



UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA DE INGENIERÍA INDUSTRIAL
DE TOLEDO

TRABAJO FIN DE GRADO N° 19-B-225530

ANALIZADOR USB

Autor:
Mario Rubio Asensio

Director:
Francisco Moya Fernández



JUNIO DE 2019

Mario Rubio Asensio

Toledo – Spain

E-mail: mario.rubio2@alu.uclm.es

© 2019 Mario Rubio Asensio

Esta obra está bajo una [Licencia Creative Commons Atribución 4.0 Internacional](#).

Muchos de los nombres usados por las compañías para diferenciar sus productos y servicios son reclamados como marcas registradas. Allí donde estos nombres aparezcan en este documento, y cuando el autor haya sido informado de esas marcas registradas, los nombres estarán escritos en mayúsculas o como nombres propios.

Agradecimientos

Redactar agradecimientos

En primer lugar, quisiera agradecer al director del presente Trabajo Fin de Grado, D. Francisco Moya Fernández, por brindarme la oportunidad de realizar este proyecto y por la ayuda ofrecida.

A mis padres y hermana, que siempre han estado y estarán para apoyarme en todo momento.

A mis compañeros

Me gustaría también agradecer a los profesores de la Escuela de Ingeniería Industrial de Toledo ...

Compañeros ...

Resumen

El objetivo del presente Trabajo Fin de Grado, consiste en el desarrollo e implementación de un sistema *hardware* capaz de obtener, y posteriormente almacenar, tramas de bus *USB* (del inglés, *Universal Serial Bus*, o en español, Bus Serie Universal). Se busca a su vez, que dicho sistema sea lo más económico posible, utilizando únicamente herramientas y utilidades de código libre.

Previamente a su elaboración, se ha realizado un estudio del estado del arte en cuanto a sistemas de captación y análisis tanto *hardware* como *software*, observando el elevado precio de estos, reafirmando la necesidad de disponer un sistema de bajo coste.

Para su desarrollo se ha utilizado en primer lugar una placa de evaluación que incluye el circuito integrado especializado *USB3300*, encargado de la propia capa física del bus USB. Esta placa también incluye dos conectores hembra USB, uno de tipo A y el otro de tipo mini-B, a los cuales se conectan ambos extremo del bus a analizar.

Seguidamente, se ha usado la placa de desarrollo *iCEstick*, que entre otros componentes, incluye una *FPGA iCE40HX1k* de la compañía *Lattice* y un conversor de USB a puerto serie *FTDI FT2232HL*. Dicha placa, tiene la función de comunicarse tanto con el integrado anterior encargado de la capa física, como con el equipo de control, siendo este un PC.

A partir de este punto, el desarrollo del sistema se ramifica en dos grupos diferenciados, el primero encargado de todo lo referente a la FPGA, y el segundo encargado del software de control. Para su realización se sigue un procedimiento ágil basado en *Scrum*, que lo divide en iteraciones de varias semanas denominadas *sprints*.

Para el primer grupo, se define previamente una serie de metodologías con las que poder desarrollar de forma efectiva, limpia y segura los diversos módulos necesarios. De esta forma, se consigue programar en lenguaje descriptor de *hardware* Verilog varios módulos con funcionalidades muy diversas, entre los que se encuentra:

- **Módulo de comunicación serie.** Cuya función es la de procesar y generar señales, de entrada y salida respectivamente, que permitan una comunicación serie con el PC.
- **Módulo de comunicación ULPI.** Cuya función, igual que en el caso anterior, es la de procesar y generar señales, pero en este caso para el bus ULPI generado por el integrado *USB3300*, tal como dicta su especificación. Gracias a este módulo, el sistema es capaz por un lado de escribir y leer registros, y por otro, de obtener los datos USB.
- **Módulo de almacenaje FIFO.** Cuya función es la de generar una memoria *FIFO* (*First-In First-Out*) a partir de los bloques de RAM internos disponibles en la FPGA. La peculiaridad de esta memoria es la forma en la que almacena y vuelca los datos, siendo el primer dato en ser almacenado el primero en ser devuelto.
- **Módulo de control general.** Cuya función es la unir y gestionar todos los módulos implicados, permitiendo un funcionamiento sincronizado entre todos ellos.

Todos ellos, han soportado varias simulaciones con las que asegurar su correcto funcionamiento antes de su despliegue en el sistema final.

Junto con los anteriores módulo, se desarrolla un protocolo simple con el que poder comandar el sistema. Este protocolo esta formado por 2 bytes, en los que se incluyen el comando a realizar, una dirección de 6 bits, y un byte de datos.

Hay que comentar también el diseño e impresión de una pequeña base en 3D, donde situar todos los componentes de forma segura.

El segundo grupo, aun teniendo una menor dificultad respecto al primero, es esencial para poder ver los resultados obtenidos. Para controlar la totalidad del sistema se ha creado una aplicación en lenguaje C, que por medio de un simple menú permita la interacción entre la FPGA y el usuario. Este menú permite por un lado configurar y abrir el puerto al que está conectado la FPGA, y por otro, permite enviar los comandos necesarios para hacer funcionar al sistema, siendo estos de escritura de registros, lectura de registros y activación/desactivación del envío de datos al PC.

Esta aplicación también se encarga de almacenar de forma estructurada dichos datos en un archivo formato *JSON*, que posteriormente puede ser estudiado. Secundariamente, y tras la finalización de todo lo anterior, se ha creado una pequeña utilidad, que convierte el fichero *JSON* a *PCAP*, lo que permite ser visualizado en el programa *Wireshark*.

Hay que destacar que todo el sistema en su conjunto ha costado aproximadamente menos de 45\$ (aproximadamente 40€), precio muy por debajo de cualquier otro método de captura comercial.

Para finalizar, hay que comentar varios aspectos a incluir o mejorar.

- Mejorar el método de transmisión hacia el PC, para aumentar su velocidad de transferencia, y evitar posibles perdidas de datos. Por ejemplo, se puede prescindir del puerto serie, y utilizar otro protocolo soportado por el integrado FTDI, que permita una mayor tasa de transferencia.
- Incorporar memoria RAM externa, que aumente significativamente la cantidad de datos que puede almacenar temporalmente el sistema.
- Crear un disector para *Wireshark*, que sea capaz de dar más información sobre los datos capturados.

Abstract

The main purpose of this Final Degree Project is the development and implementation of a hardware base system capable of capture and store a USB (Universal Serial Bus) data frame. Furthermore, this system has to be as affordable as possible, using only open source tools and utilities.

Previously to the elaboration of the project, a study on the current State of the Art of hardware and software based capture and analysis systems has been made. Noting the high price of them, the necessity of having an inexpensive system were reaffirmed.

To obtain the desired result, a evaluation board containing the USB3300 specialised integrated circuit has been used, that integrated circuit handles the Physical layer of the bus, reducing the complexity level of the final design. This board also includes two USB female connectors, one of type A and the other one of type mini-B, where both ends of the USB bus to analyze are connected. It is worth mentioning that this integrated circuit uses the parallel protocol ULPI (UTMI+ Low Pin Interface) to interface with other devices.

The iCEstick development board, that includes the FPGA (Field-Programmable Gate Array) iCE40HX1K from the Lattice Semiconductor manufacturer and a the FTDI FT2232HL USB to Serial converter integrated circuit, is then use to communicate with both the USB3300 and the control equipment, in this case a PC.

Terminar!!

Índice general

Índice de tablas	v
Índice de figuras	vii
1. Introducción	1
1.1. Organización de la memoria	1
1.2. Repositorio de información	2
2. Motivación y antecedentes	3
2.1. Sistemas <i>software</i> actuales	4
2.1.1. Licencia de tipo <i>shareware</i>	4
2.1.2. Licencia de código libre	6
2.2. Sistemas <i>hardware</i> actuales	7
2.3. Comparativa de sistemas <i>software</i>	11
2.4. Comparativa de sistemas <i>hardware</i>	11
3. Objetivos	13
3.1. Selección de componentes <i>hardware</i>	13
3.2. Implementación de memoria <i>FIFO</i>	13
3.3. Implementación de un sistema de comunicación serie	14
3.3.1. Diseño de un módulo de emisión serie	14
3.3.2. Diseño de un módulo de recepción serie	15
3.4. Implementación de un sistema que procese el protocolo ULPI	15
3.4.1. Diseño de un módulo de escritura de registros ULPI	15
3.4.2. Diseño de un módulo de lectura de registros ULPI	15
3.4.3. Diseño de un módulo de captación USB	16
3.5. Implementación de un sistema que gestione el envío de datos	16
3.6. Implementación de un sistema que gestione los comandos entrantes	16
3.7. Creación de una aplicación de control	16
4. Contribuciones	19
4.1. Etapas de diseño de un módulo	19
4.2. Pruebas y simulaciones	23

4.3. Herramientas de utilizadas	23
4.4. Automatización de tareas	24
4.5. Distribución de archivos	25
5. Procedimiento	27
5.1. Diferencias con Scrum	27
5.1.1. Roles	27
5.1.2. Historias de usuario	28
5.1.3. Planificación de <i>sprints</i>	28
5.1.4. Flujo de trabajo	28
5.1.5. Herramientas de ayuda	29
5.2. Aprendizaje de funcionamiento y utilización de FPGAs	29
5.3. Uso de <i>software</i> libre	29
5.4. Problemas encontrados durante la elaboración del proyecto	30
5.4.1. Fallos intrínsecos	30
5.4.2. Fallos extrínsecos	30
6. Resultados <i>hardware</i>	33
6.1. Componentes utilizados	33
6.2. Precio final	36
6.3. Módulos integrados en la FPGA	37
6.3.1. Módulo divisor de reloj (<i>clk_div</i>)	37
6.3.2. Módulo de generación de reloj de baudios (<i>clk_baud_pulse</i>)	38
6.3.3. Módulo de memoria FIFO (<i>FIFO_BRAM_SYNC</i>)	38
6.3.4. Módulo de registro de desplazamiento universal (<i>shift_register</i>)	39
6.3.5. Módulo de comunicación serie (<i>UART</i>)	40
6.3.6. Módulo de comunicación ULPI	41
6.3.7. Módulo de procesado y almacenaje de comandos entrantes (<i>ULPI_op</i>)	45
6.3.8. Módulos de control de botonera (<i>btn_debouncer</i> y <i>signal_trigger</i>)	45
6.3.9. Módulo de control maestro (<i>main_controller</i>)	46
6.4. Simulaciones finales	46
7. Resultados <i>software</i>	49
7.1. Aplicaciones de apoyo	49
7.2. Información de la aplicación	50
7.3. Información del archivo generado	53
8. Conclusiones	55
8.1. Conclusiones	55
8.2. Trabajos futuros	56
9. Bibliografía	57

A. Diagramas internos de la <i>FPGA</i>	59
B. Máquinas de estados <i>Mealy</i> de los módulos diseñados	69
C. Resultado de la simulación final	79
D. Manual de instalación y utilización	85
D.1. Requisitos <i>software</i> previos	85
D.2. Descarga del repositorio	86
D.3. Generación y programación del archivo binario	87
D.4. Instalación de las aplicaciones de control	87
D.5. Obtención de la captura	87
D.6. Conversión de la captura a formato <i>PCAP</i>	88

Índice de tablas

2.1.	Componentes y precio de la propuesta de analizador USB de <i>Ultra-Embedded</i>	10
2.2.	Comparativa de sistemas <i>software</i>	11
2.3.	Comparativa de sistemas <i>hardware</i>	12
6.1.	Presupuesto del sistema	36
6.2.	Información de los <i>bytes</i> enviados a la <i>FPGA</i>	45

Índice de figuras

2.1.	Esquema de captura por medio de <i>software</i>	3
2.2.	Esquema de captura por medio de <i>hardware</i>	3
2.3.	Interfaces gráficas de los analizadores <i>software</i> de tipo <i>shareware</i> . Imágenes extraídas de las páginas web de los desarrolladores.	6
2.4.	Interfaz gráfica del analizador de paquetes <i>Wireshark</i>	7
2.5.	<i>Ellisys USB Explorer 200</i> . Imagen extraída de la página web del fabricante.	8
2.6.	<i>Teledyne LeCroy Mercury T2</i> . Imagen extraída de la página web del fabricante.	9
2.7.	Productos de <i>Total Phase</i> . Imágenes extraídas de la página web del fabricante.	10
2.8.	<i>USB Sniffer</i> . Imagen extraída de la página web del proyecto.	10
3.1.	Esquema de funcionamiento de una memoria <i>FIFO</i>	14
3.2.	Señal típica de una comunicación serie.	14
3.3.	Trama de escritura de registros ULPI.	15
3.4.	Trama de lectura de registros ULPI.	16
3.5.	Trama ULPI de recepción de datos USB.	16
4.1.	Ejemplo del resultado de una simulación en <i>GTKWave</i>	23
4.2.	Interfaz gráfica de la herramienta <i>Nextpnr</i>	24
4.3.	Estructura de carpetas del código de la FPGA	25
6.1.	Esquema de analizadores <i>hardware</i>	33
6.2.	Sistema de captura final	34
6.3.	Placa de desarrollo <i>IceStick</i>	34
6.4.	PCB con el integrado <i>USB3300</i>	35
6.5.	Esquema de los botones auxiliares	35
6.6.	Resto de elementos usados en el sistema de captación	36
6.7.	Divisor de reloj con <i>Flip-Flops</i>	37
6.8.	Ejemplo de funcionamiento del registro de desplazamiento hacia la derecha.	39
6.9.	Diagrama de funcionamiento del submódulo de recepción serie (<i>UART_Rx</i>)	41
6.10.	Diagrama de funcionamiento del submódulo de emisión serie (<i>UART_Tx</i>)	41
6.11.	Diagrama de funcionamiento de la unión de módulos ULPI	43
6.12.	Diagrama de funcionamiento del submódulo de escritura de registros ULPI (<i>ULPI_REG_WRITE</i>)	43

6.13. Diagrama de funcionamiento del submódulo de lectura de registros ULPI (<i>ULPI_REG_READ</i>)	44
6.14. Diagrama de funcionamiento del submódulo de recepción USB (<i>ULPI_RECV</i>)	44
6.15. Esquema de funcionamiento del módulo <i>btn_debouncer</i>	46
7.1. Ejemplo de escritura de registro y su posterior lectura usando el menú de la aplicación	51
7.2. Diagrama de funcionamiento de la aplicación	52
7.3. Estructura del archivo <i>JSON</i> donde se almacena la captura	53
A.1. Esquema principal usado en la <i>FPGA</i>	60
A.2. Esquema principal del módulo de comunicación serie.	61
A.3. Esquema del submódulo de transmisión serie.	62
A.4. Esquema del submódulo de recepción serie.	63
A.5. Esquema principal del módulo de comunicación <i>ULPI</i>	64
A.6. Esquema del submódulo de escritura de registros <i>ULPI</i>	65
A.7. Esquema del submódulo de lectura de registros <i>ULPI</i>	66
A.8. Esquema del submódulo de recepción de datos <i>USB</i> a través de <i>ULPI</i>	67
A.9. Esquema del módulo de almacenaje de las operaciones pendientes.	68
B.1. Máquina de estados del módulo <i>UART_Rx</i>	70
B.2. Máquina de estados del módulo <i>UART_Tx</i>	71
B.3. Máquina de estados del módulo <i>ULPI</i>	72
B.4. Máquina de estados del módulo <i>ULPI_REG_WRITE</i>	73
B.5. Máquina de estados del módulo <i>ULPI_REG_READ</i>	74
B.6. Máquina de estados del módulo <i>ULPI_RECV</i>	75
B.7. Máquina de estados del módulo <i>ULPI_op</i>	76
B.8. Máquina de estados del módulo <i>main_controller</i>	77
C.1. Simulación final de botones externos (1 y 2) y lectura de registro (3)	80
C.2. Simulación final de transmisión de registro (4), escritura de registro (5) y captación <i>USB</i> (6)	81
C.3. Simulación final de envío de captura al PC (7)	82
C.4. Simulación final de captación <i>USB</i> (8), cambio de estado del bus (9) y desactivación de envío (10)	83

Lista de acrónimos, siglas y símbolos

Acrónimos

- bit* — Acrónimo inglés de *binary digit* (dígito binario en español). Es la unidad más básica de información en informática, electrónica y telecomunicaciones.

Siglas

- API — Siglas inglesas de *Application programming interface* (Interfaz de Programación de Aplicaciones en español).
- FIFO — Siglas inglesas de *Firts-In Firts-Out* (Primero-en-Entrar Primero-en-Salir en español).
- FPGA — Siglas inglesas de *Field-Programmable Gate Array* (Matriz de Puertas Programables en español).
- HID — Siglas inglesas de *Human Interface Device* (Dispositivo de Interfaz Humana en español).
- PC — Siglas inglesas de *Personal Computer* (Computadora Personal en español).
- PCB — Siglas inglesas de *Printed Circuit Board* (Placa de Circuito Impreso en español).
- LED — Siglas inglesas de *Light Emitting Diode* (Diodo Emisor de Luz en español).
- ULPI — Siglas inglesas de *UTMI+ Low Pin Interface* (Interfaz UTMI+ de Bajos Pines en español).
- USB — Siglas inglesas de *Universal Serial Bus* (Bus Universal en Serie en español).

Símbolos

- Bd* — Baudio. Unidad que indica la cantidad de símbolos enviados por segundos en transmisiones digitales.
- Hz* — Hercio. Unidad del Sistema Internacional que indica la frecuencia de oscilación de una partícula en un periodo de un segundo.
- V* — Voltio. Unidad del Sistema Internacional que indica la tensión eléctrica.

Capítulo 1

Introducción

Durante estas dos últimas décadas, el bus USB[30] (*Universal Serial Bus*) se ha impuesto como uno de los grandes pilares de la comunicación en la informática moderna, no solo en productos de grandes masas (como dispositivos de almacenamiento o periféricos HID[31]), sino también en sectores más complejos y robustos como lo son el militar o el industrial.

Al ser un bus tan ampliamente utilizado, existen multitud de ocasiones en las que sería de gran interés poder disponer de los datos que circulan por él, ya sea para fines de seguridad[19], control, ingeniería inversa o búsqueda y solución de problemas. Por ello, es de gran interés disponer de sistemas capaces de realizar dicha tarea de forma económica y sencilla.

Se considera entonces la elaboración del presente proyecto, en el que se plantea un sistema de captación de tramas USB por *hardware* que sea capaz como mínimo de obtener señales tanto *Low Speed* como *Full Speed*, todo ello a un coste bajo que pueda suplir la demanda existente, y utilizando en gran medida herramientas y utilidades de código libre.

1.1. Organización de la memoria

La organización de este documento es conforme al anexo TFG-08b[24] de la Normativa Trabajo Fin de Grado de la Escuela de Ingeniería Industrial de Toledo, de la Universidad de Castilla-La Mancha, aprobada en Junta de Escuela el 23 de junio de 2016. Se descompone en los siguientes capítulos.

Capítulo 1 Presente capítulo. Se introduce el problema a tratar.

Capítulo 2 Analiza los antecedentes y el estado del arte de los dispositivos de captura USB.

Capítulo 3 Enumera y justifica los objetivos del proyecto.

Capítulo 4 Enumera las principales contribuciones y aportaciones en este TFG.

Capítulo 5 Describe las directivas de desarrollo empleadas durante la ejecución del TFG.

Capítulo 6 Describe en detalle las pruebas realizadas y los resultados obtenidos de los elementos *hardware* del sistema.

Capítulo 7 Describe en detalle los resultados obtenidos de los elementos *software* del sistema.

Capítulo 8 Recopila las principales conclusiones del proyecto y posibles trabajos futuros.

Anexo A Muestra gráficamente las relaciones entre los módulos diseñados.

Anexo B Muestra los esquemas de las maquinas de estados internas de los módulos diseñados.

Anexo C Muestra el resultado de la simulación final del sistema.

Anexo D Detalla los procedimientos a seguir para la instalación y utilización del sistema.

1.2. Repositorio de información

Todo el material generado durante la ejecución de este proyecto está disponible tanto en el disco adjunto, como en <https://github.com/mario-mra/tfg>. Se incluye el código L^AT_EX del presente documento, el código fuente de los programas y módulos realizados, y todos los datos generados en la evaluación de resultados.

Capítulo 2

Motivación y antecedentes

A la hora de realizar una captura USB, existen dos técnicas diferentes, cada una con ciertas ventajas e inconvenientes, comentadas a continuación.

1. Por medio de un *software* de captura en uno de los equipos implicados (Figura 2.1).

Método más sencillo y más económico, ya que no es necesario utilizar herramientas externas a parte del propio *software*, pero con ausencia de varios datos de interés, como pueden ser el estado del Bus o los identificadores de paquetes (*PIDs*).

Posee a su vez limitaciones en su ejecución, ya que es necesario un completo acceso al equipo donde se realice la captura, existiendo además la posibilidad de no disponer de *software* compatible para el mismo.

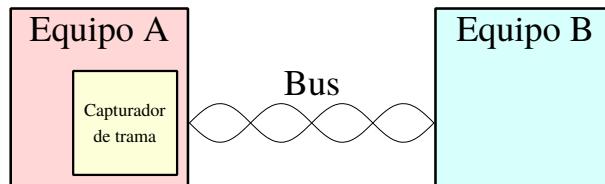


Figura 2.1: Esquema de captura por medio de *software*

2. Utilizando un elemento *hardware* intermedio entre los equipos (Figura 2.2).

Método que permite un mayor control y compatibilidad al ser totalmente independiente a los equipos implicados, pero que debido al uso de herramientas externas y a la necesidad de utilizar un equipo extra donde grabar y procesar los datos obtenidos, hace que sea un método mas laborioso, y por lo general, mas costoso. Se obtiene una mayor cantidad de información comparado al método *software* al tener un acceso directo al propio bus.

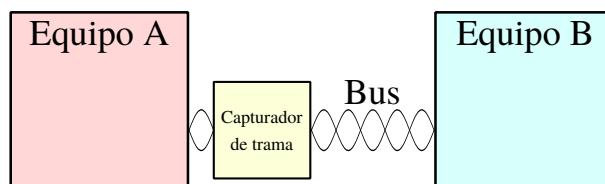


Figura 2.2: Esquema de captura por medio de *hardware*

2.1. Sistemas *software* actuales

Debido a la gran facilidad y bajo coste que trae consigo el utilizar sistemas de captación *software*, no es de extrañar que la gran mayoría de productos comerciales ya existentes se encuentran dentro de este grupo.

A continuación se muestra una lista detallada de los sistemas de captación *software* más relevantes, clasificados según su tipo de licencia y ordenados según la fecha de última actualización.

2.1.1. Licencia de tipo *shareware*

Se trata de *software* no gratuito, de código cerrado y libre distribución entre usuarios^[10], que por lo general, posee un periodo de prueba en el que disfrutar la mayoría de las funciones sin coste.

Todos las aplicaciones descritas a continuación poseen una interfaz gráfica e intuitiva (Figura 2.3), y funcionan únicamente bajo al sistema operativo *Windows™*.

- ***USB Monitor*¹ de HHD Software Ltd.**

Posee un periodo de prueba de 14 días en el que se permite utilizar todas las funciones sin restricciones. Su fecha de última actualización data del 15 de abril del 2019.

Cabe destacar que este sistema tiene cuatro ediciones², comentadas a continuación.

1. **Edición standard.**

Es la versión más básica. Incluye las siguientes funciones:

- Soporte para todas las versiones de USB.
- Funciones básicas de monitorización y visualizado.
- Filtros básicos.
- Guardado básico.
- Estadísticas de uso.
- Monitorización remota.

Su precio, a 27 de Marzo de 2019, es de 54.99€ para uso NO comercial y 74.99€ para uso comercial.

2. **Edición gratuita³.**

Permite las mismas funciones que la edición *standard*, limitando su uso máximo a cinco sesiones diarias, cada una de 10 minutos.

3. **Edición professional.**

Incluye las ventajas de la edición *standard*, añadiendo:

- Conversión de datos y visionado de datos *HID*, imagen o audio, entre otros.
- Filtros en la captura.
- Capacidad de exportar los datos capturados de forma avanzada.

Su precio, a 27 de Marzo de 2019, es de 119.99€ para uso NO comercial y 169.99€ para uso comercial.

4. **Edición ultimate.**

Incluye las ventajas de la edición *professional*, añadiendo:

- Visor de paquetes USB en bruto.

¹Página web del producto: <https://www.hhdsoftware.com/usb-monitor>

²Comparación completa de las ediciones de pago: <https://www.hhdsoftware.com/usb-monitor/compare>

³Se distribuye bajo el nombre de *Free USB Analyzer* en: <https://freeusb analyzer.com/>

- Monitorización simultanea de varios dispositivos.
- Scripts personalizados usando el lenguaje *TypeScript*.

Su precio, a 27 de Marzo de 2019, es de 159.99€ para uso NO comercial y 224.99€ para uso comercial.

■ ***USB Sniffer*⁴ de Eltima Software.**

Software con 14 días de prueba, cuya versión completa tiene un precio único de mercado, a 27 de Marzo de 2019, de \$69.95 (sin incluir comisiones de cambio de divisa e impuestos). Su fecha de última actualización data del 15 de marzo del 2019.

Hay que destacar las siguientes características⁵.

- Vista completa y detallada de los datos entrantes y salientes.
- Soporte para *HUBs* USB.
- Control de la captura según los datos de entrada.
- Soporte para todas las versiones de USB.
- Guardado avanzado de la trama capturada.
- Capacidad de marcar datos en la interfaz.
- Monitorización simultanea de varios dispositivos.

■ ***USBlyzer*⁶.**

De manera similar que en los casos anteriores, brinda 33 días en los que probar todas la funcionalidades que ofrece sin ningún tipo de restricción. Su fecha de última actualización data del 15 de mayo del 2016.

Caben destacar las siguientes características.

- Vista avanzada jerarquizada de todos los USB disponibles.
- Análisis de actividad del bus.
- Filtrado avanzado cuando se realizan búsquedas o capturas.
- Decodificación de una gran variedad de tipos de datos.
- Exportación avanzada.

El precio de una licencia, a 27 de Marzo de 2019, es de \$200 (sin incluir comisiones de cambio de divisa e impuestos), con descuentos de hasta un 46 % si se adquieren varias licencias.

■ ***USBTrace*⁷ de SysNucleus.**

Igual que todos los anteriores, este programa utiliza una interfaz en la que mostrar y analizar en tiempo real el tráfico de puerto.

Su periodo de prueba tiene una duración de 15 días, limitando además la cantidad total de la captura a 256Kb por sesión. Su fecha de última actualización data del 11 de noviembre del 2014.

Entre otras⁸, sus funciones más destacables son:

- Facilidad de uso y visualización.

⁴Página web del producto: <https://www.eltima.com/products/usb-sniffer/>

⁵Lista completa de funciones: <https://www.eltima.com/products/usb-port-monitor#tableCheckList>

⁶Página web del producto: <http://www.usblyzer.com/>

⁷Página web del producto: <http://www.sysnucleus.com/>

⁸Lista de funciones completas: http://www.sysnucleus.com/usbtrace_features.html

- Variedad de filtros.
- Guardado avanzado.
- Permite análisis de los drivers utilizados por el equipo.
- Captura en segundo plano.
- Decodificaciones personalizadas de los datos del bus.

Su precio unitario, a 27 de Marzo de 2019, asciende hasta \$125.00 (sin incluir comisiones de cambio de divisa e impuestos).

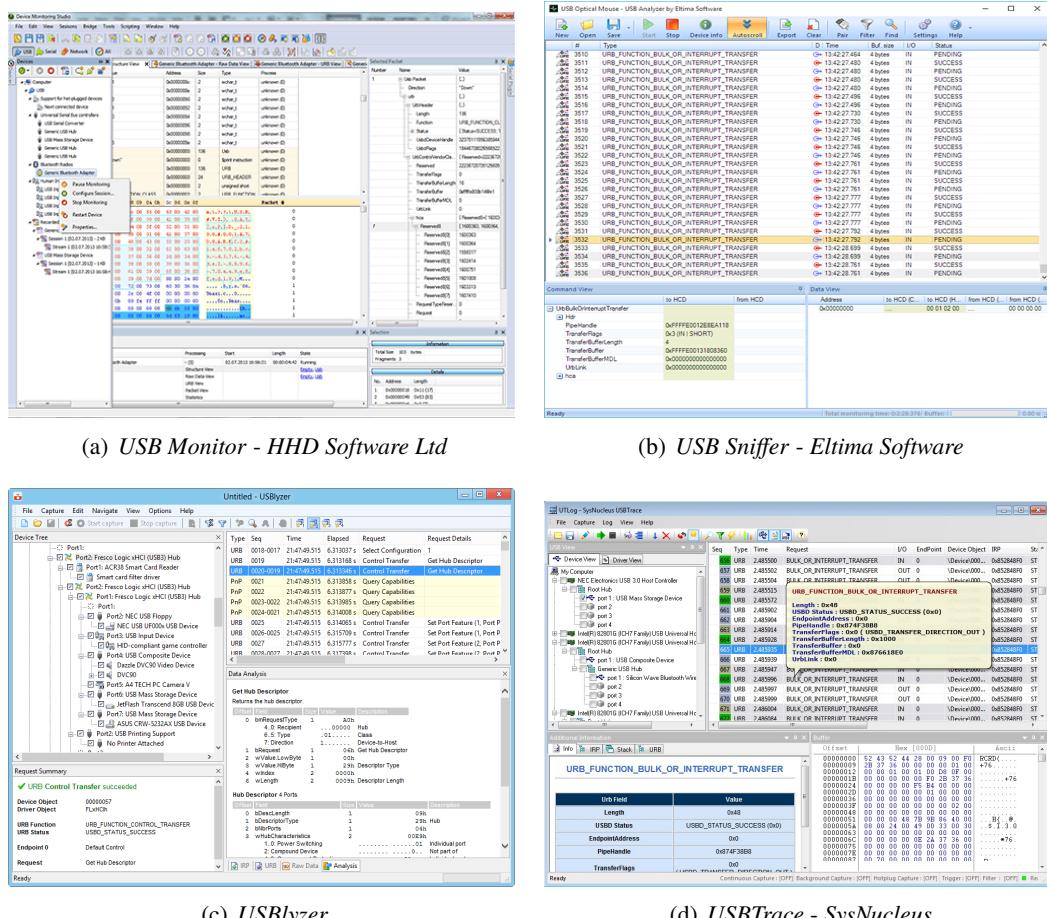


Figura 2.3: Interfaces gráficas de los analizadores *software* de tipo *shareware*. Imágenes extraídas de las páginas web de los desarrolladores.

2.1.2. Licencia de código libre

Se trata de *software* gratuito, cuyo código fuente está abiertamente disponible para su revisión y posibles modificaciones^[8].

Dentro de este grupo hay que destacar dos herramientas, con funcionalidades similares, pero disponibles una bajo el sistema operativo *Windows™*, y la otra bajo sistemas *UNIX-like*^[9].

^[9]Sistemas con comportamiento similar a sistemas Unix, sin ser necesariamente certificado a ninguna *Single Unix Specification* (Especificación Única Unix en español).

- **Tcpdump¹⁰**.

Herramienta bajo licencia BSD¹¹, que a través de una linea de comandos, es capaz de capturar, analizar y almacenar en un archivo PCAP paquetes de una amplia variedad de interfaces, entre las que se encuentra dispositivos *USB*.

Este método, debido a su especialización, no sería capaz de realizar un análisis detallado, por lo que sería necesario un programa adicional que posteriormente haga el estudio y clasificación de los datos de forma gráfica.

Funciona en sistemas *UNIX-like*, siempre que se tenga acceso la interfaz USB.

- **USBPcap¹²**.

Realiza funciones similares a las que realiza *Tcpdump* en cuanto a captura USB, pero funcionando bajo el Sistema Operativo Windows™. Posee licencias GPLv2 y BSD.

De igual manera que en el caso anterior, sigue siendo necesario el uso de *software* de terceros para su análisis completo.

Dichos archivos generados, por lo general de tipo *pcap*[9], se pueden analizar a través de la herramienta gráfica de análisis de paquetes *Wireshark*¹³ (véase figura 2.4). Esta está disponible para cualquiera de los dos sistemas operativos anteriores.

Hay destacar, que esta herramienta, y únicamente bajo sistemas basado en *Linux*, puede ser capaz de capturar directamente y en tiempo real las tramas *USB*.

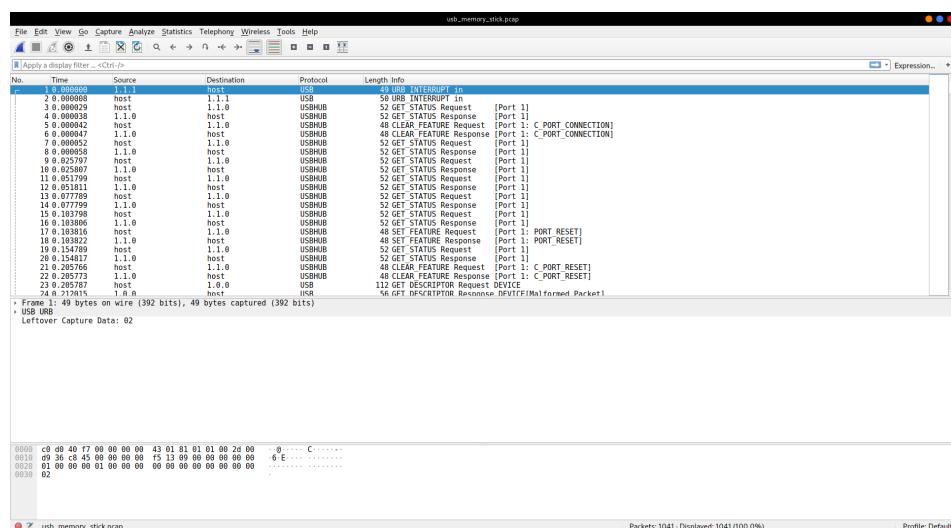


Figura 2.4: Interfaz gráfica del analizador de paquetes *Wireshark*

2.2. Sistemas *hardware* actuales

En caso de que no sea posible la utilización de una captura por *software*, ya sea por incompatibilidades, falta de acceso, necesidad de obtener más datos de interés o cualquier otra causa, se tendría que utilizar un sistema más complejo y costoso, por medio de *hardware*.

A continuación se muestran varios ejemplos ya existentes de este tipo, todos ellos soportan hasta la versión 2.0 de USB[30].

¹⁰Página web del *software*: <https://www.tcpdump.org/>

¹¹Información detallada de la licencia: <https://www.tcpdump.org/license.html>

¹²Página web del *software*: <https://desowin.org/usbpcap/>

¹³Página web del *software*: <https://www.wireshark.org/>

Nota. Todos los precios de la siguiente lista, excepto que se indique, no incluyen envío, impuestos locales, o tasas de cambio de divisa.

■ **USB Explorer 200 de Ellisys¹⁴.**

Sistema comercial, que entre otras características^[5], se destaca:

- Soporte para USB 2.0 *high speed* (480 Mbit/s), *full speed* (12 Mbit/s) y *low speed* (1,5 Mbit/s).
- *Trigger* externo de hasta 5V de entrada o 3.3V de salida.
- Memoria interna FIFO de 32 MBytes.
- Alimentación a través de la propia conexión USB en el equipo de análisis.
- Dimensiones aproximadas de 150x120x65mm con un peso de 750 gramos. Véase figura 2.5.

Aun siendo el mismo producto, este está limitado, existiendo tres variaciones, cada una con más funcionalidades y mayor precio respecto a la anterior.

1. Edición básica.

Es la versión más básica, permitiendo únicamente capturar y posteriormente almacenar en el equipo de análisis la trama USB.

Su precio, a 27 de Marzo de 2019, es de 799€.

2. Edición estándar.

Incluye las ventajas de la edición básica, añadiendo:

- Captura en tiempo real.
- Filtros de captura.
- Resumen del tráfico existente.
- Decodificación parcial del protocolo USB.

Su precio, a 27 de Marzo de 2019, es de 1599€.

3. Edición profesional.

Incluye las ventajas de la edición estándar, añadiendo:

- Decodificación completa USB.
- Análisis del protocolo.
- Capacidad de usar disparos (*triggers*) externos.

Su precio, a 27 de Marzo de 2019, es de 3199€.



Figura 2.5: Ellisys USB Explorer 200. Imagen extraída de la página web del fabricante.

■ **Mercury T2 de Teledyne LeCroy¹⁵.**

Sistema comercial, que entre otras características^[26], se destaca:

¹⁴Página web del producto: <https://www.ellisys.com/products/usbex200/index.php>

¹⁵Página web del producto: <https://teledynelecroy.com/protocolanalyzer/usb/mercury-t2>

- Soporte para USB 2.0 *high speed* (480 Mbit/s), *full speed* (12 Mbit/s) y *low speed* (1,5 Mbit/s).
- Memoria interna de 256 MBytes.
- Almacenado automática en disco para permitir capturas de larga duración.
- *Trigger* externo (necesario adaptador externo no incluido).
- Alimentación a través de la propia conexión USB en el equipo de análisis.
- Dimensiones aproximadas de 80x90x24mm con un peso de 158 gramos. Véase figura 2.6.

De igual manera que en el producto anterior, este posee varias versiones.

1. Edición estándar.

Incluye las siguientes opciones:

- Captura y almacenaje de cualquier trama USB hasta 2.0 en tiempo real.
- Disparos (*triggers*) tanto externos, como detectando patrones en la trama, usados para iniciar una captura.

Su precio, a 27 de Marzo de 2019, es de \$901.

2. Edición avanzada.

Incluye las ventajas de la edición estándar, añadiendo:

- Estadísticas en tiempo real del bus.
- Exportación en formato .csv.
- API de automatización.

Su precio, a 27 de Marzo de 2019, es de \$1235.



Figura 2.6: Teledyne LeCroy Mercury T2. Imagen extraída de la página web del fabricante.

■ Beagle USB de Total Phase¹⁶.

Dentro de la amplia gama de productos que disponen, hay dos que destacan principalmente.

1. Beagle USB 480[27].

De todas las versiones que poseen, esta es la más básica que permite hasta capturas *high speed* de (480 Mbit/s).

- Captura en tiempo real.
- Estadísticas en tiempo real.
- 64MBytes de memoria integrada.
- *Triggers* externos de entrada y salida.
- Sincronización básica.
- API de control.

Su precio, a 27 de Marzo de 2019, es de \$2250.

¹⁶Página web del producto: <https://www.totalphase.com/protocols/usb/>

2. Beagle USB 480 Power[28] - Edición estándar.

Posee las mismas ventajas que el producto anterior, pero aumentando la memoria integrada de $64M\text{Bytes}$ a $256M\text{Bytes}$ y añadiendo capacidad de medir la tensión y corriente del propio Bus.

Su precio, a 27 de Marzo de 2019, es de \$1599.

3. Beagle USB 480 Power[28] - Edición ultimate.

Mejora las capacidades de disparos (*triggers*) respecto a la versión estándar.

Su precio, a 27 de Marzo de 2019, es de \$2950.



(a) *Beagle USB 480*

(b) *Beagle USB 480 Power*

Figura 2.7: Productos de *Total Phase*. Imágenes extraídas de la página web del fabricante.

■ **USB Sniffer¹⁷ de Ultra-EMBEDDED**

Proyecto NO comercial, que usando una *FPGA Xilinx Spartan 6 LX9*, y una placa de desarrollo encargada de procesar la capa física, es capaz de capturar hasta *USB 2.0 high speed* (480 Mbit/s). En la figura 2.8 se muestra una imagen del proyecto.

Al ser un proyecto personal, no dispone de un producto final que se pueda adquirir, por lo que si se quiere utilizar se deben comprar las piezas individualmente. Además, el código utilizado es *open source*, por lo que se puede replicar sin ningún tipo de problema. Su precio aproximado se muestra en la tabla 2.1.

Tabla 2.1: Componentes y precio de la propuesta de analizador USB de *Ultra-EMBEDDED*

Componente	Unidades	Precio
FPGA Xilinx Spartan 6 LX9	1	\$75
Placa de desarrollo USB3300	1	\$8
Cable IDC	1	\$5
Precio total:		\$88

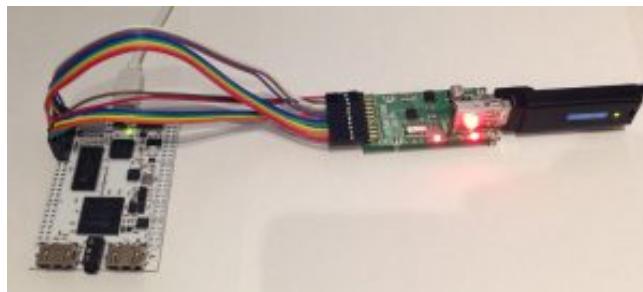


Figura 2.8: *USB Sniffer*. Imagen extraída de la página web del proyecto.

¹⁷Página web del proyecto: http://ultra-embedded.com/usb_sniffer/

2.3. Comparativa de sistemas *software*

En la tabla 2.2 se muestra una comparativa de los productos de captura *software* más destacados.

Se aprecia, que dentro del *software* de pago, el que más funcionalidades ofrece a razón de su precio, es *USB Sniffer* de *Eltima Software*, seguido de *USB Monitor* de *HHD Software Ltd*. Las otras dos opciones no poseen actualizaciones recientes, y su precio es más elevado, por lo que son menos recomendables.

Por otro lado, las opciones de *código libre*, aun teniendo una curva de aprendizaje muy superior, son opciones totalmente viables y recomendables.

Tabla 2.2: Comparativa de sistemas *software*

Nombre	Licencia y precio	Ventajas	Desventajas
USB Monitor de <i>HHD Software Ltd</i>	<i>Shareware</i> Desde 54.99€	Actualizaciones frecuentes. Decodificación «en vivo». Gran sencillez de uso. Posee versión gratuita.	Extra para uso comercial. Varias ediciones.
USB Sniffer de <i>Eltima Software</i>	<i>Shareware</i> \$69.95	Monitorización simultanea. Soporte para <i>HUBs USB</i> . Gran sencillez de uso.	Sin capacidad de decodificación.
USBlyzer	<i>Shareware</i> Hasta \$200	Decodificación incluida. Exportación avanzada.	Sin actualizaciones recientes. Precio elevado.
USBTrace de <i>SysNucleus</i>	<i>Shareware</i> \$125.00	Decodificado personalizado. Análisis de <i>drivers</i> .	Sin actualizaciones recientes. Pocas funcionalidades.
Tcpdump y USBPcap	<i>Código libre</i> –	Soporte de la comunidad. Multiplataforma. Gratis.	Solo realiza capturas de trama. Pocas funcionalidades extra. Sin interfaz gráfica.
Wireshark	<i>Código libre</i> –	Variedad de funcionalidades. Filtrado avanzado. Multiplataforma. Gratis.	No realiza capturas. Poco intuitivo.

2.4. Comparativa de sistemas *hardware*

En la tabla 2.3 se muestra una comparativa de los productos de captura *hardware* más destacados.

Se aprecia que dentro de este grupo, el equipo más recomendable a razón de sus funcionalidades ofrecidas y precio es el *Mercury T2* de *Teledyne LeCroy*. Si se buscan más funcionalidades y almacenamiento, es más viable utilizar el *Beagle USB 480 Power* de *Total Phase*.

Tabla 2.3: Comparativa de sistemas *hardware*

Nombre	Licencia y precio	Ventajas	Desventajas
USB Explorer 200 de <i>Ellisys</i>	Cerrado. Desde 799€ a 3199€	Posibilidad de análisis completo. Captura en tiempo real. Disparos externos.	Versión básica simple. Escasa memoria. Sistema costoso.
Mercury T2 de <i>Teledyne LeCroy</i>	Cerrado. Desde \$901 a \$1235	API de automatización. Disparos avanzados. Tamaño reducido.	Adaptador para disparos externos no incluido. Pocas funciones extras.
Beagle USB 480 de <i>Total Phase</i>	Cerrado. \$2250	Estadísticas en tiempo real. API de automatización. Sincronización básica.	Sin funciones de análisis. Escasa memoria.
Beagle USB 480 Power de <i>Total Phase</i>	Cerrado. Desde \$1599 a \$2950	Alta capacidad de memoria. Análisis de potencia.	Versión estándar sin disparos avanzados.
USB Sniffer de <i>Ultra-Embedded</i>	Código libre. Aprox. \$88	Muy económico. Código libre.	No está en venta directa. Solo permite captura. Memoria escasa.

Capítulo 3

Objetivos

Bajo la premisa de realizar un **anificador USB hardware de bajo coste** con herramientas y utilidades de código libre, el presente Trabajo Fin de Grado se ha descompuesto en varios objetivos parciales, estos siguen siempre una metodología SMART, es decir, deben ser Simples, Medibles, Acordados, Realista y Temporizados.

3.1. Selección de componentes *hardware*

Debido a la flexibilidad que supone utilizar una FPGA[18], se propone diseñar el sistema de captura a partir de una. Por otra parte, ya que existen circuitos especializados en procesar capas física USB, es de gran interés incluir uno de ellos¹, pudiendo abstraer de esa forma ciertas partes de la comunicación USB de poco interés, y reduciendo en cierta medida posibles errores de sincronización.

Se deben buscar por tanto, todos aquellos componentes necesarios, de la forma más económica posible, para elaborar el sistema de captura. Seguidamente, se ha de elegir como se conexionan mutuamente.

3.2. Implementación de memoria *FIFO*

Diseño e implementación de un módulo en lenguaje Verilog que actúe como memoria *FIFO*. Este será capaz de almacenar información de tal manera que el primer *bit* introducido sea el primero en ser recuperado.

Dado que la FPGA a utilizar dispone de varios bloques de RAM específicos, se utilizarán varios de estos para no depender únicamente de registros, consiguiendo una mayor capacidad de almacenamiento, con un circuito lo mejor optimizado posible.

La gran mayoría de los módulos a diseñar dependen en gran medida de este tipo de memoria, por lo que es esencial disponer de ella en la mayor brevedad posible, a su vez, y debido a su gran utilidad, la velocidad[21] máxima a la que puede funcionar el sistema en su conjunto está dada por su velocidad de escritura/lectura de datos.

En la figura 3.1 se muestra un esquema del resultado esperado.

¹La idea de utilizar un integrado especializado, se obtiene del sistema *USB Sniffer* planteado por *Ultra-Embedded*, nombrado en el capítulo 2 de antecedentes.

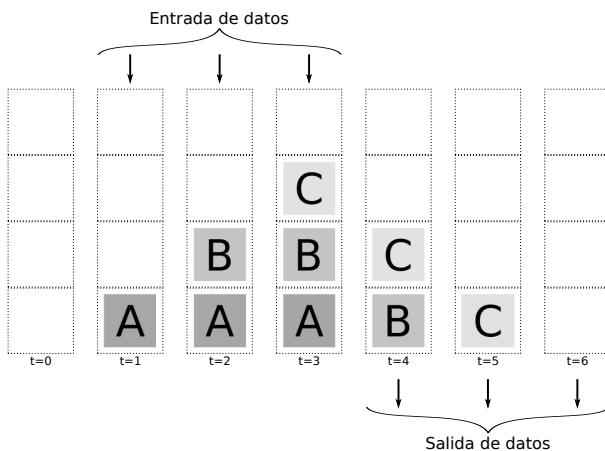


Figura 3.1: Esquema de funcionamiento de una memoria *FIFO*.

3.3. Implementación de un sistema de comunicación serie

Diseño e implementación de un sistema en lenguaje Verilog que sea capaz de comunicarse bidireccionalmente usando un puerto serie simple^[33], pudiendo conectarse a él por medio del circuito integrado FT2232HL² (disponible en la placa de desarrollo *IceStick*^[14]), o utilizando un equipo externo compatible.

Por dicho puerto, la FPGA transmitirá tanto la trama USB capturada como información del bus, y recibirá los comandos que debe seguir.

En la figura 3.2 se muestra una señal típica serie, esta es la señal que debe ser capturada y generada en la FPGA.

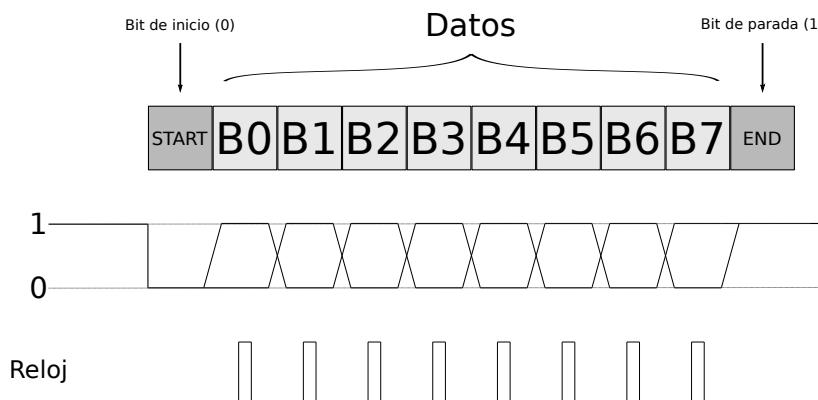


Figura 3.2: Señal típica de una comunicación serie.

A su vez, este objetivo se puede descomponer en varios subobjetivos.

3.3.1. Diseño de un módulo de emisión serie

Módulo capaz de controlar los datos almacenados en una memoria *FIFO*, para posteriormente transmitirlos por el puerto serie.

²Circuito encargado de convertir una conexión *USB High Speed* a dos protocolos configurables distintos. En el caso de la placa de desarrollo *IceStick*, se utiliza un canal para programar la memoria *Flash SPI* utilizada por la FPGA, y otro para proporcionar una comunicación *UART* hacia el equipo.

3.3.2. Diseño de un módulo de recepción serie

Módulo capaz almacenar en una memoria *FIFO* los datos paralelizados capturados en el puerto serie.

3.4. Implementación de un sistema que procese el protocolo ULPI

Diseño e implementación de un sistema en lenguaje Verilog capaz de comunicarse con otros dispositivos compatibles con el protocolo ULPI (*UTMI+ Low Pin Interface*), siendo en este caso, el integrado USB3300 de *Microchip*, encargado de la capa física USB.

Todas las características y casos del protocolo ULPI están completamente definidas tanto en la hoja de especificaciones del propio protocolo[1] como en la hoja de características del circuito integrado USB3300[17]. Hay que destacar que el bus consta de una señal de reloj a 60MHz a la que se referencia el resto de señales (*CLK*), 8 señales de datos bidireccionales paralelos (*DATA*), una señal de control de dirección (*DIR*) y dos señales extra de control (*STP* y *NXT*).

Puesto que únicamente queremos obtener de forma pasiva los datos USB, de los cuatro posibles modos de funcionamiento ULPI, escritura de registros, lectura de registros, recepción de datos USB y envío de datos USB, solo es interesante diseñar los tres primeros, por lo que este objetivo se puede dividir en las siguientes partes.

3.4.1. Diseño de un módulo de escritura de registros ULPI

Módulo capaz de generar y procesar las señales ULPI necesarias para almacenar un valor arbitrario de 8 bits, en un registro del integrado conectado por ULPI.

En la figura 3.3 se puede apreciar la comunicación básica usada para realizar dicha escritura.

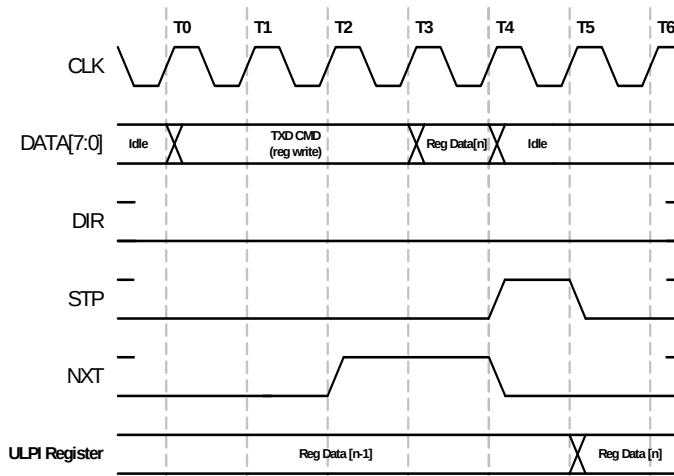


Figura 3.3: Trama de escritura de registros ULPI.

3.4.2. Diseño de un módulo de lectura de registros ULPI

Módulo capaz de generar y procesar las señales ULPI necesarias para obtener un valor almacenado en un registro arbitrario del integrado conectado por ULPI.

En la figura 3.4 se puede apreciar la comunicación básica usada para realizar dicha lectura.

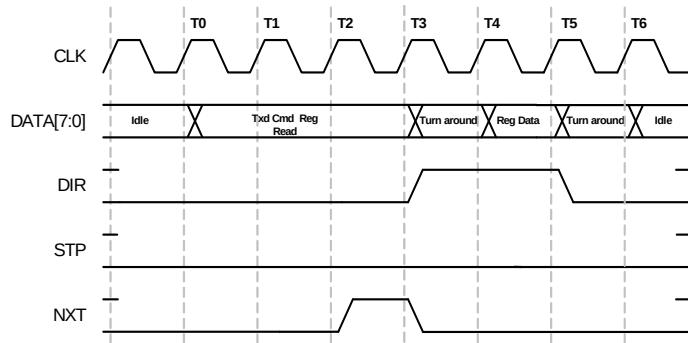


Figura 3.4: Trama de lectura de registros ULPI.

3.4.3. Diseño de un módulo de captación USB

Módulo que ante la llegada de datos USB, sea capaz de procesar las señales ULPI para obtener y clasificar la trama transmitida. En la figura 3.5 se puede apreciar la comunicación básica existente durante la lectura de datos.

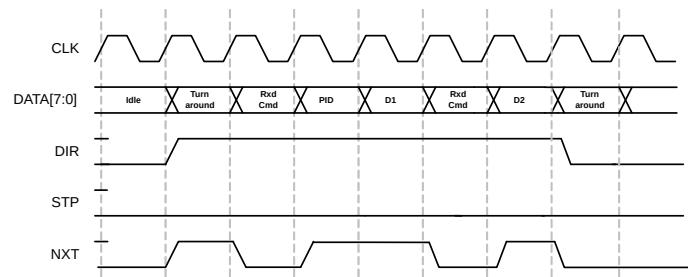


Figura 3.5: Trama ULPI de recepción de datos USB.

3.5. Implementación de un sistema que gestione el envío de datos

Diseño e implementación de un módulo en lenguaje Verilog, que gestione el envío al PC de los datos capturados. Además, sería interesante añadir una memoria intermedia en la que almacenar los datos antes de su envío, aminorando posibles perdidas.

3.6. Implementación de un sistema que gestione los comandos entrantes

Diseño e implementación de un módulo en lenguaje Verilog, que ante cualquier petición realizada por el usuario por el puerto serie anterior, deberá ser capaz de procesarla, obteniendo sus diversas partes (por ejemplo, dirección y datos en una petición de escritura de registro) y las almacenará temporalmente hasta que se puedan ejecutar.

3.7. Creación de una aplicación de control

Tras la finalización de la parte *hardware*, es necesaria una aplicación, que conectándose por medio del puerto serie anteriormente implementado, sea capaz de en primer lugar enviar los

diversos comandos necesarios para el correcto funcionamiento del sistema, y en segundo lugar, reciba los datos capturados de interés.

Debe ser como mínimo capaz de realizar las siguientes tareas.

- Proveer de una sencilla interfaz en la que controlar todos los aspectos del sistema de captura.
- Controlar y procesar los datos entrantes y salientes del puerto serie.
- Almacenar los datos capturados en un archivo que permita un fácil análisis futuro, preferiblemente *PCAP*[25].

Capítulo 4

Contribuciones

Durante el proceso de diseño del sistema de la FPGA, y ante la necesidad de crear módulos o bloques funcionales de utilidad en lenguaje Verilog[23], se siguen una serie de procedimientos definidos desde un comienzo. Gracias a su uso, a parte de establecer unos mínimos requerimientos de calidad, se obtienen con mayor eficiencia los resultados buscados, permitiendo a su vez una alta legibilidad del código, y una buena capacidad de modificación y reutilización.

Para garantizarlo, estos procedimientos deben de ser comunes en todos los módulos, e invariables a lo largo del proyecto.

4.1. Etapas de diseño de un módulo

En primer lugar, tras requerir la utilización de un módulo que logre una función específica, se analiza la posibilidad de reutilizar uno o varios módulos anteriormente realizados, evitando así un derroche innecesario de tiempo y recursos.

Seguidamente, se estudia la complejidad del módulo a desarrollar, dividiéndolo, si fuera necesario, en varios submódulos con funciones más concretas que disminuyan el grado de dificultad general en su elaboración, permitiendo a su vez una mayor reutilización futura de código.

Estando definida ya su distribución, se realiza un pequeño esquema sobre el que partir, que de forma gráfica muestre sus diversas partes y relaciones, junto a este, si fuera necesario, se dibuja otro que muestre las etapas de la máquina de estados, pudiendo a continuación empezar con la programación del mismo.

Cada módulo dispone de un estilo común, distribuido de la siguiente forma:

1. Comentario de cabecera.

Tal como se muestra en el listado 4.1, se nombra al módulo, incluyendo una descripción breve de sus funciones. Además, se enumeran y explican todas las entradas, salidas, parámetros y estados de los que está formado.

Listado 4.1: Ejemplo de comentario de cabecera del módulo.

```
1 /*
2  * <Nombre del modulo> module
3  * <Descripción del modulo>
4  *
5  * Parameters:
6  * - <Nombre del parámetro>. <Descripción del parámetro>
7  * - etc ..
8  *
```

```

9  * Inputs:
10 * - <Nombre de la entrada>. <Descripción de la entrada>
11 * - etc ..
12 *
13 * Outputs:
14 * - <Nombre de la salida>. <Descripción de la salida>
15 * - etc ..
16 *
17 * States:
18 * - <Nombre del estado>. <Descripción del estado>
19 * - etc ..
20 */

```

2. Control de reinicio síncrono/asíncrono (listado 4.2).

Se trata de un pequeño bloque que controla la forma de funcionamiento de los reinicios. Si en el archivo principal se define la constante *ASYNC_RESET*, entonces todos los reinicios serán asíncronos, en caso contrario, serán síncronos a la señal de reloj (*clk*).

Listado 4.2: Ejemplo de control de reinicio síncrono/asíncrono.

```

1 // X es el nombre del modulo en cada caso
2 `ifdef ASYNC_RESET
3   `define X_ASYNC_RESET or negedge rst
4 `else
5   `define X_ASYNC_RESET
6 `endif

```

3. Incorporación de módulos necesarios (listado 4.3).

Suele ser habitual, que el módulo que se está desarrollando haga uso de otros, los cuales son incorporar en este momento.

Listado 4.3: Ejemplo de incorporación de módulos.

```

1 `include "./modules/nombre_del_modulo.vh"
2 \\ etc ..

```

4. Creación del módulo (listado 4.4).

Se define el módulo, agrupando las entradas y salidas según características similares, posicionando en primer lugar las entradas, seguido de las salidas.

Listado 4.4: Ejemplo de creación de módulo.

```

1 module nombre_modulo
2   #(
3     parameter nombre_parametro = valor_parametro
4     \\ etc ..
5   )
6   (
7     // <Nombre del primer grupo de entradas/salidas>
8     input wire nombre_entrada_1, // <breve descripción>
9       // etc ..
10    output wire nombre_salida_1, // <breve descripción>
11      // etc ..
12  );

```

5. Inicialización de módulos necesarios (listado 4.5).

Todos los módulos incorporados previamente necesitan ser inicializados, relacionando sus entradas/salidas con el resto de señales del módulo.

Listado 4.5: Ejemplo inicialización de módulos.

```

1 // Ejemplo inicializando el modulo clk_baud_pulse, que tiene dos parámetros, una entrada y
   dos salidas
2 clk_baud_pulse #(
3     .COUNTER_VAL(BAUDS),
4     .PULSE_DELAY(BAUDS/2)
5 )
6     clk_baud_Rx (
7         .clk_in(clk),           // Input
8         .clk_pulse(clk_Rx),    // Output
9         .enable(enable)        // Output
10    );

```

6. Creación de los registros de control (listado 4.6).

Se crean todos los registros encargados de controlar el módulo o almacenar las variables.

Listado 4.6: Ejemplo de creación de registros.

```

1 // Control registers
2 reg [1:0]X_state_r = 2'b0; // Register that stores the current X state
3 reg [3:0]X_counter_r = 4'b0; // Register that counts ...
4 reg [7:0]DATA_buff = 8'b0; // Buffer where ...

```

7. Creación de *flags* o señales de control (listado 4.7).

Para poder identificar con facilidad si el sistema se encuentra en un estado concreto, se crean estas señales, cuyo valor será *1* cuando la máquina de estados esté en dicho estado, y *0* en caso contrario.

Listado 4.7: Ejemplo de creación de *flags*.

```

1 // Flags
2 wire X_s_IDLE; // HIGH if X_state_r == X_IDLE, else LOW
3 wire X_s_RUN; // HIGH if X_state_r == X_RUN, else LOW

```

8. Asignación de valores de salida y señales internas (listado 4.8).

Se asignan los valores que tomarán los diversos *wires*, tanto de uso interno, como de salida, estos pueden relacionarse con registros, salidas de módulos ya inicializados u otros *wires*. Además, al final de cada asignación, se comenta cual va a ser su uso, por ejemplo, una asignación de salida o control.

Listado 4.8: Ejemplo de asignación de valores.

```

1 // Assigns
2 assign X_s_IDLE = (X_state_r == X_IDLE) ? 1'b1 : 1'b0; // #FLAG
3 assign X_s_RUN = (X_state_r == X_RUN) ? 1'b1 : 1'b0; // #FLAG
4
5 assign nombre_salida_1 = DATA_buff[2]; // #OUTPUT

```

9. Enumeración de estados (listado 4.9).

Se crean tantos parámetros locales como estados tenga el módulo, cada uno con un valor identificativo único.

Listado 4.9: Ejemplo de enumeración de estados.

```
1 // X States (See module description at the beginning of this file to get more info)
2 localparam X_IDLE = 1'b0;
3 localparam X_RUN = 1'b1;
```

10. Máquina de estados (listado 4.10).

Se fijan y distribuyen los diversos caminos que puede tomar el sistema. Para ello se realiza una máquina de estados *Mealy*[16], es decir, los nuevos estados dependen de las entradas y del propio estado actual. Un pequeño ejemplo de una máquina de estados se muestra en el listado 4.10

Listado 4.10: Ejemplo de máquina de estados.

```
1 always @(posedge clk `X_ASYNC_RESET) begin
2     if(!rst) X_state_r <= X_IDLE;
3     else begin
4         case(X_state_r)
5             X_IDLE: begin
6                 if(!nombre_entrada_1) X_state_r <= X_RUN;
7                 else X_state_r <= X_IDLE;
8             end
9             X_RUN: begin
10                X_state_r <= X_IDLE;
11            end
12            default: X_state_r <= X_IDLE;
13        endcase
14    end
15 end
```

11. Actualización de registros (listado 4.11).

Por último, se actualizan los valores de los registros del módulo, teniendo en cuenta que se trata de una máquina de estados *Mealy*[2].

Listado 4.11: Ejemplo de actualización de registros.

```
1 always @(posedge clk `UART_RX_ASYNC_RESET) begin
2     if(!rst) begin
3         X_counter_r <= 0;
4     end
5     else if(X_s_IDLE) begin
6         X_counter_r <= X_counter_r - 1'b1;
7     end
8     else if(X_s_RUN) begin
9         X_counter_r <= X_counter_r + 1'b1;
10    end
11 end
```

4.2. Pruebas y simulaciones

Para asegurar el correcto funcionamiento de cada módulo, estos deben superar varias pruebas que comprueben cada una de sus funcionalidades. En caso de superar todas ellas satisfactoriamente, el módulo estaría listo para ser usado en la síntesis de la FPGA, en caso contrario, tendría que volver a la fase de desarrollo, en la que buscar y solventar los fallos, utilizando los resultados de las simulaciones realizadas.

Estas pruebas hacen uso de las herramientas de código abierto *Icarus Verilog*¹ (abreviado *iverilog*), encargada de simular el propio código de *Verilog*, y el visor de ondas *GTKWave*[6], permitiendo generar y mostrar gráficamente todas las señales del circuito en cualquier instante de tiempo.

Para llevar a cabo dicha simulación, *Iverilog* tiene como entrada un archivo en lenguaje *Verilog* en el que se inicializa el propio módulo a analizar, y seguidamente, en ese mismo archivo, se van variando las entradas al módulo según su instante de tiempo. Posteriormente, y tras obtener el archivo con el resultado de la simulación, se abre en *GTKWave* y se compara el resultado generado en el módulo, respecto al esperado.

En la figura 4.1 se muestra el resultado gráfico de una simulación de ejemplo.

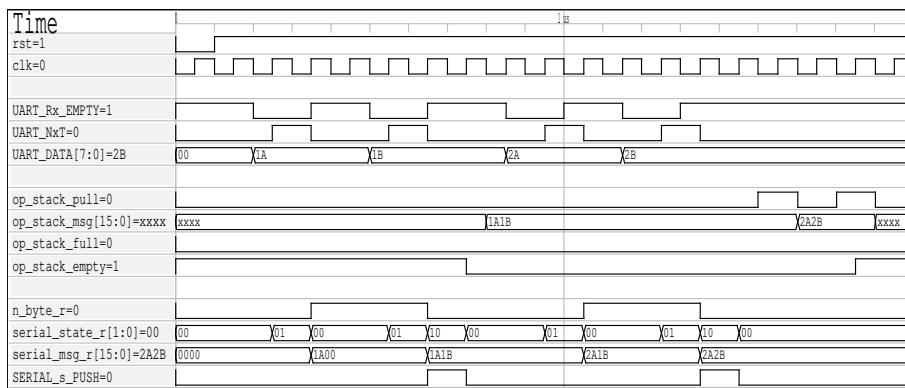


Figura 4.1: Ejemplo del resultado de una simulación en *GTKWave*.

4.3. Herramientas de utilizadas

A parte de la metodología implementada, valida para cualquier sistema a diseñar que utilice lenguaje Verilog, es necesario el uso de herramientas específicas según la FPGA usada.

Particularmente para el presente proyecto, en el que usa una FPGA iCE40HX1k del fabricante *Lattice*, se han utilizado las siguientes herramientas, todas ellas bajo la filosofía de código libre

- **Yosys**². Herramienta de síntesis[32] para Verilog. Provee una colección de algoritmos básicos para varios tipos de aplicaciones, lo que permite poder procesar la gran mayoría de diseños que se planteen. Igualmente, incluye soporte específico para la familia de FPGAs iCE40 de *Lattice*.

Es además una herramienta de código libre bajo licencia ISC³, con gran capacidad de expansión a la hora de realizar pasos de sintetizado por medio de módulos en lenguaje

¹Véase su repositorio: <https://github.com/steveicarus/iverilog>

²Página web del proyecto: <http://www.clifford.at/yosys/>

³Licencia compatible GPL, con términos similares a los definidos por una licencia MIT. Véase <https://www.isc.org/licenses/>.

C++.

- **Nextpnr**⁴. Herramienta de posicionado y encaminamiento (*Place-And-Route*) de los diversos bloques lógicos internos de la FPGA, con soporte a las familias de FPGAs iCE40 y ECP5 de *Lattice*. A parte de una interfaz de líneas de comandos, posee una interfaz gráfica (véase figura 4.2) donde observar con detalle la posición de los bloques utilizados y sus uniones.

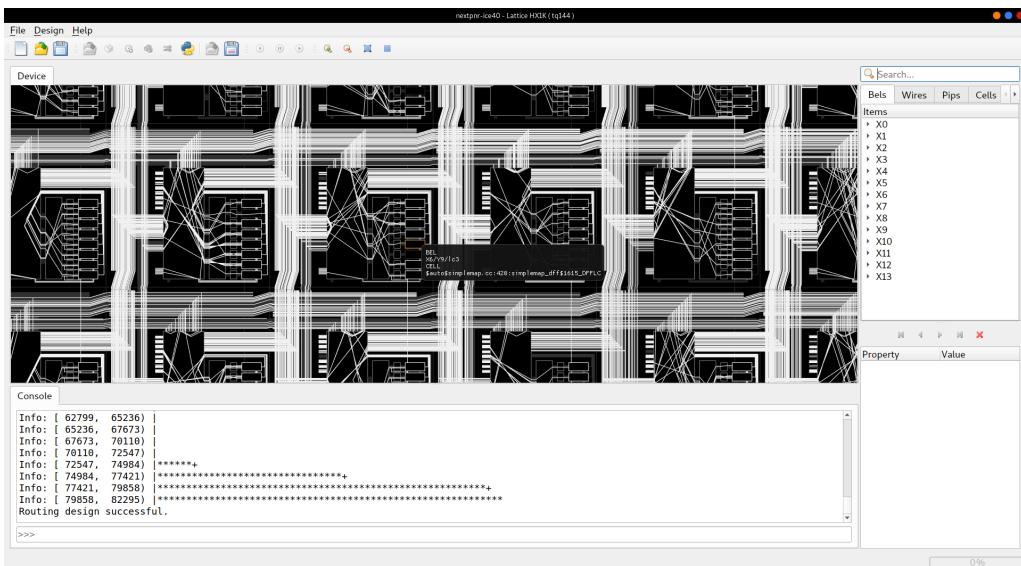


Figura 4.2: Interfaz gráfica de la herramienta *Nextpnr*.

- **IceStorm tools**⁵. El proyecto *IceStorm* tiene la finalidad de hacer ingeniería inversa y documentar el flujo de bits usado por las FPGAs iCE40 de *Lattice*, y han creado varias utilidades, que entre otras, permiten crear un binario compatible, o programar la FPGA.

4.4. Automatización de tareas

Debido a la repetitividad de ciertas tareas durante el proceso de desarrollo, se ha creado un archivo *Makefile*⁶, que de forma rápida ejecute varias acciones.

Entre otras, hay que destacar las siguientes tareas incluidas.

- **synt**. Realiza la síntesis del sistema a partir de la herramienta *Yosys*.
- **pnr**. Gracias a la herramienta *Nextpnr*, hace la tarea de posicionado y encaminamiento (*Place-And-Route*) de la síntesis anterior.
- **pnr_gui**. Realiza la misma tarea que *pnr*, pero muestra una interfaz gráfica donde ver el resultado.
- **pack**. A partir del resultado de *pnr*, genera un archivo binario listo para ser programado en la memoria de la FPGA.
- **prog**. Envía el archivo binario generado a la memoria de la FPGA.

⁴Repositorio del proyecto: <https://github.com/YosysHQ/nextpnr>

⁵Página web del proyecto: <http://www.clifford.at/icestorm/>

⁶Archivo que gracias a la herramienta *Make* disponible en sistemas *unix-like*, gestiona dependencias y automatiza tareas.

- **timing.** Realiza un pequeño análisis temporal para comprobar la velocidad máxima del sistema.
- **plot.** Genera una representación gráfica de todos los bloques y uniones del código Verilog.
- **sim.** Realiza la simulación del sistema por medio de la herramienta *iverilog*.
- **gtk.** Muestra en la aplicación *GTKWave* el resultado de la simulación.

4.5. Distribución de archivos

Todo el código fuente utilizado por el sistema de la FPGA sigue una estructura concreta, reflejada en la figura 4.3.

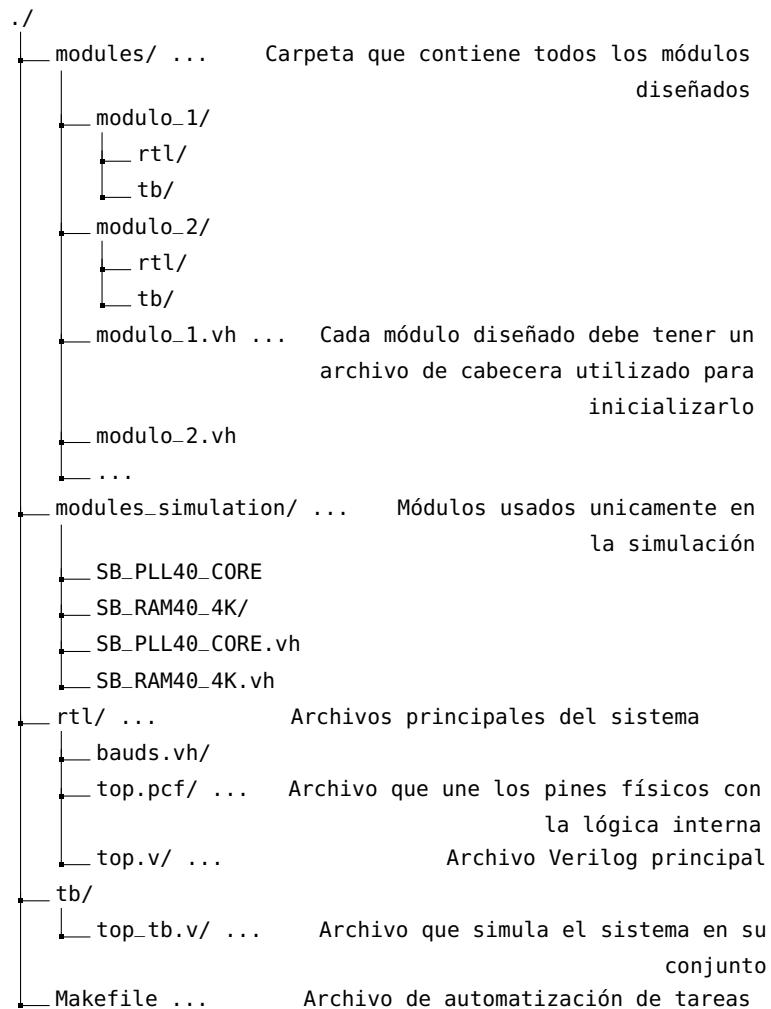


Figura 4.3: Estructura de carpetas del código de la FPGA

Capítulo 5

Procedimiento

En el desarrollo del presente TFG se ha utilizado una metodología ágil basada en *Scrum* [22], definida por el director. El trabajo se ha dividido en iteraciones de varias semanas denominadas *sprints*. Las unidades de trabajo se presentan en forma de historias de usuario (*user stories*) que definen mini-proyectos de muy corta duración que aportan valor al proyecto. Es decir, cada historia de usuario cumple o ayuda a cumplir alguno de los objetivos. Medir el valor percibido corresponde al propietario del producto (*Product Owner*), que participa activamente en la planificación del proceso priorizando las unidades de trabajo.

5.1. Diferencias con Scrum

Scrum es una metodología estrictamente centrada en el cliente. El cliente es el responsable de priorizar y, en cierto modo, planificar las iteraciones. Esto garantiza que la ejecución del proyecto responda al máximo con las expectativas del cliente, aún cuando los imprevistos impidan alcanzar alguno de los objetivos iniciales. Esta característica de *Scrum* es la única que se ha intentado mantener inalterada. Sin embargo, al ser el TFG un proyecto individual, ha requerido modificar significativamente otros aspectos de la metodología.

5.1.1. Roles

La única remuneración que se obtiene con la ejecución de un TFG es la calificación de los distintos aspectos (anteproyecto, valoración del director, valoración del tribunal, etc.). Por tanto, el cliente del TFG se compone por el director y el tribunal de la defensa. Desgraciadamente no es posible conocer a priori el tribunal. Por este motivo el director es el único representante del cliente en el proceso de desarrollo (*Product Owner*).

La labor de dirección del TFG se asimila a la de dirección del proyecto y, por tanto, el director también actúa como coordinador del proceso, o *Scrum Master*. Nótese que hay dos roles representados por la misma persona. Desde un punto de vista purista esto implica que puede haber conflicto de intereses y los intereses del cliente pueden estar insuficientemente representados. Es una limitación extrínseca, que no es posible solucionar con el proceso actual. Aún así, el uso de una metodología ágil centrada en el cliente debe mejorar el alineamiento de intereses cuando surgen problemas que afectan o pueden afectar a la consecución de alguno de los objetivos iniciales.

5.1.2. Historias de usuario

Scrum, como la mayoría de los métodos ágiles, está enfocada al desarrollo de proyectos en entornos de alta incertidumbre por equipos multidisciplinares bien formados. El desarrollo de un TFG, al tratarse de un primer proyecto profesional, también está sometido a gran cantidad de incertidumbre. Sin embargo, no siempre se cuenta con la formación previa necesaria para abordar todos los problemas. Esto implica que, en ocasiones, se necesita aprender o leer, sin repercusión medible en el valor percibido por el *Product Owner*. En esos casos se planifican unidades de trabajo que no corresponden estrictamente a historias de usuario en el sentido de Scrum. Se ha intentado mantener al mínimo este tipo de historias de usuario para tener el proceso lo más controlado posible.

Puntualmente ha sido necesario planificar historias de usuario que solo pretenden explorar opciones. Este tipo de historias de usuario están contempladas en *Scrum*, se denominan *spikes*. Sin embargo, en la ejecución de este TFG se ha procurado reducir al mínimo para que la explotación de alternativas no domine en el tiempo dedicado al TFG.

5.1.3. Planificación de sprints

Para la planificación y el seguimiento se ha utilizado un tablero [Trello](#). Los tableros Trello permiten agrupar tarjetas en una serie de listas con nombre. Se ha utilizado el esquema propuesto en [15].

El autor ha sido responsable de añadir la mayoría de las historias de usuario a la lista *Backlog*. Se trata de un proceso continuo, durante toda la ejecución del proyecto. El director, como *product owner*, prioriza las historias, moviendo las tarjetas dentro de la lista *Backlog*. Justo antes de cada iteración se realiza una reunión presencial o virtual para revisar la iteración pasada y planificar la siguiente iteración.

Usando la técnica de *planning poker* (ver [22]) se dimensionan las historias de usuario en días de trabajo. Esta técnica consiste en un proceso de generación de consenso entre el autor y el director sobre el tiempo requerido para la ejecución de cada historia de usuario. La unidad empleada ha sido de un día.

El director, como *product owner* traslada las tarjetas correspondientes a las primeras historias de la lista *Backlog* a la lista *ToDo* hasta completar los días de trabajo de la iteración.

5.1.4. Flujo de trabajo

El flujo de trabajo diario del autor corresponde a la siguiente secuencia:

- Dentro de la lista *ToDo* puede elegirse cualquier tarjeta para trabajar en ella. Antes de comenzar el trabajo se arrastra la tarjeta a la lista *Doing*. Esto proporciona información en tiempo real al director del progreso de la iteración.
- Al terminar una historia de usuario la tarjeta correspondiente se arrastra a la lista *QC* (*quality control*).
- El director revisa que la historia está realmente acabada y, si así es, la traslada a la lista *Done*. En caso contrario la traslada a la lista *Doing* otra vez, añadiendo un comentario que lo justifique.
- Si en el transcurso del trabajo se encuentra un obstáculo que impide progresar con una historia, se traslada a la lista *Blocked*, añadiendo un comentario que lo justifique.

En todo momento es posible ver el estado global de ejecución del proyecto. Al finalizar, la lista *Done* contiene todas las historias de usuario ejecutadas por orden de terminación. Y las listas *Blocked* y *Backlog* contienen (en este orden) todas las historias de usuario que corresponderían a trabajo futuro, ya priorizadas por el director.

5.1.5. Herramientas de ayuda

El proceso de desarrollo está fuertemente ligado a la herramienta [Trello](#). Se trata de una herramienta colaborativa en línea, que permite mantener una serie de tarjetas agrupadas en listas con nombre. Cada tarjeta puede tener un título, una descripción, un conjunto de adjuntos, y un conjunto de comentarios. Trello se ha usado con éxito en la planificación de proyectos de nivel de complejidad muy variable. Por ejemplo, Epic Games utiliza un tablero Trello para planificar las características a incorporar a cada nueva versión de Unreal Engine. Por otro lado, un problema de Trello es el manejo limitado de la historia de modificaciones en las tarjetas y en los movimientos entre listas de tarjetas. Esto dificulta en cierto modo el seguimiento de los cambios y, sobre todo, la corrección de errores en el proceso. Por este motivo, Trello solo se ha empleado en la coordinación del trabajo, mientras que toda la gestión de cambios se ha delegado en otra herramienta.

Todo el proyecto ha sido gestionado desde su inicio con una herramienta de control de versiones distribuido[3] en un repositorio público de GitHub, disponible en <https://github.com/mario-mra/tfg>. Cada vez que se completa con éxito una historia de usuario se notifica mediante un comentario en la tarjeta correspondiente. Este comentario tan solo contiene el identificador del paquete de cambios (*commit*) que da por concluida la historia. La evolución del proyecto se puede consultar en todo momento desde la propia página del repositorio.

5.2. Aprendizaje de funcionamiento y utilización de FPGAs

Debido a la utilización de una tecnología de la cual el no disponía un conocimiento previo, una fase del TFG se ha centrado en el estudio y aprendizaje de varios aspectos de las FPGAs.

A grandes rasgos, una FPGA[11] se puede considerar como una plataforma con multitud de puertas lógicas y registros biestables, todos ellos dispersos en el propio circuito integrado, y que permiten ser interconectados de forma arbitraria. Este tipo de sistemas, permiten crear en él una infinidad de diseños reprogramables de cualquier complejidad. Otra de sus grandes ventajas, es su gran nivel de paralelización, permitiendo realizar multitud de operaciones sin ningún tipo de relación entre ellas.

El diseño del circuito interno se realiza mediante un lenguaje descriptor de *hardware*, siendo este posteriormente convertido a un fichero binario (denominado *bitstream*), que contiene toda la configuración interpretable por la FPGA, permitiendo que realice las conexiones deseadas.

Para el presente caso, se ha utilizado el lenguaje descriptor *Verilog*. Se ha seguido un tutorial incremental[20], en el que los diversos conceptos se han introducido paulatinamente, refinando y mejorando los anteriores.

5.3. Uso de *software* libre

Durante los cerca de 35 años que llevan existiendo las FPGAs, los fabricantes han mantenido cerrada toda información sobre el *software* y las especificaciones de sus binarios *bitstreams*. Lo que ha permitido que dichos fabricantes tengan un control absoluto sobre el método de

diseño que se debe utilizar y los equipos compatibles. Esto ha disminuido la capacidad del desarrollador final de usar FPGAs y cerrado las puertas a nuevos.

Estos últimos años, han empezado a surgir proyectos, que por medio de ingeniería inversa, han permitido poder utilizar con herramientas totalmente libres las FPGA. Un gran hito a destacar es la introducción en el año 2015 del proyecto *IceStorm*, liderado por *Clifford Wolf*.

El uso de *software libre* trae consigo los siguiente beneficios:

- **Acceso libre al conocimiento.** Sin depender completamente del fabricante, se dispondría de un gran abanico de fuentes de información de los que poder obtener cualquier conocimiento relacionado, fomentando su estudio y mejora. Todo ello sin variar el resultado del producto.
- **Mayor nivel de participación.** Se puede pasar de ser usuario de los productos, a participar directamente en su evolución y mejora.
- **Autonomía.** Los desarrolladores pueden desarrollar sus sistemas sin la imposición de los criterios y requisitos del fabricante.
- **Nuevas aplicaciones.** Al no depender de las previsiones del fabricante, pueden surgir multitud de nuevas aplicaciones sin ningún tipo de restricción.

5.4. Problemas encontrados durante la elaboración del proyecto

Durante el desarrollo del proyecto, han surgido varios problemas tanto intrínsecos como extrínsecos, los cuales han tenido que ser resueltos o mitigados de diversas formas.

5.4.1. Fallos intrínsecos

■ Metodología inicial.

Durante las primeras iteraciones de desarrollo de módulos, no se había definido una metodología a utilizar, lo que se supuso ineficiencias y errores continuos. Tras definir la metodología expuesta en el capítulo 4 y replantear varias iteraciones, se pudo continuar sin problema alguno.

■ Inicialización de bloques BRAM.

Debido un entendimiento escaso del método de Inicialización de los bloques BRAM de la FGPA, se produjeron errores al almacenar datos. Tras una lectura más detallada de la librería de referencia[12] del fabricante *Lattice*, se pudo solucionar el problema.

5.4.2. Fallos extrínsecos

Debido al continuo desarrollo de la herramienta de sintetizado *Yosys*, han surgido en varias ocasiones varios errores graves que han obligado a pausar el proyecto, o a buscar alternativas, lo que ha supuesto un trabajo adicional no medible en los resultados finales. Hay que destacar los siguientes.

■ Sintetizados incorrectos.

En las primeras versiones de *Yosys* usadas, en varias ocasiones, al crear el binario, este no funcionaba completamente, teniendo que probar distintas opciones de síntesis.

- **Error con las puertas triestato.**

En una de las actualizaciones de *Yosys* que fueron surgiendo a lo largo del desarrollo del TFG, a la hora de inicializar los puertos de la FPGA de forma bidireccional, es decir, con entradas/salidas triestato, el sintetizado fallaba. Este problema se tuvo que solventar inicializando dichos puertos de forma manual. Tras varias actualizaciones, se solucionó el problemas, pudiendo volver a utilizar el método de inicialización del principio.

Capítulo 6

Resultados *hardware*

Partiendo del esquema dado de analizadores USB *hardware* (figura 6.1), se aprecian dos partes implicadas en la captura, por un lado el encargado de capturar la trama, y por otro, el encargado de controlar dicha captura y almacenar los resultados. En este capítulo se van a comentar los resultados de la primera, incluyendo además varias imágenes que complementen las explicaciones dadas.

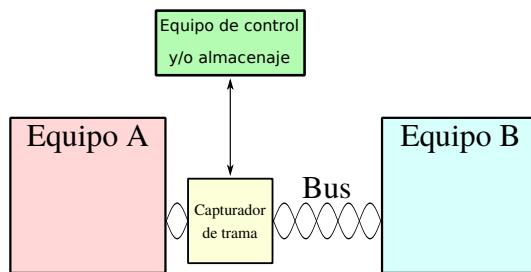


Figura 6.1: Esquema de analizadores *hardware*

6.1. Componentes utilizados

En la figura 6.2 se muestra el resultado *hardware* del presente trabajo, este a su vez, está formado por los siguientes componentes.

1. Placa de desarrollo *IceStick* [14] (Véase figura 6.3).

Se trata de una placa de desarrollo, que incorpora, sin contar con todos los conectores, indicadores *LEDs*, elementos pasivos y componentes de regulación, la *FPGA iCE40HX-1k*[13] del fabricante *Lattice*¹, memoria SPI de 32MBit para almacenar el sintetizado generado, conversor USB a doble puerto de comunicación FIFO *FTDI 2232H*[7] para comunicarse con el PC y oscilador de 12MHz con el que referenciar ciertas partes del circuito.

2. Módulo encargado de la capa física USB.

PCB que incorpora el circuito integrado *USB3300*[14] del fabricante *Microchip*. Este se encarga de manejar la capa física del bus USB, comunicándose con la FPGA por medio del protocolo ULPI[1].

En la figura 6.4, se aprecia que dicha PCB incluye dos conectores USB, uno tipo A hembra, y otro tipo mini-B hembra. Sus señales de datos están interconectadas, por lo que es

¹Página web del producto: <https://www.latticesemi.com/Products/FPGAandCPLD/iCE40.aspx>

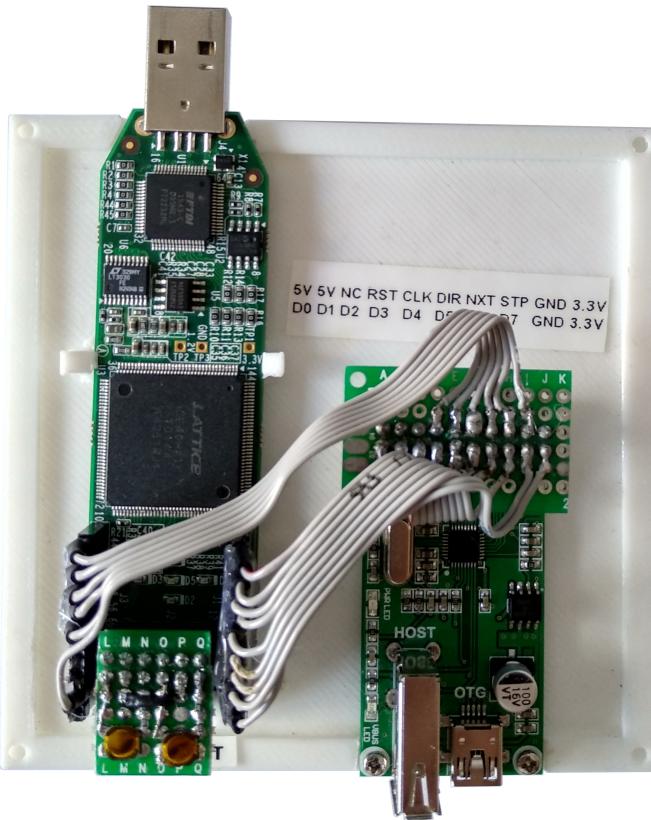


Figura 6.2: Sistema de captura final

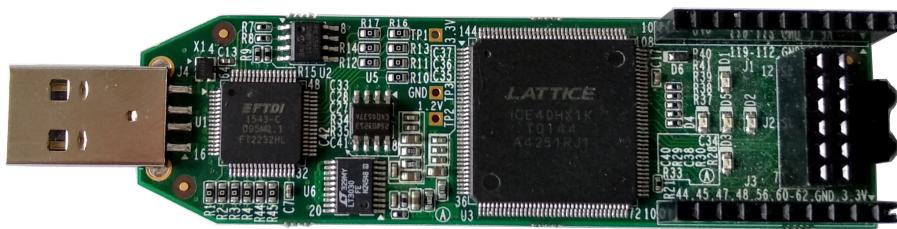


Figura 6.3: Placa de desarrollo *IceStick*

en este punto donde ambos extremos del bus a analizar se unen, pudiendo capturar la trama sin interrumpir la conexión, debido a que el integrado *USB3300* se puede configurar para mantener sus patillas de datos en alta impedancia.



Figura 6.4: PCB con el integrado *USB3300*

3. Cableado de unión (Figura 6.6(a)).

Para conectar las dos placas, se utilizan varios cables entre el conector del módulo con el integrado *USB3300* y los conectores laterales de la placa *IceStick*.

La *FPGA iCE40HX-1k* posee 4 bancos de señales de entrada/salida, para evitar posibles retrasos en las señales^[4], los 8bits de datos paralelos se conectan al banco 0 (pines del 112 al 119), mientras que el resto de señales ULPI (DIR, STP, RST y NXT) al banco 2.

4. Pulsadores externos (Figura 6.6(b)).

Se han incluido dos botones auxiliares externos, con los que poder tanto reiniciar el sistema en su totalidad, como enviar un *byte* de prueba por el puerto serie. Su implementación se debe a que en la fase de diseño era necesario reiniciar el sistema repetidamente, por lo que introduciendo el pulsador de reinicio, se agilizaba la tarea.

Las señales son activas a nivel bajo, por lo que se mantienen siempre a nivel alto por medio de unas resistencias de *Pull-Up* (Figura 6.5).

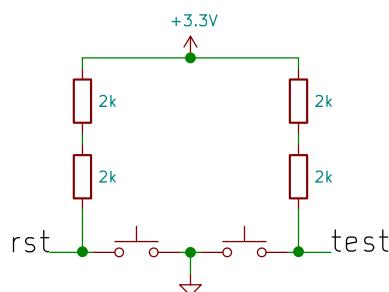


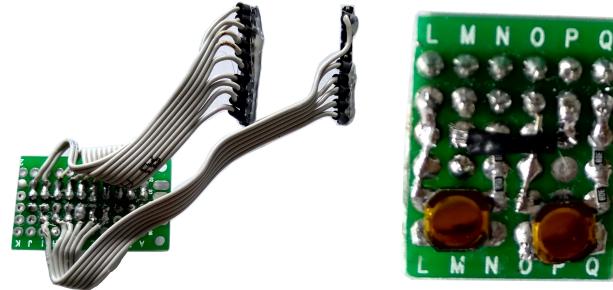
Figura 6.5: Esquema de los botones auxiliares

5. Base impresa en 3D (Figura 6.6(c)).

Se ha diseñado con *SolveSpace*², y posteriormente impreso, una pequeña base en 3D, en

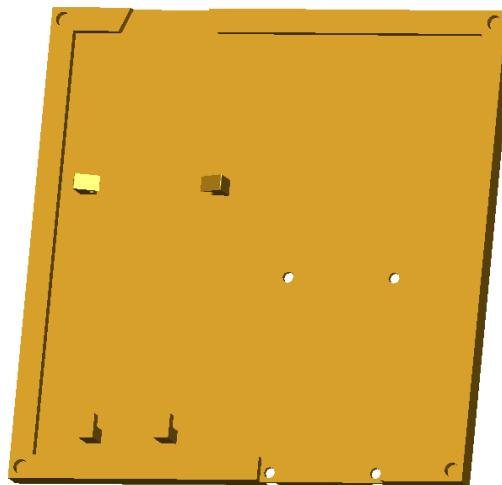
²Herramienta de diseño paramétrico CAD 2D/3D. <http://solvespace.com>

la cual tener las dos placas unidas y organizadas.



(a) Cables de interconexión

(b) Botones auxiliares



(c) Base 3D

Figura 6.6: Resto de elementos usados en el sistema de captación

6.2. Precio final

Uno de los grandes pilares del trabajo, era la realización de un producto lo más económico posible. Su precio final ha sido aproximadamente de 41€, lo que supone una gran diferencia respecto a los productos comerciales. Comparado con el sistema *USB Sniffer* de *Ultra-EMBEDDED*, que posee características similares, se ha conseguido una reducción de 37€ (un 47 % menos).

En la tabla 6.1 se desglosa el precio final según los componentes utilizados.

Tabla 6.1: Presupuesto del sistema

Componente	Unidades	Precio
Placa de desarrollo iCEstick	1	30€
Placa con integrado USB3300	1	7€
Base impresa en 3D	1	2€
Cableado de unión x10	2	1€
Resto de materiales (pulsadores, estaño)	1	1€
Precio aproximado total:		41€

6.3. Módulos integrados en la FPGA

Según los elementos *hardware* usados, y teniendo en cuenta los requerimientos definidos en el capítulo 3, se han diseñado e implementado los siguientes módulos en la FPGA, todos ellos en lenguaje descriptor de *hardware* Verilog. Todos los módulos se sitúan en el directorio `.\VCEstick\USB3300_sniffer\modules\` del repositorio.

Nota. En el anexo A se muestran de forma gráfica todas las relaciones existentes entre los módulos, mientras que en el anexo B se plasman sus máquinas de estados *Mealy* internas.

6.3.1. Módulo divisor de reloj (*clk_div*)

Módulo que divide un reloj de entrada, tal que la frecuencia de salida sea $f_{out} = \frac{f_{in}}{2^n}$, donde n es el número de *Flip-Flops* tipo D utilizados, tal como se aprecia en la figura 6.7. En el listado 6.1 se plasman los parámetros, entradas y salidas del módulo.

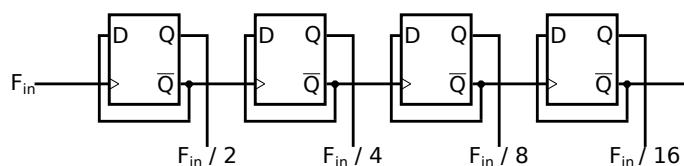


Figura 6.7: Divisor de reloj con *Flip-Flops*

A partir de él se generan tres relojes. Los dos primeros se basan en el reloj de $12MHz$ procedente de la placa *iCEstick*, dividiéndolo por un lado 24 veces para obtener la señal *clk_div_ice* (frecuencia de $0,7Hz$ y periodo de $1,4s$), y por otro lado 19 veces para obtener la señal *clk_debounce* (frecuencia de $22,9Hz$ y periodo de $43ms$). El tercer reloj se basa en el reloj de $60MHz$ del módulo *USB3300*, y es dividido 24 para obtener la señal *clk_div_ULPI* (frecuencia de $3,6Hz$ y periodo de $0,3s$).

Los relojes *clk_div_ice* y *clk_div_ULPI* tienen la única funcionalidad de comprobar el funcionamiento del sistema por medio de los LEDs integrados en la placa. El reloj *clk_debounce* es usado en el módulo *btn_debouncer*, comentado más adelante.

Listado 6.1: Parámetros, entradas y salidas del módulo *clk_div*.

```

1 /*
2 *
3 * Módulo clk_div
4 *
5 * Parámetros:
6 * - DIVIDER. Número de veces que el reloj de referencia es dividido. f_clk_out = f_clk_in / (2^
7 * DIVIDER)
8 *
9 * Entradas:
10 * - enable. Cuando esta señal esté a nivel alto, el módulo estará activo, y en caso contrario la
11 * salida estará siempre a nivel bajo.
12 * - clk_in. Reloj de referencia a partir del cual se genera el de salida.
13 *
14 * Salidas:
15 * - clk_out. Señal de reloj generada con la frecuencia deseada.
16 * - clk_pulse. Señal con la misma frecuencia que el reloj de salida, pero con un ancho de pulso
17 * igual al del reloj de entrada.
18 */

```

6.3.2. Módulo de generación de reloj de baudios (*clk_baud_pulse*)

De igual manera que el módulo *clk_div*, este también genera un reloj, pero en esta ocasión en vez de utilizar directamente *Flip-Flops*, se utiliza un contador, lo que permite una mayor precisión en la salida. El ancho de pulso resultante es igual al del reloj de entrada. En el listado 6.2 se plasman los parámetros, entradas y salidas del módulo.

Este reloj generado, configurable por medio de un parámetro, es usado para controlar la velocidad (baudios) de lectura y escritura en el módulo de comunicación serie, comentado más adelante.

Listado 6.2: Parámetros, entradas y salidas del módulo *clk_baud_pulse*.

```

1  /*
2   *
3   * Módulo clk_baud_pulse
4   *
5   * Parámetros:
6   * - COUNTER_VAL. Valor optimo a contar para generar el pulso deseado.
7   * - PULSE_DELAY. Número de retrasos en la señal de salida antes producir un pulso.
8   *
9   * Entradas:
10  * - enable. Cuando esta señal esté a nivel alto, el módulo estará activo, y en caso contrario la
11  *   salida estará siempre a nivel bajo, reiniciando además sus registros internos.
12  * - clk_in. Reloj de referencia a partir del cual se genera el de salida.
13  *
14  * Salidas:
15  * - clk_pulse. Pulso generado.
16  */

```

6.3.3. Módulo de memoria FIFO (*FIFO_BRAM_SYNC*)

Módulo que de forma sincrona en lectura y escritura al reloj de entrada, es capaz de almacenar datos en los bloques de RAM internos de la FPGA, de tal manera que el primer dato en ser introducido sea el primero en ser extraído. En el listado 6.3 se plasman los parámetros, entradas y salidas del módulo.

Se han creado dos variantes, *FIFO_BRAM_SYNC* y *FIFO_BRAM_SYNC_CUSTOM*, ambas parten de la misma base de funcionamiento, pero la segunda permite utilizar más de un único bloque de RAM interna.

Listado 6.3: Parámetros, entradas y salidas del módulo *FIFO_BRAM_SYNC_CUSTOM*.

```

1  /*
2   *
3   * Módulo FIFO_BRAM_SYNC_CUSTOM
4   *
5   * Parámetros:
6   * - ALMOST_FULL. Porcentaje mínimo de la memoria que hace que se active la señal wr_almost_full.
6   *   Por defecto vale 0.9.
7   * - ALMOST_EMPTY. Porcentaje máximo de la memoria que hace que se active la señal rd_almost_empty.
7   *   Por defecto vale 0.1.
8   * - DATA_WIDTH. Tamaño de cada valor almacenado. Por defecto vale 8bits.
9   * - FIFO_SIZE_ML. Número de bloques de RAM a usar. Por defecto se utiliza uno.
10  *

```

```

11  * Entradas:
12  * - rst. Señal de reinicio, activa a nivel bajo.
13  * - clk. Reloj de referencia.
14  * - wr_dv. Señal que indica que los datos de entrada deben ser almacenados.
15  * - wr_DATA. Datos a ser almacenados. Debe tener el mismo ancho que el parámetro DATA_WIDTH.
16  * - rd_en. Señal que indica que se desea leer los datos almacenados.
17  *
18  * Salidas:
19  * - wr_full. Señal que indica que la memoria esta llena. Futuras operaciones de escritura serán
19  ignoradas.
20  * - wr_almost_full. Señal que indica que la memoria esta casi llena.
21  * - rd_DATA. Datos leidos de la memoria.
22  * - rd_empty. Señal que indica que la memoria esta vacía. Futuras operaciones de lectura serán
22  ignoradas.
23  * - rd_almost_empty. Señal que indica que la memoria esta casi vacía.
24  *
25 */

```

6.3.4. Módulo de registro de desplazamiento universal (*shift_register*)

Módulo capaz de desplazar tanto a izquierda como a derecha la información almacenada, y que a su vez, permite una carga y lectura en paralelo. En el listado 6.4 se plasman los parámetros, entradas y salidas del módulo.

La finalidad de este submódulo es poder convertir datos de serie a paralelo o de paralelo a serie, cuando ocurra una recepción o transmisión de datos serie respectivamente.

En la figura 6.8 se muestra, mostrando sus señales, el funcionamiento del registro de desplazamiento diseñado, en esta ocasión, con desplazamiento hacia la derecha.

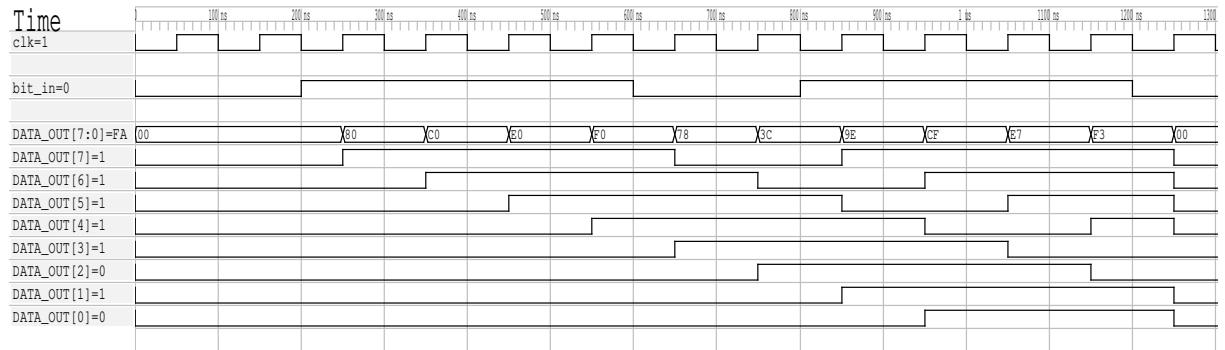


Figura 6.8: Ejemplo de funcionamiento del registro de desplazamiento hacia la derecha.

Listado 6.4: Parámetros, entradas y salidas del módulo *shift_register*.

```

1 /*
2 *
3 * Módulo shift_register
4 *
5 * Parámetros:
6 * - BITS. Tamaño en bits del registro de desplazamiento. El tamaño total será de BITS + 1 (bit de
6 * salida).
7 *
8 * Entradas:
9 * - clk. Reloj de referencia. Todas las operaciones se ejecutarán en el flanqueo de subida.
10 * - rst. Señal de reinicio, activa a nivel bajo.
11 * - bit_in. Bit a ser desplazado dentro del registro (modos 01 y 10).

```

```

12 * - DATA_IN. Datos a ser cargados de forma paralela (modo 11).
13 * - mode. Selector de operación a realizar [2 bits].
14 *     > 00. No hacer nada.
15 *     > 01. Desplazamiento a izquierda.
16 *     > 10. Desplazamiento a derecha.
17 *     > 11. Carga paralela.
18 *
19 * Salidas:
20 * - bit_out. Bit de salida tras el desplazamiento (modos 01 y 10).
21 * - DATA. Datos paralelos almacenados.
22 *
23 */

```

6.3.5. Módulo de comunicación serie (*UART*)

Sistema capaz de generar y recibir señales compatibles con una comunicación bidireccional serie 8N1³, consiguiendo una tasa de transferencia estable máxima de 3750000 *bauds*. Además, tanto para la lectura como para la escritura, se han incorporado unos *buffers* basados en la memoria FIFO anterior, de 512 *bytes* cada uno. En el listado 6.5 se plasman los parámetros, entradas y salidas del módulo.

Listado 6.5: Parámetros, entradas y salidas del módulo UART.

```

1 /*
2 *
3 * Módulo UART
4 *
5 * Parámetros:
6 * - BAUDS. Valor óptimo a contar para generar los baudios deseado.
7 *
8 * Entradas:
9 * - rst. Señal de reinicio, activa a nivel bajo.
10 * - clk. Reloj de referencia.
11 * - Rx. Datos serializados de entrada.
12 * - I_DATA. Señal de 8bits que contiene los datos a ser enviados.
13 * - send_data. Señal que inicializa una transmisión de datos.
14 * - NxT. Señal para extraer un byte de datos del buffer de lectura.
15 *
16 * Salidas:
17 * - Tx. Datos serializados de salida.
18 * - clk_Rx. Reloj, a la velocidad fijada por BAUDS, usado para recibir los datos.
19 * - clk_Tx. Reloj, a la velocidad fijada por BAUDS, usado para transmitir los datos.
20 * - O_DATA. Señal de 8bits que contiene los datos recibidos.
21 * - TiP. Señal que indica que hay una transmisión en proceso.
22 * - NrD. Señal que indica la llegada de nuevos datos. Esta estará activa un pulso de clk.
23 * - Tx_FULL. Señal que indica que el buffer de transmisión interno está lleno.
24 * - Rx_FULL. Señal que indica que el buffer de recepción interno está lleno.
25 * - Rx_EMPTY. Señal que indica que el buffer de recepción interno está vacío.
26 *
27 */

```

Este módulo se ha dividido a su vez en los siguientes submódulos.

- **Submódulo de recepción serie (*UART_Rx*).**

Está continuamente esperando a que la señal de entrada de datos serie pase de un nivel alto

³8N1: 8 *bits* de datos, sin *bit* de paridad y un *bit* de parada

a otro bajo, y una vez detectada esa caída, empezar a almacenar los *bits* que conforman la trama a la velocidad dictada por el reloj de baudios. En la figura 6.9 se muestra su flujo de funcionamiento.

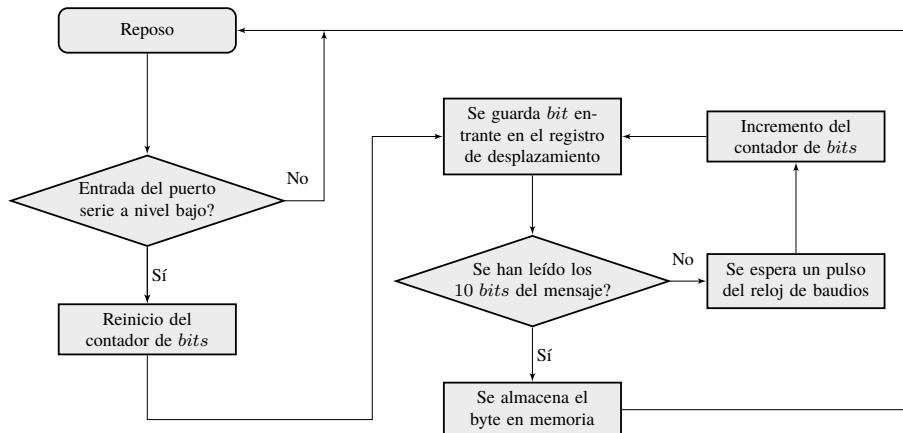


Figura 6.9: Diagrama de funcionamiento del submódulo de recepción serie (UART_Rx)

■ Submódulo de emisión serie (UART_Tx).

En este caso, cuando internamente se quiera enviar un mensaje, se exportan sucesivamente los 10*bits* que conforman la trama a la velocidad dada por el reloj de baudios. En la figura 6.10 se muestra su flujo de funcionamiento.

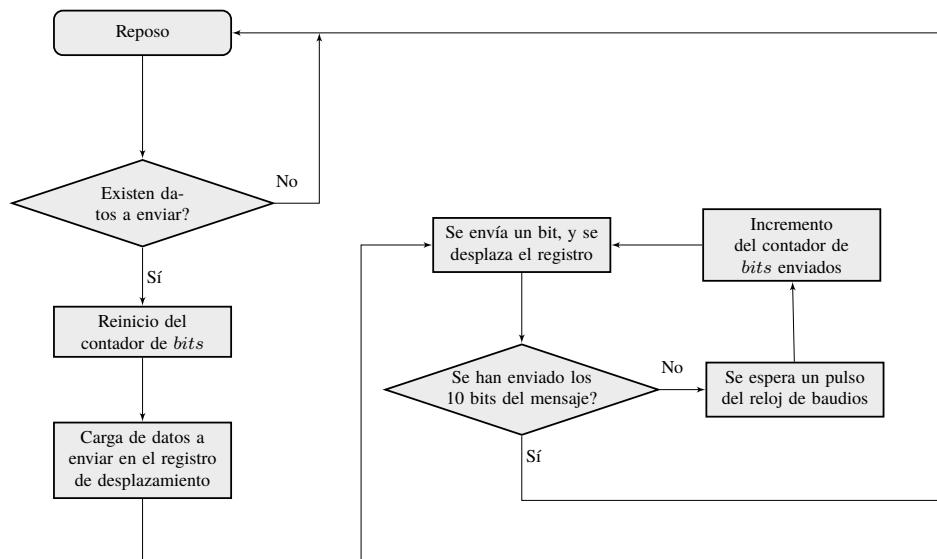


Figura 6.10: Diagrama de funcionamiento del submódulo de emisión serie (UART_Tx)

6.3.6. Módulo de comunicación ULPI

Sistema, que de forma síncrona al reloj de 60MHz generado por el integrado USB3300, es capaz tanto de procesar como de generar señales ULPI, tal como se contempla en su especificación[1]. En el listado 6.6 se plasman las entradas y salidas del módulo.

Listado 6.6: Entradas y salidas del módulo ULPI.

```

2  *
3  * Módulo ULPI
4  *
5  * Entradas:
6  * - rst. Señal de reinicio, activa a nivel bajo.
7  * - clk_ice. Reloj interno iCEstick de 12MHz.
8  * - clk_ULPI. Reloj externo de 60MHz.
9  * - PrW. Señal que activa la escritura de registro.
10 * - PrR. Señal que activa la lectura de registro.
11 * - ADDR. Dirección de 6bits que indica donde se van a escribir/leer los datos.
12 * - REG_VAL_W. Valor a ser escrito en el registro ULPI.
13 * - DATA_re. Señal que extrae un byte de los datos de captura.
14 * - INFO_re. Señal que extrae un paquete con la información de la última captura.
15 * - DIR. Señal del bus ULPI (DIRection).
16 * - NXT. Señal del bus ULPI (NeXT).
17 * - DATA_in. Señal de datos de entrada del bus ULPI.
18 *
19 * Salidas:
20 * - status. Señal que indica en que estado se encuentra el módulo (lectura, escritura, etc..)
21 * - busy. Señal que indica que el módulo está ocupado.
22 * - REG_VAL_R. Valor del registro leído.
23 * - RxCMD. Señal que contiene información relevante del bus USB.
24 * - RxLineState. Información del bus USB almacenada en RxCMD.
25 * - RxVbusState. Información del bus USB almacenada en RxCMD.
26 * - RxActive. Información del bus USB almacenada en RxCMD.
27 * - RxError. Información del bus USB almacenada en RxCMD.
28 * - RxHostDisconnect. Información del bus USB almacenada en RxCMD.
29 * - RxID. Información del bus USB almacenada en RxCMD.
30 * - USB_DATA. Datos capturados USB.
31 * - USB_INFO_DATA. Información de la última captura USB (RxCMD y tamaño).
32 * - DATA_buff_full. Señal que indica que el buffer de captura está lleno.
33 * - DATA_buff_empty. Señal que indica que el buffer de captura está vacío.
34 * - INFO_buff_full. Señal que indica que el buffer de información está lleno.
35 * - INFO_buff_empty. Señal que indica que el buffer de información está vacío.
36 * - DATA_out. Señal de datos de salida del bus ULPI.
37 * - STP. Señal del bus ULPI (SToP).
38 * - U_RST. Señal de reinicio del integrado USB3300.
39 */

```

De todos los modos de funcionamiento de dicho bus, y tal como se ha comentado en el capítulo 3 de objetivos, se han creado únicamente los submódulos encargados de leer y escribir registros, y el modo de recibir datos USB.

En la siguiente lista se nombran los diversos submódulo que lo forman, inicializados posteriormente todos ellos en un mismo archivo siguiendo el diagrama mostrado en la figura 6.11.

- **Submódulo de escritura de registros (*ULPI_REG_WRITE*).**

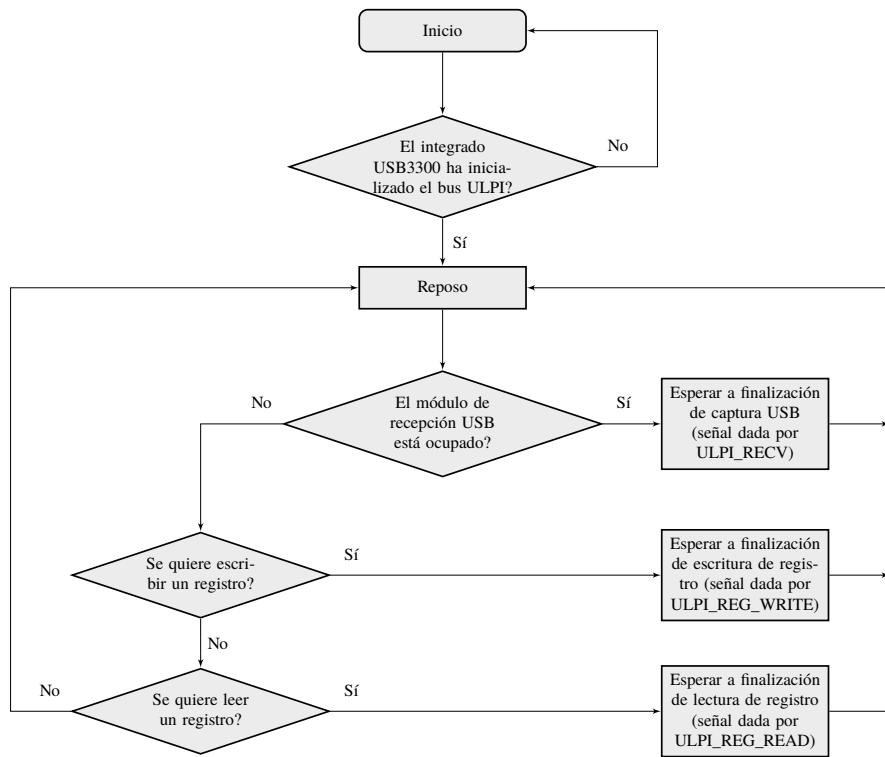
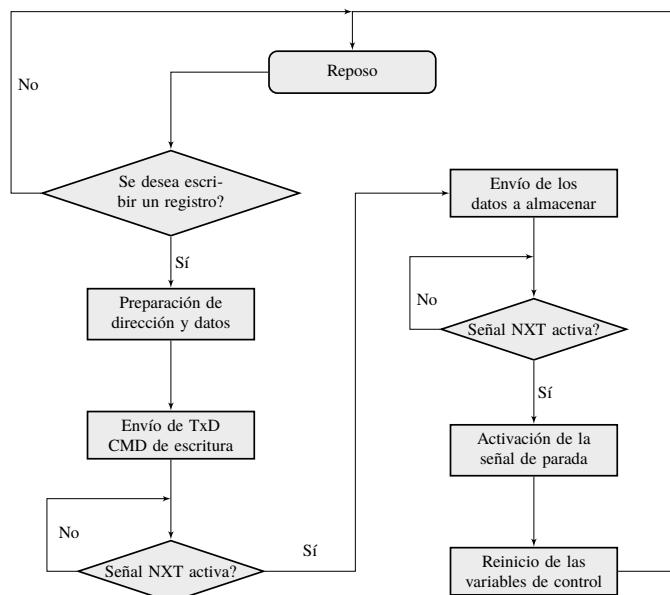
Dándole una dirección de 6bits y un byte de datos, este módulo genera las señales necesarias para la escritura de registros. En la figura 6.12 se muestra su flujo de funcionamiento.

- **Submódulo de lectura de registros (*ULPI_REG_READ*).**

Dándole en esta ocasión únicamente la dirección de 6bits, este genera las señales necesarias para que el integrado *USB3300* envíe el valor del registro solicitado, valor que debe ser leído por este modulo. En la figura 6.13 se muestra su flujo de funcionamiento.

- **Submódulo de obtención de datos USB (*ULPI_RECV*).**

Cada vez que el integrado *USB3300* quiera enviar una captura de datos, pone a nivel alto la señal DIR y empieza a transmitir los datos. Este módulo está continuamente esperando dicho cambio, a partir del cual obtiene, clasifica y almacena los datos entrantes. En la figura 6.14 se muestra su flujo de funcionamiento.

**Figura 6.11:** Diagrama de funcionamiento de la unión de módulos ULPI**Figura 6.12:** Diagrama de funcionamiento del submódulo de escritura de registros ULPI (ULPI_REG_WRITE)

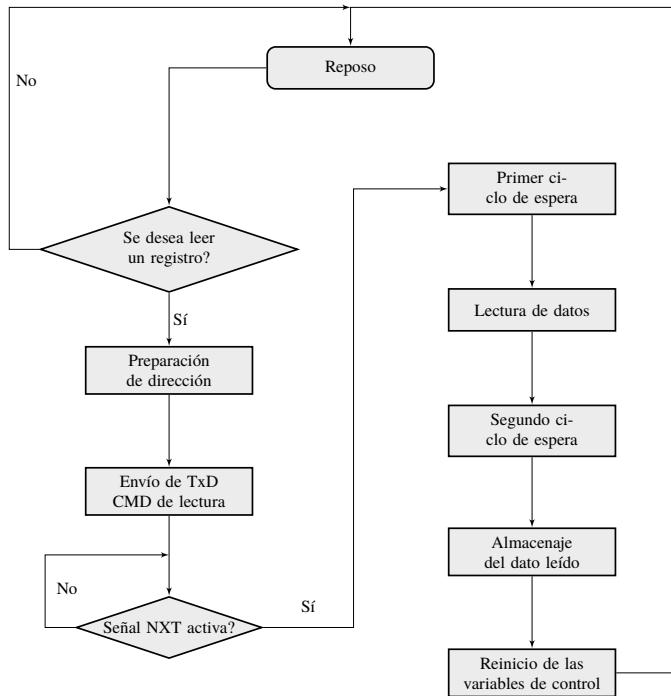


Figura 6.13: Diagrama de funcionamiento del submódulo de lectura de registros ULPI (*ULPI_REG_READ*)

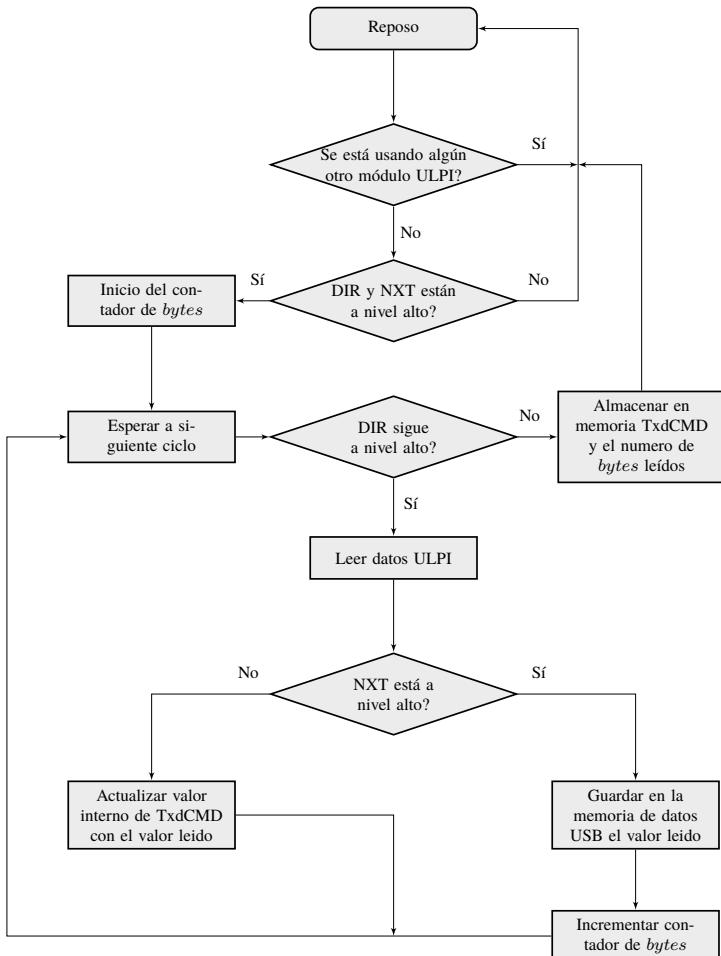


Figura 6.14: Diagrama de funcionamiento del submódulo de recepción USB (*ULPI_RECV*)

6.3.7. Módulo de procesado y almacenaje de comandos entrantes (*ULPI_op*)

Se ha diseñado e implementado un simple protocolo, con el que ser capaz de recibir las ordenes del sistema y de enviar los datos capturados.

Siempre que se desee ejecutar un comando en el sistema de captura, el PC envía 2 *bytes* (YYZZZZZZ_XXXXXXX), separados a su vez en tres grupos, de 2, 6 y 8 *bits* respectivamente. Tal como se recoge en la tabla 6.2, el primer grupo indica que comando se debe realizar, el segundo la dirección en la que realizar dicha operación, y el tercero, los datos a utilizar.

Tabla 6.2: Información de los *bytes* enviados a la *FPGA*.

Bits comando	Bits dirección	Bits datos	Descripción	Respuesta
00	Indiferente	10010110	Activar/desactivar envío de datos USB	–
01	Indiferente	Indiferente	Enviar último valor de registro leído	1 byte
10	Dirección a escribir	Datos a escribir	Escribir registro <i>ULPI</i>	–
11	Dirección a leer	Indiferente	Leer registro <i>ULPI</i>	–

Estos *bytes*, tras ser recibidos por el módulo de comunicación serie, son recogidos por este módulo, el cual los clasifica y almacena hasta su ejecución. En el listado 6.7 se plasman las entradas y salidas del módulo.

Listado 6.7: Entradas y salidas del módulo *ULPI_op*.

```

1 /*
2 *
3 * Módulo ULPI_op
4 *
5 * Entradas:
6 * - rst. Señal de reinicio, activa a nivel bajo.
7 * - clk. Reloj de referencia.
8 * - UART_DATA. Datos del puerto serie.
9 * - UART_Rx_EMPTY. Señal que indica que el puerto serie no tiene datos disponibles.
10 * - op_stack_pull. Señal que indica que se quiere obtener un nuevo valor del buffer de operaciones.
11 *
12 * Salidas:
13 * - UART_NxT. Señal usada para obtener, si fuera posible, un nuevo byte.
14 * - op_stack_msg. Comando completo recibido y almacenado.
15 * - op_stack_full. Señal que indica que el buffer de operaciones está lleno.
16 * - op_stack_empty. Señal que indica que el buffer de operaciones está vacío.
17 *
18 */

```

6.3.8. Módulos de control de botonera (*btn_debouncer* y *signal_trigger*)

Al haber introducido varios botones externos, estos sufren una propiedad física de rebote, por la que el botón, al ser pulsado, salta entre varios estados antes de estabilizarse, produciendo pulsaciones no deseadas. Para solucionarlo, se ha creado un simple módulo que concatena varios *Flip-Flops* tal como se muestra en la figura 6.15.

Por otro lado, como las pulsaciones del usuario tienen una duración mucho mayor a la de un ciclo del reloj de control, se ha creado un segundo módulo que ante la pulsación de larga duración, envía un único pulso con el mismo ancho que el del reloj de entrada.

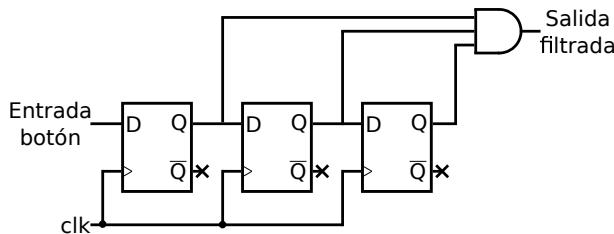


Figura 6.15: Esquema de funcionamiento del módulo btn_debounce

6.3.9. Módulo de control maestro (*main_controller*)

Los módulos anteriormente explicados, no poseen relación entre ellos, por lo que necesitan un módulo extra encargado de temporizar y distribuir que tareas deben realizar. Por tanto, este módulo a partir de las señales generadas por el resto de ellos, prioritiza las tareas de la siguiente manera.

1. Ejecución de un comando de activación/desactivación de envío de datos.
2. Ejecución de un comando de envío del último registro leído.
3. Ejecución de un comando de escritura de registro.
4. Ejecución de un comando de lectura de registro.
5. Envío de datos de captura.
6. Envío de un *byte* de prueba al pulsar el botón externo.

6.4. Simulaciones finales

Con el objetivo de eliminar posibles errores capaces de dañar los propios componentes *hardware*, y para comprobar el correcto funcionamiento del sistema, se han realizado diversas pruebas previas al sintetizado y utilización de la configuración final. Las gráficas resultantes de las simulaciones se pueden ver en el anexo C.

1. Prueba del botón auxiliar de reinicio.

Se simula la pulsación del botón externo de reinicio, poniendo el sistema en su estado inicial.

2. Prueba del botón auxiliar de *test*.

Se simula la pulsación del botón externo de *test*, produciendo un envío de un *byte* de prueba por el puerto serie.

3. Prueba de lectura de registro.

Se simula una petición enviada por el puerto serie que realice una lectura de registro *ULPI*.

4. Prueba de transmisión del último registro leído.

Se simula una petición enviada por el puerto serie para enviar el valor del registro anteriormente leído al PC.

5. Prueba de escritura de registro.

Se simula una petición enviada por el puerto serie que escriba un *byte* en un registro del integrado *USB3300*.

6. Prueba de captación de 6 bytes USB.

Prueba en la que se simula la llegada de 6 *bytes* de datos USB, se comprueba también el correcto guardado de los mismos en la memoria FIFO interna.

7. Prueba de activación de la transmisión de datos capturados.

Se simula una petición enviada por el puerto serie, con la que activar el envío de los datos capturados.

8. Prueba de captación de 4 bytes USB.

De igual manera que la prueba 6, pero enviando esta vez 4 *bytes* con el envío de datos al PC activado.

9. Prueba de captación de cambios de estado del bus USB.

Se simula un cambio de estado USB enviado por el bus *ULPI*, observando si se actualiza en la FPGA.

10. Prueba de finalización de la transmisión de datos capturados.

De igual manera que en la prueba 7 se activaba el envío de datos automático, se realiza la misma petición, desactivándolo en este caso.

Capítulo 7

Resultados *software*

Aun teniendo más importancia la parte *hardware* comentada en el capítulo 6, el sistema no sería funcional sin el *software*, que utilizando los recursos creados, pueda recibir y almacenar la captura. Además, se han creado varias utilidades que han sido ayuda en las fases de diseño *hardware*.

7.1. Aplicaciones de apoyo

A medida que se ha ido desarrollando la parte *hardware*, se han creado varias utilidades en lenguaje *Python*, con las que realizar cálculos de forma rápida o con las que comprobar el funcionamiento de ciertas partes del sistema. Dichas aplicaciones se encuentran en el directorio *.NCEstick\USB3300_sniffer\tools* del repositorio.

Hay que destacar las siguientes:

1. Utilidad de divisiones del reloj (*get_divider.py*).

Utilidad con la que poder realizar de forma rápida varios cálculos relacionados con los relojes de entrada. Por ejemplo, obtener los valores óptimos con los que dividir un reloj para generar unos baudios deseados. En el listado 7.1 se plasma un ejemplo de uso.

Listado 7.1: Ejemplo de la utilidad de divisiones de reloj.

```
1 $ ./get_divider.py
2 Usage: ./get_divider.py [option] arg1 arg2
3 Options:
4   -o: Print the optimal counter value for a given clock (arg1) and Serial baudrate (arg2)
      . [Default]
5   -b: Print the minimal divider value for a given clock (arg1) and Serial baudrate (arg2)
      .
6   -d: Print the minimal divider value for a given clock (arg1) and time (arg2).
7   -t: Print the time (and serial baudrate) for a given clock (arg1) and divider (arg2).
8   -a: Print, for a given clock (arg1), all the possible periods that can be generated
      between 0 and arg2.
9   -c: Print the clock for a given time (arg1) and divider (arg2).
10 $ ./get_divider.py -o 12000000 115200
11   Optimal counter value: 105
12 $ ./get_divider.py -b 12000000 9600
13   Recomended divider (with 18% error): 10 [8.53333333333334e-05s]
14   10 [8.53333333333334e-05s] - [0.0001041666666666667s] - 11 [0.000170666666666668s]
```

2. Utilidad de generación de baudios (*gen_bauds.py*).

Pequeña aplicación que, para los *baudios* más comunes, genere varias definiciones en lenguaje *Verilog* a usar por el módulo encargado de generar el reloj de baudios. En el listado 7.2 se muestra un ejemplo de la salida de la aplicación ante un reloj de $60MHz$.

Listado 7.2: Ejemplo de la utilidad de generación de baudios ante $60MHz$ de entrada.

```

1 $ ./gen_bauds.py
2 `define B921600 66
3 `define B460800 131
4 `define B256000 235
5 `define B230400 261
6 `define B153600 391
7 `define B128000 469
8 `define B115200 521
9 `define B57600 1042
10 `define B56000 1072
11 `define B38400 1563
12 `define B28800 2084
13 `define B19200 3125
14 `define B14400 4167
15 `define B9600 6250
16 `define B4800 12500
17 `define B2400 25000
18 `define B1200 50000
19 `define B600 100000
20 `define B300 200000
21 `define B110 545455

```

3. Utilidad de control serie (*serial_control.py*).

Aplicación, que usando una librería encargada de controlar puertos serie, realiza varias comprobaciones del protocolo creado. Esta se puede configurar para abrir el puerto serie a cualquier velocidad, y poder enviar comandos de prueba.

7.2. Información de la aplicación

Tal como se ha comentado al principio de este capítulo, aun teniendo mayor carga la parte *hardware*, es esencial implementar una aplicación que se comunique con la *FPGA* y permita su funcionamiento.

Para ello, utilizando lenguaje C, se ha creado una aplicación que dando opciones de control con una simple interfaz, es capaz de enviar comandos o recibir datos USB, y a su vez, es capaz de almacenarlos para su posterior estudio. A grandes rasgos, en la figura 7.2 se muestra un esquema de funcionamiento de la aplicación.

Esta aplicación se ha dividido de la siguiente manera:

1. Hilo encargado de controlar las entradas de usuario.

Tanto la configuración a usar para conectarse a la *FPGA* (puerto serie y su velocidad), como los comandos a enviar, son introducidos al programa por medio de un simple menú basado en texto desde la terminal. Este permite:

- Configurar el puerto serie (puerto a usar y velocidad en baudios).
- Abrir/cerrar el puerto serie seleccionado.
- Realizar una operación de lectura de registro.
- Realizar una operación de escritura de registro.

- Activar/desactivar el envío de datos USB al PC.

En la figura 7.1 se muestra dicho menú, y a su vez, se ejemplifica un proceso de escritura y posterior lectura del registro con dirección *0x16* (registro que según la hoja de características del integrado *USB3300*[17] es de libre utilización).

```
## MAIN MENU ##
0. Config port.
1. Close port [/dev/ttyUSB1 @ 3750000].
2. Command: REG READ.
3. Command: REG_WRITE.
4. Command: RECV_TOGGLE [Currently OFF].
5. Exit.
> 3

## REGISTER WRITE MENU ##
Last known address: 0x0 (0 - b00000000)
Last known data: 0x0 (0 - b00000000)
0. Write Address [DEC].
1. Write Address [HEX].
2. Write Address [BIN].
3. Write Data [DEC].
4. Write Data [HEX].
5. Write Data [BIN].
6. Perform Register Write.
> 1
Addr> 16
New Address: 0x16
> 4
Data> 19
New Data: 0x19
> 6
Writing register...
The register with address 0x16 has been written with 0x19 (25 - b00011001)
```

(a) Operación de escritura

```
## MAIN MENU ##
0. Config port.
1. Close port [/dev/ttyUSB1 @ 3750000].
2. Command: REG READ.
3. Command: REG_WRITE.
4. Command: RECV_TOGGLE [Currently OFF].
5. Exit.
> 2

## REGISTER READ MENU ##
Last known address: 0x16 (22 - b00010110)
0. Read Address [DEC].
1. Read Address [HEX].
2. Read Address [BIN].
3. Perform Register Read.
> 3
Reading register...
The value of the register with address 0x16 is 0x19 (25 - b00011001)
```

(b) Operación de lectura

Figura 7.1: Ejemplo de escritura de registro y su posterior lectura usando el menú de la aplicación

2. Hilo encargado de gestionar la comunicación con la *FPGA*, así como almacenar la trama capturada.

Haciendo uso de la librería *libserialport*¹, se establece una comunicación serie entre la *FPGA* por la que enviar las solicitudes de comandos, y recibir los datos de captura. Al finalizar el proceso de recepción de datos, es este hilo también el encargado de guardar la captura en un archivo para su posterior análisis.

¹Librería compatible con sistemas operativos Linux, Mac OS X, FreeBSD, Windows y Android. Página web de referencia: <https://sigrok.org/wiki/Libserialport>

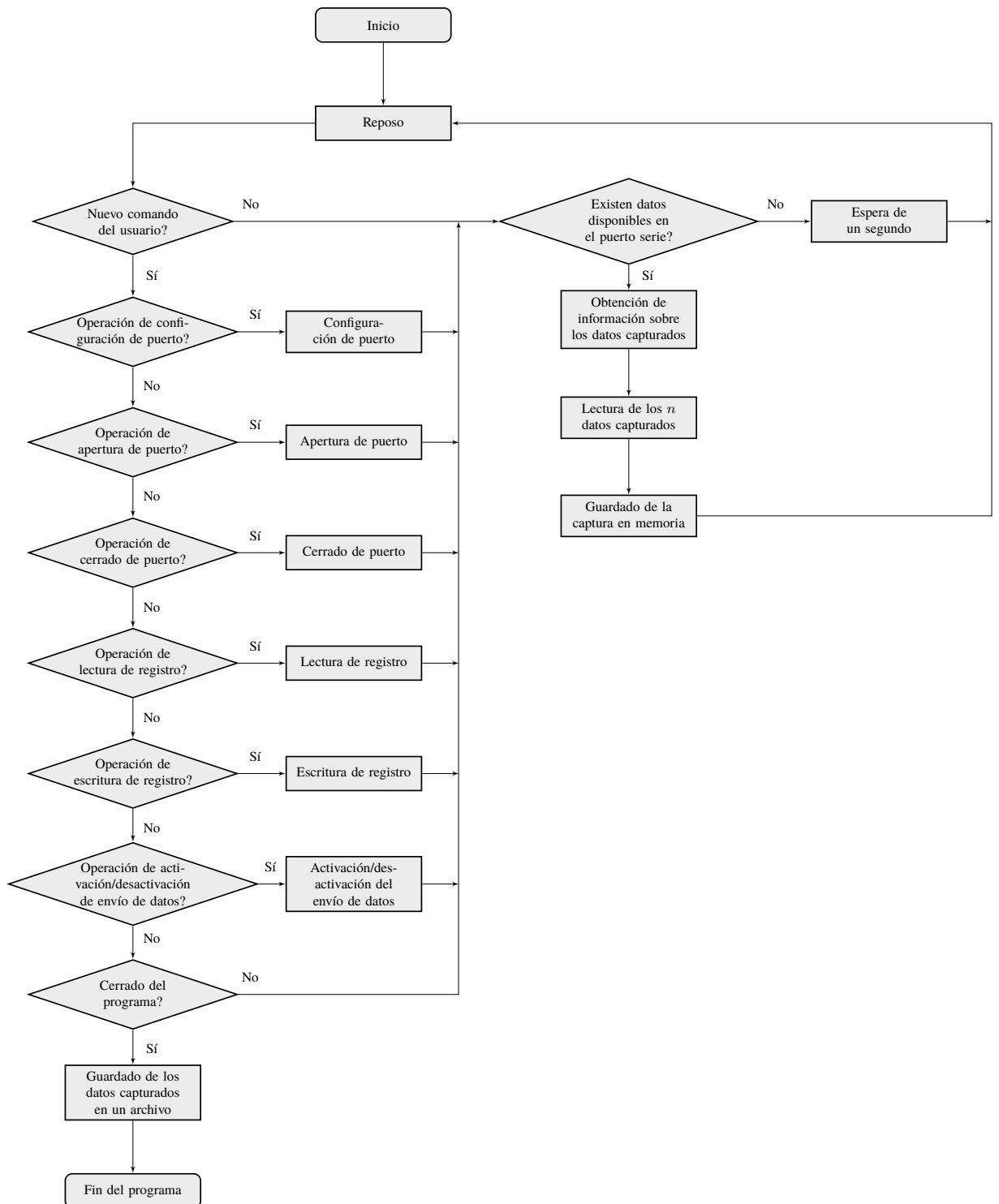


Figura 7.2: Diagrama de funcionamiento de la aplicación

7.3. Información del archivo generado

Se ha elegido almacenar la información capturada en un archivo formato *JSON*². Para cada paquete capturado se almacena el último estado del bus conocido (*TxCMD*), la longitud de paquetes capturados y los propios datos obtenidos. En la figura 7.3 se muestra la estructura utilizada en el archivo.

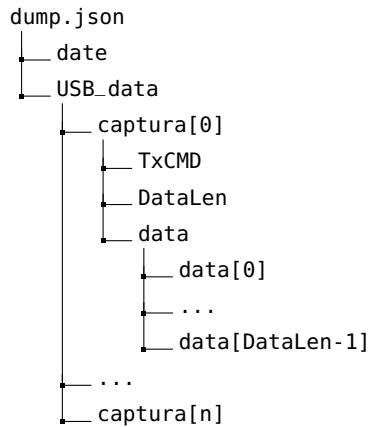


Figura 7.3: Estructura del archivo *JSON* donde se almacena la captura

Para poder analizar la captura en la aplicación de análisis *Wireshark*, se ha creado otra utilidad, que a partir del archivo *JSON* generado anteriormente, lo transforme a *PCAP*. Esta transformación no sigue la estructura de ningún disector^[29] actual, por lo que en *Wireshark* solo se consigue visualizar los datos de una forma más estructurada sin dar información adicional.

²Formato de texto, que permite almacenar información de forma estructurada.

Capítulo 8

Conclusiones

En este capítulo se plasman las principales conclusiones del presente trabajo, y a su vez, se proponen posibles mejoras y trabajos futuros de interés.

8.1. Conclusiones

A continuación se detallan las principales conclusiones obtenidas durante la realización de este proyecto:

- Se ha procedido, en un primer lugar, a un análisis de mercado, confirmando la escasez de analizadores USB *hardware* de bajo coste. Reafirmando de esa forma la finalidad del proyecto.
- Se ha procedido a la selección y posterior conexión de los diversos componentes utilizados. Estos incluyen la placa de evaluación *USB3300*, la placa de desarrollo *iCEstick*, cableado de conexión, botonera y una base impresa en 3D.
- Todo ello ha supuesto un precio de aproximadamente 45\$ (40€ al cambio), lo que supone una gran diferencia respecto a los analizadores comerciales, y una reducción de 43\$ respecto al proyecto de *Ultra-Embedded* de similares características.
- Se ha establecido una metodología de diseño y elaboración de módulos, que ha permitido una gran eficiencia en su elaboración, consiguiendo a su vez, una menor probabilidad de fallos y una alta reutilización.
- Para el funcionamiento del sistema, se han diseñado y elaborado varios módulos, incluidos los establecidos en los objetivos, en el lenguaje descriptor de *hardware* Verilog. Todos ellos han utilizado el 63 % de los bloques lógicos y el 100 % de los bloques de RAM disponibles en la FPGA usada.
Principalmente hay que destacar los siguientes módulos.
 1. Memoria FIFO.
 2. Registro de desplazamiento.
 3. Módulo de comunicación ULPI.
 4. Módulo de comunicación serie.
 5. Módulos generadores de relojes.
 6. Controlador maestro del sistema.

- Se ha desarrollado una aplicación con las todas opciones necesarias para poder configurar la FPGA y recibir una captura, siendo esta almacenada en un archivo *JSON* de fácil lectura.
Posteriormente, y para cumplir el objetivo propuesto, se ha creado una herramienta que convierte la trama capturada de formato *JSON* a *PCAP*.
- Como observación final, hay que comentar varios aspectos a destacar en el uso de las FPGAs en general.
 - Permiten una gran flexibilidad a la hora de realizar cualquier diseño de electrónica digital, y a diferencias de otros elementos como microcontroladores, permiten un alto nivel de paralelización de tareas, por lo que son de gran utilidad en ciertos diseños.
 - Como contrapartida, las FPGAs introducen la necesidad de controlar las frecuencias máximas que pueden soportar, y una dificultad extra a la hora de realizar depuraciones.

Con todo lo anteriormente expuesto, se puede afirmar que se han cumplido todos los objetivos establecidos al principio del trabajo, recogidos en el capítulo 3.

8.2. Trabajos futuros

A continuación se enumeran varios trabajos futuros que se pueden realizar para mejorar el resultado del proyecto.

- Debido a la baja cantidad de memoria disponible en la FPGA usada, puede ser de gran interés añadir un módulo de memoria RAM externa, en la que almacenar de forma temporal los datos.
- Implementación de un nuevo sistema de comunicación entre la FPGA y el PC, que prescindiendo del puerto serie actual, permita una mayor tasa de transferencia.
- Diseño de una PCB que unifique los componentes utilizados en una sola placa, y se deshaga de todos aquellos innecesarios, como puede ser el emisor/receptor de infrarrojos de la placa iCEstick. Realizando esta tarea, a parte de obtener un producto más refinado, se puede disminuir aun más el precio final del sistema.
- Creación de un disector para la utilidad de análisis *Wireshark*, que sea capaz de mostrar más detalles de los datos capturados en el archivo *PCAP*.
- Introducción de herramientas de filtrado USB en la propia FPGA, que facilite el análisis. Por ejemplo, se podrían añadir filtros según el PID o la longitud de los datos a capturar.
- Introducción de un sistema que permita saber en que instante temporal a ocurrido la captura de cada paquete de datos. Para ello, se puede añadir por ejemplo un circuito RTC (*Real Time Clock*, o reloj en tiempo real en español) que añada una marca temporal a cada paquete de datos.

Capítulo 9

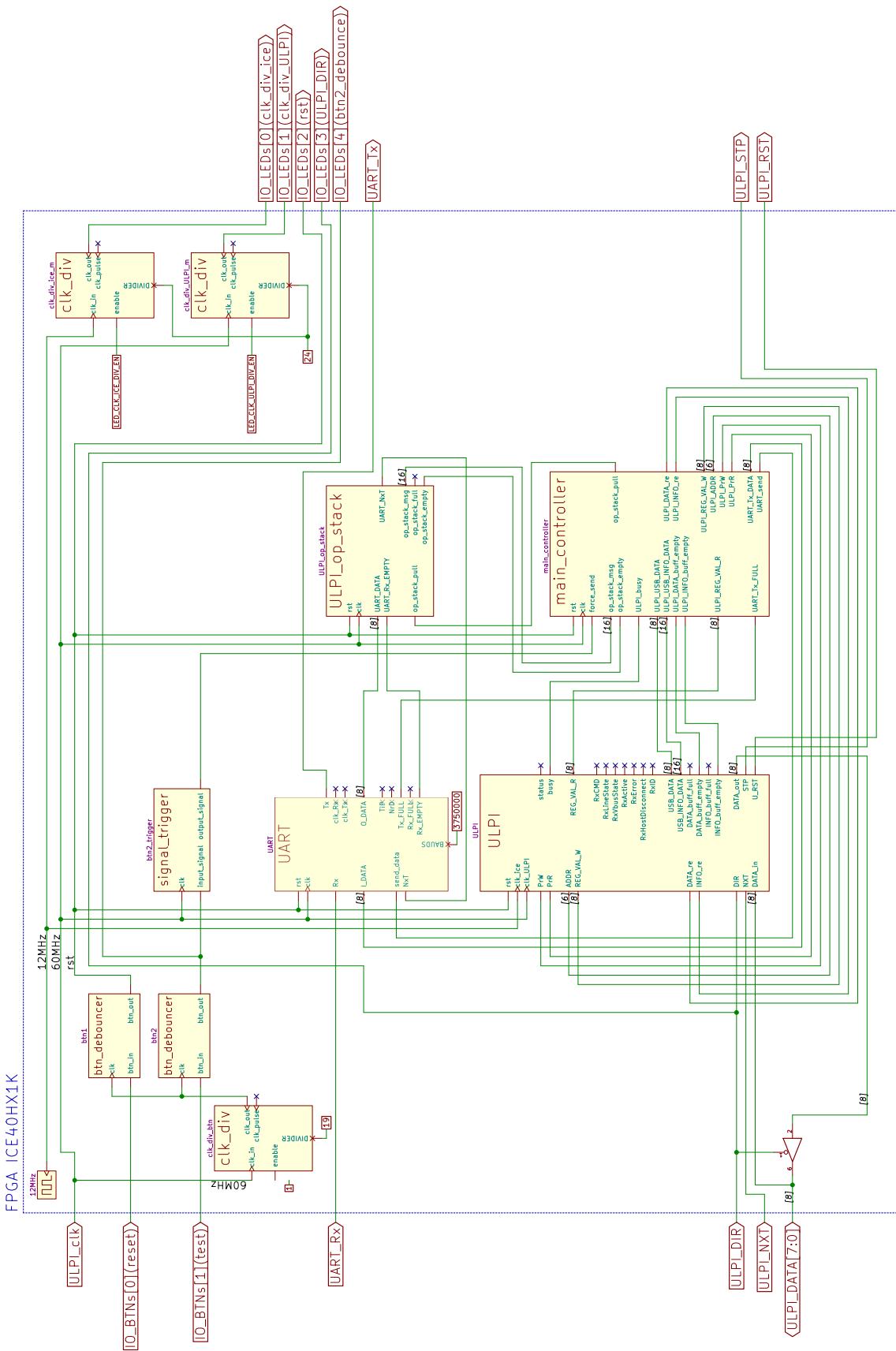
Bibliografía

- [1] *UTMI+ Low Pin Interface (ULPI) Specification*. ARC International, Conexant Systems, Mentor Graphics Corporation, Philips, SMSC, TransDimension, oct 2004. Rev. 1.1.
- [2] A. A. Barkalov and A. Barkalov. Design of mealy finite-state machines with the transformation of object codes. *International Journal of Applied Mathematics and Computer Science*, 15:151–158, 2005.
- [3] S. Chacon and B. Straub. *Pro Git v. 2.1.78*. APress, second edition, jul 2018. Disponible en línea en <https://git-scm.com/book/en/v2>.
- [4] M. C. D. Gomez-Prado. *A Tutorial on FPGA Routing*. Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, USA, sep 2007.
- [5] *Ellisys USB Exporer 200 Analyzer Datasheet*. Ellisys, Chemin du Grand-Puits 38, CH-1217 Meyrin Geneva, Switzerland, 2008.
- [6] U. Finkelstein, K. E. Fleming, and D. Baltus. *GTKWave 3.3 Wave Analyzer User's Guide*. GTKWave, may 2019.
- [7] *FTDI FT2232HL IC Data Sheet*. Future Technology Devices International Limited, Jul. 2016. Rev. 2.5.
- [8] J. González Barahona, J. Seoane Pascual, and G. Robles. *Introducción al software libre*. Barcelona: Fundació per a la Universitat Oberta de Catalunya, 2003., 2003.
- [9] GuyHarris. Libpcap file format. Technical report, The Wireshark Foundation, aug 2015.
- [10] W. Hui, B. Yoo, and K. Y. Tam. Economics of shareware: How do uncertainty and piracy affect shareware quality and brand premium? *Decision Support Systems*, 44(3):580–594, 2008.
- [11] I. Kuon, R. Tessier, J. Rose, et al. Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253, 2008.
- [12] *Lattice ICE Technology Library*. Lattice Semiconductor, mar 2015. Rev. 2.9.
- [13] *ICE40 LP/HX Family Data Sheet*. Lattice Semiconductor, Mar. 2017. Rev. 3.3.
- [14] *iCEstick Evaluation Kit manual*. Lattice Semiconductor Corp, aug 2013. Rev. 1.0.
- [15] A. Littlefield. The Beginners Guide To Scrum And Agile Project Management. Disponible online en <https://blog.trello.com/beginners-guide-scrum-and-agile-project-management>, sep 2016.

- [16] G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.
- [17] *USB3300 USB PHY IC Data Sheet*. Microchip Technology Inc., Jan. 2013. Rev. 1.1.
- [18] E. Monmasson and M. N. Cirstea. Fpga design methodology for industrial control systemsa review. *IEEE transactions on industrial electronics*, 54(4):1824–1842, 2007.
- [19] N. Nissim, R. Yahalom, and Y. Elovici. Usb-based attacks. *Computers & Security*, 70: 675 – 688, 2017. ISSN 0167-4048. doi: <https://doi.org/10.1016/j.cose.2017.08.002>. URL <http://www.sciencedirect.com/science/article/pii/S0167404817301578>.
- [20] J. G.-G. (Obijuan). Tutorial de fpga utilizando lenguaje descriptivo verilog, nov 2015. URL <https://github.com/Obijuan/open-fpga-verilog-tutorial/wiki>.
- [21] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, 1993.
- [22] K. Schwaber and J. Sutherland. The Scrum Guide™ – The Definitive Guide to Scrum: The Rules of the Game, sep 2016.
- [23] S. Sutherland. *Verilog HDL - Quick Reference Guide*. Sutherland HDL, 2001.
- [24] L. Sánchez. Directrices de organización del documento final del trabajo fin de grado tareas científico-técnicas. Disponible en línea en <https://campusvirtual.uclm.es/mod/resource/view.php?id=1082293>, apr 2013.
- [25] *PCAP reference manual*. The Tcpdump Group, jul 2018.
- [26] *Mercury T2 USB 2.0 Analyzer Datasheet*. Teledyne Lecroy, jun 2014.
- [27] *Beagle USB 12 Protocol Analyzer Datasheet*. Total Phase, 2018.
- [28] *Beagle USB 480 Protocol Analyzer Datasheet*. Total Phase, 2018.
- [29] G. B. Ulf Lamping, Luis E. Garcia Ontanon. Adding a basic dissector, dec 2014. URL https://www.wireshark.org/docs/wsdg_html_chunked/ChDissectAdd.html.
- [30] *Universal Serial Bus Rev2.0 Specification*. USB Implementers Forum, Inc., apr 2000.
- [31] *Device Class Definition for Human Interface Devices (HID) V1.11*. USB Implementers Forum, Inc, jun 2001.
- [32] C. Wolf. Yosys open synthesis suite, 2016.
- [33] Y. yuan Fang and X. jun Chen. Design and simulation of uart serial communication module based on vhdl. *2011 3rd International Workshop on Intelligent Systems and Applications*, pages 1–4, 2011.

Apéndice A

Diagramas internos de la *FPGA*

Figura A.1: Esquema principal usado en la *FPGA*.

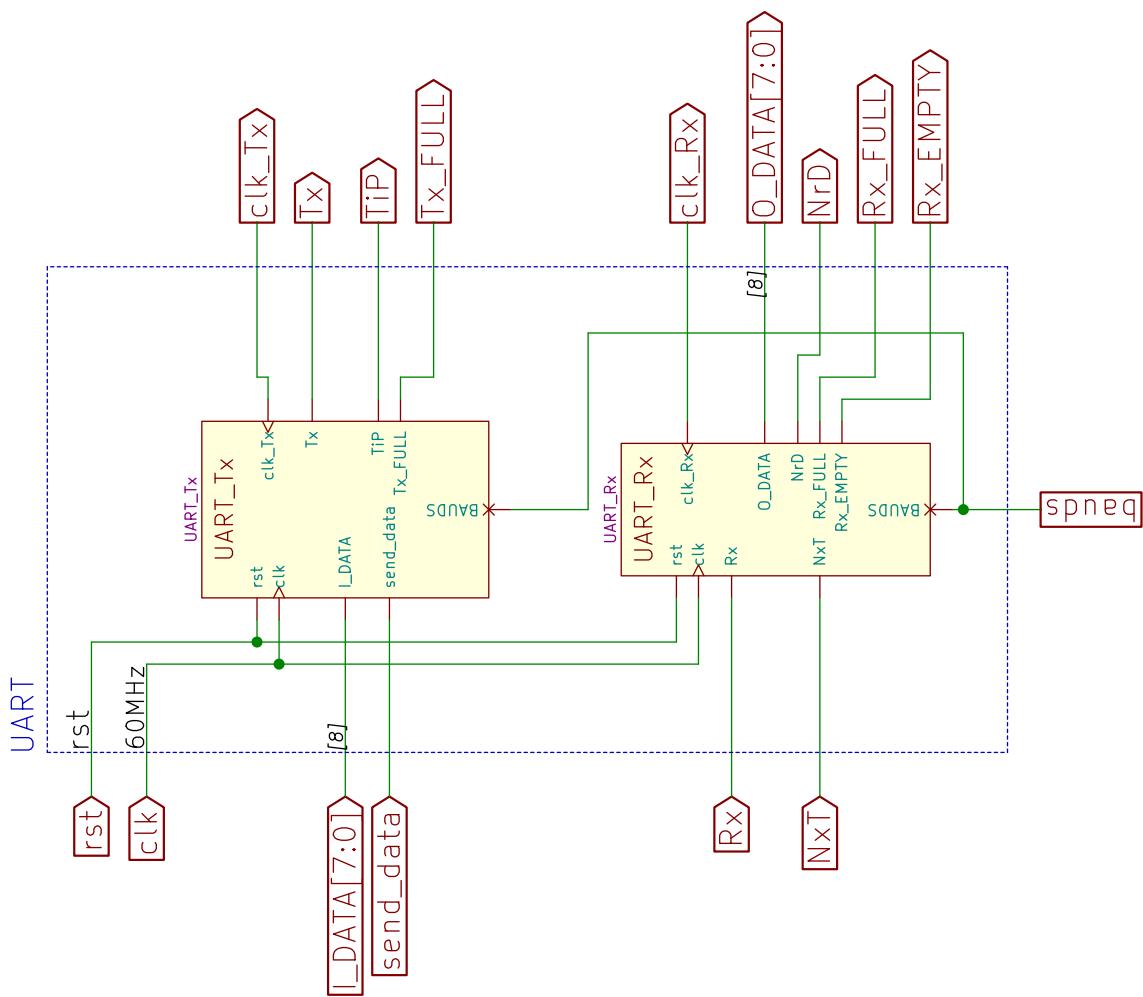


Figura A.2: Esquema principal del módulo de comunicación serie.

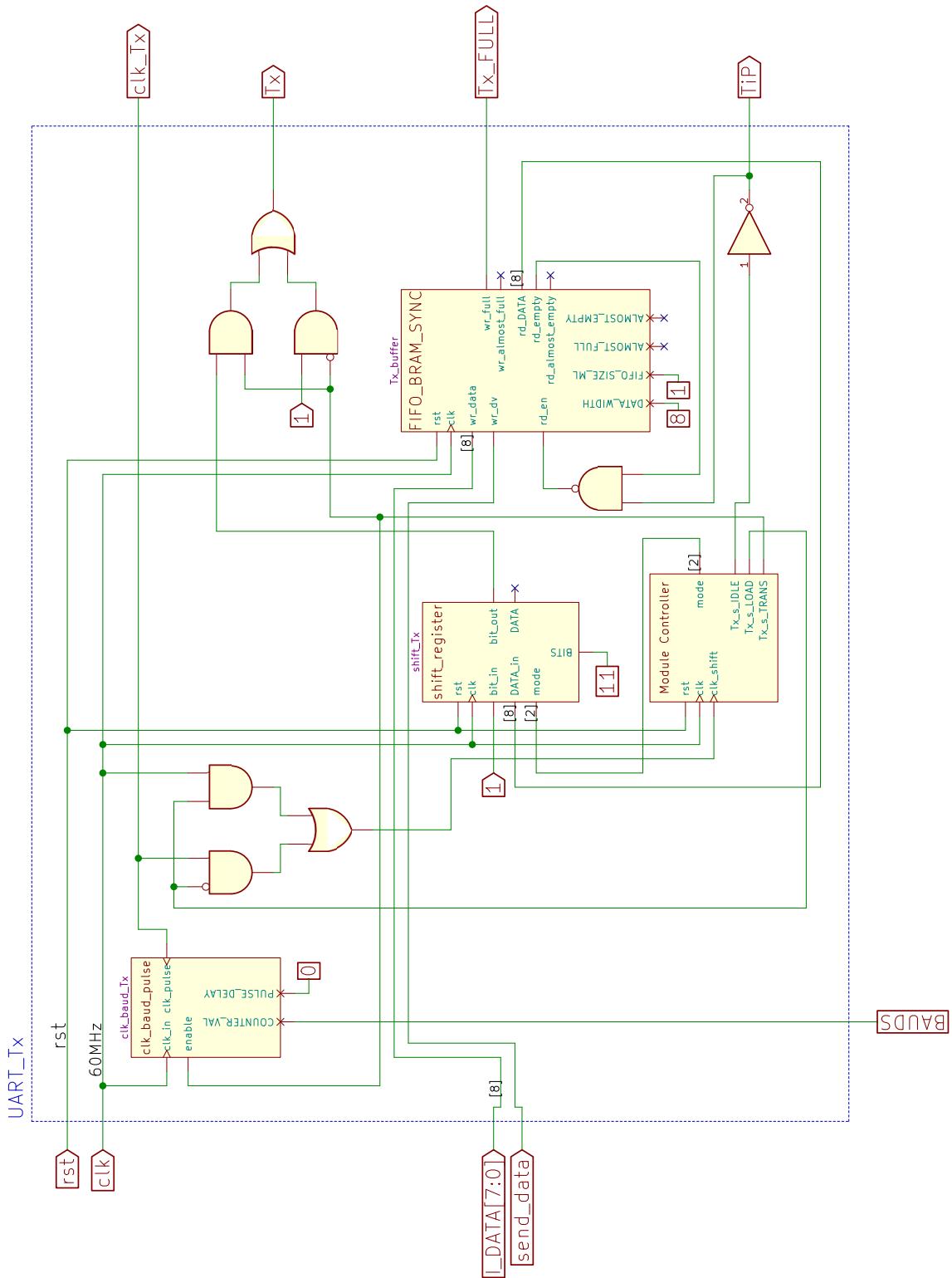


Figura A.3: Esquema del submódulo de transmisión serie.

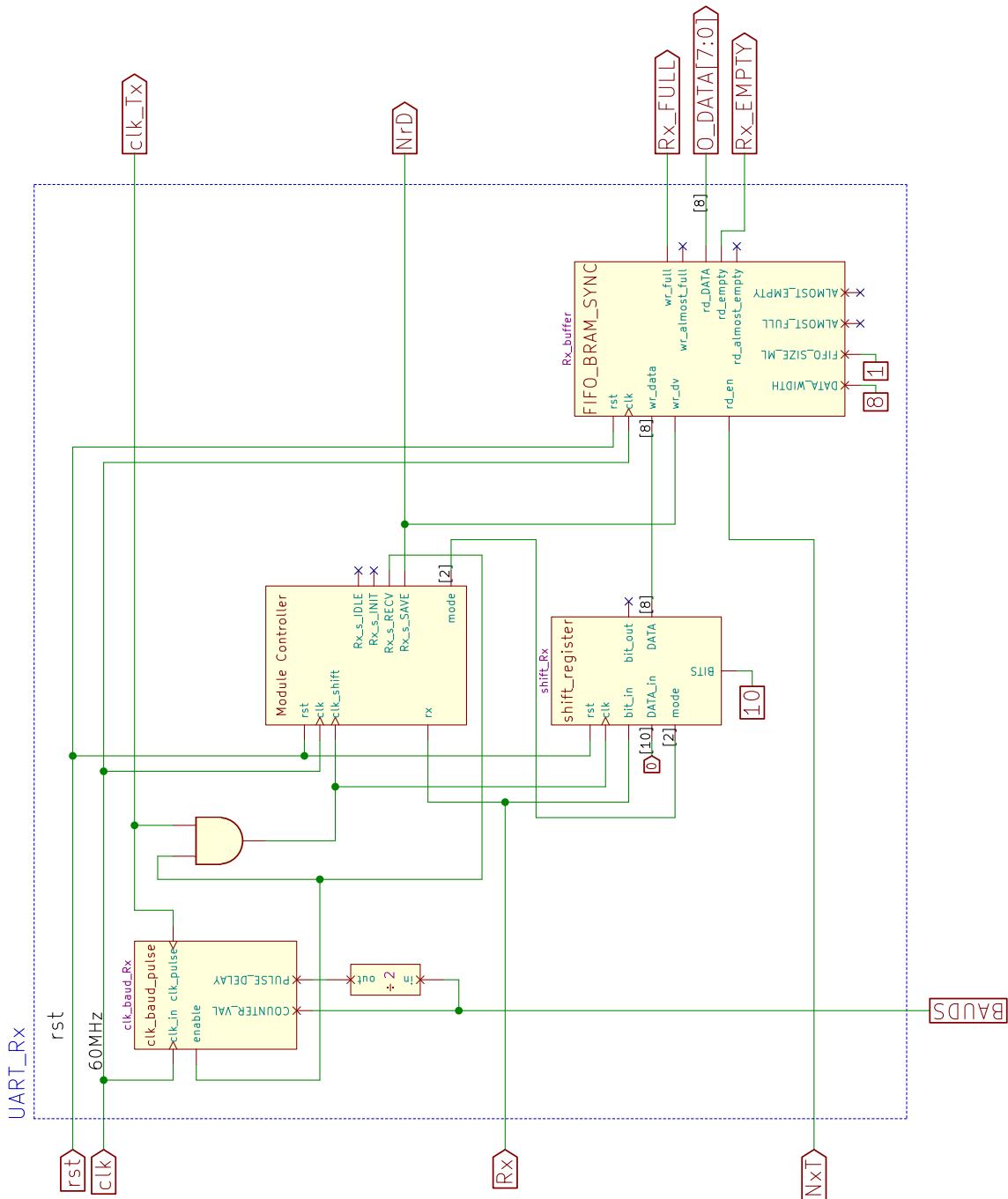
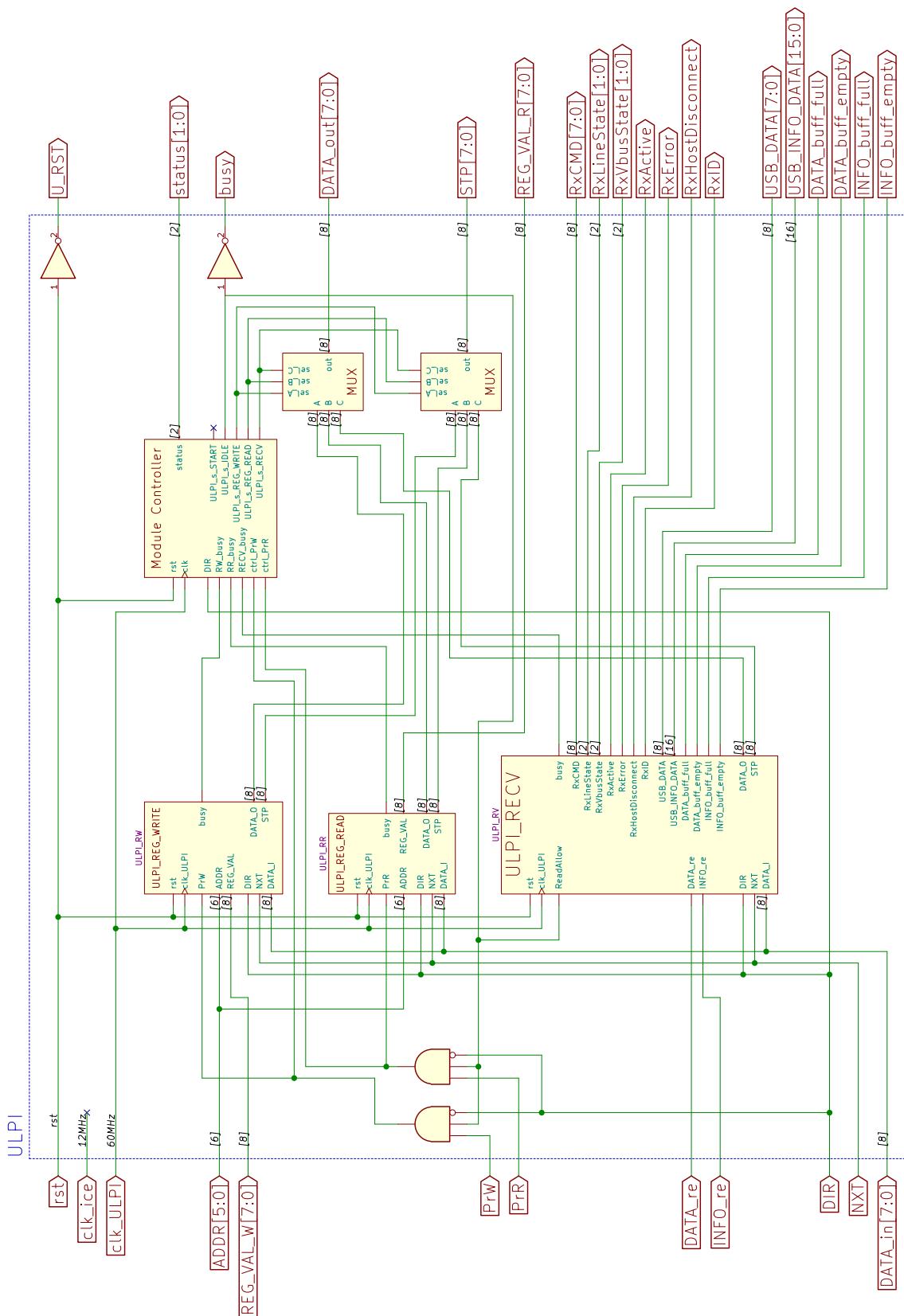


Figura A.4: Esquema del submódulo de recepción serie.

Figura A.5: Esquema principal del módulo de comunicación *ULPI*.

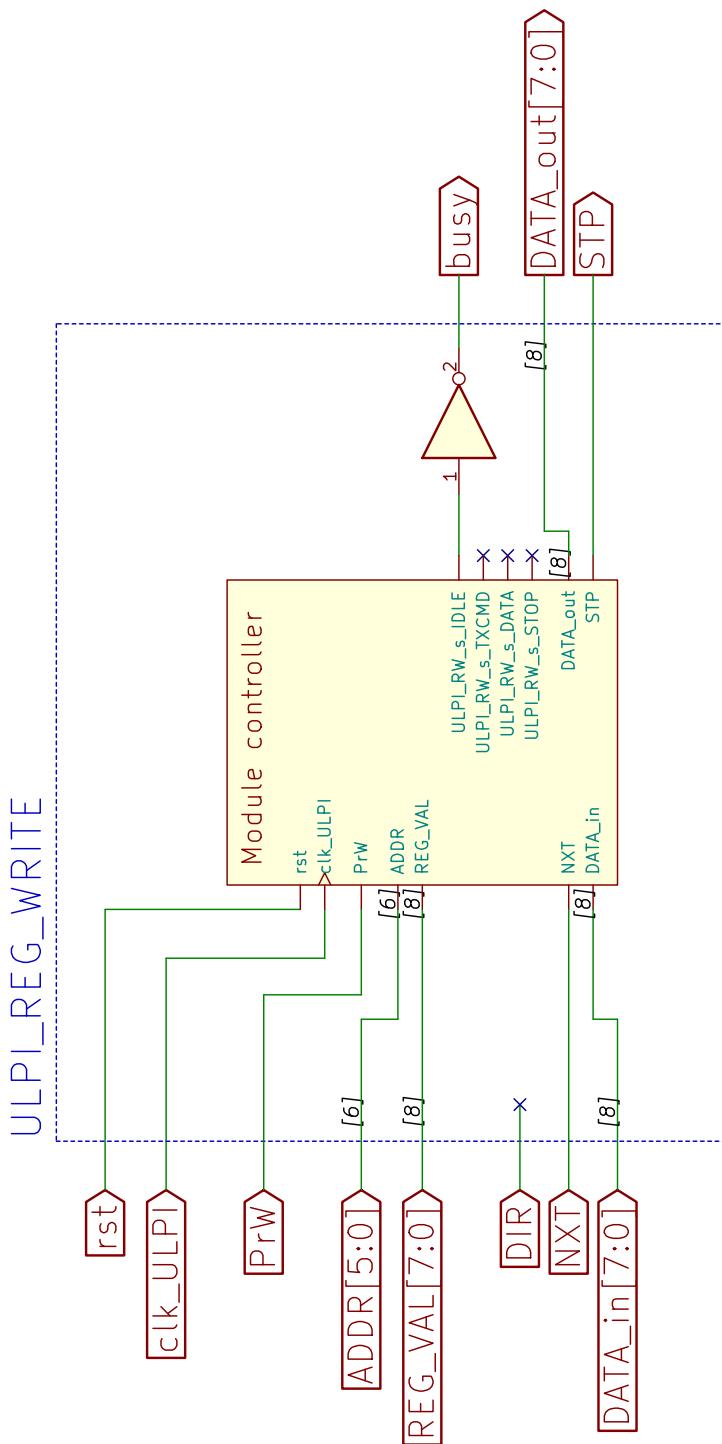


Figura A.6: Esquema del submódulo de escritura de registros *ULPI*.

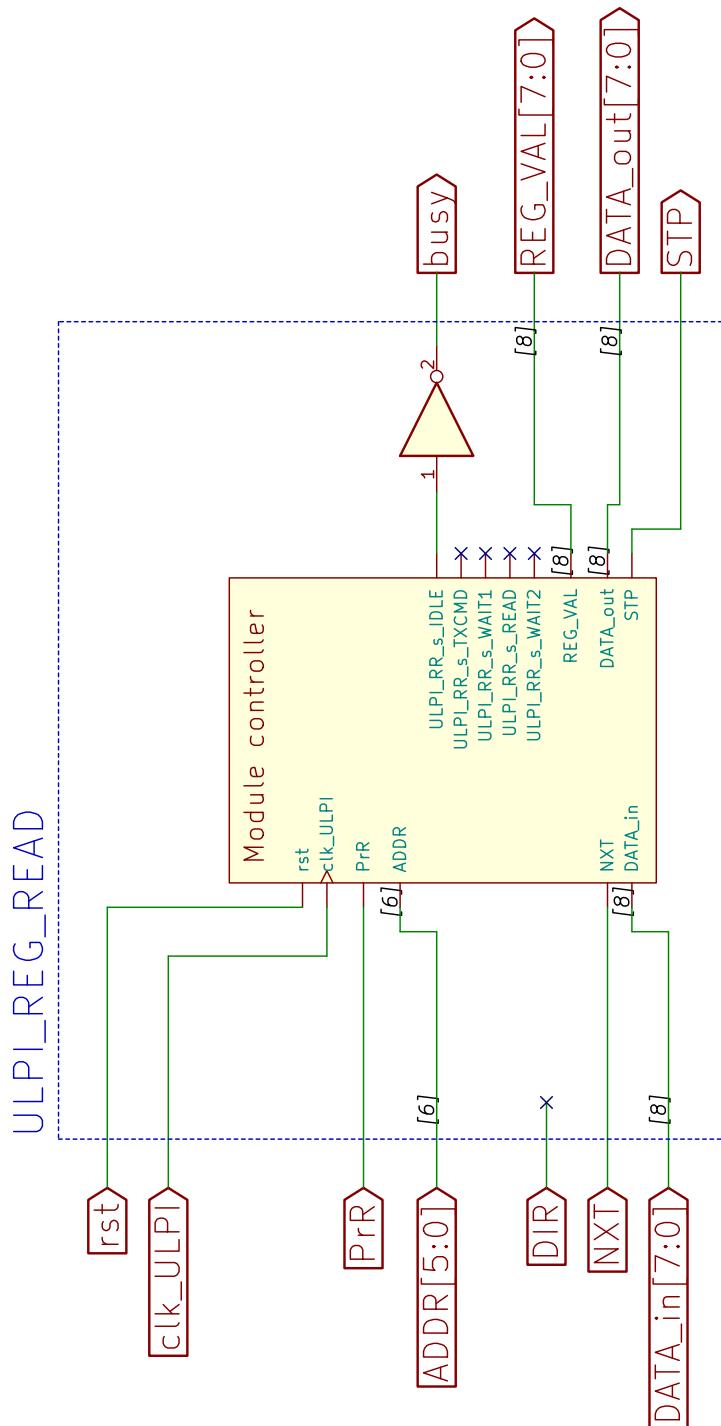


Figura A.7: Esquema del submódulo de lectura de registros *ULPI*.

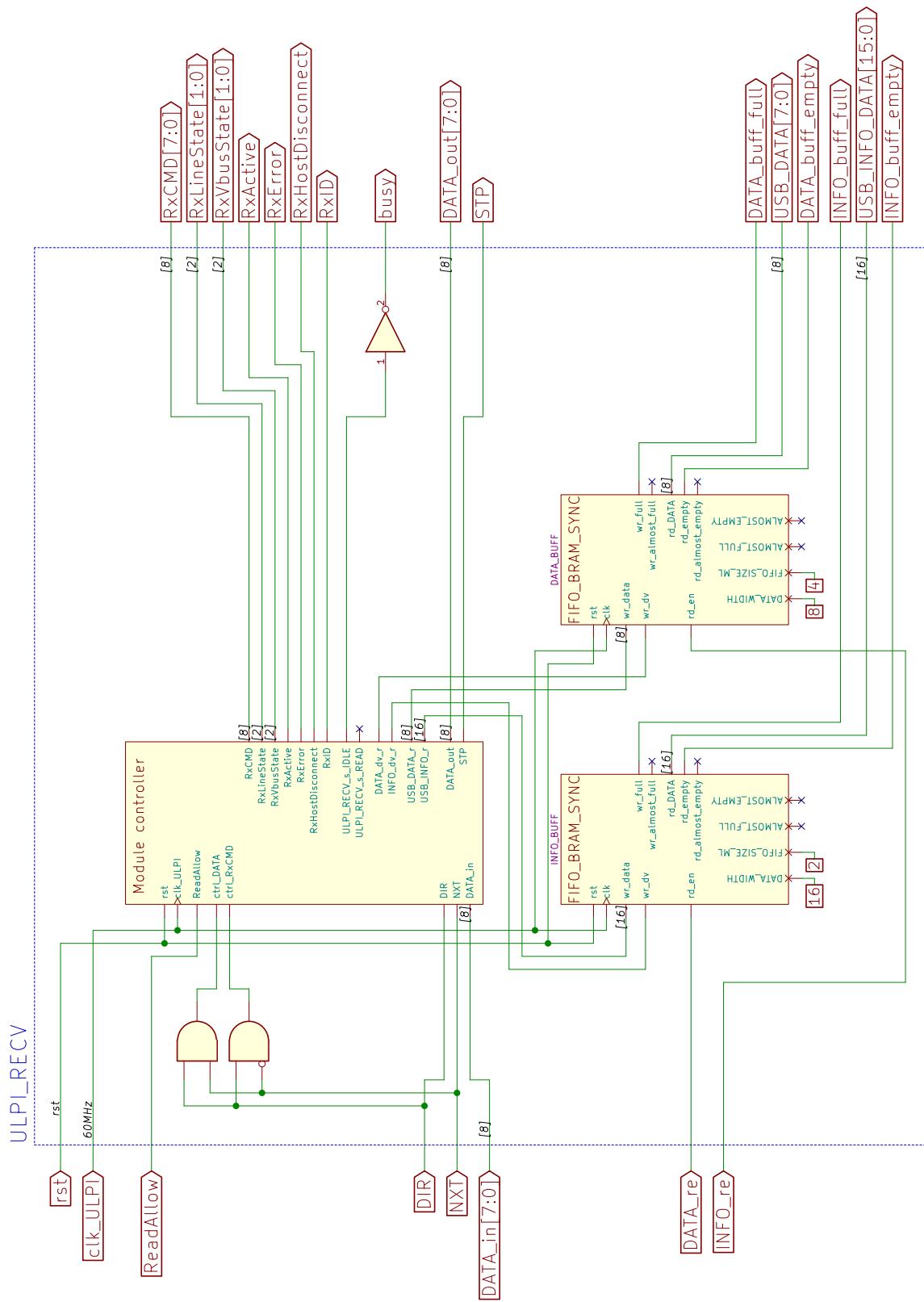


Figura A.8: Esquema del submódulo de recpción de datos USB a través de ULPi.

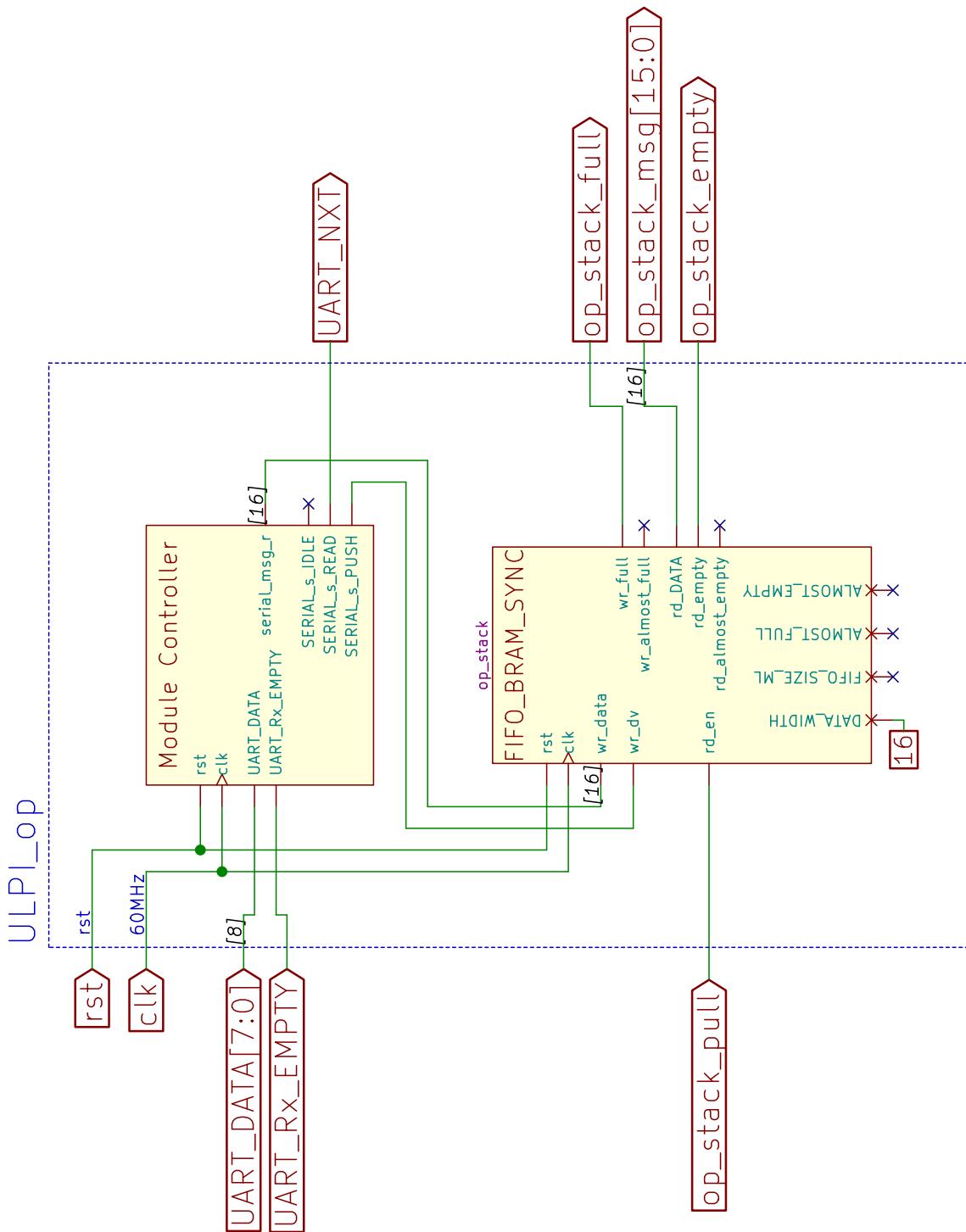


Figura A.9: Esquema del módulo de almacenaje de las operaciones pendientes.

Apéndice B

Máquinas de estados *Mealy* de los módulos diseñados

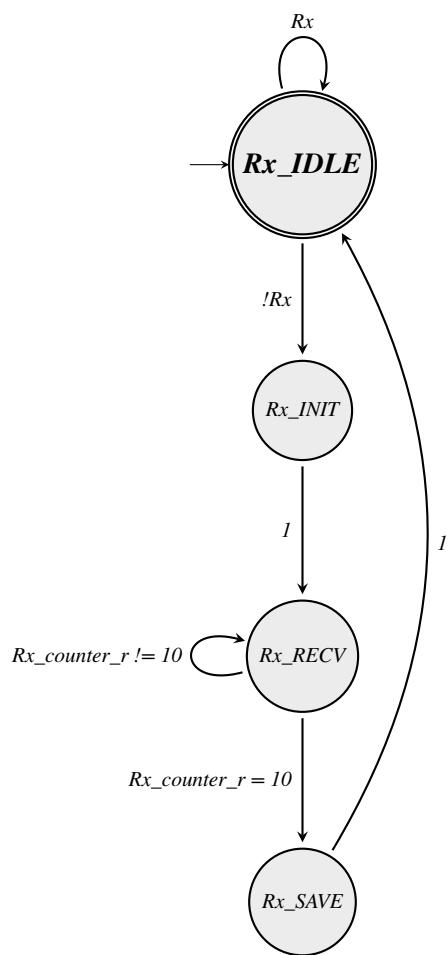


Figura B.1: Máquina de estados del módulo UART_RX

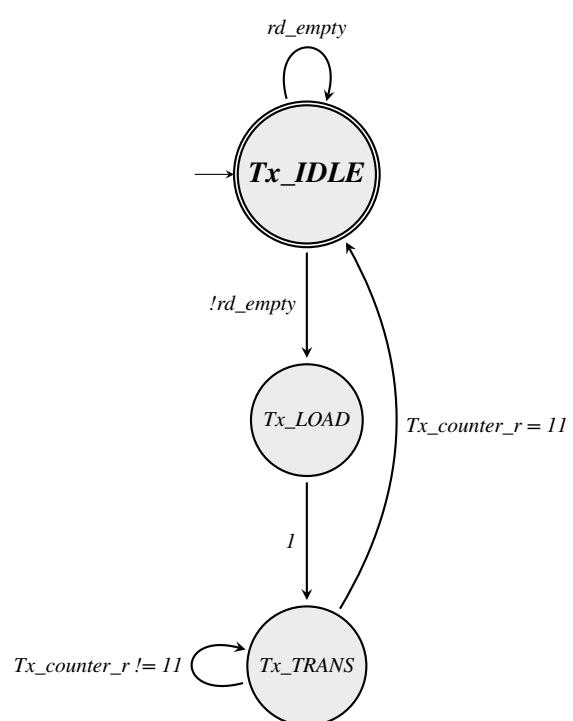
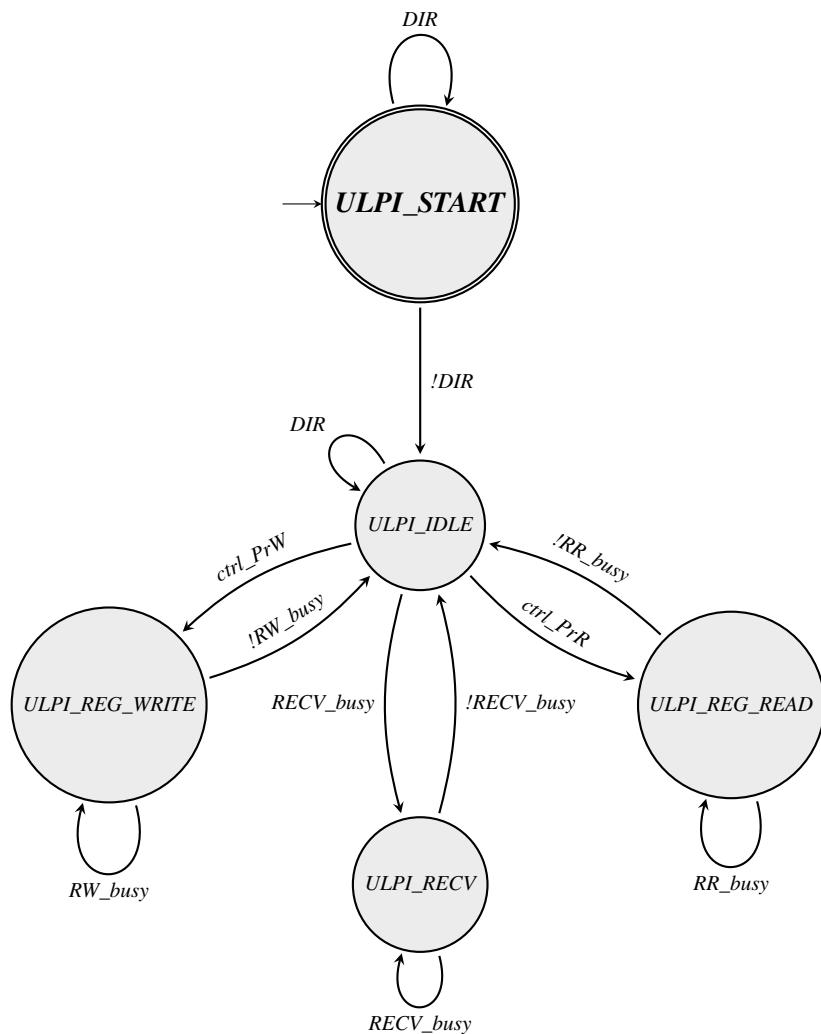


Figura B.2: Máquina de estados del módulo UART_Tx

**Figura B.3:** Máquina de estados del módulo ULPI

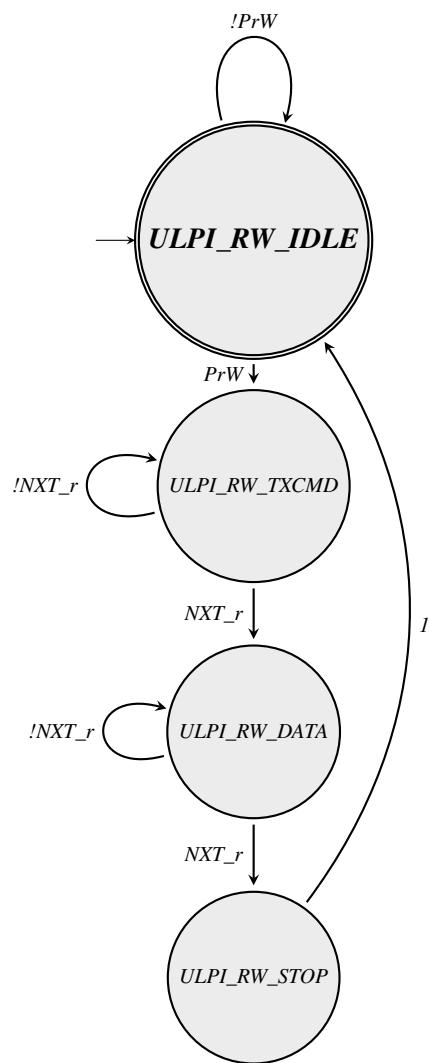


Figura B.4: Máquina de estados del módulo ULPI_REG_WRITE

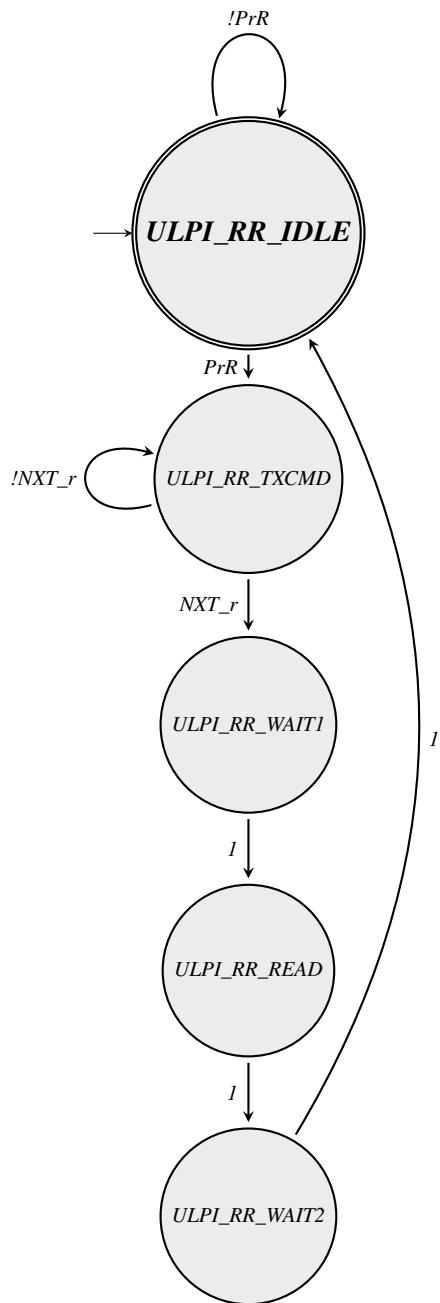


Figura B.5: Máquina de estados del módulo `ULPI_REG_READ`

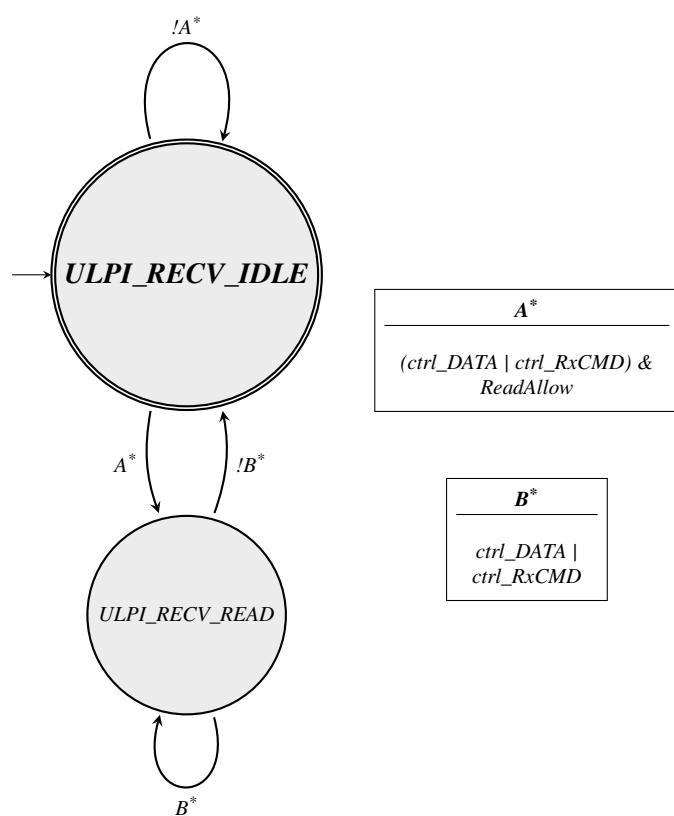


Figura B.6: Máquina de estados del módulo ULPI_RECV

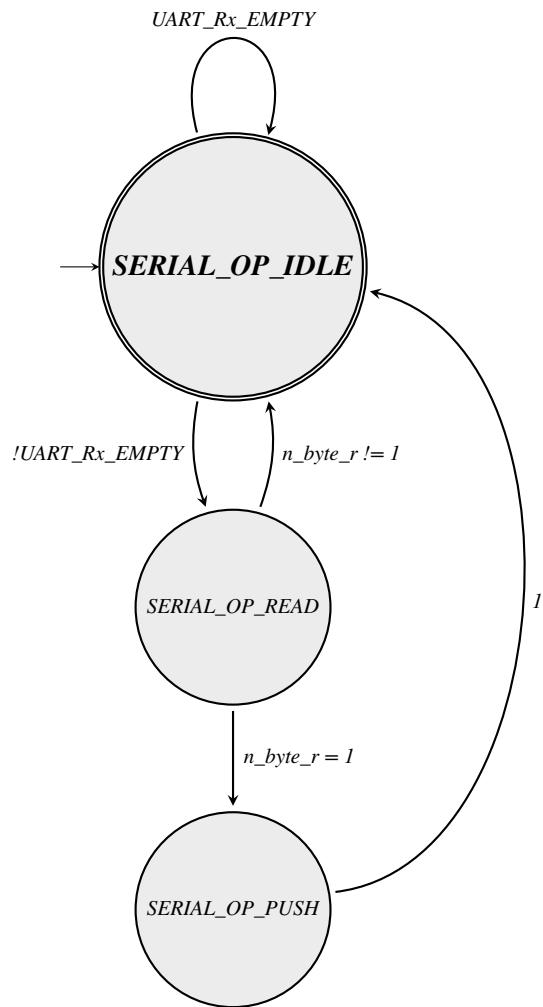


Figura B.7: Máquina de estados del módulo `ULPI_op`

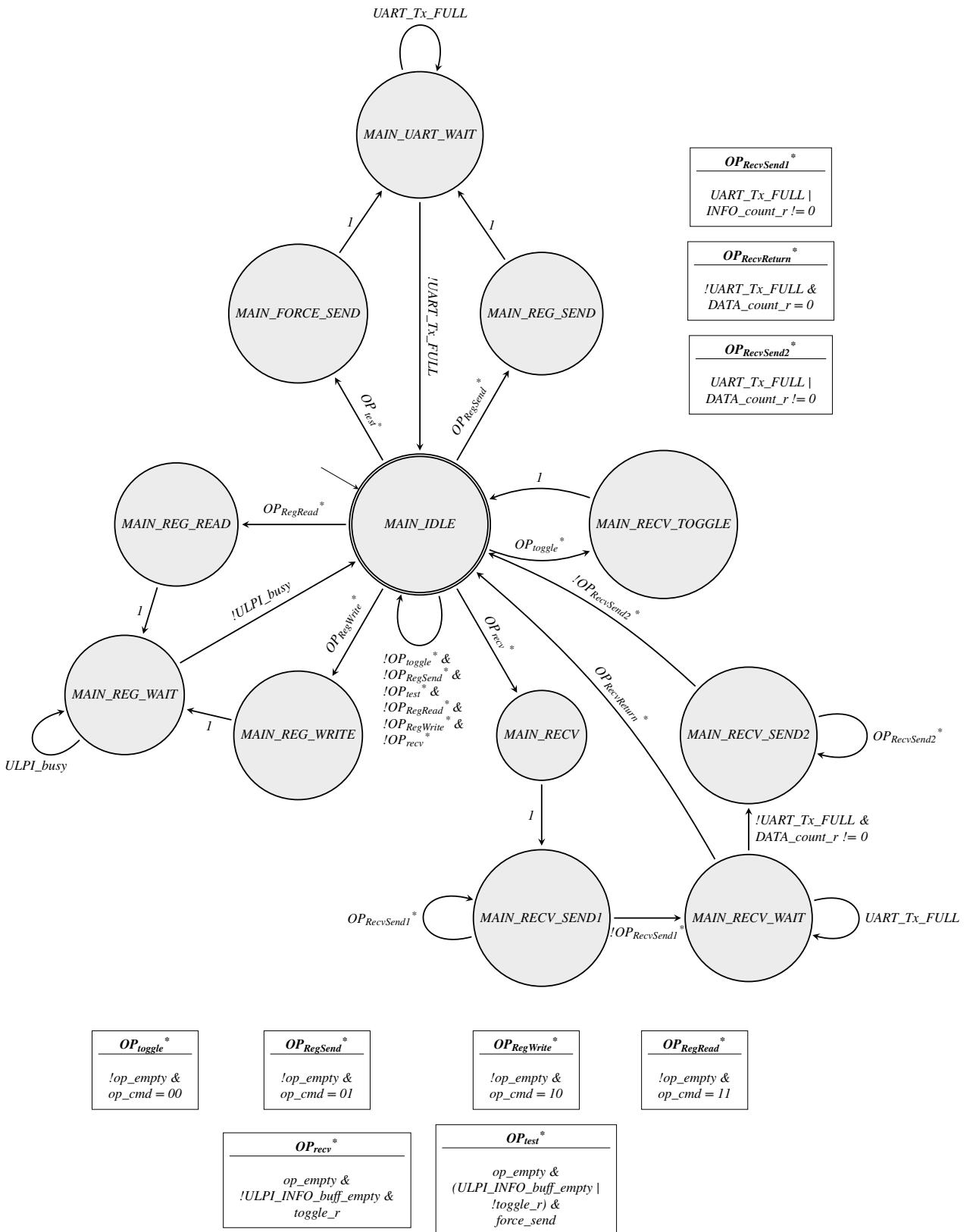


Figura B.8: Máquina de estados del módulo main_controller

Apéndice C

Resultado de la simulación final

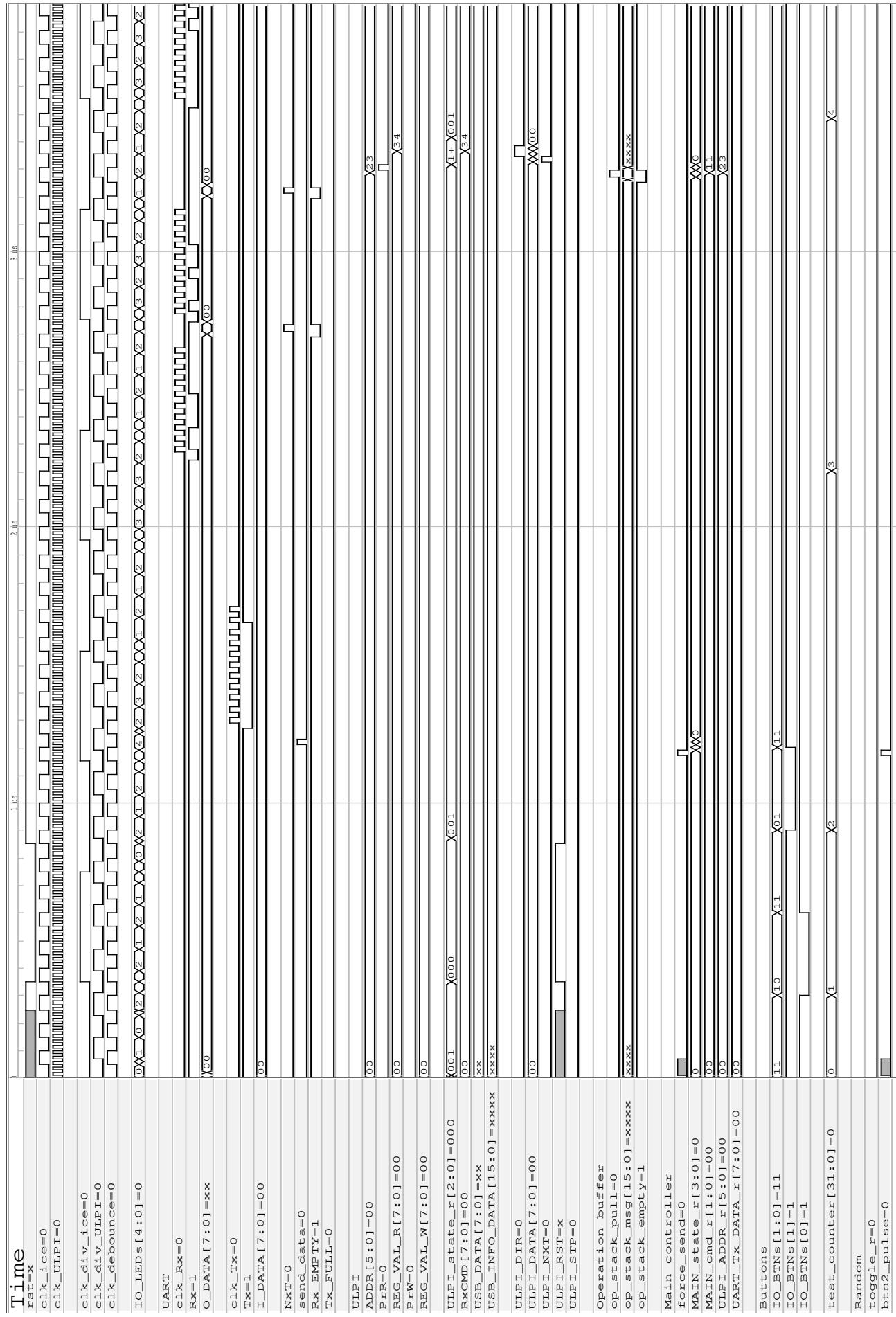


Figura C.1: Simulación final de botones externos (1 y 2) y lectura de registro (3)

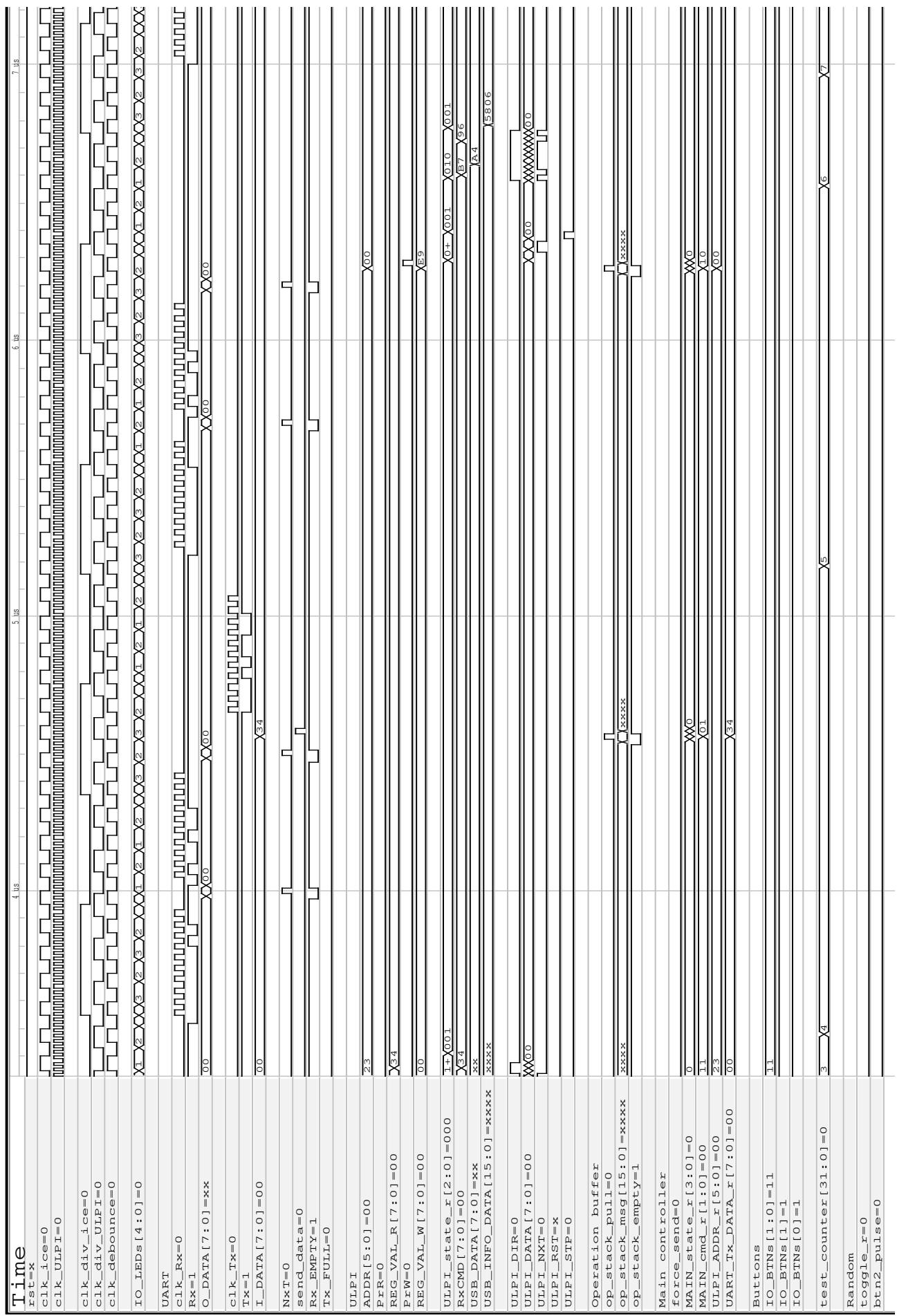


Figura C.2: Simulación final de transmisión de registro (4), escritura de registro (5) y captación USB (6)

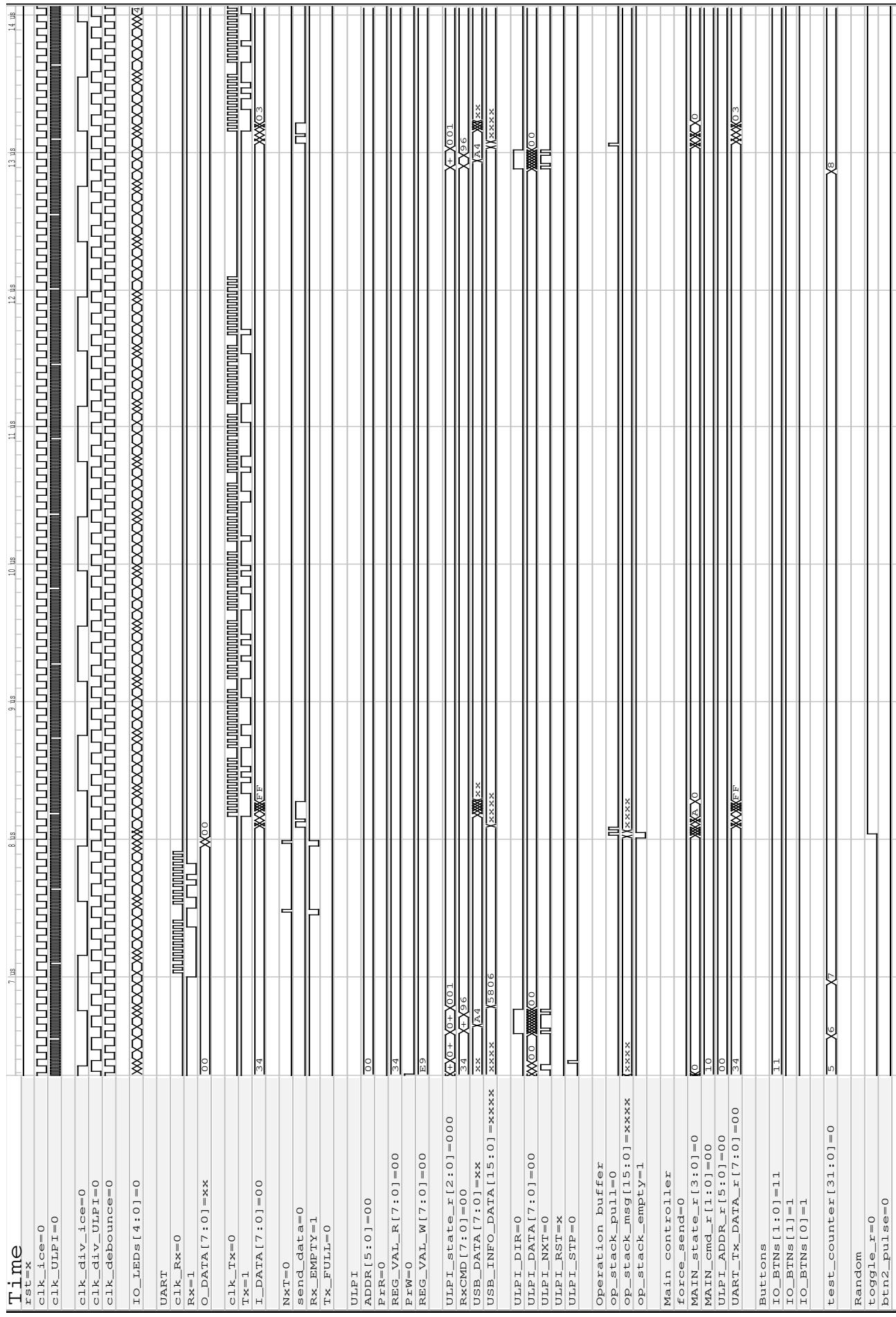


Figura C.3: Simulación final de envío de captura al PC (7)

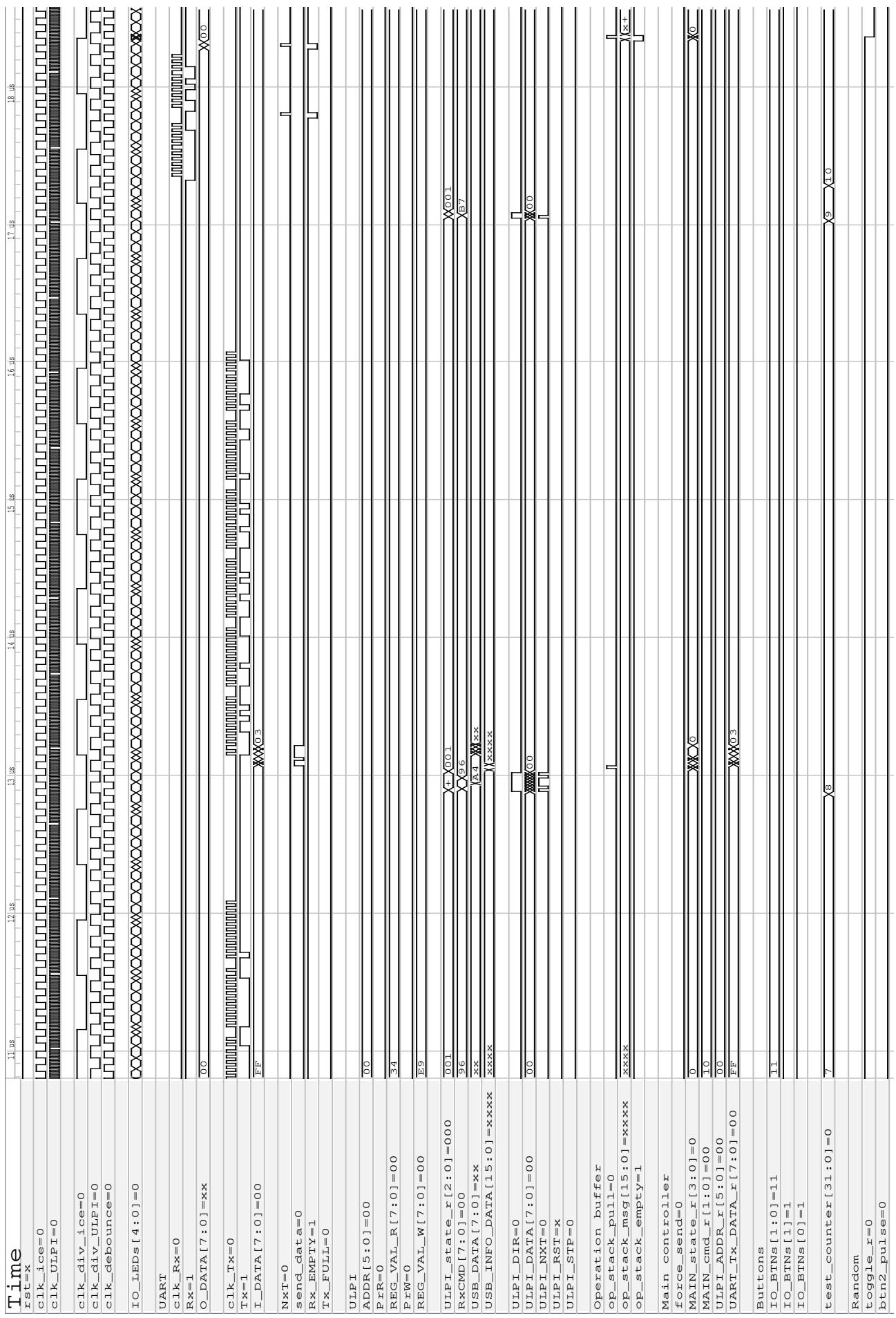


Figura C.4: Simulación final de captación USB (8), cambio de estado del bus (9) y desactivación de envío (10)

Apéndice D

Manual de instalación y utilización

En el presente anexo, se detallan los procedimientos a seguir para la instalación y utilización del sistema bajo un sistema Linux.

D.1. Requisitos *software* previos

Antes de poder generar un archivo binario compatible con la FPGA *Lattice iCE40HX1K*, es necesario instalar las aplicaciones encargadas de su creación.

Dependencias de las aplicaciones

Previo a la instalación de dichas aplicaciones, hay que instalar también sus dependencias. A continuación se dan la lista de paquetes a instalar según tres distribuciones Linux distintas.

- **Para una instalación en *Ubuntu***

```
$ sudo apt-get install build-essential clang bison flex libreadline-dev \
    gawk tcl-dev libffi-dev git mercurial graphviz \
    xdot pkg-config python python3 libftdi-dev \
    qt5-default python3-dev libboost-all-dev cmake}
```

- **Para una instalación en *Fedor*a**

```
$ sudo dnf install make automake gcc gcc-c++ kernel-devel clang bison \
    flex readline-devel gawk tcl-devel libffi-devel git mercurial \
    graphviz python-xdot pkgconfig python python3 libftdi-devel \
    qt5-devel python3-devel boost-devel boost-python3-devel
```

- **Para una instalación en *ArchLinux***

```
$ sudo pacman -S base-devel clang boost-libs python qt5-base boost \
    cmake eigen git trellis libffi tcl xdot mercurial \
    graphviz libftdi-compat
```

Instalación de las herramientas *IceStorm*

Se procede a instalar las herramientas y bases de datos del proyecto *IceStorm* (*icepack*, *icebox*, *iceprog*, *icetime* y bases de datos de las diversas FPGAs).

```

1 $ git clone https://github.com/cliffordwolf/icesstorm.git icesstorm
2 $ cd icesstorm
3 $ make -j$(nproc)
4 $ sudo make install

```

Instalación de *NextPNR*

Se procede a instalar la herramienta de posicionado y encaminamiento (*Place-And-Route*) *NextPNR*.

```

1 $ git clone https://github.com/YosysHQ/nextpnr nextpnr
2 $ cd nextpnr
3 $ cmake -DARCH=ice40 -DCMAKE_INSTALL_PREFIX=/usr/local .
4 $ make -j$(nproc)
5 $ sudo make install

```

Instalación de *Yosys*

Se procede a instalar la herramienta de síntesis del lenguaje Verilog *Yosys*.

```

1 $ git clone https://github.com/cliffordwolf/yosys.git yosys
2 $ cd yosys
3 $ make -j$(nproc)
4 $ sudo make install

```

Permisos de subida

Si se desea subir el archivo binario generado sin utilizar permisos administrativos del sistema, se ha de ejecutar el siguiente comando, en el que se da un mayor nivel de acceso al integrado *FTID* de la placa *iCEstick*.

```

1 $ sudo sh -c 'echo "ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010", MODE="0660", GROUP="plugdev",
TAG+="uaccess"" > /etc/udev/rules.d/53-lattice-ftdi.rules'

```

D.2. Descarga del repositorio

Para poder utilizar el sistema, hay que descargar en primer lugar los datos del repositorio.

```

1 $ git clone https://github.com/mario-mra/tfg.git analizador_usb
2 $ cd analizador_usb

```

A partir de este momento, todas las operaciones a realizar estarán referencias al directorio del repositorio: `.\anализатор_usb`.

D.3. Generación y programación del archivo binario

Los archivos a utilizar para la generación del binario están situados en la carpeta `./ICEstick/USB3300_sniffer`. Una vez situados en esa carpeta, y si todos los requisitos están instalados, ya sería posible realizar la generación y posterior programación del archivo binario.

```

1 $ cd ./ICEstick/USB3300_sniffer
2 $ make pnr
3 $ make prog
4 $ cd ../../

```

D.4. Instalación de las aplicaciones de control

Las aplicaciones creadas están situadas en la carpeta `./PC/`. Previamente a su ejecución hay que compilarlas e instalarlas.

Instalación de la aplicación de control

```

1 $ cd ./PC/app
2 $ make
3 $ sudo make install
4 $ cd ../../

```

Instalación de la aplicación de conversión *JSON* a *PCAP*

```

1 $ cd ./PC/json2pcap
2 $ make
3 $ sudo make install
4 $ cd ../../

```

D.5. Obtención de la captura

Con la aplicación de control ya instalada, y el sistema conectado al PC, ya se podría comenzar la captura. Para evitar posibles perdidas de datos capturados, se recomienda conectar el bus USB a analizar con el sistema en funcionamiento y la señal *RECV_TOGGLE* activa.

Nota. La aplicación se encarga de escribir los registros ULPI necesarios para poder realizar la captura, por lo que las opciones **2** y **3** no son necesarias utilizarlas.

1. Ejecutar la aplicación. `$ FPGA-usb-capture`.

2. Configurar el puerto si fuera necesario, eligiendo el dispositivo a usar. **[Opción 0]**
3. Abrir el puerto serie con la configuración dada. **[Opción 1]**
4. Empezar la recepción de datos, activando la señal *RECV_TOGGLE*. **[Opción 4]**
5. Conectar el dispositivo a analizar a la placa *USB3300*.
6. Realizar todas las operaciones deseadas.
7. Cerrar el programa (automáticamente se apaga la señal *RECV_TOGGLE* y se cierra el puerto). **[Opción 5]**

La captura generada se guarda automáticamente en el directorio desde el que se llama a la aplicación. Tras la captura, es recomendable almacenar dicho archivo, ya que si se vuelve a realizar una nueva, este archivo será sobrescrito.

D.6. Conversión de la captura a formato *PCAP*

Para realizar la conversión a formato *PCAP*, simplemente hay que llamar a la herramienta `$ capture-json2pcap <ruta archivo JSON>`, indicando como parámetro la ruta del archivo *JSON*. El archivo convertido, se guarda en el directorio donde se encuentra la captura original, manteniendo el mismo nombre.