

UNIVERSIDAD NACIONAL DE SAN JUAN
FACULTAD DE INGENIERIA
Departamento de Electrónica y Automática



**Universidad Nacional
de San Juan**

Trabajo Final
COMUNICACIÓN USB 2.0 PARA SISTEMAS
CIENTÍFICOS IMPLEMENTADOS EN FPGA
Informe

Edwin Barragán
Autór

Ing. Cristian Sistera Mgtr. Ing. Martín Pérez Dr. Marcelo Segura
Asesores

Agradecimientos

Colocar aquí todos los agradecimientos que el alumno considera.

Índice general

1. Introducción	7
1.1. Motivación	7
1.2. Protocolos disponibles para la transmisión de datos entre PC y FPGA	11
1.3. Bus Serial Universal 2.0	15
1.3.1. Descripción general de un sistema USB	16
1.3.2. Dispositivos que componen un sistema USB	18
1.3.3. Paquetes USB	19
1.3.4. Tipos de Transferencias	21
1.3.5. Descriptores	23
1.4. Objetivos	24
1.4.1. Objetivo Principal	24
1.4.2. Objetivos Particulares	24
1.5. Estructura del Informe	24
1.6. Sumario del capítulo	25
2. Interfaz USB	27
2.1. Elección de la Interfaz	27
2.2. El controlador FX2LP EZ-USB y su configuración	28
2.2.1. Microcontrolador Cypress 8051 Mejorado	30
2.2.2. Frecuencia de trabajo del sistema	31
2.2.3. Memoria FIFO	32
2.2.4. Modos de entrada y salida automáticos	34
2.2.5. Encabezado y declaraciones importantes	36
2.2.6. Descriptores USB	37
2.3. Depuración y verificación de funcionamiento	43
2.3.1. Biblioteca FX2LPSerial	43
2.3.2. Testigos LED	44
2.3.3. Prueba de envío y recepción de datos	45
2.4. Sumario del capítulo	46
3. Desarrollo del sistema maestro para la comunicación entre la FPGA y la interfaz Cypress	47
3.1. Elección del FPGA	47
3.2. Señales de comunicación de la Máquina de Estados Finitos	49
3.2.1. Señales de comunicación el FPGA y el controlador FX2LP	50
3.2.2. Comunicación interna del FPGA	52

ÍNDICE GENERAL

3.3. Diseño de la Máquina de Estados Finitos	54
3.4. Síntesis de la Máquina de Estados en VHDL	55
3.5. Verificación funcional de la síntesis	59
3.6. Placa de Interconexión	61
3.7. Sumario del capítulo	62
4. Pruebas de funcionamiento y desempeño del sistema desarrollado	65
4.1. Sistema de pruebas implementado en FPGA	65
4.1.1. Verificación Funcional del sistema de pruebas	67
4.1.2. Carga del sistema de pruebas en el FPGA	69
4.2. Desarrollo del programa de PC para enviar y recibir datos	70
4.2.1. Elección de la biblioteca libusb-1.0	70
4.2.2. Programa de PC desarrollado	70
4.3. Pruebas de la comunicación entre el FPGA y la PC	71
4.4. Resultados	72
4.5. Sumario del capítulo	73
5. Conclusiones	75
5.1. Consideraciones Finales	76
Apéndice A. Archivos de configuración del controlador FX2LP	81
Apéndice B. Códigos de la síntesis en FPGA	99
Apéndice C. Códigos para PC del programa de pruebas	121
Apéndice D. Esquemático de la Placa de Interconexión	135

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 [1–3]. 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 [4–6].

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

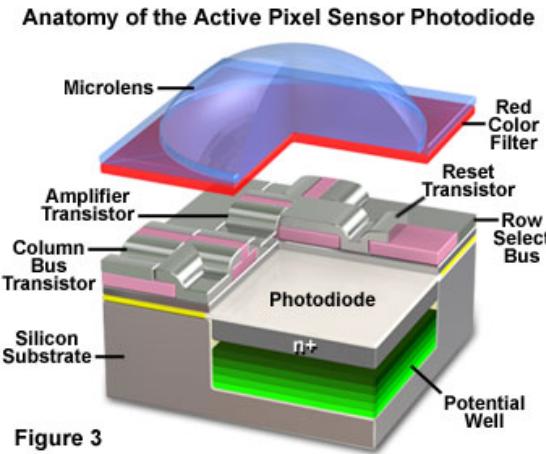


Figura 1.1: Esquema físico de un APS [8]

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 intensidades 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 unos pocos transductores y se desplaza un sistema de espejos 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 Gd_2O_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.

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 más 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 el calor 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*). Así, 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 μ 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 Mbit s⁻¹ [23]. Por su parte, la serie de sensores para adquirir imágenes monocromáticas MT9M001, comercializado por ON Semiconductors posee 1280x1024 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 310 Mbit s⁻¹.

Un requerimiento que normalmente se busca en periféricos 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

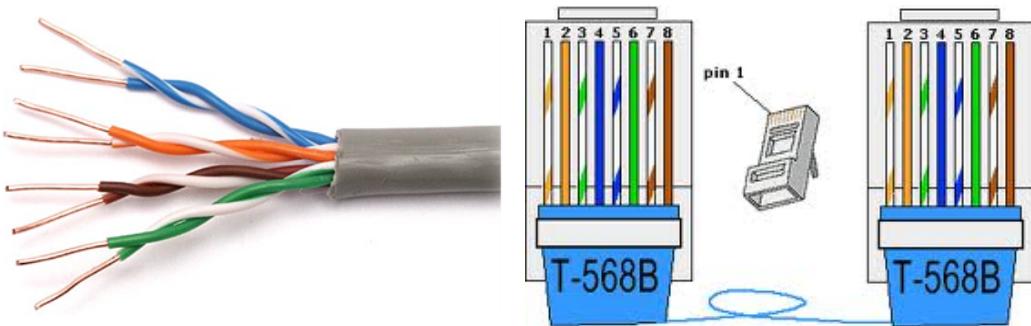


Figura 1.2: Par Trenzado y un dibujo de su ficha de conexión.

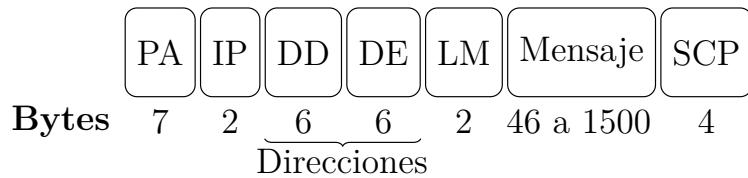
mediante cables; Wi-Fi, utilizado para el acceso a la red de forma inalámbrica; y USB, dirigido a la comunicación de periféricos con la PC.

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. Queda 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 PC 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

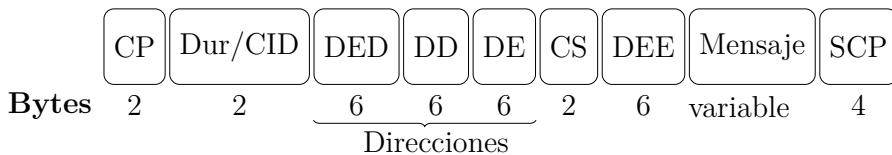
DE: Dirección de Emisión

LM: Longitud del Mensaje

SCP: Secuencia de chequeo del paquete

Figura 1.3: Estructura de un paquete Ethernet

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.



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

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 2 B de control que indican el tipo de paquete a enviar. Luego siguen 2 B 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 B de direcciones del enrutador que recibe los datos, el nodo emisor y el destinatario. Continúan, 2 B de control de secuencia se utilizan para fragmentar transmisiones largas. Continua un campo más para dirección que corresponde a la red emisora de 6 B. 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 B) y un código de chequeo por redundancia cíclica de 32 bit (4 B).

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 Mbit s^{-1} y la segunda hasta 12 Mbit s^{-1} . USB 2.0 fue presentado en Septiembre del 2000 y es capaz

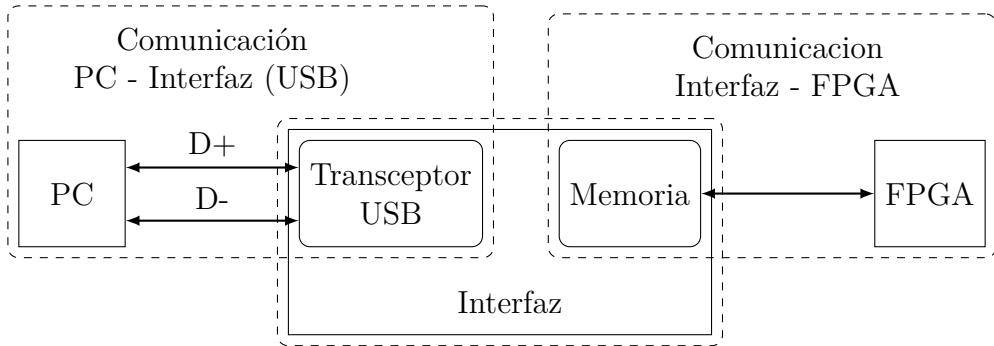


Figura 1.5: Partes en que se desglosa el trabajo

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

Se elige para el desarrollo de este trabajo la norma USB 2.0 debido a que presenta una tasa de transferencia de datos suficiente para la transmisión de imágenes. A su vez, resulta ideal para los objetivos buscados, ya que se encuentra presente en la mayoría de las PC y no interfiere 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 que serán usuarios 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 diferentes entre sí, tales como fichas, niveles de tensión, velocidades, drivers, etc. Esta diversidad no solo dificultaba a los usuarios, quienes no podían estar al día con las posibilidades y dispositivos que el mercado ofrecía, sino que también complicaba a las mismas empresas productoras, porque la introducción de un nuevo producto requería de mucho soporte extra para poder compatibilizar con 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 básicos, como mouse y teclado.

El diseño original de la norma USB buscaba resolver tres problemáticas interrelacionadas: 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 [27]. Para satisfacer estas tres demandas, la norma USB 2.0 estableció un conjunto de metas apuntadas 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. Estas metas, son alcanzadas a través de una interfaz estándar, ancho de banda que soporta comunicaciones audiovisuales de calidad aceptable y un bajo costo.

En esta sección 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 denominado Host. Es decir, la comunicación se realiza siempre a través de una única línea a la que se conectan todos los dispositivos. Dicha configuración recibe el nombre de 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 sistema maestro. El maestro se encarga de solicitar a cada dispositivo su intervención. Posteriormente, el dispositivo indicado debe responder al pedido del maestro. Este esquema es conocido como maestro-esclavo. De esta forma, la comunicación USB asegura que el bus sea utilizado por un dispositivo a la vez para enviar o recibir datos.

En un sistema USB solo un dispositivo puede ser maestro. Este rol lo ejerce generalmente una PC, pero podría ser 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 [27].

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

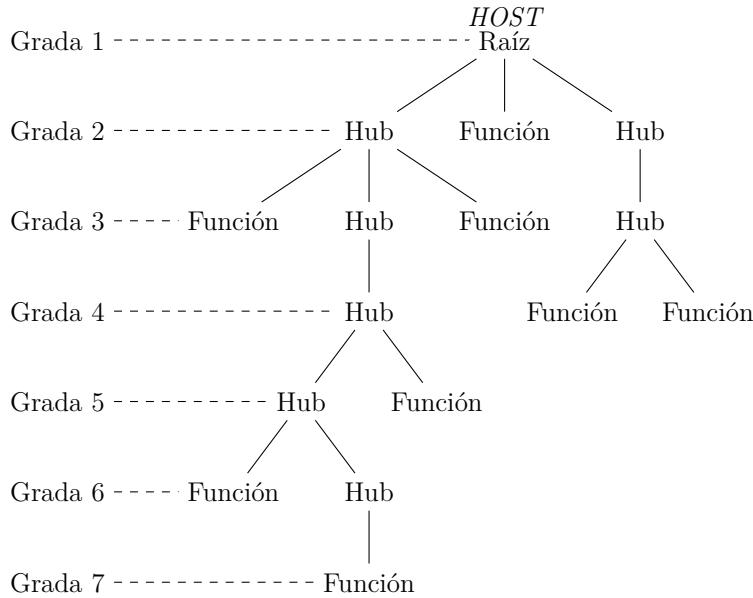


Figura 1.6: Topología de un sistema USB

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*)

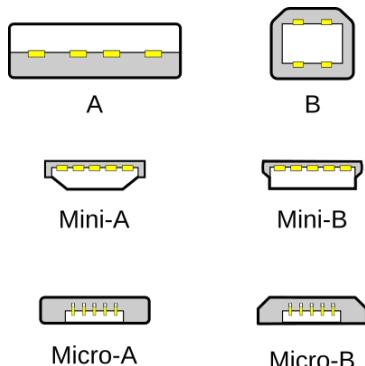


Figura 1.7: Tipos de conectores USB. Los tipo A deben ser usados en el extremo del Host y los tipo B hacia los periféricos [28]

implica una tasa de bit de 480 Mbit s^{-1} , máxima velocidad (*Full-Speed*) posee una tasa de bit de 12 Mbit s^{-1} y baja velocidad (*Low-Speed*) transmite a una tasa de bit de 1.5 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 cierta cantidad de recursos para almacenamiento y procesamiento, por lo general este rol es llevado a cabo por una PC. 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. Estas, desde un punto de vista de software, son independientes unas de otras, por lo que cuando un programa, llamado cliente, necesita utilizar una función, puede acceder a ésta en forma directa, 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 [27]. 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 toda velocidad, es decir, con una señal de 12 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 $125 \mu\text{s}$ cada uno. Los fragmentos de información que envían los dispositivos 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 y 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 bit, 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 da 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 B. Cada byte enviado se ordena con el bit menos significativo (LSb del inglés *Less Significative bit*) primero y el bit más 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.

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.

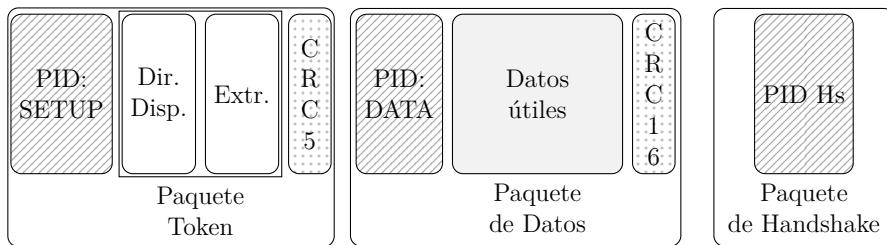


Figura 1.8: Formatos de paquetes

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 μ 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

Existen cuatro tipos de transferencias definidas por la norma USB: Transferencias de Control, transferencias en masa, transferencias isócronas y transferencias de interrupción. Cada una de ellas tiene un propósito y características diferentes.

Cada extremo presente en un dispositivo USB, se podrá comunicar con el Host a través de un único tipo de transferencias, la cual será informada por el dispositivo durante su conexión. 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.

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 isócronas

El término isócrono se refiere a eventos que ocurren en períodos de tiempo con igual duración. En sistemas digitales, se emplea el término para aquellos sistemas sincrónicos que poseen la particularidad de que suceden una cantidad determinada de eventos en intervalos idénticos 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 μ s, 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 isócrono, aún con señales de reloj provenientes de fuentes diferentes.

La principal característica de las transferencias isócronas 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 isócronas. 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 isócrona 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 que transmite a toda velocidad, 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 cliente 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 número que identifica al fabricante y otro que corresponde al producto, es decir al dispositivo. Esto puede ser utilizado por el Host para ejecutar el software de control adecuado 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 función de impresión y otra para la de escaneo. A su vez, cada interfaz puede variar el ancho de banda requerido a través de una característica denominada *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, al cambiar de configuración, todos los extremos y las interfaces son desactivadas.

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 que pueda ser utilizada en aplicaciones científicas.

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.

1.5. Estructura del Informe

El presente trabajo puede ser descompuesto en tres etapas bien diferenciados. La primera de ellas está referida a la selección y configuración de la interfaz. La segunda es dedicada al FPGA, su elección y la elaboración de un módulo que permita comunicar el FPGA a la interfaz. Luego,

1. Introducción

en la tercer etapa, se elabora un sistema que ensambla las dos etapas anteriores. Este esquema se expone, a lo largo de este trabajo, en cinco 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. **Interfaz USB:** Se justifica la elección de la interfaz USB, compuesta por el controlador FX2LP EZ-USB comercializado por Cypress Semiconductor. Se presenta la arquitectura, configuración y código desarrollado para el presente trabajo.
3. **Desarrollo del sistema maestro para la comunicación entre la FPGA y la interfaz Cypress:** Este capítulo detalla el desarrollo de una Máquina de Estados Finita para implementar la comunicación entre la FPGA y la interfaz USB. Se justifica la elección del FPGA Spartan 6 fabricado por Xilinx Inc., utilizado en este trabajo. La Máquina de Estados Finita es descripta en VHDL y sintetizada en un FPGA. También se describe un circuito impreso realizado para conectar el FPGA con la Interfaz USB.
4. **Pruebas de funcionamiento y desempeño del sistema desarrollado:** Se detalla el desarrollo de un sistema de pruebas que conecta el FPGA con la interfaz, y a través de una memoria FIFO sintetizada en FPGA, se envían y se reciben paquetes en y desde una PC, respectivamente. Los paquetes son generados y transmitidos desde la PC a través de un programa elaborado para este trabajo. Finalmente se exponen las pruebas, los resultados de ellas y las conclusiones de este trabajo.
5. **Conclusiones:** Se presentan las conclusiones obtenidas a lo largo del trabajo, y se presentan algunas líneas de trabajo futuro para profundizar el aprendizaje y mejorar el desarrollo realizado.

Al finalizar este trabajo, se agregan cuatro apéndices en donde se encuentran los códigos y esquemáticos elaborados durante la realización del este trabajo final.

1.6. Sumario del capítulo

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

Se propuso utilizar una interfaz comercial como intermediario los dos dispositivos y se brindó una justificación del empleo del protocolo USB 2.0 de alta velocidad para una comunicación óptima que satisface los requerimientos.

Además, se repasaron algunos conceptos importantes inherentes a la versión 2.0 de la norma USB.

Se presentaron también los objetivos formales del trabajo y la estructura del presente informe.

Capítulo 2

Interfaz USB

Para implementar un sistema de comunicación entre la PC y el FPGA a través del protocolo USB, se propuso utilizar un dispositivo intermedio que cumpla el rol de interfaz. El dispositivo utilizado para este propósito es el controlador EZ-USB FX2LP, diseñado por la empresa Cypress Semiconductor, el cual viene incorporado en el kit de desarrollo CY3684, fabricado por la misma empresa.

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. La función de este módulo es comunicarse con una 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 escribir los datos arribados desde la PC en el FPGA con un protocolo más simple con el objetivo de utilizar menos recursos programables de este último dispositivo. 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). Las empresas FTDI y Cypress Semiconductor 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.

Por su parte, la línea de circuitos integrados FX2/FX2LP de Cypress Semiconductor ofrece 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 supone 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), posibilitando la carga no volátil de la configuración del controlador; una memoria Flash con una capacidad de 64 kB que es utilizada durante la ejecución del firmware, un puerto USB y dos UART con zócalos DE-9. Adicionalmente, cuenta con 6 puertos de 20 pines, compatibles con Analizadores Lógicos estándar y un puerto con 40 pines, compatible con el protocolo ATA, que permiten comunicarse con el controlador.

2.2. El controlador FX2LP EZ-USB y su configuración

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 que se alimentan a baterías y poseen una autonomía energética 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 configurables para datos. Incorpora también una versión del microcontrolador (μ C) Intel MCS-51, más conocido como el μ 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 los 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 I²C y una memoria FIFO (*FirstInFirstOut*; Primero Entrado, Primero Salido).



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

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, μ C);

Bajo el criterio de este autor, el componente de mayor trascendencia en el funcionamiento del controlador FX2LP es el μ 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 μ C. El firmware es escrito en lenguaje C para microcontroladores.

La estructura de la interfaz implementada en este trabajo utiliza la memoria FIFO en modo esclavo, es decir, la memoria 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 μ C es una modificación del 8051 desarrollado por Intel, para que sea más veloz en sus tiempos de ejecución y mejore el desempeño del μ 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 μ 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 μ Vision. Los archivos resultantes de la configuración realizada se pueden encontrar en el Apéndice A.

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 fue elaborado 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 todos 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().

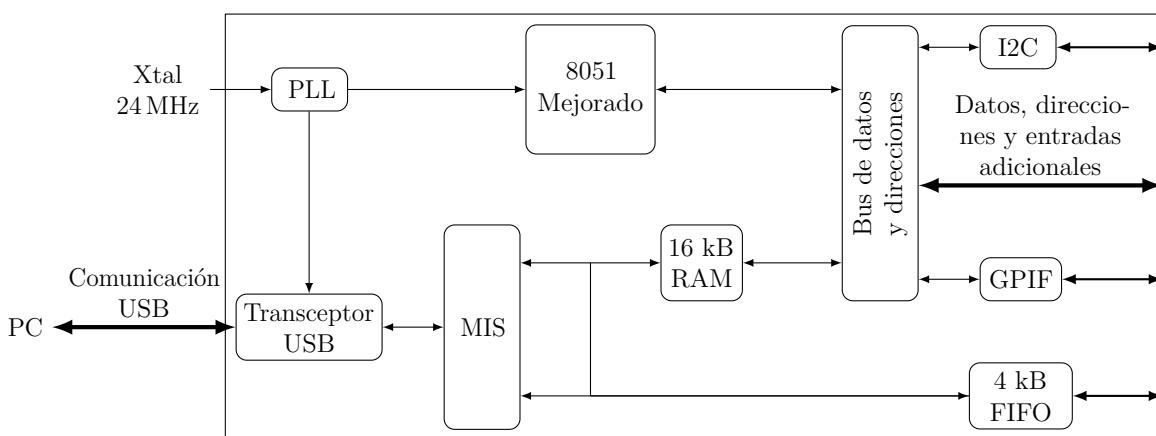


Figura 2.2: Arquitectura FX2LP [30]

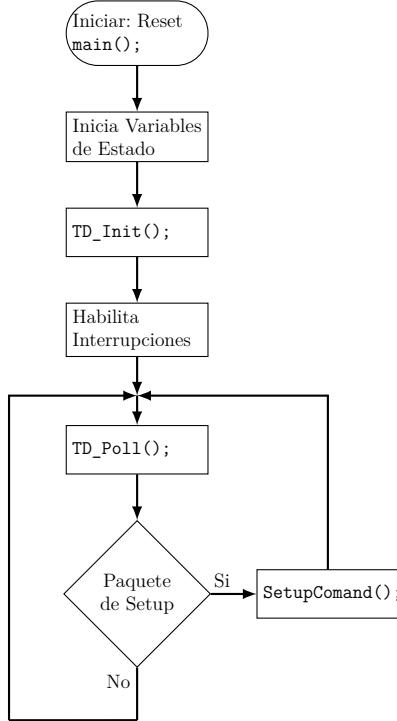


Figura 2.3: Diagrama en bloques del firmware que ejecuta el μ C de la interfaz

2.2.2. Frecuencia de trabajo del sistema

La configuración principal del sistema se realiza a través de la función `TD_Init()`. El primer módulo que configura es el PLL (*Phase-Locked Loop*). Un PLL es un lazo de servocontrol cuyo parámetro controlado es la fase de una réplica, generada en forma local, de una señal de entrada [31]. 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 μ 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 μ 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 se comunican 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 EP¹. 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 μ 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 EP2², 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 isócronas.

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.

¹EP 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.

²EPx será un extremo con dirección x, siendo x un número entero. En este caso, EP con dirección 2.

Interfaz hacia los periféricos

Cypress provee varias interfaces para comunicar el controlador hacia los periféricos. I²C 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 bien, a través de la interfaz GPIO, puede ser comandada por el μC 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 Token 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 μ 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.

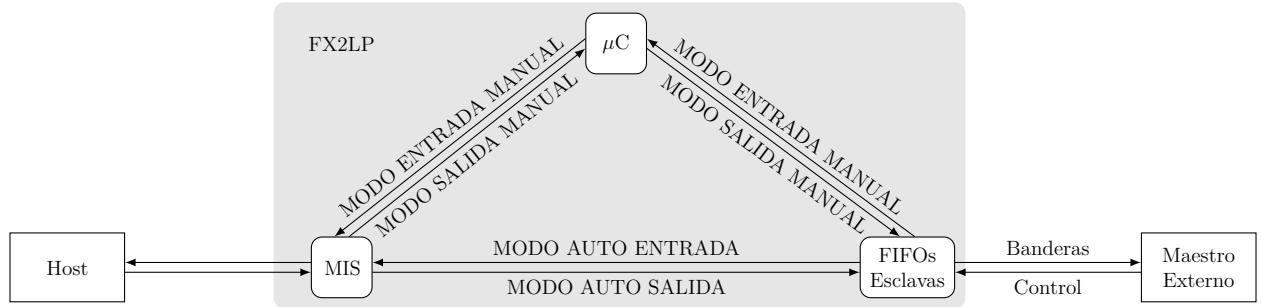


Figura 2.4: Modos de conexión de la memoria FIFO, el micrcontrolador y el MIS

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 μ 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.

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 asegurar su correcto funcionamiento.

rarse 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
EP8FIFOCFG = 0x11;
SYNCDELAY;
EP2FIFOCFG = 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 [30].

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 de un archivo objeto provisto en el framework de Cypress Semiconductor, denominado *usbjmpbt.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 d<á> informaci<ón> sobre otras velocidades de operaci<ón>. Este descriptor es necesario debido a que el sistema implementado cumple con la versi<ón> 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            ; ; Versi<ón> 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               ; ; Tama<ñ>o m<á>ximo de paquetes
10    db 1                ; ; N<ú>mero de comunicaciones
11    db 0                ; ; Reservado
```

La norma USB especifica que la informaci<ón> 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 condici<ón>, se recurre a la sentencia `org (($/2)+1)*2`, la que se puede leer en la primera l<í>neas del c<ódigo> mostrado anteriormente. La directiva `org` le ordena al ensamblador la direcci<ón> espec<íf>ica de memoria en la que debe colocar las instrucciones precedentes. El s<ímbolo> (\$) representa al puntero que contiene el valor de memoria de la <ú>ltima instrucci<ón> ensamblada. Considerando que las direcciones de memoria son enteros y que las operaciones de ensamblador truncan decimales (no los redondean), la ecuaci<ón> indicada entrega, como resultado, el entero par siguiente a la direcci<ón> actual.

Luego de esta directiva, se especifica el descriptor de tipo DEVICE_QUALIFIER asegurando que se encontrará en una direcci<ón> par de memoria.

Seguidamente, la norma 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 m<á>xima velocidad (*Full-Speed*), que se especifica en el descriptor DEVICE_QUALIFIER. Se expone a continuaci<ón> el descriptor CONFIGURATION de alta velocidad, unido al descriptor INTERFACE (l<í>nea 16 en adelante) y los descriptores ENDPOINT (a partir de la l<í>nea 27), que determinan en forma completa la configuraci<ón> 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                ; ; N<ú>mero de interfaces
10    db 1                ; ; Indice de configuraci<ón>
11    db 0                ; ; String de configuraci<ón>
12    db 80H              ; ; Atributos (b7->1, b6 - selfpw, 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                  ; ; 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 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 isócronas.

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 - selfpw, 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 mensajes en formato de cadena de caracteres, para lectura 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 de este trabajo.

```
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 provee 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 un desarrollo cuya descarga es libre y permite configurar y recibir mensajes a través de diferentes puertos.

Como testigo del funcionamiento de las señales *FLAG*, se asociaron sus valores a los LED de propósito general, de forma tal que se pudiese 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 UART0 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 frecuencia 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 % [30] 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 módulos, procede a la desconexión del controlador con el puerto USB, a través de una rutina que Cypress denomina ReNumertion [30]. Durante el proceso de reconexión ocurre algún efecto, aún no identificado, que hace que el programa se detenga. Este desperfecto se solucionó enviando un

mensaje de comunicación después activar la reconexión. Esto demostró ser lo suficientemente robusto ya que eliminando el resto de envíos de datos para monitoreo, no apareció nuevamente la detención del programa, mas 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 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 [34]. 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 lazo infinito del controlador FX2LP, se colocó la el siguiente código:

```
void TD_Poll( void )  
{  
    BYTE dum;  
  
    if (EP8FIFOFLGS & bmBIT1) //ep8 fifo vacia  
    {  
        dum = D4ON;  
    }  
    else  
    {  
        dum = D4OFF;  
    }  
  
    if (EP8FIFOFLGS & bmBIT0) //ep8 fifo llena  
    {
```

```
        dum = D3ON;
    }
else
{
    dum = D3OFF;
}
if (EP2FIFOFLGS & bmBIT1) //ep2 fifo vacia
{
    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 encienden o apagan las luces respectivas. Los LED multipropósito se encuentran numerados en la placa de desarrollo desde el LED 2 hasta el LED 5. En la programación de su funcionamiento, el LED 2 se enciende cuando el EP2 se encuentra vacío, el LED 3 encendido indica que el EP8 está lleno y el LED 4 señala que el EP8 no contiene datos. Por su parte el LED 5 parpadea cuando el controlador establece una conexión de alta velocidad con el Host y queda encendido siempre cuando la conexión es a toda velocidad. 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 Semiconductor 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 estado del LED 4, que indica que el EP8 se encuentra vacío, se infirió que los datos arribaban al controlador. A continuación, se enviaron más datos hasta lograr que el LED 3 se encendiese, señalando que el EP8 llegó al límite de su capacidad. 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 [30].

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 registra la cantidad de datos 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 μ C para esta tarea.

Si se desea agregar información en un paquete de datos, esta debe ser escrita en la memoria FIFO. Para ello, el armado automático de paquetes debe estar desactivado. Luego, al escribir datos en la memoria, el μ C debe informar al MIS el agregado de datos, colocando el valor adecuado en el contador correspondiente. Se debe considerar que cuando arriban datos, el controlador también debe emparejar los contadores.

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 en 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 TD_Init().

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.

Finalmente se explicaron problemas identificados durante las pruebas y la depuración del programa de configuración, su solución y las herramientas utilizadas para hallarlas.

Capítulo 3

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

En el Capítulo anterior, se logró intercambiar datos entre una PC y el controlador FX2LP a través del protocolo USB. Este enlace constituye una primer etapa del desarrollo realizado. En una segunda etapa, se debe lograr la comunicación de datos entre un FPGA y el controlador FX2LP de Cypress. Así, los datos estarían en condiciones de fluir entre el Host y el FPGA, a través de la interfaz USB, utilizando las memorias FIFO del controlador FX2LP.

Para diseñar un sistema de comunicación entre el FPGA y la interfaz USB, es necesario identificar cuales son los protocolos a través de los cuales se leen y escriben datos en el controlador FX2LP, como así también las señales y los puertos con que debe interactuar el FPGA. Una vez identificado el procedimiento, se pudo desarrollar el sistema de comunicación, que fue implementado en una Máquina de Estados Finitos (MEF), a través de la cual, se leen las señales que provienen del controlador FX2LP y se generan las señales necesarias para comandar su memoria FIFO.

En este Capítulo, se justifica la elección del FPGA y la placa de desarrollo utilizados para la implementación del sistema de comunicación. También se detallan las señales que intervienen en el funcionamiento de la interfaz USB y los protocolos de lectura y escritura de modo asíncrono. A continuación se desarrolla el diseño de la MEF y su descripción en lenguaje VHDL, para su posterior síntesis en el FPGA. Luego, se exponen la verificación funcional del sistema descripto. 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 del FPGA

Para trasmisitir información en un sistema de comunicación, los componentes que intervienen siguen un protocolo determinado. Así, no solo se facilita el envío y la recepción del mensaje, sino también que determina a cada dispositivo los procedimientos que debe efectuar. 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 cuál 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, tiene como dispositivo central un FPGA Spartan 3. Además, posee una gran cantidad de periféricos, entre los que se destacan su pantalla LCD, los 18 MB de memoria flash sumados a 64 MB de SDRAM y transceptores varios, tales como Ethernet, PS/2 para teclados y JTAG para programación y depuración.

Por su parte, la placa Nexys 3, tiene como núcleo un FPGA Spartan 6. Este dispositivo brinda mayor cantidad de bloques lógicos programables que la versión Spartan 3, debido a que posee transistores más pequeños gracias a un proceso de fabricación CMOS más moderno. 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 de máxima velocidad (12 Mbit s^{-1}).

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 posee un FPGA superior a la placa Spartan 3E Starter, brindando la posibilidad de elaborar sistemas más complejos y veloces. A su vez, es más económica que la placa Nexys 3, que posee un FPGA de similares características. En otras palabras la Mojo se selecciona por su bajo costo, versatilidad y por que está dotada por un Spartan 6 de Xilinx, que es un FPGA con una buena relación entre recursos, rendimiento, velocidad y precio. La menor disponibilidad de periféricos de la placa Mojo v3 con respecto a las otras placas de desarrollo, no es un inconveniente porque el kit CY3684 de Cypress incorpora los dispositivos necesarios para los fines de este trabajo.

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*¹. El usuario también puede diseñar sus propios *shields* o conectar las entradas y salidas de otros dispositivo mediante cables, conformando así una placa de desarrollo a la medida de las necesidades de cada proyecto.

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 μC ATmega32U4 diseñado por la empresa Atmel con un programa de tipo bootloader, que se encarga de transferir

¹*Shield* es una palabra del habla inglesa que en español significa escudo o armadura. Su pronunciación suena como *shild*

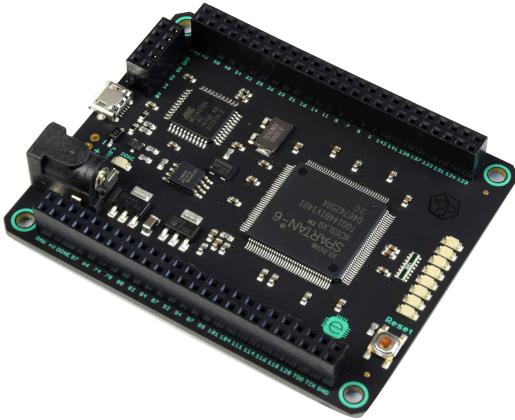


Figura 3.1: Placa de prototipado rápido MOJO v3, diseñada por Embedded Micro

la 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 μ C. 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 μ C ATmega32U4, luego de que el FPGA es programado.

Es importante aclarar que si bien el sistema posee alguna forma de comunicación USB utilizando el μ C ATmega32U4 como interfaz, este enlace posee un menor ancho de banda que el sistema desarrollado en el presente trabajo. Esto se debe a que la línea de controladores ATmega incorpora puertos USB 2.0 de máxima velocidad (12 Mbit s^{-1}) [35]. Además, la comunicación entre ambos chips se realiza vía SPI (*Serial Peripheral Interface*, o en español Interfaz Serie de Periféricos), con una tasa de transferencia máxima de 8 Mbit s^{-1} [35], ya sea para transmitir por el puerto USB como para comunicar las entradas analógicas digitalizadas, 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 de alta velocidad, de hasta 480 Mbit s^{-1} .

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. Se denomina protocolo a 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, para el diseño de una MEF es importante conocer los pasos que debe seguir para cumplir con su propósito.

A continuación se presentará la secuencia de señales 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 son la base de diseño para el desarrollo de la MEF.

3.2.1. Señales de comunicación el FPGA y el controlador FX2LP

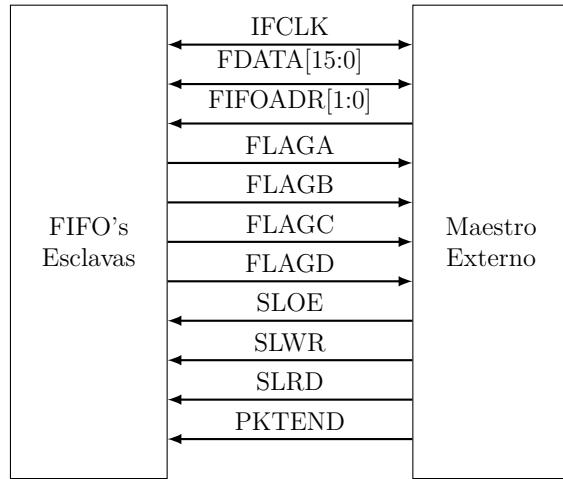


Figura 3.2: Señales de la interfaz entre las FIFO's y un maestro externo

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 Sparta 6 de la placa Mojo v3. Las señales de control son:

- IFCLK: señal de reloj. No es necesaria en caso de utilizar la memoria FIFO en modo asíncrono. El controlador FX2LP puede ser configurado para que esta señal sea provista por el mismo controlador o por el dispositivo de control.
- 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 que accede al bus de datos.
- FLAGx: Los cuatro puertos de flag indican el estado de las memorias y son configurables. La x es una variable que corresponde al nombre asignado al flag, sea A, B, C o D.
- SLOE, SLWR, SLRD: son las señales de control. A través de ellas el maestro ordena la lectura/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 μ C 8051 que posee el controlador 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.

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

Tabla 3.1: Direcciones de selección de memoria activa

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 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 implementada en este trabajo, como se detalla 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. 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* es '0', el espacio de memoria destinado al EP2 se encuentra lleno. Si en cambio, el valor es '1', la memoria aún posee espacio para el almacenamiento de datos.

Lectura de datos desde la memoria FIFO

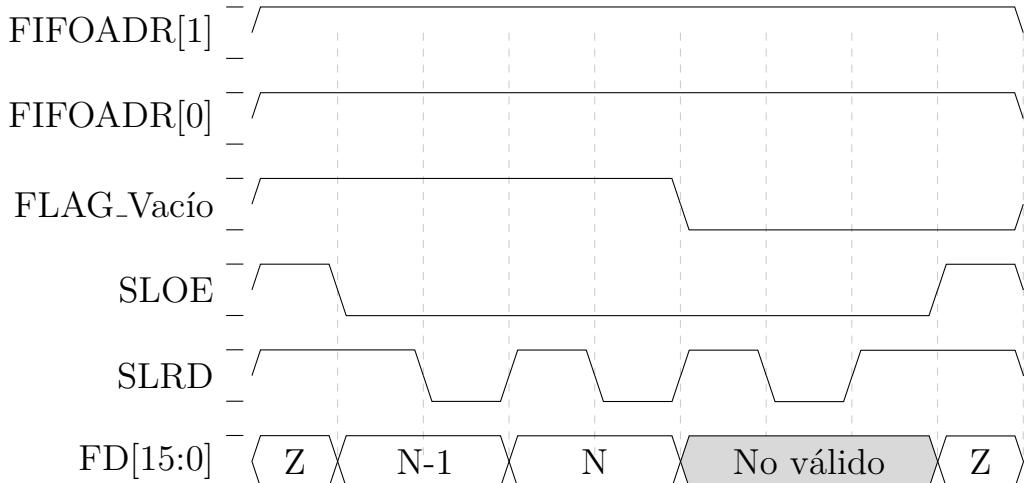


Figura 3.3: Diagrama temporal de la lectura de datos desde la memoria FIFO por un FPGA

Para efectuar operaciones de lectura en régimen asíncrono, como se muestra en la Figura 3.3, en primer lugar, el FPGA debe colocar la dirección de la memoria sobre la que efectúa la operación en los puertos *FIFOADR[1:0]* ("11" en este trabajo, 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 que fue activada por la señal *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

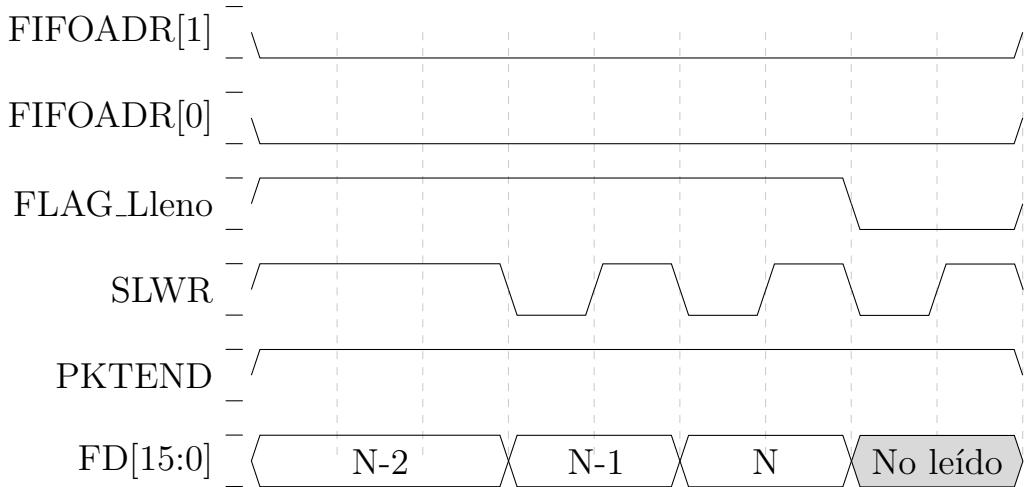


Figura 3.4: Diagrama temporal de la escritura de datos en la memoria FIFO desde un FPGA

el flanko ascendente de la señal *SLRD*, la memoria FIFO aumenta un 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 y su funcionamiento se encuentran detallados en el diagrama temporal de la Figura 3.4. En primer lugar el FPGA debe activar la memoria FIFO a través de *FIFOADR[1:0]*. El sistema desarrollado en este trabajo utiliza la dirección "00", correspondiente al EP2. Luego, se coloca en el bus de datos (conectado a los puertos *FD[15:0]*), el dato a escribir. En esta operación es importante que *SLOE* esté inactivo, de modo tal que el bus *FD[15:0]* del controlador FX2LP se encuentre en modo de alta impedancia y no interfiera con la escritura. Una vez colocado el dato en el bus, se debe activar la señal *SLWR*. En el flanko 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 este motivo, debe ser capaz de efectuar operaciones de lectura y escritura de datos en ambas direcciones. En

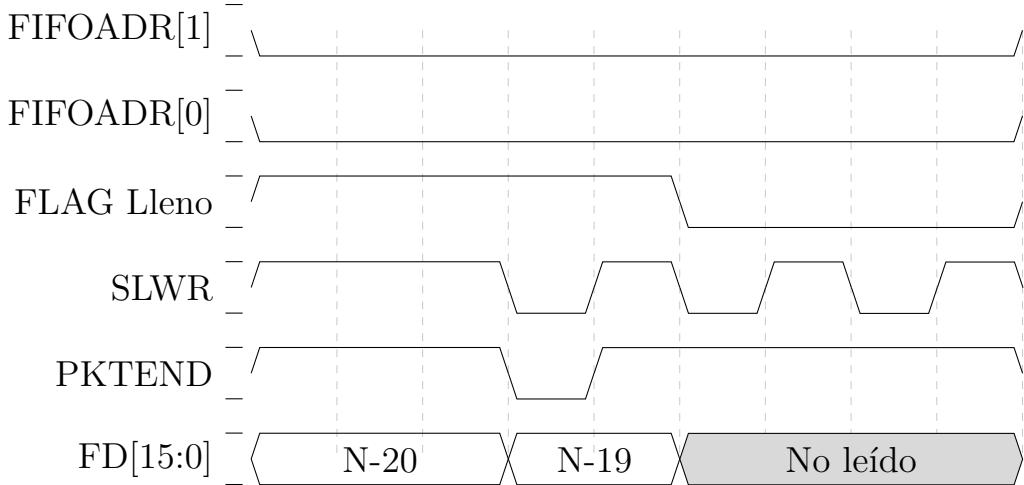


Figura 3.5: Diagrama temporal del funcionamiento del finalizado manual de mensajes

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, denominadas *Dato_Recibido* y *Dato_a_enviar* respectivamente, entrada de reloj y reset que será suministrada por el FPGA, como así también de una señal que controla el envío de datos, llamada *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.

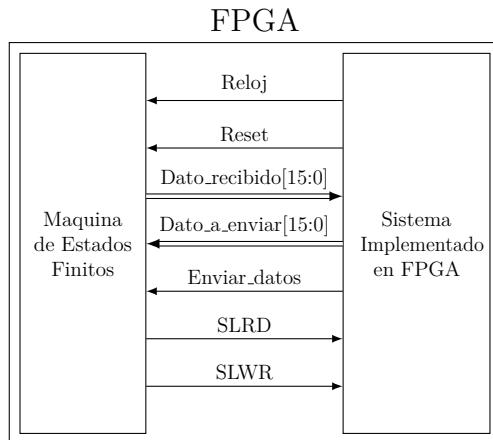


Figura 3.6: Diagrama de las señales que interconectan los módulos dentro del FPGA

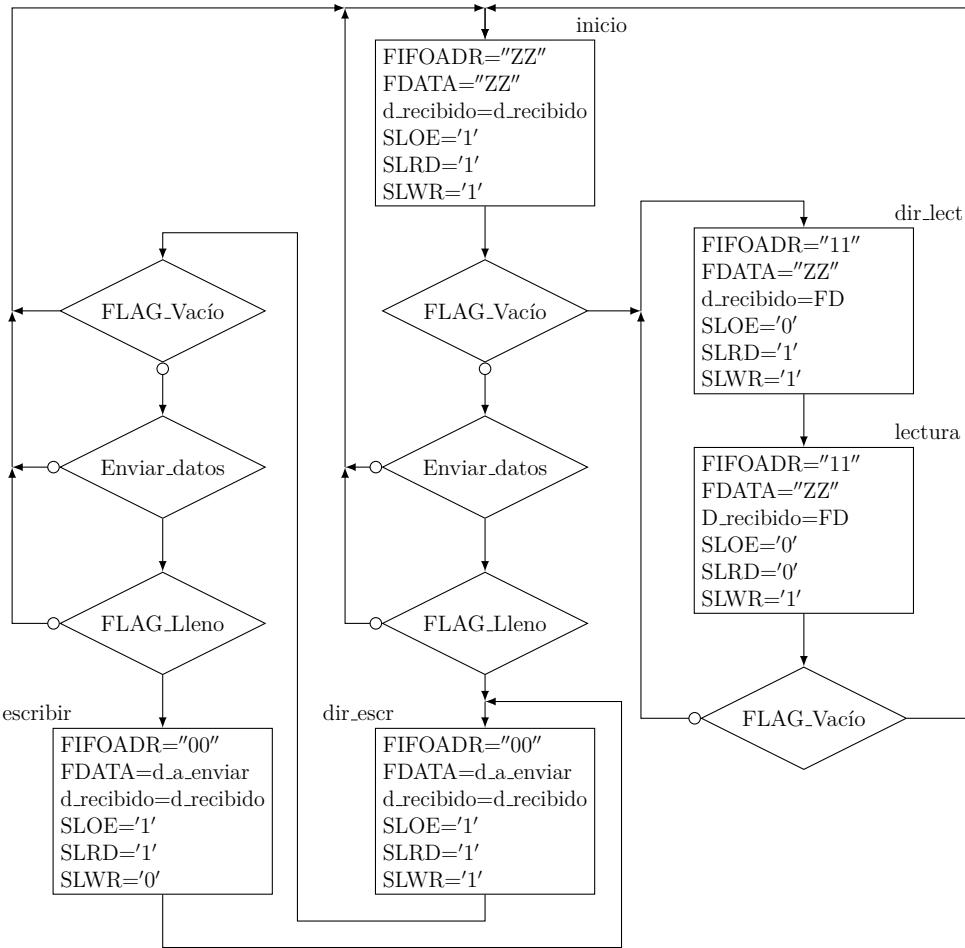


Figura 3.7: Diagrama de flujo de la máquina de estados desarrollada

3.3. Diseño de la Máquina de Estados Finitos

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 la operaciones de la MEF implementada en el presente trabajo, se recurrió a la confección del diagrama de flujo para un máquina de estados algorítmica.

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

trasmisir los datos que adquieran del medio en que se encuentren. Se espera entonces, que la información que contienen los datos de configuración posea mayor importancia, ya que podría tener ordenes 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 maquina de estados algorítmica que se observa en el diagrama de flujo de la Figura 3.7. En un estado inicial, todos 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 3.7 como *d_recibido*, retiene su propio valor.

Cuando la señal *FLAG_Vacio* se activa, se procede a la operación de lectura. Si, en cambio, *FLAG_Vacio* esta inactivo implica que no hay datos que leer. Entonces, se debe conocer si el sistema implementado en el FPGA tiene activa la señal *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. Así, se procede con 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 3.7 por *d_a_enviar*. Si las variables de 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 3.7, se procedió a describir el comportamiento del sistema en VHDL. La Figura 3.8 muestra la estructura conceptual de una MEF. Se puede apreciar que una MEF se compone de tres partes: la función del próximo estado, el estado actual y la función de salida. Estas partes pueden ser descriptas en VHDL agrupadas en un mismo proceso, o bien cada una de ellas en un proceso diferente. 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 declaraciones concurrentes.

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

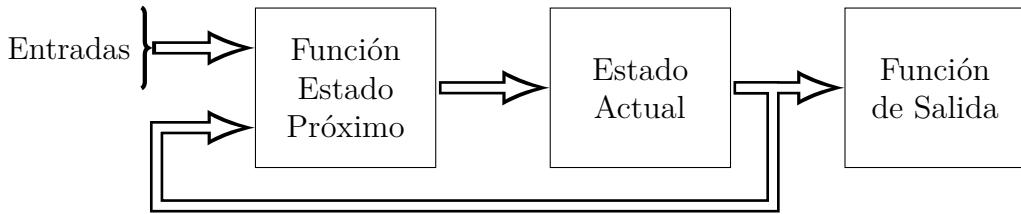


Figura 3.8: Estructura de una máquina de estados finitos

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 B.

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 3.7, 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
    (
    
```

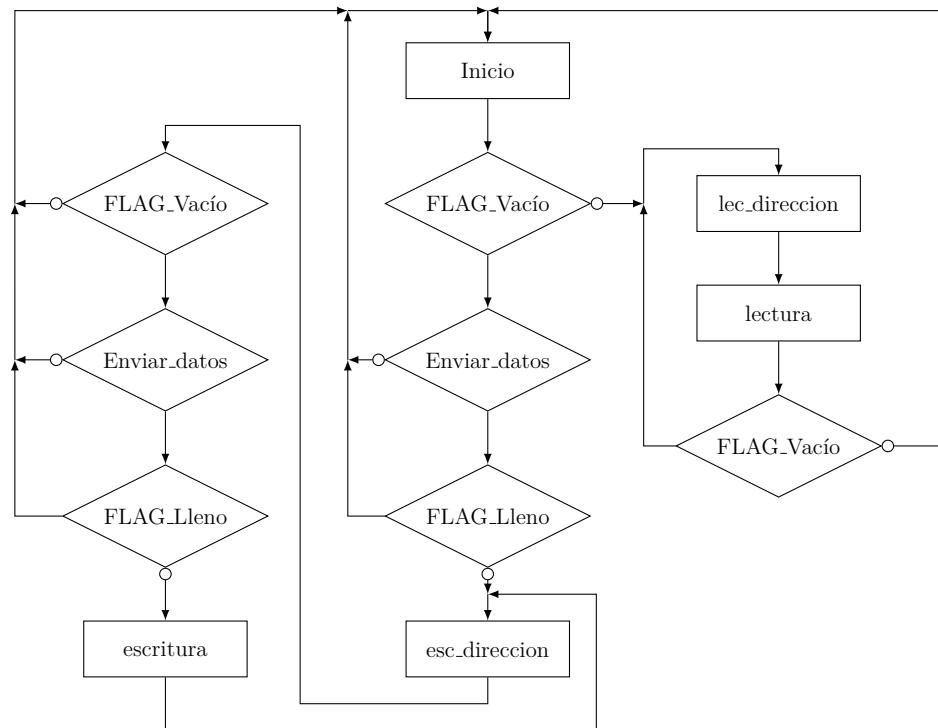


Figura 3.9: Diagrama de simplificado del flujo de la MEF

```

        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 que consta de un reloj que 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 [36]. 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, lo que 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 corroborar que lo descripto en lenguaje de alto nivel (VHDL en este trabajo), se corresponde con el funcionamiento que se esperaba antes de realizar la descripción. Para efectuar la 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 (el controlador FX2LP), como hacia el interior (los módulos implementados en el FPGA). También se simuló la entrada de datos desde el FPGA, generando la rutina de salida de datos hacia la interfaz.

En la implementación de la arquitectura se declaró e instanció el componente bajo prueba, es decir, la MEF y las señales necesarias para su correcta instancia. 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 implementó el proceso que contiene los estímulos, emulando el funcionamiento de la interfaz. En un primer momento, las memorias FIFO se encuentran descargadas, es decir, las señales que indican “memoria FIFO vacía” se encuentran activos (*FLAGB* y *FALGD* en bajo), en tanto las 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.

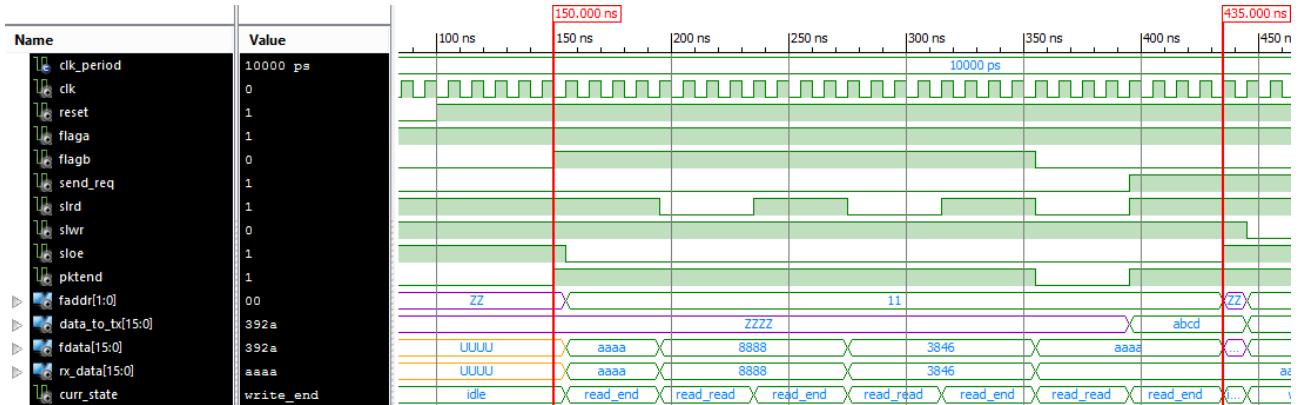


Figura 3.10: Diagrama temporal de la operación de lectura entregado por el simulador

Estas condiciones se mantuvieron durante 5 ciclos de reloj después de desactivada la señal *reset*. Así, se corrobora que ante la ausencia de estímulos, el sistema se mantiene invariable.

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

Una vez realizada la operación de escritura se procedió a describir la operación de escritura. Para esto, se colocó un dato en el bus de envío de datos, *DATA_TO_TX* y se activó la señal de envío de datos *SEND_REQ*. Una vez activada la señal *SLWR*, se cambió el dato a enviar.

Se realizan también dos pruebas extra, en el desempeño de la máquina de estados desarrollada. En primer lugar, se observó 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 activa.

Además, se observó 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*.

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 *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, 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. Tal como indica 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 alcanza el estado inicial (*IDLE*, como se ve en la Figura 3.11), procede

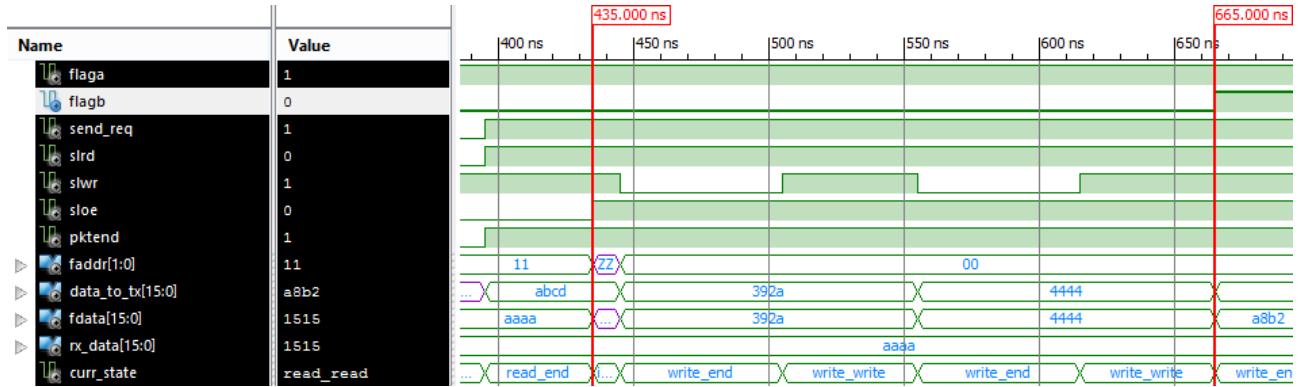


Figura 3.11: Diagrama temporal de la operación de escritura entregado por el simulador

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 encuentra en el bus de envío de datos *DATA_TO_TX*.

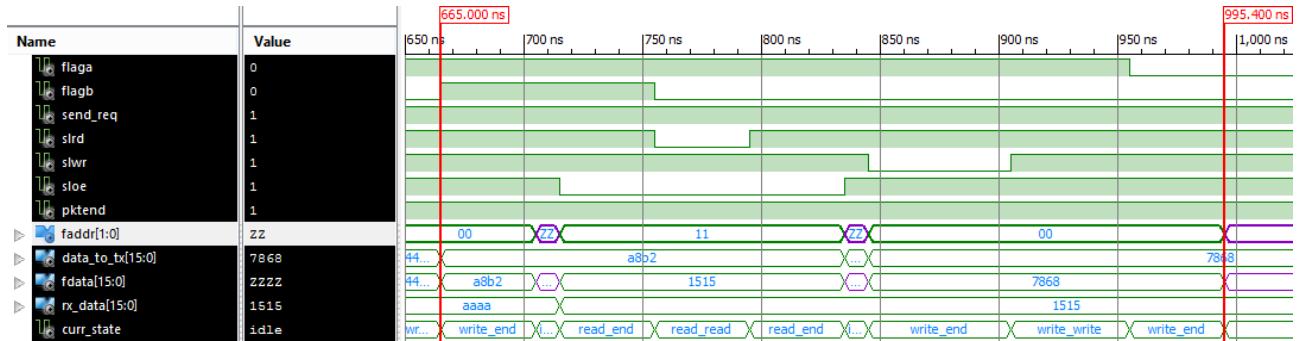


Figura 3.12: Diagrama temporal que muestra las interrupciones de la operación de escritura

La Figura 3.12 muestra cómo el sistema finaliza la operación de escritura cuando 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 estuvo en condiciones de grabar dicha síntesis en el FPGA y se puede proceder a realizar pruebas de funcionamiento más avanzadas. El código completo de la simulación, se pueden leer en el Apéndice B.

3.6. Placa de Interconexión

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

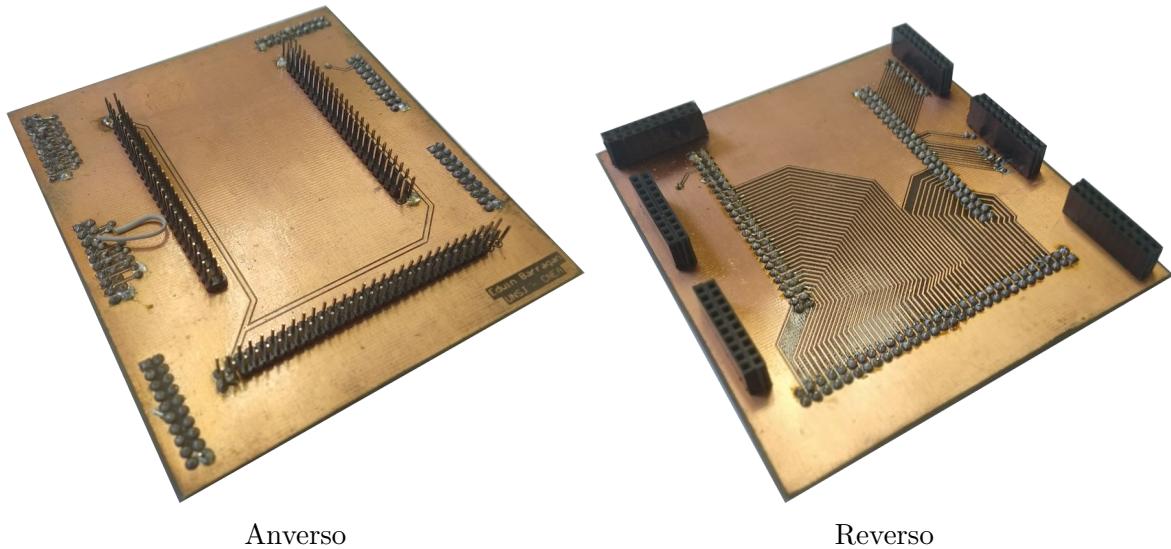


Figura 3.13: Imagen de la placa de interconexión

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 se puede consultar en el Apéndice D.

El circuito impreso desarrollado determina en forma definitiva 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.

3.7. Sumario del capítulo

Durante el presente capítulo se justificó la selección de la placa Mojo v3, la cual posee incorporado un FPGA Spartan 6 desarrollado por Xilinx para la realización de este trabajo.

Se expuso también cuáles son las señales de control que intervienen y cómo es su funcionamiento durante las operaciones de lectura y escritura externa en las memorias FIFO del controlador FX2LP.

Con base en el mecanismo que siguen las señales de control en las operaciones mencionadas se explicó el diseño de una MEF que intercambia datos entre un sistema implementado en FPGA y el controlador FX2LP. Se realizó una descripción física de la MEF, utilizando VHDL como lenguaje de descripción, para su posterior síntesis en el FPGA Spartan 6. 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 6 y como éste circuito impreso determina la correlación entre los puertos de cada uno de los circuitos integrados.

FX2LP	Spartan 6
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.

Capítulo 4

Pruebas de funcionamiento y desempeño del sistema desarrollado

El sistema de comunicación desarrollado fue sometido a una prueba de desempeño a fin de verificar que sea capaz de enviar y recibir datos en forma efectiva y que, a su vez, cumpla con el objetivo de establecer un enlace capaz de enviar datos a una tasa de 480 Mbit s^{-1} . La prueba realizada consistió en el envío de un conjunto de datos desde una PC, que fue almacenado en el FPGA para luego ser retransmitidos hacia la PC y fue repetida en forma automática durante 24 horas. Para efectuar dicha prueba, se elaboró un sistema en FPGA, dotado con la capacidad de memoria necesaria para recibir los datos que luego enviará, la MEF presentada en el Capítulo 3 que provee lo necesario para la comunicación con el controlador FX2LP y una señal de reloj que permite sincronizar la MEF y la memoria implementada.

Además, se realizó un programa de computadora, escrito en lenguaje C++, que permite enviar una trama de datos aleatorios hacia el dispositivo desarrollado en el presente trabajo, a través del protocolo USB. A lo largo de este Capítulo, se describirán los componentes utilizados para realizar las pruebas de desempeño del sistema de comunicación desarrollado y se expondrán los resultados obtenidos.

4.1. Sistema de pruebas implementado en FPGA

Para realizar las pruebas de desempeño, se elaboró un sistema tipo Eco, es decir, un dispositivo que ante la recepción de un mensaje, responde con la retransmisión del mismo hacia el emisor original. En este caso, el emisor es la PC y el dispositivo que reenvía el mensaje es el FPGA. Para poder retransmitir el mensaje, se implementó el sistema presentado en el esquema de la Figura 4.1. En ella se observa que el FPGA Spartan 6 fue dotado de un bloque con la MEF presentada en el Capítulo anterior. A dicho bloque, fue conectada una memoria FIFO. Así, los datos que son enviados desde la PC, se almacenan en la memoria FIFO a medida que arriban, atravesando la interfaz. Luego, estos datos son reenviados desde la misma memoria FIFO, emprendiendo el circuito hacia la PC.

La señal de reloj del FPGA Spartan 6 es provista por una de las salidas de un módulo PLL que el FPGA tiene incorporado. A través de este módulo, se disminuye la frecuencia de 50 MHz que entrega el oscilador de la placa de desarrollo Mojo v3 a los 48 MHz necesarios para el conectarse con el controlador FX2LP. La señal de reloj emitida por la salida del PLL sincroniza

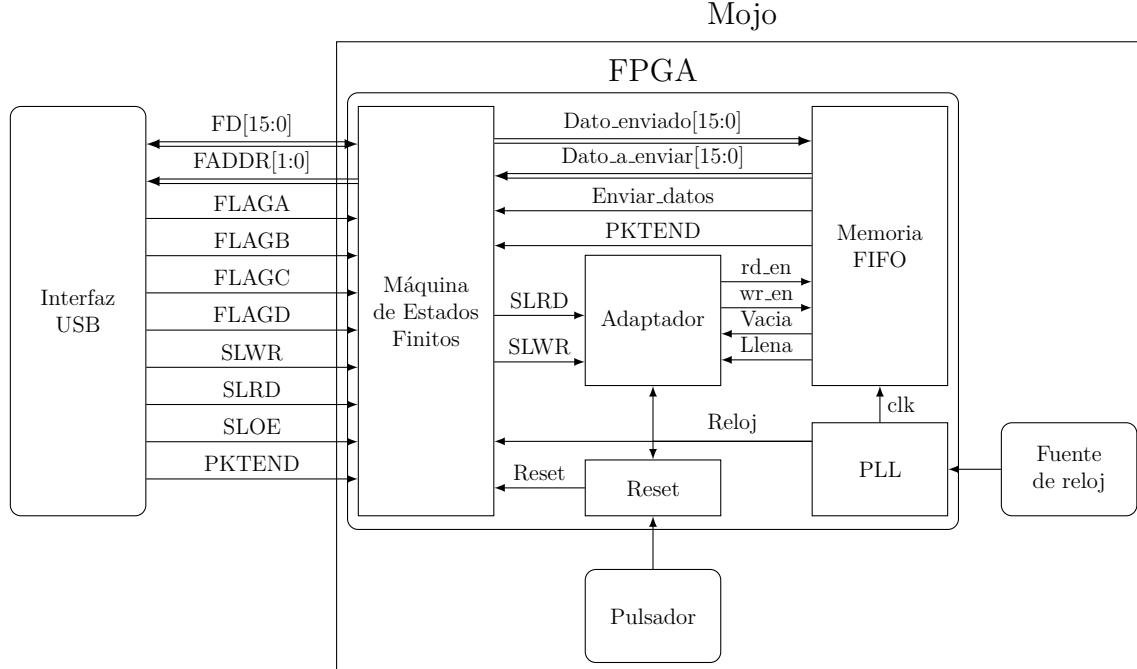


Figura 4.1: Diagrama en bloques del sistema implementado en el FPGA

a todo el sistema implementado en el FPGA.

Dos de los módulos implementados en el sistema de pruebas no fueron desarrollados en el presente trabajo, sino que se generaron a través de soluciones que provee la empresa Xilinx Inc. para el desarrollo de sistemas. Tanto el PLL como la memoria FIFO incorporadas al sistema implementado dentro del FPGA, fueron generadas por la herramienta *Core Generator*, provista por la empresa Xilinx Inc. en el software de programación ISE. Las señales SLRD y SLWR debieron ser adaptadas para compatibilizar el funcionamiento de la memoria generada por la herramienta *Core Generator*.

Por su parte, la placa Mojo v3 incorpora un pulsador, que es usado en este trabajo como señal de reset asíncrona. El sistema tiene un reset síncrono realizado a través de un contador, que es utilizado para establecer los valores iniciales del circuito.

El código completo de la descripción realizada en lenguaje VHDL del sistema de pruebas, en donde se declaran e instancian los componentes que se describen en esta sección junto con la MEF desarrollada y las señales necesarias, se puede apreciar en el Apéndice B.

Generación de Señal de Reloj

Las especificaciones de la interfaz indican que la máxima frecuencia de funcionamiento del reloj debe ser de 48 MHz [36]. A su vez, la placa de desarrollo Mojo v3 posee un oscilador que provee al FPGA una señal de 50 MHz. Para lograr la señal de reloj con la frecuencia 48 MHz, 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 [37]. A través de esta herramienta, se indicó que la señal de entrada es de 50 MHz. Se aprovechó la característica del PLL integrado en el FPGA Spartan 6 de configurar hasta 4 salidas con frecuencias diferentes y se seleccionaron

señales de salida con 50, 48, 40 y 35 MHz, para tener una forma de reducir la frecuencia si se presentaban problemas de sincronismo durante la prueba del sistema. Luego, *Core Generator* entregó un código de VHDL en donde se declara una entidad para que pueda ser utilizada como componente y se instancia el PLL. De esta forma, la entidad de dicho código pudo ser declarada e instanciada en la descripción del sistema de pruebas.

Implementación de la memoria FIFO en el FPGA

La memoria FIFO sintetizada en el FPGA también se implementó a través de la herramienta *Core Generator* de Xilinx. Dicha herramienta permite configurar una memoria de 512 o 1024 B. Se seleccionó el valor de 1024 B como capacidad para la memoria, con un ancho de bus de 16 bits. Además posee señal de reconocimiento de escritura, es decir que indica cuando el dato fue leído en forma efectiva. La configuración de la memoria permite utilizar señales diferentes para cada puerto, o bien, una sola señal de reloj que comande ambos puertos. En este trabajo se utilizó una sola señal de reloj tanto para el puerto de entrada como el de salida. Además, la memoria cuenta con puertos de entrada y salida independientes. Con la configuración mencionada, *Core Generator* entregó una plantilla para utilizar la memoria generada. Dicha plantilla se utilizó para declarar e instanciar el componente en el sistema implementado.

La memoria FIFO generada por la herramienta *Core Generator* no es directamente compatible con la MEF diseñada, debido a que la MEF utiliza los flancos como ordenes de lectura y escritura, pero la memoria FIFO almacena los datos mientras las habilitaciones de lectura escritura estén activas. Por este motivo, se realizó un adaptador para adecuar las ordenes de lectura/escritura emitidas por la MEF a las necesarias para comandar la memoria FIFO. El adaptador está compuesto por dos máquinas de estados que habilitan la escritura y la lectura con los flancos descendentes de las señales *SLRD* y *SLWR*, en función del estado (vacío o lleno) de la memoria.

4.1.1. Verificación Funcional del sistema de pruebas

Debido a la incorporación de los módulos necesarios para realizar el sistema de pruebas y la adaptación de las señales para el correcto funcionamiento de los mismos, se realizó una nueva verificación funcional del sistema implementado en FPGA. Para ello, se generó un código de descripción en VHDL, en donde se simula la transmisión de datos del sistema. El código completo realizado para la verificación se puede leer en el Apéndice B.

La simulación de los datos que ingresan al FPGA se realizó mediante un contador, el cual es incrementado con el flanco negativo de la señal *SLRD*. También se coloca en bajo la señal *FLAGB*, indicando que la memoria del controlador FX2LP contiene datos. De esta forma, se simula la operación de lectura sobre la interfaz USB por lo que requiere efectuar sobre ella una operación de lectura.

Una vez que el contador alcanza un nivel determinado, la señal de entrada *FLAGB* se coloca en '0', indicando que la memoria de entrada de datos se encuentra vacía. Resta esperar que el sistema devuelva los datos almacenados a la interfaz USB a través de la operación de escritura.

La Figura 4.2 muestra el diagrama temporal entregado por el simulador en donde se detalla el inicio de la operación de lectura de la memoria de la interfaz USB. Cuando la señal *FLAGB* se encuentra en '1', el sistema activa la memoria FIFO del controlador FX2LP con dirección "00" y habilita también su salida a través de la señal *SLOE*. Luego, procede a leer la memoria colocando en bajo la señal *SLRD*. Una vez leído el dato, el contador generador de datos es

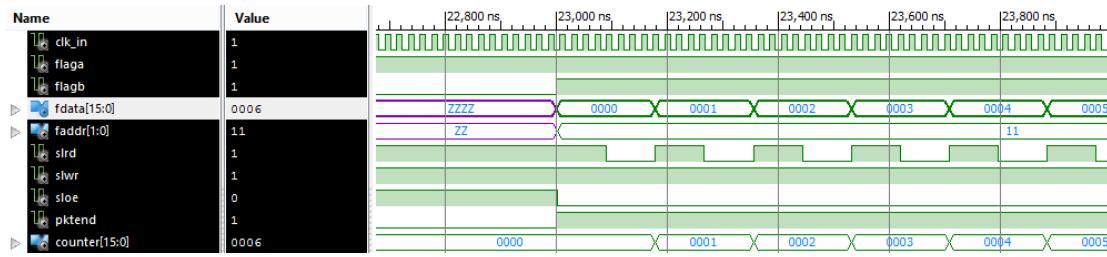


Figura 4.2: Simulación del inicio del ciclo de lectura

incrementado y el nuevo dato es leído con el siguiente flanco descendente de la señal *SLRD*. Esta secuencia es repetida hasta que la señal *FLAGB* sea '0'.

Cuando fueron leídos todos los datos contenidos en el controlador FX2LP, este colocara en bajo la señal *FLAGB*. Con esta señal, el FPGA comenzará el ciclo de escritura, reenviando los datos almacenados en su memoria. En la Figura 4.3 se puede observar que cuando la señal *FLAGB* baja a '0', se interrumpe el ciclo de lectura y el sistema vuelve al estado de reset, y a continuación, inicia el ciclo de escritura. Durante la operación de escritura, la MEF coloca el bus *FADDR[1:0]* en "11" por el FGPA, activando la memoria FIFO con esta dirección, es decir la receptora de los datos que emite el FPGA. Luego, activa y desactiva, en forma intermitente la señal *SLWR*, enviando los datos almacenados en la memoria del FPGA hacia la memoria de la interfaz.

Si por algún motivo la memoria de la interfaz USB se queda sin más espacio para el almacenamiento de los datos que provienen del FPGA, el controlador FX2LP coloca en bajo la señal *FLAGA*. En la Figura 4.4 se observa que luego del flanco negativo de la señal *FLAGA*, el FPGA interrumpe el envío. Cuando la señal *FLAGA* vuelve a '1', la memoria reanuda el envío de los datos almacenados.

Una vez que la memoria interna del FPGA no posee más datos para transmitir, la MEF coloca la señal *PKTEND* en '0', indicando que terminó el envío de datos. Esto se puede observar en la Figura 4.5. Se puede inferir que todos los datos de la memoria fueron transmitidos debido a que el último dato enviado a través del bus de datos es igual al último valor del contador utilizado como generador de datos de entrada al sistema.

Luego de realizada la simulación y verificado que el funcionamiento del sistema es el esperado, se está en condiciones de cargar la síntesis del circuito en el FPGA. Al elaborar la síntesis, el entorno de desarrollo ISE otorga un informe en donde consta, entre otras cosas, la cantidad de recursos del FPGA que utilizará la síntesis al ser cargada en el circuito integrado. En el caso de

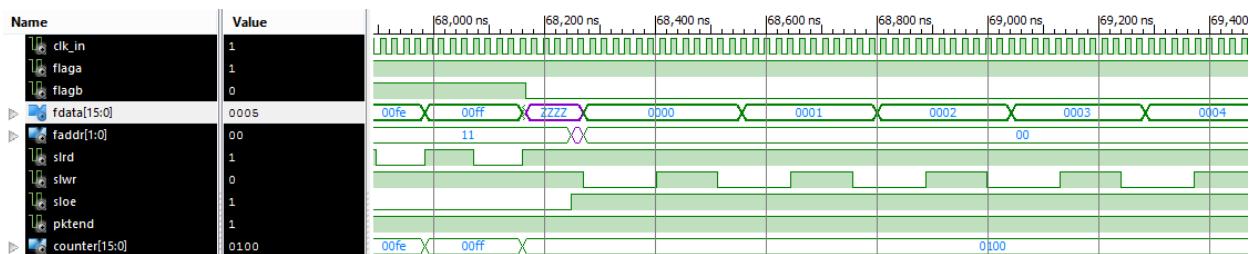


Figura 4.3: Diagrama temporal del final del ciclo de lectura e inicio del ciclo de escritura, entregado por el simulador

4. Pruebas de funcionamiento y desempeño del sistema desarrollado

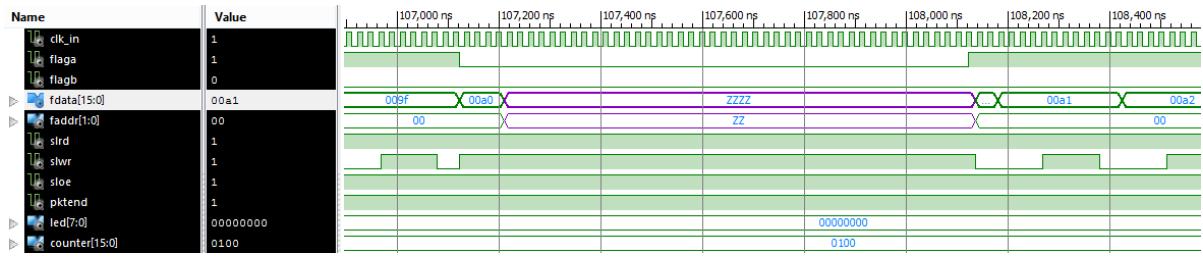


Figura 4.4: Simulación de una interrupción en el ciclo de escritura por falta de espacio en la memoria de la interfaz USB

este desarrollo, el informe muestra que se utiliza menos del 2 % de los recursos programables del FPGA Spartan 6. Tal vez el punto menos favorable del desarrollo es la cantidad de puertos utilizados, sin embargo quedan disponibles más de 60, lo cual no es un número despreciable . La transcripción del informe se puede observar en el Apéndice B.

4.1.2. Carga del sistema de pruebas en el FPGA

Para efectuar la carga del sistema fue necesario especificar en el entorno de desarrollo ISE provisto por la compañía Xilinx Inc, en qué pines del FPGA debe asignar cada uno de los puertos descriptos en el código del sistema. Con este propósito, se elabora un archivo denominado *User Constraints File* (que significa Archivo de Restricciones del Usuario) y lleva como extensión la sigla *ucf*. El texto completo de este archivo se observa en el Apéndice B.

El archivo *ufc* se generó a través de una interfaz gráfica provista por Xilinx Inc., denominada *Plan Ahead*. En ella se puede cargar en una planilla la equivalencia entre los pines y los puertos, como así también los niveles de tensión lógicos necesarios. En dicha planilla se cargaron los puertos como se indican en la Tabla 3.2. Estos puertos fueron configurados utilizando el estándar LVTTL, que es utilizado tanto en la placa Mojo v3 [38], como en el controlador FX2LP [36].

Otra configuración indicada en el archivo *ucf* fue la frecuencia de la entrada de reloj. Esta indicación es importante para detectar posibles errores temporales durante la fase de ruteo del compilador. Finalmente, se ejecutó el compilador para elaborar el archivo de configuración del FPGA, el cual fue cargado en la memoria Flash de la placa de desarrollo Mojo v3 y se constató que no existieron errores en la carga.

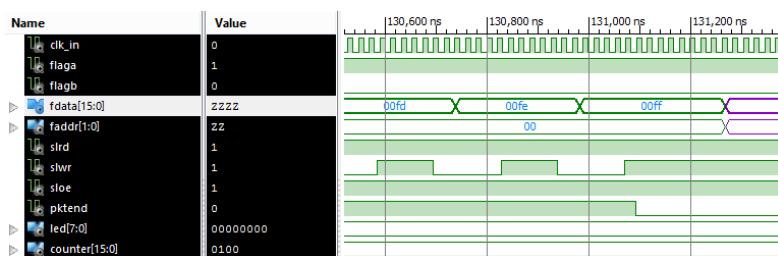


Figura 4.5: Final de la simulación

4.2. Desarrollo del programa de PC para enviar y recibir datos

La prueba del sistema consintió en el envío de una trama de datos desde la PC. La trama debe arribar al FPGA, atravesando la interfaz FX2LP. El FPGA, por su parte, almacena la trama de datos y, una vez finalizada la recepción, procede a la transmisión de la trama en el camino de regreso. Se generó un programa de computadoras, escrito en lenguaje C++, con el objetivo de poder crear tramas con datos aleatorios y repetir la prueba en forma automática. Se transmitieron datos entre la PC y el sistema desarrollado, en forma ininterrumpida, durante 24 horas con el objetivo de probar la robustez en la comunicación y calcular la tasa de datos transmitida por el bus.

Podrían existir capacidades parásitas en el circuito, a través del cual los datos son transmitidos, que disminuirían la tasa de transferencia a la que los datos podrían ser transmitidos. Por esto, los datos fueron generados en forma aleatoria logrando que las capacidades no deseadas en el circuito no sea enmascaradas por una trama invariable en el bus de datos.

Para elaborar el programa que se encarga de enviar y recibir los datos se utilizó la biblioteca **libusb-1.0**. En esta Sección se justificará la elección de la biblioteca **libusb-1.0** y se detallará la secuencia del programa desarrollado. El código completo con la implementación del programa se encuentra en el Apéndice C.

4.2.1. Elección de la biblioteca libusb-1.0

Existen bibliotecas que son utilizadas para obtener acceso a diversos periféricos. Entre ellas, una solución conocida y muy documentada es la biblioteca **libusb-1.0**, que permite transmitir datos por parte de una programa de computadoras escrito en lenguaje C++, a través de los puertos USB de la PC. Se eligió esta biblioteca para la realización del programa debido a que es una biblioteca de código abierto, es decir, que sus archivos fuente pueden ser leídos, modificados y/o utilizados por cualquier persona sin la necesidad de pagar una licencia. Otro motivo para su adopción es su característica de multiplataforma, que permite escribir códigos que funcionen en sistemas operativos tan diversos como Windows, Linux, Mac Os, Android, entre otros. Lo que no puede ser logrado con el uso de bibliotecas privativas como WinUSB.

Adicionalmente, la biblioteca **libusb-1.0** no tiene un autor específico, sino que existe una gran comunidad que contribuye al crecimiento del proyecto, como así también otros proyectos que utilizan esta biblioteca. Con el tiempo, se ha vuelto una biblioteca muy conocida, a través de los documentos que ha generado una gran comunidad de usuarios, existiendo manuales [39], ejemplos y foros [40] que facilitan el aprendizaje en su utilización y adaptaciones para diferentes lenguajes de programación, que se adapte a los conocimientos previos de la persona que desarrolla programas.

4.2.2. Programa de PC desarrollado

El programa de PC desarrollado para el envío y recepción de datos desde la PC hasta el FPGA consta de tres módulos. El primero de ellos se encarga de inicializar la biblioteca, identificar que el dispositivo USB conectado se corresponda con la comunicación con el FPGA y solicitar al sistema operativo el acceso a la comunicación. El segundo módulo se encarga de

	Paridad columna			
Fila 1	0	0	1	1
Fila 2	1	1	0	0
Fila 3	0	1	0	1
Paridad fila	1	0	1	0

Figura 4.6: Esquema de paridad par bidimensional de 4x4 bits.

No tengo esta foto!!! ahora tengo que hacerme un escape al CAB para sacarla!

Figura 4.7: El sistema desarrollado en funcionamiento

generar la trama de datos, enviarla hacia el FPGA y esperar su recepción. La trama de datos generados es almacenada para que en el tercer módulo del programa que, una vez recibida la transferencia desde el FPGA, se corrobore que los datos recibidos en la PC sean iguales a los enviados. El código completo del programa desarrollado, escrito en C++, se puede observar en el Apéndice C

En la primera etapa del programa, se inicializa la biblioteca `libusb-1.0` y se identifica el dispositivo. Para ello es necesario generar una lista con todos los dispositivos USB conectados. La lista de los dispositivos es informada por el sistema operativo a través de los identificadores contenidos en los descriptores del dispositivo. Si en la lista de dispositivos se encuentra el sistema desarrollado, se solicita acceso al sistema operativo. En caso contrario, el programa finaliza informando la situación.

Durante el segundo módulo del programa, se generan datos en forma aleatoria. Con el objetivo de obtener información adicional en caso de que existan errores en la información transmitida, se agregan bits de paridad par bidimensional (ver Figura 4.6), en tramas de 8x8 bits. Es decir, se genera un número aleatorio de 7 bits. Luego, se agrega un bit adicional, de forma tal que la cantidad de unos existentes en la palabra enviada sea par. Finalmente, se calcula la paridad de los bits con igual significancia en grupos de 7 números y se agrega el número resultante en el octavo lugar del grupo. Una vez generados los datos, estos son transmitidos hacia el FPGA. Los datos generados suman 128 B para cada una de las transferencias realizadas.

En la tercera etapa, el programa espera hasta la recepción del mensaje reenviado. Una vez que este arriba, se procede a corroborar que los datos recibido sean iguales a los enviados. La función encargada de verificar los datos corrobora que los bits de paridad sean correctos y, en caso de haber un error, informa cuál fue el dato erróneo. Tanto los datos enviados como los recibidos son almacenados en un archivo mientras el programa es ejecutado para un análisis posterior de los resultados.

4.3. Pruebas de la comunicación entre el FPGA y la PC

El sistema completo, montado y en funcionamiento, se muestra en la Figura 4.7. En él se puede apreciar el FPGA conectado a la interfaz USB través de la Placa de Interconexión. A su vez, la interfaz USB se enlaza con la PC a través de un cable. La conexión entre la interfaz USB con la PC sirve no solo para transferir los datos que llegarán el FPGA, sino también el

programa que ejecutará el controlador FX2LP.

Además, el FPGA también se conecta a la PC a través de un cable con el propósito de transferirle el archivo de programación y de proveerle alimentación a través del puerto USB. Con los dispositivos dispuestos en la configuración descripta, se procedió a cargar los diferentes programas elaborados para cada uno de ellos y, finalmente, se ejecutó el programa de pruebas. El programa de pruebas fue ejecutado por más de 24 horas con el objetivo de tener una buena cantidad de datos estadísticos como para probar la robustez del sistema, como así también su tasa de transferencia. Todas las transferencias, tanto de entrada como de salida a la PC fueron guardadas en un archivo de registro, documentando la fecha, el sentido de la comunicación y los datos intercambiados.

4.4. Resultados

A través de un análisis del registro de los datos enviados y recibidos se pudo determinar con mayor precisión cuanta información fue transmitida y durante qué intervalo de tiempo.

El registro dió cuenta de que el sistema estuvo funcionando durante 87134 segundos. Durante este lapso de tiempo fueron enviados 388.191.289 paquetes. Se recibió la misma cantidad de paquetes sin pérdida de información ni errores en la transmisión.

El programa de PC desarrollado genera paquetes de 128 B. Se debe considerar que cada paquete contiene un encabezado y una cola que también se transmite junto a los datos. Los valores típicos de encabezado y cola para transferencias en masa, como las que emite la PC en este trabajo, es de 55 B y para transferencias isócronas, que es el tipo de transferencia que llega a la PC desde la interfaz USB, este valor es de 38 B [27]. Por tanto, por cada una de los ciclos de envío y recepción de datos, se transfirieron 349 B:

$$2 \cdot 128 B + 55 B + 38 B = 349 B$$

Multiplicando el valor de datos transmitidos, por la cantidad de veces que la operación fue realizada, otorga la cantidad de bytes totales enviados. Esto es:

$$388 \times 10^3 \cdot 349 B = 135,5 \times 10^6 B$$

Cada Byte contiene 8 bits, por lo que al efectuar la operación de conversión se tiene que:

$$135,5 \times 10^6 B \cdot 8 \frac{bit}{B} = 1,08 \times 10^{12} bit$$

Finalmente, se puede calcular la tasa de bit, al dividir los bits transmitidos por la cantidad de tiempo total empleada en la prueba.

$$\frac{1,08 \times 10^{12} bit}{87,1 \times 10^3 s} = 12,4 \times 10^6 \frac{bit}{s}$$

La tasa de información útil intercambiada (sin contar encabezados y cola de la comunicación) se calcula multiplicando los 128 B de cada intercambio por la cantidad de paquetes, dividida en el tiempo que duró la prueba:

$$\frac{2 \cdot 128 B \cdot 388 \times 10^3}{87,1 \times 10^3 s} = 1,14 \times 10^3 \frac{B}{s}$$

El ancho de banda obtenida en esta prueba otorga 12.4 Mbit s^{-1} . Sin embargo, la tasa de bit real empleada en la comunicación podría ser mayor debido a que la prueba estaba usando dos EP con diferentes características, lo que limita el desempeño de la transmisión isócrona al de transferencias en masa y de las demoras propias de la prueba del sistema elaborado en la transferencia de los datos, considerando que se debe esperar recibir el paquete enviado antes de hacer un nuevo envío. No obstante, se constató que el LED D5 de la placa de desarrollo de Cypress destellaba durante la prueba, dando cuenta de que el sistema de comunicación USB funcionaba a alta velocidad (480 Mbit s^{-1}).

Aún en condiciones desfavorables de medida, la tasa de información útil obtenida en las pruebas fue de 1.14 MB s^{-1} , lo que es equivalente a 9.12 Mbit s^{-1} , la cual es superior a los 8 Mbit s^{-1} , máxima tasa alcanzable a través del sistema SPI de la placa Mojo [35] y a los 5 Mbit s^{-1} que otorgan los mejores dispositivos que emplean el sistema UART [41], ampliamente utilizado en la comunicación de desarrollos.

4.5. Sumario del capítulo

A lo largo del presente capítulo se presentaron las pruebas de desempeño del sistema realizado y los resultados de estas. Para la realización de las pruebas, los sistemas configurados en los capítulos precedentes fueron conectados. A su vez, se expuso la elaboración de un sistema tipo Eco en el FPGA con cada uno de los módulos que lo componen. Debido a la incorporación de estos componentes, se realizó una nueva verificación funcional de la síntesis del circuito con resultados positivos.

Luego se detalló el desarrollo de un software para PC, utilizando la biblioteca `libusb-1.0`, encargado de generar paquetes de datos, transmitirlos y comparar los resultados a fin de detectar errores. La prueba de desempeño fue repetida en forma ininterrumpida durante más de 24 horas logrando transmitir $1,08 \times 10^{12} \text{ bit}$ sin errores con el enlace USB funcionando a alta velocidad.

Capítulo 5

Conclusiones

En el transcurso del trabajo reportado por este informe fue cumplido el objetivo general, el cual consistió en el desarrollo de un sistema de comunicación USB 2.0 de alta velocidad destinado al intercambio de datos entre una PC y un FPGA. El sistema desarrollado fue capaz de transmitir $1,08 \times 10^{12}$, bit sin errores ni interrupciones durante 24 horas seguidas.

Además de cumplimentar con el objetivo general perseguido por este trabajo, se logró entender conceptos fundamentales del funcionamiento del USB, tal como el empaquetamiento de los datos y el tipo de transferencias que pueden realizarse a través de él. También se logró comprender cómo debe ser descripto un dispositivo USB al ser desarrollado y cómo debe ser informado a la PC. El sistema de comunicación implementado se compone de un software de computadora, una interfaz USB y un FPGA.

Se utilizó el controlador FX2LP, comercializado por la empresa Cypress Semiconductor como interfaz USB y a través de su estudio se pudo configurar dicho dispositivo, obteniendo un funcionamiento acorde a los requerimientos. El controlador FX2LP recibe transferencias en masa y transmite transferencias isócronas desde y hacia la PC respectivamente. Con el FPGA se comunica por un bus de 16 bits a través de una memoria FIFO comandada por el FPGA.

Se utilizó un FPGA Spartan 6 comercializado por la empresa Xilinx para implementar, en lenguaje VHDL, una Máquina de Estados Finitos capaz de enviar y recibir datos desde la interfaz USB. Dicha máquina de estados es utilizada para comandar la memoria FIFO presente en el controlador FX2LP. Se destaca el bajo consumo de recursos programables de FPGA por parte del sistema desarrollado, dejando lugar a la implementación de aplicaciones que utilicen la comunicación desarrollada, por ejemplo el desarrollo de sensores para equipos de radiografías y neutrógrafías o también de detectores de partículas utilizando sensores de imagen CMOS. Además se elaboró un circuito impreso destinado a la conexión física entre el FPGA y la interfaz USB.

Se desarrolló un software de computadoras que genera, envía y recibe datos hacia y desde el FPGA. Para la elaboración de este programa, se utilizó la biblioteca `libusb-1.0`, que permite la comunicación de programas con dispositivos conectados a través del bus USB. Esta es una biblioteca de código abierto y que puede ser ejecutado en cualquier sistema operativo.

Se implementó a su vez un sistema de pruebas compuesto de una memoria FIFO implementada en el FPGA, que recibe mensajes desde la interfaz USB y los retransmite de vuelta. Este sistema permitió testear la funcionalidad y robustez de la comunicación desarrollada. El sistema desarrollado fue probado y logró una conexión efectiva entre una PC y un FPGA, logrando en la prueba un intercambio de más de 1×10^{12} bits sin pérdida de información. La tasa de

transferencia de información útil lograda por la prueba de comunicación fue de 9.12 Mbit s^{-1} , superior a la máxima tasa posible a través del SPI de la placa Mojo o del protocolo UART.

5.1. Consideraciones Finales

El desarrollo expuesto en el presente trabajo puede ser de gran utilidad en el desarrollo de aplicaciones científicas. Se espera que el mismo sea utilizado en la lectura de imágenes adquiridas a través de sensores CMOS para aplicaciones tales como neutrógrafía y detección de radiación, entre otros.

Se plantea para trabajos futuros la posibilidad de implementar una nueva prueba de funcionamiento a través del envío ininterrumpido de tramas conocidas de datos, generados en forma local desde el FPGA, lo que permitirá conocer en mayor detalle la tasa máxima de bit posible con la configuración desarrollada en este trabajo.

Otra mejora a implementar en el sistema desarrollado consta de la realización sincrónica de la comunicación entre el FPGA y la Interfaz USB. Para ello, es necesario reconfigurar el controlador FX2LP, indicando que el funcionamiento de la memoria FIFO será de modo sincrónico. A su vez se debe rediseñar la placa de interconexión de forma tal que tanto el controlador FX2LP como el FPGA puedan compartir el reloj. Se espera que esta mejora brinde mayor velocidad al sistema, como así también se releve al FPGA de utilizar un PLL para brindar la señal de reloj necesaria, liberando recursos para su utilización en otro tipo de desarrollos implementados dentro de este.

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.” <https://www.olympus-lifescience.com/en/microscope-resource/primer/digitalimaging/cmosimagesensors/>. (Accesso: 10/07/2019).
- [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.

- [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, “ 32×32 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 Transmission 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. Acceso: 08/08/2019.
- [29] T. Riihonen, *Desing and analysis of duplexing Modes and Forwarding Protocols for OFDM(A) Relay Links*. PhD thesis, 2015.
- [30] Cypress Semiconductor, “EZ-USB ® Technical Reference Manual,” tech. rep., 2014.
- [31] B. Sklar, *Digital communications: fundamentals and applications*. 2001.
- [32] HW Group, “Hercules setup utility.” <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 Semiconductor, “CY3684/CY3684 EZ-USB Development Kit User Guide,” tech. rep., 2014.
- [35] Atmel, “ATmega16U4/ATmega32U4 Datasheet,” tech. rep., 2016.
- [36] Cypress, “CY7C68013A, CY7C68014A, CY7C68015A, CY7C68016A, EZ-USB(R) FX2LP(TM) USB Microcontroller High-Speed USB Peripheral Controller,” 2017.
- [37] Xilinx Inc., “Working with CORE Generator IP -” Acceso 03/03/2020.” https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_using_coregen_ip.htm. Acceso: 03/03/2020.
- [38] Embedded Micro, “Mojo Base Project.” <https://github.com/embmicro/mojo-base-project>. Acceso: 23/04/2020.
- [39] libusb, “libusb 1.0.” <https://libusb.info/>. Acceso: 04/11/2019.
- [40] The Persian Coder, “Introduction to using libusb-1.0.” <https://www.dreamincode.net/forums/topic/148707-introduction-to-using-libusb-10/>, 2010. Acceso 22/05/2020.
- [41] Texas Instrument, “NS16C2552 / NS16C2752 Dual UART with 16-byte / 64-byte FIFO ’ s and up to 5 Mbit / s Data Rate,” Tech. Rep. April, 2013.
- [42] Cypress Semiconductor, “CY3684 EZ-USB FX2LP Development Kit.” <https://www.cypress.com/documentation/development-kitsboards/cy3684-ez-usb-fx2lp-development-kit>. Acceso: 02/06/2020.

BIBLIOGRAFÍA

Apéndice A

Archivos de configuración del controlador FX2LP

En el presente apéndice se detallan los códigos fuente, los cuales fueron escritos en lenguaje C para microcontroladores. Estos códigos se utilizaron para configurar la interfaz FX2LP, incluyendo los archivos:

- **interfaz.c:** Funciones TD_Init(), TD_Poll() desarrolladas en este trabajo y funciones necesarias para el correcto funcionamiento del puerto USB, provistas por Cypress Semiconductor.
- **fw.c:** Función main() modificada por robustez.
- **leds.h:** Encabezado que contiene los registros necesarios para encender y apagar los LED multipropósito de la placa de desarrollo CY3684 EZ-USB de Cypress Semiconductor.
- **FX2LPSerial.h:** Encabezado de la biblioteca utilizada para transmitir datos a través del puerto UART.
- **FX2LPSerial.c:** Implementación de las funciones utilizadas para transmitir datos a través del puerto UART.

Además de los archivos que aquí se desarrollan, es necesario utilizar el framework provisto por Cypress [42].

interfaz.c

```
#pragma noiv                                // Do not generate interrupt vectors
#include "fx2.h"
#include "fx2regs.h"
#include "syncdly.h"                         // SYNCDELAY macro
5 #include "FX2LPSerial.h"

#include "leds.h"

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

```
10 extern BOOL    Sleep ;
extern BOOL    Rwuen ;
extern BOOL    Selfpwr ;

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

//-----
// Task Dispatcher hooks
//   The following hooks are called by the task dispatcher.
20 //-----
```

```
WORD blinktime = 0;
BYTE inblink = 0x00;
BYTE outblink = 0x00;
25 WORD blinkmask = 0;          // HS/FS blink rate

void TD_Init(void)           // Called once at startup
{
    BYTE aux;
30 // WORD i;

    // apago los 4 LED's
    aux = D2OFF;
    aux = D3OFF;
    aux = D4OFF;
35 aux = D5OFF;

    // Serial_Init configura el reloj
    FX2LPSerial_Init();
40 FX2LPSerial_XmitString("Serial_Port_Initialized");
    FX2LPSerial_XmitChar('\n');
    FX2LPSerial_XmitChar('\n');

    // Configuración de la interfaz FIFO a 48MHz
    // fifo con reloj interno , no salida de reloj ,
    // reloj no invertido , asncrono , fifo esclava(11)
45 IFCONFIG = 0xcb;

    SYNCDELAY;

50 //Configuraciones de los Flags en los puertos
PINFLAGSAB = 0xBC; // FLAGA <- EP2 Full Flag
                  // FLAGD <- EP2 Empty Flag
    SYNCDELAY;
55 PINFLAGSCD = 0x8F; // FLAGC <- EP8 Full Flag
                  // FLAGB <- EP8 Empty Flag
```

```
// Configuración de los extremos
EP1OUTCFG = 0xA0;
60 SYNCDELAY;
EP1INCFG = 0xA0;
SYNCDELAY;
EP4CFG &= 0x7F;
SYNCDELAY;
EP6CFG &= 0x7F;
65 SYNCDELAY;
EP8CFG = 0xA0; //EP8 is DIR=OUT, TYPE=BULK, SIZE=512, BUF=2x
SYNCDELAY;
EP2CFG = 0xD2; // EP2 is DIR=IN, TYPE=ISOC, SIZE=512, BUF=2x
70 SYNCDELAY;

// resetear la memoria para asegurarse de que est vaca
FIFORESET = 0x80;
75 SYNCDELAY;
FIFORESET = 0x02;
SYNCDELAY;
FIFORESET = 0x04;
SYNCDELAY;
FIFORESET = 0x06;
80 SYNCDELAY;
FIFORESET = 0x08;
SYNCDELAY;
FIFORESET = 0x00;
SYNCDELAY;

85 EP8FIFO CFG = 0x00;
SYNCDELAY;
EP2FIFO CFG = 0x00;
SYNCDELAY;

90 // Establecer modo automático. Se establece con el flanco ascendente
EP8FIFO CFG = 0x11;
SYNCDELAY;
EP2FIFO CFG = 0x0D;
95 SYNCDELAY;

98 // Se habilitan las interrupciones de para recibir paquetes SOF
USBIE |= bmSOF;

100 FX2LPSerial_XmitString("Interfaz _Configurada\n");
105 }
```

```
void TD_Poll(void) // Llamada por el lazo principal de forma repetida
{
    BYTE aux;
```

```
110      if(EP8FIFOFLGS & bmBIT1)// ep8 fifo vacia
{           aux = D4ON;
}
else
{
    aux = D4OFF;
}

115      if(EP8FIFOFLGS & bmBIT0)// ep8 fifo llena
{           aux = D3ON;
}
else
{
    aux = D3OFF;
}

120      if(EP2FIFOFLGS & bmBIT1)// ep2 fifo vacia
{           aux = D2ON;
}
else
{
    aux = D2OFF;
}

125      }

130      }

135      }

BOOL TD_Suspend( void ) // Called before the device goes into suspend mode
{
140      return(TRUE);
}

BOOL TD_Resume( void ) // Called after the device resumes
{
145      return(TRUE);
}

BOOL DR_GetDescriptor( void )
{
150      FX2LPSerial_XmitString( "Getting_Descriptor:\r\n");
      return(TRUE);
}
```

```
155    BOOL DR_SetConfiguration( void ) // Called when a Set Configuration
           // command is received
    {
        FX2LPSerial_XmitString( "Setting_Config\n\n" );
        Configuration = SETUPDAT[ 2 ];
        return(TRUE); // Handled by user code
    }
160

165    BOOL DR_GetConfiguration( void ) // Called when a Get Configuration
           // command is received
    {
        FX2LPSerial_XmitString( "Getting_Config\n\n" );
        EP0BUF[ 0 ] = Configuration;
        EP0BCH = 0;
        EP0BCL = 1;
        return(TRUE); // Handled by user code
    }
170

175    BOOL DR_SetInterface( void ) // Called when a Set Interface
           // command is received
    {
        FX2LPSerial_XmitString( "Setting_Interface\n\n" );

        AlternateSetting = SETUPDAT[ 2 ];
        return(TRUE); // Handled by user code
    }
180

185    BOOL DR_GetInterface( void ) // Called when a Set Interface
           // command is received
    {
        FX2LPSerial_XmitString( "Getting_Interface\n\n" );
        EP0BUF[ 0 ] = AlternateSetting;
        EP0BCH = 0;
        EP0BCL = 1;
        return(TRUE); // Handled by user code
    }
190

195    BOOL DR_GetStatus( void )
    {
        FX2LPSerial_XmitString( "Getting_Status\n\n" );
        return(TRUE);
    }

200    BOOL DR_ClearFeature( void )
    {
        FX2LPSerial_XmitString( "Clearing_Features\n\n" );
        return(TRUE);
    }
```

```

BOOL DR_SetFeature( void )
{
    FX2LPSerial_XmitString( "Setting_Features\n\n" );
    return(TRUE);
}

BOOL DR_VendorCmnd( void )
{
    return(TRUE);
}

//-----
215 // USB Interrupt Handlers
//    The following functions are called by the USB interrupt jump table.
//-----

// Setup Data Available Interrupt Handler
220 void ISR_Sudav( void ) interrupt 0
{
    FX2LPSerial_XmitString( "SUDAv_ISR\n" );
    GotSUD = TRUE;           // Set flag
225    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSUDAV;        // Clear SUDAV IRQ
}

// Setup Token Interrupt Handler
230 void ISR_Sutok( void ) interrupt 0
{
    FX2LPSerial_XmitString( "SUTok_ISR\n" );
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSUTOK;        // Clear SUTOK IRQ
235 }

void ISR_Sof( void ) interrupt 0
{
    BYTE aux;
240    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSOF;          // Clear SOF IRQ
    blinktime++;
    if( blinktime&blinkmask )
        aux=D5OFF; // ~1/sec blinking LED
    else
        aux=D5ON;

    if(--inblink == 0)
        aux = D4OFF;
}

```

```

250     if (--outblink == 0)
251         aux = D3OFF;
252     }
253
254     void ISR_Ures( void ) interrupt 0
255     {
256         BYTE aux;
257
258         // Whenever we get a USB Reset , we should revert to full speed mode
259         FX2LPSerial_XmitString( "URst_ISR\n" );
260         pConfigDscr = pFullSpeedConfigDscr;
261         ((CONFIGDSCR xdata *) pConfigDscr)->type = CONFIG_DSCR;
262         pOtherConfigDscr = pHighSpeedConfigDscr;
263         ((CONFIGDSCR xdata *) pOtherConfigDscr)->type = OTHERSPEED_DSCR;
264
265         EZUSB_IRQ_CLEAR();
266         USBIRQ = bmURES;           // Clear URES IRQ
267         aux = D2ON;                // Turn off high-speed LED
268         blinkmask = 0x0400;         // 2 sec period for FS
269     }
270
271     void ISR_Susp( void ) interrupt 0
272     {
273         Sleep = TRUE;
274         EZUSB_IRQ_CLEAR();
275         USBIRQ = bmSUSP;
276     }
277
278     void ISR_Highspeed( void ) interrupt 0
279     {
280         blinkmask = 0x1000;        // 1 sec period for HS
281         if (EZUSB_HIGHSPED())
282         {
283             pConfigDscr = pHighSpeedConfigDscr;
284             ((CONFIGDSCR xdata *) pConfigDscr)->type = CONFIG_DSCR;
285             pOtherConfigDscr = pFullSpeedConfigDscr;
286             ((CONFIGDSCR xdata *) pOtherConfigDscr)->type = OTHERSPEED_DSCR;
287
288             // This register sets the number of Isoc packets to send per
289             // uFrame. This register is only valid in high speed.
290             EP2ISOINPKTS = 0x81;//0x83;    // 3 packets per microframe , AADJ=1
291                         SYNCDELAY;
292         }
293         else
294         {
295             pConfigDscr = pFullSpeedConfigDscr;
296             pOtherConfigDscr = pHighSpeedConfigDscr;
297         }

```

```
500     EZUSB_IRQ_CLEAR();  
      USBIRQ = bmHSGRANT;  
    }  
    // Interrupciones no utilizadas fueron borradas  
    // Para mayor detalle de las interrupciones  
    // vectorizadas disponibles , consultar la  
305    // documentacion de Cypress Semiconductor
```

fw.c

```
#define DELAY_COUNT 0x9248*8L // Delay for 8 sec at 24Mhz,  
                           // 4 sec at 48  
#define _IFREQ 48000          // IFCLK constant for  
                           // Synchronization Delay  
5 #define _CFREQ 48000         // CLKOUT constant for  
                           // Synchronization Delay  
  
#include "fx2.h"  
#include "fx2regs.h"  
10 #include "syncdly.h"        // Define _IFREQ and _CFREQ above this #include  
#include "FX2LPSerial.h"  
  
#define min(a,b) (((a)<(b))?(a):(b))  
15 #define max(a,b) (((a)>(b))?(a):(b))  
  
// Registers which require a synchronization delay  
// FIFORESET           FIFOPINPOLAR  
// INPKTEND            OUTPKTEND  
20 // EPxBCH:L          REVCTL  
// GPIFTCB3            GPIFTCB2  
// GPIFTCB1            GPIFTCB0  
// EPxFIFOPFH:L       EPxAUTOINLENH:L  
// EPxFIFOCFG          EPxGPIFFLGSEL  
25 // PINFLAGSxx         EPxFIFOIRQ  
// EPxFIFOIE           GPIFIRQ  
// GPIFIE              GPIFADRH:L  
// UDMACRCH:L          EPxGPIFTRIG  
// GPIFTRIG  
30 // Note: The pre-REVE EPxGPIFTCH/L register are affected , as well...  
// ...these have been replaced by GPIFTC[B3:B0] registers  
  
volatile BOOL GotSUD;  
35 BOOL Rwuen;  
BOOL Selfpwr;  
volatile BOOL Sleep;
```

```
WORD    pDeviceDscr; // Pointer to Device Descriptor;  
40      // Descriptors may be moved  
WORD    pDeviceQualDscr;  
WORD    pHightSpeedConfigDscr;  
WORD    pFullSpeedConfigDscr;  
WORD    pConfigDscr;  
45 WORD    pOtherConfigDscr;  
WORD    pStringDscr;  
  
void SetupCommand( void );  
void TD_Init( void );  
50 void TD_Poll( void );  
BOOL TD_Suspend( void );  
BOOL TD_Resume( void );  
  
BOOL DR_GetDescriptor( void );  
55 BOOL DR_SetConfiguration( void );  
BOOL DR_GetConfiguration( void );  
BOOL DR_SetInterface( void );  
BOOL DR_GetInterface( void );  
BOOL DR_GetStatus( void );  
60 BOOL DR_ClearFeature( void );  
BOOL DR_SetFeature( void );  
BOOL DR_VendorCmnd( void );  
  
// this table is used by the epcs macro  
65 const char code EPSCS_Offset_Lookup_Table[] =  
{  
    0,      // EP1OUT  
    1,      // EP1IN  
    2,      // EP2OUT  
70    2,      // EP2IN  
    3,      // EP4OUT  
    3,      // EP4IN  
    4,      // EP6OUT  
    4,      // EP6IN  
75    5,      // EP8OUT  
    5,      // EP8IN  
};  
  
// macro for generating the address of an endpoint's control  
80 // and status register (EPnCS)  
#define epcs(EP) (EPSCS_Offset_Lookup_Table[(EP & 0x7E) | (EP > 128)] + 0xE6A1)  
  
// Task dispatcher  
void main( void )  
85 {  
    DWORD i;
```

```
WORD    offset ;
DWORD   DevDescrLen ;
DWORD   j=0;
90      WORD   IntDescrAddr ;
WORD   ExtDescrAddr ;

// Initialize Global States
Sleep = FALSE; // Disable sleep mode
95      Rwuен = FALSE; // Disable remote wakeup
Selfpwr = FALSE; // Disable self powered
GotSUD = FALSE; // Clear "Got_Setup_Data" flag

// Initialize user device
100     TD_Init();

FX2LPSerial_XmitString("Initialized\n\n");
// The following section of code is used to relocate the descriptor
// table.
// Since the SUDPTRH and SUDPTRL are assigned the address of the
// descriptor table, the descriptor table must be located in on-part
// memory.
// The 4K demo tools locate all code sections in external memory.
// The descriptor table is relocated by the frameworks ONLY if it
// is found to be located in external memory.
110     pDeviceDscr = (WORD)&DeviceDscr;
pDeviceQualDscr = (WORD)&DeviceQualDscr;
pHighSpeedConfigDscr = (WORD)&HighSpeedConfigDscr;
pFullSpeedConfigDscr = (WORD)&FullSpeedConfigDscr;
pStringDscr = (WORD)&StringDscr;

115     FX2LPSerial_XmitString("DeviceDscr:0x");
FX2LPSerial_XmitHex4(pDeviceDscr);
FX2LPSerial_XmitChar('\n');

120     if ((WORD)&DeviceDscr & 0xe000)
{
    IntDescrAddr = INTERNAL_DSCR_ADDR;
    ExtDescrAddr = (WORD)&DeviceDscr;
    DevDescrLen = (WORD)&UserDscr - (WORD)&DeviceDscr + 2;
    125    for (i = 0; i < DevDescrLen; i++)
        *((BYTE xdata *)IntDescrAddr+i) = 0xCD;
    for (i = 0; i < DevDescrLen; i++)
        *((BYTE xdata *)IntDescrAddr+i) = *((BYTE xdata *)ExtDescrAddr+i);
    pDeviceDscr = IntDescrAddr;
    130    offset = (WORD)&DeviceDscr - INTERNAL_DSCR_ADDR;
    pDeviceQualDscr -= offset;
    pConfigDscr -= offset;
    pOtherConfigDscr -= offset;
```

```

135     pHIGHSpeedConfigDscr -= offset;
      pFULLSpeedConfigDscr -= offset;
      pStringDscr -= offset;
    }

140     EZUSB_IRQ_ENABLE();           // Enable USB interrupt (INT2)
      EZUSB_ENABLE_RSMIRQ();        // Wake-up interrupt

      // What is INT2 is for USB & INT4 is for the Slave FIFOs
      INTSETUP |= (bmAV2EN | bmAV4EN); // Enable INT 2 & 4 autovectoring

145     // Enable selected interrupts
      USBIE |= bmSUDAV | bmSUTOK | bmSUSP | bmURES | bmHSGRANT;

      // Global interrupt enable. Controls masking of all interrupts except
      // USB wakeup (resume). EA = 0 disables all interrupts except USB wakeup.
      // When EA = 1, interrupts are enabled or masked by their individual
      // enable bits.
      EA = 1; // Enable 8051 interrupts

155 #ifndef NORENUM
      // Renumerate if necessary. Do this by checking the renum bit. If it
      // is already set, there is no need to reenumerate. The renum bit will
      // already be set if this firmware was loaded from an eeprom.
      if (!(USBCS & bmRENUM))
160 {
      EZUSB_Discon(TRUE); // reenumerate
    }
#endif

165 // unconditionally re-connect. If we loaded from eeprom we are
// disconnected and need to connect. If we just renumerated this
// is not necessary but doesn't hurt anything
      USBCS &= ~bmDISCON;

170 // esta linea soluciona problemas de booteo
      FX2LPSerial_XmitString("Reconnecting...\n\n");
      // The three LSBs of the Clock Control Register (CKCON, at SFR
      // location 0x8E) control the stretch value; stretch values
      // between zero and seven may be used. A stretch value of
      // zero adds zero instruction cycles, resulting in MOVX
      // instructions which execute in two instruction cycles.

      // Set stretch to 0 (after reenumeration)
      CKCON = (CKCON & (~bmSTRETCH)) | FW_STRETCH_VALUE;

175 // clear the Sleep flag.
      Sleep = FALSE;

```

```

// Task Dispatcher
185  while(TRUE)           // Main Loop
{
    if(GotSUD)           // Wait for SUDAV
    {
        FX2LPSerial_XmitString("Get_Setup_Data_Available\n");
        190   SetupCommand();          // Implement setup command
        GotSUD = FALSE;           // Clear SUDAV flag
    }

    // Poll User Device
    // NOTE: Idle mode stops the processor clock. There are only two
    // ways out of idle mode, the WAKEUP pin, and detection of the USB
    // resume state on the USB bus. The timers will stop and the
    // processor will not wake up on any other interrupts.
    if (Sleep)
    200   {
        if(TD_Suspend())
        {
            Sleep = FALSE;           // Clear the "go_to_sleep" flag.
            // Do it here to prevent any race
            // condition between wakeup and the
            // next sleep.
            do
            {
                EZUSB_Susp();         // Place processor in idle mode.
            }
            210   while(!Rwuen && EZUSB_EXTWAKEUP());
                // Must continue to go back into suspend if the host has
                // disabled remote wakeup *and* the wakeup was caused by
                // the external wakeup pin.

            // 8051 activity will resume here due to USB bus or Wakeup# pin
            // activity.
            EZUSB_Resume();          // If source is the Wakeup# pin, signal the
            // host to Resume.
            220   TD_Resume();
        }
    }
    TD_Poll();
}

// Device request parser
void SetupCommand(void)
{
    void *descr_ptr;

```

```

FX2LPSerial_XmitString( "Setup_Command\n" );

switch (SETUPDAT[1])
{
    case SC_GET_DESCRIPTOR:                                // *** Get Descriptor
        if (DR_GetDescriptor())
            switch (SETUPDAT[3])
            {
                case GD_DEVICE:                            // Device
                    FX2LPSerial_XmitString("Device\n\n");
                    SUDPTRH = MSB(pDeviceDscr);
                    SUDPTRL = LSB(pDeviceDscr);
                    break;
                case GD_DEVICE_QUALIFIER:                 // Device Qualifier
                    FX2LPSerial_XmitString("Device_Qualifier\n\n");
                    SUDPTRH = MSB(pDeviceQualDscr);
                    SUDPTRL = LSB(pDeviceQualDscr);
                    break;
                case GD_CONFIGURATION:                  // Configuration
                    FX2LPSerial_XmitString("Configuration\n\n");
                    SUDPTRH = MSB(pConfigDscr);
                    SUDPTRL = LSB(pConfigDscr);
                    break;
                case GD_OTHER_SPEED_CONFIGURATION: // Other Speed Configuration
                    FX2LPSerial_XmitString("Other_Speed_Configuration\n\n");
                    SUDPTRH = MSB(pOtherConfigDscr);
                    SUDPTRL = LSB(pOtherConfigDscr);
                    break;
                case GD_STRING:                           // String
                    FX2LPSerial_XmitString("String\n\n");
                    if (dscr_ptr = (void *)EZUSB_GetStringDscr(SETUPDAT[2]))
                    {
                        SUDPTRH = MSB(dscr_ptr);
                        SUDPTRL = LSB(dscr_ptr);
                    }
                    else
                        EZUSB_STALL_EP0();      // Stall End Point 0
                    break;
                default:                               // Invalid request
                    EZUSB_STALL_EP0();      // Stall End Point 0
            }
        break;
    case SC_GET_INTERFACE:                                // *** Get Interface
        DR_GetInterface();
        break;
    case SC_SET_INTERFACE:                                // *** Set Interface
        DR_SetInterface();
}

```

```

        break;
280    case SC_SET_CONFIGURATION:           // *** Set Configuration
        DR_SetConfiguration();
        break;
285    case SC_GET_CONFIGURATION:         // *** Get Configuration
        DR_GetConfiguration();
        break;
case SC_GET_STATUS:                // Get Status
290    if(DR_GetStatus())
        switch(SETUPDAT[0])
    {
        case GS_DEVICE:                  // Device
            EP0BUF[0] = ((BYTE)Rwuen << 1) | (BYTE)Selfpwr;
            EP0BUF[1] = 0;
            EP0BCH = 0;
            EP0BCL = 2;
            break;
295    case GS_INTERFACE:               // Interface
            EP0BUF[0] = 0;
            EP0BUF[1] = 0;
            EP0BCH = 0;
            EP0BCL = 2;
            break;
300    case GS_ENDPOINT:               // End Point
            EP0BUF[0] = *(BYTE*xdata*)epcs(SETUPDAT[4]) & bmEPSTALL;
            EP0BUF[1] = 0;
            EP0BCH = 0;
            EP0BCL = 2;
            break;
305    default:                      // Invalid Command
            EZUSB_STALL_EP0();          // Stall End Point 0
310    }
        break;
case SC_CLEAR_FEATURE:             // *** Clear Feature
315    if(DR_ClearFeature())
        switch(SETUPDAT[0])
    {
        case FT_DEVICE:                 // Device
            if(SETUPDAT[2] == 1)
                Rwuen = FALSE;        // Disable Remote Wakeup
            else
                EZUSB_STALL_EP0();    // Stall End Point 0
320        break;
        case FT_ENDPOINT:               // End Point
            if(SETUPDAT[2] == 0)
        {
            *(BYTE*xdata*)epcs(SETUPDAT[4]) &= ~bmEPSTALL;
325            EZUSB_RESET_DATA_TOGGLE(SETUPDAT[4]);
        }
    }
}

```

```

        }
        else
            EZUSB_STALL_EP0();    // Stall End Point 0
330      break;
    }
    break;
case SC_SET_FEATURE:           // *** Set Feature
    if(DR_SetFeature())
335      switch(SETUPDAT[0])
    {
        case FT_DEVICE:          // Device
        if(SETUPDAT[2] == 1)
            RwuEn = TRUE;      // Enable Remote Wakeup
40      else if(SETUPDAT[2] == 2)
            // Set Feature Test Mode. The core handles this request.
            // However, it is necessary for the firmware to complete
            // the handshake phase of the control transfer before the
            // chip will enter test mode. It is also necessary for FX2
            // to be physically disconnected (D+ and D-) from the host
            // before it will enter test mode.
            break;
        else
            EZUSB_STALL_EP0();    // Stall End Point 0
50      break;
    case FT_ENDPOINT:          // End Point
        *(BYTE xdata *) epcS(SETUPDAT[4]) |= bmEPSTALL;
        break;
    }
    break;
355  default:                // *** Invalid Command
    if(DR_VendorCmnd())
        EZUSB_STALL_EP0();    // Stall End Point 0
    }

360  // Acknowledge handshake phase of device request
    EP0CS |= bmHSNAK;
}

365 // Wake-up interrupt handler
void resume_isr(void) interrupt WKUP_VECT
{
    EZUSB_CLEAR_RSMIRQ();
}

```

leds.h

```

xdata volatile const BYTE D2ON      _at_ 0x8800;
xdata volatile const BYTE D2OFF     _at_ 0x8000;
xdata volatile const BYTE D3ON      _at_ 0x9800;

```

```
5    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;
```

FX2LPSerial.h

```
#ifndef INCLUDED_FX2LPSERIAL_H  
#define INCLUDED_FX2LPSERIAL_H  
  
5 #define FX2LP_SERIAL  
#ifdef FX2LP_SERIAL  
  
    extern void FX2LPSerial_Init() ;  
10   extern void  
        FX2LPSerial_XmitChar(char ch) reentrant;  
  
    extern void  
15   FX2LPSerial_XmitHex1(BYTE b) ;  
  
    extern void  
        FX2LPSerial_XmitHex2(BYTE b) ;  
  
20   extern void  
        FX2LPSerial_XmitHex4(WORD w) ;  
  
    extern void  
        FX2LPSerial_XmitString(char *str) reentrant;  
25  
#endif  
#endif
```

FX2LPSerial.c

```
#include "fx2.h"  
#include "fx2regs.h"  
#include "FX2LPSerial.h"  
  
5 void FX2LPSerial_Init() // initializes the registers for using Timer2 as  
// baud rate generator for a Baud rate of 38400.  
{  
    T2CON = 0x34 ;  
10   RCAP2H = 0xFF ;  
    RCAP2L = 0xD9;
```

```
SCON0 = 0x5A ;
TI = 1;

15 CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1) ; // Setting up the clock frequency

FX2LPSerial_XmitString("\r\n->") ; // Clearing the output screen

20 }

void FX2LPSerial_XmitChar(char ch) reentrant // prints a character
{
    while (TI == 0) ;
    TI = 0 ;
    SBUF0 = ch ; // print the character
}

25 void FX2LPSerial_XmitHex1(BYTE b) // intermediate function to print the
                                    // 4-bit nibble in hex format
{
    if (b < 10)
        FX2LPSerial_XmitChar(b + '0') ;
    else
        FX2LPSerial_XmitChar(b - 10 + 'A') ;
}

30 void FX2LPSerial_XmitHex2(BYTE b) // prints the value of the BYTE
                                    // variable in hex
{
    FX2LPSerial_XmitHex1((b >> 4) & 0x0f) ;
    FX2LPSerial_XmitHex1(b & 0x0f) ;
}

35 void FX2LPSerial_XmitHex4(WORD w) // prints the value of the WORD
                                    // variable in hex
{
    FX2LPSerial_XmitHex2((w >> 8) & 0xff) ;
    FX2LPSerial_XmitHex2(w & 0xff) ;
}

40 }

45 void FX2LPSerial_XmitString(char *str) reentrant
{
    while (*str)
        FX2LPSerial_XmitChar(*str++) ;
}

50 }

55 }
```


Apéndice B

Códigos de la síntesis en FPGA

El presente apéndice contiene los códigos desarrollados para la síntesis de los sistemas implementados en el FPGA Spartan 6 utilizados en este trabajo. Estos códigos fueron escritos en lenguaje VHDL, salvo por el archivo de restricciones, que posee como lenguaje XST, desarrollado por Xilinx para dicho propósito. A su vez, se exponen el resumen de síntesis, en donde constan los recursos del FPGA utilizados. El contenido mostrado en este Apéndice contiene los códigos de los archivos:

- **fx2lp_interface.vhd:** Implementación de la Máquina de Estados Finitos (MEF) a través de la cual se generan las señales de control que comandan la memoria FIFO del controlador FX2LP
- **mef_tb.vhd:** Código utilizado para realizar la verificación funcional de la MEF
- **fx2lp_interface_top.vhd:** Implementación del sistema de pruebas. En él se instancia la MEF, la configuración del PLL y la memoria FIFO generada a través de la herramienta *Core Generator* de Xilinx Inc.
- **top_tb.vhd:** Código de verificación funcional del sistema de pruebas
- **fx2lp_interface_top.ucf:** Archivo de restricciones en donde se le indica al compilador la frecuencia de la señal de entrada al sistema y la asignación de los puertos del sistema desarrollado a los pines del FPGA Spartan 6.
- **Sumario de síntesis:** Tabla en donde se indica el consumo de recursos de FPGA por parte de la síntesis realizada a través del entorno de desarrollo ISE de Xilinx Inc.

fx2lp_interface.vhd

```
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";
    );
    port(
        ent_ep : in std_logic;
        sal_ep : out std_logic;
        ent_din : in std_logic_vector(7 downto 0);
        sal_dout : out std_logic_vector(7 downto 0)
    );
end entity;
```

```

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
    pktend   :out std_logic;
);
end fx2lp_interfaz;

architecture Behavioral of fx2lp_interfaz is

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";
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';
signal esc_fflag : std_logic := '1';

signal reloj_sist : std_logic;

-- senales de temporizacion
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
type estados_mef is

```

```

(
    inicio ,
    lec_direccion , lectura ,
    esc_direccion , escritura
60 );
    signal estado_actual , prox_estado : estados_mef := inicio ;

begin
    -- conexion 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;

100   with estado_actual select
            fdata_ent <= fdata      when lectura | lec_dir ,
                                         (others => 'Z') when others;

```

```
105          fdata_ent  when others;  
--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;  
125  
        when lectura =>  
            if lec_eflag = '0' then  
                prox_estado <= lec_dir;  
            else  
                prox_estado <= inicio;  
            end if;  
  
        when esc_direccion =>  
            prox_estado <= escritura;  
135  
        when escritura =>  
            if enviar_datos = '1' then  
                if lec_eflag = '1' and esc_fflag = '0' then  
                    prox_estado <= esc_direccion;  
                else  
                    prox_estado <= inicio;  
                end if;  
            else  
                prox_estado <= inicio;  
            end if;  
145  
        when others =>  
            prox_estado <= inicio;  
    end case;  
end process interfaz_mef;
```

```
-- temporizaciones
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 lectura | lec_dir | escritura ,
    '0' when others;

end Behavioral;
mef_tb.vhd

LIBRARY ieee;
```

```
USE ieee.std_logic_1164.ALL;
ENTITY mef_tb IS
5 END mef_tb;

ARCHITECTURE behavior OF mef_tb IS
10
-- Component Declaration for the Unit Under Test (UUT)
COMPONENT fx2lp_interfaz
15 PORT(
    reloj           : in    std_logic;
    reset           : in    std_logic;
    enviar_datos   : in    std_logic;
    dato_recibido  : out   std_logic_vector(15 downto 0);
    dato_a_enviar  : in    std_logic_vector(15 downto 0);
    fdata           : inout std_logic_vector(15 downto 0);
    faddr           : out   std_logic_vector(1 downto 0);
20
    slrd            : out   std_logic;
    slwr            : out   std_logic;
    flaga           : in    std_logic;
    flagb           : in    std_logic;
    flagc           : in    std_logic;
25
    flagd           : in    std_logic;
    sloe            : out   std_logic;
    pktend          : out   std_logic
);
30
END COMPONENT;
-- Inputs
35
signal reloj      : std_logic := '0';
signal reset      : std_logic := '0';
signal flaga      : std_logic := '0';
signal flagb      : std_logic := '0';
signal flagc      : std_logic := '0';
40
signal flagd      : std_logic := '0';
signal enviar_datos: std_logic := '0';
signal dato_a_enviar: std_logic_vector(15 downto 0) := (others => '0');

-- BiDirs
signal fdata : std_logic_vector(15 downto 0);

-- Outputs
45
signal faddr      : std_logic_vector(1 downto 0);
signal slrd       : std_logic;
signal slwr       : std_logic;
signal sloe       : std_logic;
```

```
50      signal pktend      : std_logic;
      signal dato_recibido : std_logic_vector(15 downto 0);

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

55      BEGIN

      -- Instantiate the Unit Under Test (UUT)
      uut: fx2lp_interfaz PORT MAP (
          reloj => reloj,
          reset => reset,
          enviar_datos => enviar_datos,
          dato_recibido => dato_recibido,
          dato_a_enviar => dato_a_enviar,
          fdata => fdata,
          faddr => faddr,
          slrd => slrd,
          slwr => slwr,
          flaga => flaga,
          flagb => flagb,
          flagc => flagc,
          flagd => flagd,
          sloe => sloe,
          pktend => pktend
        );
75

      reloj <= not reloj after periodo_reloj/2;
      reset <= '0', '1' after 100 ns;

80      -- Stimulus process
      stim_proc: process
      begin
          flaga <= '1';
          flagb <= '0';
85          flagc <= '1';
          flagd <= '0';
          enviar_datos <= '0';
          dato_a_enviar <= (others => 'Z');
          wait until reset = '1';

90          wait for periodo_reloj*5;

          flagb <= '1';
          wait until sloe = '0';

95          fdata <= x"AAAA";
          wait until slrd = '0';
```

```
100      fdata <= x"8888";
          wait until slrd = '0';

          fdata <=x"3846";
          wait until slrd = '0';

105      —no interrupcion de lectura
          fdata <= x"aaaa";
          flagb <= '0';
          wait until rising_edge(slrd);
          dato_a_enviar <= x"abcd";
110      enviar_datos <= '1';

          —operacion de escritura
          wait until sloe='1';
          fdata <=(others => 'Z');
          wait until slwr='0';

          dato_a_enviar <= x"392a";
          wait until slwr='0';

120      dato_a_enviar <= x"4444";
          wait until slwr='0';

          dato_a_enviar <= x"a8b2";
          —interrupcion por prioridad
125      flagb <= '1';
          wait until sloe = '0';
          fdata <= x"1515";
          wait until slrd='0';
          flagb <= '0';
130      wait until sloe = '1';
          — operacion de escritura
          dato_a_enviar <= x"2525";
          fdata <= (others => 'Z');
          wait until slwr = '0';

135      dato_a_enviar <= x"7868";
          wait until slwr = '0';
          —interrupcion por llenado de memoria
          flaga <= '0';
          —final de operacion escritura
140      wait for periodo_reloj*10;
          wait;
      end process;
END;
```

fx2lp_interface_top.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library unisim;
5 use unisim.vcomponents.all;

entity fx2lp_interface_top is
generic(
    — direcciones EP:
    — * EP8 = "11"
    — * EP2 = "00"
10   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
15 );
port(
    — senales que van y vienen desde y hacia el FX2LP
    fdata :inout std_logic_vector(ancho_bus-1 downto 0);
    — entrada y salida de datos FIFO
20   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_flag_lleno
25   flagb :in std_logic;— EP8_empty—>lec_flag_vacio
    flagc :in std_logic;— EP8_full—>lec_flag_lleno
    flagd :in std_logic;— EP2_empty—>esc_flag_vacio
    sloe :out std_logic;— senal de habilitacion de salida
    pktend :out std_logic;
30   — reloj
    reloj_sal :out std_logic;— salida de reloj
    — senales desde y hacia mojo
    reloj_ent :in std_logic;— entrada de reloj
    pulsador :in std_logic;— activo en bajo
35   led :out std_logic_vector(7 downto 0)

    );
end fx2lp_interface_top;

40 architecture fx2lp_interface_arq of fx2lp_interface_top is

    — declaracion de componentes
COMPONENT fx2lp_interface
    GENERIC(
        constant ent_ep_addr:std_logic_vector(1 downto 0);
        constant sal_ep_addr:std_logic_vector(1 downto 0);
        constant ancho_bus :integer
45
```

```
 );
PORT(
50    reloj : IN std_logic;
    reset : IN std_logic;
    enviar_datos : IN std_logic;
    dato_a_enviar : IN std_logic_vector(15 downto 0);
    flaga : IN std_logic;
55    flagb : IN std_logic;
    flagc : IN std_logic;
    flagd : IN std_logic;
    fdata : INOUT std_logic_vector(15 downto 0);
    dato_recibido : OUT std_logic_vector(15 downto 0);
60    faddr : OUT std_logic_vector(1 downto 0);
    slrd : OUT std_logic;
    slwr : OUT std_logic;
    sloe : OUT std_logic;
    pktend : OUT std_logic
65    );
END COMPONENT;

COMPONENT fifo_generator_v9_3
70    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;
45        rd_en : IN STD_LOGIC;
        dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        full : OUT STD_LOGIC;
        empty : OUT STD_LOGIC;
80        valid : OUT STD_LOGIC
    );
END COMPONENT;

component clk_wiz_v3_6
85    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
    );
90    end component;
```

95

```

-- declaracion de señales
-- relojes
    -- reloj usado en el sistema
signal reloj_sistema : std_logic := '0';
    -- salidas de pll
signal pll_50      : std_logic := '0';
signal pll_48      : std_logic := '0';
signal pll_40      : std_logic := '0';
signal pll_35      : std_logic := '0';
signal locked      : std_logic := '0';

100
105

-- cables de utilidad
signal enviar_datos : std_logic := '0';
signal slwr_sig     : std_logic := '0';
signal slrd_sig     : std_logic := '0';

110

-- cables necesarios para la memoria fifo
signal fifo_llena   : std_logic;
signal fifo_vacia   : std_logic;
signal wr_en        : std_logic;
signal rd_en        : std_logic;
signal valid         : std_logic;
signal dout          : std_logic_vector( ancho_bus-1 downto 0);
signal din           : std_logic_vector( ancho_bus-1 downto 0);

115
120
-- señales de sistema
    --init
signal reset       : std_logic := '0';

125
130
135
140
begin
    -- leds. Se conectan en la implementación física .
    -- NO COMENTAR!!!
    led(7 downto 0) <= (others => '0');

```

```
145     oaddr_y : ODDR2  -- buffer para reloj salida
      port map(
        D0  => '1',
        D1  => '0',
        CE  => '1',
150      C0  => pll_50 ,
        C1  => (not pll_50),
        R   => '0',
        S   => '0',
        Q   => reloj_sal
155    );
      --instanciaciones
      -- interfaz
      interface: fx2lp_interface PORT MAP(
160        reloj => reloj_sistema,
        reset => reset,
        fdata => fdata,
        faddr => faddr,
        slrd => slrd_sig,
165        slwr => slwr_sig,
        flaga => flaga,
        flagb => flagb,
        flagc => flagc,
        flagd => flagd,
170        sloe => sloe,
        pktend => pktend,
        enviar_datos => enviar_datos,
        dato_recibido => din,
        dato_a_enviar => dout
175    );
      --fifo
      fifo : fifo_generator_v9_3
      PORT MAP(
180        rst => not reset,
        clk => reloj_sistema,
        din => din,
        wr_en => wr_en,
        rd_en => rd_en,
185        dout => dout,
        full => fifo_llena,
        empty => fifo_vacia,
        valid => valid
      );
190    pll : clk_wiz_v3_6
```

```

port map(
    CLK_IN1  => reloj_ent ,
    CLK_OUT1 => pll_50 ,
    CLK_OUT2 => pll_48 ,
    CLK_OUT3 => pll_40 ,
    CLK_OUT4 => pll_35 ,
    RESET     => '0' ,
    LOCKED    => locked
);
env_sol_proc : process (reloj_sistema , slwr_sig , fifo_vacia)
begin
    if rising_edge(reloj_sistema) then
        if fifo_vacia = '0' then
            enviar_datos <= '1';
        else
            if slwr_sig = '1' then
                enviar_datos <= '0';
            else
                enviar_datos <= enviar_datos;
            end if;
        end if;
    end if;
end process env_dat_proc;

-- reloj
reloj_sistema <= pll_48;

-- conexiones de señales internas hacia el exterior
slwr <= slwr_sig;
slrd <= slrd_sig;

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

-- fifo señales
wr_en <= '1' when fifo_esc_act = dehab else '0';

rd_en <= '1' when fifo_lec_act = hab else '0';

-- maquina de estados de lectura fifo
leer_fifo_mef : process(fifo_lec_act , fifo_vacia , slwr_sig)
begin
    case fifo_lec_act is
        when inicio =>
            if slwr_sig = '0' then
                if fifo_vacia = '0' then
                    fifo_lec_prox <= hab;

```

```
240           else
241               fifo_lec_prox <= inicio;
242           end if;
243       else
244           fifo_lec_prox <= inicio;
245       end if;

250   when hab =>
251       fifo_lec_prox <= dehab;

255   when dehab =>
256       if slwr_sig = '1' then
257           fifo_lec_prox <= inicio;
258       end if;

259   when others =>
260       fifo_lec_prox <= inicio;
261   end case;
262 end process leer_fifo_mef;

263 -- maquina de estados escritura fifo
264 escr_fifo_mef: process(fifo_esc_act, slrd_sig, fifo_llena)
265 begin
266     case fifo_esc_act is
267         when inicio =>
268             if slrd_sig = '0' then
269                 if fifo_llena = '0' then
270                     fifo_esc_prox <= hab;
271                 else
272                     fifo_esc_prox <= inicio;
273                 end if;
274             else
275                 fifo_esc_prox <= inicio;
276             end if;
277         when hab =>
278             fifo_esc_prox <= dehab;

279         when dehab =>
280             if slrd_sig = '1' then
281                 fifo_esc_prox <= inicio;
282             end if;

283         when others =>
284             fifo_esc_prox <= inicio;
285     end case;
286 end process escr_fifo_mef;
```

```
reloj_mef_global: process (reloj_sistema , reset)
begin
  if reset = '0' then
    fifo_lec_act <= inicio;
    fifo_esc_act <= inicio;
  elsif rising_edge(reloj_sistema) then
    fifo_lec_act <= fifo_lec_prox;
    fifo_esc_act <= fifo_esc_prox;
  end if;
end process reloj_mef_global;

inic_rst: process(reloj_sistema)
begin
  if rst_cont /= 0 then
    if rising_edge(reloj_sistema) then
      rst_cont <= rst_cont - 1;
    end if;
  end if;
end process inic_rst;
end fx2lp_interface_arq;
```

top_tb.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.all;

5 ENTITY interface_test_bench IS
END interface_test_bench;

ARCHITECTURE behavior OF interface_test_bench IS

10   -- Component Declaration for the Unit Under Test (UUT)
COMPONENT fx2lp_interface_top
PORT(
  fdata : INOUT std_logic_vector(15 downto 0);
  faddr : OUT std_logic_vector(1 downto 0);
  15   slrd : OUT std_logic;
  slwr : OUT std_logic;
  flaga : IN std_logic;
  flagb : IN std_logic;
  flagc : IN std_logic;
  20   flagd : IN std_logic;
  sloe : OUT std_logic;
  pktend : OUT std_logic;
  reloj_ent : IN std_logic;
  reloj_sal : OUT std_logic;
  25   pulsador : IN std_logic;
  led : OUT std_logic_vector(7 downto 0)
```

```
 );
END COMPONENT;

30
--Inputs
signal flaga : std_logic := '0';
signal flagb : std_logic := '0';
signal flagc : std_logic := '0';
35 signal flagd : std_logic := '0';
signal reloj_ent : std_logic := '0';
signal pulsador : std_logic := '0';

--BiDirs
40 signal fdata : std_logic_vector(15 downto 0);

--Outputs
45 signal faddr : std_logic_vector(1 downto 0);
signal slrd : std_logic;
signal slwr : std_logic;
signal sloe : std_logic;
signal pktend : std_logic;
signal reloj_sal : std_logic;
signal led : std_logic_vector(7 downto 0);

50
-- Clock period definitions
constant periodo_reloj : time := 21 ns;

-- Data generator
55 signal contador: std_logic_vector(15 downto 0) := x"0000";

BEGIN
-- Instantiate the Unit Under Test (UUT)
60 uut: fx2lp_interface_top PORT MAP(
    fdata => fdata,
    faddr => faddr,
    slrd => slrd,
    slwr => slwr,
    flaga => flaga,
65    flagb => flagb,
    flagc => flagc,
    flagd => flagd,
    sloe => sloe,
    pktend => pktend,
70    reloj_ent => reloj_ent,
    reloj_sal => reloj_sal,
    pulsador => pulsador,
    led => led
);
```

```
75      -- Clock process definitions
ent_reloj_process :process
begin
    clk_in <= '0';
80    wait for periodo_reloj/2;
    clk_in <= '1';
    wait for periodo_reloj/2;
end process;

85      -- Start up reset
pulsador <= '0', '1' after 100 ns;

-- estímulos flags. todas activas en bajo
flags_proc: process
begin
90    flaga <= '1'; -- ep2full
    flagb <= '0'; -- ep8empty
    flagc <= '1'; -- ep8full
    flagd <= '0'; -- ep2empty
    wait for 23 us;

    flagb <= '1';
    wait until counter = x"0100";
    wait until falling_edge(clk_in);

100   flagb <= '0';
    wait until fdata = x"00f0";
    flaga <= '0';
    wait for 1 us;
    flaga <= '1';
    wait until pktend = '0';
    wait;
end process;

110   -- emulador de datos
data_proc: process(slr)
begin
    if pulsador = '0' then
        contador <= x"0000";
115    elsif rising_edge(slr)then
        contador <= contador + 1;
    end if;
end process;

120   with flagb select
        fdata <= contador when '1',(others => 'Z')when others;
END;
```

fx2lp_interface_top.ucf

```
NET "clk_in" TNMNET = "clk_in";
TIMESPEC TS_clk_in = PERIOD "clk_in" 50 MHz HIGH 50 %;

NET "clk_in" LOC = P56 | IOSTANDARD = LVTTL;
5 NET "button" LOC = P38 | IOSTANDARD = LVTTL;

NET "fdata[15]" LOC = P50 | IOSTANDARD = LVTTL;
NET "fdata[14]" LOC = P51 | IOSTANDARD = LVTTL;
NET "fdata[13]" LOC = P40 | IOSTANDARD = LVTTL;
10 NET "fdata[12]" LOC = P41 | IOSTANDARD = LVTTL;
NET "fdata[11]" LOC = P34 | IOSTANDARD = LVTTL;
NET "fdata[10]" LOC = P35 | IOSTANDARD = LVTTL;
NET "fdata[9]" LOC = P32 | IOSTANDARD = LVTTL;
NET "fdata[8]" LOC = P33 | IOSTANDARD = LVTTL;
15 NET "fdata[7]" LOC = P29 | IOSTANDARD = LVTTL;
NET "fdata[6]" LOC = P30 | IOSTANDARD = LVTTL;
NET "fdata[5]" LOC = P26 | IOSTANDARD = LVTTL;
NET "fdata[4]" LOC = P27 | IOSTANDARD = LVTTL;
NET "fdata[3]" LOC = P23 | IOSTANDARD = LVTTL;
20 NET "fdata[2]" LOC = P24 | IOSTANDARD = LVTTL;
NET "fdata[1]" LOC = P21 | IOSTANDARD = LVTTL;
NET "fdata[0]" LOC = P22 | IOSTANDARD = LVTTL;

NET "faddr[1]" LOC = P9 | IOSTANDARD = LVTTL;
25 NET "faddr[0]" LOC = P8 | IOSTANDARD = LVTTL;

NET "slrd" LOC = P16 | IOSTANDARD = LVTTL;
NET "slwr" LOC = P17 | IOSTANDARD = LVTTL;
NET "flaga" LOC = P12 | IOSTANDARD = LVTTL;
30 NET "flagb" LOC = P14 | IOSTANDARD = LVTTL;
NET "flagc" LOC = P15 | IOSTANDARD = LVTTL;
NET "flagd" LOC = P11 | IOSTANDARD = LVTTL;
NET "sloe" LOC = P6 | IOSTANDARD = LVTTL;

NET "pktend" LOC = P10 | IOSTANDARD = LVTTL;
35 NET "led[0]" LOC = P134 | IOSTANDARD = LVTTL;
NET "led[1]" LOC = P133 | IOSTANDARD = LVTTL;
NET "led[2]" LOC = P132 | IOSTANDARD = LVTTL;
NET "led[3]" LOC = P131 | IOSTANDARD = LVTTL;
40 NET "led[4]" LOC = P127 | IOSTANDARD = LVTTL;
NET "led[5]" LOC = P126 | IOSTANDARD = LVTTL;
NET "led[6]" LOC = P124 | IOSTANDARD = LVTTL;
NET "led[7]" LOC = P123 | IOSTANDARD = LVTTL;
```

Sumario de síntesis

fx2lp_interface_top Project Status (04/23/2020 - 15:52:42)			
Project File:	Spartan6.xise	Parser Errors:	No Errors
Module Name:	fx2lp_interface_top	Implementation State:	Placed and Routed
Target Device:	xc6slx9-2tqg144	Errors:	
Product Version:	ISE 14.7	Warnings:	
Design Goal:	Balanced	Routing Results:	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	Timing Constraints:	All Constraints Met
Environment:	System Settings	Final Timing Score:	0 (Timing Report)

Device Utilization Summary				
Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	83	11,440	1 %	
Number used as Flip Flops	79			
Number used as Latches	4			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	80	5,720	1 %	
Number used as logic	79	5,720	1 %	
Number using O6 output only	35			
Number using O5 output only	1			
Number using O5 and O6	43			
Number used as ROM	0			
Number used as Memory	0	1,440	0 %	
Number used exclusively as route-thrus	1			
Number with same-slice register load	1			
Number with same-slice carry load	0			
Number with other load	0			
Number of occupied Slices	39	1,430	2 %	
Number of MUXCYs used	44	2,860	1 %	
Number of LUT Flip Flop pairs used	106			
Number with an unused Flip Flop	35	106	33 %	
Number with an unused LUT	26	106	24 %	

Continúa en la siguiente página

Device Utilization Summary (cont.)				
Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of fully used LUT-FF pairs	45	106	42 %	
Number of unique control sets	14			
Number of slice register sites lost to control set restrictions	77	11,440	1 %	
Number of bonded IOBs	37	102	36 %	
Number of LOCed IOBs	36	37	97 %	
IOB Flip Flops	1			
IOB Latches	16			
Number of RAMB16BWERS	1	32	3 %	
Number of RAMB8BWERS	0	64	0 %	
Number of BUFINO2/BUFINO2_2CLKs	1	32	3 %	
Number used as BUFINO2s	1			
Number used as BUFINO2_2CLKs	0			
Number of BUFINO2FB/BUFINO2FB_2CLKs	1	32	3 %	
Number used as BUFINO2FBs	1			
Number used as BUFINO2FB_2CLKs	0			
Number of BUFG/BUFGMUXs	3	16	18 %	
Number used as BUFGs	3			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	4	0 %	
Number of ILOGIC2/ISERDES2s	16	200	8 %	
Number used as ILOGIC2s	16			
Number used as ISERDES2s	0			
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	200	0 %	
Number of OLOGIC2/OSERDES2s	1	200	1 %	
Number used as OLOGIC2s	1			
Number used as OSERDES2s	0			
Number of BSCANs	0	4	0 %	
Number of BUFHs	0	128	0 %	
Number of BUFPLLs	0	8	0 %	
Number of BUFPLL_MCBs	0	4	0 %	

Continúa en la siguiente página

B. Códigos de la síntesis en FPGA

Device Utilization Summary (cont.)					
Slice Logic Utilization		Used	Available	Utilization	Note(s)
Number of DSP48A1s		0	16	0 %	
Number of ICAPs		0	1	0 %	
Number of MCBs		0	2	0 %	
Number of PCILOGICSEs		0	2	0 %	
Number of PLL_ADVs		1	2	50 %	
Number of PMVs		0	1	0 %	
Number of STARTUPs		0	1	0 %	
Number of SUSPEND_SYNCs		0	1	0 %	
Average Fanout of Non-Clock Nets		2.91			

Detailed Reports					
Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report	Current	jue 23. abr 15:51:44 2020	0	26 Warnings (0 new)	14 Infos (10 new)
Translation Report	Current	jue 23. abr 15:51:53 2020	0	0	2 Infos (0 new)
Map Report	Current	jue 23. abr 15:52:24 2020			
Place and Route Report	Current	jue 23. abr 15:52:34 2020	0	6 Warnings (0 new)	0
Power Report					
Post-PAR Static Timing Report	Current	jue 23. abr 15:52:40 2020	0	0	3 Infos (0 new)
Bitgen Report	Out of Date	mié 1. abr 14:38:01 2020	0	4 Warnings (0 new)	1 Info (0 new)

Date Generated: 05/31/2020 - 15:05:40

Apéndice C

Códigos para PC del programa de pruebas

El presente apéndice contiene los códigos escritos en lenguaje C++, los cuales fueron desarrollados durante el transcurso de trabajo aquí informado con el objetivo de obtener un programa que transmita datos a través del puerto USB. Los archivos exhibidos en este apéndice son:

- **tfUSBCheck.h:** Encabezado utilizado para la declaración de clases y funciones, como así también la definición de códigos de variables utilizadas en la ejecución del programa de pruebas
- **tfUSBCheck.cpp:** Archivo de implementación del programa de pruebas, en donde se define la función `main()` y las declaradas en el encabezado.

tfUSBCheck.h

```
#ifndef TFUSBCHECK_H_
#define TFUSBCHECK_H_

#include <iostream>
#include <fstream>
#include <libusb-1.0/libusb.h>
#include <cstdio>
#include <csignal>
#include <cstdlib>
#include <vector>
#include <ctime>

//errores: 9 es error, xx es el identificador del error
#define NO_DOI      901 //Dispositivo de interes no encontrado
#define NOPROGR    902 //El dispositivo no se encuentra programado
#define DRIVER_BLOCKED 903 //El driver no pudo ser desconectado
#define NO_LIBUSB   904 //LIBUSB no está funcionando
```

```
// datos de Cypress
20 #define CY_VID          0x04b4
#define CY_FX2LP_PID      0x1003
#define OUT_EP             (LIBUSB_ENDPOINT_OUT | 8)
#define IN_EP              (LIBUSB_ENDPOINT_IN | 2)
#define INTERFACE          0

25 // ASCII
#define ESC                0x1B

#define MAX_OUT_DATA       256
30 #define MAX_IN_DATA      512

using namespace std;

typedef struct{
35     char row;
     char block;
     char check;
} err;

40 class USBChecker
{
private:
    struct libusb_context* ctx;
    struct libusb_device* dev;
45    struct libusb_device_descriptor *dscr;
    struct libusb_device_handle* handler;
    struct libusb_transfer *transfer_out;
    struct libusb_transfer *transfer_in;

50 public:
    USBChecker(void);
    USBChecker(libusb_context*, libusb_device*, libusb_device_handle*);
    ~USBChecker(void){}

55 void init_myusb_device(void) throw(int);
void get_usb_information(void);
void closeChecker(void);

60 void dataGenerator(unsigned char* );
void dataChecker(unsigned char*, vector<err>*);

void sendData(unsigned char* );
65 void receiveData(unsigned char* );
```

```
void static receive_cb(struct libusb_transfer *);  
void static send_cb(struct libusb_transfer *);  
  
70 };  
  
void static close_myusb_device(int ignored);  
75 void err_timestamp(void);  
void out_timestamp(void);  
  
#endif //TFUSBCHECK_H
```

tfUSBCheck.cpp

```
//=====  
// Name : tfUSBCheck.cpp  
// Author : Edwin Barragan  
//=====  
  
5 #include "tfUSBCheck.h"  
  
bool to_exit = 1;  
bool to_send = 1;  
10 bool to_receive = 0;  
bool to_check = 0;  
bool to_log = 0;  
int actual_length = 1;  
  
15 USBChecker::USBChecker(void){  
    ctx = NULL;  
    dev = NULL;  
    dscr = NULL;  
    handler = NULL;  
20    transfer_in = NULL;  
    transfer_out = NULL;  
}  
  
void USBChecker::closeChecker(void){  
25    if(transfer_out)  
        libusb_cancel_transfer(transfer_out);  
    if(transfer_in)  
        libusb_cancel_transfer(transfer_in);  
    libusb_release_interface(handler, INTERFACE);  
30}  
  
USBChecker::USBChecker(libusb_context* context, libusb_device* device  
                      , libusb_device_handle * hand){  
    ctx = context;
```

```

35     dev = device;
      handler = hand;
      dscr = NULL;
      transfer_in = NULL;
      transfer_out = NULL;
40 }
void USBChecker::init_myusb_device(void) throw(int){

    int is_not_ok;
45    libusb_device **list;
    ssize_t cont;
    ssize_t dev_idx;

    //inicialización del usb
    // inicialización de la biblioteca llamando a la función libusb_init.
50    // Esto crea una sesión
    is_not_ok = libusb_init(&ctx);
    if(is_not_ok)
    {
        err_timestamp();
        cerr << "No funciona libusb" << endl;
        throw(NOLIBUSB);
    }

55    //todos los msjs serán mostrados en el archivo de registro de eventos
    libusb_set_debug(ctx, LIBUSB_LOGLEVEL_DEBUG);

    // Llamar a la función libusb_get_device_list para obtener una lista
    // de los dispositivos conectados
60    cont = libusb_get_device_list(ctx, &list);

    // Obtener los descriptores de los dispositivos
    for(dev_idx = 0; dev_idx < cont; dev_idx++)
    {
        libusb_device *aux_dev = list[dev_idx];
        libusb_device_descriptor aux_dscr;

65        is_not_ok = libusb_get_device_descriptor(aux_dev, &aux_dscr);

        if(is_not_ok)
        {
            err_timestamp();
            cerr << "No pudo acceder a un descriptor" << endl;
        }

70        // Buscar el dispositivo desarrollado
        if(aux_dscr.idVendor == CY_VID)

```

```

    {
        if( aux_dscr . idProduct == CY_FX2LP_PID)
    {
        dev = aux_dev;
        dscr = &aux_dscr;
    }
    else
    {
        err_timestamp();
        cerr << "El dispositivo no se encuentra programado" << endl;
        throw(NO_PROGR);
    }
}
if( dev == NULL)
{
    cerr << "No se encuentra el dispositivo" << endl;
    throw(NO_DOI);
}

// alcanzar acceso al dispositivo
handler = libusb_open_device_with_vid_pid(ctx, CY_VID, CY_FX2LP_PID);

// liberar la lista obtenida
libusb_free_device_list(list, 1);

// seleccionar la interfaz a utilizar.
// primero se debe desactivar el driver (solo en linux)
is_not_ok = libusb_set_auto_detach_kernel_driver(handler, 1);
if(is_not_ok)
{
    cerr << "detached failed" << endl;
    throw(DRIVER_BLOCKED);
}
libusb_claim_interface(handler, INTERFACE);
libusb_set_interface_alt_setting(handler,0,0);
//para debug
out_timestamp();
cout << "Tengo control sobre la placa" << endl;
}

void close_myusb_device(int ignored){
    cout << "Sali con la señal" << ignored << endl;
    to_exit = 0;
}

```

```
void USBChecker::dataGenerator(unsigned char* data){  
    char number;  
    unsigned char block = 0, bit;  
    uint row;  
    uint i;  
    char rowParity = 0, colParity = 0;  
    const char mask = 0x01;  
  
    while(block <= MAX_OUT_DATA / 8)  
    {  
        i = block * 8;  
        colParity = 0;  
        for(row = i; row < i + 7; row++)  
        {  
            rowParity = 0;  
            number = char(rand() & 0x7f);  
            data[row] = number << 1;  
            for(bit = 0; bit < 7; bit++)  
            {  
                rowParity = rowParity ^ (number & mask);  
                number = number >> 1;  
            }  
            if(rowParity)  
                data[row] = data[row] | 0x01;  
            else  
                data[row] = data[row] & 0xF7;  
            colParity = colParity ^ data[row];  
        }  
        data[row] = colParity;  
        block++;  
    }  
}  
  
void USBChecker::dataChecker(unsigned char* data, vector<err> *errores){  
    unsigned char block = 0, bit;  
    uint row;  
    char mask = 0x01;  
  
    cout << "Estoy_en_el_checker" << endl;  
    while(block < actual_length/8)  
    {  
        err error;  
        error.block = 0;  
        error.row = 0;  
        error.check = 0;  
        uint i = block * 8;  
        char colParity = 0;
```

```

180      for(row = i; row < i + 8; row++)
181      {
182          char rowParity = 0;
183          char aux_data = data[row];
184          for(bit = 0; bit < 8; bit++)
185          {
186              rowParity = rowParity ^ (aux_data & mask);
187              aux_data >>= 1;
188          }
189          if (!rowParity)
190          {
191              error.row = row;
192              errores->push_back(error);
193          }
194          colParity ^= data[row];
195      }
196      if (!colParity)
197      {
198          error.block = block;
199          error.check = colParity;
200          errores->push_back(error);
201      }
202      to_log = 1;
203      cout << "sali del Checker" << endl;
204  }
205
void USBChecker::sendData(unsigned char* data){
206     transfer_out = libusb_alloc_transfer(0);
207     libusb_fill_bulk_transfer(transfer_out, handler, OUT_EP, data, MAX_OUT_DATA
208         , send_cb, NULL, 0);
209     int c = libusb_submit_transfer(transfer_out);
210     out_timestamp();
211     cout << "Mandé datos" << c << endl;
212 }
213
void USBChecker::send_cb(struct libusb_transfer* transfer){
214     out_timestamp();
215     cout << "Estoy en el callback de envío" << endl;
216     cout << transfer->status << endl;
217     switch(transfer->status)
218     {
219         case LIBUSB_TRANSFER_COMPLETED:
220             to_receive = 1;
221             break;
222         case LIBUSB_TRANSFER_ERROR:
223             break;

```

```

        case LIBUSB_TRANSFER_TIMED_OUT:
            err_timestamp();
            cerr << "El dispositivo no responde." << endl;
            to_exit = 0;
            //timers libres. no ocurriría
            break;
230    case LIBUSB_TRANSFER_CANCELLED:
            //no programado para que ocurra
            break;
235    case LIBUSB_TRANSFER_STALL:
            //no programado para que ocurra
            break;
        case LIBUSB_TRANSFER_NO_DEVICE:
            //si se desconecta el dispositivo
            err_timestamp();
            cerr << "El dispositivo se desconectó. Se cierra"
            << "el programa." << endl;
            to_exit = 0;
            return;
240
245    case LIBUSB_TRANSFER_OVERFLOW:
            err_timestamp();
            cerr << "Fallo por overflow. Revisar código de"
            << "generacion de datos. Se cierra programa." << endl;
            to_exit = 0;
            break;
250    default:
            break;
        }
255    libusb_free_transfer(transfer);
    }

void USBChecker::receiveData(unsigned char* data){
    int pktSize;
    int isoPkts = 1;
260    transfer_in = libusb_alloc_transfer(isoPkts);
    pktSize = libusb_get_max_iso_packet_size(dev, IN_EP);
    libusb_fill_iso_transfer(transfer_in, handler, IN_EP, data, pktSize
        ,isoPkts, receive_cb, NULL, 0);
    libusb_set_iso_packet_lengths(transfer_in, pktSize);
    int c = libusb_submit_transfer(transfer_in);
    out_timestamp();
    cout << "Intento recibir datos" << c << endl;
    cout << "Tengo" << pktSize << "como tamaño de paquete" << endl;
265
270    }

void USBChecker::receive_cb(struct libusb_transfer* transfer){
    out_timestamp();
    cout << "Estoy en el callback de recepcion" << endl;

```

```

275     cout << transfer->status << endl;
      switch( transfer->status )
      {
        case LIBUSB_TRANSFER_COMPLETED:
          actual_length = transfer->actual_length ;
          to_check = 1;
          libusb_free_transfer( transfer );
          break;
        case LIBUSB_TRANSFER_ERROR:
          break;
        case LIBUSB_TRANSFER_TIMED_OUT:
          err_timestamp();
          cerr << "El dispositivo no responde." << endl;
          to_exit = 0;
          //timers libres. no ocurriría
          break;
        case LIBUSB_TRANSFER_CANCELLED:
          //no programado para que ocurra
          break;
        case LIBUSB_TRANSFER_STALL:
          //no programado para que ocurra
          break;
        case LIBUSB_TRANSFER_NO_DEVICE:
          //si se desconecta el dispositivo
          err_timestamp();
          cerr << "El dispositivo se desconectó. Se cierra"
              << "el programa." << endl;
          to_exit = 0;
          return;
        case LIBUSB_TRANSFER_OVERFLOW:
          err_timestamp();
          cerr << "Fallo por overflow. Revisar código de generacion de"
              << "datos. Se cierra el programa." << endl;
          to_exit = 0;
          break;
        default:
          break;
      }
}

315 void out_timestamp(void){
    time_t timer = time(NULL);
    tm *loctimer = localtime(&timer);
    cout << "_____ " << endl;
    cout << loctimer->tm_year << " ";
    cout << loctimer->tm_mon << " ";
    cout << loctimer->tm_mday << " ";
    cout << loctimer->tm_hour << ":";
```

```

cout << loctimer->tm_min << ":" ;
cout << loctimer->tm_sec << endl;
325   cout << "-----" << endl;
}

void err_timestamp(void){
    time_t timer = time(NULL);
330   tm* loctimer = localtime(&timer);
    cerr << loctimer->tm_year << "_" ;
    cerr << loctimer->tm_mon << "_" ;
    cerr << loctimer->tm_mday << "_" ;
    cerr << loctimer->tm_hour << ":" ;
335   cerr << loctimer->tm_min << ":" ;
    cerr << loctimer->tm_sec << '\t';
}

int main() {
340   time_t pre_timer;
   time_t pos_timer;

   struct timeval tv;
   libusb_context* ctx = NULL;
345   libusb_device* dev = NULL;
   libusb_device_handle* handler = NULL;

   static unsigned char buffer_in [MAX_IN_DATA];
   static unsigned char buffer_out [MAX_OUT_DATA];
350   vector<err> errs;

   struct sigaction sigIntHandler;

   sigIntHandler.sa_handler = close_myusb_device;
355   sigemptyset(&sigIntHandler.sa_mask);
   sigIntHandler.sa_flags = 0;

   sigaction(SIGKILL, &sigIntHandler, NULL);
   sigaction(SIGTERM, &sigIntHandler, NULL);
360   sigaction(SIGSTOP, &sigIntHandler, NULL);
   sigaction(SIGQUIT, &sigIntHandler, NULL);

   // redirijo el stderr a un archivo de registro de eventos
   // la función freopen reinicia tbn cerr para que vaya al
365   // mismo archivo
   freopen("errlog","a", stderr);

   freopen("datalog","a", stdout);

370   USBChecker checker(ctx, dev, handler);

```

```

// iniciar el dispositivo
try
{
    checker.init_myusb_device();
}
catch(int e)
{
    err_timestamp();
    switch(e)
    {
        case NO_DOI:
            cerr << "No se encontró dispositivo. Conecte la placa"
                << "Cypress" << endl;
            break;
        case NO_PROGR:
            cerr << "La placa Cypress no se encuentra programada" << endl;
            break;
        case DRIVER_BLOCKED:
            cerr << "No se puede alcanzar control de la placa" << endl;
            break;
        case NO_LIBUSB:
            cerr << "Nada que hacer. Final del programa" << endl;
            return(904);
    }
    to_exit = 0;
}
// iniciado
if(to_exit)
{
    while(actual_length != 0)
    {
        checker.receiveData(buffer_out);
        tv.tv_sec = 0;
        tv.tv_usec = 10000;
        libusb_handle_events_timeout_completed(ctx,&tv,NULL);
    }
    to_check = 0;

//ASYNC Transfer
while(to_exit)
{
    if(to_send) //send_flag;
    {
        cout << "Entre aquí" << endl;
        checker.dataGenerator(buffer_out);
        checker.sendData(buffer_out);
        to_send = 0;
    }
    if(to_receive)

```

```

        {
420         checker.receiveData( buffer_in );
         to_receive = 0;
    }
    if( to_check )
    {
425         checker.dataChecker( buffer_in , &errs );
         to_check = 0;
    }
    if( to_log )
    {
430         out_timestamp();
         cout << "OUT_TRANSFER:" << endl;
         for( uint i = 0; i < sizeof( buffer_out ); i++ )
        {
            if( i %16 != 15 )
                cout <<(int) buffer_out [ i ] << '\t';
            else
                cout << (int) buffer_out [ i ] << endl;
        }
         cout << "IN_TRANSFER:" << endl;
         for( uint i = 0; i < sizeof( buffer_in ); i++ )
440        {
            if( i %16 != 15 )
                cout <<(int) buffer_in [ i ] << '\t';
            else
                cout <<(int) buffer_in [ i ] << endl;
        }
        if( errs.size() == 0 )
            cout << "No_hubo_errores" << endl;
        else
450        {
            cout << "Errores:" << endl;
            uint j = 0;
            for( uint i = 0; i < errs.size(); i++ )
            {
                if( errs.at( i ).row != 0 )
455                {
                    cout << "En_fila:_" << errs.at( i ).row << endl;
                    j++;
                }
                else
460                    cout << "E_bloque:_" << errs.at( i ).block
                        << ";" cols_parity_found:_" << errs.at( i ).check << endl;
                    cout << endl;
            }
465            cout << "Tasa_de_error=_" << errs.size() / sizeof( buffer_out )
                *100 << endl;
    }
}

```

```
        }
        to_send = 1;
        cout << "le_dije_que_haga_mas_envio_de_datos" << endl;
470      to_log = 0;
        to_exit = 1;
    }
    tv.tv_sec = 0;
    tv.tv_usec = 10000;
475    libusb_handle_events_timeout_completed(ctx,&tv,NULL);
}

// liberar recursos
checker.closeChecker();

480 //cierro el dispositivo
libusb_close(handler);

// cierro el contexto y libero la memoria
485 libusb_exit(ctx);

fclose(stderr); // libero stderr
fclose(stdout); // libero stdout

490 return 0;
}
```


D. Esquemático de la Placa de Interconexión

Apéndice D

Esquemático de la Placa de Interconexión

