Trabajo Fin de Grado
Grado en Ingeniería Electrónica Industrial

Creación de juegos en SoC FPGA utilizando una matriz de led RGB

Autor:
Carlos Cotano Barroso

Tutores:
Carlos Jesús Jiménez Fernández
Erica Tena Sánchez

Dept. de Tecnología Electrónica
Escuela Politécnica Superior
Universidad de Sevilla

Sevilla, 2024

Resumen

En desarrollos de sistemas complejos con altas prestaciones las opciones a las que se suele recurrir suelen pasar por las FPGA, combinaciones de microprocesadores/microcontroladores y FPGA, o por **FPGA que cuentan con núcleos hardware de procesadores como parte de su estructura (SoC-FPGA)**. [1]

En algunos casos, integrar un SoC con FPGA puede resultar en una solución más rentable y eficiente en términos de espacio en comparación con utilizar componentes separados para el procesamiento de CPU y FPGA. Esto es especialmente cierto, sobre todo en sistemas integrados complejos donde el espacio y el costo son factores críticos.

El principal objetivo de este trabajo es tomar contacto con las tecnologías SoC-FPGA, estudiando su estructura y sus capacidades, aprendiendo su metodología de diseño y realizando diseños que pongan en valor lo aprendido.

Como ejemplo de diseño, en este trabajo se ha realizado un juego que consiste en una versión simplificada de un juego estándar de recoger objetos. El objetivo de este juego es recoger todos los objetos sin que ninguno llegue al suelo. Cada objeto recogido acumulará puntos, hasta que finalmente, cuando un objeto caiga, se mostrará la puntuación total alcanzada por el jugador. Los objetos aparecerán de forma aleatoria, y su velocidad de caída podrá variar a medida que avanza el juego.

Este juego se ha implementado sobre una SoC-FPGA realizando una parte del diseño en hardware y otra en software. Este SoC será el Zynq-7000 de Xilinx y la plataforma sobre la que está montado es la placa Pynq-Z2. Como componentes adicionales se usará un Keypad de Digilent con puerto Pmod y una matriz de 32x8 LEDs RGB con protocolo WS2812, que actuarán como entrada y salida del juego respectivamente. Se ha usado Vitis-2022 como entorno de desarrollo para realizar los diseños.

Abstract

In the development of complex systems with high performance, the options usually used are FPGAs, combinations of microprocessors/microcontrollers and FPGAs, or **FPGAs that have processor hardware cores as part of their structure (SoC-FPGA**). [1]

In some cases, integrating an SoC with FPGA can result in a more cost-effective and space-efficient solution compared to using separate components for CPU and FPGA processing. This is especially true, particularly in complex embedded systems where space and cost are critical factors.

The main objective of this paper is to get acquainted with SoC-FPGA technologies by studying their structure and capabilities, learning their design methodology and creating designs that demonstrate the acquired knowledge.

As an example of design, this project includes the creation of a game that consists of a simplified version of a standard object-collecting game. The objective of this game is to collect all the objects before any of them reach the ground. Each object collected will accumulate points, until finally, when an object falls, the total score achieved by the player will be displayed. The objects will appear randomly, and their falling speed may vary as the game progresses.

This game has been implemented on a SoC-FPGA with part of the design in hardware and part in software. The SoC used is the Xilinx Zynq-7000, and the platform on which it is mounted is the Pynq-Z2 board. As additional components a Digilent Keypad with Pmod port and a 32x8 RGB LED matrix with WS2812 protocol will be used, which will act as input and output of the game respectively. Vitis-2022 has been used as development environment to realize the designs.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Los diseñadores buscan constantemente formas de diseñar sus sistemas electrónicos que proporcionen una solución óptima y que dé respuestas a todos los requisitos de sus aplicaciones. En muchas situaciones, la solución óptima a menudo se implementa sobre dispositivos FPGAs (siglas de Field Programmable Gate Arrays), pero muchos diseñadores lamentablemente desconocen las capacidades de estos dispositivos y cómo incorporarlos entre otras cosas porque la evolución tecnológica hace que tanto la tecnología como la metodología de diseño esté en continua evolución.

Hay una amplia gama de tecnologías para la implementación de sistemas digitales, cada una de las cuales puede adaptarse mejor o peor a la aplicación y al diseño a realizar. Estas tecnologías incluyen microprocesadores (MPU) y microcontroladores (MCU) comerciales, unidades de procesamiento de gráficos (GPU), FPGA y dispositivos de sistema en chip (SoC) así como tecnologías ASIC. Decidir cuál usar requiere examinar los requisitos y consideraciones de la aplicación y del diseño a realizar.

Las FPGAs son ideales para el desarrollo rápido de prototipos para aplicaciones de altas prestaciones. Permiten iterar y probar diseños de hardware más rápidamente que con soluciones ASIC (Circuitos Integrados de Aplicación Específica). Además, si los requisitos cambian, la FPGA puede ser reprogramada para adaptarse. Un claro ejemplo podrían ser las tecnologías de vanguardia como las estaciones base 5G. Aquí los diseñadores deben tener en cuenta que los estándares y protocolos subyacentes todavía están evolucionando. Esto significa que los diseñadores deben poder responder rápida y eficientemente ante cualquier cambio en las especificaciones que estén fuera de su control. Del mismo modo, también es necesario poder responder a errores inesperados en la funcionalidad del sistema o fallos en la seguridad del sistema, modificar la funcionalidad existente o agregar una nueva funcionalidad para extender la vida útil del sistema.

En desarrollos de sistemas complejos con altas prestaciones las opciones a las que se suele recurrir suelen pasar por las FPGA, combinaciones de microprocesadores/microcontroladores y FPGA, o por **FPGA que cuentan con núcleos hardware de procesadores como parte de su estructura (SoC-FPGA)**. [1]

En algunos casos, integrar un SoC con FPGA puede resultar en una solución más rentable y eficiente en términos de espacio en comparación con utilizar componentes separados para el procesamiento de CPU y FPGA. Esto es especialmente cierto, sobre todo en sistemas integrados complejos donde el

espacio y el costo son factores críticos.

El principal objetivo de este trabajo es tomar contacto con las tecnologías SoC-FPGA, estudiando su estructura y sus capacidades, aprendiendo su metodología de diseño y realizando diseños que pongan en valor lo aprendido.

Como ejemplo de diseño, en este trabajo se ha realizado un juego que consiste en una versión simplificada de un juego estándar de recoger objetos. El objetivo de este juego es recoger todos los objetos sin que ninguno llegue al suelo. Cada objeto recogido acumulará puntos, hasta que finalmente, cuando un objeto caiga, se mostrará la puntuación total alcanzada por el jugador. Los objetos aparecerán de forma aleatoria, y su velocidad de caída podrá variar a medida que avanza el juego.

Este juego se ha implementado sobre una SoC-FPGA realizando una parte del diseño en hardware y otra en software. Este SoC será el Zynq-7000 de Xilinx y la plataforma sobre la que está montado es la placa Pynq-Z2. Como componentes adicionales se ha usado un keypad con puerto Pmod y una matriz de 32x8 LEDs RGB con protocolo WS2812, que actuarán como entrada y salida del juego respectivamente. Se usará Vitis-2022 como entorno de desarrollo para realizar los diseños.

En la realización del proyecto, la matriz de leds RGB, de 256 LEDs es el elemento clave. Un elemento que se debe controlar perfectamente, y que debe cumplir con unos requisitos temporales de transmisión exactos para no dar lugar a errores. El punto de partida del proyecto es controlar la escritura de esta matriz de LEDs desde el hardware del dispositivo y con el procesador del dispositivo realizar la programación del juego añadiendo diferentes funcionalidades. Sin embargo, antes de abordar la programación específica en el procesador, se realizó un diseño hardware específico, enfocado a transmitir correctamente los datos a la matriz de LEDs utilizando un dispositivo FPGA previamente disponible en el laboratorio.

Esta decisión se tomó debido a que se requiere una precisión bastante alta en los tiempos de transmisión de datos con la matriz de LEDs. Para esta finalidad es preferible utilizar un hardware dedicado solamente a la transmisión diseñado sobre una FPGA, que un programa de aplicación que utilice los recursos de un SoC, debido a varias razones:

- Control de hardware: En una FPGA, tienes un control más directo sobre el hardware subyacente. Puedes diseñar circuitos digitales específicos y optimizarlos para cumplir con los requisitos de temporización muy estrictos. Esto permite manejar los tiempos de transmisión de datos con una precisión excepcional.
- Flexibilidad de diseño: Las FPGAs ofrecen una mayor flexibilidad en términos de diseño. Puedes implementar algoritmos personalizados y técnicas de temporización para garantizar tiempos de transmisión precisos entre diferentes partes del circuito. Además, puedes ajustar y modificar el diseño según sea necesario para cumplir con los requisitos de temporización específicos de tu aplicación.
- Menos latencia: Debido a su arquitectura altamente paralela y

configurable, las FPGAs pueden ofrecer menores latencias en comparación con los SoCs, lo que es crucial en aplicaciones que requieren tiempos de transmisión ultra precisos.

Una vez se haya realizado el diseño para el control de la matriz de LEDs, para completar el sistema, se ha programado una aplicación en el microprocesador del SoC, que permite un control más amplio de los leds, y se controla a través del teclado KYPD. Finalmente se ha realizado el juego, que será explicado en el capitulo *5*, con la finalidad de poder mostrar experimentalmente el funcionamiento de la placa.

La oportunidad de programar un SoC FPGA, que combina las funcionalidades de una FPGA y un microprocesador en un solo chip me ha llamado mucho la atención ya que es una experiencia nueva que me va a aportar muchos conocimientos.

Mi fascinación por la programación, que es algo que he descubierto cuando empecé a estudiar esta carrera, ha sido lo que me ha llevado a realizar el TFG sobre este tema. He podido aprender varios lenguajes de programación durante estos años. Cada lenguaje enfocado a un campo de la electrónica, pero dos de los lenguajes que sin duda me han parecido los más prácticos o esenciales a la hora de construir sistemas electrónicos han sido los lenguajes “C” y “VHDL”. Cada uno de ellos enfocado a una propia parte del sistema. El lenguaje “C” utilizado para la parte software, y el lenguaje “VHDL” para al diseño hardware. Este último lenguaje en particular permite construir sistemas más complejos y de alto rendimiento. En este trabajo los diseños tienen parte realizado en C y otra parte en VHDL.

Esta memoria se estructura en 6 capítulos y 2 apéndices. Este primer capítulo ha proporcionado una breve introducción al tema. En el segundo capítulo se presenta una revisión de la literatura del estado del arte, complementada con la metodología utilizada. El tercer capítulo describe la utilización de dos de los programas que son necesarios para realizar el diseño hardware y la aplicación software. El cuarto capítulo contiene una descripción de varios diseños previos a la implementación de la aplicación, que tienen gran relevancia en el desarrollo del proyecto, junto con si verificación a través de simulaciones. El quinto capítulo ofrece una descripción de la solución contemplada para realizar la aplicación del juego. Finalmente, en el sexto capítulo se extraen las principales conclusiones.

2 ESTADO DEL ARTE

En este capítulo, se explica brevemente el funcionamiento interno de los dispositivos programables y se habla sobre su evolución a lo largo de la historia. También se abarca la metodología de diseño seguida en el desarrollo del proyecto y se realiza una descripción de los elementos empleados, haciendo especial hincapié en la placa de desarrollo utilizada.

2.1 Evolución de las tecnologías de dispositivos programables

Las FPGA fueron inventadas en el año 1984 por Ross Freeman y Bernard Vonderschmitt, cofundadores de Xilinx, y surgen como una evolución de los CPLD (*Complex programmable logic device*). Exactamente, es el resultado de la unión de dos tecnologías, los dispositivos lógicos programables (PLD) y los circuitos integrados de aplicación específica (ASIC).

Los PLD, surgieron de las memorias PROM (*Programmable Read-Only Memory*), que evolucionaron a las PAL (*Programmable Array Logic*) que tenían la novedad de que incluían registros y un mayor número de entradas.

Los ASIC, sin embargo, siempre han sido potentes, pero tenían la gran desventaja de que había que realizar una gran inversión de tiempo y de dinero en ellos por la necesidad de fabricar un número muy elevado de muestras. Para reducir esto se impuso la modularización de los elementos de los circuitos, así como los ASIC basados en celdas de librería y la estandarización de máscaras.

El paso final fue combinar las dos estrategias con un mecanismo de interconexión que pudiese programarse utilizando fusibles, antifusibles o celdas RAM y celdas ROM. Estos fueron los innovadores dispositivos de Xilinx de mediados de los 80 del siglo pasado. Así nacieron las **FPGA**, cuyas características originalmente eran similares en capacidad y aplicaciones a los PLDs más grandes, aunque con diferencias puntuales que delatan puntos de partida diferentes [2].

Una FPGA está constituida esencialmente por 3 elementos básicos (Ver *Figura 1*):

- Bloques lógicos configurables (CLB).
- Recursos de interconexión.
- Bloques de Entrada/Salida.

Las FPGAs normalmente usan una memoria estática para ser programadas. Los elementos que se programan son el rutado de las interconexiones y los bloques lógicos programables (CLBs). Los recursos de interconexión actúan como

elementos de multiplexado y demultiplexado, y los CLBs implementan las funciones lógicas [3].

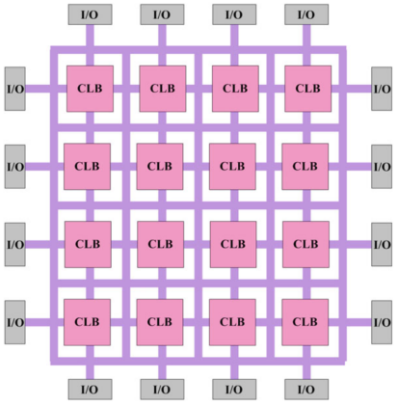

Figura 1. Arquitectura básica de una FPGA. Fuente: [3]

Los **CLB** constan esencialmente de LUTs y Flip-flops D. Los LUT (*Look-Up Table*) son utilizados para proporcionar funcionalidad lógica combinacional y, en algunos casos, almacenamiento. Además, en los CLB hay flip-flops D encargados de realizar la lógica secuencial. Cada CLB puede constar de un único elemento lógico básico (**BLE**) o de un grupo de BLE interconectados localmente. Un BLE sencillo (*Figura 2*) consta de una LUT y un Flip-Flop.

Un **LUT** es una SRAM que se utiliza para implementar una tabla de verdad. Dado que una SRAM básica no está sincronizada, la tabla de verdad funciona como cualquier elemento combinacional: al cambiar sus entradas cambian sus salidas después de un cierto retardo. El LUT más básico consta de 4 entradas. Cuantas más entradas tenga el LUT, más lógica podrá generar y menos bloques lógicos serán necesarios para implementar un circuito. Sin embargo, la complejidad del LUT crece exponencialmente con el número de entradas. La mayor eficiencia según [4] se consigue con un LUT de 4 entradas, que alcanza el mínimo consumo de energía [5].

El **Flip-flop** utilizado en FPGAs es el tipo D. Con este Flip-flop es posible implementar registros o almacenar datos secuenciales en el diseño de la FPGA.

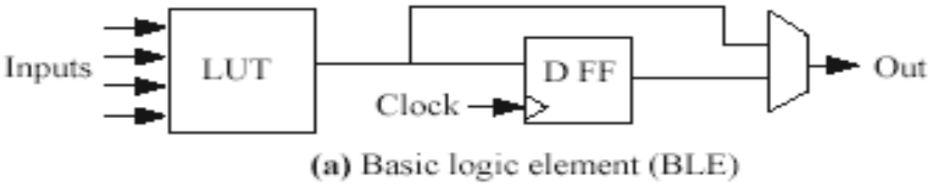

Figura 2. Elemento lógico básico (BLE). Fuente: [5]

Las conexiones entre los elementos lógicos pueden requerir rutas complejas, ya que los CLBs están dispuestos en una estructura bidimensional. Esto quiere decir que hay que realizar conexiones entre los CLBs y los “cables” y también entre los propios “cables”. Para ello, los cables suelen organizarse en canales que se extienden vertical y horizontalmente por el chip como en la *Figura 3.* Cada canal está compuesto por varios cables, el programa de diseño (o el diseñador) elegirá que cable se utilizará en cada canal para transportar una señal. Las señales de reloj también disponen de sus propias líneas de interconexión [6].

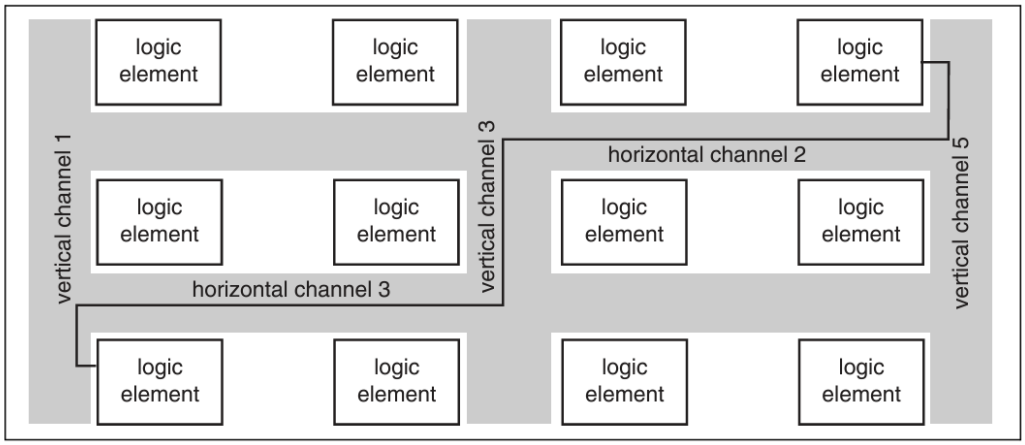

Figura 3. Recursos de interconexión. Fuente: [6]

Los pines E/S pueden denominarse bloques de E/S o **IOBs**. Generalmente se pueden programar como entradas, salidas o bidireccionales y a menudo proporcionan otras características como conexiones de bajo consumo o de alta velocidad. Estos bloques actúan como interfaz entre señales externas de entrada/salida y los bloques lógicos. Estos bloques E/S están dotados de buffers de recepción y transmisión de señales, que permiten adaptar los niveles de corriente y tensión de las señales para que sean compatibles con los estándares de comunicación requeridos.

La complejidad y composición de todos estos elementos mencionados anteriormente, varía en función de la tecnología FPGA utilizada, pero esos elementos son indispensables en la arquitectura de una FPGA.

La primera FPGA comercial fue la *XC2064* de Xilinx (*Figura 4*), que estaba compuesta por 64 bloques lógicos configurables (CLB) en una cuadrícula de 8x8,

Figura 4. FPGA XC2064 de Xilinx. Fuente: [7]

18 MHz, 58 pines de entrada/salida, 1200 puertas lógicas equivalentes y un circuito integrado muy grande [7].

La evolución de las FPGA ha venido marcada por avances tecnológicos, mejoras en la capacidad de integración, velocidades de operación y un aumento de la complejidad de los dispositivos. Todo esto de la mano de Xilinx, que puede considerarse uno de los principales fabricantes de esta tecnología. Las últimas generaciones de las FPGAs de Xilinx, pueden dividirse en los siguientes grupos de tecnologías, de acuerdo al tamaño de los transistores (Ver *Figura 5*):

- 45 nm: En este grupo, se incluye a la familia Spartan-6. Esta es una familia de baja potencia y costo eficiente, pero con un rendimiento razonable para una variedad de **aplicaciones de interfaz avanzado** entre diferentes protocolos de redes industriales, conectividad y en vehículos o video y gráficos de alta resolución. Ofrece una elevada relación lógica-pin, un pequeño factor de forma, el procesador software MicroBlaze™ y un gran número de protocolos de E/S.
    - 28 nm: Esta tecnología es la utilizada en las familias de la serie 7 de Xilinx: Spartan-7, Artix-7, Kintex-7 y Virtex-7, ordenadas desde la gama más baja hasta la más alta respectivamente. La característica de esta serie es que todos los elementos que hay dentro de los dispositivos son iguales en todas las familias, lo que hace que un diseño pueda ser implementado en cualquiera de las familias sin necesidad de realizar cambio alguno. Lo único que varía es la proporción de los elementos más avanzados, que son mayores en las familias de más prestaciones.
        - Spartan-7: son potentes y eficientes en términos de energía, además de tener un tamaño compacto. Vienen con un procesador MicroBlaze™ que puede manejar más de 200 millones de instrucciones por segundo y admiten memoria DDR3 rápida. También incluyen un convertidor analógico-digital (ADC) y características de seguridad especiales. Son ideales para aplicaciones industriales, de consumo y automotrices, como la conectividad, la fusión de datos de sensores y la visión integrada.
        - Los dispositivos Artix-7 ofrecen un buen rendimiento por cada unidad de energía utilizada, así como velocidades rápidas y procesamiento integrado en una FPGA que resulta económica. Suelen ser usados en dispositivos como radios programables, cámaras de reconocimiento visual y sistemas de comunicación inalámbrica económicos.
        - La familia Kintex-7 proporciona un equilibrio entre precio, rendimiento y eficiencia energética y cuenta con una buena compatibilidad con tecnologías comunes como PCIe® y Ethernet. Estan enfocadas en aplicaciones relacionadas con redes inalámbricas y soluciones de vídeo.

- Por último, la familia de FPGAs Virtex-7 está diseñada para ofrecer un alto rendimiento y se utilizan en una variedad de aplicaciones, desde redes hasta equipos médicos y sistemas de radar.

- Las familias fabricadas con tecnología de 20 nm, son llamadas "Ultrascale". Tienen unas mejoras significativas en rendimiento y eficiencia. Son ideales para aplicaciones que requieren un procesamiento rápido de datos, como redes de alta velocidad y equipos médicos avanzados.
- Finalmente, las FPGA de 16 nm, conocidas como "Ultrascale +", son las que contienen la tecnología más avanzada actualmente. Proporcionan aún más potencia y eficiencia energética, siendo ideales para una amplia gama de aplicaciones, desde sistemas de visión artificial hasta redes de alta velocidad y dispositivos de Internet de las cosas (IoT).

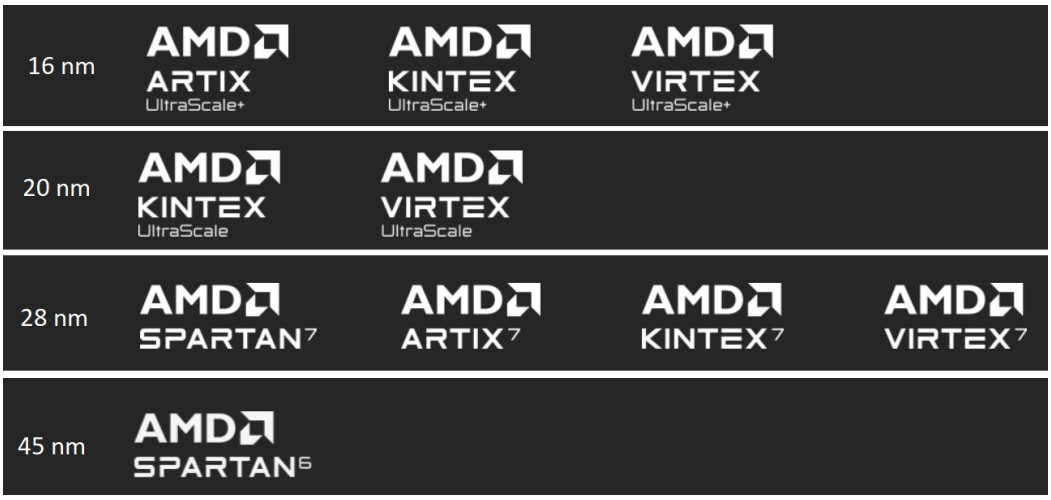

Figura 5. Familias de FPGAs de Xilinx. Fuente: www.xilinx.com

Uno de los avances que interesa destacar es la integración de FPGA y SoC. Esto se da en marzo de 2011, cuando Xilinx lanza al mercado la familia Zynq® -7000. Posteriormente, Xilinx ha desarrollado nuevos SoCs (Ver *Figura 6*) basados en tecnologías más avanzadas, que permiten crear sistemas más complejos.

Figura 6. Familias de SoCs Xilinx. Fuente: xilinx.com

La familia de dispositivos Zynq® -7000 integra en un solo chip (SoC) un sistema basado en un procesador ARM® Cortex®-A9 de doble núcleo (PS, *Processing System*), y una estructura FPGA (PL *Programmable Logic*). El Subsistema PS incluye varios periféricos dedicados (como controladores de memoria, USB, UART, I2C, SPI…) y estos pueden ser extendidos con hardware IP adicionales “superpuestos” en la PL [8]. Estas “superposiciones” o librerías hardware, son diseños FPGA que extienden y aceleran las aplicaciones del procesador, gracias a la lógica programable.

Las dos áreas del SoC (**PS** y **PL**) están interconectadas por una serie de interfaces que siguen el estándar de conexión **AXI4**. Estas interfaces permiten implementar una lógica personalizada en el PL, que puede conectarse con el PS (Ver *Figura 7*) y, así ampliar los periféricos que están disponibles y visibles para

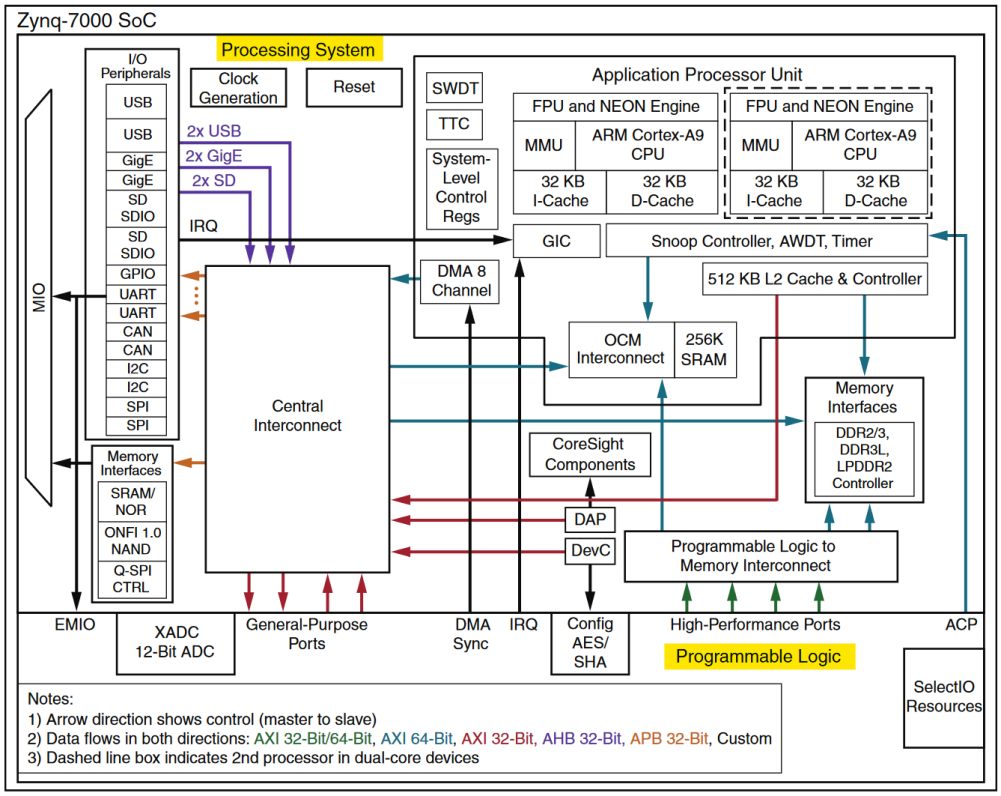

Figura 7. Estructura interna SoC Zynq-7000. Fuente: docs.xilinx.com

el procesador en su mapeo de memoria. [9].

AXI4 ofrece una amplia gama de opciones, y por ello, existe la variante AXI4-Lite con unas funciones más reducidas, y que será el protocolo que usaremos en este proyecto, ya que cumple sus necesidades con creces.

Al aumentar el tamaño y la complejidad de los dispositivos, los diseñadores necesitan nuevas técnicas y herramientas de diseño y una nueva metodología de diseño para los *System-on-chip* (SoC). Un diseñador de SoC, tiene un conocimiento limitado de la estructura interna de estos núcleos, pero con la nueva metodología, el diseñador podría ser capaz de utilizar diferentes núcleos y combinarlos en un chip para implementar funciones complejas, sin necesidad de tener un conocimiento avanzado del interior del núcleo.

En esta metodología se pueden usar **bloques hardware prediseñados y verificados previamente**, a menudo llamados *Cores* o *Intellectual Property* (**IP**). Estos se obtienen de fuentes internas o de terceros y se combinan en un solo chip. Estos *cores* pueden incluir procesadores integrados, bloques de memoria o circuitos que manejan funciones de procesamiento específicas [10]. El protocolo AXI4, hace posible interconectar todos estos *Cores* para así poder

construir un sistema electrónico completo y funcional.

El enfoque SoC es atractivo por varias razones. Al utilizar esta técnica, el diseñador no necesita comprender los detalles de cada *core*, sino que **puede centrarse en cuestiones superiores a nivel del sistema**. Mientras tanto, **los *cores* pueden ser diseñados por expertos en un protocolo o función individual y reutilizados por terceros.**

2.2 Metodología de diseño y desarrollo

La metodología de diseño para programar el SoC FPGA, estará basado en un esquema de bloques dentro del entorno de diseño Vivado y Vitis de AMD-Xilinx que integra el conjunto de herramientas necesarias para crear, verificar y programar el SoC-FPGA. Para crear los **bloques *IP*,** integrarlos e interconexionarlos correctamente en el sistema, se hace uso de la **herramienta Vivado™ Design Suite**, con su herramienta Vivado™ IP Integrator. Vivado es un software de diseño para las FPGAs y SoCs de AMD-Xilinx. Entre sus funciones incluye: Herramientas de entrada de diseño, síntesis, colocación y rutado, y verificación/simulación.

Los diseños en Vivado son normalmente construidos a nivel RT, interconectando las distintas interfaces de cada *núcleo IP* haciendo uso de un diagrama de bloques. Adicionalmente, para más precisión, es posible editar estos bloques a nivel lógico. Es decir, podemos añadirle **nuevas funcionalidades** hardware a nuestro SoC según nos convenga. Estos diseños hardware se pueden realizar en VHDL o en Verilog.

Después de finalizar el diseño hardware en Vivado, se puede exportar a través de un fichero de extensión ".xsa". Este fichero contiene los archivos XML requeridos por Vitis para interpretar las IPs usadas en el diseño hardware y el mapeo de memoria desde la perspectiva del procesador que serán los cimientos de la aplicación que se construya [11].

Vitis IDE (entorno de desarrollo integrado) forma parte de la plataforma de software unificada de Vitis. Este entorno está enfocado al desarrollo de aplicaciones software integradas dirigidas a procesadores integrados AMD y funciona con los diseños hardware creados con Vivado, como se ha expuesto anteriormente. Este IDE, está basado en el estándar de código abierto *Eclipse*.

Entre sus funciones destacan: Editor de código C/C++ con muchas funciones predefinidas, configuración de compilación de aplicaciones y generación automática de Makefiles, entorno integrado para depuración, herramientas especiales enfocadas para configurar FPGA, creación de imágenes de arranque

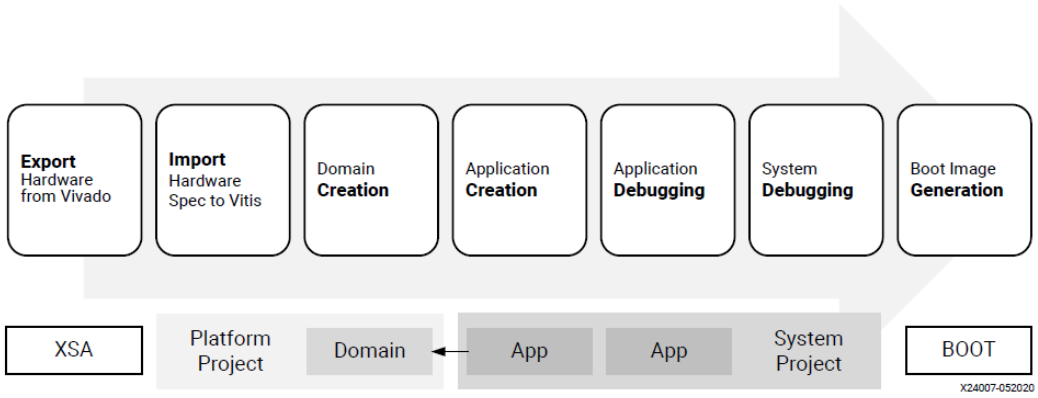

Figura 8: Flujo de diseño general. Fuente: [12]

(Bootable Image), programación de memoria flash…

La aplicación que se crea en Vitis parte del archivo XSA, exportado por el diseñador hardware. Este archivo es importado a Vitis creando una plataforma. Vitis usa una arquitectura de plataforma y aplicación para poder **unificar la arquitectura** de todos los diferentes tipos de sistemas y así ser más versátil (Ver *Figura 8*). Una plataforma incluye especificaciones hardware y configuraciones del entorno software.

Las configuraciones del entorno del software se denominan *dominios* y también forman parte de una plataforma. Los desarrolladores de software crearán **aplicaciones** basadas en la plataforma y los dominios. Durante el desarrollo, es posible **depurar** las aplicaciones en Vitis.

Una vez las aplicaciones funcionen correctamente, hay que realizar una verificación del **sistema software** (esto es necesario en sistemas complejos, ya que estos tendrán varias aplicaciones que se comunicaran entre sí, y se les debe tratar como un conjunto).

Cuando el sistema software esté listo, Vitis IDE puede ayudar a crear imágenes de arranque que inicializan el sistema y ejecutan las aplicaciones.[12]

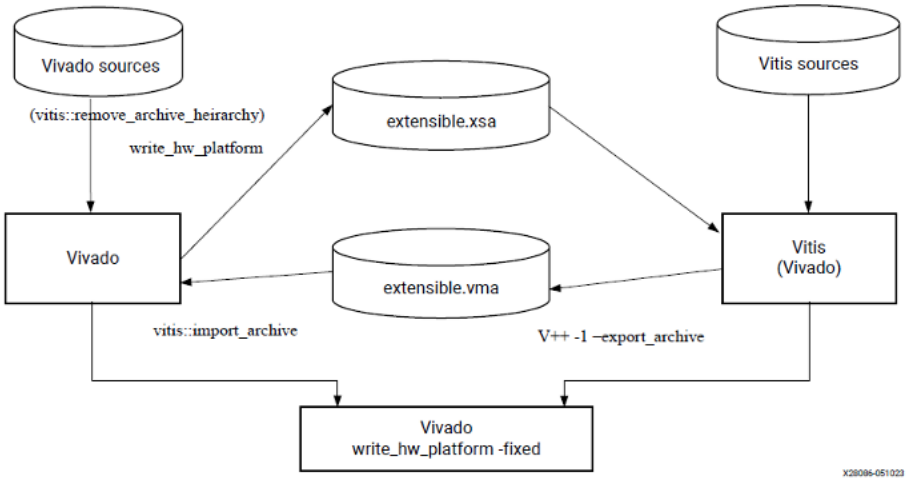

Figura 9: Flujos de diseño posibles. Fuente: [13]

La creación de una aplicación en el SoC, puede realizarse de forma **bidireccional**, sin importar el orden, tal y como se puede observar en la *Figura 9*. Es decir, primero se puede crear el hardware necesario en Vivado y posteriormente crear la aplicación en Vitis con ese hardware. O bien, se puede crear la aplicación inicialmente, exportando un fichero ".vma" desde Vitis, y posteriormente crear el hardware necesario en Vivado.[13]

2.2.1 Diseño a nivel RT de la PL

Como ya se ha mencionado, se usará una FPGA para realizar un diseño para la transmisión de datos a los leds. El proceso de diseño en una FPGA implica varios pasos, desde la especificación de los requisitos del sistema hasta la implementación y verificación en la FPGA. Una breve descripción de los pasos para realizar un diseño en una FPGA es la siguiente:

1- Definición de requisitos del sistema: Identificar los objetivos y las funcionalidades que se desean lograr con el diseño en la FPGA. Especificar los requisitos de rendimiento, consumo de energía, interfaces de E/S, y cualquier otra consideración importante.

2- Diseño y modelado del circuito: Se utilizan lenguajes de descripción de hardware (HDL) como Verilog, o VHDL (este será el que se usará en este proyecto) para crear la descripción del circuito que implementará el diseño. Se divide el diseño en bloques funcionales lógicos y se desarrollan modelos HDL para cada uno de ellos.

3- Síntesis del diseño: Se utilizan herramientas de síntesis para traducir el diseño HDL en una descripción de hardware específica de la FPGA. Durante la síntesis, se asignan recursos físicos de la FPGA (como LUTs,

flip-flops, bloques de memoria, etc.) al diseño, y se optimiza el diseño para cumplir con los requisitos de rendimiento y área.

4- Simulación del diseño: Después de la síntesis, es necesario realizar simulaciones para verificar el comportamiento funcional y temporal del diseño antes de la implementación en la FPGA. Este paso es crucial para garantizar que el diseño funcione correctamente y cumpla con los requisitos de rendimiento y temporización antes de ser implementado en hardware. Las simulaciones tras la síntesis te permiten detectar y corregir posibles problemas antes de pasar a la etapa de implementación en la FPGA, lo que ayuda a reducir el tiempo y los costos asociados con la depuración en el hardware físico.

5- Implementación y asignación de pines: Se utilizan herramientas de implementación para traducir la descripción de hardware sintetizada en un archivo de configuración específico de la FPGA. En esta etapa, se asignan los pines de entrada/salida del diseño a los pines físicos de la FPGA y se realiza el enrutamiento de las conexiones entre los diferentes bloques del diseño.

6- Generación de bitstream y programación de la FPGA: Se genera el bitstream, que es el archivo binario que contiene la configuración del diseño para la FPGA. Posteriormente, se utilizan herramientas de programación para cargar el bitstream en la FPGA y configurarla con el diseño implementado.

7- Verificación y depuración: Este último paso consiste en verificar el funcionamiento del diseño en la FPGA utilizando pruebas funcionales y verificación en hardware. De esta forma se podrá depurar cualquier problema encontrado durante la verificación, ajustando el diseño según sea necesario y repitiendo el proceso de implementación si es necesario.

Para el diseño del hardware en VHDL de este proyecto, se ha hecho uso de la placa de desarrollo **Nexys4 DDR™** del fabricante Digilent® (*Figura 10*). Esta placa incorpora una FPGA Artix-7™ de Xilinx. (Part number XC7A100T – 1CSG324C). Y es posible acceder a su programación gracias al software **ISE Design Suite de Xilinx** y al software **Adept** proporcionado por Digilent, para el volcado de la configuración a la placa. Esta placa se ha elegido, ya que era la más accesible en ese momento, y ya había una experiencia previa en su programación.

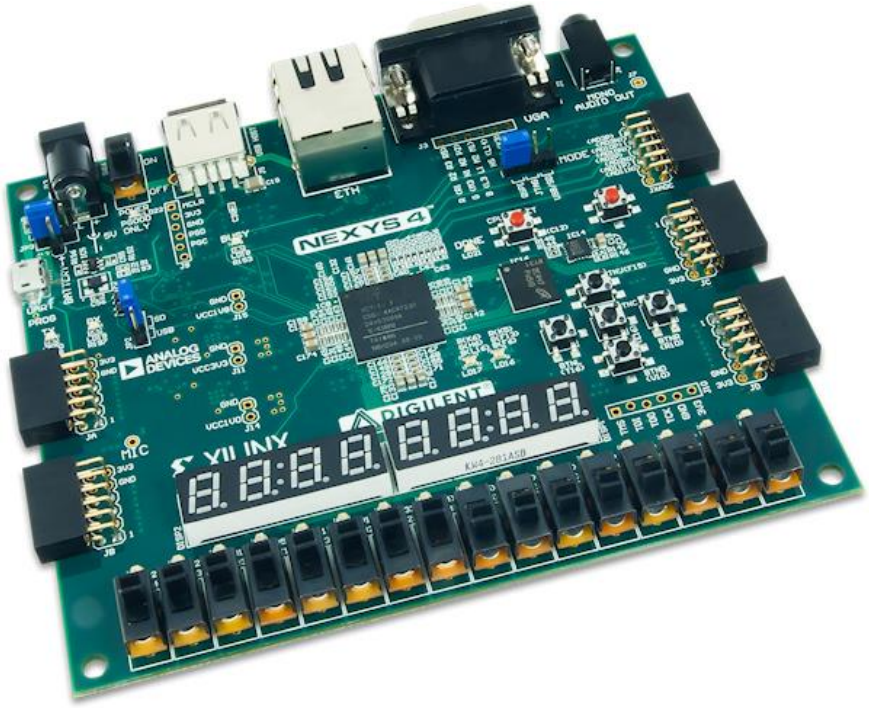

Figura 10: Placa de desarrollo Nexys4 DDR. Fuente: eu.robotshop.com

Un breve resumen de sus características sería el siguiente:

Características FPGA	▪ 15,850 slices con 6 LUTs y 8 flip-flops cada uno. ▪ 4.860 Kbits de bloques de memoria RAM de doble puerto. ▪ Seis módulos de gestión de reloj, cada uno con un bucle de bloqueo de fase (PLL). ▪ 240 CLBs DSP. ▪ Velocidades de reloj interno superiores a 450 MHz. ▪ Convertidor analógico – digital (XADC) en chip
Puertos y periféricos	▪ 16 switches. ▪ 16 LEDs. ▪ 2 displays de 7 segmentos de 4 dígitos. ▪ Puerto USB-UART. ▪ 2 LEDs tri-color. ▪ Conector para tarjeta Micro SD.

	▪ Salida VGA de 12 bits. ▪ Salida de audio PWM. ▪ Micrófono PDM. ▪ Acelerómetro de 3 ejes. ▪ Sensor de temperatura. ▪ 10/100 Ethernet PHY ▪ 128 MiB DDR2. ▪ Serial Flash. ▪ 4 puertos Pmod. ▪ Pmod para señales XADC. ▪ Puerto USB-JTAG para la programación y comunicación con la FPGA. ▪ USB HID para ratones, teclados y tarjeta de memoria.

Tabla 1: Características Nexys4 DDR. Fuente: [14]

2.2.2 Placa para el desarrollo del proyecto

La placa de desarrollo SoC-FPGA con la que se ha trabajado en este proyecto es la **PYNQ-Z2** (*Figura 11*), que incorpora el chip Zynq-7000 SoC XC7Z020-1CLG400C de la familia Zynq® -7000. En esta placa se han usado parte de los diseños probados sobre la Nexys4 y se creará un sistema basado en un diagrama de bloques para interconectar todos los componentes necesarios entre sí. Para realizar una muestra de su funcionamiento se creará además un pequeño juego que será programado en el PS del SoC.

PYNQ (Python Productivity for Zynq) es un Proyecto de Xilinx® de código abierto basado en el SoC Zynq de Xilinx. Estas placas de desarrollo están enfocadas a utilizar el lenguaje y las bibliotecas de Python para poder aprovechar las ventajas de la lógica programable y los microprocesadores de Zynq y con esto poder construir sistemas integrados con mayor capacidad [15].

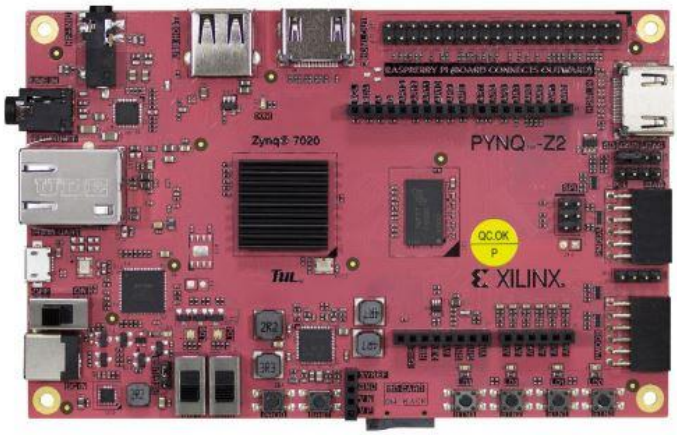

Figura 11: Placa de desarrollo PYNQ – Z2. Fuente: www.mouser.com

Existen 3 modos de programar la PYNQ - Z2:

- Linux: En este modo, la PYNQ-Z2 ejecuta un sistema operativo Linux completo, lo que permite acceder a todas las capacidades de un entorno Linux estándar. Se puede programar la PYNQ-Z2 utilizando herramientas y lenguajes de programación estándar de Linux, como C/C++, Python, shell scripting, etc. Gracias a la comunicación que se puede establecer entre la placa y el PC, vía Ethernet, es posible acceder a la terminal de Linux, lo que permite ejecutar comandos directamente en la placa y utilizar herramientas de desarrollo estándar. Este modo es útil para proyectos que requieren un mayor control sobre el hardware y el sistema operativo, así como para el desarrollo de aplicaciones complejas que no se pueden realizar fácilmente en el entorno de Jupyter.
- Python: Es posible programar la PYNQ-Z2 en Python, gracias al entorno Jupyter. Para ello, es necesario, tener la PYNQ-Z2 con su tarjeta micro SD insertada, el jumper JP1 debe estar en modo SD, y hay que conectar la PYNQ-Z2 a una fuente de alimentación y al PC a través del cable USB, y a su vez conectarla también con el PC a través del cable Ethernet. Al encender la placa, hay que abrir un navegador web y acceder a la dirección IP de la PYNQ-Z2. Esta IP es 192.168.2.99, y para acceder, la dirección sería: https://192.168.2.99:9090. Una vez dentro de la IP, hay que iniciar sesión en Jupyter, Una vez en Jupyter, para iniciar un nuevo programa, habría que crear un nuevo notebook:

  En el nuevo notebook, se puede escribir y ejecutar código Python como se haría en cualquier otro entorno de desarrollo de Python. Se pueden importar las bibliotecas de PYNQ, como pynq, pynq.lib, pynq.overlay, etc., para interactuar con los recursos de la PYNQ-Z2, como GPIO, aceleradores FPGA, periféricos, etc.

  Toda la información sobre este método puede encontrarse en la documentación oficial de PYNQ: https://pynq.readthedocs.io/en/v2.5/jupyter_notebooks.html

Este modo es ideal para el desarrollo rápido de prototipos, la experimentación y la demostración de conceptos, así como para proyectos que se centran en el procesamiento de señales, aprendizaje automático, etc.

- Vitis: Vitis es un entorno de desarrollo integrado (IDE) de Xilinx que permite el desarrollo de software y hardware para dispositivos FPGA. Con Vitis, es posible escribir código en lenguajes de programación como C, C++ y OpenCL, y compilarlo para ejecutarse en la FPGA de la PYNQ-Z2. Vitis proporciona herramientas avanzadas de diseño y depuración, así como bibliotecas optimizadas y plantillas de proyectos para acelerar el desarrollo. Este modo es adecuado para proyectos que requieren un rendimiento optimizado, una integración profunda con la arquitectura de la FPGA y un control preciso sobre los recursos de hardware. Se va a hacer hincapié en la programación a través de Vitis IDE en el capítulo *3*, ya que es el método de programación que se usará en el proyecto.

En este proyecto se ha optado por programar la placa desde el entorno Vitis.

Para arrancar la PYNQ-Z2, existen 3 modos:

- Modo SD. La placa contiene una ranura para tarjetas micro SD. De esta manera, podemos almacenar en esta una imagen de arranque para que el SoC pueda arrancar desde aquí.
- Modo QSPI. La placa contiene una memoria Flash Quad SPI, en la que se puede programar un bitstream o un firmware (Imagen de arranque) y el SoC arrancará desde esta memoria. (Estrategia similar al modo SD, pero sin necesidad de tener esta tarjeta).
- Modo JTAG. JTAG (*Joint Test Action Group*) es una interfaz hardware que proporciona a cualquier ordenador un modo fácil de comunicarse con los chips en una tarjeta. Esta interfaz permite depurar programas. Sin embargo, la aplicación no podrá ser almacenada permanentemente en la placa por este medio. Este será el modo que se va a utilizar durante el proyecto, ya que es el que permite corregir los errores que van surgiendo más rápido. Si se desea un funcionamiento fijo para la placa, usaremos cualquiera de los otros dos métodos una vez hayamos depurado nuestra aplicación.

En este proyecto, la placa se usará para desarrollar una aplicación. Por tanto, se usará el modo JTAG, y cuando se termine de desarrollar la aplicación, ya sería conveniente generar una imagen de arranque (Boot Image) y almacenarla bien en la memoria flash integrada en el SoC o en una tarjeta micro SD externa, aunque esto no está dentro del alcance de este proyecto. Para generar la imagen de arranque, se puede recurrir a la herramienta *Bootgen* de Vitis, que permite generar la imagen a partir de un proyecto de aplicación. Si, por otra parte, interesa más que el SoC arranque desde la memoria QSPI, se puede recurrir a otra de las herramientas de Vitis que es el asistente *Program Flash Memory* [12].

Las características principales de la placa Pynq-Z2 son:

FPGA	▪ Zynq-7000 SoC XC7Z020-1CLG400C
Interfaces E/S	• Circuitería de programación USB-JTAG • USB OTG 2.0 • Puente USB-UART • 1 puerto Ethernet 10/100/1G • Entrada HDMI • Salida HDMI • Interfaz I2S con DAC de 24bits con jack TRRS de 3.5mm • Conector de entrada jack de 3.5 mm
Memoria	• DDR3 de 512 Mbytes con bus de 16 bits a 1050 Mbps • Memoria Flash Quad-SPI de 128 Mbits • Conector para tarjeta Micro SD.
Interruptores y LEDs	• 2 interruptores • 2 LEDs RGB • 4 LEDs • 4 Pulsadores
Relojes	• 1 reloj de 125 MHz para la lógica programable (FPGA) • 1 reloj de 50 MHz para el sistema de procesado (Procesador del SoC)
Puertos de Expansión	• 2 puertos Pmod (Con tensión VCC = 3,3V y 1A como corriente máxima) • 16 E/S completas para la FPGA (8 pines compartidos con el conector a Raspberry Pi) • 1 conector Arduino Shield • 24 E/S totales para FPGA. • 6 entradas analógicas para el XADC Single-ended 0-3.3V • Conector para Raspberry Pi • 28 E/S completas para FPGA (8 pines compartidos con el puerto Pmod A)

Tabla 2: Características PYNQ – Z2. Fuente: [16]

Cabe destacar dos componentes de la placa, que permitirán seleccionar diferentes modos de configuracion como son:

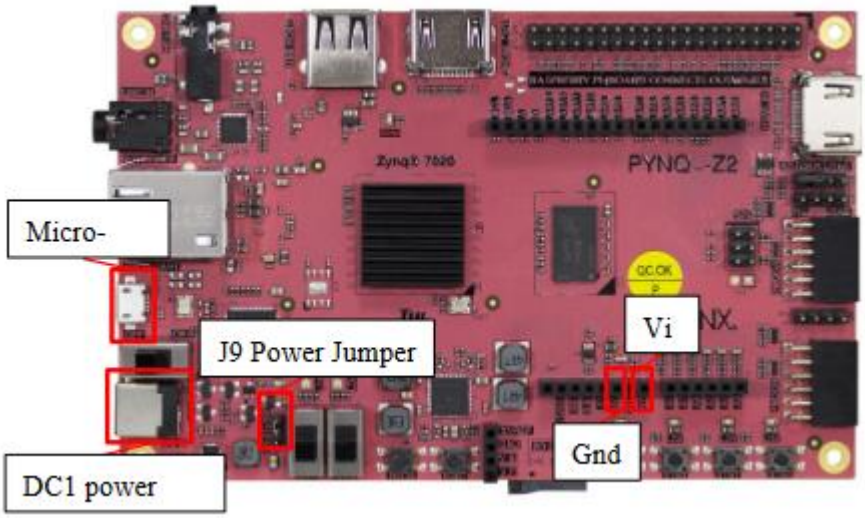

Figura 12: Alimentacion PYNQ – Z2.

1- Power Jumper J9 (*Figura 12*): Este Jumper selecciona el modo en que se va a alimentar la placa. Estos dos modos son USB y REG.

   - El modo USB sirve para alimentar la placa desde el Puerto MicroUSB, que se conectará al PC y proporcionará la potencia suficiente para la mayoría de proyectos.
   - El modo REG (*External Power REGulator/ Battery*) permite arrancar la placa desde una Fuente de alimentación externa o desde una batería. La fuente de alimentación externa se conectaría al puerto DC1, mientras que la batería se conectaría a través del pin “VIN", del grupo de conectores J7. El negativo de la batería se conectaría a un pin GND de este grupo de pines J7. Los límites de tensión de la placa van desde 7 VDC hasta 15 VDC, siendo recomendable usar como límite 12 VDC.

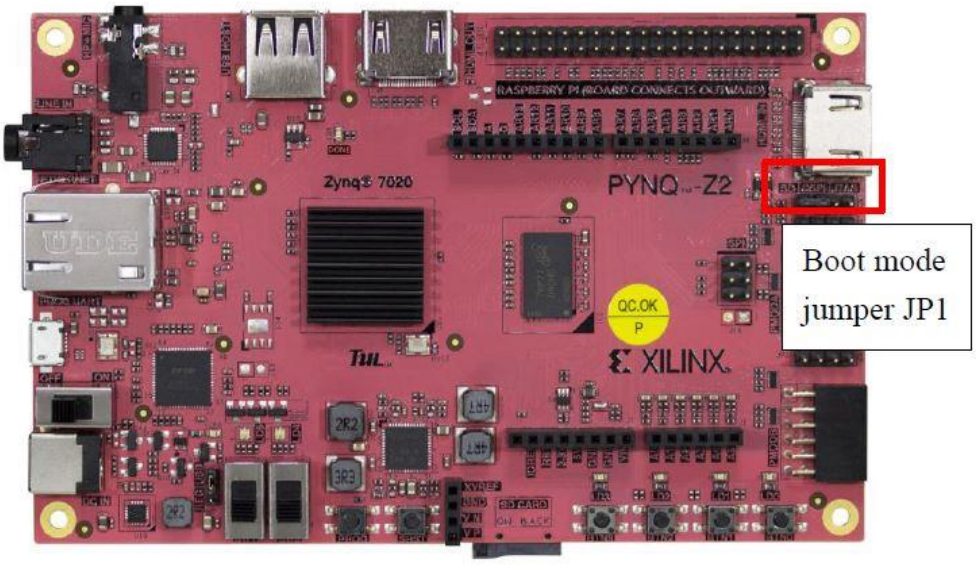

Figura 13: Cambio de modo de arranque en PYNQ-Z2

2- Jumper JP1 (*Figura 13*): Este Jumper, permite cambiar el modo de arranque de la placa según en la posición en la que esté. Entre estos modos tenemos: SD, QSPI y JTAG

2.2.3 Componentes adicionales

Destacamos aquí los dos componentes adicionales que se necesitan en el desarrollo de este proyecto, la matriz de LEDs RGB y el keypad.

2.2.3.1 Matriz de LEDs RGB 8x32

Esta matriz está compuesta por 256 leds WS2812 como se puede apreciar en la *Figura 14*, que internamente tienen un driver capaz de controlar 1 led RGB.

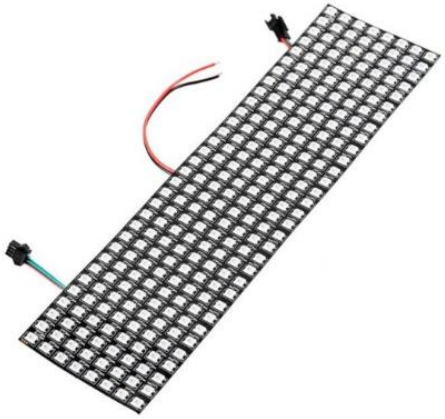

Figura 14: Matriz de leds RGB. Fuente: www.tinytronics.nl

El protocolo WS2812 se caracteriza por el encendido de LEDs RGB en cascada [17]. Por tanto, cada led tendrá además de sus terminales de alimentación y tierra, una entrada de datos y una salida de datos para que los datos se transmitan al resto de LEDs (*Figura 15*).

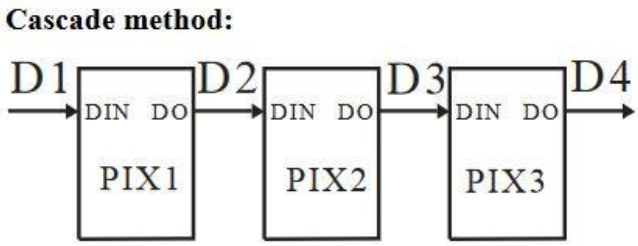

Figura 15: Conexión en cascada de los leds. Fuente: [17]

Cada dato está compuesto por una secuencia de pulsos electricos (bits), en total 24 pulsos por dato. De estos 24 bits, se dividen 8 para cada color (Rojo, Verde, Azul). Estos siguen el orden G-R-B (Verde – Rojo - Azul), desde el MSB hasta el LSB de los 24 bits de datos (Ver *Figura 16*).

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

Figura 16: Distribución de colores en los bits de cada dato enviado a un led.

Cada led toma de la línea serie de entrada (DIN) los datos que lo configuran (24 bits). El resto de datos los deja pasar por su línea serie de salida. Si el tiempo que hay entre dato y dato es mayor de 50 µs, la matriz de LEDs entiende que tiene que reprogramarse de nuevo (Ver *Figura 17*). Por tanto, comenzará a reprogramarse con los siguientes datos que le lleguen.

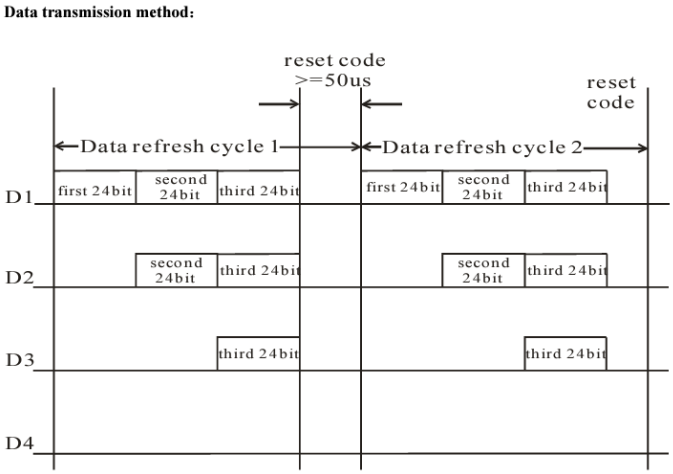

Figura 17: Modo de transmisión de datos entre los leds. Fuente: [17]

Los bits estarán codificados de cierta forma para que la transmisión en serie sea fiable. Siendo necesario mantener los pulsos que componen los datos un determinado tiempo en alto (VCC) y un determinado tiempo en bajo (GND). Este tiempo es de 400 ns en alto y 850 ns en bajo para el '0' lógico, y 800 ns en alto y 450 ns en bajo para el '1' lógico. Y como se ha mencionado anteriormente un tiempo en bajo mayor de 50 µs se traduciría en un reset (Ver *Figura 18*).

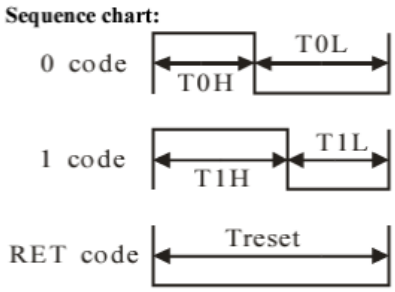

Figura 18: Tiempos de codificación de lógica. Fuente: [17]

Al tener que ser tan meticulosos con el tiempo, el hardware necesario para enviar cada uno de los datos que le llegan a los leds, se diseñará con la lógica programable del SoC, comprobando correctamente a través de una simulación que en el diseño en VHDL se cumplen los tiempos de transmisión y no ocurren fallos al intentar encender losLEDs.

2.2.3.2 Teclado KYPD

Este teclado (*Figura 19*) será el medio de comunicación del usuario con la placa que utilizaremos en la pequeña aplicación que se explica con más detalle en próximos capítulos.

Figura 19: Teclado PmodKYPD. Fuente: digilent.com

El “Keypad” consta de una interfaz Pmod compuesta por 12 pines. 8 ellos estarán dedicados a E/S de datos y los otros 4 sobrantes a alimentación (2 pines para Vcc) y tierra (2 pines para GND). Los botones del teclado estan conectados en forma de matriz a los pines de datos tal y como muestra la *Figura 20*.

Para controlar el teclado desde la PYNQ, se ha usado un IP Core, proporcionado por una de las librerías de Vivado enfocada al control de este dispositivo desde el puerto Pmod. Esta librería además contiene un programa de ejemplo en C, para poder leer correctamente los botones que se pulsan.

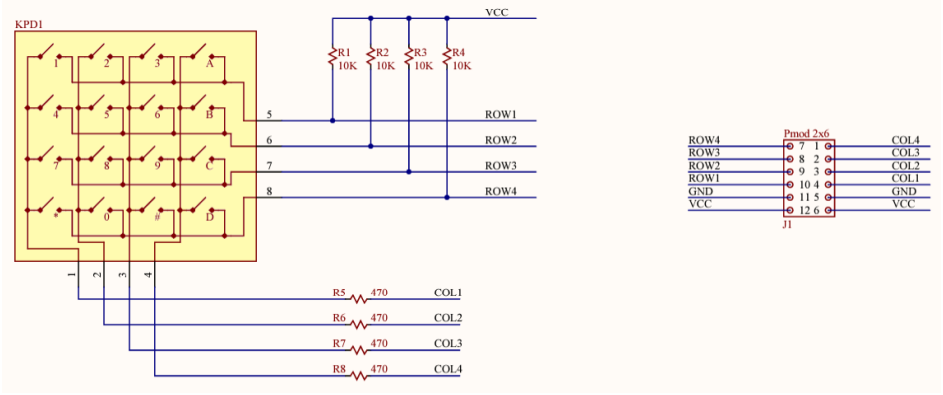

Figura 20: Circuito interno PmodKYPD. Fuente: digilent.com

3 SOFTWARE EMPLEADO

En este capítulo, se hace una breve introducción del software que se ha usado en el proyecto, para la programación de la placa de desarrollo. Se da una breve explicación sobre el manejo de las herramientas Vivado y Vitis, que proporciona Xilinx. Se describen los pasos fundamentales para realizar un proyecto en Vivado, crear bloques personalizados, interconectar todos los elementos necesarios entre sí y generar los archivos necesarios para la programación de la placa de desarrollo que se desee usar. Con respecto a Vitis, se hablará sobre la creación de la plataforma hardware, la creación del proyecto de aplicación, la depuración de esta aplicación, y la creación de la imagen de arranque del sistema.

3.1 Creación de un proyecto genérico en Vivado.

En este apartado, vamos a desarrollar los pasos pertinentes para crear un diseño de sistema hardware en Vivado, añadiendo adicionalmente la creación de un *núcleo IP* personalizado, que destinaremos al control de la matriz de leds.

Para crear un proyecto de Vivado, hay que seleccionar la opción **File** → **Project** → **New…** (*Figura 21*). Una vez hecho esto, aparecerá el asistente de creación de proyecto de Vivado, donde se le dará un nombre al proyecto, y se podrán configurar algunos pequeños ajustes iniciales como el lenguaje de descripción que se va a emplear, VHDL en este caso, y el tipo de proyecto. Se elegirá **RTL Project,** que permite crear o añadir un diseño de bloques.

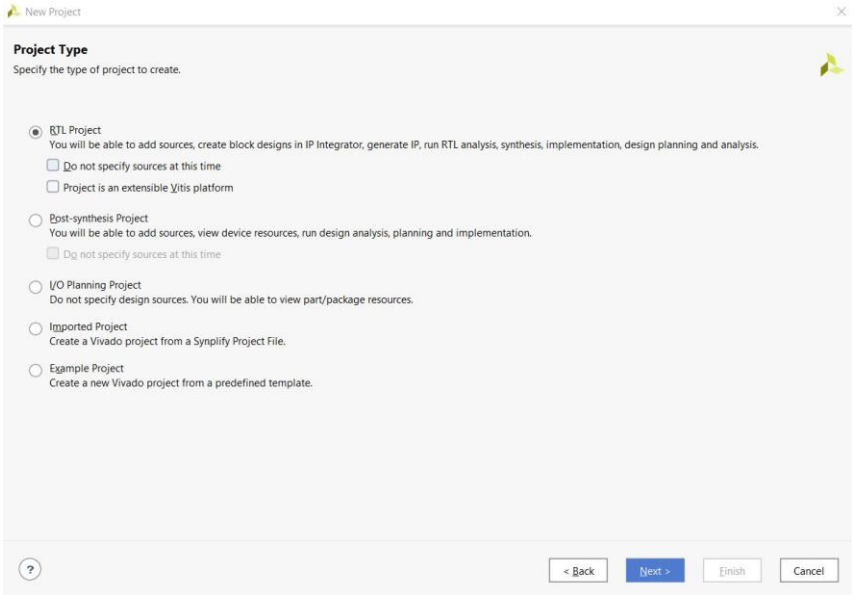

Figura 21: Elección de tipo de proyecto en Vivado.

En las siguientes pestañas del asistente será posible añadir algún diseño existente y el fichero de restricciones de pines.

En la última pestaña se selecciona la pestaña **Boards** (*Figura 22*), y será momento de elegir la placa con la que se va a trabajar. En el caso de estudio, buscaremos la placa "pynq-z2". Solo quedaría pulsar en finalizar y ya estaría creado el proyecto.

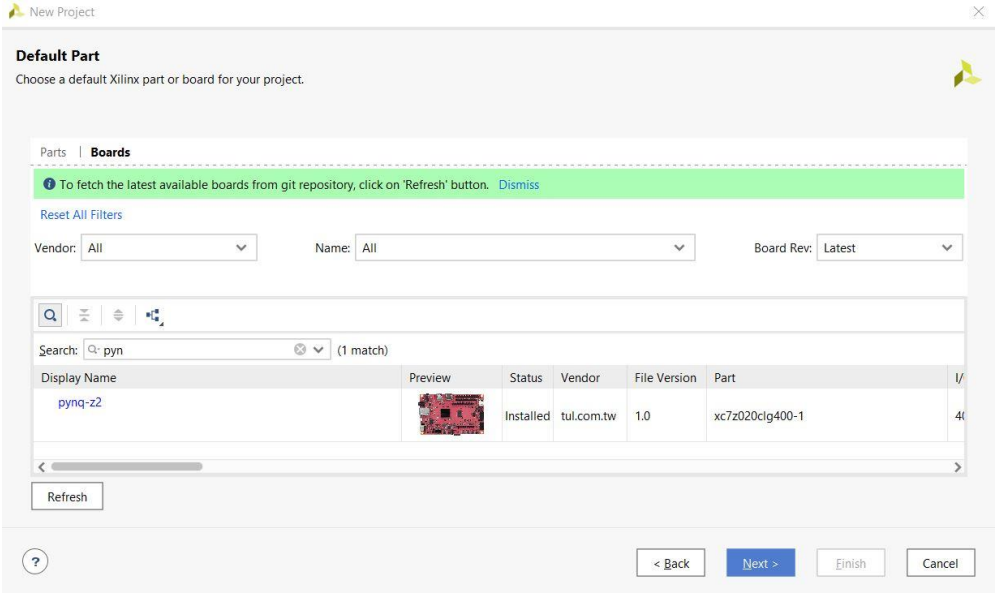

Figura 22: Selección de placa en Vivado.

3.2 Creación de un IP Core personalizado.

Ahora sería conveniente crear y configurar los nuevos *núcleos IP* que se incorporen al diseño para así tener todos los componentes disponibles una vez se construya el diagrama de bloques. Para ello se seleccionará la opción **Tools → Create and Package New IP…** y aparecerá el asistente de creación de *núcleos IP.* Una vez en el asistente, se seleccionará **Create a New AXI4 peripheral.**

Después de esto, se asigna el nombre, descripción y ruta para almacenar la IP, y se pasa a la siguiente pantalla de configuración donde se podrán configurar las características del *núcleo,* así como su modo de interfaz (Maestro o esclavo) y su número de registros.

Una vez creado el *Core*, solo habría que añadir el código de la descripción hardware a añadir (**Add sources → Add or create design**). De esta manera, se puede crear un IP Core basado en un diseño hardware ya implementado a traves de su fichero ".vhd", o se puede optar por crear uno nuevo desde cero, escribiendo el código necesario. Sin embargo, aunque ya se tenga un diseño previo, habrá que hacer ligeras modificaciones para poder conectarlo adecuadamente con los registros internos del SoC creados e instanciarlo adecuadamente en el "*Top Module*" de la IP. El diseño introducido, será tratado para la IP, como si fuese un componente en un diseño VHDL normal que hay

que instanciar.

Para Comprobar si se ha instanciado correctamente el diseño y esta correctamente conectado con el Top Module, es necesario realizar una síntesis de todo el conjunto y una implementación y comprobar que no aparece ningún error. Esto se hace en el menú izquierdo de Vivado: **Run Synthesis,** y posteriormente, **Run Implementation**

Si todo ha salido bien, el último paso final tras la instanciación del componente es el de confirmar los cambios en la ventana **Package IP** del proyecto de Vivado y seleccionar la opción **Merge changes**, tras esto se termina de empaquetar la IP mediante la opción **Re-Package IP (***Figura 23***)** y se guardan todos los cambios realizados.

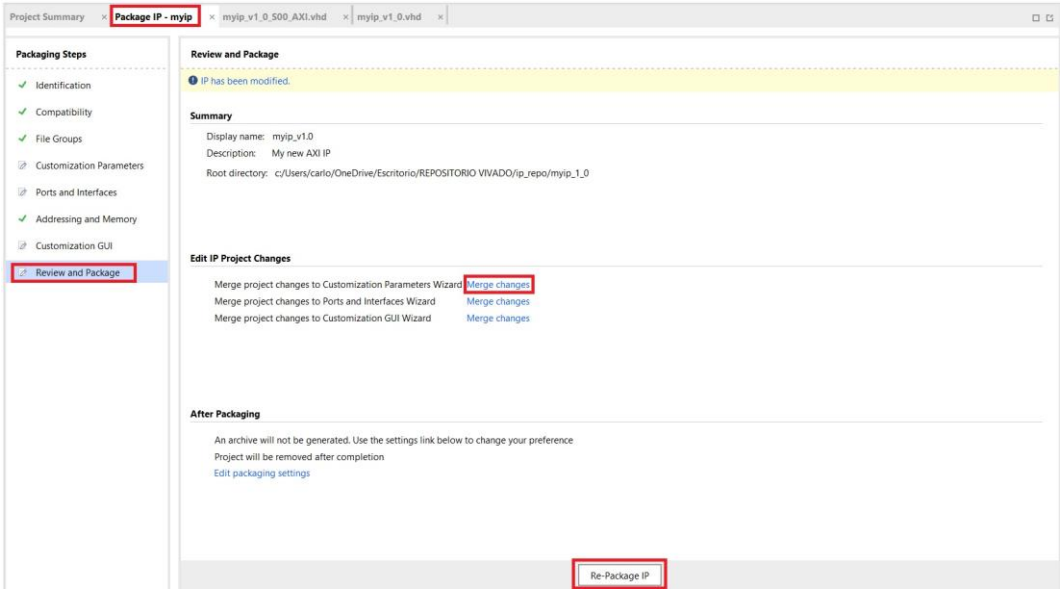

Figura 23: Empaquetamiento de bloque IP personalizado en Vivado.

Antes de pasar al siguiente apartado, hay que vincular el *núcleo* creado para que aparezca en el buscador de Vivado. Para ello, en la pestaña **Settings,** a la izquierda en el navegador de Vivado, hay que acceder a la pestaña **IP → Repository,** y añadir la carpeta donde se ha guardado el componente.

3.3 Creación del diagrama de bloques.

Ahora hay que crear un diseño de bloques y añadir todos los componentes que intervendrán en el diseño deseado. Para ello hay que crear el diagrama de bloques pinchando en el desplegable **IP INTEGRATOR → Create Block Design.** Aparecerá una ventana para nombrar el diseño y establecer su ubicación y posteriormente una página en blanco en la que añadir cada bloque IP. Habrá que usar el icono “+” de la barra superior e introducir los nombres de los *núcleos* que se van a utilizar. Es importante añadir el procesador del SoC, si

se desea crear un programa de aplicación. En este caso, sería añadir el componente “ZYNQ7 Processing System”.

Las librerías integradas de Vivado permiten una forma más sencilla de añadir componentes. Basta con pinchar en la pestaña **Board (***Figura 24***)**, y ahí aparecerán un conjunto de bloques IP asociados a algunos de los periféricos de la placa elegida para el proyecto. Algunos de estos periféricos no tienen ningún IP Core definido en las librerías predeterminadas, pero se puede descargar la librería *Vivado-library-master* desde la web de Digilent (https://digilent.com/reference/vivado/library). En esa página se pueden encontrar, entre otras, las librerías para los puertos Pmod, que serán de gran ayuda en el diseño del proyecto.

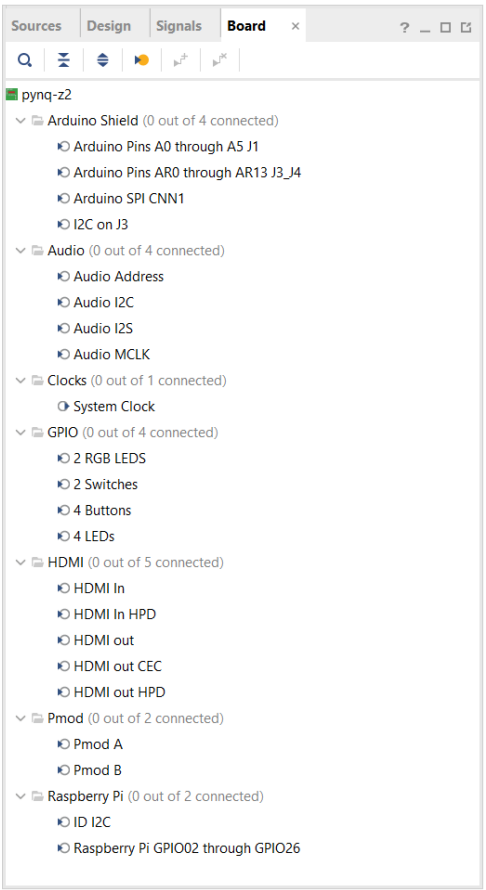

Figura 24: Contenido de la pestaña “Board” en Vivado.

Al añadir todos los componentes, hay que interconectarlos, y esto se hace a través de las opciones **Run Block Automation** y **Run Connection Automation** (*Figura 25*)**.** La primera opción se dedica a crear un sistema básico añadiendo componentes o E/S que faltan para un diseño completo. La segunda opción conecta todos los componentes adecuadamente entre sí, excepto las E/S externas, esas hay que conectarlas manualmente.

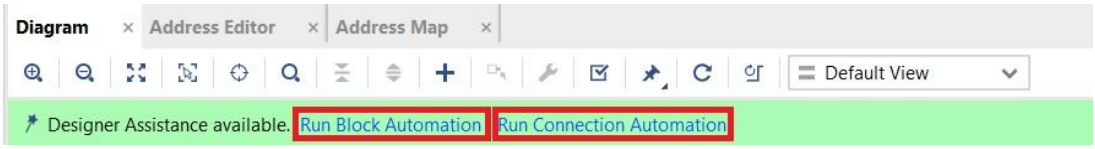

Figura 25: Opciones para interconexionar elementos del diagrama de bloques.

3.4 Generación del Bitstream.

Para crear el bitstream es necesario, en primer lugar, integrar el diagrama de bloques en el nivel más alto de la jerarquía del diseño. Para ello hay que generar el *HDL Wrapper* (Clic derecho en el diagrama → **Create HDL Wrapper.**

Una vez creado el HDL Wrapper ya es posible generar el Bitstream, que es un archivo binario que contiene la configuración hardware espécifica que se ha implementado). Es importante añadir el **fichero de restricciones de pines** antes de generar el Bitstream (**Sources → Add sources → Add or create constraints**). Este archivo se puede crear desde cero, aunque es mucho más fácil buscar una plantilla acorde a nuestra placa de desarrollo, que contenga todas las E/S físicas de la placa.

Con la opción **Generate Bitstream** en menú izquierdo de Vivado, se realizará una **síntesis del diseño**, comprobando que no hay errores de sintaxis, problemas de conectividad, de temporización, etc. Seguidamente se realiza una **implementación** del diseño ya sintetizado, y finalmente se genera el Bitstream.

3.5 Exportación de la plataforma para Vitis.

Tal y como se comentó en el apartado ***Metodología de diseño y desarrollo***, para poder comenzar a desarrollar una aplicación software para el diseño realizado en Vivado, es necesario exportar un archivo que contenga las configuraciones relativas al procesador y sus componentes y que sea interpretable por Vitis. Este fichero tiene extensión ".xsa" y para exportarlo hay que realizar los siguientes pasos:

- El primer paso es haber realizado la implementación del diseño y la generación del Biststream
- El segundo paso consiste en generar las salidas del diseño, haciendo click derecho en el diagrama de bloques en la pestaña **Sources** y ejecutando **Generate Output Products.** Aparecerá un asistente similar a los anteriores y con hacer click en **Generate** es suficiente, se dejará todo tal y como estaba predefinido.

- El último paso consiste en exportar el hardware (**File** → **Export** → **Export Hardware…**) y en el asistente (*Figura 26*) tendremos dos modos de exportación, pre-sintesis y post-implementación (este último, incluye el bitstream dentro del archivo ".xsa" que se va a generar).

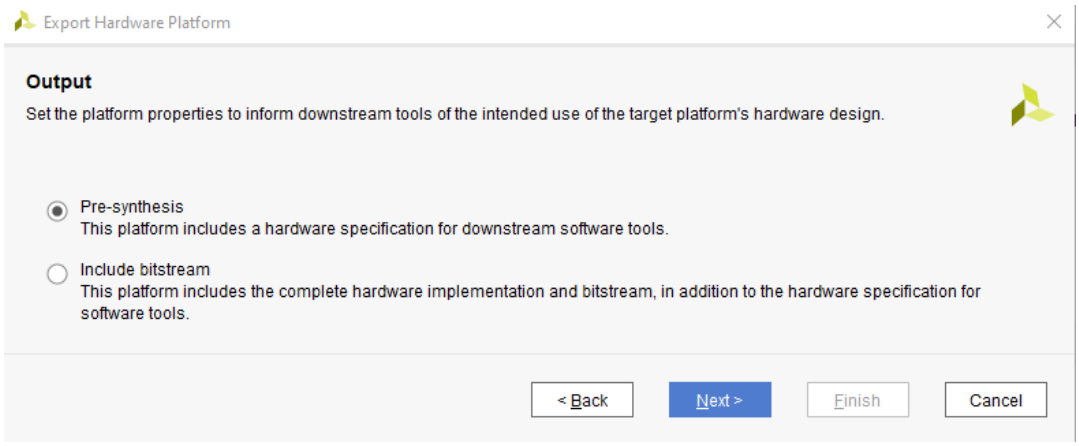

Figura 26: Modos de exportación de la plataforma.

3.6 Creación de una Plataforma en Vitis

Como ya se ha mencionado, la plataforma serán los cimientos de nuestro sistema. Para crearla, una vez arrancado Vitis se ejecutará **File → New → Platform Project…**

Aparecerá un asistente como el de la *Figura 27*:

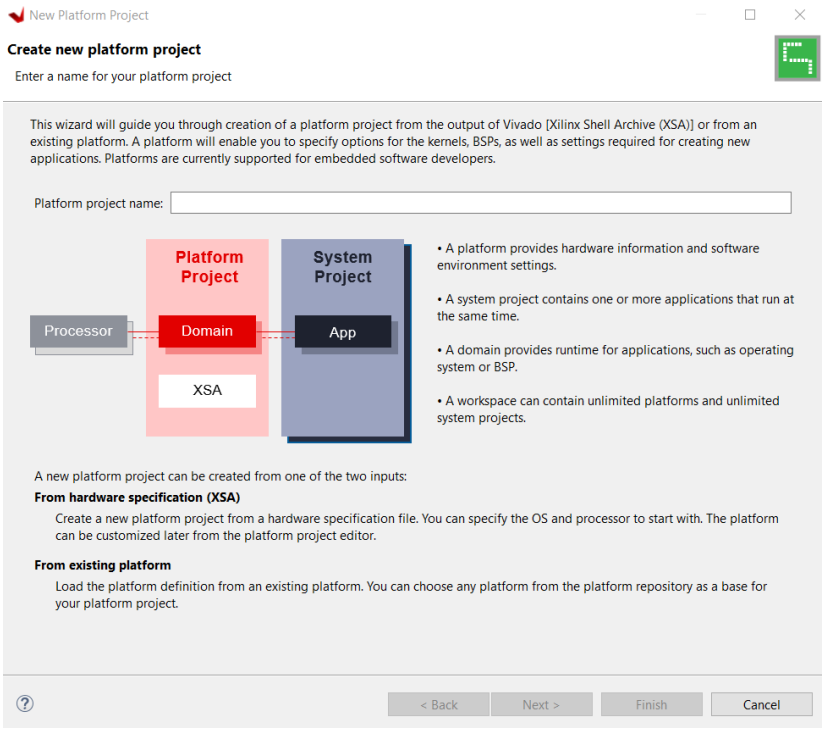

Figura 27: Asistente de creación de plataforma en Vitis.

Tenemos dos formas de crear una nueva plataforma; a partir de una ya existente o a partir del **archivo XSA**. En este estudio, se hará mediante segunda opción. Para ello, en primer lugar hay que añadir un nombre a la plataforma hardware creada y ya será posible pasar a la siguiente página.

En la siguiente página *(Figura 28)* se elegirá la pestaña **Create a new platform from hardware (XSA)** y se hará clic en **Browse…** Aquí se seleccionará el fichero ".xsa" exportado en Vivado.

En la siguiente página del asistente, hay que crear el *dominio* inicial de la plataforma (una plataforma puede tener dominios ilimitados), que está compuesto por el sistema operativo y el procesador. Un dominio tiene un *BSP* (*Board Support Package*) asociado, que es un conjunto de software que se adapta a la plataforma hardware y contiene los controladores y archivos necesarios para que se pueda ejecutar el sistema operativo (aplicación) en esa plataforma. Es una especie de interfaz entre el hardware y el software.

Durante la creación del dominio, para la placa Pynq-Z2 en la opción “Processor” se elegirá “ps7_cortexa9_0”, aunque el procesador correspondiente saldrá predeterminadamente al importar el fichero “.xsa”. A su vez, hay que elegir el sistema operativo del sistema, entre los que se pueden distinguir:

- Standalone. Esto significa que no hay sistema operativo, es decir, la aplicación es autónoma y no necesita nada más para ponerse en marcha.
- Freertos10_xilinx. Se trata de un sistema operativo de código abierto en tiempo real. Muy útil en este campo, sobre todo para aceleración de aplicaciones.
- Linux. Sistema operativo de código abierto. Este está enfocado a tareas de propósito general, a diferencia de FreeRTOS, que ha sido diseñado específicamente para sistemas embebidos y microcontroladores.

Una vez se han elegido estos parámetros, se clicará en **Finish**, y el proyecto de plataforma será creado.

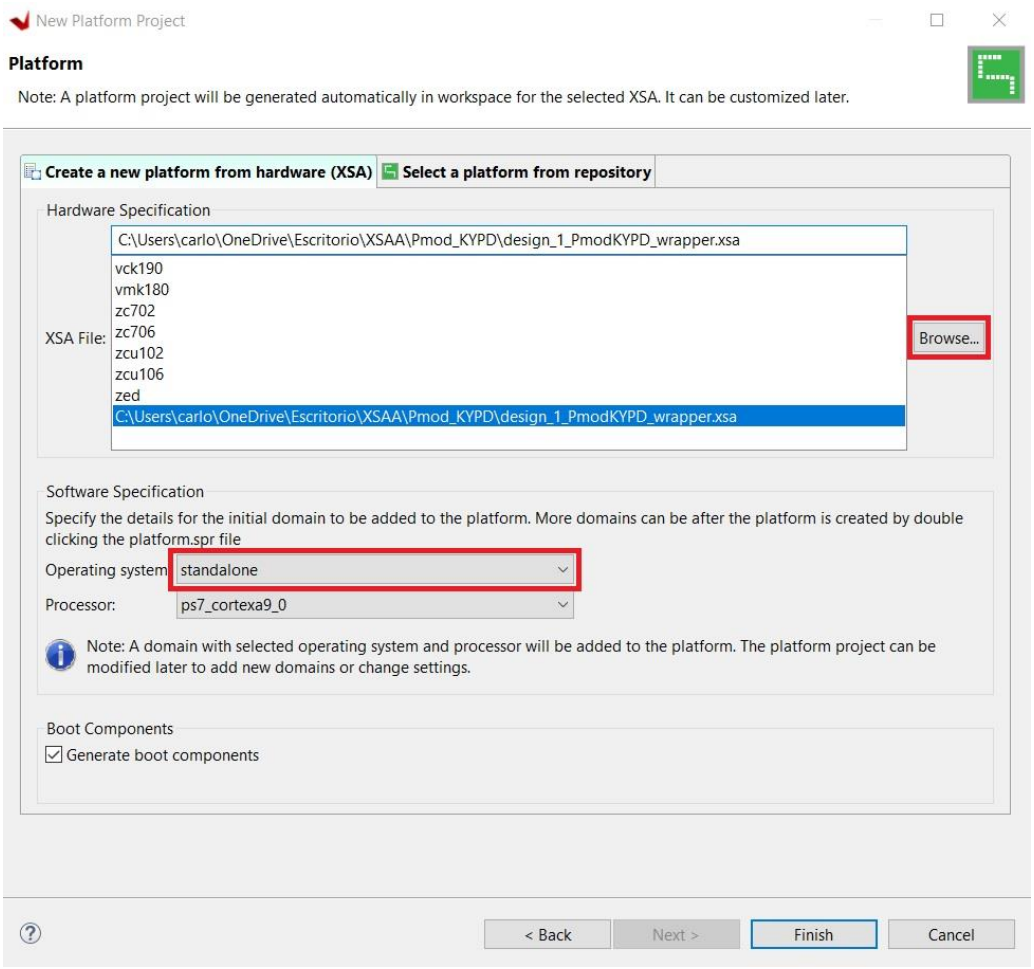

Figura 28: Elección de plataforma y de Sistema Operativo para la misma.

Para generar correctamente la plataforma, hay que compilar el proyecto. Es muy común que salgan a la luz errores al compilar el proyecto, especialmente, si en el diseño de Vivado se ha añadido al menos un bloque IP personalizado. El fallo está en que Vitis no genera correctamente los *Makefiles* (este error y la forma de corregirlo se explicará en el apartado *3.6.1*).

Una vez el proyecto haya compilado sin errores, la plataforma habrá sido generada correctamente. Si la plataforma no ha sido generada correctamente, se podrá saber rápidamente observando el nombre de la plataforma en el *Explorer* (Menú superior izquierdo), donde además del nombre, aparecerá "*(Out-of-date)*" a la derecha y un símbolo ( ) si hubiese un error.

3.6.1 Error al generar Makefiles

Un Makefile es un archivo de configuración que se incluye en la carpeta raíz de un proyecto. Este archivo contiene instrucciones sobre como compilar y vincular un conjunto de archivos de código fuente. Un programa (*Make*) se encarga de llamar a un compilador, un vinculador y otros programas para crear un archivo ejecutable.

El error al generar un Makefile es uno de los fallos más comunes de Vitis. Para solucionarlo, habrá que editar todos los makefiles que estén dentro de las carpetas con el nombre de la/s IP personalizada/s creada/s en Vivado. Normalmente se suelen crear 3 makefiles en estos 3 directorios:

- `<Nombre_de_la_plataforma>/ps7_cortexa9_0/<Nombre_del_sistema_operativo_elegido>/bsp/ps7_cortexa9_0/libsrc/>Nombre_de_la_IP_personalizada>/src/Makefile`
- `<Nombre_de_la_plataforma>/zynq_fsbl/zynq_fsbl_bsp/ps7_cortexa9_0/libsrc/<Nombre_de_la_IP personalizada>/src/Makefile`
- `<Nombre_de_la_plataforma>/hw/drivers/<Nombre_de_la_IP_personalizada>/src/Makefile`

Estos 3 archivos, habrá que abrirlos y borrar completamente su contenido y copiar el siguiente (*Figura 29*), quedando de esta forma:

```
COMPILER=
ARCHIVER=
CP=cp
COMPILER_FLAGS=
EXTRA_COMPILER_FLAGS=
LIB=libxil.a

RELEASEDIR=../../../lib
INCLUDEDIR=../../../include
INCLUDES=-I./. -I${INCLUDEDIR}

INCLUDEFILES=$(wildcard *.h)
LIBSOURCES=$(wildcard *.c)
OUTS=$(wildcard *.o)
OBJECTS = $(addsuffix .o, $(basename $(wildcard *.c)))
ASSEMBLY_OBJECTS = $(addsuffix .o, $(basename $(wildcard *.S)))

libs:
    echo "<CustomIP_name>..."
    $(COMPILER) $(COMPILER_FLAGS) $(EXTRA_COMPILER_FLAGS) $(INCLUDES) $(LIBSOURCES)
    $(ARCHIVER) -r ${RELEASEDIR}/${LIB} ${OBJECTS} ${ASSEMBLY_OBJECTS}
    make clean

include:
    ${CP} $(INCLUDEFILES) $(INCLUDEDIR)

clean:
    rm -rf ${OBJECTS} ${ASSEMBLY_OBJECTS}

```

Figura 29: Corrección de código en makefiles.

3.7 Creación de un Proyecto de aplicación

Este proyecto de aplicación, contendrá toda la funcionalidad software que se desea cargar en la placa y que ejecutará el procesador. La aplicación se programará en C o C++.

Para crear el proyecto de aplicación hay que ejecutar **File → New → Application**

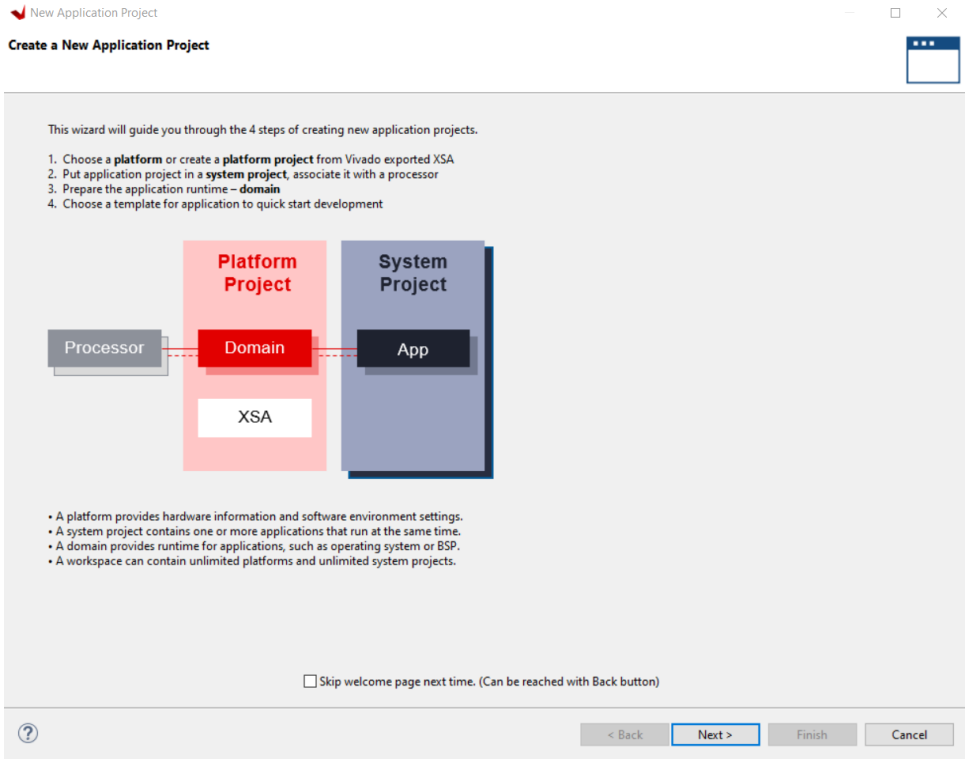

Figura 30: Asistente de creación de proyecto de aplicación en Vitis.

Project. Aparecerá un asistente (*Figura 30*), en la primera página aparecerá un resumen de los pasos a seguir y para qué sirve cada elemento.

En la siguiente página (*Figura 31*), se deberá elegir la plataforma base para la aplicación que se va a crear. Se elegirá la pestaña **Select a platform from repository,** si se han seguido los pasos anteriores, y se elegirá la plataforma

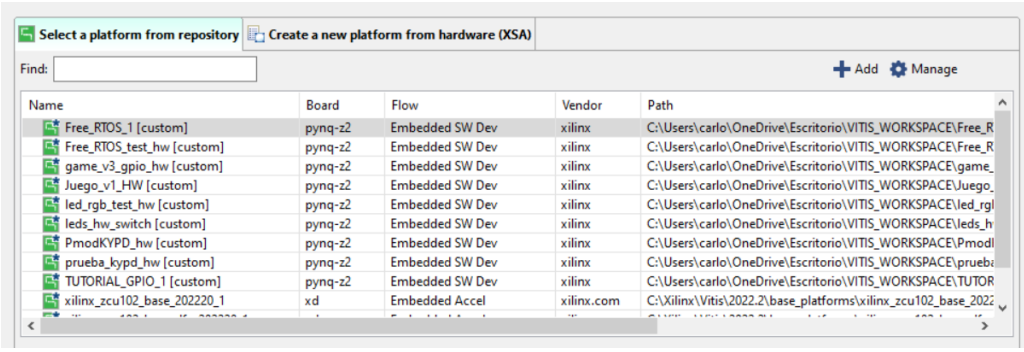

Figura 31: Selección de plataforma para el proyecto de aplicación.

creada.

En la siguiente página habrá que añadir un nombre para la aplicación y elegir el procesador que se va a usar en la aplicación. Al nombrar la aplicación, automáticamente se nombrará también al sistema software que engloba esta aplicación (el nombre será el mismo, pero con el sufijo "_system").

Se pulsa **Next,** y se pasará a la página de selección de *dominio* (*Figura 32*). Aquí se puede crear también un nuevo dominio, o elegir el creado anteriormente. Lo normal si se ha creado un domino anteriormente es elegir ese mismo dominio.

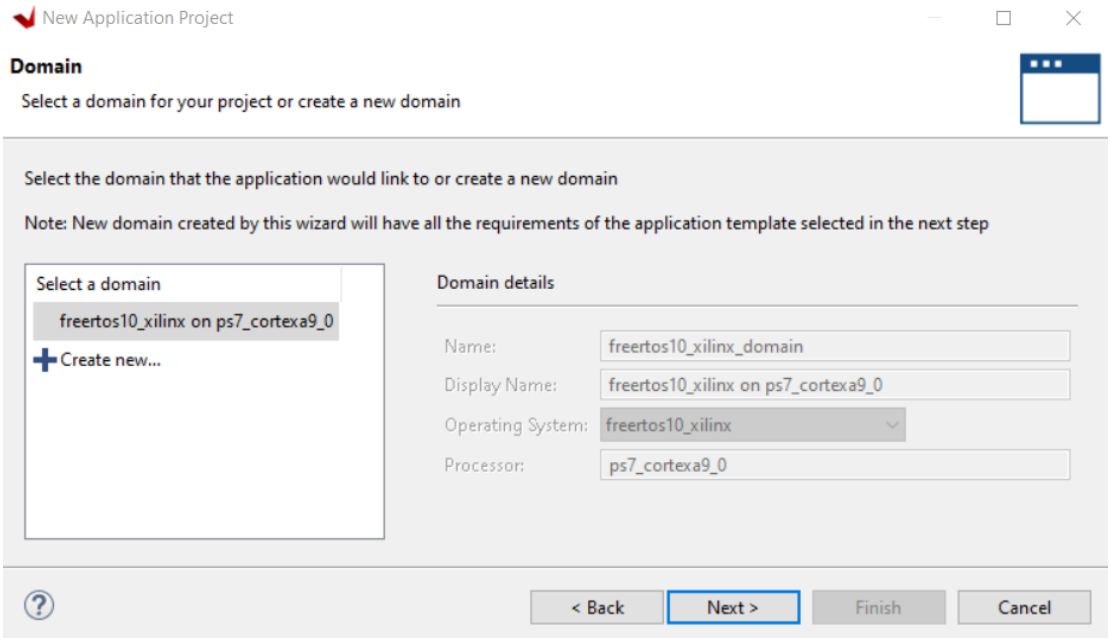

Figura 32: Página de selección de dominio.

Se pulsa **Next,** y aparecerá la última página en la que se podrá seleccionar una plantilla ya predefinida por Vitis para la aplicación, o una hoja en blanco (en C o C++). Para desarrolladores con poca experiencia es recomendable elegir la plantilla "Hello World", ya que contendrá funciones básicas para el correcto funcionamiento de una aplicación, sobre todo, si es una aplicación con un sistema operativo FreeRTOS, ya que aquí hay que configurar más parámetros para iniciar la aplicación, así como creación de tareas y manejadores.

Finalmente, se pulsa en **Finish** y se generará automáticamente el sistema software del proyecto. Ahora es momento de desarrollar la aplicación.

3.8 Depuración de una aplicación

Una vez creado un prototipo para la aplicación, habría que compilarlo y ver que la aplicación no tiene errores. Cuando esto ocurra ya se podrá depurar la aplicación para comprobar su correcto funcionamiento. Para ello, hay que hacer

clic derecho sobre el nombre o el icono de la aplicación ( ) y seleccionar la opción **Debug As → Debug Configurations…** Se abrirá un asistente, en el que habrá que seleccionar en el menú izquierdo: "**Single Application Debug**". Posteriormente aparecerán varias pestañas en el asistente (*Figura 33*).

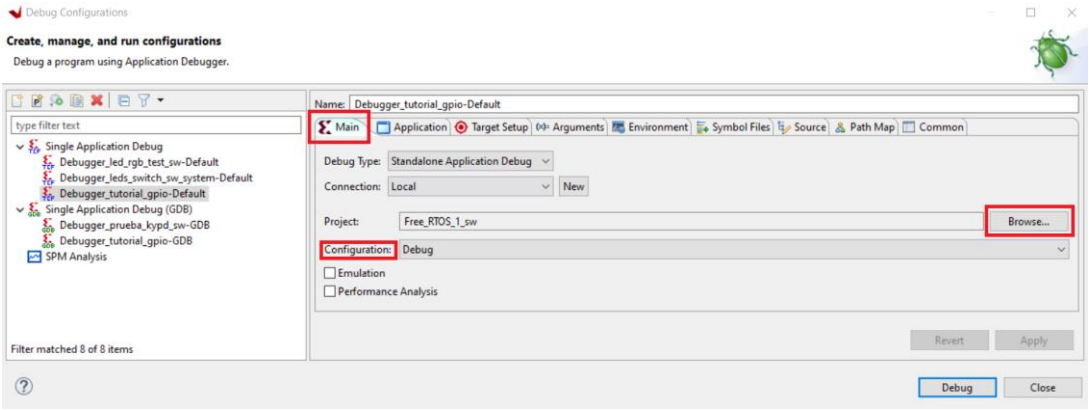

Figura 33: Asistente para la configuración de la depuración en Vitis.

Para hacer una depuración normal, sin complejidad, solo hay que clicar en **Browse…**, en la pestaña **Main** y se elige la aplicación a depurar.

En la opción **Configuration,** se mantiene la opción **Debug.**

Despues hay que clicar en **Debug**, y el entorno Vitis pasará al modo depuración (*Figura 34*).

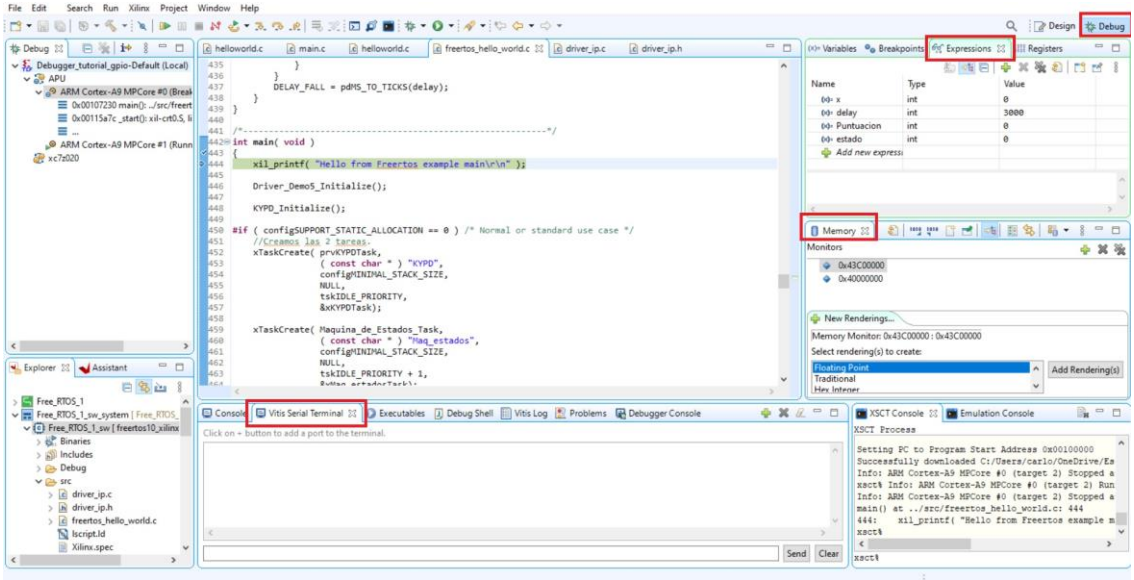

Figura 34: Modo depuración.

Durante la depuración se pueden utilizar diferentes opciones de monitorización:

- Se puede arrancar un terminal serie desde Vitis, para comprobar mensajes que se transmiten en tiempo real con la función **"xil_printf();".** Para abrir un terminal, basta con ir a la pestaña **Vitis Serial Terminal** (*Figura 34*)**,** en la ventana inferior del entorno de depuración y hacer clic en **Connect to serial port (icono).** Aparecerá una ventana en la que podremos configurar la comunicación serie. Para ello hay que asegurarse de el puerto COM que se ha habilitado en el PC y seleccionarlo. También hay que configurar otros parámetros como la velocidad de transmisión, tamaño de los datos, etc. En la *Figura 35* se muestra la configuración para la placa del proyecto.

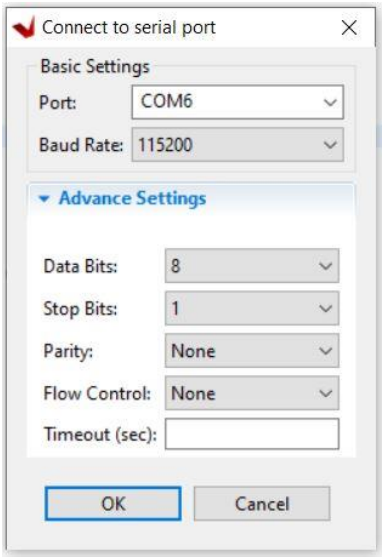

Figura 35: Configuración del puerto serie en modo depuración.

- También se puede acceder al contenido de los registros internos que comunican la FPGA con el procesador a través de la ventana **Memory**. Para ello, en la ventana **Memory** se añade una nueva dirección de memoria a mapear (icono) y aparecerá una pestaña abajo llamada **New Renderings…** (*Figura 36*) donde habrá que seleccionar el formato numérico que se desea para visualizar los datos. En la librería **"xparameters.h"**, generada por Vitis, aparecerán las direcciones de memoria correspondientes al registro base de cada bloque IP añadido en el diseño hardware. Estas direcciones aparecerán nombradas como: "XPAR_*NOMBRE DEL BLOQUE IP*_AXI_BASEADDR". Cada dirección de memoria tiene una capacidad de 32 bits, es decir, 4 bytes. Por tanto, la dirección siguiente a una dada será la dirección dada + 4. Por ejemplo: Dirección base 0x00000000, la siguiente seria 0x00000004 y la siguiente a esta 0x00000008.

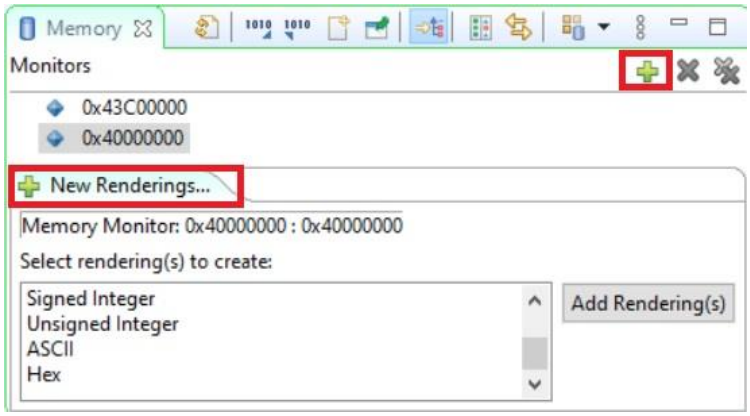

Figura 36: Control de contenido de direcciones de memoria en depuración.

- Otra de las funcionalidades es que podemos añadir cualquier variable creada para conocer su valor. Esto se puede revisar en la ventana **Expressions** (*Figura 34*)**.** El entorno también dispone de otras pestañas adicionales para ver los puntos de ruptura que se han asignado, el valor de los registros internos del procesador y las variables del sistema.

3.9 Ensamblaje de todo el sistema y creación de la imagen de arranque

Una vez construida la aplicación y depurada comprobando que le funcionamiento se ajusta a los requisitos, solo quedaría montarla en una imagen de arranque para que la placa pueda usar la aplicación sin necesidad de estar conectada al PC.

El primer paso sería compilar todo el sistema software (o proyecto de aplicación) y ver que no aparece ningún error. Para ello, se hace clic derecho en el sistema software (icono) y **Build Project.**

Cuando todo este correctamente compilado ya será posible generar una imagen de arranque. Esto se consigue seleccionando el proyecto de aplicación en la ventana **Explorer** y posteriormente se hace clic derecho en el proyecto de aplicación y se selecciona **Create Boot Image.** Se abrirá una ventana donde habrá que especificar como será la imagen (particiones, autentificación, encriptación…).

4 DISEÑOS PREVIOS A LA APLICACIÓN

En este capítulo se describen los diseños creados previamente antes de realizar la aplicación. Estos diseños se han realizado con la finalidad de entender y comprobar el funcionamiento de los componentes electrónicos del proyecto, así como todo el proceso que conlleva la programación de la placa de desarrollo

Principalmente se van a describir tres diseños:

- El primero de ellos se ha usado para controlar en encendido de la matriz de LEDs.
- El segundo diseño complementa al primero y tiene la finalidad de conocer el mapeo de los LEDs físicamente y ver en qué orden se encienden.
- El tercer diseño irá destinado a poner a prueba el flujo de diseño y programación de la placa de desarrollo junto con la matriz de leds.
- El cuarto diseño incorpora el teclado KYPD y comprueba el funcionamiento de todas sus teclas.

4.1 Diseño Hardware para el control automático de la matriz de leds: LEDS_RGB_32X8.vhd

En este apartado se explica cómo se ha realizado el diseño hardware para controlar el encendido de la matriz de leds. El diseño se ha realizado siguiendo la carta ASM que se muestra en la *Figura 37*.

El diseño consta de 35 entradas y 1 salida. De estas entradas, 32 son entradas de 32 bits, definida cada una de ellas como **"std_logic_vector(31 downto 0)"**. Estas entradas contienen los colores a poner en los 256 bits de la matriz de leds, aunque para reducir el número de bits necesarios, sólo se permiten 16 posibles colores. De esta forma en estas entradas el color de cada led se indica únicamente con 4 bits. Cada entrada de 32 bits contiene los colores de 8 leds. Las otras tres entradas corresponden a:

- Reset: Al activar reset ('1' lógico) pasaremos al estado inicial del proceso creado en la carta ASM y se inicializarán todas las variables.
- Init: sirve para arrancar la máquina de estados cuando este en alto ('1' lógico) y comenzar a enviar datos por la salida "dato_rgb" para cargar

los leds.

- Clk: Reloj del sistema, de 100 MHz.

Solo será necesaria una salida, que es “dato_rgb” y que es la entrada de datos de la matriz de leds.

Se utiliza un multiplexor 16 a 1 para cargar en la señal “color_act” el valor los 24 bits del color del led que se está cargando en cada momento. La relación de los 4 bits con el color que se carga se muestra en el ejemplo: “IMPLEMENTACIÓN DEL MUX 16 a 1”. De las 16 entradas del MUX, solo se usarán 9. Las 7 entradas restantes serán repetidas y conectarán con el estado *apagado.*

IMPLEMENTACIÓN DEL MUX 16 a 1:

```
    with color_act_4bits select
color_act <= apagar   when "0000",
       red      when "0001",
       green    when "0010",
       blue     when "0011",
       magenta  when "0100",
       yellow   when "0101",
       cyan     when "0110",
       gray     when "0111",
       white    when "1000",
       apagar   when others;
```

El diseño tiene además las siguientes señales internas:

color_act	Señal que almacena los 24 bits correspondientes al color. Esta variable será la salida del decodificador.
cont_v	Contador que temporiza el tiempo que mantenemos la salida a ‘1’ y a ‘0’ para transmitir un bit de datos.
cont_bits (integer)	Contador que indica el número del bit que se está transmitiendo, de los 24 necesarios para completar un dato de un led.
cont_led (integer)	Contador que indica el número de led que se está cargando de los 8 que hay en un registro.
cont_reg (integer)	Contador que indica el número de registro que se está cargando de los 32 registros que hay.

color_act_4bits	Señal que almacena el color codificado correspondiente a un led. Esta señal será de 4 bits.
color_act_8bits	Señal que almacena los 32 bits del registro que se está cargando actualmente.

Tabla 3: Señales del diseño LEDS_RGB_32X8.

A continuación, se explica brevemente el funcionamiento de la máquina de estados. Se ha diseñado con 5 estados, y son los siguientes:

- **Estado *s0***: Es el estado inicial. Este estado estará en continua espera y cuando detecte un '1' lógico en la entrada *init* se inicializarán todas las señales y se pasará al siguiente estado, *s1*.
- **Estado *s1***: Como se mencionó en el capítulo *2.2.3.1*, para cargar un bit en cada led, se necesita tener la salida en alto un determinado tiempo y en bajo otro determinado tiempo (según tabla tiempos). En este estado se pone la salida a '1' durante 400 ns u 800 ns según el bit que vayamos a transmitir ('0' o '1'). Esto se hará contando los ciclos de reloj que mantenemos a un determinado valor, cada ciclo tiene una duración de 10 ns, por tanto, se necesitarán "40" u "80" ciclos respectivamente. Para ello se utiliza un contador decreciente (*cont_v*), y en este estado se le asigna el valor de inicio de cuenta. Una vez hecho esto se pasará al siguiente estado, *s2*.
- **Estado *s2***: Este estado comenzará decrementando el valor del contador que temporiza el tiempo a "1" de la salida. Una vez el contador finalice su cuenta, se actualizará de nuevo el contador, pero ahora con el tiempo en baja. Este tiempo será de 850 ns o 450 ns según se vaya a transmitir un '0' o '1' lógico respectivamente. Esto será lo mismo que 85 o 45 ciclos de reloj. Una vez hecho esto se pasará al siguiente estado, *s3*.
- **Estado *s3***: En este estado se decrementa el contador como en el estado anterior, hasta que llegue a cero. Al finalizar esta cuenta, ya se habrá transmitido un bit completo por la salida de datos. Ahora lo que faltaría sería transmitir el resto de bits. Para ello se comprueba si se han transmitido todos los bits (*cont_bits*). Si no es así, se tranmite el siguiente bit y para ello hay que volver al estado s1. Si se han transmitidos todos los bits correspondientes a un led, se pasa a cargar los datos del siguiente led (*cont_led*). Si ya se han transmitidos los datos correspondientes a los 8 leds de un registro pasaríamos al siguiente registro (*cont_reg*). Conforme se van transmitiendo nuevos datos, se irán actualizando las variables *color_act_8bits* y *color_act_4bits* según corresponda. Una vez se hayan cargado todos los datos, se va al estado final *sf*.

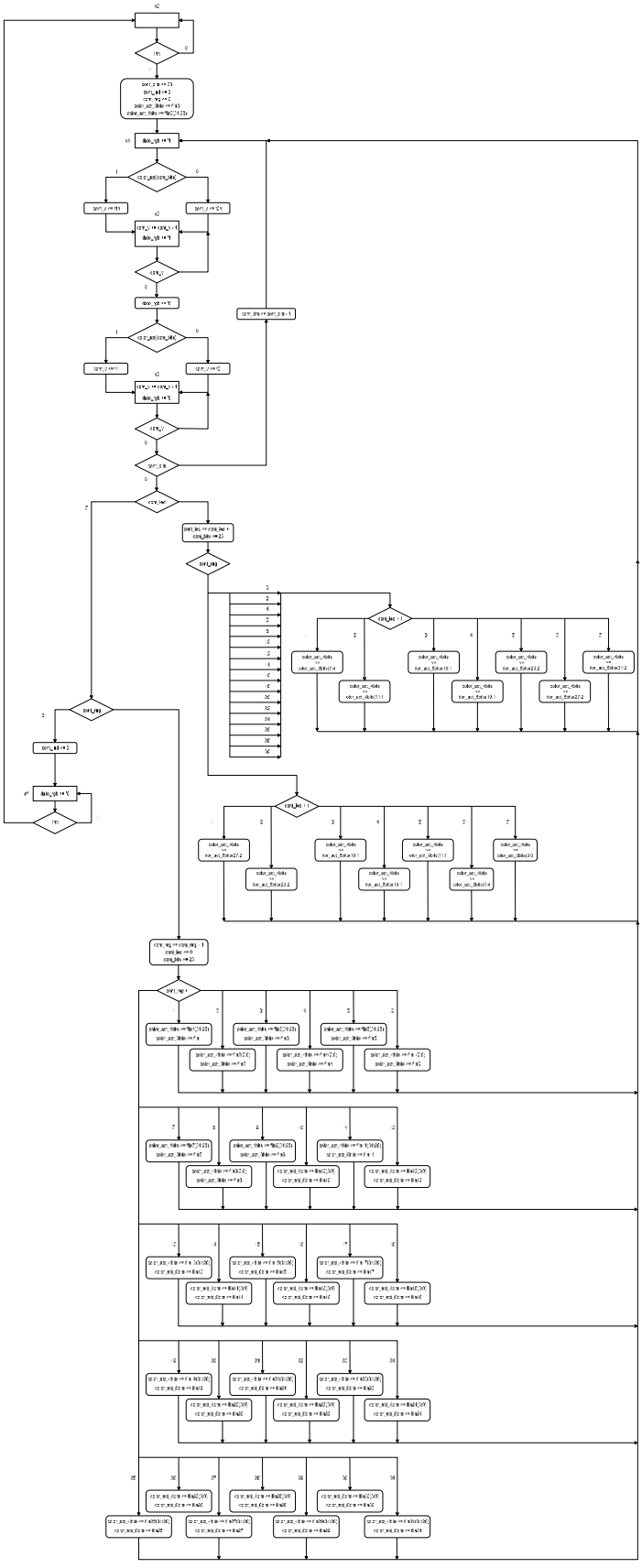

Figura 37: Carta ASM del diseño de encendido de leds.

- **Estado *sf***: En este estado, se mantiene la salida a '0', para indicar el fin de la transmisión. Una vez hecho esto, se pasa al estado inicial. Para asegurar de que el *init* no está puesto a '1' durante un tiempo log provocar un bucle infinito, solo se vuelve al estado inicial si esta entrada está a '0'.

Tras realizar las pruebas en campo sobre la placa, se modificó el estado s3, ya que se observó que los leds de la matriz tenían una conexión en zig-zag a lo largo de su ancho. Por ello hubo que invertir el contenido de ciertos registros como se muestra en la *Figura 38*. Estos registros coinciden con los registros pares, y por tanto se añadió un fragmento de código para corregir esto (esta corrección ya está incluida en la carta ASM de la *Figura 37*. La comprobación experimental de este error se puede ver en la *Figura 45*.

De esta forma, los únicos parámetros que hay que controlar para que se enciendan los leds son los 32 "registros" y las entradas "Init" y "Reset" para dar inicio o anular la transmisión. Las simulaciones realizadas para comprobar el funcionamiento de la placa, se han añadido en el apartado *4.2.1*, ya que antes de eso es necesario explicar el funcionamiento de un diseño complementario (*4.2*).

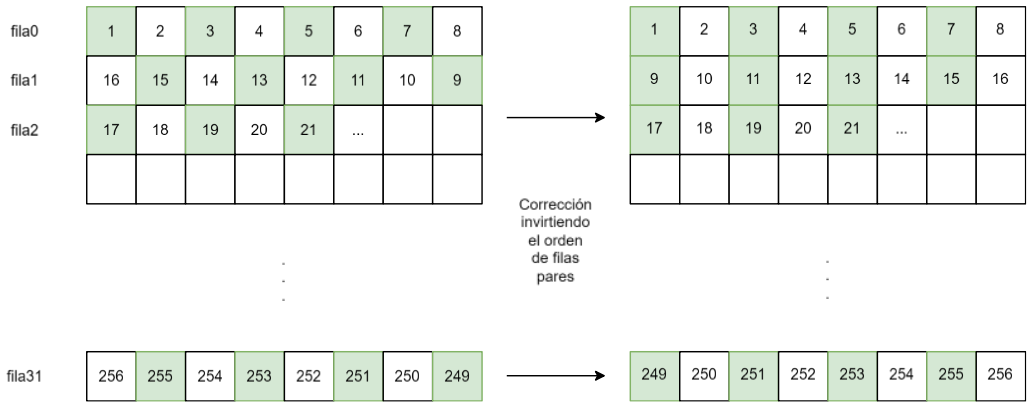

Figura 38: Corrección de orden de leds.

4.2 Diseño complementario para pruebas Físicas: CONTROL_MAIN.vhd

El diseño para comprobar el funcionamiento de los leds, se realizó sobre la placa de desarrollo Nexys4 DDR™ de Digilent. Para poder "simular" los registros en la placa, y poder probar físicamente a través de los switches de la placa que los leds se encienden correctamente, se realizó otro diseño hardware.

Este diseño consta de 4 entradas y una salida. Entre las entradas se hallan:

- clk: Reloj del sistema (100 Mhz)
- reset: Reset del sistema, activo en alta.
- init_config: Entrada para dar la orden para cargar los leds.
- switches: Es un std_logic_vector de 2 bits, que actuarán como una entrada de selección.

Y la salida es:

- dato_rgb: Salida de datos hacia los leds. Esta salida es necesaria, ya que este diseño será el “Top module” del sistema y necesitamos conectar la salida del diseño anterior con la de este diseño, que será el que esté vinculado a las E/S físicas de la placa.

Una breve representación de la conexión entre el diseño de este apartado y el

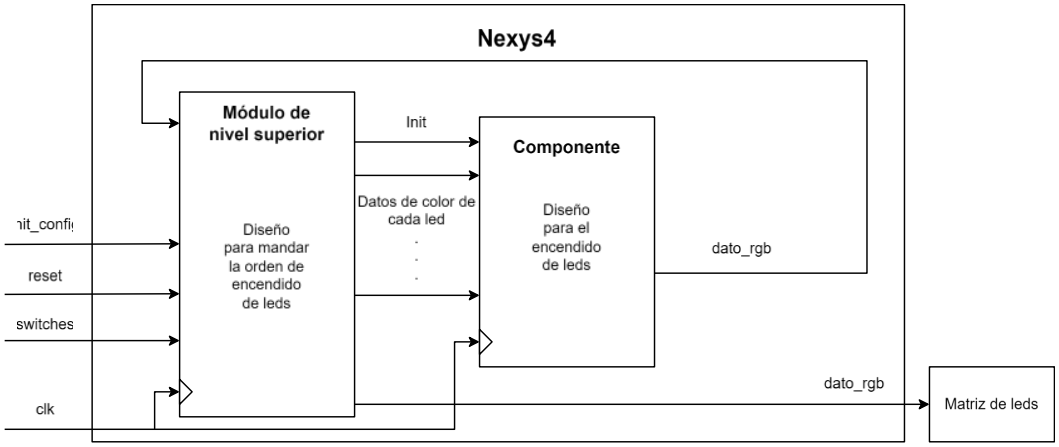

Figura 39: Estructura general del diseño Inicial en la placa Nexys4.

del anterior se muestra en la *Figura 39.*

Como señales internas en el diseño tenemos:

fila0 a fila31	Señales de 32 bits que se conectan a las 32 entradas de 32 bits. Cada una de ellas almacena el valor de color de los 8 leds correspondientes a su fila.
permiso_leds	Esta señal actúa como un “ENABLE” para el diseño del apartado *4.1 y* se conectará directamente con su entrada “init”.

Tabla 4: Señales del diseño CONTROL_MAIN.

El diseño se ha realizado con una máquina de estados que se explica a continuación:

- **Estado *s0***: Este estado comienza cuando se haya dado la orden de carga, es decir, cuando init_config = ‘1’. Una vez ocurra esto, se deshabilitará el enable (“permiso_leds = ‘0’ ”). Esto se hace para evitar que mientras se actualizan los valores de los registros, estos valores pasen simultáneamente a la entrada de datos de los leds.

  En este estado se utiliza un MUX 4 a 1, en el que la línea de selección serán las entradas *switches* del diseño. Las entradas del MUX, serán las indicadas en los estados s1, s2, s3 y s4, asociados a las entradas

de selección 00, 01, 10, 11, respectivamente. En esos estados, se cargarán los registros de diferentes formas.

- **Estado *s1***: En este estado se cargan todos los registros a 0, es decir, este estado configurará el apagado de los leds. Al cargar los leds se pasará al estado s5.

CONFIGURACIÓN 1

```
fila0 <= (others => '0');
…
Fila31 <= (others => '0');
```

- **Estado *s2***: Aquí se realiza otra configuración de leds. En este estado se encenderán todos los leds con el color verde, para comprobar que a todos los leds le llegan los datos. Al cargar los leds se pasará al estado s5.

CONFIGURACIÓN 2

```
fila1 <= X"11111111";
…
fila31 <= X"11111111";
```

- **Estado *s3***: En esta configuración se cargarán 4 filas en verde, 4 apagadas, y así sucesivamente hasta llegar a la última. Al cargar los leds se pasará al estado s5.

CONFIGURACIÓN 3

```
fila0 <= X"11111111";
fila1 <= X"11111111";
fila2 <= X"11111111";
fila3 <= X"11111111";
fila4 <= (others => '0');
fila5 <= (others => '0');
fila6 <= (others => '0');
fila7 <= (others => '0');
fila8 <= X"11111111";
fila9 <= X"11111111";
fila10 <= X"11111111";
fila11 <= X"11111111";
…
fila31 <= (others => '0');
```

- **Estado *s4***: En este estado se realiza la última de las configuraciones. Se encenderá un led y el siguiente no, y así sucesivamente en cada registro. Quedará como en el siguiente ejemplo. Al cargar los leds se pasará al estado s5. En este estado, se detectó el error mencionado en el apartado anterior.

CONFIGURACIÓN 4

```
fila0 <= X"01010101";
…
fila31 <= X"01010101";
```

- **Estado *s5***: En este estado se activa el "ENABLE" (*permiso_leds*) para que la información de los registros se transmita a los leds y se enciendan acorde a los valores cargados. No se podrá salir de este estado hasta que la entrada "init_config" esté a '0'. Esto se ha añadido como medida de seguridad para evitar posibles bucles infinitos si la entrada queda mantenida a '1'. Una vez se produzca su vuelta al valor '0' se pasaría al estado s0, y se volvería a comenzar un nuevo ciclo de la máquina de estados.

4.2.1 Simulaciones para la comprobación del correcto funcionamiento.

Para poder corroborar que el diseño está correctamente implementado, se realizó una simulación donde se prestó detalle a diferentes cosas:

- **Tiempos de transmisión de datos:**

Los tiempos deben coincidir con la según se mencionó en el apartado *2.2.3.1*.

Tiempo a '1' del 0 lógico (T0H)	400 ns
Tiempo a '0' del 0 lógico (T0L)	850 ns
Tiempo a '1' del 1 lógico (T1H)	800 ns
Tiempo a '0' del 1 lógico (T1L)	450 ns

Tabla 5: Tiempos de transferencia de datos. Fuente: [17]

Figura 40: Tiempo a 1 del '1' lógico.

Figura 41: Tiempo a 0 del '1' lógico.

Figura 42: Tiempo a 0 del '0' lógico.

Figura 43: Tiempo a 1 del '0' lógico.

- **<u>Comprobación del orden de bits:</u>**

Se pueden observar en la *Figura 44* los 24 bits que componen el color de un led. En este caso los leds corresponden al color verde, y se puede observar que se transmiten los bits desde el MSB hasta el LSB. Los 8 bits del medio de la trama corresponden al color rojo, y los 4 bits MSB de estos 8, no se usan para disminuir el consumo de los leds. Se puede observar que los datos coinciden con el color verde según la Tabla 6. Figura 42: Tiempo a 0 del '0' lógico.

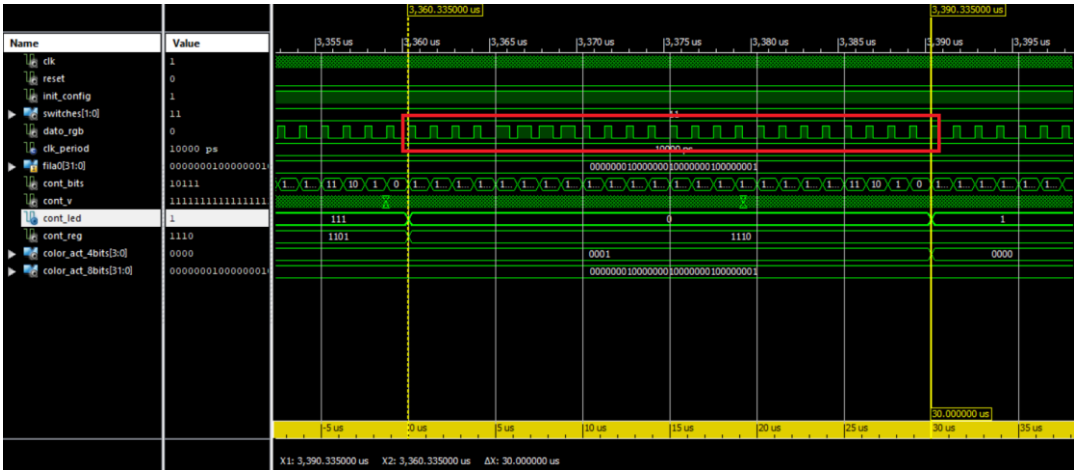

Figura 44: Transmision de 24 bits de configuracion de un LED.

Después de realizar las simulaciones, se realizó una prueba experimental con la placa Nexys4. Aquí se pudo apreciar otro error al enviar el color, ya que inicialmente se definieron las constantes de los 24 bits del color como si el orden de colores en la transmisión fuese RGB (Rojo – Verde - Azul), mientras que el orden real es GRB (Verde – Rojo – Azul). Este error fue detectado, ya que se ordenó cargar los LEDs en rojo y estos se encendian en verde. El error no es demasiado significativo, y se corrigió posteriormente en el diseño final que se usó para implementar el juego.

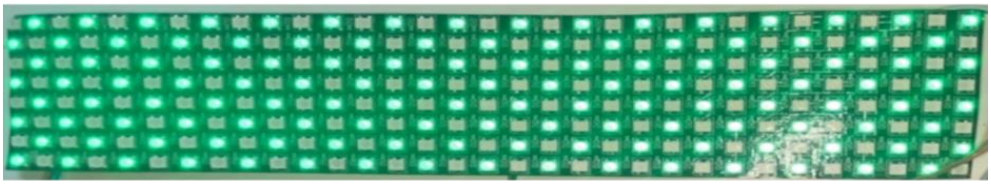

Figura 45: Configuración 4, comprobación experimental.

4.3 Diseño Hardware y programación en Vitis incluyendo el componente de la matriz de LEDs para comprobar que la escritura en los registros es correcta.

Para este diseño se trabajó con Vivado, donde se tuvo que crear un bloque IP basado en el diseño VHDL expuesto en el apartado *4.1.* Este bloque IP, tiene conectar el diseño mencionado con el procesador del SoC a través de un conjunto de registros internos (*Figura 46*) que servirán de comunicación entre el procesador del SoC y el controlador hardware diseñado para controlar los leds.

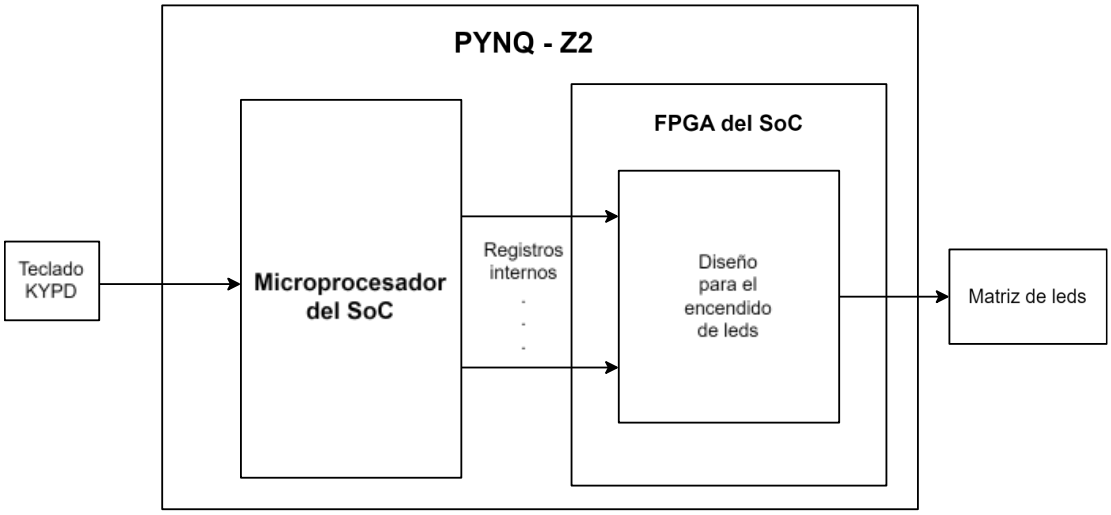

Figura 46: Estructura general del diseño en la placa PYNQ-Z2.

Estos registros serán de 32 bits, ya que es la capacidad que nos permite crear Vivado. En total se necesitan 33 registros según el diseño propuesto, 32 registros irán destinados a almacenar el valor de color que tiene que tener cada led (*Figura 47*) y un registro de control. Esto se ha hecho de la siguiente forma:

- Se codificó el color asignado a cada led de 24 bits con 4 bits según la Tabla 6. De estos 24 bits **no se usarán los 4 bits más significativos de cada color**, ya que la placa no puede suministrar tanta potencia (ver *Figura 16*). De esta manera, al usar 4 bits por led, con 32 registros es suficiente para cubrir los 256 leds. Los registros serán nombrados como: **fila0, fila1, … fila31**.

Color	***Codificación (4 bits)***	***Dato real (24 bits)***
Apagado	*0000*	*00000000 00000000 00000000*
Azul	*0001*	*00000000 00000000 00001111*
Verde	*0010*	*00001111 00000000 00000000*
Rojo	*0011*	*00000000 00001111 00000000*
Magenta	*0100*	*00000000 00001111 00001111*
Amarillo	*0101*	*00001111 00001111*

		00000000
Cyan	*0110*	*00001111 00000000 00001111*
Gris	*0111*	*00000111 00000111 00000111*
Blanco	*1000*	*00001111 00001111 00001111*

Tabla 6: Codificación de colores RGB en 4 bits.

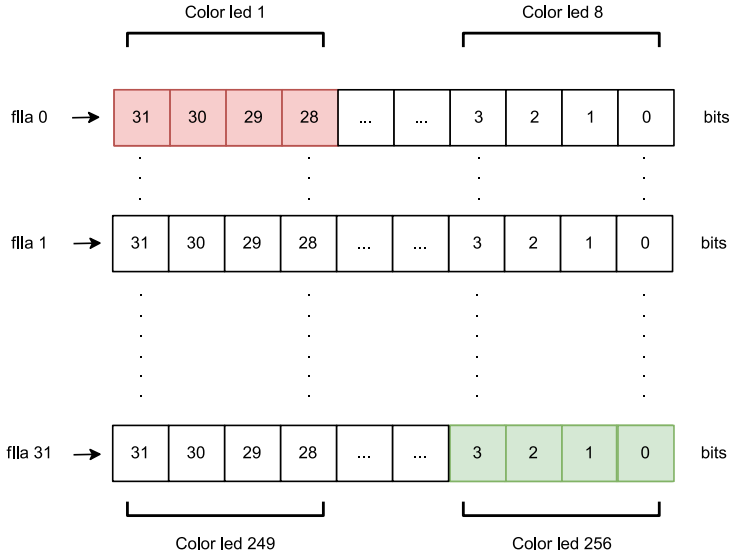

Figura 47: 32 registros de datos.

- El último registro es utilizado como registro de control (*Figura 48*) y su función consiste en transmitir variables de control como son las señales “Init” y “Reset”, ambas activas en alta. Que se usarán para iniciar una transmisión de datos a los LEDs (INIT), o finalizará la transmisión (RESET).

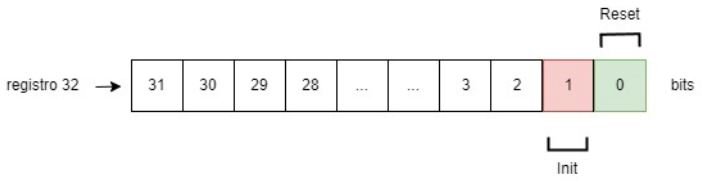

Figura 48: Registro de control.

Una vez creado ya este bloque, se instanció en el diagrama de bloques de Vivado

y se conectó al procesador y al resto de componentes para poder exportar la plataforma hardware. Posteriormente, con la plataforma hardware se creó un programa de aplicación que realizará unas funciones similares a las del diseño del apartado *4.2,* salvo que se han añadido dos funciones adicionales. Una de ellas escribirá la palabra ”CARLOS” en la matriz de LEDs, y otra realizará un desplazamiento del contenido de los registros hacia la derecha, lo que deberá provocar el movimiento de la imagen representada en la matriz de LEDs.

4.3.1 Creación del IP Core personalizado en Vivado para controlar los leds.

Para la creación del IP Core se siguieron los mismos pasos que en el apartado *3.2*. Se creó un bloque AXI4 IP esclavo, con 33 registros (*Figura 49*).

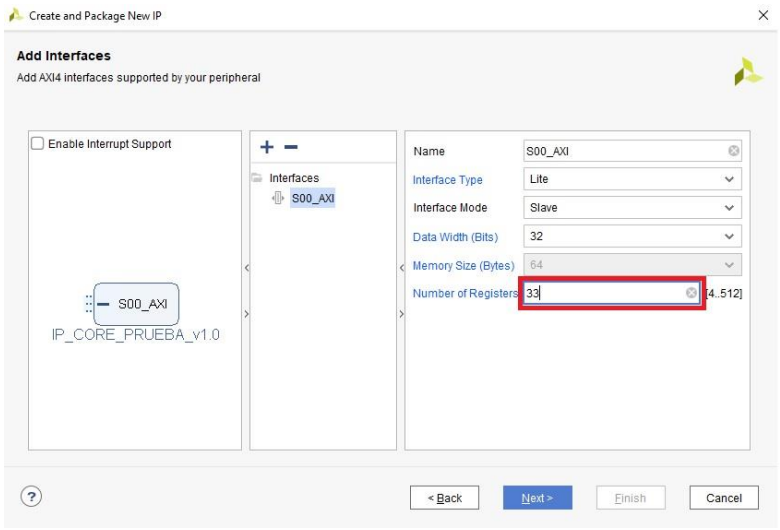

Figura 49: Creación de bloque IP personalizado.

Al crear la IP se generan automáticamente dos plantillas para la creación de instancias (*Figura 50*). Hay que instanciar el diseño en VHDL para encender los leds correctamente. Primero, se añade el fichero “.vhd” del diseño y posteriormente se edita el código de estas dos plantillas.

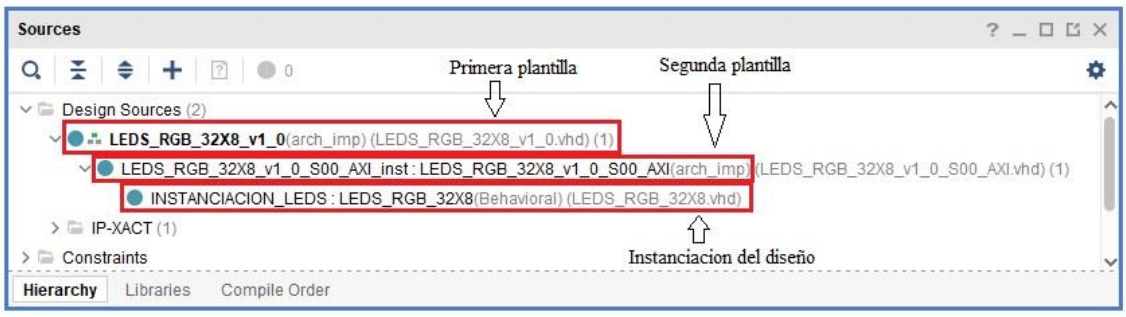

Figura 50: Instancias en el bloque IP personalizado.

La primera plantilla ***(LEDS_RGB_32X8_v1_0)*** que se crea, es usada para definir los puertos de la interfaz del bloque e instanciar la segunda plantilla.

La segunda plantilla ***(LEDS_RGB_32X8_v1_0_S00_AXI_inst: LEDS_RGB_32X8_v1_0_S00_AXI)*** se encarga de configurar la funcionalidad de los registros internos del bloque. Es aquí donde se pueden añadir nuevas funcionalidades.

La instanciación del diseño de control del encendido de leds ***(INSTANCIACION_LEDS: LEDS_RGB_32X8)*** se realiza sobre la segunda plantilla. Las entradas del diseño: *fila0, fila1, fila2…* se conectan directamente con los registros internos del esclavo (*slv_reg0, slv_reg1, slv_reg1…)* en orden. En el registro 32, que es el último, se conectan las entradas *init* y *reset* a los bits 1 y 0 del registro respectivamente. Es decir, este registro actuará como **registro de control**.

Como último paso, hay que crear una salida en la interfaz del bloque para que pueda conectar externamente con la entrada de datos de la matriz de leds. Esto se hace creando una salida en la primera plantilla y otra la segunda, interconectándolas entre sí, y conectando la salida de la segunda plantilla con la salida *dato_rgb* del componente instanciado.

Una vez editado todo, se procede a realizar la síntesis del diseño y la implementación. Si todo se ha conectado correctamente no aparecen errores, se guardan los cambios y se empaqueta el diseño para poder usar el bloque.

4.3.2 Diagrama de bloques del diseño

Inicialmente, al crear el bloque IP personalizado para estas pruebas previas, se le dio el nombre de “Demo5_v1.0”. Junto con el procesador “ZYNQ7 Processing System”, se añadió también un bloque IP “AXI_GPIO” con 2 canales para poder controlar los dos switches y los 4 botones de los que dispone la placa de desarrollo. La entrada de estos elementos está designada como “bnts” y “sws”, que corresponden a los botones y los switches respectivamente. Como salida, se ha añadido “dato_rgb”, que será la que transmita los datos a la matriz de LEDs (*Figura 51*).

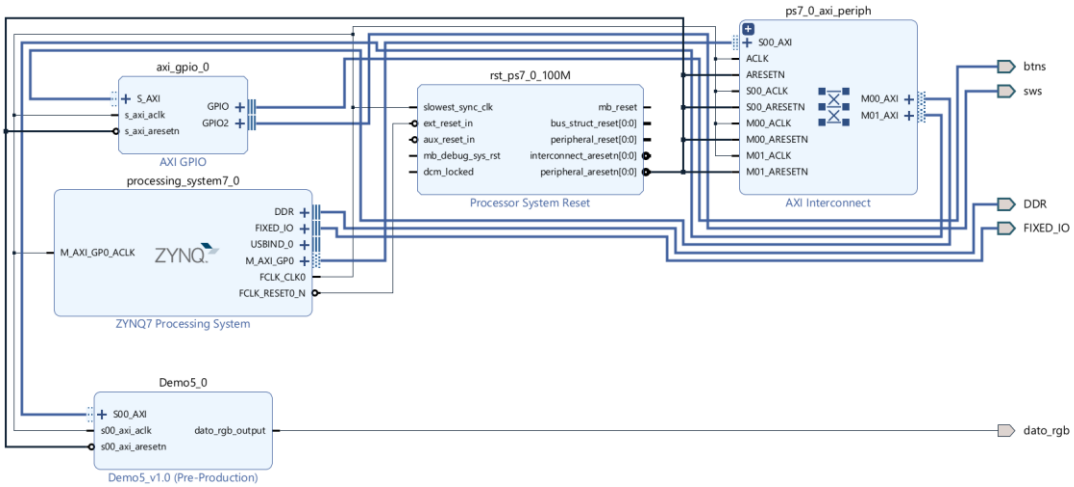

Figura 51: Diagrama de bloques del primer diseño previo

4.3.3 Programa de aplicación

El programa de aplicación tratará de usar los botones y los switches de la placa como las entradas de selección de un MUX. Hay 2 switches y 4 botones, lo que supone un total de 16 combinaciones posibles. Estas combinaciones irán repartidas de 4 en 4, y vendrá determinada por la posición de los switches según la *Figura 52*.

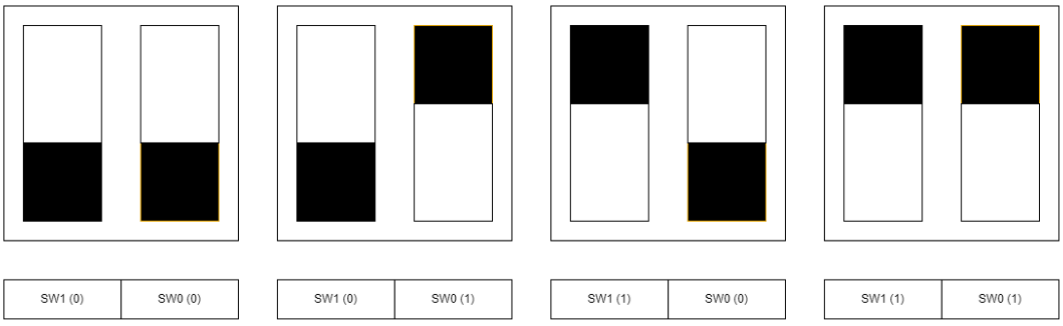

Figura 52: Combinaciones posibles de los switches.

Las funciones para cada estado serán las siguientes:

- **<u>Estado 1 (SW1(0)/SW0(0)):</u>**
  - Boton1: Se cargarán todos los LEDs del mismo color.
  - Boton2: Se cargará una fila de LEDs si y la siguiente no, alternando.
  - Boton3: Leer registros por el terminal serie.
  - Boton4: Apagar los LEDs.

- **Estado 2 (SW1(0)/SW0(1)):**
  - Boton1: Se cargará un LED si y el siguiente no, alternando, aunque siempre igual en cada fila.
  - Boton2: Se cargará un LED si y el siguiente no, alternando el orden en cada fila.
  - Boton3: Leer registros por el terminal serie.
  - Boton4: Apagar los leds.
- **Estado 3 (SW1(1)/SW0(0)):**
  - Boton1: Escribir "CARLOS" en la matriz de LEDs.
  - Boton2: Comenzar bucle de desplazamiento de los registros hacia la izquierda con un retraso de 1 segundo. Para salir del bucle, basta con cambiar la posición de los switches.
  - Boton3: Leer registros por el terminal serie.
  - Boton4: Apagar los leds.
- **Estado 4 (SW1(1)/SW0(1)):** (Este estado se ha creado una vez depurados los 3 estados anteriores en la placa de desarrollo).
  - Boton1: Escribir "CARLOS" en la matriz de LEDs.
  - Boton2: Comenzar bucle de desplazamiento de los registros hacia la derecha con un retraso de 1 segundo. Para salir del bucle, basta con cambiar la posición de los switches.
  - Boton3: Leer registros por el terminal serie.
  - Boton4: Apagar los leds.

El código en C se encuentra en el Apendice A.

4.3.4 Conclusiones del diseño

La Finalidad de este diseño era crear la IP personalizada basada en el diseño que se realizó en VHDL, aprender a crear programas de aplicación en Vitis, controlar el "movimiento" de los LEDs y poder representar palabras en la matriz.

Al cargar el diseño dentro de la placa de desarrollo, se pudo detectar un notable error, ya que la palabra "CARLOS" se veía como si estuviese reflejada en un espejo (*Figura 53*). Esto se solucionó cambiando el contenido de todos los registros, desde el contenido del registro más significativo al del menos significativo (*Figura 55*). Con respecto al desplazamiento de los registros, la prueba fue exitosa. Se podía comprobar físicamente en la matriz de LEDs como se desplazaba, y además, gracias a la función de "Leer registros por el terminal serie", se pudo comprobar que el contenido de los registros se desplazaba correctamente (*Figura 54*).

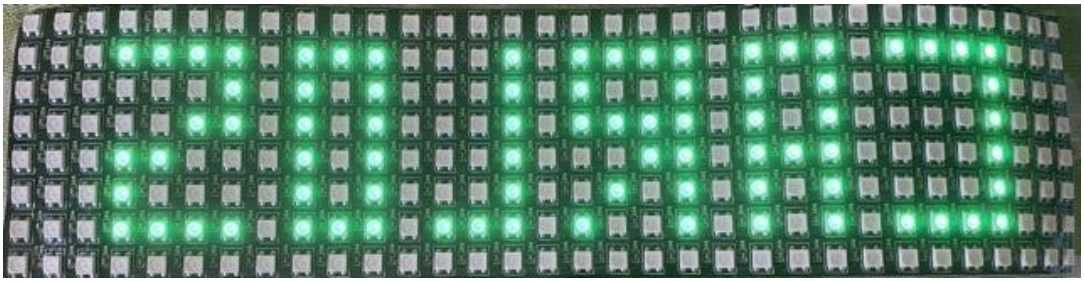

Figura 53: Error al escribir "CARLOS" en la matriz de LEDs.

SIN DESPLAZAMIENTO

Registro 0: 0
Registro 1: 0
Registro 2: 0
Registro 3: 1111110
Registro 4: 1000010
Registro 5: 1000010
Registro 6: 1000010
Registro 7: 0
Registro 8: 1111110
Registro 9: 1001000
Registro 10: 1111110
Registro 11: 0
Registro 12: 1111110
Registro 13: 1011000
Registro 14: 1010100
Registro 15: 1110010
Registro 16: 0
Registro 17: 1111110
Registro 18: 10
Registro 19: 10
Registro 20: 0
Registro 21: 1111110
Registro 22: 1000010
Registro 23: 1111110
Registro 24: 0
Registro 25: 1110010
Registro 26: 1010010
Registro 27: 1001010
Registro 28: 1001110
Registro 29: 0
Registro 30: 0
Registro 31: 0

PRIMER DESPLAZAMIENTO

Registro 0: 0
Registro 1: 0
Registro 2: 0
Registro 3: 0
Registro 4: 1111110
Registro 5: 1000010
Registro 6: 1000010
Registro 7: 1000010
Registro 8: 0
Registro 9: 1111110
Registro 10: 1001000
Registro 11: 1111110
Registro 12: 0
Registro 13: 1111110
Registro 14: 1011000
Registro 15: 1010100
Registro 16: 1110010
Registro 17: 0
Registro 18: 1111110
Registro 19: 10
Registro 20: 10
Registro 21: 0
Registro 22: 1111110
Registro 23: 1000010
Registro 24: 1111110
Registro 25: 0
Registro 26: 1110010
Registro 27: 1010010
Registro 28: 1001010
Registro 29: 1001110
Registro 30: 0
Registro 31: 0

TRAS 5 ITERACIONES

Registro 0: 1001010
Registro 1: 1001110
Registro 2: 0
Registro 3: 0
Registro 4: 0
Registro 5: 0
Registro 6: 0
Registro 7: 0
Registro 8: 1111110
Registro 9: 1000010
Registro 10: 1000010
Registro 11: 1000010
Registro 12: 0
Registro 13: 1111110
Registro 14: 1001000
Registro 15: 1111110
Registro 16: 0
Registro 17: 1111110
Registro 18: 1011000
Registro 19: 1010100
Registro 20: 1110010
Registro 21: 0
Registro 22: 1111110
Registro 23: 10
Registro 24: 10
Registro 25: 0
Registro 26: 1111110
Registro 27: 1000010
Registro 28: 1111110
Registro 29: 0
Registro 30: 1110010
Registro 31: 1010010

Figura 54: Desplazamiento del contenido de los registros.

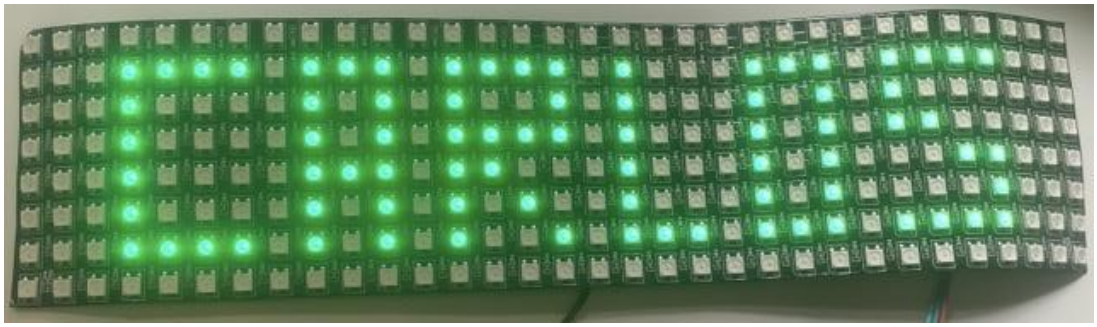

Figura 55: Corrección al escribir nombre en la matriz de LEDs.

4.4 Diseño Hardware y programación para controlar teclado PmodKYPD

En este apartado se va a hablar sobre el diseño que se desarrolló para comprobar el funcionamiento del teclado PmodKYPD. Este diseño consta de 3

elementos: el teclado, la matriz de LEDs y la placa de desarrollo.

El bloque IP necesario para controlar el teclado se puede descargar de la librería vivado-master, disponible en: https://digilent.com/reference/vivado/library. Hay que buscar en el navegador de IPs cada bloque y añadirlo.

Para añadir el bloque PmodKYPD_1 o cualquier otro procedente de las librerías de Vivado, es recomendable hacerlo desde la pestaña **Board,** ya que de esta forma Vivado creará automáticamente los puertos necesarios para el bloque y los conectará correctamente. Se va a dedicar el puerto Pmod B para la comunicación con el teclado, por tanto, se hará doble click en PmodB dentro de la pestaña **Board** y aparecerán todos los bloques de la librería disponibles que usan este puerto. Entre todos los que hay se elegirá el nombrado anteriormente, y Vivado automáticamente conectará el bloque con el puerto Pmod. (En este caso se ha designado al puerto como “jb”)

El diagrama de bloques quedará de la forma mostrada en la *Figura 56*.

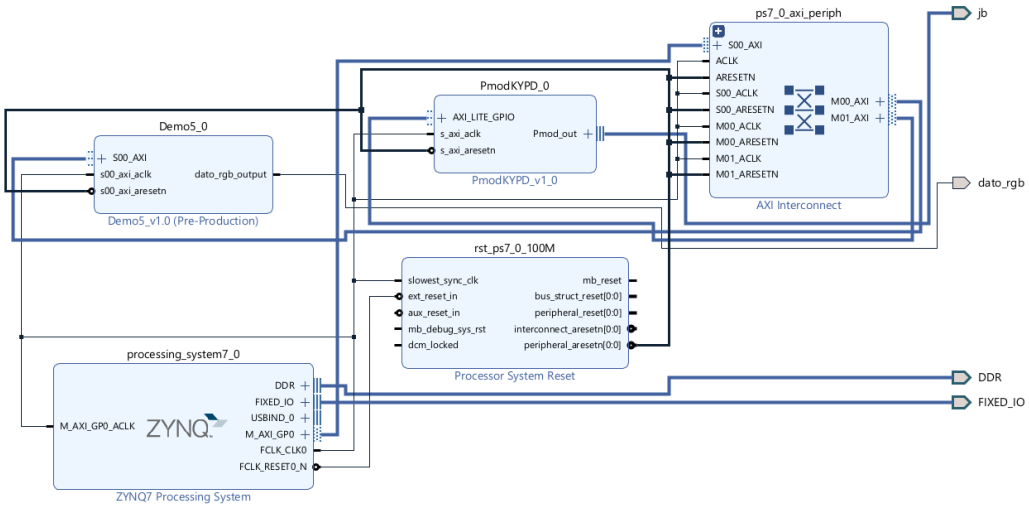

Figura 56: Diagrama de bloques del segundo diseño previo.

La aplicación que se cargará en la placa de desarrollo para este diseño, será bastante simple. Cada vez que se pulse una tecla, en la matriz de LEDs se mostrará el carácter de la tecla pulsada, excepto si se pulsa la tecla ‘0’, en cuyo caso se apagaran todos los LEDs.

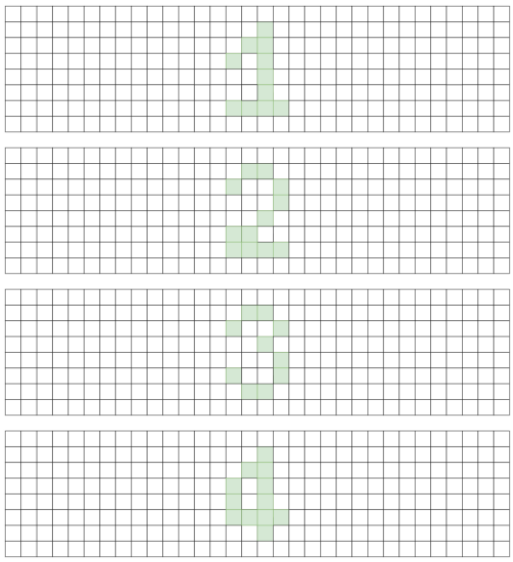

Figura 60: Representacion caracteres del teclado (I)

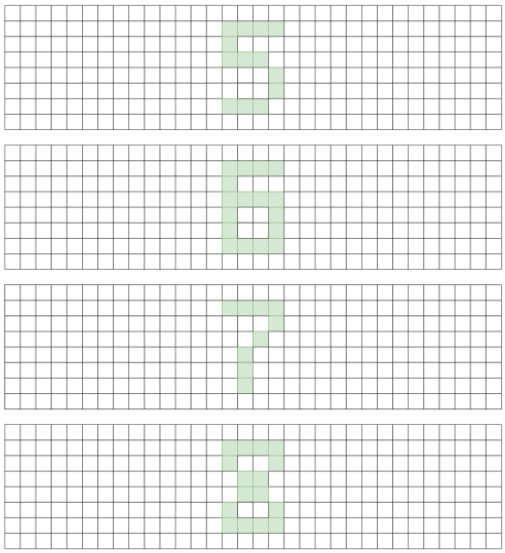

Figura 59: Representacion caracteres del teclado (II)

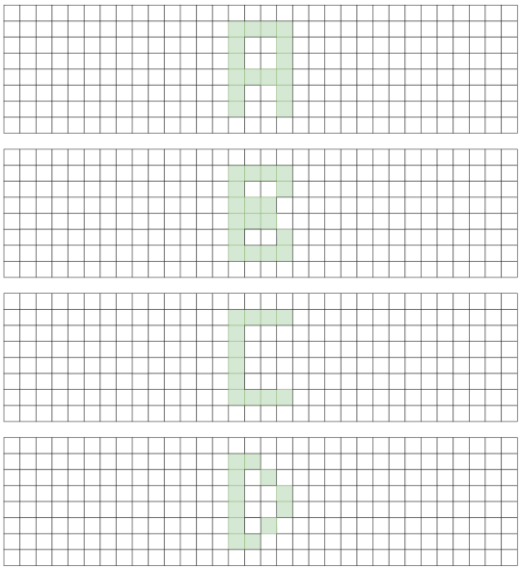

Figura 57: Representacion caracteres del teclado (III)

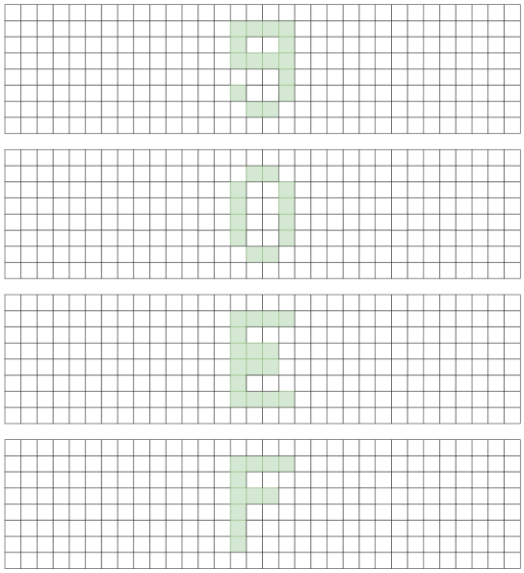

Figura 58: Representacion caracteres del teclado (IV)

Para poder leer las teclas correctamente, se usó la librería “PmodKYPD.h”. Para poder controlar el teclado, primero será necesario inicializar el telado, y para ello se utilizarán 3 funciones pertenecientes a esta librería:

```
void KYPD_Initialize()//Inicialización del bloque de control para el PMODB
para el teclado.
{
   KYPD_begin(&myDevice, XPAR_PMODKYPD_0_AXI_LITE_GPIO_BASEADDR);
   KYPD_loadKeyTable(&myDevice, (u8*) DEFAULT_KEYTABLE);
//    PARA INDICAR QUE TODAS LAS SALIDAS SE COMPORTARAN COMO SALIDA DE
ENTRADA DE DATOS.
   Xil_Out32(myDevice.GPIO_addr, 0xF);
}
```

- En primer lugar, una función que incluya la dirección de memoria asignada al periférico y lo inicialice.

- Una segunda, que se utilizará para cargar una matriz alfanumérica correspondiente a los valores de cada una de las teclas del PmodKYPD. Esta función servirá para poder relacionar el valor numérico que devolverá la fila y columna de cada tecla al ser pulsada y transformarlo en el número o letra que se desee.

- Y, por último, habrá que utilizar una función para indicar que todas las salidas se comportaran como salidas de entrada de datos.

Una vez configurado el teclado, se leerán las teclas pulsadas por medio de otras dos funciones:

- Primero, se observará cual es el estado del teclado, es decir, que filas y columnas han sido pulsadas. Esta operación se realiza a través de la función “KYPD_getKeyStates”.

- Una vez se sabe si se ha pulsado una sola tecla o varias, mediante otra función se observa que tecla se ha pulsado. Para ello se utiliza la función “KYPD_getKeyPressed”. En el caso de que solo se haya pulsado una tecla guardará el valor de esta tecla en la variable “key”, para poder utilizarla más adelante.

- Para que una misma tecla no se repita al mantenerse pulsada, se guardará el estado actual y el anterior del teclado. De esta forma, hasta que el estado no cambie, no se detectará una nueva pulsación. Todo este proceso de obtención de datos se repetirá cada vez que se quiera obtener información del teclado.

El fragmento de código es el siguiente:

```
   //Variables
   u16 keystate; //Estado de filas y columnas
   XStatus status, last_status = KYPD_NO_KEY; //Estado actual y anterior
   u8 key, last_key = 'x'; //Tecla actual y anterior

   // El valor inicial de la ultima tecla no puede ser contenida en la
cadena de la tabla de teclas cargada (x).

//   Xil_Out32(myDevice.GPIO_addr, 0xF);

   xil_printf("Pmod KYPD demo started. Press any key on the Keypad.\r\n");
   while (1) {
      // Captura el estado de cualquier tecla
```

```
        keystate = KYPD_getKeyStates(&myDevice);

        // Si solo se ha pulsado una tecla, determina cual es
        status = KYPD_getKeyPressed(&myDevice, keystate, &key);

        // Print key detect if a new key is pressed or if status has changed
        if (status == KYPD_SINGLE_KEY
              && (status != last_status || key != last_key)) {
           xil_printf("Key Pressed: %c\r\n", (char) key);
           last_key = key;
           Imprime_tecla(key);
        }
        else if (status == KYPD_MULTI_KEY && status != last_status)
           xil_printf("Error: Multiple keys pressed\r\n");

        last_status = status;

        usleep(1000);
    }
```

Tras finalizar esta aplicación y comprobar su funcionamiento, no se detectó ningún fallo y todo funcionó correctamente. Una vez comprobada la metodología de diseño y realizadas las pruebas necesarias para garantizar que todo se estaba haciendo correctamente, se pasó al diseño de la aplicación principal de este trabajo.

5 DESARROLLO DE LA APLICACIÓN

En este capítulo se van a desglosar todos los pasos que se han realizado para poder implementar la aplicación y se hará una breve descripción general sobre el juego. Describiremos todos los pasos llevados a cabo para su implementación:

- Diseño del diagrama de bloques adecuado en Vivado.
- Diseño del programa en C en Vitis para el Juego.

5.1 Diseño del diagrama de bloques en Vivado.

El último paso para dar por concluida la parte hardware es crear el diagrama de bloques. Esta forma de crear diseños es mucho más intuitiva y sencilla, ya que trabajaremos directamente con las interfaces de los bloques IP. El diagrama estará compuesto por 3 elementos principales, que son:

- El procesador, que en este caso es el Zynq-7000. El nombre de su bloque IP es *ZYNQ7 Processing System* (*Figura 61*).

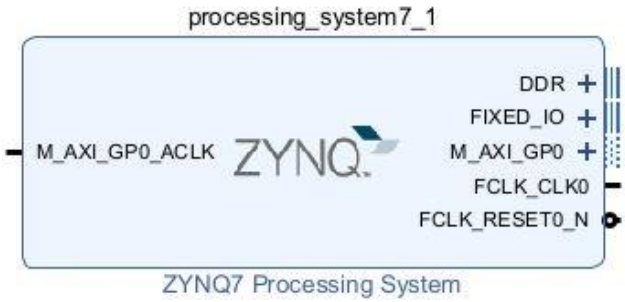

Figura 61: Bloque ZYNQ7 Processing System.

- El bloque IP creado en el apartado anterior para controlar los leds, al que se le ha dado el nombre de *LEDS_RGB_32X8_v1.0* (*Figura 62*).

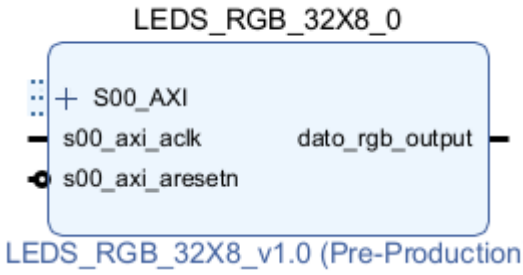

Figura 62: Bloque LEDS_RGB_32X8_v1.0.

- El bloque IP que contiene el diseño hardware necesario para poder controlar de forma sencilla el teclado KYPD. Se usará el bloque IP llamado *PmodKYPD_v1_0* (*Figura 63*).

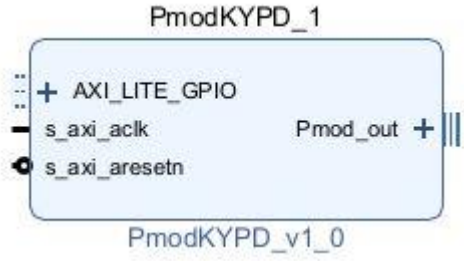

Figura 63: Bloque PmodKYPD_v1_0.

Una vez esten los 3 bloques principales en el diagrama, se siguen los pasos del apartado *3.3* y se interconectará todo, quedando un diagrama final como el de la *Figura 64*.

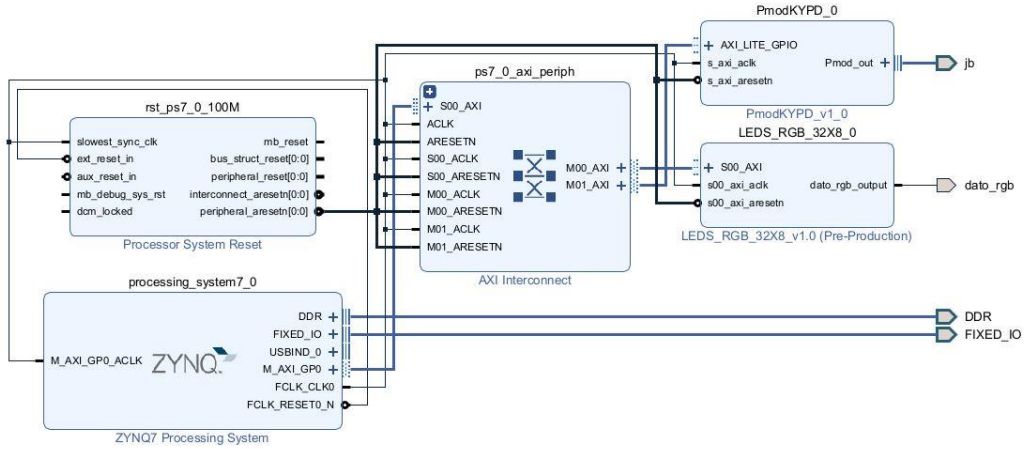

Figura 64: Diagrama de bloques final.

Al interconectar todos los elementos, se pueden ver varios datos interesantes:

En el bloque *ZYNQ7 Processing System* cabe destacar dos señales importantes como son: M_AXI_GP0_ACLK y M_AXI_GP0.

- M_AXI_GP0_ACLK contiene el reloj del bus AXI, que proviene del reloj FCLK_CLK0 del SoC y controlará al resto de dispositivos.
- M_AXI_GP0 que se trata de una señal bidireccional, que se utiliza para comunicarse con el resto de esclavos. En nuestro diagrama tenemos dos esclavos, que conforman nuestro sistema hardware: *PmodKYPD_v1_0* y *LEDS_RGB_32X8_v1.0*, que serán comandados por las señales M01_AXI y M00_AXI respectivamente.

Aparecen dos nuevos bloques: **AXI Interconnect** y **Processor System Reset.** El primer bloque permite conectar y comunicar los diferentes componentes AXI entre sí. Este bloque requiere de un sistema de reset, y para eso se utiliza el bloque Processor System Reset, que parte de la conexión del reset del reloj principal del procesador FCLK_RESET0_N y llega hasta los diferentes resets del resto de bloques.

Los siguientes pasos serán los que se mencionan en los apartados *3.4* y *3.5*. hasta exportar el archivo XSA para la plataforma.

5.2 Diseño del programa de aplicación (en C) para el juego.

En este apartado se realizará una descripción de los requisitos para el pequeño juego, y después se entrará en detalle en cómo se ha abarcado la solución.

El juego será una versión simple de un juego estándar de recoger objetos. La matriz de leds actuará como pantalla, y los distintos leds, serán los pixeles de la pantalla. El teclado KYPD será la interfaz con el usuario y a través de él podremos tomar control del juego.

Los objetos serán representados por leds encendidos y caerán en lugares aleatorios. La velocidad de caída se irá incrementando cada cierto tiempo. El usuario podrá mover una barra de izquierda a derecha que tendrá una longitud de 4 leds. Esta barra actuará como una cesta para recoger los objetos. Cada vez que un objeto alcance la cesta, se incrementará la puntuación (Ver *Figura 66*). Una vez que un objeto alcance el suelo y no la barra (Ver *Figura 65*), el juego finalizará y se mostrará en la “pantalla” la puntuación alcanzada.

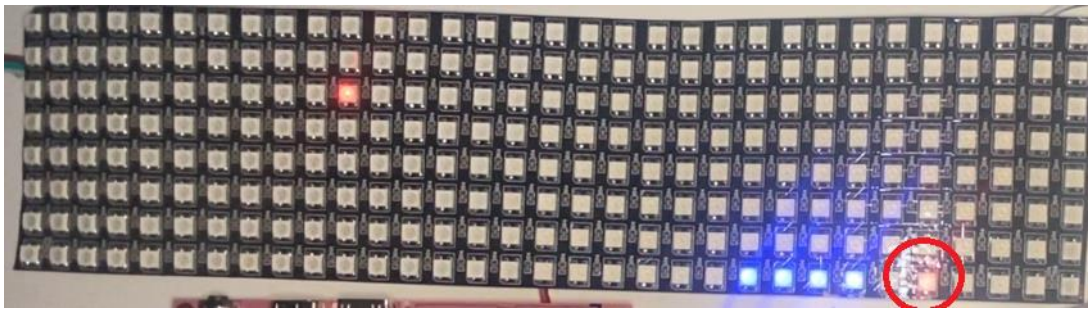

Figura 65: Objeto alcanza el suelo.

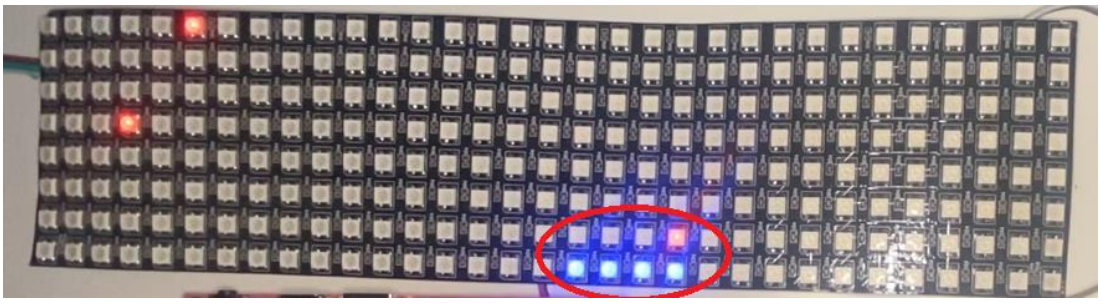

Figura 66: Objeto alcanza la cesta.

Para realizar este software, se ha optado por usar un sistema operativo en tiempo real *FreeRTOS.* Se va a usar este sistema operativo, ya que necesitamos un sistema *Multitasking* para poder ejecutar dos tareas independientes simultáneamente, una de ellas irá destinada a leer los datos que se reciben desde el teclado, y la otra realizará todo el control del juego.

El código del programa en C, esta dividido en dos ficheros; un fichero principal (*main),* en el cual se encuentran las funciones principales del juego (Apartado *5.2.1*), y las dos tareas mencionadas en el párrafo anterior (Apartado *5.2.3*), y un fichero secundario (*driver_ip*), que es una librería que se ha creado y que contiene funciones complementarias. A continuación, se describirán todas las funciones usadas en el desarrollo de la aplicación.

5.2.1 Funciones del fichero main

5.2.1.1 Driver_LEDS_RGB_Initialize(void)

Para manejar la mayoria de funciones se ha creado una estructura *"XDriver"* que contenga los datos esenciales de cada bloque IP, así como su dirección base en memoria y su ID de dispositivo.

Esta función, se encargará de cargar los parámetros pertinentes en la estructura *custom_ip,* que es de tipo de *XDriver* para que pueda usarse posteriormente.

5.2.1.2 KYPD_Initialize(void)

Esta inicialización realiza la misma función que la funcion anterior. Es decir, trata de rellenar los diferentes parámetros de la estructura *PmodKYPD*, que incluyen:

- Dirección base del bloque IP en memoria.
- Una tabla que contendrá los diferentes caracteres de las teclas.
- Una confirmación de si se ha cargado la tabla de caracteres con éxito o no.
- Se configuran las columnas de la matriz del teclado como salidas y las filas como entradas (los 4 bits LSB son las 4 columnas y los 4 bits anteriores son las 4 filas):

```
Xil_Out32(InstancePtr->GPIO_addr + 4, 0xF0);
```

El bloque *PmodKYPD_v1_0* del diagrama de bloques actúa como un módulo GPIO, y por ello, la siguiente dirección en memoria a su dirección base, es el Registro de Control, según la *Tabla 8.*

Address Space Offset[3]	Register Name	Access Type	Default Value	Description
0x0000	GPIO_DATA	R/W	0x0	Channel 1 AXI GPIO Data Register.
0x0004	GPIO_TRI	R/W	0x0	Channel 1 AXI GPIO 3-state Control Register.
0x0008	GPIO2_DATA	R/W	0x0	Channel 2 AXI GPIO Data Register.
0x000C	GPIO2_TRI	R/W	0x0	Channel 2 AXI GPIO 3-state Control.
0x011C	GIER[1]	R/W	0x0	Global Interrupt Enable Register.
0x0128	IP IER[1]	R/W	0x0	IP Interrupt Enable Register (IP IER).
0x0120	IP ISR[1]	R/TOW[2]	0x0	IP Interrupt Status Register.

Tabla 8: Registros de un módulo AXI GPIO. Fuente: docs.xilinx.com

En este registro, se configuran los puertos del módulo bien como entrada o como salida, escribiendo un '1' o un '0' en el bit correspondiente según la *Tabla 7*.

Bits	Field Name	Access Type	Reset Value	Description
[GPIOx_Width-1 :0]	GPIOx_TRI	Read/ Write	GPIO: Default Tri State Value GPIO2: Default Tri State Value	AXI GPIO 3-State Control Register. Each I/O pin of the AXI GPIO is individually programmable as an input or output. For each of the bits: 0 = I/O pin configured as output. 1 = I/O pin configured as input.

Tabla 7: Configuración de puertos E/S en el Registro de Control. Fuente: docs.xilinx.com

5.2.1.3 Inicialización_del_juego(XDriver* Puntero_inst)

Esta función coloca cada elemento en su posición inicial. Esto es, colocar la cesta en el centro de la pantalla encendiendo solo esos leds, y apagando el resto de leds (Ver *Figura 67*).

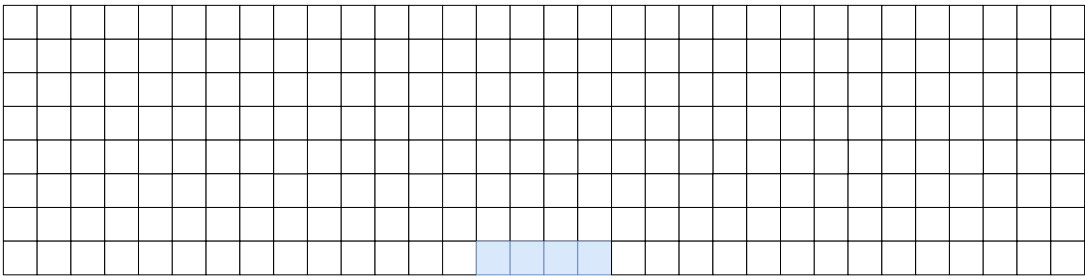

Figura 67: Posición inicial de elementos en matriz de leds.

5.2.1.4 Desplazar_barra_dcha(XDriver *Puntero_inst) y Desplazar_barra_izq(XDriver *Puntero_inst)

Estas dos funciones se encargarán de mover la "cesta" de izquierda a derecha cuando sea necesario. Basicamente lo que se hace es desplazar el valor de los

registros 1 posición de memoria más o 1 posición menos. Esto se hace modificando solo los 4 bits menos significatidos en los registros pertinentes, que corresponden a los leds inferiores, el resto de bits mantendrán su valor anterior. Para controlar donde está la "cesta" en todo momento, se usarán dos variables que marcarán los extremos de la "cesta". Estas variables serán *slicer_izq* y *slicer_dcha.* Cuando la barra llegue a alguno de los dos extremos, la barra no podrá moverse mas hacia ese extremo.

Ejemplo:

Si la barra llega al extremo izquierdo y se manda una orden de desplazamiento hacia la izquierda, la barra no se moverá, solo será posible el movimiento hacia la derecha.

5.2.1.5 int Generación_bolas(void)

Esta función se encargará de generar números aleatorios en un rango entre 0 y 31 que determinará la posición en la que se introducirá la nueva bola que debe caer.

5.2.1.6 Led_fall(XDriver *Puntero_inst)

Esta es una de las funciones más importantes. Se encarga del movimiento de caída de los objetos y comprobar si un objeto ha sido recogido. Para ello se desplaza el contenido de cada registro 4 bits a la derecha (realizando el movimiento de caída). Si el "objeto" no ha alcanzado el "suelo" continuará el movimiento de caída. Si los 4 bits del color del led han alcanzado la posición del "suelo", se comprueba si en ese registro hay uno de los leds de la "cesta" activos. Si coinciden, se continuará con el juego, y si no, el juego terminará. Esto será indicado por la variable *Game_over.* También se usará la variable *colision* que indicará los 3 casos mencionados anteriormente:

1- Ninguno de los "objetos" ha llegado al suelo aún.

2- Un "objeto" ha caido, pero no ha caido en la "cesta".

3- Un "objeto" ha caido dentro de la "cesta".

Adicionalmente, se han añadido 2 parámetros *x* y *siguiente_caida*. El parámetro ***x*** actuará como un contador de ciclos, y el parámetro ***siguiente_caida*** indicará en que ciclo hay que introducir un nuevo objeto. El valor de esta variable se generará aleatoriamente (Ver *Figura 68*). Entre objeto y objeto habrá una distancia mínima de dos LEDs, para que el jugador pueda responder a tiempo.

A medida que se va aguantando más en el juego, la velocidad de caída de los "objetos" irá variando. Esto se hace decrementando el retraso con el que se ejecuta esta función. Habrá dos umbrales de velocidad; al llegar al primer umbral, la reducción del retraso será más pequeña, y al llegar al segundo umbral, se va a reducir la velocidad de caída, y por tanto, el retraso será aumentado. Para realizar esta tarea se ha utilizado la variable *x* mencionada anteriormente, y dos constantes: DELAY_1_SECOND y DELAY_0_9_SECOND, que marcarán los dos umbrales de velocidad. La reducción del retraso viene definido por las constantes *incremento_caida* e *incremento_caida_2,* cada uno para una de las dos zonas separadas por los umbrales.

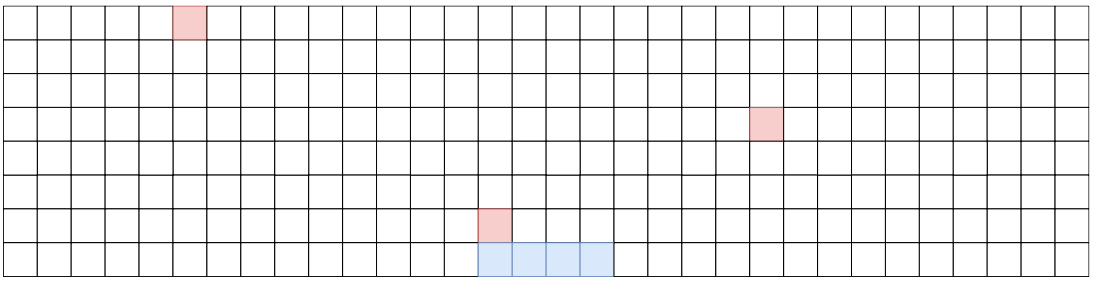

Figura 68: Ejemplo de funcionamiento del juego.

5.2.2 Funciones de la librería creada "driver_ip.c"

5.2.2.1 Dibujar_pantalla_de_inicio(XDriver *Puntero_inst, short int)

Aquí se cargará la tira de leds de tal forma que muestre el texto: "PULSE : A" (Ver *Figura 69*). Con esta información, el jugador podrá saber que tecla tiene que pulsar para que el juego comience. Para ello, se cargarán todos los registros con la información necesaria para mostrar el texto correctamente, en el color que se le indique a la función.

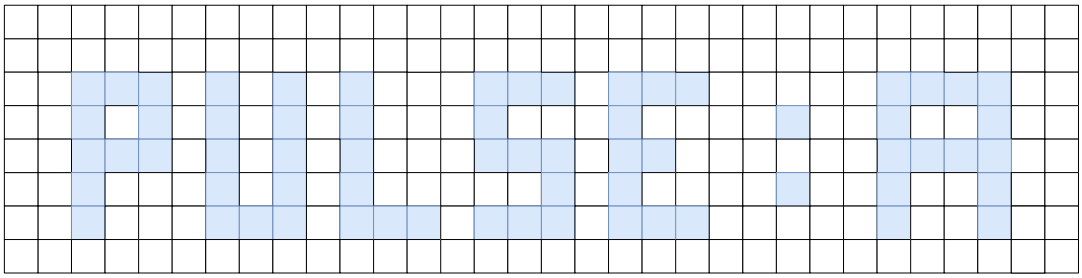

Figura 69: Escena: "PULSE: A" en matriz de leds.

5.2.2.2 Mostrar_puntuación(XDriver *Puntero_inst, short int color, int puntuacion)

Esta función recibe la puntuación que ha conseguido el jugador tras perder la partida, la cual se divide en 3 digitos: centenas, decenas y unidades, y se muestra por pantalla. El texto que se mostrará será "PTS : 000" siendo "000" los 3 dígitos

de la puntuación (Ver *Figura 70*).

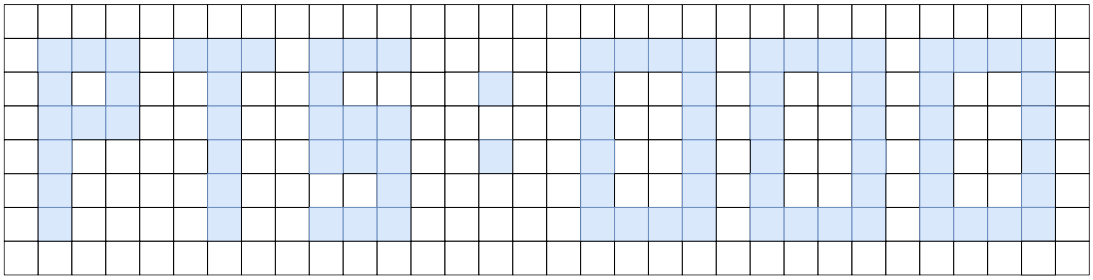

Figura** 70**: Escena: Puntuación total.

5.2.2.3 Apagar_leds(XDriver *Puntero_inst, int num_reg)

Esta es una función esencial. Aquí se limpiará la "pantalla", y para ello se pondrán todos los registros a 0, y, en consecuencia, se apagarán todos los leds una vez reciban los datos. Esta función también es usada para el parpadeo de la puntuación en la función ***Mostrar_puntuación(***XDriver *Puntero_inst, short int color, int puntuacion)***.** En esta función apaga una parcialmente los leds, el resto siguen encendidos según corresponda. Para ello lo unico que hay que alterar es el parámetro *num_reg* de entrada de la función.

5.2.2.4 Activar_init(long unsigned int Base_ip_address, int num_reg) y Desactivar_init(long unsigned int Base_ip_address, int num_reg):

Estas dos funciones son necesarias cada vez que se va a actualizar la pantalla. Estas cargan un '1' o un '0' en el bit 1 del registro 32, que es el registro de control. Al activar init (escribir un '1') se pasarán los datos de los registros a los leds.

Desactivar init (escribir un '0') se usa para volver a cargar nuevos datos en la tira de leds. Es necesario, ya que el diseño realizado requiere una activación de Init por flanco, por tanto, hay que crear un flanco de subida para volver a cargar datos, y esto se obtiene gracias a la bajada creada por "Desactivar_init". En la *Figura 71*: Nueva transmisión de datos tras flanco de subida de "init". se muestra cómo se inicia una transmisión después del flanco de subida de la señal "init".

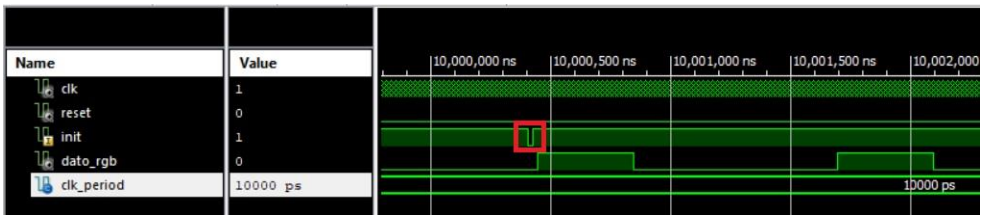

Figura** 71**: Nueva transmisión de datos tras flanco de subida de "init".

5.2.2.5 int main(void)

En la función principal se ejecutaran las inicializaciones de los dispositivos: ***Driver_LEDS_RGB_Initialize(***void) y ***KYPD_Initialize(***void). Y adicionalmente se crearán todos los elementos necesarios relativos a las tareas del Sistema Operativo.

5.2.3 Tareas del Sistema Operativo

El juego debe ejecutar dos tareas de forma simultánea según se ha mencionado anteriormente. Para ello, se usarán dos tareas: una será aquella que se encargue de leer las teclas que se pulsan y actuar en consecuencia: ***prvKYPDTask(***void*), y la otra será la que se encargue de controlar el juego, así como la caída de los objetos y el cambio entre las diferentes etapas del juego: ***Maquina_de_Estados_Task(***void*)***.***

5.2.3.1 prvKYPDTask(void*)

Para leer las teclas pulsadas se usará esta tarea. La mayoría del contenido de esta tarea ha sido sacada de las librerías que genera la plataforma extraida desde Vivado: **PmodKYPD.c**. El juego estará controlado principalmente por 4 teclas: **4, 6, A** y **D**. El resto de teclas no tienen efecto alguno en el juego.

Las teclas **4** y **6** se usarán para controlar el desplazamiento de la "cesta". La tecla **4** comandará el movimiento hacia la izquierda, y la tecla **6** comandará el movimiento hacia la derecha. Todo esto, siempre y cuando la ***Maquina_de_Estados***_Task(void*) se encuentre en el estado de "*Juego en marcha".*

La tecla **A** solo tendrá uso en el "*Estado de reposo*", cuando la pantalla formada por los leds muestre el texto "PULSE: A". Al pulsar esta tecla se cambiará del "*Estado 0*" al "*Estado 1*" en la máquina de estados.

La tecla **D** es la única que tendrá efecto en todo momento. Al pulsar esta tecla se provoca un reinicio del juego, pasando al estado de reposo (Estado 0).

Si se pulsan varias teclas simultáneamente, se indicará por el puerto serie. Al igual que, si solo se pulsa una tecla, se mostrará cual es la que se ha pulsado.

5.2.3.2 Maquina_de_Estados_Task(void*)

Esta tarea se encarga de gestionar una máquina de estados, de 5 estados que representarán distintas etapas del juego. Entre ellos se encuentran:

- **Estado 0: Estado de reposo:**

  En este primer estado, se mostrará "PULSE: A" en los leds, gracias a la función ***Dibujar_pantalla_de_inicio(***XDriver *Puntero_inst, short int***)***. Una vez enviados los datos, se esperará un tiempo de 1,5 segundos en la tarea para asegurar que los leds se han cargado correctamente.

- **Estado 1: Inicialización del juego:**

En este caso se iniciará la puntuación a 0, y se ejecutará la función ***Inicialización_del_juego(***XDriver* Puntero_inst) para encender los correspondientes leds iniciales. Una vez hecho esto, se pasará automáticamente al estado 2.

- **Estado 2: Juego en marcha:**

Este será el estado principal de la máquina de estados. Aquí se ejecutará la función que controlará la caída de los "objetos" del juego: ***Led_fall(***XDriver *Puntero_inst). Posteriormente se esperará el tiempo necesario según el retardo de la caída de los leds, que será un parámetro variable.

- **Estado 3: Fin del juego - Borrar pantalla:**

Este estado será alcanzado cuando un objeto haya caido al suelo y por tanto, el juego habrá finalizado. Se apagarán todos los leds y se esperará 1 segundo para pasar al siguiente estado.

- **Estado 4: Fin del juego - Mostrar puntuación:**

En este último estado, se mostrará la puntuación con la función ***Mostrar_puntuación(****XDriver* *Puntero_inst, short int color, int puntuacion) y también se creará un parpadeo de 1 segundo en la numeración de la puntuación.

6 CONCLUSIONES

En este último capítulo, se presenta un último resumen de los puntos que se han trabajado durante el desarrollo de este trabajo. A su vez, se mencionan las habilidades y conocimientos adquiridos, y el cumplimiento de los objetivos.

El objetivo fundamental de este trabajo fue trabajar sobre tecnologías SoC-FPGA, en concreto con la placa de desarrollo PYNQ-Z2 de Xilinx. Esta tecnología resulta novedosa con respecto a las usadas en la carrera, ya que combina la programación de sistemas empotrados con el diseño de sistemas digitales. Y desde el primer momento se tuvo como objetivo hacer este trabajo de forma práctica implementando un juego que además utiliza dos componentes adicionales (Matriz de LEDs RGB y teclado PmodKYPD) y asi demostrar experimentalmente que se ha dominado la programación de la placa.

Para cumplir este objetivo fundamental, se han realizado una serie de trabajos que han ido dando lugar al cumplimiento de objetivos parciales.

- Se realizó un estudio del SoC Zynq-7000, que es el componente principal sobre el que se ha desarrollado el trabajo. En este estudio se analizó las potencialidades de la parte hardware y de la parte software, así como la metodología de diseño a seguir. Durante este estudio se empezó a planificar que para construir el juego era necesario realizar una parte hardware y otra software y que se interconectan a través del bus de comunicación AXI4.
- Para realizar la parte hardware se aprendió el uso del software Vivado de Xilinx, trabajando sobre la versión Vivado 2022.2. Se aprendió a crear un proyecto en Vivado y construir un bloque IP personalizado, donde se tuvieron que interconectar las Entradas/Salidas correspondientes del diseño para controlar la matriz de LEDs con los registros internos del procesador del SoC. Se aprendió a crear diagramas de bloques, instanciando los bloques necesarios e interconectándolos entre sí. Estos diagramas han sido validados, sintetizados e implementados por Vivado, donde posteriormente se ha generado el Bitstream y la plataforma Hardware que son los dos archivos de mayor importancia a la hora de finalizar un diseño.
- Para la parte software, se realizó un análisis entre las opciones posibles de programación, optando por el manejo de la plataforma Vitis IDE de Xilinx, ya que permitía realizar la depuración del software desarrollado. Se trabajó también sobre la versión Vitis 2022.2. Se aprendieron los diferentes pasos necesarios para crear un proyecto genérico, así como: crear la plataforma hardware desde el diseño exportado en Vivado, y crear una aplicación basada en esta plataforma. Se ha creado una aplicación software basada en el lenguaje C, incluyendo el sistema operativo FreeRTOS para poder realizar varias tareas simultáneamente.

- Se analizaron los diferentes modos de programación de la placa, entre los cuales se eligió el modo de programación a través del puerto JTAG, ya que era el más adecuado para realizar un programa en desarrollo, donde no es necesario crear una Imagen de arranque.
- Previamente a la realización de diseños en la plataforma SoC, se desarrolló un diseño hardware en VHDL para controlar la transmisión de datos en la matriz de LEDs RGB 38x8, basado en el protocolo WS2812. El diseño se realizaó en la versión 14.7 del software ISE Design Suite de Xilinx, donde se examinó mediante simulaciones que se cumplía con rigurosidad los tiempos de transmisión requeridos por el protocolo de los LEDs.
- Complementando el diseño anterior, se realizó un segundo diseño que permitía encender los LEDs de diferentes formas según la posición de 2 de los switches de la placa. La finalidad de este diseño fue comprobar de qué forma estaba conectada físicamente una fila de LEDs con la siguiente en la matriz. Este diseño también fue verificado mediante simulaciones, aunque no se han incluido en la memoria.
- Estos dos diseños incorporaron en un único diseño y se comprobó experimentalmente su funcionamiento sobre la placa Nexys4 DDR. Tras esta comprobación, se observó que los LEDs se encendían con el color adecuado, aunque debido a la forma en la que estaban conectados los LEDs en la matriz, los LEDs de las filas pares se encendían de forma inversa, por lo que hubo que realizar unos pequeños ajustes sobre el primer diseño para corregir este error.
- Para ir comprobando los conocimientos adquiridos sobre la programación de la PYNQ-Z2, se desarrollaron dos aplicaciones, que controlaban la matriz de LEDs y el teclado PmodKYPD. Los diseños hardware implementados en Vivado, se usaron como “drivers” para controlar estos dispositivos. Y con la programación de la aplicación en C se aportó un control más amplio y la capacidad de realizar operaciones más complejas actuando sobre estos drivers.
    - La primera aplicación que se desarrolló consistió en transformar el diseño que premitía encender los LEDs en función de la posición de dos switches en una aplicación en C.
    - La segunda aplicación desarrollada controlaba el teclado PmodKYPD y mostraba en la matriz de LEDs la tecla que ha sido pulsada para verificar que se leen las teclas correctamente. Para escribir en la matriz de LEDs, se puede hacer una análogía con los píxeles de una imagen.
- Una vez que ya se aprendió a controlar la matriz de LEDs y el teclado PmodKYPD desde la PYNQ-Z2, se realizó una aplicación más compleja, para mostrar experimentalmente el dominio de la programación. La funcionalidad de este juego, descrita en la Introducción de esta memoria, consiste en realizar dos tareas fundamentales:
    - Controlar el teclado y actuar en consecuencia, cuando una tecla

se ha pulsado, independientemente de lo que esté ocurriendo en el juego

- Mantener el movimiento de los objetos que caen (LEDs en movimiento), generar nuevos objetos y acelerar o reducir su velocidad de caída según el ciclo del programa.

- Era inviable ejecutar estas dos tareas en un programa “standalone”, el cual es ejecutado directamente sobre el hardware de la placa de forma independiente sin necesidad de un sistema operativo. Por tanto, se recurrió a usar un sistema operativo FreeRTOS, el cual permite gestionar varias tareas concurrentes.
- En la aplicación en C, se ha trabajado leyendo y escribiendo datos sobre los registros internos del procesador del SoC. Estos registros se crean junto con los bloques IP en Vivado y son los que interconectan el procesador del SoC con la lógica programable. Los datos sobre los que se ha actuado son los referidos a la matriz de LEDs, y el teclado.
- En el juego también se ha creado una escena inicial sobre la matriz de LEDs, la cual indica al jugador la tecla que hay que pulsar para iniciar el juego, y otra escena final, que muestra la puntuación adquirida por el jugador.

Con todas las tareas realizadas, podemos concluir que se han cumplido los objetivos de este proyecto. Adicionalmente se ha realizado un tutorial (Apendice B), para realizar un diseño en la placa PYNQ-Z2 en el que se puede observar el funcionamiento de los LEDs fijos, uno de los LEDs RGB de la placa, los 4 botones y los 2 switches. El tutorial contiene toda la información necesaria para crear el sistema desde el diseño hardware en Vivado, hasta la aplicación software en Vitis.

REFERENCIAS

[1] «Conceptos fundamentales de los FPGA: ¿Qué son los FPGA y por qué son necesarios?», DigiKey. Accedido: 28 de diciembre de 2023. [En línea]. Disponible en: https://www.digikey.com/es/articles/fundamentals-of-fpgas-what-are-fpgas-and-why-are-they-needed

[2] «FPGA», pdfcoffee.com. Accedido: 28 de diciembre de 2023. [En línea]. Disponible en: https://pdfcoffee.com/fpga-7-pdf-free.html

[3] U. Farooq, Z. Marrakchi, y H. Mehrez, «FPGA Architectures: An Overview», en *Tree-based Heterogeneous FPGA Architectures*, New York, NY: Springer New York, 2012, pp. 7-48. doi: 10.1007/978-1-4614-3594-5_2.

[4] K. K. W. Poon, S. J. E. Wilton, y A. Yan, «A detailed power model for field-programmable gate arrays», *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, n.º 2, pp. 279-302, abr. 2005, doi: 10.1145/1059876.1059881.

[5] K. Tatas *et al.*, «FPGA Architecture Design and Toolset for Logic Implementation», en *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, vol. 2799, J. J. Chico y E. Macii, Eds., en Lecture Notes in Computer Science, vol. 2799. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 607-616. doi: 10.1007/978-3-540-39762-5_67.

[6] «0131424610_ch03.pdf» [En linea] Disponible en: https://ptgmedia.pearsoncmg.com/images/0131424610/samplechapter/0131424610_ch03.pdf.

[7] «Chip Hall of Fame: Xilinx XC2064 FPGA - IEEE Spectrum». Accedido: 29 de diciembre de 2023. [En línea]. Disponible en: https://spectrum.ieee.org/chip-hall-of-fame-xilinx-xc2064-fpga

[8] «PYNQ Overlays — Python productivity for Zynq (Pynq)». Accedido: 28 de diciembre de 2023. [En línea]. Disponible en: https://pynq.readthedocs.io/en/latest/pynq_overlays.html

[9] «axi_slave.pdf» [En linea] Disponible en: https://www.webpages.uidaho.edu/~jfrenzel/440/Handouts/Xilinx%20Vivado/Embedded%20Processor/axi_slave.pdf.

[10] S. J. E. Wilton y R. Saleh, «Programmable logic IP cores in SoC design: opportunities and challenges», en *Proceedings of the IEEE 2001 Custom Integrated Circuits Conference (Cat. No.01CH37169)*, San Diego, CA, USA: IEEE, 2001, pp. 63-66. doi: 10.1109/CICC.2001.929724.

[11] «Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator», 2023. Disponible en: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_1/ug994-vivado-ip-subsystems.pdf.

[12] «Vitis Unified Software Platform Documentation: Embedded Software Development», 2023. Disponible en: https://docs.xilinx.com/viewer/book-attachment/X1QD5u5YoN5xSktsiEhkDA/zld1saCKysgBEBLkAJe75A.

[13] «ug994-vivado-ip-subsystems-en-us-2023.1.pdf». Disponible en: https://docs.xilinx.com/viewer/book-attachment/445TxxbAsQBK08D8hEev3g/XKRXLetZ2~i~b~OBgoyUzA.

[14] «Nexys4-DDR_rm.pdf». Disponible en: https://www.xilinx.com/support/documents/university/XUP%20Boards/XUPNexys4DDR/documentation/Nexys4-DDR_rm.pdf.

[15] «PYNQ - Python productivity for Zynq», PYNQ - Python productivity for Zynq. Accedido: 28 de diciembre de 2023. [En línea]. Disponible en: http://www.pynq.io/

[16] «XUP PYNQ-Z2», AMD. Accedido: 15 de febrero de 2024. [En línea]. Disponible en: https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html

[17] «WorldSemi WS2812B Digitale 5050 RGB LED - Matrix 32x8 - Flexible». Accedido: 15 de febrero de 2024. [En línea]. Disponible en: https://www.tinytronics.nl/en/lighting/matrix/ws2812b-digitale-5050-rgb-led-matrix-32x8-flexible