.

Otro problema que presentan los circuitos electrónicos digitales tiene que ver con los tiempos de propagación de las corrientes y tensiones que circulan a través de ellos. Cuando se aplica un impulso en un conductor, debido a las capacidades propias de los materiales utilizados, las tensiones pueden demorar unos instantes en establecerse. Puede suceder que varias señales lleguen a los puertos de un dispositivo por conductores con distintas longitudes y generen retardos diferentes. Esto puede ocasionar un comportamiento indeseado si no se toman los recaudos adecuados.

Aún suponiendo un perfecto diseño, los circuitos digitales de alta velocidad se encuentran limitados en la frecuencia de conmutación por la temperatura que se necesita disipar. La potencia consumida por estos dispositivos es proporcional a la frecuencia de funcionamiento[18]. Parte de esta potencia se transforma en calor y produce un aumento en la temperatura. Si el incremento es indiscriminado, puede destruir los circuitos.

Una posible solución para disminuir la frecuencia de las señales sin perjudicar la tasa de transferencia es la incorporación de varios conductores para enviar datos en paralelo. La cantidad de conductores a través de los cuales circula la información, se denomina ancho de bus. Idealmente, para lograr una tasa de transferencia determinada, se podría disminuir la frecuencia tantas veces como conductores se agreguen. Por ejemplo, transmitiendo por cuatro filamentos, se podría enviar la misma información a un cuarto de la frecuencia que se necesitaría con uno solo de iguales características.

Existen distintas tecnologías para efectuar la lectura de los datos generados por los sensores y su posterior transmisión. La incorporación y evolución de microcontroladores permite capturar y procesar volúmenes crecientes de datos. Sin embargo, este tipo de dispositivos posee una

estructura rígida: su capacidad de procesamiento se encuentra limitada a una instrucción por ciclo de reloj y a un ancho de bus definido. Para aumentar los volúmenes de datos que circulan a través de ellos, no es posible aumentar el ancho de bus, sino que se torna necesario incrementar la frecuencia de funcionamiento, generando los problemas anteriormente detallados.

Una solución óptima, sin considerar los costos asociados a esto, sería el desarrollo de un circuito integrado de aplicación específica (ASIC del inglés *Application Specific Integrated Circuit*). En este tipo de circuitos, el diseñador elabora un circuito que puede operar a altas velocidades y, a su vez, obtener un ancho de bus sin restricciones, más que las dimensiones físicas del área donde será realizado el circuito. Sin embargo, cuando sí se considera el costo asociado a este enfoque, se vuelve una solución ineficiente en bajas cantidades. La manufactura de este tipo de dispositivos puede tener un costo de miles hasta cientos de miles de dólares, dependiendo del proceso de fabricación utilizado. Gran parte de estos costos son no recurrentes, es decir, solo se pagan una vez por proyecto. En grandes cantidades de dispositivos, este tipo de soluciones se vuelven más convenientes.

Otro enfoque, es la utilización de Arreglos de Compuertas Programables por Campo (FPGA, acrónimo del inglés *Field-Programmable Gate Array*). Un FPGA es un dispositivo electrónico que posee la capacidad de sintetizar casi cualquier circuito digital. En esencia, es una matriz de bloques lógicos (también llamadas *slices* o celdas lógicas, dependiendo del fabricante), que contienen Tablas de Verdad(LUTs o *Look-Up-Table*) y flip-flops (ff), entre otras cosas, y pueden ser interconectadas entre sí, según el criterio del usuario. Así, permite implementar una solución digital en un circuito físico, a diferencia de los microcontroladores, lo realiza a través de un algoritmo almacenado en una memoria, incorporando la ventaja de definir el ancho de bus necesario para relevar una gran cantidad de datos y transmitirlos a frecuencias de trabajo menores, además de ejecutar tareas en paralelo, disminuyendo los tiempos de procesamiento. A su vez, al ser implementado en un área muy pequeña, debido a la integración del sistema, este tipo de sistemas puede trabajar a frecuencias muy elevadas, lo que implica una mayor tasa de datos aún. A pesar de la gran diversidad de precios existentes en el mercado, una FPGA de costos menores a la centena de dólares suele tener muy buenas prestaciones para la mayor parte de las aplicaciones.

Existen diversas publicaciones en donde se observa el uso de FPGAs para la implementación de sistemas que producen imágenes. Por ejemplo, el desarrollo de un detector de radiación ionizante utilizando una sensor de imagen CMOS comercial. Para ello, los autores utilizaron una FPGA para configurar diversos parámetros del sensor con el fin de generar estrategias para la identificación de partículas alfa en campos de radiación mixtos y transmitir imágenes a una computadora personal (PC) a través de un puerto UART[19]. Se denomina ultrasonografía a la técnica de adquirir imágenes basándose en reflexiones de ultrasonido. Sus aplicaciones son múltiples, en las que se destaca el diagnóstico médico debido. Un trabajo reciente desarrolló un sistema que mejora la obtención de ecografías médicas con bajo costo utilizando una FPGA[20]. El autor presentó un algoritmo para la supresión de ruido de impulso en tiempo real para imágenes codificadas como JPEG 2000 realizado y probado en Matlab e implementado en una FPGA. Yanagisawa *et al*, desarrollaron un sistema con telescopios pequeños para explorar objetos de campo cercano con la finalidad de monitorear cuerpos celestes que puedan colisionar con el planeta[21]. En este trabajo, se aprovechó la velocidad de los circuitos implementados en FPGA para minimizar el tiempo de adquisición.

El desarrollo de nuevos sensores brinda a los investigadores un gran volumen de datos. En

muchos casos, la obtención de datos por si misma no otorga información, sino que es necesario procesar y analizar los mismos. La invención y evolución de las computadoras, como así también el desarrollo de nuevos algoritmos, dan lugar a procesamiento de datos cada vez más complejos en tiempos mucho menores. Las primeras ENIAC, computadoras de propósito general desarrollada en el año 1946 para el cálculo de tablas balísticas de las fuerzas armadas estadounidenses, podía ejecutar 20 operaciones cada  $10\text{ }\mu\text{s}$  [22], es decir, ejecutaba instrucciones con una frecuencia máxima de 200 kHz. A su vez, tuvo un costo aproximado de U\$S 500.000, pesaba 5 t y consumía 175 kW. En contraste con aquello, es posible conseguir en el mercado actual, computadoras con tamaño y peso reducido, que ejecutan instrucciones en cuenstión de nanosegundos, (5 ordenes de magnitud menos), consumen menos de 1 kW y cuestan algunos cientos de U\$S. A tal punto ha evolucionado esta tecnología, que se cuenta con computadoras muy potentes en casi cualquier laboratorio, oficina u hogar. La capacidad de cálculo que exhiben estos dispositivos, sumada al desarrollo de nuevos métodos y algoritmos de cálculo, permite a los investigadores procesar datos en tiempo reducido, facilitando el análisis y la generación de nueva información.

En todos los casos que se consideran en este trabajo, la generación de datos y el procesamiento de lo mismos se da en sistemas diferentes. Es decir, los datos son relevados por los sensores y adquiridos luego por los FPGAs. Finalmente llegan a una PC para su posterior procesamiento y análisis. Se requiere, por tanto, de una conexión a través de la cual los datos puedan ser transferidos del sistema de adquisición, la FPGA, a la PC y viceversa. Se torna de suma utilidad, entonces, proveer una comunicación efectiva y robusta que permita transmitir grandes volúmenes de datos en poco tiempo, y de esta forma facilitar los tiempos de desarrollo, pruebas, depuración, procesamiento y análisis.

La implementación de un sistema de comunicación en una FPGA puede ser resuelta de muchas maneras, quedando a criterio del desarrollador utilizar algún protocolo estándar, o bien diseñar uno propio. Sin embargo, en una computadora, las formas de comunicar datos se vuelven un poco más restrictivas y acotadas a los puertos y señales que puede manejar el equipo, conforme el fabricante haya establecido. Este trabajo busca implementar una comunicación entre una computadora personal y una FPGA, utilizando un protocolo estándar, que esté disponible en cualquier computadora comercial y que posea una tasa de bit suficiente para poder transmitir imágenes.

## **1.2. Protocolos disponibles para la transmisión de datos entre PC y FPGA**

El estándar más exigente de la norma americana de la SCTE (Sociedad de Ingenieros de Comunicación por Cable) utilizada Televisión Digital, posee una tasa de  $38.8\text{ Mbit s}^{-1}$ [23]. Por su parte, la serie de sensores para adquirir imágenes monocromáticas MT9M001, comercializado por ON Semiconductors posee  $1280\times 1024$  pixeles, con profundidad de 10 bits y puede operar hasta a 30 cuadros por segundo[24]. La tasa de transmisión necesaria es, por tanto, de  $393.2\text{ Mbit s}^{-1}$ .

Un requerimiento que posee cualquier periférico informático es el de compatibilidad. No es conveniente utilizar puertos que requieran acceso a la placa madre, como el caso de tarjetas de tipo PCI o PCI express, debido a que no todos los equipos los tienen accesible, como ser computadoras portátiles, y en algunos casos estos pueden estar todos ocupados. Se opta, entonces, por alguno de los tres puertos de moda: Ethernet, dedicado principalmente a conexión de redes

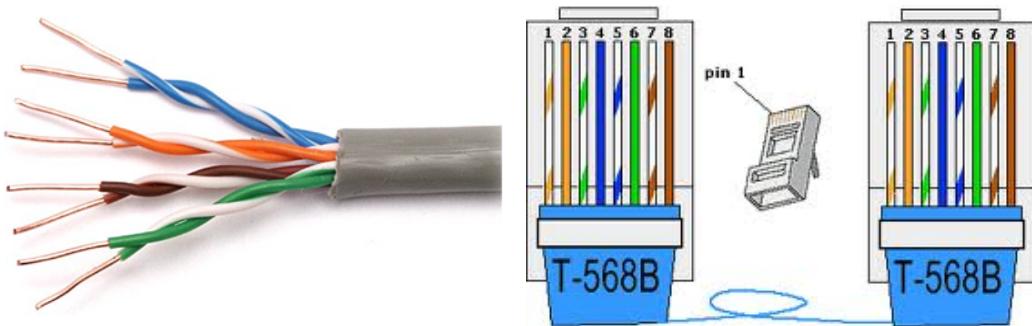


Figura 1.2: Par Trenzado y un dibujo de su ficha de conexión.

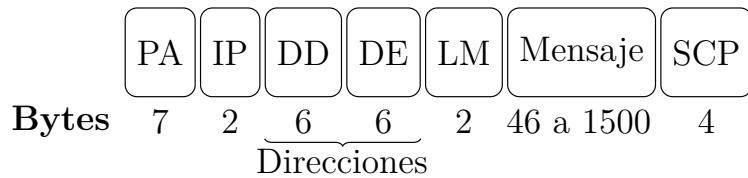
mediante cables; Wi-Fi, utilizado para el acceso a la red de forma inalámbrica; y USB, dirigido a la comunicación de periféricos con la PC.

Al hablar de Ethernet o Wi-Fi, se hace referencia a dos formas diferentes de conectarse a una red de computadoras. En otras palabras, se habla de dos o más nodos, compuestos por PCs o cualquier dispositivo electrónico con capacidad de realizar cálculo binario, que pueden intercambiar datos a través de una trama bastante compleja de componentes diferentes. Ambos protocolos hacen referencia solo a la conexión física de los dispositivos y el control de acceso de cada uno de ellos a la conexión. Quedando a cargo de otros sistemas, con sus protocolos, que los datos enviados puedan ser correctamente recibidos por el usuario de la PC. La gran diferencia entre ellos radica en el medio físico que utilizan: Wi-Fi emplea ondas electromagnéticas emitidas mediante radiofrecuencia, mientras que en Ethernet, estas ondas son acarreadas por uno o más conductores, como ser cable coaxial, cables de par trenzado o fibra óptica.

Ethernet, también conocido como IEEE 802.3, es una norma que define cómo se deben conectar nodos a través de conductores para conformar redes de área local (LAN o *Local Area Network*), es decir, redes pequeñas, como ser domésticas, de oficinas o de pequeñas empresas, de forma que puedan transmitir información a velocidades seleccionables entre 1 Mbit/s y 400 Gbit/s [25]. Utiliza una tecnología denominada Acceso Múltiple Sensando la Portadora con Detección de Colisiones (CSMA/CD del inglés *Carrier Sense Multiple Access with Collision Detection*). En una red con CSMA/CD, cada dispositivo debe sensar en forma permanente la conexión a la red, es decir, no existe un dispositivo que dirija el uso del bus, sino que cada uno debe identificar el estado de la red. Los mensajes se envían modulados. Cuando una señal portadora es detectada, todos chequean la dirección del paquete de información que viaja y el mensaje es recibido solamente por el dispositivo que se corresponda con esa dirección. Siempre que exista una señal portadora en el bus, los dispositivos que deseen transmitir información deberán esperar a fin de evitar colisiones, o sea, que dos dispositivos envíen mensajes a la vez y estos se interfieran.

Dependiendo de la frecuencia de la portadora y la tasa de transferencia a la que transporta el mensaje, la norma especifica el conector y la distancia máxima a la que debe conectarse una repetidora, es decir, un dispositivo que reciba, reconstruya y emita la señal recibida. Estos conectores pueden ser cable coaxial, fibra óptica o cable de par trenzado. Este último es el más usual en las PCs comerciales y se muestra, junto a su ficha característica en la Figura 1.2.

La información se estructura en paquetes para permitir la comunicación entre muchos nodos de la red. Un paquete, como se observa en la Figura 1.3, se compone de un preámbulo con 7 B que sirve para sincronizar los dispositivos en cada extremo de la conexión, 1 B de inicio,



### Referencias

**PA:**Preámbulo

**IP:** Inicio de Paquete

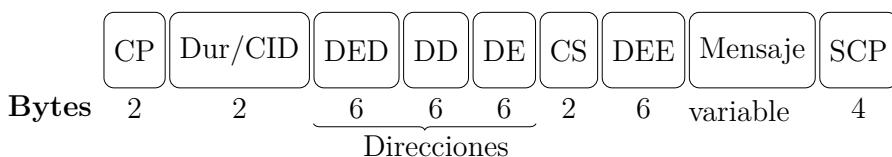
**DD:** Dirección de Destino

**DE:** Dirección de Emisión

**LM:** Longitud del Mensaje

**SCP:** Secuencia de chequeo del paquete

Figura 1.3: Estructura de un paquete Ethernet



### Referencias

**CP:** Control de Paquete

**Dur/CID:** Duración del paquete/Identificación de conexión

**DED\*:** Dirección de Enrutador de Destino

**DD\*:** Dirección de Destino

**DE:** Dirección de Emisión

**CS\*:** Control de Secuencia

**DEE\*:** Dirección de Enrutador de Emisión

**SCP:** Secuencia de chequeo del paquete

\*Pueden no estar dependiendo del tipo de mensaje

Figura 1.4: Estructura de un paquete Wi-Fi

12 B de direcciones, que corresponden 6 al nodo destinatario y 6 al emisor respectivamente, 2 B que indican la longitud del mensaje, entre 46 y 1500 B de datos y 4 B para la verificación de la transmisión. Otra definición importante de la norma, son las características eléctricas de las señales, pero no se detallan en este trabajo porque varían en función de la velocidad del puerto.

Por su parte Wi-Fi, perteneciente a la asociación de compañías denominada Wi-Fi Alliance, se rige por la norma que estableció esta última. Existe una norma equivalente, encuadrada en la especificación IEEE 802.11, referida a las redes de área local inalámbrica, o WLAN (siglas del inglés *Wireless Local Area Network*). Wi-Fi se enfoca en las que se refieren a las comunicaciones de radiofrecuencia con portadora de 2.4 GHz, que se incorporan en las revisiones b, g y n de la norma IEEE. IEEE 802.11 está pensado especialmente para dispositivos portátiles y móviles. La norma define a los dispositivos portátiles como aquellos que pueden ser trasladados con facilidad pero operan estáticos y los móviles se identifican por poder trabajar en movimiento [26]. La principal característica que posee este tipo de comunicación es la falta de conductores para la elaboración de la red, sin contar las conexiones entre los transceptores que emiten y reciben las

señales de radiofrecuencias y los nodos, en donde la información es producida y/o consumida. En cuanto al formato del paquete de datos, el cuál se muestra en la Figura 1.4, es bastante similar al de Ethernet. En primer lugar, se envían dos bytes de control que indican el tipo de paquete a enviar. Luego siguen dos bytes que, dependiendo de la etapa de la comunicación puede indicar la duración del mensaje a transmitir o un identificador de una conexión establecida previamente. Siguen entre 6 y 18 bytes de direcciones del enrutador que recibe los datos, el nodo emisor y el destinatario. Continúan, dos bytes de control de secuencia se utilizan para fragmentar transmisiones largas. Continua un campo más para dirección que corresponde a la red emisora de 6 bytes. Todos los campos de dirección pueden variar en función del tipo de mensaje que se envía. Los últimos dos campos de la trama corresponden a la información que se quiere comunicar (hasta 2312 bytes) y un código de chequeo por redundancia cíclica de 32 bits (4 bytes).

Existen múltiples ventajas de utilizar radiofrecuencias para conectarse a la red, tales como la libertad de mover el punto de trabajo y la economía a la hora de armar redes con muchos nodos. Sin embargo, posee algunas desventajas notorias, propias del medio de propagación, que lo hacen no tan óptimo para los fines del presente trabajo. Las redes inalámbricas tiene la característica de que no son del todo confiables: posee múltiples fuentes de interferencia, ya que varias tecnologías que utilizan la misma frecuencia (Bluetooth, Zig-Bee, WUSB, microondas). A su vez, suele presentar variaciones temporales y asimetrías en las propiedades de propagación, lo que puede provocar interrupciones en la comunicación.

Ambos protocolos proporcionan una solución de conexión de redes de nivel físico y ejecutan tareas de control de acceso al medio (MAC) a fin de evitar colisión en los datos, es decir, que dos dispositivos transmitan en forma simultánea e interfieran la comunicación. Sin embargo, para establecer una red, faltan componentes físicos y lógicos tales como un sistema de control enlace lógico (Logic Link Control), un sistema de direccionamiento, como el Protocolo de Internet (IP), una capa de transporte de datos, (como el protocolo TCP) y las capas de software que permiten acceder a los protocolos anteriormente mencionados.

A pesar de lo anterior, es posible establecer comunicaciones punto a punto con ambos protocolos, simplificando mucho el sistema de transmisión de datos. Sin embargo esta solución presenta un inconveniente no menor: se le quita a la PC un acceso a la red, que en la mayoría de los casos es el único. Esto no es deseable ya que la conectividad es un requisito fundamental en cualquier hogar u organización, ya sea empresarial, gubernamental, científica o de cualquier tipo.

Por su parte el protocolo USB (acrónimo de *Universal Serial Bus*), es una norma desarrollada por seis de las empresas más grandes de la industria informática, pensada y desarrollada para la conexión de teléfonos y periféricos a PCs [27]. En la versión original, USB posee conectores cableados de 4 conductores y presenta una topología de bus, es decir todos los dispositivos se conectan a un mismo circuito conductor. La conexión es manejada por una PC y solo transmite o recibe un dispositivo a la vez. Tal fue la penetración de USB en el mercado, que se transformó en una norma de facto. Actualmente es incorporada casi por defecto en casi todas las computadoras disponibles en el mercado y es necesaria a la hora de comprar e instalar periféricos.

USB presenta diferentes versiones de su norma, cada cual con una o más tasas de transmisión y señalización. La versión 1 posee dos revisiones, 1.0 fue lanzada al mercado en el año 1996 y 1.1 que se presentó en Agosto de 1998. La primera alcanza una tasa máxima de  $1.5 \text{ Mbit s}^{-1}$  y la segunda hasta  $12 \text{ Mbit s}^{-1}$ . USB 2.0 fue presentado en Septiembre del 2000 y es capaz

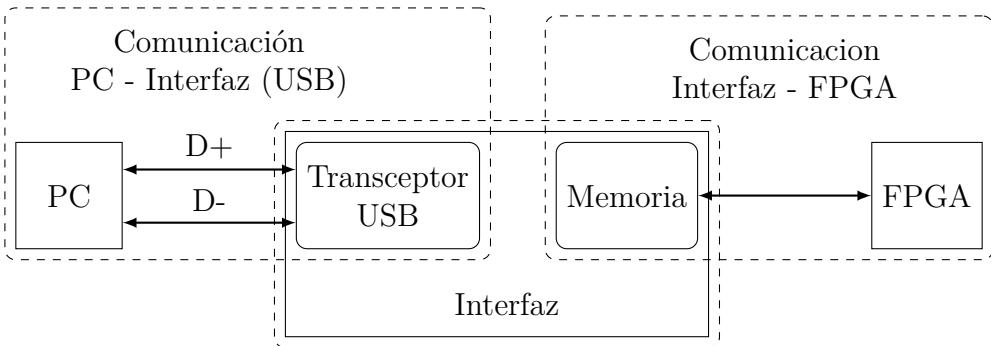


Figura 1.5: Partes en que se desglosa el trabajo

de transmitir a  $480 \text{ Mbit s}^{-1}$ . La tercera versión, USB 3.0, fue lanzada al mercado en 2011 y transmite a una tasa de  $5 \text{ Gbit s}^{-1}$ . Esta última versión fue revisada en julio de 2013 y en septiembre de 2017, ofreciendo  $10 \text{ Gbit s}^{-1}$  y  $20 \text{ Gbit s}^{-1}$  respectivamente.

Se elige para el desarrollo de este trabajo la norma ya que USB 2.0 presenta una tasa de transferencia de datos suficiente para la transmisión de imágenes. A su vez, resulta ideal para los objetivos buscados debido a encontrarse presente en la mayoría de las computadoras y no interferir en la conexión a internet de las mismas. En el Capítulo 1.3 se profundizarán en conceptos específicos de la norma USB.

Es posible implementar una comunicación USB completa a través de una FPGA. Sin embargo, esto sería demasiado oneroso en términos de tiempos de desarrollo y de recursos de FPGA disponibles para la implementación de otros sistemas, los cuales son el objetivo de la comunicación. Se plantea, entonces, un esquema como el que se observa en la Figura 1.5 en la cual se utiliza una interfaz externa al FPGA. La comunicación USB propiamente dicha será efectuada entre la interfaz y la PC, mientras que se plantea una comunicación diferente entre la interfaz y el FPGA. Este último, por su parte, tendrá la tarea de realizar el control de esta comunicación.

### 1.3. Bus Serial Universal 2.0

El Bus Serial Universal, o USB por sus siglas en inglés, es un sistema de comunicación diseñado durante los años 90 por seis fabricantes vinculados a la industria informática, Compaq, Intel, Microsoft, Hewlett-Packard, Lucent, NEC y Philips, con la idea de proveer a su negocio de un sistema que permita la conexión de PCs con teléfonos y periféricos con un formato estándar, fácil de usar y que permita la compatibilidad entre los distintos fabricantes.

Hasta ese momento, el gran ecosistema de periféricos, sumado a los nuevos avances y desarrollos, hacia muy compleja la interoperatividad de todos ellos. Cada uno de los fabricantes desarrollaba componentes con características, tales como fichas, niveles de tensión, velocidades, drivers, lo cuál dificultaba al usuario estar al día y poder utilizar cada componente que compraba. Esto también complicaba a las mismas empresas productoras, por que la introducción de un nuevo sistema requería de mucho soporte extra para poder conectar lo existente en forma previa.

Todo esto, quedó saldado con el aparición de la norma USB que, gracias a la gran cuota de mercado de sus desarrolladores, fue adoptada en forma rápida y se transformó en la especificación casi por defecto a la hora de seleccionar un protocolo para periféricos. La penetración en el

mercado fue tal que hoy, más de 20 años después, es difícil encontrar PC con otro tipo de puertos, salvo que en el momento de compra se solicite de manera especial. No obstante, cualquier PC nueva disponible en el mercado debe poseer puertos USB para la conexión de, al menos, los periféricos.

El diseño de la norma USB busca resolver tres problemáticas interrelacionadas, que son: La conexión de teléfonos con las PC, la facilidad de uso, es decir, que el usuario solo conecte su dispositivo y pueda utilizarlo, y la expansión en la cantidad de puertos disponibles para conectar periféricos<sup>[27]</sup>. Para satisfacer estas tres demandas, la norma USB 2.0 busca alcanzar un conjunto de metas que apuntan a la facilidad del uso, la compatibilidad entre versiones diferentes de la misma tecnología, la robustez en el flujo de datos, y la convivencia de diferentes configuraciones temporales en único bus. Para alcanzar estas metas, la norma provee una interfaz estándar, ancho de banda que soporte comunicaciones audiovisuales de calidad aceptable y un bajo coste.

El presente capítulo intenta ser un breve resumen con los aspectos más relevantes de la norma en cuanto a su composición física, su topología, los dispositivos que intervienen, la importancia de los mismos y como los datos son transmitidos desde y hacia una PC.

### **1.3.1. Descripción general de un sistema USB**

Un sistema USB posee un esquema en forma de árbol cuyo nodo principal es el host. Es decir, la comunicación se realiza siempre a través de una única línea a la que se conectan todos los dispositivos (bus). Dado el campo de direcciones provisto por la norma, un sistema USB puede conectar hasta 128 dispositivos. El acceso al bus es administrado por un maestro. El maestro se encarga de solicitar a cada uno de los dispositivos su intervención. Posteriormente, el dispositivo debe responder al pedido del maestro. Este esquema es lo que se conoce como maestro-esclavo. De esta forma, el sistema se asegura que el bus sea utilizado por un dispositivo a la vez para enviar o recibir datos.

En un sistema USB no cualquier dispositivo puede ser maestro. Este rol lo cumple solo uno: una PC, o cualquier dispositivo con capacidad de llevar a cabo las tareas asignadas (que se detallan más adelante); denominado Host por la norma. La palabra *HOST* proviene del habla inglesa y se traduce como anfitrión, aunque en la jerga se conoce comúnmente por su nombre en inglés.

La topología del bus, que se observa en la Figura 1.6, posee forma de árbol, es decir, puede ser pensada como una comunicación vertical, donde en el punto más alto se encuentra el Host. Siguiendo hacia abajo, el bus puede encontrar dos tipos diferentes de dispositivos: Funciones, cuyo rol es el de proveer una utilidad al sistema, como ser la de captura de imagen, reproducción de audio o el ingreso de comandos; y Hubs (concentradores o distribuidores), que se encargan de conectar una o más funciones al sistema. La norma USB establece gradas, en donde cada Hub introduce una nueva grada que contiene a las Funciones conectadas. En otras palabras, las gradas configuran una suerte de distancia lógica entre las Funciones y el Host, separada por Hubs. Por cuestiones de restricciones temporales y tiempos de propagación en los cables, no se permiten más de 7 gradas, incluyendo al Host en la primera. Es decir, no se puede conectar más de 5 Hubs en cascada. La grada 7 sólo puede contener Funciones<sup>[27]</sup>.

Cada uno de estos dispositivos diferentes, se interconectan entre sí a través de cables y conductores específicos, diseñados en forma tal que no sea posible conectarlos en forma equivocada. Para cumplir con la norma, el Host debe tener siempre un zócalo compatible con conectores

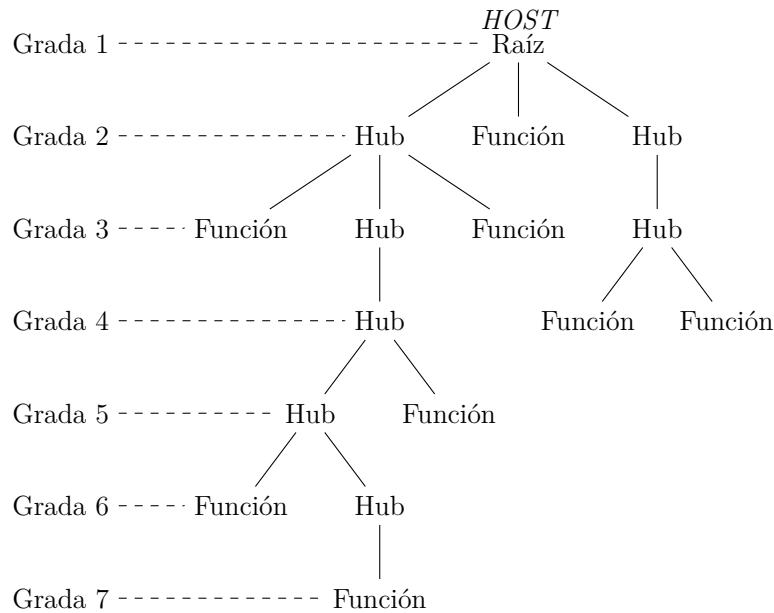


Figura 1.6: Topología de un sistema USB

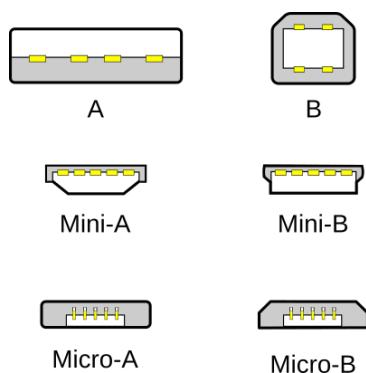


Figura 1.7: Tipos de conectores USB. Los tipo A deben ser usados en el extremo del Host y los tipo B hacia los periféricos[28]

tipo A y los periféricos, de tipo B. Se observan las diferencias entre uno y otro en la Figura 1.7. Los cables de conexión poseen dos pares de conductores: uno para la señal de alimentación de 5 V ( $V_{BUS}$  y  $GND$ ) y otro para el flujo de datos ( $D+$  y  $D-$ ).

A nivel eléctrico, la interconexión de datos en los dispositivos se lleva a cabo a través de una codificación de inversión sin retorno a cero, es decir, el cambio de nivel de tensión representa un '0' y la invariabilidad, un '1'. Además, la señal de datos es diferencial. Esto implica que cuando  $D+$  es positivo,  $D-$  debe ser negativo y viceversa.

Cabe destacar que, al tener una señal diferencial, la norma USB es half-dulpex , es decir, puede transmitir en los dos sentidos (desde Host hacia Funciones y viceversa), pero no puede hacerlo en simultáneo[29], sino que primero debe transmitir un dispositivo y, al finalizar este, el bus queda liberado para que otros dispositivos puedan transmitir.

La velocidad de conmutación en los niveles de tensión de la señal de comunicación puede darse a diferentes valores, dando lugar a tres tipos de tasa de bit: Alta-velocidad (*High-Speed*) implica una tasa de bit de  $480 \text{ Mbit s}^{-1}$ , Velocidad-completa (*Full-Speed*) posee una tasa de bit de  $12 \text{ Mbit s}^{-1}$  y baja-velocidad (*Low-Speed*) transmite a una tasa de bit de  $1.5 \text{ Mbit s}^{-1}$ .

Cuando el Host se comunica con las diferentes Funciones, lo realiza a través de paquetes. Los paquetes implican que la información que se transmite a través del bus está encapsulada en un formato establecido. Cada vez que un dispositivo accede al bus, lo debe hacer de una manera particular, definida por el tipo de transferencia, por su rol (Host, Hub o Función) y por el estado de la transmisión dentro del protocolo establecido.

### **1.3.2. Dispositivos que componen un sistema USB**

Dentro de un sistema USB existen tres tipos diferentes de dispositivos: Host, Hubs y Funciones. Cada uno de ellos tiene asignado un rol específico dentro de la comunicación. Se detallan a continuación las tareas pertinentes a cada uno de ellos.

#### **Host USB**

El Host es quien comanda las comunicaciones. Este dispositivo debe tener capacidades de memoria y procesamiento necesarias para almacenar y ejecutar el software de control. A su vez, necesita de hardware que le permita llevar un monitoreo y control de los eventos que suceden en el bus. Entre las tareas que debe llevar a cabo, se encuentran:

- Detectar la conexión y desconexión de dispositivos.
- Administrar el flujo de los comandos de control con los diferentes dispositivos.
- Administrar el flujo de la información entre él (Host) y los diferentes dispositivos.
- Llevar estadísticas de actividad y estado del bus.
- Proveer potencia a los dispositivos conectados, cuando estos así lo requieran.

Debido a que las tareas que ejecuta el Host requiere una cantidad de recursos de almacenamiento y procesamiento, es usual que el sea una PC la que lleve el rol. El Host es quien inicia la comunicación con las Funciones. Las Funciones, a su vez, responden a lo que fue solicitado por el Host, cuando él lo indique.

## Hubs USB

Un Hub USB tiene la función de proveer puertos al bus. El primer Hub esta incorporado en el Host y cada vez que se requiere más puertos a los cuales incorporar periféricos, se puede ir agregando a través de Hubs. Otra función importante es la de servir como interfaz entre dispositivos con diferentes velocidades, optimizando así el ancho de banda disponible para la comunicación.

## Funciones USB

La norma define como Función a todo aquel dispositivo que se conecta al bus y brinda al Host la capacidad de realizar una nueva tarea. Por ejemplo, un teclado otorga un método de entrada adicional, un mouse permite manejar un puntero de la interfaz gráfica, un parlante y un micrófono posibilitan la emisión y recepción de sonidos, respectivamente. Cada una de estas utilidades, compone una Función USB. A su vez, un dispositivo que brinda más de una capacidad es visto por el Host como Funciones separadas conectadas a través de un Hub. Por ejemplo, si se piensa en unos auriculares con micrófono, aunque se presenten integrados en un mismo producto y tengan un único puerto de conexión al bus, el Host los considera como dos Funciones separadas. Las Funciones, desde un punto de vista de software, son independientes unas de otras, por lo que cuando un programa, llamado cliente, necesita utilizar una de ellas, puede acceder a ésta directamente sin conocer cuantas y cuales funciones diferentes existen en el bus.

Cada Función se compone de un conjunto de extremos. Un extremo es una porción de dispositivo identificable en forma unívoca<sup>[27]</sup>. Cada extremo tiene características definidas por el diseñador del sistema que deben estar adecuadas a los requerimientos de cada dispositivo. Los extremos tienen un solo sentido de comunicación y un tamaño máximo de mensaje a transmitir o recibir. Cuando se conecta al bus, un dispositivo debe enviar una descripción en donde consten sus extremos y las diferentes formas de configuración de cada uno, con el tipo de mensajes que soporta, el sentido de la comunicación, el tamaño, entre otros parámetros. Esta descripción se lleva a cabo través de lo que la norma llama descriptores.

Todo dispositivo debe contener un extremo con dirección cero dedicado exclusivamente al control de la Función por parte del Host. Debe, como mínimo, poder comunicarse a velocidad completa, es decir, con una señal de  $12 \text{ Mbit s}^{-1}$  y, a su vez, responder a los comandos de control básicos cómo adquirir la dirección, recibir la configuración y enviar los descriptores del dispositivo y sus diferentes configuraciones. Dependiendo de los diferentes requerimientos, el dispositivo puede incorporar otros extremos (15 de entrada y 15 de salida como máximo). Cada extremo no-cero tiene diferente latencia, acceso al bus, ancho de banda, manejo de errores, tamaño máximo de paquete soportado y dirección.

### 1.3.3. Paquetes USB

Los dispositivos transmiten información a través del bus con un formato particular, establecido por el protocolo que dicta la norma USB. Cada 1 ms, el Host debe emitir una señal de sincronismo. El intervalo que transcurre entre una señal y la siguiente, se denomina cuadro. El Host asigna una porción de cuadro a cada uno de los dispositivos, asignando ancho de banda y tiempos de retardo a cada uno, según los requerimientos. A su vez, en comunicaciones de Alta-Velocidad, cada

cuadro se subdivide en 8 microcuadros de 125 µs cada uno. Los fragmentos de información que envían los dispositivo mientras transcurre un cuadro, se denominan paquetes. Un paquete está compuesto por diferentes campos. El sistema reconoce cada campo, decodifica su información e identifica cada paquete, su emisor, el tipo de datos que envía, el sentido de circulación. Luego, corrobora que los datos transmitidos llegaron a destino en forma satisfactoria.

## Campos de paquetes

Existe un número finito de campos y todos pueden resumirse en el presente documento. Sin embargo, se detallan a continuación los que el autor considera más relevantes para el objetivo de este trabajo, quedando de lado algunos comandos, por ejemplo, inherentes a los hubs que conectan dispositivos de diferentes velocidades.

- **Identificador de paquete:** El campo identificador de paquete (PID del inglés *Packet Identifier*) le da a conocer a los distintos dispositivos el tipo de información que contiene el paquete. Por ejemplo, indica si el Host solicita envío o recibo de datos, si envía un comando o si un dispositivo está transmitiendo los datos. Se compone de un campo de 8 bits, de los cuales 4 corresponden al identificador propiamente dicho y los otros cuatro son el complemento a uno de los mismos datos, permitiendo corroborar que no hubo una pérdida de información.

Existen 4 tipos de PID: Token, que antecede a cualquier transmisión y es emitido por el host; Data, indica paquetes que contienen datos transmitidos; Handshake, a través del cual los componentes del sistema se enteran si la comunicación fue efectiva o no y Special, cuya función no es de interés para este trabajo.

A su vez, los PID Token se dividen en 4 tipos: IN, para indicar que se va a realizar una envío de datos desde un extremo al Host; OUT, antecede a una transmisión de datos en el sentido contrario, es decir del Host a un extremo; SETUP, que señala una secuencia de comandos y SOF (del inglés Start of Frame) que emite una señal de inicio de cuadro, utilizada para sincronismo y control.

Dentro de los PID Data, solo existen diferentes etiquetas que se usan dependiendo del tipo de transmisión. Los PID de Handshake contienen 4 mensajes diferentes: ACK para indicar que el mensaje fue recibido satisfactoriamente y NAK señala que no se pudo enviar o recibir, STALL significa que el extremo se detuvo y NYET de cuenta sobre demoras en la respuesta del receptor.

- **Dirección:** El campo de Dirección señala cuál es la Función que debe responder o recibir alguna directiva emitida por el host. A su vez, se divide en dos subcampos: uno que indica un dispositivo y la segunda que señala el extremo específico con el cual desea comunicarse.
- **Datos:** Es el campo que contiene la información útil transferida. Puede tener un largo de hasta 1024 bytes. Cada byte enviado se ordena con el bit menos significativo (LSb del inglés *Less Significative bit*) primero y el bit mas significativo (MSb por sus siglas en inglés) al final.
- **Chequeos de redundancia cíclica:** El campo de chequeo de redundancia cíclica (CRC) contiene verificadores para corroborar que no hubo pérdida de información. Dependiendo de que tipo de paquete se esté transmitiendo, el CRC puede tener 5 o 16 bits.

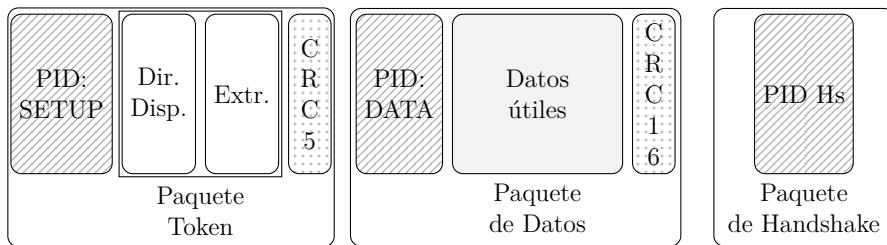


Figura 1.8: Formatos de paquetes

### Formato de paquetes

Cada uno de los paquetes que intervienen en la comunicación USB utilizan diferentes tipos de campos, dando lugar a distintos tipos de paquetes. La figura 1.8 muestra como se conforman algunos de ellos.

Un paquete de tipo Token está conformado por los campos PID, Dirección y CRC-5 (CRC de 5 bits). Un paquete Token que indica SOF en su campo PID, lleva un formato un poco diferente. En lugar de la dirección, se envía un contador de 11 bits que señala la cantidad de cuadros que han transcurrido desde la puesta en marcha del sistema, seguido de un código CRC-5.

Cada 1 ms el host transmite un SOF e incrementar el contador de cuadros. En sistemas USB 2.0 de Alta velocidad, además, se transmiten 8 subcuadros de 125  $\mu$ s por cada cuadro. Cada uno de estos subcuadros inicia con un paquete SOF. Sin embargo, el host no actualizará el número de cuadros hasta pasado 1 ms.

El paquete de datos iniciará con un PID que indique que es un paquete de este tipo, luego enviará los datos desde el LSb hasta el MSb y, finalmente, enviará un código CRC-16 (CRC de 16 bits de longitud).

Los paquetes de tipo Handshake (Hs) solo envía un PID con información sobre si el mensaje fue recibido en forma correcta o no.

#### 1.3.4. Tipos de Transferencias

Cada extremo presente en un dispositivo USB, puede estar configurado, en simultaneo, con un solo tipo de transferencias. Es importante, para el diseñador del dispositivo, entender y seleccionar el tipo de transferencia adecuada para cada uso debido a que, de ello depende las características que poseerán las comunicaciones que se efectúen.

Existen cuatro tipos de transferencias definidas por la norma USB: Transferencias de Control, transferencias en masa, transferencias isocrónicas y transferencias de interrupción. Cada una de ellas tiene un propósito y características diferentes, las que se detallan a continuación.

#### Transferencias de control

Las transferencias de control son utilizadas por el Host para configurar, emitir comandos y conocer el estado de los distintos dispositivos acoplados al bus. Se caracteriza por ser una comunicación de ráfagas, es decir, de corta duración, tener alta prioridad y ser no periódica. Habitualmente, son utilizadas para emitir comandos hacia los dispositivos, o bien, para conocer su estado. Sin embargo, esto no quiere decir que pueda ser empleada para transmitir mensajes que no sean específicamente de comando. Debido a la sensibilidad que los mensajes de control

poseen para el sistema USB, estos están dotados del protocolo más estricto de chequeo, corrección y/o retransmisión de datos.

Las transferencias de control poseen dos o tres etapas en su ejecución. En la primera de ellas, el Host debe enviar un Paquete Token que indique SETUP, luego envía un paquete Data con 8 B y esto es respondido por el dispositivo con un paquete Handshake indicando la recepción. Si hiciese falta enviar información extra, en una segunda etapa, el Host transmitirá un paquete Token indicando la necesidad de información. Luego, dependiendo del sentido de los datos solicitado, se enviará un paquete Data con hasta 64 B más y el receptor responderá con un paquete Handshake. Finalmente, en la última etapa, se le permite al dispositivo informar su estado. Para ello, el Host le envía un paquete Token de solicitud de datos, luego la Función responderá con un paquete Data y el Host emitirá con un paquete Handshake, indicando si recibió o no la información.

## Transferencias en masa

Las transferencias en masa son usadas para transferir paquetes grandes en forma de ráfagas, en forma no periódica. Su utilidad consiste en que aprovecha al máximo cualquier espacio de ancho de banda disponible. Gracias al sistema de chequeo de errores, es posible solicitar retransmisiones, asegurando la integridad de la comunicación. Esta transferencia es ideal para comunicar cantidades relativamente grandes de datos que requieren una comunicación fidedigna a costa de sacrificar velocidad en los tiempos de entrega, por ejemplo, una impresora. En un bus que no posee un gran uso, los mensajes alcanzarán el destino en tiempos cortos. Sin embargo, cuando exista una gran cantidad de dispositivos conectados y el ancho del bus se encuentre congestionado, un mensaje largo puede verse demorado.

Cuando se lleva a cabo una operación de este tipo, el Host envía un paquete Token de tipo OUT cuando desea transmitir datos o IN si desea recibirlos, la dirección de la Función y su extremo. Luego, el emisor comunica un paquete Data, y finalmente, el receptor de la transferencia responde con un paquete Handshake. Una transferencia en masa (*bulk transfer*) puede poseer un tamaño máximo de 512 B de datos por paquete transmitido.

## Transferencias isocrónicas

El término isócrono o isocrónico está referido a sistemas digitales sincrónicos con la particularidad de que se supone que sucede una cantidad determinada de sucesos en intervalos regulares de tiempo. Esto puede ser logrado compartiendo la misma fuente de sincronismo, o bien, sincronizando los relojes de cada componente.

En un sistema USB, el Host envía una señal SOF por cada cuadro de 1 ms y por cada subcuadro de 125 µs, en los sistemas de alta velocidad. Es posible sincronizar sistemas que poseen fuentes de reloj diferentes a través de la captura de esta señal. Esto permite tener este comunicaciones de tipo isocrónico, aún con señales de reloj provenientes de fuentes diferentes.

La principal característica de las transferencias isocrónicas es que son periódicas y continuas entre el Host y las Funciones. Se utiliza este tipo de transferencias para comunicar datos que pierden validez cuando no son entregados en un tiempo establecido. Para lograr esto, el Host asigna una porción fija de ancho de banda por cada cuadro (1 ms) a cada dispositivo que se comunique por transferencias de tipo isocrónicas. Gracias a que los datos pierden su validez a lo

largo del tiempo, también los errores la pierden, por lo que no se prevé una retransmisión de los datos enviados por este sistema.

La ejecución de una transferencia isocrónica se da cuando el host envía un paquete Token con la dirección de un extremo de este tipo de transferencias. Luego, el emisor envía un paquete Data cuyo campo de datos puede poseer hasta 1024 B y un CRC-16. Finalmente el receptor envía un paquete Handshake. Si, dado el caso, el receptor envía un Handshake indicando que el paquete no pudo ser recibido en forma correcta, el mensaje es descartado, sin existir una retransmisión posterior del mismo paquete.

### Transferencias de interrupción

Cuando se requiere de una comunicación cuya demora en la entrega de datos sea menor que un tiempo máximo y que, a su vez, posea una baja probabilidad de ocurrencia, el tipo de transferencia óptimo para utilizar, son las transferencias de interrupción. En este tipo de trasferencias, el Host consulta cada un periodo de tiempo determinado el estado de los extremos que se encuentran configurados para efectuar este tipo de transferencias. Para ello, envía un paquete Token, luego el emisor transmite un paquete de datos con hasta 64 B, si se trata de dispositivos de velocidad completa, y 1024 B, en el caso de una comunicación de alta velocidad. Finalmente, el receptor responderá con un paquete Handshake.

#### 1.3.5. Descriptores

Cuando un dispositivo es conectado al bus, debe informar sus características al Host a través de descriptores. Un descriptor es un estructura de datos con formato definido. De esta forma, el sistema conoce las diferentes configuraciones que puede tener cada una de las Funciones conectadas. El conocimiento detallado de estos descriptores por parte de los diseñadores de dispositivos, facilita luego la tarea de selección de cada uno de los atributos que tendrá, como así también, la elaboración de software de control en la PC.

Cada uno de los descriptores comienzan con su longitud en bytes y el tipo de descriptor que se está enviando. En orden jerárquico, se utilizan categorías de descriptores que van desde los atributos generales a los particulares. En primer lugar, se envía el descriptor DEVICE que informa la versión de la norma USB que cumple el dispositivo, un numero que identifica al fabricante y otro que corresponde al producto, es decir al dispositivo. Esto le permite saber al Host que software de control debe utilizar para comunicarse con el dispositivo. A su vez, comunica la cantidad de posibles configuraciones. Luego, si el dispositivo cumple con la norma 2.0 (o más moderna) envía un descriptor de tipo DEVICE\_QUALIFIER con información sobre otras velocidades de comunicación soportadas.

El protocolo USB diferencia una configuración de otra dependiendo de las necesidades de energía. Un dispositivo podría operar conectado a una fuente de energía externa, o bien, ser alimentado por el mismo bus. Si las potencias de la fuente y del bus son diferentes, podrían verse limitadas las utilidades que ejecutaría la Función. Entonces, cuando el dispositivo funcione con la fuente podría tener una configuración pero cuando se desconecta, deberá informar esta situación al Host, indicando que se debe cambiar la configuración. Esta comunicación se lleva a cabo a través del descriptor de tipo CONFIGURATION. Debe haber tantos descriptores de este tipo como se indicó en el descriptor DEVICE.

Debido a que cada configuración puede tener diferentes limitaciones en sus funciones dependiendo de la potencia que consuma, se establece que cada configuración tenga a su vez diferentes interfaces. La cantidad de interfaces que tiene una configuración, también debe estar informada en el descriptor CONFIGURATION.

Una interfaz puede verse como el conjunto de extremos que son utilizados por un dispositivo para realizar una función específica. Por ejemplo, se podría pensar en una impresora multifunción. Se puede tener una interfaz para la parte que imprime y otra para el scanner. A su vez, cada interfaz puede variar el ancho de banda requerido a través de los denominados AlternateSettings. Las interfaces y sus diferentes alternativas, se comunican al Host a través del descriptor de tipo INTERFACE.

A su vez, un extremo define la dirección de la comunicación, es decir, si es desde o hacia el Host, un tipo de transferencia, si la comunicación es sincrónica o no, el tamaño máximo de paquete y el ancho de banda necesario. Los extremos se describen a través del descriptor ENDPOINT.

En resumen, la comunicación entre los dispositivos y el Host se efectúa a través de los extremos. Los extremos, a su vez, se agrupan en interfaces y un grupo de interfaces conforman una configuración. Una característica a tener en cuenta es que un dispositivo puede tener diferentes interfaces activas a la vez y las interfaces pueden cambiar durante la operación de características alternativas (AlternateSettings). Sin embargo, para cambiar de configuración, todos los extremos y las interfaces se desactivan.

También existe un tipo de descriptores, denominados STRING, que sirven para colocar a cada uno de los atributos una forma legible por el usuario, aunque puede no ser utilizada.

## **1.4. Objetivos**

### **1.4.1. Objetivo Principal**

El objetivo del presente trabajo es obtener una comunicación USB 2.0 de alta velocidad entre una PC y un FPGA.

Esta comunicación debe realizarse y documentarse de forma tal que pueda ser usado posteriormente en aplicaciones científicas desarrolladas con FPGA's.

### **1.4.2. Objetivos Particulares**

Para la consecución del objetivo general, se deben cumplir los siguientes objetivos particulares:

- Comprender el funcionamiento del protocolo USB.
- Seleccionar los componentes a utilizar.
- Configurar los componentes seleccionados.
- Desarrollar un núcleo en VHDL que sirva de interfaz.
- Diseñar e implementar la interconexión de los componentes seleccionados.
- Verificar el sistema desarrollado.
- Desarrollar un documento que explique el modo de uso del código VHDL utilizado.

## 1.5. Estructura del Informe

El presente informe se divide en 2 bloques principales: uno referido al desarrollo del sistema y el siguiente a su forma de uso y verificación.

Dentro del bloque referido al desarrollo del sistema, se encuentran los primeros 5 capítulos:

1. **Introducción:** En este capítulo se intenta exponer lo que motiva el presente trabajo, la propuesta que da solución a la motivación, el objetivo y alcance que el trabajo busca y la estructura del mismo. Se brindan, además, conceptos importantes de la norma USB que son significativos para los objetivos de este trabajo.
2. **??:** Se describe aquí todas las herramientas de las que se vale este trabajo para cumplir con los objetivos propuestos.
3. **Interfaz USB:** Se presenta la arquitectura, configuración y código desarrollado para el presente trabajo, como así también las herramientas específicas provistas por el fabricante, que facilitan el desarrollo.
4. **Desarrollo del sistema maestro para la comunicación entre la FPGA y la interfaz Cypress:** Este capítulo detalla lo desarrollado para implementar la comunicación entre la FPGA y la interfaz. Se expone una máquina de estados descrita en VHDL y sintetizada en FPGA. También se describe un circuito impreso realizado para conectar ambas partes.
5. **Pruebas de funcionamiento y desempeño del sistema desarrollado:** Se desarrolla las tareas desarrolladas a fin de realizar las depuraciones del sistema y la verificación del cumplimiento de las especificaciones.

## 1.6. Sumario del capítulo

En el presente capítulo se expone la necesidad de la elaboración de un sistema de comunicación que permita la transferencia de datos entre una PC y un FPGA para ser utilizados por sistemas implementados con este último dispositivo.

Se plantea una solución utilizando una interfaz comercial que sirve de intermediario entre estas herramientas y se brinda una justificación del empleo del protocolo USB 2.0 de alta velocidad como la implementación óptima del sistema.

Se presenta también la estructura del presente informe y se dan detalles relevantes para este trabajo de la norma USB.



# Capítulo 2

## Interfaz USB

Como se menciona en la Sección 1.2, la comunicación USB fue implementada a través de una interfaz entre un FPGA y una PC. Para cumplir el rol de interfaz, se utilizó el controlador EZ-USB FX2LP de Cypress, el cual viene incorporado en el kit de desarrollo CY3684.

El kit de desarrollo CY3684 puede ser descompuesto en dos partes: una de hardware, que posibilita la conexión eléctrica entre los componentes y una parte de software que facilita al desarrollador tanto la elaboración del programa que es cargado y ejecutado por el microcontrolador (denominado firmware), como las pruebas del sistema en desarrollo.

En este capítulo se justifica la selección de la interfaz, se presenta la configuración que mejor se adapta a los objetivos, se detalla el firmware elaborado y se abordan algunos aspectos conceptuales sobre la estructura y arquitectura del circuito integrado seleccionado como interfaz y las herramientas utilizadas.

### 2.1. Elección de la Interfaz

La parte central del sistema desarrollado en el presente trabajo está constituida por el módulo de interfaz entre el FPGA y la PC. Dicho módulo tiene como tarea la de comunicarse con un PC a través del protocolo USB 2.0, decodificando los paquetes que arriban, comprobando que lleguen sin errores, separando la información del protocolo USB (encabezado y cola), de la que es útil para el sistema implementado en un FPGA. Además, debe poder escribir los datos en el FPGA con un protocolo más simple para utilizar menos recursos programables del mismo. También debe efectuar el camino inverso de comunicación, es decir leer datos del FPGA, colocar la información que requiere el protocolo y transmitir los paquetes hacia la PC.

En el mercado de componentes electrónicos, existen dos fabricantes que ofrecen interfaces USB (también llamadas puentes USB). FTDI y Cypress exhiben en sus catálogos, sendas líneas de productos que proveen circuitos integrados que podrían servir a los fines del desarrollo buscado. Durante la elaboración de este trabajo, se evaluó la alternativa que más se ajusta a las necesidades del sistema desarrollado que brinda cada uno de estos proveedores.

El chip FT4222H de FTDI es un puente USB relativamente simple de configurar, ya que no es necesario elaborar software adicional para que ejecute las tareas relativas a la comunicación. Hacia el lado de los periféricos, la comunicación se realiza mediante el protocolo SPI, con un reloj de hasta 30 MHz. Es posible alcanzar la velocidad máxima permitida por el protocolo USB mediante el uso de cuatro puertos SPI.



Figura 2.1: Circuito impreso principal del kit de desarrollo CY3684 EZ-USB FX2LP

Por su parte, la línea de circuitos integrados FX2/FX2LP de Cypress brinda controladores USB muy versátiles y potentes. Los puentes USB poseen, como interfaz hacia los periféricos, un conjunto de memorias FIFO a las que se puede acceder por un puerto paralelo de 16 bits de ancho de bus, que pueden operar a 48 MHz. También incorporan un microcontrolador 8051, a través del cuál se implementa el protocolo USB y puede ser utilizado por el usuario para implementar sistemas adicionales.

Se escoge entonces, para el desarrollo del sistema de comunicación, el controlador FX2/FX2LP de Cypress en lugar de la interfaz fabricada por FTDI ya que el uso de un puerto cuádruple SPI posee una implementación un poco más costosa, en términos de lógica programable, que la comunicación de un puerto paralelo de 16 bits.

Cypress comercializa un kit de desarrollo destinado al diseño de sistemas basados en la serie de controladores FX2LP. Dicho kit de desarrollo se denomina CY3684 EZ-USB FX2LP. El kit posee una placa de desarrollo como la que se observa en la Figura 2.1. El componente principal del kit es el controlador EZ-USB FX2LP e incorpora un display de 7 segmentos, 4 luces led multipropósito, 6 pulsadores, de los cuales 4 son de propósito general, uno de reinicio y otro que envía una señal especial para salir de un modo de bajo consumo. También tiene dos bloques de memorias EEPROM destinadas al almacenamiento del firmware (programa que ejecuta un microcontrolador), lo que otorga la posibilidad de realizar una carga no volátil de la configuración del controlador, memoria flash con una capacidad de 64 kB que es utilizada para almacenar el programa del controlador, un puerto USB y dos UART con zócalos DE-9. Adicionalmente, cuenta con 6 puertos de 20 pines y uno de 40, compatible con el protocolo ATA, que permiten comunicarse con el controlador.

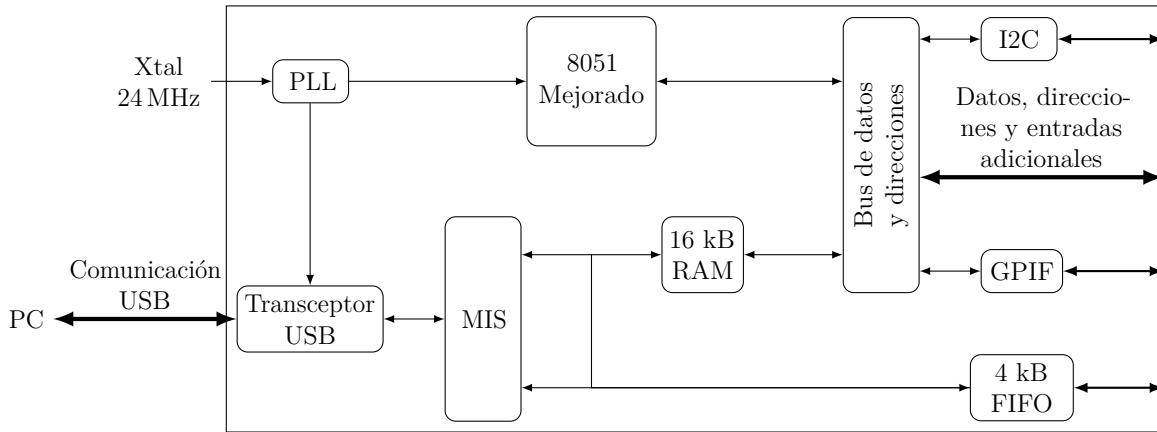


Figura 2.2: Arquitectura FX2LP

## 2.2. El controlador FX2LP EZ-USB y su configuración

Como se menciona anteriormente, el núcleo del kit de desarrollo CY3684 es el controlador EZ-USB FX2LP. La serie de controladores FX2LP se caracteriza por brindar una conexión USB 2.0 de alta velocidad y bajo consumo energético. Está diseñada, preferentemente más no exclusivamente, para periféricos con autonomía limitada.

La arquitectura de controlador FX2LP, tal como se presenta en la Figura 2.2, integra un controlador USB completo, es decir, incluye un transceptor USB, un Motor de Interfaz Serie (MIS) y buffers de datos configurables. Incorpora también una versión del microcontrolador ( $\mu$ C) Intel MCS-51, más conocido como el  $\mu$ C 8051, que contiene registros y funciones adicionales orientadas a mejorar el rendimiento de la comunicación USB y memoria RAM de 16 kB de capacidad, para almacenar programas y datos. El modelo del flujo de datos posee dos extremos entre las cuales el controlador cumple el rol de interfaz. Estos extremos son la PC y el FPGA respectivamente. El controlador necesita, entonces, poder comunicarse tanto con el Host como con los periféricos. Para este propósito, Cypress agrega al circuito integrado del controlador dos puertos USART ((acrónimo de Transmisión y Recepción Asíncrona en Serie Universal, en inglés)), una interfaz de propósito general (GPIF), un puerto I<sup>2</sup>C y una memoria FIFO (*FirstInFirstOut*; Primero Entrado, Primero Salido).

La GPIF está pensada principalmente para poder utilizar sistemas que deban ser comandados en forma externa, como por ejemplo un registro de desplazamiento. Por su parte, la memoria FIFO posee 4 kB de capacidad reservados para almacenar los datos que se intercambian y se destina a aquellos sistemas que pueden proveer las señales de control, aunque también puede ser comandada por el GPIF. Con estas interfaces se posibilita la conexión con casi cualquier dispositivo, ya sea estandarizado (ATA, PCMCIA, EPP, etc) o personalizable (DSP, FPGA,  $\mu$ C);

Bajo el criterio de este autor, el componente de mayor trascendencia en el funcionamiento del controlador FX2LP es el  $\mu$ C 8051. Es este componente el encargado de configurar los bloques programables y de inicializar todos los registros que determinan la forma en la que el sistema funciona: la frecuencia de trabajo, la gestión de las memorias y el modo en que fluyen los datos son algunas de las tareas que configura el  $\mu$ C. El firmware es escrito en lenguaje C para microcontroladores.

Durante el desarrollo de la interfaz que se implementó en este trabajo, se utilizó la memoria FIFO en modo esclavo, es decir, que responde a señales que proporciona un maestro externo sintetizado en un FPGA. Se escogió la frecuencia de funcionamiento del PLL y se configuraron los extremos que intervienen en la comunicación USB y el modo de funcionamiento, por lo que a continuación se explicitan los detalles referidos a la configuración realizada, con lo que se busca aclarar el funcionamiento y que el lector comprenda los fundamentos de las configuraciones que se plasman en el código del firmware.

### 2.2.1. Microcontrolador Cypress 8051 Mejorado

Las tareas que ejecuta el controlador FX2LP son llevadas a cabo por un microcontrolador incorporado al circuito integrado. Dicho  $\mu$ C es una modificación del 8051 desarrollado por Intel, para que sea más veloz en sus tiempos de ejecución y mejore el desempeño del  $\mu$ C como interfaz, mediante la incorporación de registros especiales adicionales. De esta forma, la manera a través de la cual el desarrollador elabora la configuración del controlador, es a través de la programación de este  $\mu$ C 8051.

Para elaborar el firmware que ejecuta el controlador FX2LP, se desarrolló un programa en C para microcontroladores y se compiló mediante el compilador C51 de Keil, a través del entorno de desarrollo integrado Keil  $\mu$ Vision.

Cypress provee, dentro del kit de desarrollo CY3684, un conjunto de archivos que contienen código base sobre el cual el desarrollador implementa la configuración. Este conjunto de archivos es denominado framework, el cual posee, entre otras cosas, encabezados con definiciones de macros, constantes, registros, tipos de datos y declaración de funciones prototipo. También incorpora algunas funciones precompiladas para utilizar los periféricos que contiene la placa de desarrollo.

La Figura 2.3 muestra un diagrama de flujo del firmware que se desarrolló en el presente trabajo. El mismo se elaboró utilizando la estructura propuesta por Cypress para el desarrollo de la comunicación que se implementó. Se puede observar que al inicio del programa se inicializan variables de estado que corresponden a una máquina de estados, desarrollada por Cypress, que ejecuta las tareas de la comunicación USB.

Luego, se invoca una función llamada `TD_Init()`. Esta es la función a través de la cual se implementa la configuración que se desarrolló en este trabajo. En las secciones siguientes se profundiza cada uno de los bloques que intervienen.

Una vez configurado el funcionamiento del controlador, se habilitan las interrupciones, lo que da lugar a que todas los bloques del circuito integrado puedan funcionar e intercambiar información. Seguidamente, el programa entra en un lazo infinito, donde en primer lugar ejecuta la función `TD_Poll()`, en la cual el desarrollador programa las tareas que ejecuta el controlador durante la rutina de funcionamiento. Como segundo paso, el controlador chequea si arribó desde el Host una transferencia de control cuyo PID indique Setup. En caso afirmativo, ejecuta lo solicitado por el Host. En caso contrario, vuelve a ejecutar la función `TD_Poll()`.

### 2.2.2. Frecuencia de trabajo del sistema

Como se menciona en la sección anterior, la configuración principal del sistema se realiza a través de la función `TD_Init()`. El primer módulo configurado es el PLL (*Phase-Locked Loop*). Un PLL es un lazo de servocontrol cuyo parámetro controlado es la fase de una réplica, generada

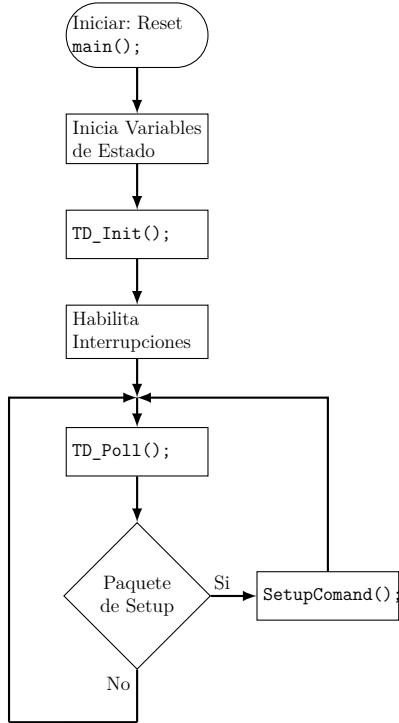


Figura 2.3: Diagrama en bloques del firmware que ejecuta el  $\mu$ C de la interfaz

en forma local, de una señal de entrada[30]. En otras palabras, permite obtener dos señales iguales a través de un detector de fase. Si se incorpora un contador entre la señal generada y la entrada del comparador de fase, la señal generada tendrá una frecuencia igual al producto de la entrada por el recorrido del contador. Si, en cambio, se coloca el contador a la salida del PLL, la frecuencia puede ser dividida. Así, es posible obtener señales de frecuencia modificable.

El PLL incorporado en el controlador permite elevar la frecuencia de un cristal de 24 MHz hasta los 480 MHz que necesita el transceptor USB para el cumplimiento de la norma USB. A su vez, a través de un divisor de frecuencias, permite seleccionar diferentes frecuencias de trabajo del  $\mu$ C 8051, entre 12, 24 o 48 MHz.

A través de los bits especiales CLKSPD[1:0] del registro de Control y Estado de CPU (CPUCS). En la implementación realizada, se seleccionó la frecuencia de trabajo del  $\mu$ C a 48 MHz.

```

//CPUCS – Registro de Control y Estado del CPU
// CLKSPD[1:0] -> "00" => 12 MHz
//           -> "01" => 24 MHz
//           -> "10" => 48 MHz
CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1); // 48 MHz
  
```

### 2.2.3. Memoria FIFO

El controlador FX2LP posee una sección especial de memoria destinada al almacenamiento de los datos que fluyen desde cada uno de los extremos de la comunicación. A esta memoria pueden acceder tanto los componentes del propio controlador, como también los periféricos que

desean comunicarse a través de él. Desde el punto de vista de la electrónica digital, cada uno de los componentes que acceden a esta memoria pueden tener diferentes fuentes de señal de reloj. Para salvar los inconvenientes que puede acarrear el uso de sistemas con fuentes de reloj independientes, esta porción de memoria reservada es de tipo FIFO. Debido a que se puede acceder a estas memorias FIFO tanto desde el interior de controlador FX2LP, como desde el exterior, deben ser configuradas en ambos sentidos.

La memoria FIFO puede ser programada y configurada de diferentes formas, en función de los requerimientos sistemas periféricos acoplados a ella. Cada uno de los periféricos conectados a la memoria FIFO se denomina extremo o EP<sup>1</sup>. Las características a configurar son el tamaño (64, 512 o 1024 B), la cantidad de bloques o partes en que se divide la memoria (puede estar dividida hasta en 4 extremos) y la cantidad de buffers de datos utilizados para almacenar los datos de cada bloque de memoria.

Los buffers son porciones de memoria físicamente separadas pero que, en la operación, el controlador puede intercambiar de forma tal que se acceda a ellos a través de una misma dirección de memoria. El uso de buffers múltiples implica que un EP utiliza más de un buffer. Los buffers múltiples poseen la función de evitar la congestión de datos. Con doble buffer, un periférico coloca o extrae datos del buffer de un EP, mientras el  $\mu$ C, utiliza otro del mismo EP. La selección del buffer donde cada componente escribe y/o lee los datos lo asigna e intercambia la interfaz en forma automática. Se pueden configurar también un triple o cuádruple buffer, lo que agrega sendas porciones de memoria extra a la reserva. De esta forma, se le otorga al sistema, en forma simultánea, gran capacidad de datos y ancho de banda.

En este desarrollo, se configuró la memoria FIFO con dos EP. El EP<sup>2</sup>, es un EP de entrada (envía datos al Host). Requiere una gran cantidad de datos, debido a que será por donde los sensores transmitirán todos los datos que adquieran. Además, es necesario que posea una buena cantidad de almacenamiento de datos y que estos datos sean enviados de la forma más rápida posible. Por tanto, el EP2 se configuró con dos buffers de 1024 B, para que efectúe transferencias Isocrónicas.

Por su parte, se configura el EP8 como EP de salida (recibe datos desde el Host). Este EP se utiliza para recibir la configuración de los sensores, que se espera que sea de menor cantidad y más distanciada en el tiempo que los datos adquiridos. Se configuró, entonces, con dos buffers de 512 B para transferencias en masa.

Debido a que la memoria FIFO cumple el rol de interfaz entre los periféricos y el módulo del controlador FX2LP que efectúa las tareas propias de la comunicación USB, la configuración de dicha memoria se efectúa por separado, conteniendo información relevante a cada etapa de la comunicación.

## Interfaz hacia los periféricos

Cypress provee varias interfaces para comunicar el controlador hacia los periféricos. I<sup>2</sup>C y UART son dos posibilidades, aunque poseen un ancho de banda muy limitado. La interfaz que opera con mayor ancho de banda es la memoria FIFO. Esta puede ser utilizadas en modo esclavo, es decir, que un sistema externo comande la lectura y la escritura de datos en ellas, o

<sup>1</sup>EP es una abreviación del término inglés *endpoint*, que significa “Extremo”. Esto quiere decir que cada uno de los periféricos conectados a la memoria FIFO es un extremo de la comunicación.

<sup>2</sup>EP con dirección 2.

bien, a través de la interfaz GPIO, puede ser comandada por el  $\mu$ C 8051. La implementación que se realiza en el desarrollo de la comunicación utiliza la memoria FIFO en modo esclavo.

La frecuencia de funcionamiento de estas interfaz es independiente del reloj del sistema. Puede ser configurado para usar una señal de reloj interna de 30 o 48 MHz, propia de la interfaz, o bien, ser provista por un sistema externo al controlador. También, es importante indicarle al controlador si la interfaz funcionará en modo asíncrono. Todos estos parámetros son configurados a través del registro Configuración de Interfaz (IFCONFIG).

La configuración que se realizó en esta implementación, utiliza el reloj interno de la interfaz, corriendo a 48 MHz. Además, se indica que las memorias FIFO esclavas son utilizadas en modo asíncrono. Dicha configuración se plasma en las siguientes líneas de código:

```
//IFCONFIG - Registro de Configuración de la Interfaz
// b7      -> fuente de reloj: '1' interna , '0' externa
// b6      -> freq: '1' 48 Mhz, '0' 30 MHz
// b3      -> asinc: '1' asíncrono
// b[1:0] -> modo de interfaz: "11" FIFO esclava
IFCONFIG = 0xCB;
SYNCDELAY;
```

El controlador FX2LP posee cuatro puertos que emiten señales del estado de las memorias FIFO. Estos puertos pueden ser programados para que indiquen si una porción particular de memoria se encuentra vacía, llena o si sobrepasa un nivel programable de datos. También pueden ser configurados para que indiquen el estado completo (vacío, lleno y el nivel programable) de la porción de memoria activa. Cada porción de memoria se activa a través de dos puertos de dirección, comandados por un sistema externo al controlador FX2LP.

Para la comunicación desarrollada, solo son importantes las señales que indican cuando el puerto que se corresponde al EP8 está vacío y el que señala al EP2 está lleno. Si bien no son necesarias, por completitud, también se configuraron las señales EP2 vacío y EP8 lleno. Cada uno de los puertos de señal se denominan A, B, C y D y se configuran por pares a través de los registros PINFLAGSAB y PINFLAGSCD, de la forma en que se muestra a continuación.

```
PINFLAGSAB = 0xBC;    // FLAGA <- EP2 Full Flag
                     // FLAGD <- EP2 Empty Flag
SYNCDELAY;
PINFLAGSCD = 0x8F;    // FLAGC <- EP8 Full Flag
                     // FLAGB <- EP8 Empty Flag
```

## Interfaz hacia el módulo de comunicación USB

Desde el extremo interno del controlador FX2LP, la memoria FIFO se conecta al Motor de Interfaz Serial (MIS). El MIS es un módulo que se encarga de tomar datos en paralelo y convertirlos en una secuencia seriada. Para cumplir con la norma USB, el MIS debe ser capaz de empaquetar, enviar, recibir y desempaquetar toda la información, así como leer los tokens que emite el host, calcular y corroborar los códigos cíclicos de detección de errores y todo lo relacionado al protocolo propiamente dicho. Luego, el transceptor USB efectúa las tareas de codificación y decodificación de los mensajes transmitidos a través del bus.

Para la configuración, es necesario indicarle al controlador FX2LP el funcionamiento que tendrá cada uno de los EP. Los parámetros programables son: si está activo o no, el sentido de

la comunicación (sea hacia o desde el Host), el tipo de transferencia, el tamaño de la misma y la cantidad de buffers múltiples que se utilizan. En el desarrollo que se presenta se configura el EP2 como entrada de 1024 B con dos buffers y el EP8 como salida con dos buffers de 512 B. También se configura el EP1 con un buffer de 64 B como entrada y otro igual como salida, ya que viene implementado en una memoria separada dentro del circuito integrado FX2LP y no interfiere con el desempeño pretendido en este trabajo. Los otros EP válidos (EP4 y EP6) no se utilizan, con el objetivo de maximizar la memoria disponible para los datos útiles. De esta forma, la configuración se realiza a través de la siguiente línea de código:

```
//EPxCFG – Registros de configuración de extremos
// b7      -> '1' EP activo
// b6      -> dir: '0' salida , '1' entrada
// b[5:4] -> tipo: "01" => isocronico
//           "10" => masa
//           "11" => interrupción
// b3      -> tamaño: '0' 512 bytes , '1' 1024 bytes
// b[1:0] -> buffer:   "00" => x4
//           "10" => x2
//           "11" => x3
EP1OUTCFG = 0xA0;
SYNCDELAY;
EP1INCFG = 0xA0;
SYNCDELAY;
// dir:entrada , tipo:isoc , tam:1024 , x3
EP2CFG = 0xDB;
SYNCDELAY;
EP4CFG = 0x7F; //Inactivo
SYNCDELAY;
EP6CFG = 0x7F; //Inactivo
SYNCDELAY;
// dir:salida , tipo:masa , tam:512 , x2
EP8CFG = 0xA2;
SYNCDELAY;
```

## 2.2.4. Modos de entrada y salida automáticos

Los datos se reciben o envían a través del MIS. Dichos datos, pueden ser enviados en forma automática desde y hacia las memorias FIFO, o bien, pueden ser dirigidos hacia el  $\mu$ C, el cual debe dirigir los datos desde y hacia su destino (el MIS o las memorias). Esto último permite leer, modificar, suprimir, agregar y/o generar nuevos datos antes de ser remitidos a su destinatario. Estos caminos se pueden ver en la Figura 2.4.

Aunque el envío de datos se hace siempre sin intervención de una persona, el fabricante llama a estos caminos "MODO MANUAL", en caso de enviar los datos a través del  $\mu$ C 8051, y "MODO AUTOMÁTICO", cuando la comunicación es directa entre el MIS y las FIFO. Además, se programan en forma independiente para cada extremo, sea este de salida o entrada. Es decir, la entrada de un EP puede ser manual y la entrada de otro puede ser automática.

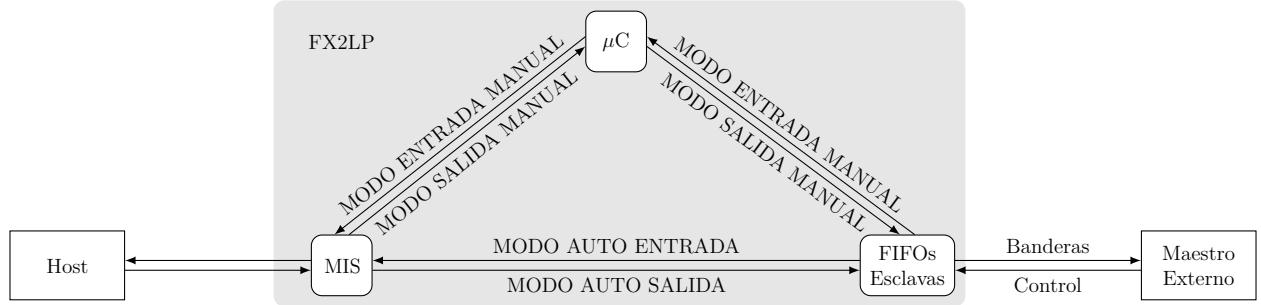


Figura 2.4: Modos de conexión de la memoria FIFO, el micrcontrolador y el MIS

Se debe notar en la Figura 2.4 que se refiere a paquetes de entrada cuando estos poseen una dirección que se inicia en un periférico y termina en el host y de salida cuando llevan el sentido contrario. Esto se debe al rol central que ejerce el host en la comunicación USB.

Para efectuar la configuración del modo de funcionamiento de cada EP, se recurre a los Registros de Configuración Extremo-FIFO esclava (EPxFIFOFCFG). A continuación se muestra la programación efectuada en este trabajo, en donde se envían los datos en forma automática, tanto de entrada como de salida. Se debe notar que la activación del modo automático se produce por el flanco ascendente de la variable de configuración, por lo que primero se coloca el registro en cero y luego se establece el valor de la configuración. También se indica en este registro que los datos tendrán un ancho de 16 bits.

```
//EPxFIFOFCFG – Registro de configuracion extremo/FIFO
// b6 -> '1' Indica lleno un byte antes
// b5 -> '1' Indica vacío un byte antes
// b4 -> '1' Modo Auto Salida
// b3 -> '1' Modo Auto Entrada
// b2 -> '1' Permite paquetes de entrada con largo 0
// b0 -> '1' bus de 16 bits , '0' bus de 8 bits
EP8FIFOFCFG = 0x00;
SYNCDELAY;
EP2FIFOFCFG = 0x00;
SYNCDELAY;

//establecer modo auto. se necesita flanco ascendente
EP8FIFOFCFG = 0x11;
SYNCDELAY;
EP2FIFOFCFG = 0x0D;
SYNCDELAY;
```

Una vez configuradas las interfaces, se deben restablecer las memorias FIFO, a fin de asegurarse que se encuentran vacías para iniciar la comunicación, a través del registro FIFORESET. El bit 7 de este registro le indica al MIS que la memoria FIFO no se encuentra disponible, y el MIS, a su vez, lo indica al Host si es necesario. Luego, a través de los cuatro bits menores se indica la dirección del EP a restablecer. Finalmente, libera la memoria y se le indica la situación al MIS.

```
//FIFORESET – Registro de restablecimiento FIFO
```

```
// b8      -> '1' Desabilitado
// b[3:0]  -> '1' Dirección de EP
FIFORESET = 0x80;
SYNCDELAY;
FIFORESET = 0x82;
SYNCDELAY;
FIFORESET = 0x84;
SYNCDELAY;
FIFORESET = 0x86;
SYNCDELAY;
FIFORESET = 0x88;
SYNCDELAY;
FIFORESET = 0x00;
SYNCDELAY;
// establecer modo auto. se necesita flanco ascendente
EP8FIFOFG = 0x11;
SYNCDELAY;
EP2FIFOFG = 0x0D;
SYNCDELAY;
```

En las líneas de código mostradas hasta acá se utiliza el macro *SYNCDELAY*. Dicho macro es una secuencia de espera requerida por Cypress para cumplir con los tiempos de mantenimiento asociados a la escritura y lectura de determinados registros[31], los cuales se explicitan en el Anexo ??.

### 2.2.5. Encabezado y declaraciones importantes

Para el correcto funcionamiento del código que se describe a lo largo de esta sección, es necesario incorporar el encabezado que se observa a continuación.

```
#pragma noiv                      // No generar vectores de interrupción
#include "fx2.h"
#include "fx2regs.h"
#include "syncdly.h"                // SYNCDELAY macro
#include "leds.h"
```

Las primeras 4 líneas de encabezados son provistas por Cypress, a través de su framework. En ellas, la directiva de ensamblador `noiv`(identificada con `#pragma`), le indica al compilador que no debe habilitar las interrupciones vectorizadas. Estas, en cambio, serán manejadas y direccionadas a través del archivo objeto *usbjmptb.obj*.

El encabezado *leds.h* cuyo código se muestra a continuación, sirve para encender y apagar las luces de la placa de desarrollo. Los LED fueron utilizado para realizar las tareas de prueba. Se explicará su funcionamiento más adelante.

Luego, el framework define algunas variables globales que utiliza en las funciones implementadas para el manejo de las tareas relacionadas al protocolo USB. Se listan estas variables a continuación.

```
extern BOOL GotSUD;                  // Received setup data flag
```

```

extern BOOL Sleep ;
extern BOOL Rwuen ;
extern BOOL Selfpwr ;

BYTE Configuration ; // Current configuration
BYTE AlternateSetting = 0; // Alternate settings

//-
// Task Dispatcher hooks
// The following hooks are called by the task dispatcher.
//-
WORD blinktime = 0;
BYTE inblink = 0x00;
BYTE outblink = 0x00;
WORD blinkmask = 0; // HS/FS blink rate

```

### 2.2.6. Descriptores USB

Los descriptores son una estructura definida de datos. A través de ellos, el dispositivo USB le comunica al anfitrión sus atributos, tales como velocidad de trabajo, cantidad de configuraciones e interfaces posibles, número de EPs, dirección de cada uno de ellos, tamaño máximo en bytes de paquetes que puede enviar en una comunicación, entre otros.

El framework de Cypress coloca toda la información sobre los descriptores del dispositivo desarrollado en un archivo escrito en lenguaje de ensamblador, denominado *dscr.a51*. Este archivo contiene una plantilla en donde el desarrollador coloca la descripción de la configuración realizada en el firmware. Luego, el archivo es enviado por el dispositivo al host comunicando las características del sistema desarrollado.

En primer lugar, posee un encabezado en donde se establecen etiquetas que hacen más legible el código para el programador:

```

1 DSCR_DEVICE equ 1 ; ; Descriptor type: Device
2 DSCR_CONFIG equ 2 ; ; Descriptor type: Configuration
3 DSCR_STRING equ 3 ; ; Descriptor type: String
4 DSCR_INTRFC equ 4 ; ; Descriptor type: Interface
5 DSCR_ENDPNT equ 5 ; ; Descriptor type: Endpoint
6 DSCR_DEVQUAL equ 6 ; ; Descriptor type: Device Qualifier
7
8 DSCR_DEVICE_LEN equ 18
9 DSCR_CONFIG_LEN equ 9
10 DSCR_INTRFC_LEN equ 9
11 DSCR_ENDPNT_LEN equ 7
12 DSCR_DEVQUAL_LEN equ 10
13
14 ET_CONTROL equ 0 ; ; Endpoint type: Control
15 ET_ISO equ 1 ; ; Endpoint type: Isochronous

```

```
16      ET_BULK      equ    2      ; ; Endpoint type: Bulk
17      ET_INT       equ    3      ; ; Endpoint type: Interrupt
```

Luego, el programador se debe asegurar que el código se guarda en un lugar de memoria adecuado.

```
1 DSCR  SEGMENT  CODE PAGE
2
3 ;;
4 ; ; Global Variables
5 ;;
6     rseg DSCR      ; ; locate the descriptor table in on-part memory.
```

Debido a la estructura rígida del formato de los descriptores, impuesto por la norma USB y por la implementación de Cypress, la memoria debe contener en primer lugar el descriptor DEVICE, el cual se muestra a continuación.

```
1 DeviceDscr :
2     db  DSCR_DEVICE_LEN      ; ; Largo del descriptor
3     db  DSCR_DEVICE      ; ; Tipo de descriptor
4     dw  0002H      ; ; Versión de la norma (BCD)
5     db  00H        ; ; Clase de Dispositivo
6     db  00H        ; ; Sub-Clase de Dispositivo
7     db  00H        ; ; Sub-sub-Clase de dispositivo
8     db  64         ; ; Tamaño máximo de paquete
9     dw  0B404H      ; ; Identificador de Vendedor (Cypress)
10    dw  0310H      ; ; Identificador de Producto (Sample Device)
11    dw  0000H      ; ; Identificador de versión del producto
12    db  1          ; ; Índice de Fabricante en string
13    db  2          ; ; Índice de Producto en string
14    db  0          ; ; Índice de número de serie en string
15    db  1          ; ; Número de configuraciones
```

El descriptor de tamaño máximo de paquete está referido especialmente al EP0, es decir, al extremo que el host y el dispositivo utilizan para intercambiar mensajes de control.

Se debe notar que los penúltimos tres parámetros mostrados corresponden a una descripción realizada en cadena de caracteres al final del archivo, a fin de poder mostrar un mensaje que pueda ser leído por un usuario humano.

El último parámetro está relacionado con el número de configuraciones que posee este dispositivo. Esto determina también la cantidad de descriptores de tipo CONFIGURATION que debe tener el archivo de descripción. No obstante, antes de comenzar con los descriptores CONFIGURATION, se debe especificar el descriptor DEVICE\_QUALIFIER, el cual da información sobre otras velocidades de operación. Este descriptor es necesario debido a que el sistema implementado cumple con la versión 2.0 de la norma USB.

```
1 org (( $ / 2 ) +1) * 2
2 DeviceQualDscr :
3     db  DSCR_DEVQUALLEN      ; ; Largo del descriptor
4     db  DSCR_DEVQUAL      ; ; Tipo de descriptor
```

```

5   dw  0002H      ; ; Versión de la norma (BCD)
6   db  00H       ; ; Clase de dispositivo
7   db  00H       ; ; Sub-clase de dispositivo
8   db  00H       ; ; Sub-sub-clase de dispositivo
9   db  64        ; ; Tamaño máximo de paquetes
10  db  1         ; ; Número de comunicaciones
11  db  0         ; ; Reservado

```

La norma USB especifica que la información que poseen los descriptores debe estar alineada por palabra (dos bytes), es decir, agrupada cada dos bytes y almacenada en direcciones de memoria par. Para asegurar esta condición, se recurre a la sentencia `org (($/2)+1)*2`, la que se puede leer en la primera líneas del código mostrado anteriormente. La directiva `org` le ordena al ensamblador la dirección específica de memoria en la que debe colocar las instrucciones precedentes. El símbolo (\$) representa al puntero que contiene el valor de memoria de la última instrucción ensamblada. Considerando que las direcciones de memoria son enteros y que las operaciones de ensamblador truncan decimales (no los redondean), la ecuación indicada entrega, como resultado, el entero par siguiente a la dirección actual.

Luego de esta directiva, se especifica el descriptor de tipo `DEVICE_QUALIFIER` asegurando que se encontrará en una dirección par de memoria.

Seguidamente, el protocolo indica que se debe detallar cada una de las configuraciones y las interfaces que se indican en los descriptores `DEVICE` y `DEVICE_QUALIFIER`. En este caso, se especifican dos configuraciones: una de alta velocidad, indicada en el descriptor `DEVICE` y otra de velocidad completa (*Full-Speed*), que es especifica en el descriptor `DEVICE_QUALIFIER`. Se muestran el descriptor `CONFIGURATION` de alta velocidad, unido al descriptor `INTERFACE` (línea 16 en adelante) y los descriptores `ENDPOINT` a partir de la línea 27, que determinan en forma completa la configuración de Alta Velocidad.

```

1 org (( $ / 2 ) +1 ) * 2
2 HighSpeedConfigDscr :
3   db  DSCR_CONFIG_LEN      ; ; Largo del descriptor
4   db  DSCR_CONFIG          ; ; Tipo de descriptor
5   db  (HighSpeedConfigDscr_End-HighSpeedConfigDscr) mod 256
6           ; ; Largo total (LSB)
7   db  (HighSpeedConfigDscr_End-HighSpeedConfigDscr) / 256
8           ; ; Largo total (MSB)
9   db  1        ; ; Número de interfaces
10  db  1        ; ; Índice de configuración
11  db  0        ; ; String de configuración
12  db  80H      ; ; Atributos (b7->1, b6 - selfpwr , b5 - rwu)
13  db  50       ; ; Consumo de potencia (div 2 ma)
14
15  ; ; Alt Interface 0 Descriptor – Bulk IN
16  db  DSCR_INTRFC_LEN     ; ; Largo del descriptor
17  db  DSCR_INTRFC         ; ; Tipo de descriptor
18  db  0        ; ; Índice de interfaz
19  db  0        ; ; Índice de ajuste alternativo
20  db  2        ; ; Número de extremos

```

```

21      db  0ffH           ; ; Clase de interfaz
22      db  00H           ; ; Sub-clase de interfaz
23      db  00H           ; ; Sub-sub-clase de interfaz
24      db  0             ; ; Indice de string descriptor de interfaz
25
26      ; ; Iso IN Endpoint Descriptor
27      db  DSCR_ENDPNT_LEN    ; ; Largo del descriptor
28      db  DSCR_ENDPNT        ; ; Tipo de descriptor
29      db  82H              ; ; Extremo de entrada EP2
30                  ; ; b7 -> IN/OUT, b[4:0] -> dir
31      db  ET_ISO           ; ; Tipo de transferencia
32      db  00H              ; ; Tamaño máximo de paquete (LSB)
33      db  02H              ; ; Tamaño máximo de paquete (MSB)
34      db  01H              ; ; Intervalo de consulta
35
36      ; ; Bulk OUT Endpoint Descriptor
37      db  DSCR_ENDPNT_LEN    ; ; Largo del descriptor
38      db  DSCR_ENDPNT        ; ; Tipo de descriptor
39      db  08H              ; ; Extremo de salida EP8
40      db  ET_BULK           ; ; Tipo de transferencia
41      db  00H              ; ; Tamaño máximo de paquete (LSB)
42      db  02H              ; ; Tamaño máximo de paquete (MSB)
43      db  00H              ; ; Intervalo de consulta
44
45 HighSpeedConfigDscr_End :

```

En la línea 12 del código anterior se debe colocar cuál es la fuente de la potencia que consume el dispositivo, es decir, de donde proviene la energía utilizada para el funcionamiento. El bit 7 debe estar siempre establecido a 1 por razones históricas de la norma USB[27]. El bit 6 en 1 define que el dispositivo está energizado por una fuente propia. En el caso contrario, toma potencia del bus. El bit 5, por su parte, señala que el dispositivo tiene modo de baja energía y que es posible establecer el modo de funcionamiento de mayor consumo con un comando del host. La línea 13 se informa cuánta potencia consume, lo que le brinda al host la posibilidad de establecer un control de la potencia suministrada en el bus.

El último campo del descriptor ENDPOINT, correspondiente al intervalo de consulta, se utiliza para establecer cada cuanto tiempo el host debe asignar ancho de banda para transferencias isocrónicas.

Luego de enviar la configuración de Alta Velocidad, se informa de la misma manera los descriptores que detallan la configuración del dispositivo, cuando trabaja con Velocidad Completa, cuyo código se observa a continuación.

```

1 org (( $ / 2 ) +1) * 2
2 FullSpeedConfigDscr :
3     db  DSCR_CONFIG_LEN    ; ; Largo del descriptor
4     db  DSCR_CONFIG        ; ; Tipo de descriptor
5     db  (FullSpeedConfigDscr_End - FullSpeedConfigDscr) mod 256
6                  ; ; Largo total (LSB)

```

```

7   db  (FullSpeedConfigDscr_End-FullSpeedConfigDscr) / 256
8   ; ; Largo total (MSB)
9   db  1      ; ; Número de interface
10  db  1      ; ; Numero de configuraciones
11  db  0      ; ; Indice de string de configuración
12  db  80H    ; ; Atributos (b7 -<'1', b6 - selfpwr, b5 - rwu)
13  db  50     ; ; Requerimiento de potencia (div 2 ma)
14
15  ; ; Interface Descriptor
16  db  DSCR_INTRFC_LEN ; ; Largo del descriptor
17  db  DSCR_INTRFC    ; ; tipo de descriptor
18  db  0              ; ; Indice de interfaz
19  db  0              ; ; Indice de ajuste alternativo
20  db  2              ; ; Número de extremos
21  db  0ffH            ; ; Clase de interfaz
22  db  00H            ; ; Sub-clase de interfaz
23  db  00H            ; ; Sub-sub-clase de interfaz
24  db  0              ; ; Indice de string de interfaz
25
26  ; ; Endpoint Descriptor
27  db  DSCR_ENDPNT_LEN ; ; Largo del descriptor
28  db  DSCR_ENDPNT    ; ; Tipo de descriptor
29  db  82H            ; ; Dirección y sentido de extremo
30  db  ET_ISO          ; ; Tipo de extremo
31  db  0FFH            ; ; Tamaño máximo de paquete (LSB)
32  db  03H            ; ; Tamaño máximo de paquete (MSB)
33  db  01H            ; ; Intervalo de consulta
34
35  ; ; Endpoint Descriptor
36  db  DSCR_ENDPNT_LEN ; ; Largo del descriptor
37  db  DSCR_ENDPNT    ; ; tipo de descriptor
38  db  08H            ; ; Dirección y sentido de extremo
39  db  ET_BULK         ; ; Tipo de extremo
40  db  040H            ; ; Tamaño máximo de paquete (LSB)
41  db  00H            ; ; Tamaño máximo de paquete (MSB)
42  db  01H            ; ; Intervalo de consulta
43
44 FullSpeedConfigDscr_End :

```

Finalmente, el diseñador puede escribir todos los mensajes en formato de cadena de caracteres, para una lectura más sencilla por parte del usuario. En este trabajo solo se usan dos que sirven como ejemplo pero no se profundizó más en el estudio de estos mensajes debido a que no son relevantes para los objetivos.

```

1 org (($ / 2) +1) * 2
2 StringDscr:
3

```

```
4 StringDscr0:  
5     db    StringDscr0_End-StringDscr0      ; ; Largo del descriptor  
6     db    DSCR_STRING                      ; ; Tipo de descriptor  
7     db    09H,04H  
8 StringDscr0_End:  
9  
10 StringDscr1:  
11    db    StringDscr1_End-StringDscr1      ; ; Largo del descriptor  
12    db    DSCR_STRING                      ; ; Tipo de descriptor  
13    db    'E',00                            ; ; Mensaje  
14    db    'd',00  
15    db    'w',00  
16    db    'i',00  
17    db    'n',00  
18    db    ' ',00  
19    db    'B',00  
20    db    'a',00  
21    db    'r',00  
22    db    'r',00  
23    db    'a',00  
24    db    'g',00  
25    db    'a',00  
26    db    'n',00  
27 StringDscr1_End:  
28  
29 StringDscr2:  
30    db    StringDscr2_End-StringDscr2      ; ; Largo del descriptor  
31    db    DSCR_STRING                      ; ; Tipo de descriptor  
32    db    'L',00                            ; ; Mensaje  
33    db    'a',00  
34    db    ' ',00  
35    db    'T',00  
36    db    'e',00  
37    db    's',00  
38    db    'i',00  
39    db    's',00  
40 StringDscr2_End:
```

## 2.3. Depuración y verificación de funcionamiento

Se realizaron diversas pruebas y versiones tanto para resolver problemas como para verificar el correcto funcionamiento de la interfaz.

El primer problema que se presentó fue una inestabilidad en la ejecución del código, la cual se mostraba en forma intermitente al cargar el código compilado. La inestabilidad hacía que el código repentinamente se detuviera en la ejecución de la inicialización del dispositivo. A fin de

salvar dicho problema, se recurrió al envío de mensajes de seguimiento a través de los puertos UART que prevee el controlador FX2LP.

También se realizaron pruebas de robustez en la comunicación, pudiendo enviar y recibir datos por el mismo puerto UART, aprovechando la configuración establecida. Para la recepción de datos en la PC se utilizó el programa Hercules[32]. Dicho programa es una desarrollo cuya descarga es libre y permite configurar y recibir mensajes a través de diferentes puertos.

Como testigo del funcionamiento de los flags, se asociaron sus valores a los LED de propósito general, de forma tal que se pueda corroborar que los endpoint usados recibían y enviaban los datos cuando se emitía la orden desde el Host.

### 2.3.1. Biblioteca FX2LPSerial

Para la configuración y utilización del puerto UART 0 se recurrió a la biblioteca FX2LPSerial[33]. Esta biblioteca es un conjunto de funciones de C para microcontroladores que resuelven la configuración los puertos UART y de las rutinas asociada a la recepción y envío de datos por dichos puertos.

La configuración del puerto UART se realizó a través de la función `FX2LPSerial_Init()`. Esta función configura los registros de los contadores para establecer una tasa de transmisión de 38400 baudios. Luego, asegura que el reloj que utiliza el puerto UART se encuentre configurado. Esto se realiza a través del Registro de Configuración de la Interfaz (IFCONFIG), el cual se setea para correr a 48 MHz, la misma a la cual funciona el sistema desarrollado, por lo que no presenta problemas de compatibilidad. Si bien es posible cambiar la velocidad de transmisión, esta configuración posee una desviación del 0,16 %[31] entre la tasa nominal y la tasa real de transferencia. Esta diferencia de tasas es suficiente para asegurar que funcione en cualquier dispositivo. Por lo tanto, se decidió utilizar la configuración con los valores por defecto.

Para la recepción de datos en la PC se utilizó el programa Hercules[32]. Dicho programa es de descarga libre y permite configurar y recibir mensajes a través de protocolo Ethernet o por puerto Serie. Se configuró el puerto UART, en la pestaña destinada al monitoreo del puerto Serie, con una tasa de transmisión de 38400 baudios, 8 bit, sin paridad y sin handshaking. A través de esta configuración recibe cualquier dato enviado a través de los puertos UART del controlador FX2PL y los muestra en formato ASCII.

Una vez configurado el envío de datos a través del puerto UART, se procedió a enviar mensajes de control para poder corroborar la funcionalidad y comprobar si efectivamente el controlador efectuaba sus tareas y en qué momento dejaba de hacerlo.

Se pudo determinar que en la función `main()`, luego de realizar la inicialización de todos los modulos, procede a la desconexión del controlador con el puerto USB, a través de una rutina que Cypress denomina ReNumertion[31]. Durante la reconexión ocurre algún efecto, aún no identificado, que hace que el programa se detenga cuando debe reconectarse. Este desperfecto fue solucionado con el agregado de una línea de comunicación después de el activado del reconectado. Esto demostró ser lo suficientemente robusto ya que eliminando todas las líneas que envían datos de monitoreo no aparcó nuevamente la detención del programa, más sí, eliminando solo esa línea de código. De esta manera, el código queda de la siguiente forma en donde la línea 190 es la agregada en este trabajo:

```
189 USBCS &= ~bmDISCON;  
190 FX2LPSerial_XmitString( "Reconectando . . . \n\n");
```

### 2.3.2. Testigos LED

Los LED multipropósito incorporados en la placa de desarrollo CY3684, fueron programados para que repliquen el estado de los flags de vaciado y de llenado de los EP. De esta forma, se pudo monitorear la carga de datos en el controlador y la descarga a través del puerto USB y vicersa.

Los LED se encuentran conectados a través de un decodificador. Para su encendido, es necesario la lectura de las direcciones hexadecimales de memoria 80xx, 90xx, A0xx y B0xx, mientras que para su apagado, las direcciones a leer son 88xx, 98xx, A8xx y B8xx[? ]. Por ello, se elaboró el encabezado *leds.h*, el que se observa a continuación.

```
xdata volatile const BYTE D2ON _at_ 0x8800;
xdata volatile const BYTE D2OFF _at_ 0x8000;
xdata volatile const BYTE D3ON _at_ 0x9800;
xdata volatile const BYTE D3OFF _at_ 0x9000;
xdata volatile const BYTE D4ON _at_ 0xA800;
xdata volatile const BYTE D4OFF _at_ 0xA000;
xdata volatile const BYTE D5ON _at_ 0xB800;
xdata volatile const BYTE D5OFF _at_ 0xB000;
```

Luego, con el propósito de encender o apagar los diodos emisores de luz, es necesario declarar una variable auxiliar y asignarle los punteros declarados. Así, a través de la función *TD\_Poll()*, que es la función que se ejecuta en el loop infinito del controlador FX2LP, se colocó la siguiente código:

```
void TD_Poll(void) // Called repeatedly while the device is idle
{
    BYTE dum;

    if (EP8FIFOFLGS & bmBIT1) //ep8 fifo empty
    {
        dum = D4ON;
    }
    else
    {
        dum = D4OFF;
    }

    if (EP8FIFOFLGS & bmBIT0) //ep8 fifo full
    {
        dum = D3ON;
    }
    else
    {
        dum = D3OFF;
    }
    if (EP2FIFOFLGS & bmBIT1) //ep2 fifo empty
    {
```

```
        dum = D2ON;
    }
else
{
    dum = D2OFF;
}
}
```

A través de los registros EPxFIFOFLGS se puede conocer el estado de cada uno. Aplicando una máscara bit a bit, se obtiene el estado de cada uno de los flags. Así, dependiendo del estado de cada uno de ellos, se enciende o se apagan las luces. Esta rutina resultó de mucha utilidad a la hora de realizar las diferentes pruebas, tanto del funcionamiento de la interfaz, como de la conexión entre la interfaz y el FPGA.

### 2.3.3. Prueba de envío y recepción de datos

Para la verificación de la conexión entre la PC y la interfaz, a través del protocolo USB, se utilizó el programa *Cypress USB Control Center*, provisto por Cypress dentro del kit de desarrollo CY3684, con el cual se desarrolló este trabajo. Dicho software permite cargar el firmware desarrollado en la interfaz y, a su vez, detalla la información recibida a través de los descriptores y posibilita el envío y recepción de mensajes.

Para corroborar que el envío de datos fue exitoso, se procedió a enviar datos a través del *Cypress USB Control Center*. Así, en primer lugar, corroborando que el LED que indica que el EP8 se encuentra vacío se apaga, se infirió que los datos estaban llegando. A continuación, se enviaron más datos hasta lograr que el LED asociado al límite de capacidad del EP8 se encendiese. Luego, a través de un pulsador, se activa una rutina de envío de los datos guardados en el EP8 a través del puerto UART. Para ello, fue necesario realizar unos pequeños ajustes en la configuración.

El controlador FX2LP de Cypress permite manipular los datos de los paquetes USB. Sin embargo, la configuración por defecto, no permite realizar esta tarea. Para ello es necesario, en primer lugar, desactivar lo que Cypress llama el armado automático de paquetes y habilitar la manipulación avanzada de paquetes[31].

A medida que los datos van ingresando al controlador FX2LP a través del puerto USB, el Motor de Interfaz Serial (MIS) los coloca en el espacio de memoria asignado y va incrementando un contador por cada byte recibido. De esta forma, la interfaz puede saber cuantos datos, efectivamente, posee almacenados. Sin embargo, cuando los datos ingresan a través de la memoria FIFO, es otro el contador incrementado. El armado automático de paquetes realiza la tarea de emparejar estos contadores, de forma tal que no se deba destinar tiempo de ejecución de  $\mu$ C para esta tarea.

Para poder escribir datos en la memoria y informarle al MIS que se agregaron datos, es necesario que este armado automático está desactivado. Esto implica que cuando lleguen datos, el controlador debe poder realizar esta operación. A su vez, por defecto los paquetes que llegan desde la PC no pueden ser modificados. Para realizar esto, es necesario habilitar el manejo mejorado de paquetes. Tanto la manipulación avanzada de paquetes como el armado automático se encuentran los dos bits menos significativos del Registro de Control de Revisión (REVCTL).

Una vez activados ambos bits del registro REVCTL, se debe efectuar la rutina que armará

los paquetes en forma “manual”. Para este propósito, se activó una interrupción que se dispara cada vez que llegan mensajes a un EP determinado, en este caso, el EP8. Toda esta configuración se realizó en las últimas líneas de la función `TV_Init()`, la cual quedó de la siguiente forma:

## **2.4. Sumario del capítulo**

En el presente capítulo se desarrolló y justificó la elección del controlador FX2LP como nexo entre la FPGA y la PC, brindando la conexión USB necesaria. Luego, se explicaron algunos componentes de la arquitectura implementada por Cypress a fin de proveer la comunicación USB. Finalmente, se detalló paso a paso cada uno de los componentes configurados, como así también el código desarrollado para dicho fin.

Además, se mostraron algunos detalles del framework provisto por Cypress y los encabezados necesarios para su utilización y se explicitaron los descriptores a través de los cuales se le informa al sistema las características de la comunicación que se implementa.

## Capítulo 3

# Desarrollo del sistema maestro para la comunicación entre la FPGA y la interfaz Cypress

En el Capítulo anterior, se ha descripto una comunicación que intercambia datos entre una PC y el controlador FX2LP a través del protocolo USB. Dicha comunicación constituye una primer etapa del desarrollo realizado.

En una segunda etapa, se debe lograr la comunicación de datos entre un FPGA y la interfaz. Con este enlace, los datos estarían en condiciones de fluir desde el Host y el FPGA, a través de la interfaz. Los datos son transferidos a través de las memorias FIFO que se detallaron en el Capítulo 2.

Es necesario identificar cuales son los mecanismos para la lectura y escritura de datos, las señales que intervienen y los puertos con los que debe interactuar el FPGA e implementar dentro de este dispositivo un módulo que sea capaz de interactuar con las memorias FIFO del controlador FX2LP.

El módulo que se implementó en el FPGA es una pequeña Máquina de Estados Finitos, a través de la cual, se leen las señales que provienen de la interfaz y se generan las señales necesarias para comandar su memoria FIFO.

A continuación, se justifica la elección del FPGA y la placa de desarrollo utilizados. También se detallan las señales que intervienen en el funcionamiento de la interfaz y los protocolos de lectura y escritura de modo asíncrono.

Los mecanismos que se deben seguir para las operaciones de intercambio de datos dan lugar a la elaboración de la máquina de estados se sintetiza en el FPGA. Se utiliza VHDL como lenguaje de descripción para elaborar el código que implementa de la máquina de estados en el FPGA se utiliza el lenguaje de descripción de hardware VHDL.

Además, se explica el desarrollo de un circuito impreso utilizado para la interconexión entre las distintas placas de desarrollo que se utilizan en este trabajo.

### 3.1. Elección de la FPGA

En la implementación de una comunicación, para poder transmitir y recibir datos, los componentes que intervienen deben seguir un protocolo establecido. Así, no solo se facilita el

envío y la recepción del mensaje, sino también que determina a cada dispositivo los procedimientos que efectúa. Por este motivo, una vez definido que se utiliza una interfaz intermedia entre la PC y un FPGA (Capítulo 1), y que dicha interfaz es el circuito integrado EZ-USB FX2LP de Cypress (Capítulo 2), se determina cuál es el protocolo a través del cual se comunica cada uno de los dispositivos y se puede configurar un FPGA para que reciba y envíe datos a la interfaz.

En base a los materiales disponibles, se evaluaron tres placas de desarrollo diferentes, todas con FPGA diseñadas y comercializadas por la empresa Xilinx Inc. La placa Spartan-3E Starter, comercializada por Digilent es una placa muy potente debido a la gran cantidad de periféricos que incorpora, entre los que se destacan su pantalla LCD, los 18 MB de memoria flash sumados a 64 MB de SDRAM y la gran cantidad de transceptores tales como Ethernet, PS/2 para teclados y JTAG para programación y depuración.

Por su parte, la Nexys 3, tiene como núcleo un FPGA Spartan-6, es decir, un FPGA tecnológicamente superior a los FPGA Spartan-3,. El FPGA Spartan 6 brinda mayor cantidad de bloques lógicos programables, debido a que posee un método de fabricación más avanzado que permite elaborar transistores más pequeños en su circuito integrado. La miniaturización de los transistores, a su vez, otorga la posibilidad de una mayor velocidad de operación. La placa Nexys 3 también posee una gran gama de periféricos tales como pulsadores, interruptores, displays led de 7 segmentos, diferentes tipos de memorias, CODEC para comunicarse por Ethernet 10/100 o USB 2.0.

A diferencia de las anteriores, la placa de desarrollo Mojo v3 comercializada por la empresa Alchitry, es una placa de prototipado rápido. Esto quiere decir que en lugar de poseer una gran cantidad y variedad de periféricos, se dota a la placa de una gran cantidad de puertos para que el usuario pueda colocar los periféricos que desea. Al igual que la Nexys 3, cuenta con un FPGA Spartan 6 de Xilinx. Dispone de 84 puertos digitales configurables como entrada y/o salida, 8 entradas analógicas, 8 LED's de propósito general, un botón de tipo pulsador. La principal ventaja de esta placa de desarrollo es que posee un costo notablemente inferior a las anteriores debido en gran medida a la ausencia de periféricos que, dependiendo la aplicación, pueden ser innecesarios.

Se elige para el desarrollo que se presenta en este trabajo la placa de desarrollo Mojo v3 debido a que se incorporan varios periféricos al kit CY3684 utilizado para el desarrollo de la interfaz. Además de esto, posee un FPGA superior a la placa Spartan 3E Starter, por lo que brinda la posibilidad de elaborar sistemas más complejos y veloces. En otras palabras la Mojo se selecciona por su bajo costo, versatilidad y por que está dotada por un Spartan-VI de Xilinx, que es un FPGA con una buena relación entre recursos, rendimiento, velocidad y precio.

La Mojo v3, la cual se observa en la Figura 3.1, es una placa de desarrollo muy económica para prototipado, es decir, la fabricación de modelos funcionales. Para ello los puertos se disponen en un arreglo de pines a través de los cuales es posible acoplar el dispositivo que sea necesario. Se dispone en el mercado de otros circuitos impresos que se conectan a los pines y contienen un grupo de periféricos para propósito general. Estos circuitos impresos, se denominan shields (*escudo* traducido al castellano). Se obtiene así una placa de desarrollo a la medida de las necesidades de cada proyecto. El usuario también puede diseñar sus shields o conectar las entradas y salidas de otros dispositivo mediante cables.

Además de los shields, los diseñadores pensaron en que no sea necesario ninguna herramienta extra a la hora de programar la FPGA. Para ello, dotaron al sistema de un microcontrolador ATmega32U4 de Atmel con un programa de tipo bootloader, que se encarga de transferir la

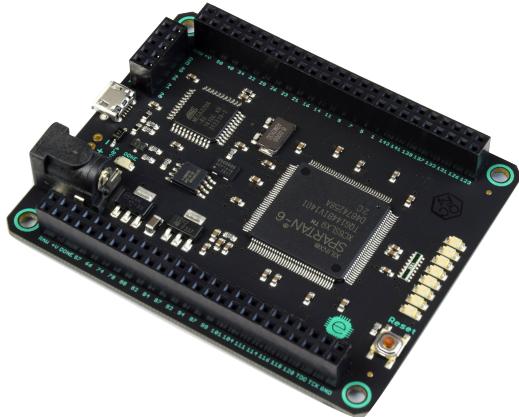


Figura 3.1: Placa de prototipado rápido MOJO v3, diseñada por Embedded Micro

configuración del FPGA (cargada desde una memoria flash incorporada, o transmitida por el usuario desde una PC), a través de un transceptor USB que contiene el microcontrolador. Luego, el controlador es colocado en modo esclavo y se configura de forma tal que dota al sistema de una comunicación entre la FPGA y una PC, vía USB. Las entradas analógicas que posee esta placa de desarrollo también son leídas a través del microcontrolador ATmega32U4, luego de que el FPGA es programado.

Es importante aclarar que si bien el sistema posee alguna forma de comunicación USB utilizando el microcontrolador ATmega32U4 como interfaz, el enlace que forma posee un menor ancho de banda que el sistema en desarrollo por el presente trabajo. Esto se debe a que la línea de controladores ATmega incorpora puertos USB 2.0 full-speed. Esto quiere decir que puede enviar datos a una tasa de  $12 \text{ Mbit s}^{-1}$ . Además, la comunicación entre ambos chips se realiza vía SPI (*Serial Peripheral Interface*, o en español Interfaz Serie de Periféricos), comandada por un cristal de cuarzo de 50 MHz, ofreciendo una velocidad de salida que puede resultar insuficiente a los fines de este trabajo. Se pretende dotar al sistema del mayor ancho de banda posible, utilizando la capacidad de USB 2.0 High-Speed, de hasta 480 Mbps.

## 3.2. Señales de comunicación de la Máquina de Estados Finitos

La comunicación entre un FPGA y el controlador FX2LP requiere de la elaboración de una interfaz que sea capa de responder en forma adecuada al protocolo establecido para la lectura y escritura de datos en la memoria FIFO del controlador FX2LP.

Un protocolo es una secuencia estructurada de pasos a seguir para efectuar un propósito. El sistema electrónico que sigue una secuencia estructurada de pasos es una Máquina de Estados Finitos (MEF). Por esto, a continuación se detallará la metodología a través de la cual se efectúan las operaciones de lectura y escritura en la memoria FIFO, y la comunicación interna de los diferentes módulos del FPGA, a fin de identificar las señales de entrada y de salida que dan lugar al programa que sigue la MEF diseñada en su funcionamiento.

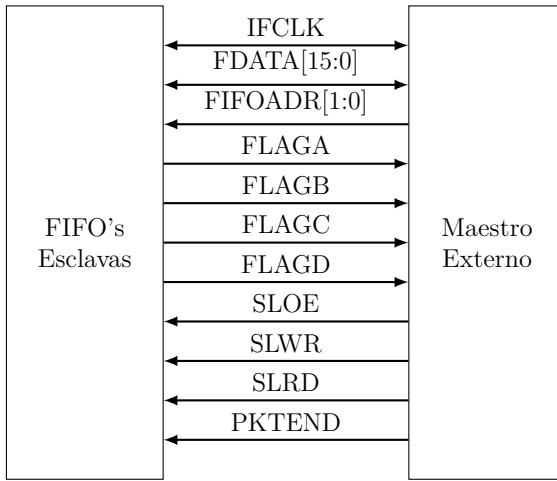


Figura 3.2: Señales de la interfaz entre las FIFO's y un maestro externo

### 3.2.1. Señales de comunicación el FPGA y el controlador FX2LP

La Figura 3.2 muestra los puertos a través de los cuales se conectan las memorias FIFO del controlador FX2LP con un dispositivo de control externo, el cual se implementa en este desarrollo a través del FPGA Spartan-IV de la placa Mojo. Las señales de control son:

- IFCLK: señal de reloj. No es necesario en caso de conectar la interfaz en modo asincrónico. La señal de reloj puede ser provista por el controlador o por el dispositivo de control en forma programable.
- FDATA[15:0]: constituye el bus de datos. Según se programe, este puede ser de 8 o 16 bits, en forma independiente para cada EP.
- FIFOADDR[1:0]: puerto de direcciones. A través de él se selecciona la memoria activa en el bus.
- FLAGx: Los cuatro puertos de flag indican el estado de las memorias y son configurables.
- SLOE, SLWR, SLRD: son las señales de control. A través de ellas el maestro entrega las ordenes de lectura y escritura.
- PKTEND: a través de este puerto el maestro indica que terminó una transferencia de datos.

Las señales FIFOADR[1:0] se utilizan para seleccionar la memoria FIFO sobre la que se escriben o leen los datos. Cada una de estas memorias está asociada a un extremo (EP) determinado. Estos extremos poseen dirección hexadecimal 02, 04, 06 y 08 para el sistema USB comandado por el  $\mu$ C 8051 incorporado en el circuito integrado FX2LP. Las memorias FIFO tienen dirección binaria "00", "01", "10" y "11" en los puertos FIFOADR[1:0]. Se muestra en la Tabla 3.1 las direcciones asociadas entre cada una de las memorias FIFO y los EP. Se destaca que '0' y '1' en cada puerto FIFOADR equivalen a niveles de tensión bajo y alto, respectivamente.

Los puertos que indican el estado de las memorias son programables. Pueden indicar si la memoria se encuentra llena o vacía. También es posible que señalen que se alcanzó una

FIFOADR[1:0]	EP (USB)
00	0x02
01	0x04
10	0x06
11	0x08

Tabla 3.1: Direcciones de selección de memoria activa

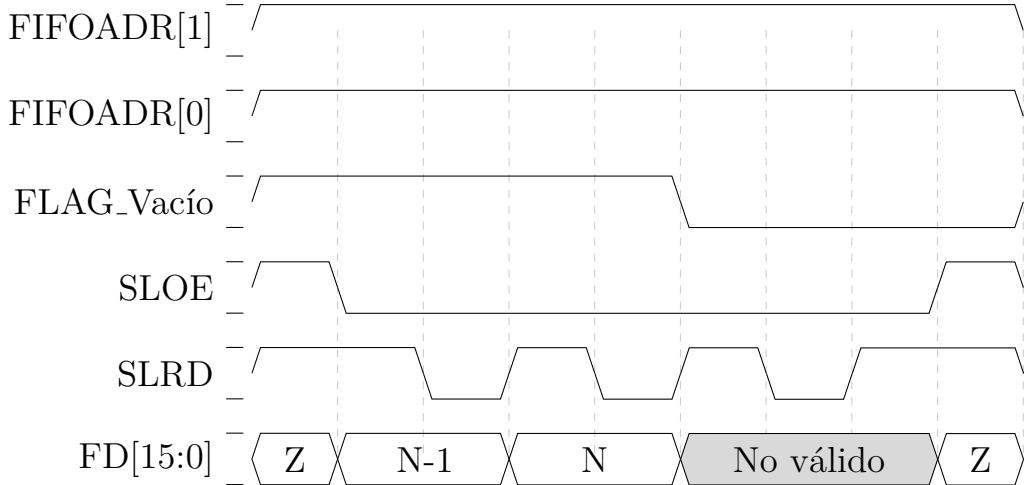


Figura 3.3: Diagrama temporal de la lectura de datos desde la memoria FIFO por un FPGA

cantidad de datos superior a un umbral programable. Según la configuración que el usuario realice, estarán asociadas a una memoria específica o a la memoria activa, seleccionada a través de FIFOADDR[1:0].

La configuración que se implementó en este trabajo, como ya se menciona en el Capítulo 2, dispone al EP 0x02 como puerto de entrada USB (es decir, salida desde el FPGA) y al EP 0x08 como salida USB (o sea, entrada para el FPGA). A su vez, el puerto FLAGA señala que la memoria FIFO relacionada al EP 0x02 está llena y el FLAGB indica que la memoria FIFO relacionada al EP 0x08 está vacía.

Se debe destacar que todas las señales que emite la memoria FIFO del controlador FX2LP son activas en bajo. Esto quiere decir que, por ejemplo, si la señal FLAGA posee un nivel de tensión bajo (cerca de 0 V), el espacio de memoria destinado al EP2 se encuentra lleno. Por su parte, si el valor de tensión es más cercano a la tensión de alimentación del controlador (3.3 V), la memoria aún posee espacio para el almacenamiento de datos.

### Lectura de datos desde la memoria FIFO

Para efectuar operaciones de lectura en régimen asíncrono, como se muestra en la Figura 3.3, en primer lugar, el FPGA debe colocar en los puertos FIFOADR[1:0] la dirección de la memoria sobre la que desea efectuar esta operación. En el caso de la configuración planteada en este trabajo, "11", la que corresponde al EP8. Luego, debe ser activada la señal SLOE, la cual coloca en los puertos FD[15:0] los datos almacenados en la memoria FIFO activa por FIFOADR[1:0]. El dato disponible en la salida de la memoria FIFO siempre será el más antiguo, es decir, el que se almacenó antes. En el flanco negativo de la señal SLRD, la memoria FIFO aumenta un

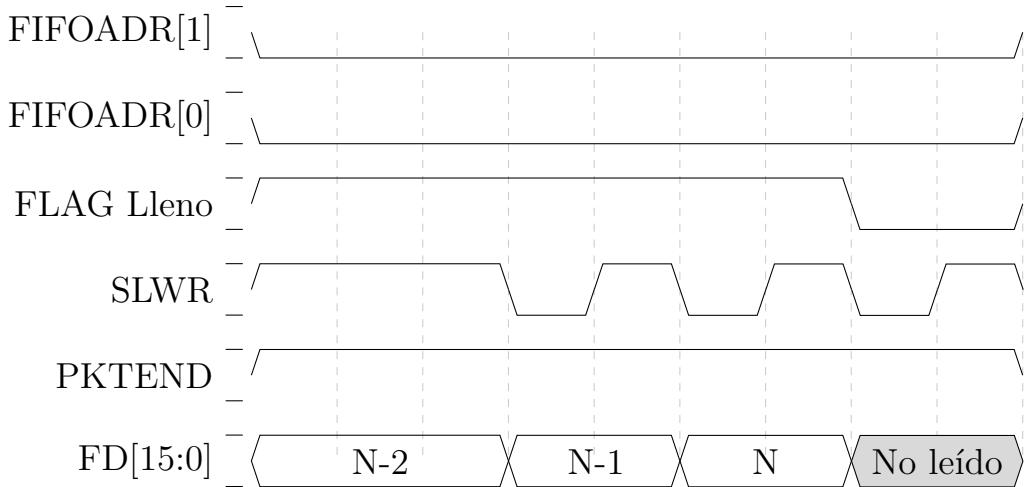


Figura 3.4: Diagrama temporal de la escritura de datos en la memoria FIFO desde un FPGA

contador que selecciona la dirección del próximo dato, y coloca este dato en el puerto FD[15:0].

Una vez que todos los datos fueron leídos, es decir, que el contador de la memoria ha alcanzado un valor N de datos, iguales a los almacenados, se activa la señal FLAG\_Vacío (para este trabajo, FLAGB). Mientras SLOE no está activo, el puerto FD[15:0] permanece en estado de alta impedancia para dejar el bus disponible a otros dispositivos.

### Escritura de datos en la memoria FIFO

Las señales que intervienen en el proceso de escritura de datos en la memoria FIFO, se encuentran detalladas en el diagrama temporal de la Figura 3.4. Para escribir datos en una memoria FIFO, el FPGA debe seleccionar la memoria a través de FIFOADR[1:0] en primer lugar. Para la configuración de este trabajo, esto es "00", correspondiente al EP2. Luego, se coloca en el bus de datos, donde se encuentran conectados los puertos FD[15:0], el dato a escribir. Es importante aclarar que SLOE no debe estar activo, de modo tal que el bus FD[15:0] se encuentre en modo de alta impedancia por parte del controlador FX2LP y no interfiera con la escritura.

Una vez colocado el dato en el bus, se debe activar la señal SLWR. En el flanco negativo de SLWR, el controlador incrementa el contador que indica la dirección de memoria en donde será almacenado el dato siguiente y deja guardado el dato que leyó en los puertos del bus FD.

La interfaz FX2LP espera siempre un número determinado de datos, señalizado como N en los diagramas de la Figura 3.4 y la Figura 3.5. Una vez alcanzado dicho número, el paquete queda listo para ser enviado cuando el host lo solicite. Sin embargo, puede ser enviado un número menor de datos en forma manual. Este funcionamiento es provisto a través de la señal PKTEND. Como se observa en la Figura 3.5, cuando se activa PKTEND, también lo hace la señal FLAG\_Lleno y la memoria FIFO ignora cualquier dato que se envíe a continuación.

### 3.2.2. Comunicación interna del FPGA

La MEF desarrollada es un nexo entre el controlador FX2LP y el FPGA. Por ello, la MEF debe ser capaz de proveerle a este último dispositivo los datos leídos en el primero y, en sentido

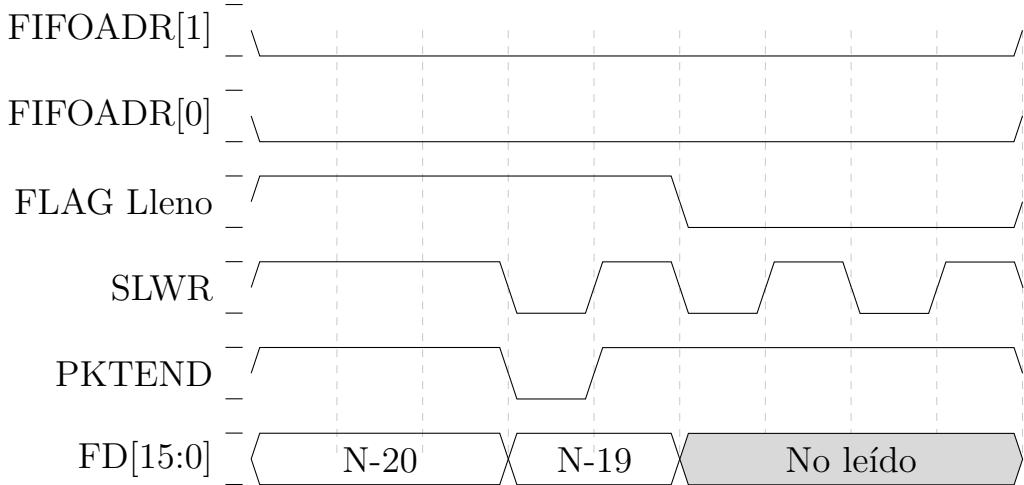


Figura 3.5: Diagrama temporal del funcionamiento del finalizado manual de mensajes

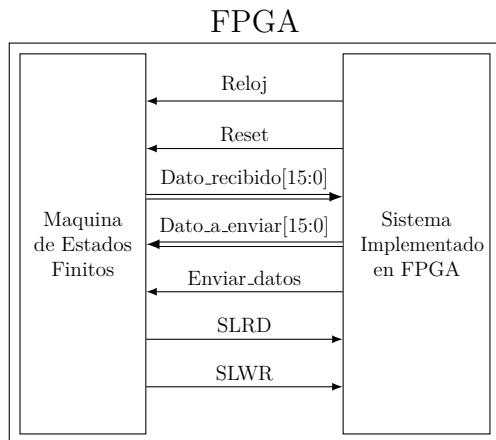


Figura 3.6: Diagrama de las señales que interconectan los módulos dentro del FPGA

inverso, poder escribir en la interfaz FX2LP los datos que le suministre un sistema sintetizado en el FPGA cuando lo indique. En adición, el FPGA debe proveerle al sistema una señal de reloj a fin de que la MEF tenga un funcionamiento sincronizado con el sistema.

La Figura 3.6 muestra un diagrama en bloques en donde se detalla cuáles son las señales internas a través de las cuales se comunican los distintos módulos que se implementan en un FPGA. La MEF desarrollada posee entrada y salida de datos independientes *Dato\_a\_enviar* y *Dato\_Recibido*, entrada de reloj y reset que será suministrada por el FPGA, como así también de una señal que controla el envío de datos, *Enviar\_Datos*.

Para indicar al FPGA que los datos a enviar fueron enviados, la MEF posee como salida *SLWR*. Para indicar que leyó datos de la interfaz FX2LP, se utiliza la señal *SLRD*. Se debe notar que ambas señales son las mismas a través de las cuales la MEF se comunica con la memoria FIFO del controlador.

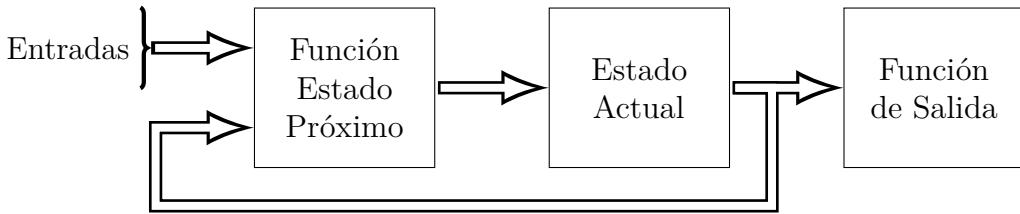


Figura 3.7: Estructura de una máquina de estados finitos

### 3.3. Diseño de la Máquina de Estados Finitos

Una Máquina de Estados Finitos (MEF) es un tipo de dispositivo lógico secuencial que puede estar en uno de un conjunto finito de elementos. Tal como se observa en la Figura 3.7, el estado siguiente que adoptará la MEF será función de una combinación lógica entre el estado actual y las entradas del sistema. A su vez, las salidas dependen del estado actual, configurando la denominada máquina de Moore, aunque también pueden variar en función del estado de las entradas, lo que se conoce como máquina de Mealy[18]. Se podría decir que una máquina de Moore es una particularidad de la máquina de Mealy, en la cual las salidas no varían en función de las entradas.

A modo conceptual, la máquina de estados finitos (MEF) que se implementa en este trabajo es capaz de realizar dos tareas, bien definidas: leer datos desde la memoria FIFO destinada al EP de salida (desde la PC) y escribir datos en la memoria FIFO que corresponde al EP de entrada (hacia el host). Para el diseño de cada una de las operaciones de la MEF implementada en el presente trabajo, se recurrió a la confección del diagrama de flujo para un máquina de estados algorítmica.

Como se indica en la Sección 3.2, las señales de salida de la MEF diseñada que se comunican con el controlador FX2LP son *FIFOADR*, *SLOE*, *SLRD*, *SLWR* y *PKTEND*. El bus de comunicación de datos hacia el interior del FPGA, *dato\_recibido[15:0]* también es una salida del sistema desarrollado en este trabajo. Si bien *FDATA[15:0]* es un puerto de entrada y salida, se maneja también como puerto de salida, cuidando que, cuando funciona como puerto de entrada, se encuentre en alta impedancia el buffer que maneja la salida. Por su parte, los puertos de entrada son *FLAG\_Vacio*, *Enviar\_Datos* y *Flag\_Lleno*.

Una consideración que se hizo en la implementación de este trabajo es que la lectura de las memorias FIFO es prioritaria con respecto a su escritura. Esto se basa en que se espera que este desarrollo sirva de manera fundamental para la lectura de sensores. Dichos sensores serán configurados a través de los datos que lleguen al FPGA y, una vez configurados, deberán transmitir los datos que adquieran del medio en que se encuentre. Se espera entonces, que la información que contienen los datos de configuración posea mayor importancia, ya que podría tener órdenes que detengan los sensores o cambien su funcionamiento. Así mismo, los datos deben ser enviados durante todo el tiempo que el sensor esté adquiriendo, por lo que se espera que los datos de entrada al FPGA sean menos probables que los datos que se envíen.

Con las consideraciones mencionadas, se confeccionó la máquina de estados algorítmica que se observa en el diagrama de flujo de la Figura 3.8. En un estado inicial, todas las salidas se encuentran inactivas (en alto, dado que son activas en bajo). En el caso de salidas que se conectan a un bus (*FIFOADR[1:0]* y *FDATA[15:0]*) se colocan en estado de alta impedancia. El puerto *dato\_recibido*, señalado en el diagrama de la Figura 3.8 como *d\_recibido*, retiene su

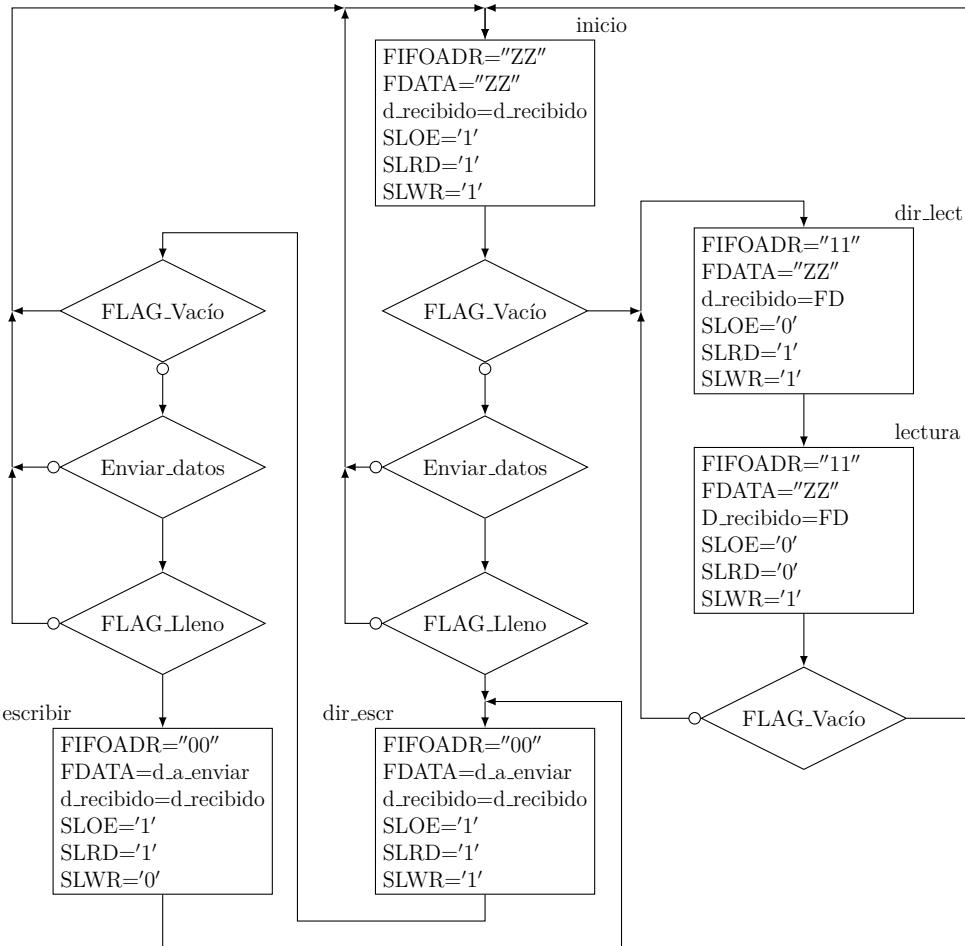


Figura 3.8: Diagrama de flujo de la máquina de estados desarrollada

propio valor.

Cuando el *FLAG\_Vacio* se activa, se procede a la operación de lectura. Si, en cambio, *FLAG\_Vacio* esta inactivo, se debe conocer si el sistema genérico indica que envía datos, a través de *Enviar\_datos*. Si esto ocurre, el sistema de comunicación debe corroborar que la memoria FIFO se encuentra en condiciones de recibir los datos, es decir, que no se encuentre *FLAG\_Lleno* activo. Si estas condiciones ocurren, se debe proceder a la operación de escritura.

La operación de lectura coloca la dirección de la memoria FIFO relacionada al EP de salida (desde el Host), es decir "00". A su vez, se debe activar la salida de la memoria FIFO, activando la señal *SLOE*. El buffer de salida del bus *FDATA[15:0]* debe encontrarse en modo de alta impedancia, para no interferir con la lectura y un registro debe almacenar el valor que se indica en el buffer de entrada. El registro utilizado para almacenar la información leída es *d\_recibido*. Luego, se activa la señal *SLRD*, lo que incrementa el dato de la memoria FIFO. De esta manera, se puede volver a leer la señal de *FLAG\_Vacio* y determinar si se vuelve a implementar una operacion de lectura, o bien, se vuelve al inicio del programa.

Para efectuar la operación de escritura, en primer lugar se debe colocar la dirección de memoria FIFO que apunte al EP de entrada (hacia el Host). La dirección de la memoria FIFO en donde este trabajo escribe datos es "11". El bus de datos se conecta con el puerto interno *dato\_a\_enviar*, representado en el diagrama de la Figura 3.8 por *d\_a\_enviar*. Si las variables de

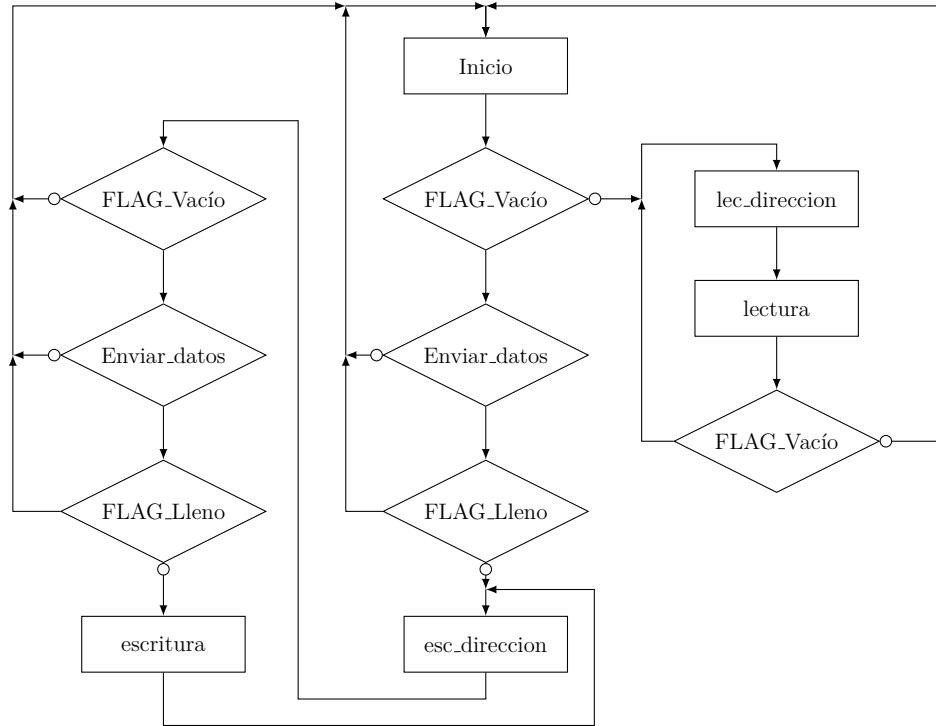


Figura 3.9: Diagrama de simplificado del flujo de la MEF

entrada no se ven alteradas, el estado siguiente activará la señal *SLWR*, de forma tal que los datos colocados en el bus *FDATA* queden almacenados en la memoria FIFO. Luego, el estado siguiente desactiva *SLWR* y vuelve a consultar las variables de entrada.

### 3.4. Síntesis de la Máquina de Estados en VHDL

Considerando las señales que se mencionaron la Sección 3.2 y el diseño de la MEF cuyo diagrama de flujo se observa en la Figura 3.8, se procedió a describir el comportamiento del sistema en VHDL.

Como se mencionó en la Sección anterior, una MEF se compone tres partes: la función del próximo estado, el estado actual y la función de salida. Cada uno de estas partes puede ser descrita en VHDL todo junto en un mismo proceso, o bien en procesos separados. Este trabajo fue implementado mediante un proceso para la función de próximo estado y otro para actualizar el registro del estado actual. Las funciones de salida se implementaron mediante estados combinacionales.

Se presentan a continuación la función de próximo estado y la actualización del estado actual debido a que son, a criterio del autor, los procesos más importantes del desarrollo. El código completo, en donde se realiza la declaración de la entidad, la declaración de las señales, las conexiones internas y se asignan las funciones de salida a los estados en forma concurrente, se puede encontrar en el Apéndice A.

La función de próximo estado, es un proceso que lee el registro *estado\_actual* y las señales de entrada y, en función de su valor, asigna el estado siguiente al registro *prox\_estado*.

Para una mejor comprensión de la descripción de la función de próximo estado, se puede

utilizar la Figura 3.9 en donde se observa una simplificación de cada uno de los estados del diagrama en bloques de la Figura 3.8, en donde se quitaron las variables de salida y se incorporó en cada uno de los estados el nombre con el que se lo asigna en el código VHDL desarrollado. Además, para facilitar la lectura del desarrollo, se colocaron las dos señales de entrada *FLAG\_Vacio* y *FLAG\_Lleno* como activos en alto.

```

architecture Behavioral of fx2lp_interfaz is
    -- maquina de estados de la interfaz
    type estados_mef is
    (
        inicio ,
        lec_direccion , lectura ,
        esc_direccion , escritura
    );
    signal estado_actual , prox_estado: estados_mef := inicio;
begin
    --implementacion de funcion de proximo estado
    proximo_estado: process(estado_actual , flag_lleno ,
        flag_vacio , enviar_dato)
    begin
        case estado_actual is
            when inicio =>
                if flag_vacio = '0' then
                    prox_estado <= lec_direccion ;
                elsif enviar_dato = '1' then
                    if flag_lleno = '0' then
                        prox_estado <= esc_direccion ;
                    else
                        prox_estado <= inicio ;
                    end if;
                else
                    prox_estado <= inicio ;
                end if;
            when lec_direccion =>
                prox_estado <= lectura ;

            when lectura =>
                if flag_vacio = '0' then
                    prox_estado <= lec_direccion ;
                else
                    prox_estado <= inicio ;
                end if;

            when esc_direccion =>
                prox_estado <= escritura ;
    
```

```

when escritura =>
    if enviar_dato = '1' then
        if flag_vacio = '1' and flag_lleno = '0' then
            prox_estado <= esc_direccion;
        else
            prox_estado <= inicio;
        end if;
    else
        prox_estado <= inicio;
    end if;

when others =>
    prox_estado <= inicio;
end case;
end process proximo_estado;
end Behavioral;

```

La transición entre estados es un proceso consta de un reloj que hace transfiere el valor del registro de *prox\_estado* al registro *estado\_actual*. A este reloj, se le acoplan dos temporizadores de habilitación. Esto se debe a que se algunas señales deben respetar ciertos tiempos de establecimiento y ancho de pulso[34]. Cuando el próximo estado es *esc\_dirección* se deben esperar tres ciclos de reloj y en el caso de que el próximo estado sea escritura, *lec\_direccion* o lectura, se debe esperar dos ciclos de reloj. Esto se implementa con dos contadores diferentes, los cuales habilitan o no el cambio de estado. Esto se detalla a continuación:

```

architecture Behavioral of fx2lp_interfaz is
    signal contador3      : natural range 0 to 4 := 0;
    signal contador2      : natural range 0 to 3 := 0;
    signal disp3          : std_logic := '0';
    signal disp2          : std_logic := '0';
begin
    reloj_mef: process (reloj_sist, reset)
    begin
        if reset = '0' then
            estado_actual <= inicio;
        elsif rising_edge(reloj_sist) then
            if contador2 = 0 and contador3 = 0 then
                estado_actual <= prox_estado;
            end if;
        end if;
    end process reloj_mef;

    tempo3: process(reloj_sist, reset, disp3)
    begin
        if reset = '0' then
            contador3 <= 0;

```

```

    elsif rising_edge(reloj_sist) then
        if contador3 > 0 then
            contador3 <= contador3 - 1;
        elsif disp3 = '1' then
            contador3 <= 4;
        end if;
    end if;
end process tempo3;

disp3 <= '1' when (prox_estado = esc_direccion) else '0';

tempo2: process(reloj_sist, reset, disp2)
begin
    if reset = '0' then
        contador2 <= 0;
    elsif rising_edge(reloj_sist)then
        if contador2 > 0 then
            contador2 <= contador2 - 1;
        elsif disp2 = '1' then
            contador2 <= 3;
        end if;
    end if;
end process tempo2;

with prox_estado select
    disp2 <=    '1' when lec_direccion | lectura | esc_direccion ,
    '0' when others;

end Behavioral

```

### 3.5. Verificación funcional de la síntesis

La verificación funcional es una rutina de control que sirve para que el desarrollador se asegure de lo que ha descripto a través de un lenguaje de alto nivel (VHDL en este trabajo), se corresponda con el funcionamiento esperado previo a la descripción. Para efectuar esta verificación, se describe el comportamiento esperado de las señales que de entrada al circuito desarrollado, se simula el sistema y se corrobora que las salidas y se comporten conforme a lo esperado.

Para efectuar la verificación funcional de la máquina de estados desarrollada, se describió en VHDL un ciclo típico de entrada de datos desde la interfaz y se realizó un chequeo de las salidas, tanto hacia el sistema exterior, como hacia el interior. También se simuló la entrada de datos desde el FPGA, generando la rutina de salida de datos hacia la interfaz.

Debido a que es una simulación y no si implementa en físcos, la declaración de la entidad se deja en blanco.

En la implementación de la arquitectura, se declara e instancia el componente bajo prueba, en este caso la MEF y las señales necesarias para su correcta instanciación.

Una vez declarado e instanciado el componente bajo prueba, se generan los estímulos. De estos estímulos, las señales que emulan el reloj y el reset se establecen de forma concurrente. La señal de reloj es la que sincroniza el funcionamiento del sistema. La señal de reset es importante ya que asegura que el dispositivo inicie en el estado determinado por el diseño.

Finalmente, se declara el proceso que contiene los estímulos, emulando el funcionamiento de la interfaz. En un primer momento, las memorias FIFO se encuentran descargadas, es decir, los flags que indican memoria FIFO vacía se encuentran activos (*FLAGB* y *FALGD* en bajo), en tanto los flags que señalan memoria FIFO llena, se encuentran inactivos (*FLAGA* y *FLAGC* en alto). A su vez, el bus de datos *FDATA[15:0]* y el de dirección *FADDR[1:0]* se encuentran en estado de alta impedancia ('Z') y las salidas *SLWR*, *SLRD* y *SLOE* se encuentran inactivas. Estas condiciones se mantuvieron hasta que se desactiva la señal de reset, y luego, se mantienen por 5 ciclos de reloj. Así, se corrobora que ante la ausencia de estímulos, el sistema se mantiene invariable.

Luego, se simula la operación de lectura de la memoria FIFO, con una serie de 3 datos aleatorios. Para este propósito, se desactivó la señal *FLAGB*, el flag vacío que corresponde al EP8, es decir, la memoria FIFO de salida (desde el Host). Seguidamente, se espera a la señal *SLOE*. Esta señal habilita a la interfaz a hacer uso del bus de datos *FDATA[15:0]*, por lo que se simulan datos aleatorios colocados en él, y luego se espera por la respuesta del sistema, el cual debería activar la señal de lectura *SLRD*.

Una vez realizada la operación de escritura se procede a describir la operación de escritura. Para esto, se colocó un dato en el bus de envío de datos, *DATA\_TO\_TX* y se activa la señal de envío de datos *SEND\_REQ*. Una vez que se activa la señal *SLWR*, se está en condiciones de cambiar el dato que se está enviando.

Se realizan también dos pruebas extra, en el desempeño de la máquina de estados desarrollada. En primer lugar, se observa la interrupción del proceso de escritura a través de la señalización del ingreso de datos. Es decir, se debe detener el envío de datos cuando la señal *FLAGB* se active.

Además, se observa que también se debe detener el envío de datos cuando la memoria FIFO que recibe los datos que salen de nuestro sistema se llena. Esta situación es avisada por la interfaz a través de la señal *FLAGA*.



Figura 3.10: Diagrama temporal de la operación de lectura entregado por el simulador

Se puede observar en la Figura 3.10 el diagrama temporal obtenido a partir de la simulación. Se resalta en este diagrama que luego de liberada la señal de reset, el sistema conserva todas las variables en el estado esperado mientras no existe estímulo.

Una vez que es desactivada la señal *FLAGB*, el bus de dirección apunta a la memoria FIFO de entrada de nuestro sistema, es decir, *FADDR[1,0] = "11"*. La señal *SLOE* se coloca en bajo, ese decir que activa el bus de datos, *FDATA[15:0]* para que sea usado por la interfaz. Como se indicó en la descripción realizada, se coloca un dato en el bus y este es volcado en el bus que comunica los datos que ingresan al interior del sistema, *RX\_DATA[15:0]*. Luego, se activa la señal *SLRD* en forma repetida, tal y como se espera.

También se comprueba que el estado actual en cada uno de los momentos es el adecuado. Además, se observa que, aunque se coloquen datos en el bus de salida interno *DATA\_TO\_TX[15:0]* y se active la señal de envío *SEND\_REQ*, el bus de datos que se comunica con la interfaz *FDATA[15:0]* permanece con los datos externos.

Una vez que la MEF alcanzó el estado inicial (*IDLE*, como se ve en la Figura 3.11), procede a la escritura de datos. A partir de allí, mientras no existan datos en la memoria de entrada (*FLAGB* permanezca en alto), la memoria de salida no se encuentre llena (*FLAGA* se encuentre en alto) y sea requerido el envío de datos, el sistema procede a activar en forma sistemática los datos que encunetra en el bus de envío de datos *DATA\_TO\_TX*.

La Figura 3.12 muestra cómo el sistema finaliza la operación de escritura cuando un sucede al menos uno de los eventos considerados para su interrupción. Cuando se activa *FLAGB* mientras se ejecuta la operación de escritura, esta última es interrumpida y el sistema procede a realizar, el próximo estado, la operación de lectura. En el caso de que active la señal *FLAGA*, indicando que la memoria que recibe los datos que envía el FPGA alcanzó el máximo de capacidad, el proceso de escritura es detenido. En ambos casos, no es relevante el valor de la señal de envío de dato *SEND\_REQ*, ya que ambos eventos poseen prioridad a esta.

Una vez realizada la simulación y verificación funcional de la síntesis desarrollada, se está en condiciones de grabar dicha síntesis en el FPGA y se puede proceder a realizar pruebas de funcionamiento más avanzadas. Las pruebas del sistema realizadas se detallan en el capítulo siguiente.

El código completo de la simulación, se pueden leer en el Anexo ??.

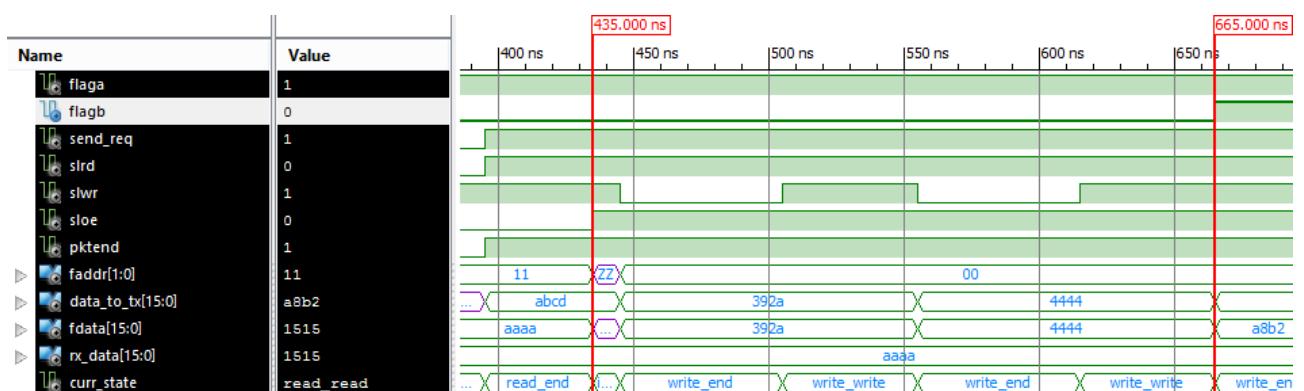


Figura 3.11: Diagrama temporal de la operación de escritura entregado por el simulador

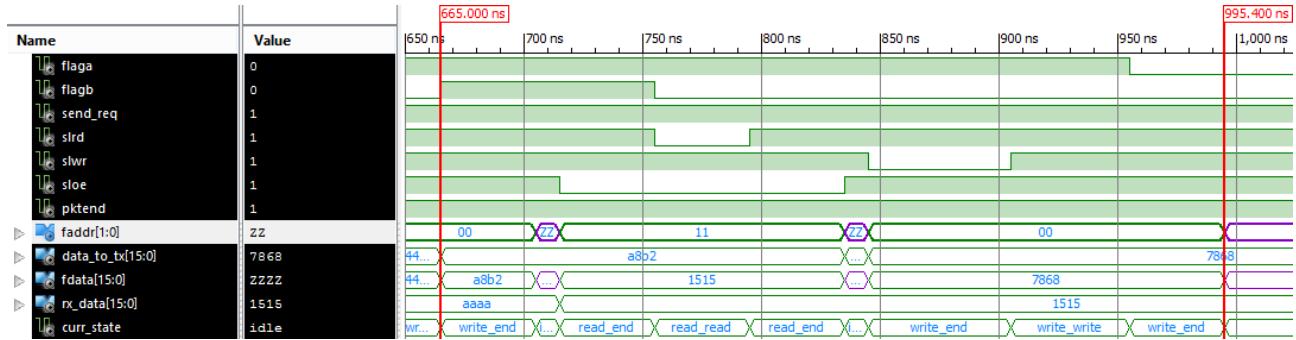


Figura 3.12: Diagrama temporal que muestra las interrupciones de la operación de escritura

## 3.6. Placa de Interconexión

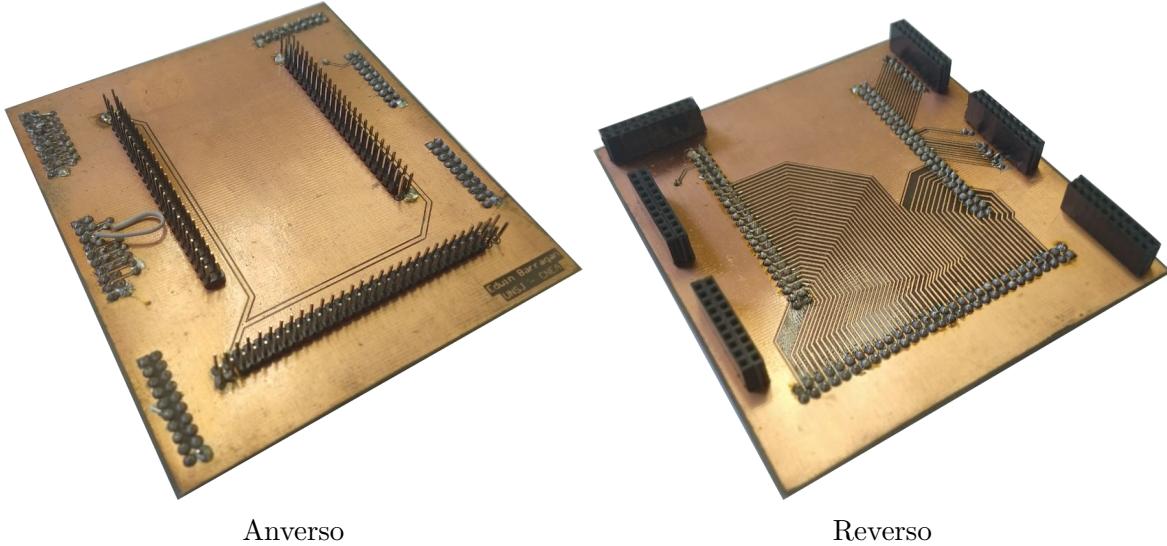


Figura 3.13: Versión 2 de la placa de interconexión

Hasta aquí, se tiene cada una de las partes que componen el desarrollo trabajando correctamente en forma individual. Sin embargo, previo a realizar pruebas del sistema en su totalidad, se debe conectar cada una de las partes en forma física. Los circuitos integrados utilizados para la implementación de la comunicación USB, estos son el controlador FX2LP de Cypress y el FPGA Spartan VI de Xilinx, vienen incorporados en sendas placas de desarrollo. Para la conexión eléctrica de estos dos chips, se desarrolló una placa de interconexión, es decir, un circuito impreso (PCB, del inglés *Printed Circuit Board*) que conecta en forma eléctrica dos o más dispositivos. Esto brinda una conexión mucho más robusta y prolífica que si fuese realizada mediante cables cintas o alambres individuales. La Figura 3.13 muestra el circuito impreso desarrollado. El plano esquemático y el dibujo utilizado para su fabricación se puede consultar en el Anexo ??.

El circuito impreso desarrollado determina en forma definita los puertos que se conectan entre la placa de desarrollo de la interfaz y la del FPGA. En la Tabla 3.2 se puede observar la correspondencia de cada una de las señales de interés del controlador FX2LP con los puertos del FPGA Spartan 6.

<b>FX2LP</b>	<b>Spartan 6</b>
FD15	P50
FD14	P51
FD13	P40
FD12	P41
FD11	P34
FD10	P35
FD9	P32
FD8	P33
FD7	P29
FD6	P30
FD5	P26
FD4	P27
FD3	P23
FD2	P24
FD1	P21
FD0	P22
SLWR	P17
SLRD	P16
SLOE	P6
FLAGA	P12
FLAGB	P14
FLAGC	P15
FLAGD	P11
PKTEND	P10
FIFOADR1	P9
FIFOADR0	P8

Tabla 3.2: Correspondencia entre las señales del controlador FX2LP y el FPGA Spartan 6 fijada por el PCB.

### **3.7. Sumario del capítulo**

Durante el presente capítulo se desarrolló cuales son las señales de control que intervienen y como es su funcionamiento en las operaciones de lectura y escritura externa en las memorias FIFO. En base a ellas se diseñó e implementó la máquina de estados finitos. Se utilizó VHDL como lenguaje de implementación de la MEF, para su posterior síntesis en FPGA. Se realizó también una verificación funcional de la síntesis realizada, a fin de corroborar su correcto funcionamiento. Finalmente se detalló la placa de interconexión realizada para la conexión eléctrica de las placas de desarrollo que contienen al controlador FX2LP y al FPGA Spartan VI y como éste circuito impreso determina la correlación entre los puertos de cada uno de los circuitos integrados.



## Capítulo 4

# Pruebas de funcionamiento y desempeño del sistema desarrollado

Hasta el momento, se explicaron aspectos varios de la norma USB, se seleccionaron las herramientas a utilizar para cumplir con los objetivos del trabajo, se configuraron las distintas partes que componen el sistema y se realizó un circuito que conecte a cada una de ellas.

Con lo anterior, podría esperarse que el sistema desarrollado sea funcional. Sin embargo, esto no puede aseverarse hasta probar que la comunicación entre una PC y el FPGA es efectiva. Por lo tanto es necesario realizar una verificación del sistema.

Para ello, es necesario implementar el sistema como se espera que funcione. Es decir, se debe implementar un sistema en FPGA que pueda enviar y recibir datos desde una PC. A continuación, se desarrollará la implementación sistema de tipo eco, que recibe datos desde una PC y los transmite, luego, en el sentido inverso.

Con este propósito se explicarán los sistemas implementados dentro del FPGA, como así también el programa de computadora realizado con el fin de probar esto y se expondrán los resultados y las conclusiones del trabajo realizado.

### 4.1. Implementación de un sistema genérico en FPGA

Con el objetivo de verificar el sistema desarrollado, se procedió a implementarlo en un sistema mínimo que sea capaz de utilizar la comunicación, de forma tal que el Host pueda establecer un enlace con el FPGA. Para ello, se implementó un sistema eco, es decir, un sistema que recibe los mensajes que envía el Host y, luego, los transmite para que el Host pueda recibirlas. De esta forma, el Host puede reconocer que los datos enviados no fueron perdidos ni modificados.

Para establecer un sistema eco, se deben implementar todas las señales de control que se observan en la Figura 3.6. Estas son, señal de reset, señal de reloj, solicitud sde envío de datos y los datos a enviar. A su vez, se deben poder leer los datos recibidos y las señales *SLWR* y *SLRD*, a través de las cuales el sistema señala el cambio de dato.

Con el objetivo de poder recibir, almacenar y reenviar los datos que llegan desde la interfaz, se sintetizó en el FPGA una memoria FIFO que almacene los datos y, cuando el EP de salida no posee más datos, es decir que el *Flag Vacío* se encuentra activo, retransmita los datos hacia la interfaz hasta que el Host los solicite.

Una vez incorporados los componentes al FPGA, se realizó su validación funcional antes de

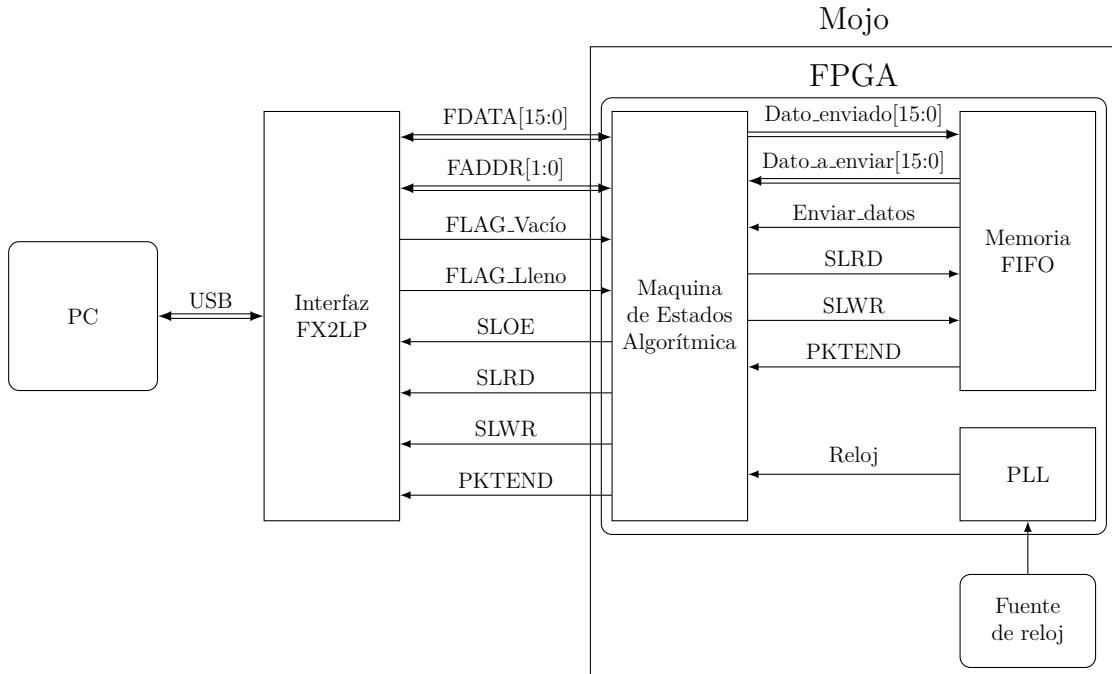


Figura 4.1: Diagrama en bloques del sistema de prueba

ser cargado todo el desarrollo al FPGA para efectuar las pruebas del sistema. A continuación, se detallarán los componentes y señales que se sintetizaron, como así también la verificación realizada.

#### 4.1.1. Declaración de la entidad

Para la implementación y síntesis del sistema, es necesario declarar los puertos que tendrán una correspondencia física con un pin de salida del FPGA en la descripción de mayor jerarquía. El archivo que posee una mayor jerarquía es usualmente llamada top.

Como se conoce cuáles son las señales a través de las cuales el FPGA debe conectarse con la interfaz, es posible declarar la entidad que se sintetizó. Además se declarará la señal de reloj, que proviene desde la placa Mojo v3. El código de descripción en donde se declaró la entidad se muestra a continuación.

```
entity fx2lp_interface_top is
    generic(
        constant in_ep_addr : std_logic_vector(1 downto 0) := "00";
        constant out_ep_addr: std_logic_vector(1 downto 0) := "11";
        constant port_width: integer := 16
    );
    port(
        fdata      : inout std_logic_vector(port_width-1 downto 0);
        faddr     : out  std_logic_vector(1 downto 0);
        slrd      : out  std_logic;
        slwr      : out  std_logic;
        flaga     : in   std_logic;
```

```

    flagb    : in      std_logic ;
    flagc    : in      std_logic ;
    flagd    : in      std_logic ;
    sloe     : out    std_logic ;
    pktend   : out    std_logic ;
    clk_in   : in      std_logic
  );
end fx2lp_interface_top ;

```

#### 4.1.2. Instanciación de la MEF

Dentro de los componentes incorporado al sistema, el más importante es la MEF elaborada en el Capítulo 3, debido a que es el componente que se desea sintetizar, verificar y probar. Para que el sistema reconozca que ese módulo debe incorporarlo al sistema, se declaró como componente los puertos de la entidad elaborada en el capítulo mencionado y se lo instanció como se observa a continuación.

```

architecture fx2lp_interface_arq of fx2lp_interface_top is
  COMPONENT fx2lp_interface
  GENERIC(
    constant in_ep_addr:      std_logic_vector(1 downto 0) := "00";
    constant out_ep_addr: std_logic_vector(1 downto 0) := "11";
    constant port_width: integer := 16
  );
  PORT(
    clk : IN std_logic;
    reset : IN std_logic;
    flaga : IN std_logic;
    flagb : IN std_logic;
    flagc : IN std_logic;
    flagd : IN std_logic;
    send_req : IN std_logic;
    data_to_tx : IN std_logic_vector(15 downto 0);
    fdata : INOUT std_logic_vector(15 downto 0);
    faddr : OUT std_logic_vector(1 downto 0);
    slrd : OUT std_logic;
    slwr : OUT std_logic;
    sloe : OUT std_logic;
    pktend : OUT std_logic;
    rx_data : OUT std_logic_vector(15 downto 0)
  );
  END COMPONENT;
begin
  interface: fx2lp_interface PORT MAP(
    clk => sys_clk,
    reset => reset,

```

```

fdata => fdata ,
faddr => faddr ,
slrd => slrd_sig ,
slwr => slwr_sig ,
flaga => flaga ,
flagb => flagb ,
flagc => flagc ,
flagd => flagd ,
sloe => sloe ,
pktend => pktend ,
send_req => write_req ,
rx_data => din ,
data_to_tx => dout
);
[...]
end fx2lp_interface_arq ;

```

Se puede observar que las constantes genéricas fueron definidas en la declaración del componente . Sin embargo, estas no fueron instanciadas debido a que la configuración a probar era la asignada por defecto.

#### 4.1.3. Otros componenetes y señales de control

##### Generación de Señal de Reloj

Las especificaciones de la interfaz indican que la máxima frecuencia de funcionamiento del reloj debe ser de 48 MHz[34]. La placa de desarrollo Mojo V3, por su parte, posee un oscilador que provee al FPGA una señal de 50 MHz. Para lograr la señal de reloj con la frecuencia adecuada, se utiliza un PLL incorporado dentro del integrado del FPGA.

El PLL fue configurado a través de la herramienta *Core Generator* provista por Xilinx junto con el entorno de desarrollo ISE, utilizado en este trabajo para el desarrollo[35]. A través de esta herramienta, se indicó que la señal de entrada es de 50 MHz. Luego, con el objetivo de poseer señales de frecuencias diferentes por si se presentaran problemas de sincronismo, se seleccionaron señales de salida de 50, 48, 40 y 35 MHz.

La herramienta *Core Generator* de Xilinx entregó un código de VHDL en donde se declara una entidad para que pueda ser utilizada como componente y se instancia el PLL. Luego, la entidad de dicho código se declaró e instanció en la descripción del sistema de pruebas de la siguiente forma:

```

architecture fx2lp_interface_arq of fx2lp_interface_top is
[...]
component clk_wiz_v3_6
port(
    CLK_IN1 : in std_logic ;
    CLK_OUT1: out std_logic ;
    CLK_OUT2: out std_logic ;
    CLK_OUT3: out std_logic ;

```

```

CLK_OUT4: out std_logic;
RESET    : in  std_logic;
LOCKED   : out std_logic
);
end component;
[...]
begin
  pll : clk_wiz_v3_6
    port map(
      CLK_IN1    => clk_in ,
      CLK_OUT1   => pll_50 ,
      CLK_OUT2   => pll_48 ,
      CLK_OUT3   => pll_40 ,
      CLK_OUT4   => pll_35 ,
      RESET      => '0',
      LOCKED    => locked
    );
    sys_clk <= pll_48 ;
[...]
end fx2lp_interface_arq ;

```

Se puede notar que las señales *pll\_50*, *pll\_48*, *pll\_40* y *pll\_35* se utilizaron de manera especial para seleccionar en forma rápida la frecuencia que se le asigna a la señal de reloj del sistema. La señal asignada al reloj del sistema fue la salida del PLL que posee una frecuencia de 48 MHz.

### Generación de señal de reset

La señal de reset fue generada por un contador con el objetivo de asegurar que, al conectar el FPGA, el sistema espere que finalice cualquier transitorio que pueda causar respuestas inesperadas e inicie con los valores iniciales preestablecidos.

```

init_rst: process(sys_clk)
begin
  if rst_cont /= 0 then
    if rising_edge(sys_clk) then
      rst_cont <= rst_cont - 1;
    end if;
  end if;
end process init_rst;

reset <= '1' when rst_cont = 0 else '0';

```

### Implementación de la memoria FIFO en el FPGA

La memoria FIFO sintetizada en el FPGA también se obtuvo a través de la herramienta *Core Generator* de Xilinx. La configuración seleccionada generó una memoria FIFO de 511

bytes (la configuración de la herramienta pierde un byte al generar la memoria), con puertos de entrada y salida dedicados, es decir uno de entrada y uno de salida, un bus de 16 bits de ancho. Posee señal de reconocimiento de escritura. También se dotó a la memoria generada con entradas de reloj y habilitación de bus independientes tanto para el puerto de entrada como el puerto de salida.

Con la configuración mencionada, *Core Generator* entregó una plantilla para utilizar la memoria generada. Esta plantilla se utilizó para declarar e instanciar el componente en el sistema implementado.

```
architecture fx2lp_interface_arq of fx2lp_interface_top is
[...]
COMPONENT fifo_generator_v9_3
  PORT (
    rst : IN STD_LOGIC;
    wr_clk : IN STD_LOGIC;
    rd_clk : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    full : OUT STD_LOGIC;
    empty : OUT STD_LOGIC;
    valid : OUT STD_LOGIC
  );
END COMPONENT;
[...]
begin
  fifo : fifo_generator_v9_3
    PORT MAP (
      rst => not reset,
      wr_clk => sys_clk,
      rd_clk => sys_clk,
      din => din,
      wr_en => wr_en,
      rd_en => rd_en,
      dout => dout,
      full => fifo_full,
      empty => fifo_empty,
      valid => valid
    );
[...]
end fx2lp_interface_arq;
```

Luego, con todos los componentes declarados e instanciados, se elaboró una pequeña máquina de estados que sea capaz de colocar y extraer los datos en la memoria FIFO generada por la herramienta *Core Generator*.

## 4.2. Pruebas de la comunicación entre el FPGA y la PC

### 4.2.1. Elección de la biblioteca libusb-1.0

La tercer parte en la que se divide el trabajo es relativa a la comunicación entre la interfaz y una PC. Ya que la interfaz se encarga en gran medida de lo relativo al empaquetamiento, codificación y decodificación y que las PC, por su parte, vienen equipadas con el hardware necesario, este trabajo debe implementar el software que comande y gestione, desde el sistema operativo el correcto acceso a los datos que se envían y reciben. Para la elaboración de software que permita el manejo de los puertos USB, se utiliza la biblioteca **libusb**.

**libusb** es una biblioteca de código abierto, muy bien documentada, escrita en C, que brinda acceso genérico a dispositivos USB. Las características de diseño que persigue el equipo de desarrollo que mantiene la biblioteca es que sea multiplataforma, modo usuario y agnóstico de versión[36].

- Multiplataforma: Se apunta a que cualquier software que contenga esta biblioteca pueda ser compilado y ejecutado en la mayor cantidad de plataformas posibles, dotando al software de portabilidad, es decir, esta biblioteca puede ser ejecutada en Windows, Linux, OS X, Android y otras plataformas sin necesidad de realizar cambios en el código.
- Modo usuario: No se requiere acceso privilegiado de ningún tipo para poder ejecutar programas escritos con esta biblioteca.
- Agnóstico de versión: Sin importar la versión de la norma USB que se utilice, el programa se podrá comunicar siempre con el dispositivo USB que se requiera.

La biblioteca **libusb** no posee un autor formal. Es decir, no hay una persona, empresa u organización formal que se encargue de la creación y el mantenimiento del software. Existe una comunidad de más de 130 desarrolladores que en forma voluntaria cooperan en el mantenimiento y desarrollo de esta biblioteca. Se garantiza así que el proyecto esté documentado en forma detallada, existiendo amplios ejemplos y tutoriales de su uso.

Se elige esta biblioteca para la realización del software que gestionara el envío y la recepción de datos debido a su amplio soporte, la factibilidad de ejecutarlo en diferentes sistemas operativos y por ser totalmente gratuito.

## 4.3. Resultados

### 4.4. Conclusiones

### 4.5. Trabajos Futuros



# Bibliografía

- [1] R. Pallàs-Areny and J. G. Webster, *Sensors and signal conditioning*. Wiley-Interscience, 2001.
- [2] D. M. Considine, *Encyclopedia of instrumentation and control*. McGraw-Hill, Inc., 1971.
- [3] A. Perez Garcia, “Curso de instrumentación,” p. 261, 2008.
- [4] J. Fraden, *Handbook of modern sensors: physics, designs, and applications*. New York, NY: Springer New York, 2010.
- [5] E. Slawiński and V. Mut, *Humanos y máquinas inteligentes: conocimiento educativo sobre el comportamiento interno de robots que actúan juno y para el hombre*. Saarbrücken, Alemania: Editorial Académica Española, 2011.
- [6] K. Ogata, *Modern control engineering*. Aeeizh, 2002.
- [7] G. Binnig and H. Rohrer, “Scanning tunneling microscopy,” *Surface Science*, vol. 126, pp. 236–244, mar 1983.
- [8] R. Turchetta, K. R. Spring, and M. W. Davidson, “Digital Imaging in Optical Microscopy - Introduction to CMOS Image Sensors,” (accessed in July 2019).
- [9] S. Mendis, S. Kemeny, and E. Fossum, “CMOS active pixel image sensor,” *IEEE Transactions on Electron Devices*, vol. 41, pp. 452–453, mar 1994.
- [10] C. Hu-Guo, J. Baudot, G. Bertolone, A. Besson, A. S. Brogna, C. Colledani, G. Claus, R. D. Masi, Y. Degerli, A. Dorokhov, G. Doziere, W. Dulinski, X. Fang, M. Gelin, M. Goffe, F. Guilloux, A. Himmi, K. Jaaskelainen, M. Koziel, F. Morel, F. Orsini, M. Specht, Q. Sun, I. Valin, and M. Winter, “CMOS pixel sensor development: a fast read-out architecture with integrated zero suppression,” *Journal of Instrumentation*, vol. 4, pp. P04012–P04012, apr 2009.
- [11] J. Baudot, G. Bertolone, A. Brogna, G. Claus, C. Colledani, Y. Değerli, R. De Masi, A. Dorokhov, G. Dozière, W. Dulinski, M. Gelin, M. Goffe, A. Himmi, F. Guilloux, C. Hu-Guo, K. Jaaskelainen, M. Koziel, F. Morel, F. Orsini, M. Specht, I. Valin, G. Voutsinas, and M. Winter, “First test results of MIMOSA-26, a fast CMOS sensor with integrated zero suppression and digitized output,” *IEEE Nuclear Science Symposium Conference Record*, pp. 1169–1173, 2009.

---

**BIBLIOGRAFÍA**

---

- [12] M. Pérez, J. Lipovetzky, M. Sofo Haro, I. Sidelnik, J. J. Blostein, F. Alcalde Bessia, and M. G. Berisso, “Particle detection and classification using commercial off the shelf CMOS image sensors,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 827, pp. 171–180, aug 2016.
- [13] M. Pérez, J. J. Blostein, F. A. Bessia, A. Tartaglione, I. Sidelnik, M. S. Haro, S. Suárez, M. L. Gimenez, M. G. Berisso, and J. Lipovetzky, “Thermal neutron detector based on COTS CMOS imagers and a conversion layer containing Gadolinium,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 893, pp. 157–163, jun 2018.
- [14] C. L. Galimberti, F. Alcalde Bessia, M. Perez, M. G. Berisso, M. Sofo Haro, I. Sidelnik, J. Blostein, H. Asorey, and J. Lipovetzky, “A Low Cost Environmental Ionizing Radiation Detector Based on COTS CMOS Image Sensors,” in *2018 IEEE Biennial Congress of Argentina (ARGENCON)*, pp. 1–6, IEEE, jun 2018.
- [15] T. Hizawa, J. Matsuo, T. Ishida, H. Takao, H. Abe, K. Sawada, and M. Ishida, “ $32 \times 32$  pH image sensors for real time observation of biochemical phenomena,” *TRANSDUCERS and EUROSENSORS '07 - 4th International Conference on Solid-State Sensors, Actuators and Microsystems*, pp. 1311–1312, 2007.
- [16] ON Semiconductor, “NOIP1SN0300A Global Shutter CMOS Image Sensors,” 2014.
- [17] N. Ida, *Engineering Electromagnetics*. Cham: Springer International Publishing, 3th ed., 2015.
- [18] J. F. Wakerly, *Digital Design: principles and practices*, vol. 1. Pearson, 1999.
- [19] M. Perez, F. Alcalde, M. S. Haro, I. Sidelnik, J. J. Blostein, M. G. Berisso, and J. Lipovetzky, “Implementation of an ionizing radiation detector based on a FPGA-controlled COTS CMOS image sensor,” in *2017 XVII Workshop on Information Processing and Control (RPIC)*, pp. 1–6, IEEE, sep 2017.
- [20] R. Biswas, *An Embedded Solution for JPEG 2000 Image Compression Based Back-end for Ultrasonography System*. PhD thesis, IIT, Kharagpur, 2018.
- [21] T. Yanagisawa, T. Ikenaga, Y. Sugimoto, K. Kawatsu, M. Yoshikawa, S.-i. Okumura, and T. Ito, “New NEO search technology using small telescopes and FPGA,” in *2018 IEEE Aerospace Conference*, vol. 2018-March, pp. 1–7, IEEE, mar 2018.
- [22] H. H. Goldstine and A. Goldstine, “The Electronic Numerical Integrator and Computer (ENIAC),” *Mathematical Tables and Other Aids to Computation*, vol. 2, p. 97, jul 1946.
- [23] S. of Cable Telecommuniocations Engineers, *American National Standard ANSI/SCTE 07 2006. Digital Tansmission Standard for Cable Television*. Society of Cable Telecommunications Engineers, Inc., 2006.
- [24] I. Micron Technology, “1 / 2-Inch Megapixel CMOS Digital Image Sensor MT9M001C12STM (Monochrome),” pp. 1–35, 2004.

## BIBLIOGRAFÍA

---

- [25] IEEE Computer Society, *IEEE Standard for Ethernet*, vol. 2018. 2018.
- [26] IEEE Computer Society, *Part 11 : Wireless LAN Medium Access Control ( MAC ) and Physical Layer ( PHY ) Specifications IEEE Computer Society Specific requirements Part 11 : Wireless LAN Medium Access Control ( MAC ) and Physical Layer ( PHY ) Specifications*, vol. 2012. 2016.
- [27] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips, *Universal Serial Bus Specification*, vol. Revision 2.0. 2000.
- [28] “Usb hardware.” [https://en.wikipedia.org/wiki/USB\\_hardware](https://en.wikipedia.org/wiki/USB_hardware). Ingreso: 8 de agosto del 2019.
- [29] T. Riihonen, *Desing and analysis of duplexing Modes and Forwarding Protocols for OFDM(A) Relay Links*. PhD thesis, 2015.
- [30] B. Sklar, *Digital communications: fundamentals and applications*. 2001.
- [31] Cypress Semiconductor, “EZ-USB ® Technical Reference Manual,” tech. rep., 2014.
- [32] HW Group, “<https://www.hw-group.com/software/hercules-setup-utility>” - acceso 17/03/2020.”
- [33] P. Kumar, “AN58009 - Serial (UART) Port Debugging of EZ-USB ® FX1/FX2LP ™ Firmware Serial Debug Files,” Tech. Rep. AN58009-rev E, Cypress Semiconductor, 2017.
- [34] Cypress, “CY7C68013A, CY7C68014A, CY7C68015A, CY7C68016A, EZ-USB(R) FX2LP(TM) USB Microcontroller High-Speed USB Peripheral Controller,” 2017.
- [35] Xilinx Inc., “Working with CORE Generator IP - Acceso 03/03/2020.”
- [36] libusb, “libusb 1.0 <https://libusb.info/>” - acceso: 04/11/2019.”

*BIBLIOGRAFÍA*

---

# Apéndice A

## Códigos de descripción escritos en VHDL

### A.1. Código de implementación de la MEF que controla la interfaz FX2LP

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fx2lp_interface is
    generic(
        constant ent_ep_addr:    std_logic_vector(1 downto 0) := "00";
        constant sal_ep_addr:    std_logic_vector(1 downto 0) := "11";
        constant ancho_bus: integer := 16
    );
    port(
        -- desde y hacia el sistema
        reloj      : in std_logic;
        reset      : in std_logic;
        enviar_datos : in std_logic;
        dato_recibido : out std_logic_vector(ancho_bus-1 downto 0);
        dato_a_enviar : in std_logic_vector(ancho_bus-1 downto 0)
        -- desde y hacia la interfaz
        fdata     : inout std_logic_vector(ancho_bus-1 downto 0);
        -- entrada y salida de datos FIFO
        faddr     : out std_logic_vector(1 downto 0);
        -- canal FIFO
        slrd      : out std_logic;    -- senal de lectura
        slwr      : out std_logic;    -- senal de escritura
        flaga     : in  std_logic;    -- EP2_full-->esc_full_flag
        flagb     : in  std_logic;    -- EP8_empty-->lec_empty_flag
        flagc     : in  std_logic;    -- EP8_full-->lec_full_flag
        flagd     : in  std_logic;    -- EP2_empty-->esc_empty_flag
        sloe     : out std_logic;    -- senal de habilitacion de salida
    );
end entity;
```

```

      pktend  : out    std_logic;
30   );
end fx2lp_interface;

architecture Behavioral of fx2lp_interface is

35   signal slwr_int : std_logic := '1';
   signal slrd_int : std_logic := '1';
   signal sloe_int : std_logic := '1';
   signal pktend_int : std_logic := '1';
   signal faddr_int : std_logic_vector(1 downto 0) := "ZZ";
40   signal fdata_sal : std_logic_vector(ancho_bus-1 downto 0);
   signal fdata_ent : std_logic_vector(ancho_bus-1 downto 0);
   signal lec_eflag : std_logic := '1';
   signal lec_fflag : std_logic := '1';
   signal esc_eflag : std_logic := '1';
45   signal esc_fflag : std_logic := '1';

   signal reloj_sist : std_logic;

-- señales de temporización
50   signal contador3 : natural range 0 to 4 := 0;
   signal contador2 : natural range 0 to 3 := 0;
   signal disp3, disp2 : std_logic := '0';

-- maquina de estados de la interfaz
55   type estados_mef is
   (
      inicio,
      lec_direccion, lectura,
      esc_direccion, escritura
60   );
   signal estado_actual, prox_estado : estados_mef := inicio;

begin
-- conexión de señales internas hacia los puertos
65   reloj_sist <= reloj;

   slwr  <= slwr_int;
   slrd  <= slrd_int;
   sloe  <= sloe_int;
70   faddr <= faddr_int;
   pktend <= pktend_int;

   esc_fflag <= not flaga;
   lec_eflag <= not flagb;

75   dato_recibido <= fdata_ent;

```

```

fdata_sal <= dato_a_enviar;

-- señalizacion
80   with estado_actual select
      faddr_int <= sal_ep_addr when lec_dir | lectura,
                                ent_ep_addr when esc_direccion | escritura,
                                (others => 'Z') when others;

85   slwr_int <= '0' when prox_estado = esc_direccion else
                  '1';

     slrd_int <= '0' when estado_actual = lec_dir else
                  '1';

90   pktend_int <= ((not lec_eflag) or enviar_datos);

with estado_actual select
      sloe_int <= '0' when lectura | lec_dir,
                  '1' when others;

95   with estado_actual select
      --dout->fdata_sal
      fdata <= fdata_sal when escritura | esc_direccion,
                            (others => 'Z') when others;

with estado_actual select
      fdata_ent <= fdata when lectura | lec_dir,
                            fdata_ent when others;

100
--implementacion de la maquina de estados
interfaz_mef: process(estado_actual, esc_fflag, lec_eflag, enviar_datos)
begin
  case estado_actual is
    when inicio =>
      if lec_eflag = '0' then
        prox_estado <= lectura;
      elsif enviar_datos = '1' then
        if esc_fflag = '0' then
          prox_estado <= escritura;
        else
          prox_estado <= inicio;
        end if;
      else
        prox_estado <= inicio;
      end if;
    when lec_dir =>
      prox_estado <= lectura;
  end case;
end process;

```

```
125      when lectura =>
126          if lec_eflag = '0' then
127              prox_estado <= lec_dir ;
128          else
129              prox_estado <= inicio ;
130          end if;

135      when esc_direccion =>
136          prox_estado <= escritura ;

140      when escritura =>
141          if enviar_datos = '1' then
142              if lec_eflag = '1' and esc_fflag = '0' then
143                  prox_estado <= esc_direccion ;
144              else
145                  prox_estado <= inicio ;
146              end if;
147          else
148              prox_estado <= inicio ;
149          end if;

150      when others =>
151          prox_estado <= inicio ;
152      end case;
153  end process interfaz_mef;

-- temporizaciones
reloj_mef: process (reloj_sist , reset)
begin
155      if reset = '0' then
156          estado_actual <= inicio ;
157      elsif rising_edge(reloj_sist) then
158          if contador2 = 0 and contador3 = 0 then
159              estado_actual <= prox_estado ;
160          end if;
161      end if;
162  end process reloj_mef;

tempo3: process(reloj_sist , reset , disp3)
begin
165      if reset = '0' then
166          contador3 <= 0;
167      elsif rising_edge(reloj_sist) then
168          if contador3 > 0 then
169              contador3 <= contador3 - 1;
170          elsif disp3 = '1' then
171              contador3 <= 4;
```

```
175      end if;
      end if;
end process tempo3;

disp3 <= '1' when (prox_estado = esc_direccion) else '0';

180 tempo2: process(reloj_sist, reset, disp2)
begin
  if reset = '0' then
    contador2 <= 0;
  elsif rising_edge(reloj_sist)then
    if contador2 > 0 then
      contador2 <= contador2 - 1;
    elsif disp2 = '1' then
      contador2 <= 3;
    end if;
  end if;
end process tempo2;

185 with prox_estado select
  disp2 <=    '1' when lectura | lec_dir | escritura ,
  '0' when others;

190
195 end Behavioral;
```

## A.2. Código de validación de la MEF

## A.3. Código de síntesis del sistema de prueba