

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR



Grado en Ingeniería de Tecnologías y Servicios de
Telecomunicación

Trabajo Fin de Grado

**Desarrollo de una placa suplementaria para
Digilent Zybo**

Autor: Umberto Benito Vargas Sánchez

Tutor: Sergio López Buedo

Julio 2020

Resumen

Este este trabajo de fin de grado se desarrolla una placa de expansión con tres periféricos sencillos que se conecta a través de los puertos *pmod* de la placa de desarrollo FPGA, Zybo Z7, para cubrir la carencia de dichos componentes en esta última. Para ello, se hace un estudio previo con el objetivo de identificar que componentes serían los adecuados, por lo que primero se estableció un proyecto de procesado de audio digital el cual se recibe desde un dispositivo móvil vía *streaming bluetooth*, que además cuenta con la posibilidad de *playback* del resultado a través del *jack* de audio de la propia Zybo. Deduciendo así que los componentes que le harían falta a la Zybo para ejecutar dicha aplicación son: un módulo bluetooth, unos displays y un puerto serie. Posteriormente, se ha llevado a cabo un estudio de mercado sobre los componentes específicos que cumplan las demandas al menor coste y estos son comparados con sus homólogos que ofrece el propio fabricante Digilent como módulos *pmod*.

Ya adquiridos dichos componente, se desarrollo el PCB de la placa de expansión mediante la herramienta Altium Designer 19, la fabricación y soldado de los componentes de la placa en laboratorio, y se desarrollo aparte del proyecto inicialmente fijado, uno más para usuarios principiantes para ayudarlos a introducirlos en el desarrollo tecnología FPGA. Este último, es un proyecto básico puramente RTL y el principal es más avanzado, y fue desarrollado mediante las herramientas Xilinx Vivado IP Integrator y Vitis.

Este trabajo trae consigo una conclusión final que sugiere mejoras sobre el proyecto avanzado, el cual no ejecuta un sistema operativo sino que trabajamos directamente con el procesador en modo *BareMetal*, por lo que se propone la inclusión de un sistema operativo y nuevas funcionalidades como añadir la fusión de parar la reproducción en curso o monitorización de temperatura en los displays.

Palabras Clave— FPGA, PCB, Digilent, Zybo, Pmod, Xilinx Vivado IP Integrator, Altium Designer 19, RTL, Xilinx Vitis, HDL, BareMetal

Abstract

In this work, an expansion board with three simple components has been developed for being connected through the *Pmod* ports of the Zybo board in order to mitigate its lack of those elements. Bearing in mind this particular issue, a research has been done with the goal to identify which component will be suitable to work with the Zybo. So, a project was set up since the beginning and consisted in a digital audio processing app which will receive digital audio via streaming through Bluetooth and also is able to playback the processed data through the audio jack of the Zybo. Consequently, a Bluetooth module, couple of displays and an serial port were chosen to fit with this requirements. After that, a market study was performned to find the proper components. In that research , cost and performance were keypoints. Finally, an comparison between the chosen componets with their counterparts available as Digilent *Pmod* modules was done.

Next, PCB design was done using Altium Designer 19, then, manufactured, and soldered in the lab. In parallel, another project was developed, this last one was easier, focus on beginners who want to introduce themselves in FPGA design. However, the main project was only for mid-level or advance users because it uses Vivado IP integrator and Vitis.

This work ends with the final conclusions, where some improvements are suggested. The target of this suggestions is that the main project, which is a baremetal application, lacks an Operating System, so one way to solve this is using RTOS. Another idea is including a stop bottom to pause the playback, and show up the current temperature in the displays from the sensor of the bluetooth module.

Keywords— FPGA, PCB, Digilent, Zybo, Pmod, Vivado IP Integrator, Altium Designer 19, RTL, Vitis, BareMetal.

Agradecimientos

Quiero mostrar mi más profundo agradecimiento a mi tutor Sergio López Buedo, a quien admiro por su gran vocación académica y extensos conocimientos en este campo de las telecomunicaciones. Sergio me ha guiado, corregido y trabajado codo a codo conmigo en este largo camino, ayudándome a ser profesional y tener una mejor atención por el detalle, además de haber mostrado una gran paciencia con mis distintas iteraciones (en ocasiones no acertadas) a lo largo del desarrollo de este proyecto, el cual, sin su ayuda no habría sido posible, gracias. Además, no olvidar a mi madre Caridad, que siempre me animó con mis estudios y ha sido siempre mi soporte para conseguir acabar esta carrera y me dio la oportunidad de venir a España a estudiar en una gran universidad.

En clase José Ignacio, Pablo y Santiago, siempre han estado ahí apoyándome y teniéndome una mano cuando los necesitaba en las muy buenas, pero también en las menos buenas. Y sin duda, las prácticas de antenas con los dos primeros fueron una verdadera experiencia de *team-work*.

Mención especial a Emma y Andrea, unos grandes amigos de los cuales afecto y ánimo nunca faltaron en la consecución de mis metas.

Y en general a todos los componentes de las universidad, y en especial a los profesores de la EPS por su trabajo y su trato profesional.

Índice general

I	Introducción	1
1	Motivación	1
2	Objetivos	2
3	Fases de realización	2
4	Estructura de la memoria	3
II	Introducción y Estado del Arte	4
1	Sistemas Empotrados Reconfigurables	4
2	Zynq SoC	5
3	¿Por que ZYBO?	6
4	AXI Interface	8
4.1	AXI Architecture	9
4.2	AXI Streaming Interface	10
III	Diseño de la placa de expansión	11
1	Diseño de la Placa	11
1.1	Elección de componentes	11
1.2	Diseño de esquemáticos	11
1.3	Diseño del PCB	14
IV	Proyecto básico	18
1	Implementación en Verilog	18
V	Proyecto avanzado	20
1	Resumen de la proyecto	20
2	MicroBlaze	22
3	DMA	22
4	<i>I²S</i> Receiver	24
5	Digilent <i>I²S</i> Transmitter	24
6	AXI <i>I²C</i> IP	24
7	AXI UARLITE IP	25
8	AXI GPIO IP	25
9	AXI STREAM FIFO IP	26
10	FIR	27
11	High Level Synthesis	29

12	FIR AXIS	30
13	Diseño del software	32
14	Comunicación con el módulo	34
15	Procesamiento y almacenamiento de audio	36
16	Reproducción de audio	37
17	Validación	38
VI	Conclusión.	40
1	Resumen	40
2	Trabajo Futuro	40
Anexos		42
1	Recursos PS	43
2	Recursos PL	44
3	Microblaze	45
4	I^2S Receiver	46
5	DMA	47
6	I^2S Transmitter	49
7	UARTLITE	51
8	I^2C	51
9	HLS Filtro FIR optimizado	52
10	Testbench proyecto básico	52
11	Fir Axi Stream extra	54
12	Microchip Bluetooth Module (BM64)	55
13	BOM	57

Índice de figuras

Figura 2:	Diagrama de Gantt.	3
Figura 3:	Sistema embebido	4
Figura 4:	Aplicaciones de los Sistemas Embebidos.	5
Figura 5:	Zynq	6
Figura 6:	ZYBO CENITAL	6
Figura 7:	Rules	7
Figura 8:	Zynqberry	7
Figura 9:	MYD-C7Z010 + MYD-C7Z010	8
Figura 10:	AXI Write	9
Figura 11:	AXI Read	10
Figura 12:	AXI Stream	10
Figura 13:	Esquemático Módulo Bluetooth	12
Figura 14:	Esquemático puerto serie	13
Figura 15:	Características del drenador. $I_D = F(V_{DS})$.	13
Figura 16:	Esquemático de los displays	14
Figura 17:	Estructura de nuestro PCB.	14
Figura 18:	Rules	15
Figura 19:	Polygon	15
Figura 20:	PCB final.	16
Figura 21:	Placa construida	17
Figura 22:	Proyecto básico 1	18
Figura 23:	Proyecto básico ondas	19
Figura 24:	Proyecto Avanzado	20
Figura 25:	Proyecto Avanzado IP Integrator	21
Figura 26:	HP	23
Figura 27:	Interacción MicroBlaze-DMA.	23
Figura 28:	I^2S Receiver	24
Figura 29:	I^2C	25
Figura 30:	UART	25
Figura 31:	XGPIO	26
Figura 32:	Configuración AXI Stream FIFO.	27
Figura 33:	Ecuación del Filtro FIR.	27
Figura 34:	diseño FIR en simulink	28
Figura 35:	DSP Simulink	28
Figura 36:	Espectro de la señal de audio	29

Figura 37:	Flujo de diseño del Vivado hls HLS [4].	30
Figura 38:	ILA block	34
Figura 39:	ILA WAVE TX	35
Figura 40:	ILA WAVE RX	35
Figura 41:	Sincronización FIFO-FIR-I2S	36
Figura 42:	Archivo de audio	38
Figura 43:	Pairing	39
Figura 44:	Processing System Resources	43
Figura 45:	Processing Logic Resources	44
Figura 46:	Microblaze	45
Figura 47:	Interrupciones.	46
Figura 48:	crossing	46
Figura 49:	Configuración I2S RX	47
Figura 50:	Direcciones predefinidas PS	47
Figura 51:	Configuración del puerto acrfullHP	48
Figura 52:	Modificación del la asignación de direcciones en Vivado	48
Figura 53:	AXI DMA	49
Figura 54:	FIFO DIGILENT	50
Figura 55:	IP Digilent	50
Figura 56:	Configuración UART	51
Figura 57:	Configuración I^2C del SSM2603.	51
Figura 58:	Esquemático de nuestro switches.	56
Figura 59:	BOM	57

Capítulo I

Introducción

1. Motivación

La curva de aprendizaje para un ingeniero que se quiere introducir en el mundo del desarrollo de sistemas empotrados reconfigurables, basados en tecnología FPGA, puede ser muy compleja. Es mucho más fácil abordar esta complejidad si trabajamos sobre placas de desarrollo que tengan una gran variedad de periféricos sobre los que practicar y aprender. Una de las placas más asequibles en cuanto a coste, la Diligent Zybo Z7, que aúna un potente dispositivo SoC ARM/FPGA con sofisticados periféricos multimedia (HDMI, Ethernet, etc.), apenas cuenta con dispositivos más sencillos (displays LED, puertos serie, etc.) con los que empezar a trabajar. Este es un problema que aparece también en otras placas, y aunque el uso de módulos Pmod pueda ser la solución, no es una opción óptima, porque son módulos pequeños que pueden perderse o desconectarse fácilmente. Por esta razón, nuestra meta durante la elaboración de este trabajo fue complementar el uso académico de la Zybo para usuarios desde principiantes a moderadamente avanzados, proponiendo una solución compacta e integrada, a diferencia de lo que ofrecen los módulos Pmod.

Primero, nuestra placa cuenta con componentes básicos para el desarrollo de prácticas en Verilog/VHDL, que resultara útil para por ejemplo estudiantes de ingeniería que quieren aprender conocimientos de diseño RTL.

Segundo, la combinación Zybo-placa permite llevar a cabo proyectos mas complejos e interesantes para usuarios moderadamente avanzados, haciendo uso de un microprocesador empotrado como MicroBlaze. Además, usaremos herramientas más sofisticadas como IP Integrator de Xilinx Vivado, para desarrollar el HW, y Xilinx Vitis, para el SW. En esta parte no se trata de reinventar la rueda escribiendo el HW desde cero en Verilog/VHDL, sino de diseccionar nuestro proyecto en bloques funcionales mas manejables y abordar su interacción con el SW, acelerando así el proceso de desarrollo de nuestro diseño.

En conclusión, el objetivo final es animar a quien esté interesado en el desarrollo de HW a dar unos primeros pasos en el desarrollo FPGA, demostrando que es asequible y que cuenta con muchas herramientas de desarrollo gratuitas proporcionadas por los distintos fabricantes del mercado como Xilinx o Intel. Y si es alguien con más conocimiento previo de SW que de HW, que sea consciente de que hoy en día no es una barrera, pues existen herramientas como Vivado-HLS o Intel FPGA SDK for OpenCL, en las que podemos implementar nuestros algoritmos en C , C++, OpenCL o incluso Python. Y estas herramientas se encargan de traducirlo a nivel RTL. Nunca es tarde para aprender diseño FPGA, y esta es una tecnología en plena efervescencia con nuevas iniciativas como las instancias F1 de Amazon EC2 que ofrecen aceleradoras reconfigurables basadas en la nube.

2. Objetivos

El objetivo de este TFG es desarrollar una placa con 4 periféricos, dos displays, un puerto serie y un módulo bluetooth. Y se conectara dicha placa a los puertos *Pmod* de la placa Zybo.

Desarrollaremos así dos proyectos, el primero, una práctica sencilla en Verilog para manejar el display. Y un segundo, desarrollamos un dispositivo capaz de recibir audio en streaming vía bluetooth, procesarlo, y almacenarlo en memoria, con la posibilidad de escuchar el audio mediante playback.

Para el desarrollo del PCB se utilizará la herramienta Altium Designer 19. Para los proyectos, las herramientas Vivado, Vivado-HLS y Vitis del fabricante Xilinx.

3. Fases de realización

Las fases que se definieron para la realización de este TFG son:

1. Realizamos un estudio previo para identificar los periféricos a añadir, para ello primero indagamos sobre las características técnicas y con cuales periféricos ya cuenta la Zybo, para así elegir unos componentes que sean compatibles y complementen a este dispositivo. Además, a la hora de elegir nos aseguramos de que los componentes estén en stock.
2. Desarrollo del PCB mediante la herramienta Altium Designer 19, consistiendo en dos tareas secuenciales principales.
 - Definición del esquemático, y con ayuda de las hojas de datos de los fabricantes realizar una conexión coherente de las señales y componentes de la placa.
 - Diseño del PCB, colocando y enrutando las señales correctamente.
3. Desarrollamos dos proyectos para verificar la placa, el primero, simple, puramente en Verilog, para comprobar el correcto funcionamiento de los displays. El segundo fue más complejo, pues al diseñar con el Vivado IP Integrator debemos ser consciente del funcionamiento de cada IP-Core, acudiendo a la documentación de referencia. Aquí la clave es partir de algún proyecto ya desarrollado e irse familiarizando con los IP-Cores más habituales, dicho esto partimos del proyecto de Diligent, Zybo DMA Audio Demo [5].
4. Redacción de la memoria, con la intención de transmitir los conocimientos adquiridos de la manera más clara y concisa posible.

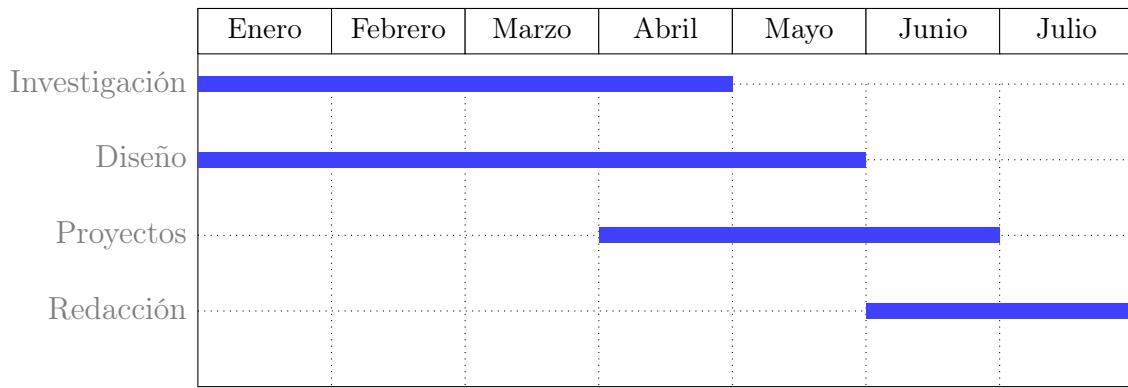


Figura 2: Diagrama de Gantt.

4. Estructura de la memoria

Estado del arte Antes de lanzarnos a diseñar, en el Capítulo 2 haremos una breve introducción básica sobre los sistemas empotrados reconfigurables, en especial el dispositivo Zynq SoC de Xilinx (comentario, Zybo es acrónimo de Zynq board), estudiaremos qué opciones de periféricos y placas tenemos en el mercado, justificando así la necesidad de fabricar nuestra propia placa. Finalmente, veremos el protocolo de comunicación AXI, fundamental para cualquier proyecto complejo, que aproveche la potencia que ofrece la herramienta Vivado IP Integrator.

Diseño de la placa de expansión: Ya introducido el dispositivo con el que vamos a trabajar, el en Capítulo 3 entramos en lo que al diseño físico de la placa de expansión de refiere, desde la elección de los componentes al estado final para ser enviado a nuestro fabricante de PCB de confianza, Eurocircuits en este caso.

Proyecto básico: Una vez tenemos nuestra placa, vamos a comprobar la funcionalidad de los displays, mediante un proyecto sencillo en Verilog detallado en el Capítulo 4.

Proyecto avanzado: Capítulo 5 en donde ya realizamos un proyecto mas completo en el que manejamos nuestro propio microprocesador empotrado, el MicroBlaze, para una aplicación de procesado de audio que recibimos vía *streaming* a través de bluetooth. Así pues, implementamos el software que se ejecutara en nuestro MicroBlaze, siendo el encargado de gestionar las interrupciones, comunicarse con el módulo y solicitar transferencias de lectura o escritura en memoria al DMA. Finalmente, validamos la funcionalidad de nuestra aplicación siguiendo una serie de pasos más detallados dentro del capítulo y comprobamos que todo funciona correctamente.

Conclusión: Capítulo 6 y final, donde resumiremos los resultados del trabajo y presentaremos los puntos a explorar en futuros desarrollos.

Capítulo II

Introducción y Estado del Arte

1. Sistemas Empotrados Reconfigurables

¿Qué es un sistema empotrado? Un sistema empotrado es un sistema de computación encargado idealmente de una sola tarea. Suelen formar parte de los dispositivos electrónicos con el propósito de controlar funciones específicas dentro de dichas máquinas. La especificidad de los sistemas embebidos contrasta por ejemplo con los *General Purpose Processors* con las que cuentan nuestros ordenadores de sobremesa o dispositivos móviles, y que podemos emplear tanto para edición de texto, audio o vídeo, como centro de entretenimiento multimedia, o para navegar en Internet. Esta especificidad de los sistemas embebidos permite su optimización basada en la tarea que ejecutan logrando un alto rendimiento y usualmente menor consumo de energía. En la figura 59 ilustramos las distintas áreas de aplicación de los sistemas embebidos en ella encontramos los siguientes ejemplos (no limitativos):

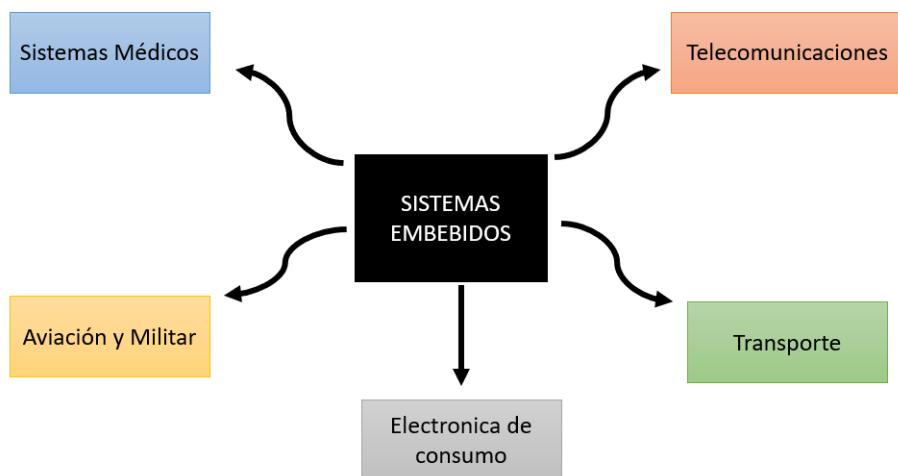


Figura 3: Campos de aplicación de los sistemas empotrados. [4]

- **Telecomunicaciones:** Teléfonos móviles, routers, televisión.
- **Sistemas médicos:** sensores cardíacos, escáner de cuerpo, etc.
- **Electrónica de consumo:** Cámaras digitales, videojuegos.
- **Aviación y militar:** RADAR y SONAR.
- **Transporte:** como GPS.

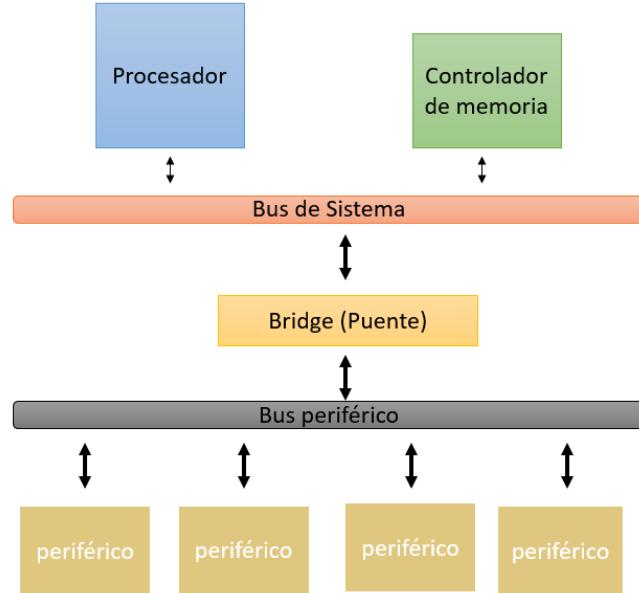


Figura 4: Estructura general de un sistema empotrado. [4]

Estructura genérica de un sistema embebido La estructura básica de un sistema empotrado está compuesta por los siguientes elementos.

- **Procesador** está programado para llevar a cabo las tareas específicas del sistema.
- **Controlador de memoria** Gestiona la escritura y lectura de datos desde o hacia la memoria principal del sistema.
- **Bus de sistema** Un sistema empotrado suele contar con múltiple buses, este suele ser el de mayor rendimiento y conecta al procesador con la memoria principal.
- **Bridge o Puente** Hace de interfaz para hacer compatible la comunicación de buses distintos, por ejemplo un bus de alto rendimiento con uno más lento.
- **Bus periférico** suele ser el bus más lento del sistema, y permite a los periféricos comunicarse entre ellos.
- **Periféricos** componentes que se comunican con el procesador para la aplicación, pueden ser circuitos integrados externos, o pueden ser instancias en el lado de la FPGA en el caso de dispositivos SoC reconfigurables como Zynq.

2. Zynq SoC

Un ejemplo de sistema empotrado reconfigurable es Xilinx Zynq System-onProgrammable-Chip (SoC) que cuenta con una arquitectura que simplificándola al máximo consiste en dos partes principales, Processing System (PS), formada por el procesador de doble núcleo ARM Cortex-A9, memoria integrada, periféricos e interfaces de comunicación de alta velocidad, y Programmable Logic (PL) que es equivalente a una FPGA. Debido a que los componentes de la PS y PL están fuertemente acoplados, la comunicación entre ellos es muy eficiente y no necesita emplear muchos recursos (ver sección 4). Para más detalles sobre los recursos podemos ir al **anexo 1** (PS) o al **anexo 2** (PL).

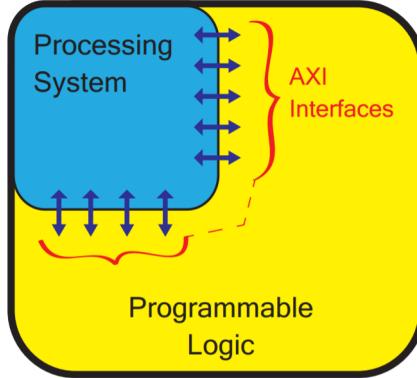


Figura 5: Modelo simplificado de la arquitectura Zynq. [4]

3. ¿Por qué ZYBO?

En el mercado existen una gran variedad de placas de desarrollo con un dispositivo Zynq embotrado, la diferencia como siempre radicará en los recursos (velocidad del procesador, capacidad de memoria, numero de puertos de entradas y salidas disponibles...), el coste, librerías o herramientas proporcionadas por los fabricantes, entre otras. A la hora de elegir tendremos como mínimo que informarnos si se ajustara a nuestras necesidades dependiendo de nuestro objetivo (profesional, pasatiempo o educativo), aplicación, presupuesto y un largo etc. Dicho esto, vamos a un tema apasionante para quienes estamos interesados en la industria de dispositivos electrónicos, las especificaciones técnicas.

Si somos usuarios principiantes/intermedios en el desarrollo con tecnología FPGA, lo más probable es que nuestros diseños no tengan requerimientos de alta densidad de entradas/salidas, en este caso las prestaciones de la placa Zybo encajan perfectamente.

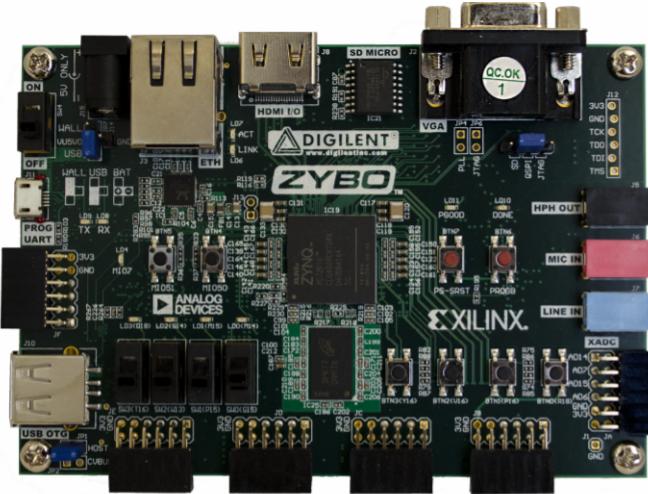


Figura 6: Zybo (desde 276,39 €) . Destacar el dispositivo Zynq en el centro de la placa que contiene un procesador de doble núcleo ARM a 650 MHz, entre los botones y los switches, la memoria DDR3 de 512 MB, además de todo lo mencionado en la introducción, mención especial de los 6 conectores *Pmod* para entrada/salida, pues serán los únicos disponibles para expandir las capacidades de la Zybo.

Como menciono en la figura 6, los conectores *pmods* son la única vía para extender las prestaciones de la placa en cuando a periféricos. Prueba de esto, son la gran variedad de

módulos *Pmod* que nos ofrece el propio fabricante Digilent.

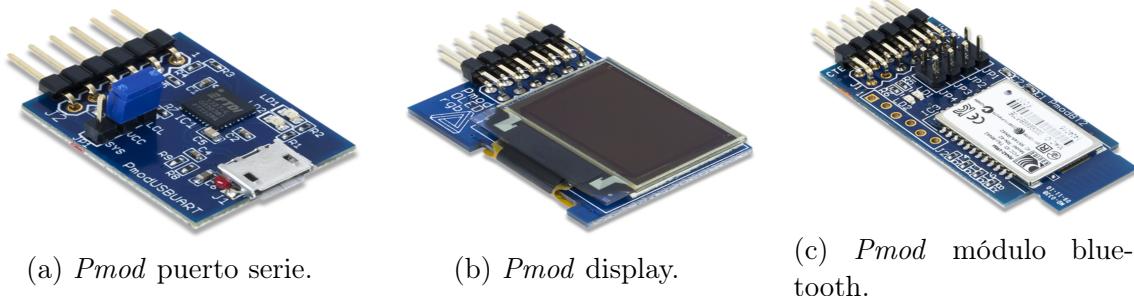


Figura 7: Pack de *pmods* que deberíamos adquirir para una aplicación similar a la del proyecto.

En el pack anterior, son diseños cerrados, no podemos elegir por ejemplo qué módulo queremos que venga en nuestro *pmod*, y además el puerto serie no cuenta con aislamiento con la alimentación de entrada del USB, produciendo situaciones extrañas al conectarlo sin alimentar previamente la placa Zybo.

La Zybo no es la única opción para empezar, otra alternativa más *low cost* y que puede abrir el abanico de experimentos a aquellos que ya están familiarizados con el desarrollo en Raspberry-Pi, es la ZynqBerry, pues dispone de un conector de 40 pines pensado para conectarse con periféricos creados para Raspberry-Pi.

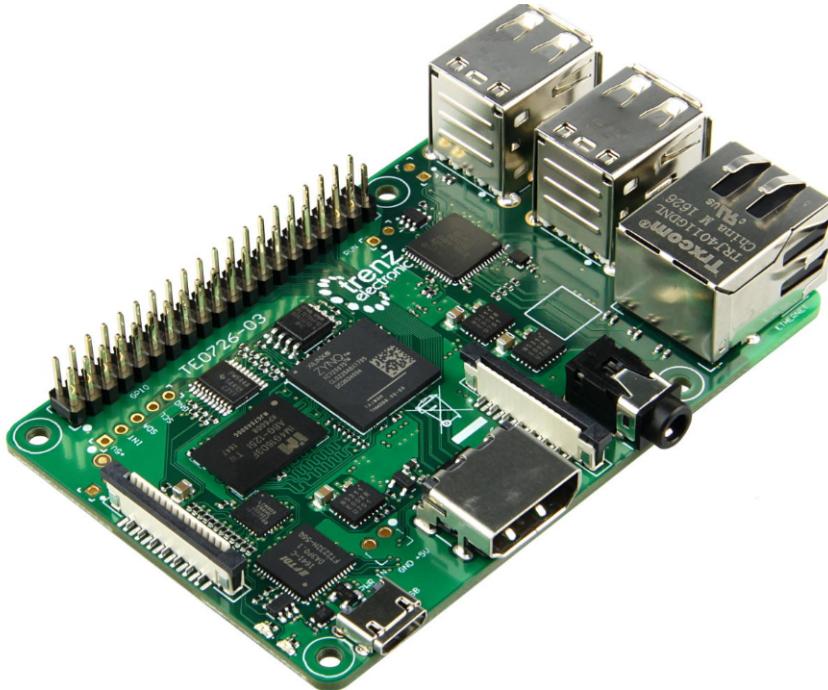


Figura 8: **ZynqBerry (desde 80.08 €)**. En este caso la reducción sustancial del precio es consecuencia de una bajada en las prestaciones, contamos ahora con una Zynq (XC7Z007S) que dispone de un procesador ARM de un solo núcleo a 766 MHz. En la página oficial del fabricante podemos encontrar demos como [esta](#), en la que se conecta una cámara a la placa mientras corre Linux.

Vistas las Zybo y la ZynqBerry, de medio y bajo coste, y con un set de periféricos más bien escaso, y muy limitados puertos de expansión (*conectores pmods* en el caso de la Zybo

y *header* Raspberry en el caso de la ZynqBerry). Pasamos a la placa MYD-C7Z010/20 que introduce el concepto System on Module (SOM), pues contamos con una placa base (o de expansión, depende el punto de vista) con una extensa lista de conectores como puerto serie RS232, LCD / Touch Screen que permite conectar módulos de pantalla LCD Táctil de 4.3 y 7 pulgadas. Además los ya comunes en este tipo de placas (Ethernet, HDMI...), y le conectamos otra placa como si fuera un módulo, esta ultima es donde se encuentra del dispositivo Zynq y la memoria principal.

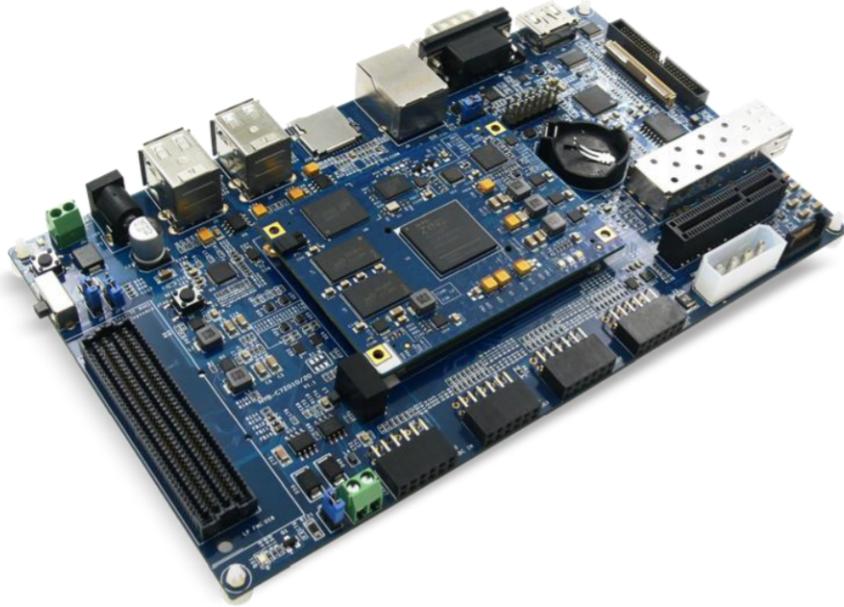


Figura 9: **MYD-C7Z010 (desde 247,19€) + MYC-C7Z010 CPU Module (desde 114.31€)**. Vemos como la placa de desarrollo (en el centro de la placa base) está conectada como un módulo en la placa de expansión, la cual permitirá compatibilidad con distintos módulos. El fabricante proporciona dos modelos, tanto para la placa base como para la que contiene la CPU en el pack de compra, con mas o menos prestaciones.

4. AXI Interface

AMBA Es una estándar abierto de interconexiones SoC fabricado por ARM . Permite la conexión y configuración de controladores y periféricos, a una alta frecuencia y rendimiento. Dentro de la esta familia de buses se encuentra el protocolo AXI, que esta optimizado para su implementación en FPGA y se utiliza como medio de comunicación entre los distintos IP-Cores dentro de un diseño de FPGA. Entre las *features* más importantes del protocolo AXI podemos destacar:

- Fase de direccionamiento y control separada de la fase de transferencia de datos.
- *Byte Strobe*, mediante el cual indicamos en una transferencia que bytes son válidos, permitiendo transferencias desalineadas.
- *Burst-based transactions*, nos permiten enviar una gran cantidad de datos a ráfagas proporcionando solo la dirección de inicio de los datos que se envían.
- Separación de los canales de lectura y escritura, permitiendo menor penalización de tiempo en los accesos a memoria.

- Habilidad para atender múltiples direcciones pendientes.
- Completar transacciones fuera de orden.
- Fácil de añadir más etapas mediante registros para cumplir con el *timing*.

AMBA 4.0 lanzado en 2010 incluye la última versión de AXI conocida como AXI4. Existen 3 tipos, según nuestras necesidades utilizaremos uno u otro:

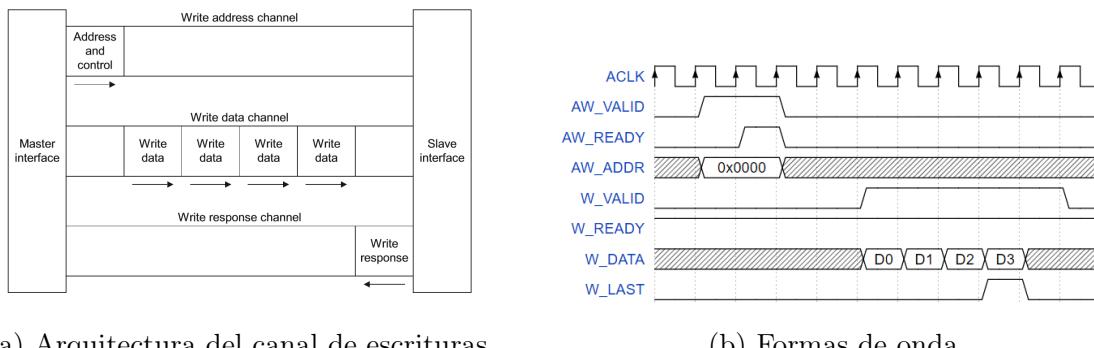
- **AXI4** - La interfaz de alto rendimiento, mediante Memory Mapped Protocol nos permite ráfagas de hasta 256 transferencias de datos por dirección proporcionada.
- **AXI4-lite** - Es una versión más simple y ligera del AXI4. Usada para *Memory Mapped Protocol* de una sola transacción de datos por dirección.
- **AXI-Stream** - Es una interfaz específica para el *Memory Mapped Protocol* de datos, no existen direcciones, solo un *handshake* previo y el flujo de datos es unidireccional del maestro hacia el esclavo para iniciar la transferencia de datos a ráfagas.

4.1. AXI Architecture

En el protocolo AXI, un gran número de AXI maestros pueden conectarse a otro gran número de AXI esclavos mediante un AXI interconnect. Las transacciones que se pueden llevar a cabo son básicamente tres, escritura, lectura o *streaming* de datos.

AXI Write-Burst Transaction

En la figura 10 observamos el proceso de una escritura. Para mayor información dirigirse a la página 29 (para información de las señales) o 37 sobre el proceso de este *handshake*, de la **guía de referencia del protocolo AMBA AXI4** [8].



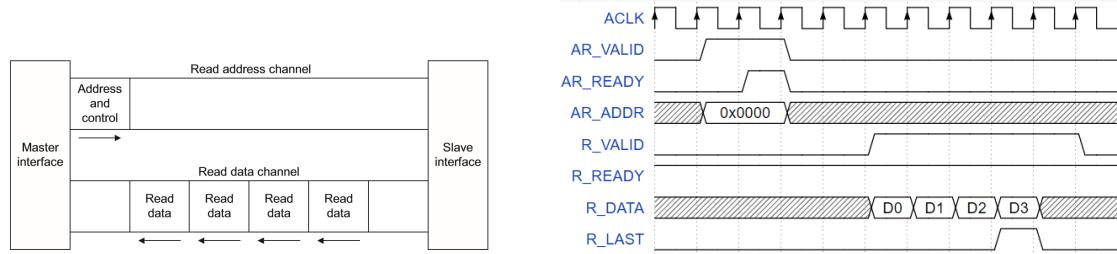
(a) Arquitectura del canal de escrituras. (b) Formas de onda.

Figura 10: Transacciones de escritura AXI4.

El maestro envía la dirección y los datos de control vía **AW_ADDR** y pone **AW_VALID** a '1' confirmando que los datos son válidos, cuando el esclavo establece **AW_READY** se inicia la transferencia en **W_DATA** y el maestro vuelve a colocar **AW_VALID** a '0'. **W_LAST** a '1' indica el final del paquete de datos.

AXI Read-Burst Transactions

En la figura 11 observamos el proceso de una lectura. Para mayor información dirigirse a la página 32 (para información de las señales) o página 37 sobre el proceso de este *handshake*, de la **guía de referencia del protocolo AMBA AXI4** [8].



(a) Arquitectura del canal de lecturas.

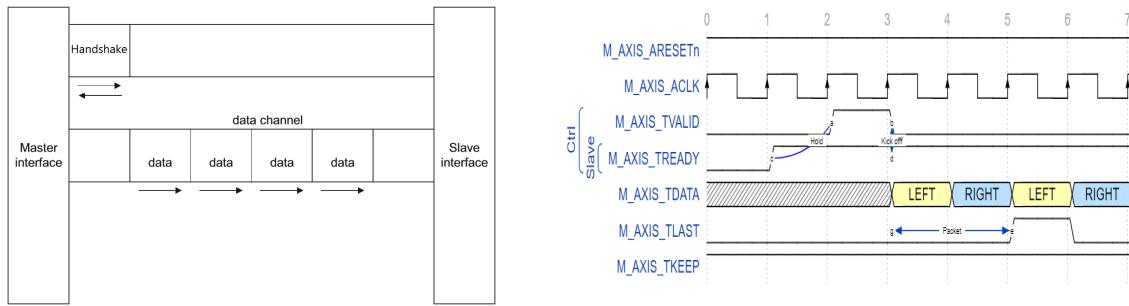
(b) Formas de onda.

Figura 11: Transacciones de lectura AXI4.

En la figura 11b el maestro coloca la dirección y los datos de control en **AR_ADDR** y pone **AR_VALID** a '1', cuando el siguiente flaco coincide que el esclavo ha puesto **AR_READY** '1' el maestro vuelve a poner inmediatamente **AR_VALID** a '0' y se inicia la transacción con el esclavo enviando datos por el puerto **R_DATA**, el último dato de la ráfaga de datos leídos lo indica en este caso el esclavo poniendo **R_LAST** a '1'.

4.2. AXI Streaming Interface

Este es el protocolo principal en nuestro proyecto por lo que detallaremos brevemente la gestión del *handshake* inicial junto con sus señales más importantes.



(a) Arquitectura del canal de Streaming.

(b) Formas de onda.

Figura 12: Streaming de audio usando AXI Stream.

En la figura 12b observamos como vamos a mover los datos de audio en nuestro proyecto, ya que estamos manejando audio estéreo LEFT es el dato del canal izquierdo y RIGHT es el del canal derecho, iremos indicando el tamaño de paquete por medio de la señal **M_AXIS_TLAST**.

Existen tres escenarios posibles antes del inicio de la transferencia, y en todos siempre se cumple la condición de que en el flanco de subida de reloj en el que **TREADY** y **TVALID** ambas estén a '1' se iniciara la transmisión.

Capítulo III

Diseño de la placa de expansión

1. Diseño de la Placa

1.1. Elección de componentes

A la hora de seleccionar nuestros componentes partimos con el proyecto avanzado en mente. Un dispositivo que realiza *playback* de audio debería incluir componentes como un display, algunos botones, memoria, un microcontrolador, un códec de audio, conexión inalámbrica para recibir *streaming* de audio, amplificador. La Zybo ya cuenta con la mayoría (como amplificador usaremos unos auriculares estándar), a excepción de el display, la capacidad de conexión *wireless* y no menos importante un puerto serie para imprimir los comandos en pantalla, clave a la hora de hacer *debug* de nuestra aplicación. En consecuencia, primero elegimos el módulo Bluetooth de Microchip, el BM64, cuyas características son explicadas con más detalle en anexo 12.

Para los display, seleccionamos dos matrices 8x8 de LEDs, en la sección de en la que detallamos su esquemático (figura 16) entramos más en detalle sobre sus características. Finalmente, seleccionamos un driver que haga de interfaz USB externa a la UART de la Zybo, mediante el chip FT232R de FTDI.

Para conocer la lista completa, ir al BOM, localizado en el anexo 13.

1.2. Diseño de esquemáticos

Esquemático Módulo Bluetooth

De este esquemático, lo más remarcable, que no se haya mencionado en el anexo 12 es el añadido del *Ambient Detection*, para poder utilizar el el convertidor analógico a digital que tiene el módulo incorporado, utilizando un termistor de 100K Ohms. En principio su propósito es monitorizar la temperatura en caso de que incluyamos una batería para el módulo, no es nuestra situación, lo usaremos como sensor de temperatura ambiente.

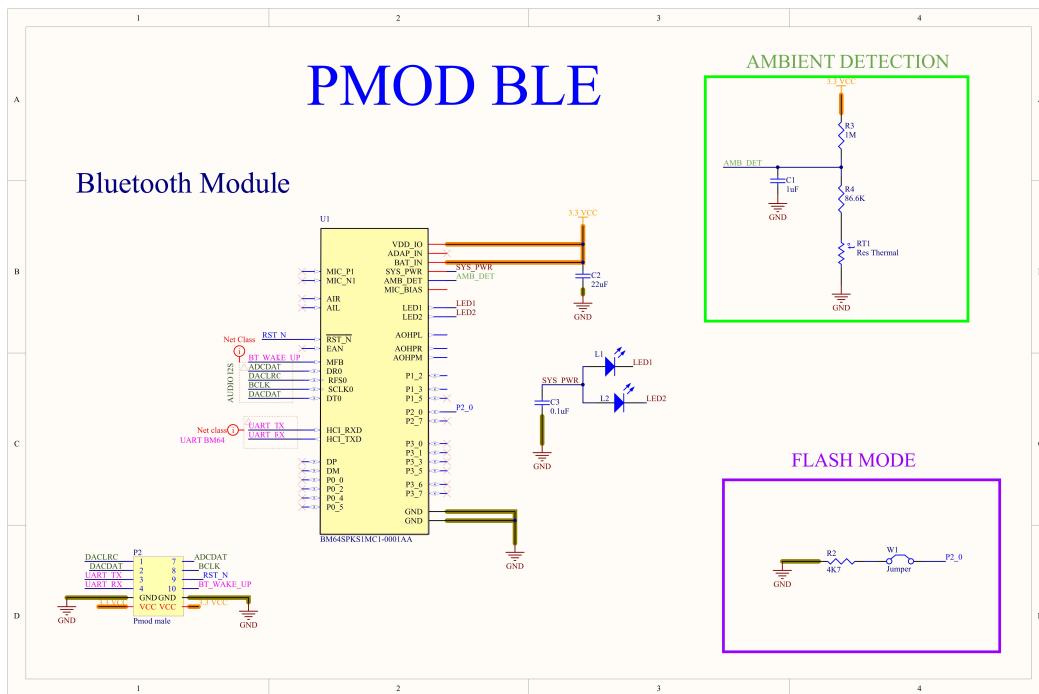


Figura 13: Esquemático del módulo bluetooth. Los bloques de sensor de temperatura (en verde) y el modo de inicio (en morado).

Esquemático puerto serie

Con la intención de estudiar los pasos que deberíamos seguir si queremos cumplir con la certificación CE, a la interfaz del USB-UART hemos añadido un diodo para protección de descargas electrostática (ESD), un par de ferritas para filtrar altas frecuencias espurias, y finalmente un driver para trasladar el voltaje de trabajo del USB 5 V a 3.3 V de la Zybo que también proporciona aislamiento en las alimentaciones, asegurándonos de que si tenemos el cable USB conectado al dispositivo, no va a proporcionar corriente a la Zybo.

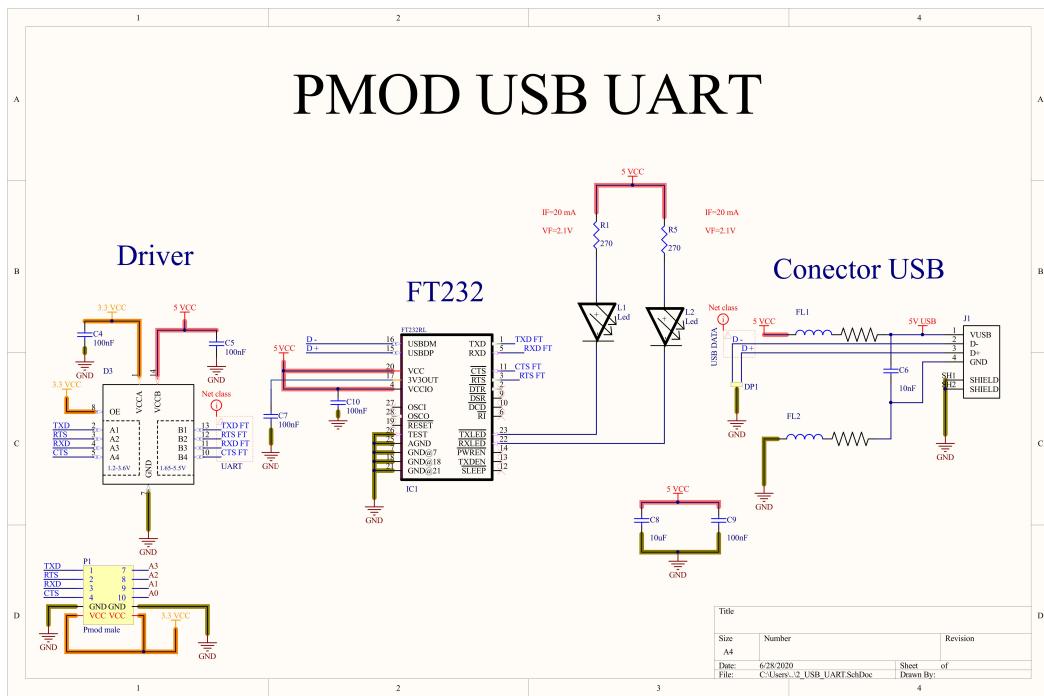
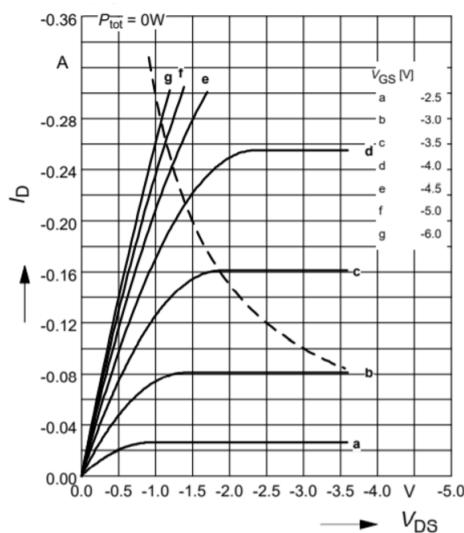


Figura 14: Esquemático puerto serie

Esquemático display

Aprovechando los 4 pines que quedaron libres de la conexión anterior con el conector *pmode* del puerto serie, vamos a utilizar un demultiplexor 4-16 para controlar las columnas, que va a los ánodos comunes, los cuales no se manejan directamente con el demultiplexor sino con transistores PMOS, como vamos a trabajar a 3.3 V como máximo elegimos unos transistores con las características de la figura 15, asegurándonos que estamos por encima del valor $-V_{GS\ threshold}$, fijando el camino de la corriente para los leds.


 Figura 15: Características del drenador. $I_D = F(V_{DS})$.

Para el caso de las filas, estas estarán conectadas a al otro *pmods* a través de resistencias que fijen la corriente de los leds.

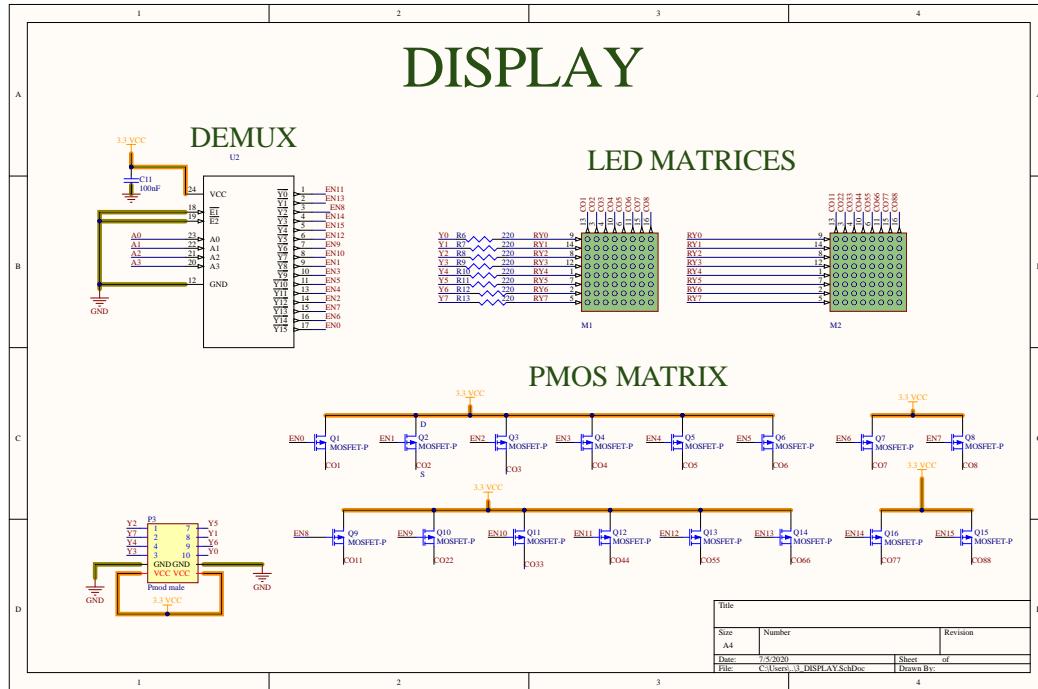


Figura 16: Esquemático de los displays. La conexión los transistores conectados a 3.3 VCC (drenador) columna del display (fuente) y demultiplexor (puerta).

1.3. Diseño del PCB

Una vez hemos compilado nuestro esquemáticos sin errores de conexión, corto circuito, etc. No ponemos en marcha con el diseño del PCB, antes de nada, nos vamos a layer stack manager, y establecemos el numero de capas de nuestro PCB y el espesor de cobre y de dieléctrico (debemos consultar nuestro fabricante), en nuestro caso será un doble capa con espesor de cobre de $35\mu m$ (ver figura 17). Posteriormente establecemos nuestras reglas de diseño, las cuales serán lo menos restrictivas posibles y podemos ver las mas importantes en la figura 18.

#	Name	Material	Type	Weight	Thickness	Dk
	Top Overlay		Overlay			
	Top Solder	Solder Resist	Solder Mask		0.01mm	3.5
1	Top Layer		Signal	1oz	0.036mm	
	Dielectric 1	FR-4	Dielectric		0.32mm	4.8
2	Bottom Layer		Signal	1oz	0.036mm	
	Bottom Solder	Solder Resist	Solder Mask		0.01mm	3.5
	Bottom Overlay		Overlay			

Figura 17: Estructura de nuestro PCB.

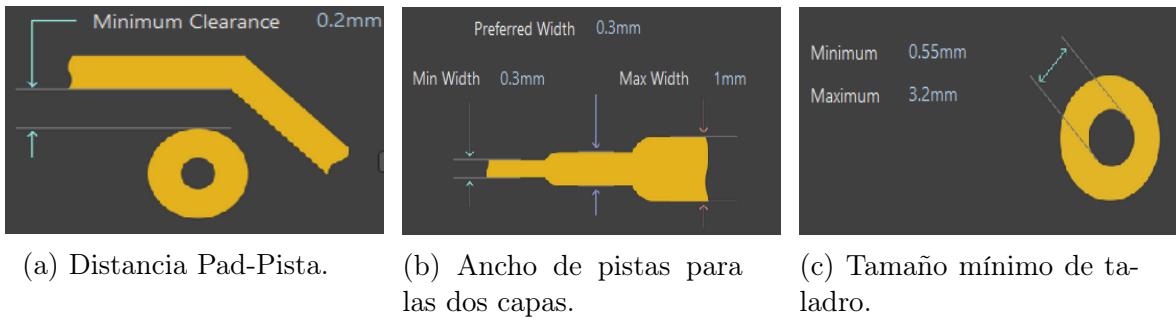


Figura 18: Reglas mas importantes en nuestro diseño.

Una vez hemos realizado el *set-up* empezamos a colocar nuestros componentes con coherencia y cumpliendo los estándares de medidas impuestos, como por ejemplo la distancia entre *pmod*, el ancho de la Zybo, y que los condensadores y componentes de protección ESD estén cerca de la línea que desacoplan o protegen respectivamente. Para ayudar a facilitar unos caminos de corriente de retorno mas cortos, esto es, para baja frecuencia, el camino de menor impedancia, y para altas frecuencias el camino de menor inductancia, hemos colocado en la capa bottom, un polígono de cobre que hará como plano de masa (estará conectado al plano de masa de la Zybo) y con esto mitigar el *cross-talk* en nuestra placa.

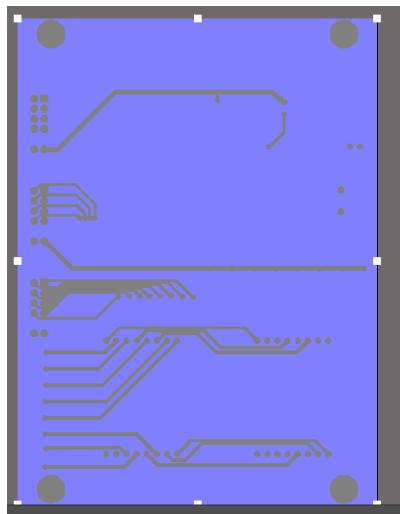


Figura 19: Polygon Pour en la capa bottom.

El PCB final tras enrutar es el siguiente:

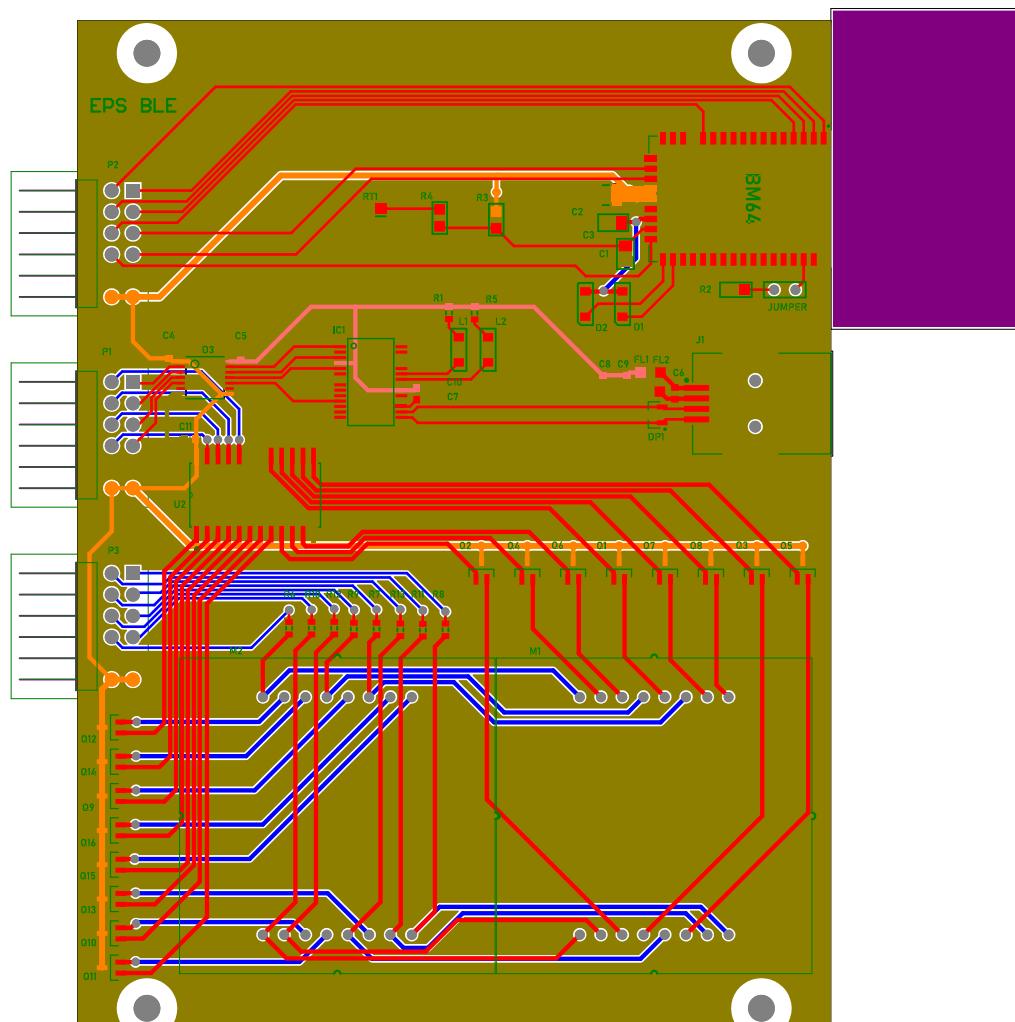


Figura 20: PCB final.

Finalmente tras fabricar, y soldar los componentes en el laboratorio, en resultado final es el de la figura 21.

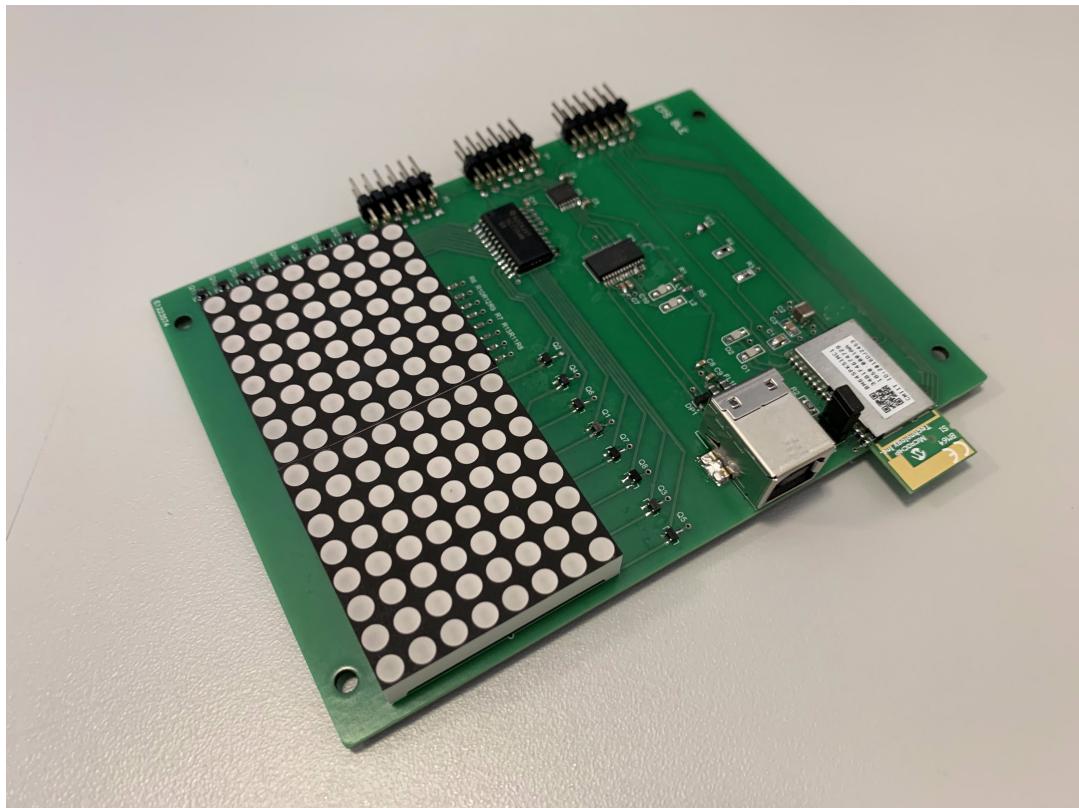


Figura 21: Placa de expansión construida.

Capítulo IV

Proyecto básico

1. Implementación en Verilog

Teniendo la placa lista, vamos a realizar una pequeña prueba de dos dispositivos en concreto, los displays. Para ello en Verilog vamos a implementar la práctica de la figura 22.

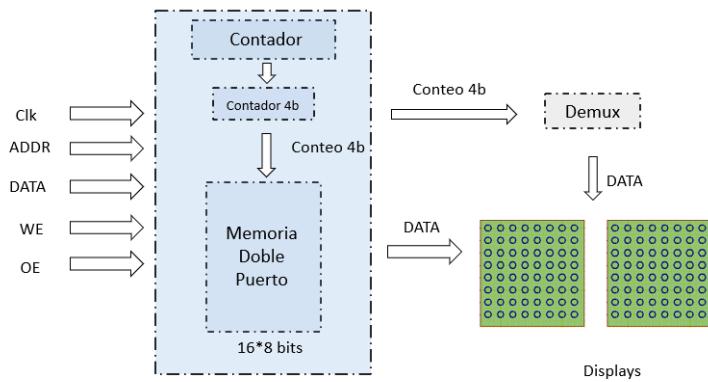


Figura 22: Control de los displays

Consiste en hacer usos de una de las memorias distribuidas de la Zybo por inferencia, con lo cual forzamos el uso de este recurso que tenemos disponible siendo más eficiente que la implementación tradicional por medio de LUTs, para guardar los datos que contienen la fila de LEDs que vamos a activar en cada momento. En cuanto a las columnas, las habilitaremos/deshabilitaremos mediante el demultiplexor que tenemos en nuestra placa.

Al ir activando una fila y columna a la vez, utilizamos el reloj que nos llega de la Zybo ($Clk = 125 \text{ MHz} = \frac{1 \text{ Cycle}}{1 \text{ Sec}}$) y gracias al contador obtenemos una frecuencia de $125 \text{ Hz} = \frac{1e6 * Clk * Cycles}{1 \text{ Sec}}$ de refresco para los datos, por encima de la frecuencia a la que el ojo humano aprecia el *flicker* de un led.

Para probar el correcto funcionamiento hemos escrito un testbench cuya función es llenar inicialmente las 64 posiciones de memoria de datos poniendo $WE = 1$ para habilitar la escritura a memoria, y un bucle for, después deshabilitar la escritura $WE = '0'$ y Habilitar el output $OE = '1'$ para así ir monitorizando los cambios en o_demux e imprimiendo el tiempo cuando estos se produzcan gracias a un *model* que irá contando y esta sincronizado con nuestro control e imprimirá los datos en pantalla, como vemos en la figura 23.

1. IMPLEMENTACIÓN EN VERILOG

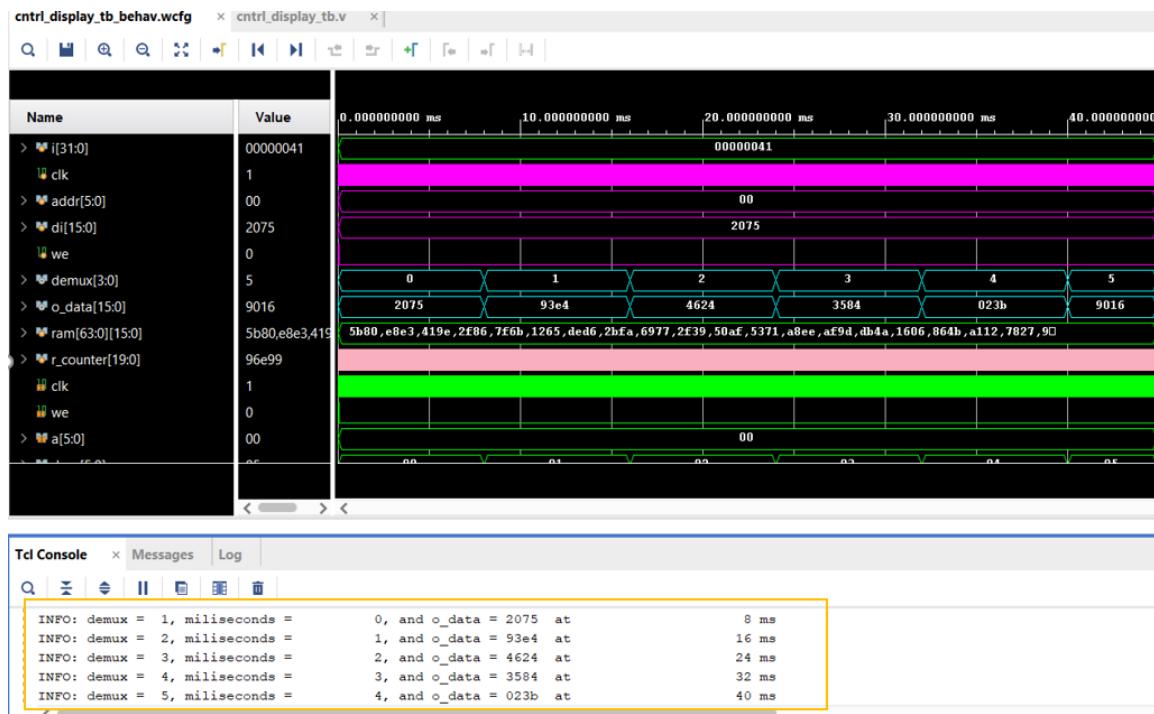


Figura 23: Llenamos de datos la memoria inicialmente, y ya luego habilitamos las salidas con OE. Y vamos generando el contador o_demux de 4 bits. El código de los archivos simulación lo encontramos en el anexo 10.

El control de los display quedaría como el código 1.

```

1 module cntrl_display#(
2     parameter ms_limit = 1000000 // Frequency divider 125 MHz (8 ns) to 125hz (8 ms)
3 )(
4     input clk, //Clk
5     input [5:0] i_addr,//ADDR
6     input [15:0] i_di, //DATA
7     input i_we, //WE
8     input i_oe, //OE
9     output reg [3:0] o_demux,// Contador 4b
10    output reg [15:0] o_data// DATA
11 );
12    wire [15:0] w_spo;
13    wire [15:0] w_o_data;
14 //Instancia de la Memoria
15 rams_dist mem(
16 .clk(clk),.we(i_we),.a(i_addr),.dpra({2'b00,o_demux}),.di(i_di),.spo(w_spo),.dpo(w_o_data)
17 );
18
19 reg [$clog2(ms_limit)-1:0] r_counter = 0;
20 initial o_demux = 0;
21 always@(posedge clk)
22 begin
23     if(r_counter == ms_limit)
24     begin
25         o_demux <= o_demux + 1;
26         r_counter <= 0;
27     end
28     else
29         r_counter <= r_counter + 1;
30 end
31 //data_out
32 always@(posedge clk)
33 if(i_oe)
34     o_data <= w_o_data;
35 endmodule

```

Listing 1: control de los displays.

Capítulo V

Proyecto avanzado

1. Resumen de la proyecto

El proyecto consiste en recibir audio digital estéreo de calidad CD a 48 kHz por bluetooth ya sea desde un móvil u otro dispositivo con tecnología bluetooth, procesarlo aplicándole un filtro FIR y un posterior almacenamiento del resultado en memoria, con la posibilidad de realizar *playback* del resultado . La secuencia sería como en la figura 24.

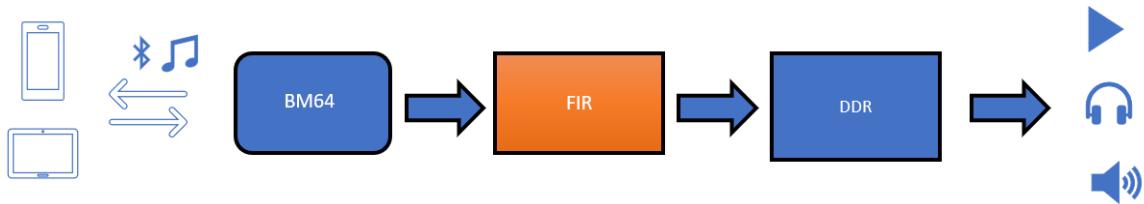


Figura 24: En el Bloque FIR será donde se lleva a cabo las operaciones con el audio digital.

Para plasmarlo en la práctica, haremos uso del **Vivado IP Integrator** que es un flujo de diseño basado en la conexión de IP-Cores, que son bloques funcionales que pueden representar un microcontrolador, una FIFO, una interfaz, etc. El IP Integrator cuenta con una interfaz gráfica interactiva mediante la cual podemos instanciar los Cores que vamos a utilizar, proporcionando conexiones automáticas inteligentes entre dichos componentes y facilidad de *debug*. Así se agiliza bastante el tiempo de desarrollo, y tenemos más control al diseccionar el proyecto en bloques funcionales configurables. En la figura 25 observamos los Cores y las conexiones que son necesarias para el proyecto.

1. RESUMEN DE LA PROYECTO

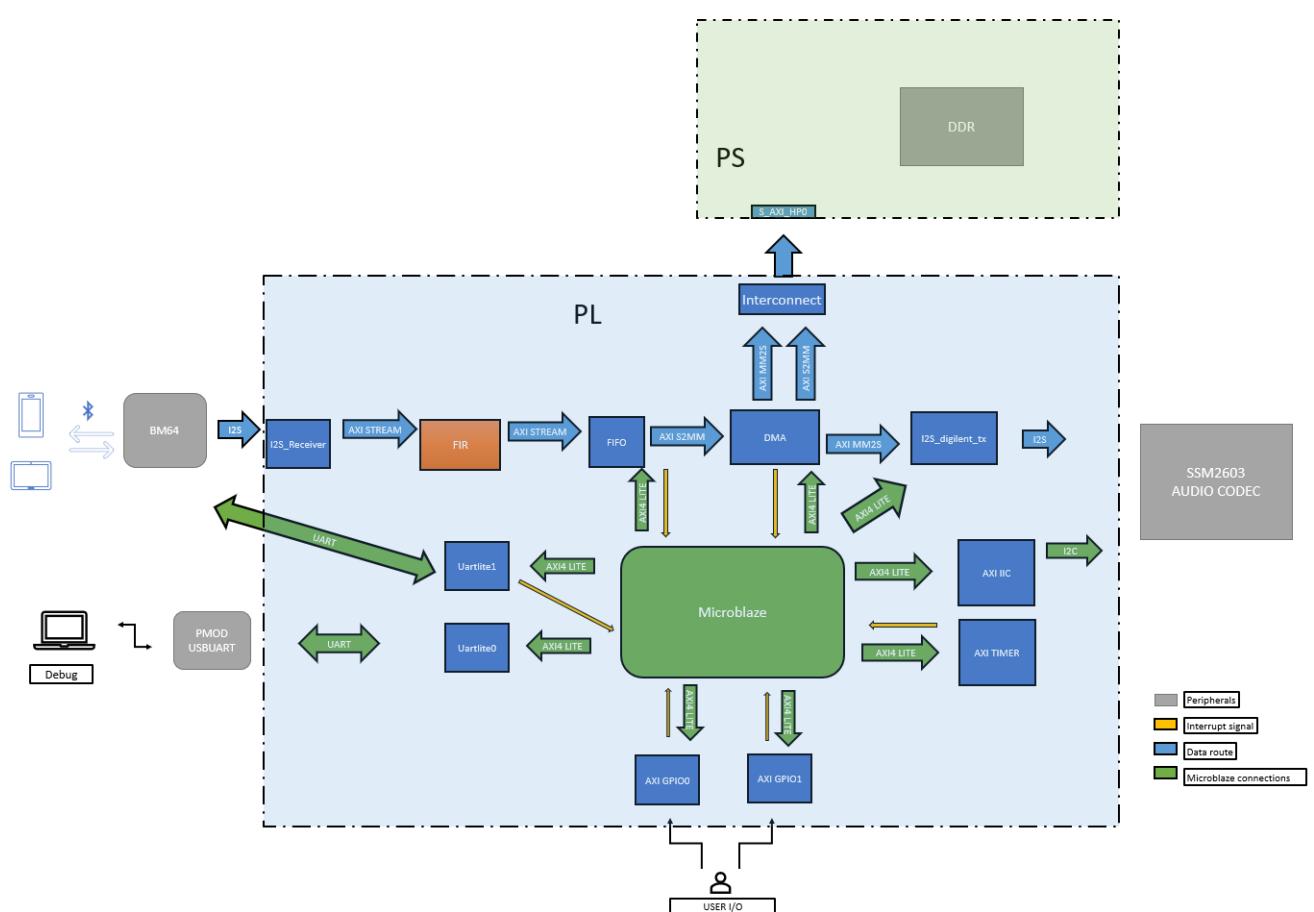


Figura 25: Implementación de nuestro proyecto al completo en Vivado IP Integrator.

A continuación nos detendremos brevemente en cada Core y su función en el proyecto.

2. MicroBlaze

MicroBlaze es un *Soft-Processor* con arquitectura Reduced Instruction Set Computer (RISC) soportado por XILINX y por tanto optimizado para sus FPGA. Debido a que nuestro diseño tiene una tarea es específica y no requiere sistema operativo, la opción que hemos escogido es implementar el MicroBlaze pues gracias a su flexibilidad a la hora de implementarlo debida a una serie de *presets* o configuraciones que podemos modificar, según la aplicación que vamos a desarrollar y nuestros requerimientos. Por ejemplo, dicho abanico va desde la configuración más básica ('*minimun area*') que necesita 900 LUTs, 700 FFs, 2 BRAM, a los 3800 LUTs, 3200 FFs, 6 DSP48E1s and 21 BRAM para '*optimized area*'. Las funciones de dicho procesador serán las siguientes:

- Establecer todas las configuraciones iniciales de los distintos Cores y periféricos de nuestro diseño.
- Gestionar las diversas interrupciones generadas por:
 - Acción del usuario en los botones y *switches* de la Zybo.
 - *Flags* de estado de la FIFO.
 - Mensajes de estado del módulo de bluetooth.
 - Acción del usuario en los botones y *switches* de la Zybo.

En el anexo 3 entramos en mas detalle sobre los parámetros de configuración utilizados.

3. DMA

Para lograr almacenar nuestros datos resultado en memoria, primero debemos acceder a ella, la manera más fácil para acceder a memoria es a través de uno o más de los 4 High-Performance (HP) de interfaz AXI, se pueden configurar para 64 o 32 bits, nosotros usaremos la configuración de 32 bits, la ruta de acceso la podemos observar en la figura 26.

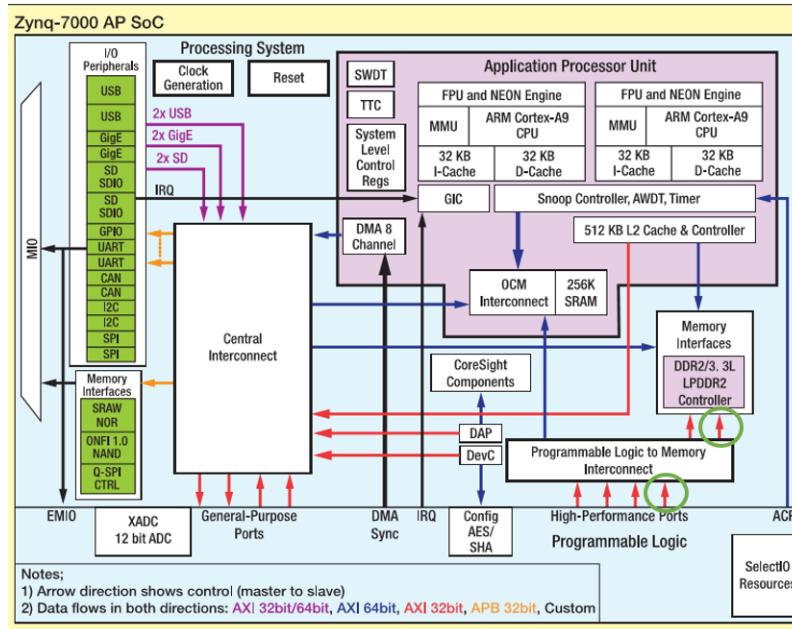


Figura 26: Siguiendo las flechas rodeadas del círculo verde llegamos al DDR SDRAM Controller [1].

Necesitamos el AXI Direct Memory Access (DMA) Core [10] para transformar los datos que recibimos en por AXI STREAM en la PL a transacciones Memory Mapped Protocol para almacenarla en la DDR SDRAM a través del canal de stream to memory-mapped (S2MM), y para leer de la memoria y transformarla de nuevo a AXI STREAM mediante el canal memory-mapped to stream (MM2S).

Además al delegar las movimientos de datos al DMA, liberamos de trabajo a nuestra CPU en nuestro caso al MicroBlaze, pues solo debemos solicitar la transferencia de memoria al DMA y este se encargara de llevar a cabo la transacción de memoria dejando a nuestro procesador disponible para la gestión de la interrupción generada por las muestras de audio que llegan a nuestra FIFO.

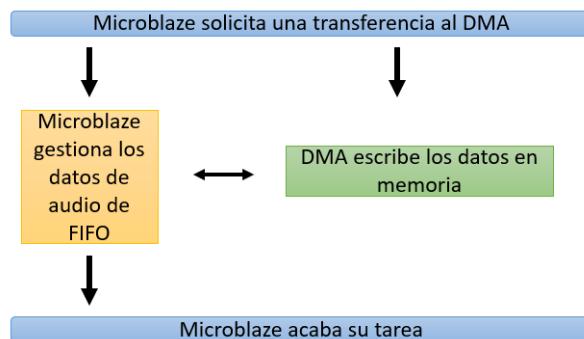


Figura 27: Interacción MicroBlaze-DMA.

Los detalles de la configuración de nuestro DMA lo podemos encontrar en el anexo 5.

4. I^2S Receiver

Al revisar el anexo 12, vemos que el módulo bluetooth implementa un bus de audio Inter-IC Sound (I^2S) y que por defecto el audio tiene una frecuencia de muestreo de 48kHz y una resolución de 16 bits. Este Core será en encargado de hacer de puente entre la interfaz I^2S del módulo y la AXI Stream como vemos en la figura 28.

Información sobre las particularidades de este Core deberíamos ir al anexo 4.

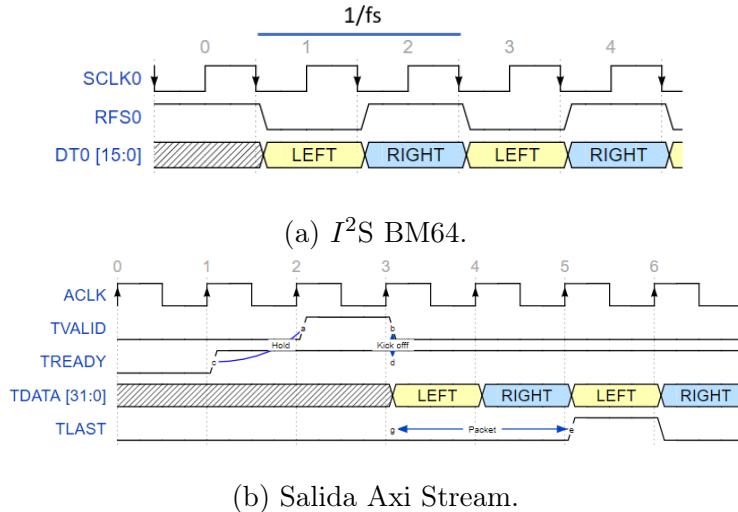


Figura 28: En la figura 28a observamos las formas de onda de de audio que nos llega del módulo a nuestro Core, y en la figura 28b observamos la salida a la interfaz AXI Stream.

5. Digilent I^2S Transmitter

La ZYBO cuenta con un códec de audio integrado, el **SSM2603**, lo utilizaremos como esclavo I^2S para realizar el *playback* de los datos almacenados en memoria. Por lo que con el DMA leeremos de la DDR SDRAM y enviaremos los datos de audio por el canal MM2S para transformarlos a AXI STREAM, necesitando finalmente una última transformación de AXI Stream a I^2S . Para ello usaremos la IP proporcionada por Digilent axi i2s audio, que contiene dos FIFOs para hacer de *buffer* con los datos audio de cada canal y un divisor de frecuencia que exige a la entrada una frecuencia de 100 MHz para generar a la salida las señales de reloj MCLK (12.288 MHz) necesaria para el correcto funcionamiento del SSM2603, además de generar BCLK y PBLRC.

En el anexo 6 se listan los pasos para utilizar este Core.

6. AXI I^2C IP

Para configurar el códec de audio **SSM2603**, primero debemos ser capaces de comunicarnos con el vía Inter-Integrated Circuit (I^2C) bus, por lo que emplearemos la IP disponible en el paquete básico de Vivado y soportada por Xilinx, el AXI IIC Core, configuraremos los parámetros de acuerdo con las especificaciones del SSM2603 [3].

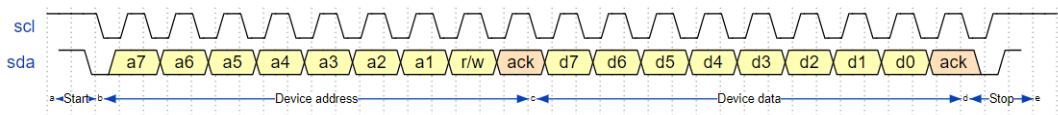


Figura 29: Formas de onda genérica de la comunicación I^2C . Es una transmisión síncrona, Half Duplex, en la que un dispositivo hace de maestro y los demás de esclavos, el bit de *start* se consigue poniendo un cero en el flanco de bajada del reloj, se transmite los bits de la dirección del esclavo seguido un bit de operación que puede ser de escritura y/o lectura, esperamos por el reconocimiento (ACK) del esclavo y enviamos los bits de datos.

7. AXI UARLITE IP

Los dos Cores UARLITE tienen tareas bien diferenciadas:

- **Uartlite0** cuya función será ayudarnos a imprimir datos en la consola a la hora de *debuggear* nuestra aplicación. Para ello hará uso del puerto serie de la placa que hemos construido.
- **Uartlite1** nuestro módulo cuenta con una interfaz UART, por tanto con este Core configuraremos el modulo por software, para ello configuraremos el Uartlite1 con los parámetros que definimos en el capítulo 12.

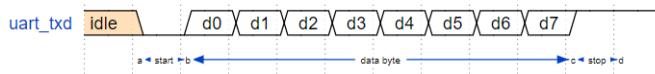


Figura 30: Formas de onda genérica de la comunicación UART. Es una transmisión asíncrona, en la que transmisor y receptor deben estar de acuerdo con el *baud rate* o bits por segundo, y tienen linea de transmisión y recepción separadas. La línea se mantiene en activo alto indicando que esta disponible (*Idle*), el transmisor coloca la línea a 0 (*Start bit*). y transmite el *byte* bit a bit, finalmente vuelve a colocar la linea a 1 (*Stop bit*).

La configuración de estos Cores podemos encontrarla en el anexo 7.

8. AXI GPIO IP

Usaremos el bloque AXI GPIO para generar las interrupciones de fuentes como los botones y *switches* de la Zybo.

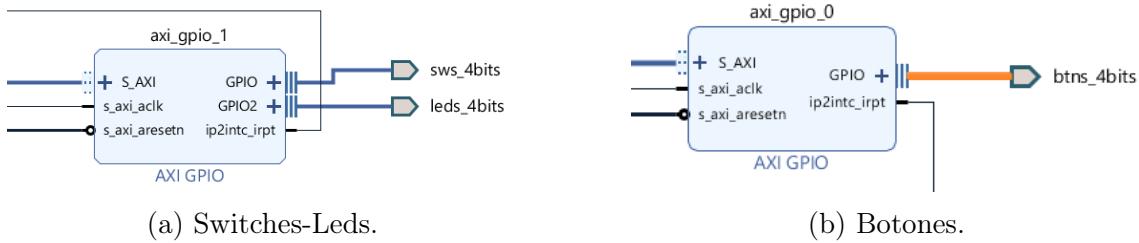


Figura 31: En la figura 31a activamos doble canal, donde la entrada serán los Switches y las salidas los led, que nos informaran que tenemos activo y que no (usaremos solo 2 switches, para encendido y reinicio del módulo). Por último en la figura 31b solo tiene un canal activo y serán los botones con los que indicamos la acción a realizar en nuestra aplicación.

9. AXI STREAM FIFO IP

Como nuestro procesador trabaja a una frecuencia mucho mayor a la que llegan y se procesan los datos de audio por el filtro, emplearemos una FIFO como buffer para guardar los datos en memoria en el orden en el que fueron generados, aquí es donde entra en juego el Core AXI Stream FIFO, pues con el podemos implementar dos FIFOs, una para recibir y otra para transmitir datos, ambas con interfaz AXI STREAM y AXI4 lite, esta última nos permite la gestión de los datos entre las FIFOs desde nuestro MicroBlaze.

Como a la salida nuestro filtro FIR, mantenemos la estructura de los paquetes , es decir, generamos TLAST cada dos palabras, nuestra FIFO de recepción actualizara su contador cada vez que reciba un paquete completo, y generaremos una interrupción para mover los paquetes de la FIFO de recepción a la de transmisión.

Una vez explicada su función, vamos con su configuración, la cual debemos comprender para relacionarla con el funcionamiento de nuestro sistema en la parte del software, por lo tanto, tendremos en cuenta una serie de campos:

- **Interfaz:** elegimos AXI4, que es totalmente compatible con nuestro MicroBlaze.
- **Transmit/Receive Fifo Depth:** cantidad de palabras de 32 bits que puede almacenar, para nuestros requerimientos con el mínimo (512) es más que suficiente.
- **Transmit/Receive Fifo Empty Threshold:** Flag que indica hasta qué número de palabras almacenadas el flag de estado de FIFO vacía está activo, podemos dejarlo por defecto.
- **Transmit/Receive Fifo Full Threshold:** Flag que indica el número de palabras almacenadas a partir del cual el flag de estado de FIFO llena se activa, podemos dejarlo por defecto.
- **Enable Transmit Cut Through:** Uno de los puntos críticos del diseño, si no lo habilitamos, la FIFO de transmisión no podrá transferir una cantidad de datos mayor que su capacidad de almacenamiento, pues el modo por defecto *Store-and-Forward Mode* debemos indicar el tamaño de datos que vamos a transferir antes de empezar la transmisión y nunca debe ser mayor a la capacidad de la FIFO. Sin embargo, en Cut Through podemos ir transfiriendo y leyendo el contador de *byte* transmitidos y generar TLAST cuando el contador alcance el número de *bytes* que queremos transmitir.

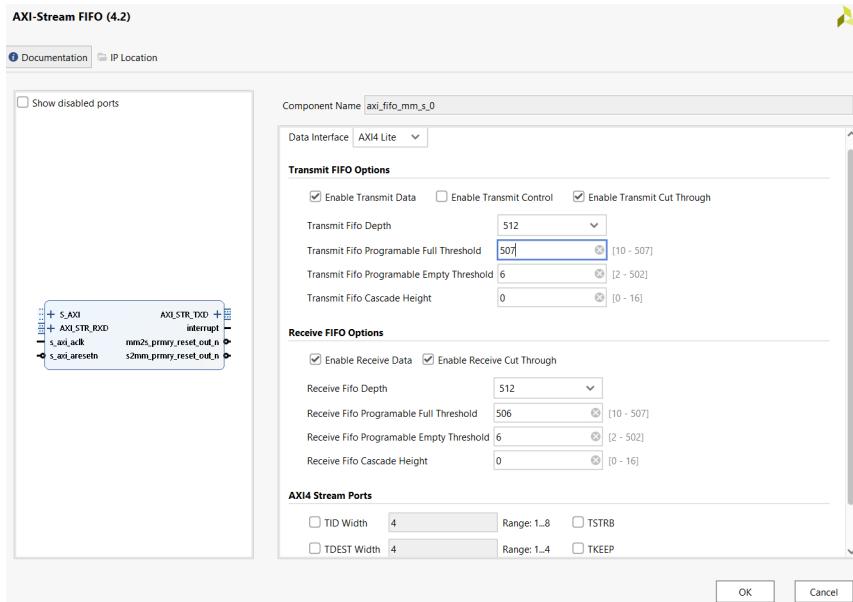


Figura 32: Configuración AXI Stream FIFO.

10. FIR

Uno de los algoritmos más extendidos para el procesamiento digital de audio son los filtros finite impulse response (FIR), usualmente con el fin de atenuar o amplificar una región particular del espectro de la señal de audio. La fórmula matemática que define un filtro es:

$$y_n = \sum_{k=0}^{N-1} b_k * x_{n-k}$$

Figura 33: Ecuación del Filtro FIR.

En la ecuación de la figura 33, $N - 1$ es el orden del filtro, b_k los coeficientes $x[n]$ entrada e $y[n]$ salida. En nuestro caso filtraremos un tono a 4000 kHz de frecuencia de una señal de audio que obtenemos por bluetooth (48 kHz), para ello empleamos un filtro banda eliminada. En la figura 34 se aprecia las especificaciones del mismo, y montando el diagrama de bloques en Simulink como en la figura 35 comprobamos que funciona como esperamos, obteniendo el resultado visual de espectro en la figura 36.

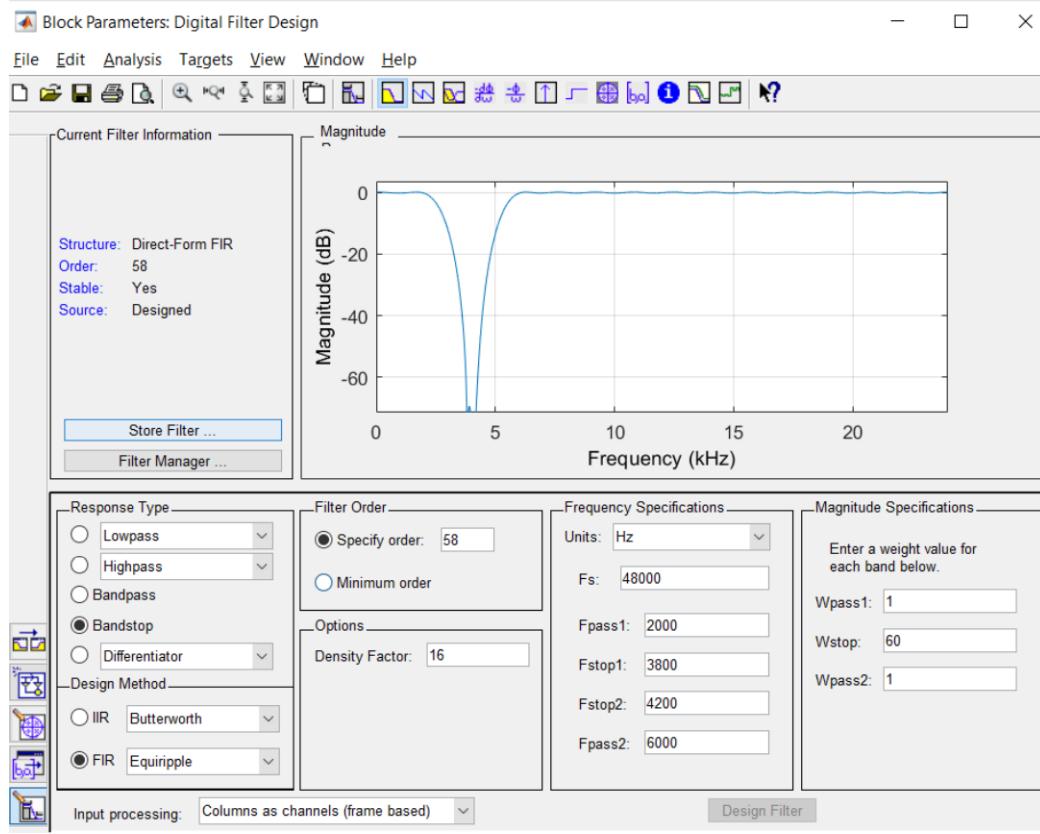


Figura 34: Diseño del filtro en Matlab, con el objetivo de hacer una simulación para tener una referencia de nuestras expectativas al acabar la aplicación.

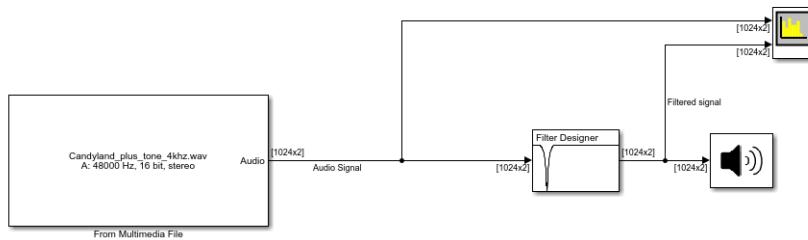


Figura 35: Nuestro aplicación de procesado de audio en Simulink. Cargamos nuestro archivo de audio de prueba al que le hemos añadido previamente el tono y simulamos obteniendo los resultados esperados.

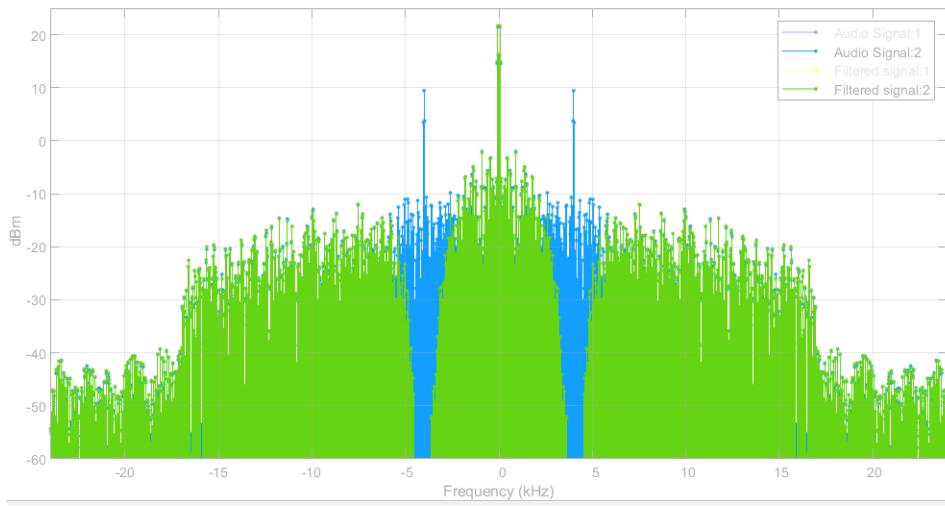


Figura 36: En la imagen los espectros solapados del audio cargado en la figura 35 (en azul), y el filtrado (en verde). Vemos como lo única diferencia es el la zona del espectro alrededor de los 4 KHz que hemos eliminado con el filtro.

11. High Level Synthesis

Para la implementación del algoritmo del filtro finite impulse response (FIR) haremos uso de herramientas de High-Level Synthesis (HLS), en específico el Vivado HLS. Como el filtro FIR es un algoritmo es altamente paralelizable (múltiples operaciones pueden ejecutarse simultáneamente), podemos acelerar dicho algoritmo empleándolo directamente en la FPGA (PL). Para ello utilizaremos Vivado Hls. La ventaja del uso de estas herramientas, es que nos permiten utilizar un lenguaje de alto nivel como C/C++ para especificar las funciones a implementar, y la herramienta implementa dichas funciones en una arquitectura en lenguaje HDL. Tendremos menor flexibilidad en comparación con realizar el diseño usando lenguajes como Verilog o VHDL, pero los tiempos de desarrollo y validación son muchos menores.

Esto se logra mediante dos procesos principales

- **Scheduling** es la traslación de la interpretación del código C/C++ en operaciones en función de su duración en ciclos de reloj, este proceso tendrá en cuenta las *constraints* que hemos definido (hemos establecido una frecuencia de 100 MHz) y los recursos de tecnología disponible, en este caso los recursos de la PL de la Zynq.
- **Binding** Asocia las operaciones definidas por el *scheduling* con recursos de *hardware* disponibles. Puede impactar al *scheduling* y en general el rendimiento del sistema, por ejemplo para si para las operaciones MAC usamos el DSP48x será más eficiente que emplear LUTs.

Seguiremos el flujo de diseño de la figura 37 , a partir del algoritmo de un algoritmo en C, diseñaremos un testbench sencillo para compararlo con nuestra *Golden Data*, lo migraremos a C++ para así poder aplicar directivas mas sofisticadas que están disponibles en Vivado Hls, en la siguiente sección lo explico con mas detalle.

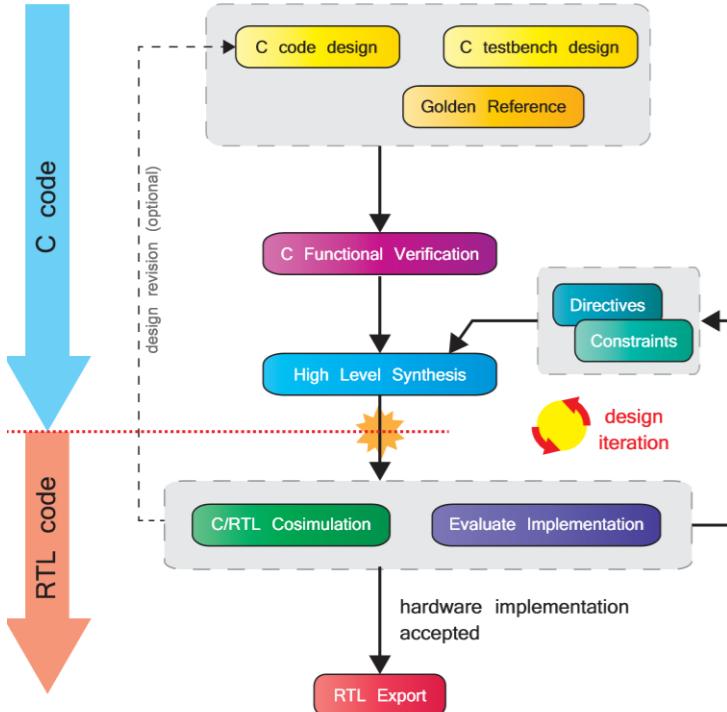


Figura 37: Flujo de diseño del Vivado hls HLS [4].

12. FIR AXIS

Como mencionamos en la sección 11, vamos a partir del código en C (3) de este repositorio [2] que implementa el algoritmo. Crearemos un *Testbench* que leerá del de un fichero que contiene las muestras de la señal δ en cada canal y genera una salida con los coeficientes del filtro en cada canal. Para más detalle, ir al código fuente del proyecto. Después migraremos nuestro algoritmo a C++ y cambiaremos los tipos de datos siguiendo los pasos de la página 97 de la *guía de usuario de Vivado Hls* [11], el resultado esta subido a este repositorio. [7].

```

1 // definition of types
2 typedef int32 coef_t;
3 typedef int32 data_t;
4 typedef int64 acc_t;
5
6 // two samples, one for left channel and the other for right
   channel
7 void fir_filter(data_t datain[2], data_t dataout[2]) {

```

Listing 2: Argumentos Filtro FIR en C.

```

1 #include <ap_cint.h>
2
3 // number of taps
4 #define N 58
5
6 // definition of types
7 typedef int32 coef_t;
8 typedef int32 data_t;
9 typedef int64 acc_t;
10

```

```

11 // two samples, one for left channel and the other for right
12 // channel
13 void fir_filter(data_t datain[2], data_t dataout[2]) {
14     const coef_t c[N+1] = {
15         -378, -73, 27, 170, 298, 352, 302, 168, 14, -80, -64, 53, 186,
16         216, 40,
17         -356, -867, -1283, -1366, -954, -51, 1132, 2227, 2829, 2647,
18         1633, 25, -1712, -3042,
19         29229,
20         -3042, -1712, 25, 1633, 2647, 2829, 2227, 1132, -51, -954,
21         -1366, -1283, -867, -356,
22         40, 216, 186, 53, -64, -80, 14, 168, 302, 352, 298, 170, 27,
23         -73, -378
24 };

```

Listing 3: Filtro FIR C.

```

1 // definition of types
2 typedef ap_int<32> coef_t;
3 typedef ap_int<32> data_t;
4 typedef ap_int<64> acc_t;
5
6
7 struct ap_axis_custom {
8     data_t      data;
9     ap_uint<1>   last;
10    };
11 void fir_filter( ap_axis_custom datain [2], ap_axis_custom dataout
12 [2]);

```

Listing 4: Filtro FIR en C++.

Otro dato a tener en cuenta, es que los valores que nos llegaban de nuestro receptor I^2S (ver sección 4) eran de 16 bits, alojados en la parte menos significativa, por tanto antes de operar con ellos debemos hacer una extensión de signo antes de operar con ellos como se ve en el código 5.

```

1 // first tap
2 if(datain[0].data & 0x8000)
3 {
4
5     datain_left = datain[0].data | 0xFFFF0000;
6
7 }
8 else
9     datain_left = datain[0].data;
10#endif

```

Listing 5: Extensión de signo de los datos de audio.

Synthesis El resultado de nuestra primera aproximación sin utilización de directivas para optimización como se ve en el código 6 , lo podemos observar en las tablas 1, 2 y 3.

```

1 set_directive_interface -mode ap_ctrl_none "fir_filter"
2 set_directive_interface -mode axis -register -register_mode both "
3     fir_filter" datain
4 set_directive_interface -mode axis -register -register_mode both "
5     fir_filter" dataout

```

Listing 6: Directivas Interfaz AXI Stream.

Clock	Objetivo	Estimado	Incertidumbre
ap_clk	10 ns	8.51 ns	1.25 ns

Cuadro 1: El filtro podría funcionar a una frecuencia mayor, 117.5 MHz para ser mas precisos. Lo mantendremos en 100 MHz que es la frecuencia a la que trabaja nuestro sistema.

Latencia (Ciclos)	Latencia (absoluta)	Intervalo (Ciclos)
467	467 ns	467

Cuadro 2: La latencia se refiere al tiempo entre la entrada de una muestra, y la muestra procesada en la salida. Así pues al sistema tarda 4.67 ns en generar un par de nuevas muestras de audio L/R, es decir trabaja a una frecuencia de 2.159 MHz, mucho mayor que los 48 KHz de la frecuencia de muestreo de nuestro modulo, garantizando el funcionamiento.

Nombre	BRAM 18K	DSP48E	FF	LUT
Expression	-	12	0	532
Memory	2	-	16	15
Multiplexer	-	-	-	276
Register	-	-	569	-
TOTAL	2	12	585	823
DISPONIBLE	120	80	35200	17600
UTILIZACIÓN %	1.6	15	1.66	4.67

Cuadro 3: Recursos utilizados por el Filtro.

La otra aproximación, no es necesaria para el objetivo del proyecto, pero incrementa sustancialmente el rendimiento a cambio de un mayor uso de recursos, para ver los resultados de su implementación ir al anexo 9.

13. Diseño del software

Para el software, hemos partido del ejemplo que tiene Digilent en este repositorio [5], que consiste en guardar 5 segundos de audio desde la entrada de micrófono o la línea de entrada del jack y poder reproducirlo en la salida jack de audio de la Zybo. Describiremos secuencialmente las tareas de nuestro función principal citando parte del código a continuación, y mas adelante entraremos en detalles en algunas particularidades a tener en cuenta.

1. **Inicialización y configuración:** Los parámetros de configuración de los dispositivos y controladores del sistema, los encontraremos en dos archivos principalmente *xparameters.h* y *x<nombre_ del_ dispositivo>_g.c..* Sabiendo esto podemos instanciar los controladores, y configurarlos.

```

1 static XIic sIic;
2 static XUartLite sUartLite;
3 static XGpio sUserIOButtons;
4 static XGpio sUserIOSw;
```

```

5 static XLlFifo fifo;
6 static XAxiDma sAxiDma; /* Instance of the XAxiDma */

```

Listing 7: Instanciación de los controladores después de identificarlos en el archivo *xparameters.h*.

- Habilitar interrupciones:** como el modo de interrupción es *Level sensitive*, cuando se genere una interrupción, esta debe permanecer activa, hasta que MicroBlaze guarde el contexto y salte a dicha interrupción mapeada en el vector de interrupciones, deberemos limpiar el vector de la interrupción generada por software antes de acabar de atenderla, para que no salte de nuevo.

```

1 const ivt_t ivt[] = {
2     //IIC
3     {XPAR_INTC_0_IIC_0_VEC_ID, (XInterruptHandler)
4         XIic_InterruptHandler, &sIic},
5     //DMA Stream to MemoryMap Interrupt handler
6     {XPAR_INTC_0_AXIDMA_0_S2MM_INTROUT_VEC_ID, (XInterruptHandler)
7         fnS2MMInterruptHandler, &sAxiDma},
8     //DMA MemoryMap to Stream Interrupt handler
9     {XPAR_INTC_0_AXIDMA_0_MM2S_INTROUT_VEC_ID, (XInterruptHandler)
10        fnMM2SInterruptHandler, &sAxiDma},
11     //User I/O (buttons, switches, LEDs)
12     {XPAR_INTC_0_GPIO_0_VEC_ID, (XInterruptHandler)fnUserIOIsr, &
13         sUserIOButtons},
14     {XPAR_INTC_0_GPIO_1_VEC_ID, (XInterruptHandler)fnUserIOSwIsr, &
15         sUserIOSw},
16     //FIFO
17     {XPAR_INTC_0_LL FIFO_0_VEC_ID, (XInterruptHandler)
18         FifoHandler, &fifo},
19     //UART BM64
20     {XPAR_INTC_0_UARTLITE_1_VEC_ID, (XInterruptHandler)
21         XUartLite_InterruptHandler, &sUartLite}
22 };

```

Listing 8: Vector de interrupciones.

- Esperar inputs:** Esperaremos a que el usuario seleccione una de las opciones disponibles a partir de los botones como se pude ver en la tabla 4

Botón	Función
BTN0	Reproducir audio
BTN1	Grabar audio
BNT2	Cambiar nombre módulo
BNT3	Reiniciar Microblaze

Cuadro 4: Controles.

Además para controlar el encendido/apagado y reinicio del módulo utilizaremos los *switches* como se aprecia en la tabla 5.

Switch	Función
SW0	POWER ON
SW1	RESET

Cuadro 5: Controles BM64.

14. Comunicación con el módulo

Una vez hemos iniciado nuestra aplicación, intentaremos cambiar el nombre del módulo bluetooth cuando está en *pairing mode* para así identificarlo. Para ello debemos asegurarnos que nos comunicamos y obtenemos respuesta, por lo que debemos hacer debug de la interfaz UART que es la que nos sirve para programar el módulo, así pues haciendo uso de un bloque ILA, monitorizaremos los puertos UART que nos comunican con dicho módulo.

En la configuración del bloque ILA tendremos en cuenta los siguientes parámetros:

- **Monitor type:** Para elegir el tipo de monitorización, elegimos interfaz.
- **Sample data width:** El número de muestras que vamos tomar, este dato está directamente relacionado con la frecuencia a la que trabaja nuestro bloque, la cual sera nuestra frecuencia de muestreo, como trabajamos a 100 MHz, después de producirse un disparo, muestreamos cada 10 ns, como seleccionamos 16384 muestras y son dos puertos (UART RX y UART TX) , la duración de nuestra captura será de $(\frac{16384}{2}) * 10^{-9} [ns] = 81,92 [ms]$ suficiente para ver algunos bits teniendo en cuenta que utilizamos un *baud rate* de 115200[bps], por tanto la entrada cambia cada 8,68[μs]. Esto se refleja en las figuras 39 y 40.
- **Interface option:** Eligiremos captura de datos y disparo.

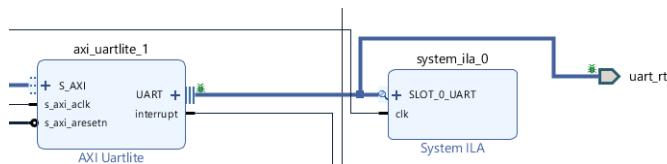


Figura 38: Bloque ILA conectado al UART.

Iniciaremos el módulo en *Flash Test Mode* (como iniciar este modo e información en general de módulo ver el anexo 12, en este modo podemos hacer escrituras parciales de los registros de la *EEPROM* del BM64. Para ello haremos dos funciones que envíen los comandos de escritura y lectura a registros.

CMD Escritura	Longitud	Dirección	Longitud	Longitud del nombre	nombre
01 27 FC	09	05 00	05	04	5A 59 42 4F

Cuadro 6: 5A 59 42 4F es ZYBO en ASCII, será el nombre de nuestro dispositivo, el cual escribimos en el registro de dirección 0550. En campo longitud, se refiere al número de *bytes* siguientes.

```

1 void readEeprom( XUartLite * sUartLite , uint8_t nBytes , uint8_t
address [2] );

```

CMD Lectura	Longitud	Dirección	Número de bytes
01 29 FC	03	05 00	04

Cuadro 7: Leeremos el registro de dirección *0550* para comprobar el valor escrito. En campo longitud, se refiere al número de *bytes* siguientes.

```
2 void writeEeprom( XUartLite * sUartLite ,uint8_t address [2] ,uint8_t
nBytes ,uint8_t* data);
```

Listing 9: Funciones para leer y escribir de la EEPROM.

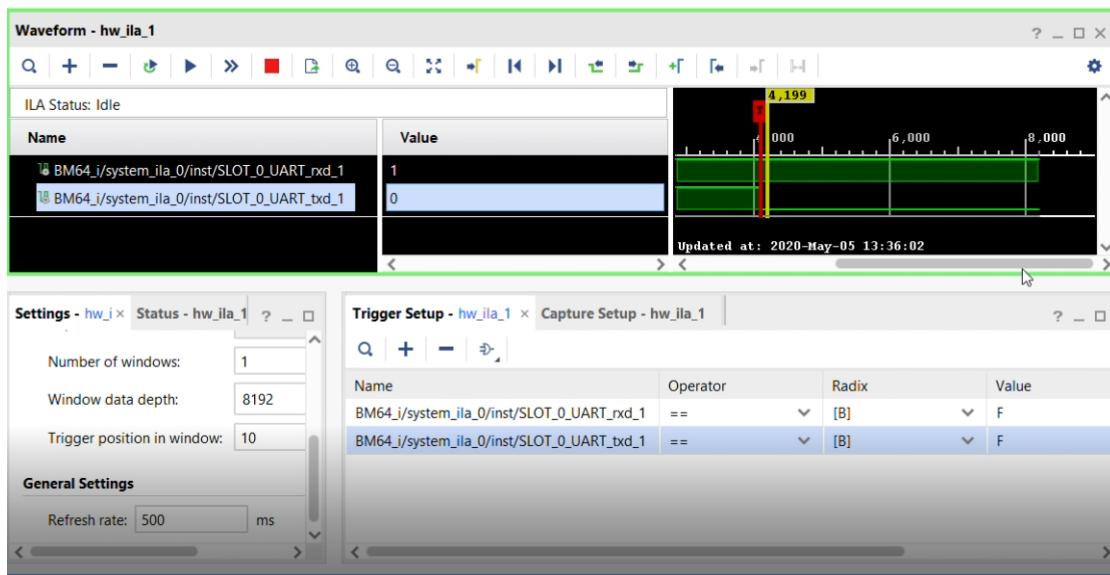


Figura 39: Primer disparo al transmitir hacia el módulo. Establecimos la condición de disparo si sucede un flanco de bajada.

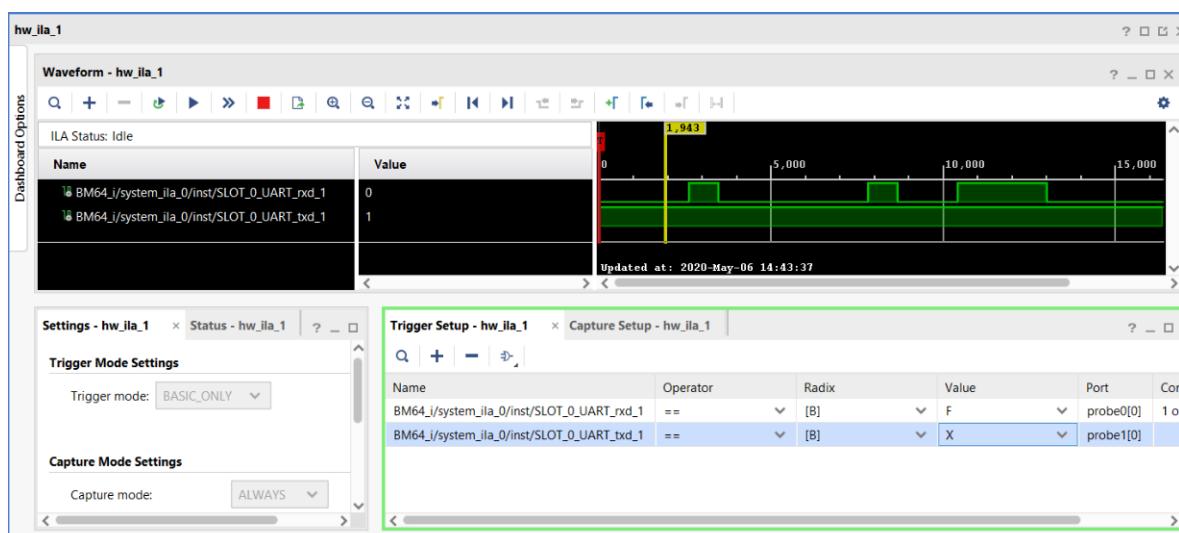


Figura 40: Primer disparo que nos indica respuesta del módulo. La condición de disparo es cuando suceda un flanco de bajada.

15. Procesamiento y almacenamiento de audio

Cuando seleccionamos guardar audio desde el botón 1 de la Zybo, lanzamos la función *fnAudioRecord* que funciona de la siguiente manera:

1. Ponemos nuestro DMA a la escucha del canal *Streaming* y le damos un valor igual o superior a la cantidad de bytes que va a recibir en el canal de *Streaming*.
2. Reiniciamos nuestra FIFO, y con ello también reiniciaremos nuestro filtro y el Core *I2S* (ver figura 41), así empezaremos a procesar con todos los cores sincronizados.
3. Habilitaremos las interrupciones de la FIFO, esta se generara cada vez que el filtro FIR haya generado un par de muestras de audio (una para el canal izquierdo y otra para el derecho) y las enviaremos al DMA.
4. Esperaremos a que la FIFO haya acabado, eso significara que los datos han sido procesados y almacenados (Se generara otra interrupción procedente del DMA informando si fue un éxito o hubo algún problema).

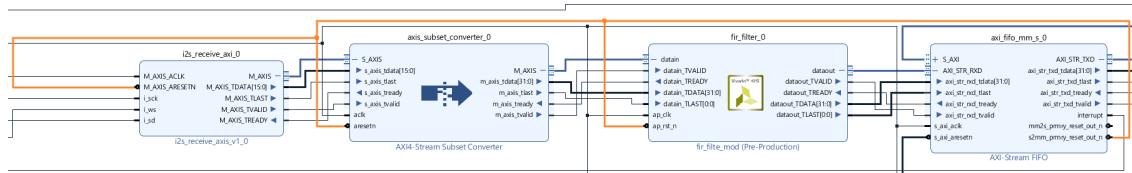


Figura 41: Desde software controlamos el reset de los cores que procesan el audio.

```

1 void fnAudioRecord(XAxiDma AxiDma, XLlFifo fifo, u32 u32NrSamples)
2 {
3     int Status;
4     xil_printf("\r\nEnter Record function");
5     fifo_done=0;
6     count_bytes=0;
7     words_number=0;
8     Status= XAxiDma_SimpleTransfer(&AxiDma,(u32) MEM_BASE_ADDR ,6*
9         u32NrSamples, XAXIDMA_DEVICE_TO_DMA);
10    if(Status==XST_INVALID_PARAM)
11        xil_printf("\nFail fnAudioRecord invalid param");
12    if (Status != XST_SUCCESS)
13        xil_printf("\n fail @ rec; ERROR: %d", Status);
14    XLlFifo_Reset(&fifo);
15    // Enable the interrupt for the FIFO
16    XLlFifo_IntEnable(&fifo,XLLF_INT_ALL_MASK);
17    //Wait until fifo stream finish
18    while(!fifo_done);
}
```

Listing 10: Almacenamiento.

```

1 void FifoStreamHandler(XLlFifo *InstancePtr)
2 {
3     int i;
4     u32 RxWord;
5     static u32 ReceiveLength;
6     int j;
```

```

7   int txVancanccy;
8   count=0;
9   // checking the occupancy
10  while(XLlFifo_iRxOccupancy(InstancePtr)>2)
11  {
12      //Read RLR to find the number of bytes
13      ReceiveLength = (XLlFifo_iRxGetLen(InstancePtr))/WORD_SIZE;
14      for (i=0; i < ReceiveLength; i++)
15      {
16
17          RxWord = XLlFifo_RxGetWord(InstancePtr);
18          *(samples_In+i) = RxWord;
19          words_number+=4;
20
21          count+=1;
22      }
23      //check TDFV for the FIFO occupancy before writing to the TX
24      // FIFO
25      txVancanccy=Xil_In32((InstancePtr->BaseAddress) +
26 XLLF_TDFV_OFFSET);
27      // to set transmit destination address
28      Xil_Out32(((InstancePtr->BaseAddress) + XLLF_TDR_OFFSET), 0
29      x00000002);
30      if(txVancanccy)
31      {
32          for (j=0; j < ReceiveLength ; j++)
33          {
34              // to write data in the transmit data FIFO data
35              // write port, eight words
36              Xil_Out32(((InstancePtr->BaseAddress) +
37 XLLF_TDFD_OFFSET), *(samples_In+j));
38
39      }
40      if(words_number > (NR_AUDIO_SAMPLES*5))
41      {
42          // Start Transmission by writing transmission length
43          // into the TLR
44          XLlFifo_iTxSetLen(InstancePtr,words_number);
45          fifo_done=1;
46          // Disable the interrupt for the FIFO
47          XLlFifo_IntDisable(InstancePtr,XLLF_INT_ALL_MASK);
48          return;
49      }
50  }

```

Listing 11: Handler principal de la interrupción generada por la FIFO.

16. Reproducción de audio

Para reproducir los datos (botón 0 de la Zybo) invocaremos dos funciones, primero *fnSetHpoOutput* para establecer como salida de audio el jack de nuestra Zybo y la función *fnAudioPlay*, el procedimiento es el siguiente:

1. Preparamos el buffer del controlador I^2S de Digilent para recibir datos.
2. Leemos 5 segundos de audio de la memoria DDR a través del DMA.
3. habilitamos el canal *Streaming* del controlador I^2S para que escuche del canal AXI Stream conectado al DMA y lo envíe por I^2S al códec de audio.

Listing 12: Reproducción del audio.

Completado el último paso, procederemos a validar nuestro proyecto en la siguiente sección.

17. Validación

Para probar nuestra aplicación, vamos a partir de un archivo de audio de calidad CD en formato MP3 con un bit rate de 192 kbps y una frecuencia de muestro de 48 kHz. En Matlab crearemos una función seno a 4 kHz, de una amplitud de 0.2, y de una duración igual a nuestra canción de prueba. Sumaremos a la canción el seno, así apreciaremos la diferencia al filtrarlo.

```
1 [y,Fs] = audioread("TobuCandyland.mp3");
2 n = length(y);
3 % interval
4 Fs = 48000; % Sampling frequency (samples per second)
5 dt = 1/Fs; % seconds per sample
6 StopTime = 120; % seconds
7 t = (0:dt:StopTime)'; % seconds
8 F = 4000; % Sine wave frequency (hertz)
9 A=0.2; % Sine amplitude
10 sine = A*sin(2*pi*F*t);
11 %%Fourier Transform:
12 SINE = fftshift(fft(sine));
13 %%For one cycle get time period
14 T = 1/F ;
15 % time step for one time period
16 tt = 0:dt:T+dt ;
17 d = sin(2*pi*F*tt) ;
18 %%Add the 4khz sine wave
19 samples=2500000;%start in 5000 samples music
20 y=[sine+y(samples:length(sine)+(samples-1),1) sine+y(samples:
length(sine)+(samples-1),2)];
```

Listing 13: Añadiendo tono a una canción de calidad CD.

 TobuCandyland	3/15/2020 9:24 AM	MP3 File	4,645 KB
 Candyland_plus_tone_4khz	5/26/2020 9:17 AM	WAV File	22,501 KB

Figura 42: Archivo generado que contiene dos minutos de la canción más el tono. Al reproducirlo se escucha un pitido durante toda la canción.

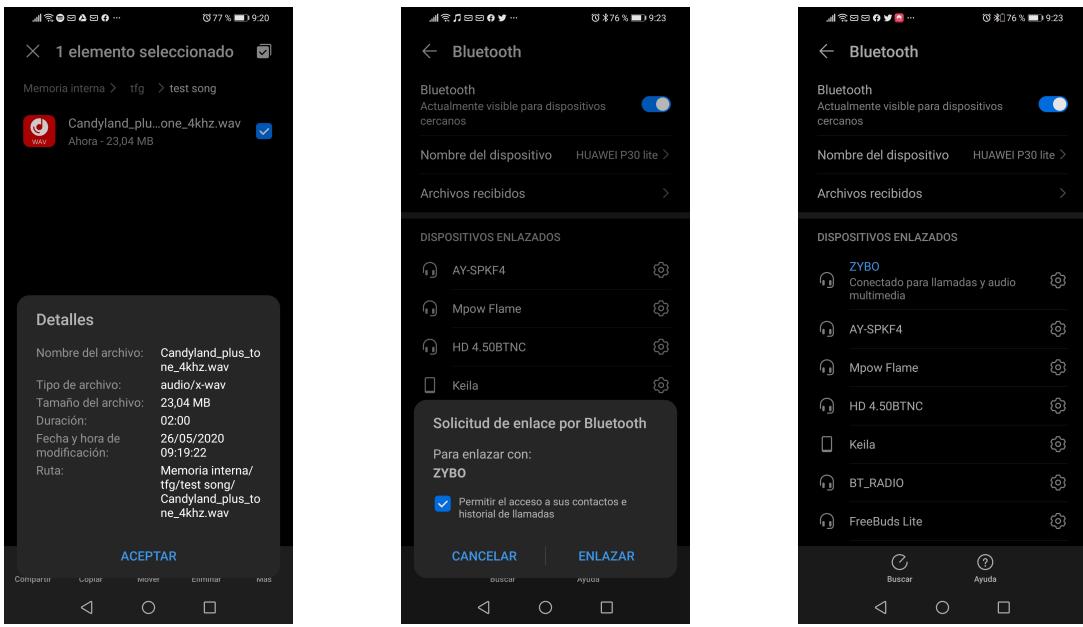


Figura 43: Enlazando con la Zybo por bluetooth.

Cuando hemos enlazado con el módulo abriremos el reproductor de nuestro móvil y reproduciremos la canción de prueba. Posteriormente con el botón de grabar de la Zybo guardaremos un intervalo de 5 segundo elegido de manera subjetiva, y realizaremos *play-back*, apreciando la diferencia, en mi caso particular, realice tres pruebas y mi percepción subjetiva se puede ver en la tabla 8.

Amplitud del tono	Percepción
1	Buena calidad de audio, pero aun se percibe ligeramente el tono
0,5	No se percibe el tono, calidad de audio buena
0,2	No se percibe el tono pero aparece un ruido de fondo. mala calidad

Cuadro 8: Percepción.

El proyecto final lo podemos encontrar en este repositorio. [6].

Capítulo VI

Conclusión.

1. Resumen

Los proyectos llevado a cabo en este trabajo, no habrían sido posible utilizando la Zybo sin ningún añadido, demostrando la necesidad de construir una placa con los componentes que se adecuaban a nuestras necesidades, o la alternativa, generalmente mas cara y con menos posibilidad de configuración, de comprar los módulos que se conecten directamente a la Zybo en el mercado. Además, hemos utilizados herramientas que aceleran el tiempo de desarrollo como Xilinx Vivado IP Integrator y Vitis, sin olvidar Vivado-HLS, al que proporcionándole el algoritmo del filtro FIR en lenguaje de alto nivel C++, nos implemento arquitectura hardware que cumplía con nuestras especificaciones, y por último vimos la utilidad de Simulink para hacer simulaciones sobre aplicaciones de procesado de señal. Sugerencias sobre mejoras, la veremos en la próxima sección.

2. Trabajo Futuro

El proyecto avanzado, no ejecuta ningún sistema operativo , es un buen punto para desarrollar una aplicación, pues tenemos control total del procesador. Sin embargo, al intentar sistemas a tiempo real mas complejos la dificultad se incrementa exponencialmente, (gestión de interrupciones y su sincronización...). Por ello una de las opciones que sugerimos es añadir un Real-Time Operating System (RTOS), un software que permite que la ejecución de múltiples tareas sea simultanea (multihilo), permitiendo también compartir recursos entre las tareas y establecer prioridades. Con su utilización en el proyecto podríamos por ejemplo actualizar los displays con la temperatura del sensor del módulo bluetooth cada 5 minutos, dar una mayor prioridad a los botones, para implementar un botón de stop para detener el audio cuando estamos reproduciendo, entre otros añadidos. Entre las opciones de RTOS, se encuentran FreeRtos, muy común para sistemas embebidos, ThreaddX, y Micrium uc/OS por mencionar algunas. Por ultimo, recomendamos echar un vistazo a las demos de ejemplo que podríamos aplicar para nuestro MicroBlaze en nuestra Zybo accediendo a FreeRtos.org.

Bibliografía

- [1] How a microblaze can peaceably coexist with the zynq soc. *Xcell Journal issue 82*, 2013. (Ver).
- [2] Fir. <https://github.com/xupgit/High-Level-Synthesis-Flow-on-Zynq-using-Vivado-HLS>, 2018.
- [3] Analog Devices. *Low Power Audio Codec Data Sheet SSM2603*, 7/2018—rev. c to rev. d edition, March 2018. (Ver).
- [4] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc.* Strathclyde Academic Media, Glasgow, GBR, 2014.
- [5] Digilent. Zybo dma. <https://github.com/Digilent/Zybo-DMA.git>, 2017.
- [6] Umberto Vargas. Bm64 demo. <https://github.com/umbertoVargas/BM64-FIR-DMA>, 2020.
- [7] Umberto Vargas. Fir axis. <https://github.com/umbertoVargas/FIR-HLS>, 2020.
- [8] XILINX. *AXI ReferenceGuide*, ug761 (v13.1) edition, March 2011. (Ver).
- [9] XILINX. *Zynq-7000 SoC Technical Reference Manual*, ug585 (v1.12.2) edition, July 2018. (Ver).
- [10] XILINX. *LogiCORE IP AXI DMA v7.1*, pg021 edition, June 2019. (Ver).
- [11] XILINX. *Vivado Design Suite UserGuide High-Level Synthesis*, ug902 edition, June 2020. (Ver).

Anexos

1. Recursos PS

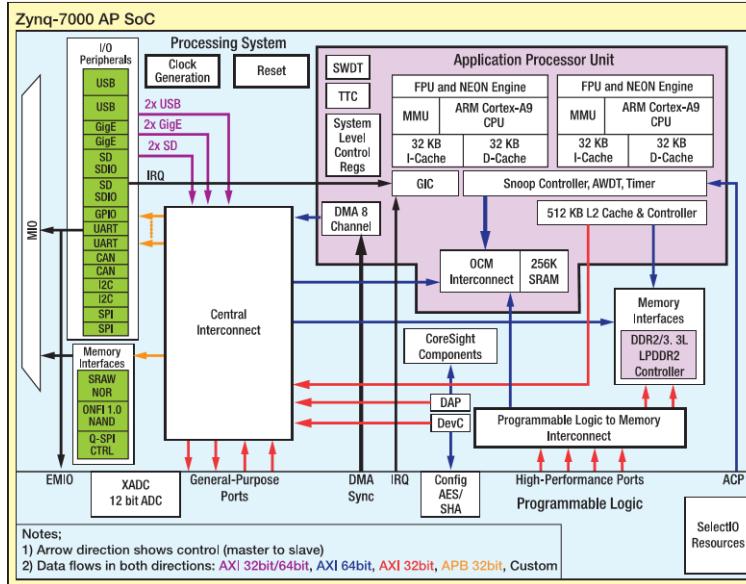


Figura 44: ZYNQ Pocessing System. Recursos privados y compartidos. [4]

■ Application Processing Unit (APU):

- Dos procesadores ARM.
- Cache L1 y L2 - una Cache L1 individual para cada procesador, la cache L2 es compartida.
- ON Chip Memory (OCP) - Memoria principal de los procesadores.
- Snoop Control Unit (SCU) - Hace de puente entre los procesadores y la cache L2 y la OCP, mantiene la consistencia de los datos en la Cache L2.
- DDR SDRAM Controller - Interfaz para acceder a la Double Data Rate Synchronous Dynamic Random-Access Memory (DDR SDRAM) de la Zybo.

■ Interfaces externas:

- Multiplexed Input/Output (MIO) - Permite la conexión de la PS con los periféricos, son un total de 53 pines y el mapeo con los periféricos es flexible, es decir como tenemos un número limitado de pins, podemos configurar que periféricos nos interesan mapear a los MIO.
- Extended Multiplexed Input/Output (EMIO) - Permite compartir los periféricos de la PS con la PL, por ejemplo *Ethernet* o *UART*, en el caso del primero por ejemplo funcionara a mitad de la frecuencia si lo utilizamos en la PL a través de EMIO. No todos los periféricos son compatibles, como observamos en la tabla 9.

Para mas información puedes consultar el capitulo 1.2 de el manual técnico de Xilinx [9].

Periférico	MIO	EMIO
SPI (x2)	disponible	no disponible
I^2C	disponible	disponible
CAN (x2)	disponible	disponible
UART (x2)	disponible	disponible
GPIO	disponible	disponible
SD (x2)	disponible	disponible
USB (x2)	disponible	no disponible
GigE (x2)	disponible	disponible

Cuadro 9: Periféricos de la PS

2. Recursos PL

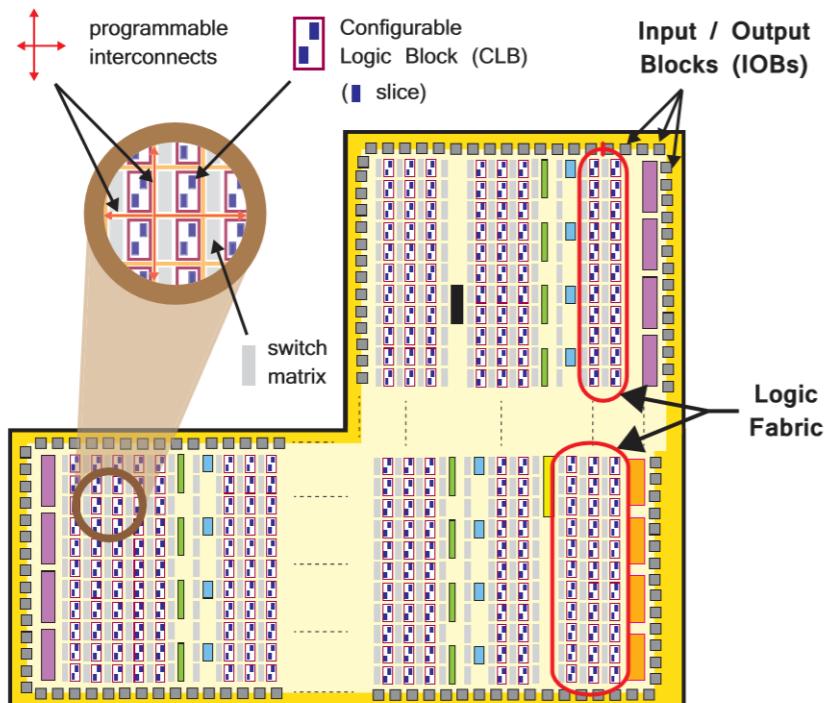


Figura 45: Recursos de la PL [4]

Elementos de la PL:

- **Configurable Logic Block (CLB)** - una celda en la que se agrupan diversos elementos lógicos, como LUT, FF y multiplexores, que permiten realizar operaciones booleanas, almacenamiento de datos y operaciones aritméticas.
- **Look Up Table (LUT)** - recurso flexible que nos permite implementar funciones lógicas, memorias o registros de desplazamiento.
- **Flip Flop (FF)** - circuito secuencial que implementa 1 bit de registro con capacidad de *reset*.

- **Switch Matrix** - elemento próximo a cada CLB y facilita el enrutado en la PL.
- **Input/Output Blocks (IOBs)** - es la interfaz entre los recursos de la PL y los elementos físicos como *pads*, *pmod*, usados para conectar elementos externos.
- **Recursos especiales**

- **DSP48E1** - componente específico para implementar operaciones aritméticas.
- **Block Ram** - componente específico para operaciones que requieren cierta cantidad de memoria, First in - First out (FIFO), Random Access Memory (RAM), Read-Only Memory (ROM)...

Una combinación de los recursos especiales listados presenta una gran mejora en las Multiply–Accumulate Operation (MAC).

Para mas información puedes consultar el capitulo 1.3 de el manual técnico de Xilinx [9].

3. Microblaze

Entre las configuraciones disponibles seleccionamos optimización orientada al rendimiento '*Performance*', una implementación de 32 bits, habilitamos la interfaz de *debug*, habilitamos memoria de instrucciones y datos de 128 kB, las interrupciones, trabajaremos con memoria física, por tanto no utilizaremos Memory Management Unit (MMU) ni cache, y la frecuencia con la que trabajaremos será 100 MHz, como vemos es una implementación sencilla, por lo que no tendrá gran impacto en lo que al área o recursos utilizados se refiere.

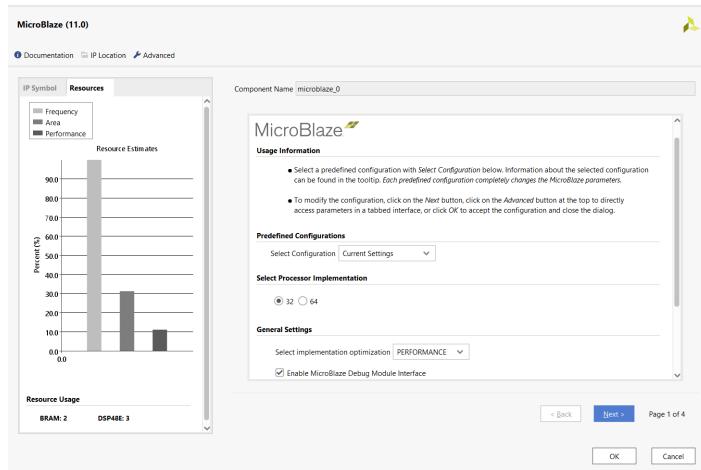


Figura 46: Resumen Microblaze.

Por ultimo debemos realizar una configuración extra, al establecer los apartados anteriores y correr el proceso automático de Vivado este instanciará el AXI Interrupt Controller usado por el MicroBlaze para gestionar las interrupciones generadas por los demás dispositivos, lo que haremos es aumentar el numero de interrupciones a 8 y el tipo de interrupciones será *Level interrupt*.

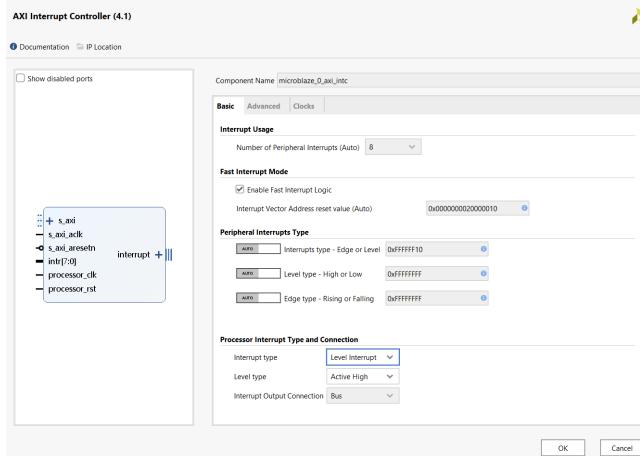


Figura 47: Configuración del control de interrupciones del Microblaze.

4. I^2S Receiver

Partiendo de las hojas se especificaciones del bus I^2S implementaremos *from scratch* (desde cero) en Verilog, le añadiremos la interfaz AXI STREAM, y empaquetaremos nuestra *Custom IP* para instanciarla en nuestro diseño. Para la sincronización del reloj del módulo bluetooth hemos usado dos FF de sincronización y un detector de flanco.

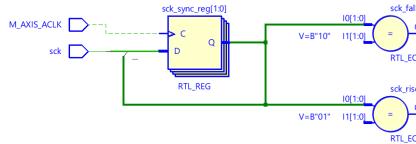


Figura 48: Sincronización del reloj del módulo bluetooth I^2S

Los datos del módulo son de 16 bits y el tamaño mínimo del puerto de datos del AXI Stream es de 32 bits, lo que haremos es alojar esos bits en la parte menos significativa (por tanto habrá que hacer una posterior extensión de signo), para esto utilizaremos el **AXI Subset converter**. y por ultimo y muy *importante* el valor del tamaño de paquetes no es trivial, es de 2, por que un paquete de audio de doble canal serán dos paquetes uno para el auricular izquierdo y otro para el derecho, este genera **TLAST** que es imprescindible para generar la interrupción en la AXI FIFO y procesar los paquetes, dicho esto, su configuración sería la que vemos en la figura 49.

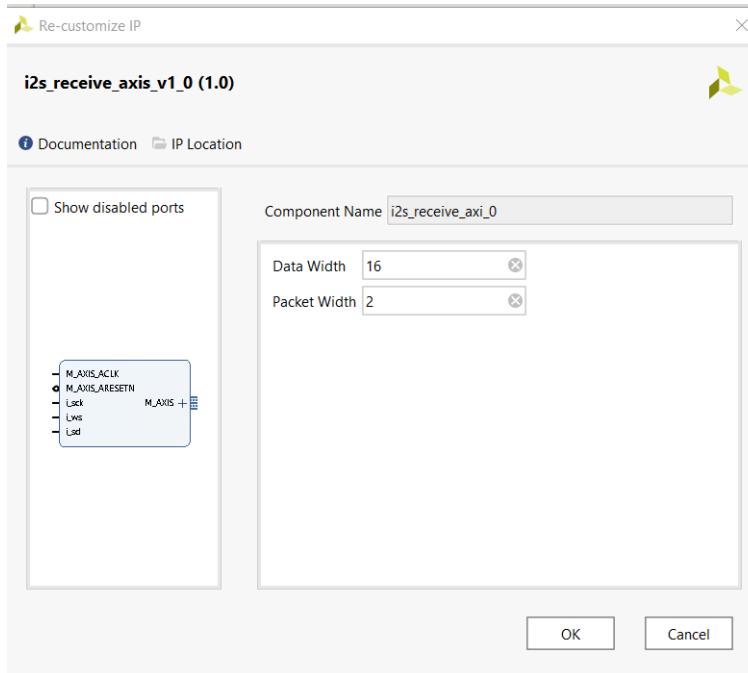


Figura 49: Configuración I2S RX.

5. DMA

Instanciaremos el DMA, y haciendo uso del manual técnico de Xilinx [9], nos vamos a la página 112 (ver figura 50), y anotamos la dirección por la que podemos acceder desde la PL a la DDR SDRAM.

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters ⁽¹⁾	Notes
0000_0000 to 0003_FFFF ⁽²⁾	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU ⁽³⁾
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to FB00_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FBFF_FFFF ⁽⁴⁾	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF ⁽²⁾	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Figura 50: Mapeo de direcciones de la PS. [10]

Nos vamos a la configuración del Core Zynq Processor, el cual se debe instanciar siempre, aunque no vayamos a utilizar el hard-processor, pues son necesarios unos *presets* mínimos para que luego funcione en la herramienta de desarrollo de software. Allí habilitamos el puerto puerto HP que vamos a utilizar.

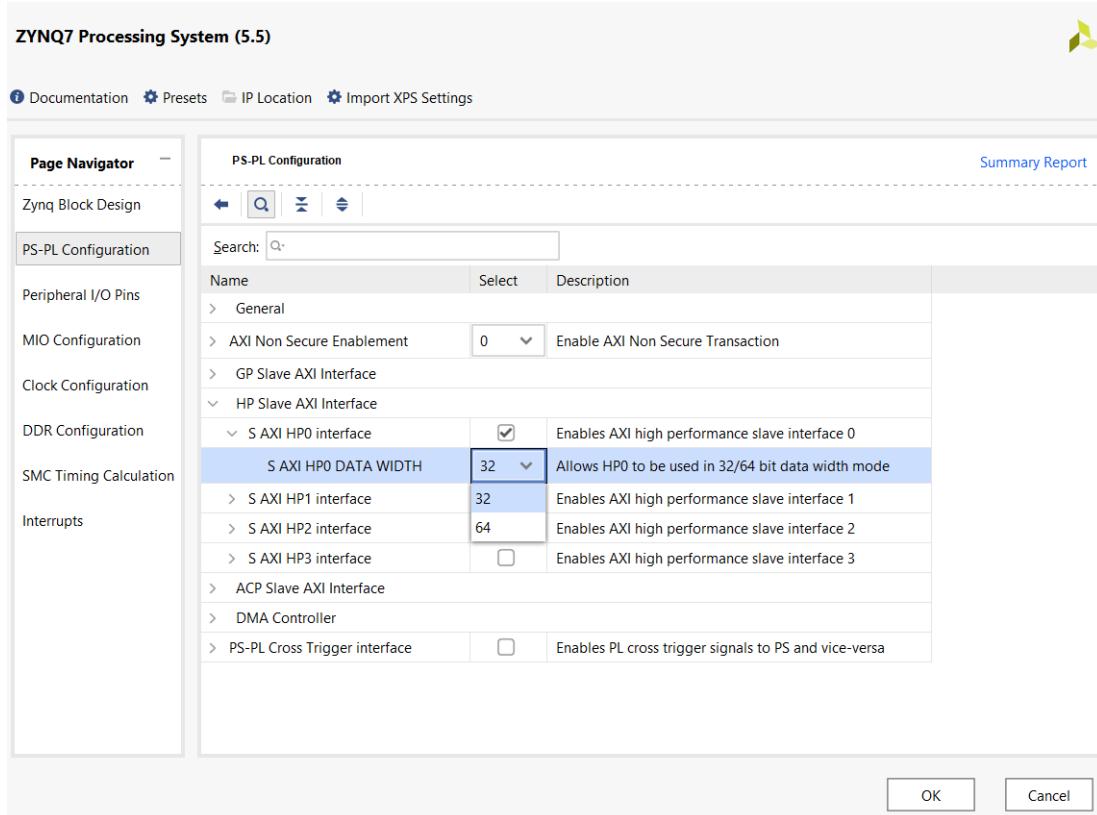


Figura 51: Elegimos la configuración de 32 bits pues son el tamaño de los paquetes de audio que vamos a guardar.

Además modificaremos las direcciones físicas donde se ha mapeado la memoria de instrucciones y datos de la MicroBlaze (por defecto es 0x00000000) la cambiamos a 0x20000000 como se puede observar en la figura 52.

microblaze_0						
Data (32 address bits : 4G)						
axi_dma_0	S_AXI_LITE	Reg	0x41E0_0000	64K	<input type="checkbox"/>	0x41E0_FFFF
axi_fifo_mm_s_0	S_AXI	Mem0	0x44A0_0000	64K	<input type="checkbox"/>	0x44A0_FFFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	<input type="checkbox"/>	0x4000_FFFF
axi_gpio_1	S_AXI	Reg	0x4001_0000	64K	<input type="checkbox"/>	0x4001_FFFF
axi_iic_0	S_AXI	Reg	0x4080_0000	64K	<input type="checkbox"/>	0x4080_FFFF
axi_timer_0	S_AXI	Reg	0x41C0_0000	64K	<input type="checkbox"/>	0x41C0_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	<input type="checkbox"/>	0x4060_FFFF
axi_uartlite_1	S_AXI	Reg	0x4061_0000	64K	<input type="checkbox"/>	0x4061_FFFF
d_axi_i2s_audio_0	AXI_L	AXI_L_reg	0x44A1_0000	64K	<input type="checkbox"/>	0x44A1_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x2000_0000	128K	<input type="checkbox"/>	0x2001_FFFF
microblaze_0_axi_intc	S_AXI	Reg	0x4120_0000	64K	<input type="checkbox"/>	0x4120_FFFF
Instruction (32 address bits : 4G)						
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x2000_0000	128K	<input type="checkbox"/>	0x2001_FFFF
axi_dma_0						
Data_MM2S (32 address bits : 4G)						
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	<input type="checkbox"/>	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)						
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	<input type="checkbox"/>	0x1FFF_FFFF

Figura 52: Modificado el rango de direcciones de la memoria de instrucciones y datos del MicroBlaze para no solaparse con la dirección del puerto HP que da acceso a la DDR SDRAM

Ahora toca definir los parámetros de configuración para ello seguiremos una serie de

pasos:

1. Habilitaremos el canal de escritura, con esto conseguimos pasar nuestros datos de Axi Stream a Memory Mapped Protocol y almacenarlo en la DDR.
2. Habilitaremos el canal de lectura, con esto conseguimos leer de la DDR SDRAM logrando el proceso inverso del apartado anterior, para enviar los datos al Core AXI I2S.
3. Tamaño de direcciones de 32 bits como establecimos en la figura 51
4. Para la longitud del *buffer* del DMA, seguimos la recomendación para modo multi-canal de la página 78 **hoja de datos del DMA** [10].
5. Marcaremos *unaligned transfers* para poder empezar a escribir en una dirección que no este alineada a direccionamiento de 32 bits.

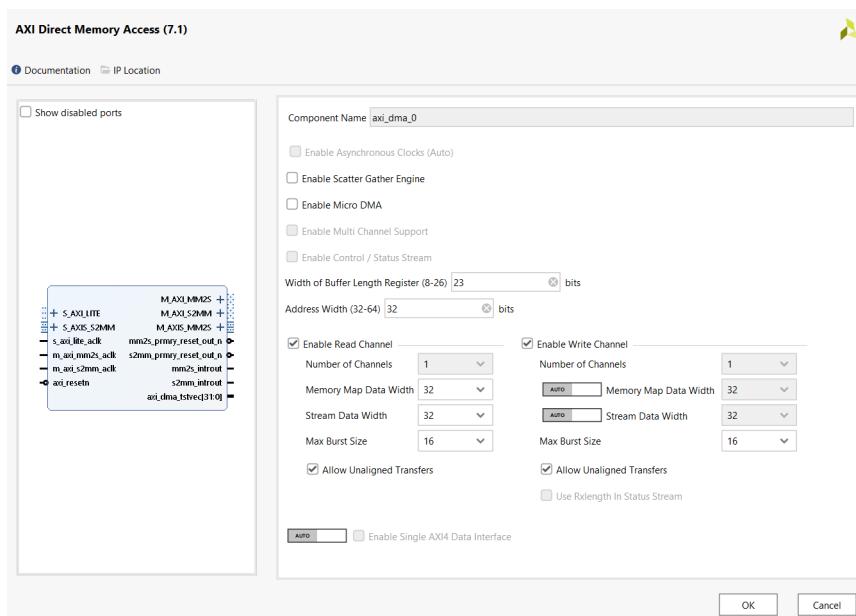


Figura 53: Parámetros de configuración del AXI DMA.

6. I^2S Transmitter

Para utilizar la IP de digilent que encontramos en este **repositorio** realizaremos las siguientes modificaciones:

- Actualizar la IP, desde Vivado, así podremos modificar los parámetros de las FIFOs que implementa.
- Cambiar el tamaño de datos de las FIFO a 16 bits (figura 54, pues esta es la resolución máxima del módulo bluetooth).
- Cambiar en la GUI de la IP el tamaño de los datos I^2S como en la figura 55.

Desarrollo de una placa suplementaria para Digilent Zybo

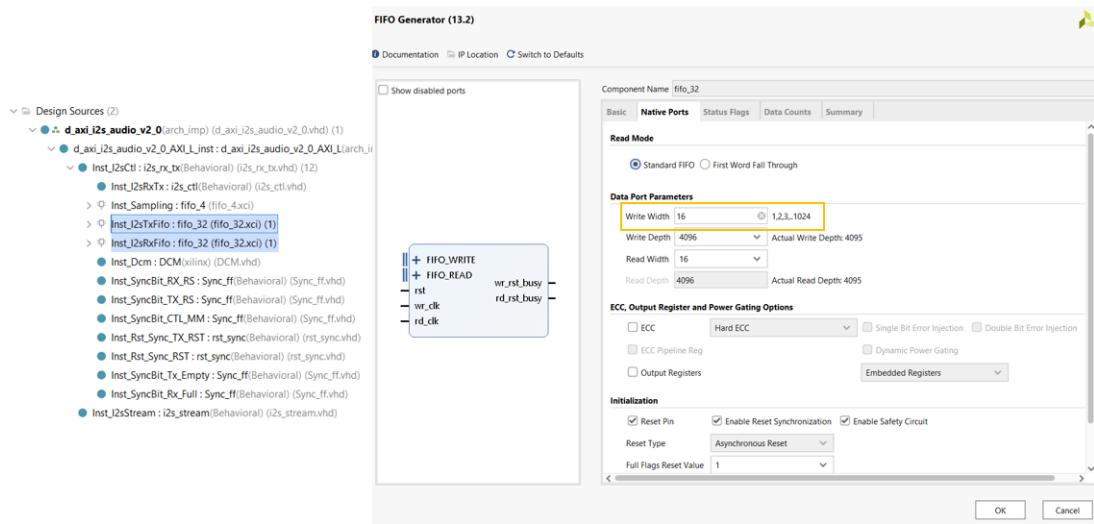


Figura 54: Modificación de las FIFOs que emplea el core.

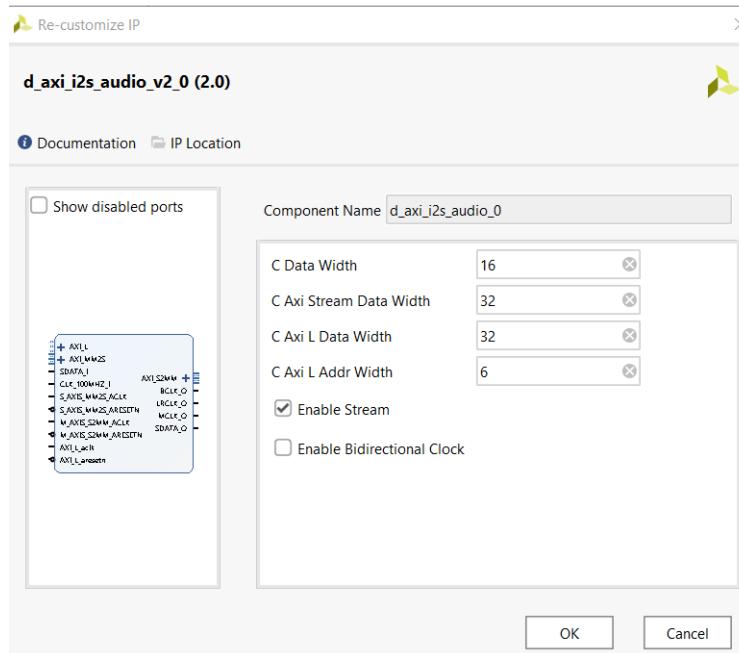
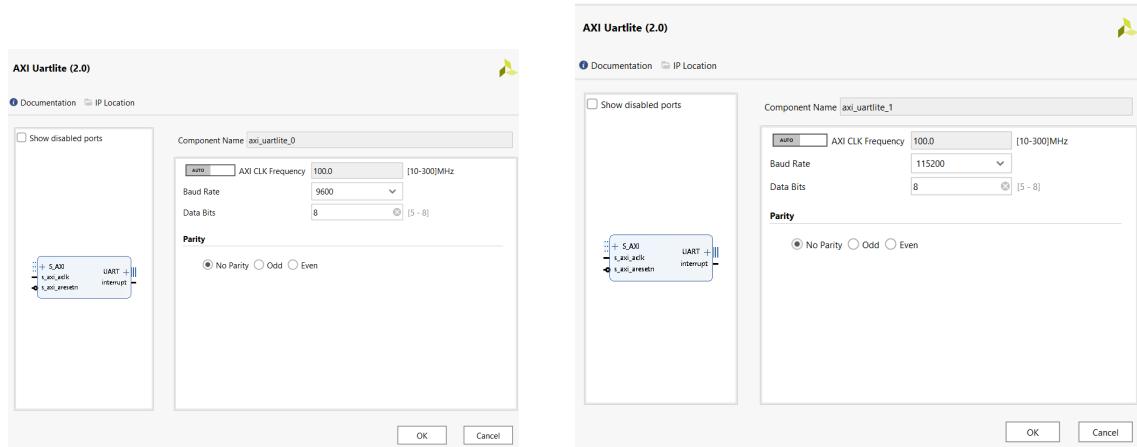


Figura 55: Modificación del tamaño de datos en la GUI de Vivado-

7. UARTLITE



(a) Configuraciones UART para *debug*, *baud rate* de 9600 bps.

(b) Configuraciones UART para comunicarse con el módulo, *baud rate* de 115200 bps.

Figura 56: Configuración UART

8. I^2C

Para configurar correctamente nuestro core I^2C debemos dirigirnos a la página 17 de la **hoja de datos de SSM2603** de ahí configuraremos como se puede apreciar en la figura 57.

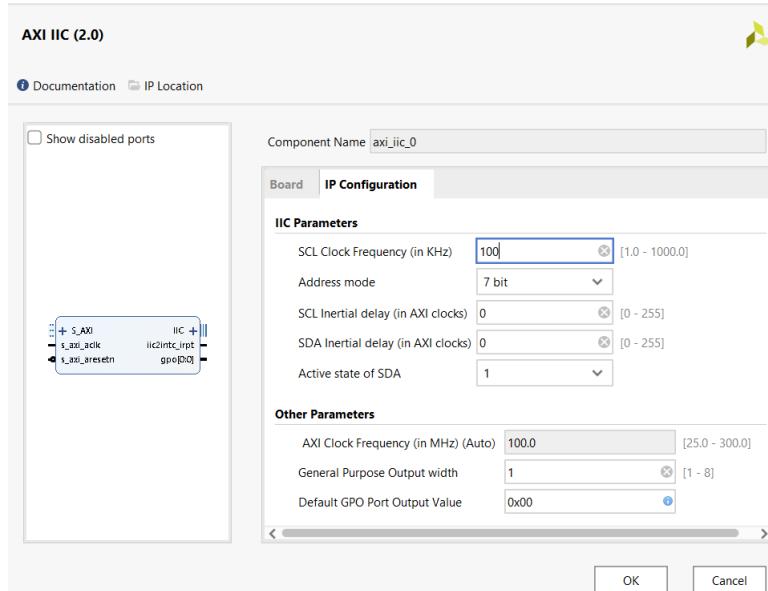


Figura 57: Configuración I^2C del SSM2603.

9. HLS Filtro FIR optimizado

10. Testbench proyecto básico

```
1 module cntrl_display_tb;
2 parameter word_size = 16;//bits
3 integer i;
4 //clock zybo
5 reg clk = 1;
6 always #4 clk = ~clk;
7
8 //wire and regs
9 reg [5:0] addr;
10 reg [word_size-1:0] di;
11 reg we = 0;
12 wire [3:0] demux;
13 wire [word_size-1:0] o_data;
14 wire [31:0] model_msseconds;
15 reg oe = 0;
16
17 cntrl_display#(.ms_limit(1000000)) dut (
18 .clk(clk),
19 .i_addr(addr),
20 .i_di(di),
21 .i_we(we),
22 .i_oe(oe),
23 .o_demux(demux),
24 .o_data(o_data)
25 );
26 model
27     #(.ms_limit(1000000))
28     model
29         (
30             .clk(clk),
31             .en(oe),
32             .miliseconds(model_msseconds)
33         );
34     always @ (demux)
35         begin
36             $display("INFO: demux = %d, miliseconds = %d, and o_data = %x
37             at %d ms",demux, model_msseconds,o_data, $time/1000000);
38         end
39 initial
40 begin
41     repeat (10) @(posedge clk);
42     //fill memory with data
43     for (i = 0; i < 65; i = i+1)
44         begin
45             @(posedge clk);
46             addr = i;
47             di = $urandom;
48             we = 1;
49         end
50         // Wait on signal
51         @(posedge clk);
```

```

52     // Now start counting
53     we = 0;
54     addr = 0;
55     //Enabling output data
56     oe = 1;
57     //simulate 100 ms ...
58     repeat (100000000) @(posedge clk);
59     $display("Simulation complete at time %0fns.",$realtime);
60     $finish;
61 end
62 endmodule

```

Listing 14: TestBench.

```

1 // 125 MHz - 1 Cycle/S - 8ns
2 // 125 Hz - 1e6 Cycle/S - 8ms
3 module model
4   #(
5     parameter ms_limit = 1000000
6   )
7   (
8     input clk,
9     input en,
10    output reg [31:0] miliseconds
11  );
12  integer counter = 0;
13  always @(posedge clk)
14    if(en)
15      counter <= counter + 1;
16
17  always @(posedge clk)
18    miliseconds <= counter / (ms_limit);
19 endmodule

```

Listing 15: Model.

```

1 module rams_dist (clk, we, a, dpra, di, spo, dpo);
2
3
4 input clk;
5 input we;
6 input [5:0] a;
7 input [5:0] dpra;
8 input [15:0] di;
9 output [15:0] spo;
10 output [15:0] dpo;
11 reg [15:0] ram [63:0];
12
13
14 always @(posedge clk)
15 begin
16   if (we)
17     ram[a] <= di;
18 end
19
20 assign spo = ram[a];
21 assign dpo = ram[dpra];
22

```

```
23 endmodule
```

Listing 16: Memoria de doble puerto.

11. Fir Axi Stream extra

Una segunda aproximación de la implementación del filtro del proyecto avanzado sería haciendo uso de las directivas listadas en la guía de optimización de Vivado hls (página 12) en nuestro caso tenemos dos loops 17 y 18, añadiremos pipeline a ambos loops para que las muestras pueden empezar a ser procesadas antes de haber acabado con las que están procesándose.

```
1 loop_left: for (i=N-1; i!=0; i--) {
2     acc_left += (acc_t) shift_reg_left[i-1] * (acc_t) c[i];
3     shift_reg_left[i] = shift_reg_left[i-1];
4 }
```

Listing 17: Loop canal izquierdo.

```
1 loop_right: for (i=N-1; i!=0; i--) {
2     acc_right += (acc_t) shift_reg_right[i-1] * (acc_t) c[i];
3     shift_reg_right[i] = shift_reg_right[i-1];
4 }
```

Listing 18: Loop canal derecho.

```
1 set_directive_interface -mode ap_ctrl_none "fir_filter"
2 set_directive_interface -mode axis -register -register_mode both "
    fir_filter" datain
3 set_directive_interface -mode axis -register -register_mode both "
    fir_filter" dataout
4 set_directive_pipeline "fir_filter/loop_left"
5 set_directive_pipeline "fir_filter/loop_right"
```

Listing 19: Directivas optimización.

Latencia (Ciclos)	Latencia (absoluta)	Intervalo (Ciclos)
131	131 ns	131

Cuadro 10: Como era de esperar disminuimos la latencia, al aplicar pipeline a los loops.

Nombre	BRAM 18K	DSP48E	FF	LUT
Expression	-	12	0	536
Memory	2	-	16	15
Multiplexer	-	-	-	305
Register	0	-	703	64
TOTAL	2	12	719	920
DISPONIBLE	120	80	35200	17600
UTILIZACIÓN %	1.6	15	2	5.2

Cuadro 11: Recursos utilizados por el Filtro después de aplicar loops. Como siempre vemos que habrá un *tradeoff* entre recursos utilizados (área) y rendimiento.

12. Microchip Bluetooth Module (BM64)

Nuestra aplicación de procesamiento de audio requiere una fuente donde obtener dichos datos, elegimos como medio la tecnología bluetooth pues la mayoría de los dispositivos *wireless* del mercado, especialmente los *Smartphones*, son compatibles con dicha tecnología, y a la hora elegir el módulo bluetooth necesitábamos que cumpliera dos premisas, que sea configurable desde nuestra aplicación sin necesidad de elementos externos y la interfaz de audio digital sea de un estándar conocido. El módulo BM64 de Microchip aúna todas esas características gracias a su interfaz UART , y su salida de audio digital I^2S .

Las *features* principales de la que vamos hacer uso son:

- **Perfil Bluetooth:** En nuestro caso trabajaremos con el perfil Advanced Audio Distribution Profile (A2DP), un estándar muy extendido y compatible con la mayoría de dispositivos bluetooth del mercado y que permite la transmisión de audio digital estéreo (dos canales). El BM64 cuenta con dos sistemas de compresión de audio low-complexity subband codec (SBC) y Advanced Audio Coding (AAC), este último es más utilizado en dispositivos Apple, pero para nuestros objetivos y debido de nuevo a cuestiones de compatibilidad hemos optado por dejar SBC que nos proporciona una frecuencia de muestreo de hasta 48 kHz y 16 bits de resolución, suficiente para nuestra aplicación.
- **Modo:** El BM64 cuenta con una serie de modos de inicialización , en los que contamos con unas funciones u otras,los dos modos que utilizamos fueron
 - ***Flash test mode*** Nos permite escribir en la EEPROM y leer sus registros, en la web de Microchip podemos descargar el archivo BM6x Software and Tools (DSPK v2.1.3), y dentro encontramos un excel en el que tenemos la lista de direcciones y su función en la EEPROM.
 - ***Flash app mode*** Sera nuestro modo principal, carga el perfil A2DP y nos permite emparejar dispositivos bluetooth.
- **Interfaz UART** con un *baud rate* por defecto de 115,200 bits por segundo.
- **Audio I^2S** a 48 kHz y 16 bits de resolución.

Seleccionar modo nuestra placa suplementaria cuenta con un *jumper* para cambiar entre *Flash app mode* (el modo en el que se ejecuta nuestra aplicación) y *Flash test mode* para escribir en la EEPROM, modificando las distintas configuraciones por defecto del módulo. Dicho esto, en la tablas 12 observamos que según establezcamos el pin P2_0 en high o low , entraremos a un modo u otro, este pin por defecto cuenta con un *pull-up* por lo que esta high todo el tiempo, con la ayuda del *jumper* lo colocamos a low. finalmente en la tabla 13, observamos las distintas posibilidades para audio, que por defecto es doble canal.

	EAN	P2_0
ROM APP	ON	OFF
ROM TEST	ON	ON
Flash App	OFF	OFF
Flash Test	OFF	ON

Cuadro 12: Configuración del modo

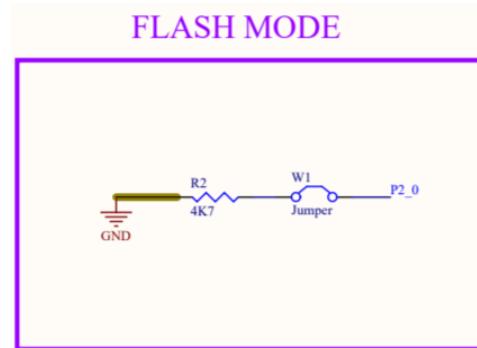


Figura 58: Esquemático de nuestro switches.

	P1_5	P3_6
Single	OFF	OFF
Double	ON	ON
L	OFF	ON
R	ON	OFF

Cuadro 13: Configuración de los canales de audio.

13. BOM

Nº De Orden	Descripción	Cantidad	Designador	Fabricante	Nº Referencia Fabricante	Nº Ref. RS-Online
1	BlueTooth v5.0 Módulos de transceptor RF 2.4GHz Integrado, rastreo Montaje en superficie	1	U1	Microchip Technology	BM64SPKS1MC1-0001AA-0001AA	BM64SPKS1MC1-0001AA-ND
2	Rojo 624nm Indicación led: discreta 2V 1205 (3x1212 métrico)	2	L1, L3	Würth Elektronik	156125RS57000	732-13395-1-ND
3	Azul 470nm Indicación led: discreta 3.2V 1205 (3x1212 métrico)	2	L2, L4	Würth Elektronik	156125BS57000	732-13391-1-ND
4	Termistores NTC 100k 0805 (2012 métrico)	1	RT1	Vishay BC Components	NTCS0805E3104FXT	BC2562CT-ND
5	2 (1 x 2) Conector de derivación de posiciones Negro Parte superior cerrada 0.100" (2.54mm) Estilo	1	W1	Sullins Connector Solutions	S9000-ND	S9000-ND
6	USB-B (USB TYPE-B) USB 2.0 Receptáculo Conector 4 Posiciones Montaje en superficie ángulo recto	1	J1	TE Connectivity AMP Connectors	1-1734346-1	A110862CT-ND
7	220 Ohms @ 100MHz 1 Línea de impedancia Perillas de ferrita y astillas 0805 (2012 métrico) 2A 45mOhm	2	FL1, FL2	Murata Electronics	BLM21PG221SN1D	490-1054-1-ND
8	Abrazadera Ipp TVS - Diodos Montaje en superficie TO-236AB	1	DP1	Nexperia USA Inc.	1727-3936-1-ND	1727-3936-1-ND
9	USB Puerlo, USB a UART USB 2.0 UART Interfaz 28-SSOP	1	IC1	FTDI, Future Technology Devices International Ltd	FT232RL-REEL	768-1007-1-ND
10	Nivel de voltaje Tractores Bidireccional 1 circuitos 4 canales 100Mbps 14-TSSOP	1	D1	Texas Instruments	TXB0104PWR	296-21929-1-ND
11	Dot Matrix Display Module 8 x 8 Common Anode Green 5V I²C 1.48" L x 1.48" W x 0.29" H (37.70mm x 37.70mm x 7.30mm)	2	M1, M2	Seeed Technology Co., Ltd	104020150	1597-104020150-ND
12	Decodificador/demultiplexor 1 x 4:16 24-SOIC	1	U2	Texas Instruments	CD74HC154M96	296-24352-2-ND
13	1μF -20%, +80% 16V Capacitores cerámicos Y5V (F) 0805 (2012 métrico)	1	C1	KEMET	C0805C105Z4VACTU	399-8011-1-ND
14	22μF ±10% 16V Capacitores cerámicos X5R 1210 (3225 métrico)	1	C2	KEMET	C1210C226K4PACTU	399-5092-1-ND
15	Canal P Montaje en superficie 60V 150mA (Ta) 300mW (Ta) PG-SOT323-3	16	Q1 a Q15	Infinion Technologies	BSS84PWH6327XTSA1	BSS84PWH6327XTSA1CT-ND
16	0.1μF -20%, +80% 50V Capacitores cerámicos Y5V (F) 0805 (2012 métrico)	1	C3	KEMET	C0805C104Z5VACTU	399-1177-1-ND
17	10μF ±10% 10V Capacitores cerámicos X5R 0603 (1608 métrico)	1	C8	Murata Electronics	GRM188R61A106KE69J	490-14372-1-ND
18	10000pF ±10% 100V Capacitores cerámicos X7R 0805 (2012 métrico)	1	C6	KEMET	C0603C103M5RACTU	399-7842-1-ND
19	200 Ohms ±0.1% 0.1W, 1/10W Resistencia en microprocesador 0603 (1608 métrico) AEC-Q200 automotriz Película delgada	9	R6 a R13, R1	Panasonic Electronic Components	ERA-3AEB201V	P200DBCT-ND
20	860 Ohms ±0.1% 0.125W, 1/8W Resistencia en microprocesador 0805 (2012 métrico) AEC-Q200 automotriz Película delgada	1	R4	Panasonic Electronic Components	ERA-6AEB8662V	P86.6KDACT-ND
21	1 MOhms ±10% 0.25W, 1/4W Resistencia en microprocesador 0805 (2012 métrico) AEC-Q200 automotriz Resistente a la humedad Película gruesa	1	R3	Yageo	RC0805FR-071ML	311-1.00MCRCT-ND
22	3.92 MOhms ±10% 0.25W, 1/4W Resistencia en microprocesador 0805 (2012 métrico) AEC-Q200 automotriz, protección contra impulsos Película gruesa	1	R2	Panasonic Electronic Components	ERJ-PB6B3921V	P20661CT-ND
23	0.1μF -20%, +80% 25V Capacitores cerámicos Y5V (F) 0603 (1608 métrico)	6	C9, C10, C7, C5, C4, C11	KEMET	C0603C104Z3VACTU	399-1100-1-ND

Figura 59: Bill Of Materials.

