Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Trabajo Fin de Grado

IMPLEMENTACIÓN DE PERIFÉRICOS EN VIVADO PARA DISPOSITIVOS ZYNQ

Autora: Sara Ortega Lázaro

Tutor: Ignacio Bravo Muñoz

2014

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Trabajo Fin de Grado

IMPLEMENTACIÓN DE PERIFÉRICOS EN VIVADO PARA DISPOSITIVOS ZYNQ

Autora: Sara Ortega Lázaro

Director: Ignacio Bravo Muñoz

TRIBUNAL:

Presidente: Alfredo Gardel Vicente

Vocal 1º: Pedro Martín Sánchez

Vocal 2º: Ignacio Bravo Muñoz

CALIFICACIÓN: .....
FECHA: .....

Nobody said it would be easy, they just promised it would be worth it.

Harvey Mackay.

AGRADECIMIENTOS

Me gustaría dedicar un espacio de este trabajo para dar las gracias a todas aquellas personas que me han mostrado su apoyo, no sólo durante este proyecto sino a lo largo de mi carrera.

En primer lugar, dar las gracias a mi familia. A mis padres, José Alberto y Juani, por estar ahí celebrando cada éxito y compartiendo cada fracaso, por enseñarme que el esfuerzo es la clave del éxito y creer en mí incondicionalmente. Y a lo mejor que tengo, mi hermano pequeño, Álvaro, por su habilidad de hacerme reír en los malos momentos, y por quitarme méritos en los buenos.

Gracias también a mis compañeros de clase, con los que he compartido miles de emociones a lo largo de estos años. Ellos han hecho que el camino fuera más llevadero y han dado color a los días más grises. En especial quiero nombrar a Cristian, Diego, Miguel Ángel y Marta porque hemos hecho un buen equipo y, lo que es más importante, una gran amistad.

A mis amigos de Guadalajara, porque ellos eran la vía de escape ante el cansancio de la semana, el consuelo de que al final de una semana dura siempre habría tiempo para divertirme junto a ellos.

A Diego, por darme ánimos en este sprint final y hacer un poquito mejor mi día a día.

Gracias a César, por su ayuda y paciencia en los momentos difíciles que suponen comenzar a trabajar con algo nuevo.

Y finalmente, gracias a mi tutor Nacho por proponerme este reto y en general a todos los profesores que he tenido la oportunidad de conocer a lo largo de la carrera, por los conocimientos que me habéis aportado cada uno y los valores personales que puedo llevarme de muchos.

GRACIAS a todos por vuestro granito de arena, porque gran parte de este éxito os pertenece.

ÍNDICE DEL TRABAJO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS Y ECUACIONES

ÍNDICE DE CÓDIGO

RESUMEN

La familia Zynq de Xilinx combina en un mismo circuito integrado un microprocesador (parte software) y una zona de lógica programable (parte hardware). Para su diseño, Xilinx proporciona herramientas de desarrollo agrupadas bajo el nombre ISE Design Suite. Por otro lado, Xilinx ofrece Vivado HLS (High-Level Synthesis) como herramienta de síntesis de alto nivel que permite trabajar a un elevado nivel de abstracción.

Durante este trabajo se emplearán ambas herramientas para diseñar un periférico controlado por procesador a través de la interfaz AXI4-Lite que los comunica. Se estudiará su funcionamiento, ventajas, y se analizará la implementación más óptima en el caso de Vivado HLS.

Palabras clave: Zynq, EDK, Vivado HLS, Directivas de optimización, Periférico

ABSTRACT

Xilinx Zynq family combines software programmability of a processor with the hardware programmability of an FPGA in the same integrated circuit. For its design, Xilinx provides a broad range of development system tools, collectively called the ISE Design Suite. Xilinx also provides Vivado HLS as high-level synthesis tool that allows working at a higher level of abstraction.

Along this work both of these tools will be used for creating a custom peripheral in the programmable logic portion and control it via AXI4-Lite interface. It will be studied how these tools work, their advantages, and the most optimal implementation will be analyzed with Vivado HLS.

Keywords: Zynq, EDK, Vivado HLS, Optimization directives, Peripheral

RESUMEN EXTENDIDO

Recientemente Xilinx ha lanzado al mercado su nueva familia de dispositivos Zynq-7000 Extensible Processing Platform, basada en la arquitectura All Programmable System-on-Chip, lo que supone una innovación dentro de la industria electrónica al ofrecer un sistema totalmente encapsulado, acercándose así a un mayor margen de audiencia que no únicamente a diseñadores hardware.

Este dispositivo combina una parte software (PS) compuesta principalmente por un doble núcleo ARM Cortex-A9, además de múltiples periféricos I/O, memoria on-chip, interfaz de memoria externa, caches y controlador DMA, entre otros, con una parte de lógica programable (PL) en un mismo circuito integrado.

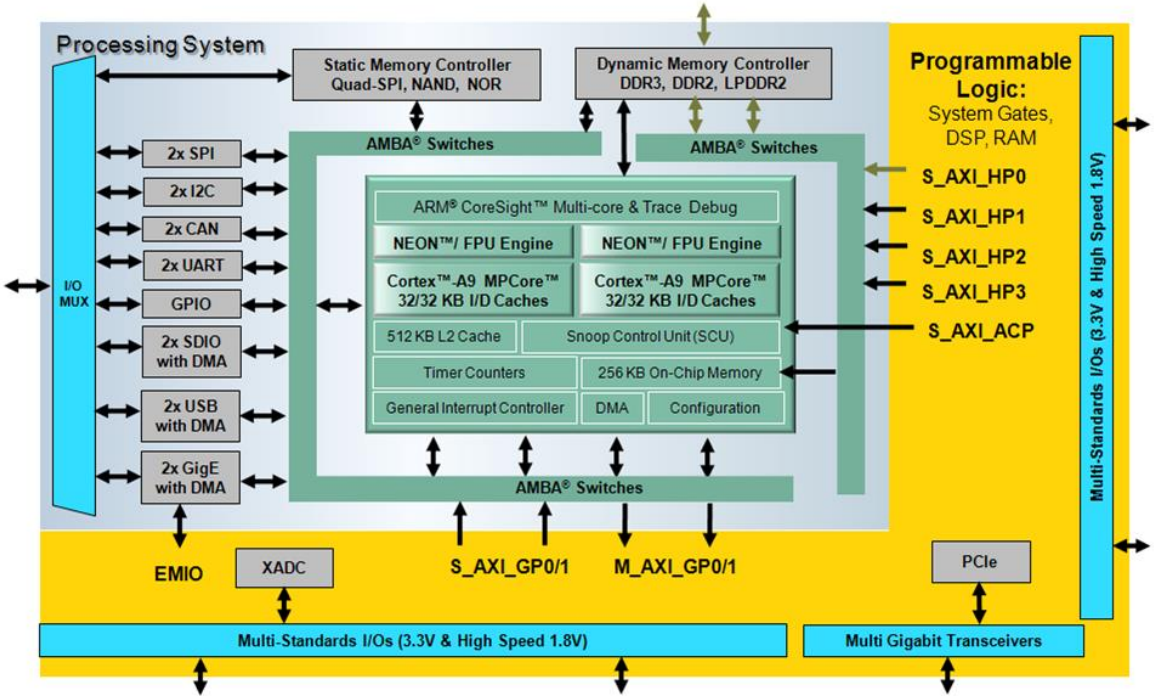


FIGURA 1. DIAGRAMA DE BLOQUES ZYNQ-7000 AP SOC [12]

Esta asociación entre FPGA y microprocesadores no es algo nuevo, pues ya a finales de los 90 los fabricantes de FPGAs comenzaron a ofrecer *soft-cores* y *hard-cores* que los diseñadores hardware podían programar en las FPGAs mediante las herramientas de síntesis y *place-and-route*. Más tarde a principios del siglo XXI los vendedores de FPGAs consiguen implementar los microprocesadores en el propio encapsulado, próximos a los bloques de lógica programable. Esto conlleva un ahorro de espacio en el chip para la lógica programable, mayor velocidad de procesado, un buen rendimiento y un bajo consumo. Esta estrategia fue la empleada en las FPGAs basadas en los procesadores Power-PC (*hard-core*) y Microblaze (*soft-core*), cuyo éxito sirvió de

base para esta nueva tecnología que se espera revolucione el mercado con este novedoso concepto de un sistema de procesador completamente encapsulado junto a la parte lógica [12].

Cabe destacar que uno de los objetivos buscados en la arquitectura Zynq-7000 es despertar también el interés de desarrolladores software asegurando el protagonismo del procesador. Es decir, cuando los usuarios encienden el dispositivo, el procesador arranca y espera órdenes para programar una porción de lógica programable. De hecho, una de las ventajas con las que cuenta este diseño es que, si el usuario lo desea, puede controlar el dispositivo como un procesador independiente y no trabajar con la parte de lógica programable del chip. Sin embargo, el gran valor añadido de Zynq-7000 es que los usuarios tienen la posibilidad de descargar funciones de procesado a la FPGA, lo cual les permitiría crear sistemas más óptimos en cuanto a funcionalidad, rendimiento y potencia [12].

Como ya se ha comentado, con la creación de este producto, Xilinx busca que el dispositivo no sólo tenga acogida entre su usuario tradicional, como son los diseñadores de hardware, sino que espera que sea también empleado por diseñadores de software. Por este motivo, proporciona herramientas de diseño que pretenden hacer la programación del dispositivo familiar para ambos campos.

Estas herramientas se agrupan bajo el nombre ISE Design Suite, proporcionada por Xilinx para los dispositivos All Programmable (AP), que incluyen *7 series* y la familia *Zynq-7000 AP SoC*, y se puede encontrar en las siguientes tres ediciones:

- *Embedded Edition*, que contiene el conjunto ISE (Integrated Software Environment), la herramienta de análisis PlanAhead, ChipScope Pro para el depurado de diseños FPGA y la herramienta EDK (Embedded Development Kit).

- *System Edition*, con el mismo contenido que Embedded Edition pero añade System Generator for DSP, para el diseño de sistemas DSP de alto rendimiento usando dispositivos *All Programmable* de Xilinx.

- WebPack Edition, que es la opción más básica, pues incluye únicamente el conjunto ISE y las opciones EDK, estando limitadas para los tres dispositivos más bajos de la familia Zynq.

De estas tres ediciones se va a trabajar con la edición más completa, System Edition, donde se va a emplear la herramienta de diseño EDK, y la herramienta PlanAhead como herramienta *top-level* para coordinar los diseños hardware y software.

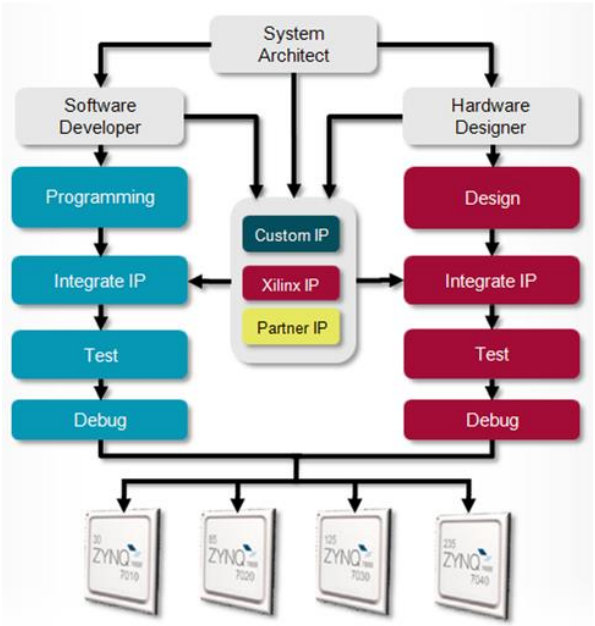


FIGURA 2. FLUJO DE DISEÑO DE UN SISTEMA PARA ZYNQ-7000 AP SOC [12]

Como se decía, la herramienta principal de la que dispone Xilinx para el diseño de sistemas con Zynq-7000 AP SoC se denomina Embedded Development Kit (EDK), y está compuesta por una plataforma para el desarrollo hardware denominada Xilinx Platform Studio (XPS) y una plataforma para el diseño software denominada Software Development Kit (SDK), basada en Eclipse y ya familiar para los desarrolladores de software.

A la hora de realizar un diseño se parte de la herramienta XPS donde comenzar creando un diseño hardware. Dispone del asistente Base System Builder (*BSB wizard*), que crea la base del proyecto hardware conteniendo los elementos básicos para comenzar a construir un sistema más complejo. Además, cuenta con una herramienta para el diseño de periféricos denominada IPIF, donde a través de módulos IPIF (AXI IP Interface) se implementa la interfaz entre AXI4 Interconnect y la lógica de usuario. Posteriormente con la herramienta IPIC (AXI IP Interconnect), se definen las señales que comunican al periférico con su módulo IPIF anteriormente definido.

Una vez terminado el diseño hardware se emplea la herramienta SDK para añadir funcionalidad software al módulo hardware creado en XPS. Dispone del asistente Board Support Package (*BSP*), donde generar una plataforma de diseño software sobre el módulo hardware importado al proyecto.

Por otro lado, PlanAhead es el gestionador de proyectos global, capaz de lanzar las herramientas XPS y SDK llevando un control de los ficheros necesarios y generados para cada una de ellas y asociando ambos diseños en un proyecto común. Su uso no es estrictamente necesario para llevar a cabo el diseño pero se recomienda para el desarrollo de sistemas empotrados usando Zynq.

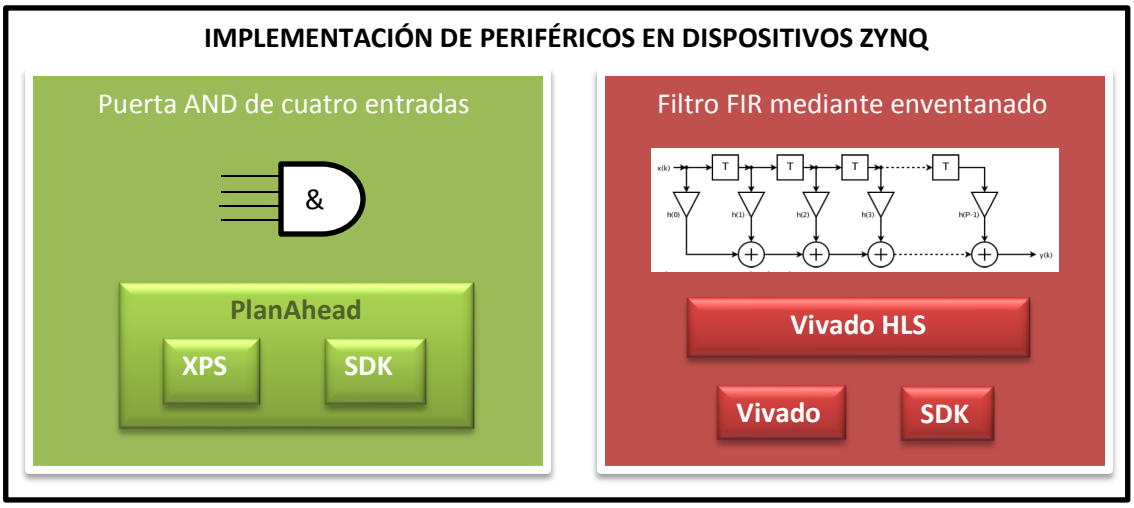


FIGURA 3. SECCIONES QUE COMPONEN LA MEMORIA

La Figura 3 muestra de forma gráfica los bloques en que está dividida la memoria.

La primera parte de este trabajo se centra en el estudio de estas herramientas en conjunto para la realización de un periférico creado por el usuario, en este caso una puerta and de cuatro entradas, detallando el procedimiento a seguir desde la opción en XPS para la creación de periféricos hasta el software desarrollado en SDK para la comunicación con el procesador. A continuación, la memoria se centrará en una nueva herramienta de Xilinx, Vivado High-Level Synthesis, para su estudio en profundidad, creando como periférico un filtro FIR (*Finite Impulse Response filter*) y detallando aspectos de su comportamiento que permiten conseguir la implementación más óptima del diseño creado.

Con Vivado HLS el usuario dispone de mayores ventajas que con la herramienta EDK comentada anteriormente, pues permite trabajar en lenguajes de alto nivel, como es ANSI-C, dejando a un lado los lenguajes de descripción hardware HDL (*Hardware Description Languages*). Esto supone una mayor rapidez de diseño evitando entrar en detalles de implementación, dado que su metodología de trabajo se basa en transformar descripciones C a implementaciones RTL para sintetizarlas en una FPGA.

Además Vivado HLS ofrece un conjunto de directivas para la síntesis con el propósito de optimizar el código sin necesidad de modificarlo y encontrar así la implementación más óptima del diseño.

En cuanto a la comunicación con el procesador, Vivado HLS crea un conjunto de drivers en el momento de exportación en bloque IP de la implementación RTL realizada, que posteriormente son usados en la herramienta SDK para desarrollar el software para la comunicación.

En definitiva, Zynq es una arquitectura centrada en el procesador que permite diseñar sistemas más inteligentes que combinan el rendimiento de un ASIC (*Application Specific Integrated Circuits*), la flexibilidad de una FPGA y la facilidad de programación de un microprocesador.

1. PLANTEAMIENTO Y OBJETIVOS

El objetivo de este proyecto es sacar partido a las ventajas que ofrece este nuevo dispositivo, logrando implementar tanto diseño hardware como software en un mismo chip de forma que, diseños que hasta ahora se habían realizado únicamente sobre microprocesador o sobre FPGA, puedan ser portados de forma independiente o conjunta sobre el mismo encapsulado.

Como punto de partida, se realiza un estudio de la tarjeta Zedboard (Zynq Evaluation & Development Board) que se va a emplear para este trabajo, detallando su arquitectura dividida en dos grandes partes, los bloques que la componen, la interfaz de comunicación entre ambas partes, y las ventajas que supone hacer uso de esta tarjeta.

Una vez estudiada la tarjeta a emplear, se va a estudiar el funcionamiento de las diferentes herramientas que proporciona Xilinx para trabajar sobre Zynq y se implementarán diseños crecientes en complejidad que permitan dar una visión clara al usuario de cómo puede trabajar con este dispositivo haciendo uso de estos recursos.

- En primer lugar, se realiza un estudio del conjunto de herramientas EDK que comprende XPS y SDK, y la herramienta PlanAhead como herramienta *top-level* controlando las distintas etapas del diseño y pudiendo invocar tanto a XPS como a SDK según corresponda.

  Empleando estas herramientas, se va a comenzar realizando un diseño que sirva de toma de contacto para el usuario. Este diseño estará compuesto por el procesador más un puerto para la comunicación serie (UART) en la parte PS (*Processing system*) y un par de periféricos GPIO (uno para *switches* y otro para botones) obtenidos del catálogo de XPS en la parte PL (*Programmable logic*), tal como puede verse en la Figura 4.

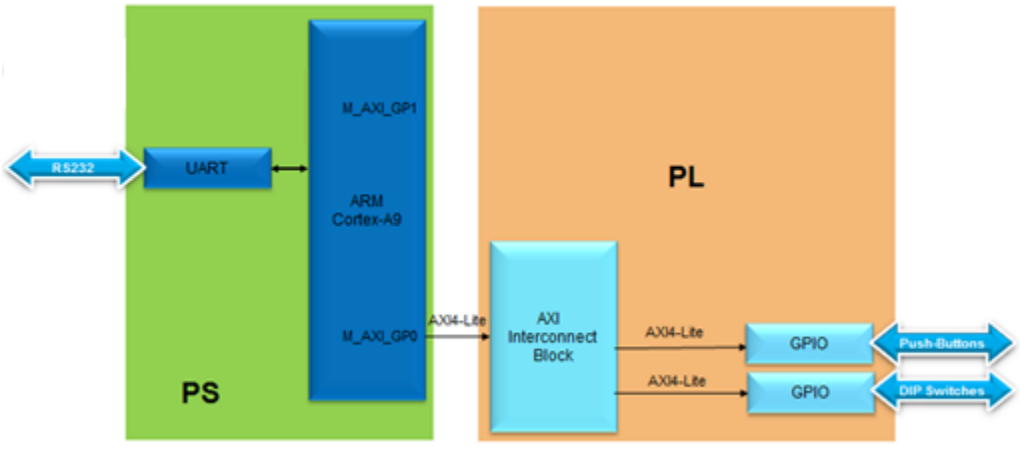


FIGURA 4. BLOQUES DE DISEÑO A ESPECIFICAR EN XPS

Sobre este diseño se creará una aplicación software que controle el estado de los periféricos GPIO desde el procesador y verifique el funcionamiento del diseño.
El procedimiento a seguir puede verse en la Figura 5.

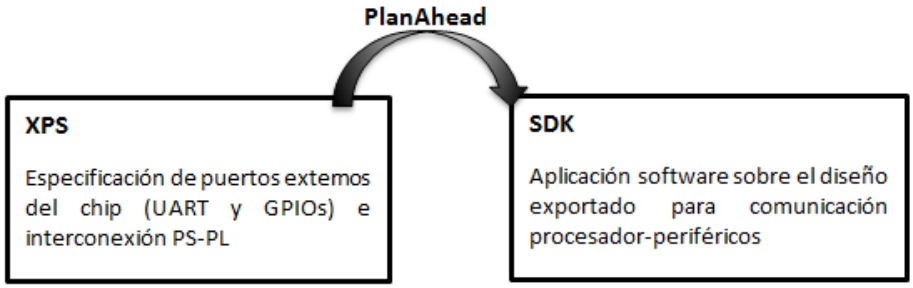


FIGURA 5. PROCEDIMIENTO A SEGUIR EN EL PRIMER DISEÑO

Posteriormente, se va a crear un periférico por el usuario desde la herramienta de XPS que consiste en una puerta AND de cuatro entradas, tres de ellas externas dadas por tres switches de la tarjeta y otra de ellas interna contenida en un registro compartido entre las partes PS y PL. Una vez creado se instanciará en la parte PL y se verificará su funcionamiento en hardware controlándolo desde el microprocesador mediante una nueva aplicación software en la cual se hará uso de los drivers creados para la comunicación con el periférico.

- En segundo lugar, se realiza un estudio de la herramienta Vivado High-Level Synthesis, donde primero se analizará el comportamiento de dicha herramienta para comprender de qué manera realiza la transformación de una descripción C a una implementación RTL (Register-Transfer Level). Además para afianzar los conceptos de esta base teórica se realiza un ejemplo de aplicación de directivas consistente en una multiplicación de matrices sobre la cual se van implementando diferentes soluciones observando el impacto que conlleva la aplicación de una directiva a la síntesis en función del momento y el lugar.

  A continuación, empleando lenguaje de alto nivel, se va a crear un periférico, que en este caso se tratará de un filtro FIR, el cual será descrito en lenguaje C en Vivado HLS. Una vez encontrada la implementación más óptima mediante el uso de directivas de optimización, será exportado como diseño RTL a Vivado Design Suite, donde se realizará un diseño en HDL (Hardware Description Languaje) de la parte PS con el periférico importado.

  Finalmente, se creará una aplicación en SDK para probar su funcionamiento comunicando el procesador con el periférico a través de la interfaz AXI4-Lite que los comunica mediante los drivers creados en el momento de exportación de Vivado HLS.

La Figura 6 muestra las fases por las que va pasando el diseño hasta introducirlo en la tarjeta Zedboard.

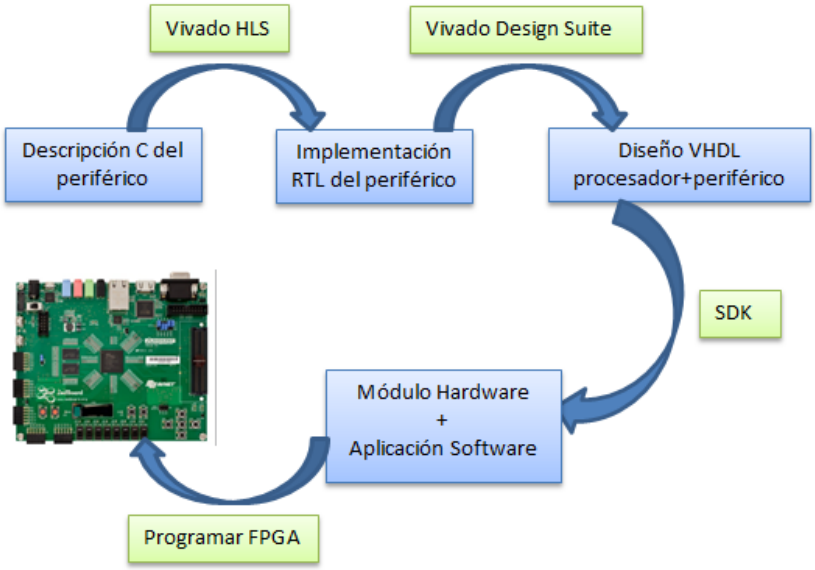


FIGURA 6. FASES DEL DISEÑO A REALIZAR EN VIVADO HLS

2. INTRODUCCIÓN A LA TARJETA ZEDBOARD

DESCRIPCIÓN

ZedBoard, proveniente de las siglas Zynq Evaluation & Development Board, es una tarjeta de desarrollo de bajo coste de la familia de dispositivos Zynq-7000 All Programmable SoC.

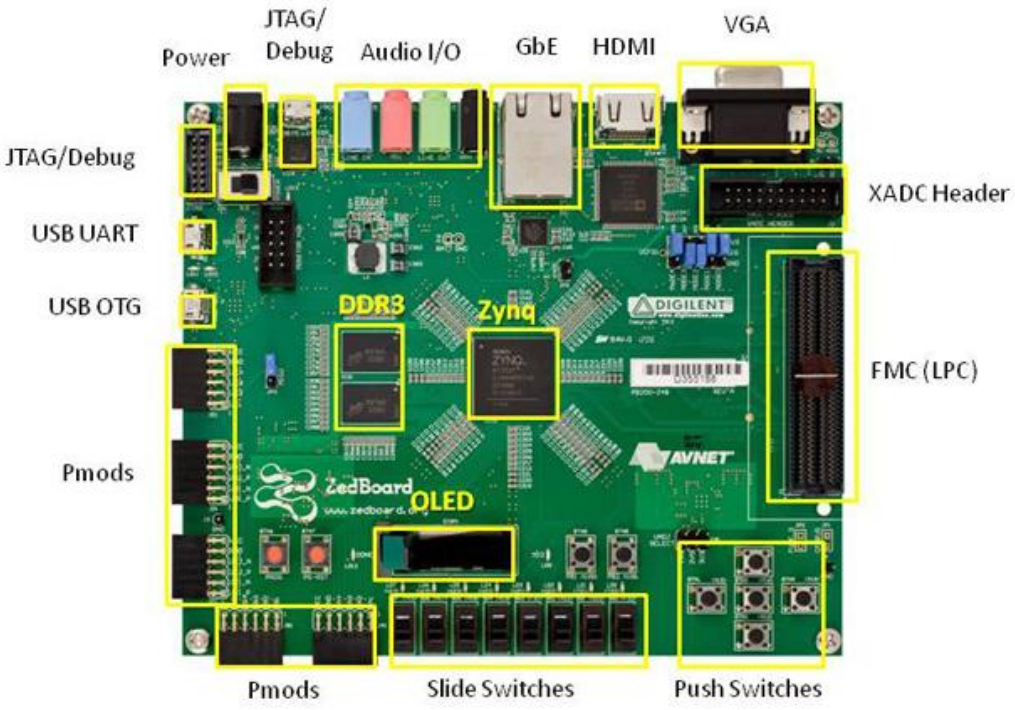


FIGURA 7. COMPOSICIÓN DE LA TARJETA ZEDBOARD

Su arquitectura se divide en dos grandes áreas:

- Parte PS (Processing System) donde se encuentran los dos procesadores ARM Cortex-A9, cachés, controladores de memoria, la señal de reloj y de reset, periféricos de entrada y salida, DMA, temporizadores, el controlador de interrupciones y la memoria RAM.

- Parte PL (Programmable Logic) donde se encuentran los periféricos GPIO (General Purpose IO) compuestos por 8 LEDs, 7 *push buttons* y 8 *switches*, el conector JTAG, displays, y puertos para comunicación USB-JTAG, USB-UART y Ethernet. Está basada en la lógica programable de la serie 7, como los dispositivos Artix y Kintex.

En la Figura 8 puede verse el diagrama de bloques correspondiente a la tarjeta ZedBoard [6].

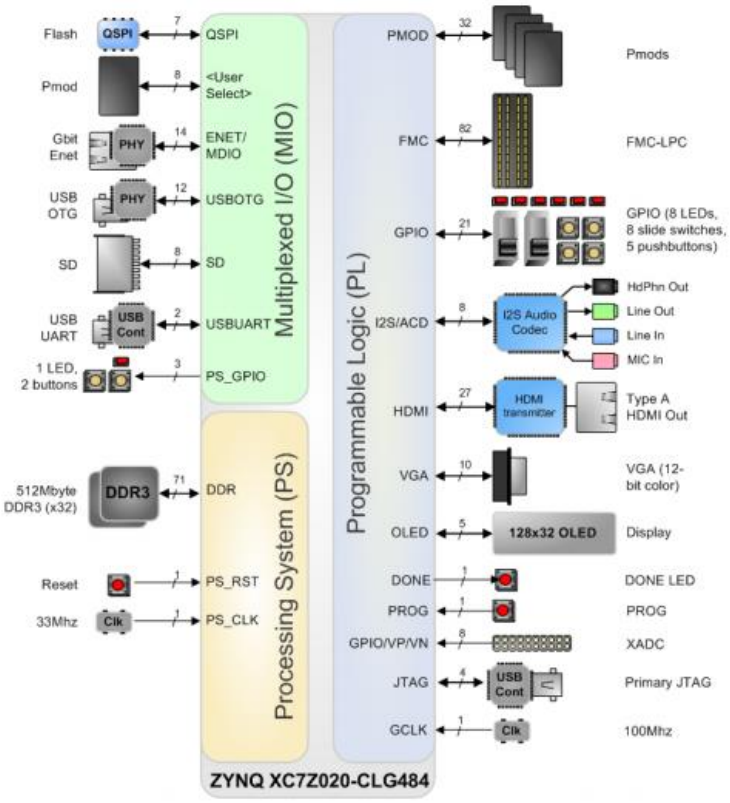


FIGURA 8. DIAGRAMA DE BLOQUES DE LA TARJETA ZEDBOARD

En cuanto a su modo de operación, son sistemas basados en procesador, lo que quiere decir que la parte PS arranca primero, cargando la configuración del primer estado de arranque (*First Stage Boot Loader*) de la memoria flash externa. Una vez ha arrancado, la parte PS espera órdenes para configurar la parte PL mediante una aplicación software que accede a la unidad de configuración del dispositivo hardware.

Ambas partes de la arquitectura Zynq, PS y PL, emplean la interfaz AXI (*Advanced Extensible Interface*) para la comunicación entre ellas, que es parte de la arquitectura AMBA (*Advanced Microcontroller Bus Architecture*) de ARM. Esta interfaz proporciona un alto rendimiento a través de conexiones punto a punto y tiene canales separados para implementar las interfaces de lectura y escritura de forma independiente.

Dentro de la interfaz AXI se encuentran las interfaces: Memory Map (AXI4), para ráfagas de múltiples datos o envío de una sola dirección de memoria; Streaming (AXI4-Stream), sólo para datos en modo ráfaga; y Lite (AXI4-Lite), donde no está permitida la ráfaga y se emplea para el envío de una sola dirección de memoria o dato [13].

La interfaz AXI4-Lite será la empleada para la comunicación maestro-esclavo a lo largo del proyecto dado que no interesa el modo ráfaga y es la más sencilla de implementar, habilitándose desde el lado PS con la interfaz M_AXI_GP0 y desde el lado PL con la interfaz S_AXI a través del módulo axi_interconnect, como se verá más adelante.

VENTAJAS

Zynq-7000 All Programmable SoC es la plataforma ideal para infundir inteligencia a los sistemas emportrados de hoy en día. Al ser *All Programmable* no sólo se le puede dotar de inteligencia a través de la programación del software sino que además se puede ejecutar el procesado de datos y decisiones en tiempo real a través del hardware programable y las interfaces del sistema se pueden optimizar y desarrollar a través de E/S programable [14].

Estos tres aspectos programables de los dispositivos Zynq hacen que se considere la forma más rápida y eficaz de crear sistemas más inteligentes capaces de hacer frente a los problemas de diseño que plantea el mercado.

A ello se suman los bajos costes de diseño, la alta flexibilidad para modificarlo, un alto rendimiento y bajo consumo.

Existen diversas razones que hacen de este dispositivo una buena elección [30]:

- ✓ Ambas partes software y hardware del chip se comunican mediante puertos AXI de AMBA, acoplándose de forma muy eficiente estas dos partes de la arquitectura Zynq. En total existen 9 interfaces AXI: 2 interfaces AXI master de 32 bits, 2 interfaces AXI slave de 32 bits, 4 interfaces AXI slave configurables y de alto rendimiento, de 64 bits, y una interfaz AXI ACP (Accelerator Coherency Port) de 64 bits.

- ✓ Soporta un amplio rango de sistemas operativos para manejar el hardware, entre los cuales se incluyen muchas variantes de Linux. Además dispone de una larga lista de herramientas de Xilinx para desarrollo software.

- ✓ Cumple con los estrictos requisitos de seguridad y confianza que exige un sistema inteligente.
  Respecto a seguridad, la plataforma Zynq, como se comentaba anteriormente, siempre arranca el lado de procesador primero, y si se desea, esta secuencia de arranque puede contener autenticación de usuario, encriptación y autenticación de datos, de forma que, a partir de una password, el código es desencriptado y almacenado en la memoria on-chip donde se ejecuta fuera de peligro.
  Respecto a confianza, uno de los parámetros que la caracterizan es conocido como SEUs (*single-event upsets*) que pueden darse en errores de memoria o errores en el sistema.

Estas tarjetas han sido probadas contra estos síntomas resultando tener la tasa más baja de estos síntomas entre cualquier tecnología basada en SRAM del mercado. Otro aspecto que brinda confianza es que estos dispositivos Zynq contienen todos un sensor de temperatura y conversores A/D de forma que controlan su propio ambiente incluyendo temperatura y voltaje de alimentación.

- ✓ Alto rendimiento, debido no sólo a los procesadores más rápidos sino también a un buen rendimiento del sistema de memoria, que consigue mediante sus controladores de memoria SDRAM, con anchos de 16 y 32 bits con paridad y de 16 bits con ECC (*Error-correcting code*).

- ✓ Permite el diseño en dispositivos Zynq a cualquier nivel de abstracción, software o hardware. Se consigue alta productividad gracias a la herramienta Vivado HLS, donde los desarrolladores pueden escribir rápidos algoritmos en C, C++ o SystemC, probarlos e implementarlos únicamente recompilando el código para correr en una plataforma Zynq. Después, para añadir más velocidad, se necesita una implementación hardware, y eso se consigue mediante la exportación del diseño HLS en código HDL.

- ✓ Gran número de bloques IP, kits de diseño y plantillas de referencia para ayudar a los equipos de diseño en la creación de nuevos sistemas.

Y por si hiciesen falta más razones, la familia Zynq-7000 ha ganado diversos premios y obtenido prestigiosas calificaciones a finales de 2012, como el premio *"Best Embedded Processor"* de Microprocessor Report [22] y el premio "*Product of the Year"* por la revista Electronic Products [23], ambos haciendo alusión al abanico de oportunidades que ofrece para los diseñadores y su combinación acertada de tecnología software y hardware.

3. HERRAMIENTAS DE DISEÑO DE XILINX

3.1. EMBEDDED DEVELOPMENT KIT (EDK)

3.1.1. **Introducción**

Como ya se comentó, ante la aparición de Zynq la empresa XILINX puso un esfuerzo adicional en proporcionar herramientas de diseño que facilitasen el trabajo tanto a diseñadores hardware como a desarrolladores software, de forma que el producto tuviese buena acogida en ambos campos.

Con este propósito, la empresa proporciona un amplio rango de herramientas para el desarrollo del sistema, las cuales se engloban de forma colectiva bajo el nombre ISE (*Integrated Software Environment*). Para el desarrollo de sistemas empotrados, ISE contiene la herramienta de alto nivel EDK (*Embedded Development Kit*) [17]. Esta plataforma software permite diseñar un sistema empotrado completo para implementarlo en un dispositivo FPGA, y se compone de la plataforma XPS (*Xilinx Platform Studio*) y el software SDK (*Software Development Kit*).

Mediante el software XPS se puede diseñar la parte hardware del sistema, además incluye una interfaz gráfica de usuario con un conjunto de herramientas que sirven como guía durante el proceso de diseño. Por otro lado, el software SDK facilita el desarrollo de aplicaciones software para la plataforma hardware creada previamente en XPS. Está basado en la plataforma Eclipse, ya familiar para los desarrolladores de software.
La Figura 9 muestra la interacción que existe entre estas herramientas.

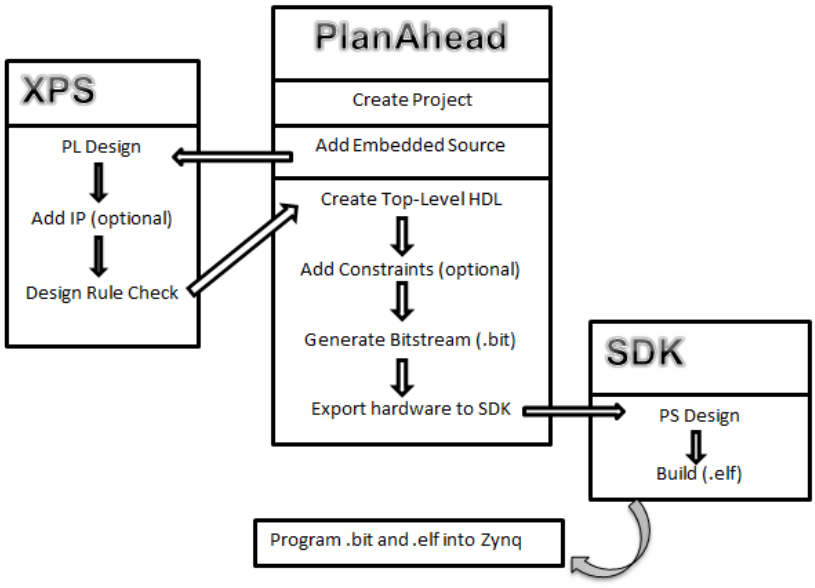


FIGURA 9. PROCESO DE DISEÑO ENTRE LAS HERRAMIENTAS EDK Y PLANAHEAD

3.1.2. **Instalación**

Las herramientas con las que se va a trabajar requieren ser descargadas e instaladas en el equipo de trabajo, lo que puede resultar tedioso si se desconoce la versión necesaria para trabajar, por ello en este apartado se pretende guiar al usuario en esta labor de arranque.

Estas herramientas están disponibles para su descarga e instalación en la página web de Xilinx accediendo a *Support>Downloads*, aunque previamente es necesario registrarse accediendo a *Sign In* e introduciendo los datos personales solicitados en el formulario. Esta cuenta de usuario Xilinx será la empleada para descargar las herramientas y para solicitar una licencia de uso una vez realizada la instalación. Otra opción es realizar la instalación desde el DVD que trae la tarjeta Zedboard para instalación de ISE Design Suite 14 y Vivado Design Suite 2012.

Desde Xilinx, en la sección *ISE Design Tools* se debe escoger la versión deseada y seleccionar el tipo de descarga. Para este proyecto se ha empleado la versión 14.3 de ISE descargando el pack *"Vivado and ISE Design Suites – 2012.3 Full Product Installation"*, no obstante pueden usarse nuevas versiones siempre que éstas permitan trabajar con EDK y PlanAhead. Además, es posible trabajar con ISE Webpack a pesar de estar ésta limitada a los tres dispositivos más pequeños de la familia Zynq, Z-7010, Z-7020, y Z-7030.

Una vez completado el proceso de descarga debe lanzarse el asistente de instalación abriendo la aplicación xsetup.exe que se encuentra en el WinRar descargado. Dentro de este asistente es donde se elige el tipo de edición a instalar, en el cual se ha elegido instalar *System Edition+SDK* por tratarse ésta de la edición más completa, sin limitación de dispositivos. Se comprueba el resumen final de componentes y su localización de instalación, y se siguen las indicaciones del asistente hasta completarse el proceso de instalación, momento en el cual aparece el asistente para la obtención de licencia. En la pestaña *Acquire a License* debe seleccionarse *Get My Purchased License* y conectar a la página de Xilinx accediendo a ella con la cuenta creada inicialmente. Xilinx muestra ahora una interfaz bajo el nombre *Create a New License File* con los productos descargados por el usuario, tal que para generar la licencia se debe marcar el producto correspondiente y seleccionar abajo *Generate Node-Locked License*, apareciendo en pantalla el producto seleccionado, información del equipo y un hueco para comentarios que se debe rellenar con alguna frase identificativa del tipo *"Zynq License for ISE and Vivado"* y hacer clic en *Next*. Al finalizar, la licencia será enviada al email registrado.

Ahora, desde el asistente para obtención de licencia lanzado durante la instalación se debe cambiar a la pestaña *Manage Xilinx Licenses* y hacer clic en el botón *Load License…* referenciando el archivo .lic recibido en el email proveniente de Xilinx. Si todo está correcto se observará que la licencia ha sido aceptada y el producto posee ahora licencia permanente. Al cerrar este asistente finalmente la instalación termina mostrándose un mensaje de éxito.

Otra alternativa hubiera sido hacer uso del código para obtención de licencia de la edición WebPack siguiendo las instrucciones que trae la tarjeta Zedboard.

El pack instalado contiene ISE 14.3 y Vivado 2012.3. Dentro de ISE se encuentran tanto EDK, compuesta por XPS y SDK, como PlanAhead. Lo conveniente ahora es localizar la carpeta de instalación, comprobar que las herramientas necesarias se encuentran accesibles como aplicación y situar accesos directos a éstas desde el escritorio para trabajar con mayor comodidad.

3.1.3. **PlanAhead**

Es la herramienta "top" con la que se va a trabajar [18][19]. Ésta debe ejecutarse cada vez que se vaya a crear un nuevo proyecto o trabajar sobre un proyecto ya creado, y desde ella se invocarán las diferentes herramientas necesarias.

Cuando se lanza la aplicación aparece la siguiente ventana de inicio.

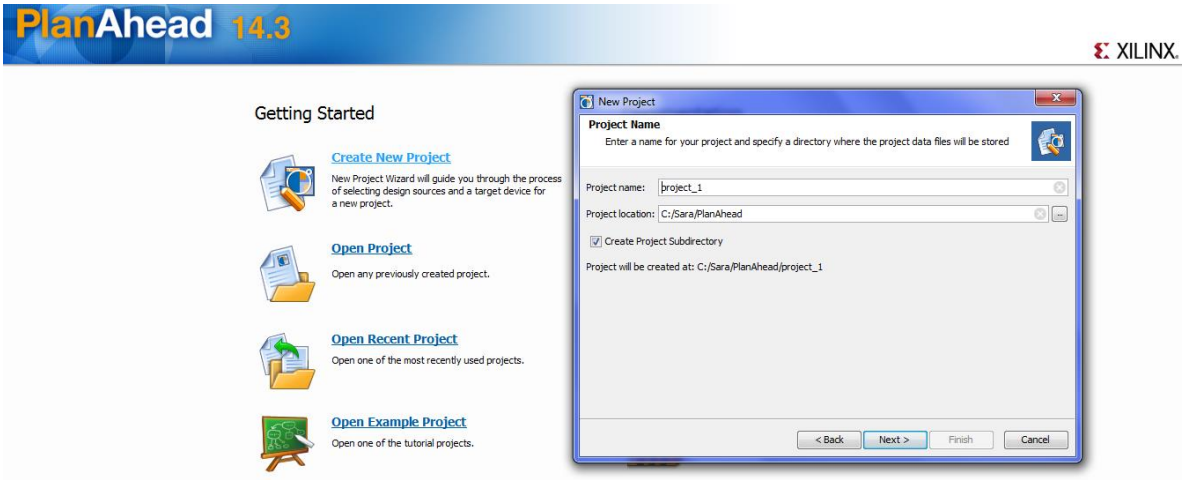


FIGURA 10. VENTANA DE INICIO DE PLANAHEAD

Esta ventana de bienvenida muestra las opciones más comunes como son *Create New Project*, para comenzar un nuevo diseño, y *Open Project* u *Open Recent Project* para continuar trabajando en un diseño ya creado.

Si se selecciona *Create New Project* aparece la ventana de *New Project* mostrada a la derecha en la Figura 10, donde se debe indicar nombre y localización del proyecto, así como marcar *Create Project Subdirectory* para tener un directorio de trabajo. Al hacer clic en *Next* se van sucediendo el resto de ventanas para las opciones de creación de un nuevo proyecto. *RTL Project*, lenguaje VHDL, dejar *Adding Sources*, *Adding IP* y *Adding Constraints* por defecto (ya que se añadirán una

vez creado el proyecto cuando sea necesario), y escoger la tarjeta *Zedboard Zynq Evaluation and Development Kit*.

En la ventana *Project Type* de la Figura 11 se selecciona tipo de proyecto RTL Project, que es el correspondiente a nuestro propósito.

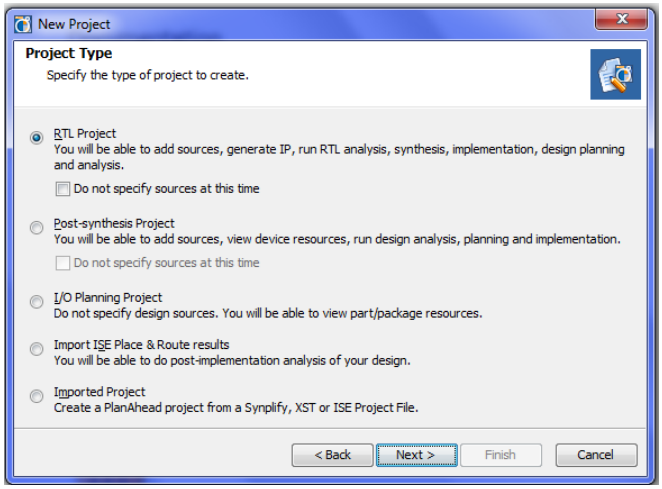


FIGURA 11. ASISTENTE DE CREACIÓN DE PROYECTOS EN PLANAHEAD (I)

En la ventana siguiente se va a dejar vacío el apartado de Add Sources puesto que en este momento no se quiere añadir ningún fichero ya generado sino que se va a generar dentro de la herramienta. Debe escogerse lenguaje VHDL como se observa en la Figura 12.

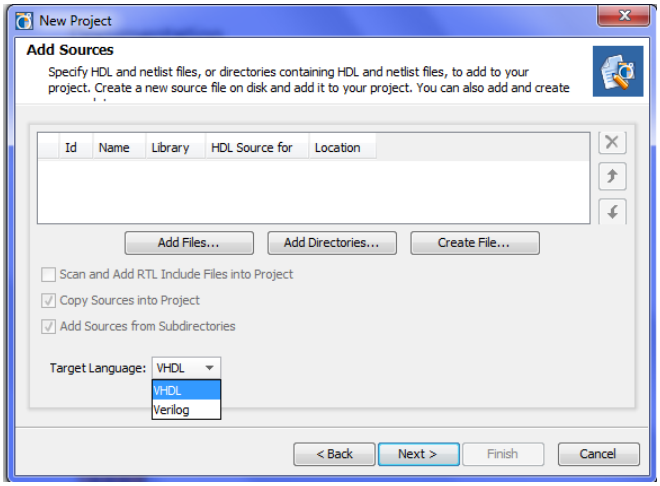


FIGURA 12. ASISTENTE DE CREACIÓN DE PROYECTOS EN PLANAHEAD (II)

De igual forma, en las ventanas *Add Existing IP* y *Add Constraints* no se especificará nada por el momento, como puede verse en la Figura 13.

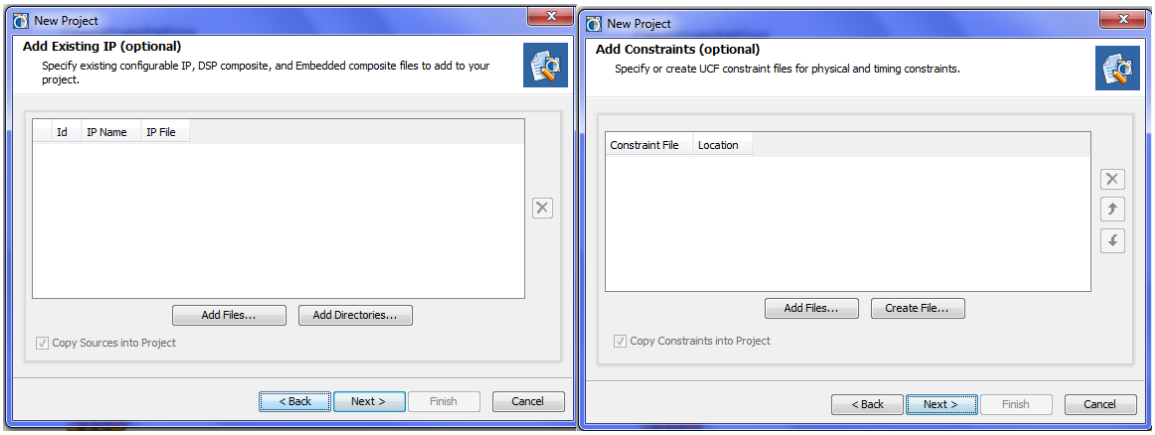


FIGURA 13. ASISTENTE DE CREACIÓN DE PROYECTOS EN PLANAHEAD (III)

En la siguiente pestaña debe escogerse la tarjeta *Zedboard Zynq Evaluation and Development Kit* de la familia Zynq-7000 en el apartado de *Boards*.

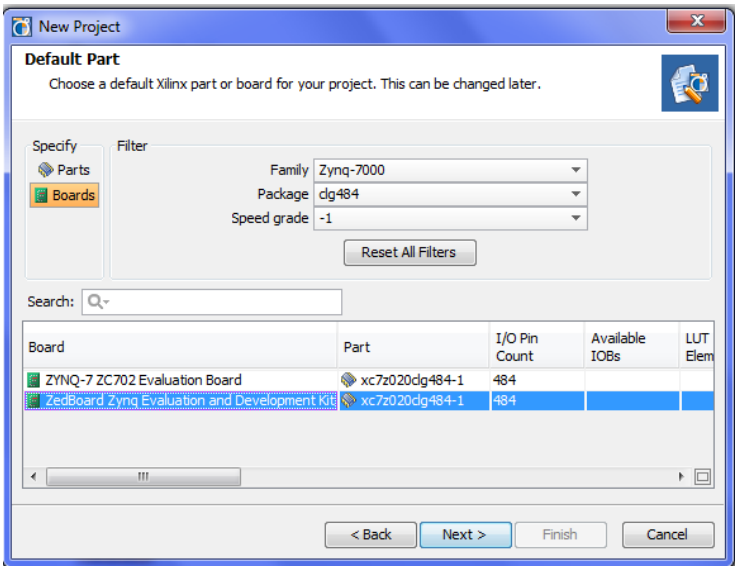


FIGURA 14. ASISTENTE DE CREACIÓN DE PROYECTOS EN PLANAHEAD (IV)

Finalmente, se muestra un resumen previo a la creación del proyecto con las opciones escogidas, como puede verse en la Figura 15.

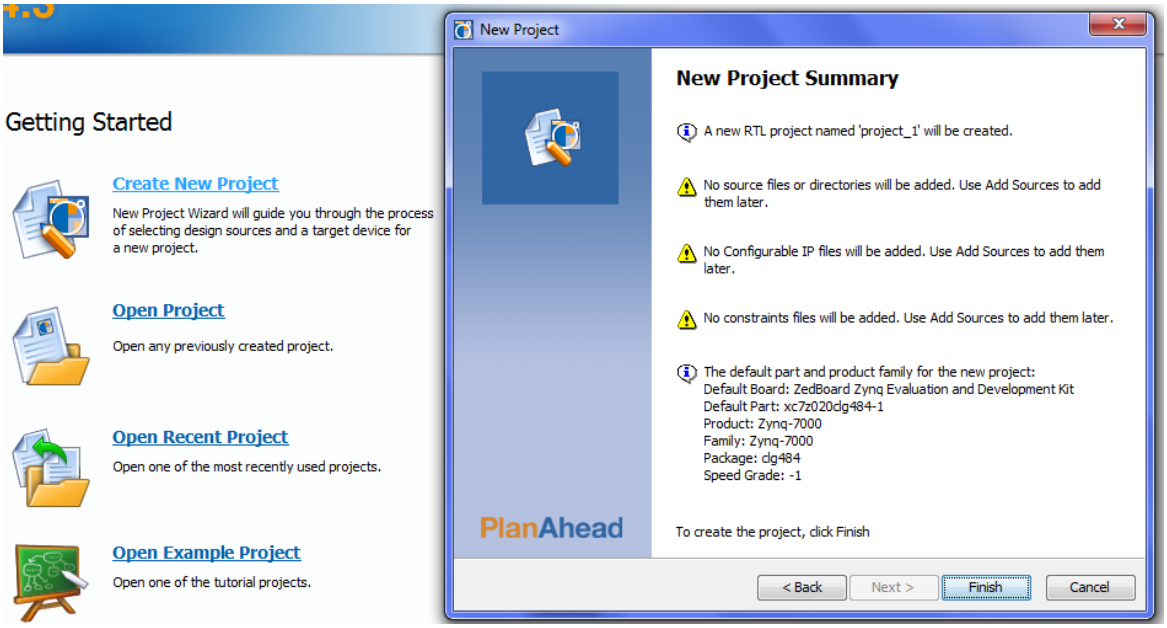


FIGURA 15. RESUMEN DEL PROYECTO CREADO EN PLANAHEAD

Una vez seleccionado *Finish* comenzará a crearse el nuevo proyecto, situando los ficheros correspondientes en el directorio indicado.

Aparece entonces la siguiente interfaz para comenzar a trabajar con PlanAhead:

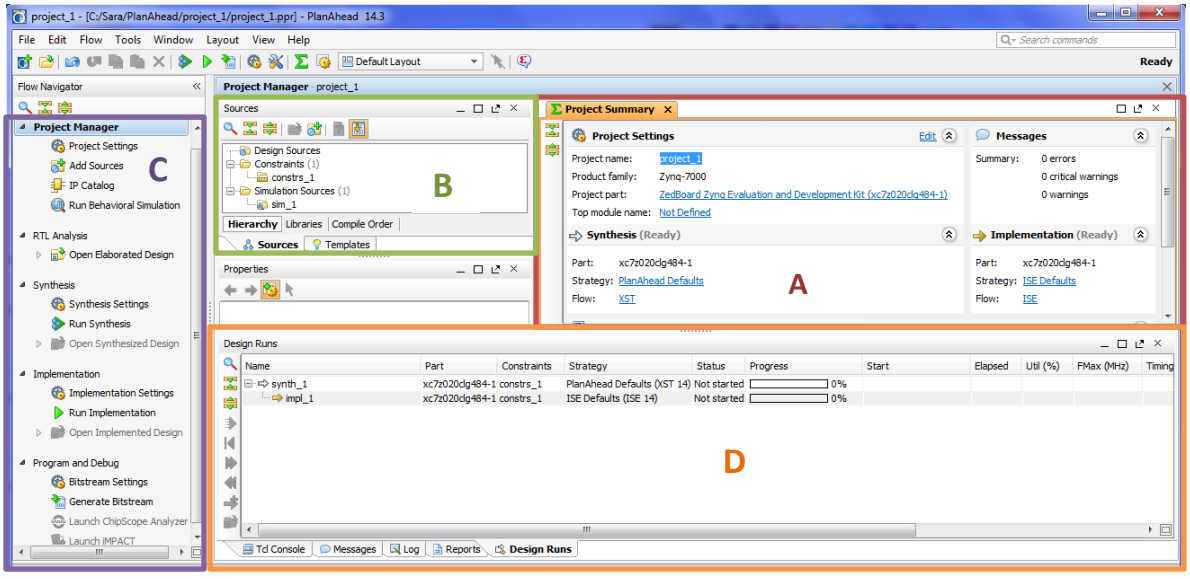


FIGURA 16. INTERFAZ DE USUARIO EN PLANAHEAD

En esta interfaz pueden distinguirse diferentes áreas.

En la zona A se encuentra un panel desde el cual se puede ver, por un lado, los datos que caracterizan al proyecto creado, y por otro, el estado en que se encuentra el proyecto en cuanto a las operaciones de síntesis e implementación, así como mensajes de warning y error que puedan ir surgiendo.
Al lado izquierdo de este panel, en la zona B, se encuentra una ventana en la cual se muestran en jerarquía los archivos que se irán creando o añadiendo al proyecto, como archivos fuente, ficheros *.ucf para definición de pines y archivos de simulación si se da el caso.
En la zona C se encuentra el explorador de proyecto, con un navegador que dispone de diferentes acciones relacionadas con el proyecto, entre ellas cabe destacar *Add Sources*, desde donde se crean los archivos anteriormente referenciados, *Run Synthesis*, *Run Implementation* y *Generate Bitstream.*
Finalmente, en la parte inferior, zona D, se encuentra la ventana de proceso, donde puede visualizarse la consola de comandos (*Tcl Console*), los mensajes de info, warning y error que surjan durante el diseño (*Messages*), los mensajes devueltos durante y al final de una operación (*Log* y *Reports*), y el estado en que se encuentran la síntesis e implementación (*Design Runs*).

Una vez creada la base del proyecto, puede comenzarse a diseñar. Para ello primeramente hay que crear la parte de diseño hardware del sistema. Para añadir un sub-diseño hardware al proyecto creado hay que lanzar la herramienta XPS desde PlanAhead.

Para ello, como se comentó anteriormente, debe seleccionarse en la columna de navegación *Add Sources*, donde se elegirá *Add or Create Embedded Sources* (ver Figura 17) para crear un nuevo sub-diseño con el que trabajar en XPS.

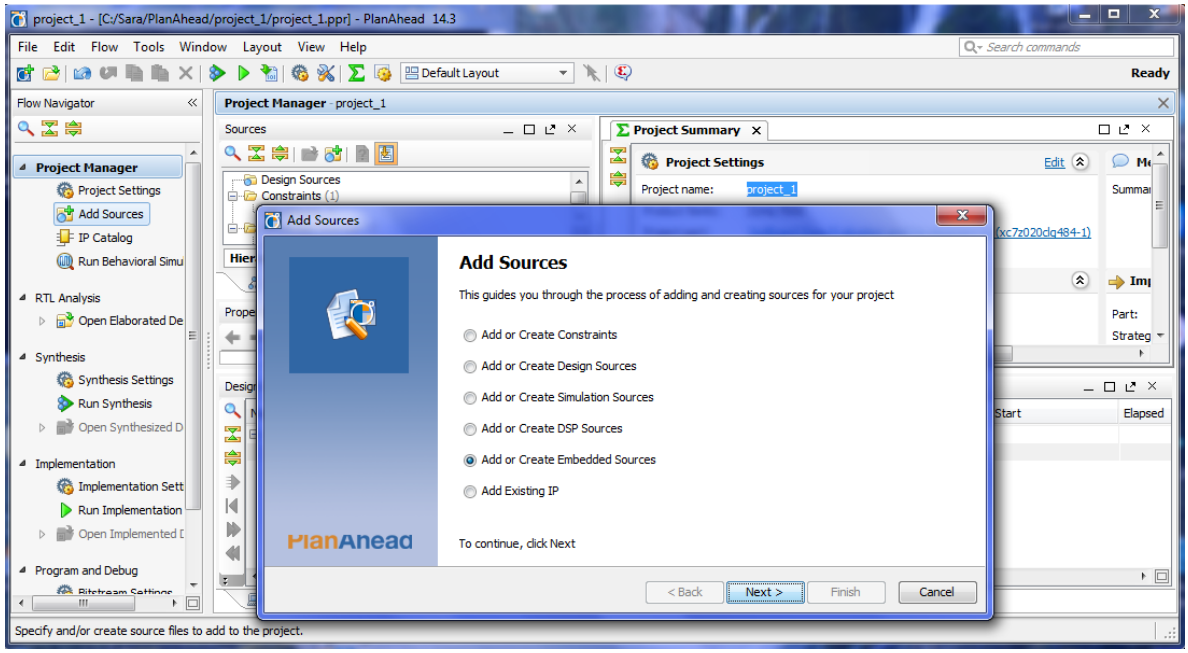


FIGURA 17. AÑADIR SUB-DISEÑO HARDWARE EN PLANAHEAD (I)

En la siguiente pestaña se da nombre al sub-diseño que se va a crear. En este caso se ha escogido llamarlo *module_1*, como se ve en la Figura 18.

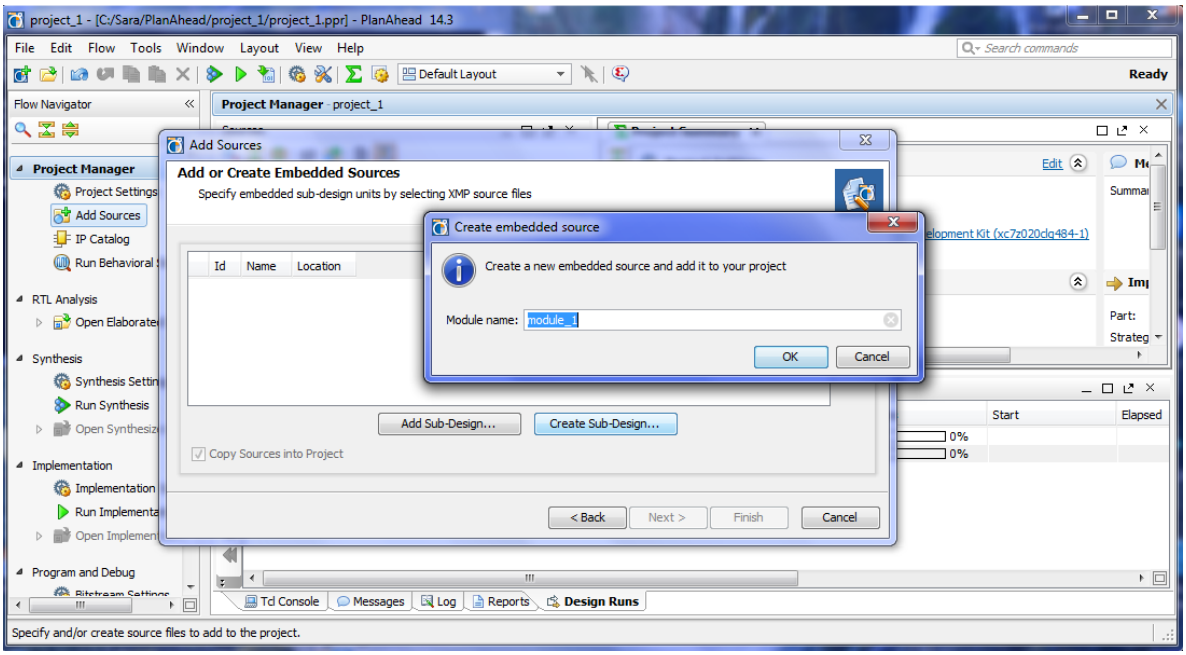


FIGURA 18. AÑADIR SUB-DISEÑO HARDWARE EN PLANAHEAD (II)

Una vez se da nombre a este diseño automáticamente PlanAhead lanza la herramienta XPS.

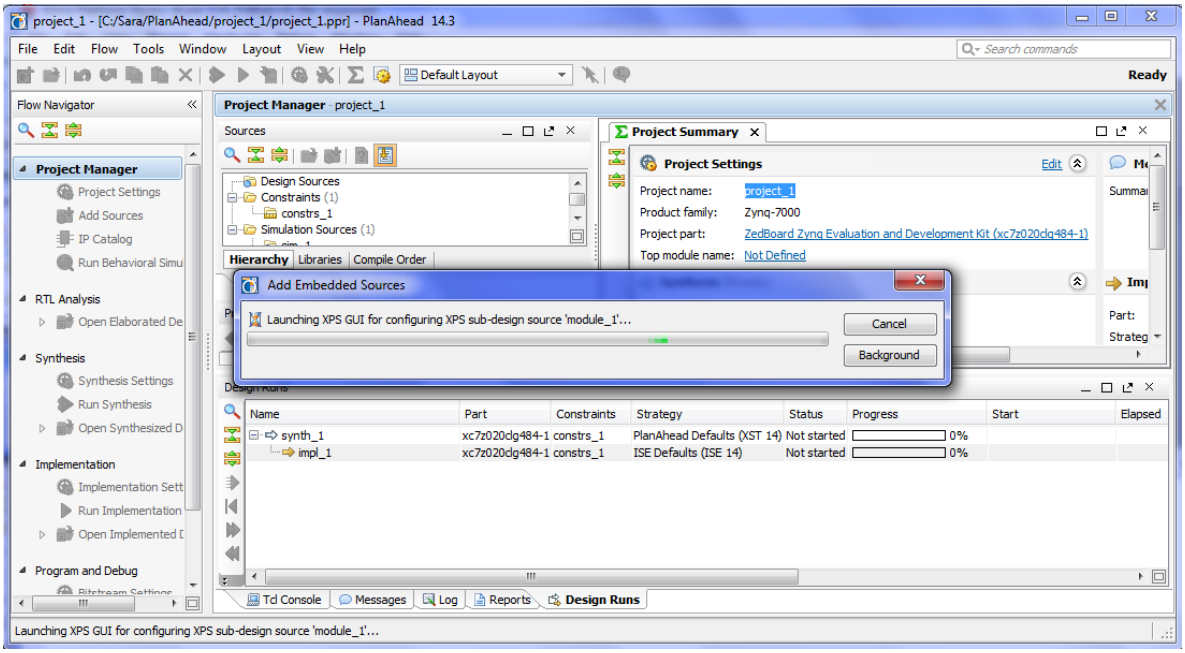


FIGURA 19. AÑADIR SUB-DISEÑO HARDWARE EN PLANAHEAD (III)

A partir de este punto el procedimiento a seguir se muestra en el apartado correspondiente a XPS (ver siguiente apartado 3.1.4 *Xilinx Platform Studio - XPS*).

Finalizado el diseño hardware desde XPS, se cierra el programa y se retorna automáticamente a PlanAhead, donde aparece el sub-diseño hardware creado en *Design Sources* con la extensión *.xmp que caracteriza a los diseños creados en XPS.

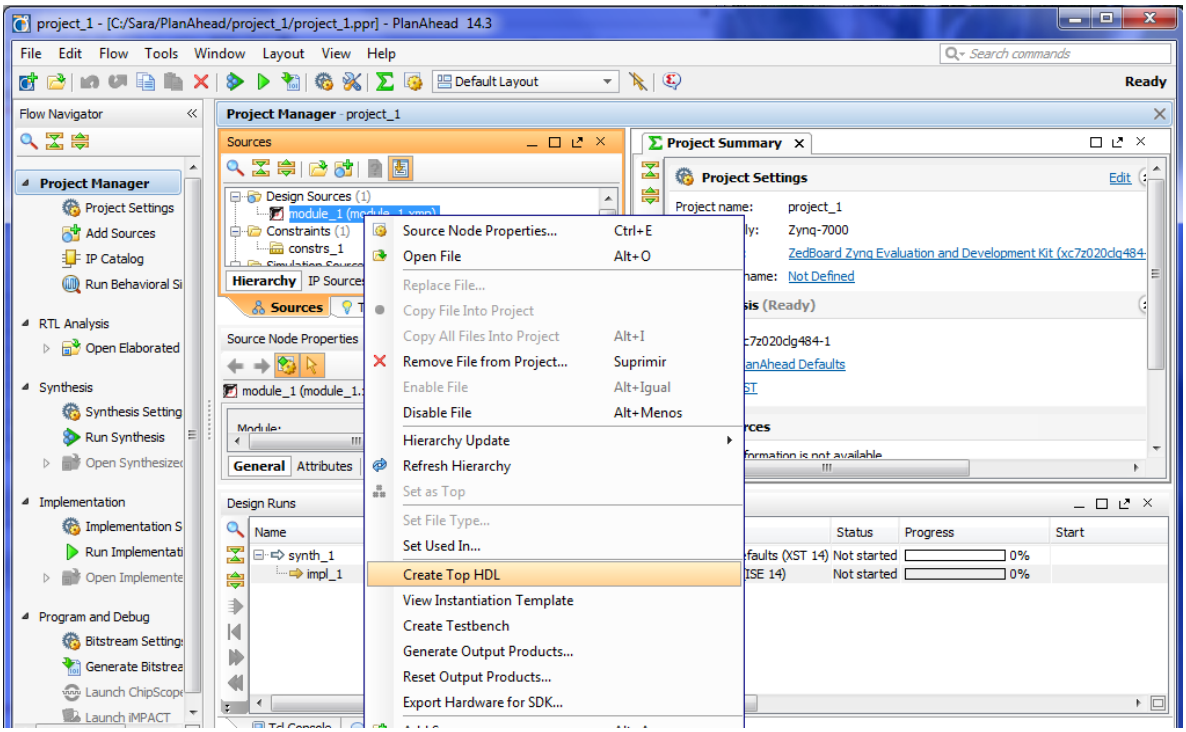


FIGURA 20. MÓDULO HARDWARE (.XMP) CREADO EN PLANAHEAD

Ahora, debe crearse el fichero en lenguaje hardware VHDL correspondiente a ese diseño, de forma que pueda ser comprendido por la tarjeta. Para ello, hay que seleccionar *Create Top HDL* (ver Figura 20) y, tras un tiempo de proceso, se observa que aparece como fichero top en la jerarquía el equivalente *.vhd al fichero *.xmp (ver Figura 21).

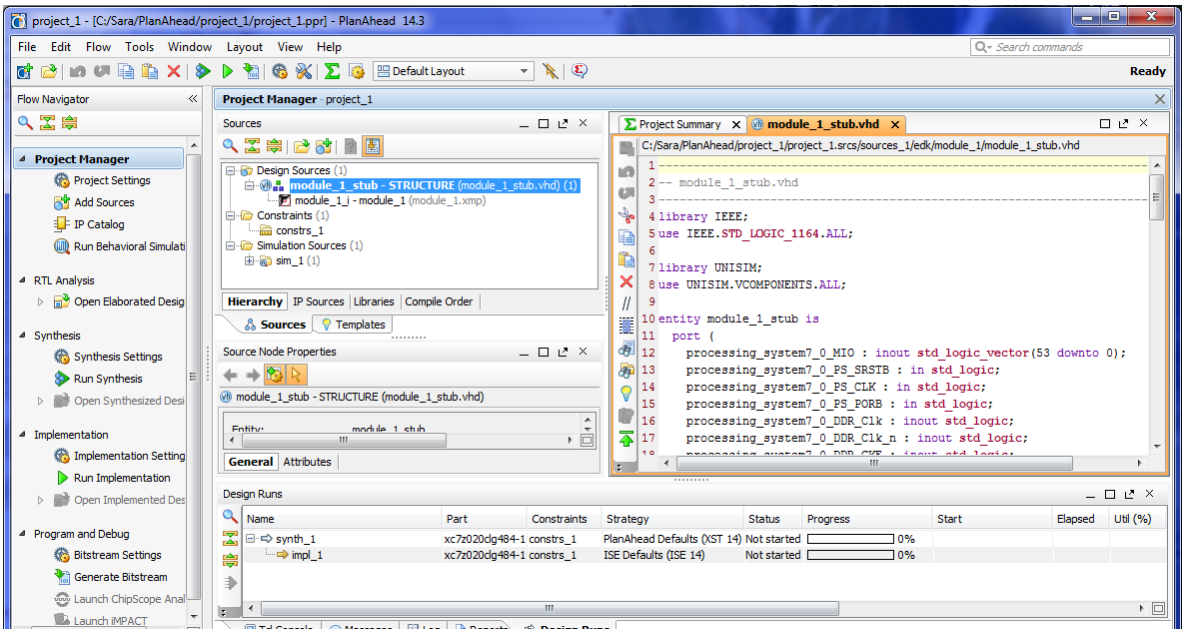


FIGURA 21. MÓDULO HARDWARE (.VHD) CREADO EN PLANAHEAD

Llegados a este punto, es conveniente crear o añadir un fichero *.ucf al proyecto, donde se indicará el nombre y la localización en la tarjeta de los puertos externos empleados en el proyecto. Nuevamente desde la pestaña de *Add Sources,* debe escogerse esta vez *Add or Create Constraints*.

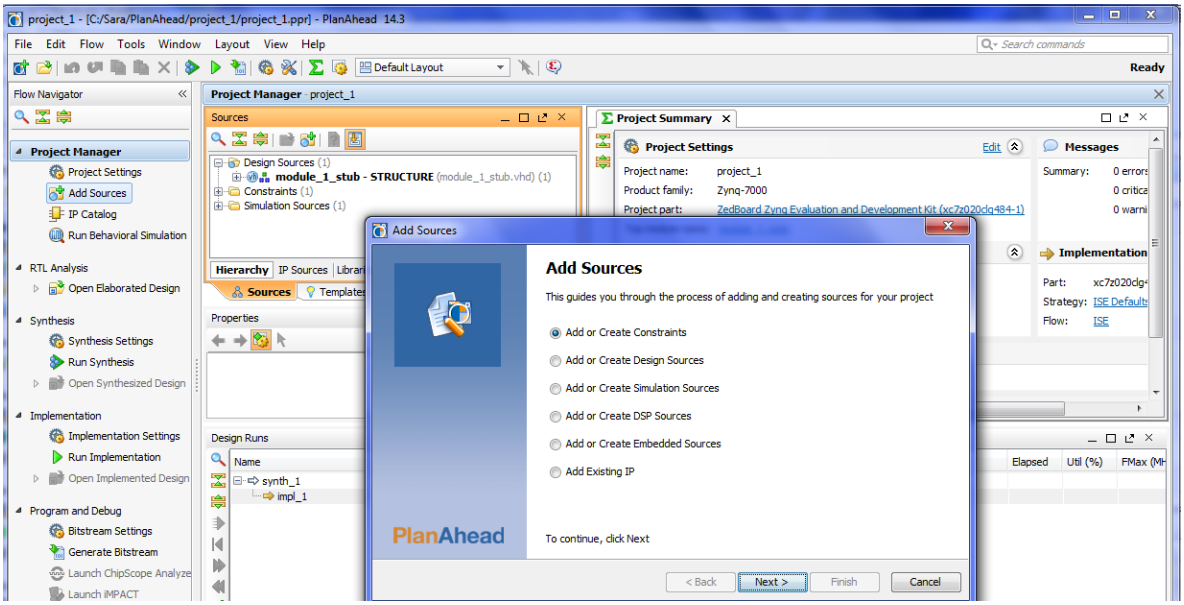


FIGURA 22. AÑADIR FICHERO CONSTRAINTS (.UCF) EN PLANAHEAD (I)

Ahora es posible añadir un fichero de restricciones *.ucf ya existente en *Add Files…* (ver Figura 23) indicando simplemente su directorio (ya que el programa se encarga de añadir físicamente ese fichero al directorio del proyecto) o crear un nuevo fichero en *Create File…,* como es el caso.

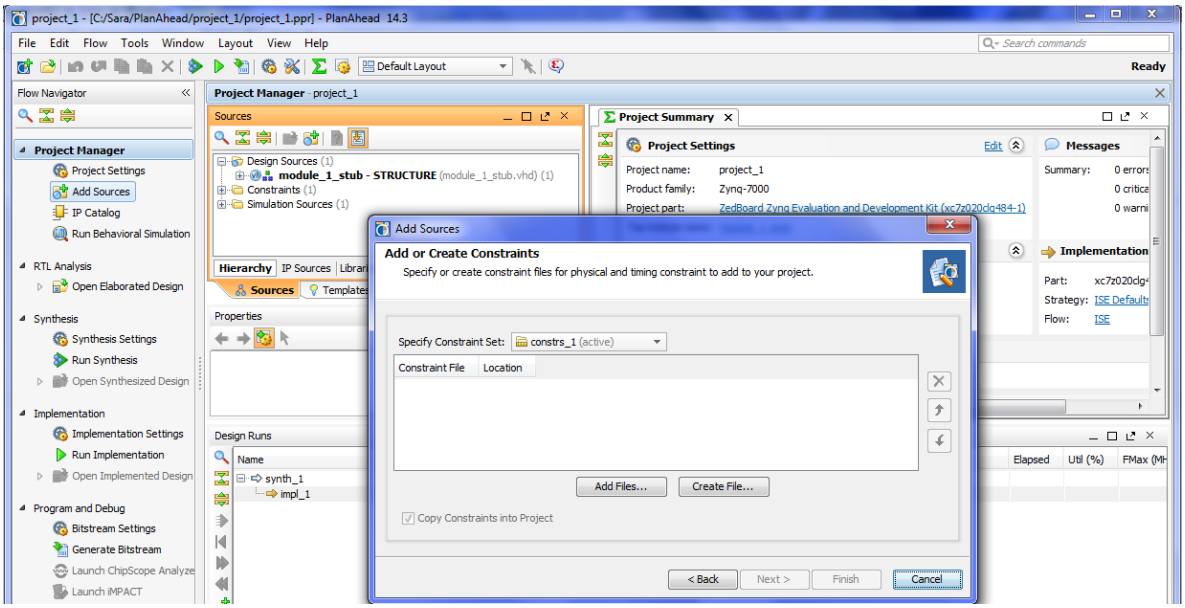


FIGURA 23. AÑADIR FICHERO CONSTRAINTS (.UCF) EN PLANAHEAD (II)

En este caso se va a crear un nuevo fichero de restricciones, por lo que se debe dar un nombre, y asegurarse de que la localización, por defecto, es <*Local to Project*>.

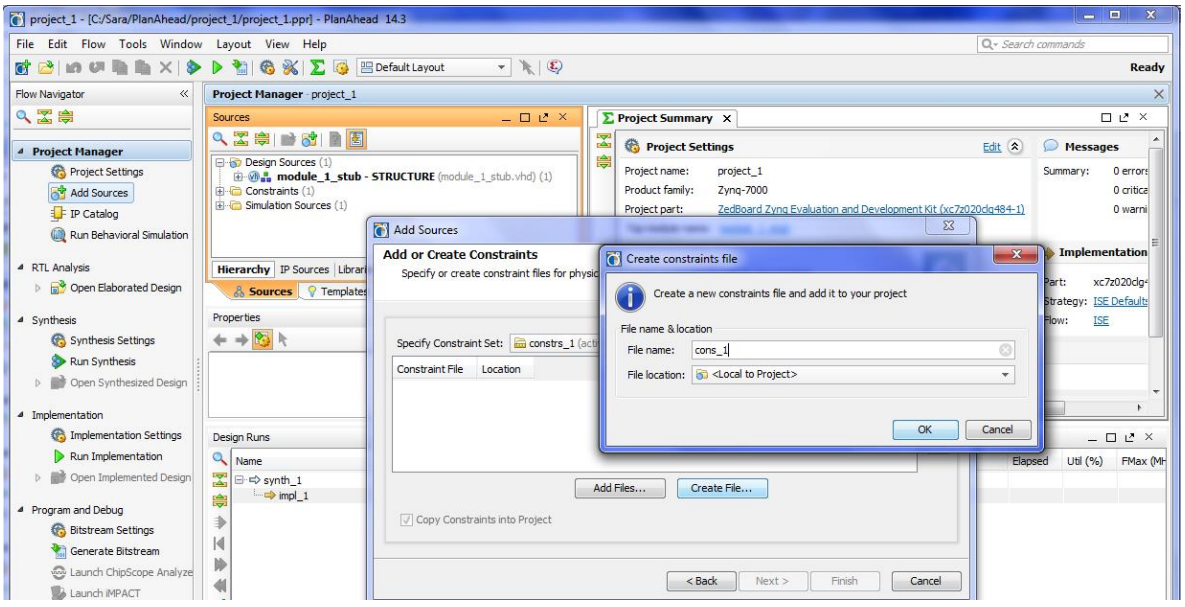


FIGURA 24. AÑADIR FICHERO CONSTRAINTS (.UCF) EN PLANAHEAD (III)

Al aceptar debe aparecer nombre y localización finales, y asegurarse de que esté marcado *Copy Constraints into Project*, como puede verse en la Figura 25.

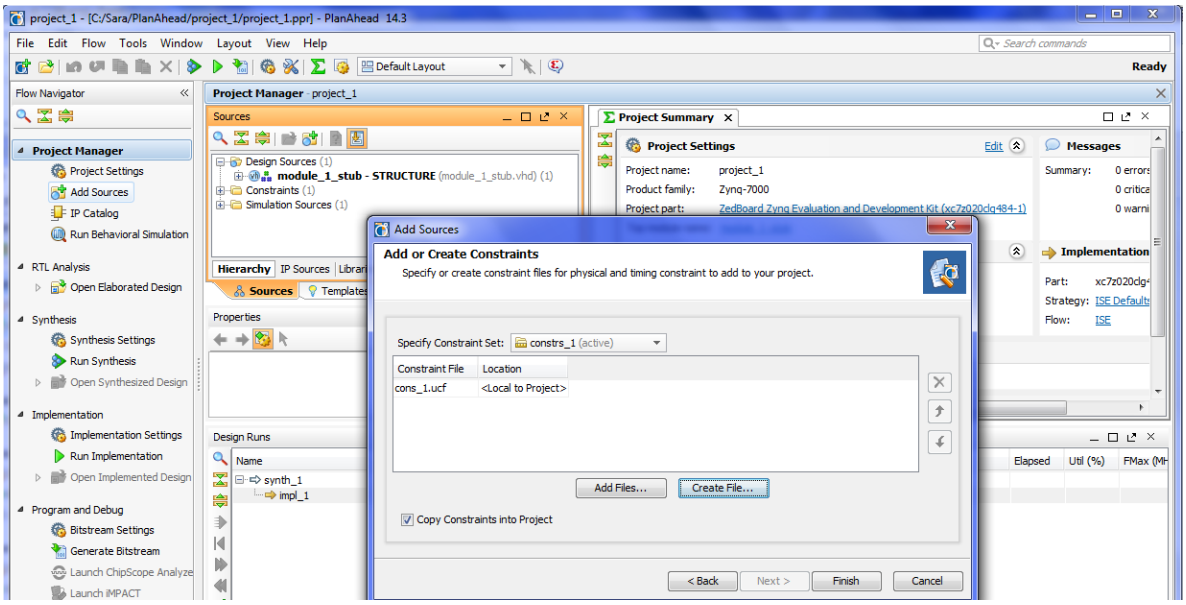


FIGURA 25. AÑADIR FICHERO CONSTRAINTS (.UCF) EN PLANAHEAD (IV)

Una vez finalizado debe aparecer el fichero *.ucf en la ventana referente a fuentes del proyecto, situada en el apartado de *Constraints*. Abriendo el fichero puede entonces realizarse la definición de los puertos GPIO como pines externos de la FPGA, de forma que queden asociados estos periféricos a los switches y los botones de la tarjeta.

En la Figura 26, aparece la asociación de puertos para ambos periféricos implementados en el diseño hardware, pudiéndose ver un vector de 8 bits para los dips y un vector de 5 bits para los botones con el nombre dado por XPS en su instanciación y fijando la localización en la tarjeta [9].

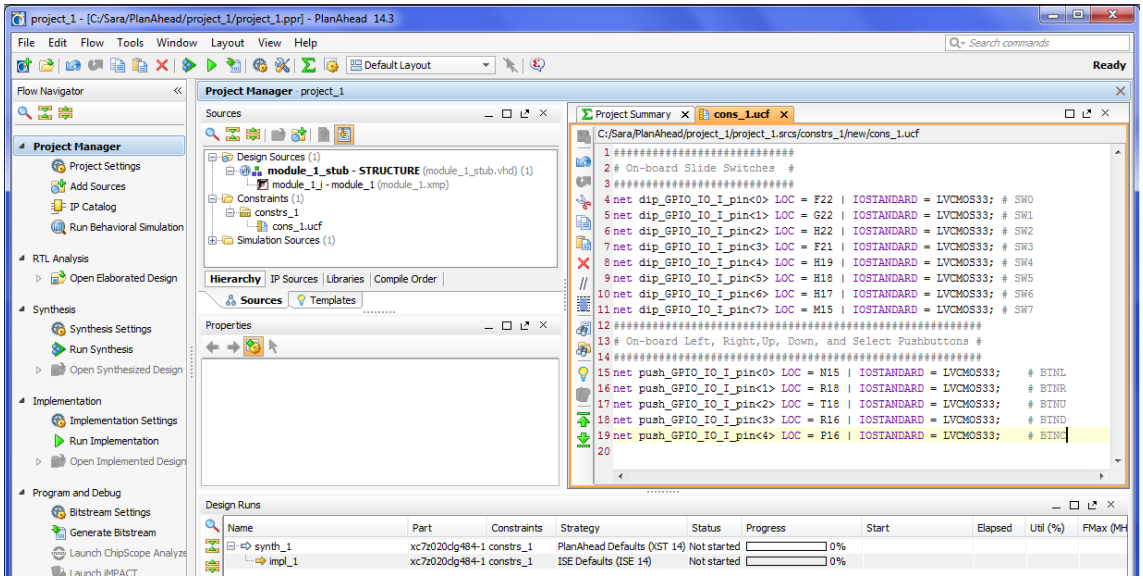


FIGURA 26. FICHERO CONSTRAINTS EN PLANAHEAD

El proceso a seguir a continuación es realizar síntesis e implementación del diseño, y generar el bitstream (*.bit) previo a exportar el diseño a la herramienta SDK para la incorporación de software. En el caso de que el diseño no contuviese hardware en la parte de lógica programable, podría exportarse directamente a SDK y sin generar *.bit.

En el explorador de proyecto se selecciona primero *Run Synthesis* para iniciar el proceso de síntesis del diseño y se espera a que finalice el proceso, el cual puede durar unos minutos.

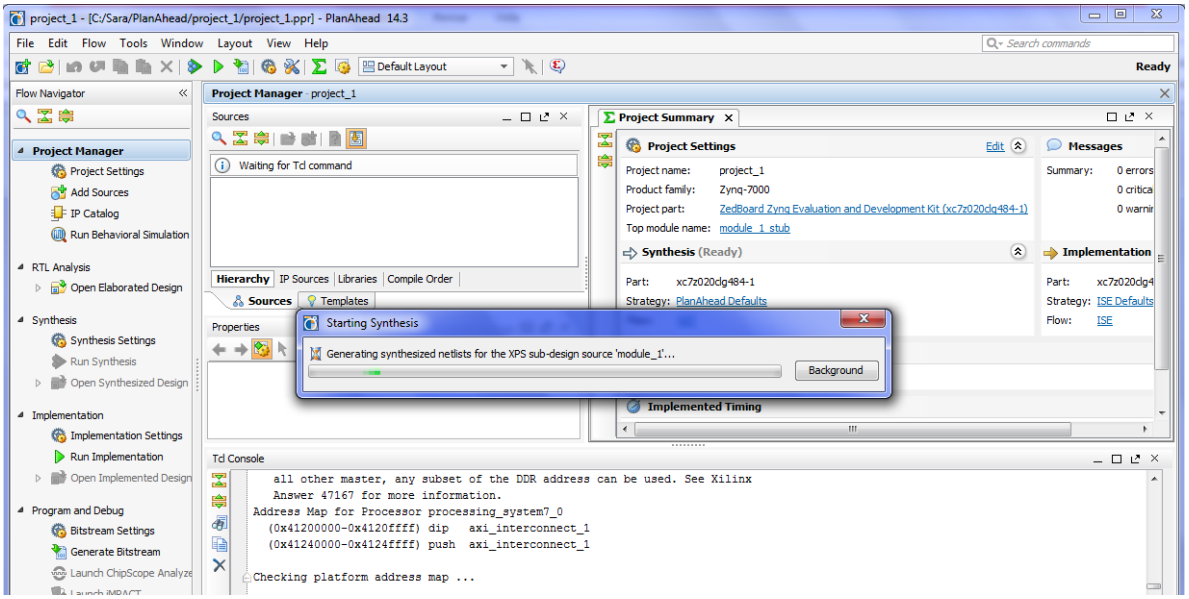


FIGURA 27. PROCESO DE SÍNTESIS EN PLANAHEAD

Al finalizar la síntesis se selecciona *Run Implementation* para continuar con la implementación.

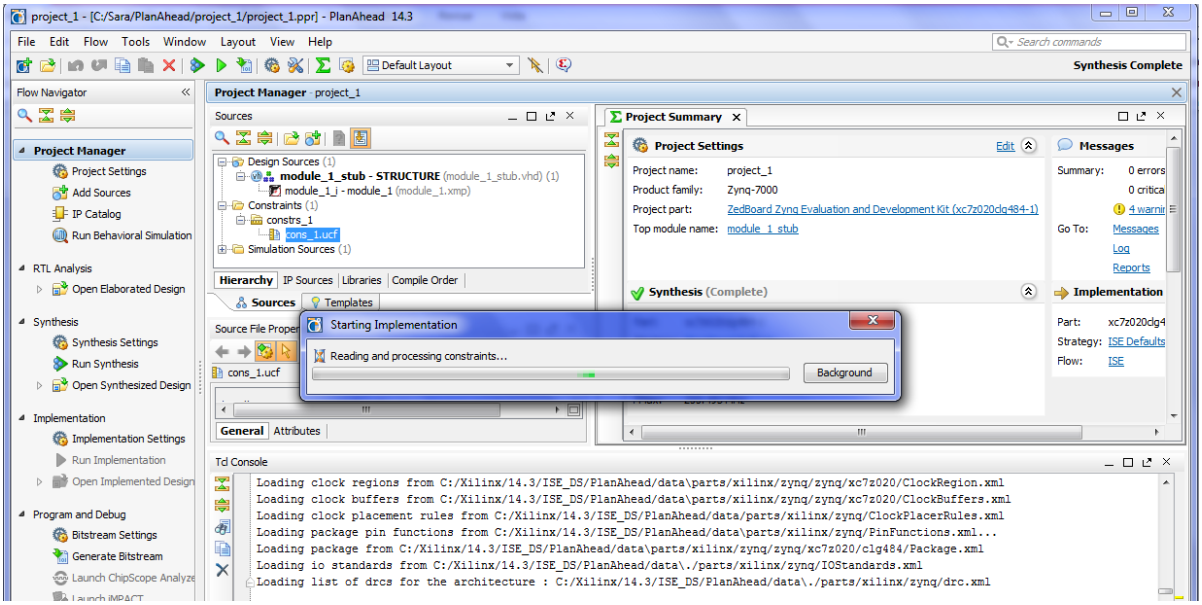


FIGURA 28. PROCESO DE IMPLEMENTACIÓN EN PLANAHEAD

Tras estos procesos de síntesis, implementación y generación del bitstream se devuelven una serie de *Reports* a modo de seguimiento referentes al mapeado, al place&route y a tiempos. Además puede verse en lo que se llamaba anteriormente zona A que el estado de *Synthesis* e *Implementation* aparece como *Complete,* al igual que muestra la pestaña *Design Runs*, y no se tienen errores ni warnings.

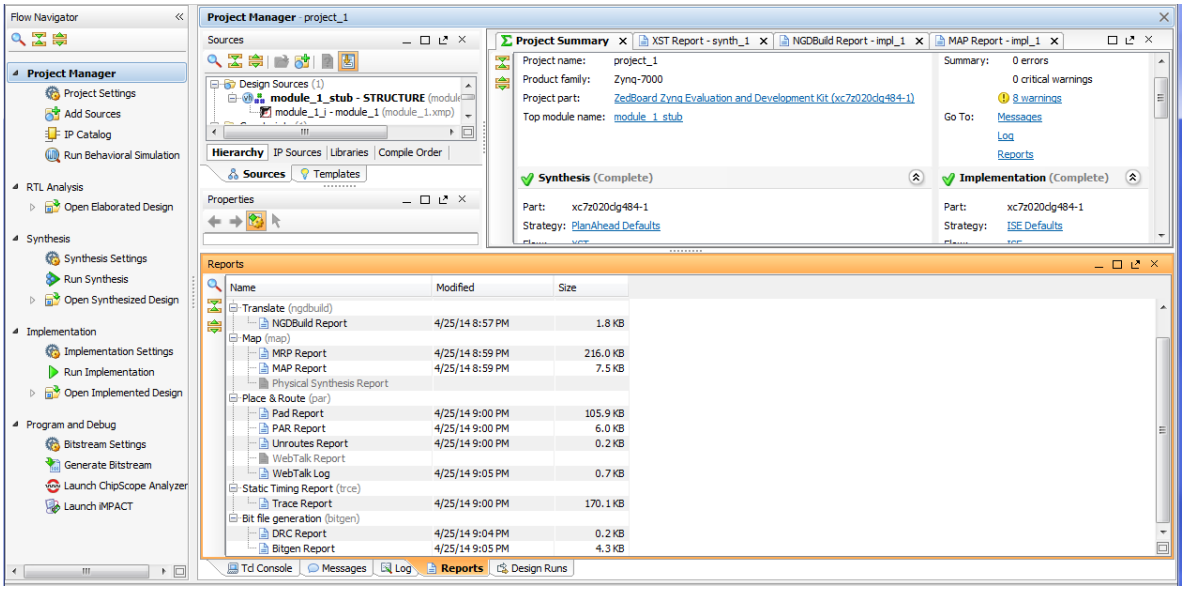


FIGURA 29. ESTADOS DE SÍNTESIS E IMPLEMENTACIÓN EN PLANAHEAD

Finalmente, se exporta el sistema hasta ahora creado a la herramienta SDK para continuar con el diseño software. Esto debe hacerse desde *File>Export>Export Hardware for SDK*, y en la ventana de lanzamiento asegurarse de marcar todas las opciones, *Include Bitstream* (fichero generado para SDK), *Export Hardware* (módulo hardware hasta ahora diseñado) y *Launch SDK*.

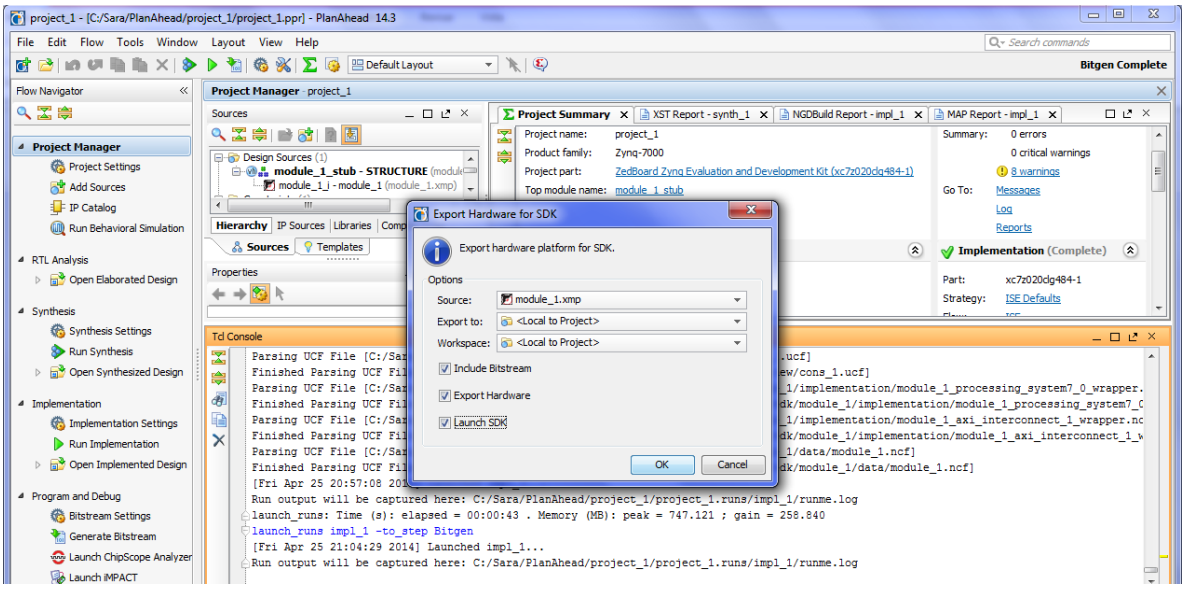


FIGURA 30. EXPORTACIÓN DEL DISEÑO HARDWARE A SDK DESDE PLANAHEAD (I)

Al aceptar comenzará el lanzamiento de la herramienta SDK, como se ve en la Figura 31.

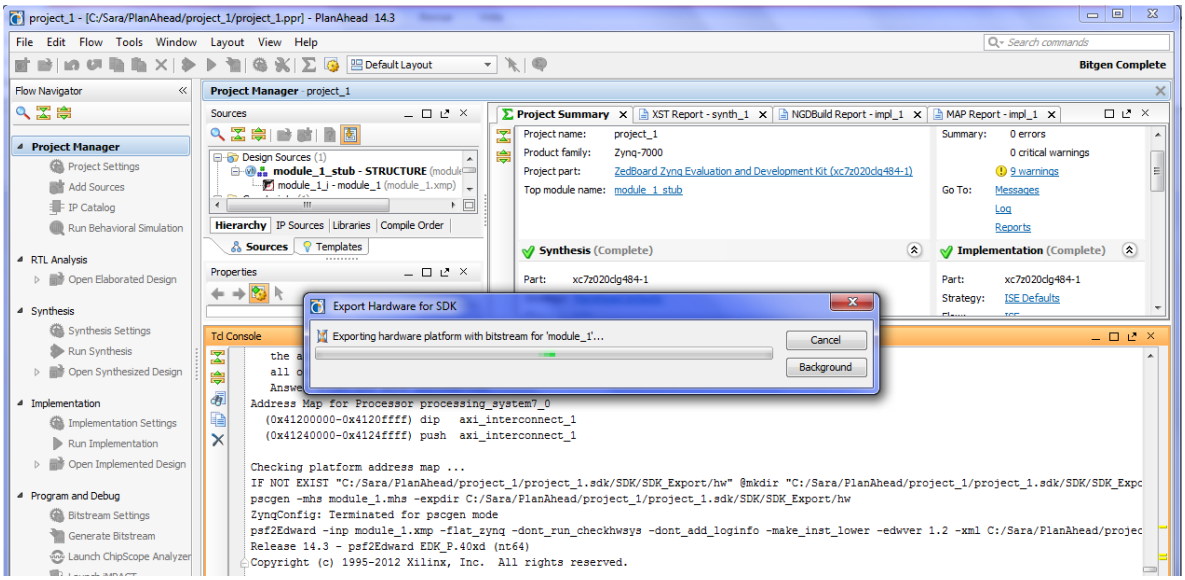


FIGURA 31. EXPORTACIÓN DEL DISEÑO HARDWARE A SDK DESDE PLANAHEAD (II)

A partir de este punto el procedimiento a seguir se muestra en la parte correspondiente a SDK (ver apartado 3.1.5 *Software Development Kit – SDK*).

3.1.4. Xilinx Platform Studio - XPS

Como ya se ha indicado, esta herramienta muestra una interfaz gráfica de usuario que permite comenzar realizando el diseño del sistema especificando el hardware y la forma de comunicación entre las partes PS y PL.

Base System Builder wizard

Se comienza creando un diseño sobre el que trabajar. Para ello XPS dispone del asistente Base System Builder (BSB) que crea la base del proyecto hardware conteniendo los elementos básicos en función de la tarjeta seleccionada con los que comenzar a construir un sistema más complejo, teniendo la opción de marcar/desmarcar elementos según interese o de añadir nuevos, como un periférico creado por el usuario con las herramientas dadas por XPS.

Al lanzar XPS desde PlanAhead se muestra un mensaje donde pregunta si se quiere hacer uso de este asistente.

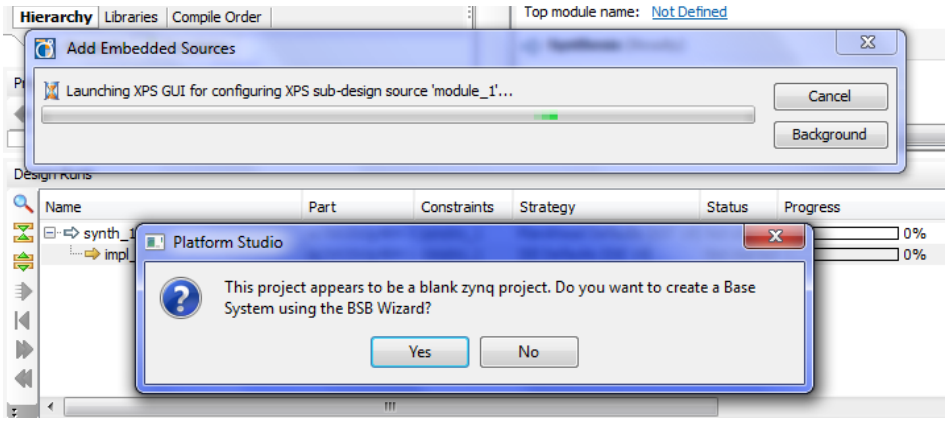


FIGURA 32. LANZAR BSB WIZARD DESDE PLANAHEAD

Al aceptar se abre dicho asistente, en el cual se configuran opciones base de diseño como la interfaz de conexión, la tarjeta sobre la que se desarrollará el diseño y la configuración inicial de periféricos que se desea tener.

En cuanto a la interfaz de conexión se escoge AXI System, puesto que sobre ella se realiza la comunicación entre la parte hardware y la parte software de Zynq, siendo ésta la interfaz estándar adoptada por Xilinx para los nuevos dispositivos, por lo que se recomienda su uso con vistas a posibles migraciones futuras.

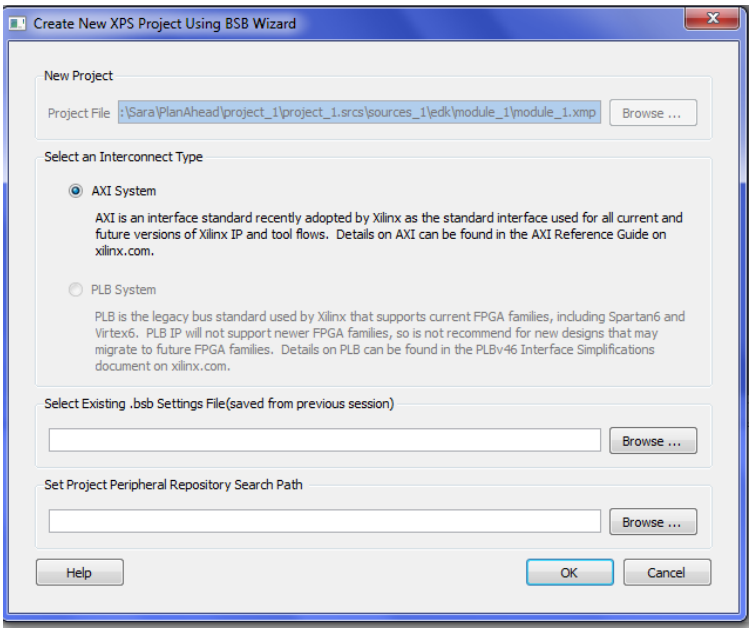


FIGURA 33. BSB WIZARD - SELECCIÓN DE INTERFAZ AXI

A continuación se escoge la tarjeta Zedboard Zynq Evaluation and Development Kit de Avnet y se deja seleccionada la plantilla por defecto, Zynq Processing System 7, la cual sirve de base para nuestro diseño y contiene la parte de procesador, PS, comunicada mediante el bus AXI a la parte lógica, PL, donde se encuentra el módulo AXI Interconnect (ya incluido por el BSB) para la conexión con periféricos GPIO.

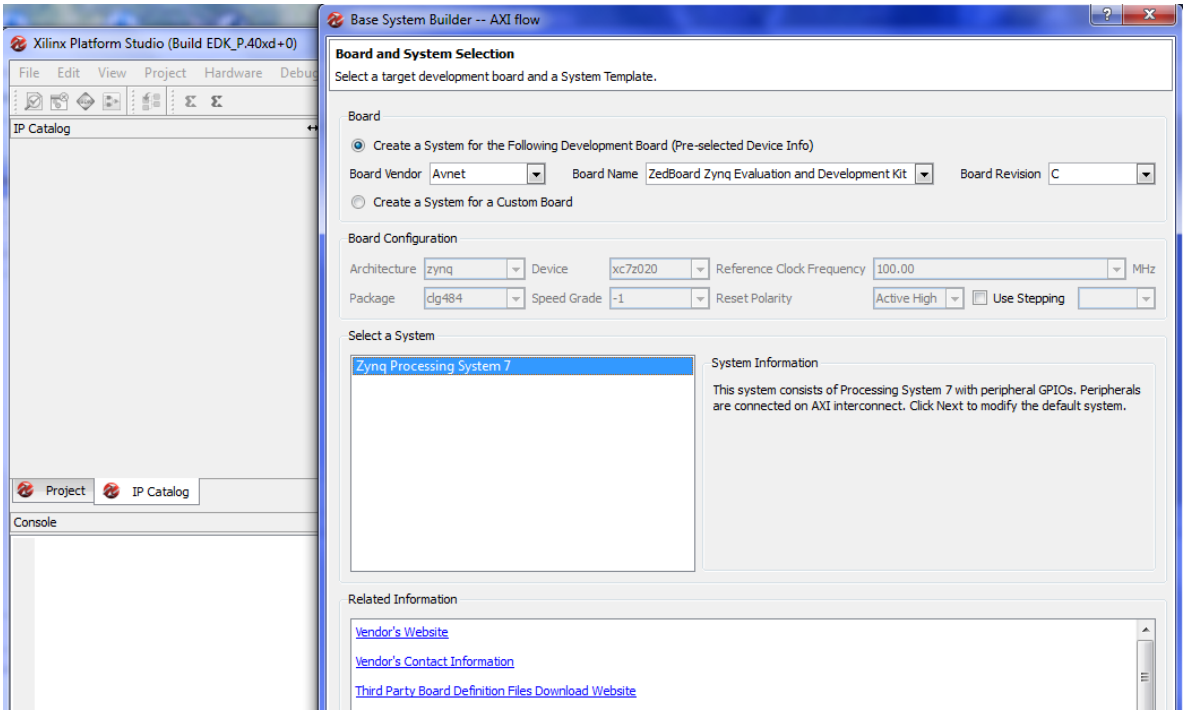


FIGURA 34. BSB WIZARD - ELECCIÓN DE LA TARJETA ZEDBOARD

Al aceptar se muestra una nueva ventana con la configuración de periféricos por defecto para la plantilla seleccionada, apareciendo periféricos tales como botones, switches y leds en el área denominada *Included Peripherals for Processing System 7*. Éstos serían útiles para el diseño que se va a desarrollar, donde se mostrará el procedimiento a seguir para añadir un par de periféricos GPIO, uno para switches y otro para botones, escogidos del catálogo del que dispone la herramienta XPS, pero con vistas a hacer un ejemplo de carácter genérico se va a eliminar esta configuración inicial, partiendo de cero en el proceso de implementación de periféricos.

Para ello, se debe seleccionar *Select All* y *Remove*, trasladándose el conjunto de periféricos al área *Available Peripherals*, tal como puede verse en la Figura 35.

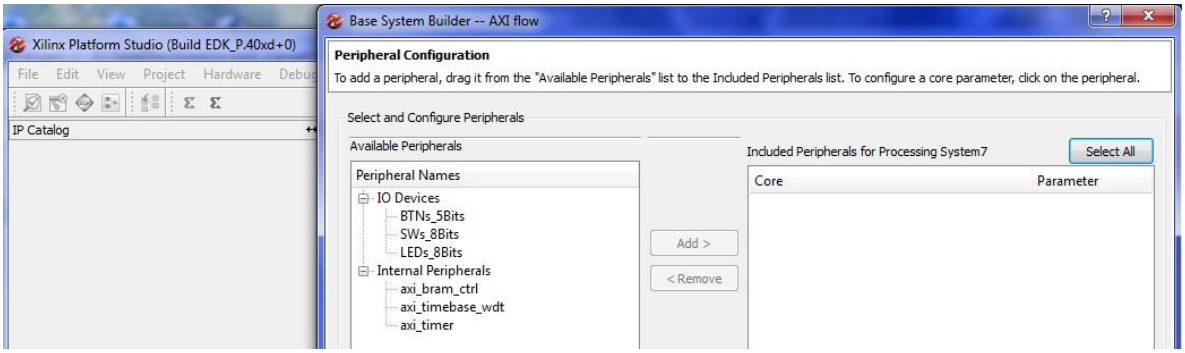


FIGURA 35. BSB WIZARD - CONFIGURACIÓN INICIAL DE PERIFÉRICOS

Al aceptar la configuración realizada se retorna a la interfaz de usuario de XPS, donde puede verse el sistema completo que ha proporcionado el asistente BSB para esta tarjeta.

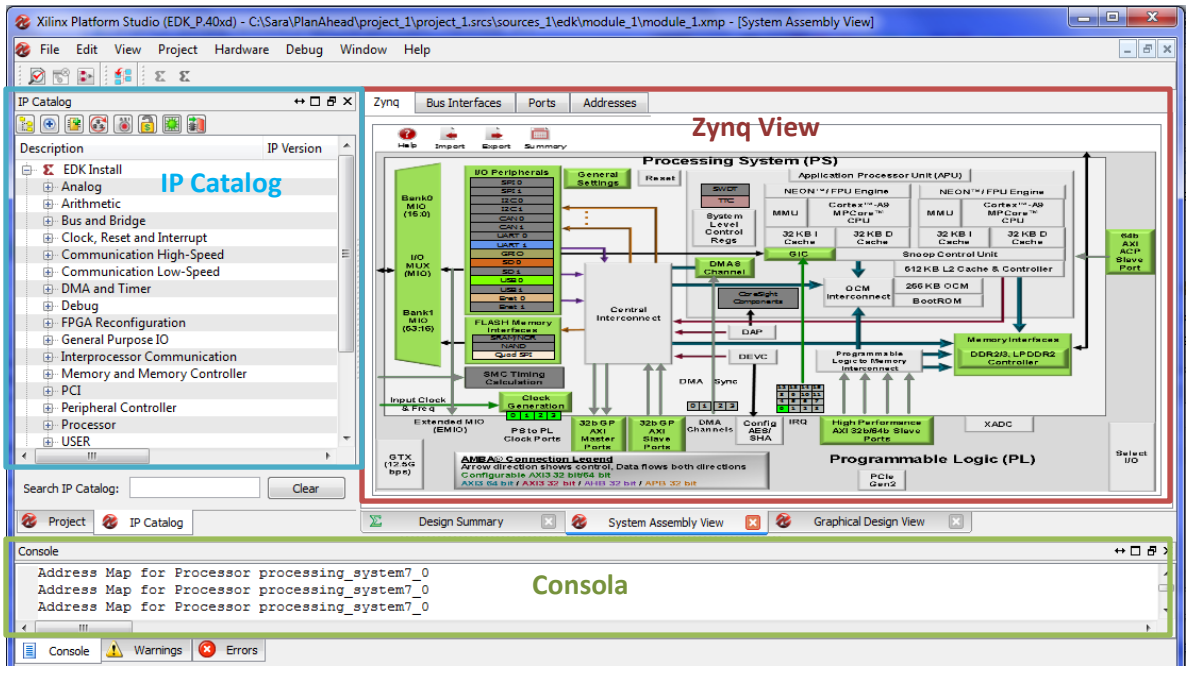


FIGURA 36. INTERFAZ DE USUARIO DE XPS

Esta interfaz puede dividirse en varias zonas.

La zona principal consta de un panel que se denomina Zynq Processing System View y muestra el dispositivo desde la perspectiva hardware, conteniendo la unidad de procesado con ambos microprocesadores, el controlador de interrupciones, la memoria on-chip, etc, así como el banco de periféricos I/O y MIO. También se muestran algunos elementos relacionados con la interconexión entre PS y PL como es el bloque *General Purpose Master AXI Interfaces* que permite habilitar la interfaz de comunicación M_AXI_GP0/1 de la parte PS con el bus AXI a través del bloque *Central Interconnect* (AXI Interconnect).
A la izquierda de este panel se encuentra el catálogo de bloques IP proporcionado por XPS, desde donde se pueden incluir distintos dispositivos al diseño sin más que hacer doble clic sobre ellos y confirmar el nombre del diseño donde se quieren añadir.
Finalmente, en la zona inferior se encuentra la consola, donde se observarán los diferentes comandos que vayan surgiendo al trabajar sobre el diseño, a la vez que posibles warnings y errores.

Como se aprecia en la Figura 37, en el panel de vista del sistema aparecen bloques con fondo verde, mientras el resto se encuentran en gris. Esto indica al usuario qué tipo de bloques puede configurar y acceder para la realización de su diseño hardware. De esta forma, puede verse que en el módulo *I/O Peripherals* dentro de la parte PS aparecen varios periféricos iluminados en color, pero para el primer diseño que se va a realizar sólo se requiere tener disponible la UART1 para comunicación serie (ver Figura 4), por lo que para aprovechar más los recursos y ahorrar tiempo de procesado, conviene abrir las opciones del bloque (haciendo clic sobre él) y desmarcar el resto de periféricos, quedando iluminada únicamente la celda UART1, como se observa en la Figura 37.

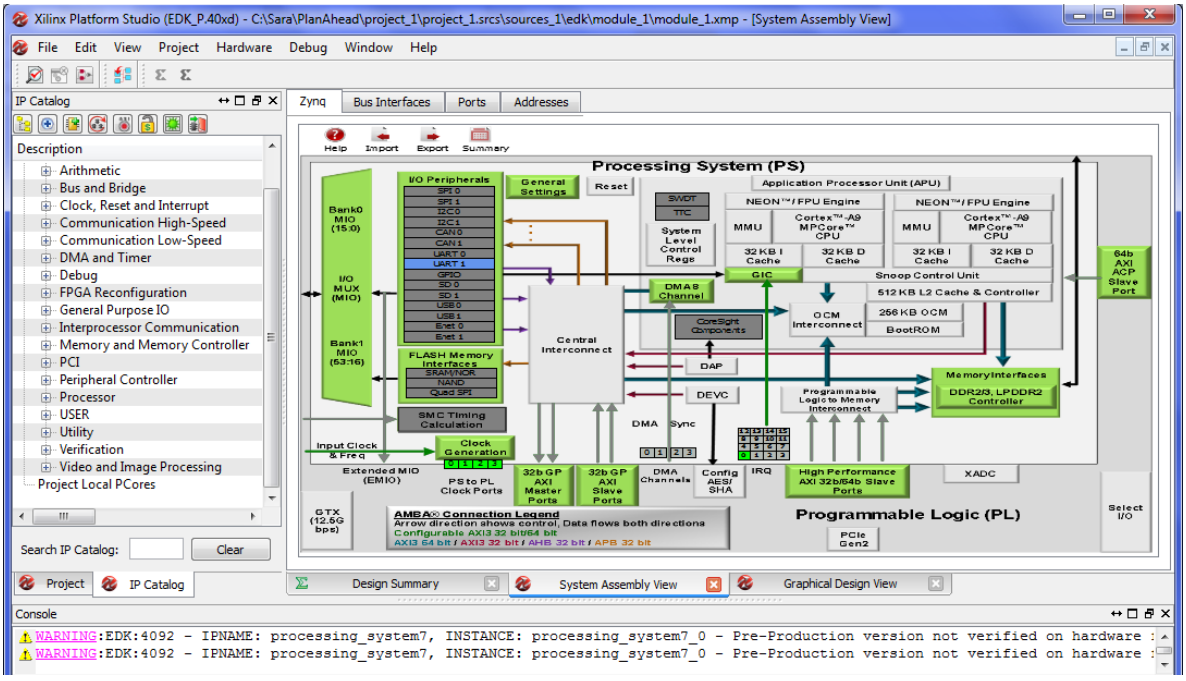


FIGURA 37. VISTA DE BLOQUES DEL SISTEMA EN XPS

Ahora es el momento de comenzar a darle un propósito al diseño. Si solo se quisiese utilizar la parte de lógica programable de la tarjeta para programar algún tipo de aplicación software podría darse por finalizada la parte de diseño hardware, cerrando la herramienta XPS y volviendo a PlanAhead. Pero en este caso el diseño que se quiere realizar sí va a incorporar diseño hardware, puesto que se añadirán dos periféricos de los disponibles en el catálogo.

El diseño final se muestra de forma esquemática en la Figura 38.

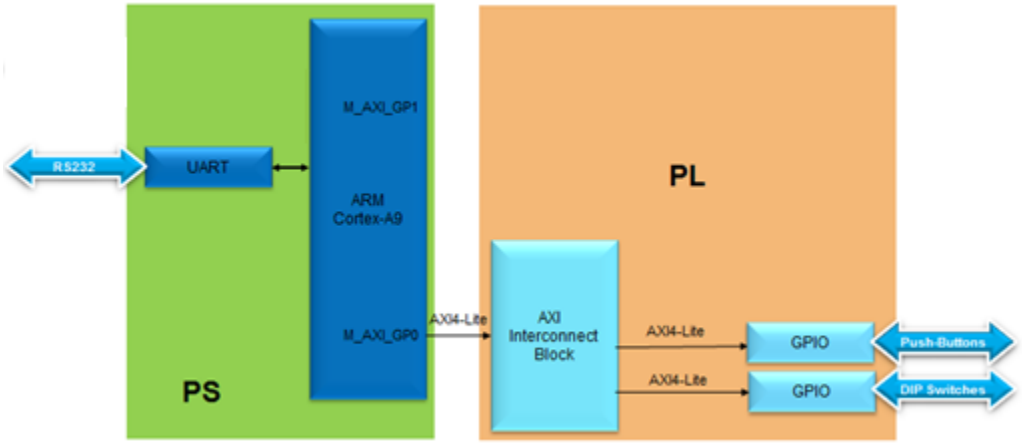


FIGURA 38. ESQUEMA DE BLOQUES DEL DISEÑO A REALIZAR INCORPORANDO PERIFÉRICOS GPIO EN XPS

Se observa que para poder comunicar la parte PS (maestro) con la parte PL (esclavo) es necesario habilitar la interfaz AXI4Lite en la parte hardware. Esto se hace desde el bloque *General Purpose Master AXI Interfaces*, como se comentó anteriormente, abriendo su configuración y marcando *Enable M_AXI_GP0 interface*.

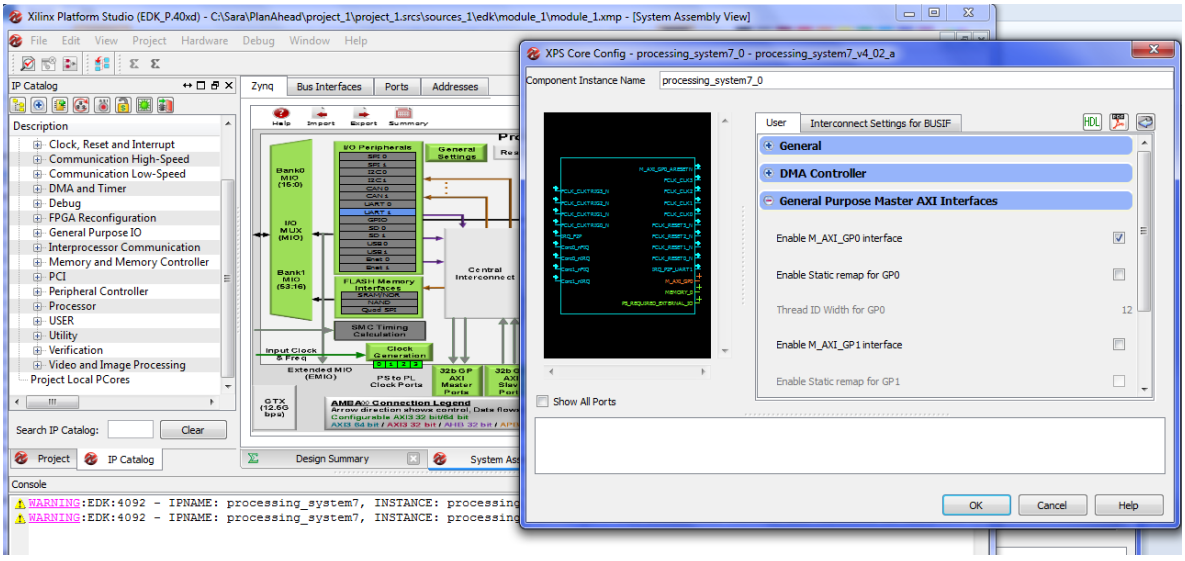


FIGURA 39. HABILITAR LA INTERFAZ AXILITE DE MAESTRO A ESCLAVO

Al guardar la configuración puede verse que desde dicho bloque al bloque Central Interconnect que gestiona las conexiones entre ambas partes PS y PL, se ha habilitado una interfaz para la comunicación en sentido PS a PL. Ésta aparece iluminada en color verde en la Figura 40.

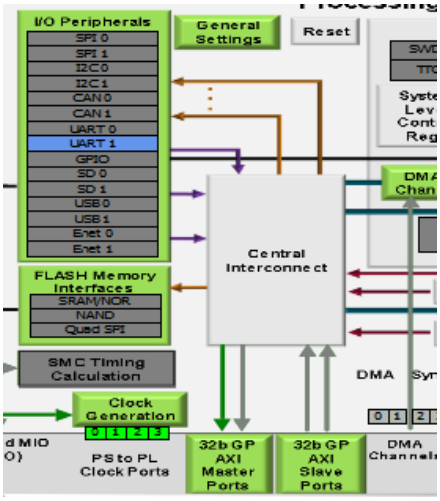


FIGURA 40. INTERFAZ M_AXI_GP0 HABILITADA

Ahora, desde el catálogo IP, pueden añadirse al diseño los dos GPIO desplegando en el catálogo *General Purpose IO*, seleccionando *AXI General Purpose IO* y aceptando su implementación. Se abrirá una ventana donde se debe dar nombre al periférico, configurarlo para un canal, y un ancho de bits.

Para el diseño que se quiere mostrar se les darán los nombres *"dip"* y *"push"* utilizando un único canal, *Channel 1*, *Input Only* y ancho de bits de 8 y 5 respectivamente, como puede verse en la Figura 41 y la Figura 42 para su configuración.

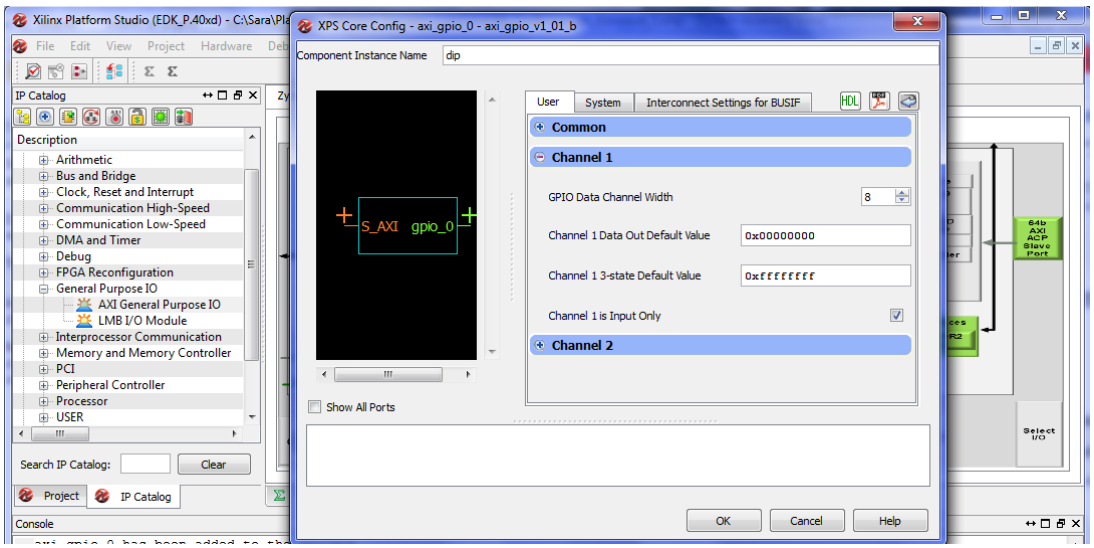


FIGURA 41. PROPIEDADES DEL BLOQUE GPIO AÑADIDO - DIP

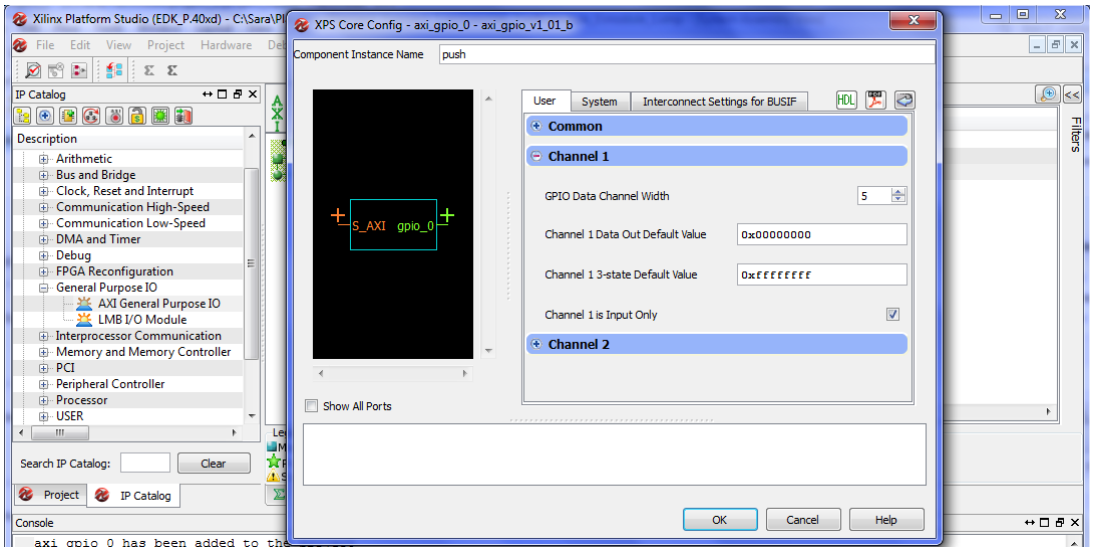


FIGURA 42. PROPIEDADES DEL BLOQUE GPIO AÑADIDO - PUSH

Una vez se han añadido puede comprobarse su correcta instanciación en la pestaña *Bus Interfaces* del panel principal, donde ambos se han conectado a PS a través de *axi_interconnect_1* y las interfaces *M_AXI_GP0* vista desde el lado maestro, PS, y *S_AXI* vista desde el lado esclavo, PL.

Haciendo doble clic sobre el periférico puede verse su configuración y comprobarse que efectivamente usa el protocolo AXI4Lite. A su vez, se les han asignado automáticamente direcciones, que pueden comprobarse desde la pestaña Addresses.

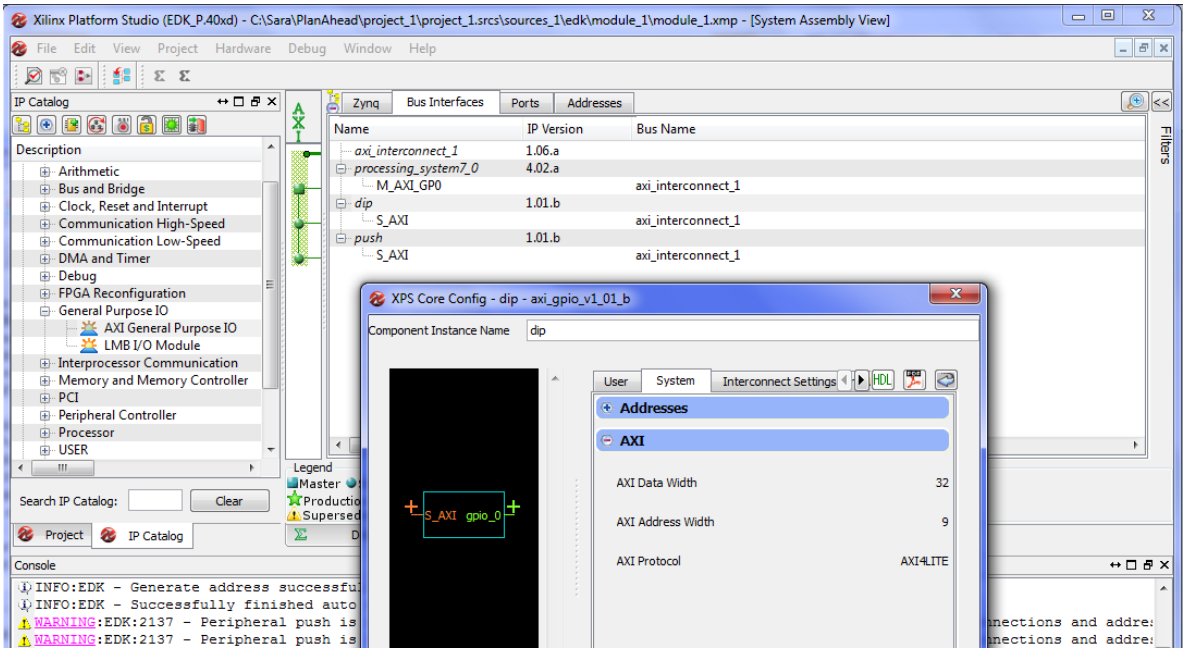


FIGURA 43. INTERFACES Y CONFIGURACIÓN DEL PERIFÉRICO AÑADIDO EN XPS

Para asociar estos periféricos a los switches y los botones de la tarjeta deben configurarse los puertos GPIO como pines externos de la FPGA y asignarles las localizaciones correspondientes en el fichero de restricciones *.ucf una vez se haya retornado a PlanAhead.

Así, en la pestaña *Ports*, visualizando la interfaz de salida *(IO_IF) gpio_0*, se observa que hay 4 puertos por cada periférico GPIO: I para *Input Only*, O para *Output Only*, T para *Tristate*, e IO para *Input/Output*. Este último es el que viene conectado por defecto, pero se quiere tener sólo entrada por ambos periféricos, por tanto, debe seleccionarse con botón derecho sobre ese puerto IO *No Connection*.

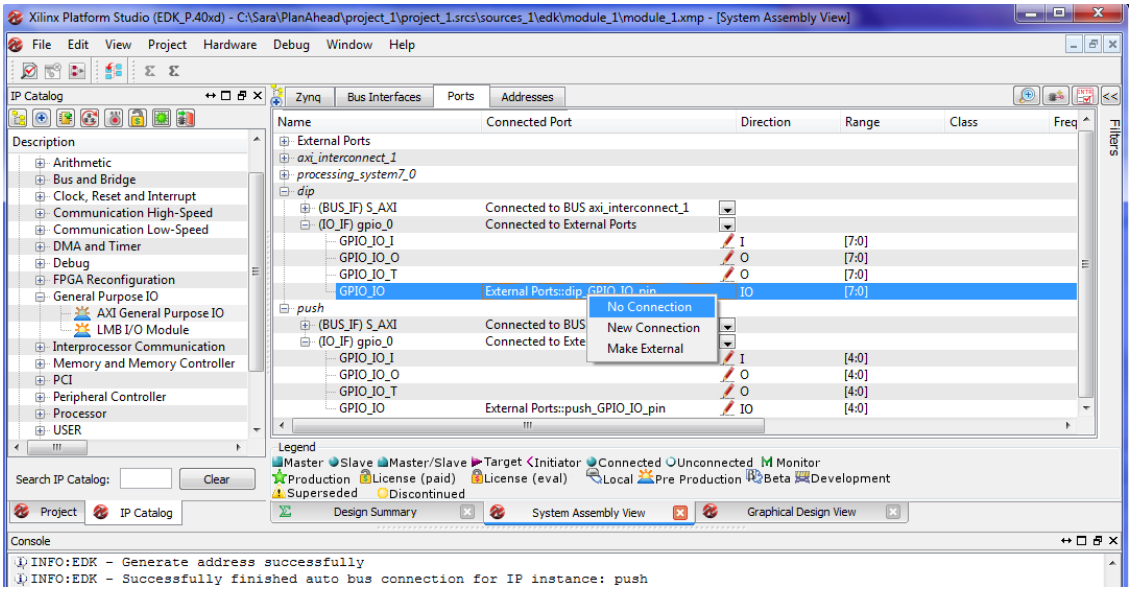


FIGURA 44. CONFIGURAR LOS PUERTOS GPIO COMO PINES EXTERNOS - DESCONECTAR POR DEFECTO

A continuación, con botón derecho sobre el puerto I seleccionar *Make External*, en ambos periféricos dip y push.

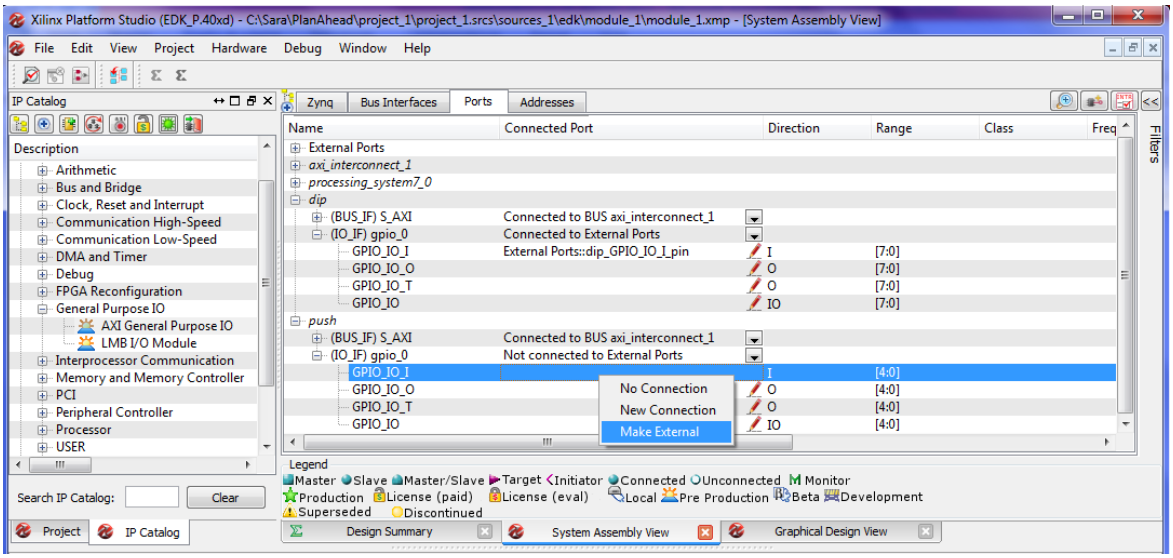


FIGURA 45. CONFIGURAR LOS PUERTOS GPIO COMO PINES EXTERNOS - CONECTAR INPUT ONLY

Una vez ambos puertos de entrada se hayan configurado como pines externos, aparecerán en el desplegable *External Ports* dentro de *Ports*, con el nombre que se utilizará en el fichero *.ucf para referirse a ellos y el rango de bits que ocupan (8 bits para el periférico *dip* y 5 bits para el periférico *push* como se configuró previamente), puesto que la tarjeta tiene 8 switches y 5 pushbuttons. Puede verse que estos nombres coinciden con los nombres dados en Constraints (Figura 26).

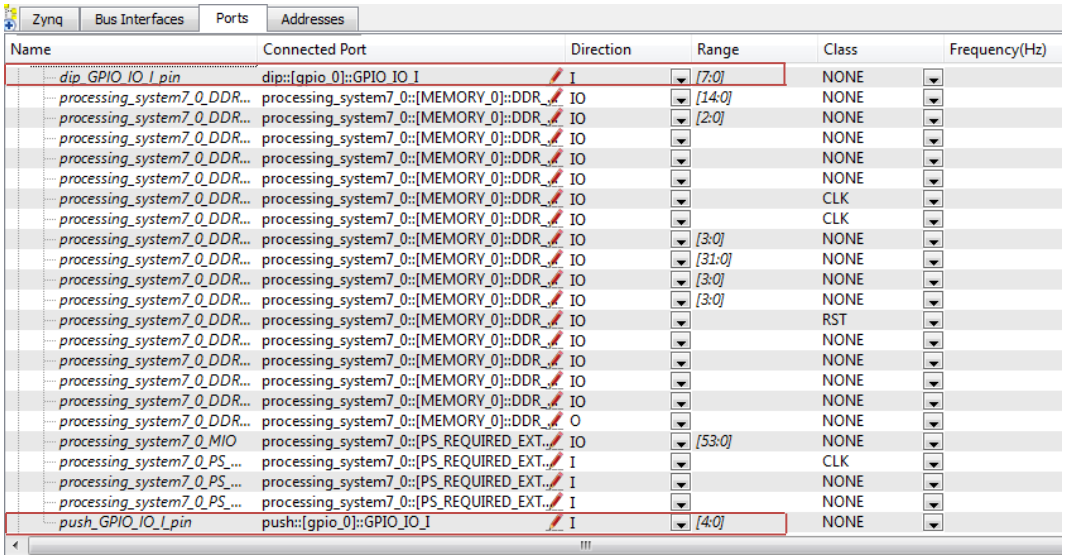


FIGURA 46. RESUMEN DE PUERTOS EXTERNOS DEL DISEÑO CREADO EN XPS

Para este diseño concluiría la parte de diseño hardware, por lo que basta con comprobar si hay errores con *Design Rule Check*, que muestra unos mensajes como se ve en la Figura 47:

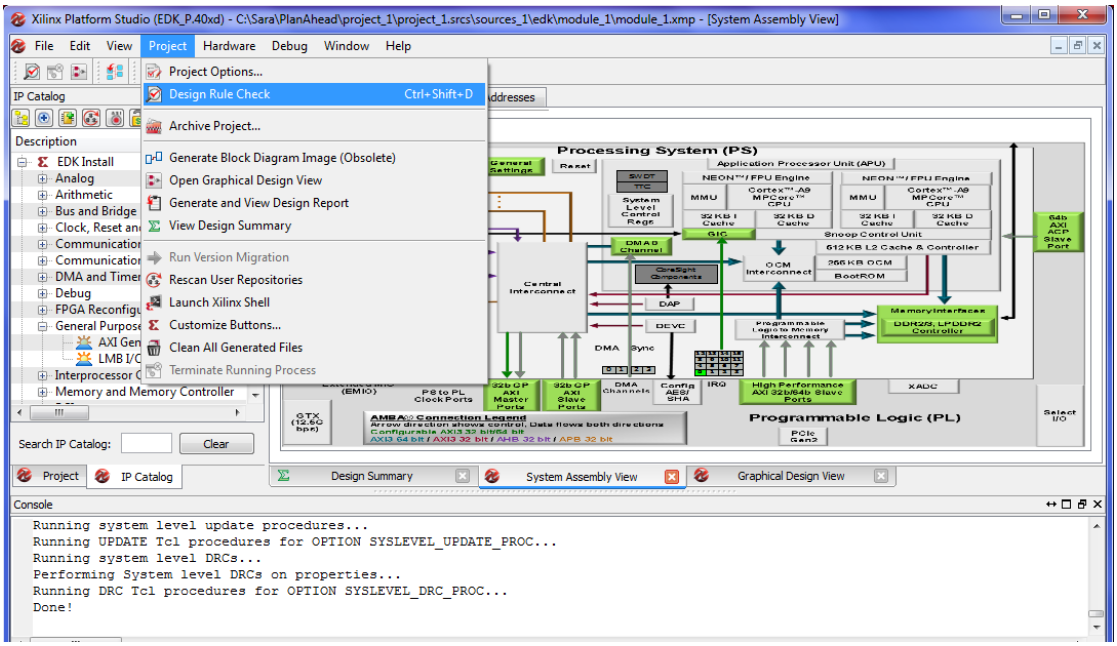


FIGURA 47. COMPROBACIÓN DE ESTADO DEL DISEÑO HARDWARE

Finalmente, *File>Exit* cerraría la herramienta XPS devolviendo el control a PlanAhead, donde aparecería reflejado en *Design Sources* el fichero *.xmp creado para el diseño hardware.

3.1.5. **Software Development Kit - SDK**

Esta herramienta es la proporcionada por Xilinx para completar el diseño añadiendo funcionalidad software a la parte PS de la tarjeta. Para ello, previamente ha sido necesario crear la base del sistema y añadir un diseño hardware utilizando la herramienta XPS. Una vez se ha finalizado con XPS, y se ha llevado a cabo la síntesis, implementación y generación del bitstream del diseño, se lanza SDK desde PlanAhead.

La interfaz de usuario que se muestra al iniciar SDK (ver Figura 48) está formada por el explorador de proyecto situado a la izquierda (zona A), un panel central de visualización (zona B), un panel visualizador de los ficheros contenidos en el fichero fuente de la aplicación situado a la derecha (zona C), y en la parte inferior la ventana de tareas, warnings, la consola y el terminal (zona D).

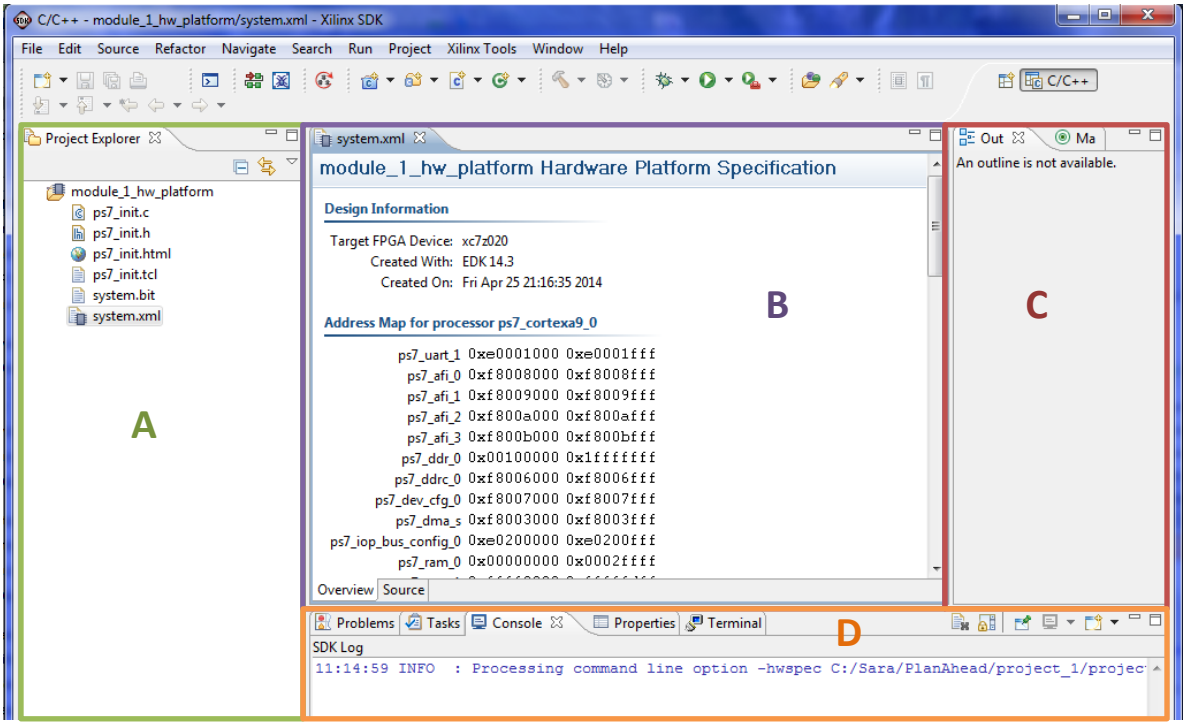


FIGURA 48. ESPECIFICACIONES DE LA PLATAFORMA HARDWARE EXPORTADA A SDK

Al iniciar únicamente se ha exportado desde PlanAhead el módulo hardware del proyecto, incluyendo la información generada desde XPS. Así, puede verse en el explorador del proyecto *module_1_hw_platform,* que contiene el fichero system.xml, el cual se encuentra abierto en el panel de visualización de ficheros (zona B).

En este fichero se muestra información del módulo hardware que se ha exportado desde PlanAhead. Muestra las direcciones de memoria para ambos procesadores (Figura 49) y los bloques IP que contiene (Figura 50).

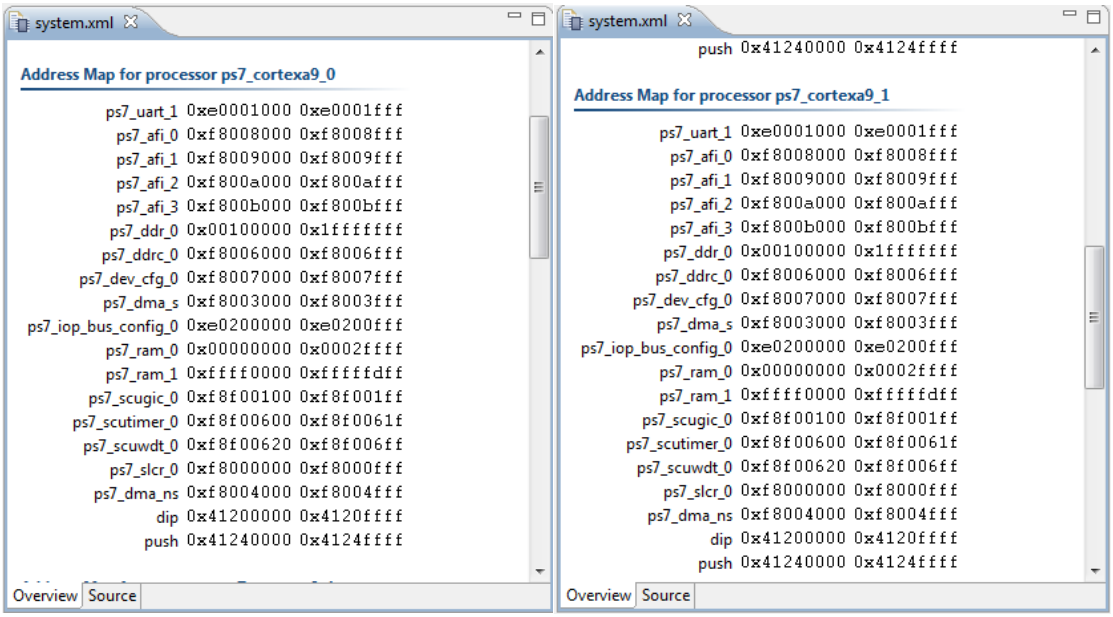


FIGURA 49. MAPA DE DIRECCIONES DE LOS DOS PROCESADORES CONTENIDOS EN LA TARJETA

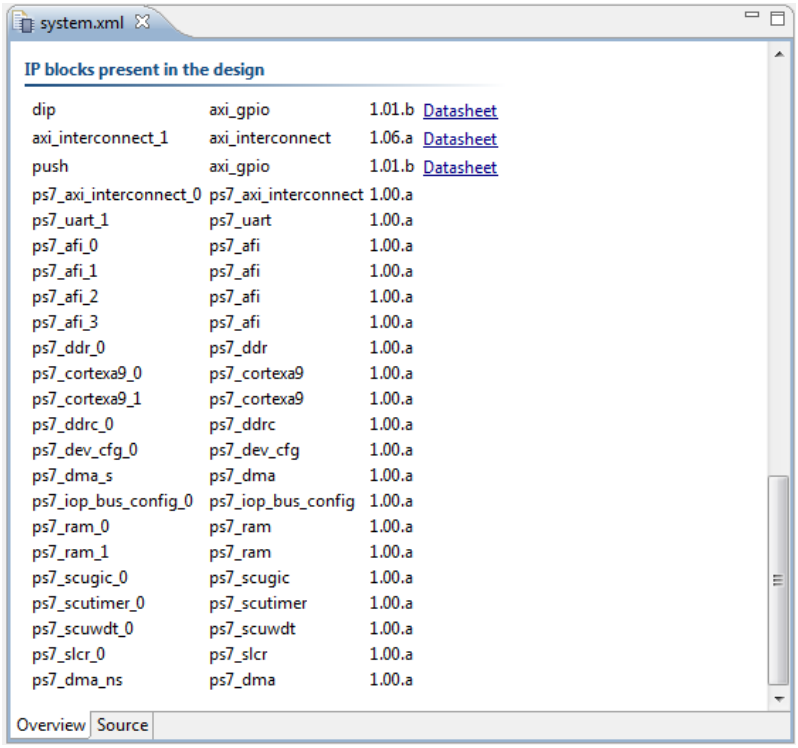


FIGURA 50. BLOQUES IP CONTENIDOS EN LA PLATAFORMA HARDWARE EXPORTADA

Para continuar con el diseño, hay que generar una plataforma de diseño software sobre la plataforma de diseño hardware que se ha exportado, lo que se conoce como Board Support Package, que es un conjunto de componentes software que permiten al sistema operativo correr sobre una plataforma hardware específica.

Contiene, entre otros: un boot loader que inicializa el hardware, carga el S.O y lanza la rutina de startup; y los drivers del dispositivo. Puede crearse desde *File* > *New* > *Board Support Package*.

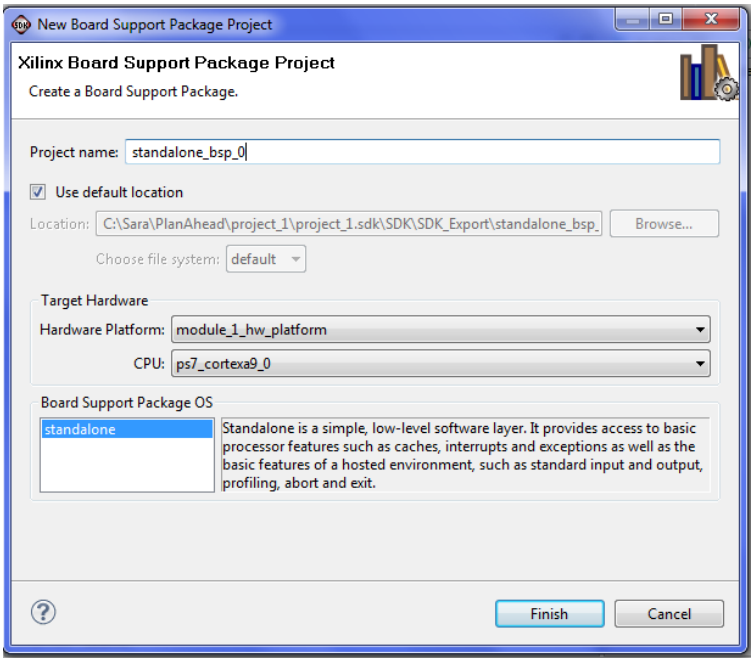


FIGURA 51. CREACIÓN BSP PARA DISEÑO SOFTWARE EN SDK

En la ventana que se abre (Figura 51) es recomendable no modificar los valores por defecto que muestra la ventana, ya que viene con un nombre identificativo, la localización recomendada dentro del proyecto, el módulo hardware asociado, el procesador a emplear y sistema operativo autónomo. Al finalizar, el BSP es añadido al explorador de proyecto y se abre una ventana mostrando su configuración actual.

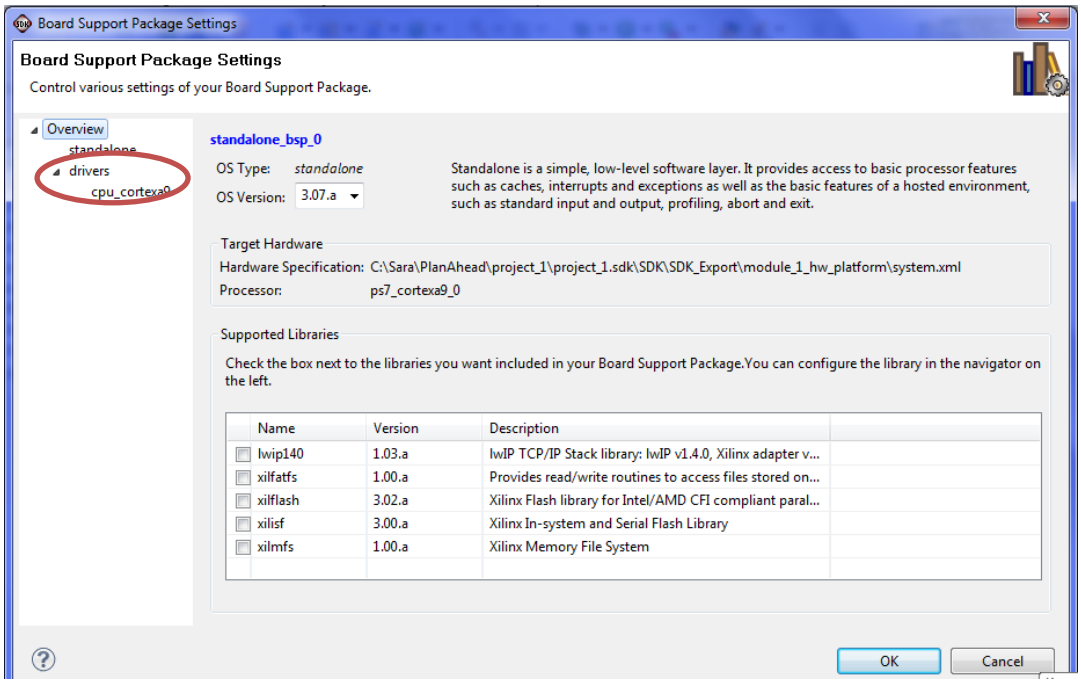


FIGURA 52. CONFIGURACIÓN ACTUAL BSP

Lo único a destacar en esta ventana de ajustes del BSP es la pestaña de drivers resaltada con el círculo en rojo en la Figura 52, donde será necesario acceder a la hora de utilizar un periférico creado por el usuario para asignarle los drivers creados en XPS para ese periférico. No conviene entrar en más detalles por el momento puesto que esto podrá verse posteriormente en el diseño correspondiente a la creación de un periférico.

A su vez, seleccionando con botón derecho sobre el BSP creado, *standalone_bsp_0*, en *Properties* > *Project Rerefences*, puede verse en la Figura 53 que el BSP está asociado a la plataforma hardware que se exportó de PlanAhead y se importó en el proyecto de SDK.

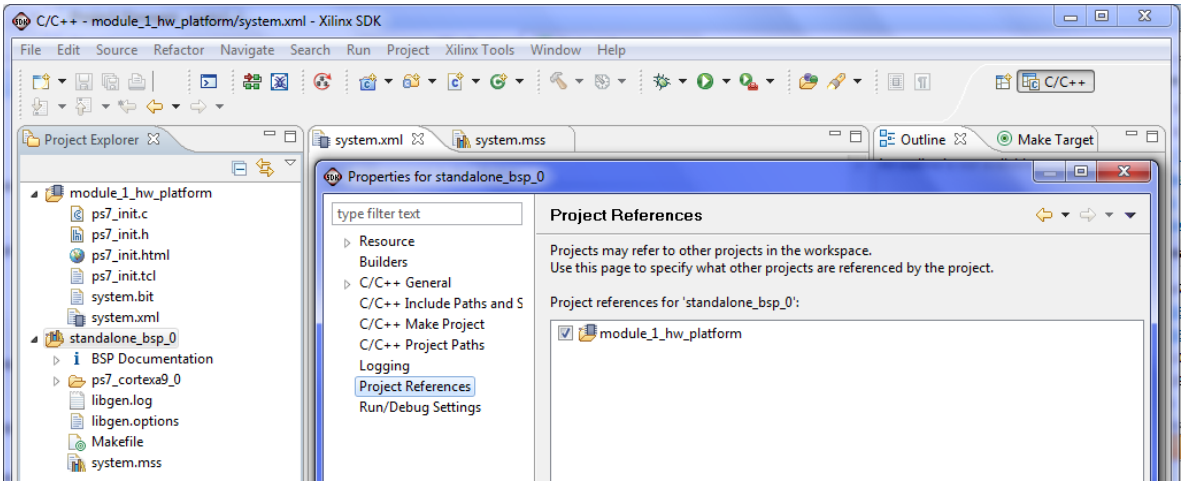


FIGURA 53. ASOCIACIÓN BSP CON EL MÓDULO HARDWARE DEL DISEÑO

Una vez se dispone de una plataforma software, va a añadirse una aplicación de usuario sobre el BSP creado. Puede crearse escogiendo *File* > *New* > *Application Project*.

En esta ventana (ver Figura 54) ya vienen marcadas las opciones recomendadas por defecto a excepción del nombre de la aplicación, donde se ha elegido llamarla Test, y de configurarlo para emplear el BSP ya creado, que debe indicarse en *Use existing: standalone_bsp_0*. Además, da la opción de seleccionar S.O Standalone o Linux, que también es soportado por la tarjeta, y de elegir lenguaje de programación C o C++.

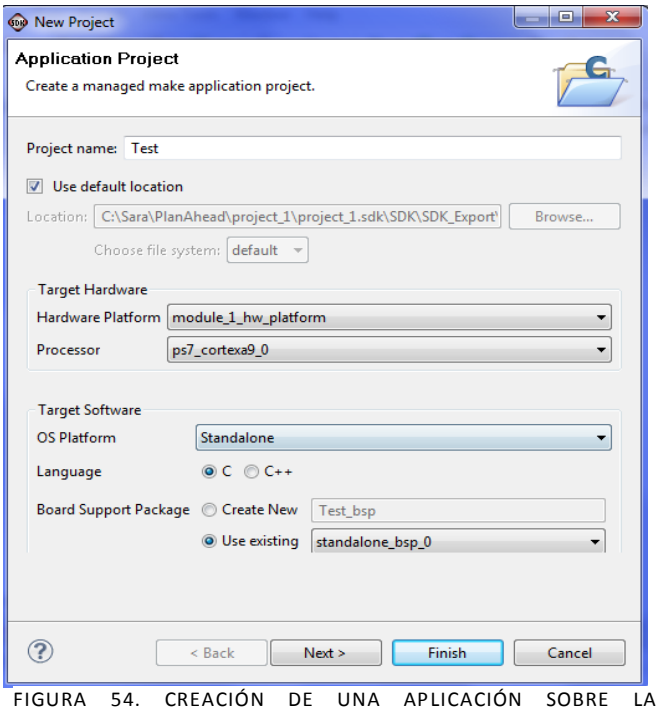


FIGURA 54. CREACIÓN DE UNA APLICACIÓN SOBRE LA PLATAFORMA SOFTWARE CREADA EN SDK

La siguiente ventana, mostrada en la Figura 55, completa el proceso de creación de la aplicación. Contiene las plantillas de las que dispone SDK, como *Hello World*, *Memory Tests* y *Peripheral Tests*, como posibles aplicaciones para probar el hardware.

En el caso general en que se quiera comenzar de cero debe seleccionarse *Empty Application*, aunque otra opción es escoger la plantilla *Hello World* para disponer de un código rápido y sencillo que probar en la FPGA.

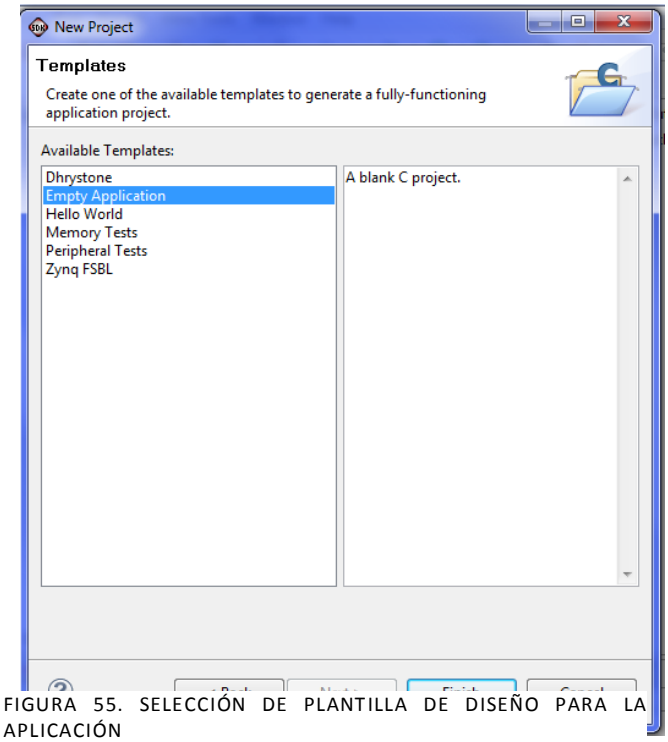


FIGURA 55. SELECCIÓN DE PLANTILLA DE DISEÑO PARA LA APLICACIÓN

Al finalizar, se habrá añadido la aplicación Test al explorador de proyecto, y consultando sus propiedades, en *Project References*, puede verse que está asociada al BSP creado previamente, el cual estaba asociado a la plataforma hardware del diseño.

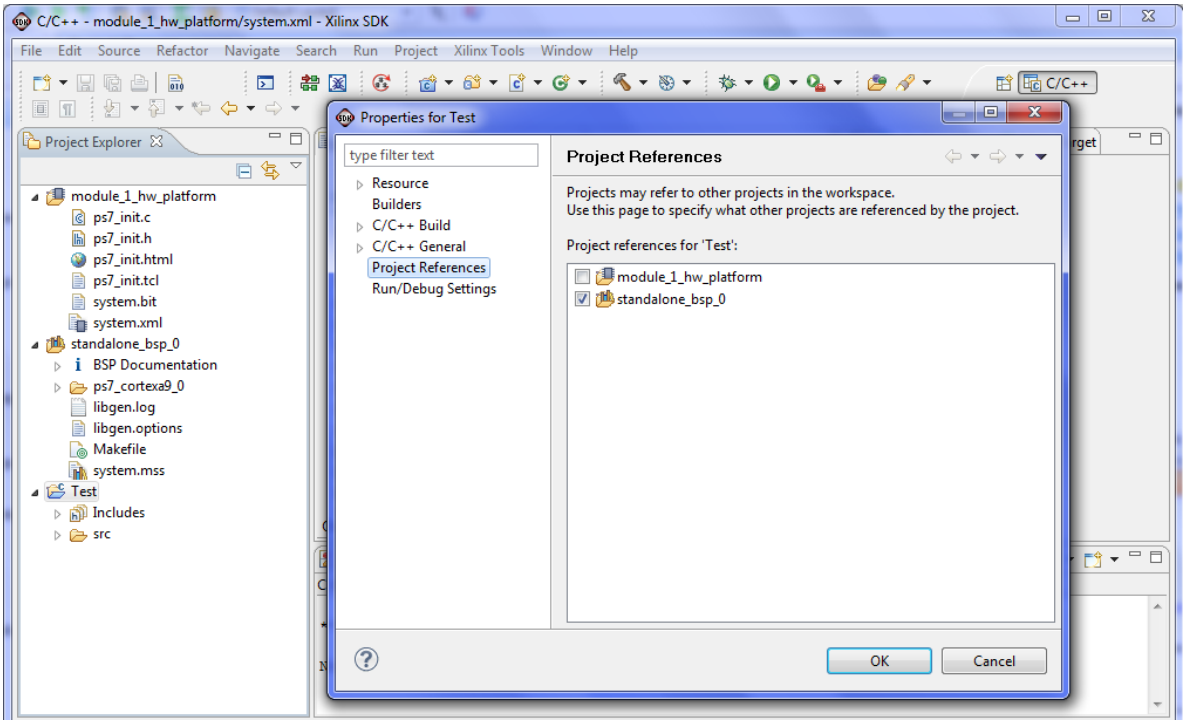


FIGURA 56. PROPIEDADES DE LA APLICACIÓN DE USUARIO CREADA EN SDK

El siguiente paso es añadir el fichero *.c (en este caso llamado program.c) que contiene la aplicación software. Para importarlo al proyecto debe hacerse mediante botón derecho sobre el proyecto, Test, escoger *Import>General>File System>Browse…* y seleccionar el directorio donde se encuentra. Todos los ficheros contenidos en ese directorio se mostrarán en la Figura 58 donde se ha de marcar el fichero deseado.

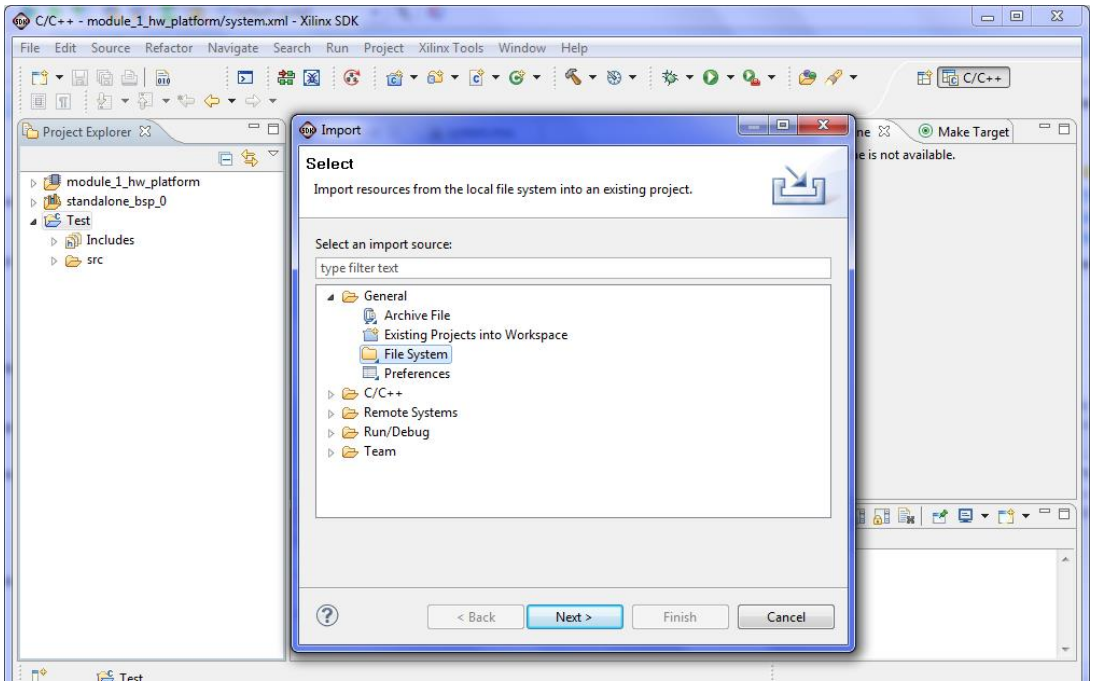


FIGURA 57. IMPORTACIÓN DEL FICHERO FUENTE QUE AÑADIR A LA APLICACIÓN EN SDK (II)

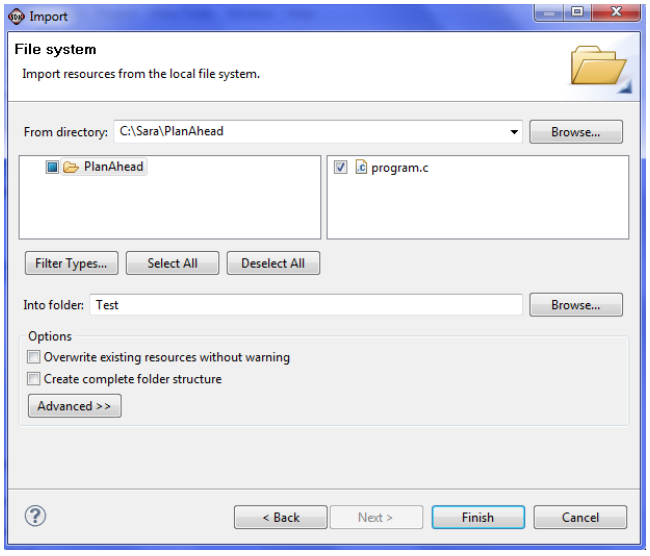


FIGURA 58. IMPORTACIÓN DEL FICHERO FUENTE QUE AÑADIR A LA APLICACIÓN EN SDK (II)

Al finalizar, el fichero es añadido al proyecto asociándolo al contexto de manera automática y compilándose el proyecto automáticamente para generar el fichero *.elf que caracteriza al fichero propio de SDK y que servirá para programar la FPGA junto con el *.bit (ver Figura 9).

Abriendo el fichero program.c con el código del programa que se quiere realizar en este caso, se observa un código sencillo cuya función es inicializar tanto switches como pushbuttons, configurando ambos como entradas, y entrar en un bucle de programa donde se lee el estado de ambos GPIO, almacenándolo en una variable que posteriormente se imprime por pantalla.

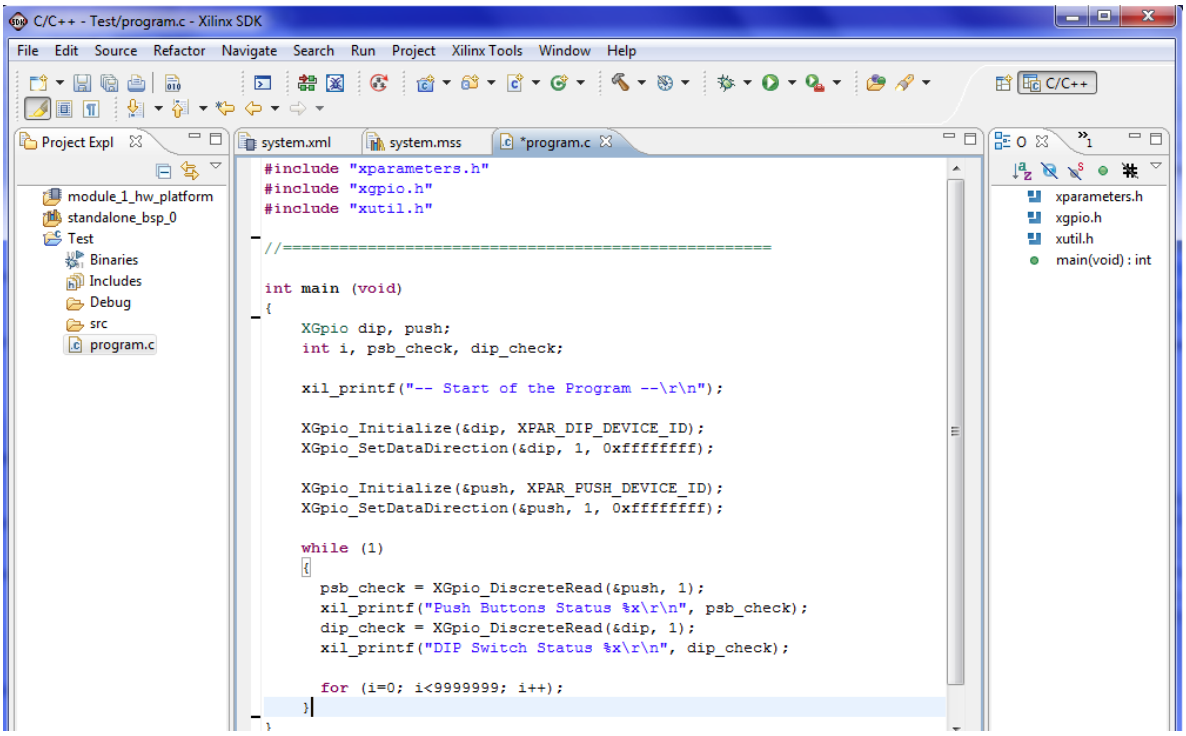


FIGURA 59. PANEL DE VISUALIZACIÓN CON EL FICHERO FUENTE IMPORTADO

Por el momento puede verse en el código que se emplea: la función *XGpio_Initialize(*InstancePtr, DevideId)* para inicializar los GPIO; la función *XGpio_SetDataDirection(*InstancePtr, Channel, DirectionMask)* para configurarlos como sentido de entrada; la función *XGpio_DiscreteRead (*InstancePtr, Channel)* para leer la entrada de los GPIO; y la función *xil_printf* para imprimir por pantalla lo que se ha leído.

El diseño incluye unas cabeceras que fueron creadas o añadidas en el proyecto de forma transparente al usuario en el momento de creación del BSP (por lo que pueden localizarse en el directorio propio del BSP *C:\...\project_1\project_1.sdk\SDK\SDK_Export\standalone_bsp_0\ps7_cortexa9_0\include*). Las cabeceras de las que hace uso program.c son:

- *Xparameters.h*, que contiene las definiciones para todos los parámetros de los periféricos contenidos en el diseño.
- *Xgpio.h*, que contiene la definición de las funciones que pueden utilizarse para el manejo de los periféricos en el código de programa.
- *Xutil.h,* contiene funciones útiles que consisten en distintos test para pruebas de memoria.

Las funciones comentadas en el código de la Figura 59 se localizan en la cabecera *xgpio.h*, y los parámetros pasados a esas funciones se encuentran en *xparameters.h*. Por tanto, a partir de ambas cabeceras puede completarse el diseño software.

En *xparameters.h* mostrada en la Figura 60 aparecen las definiciones de parámetros que caracterizan a los periféricos *dip* y *push* y al número de GPIO instanciados. No obstante, la Tabla 1 muestra otras definiciones contenidas en la cabecera que pueden resultar útiles.

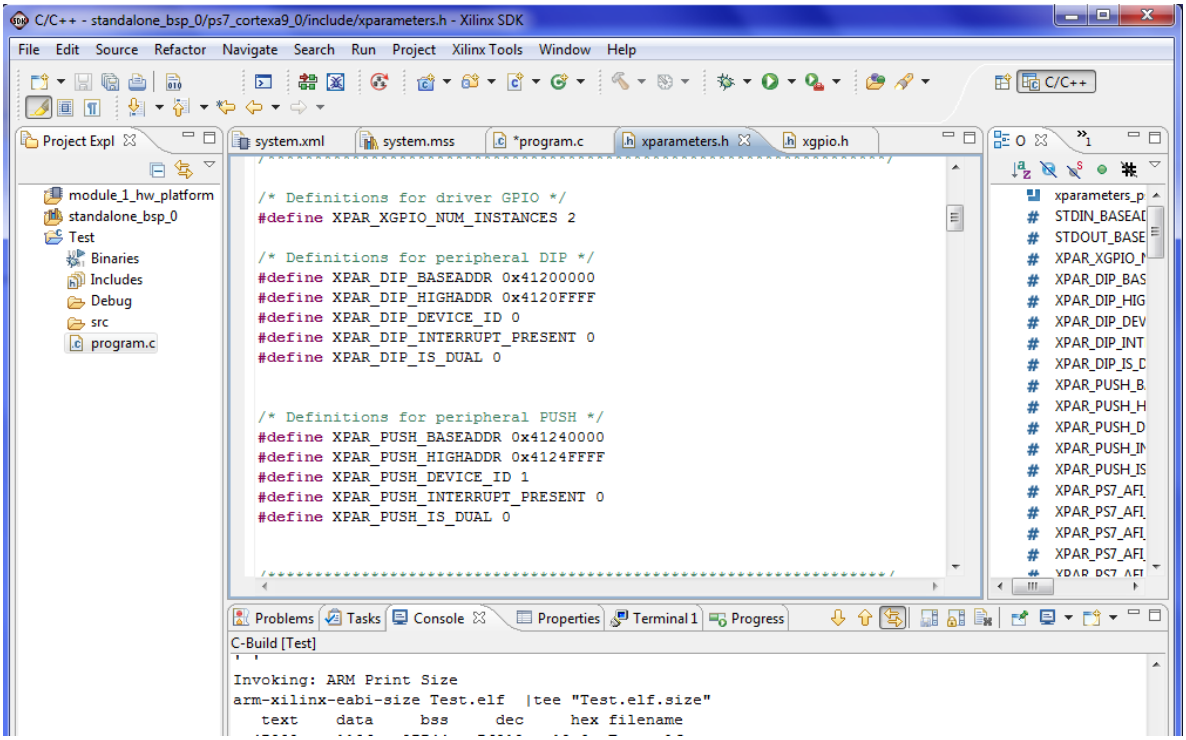


FIGURA 60. CONTENIDO DE LA CABECERA XPARAMETERS.H DEL DISEÑO SDK

CABECERA XPARAMETERS.H	
TIPO DE DEFINICIONES	*PARÁMETROS QUE CONTIENEN*
Definitions for driver GPIO	Num_Instances
Definitions for peripheral DIP	BaseAddr,HighAddr,DeviceId,InterruptPresent,Dual
Definitions for peripheral PUSH	BaseAddr,HighAddr,DeviceId,InterruptPresent,Dual
Definitions for peripheral PS7_DDR_0	BaseAddr,HighAddr,HP0BaseAddr,HP0HighAddr,…
Definitions for peripheral PS7_DDR_C	BaseAddr,HighAddr
Definitions for peripheral PS7_IOP_BUS_CONFIG_0	BaseAddr,HighAddr
Definitions for peripheral PS7_RAM_X X=0,1	BaseAddr,HighAddr,HP0HighOCMBaseAddr,HP0HighOcmHigh Addr,HP1HighOCMBaseAddr…
Definitions for peripheral PS7_UART_1	DeviceId,BaseAddr,HighAddr,UART_clk_freq_hz

TABLA 1. DEFINICIONES CONTENIDAS EN LA CABECERA XPARAMETERS.H

En *xgpio.h* mostrada en la Figura 61 aparecen las principales funciones para inicialización y recogida de valores que hacen uso de las definiciones de *xparameters.h*. De igual forma, la Tabla 2 muestra al completo todas las funciones que se pueden usar.

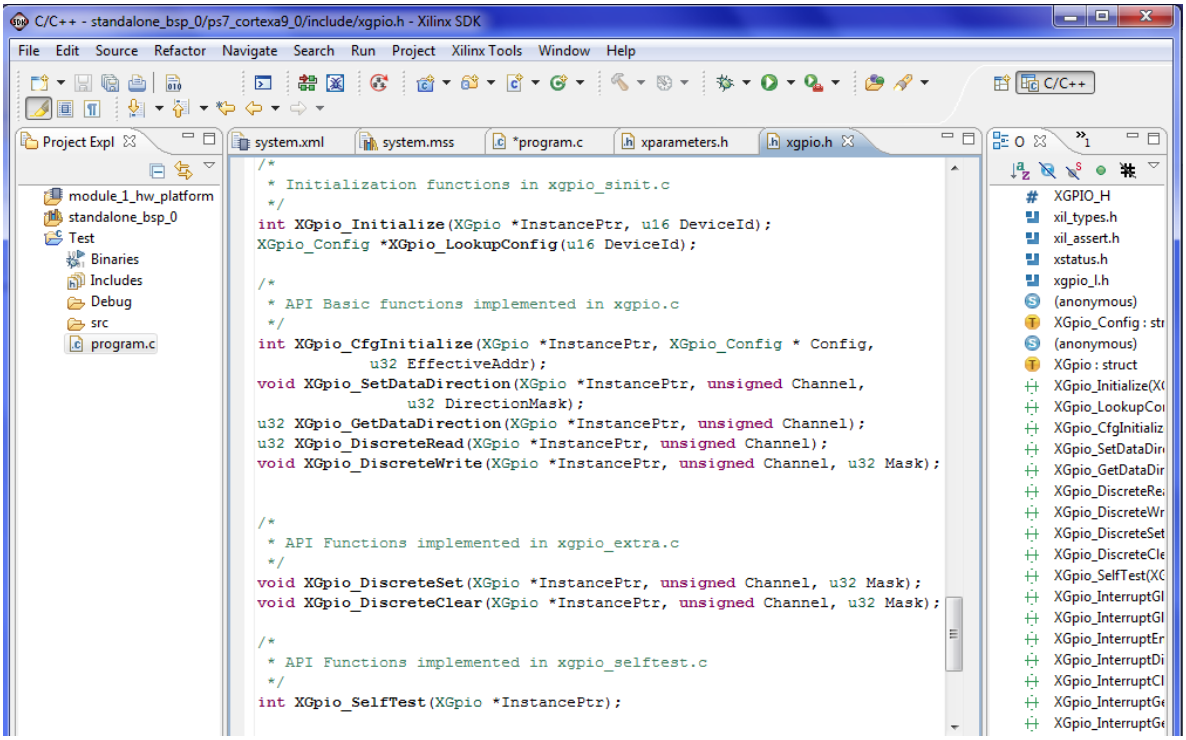


FIGURA 61. CONTENIDO DE LA CABECERA XGPIO.H DEL DISEÑO SDK

CABECERA XGPIO.H	
typedef struct { u16 DeviceId; u32 BaseAddress; int InterruptPresent; int IsDual; } XGpio_Config;	typedef struct { u32 BaseAddress; u32 IsReady; int InterruptPresent; int IsDual; } XGpio;
Funciones de inicialización	int XGpio_Initialize(XGpio *InstancePtr, u16 DeviceId); XGpio_Config *XGpio_LookupConfig(u16 DeviceId);

Funciones para manejo de GPIO	int XGpio_CfgInitialize(XGpio *InstancePtr, XGpio_Config * Config,u32 EffectiveAddr);
	void XGpio_SetDataDirection(XGpio *InstancePtr, unsigned Channel, u32 DirectionMask);
	u32 XGpio_GetDataDirection(XGpio *InstancePtr, unsigned Channel);
	u32 XGpio_DiscreteRead(XGpio *InstancePtr, unsigned Channel);
	void XGpio_DiscreteWrite(XGpio *InstancePtr, unsigned Channel, u32 Mask);
	int XGpio_SelfTest(XGpio *InstancePtr);

Funciones para interrupción	void XGpio_InterruptGlobalEnable(XGpio *InstancePtr);
	void XGpio_InterruptGlobalDisable(XGpio *InstancePtr);
	void XGpio_InterruptEnable(XGpio *InstancePtr, u32 Mask);
	void XGpio_InterruptDisable(XGpio *InstancePtr, u32 Mask);
	void XGpio_InterruptClear(XGpio *InstancePtr, u32 Mask);
	u32 XGpio_InterruptGetEnabled(XGpio *InstancePtr);
	u32 XGpio_InterruptGetStatus(XGpio *InstancePtr);

TABLA 2. FUNCIONES CONTENIDAS EN LA CABECERA XGPIO.H

Como dato útil, hay que comentar que cada vez que uno de los archivos del programa es modificado, en el momento de guardar cambios la herramienta lleva a cabo la compilación mostrando mensajes de proceso en la consola. Además, es posible observar una barra de procesos con la acción que se está llevando a cabo, la cual es útil también en el caso en que el programa se quede colgado, permitiendo al usuario terminar la tarea que lo bloquea.

El siguiente paso es probar el proyecto en hardware para comprobar si funciona correctamente y corregir posibles errores de código.

Se conecta la tarjeta Zedboard de forma adecuada al equipo y a la alimentación y se enciende [8]. Una vez el equipo la haya reconocido se debe buscar el *puerto COM* al que está asociada, ya que esto varía en función del equipo y del puerto. También debemos asegurarnos de tener conectada la salida UART de la tarjeta al puerto USB del equipo para poder visualizar lo que el programa imprimirá por pantalla desde la pestaña *Terminal* de SDK.

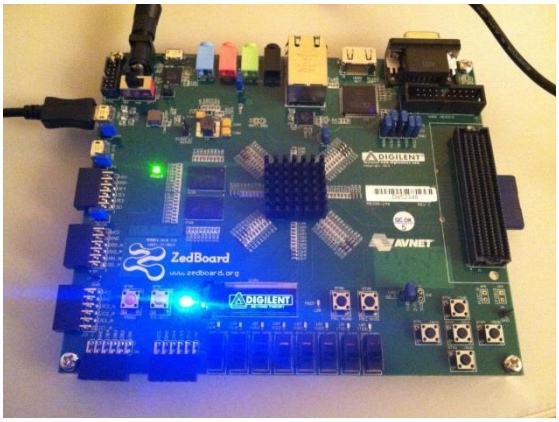


FIGURA 62. CONEXIÓN TARJETA ZEDBOARD

A continuación, hay que cargar el programa en la FPGA de la tarjeta. Para ello en SDK puede hacerse clic sobre el icono denominado *Program FPGA* o acceder a la herramienta desde *Xilinx Tools>Program FPGA*.

Aparecerá una ventana donde incluir el directorio del bitstream generado con anterioridad (*.bit), que es el fichero que se descargará a la tarjeta y que fue exportado en su momento desde PlanAhead al exportar el diseño hardware a SDK. Por lo general se muestra por defecto el directorio correcto, pero es conveniente comprobarlo como paso previo a la programación.

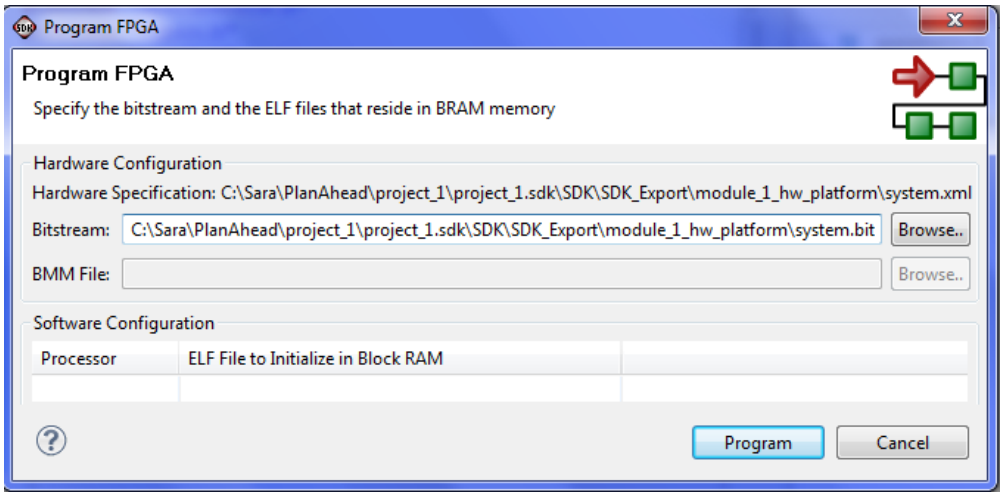


FIGURA 63. PROGRAMACIÓN DE LA FPGA PARA LA PRUEBA EN HARDWARE DEL DISEÑO REALIZADO

Tras el proceso de programación debe encenderse el led LD12 de la tarjeta en color azul (ver Figura 62) como indicación de que se ha realizado con éxito, además de no aparecer ningún mensaje de error en la consola.

Para poder visualizar las variables que se imprimirán por pantalla debe conectarse el *Terminal* a la tarjeta. Para abrir los ajustes de conexión del terminal se hace clic sobre el icono *Settings* de la pestaña *Terminal*, configurándose la comunicación serie como se indica en la Figura 64.

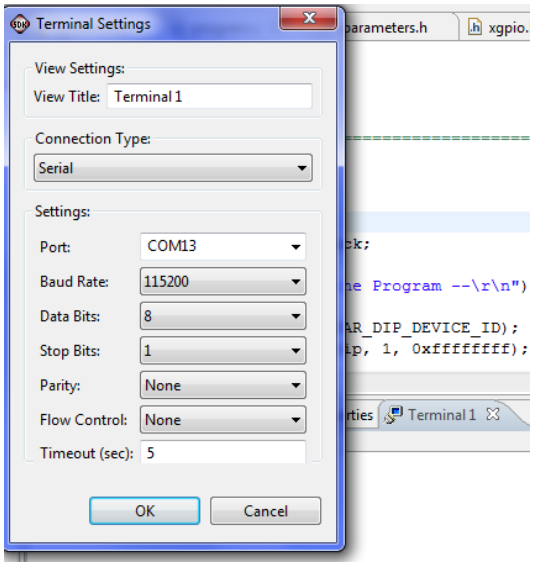


FIGURA 64. CONFIGURACIÓN DE LA COMUNICACIÓN SERIE CON LA TARJETA

Al aceptar debe mostrarse en la parte superior del terminal el mensaje: *Serial: (COM13, 115200, 8, 1, None, None – CONNECTED)*. Una vez se ha configurado la conexión ésta se queda guardada y no es necesario entrar en *Settings* para conectarla en pruebas hardware posteriores sino que puede hacerse pulsando el icono *Connect* de la pestaña *Terminal*.

Ahora, para ejecutar el programa debe hacerse botón derecho sobre el nombre de la aplicación, Test > *Run As…* > *Launch on Hardware*, y se mostrará en la esquina inferior de la herramienta el mensaje *Launching Test.elf*. Una vez haya comenzado la ejecución puede comprobarse su funcionamiento desde el terminal.

Lo que se observa en el terminal es una muestra continua debida al bucle *while(1)* de mensajes impresos por pantalla. Inicialmente se mostrarán los mensajes *DIP Switch Status 0* y *Push Buttons Status 0*, ya que tanto switches como pushbuttons están inicialmente sin pulsar. En el momento en que se pulsa alguno de los cinco botones de los que dispone la tarjeta, puede verse en el Terminal el mensaje con el valor asociado a ese pushbutton, siendo 1 (BTNL left), 2 (BTNR right), 4 (BTNU up), 8 (BTND down) y 10 (BTNC central), si se pulsan de manera independiente, o la suma de sus valores asociados en caso de pulsar varios botones a la vez. De igual forma sucede con los switches, en este caso la tarjeta dispone de ocho dips, correspondiendo los cuatro inferiores (SW3-0) a la parte menos significativa del valor hexadecimal mostrado por pantalla, y los cuatro superiores (SW7-4) a la más significativa.

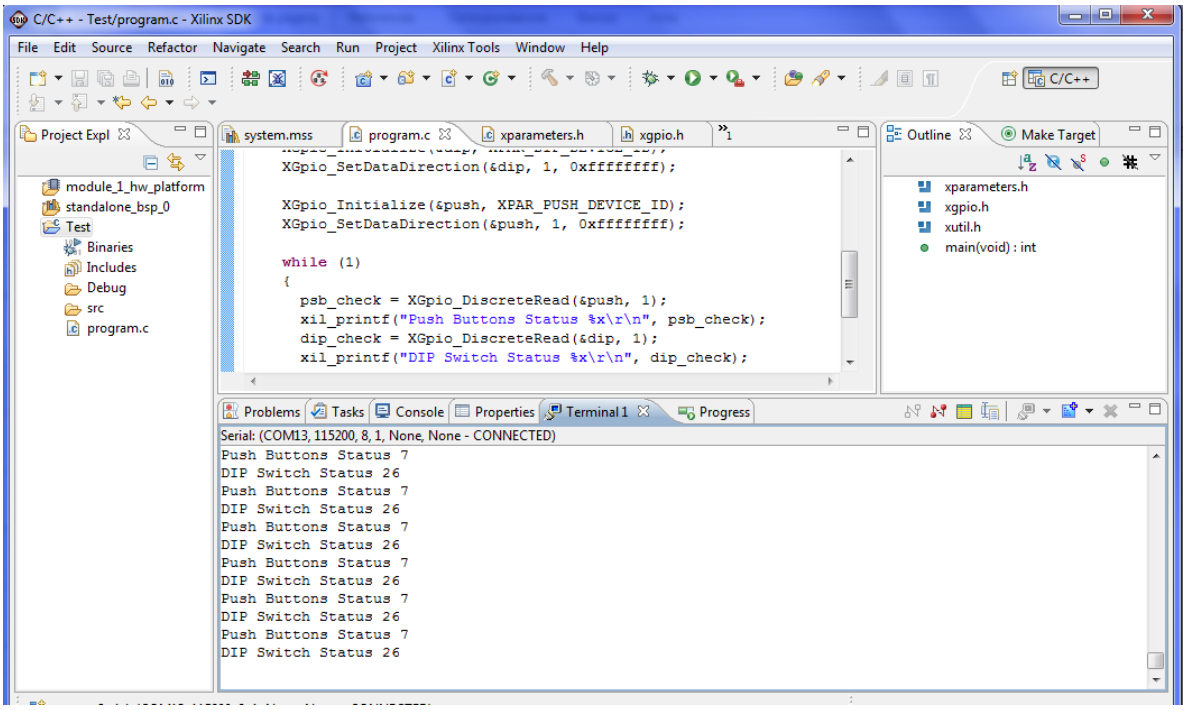


FIGURA 65. OBSERVACIÓN DE MENSAJES IMPRESOS POR EL TERMINAL RESULTADO DE LA PRUEBA EN LA TARJETA

En la parte de ejecución que se muestra en la Figura 65 se habían activado los switches *SW5, SW2* y *SW1*, y se estaban manteniendo pulsados los botones *BTNL, BTNR* y *BTNU*, por ello aparece el valor 0x26 para dips y 7 para pushbuttons.

Finalmente puede desconectarse la comunicación serie en el icono *Disconnect* de la pestaña *Terminal*, y terminar la ejecución en el icono *Terminate* de la pestaña *Console*.

3.1.6. **Periférico creado por el usuario mediante IPIF e IPIC**

El objetivo que se persigue es mostrar el procedimiento a seguir para añadir un periférico, con funcionalidad diseñada por el usuario, a la parte hardware del proyecto.

El periférico que se pretende diseñar es una puerta AND de cuatro entradas y una salida. Las entradas se dividen en tres entradas externas provenientes de tres dips de la tarjeta, las cuales serán asignadas en hardware, y una entrada interna proveniente de un registro de usuario compartido por el micro y la parte lógica.

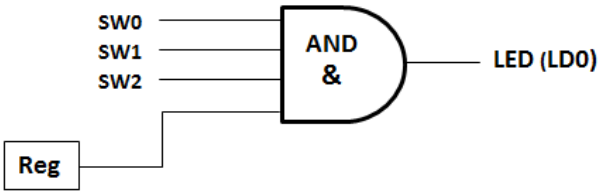


FIGURA 66. ENTRADAS Y SALIDA DE LA PUERTA AND

La salida será externa y mostrará el resultado de la operación and entre las cuatro entradas en un led de la tarjeta, iluminándose éste para el caso de que las cuatro entradas estén activas.

El código .vhd del periférico a diseñar es el siguiente:

```
library ieee;
use ieee.std_logic_1164.all;

entity and4 is
  port (
    DIP0, DIP1, DIP2 : in  std_logic;
    LED0 : out std_logic
       );
end entity and4;

architecture rtl of and4 is
  signal in4 : std_logic;

  begin
process (DIP0, DIP1, DIP2) is
  begin

  slv_reg1(0) <= DIP0;
  slv_reg2(0) <= DIP1;
  slv_reg3(0) <= DIP2;
  slv_reg1(31 downto 1) <= (others => '0');
  slv_reg2(31 downto 1) <= (others => '0');
  slv_reg3(31 downto 1) <= (others => '0');

  in4 <= slv_reg0(0);
  LED0 <= ((DIP0 and DIP1) and DIP2) and in4;

  end process;
end architecture rtl;
```

FIGURA 67. CÓDIGO EN VHDL PUERTA AND

A modo de introducción puede verse en la Figura 67 que consta de un proceso dependiente del valor de los switches, en el cual el bit referido al switch 0, 1 y 2 se va a almacenar en el bit 0 de los registros 1, 2 y 3, respectivamente, y el valor de la cuarta entrada, in4, vendrá dado por el bit 0 del registro 0. Finalmente el resultado de la operación AND se pasará al puerto LED0.

Este nuevo diseño se va a realizar sobre el proyecto anterior que sirvió de modelo para conocer el uso de las herramientas PlanAhead, XPS y SDK.

Abriendo el proyecto anterior, *proyect_1*, en PlanAhead, se lanza XPS haciendo doble clic sobre el fichero *module_1.xmp* situado en la ventana de fuentes del proyecto. En XPS puede verse que, hasta ahora, la parte PS del sistema está comunicada con la parte PL mediante el protocolo AXI4Lite, estando habilitada la interfaz *M_AXI_GP0* en PS, a la cual a través de *axi_interconnect_1* y la interfaz *S_AXI* en PL están conectados dos GPIO. Estos GPIO, dips y botones, fueron instanciados desde el catálogo IP en el diseño anterior, para lo cual se recuerda que únicamente hizo falta darles nombre y ancho de bits, configurarlos como puertos externos y asociarlos en el fichero*.ucf a los pines de la FPGA.

Para el diseño que pretende realizarse ahora, no se necesitan estos dos periféricos ya que se va a crear uno consistente en una puerta AND, por lo que se van a eliminar del proyecto con simplemente seleccionarlos y suprimirlos desde la pestaña *Bus Interfaces*, marcando *Delete Instance and all its connections* para un borrado completo.

Ahora el punto de vista es diferente, con el asistente de creación de periféricos se siguen los siguientes pasos: se le da nombre al periférico, se selecciona el protocolo de comunicación, se elige el número de registros en la lógica de usuario accesibles por software y su ancho de bits, y se generan, opcionalmente, plantillas para ayudar al usuario a implementar la funcionalidad del periférico.

Este asistente puede lanzarse en XPS desde *Hardware>Create or Import Peripheral*. Una vez se ha abierto la herramienta, se elige si se quiere crear o importar un periférico, en este caso se selecciona la opción que permite crear plantillas para un nuevo periférico (*Create templates for a new peripheral*), que dentro del proyecto que serán modificadas por el usuario para incorporar la funcionalidad del periférico.

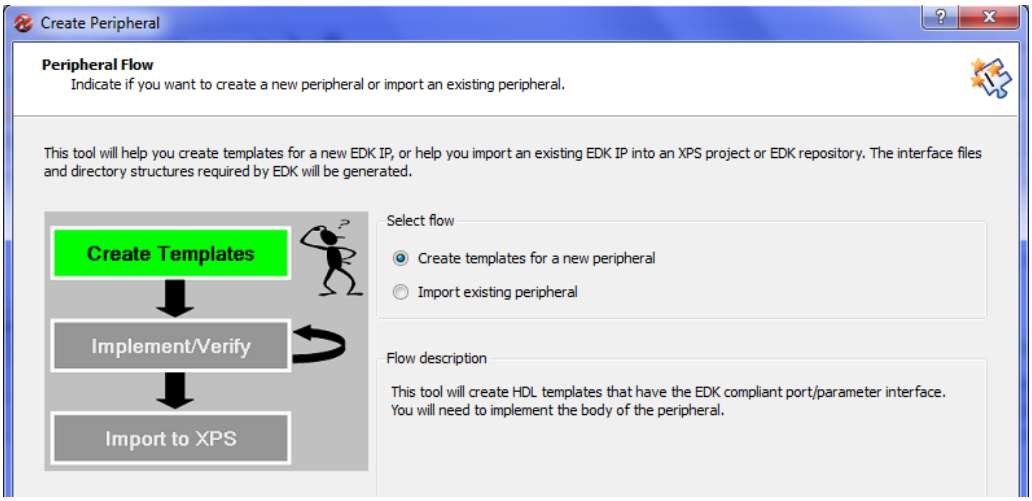


FIGURA 68. CREACIÓN DE UN PERIFÉRICO EN XPS - NUEVA PLANTILLA

En la ventana siguiente (Figura 69) puede elegirse si almacenar el periférico en un repositorio fuera del proyecto (donde queda accesible desde múltiples proyectos para su importación) o almacenarlo únicamente en el proyecto actual, que es la opción que escogemos para este caso porque por el momento no se va a hacer uso de él en próximos proyectos.

Puede verse en la parte inferior de la Figura 69 el directorio donde se va a situar el periférico una vez sea creado, contenido en la carpeta *pcores* dentro de la carpeta correspondiente a la plataforma hardware que fue creada en su momento en XPS, *module_1*.

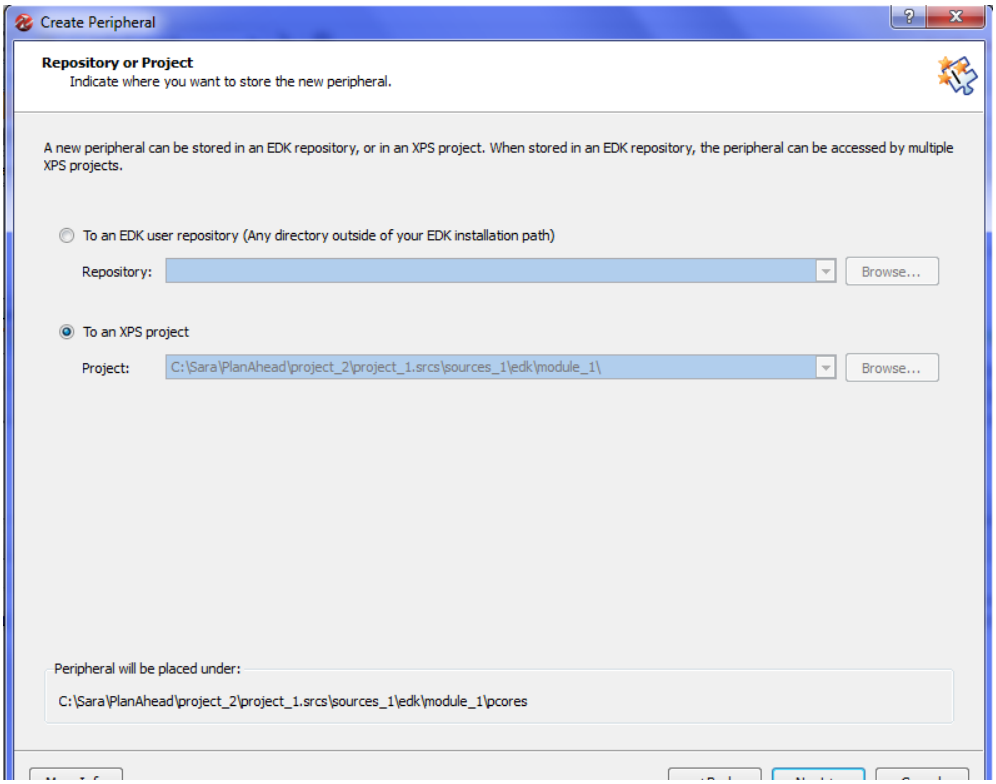


FIGURA 69. CREACIÓN DE UN PERIFÉRICO EN XPS - LOCALIZACIÓN DEL PERIFÉRICO

A continuación se le da nombre al periférico, en este caso se ha decidido darle el nombre de and4, y se selecciona la versión, donde se dejará la versión que aparece por defecto.

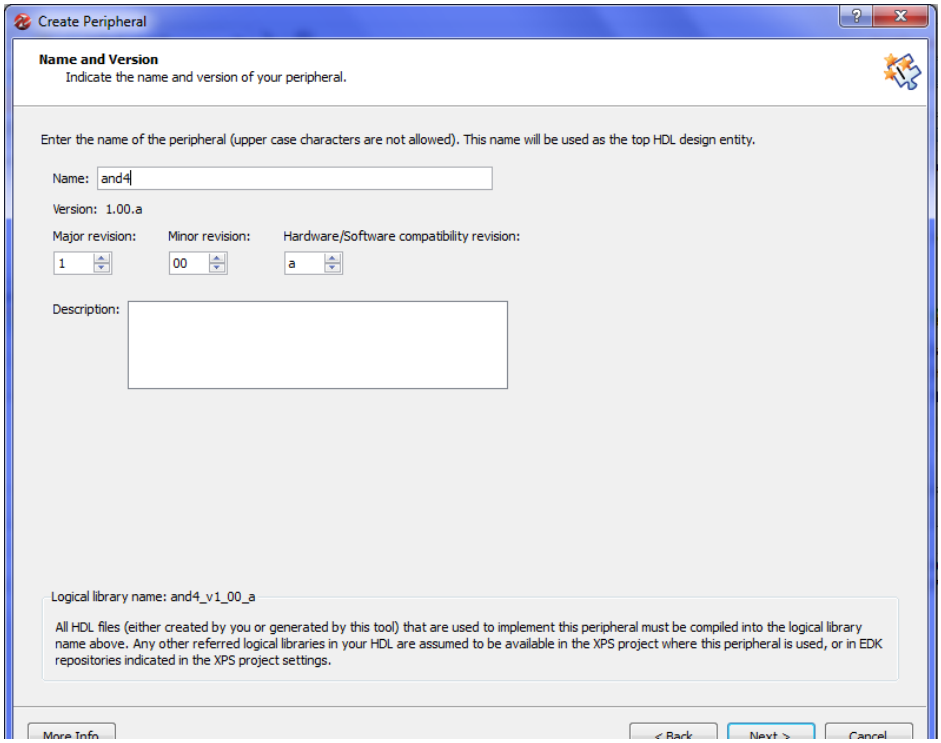


FIGURA 70. CREACIÓN DE UN PERIFÉRICO EN XPS - NOMBRE Y VERSIÓN

En la ventana siguiente, como interfaz soportada por el periférico, se escoge por simplicidad AXI4-Lite.

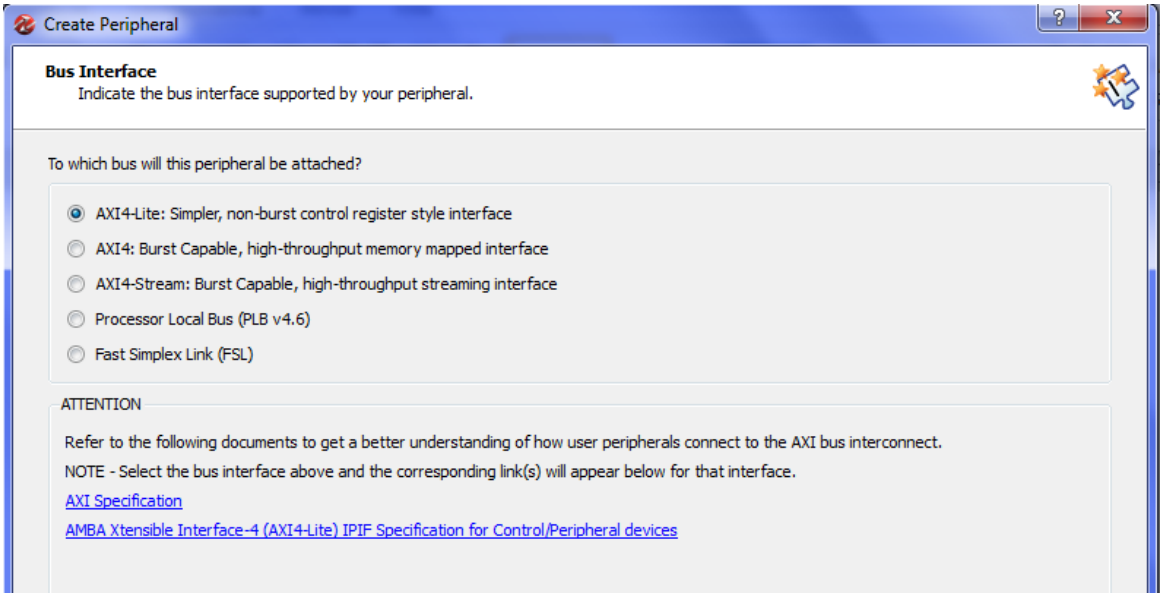


FIGURA 71. CREACIÓN DE UN PERIFÉRICO EN XPS - INTERFAZ DE COMUNICACIÓN

Se podría decir que llegados a este punto comienza en sí la herramienta para el diseño de periféricos denominada IPIF (*AXI IP Interface*).

A través de módulos IPIF se implementa la interfaz entre AXI4 Interconnect y la lógica de usuario, ofreciendo el asistente los servicios más comunes en la configuración de maestro (*Master service and configuration*) y de esclavo (*Slave service and configuration*).

En la configuración de maestro se encuentra el siguiente servicio:

- *User logic master support*: permite al periférico creado actuar de maestro e iniciar él los procesos de lectura o escritura, en vez de ser quien responda a ellos.

Por otro lado, en la configuración de esclavo pueden seleccionarse los servicios:

- *Software reset:* permite que el periférico sea reseteado mediante software a través del ARM.
- *Include data phase timer*: actúa en cierto modo como un watchdog, fijando un número entero a este parámetro se fija el número de ciclos de reloj que espera la interfaz AXI4 Lite a recibir una confirmación (*acknowledgement*) del periférico antes de forzar a terminar la transacción.
- *User logic software register*. La opción *Software reset* La opción *Include data phase timer* Y la opción User Logic software register permite el acceso a registros direccionables por software para la comunicación con el periférico.

Para este periférico, y en la mayoría de los periféricos a diseñar, se emplea la configuración de esclavo, dejando al procesador que actúe de maestro. Dentro de las configuraciones de esclavo para nuestro periférico se escogerá el servicio *User logic software register,* ya que se necesita acceder a los registros del periférico para lectura/escritura de las entradas, debido a que la cuarta entrada se encontrará en uno de estos registros y en las otras tres entradas puede ser necesario comprobar su valor leyendo de sus registros para probar el funcionamiento.

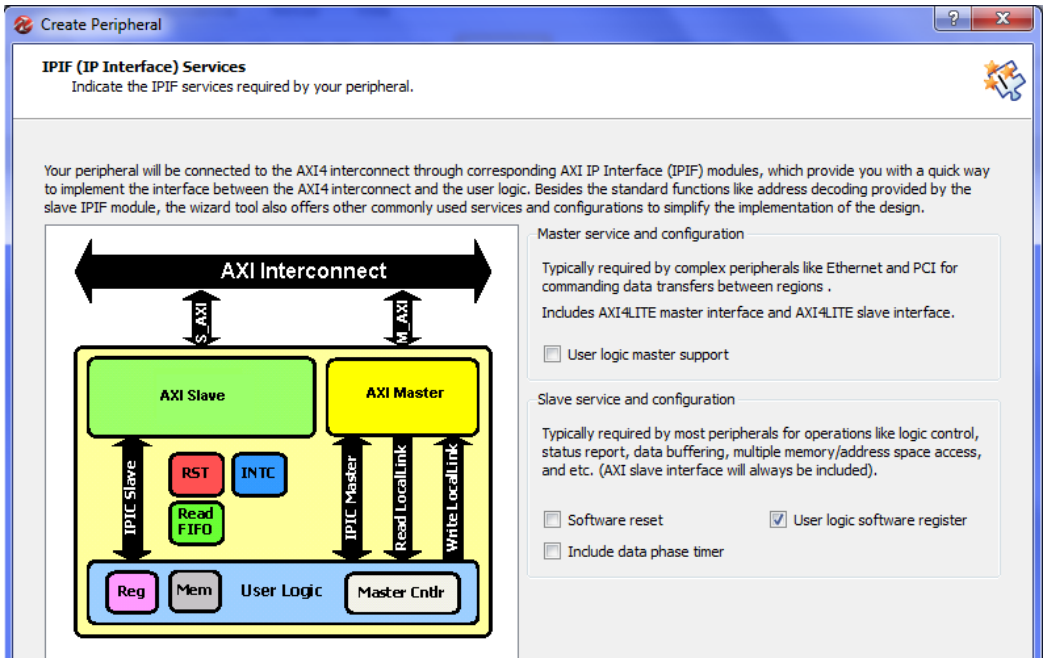


FIGURA 72. CREACIÓN DE UN PERIFÉRICO EN XPS - CONFIGURACIÓN DE ACCESO A REGISTROS POR SOFTWARE

A continuación se muestra la ventana correspondiente al servicio señalado. En esta ventana se escoge el número de registros en el diseño que se desea permitir que sean accedidos desde el software. Además se genera un conjunto de funciones personalizadas para este periférico que permiten realizar operaciones tales como lectura o escritura de dichos registros.

Puede verse en la Figura 73 que el número de registros accesibles puede configurarse para un mínimo de 1 hasta un máximo de 32 registros. Para este diseño se escoge un número de cuatro registros ya que se requieren tres registros para almacenar el valor de los dips, y un registro donde escribir el valor para la cuarta entrada a la puerta AND.

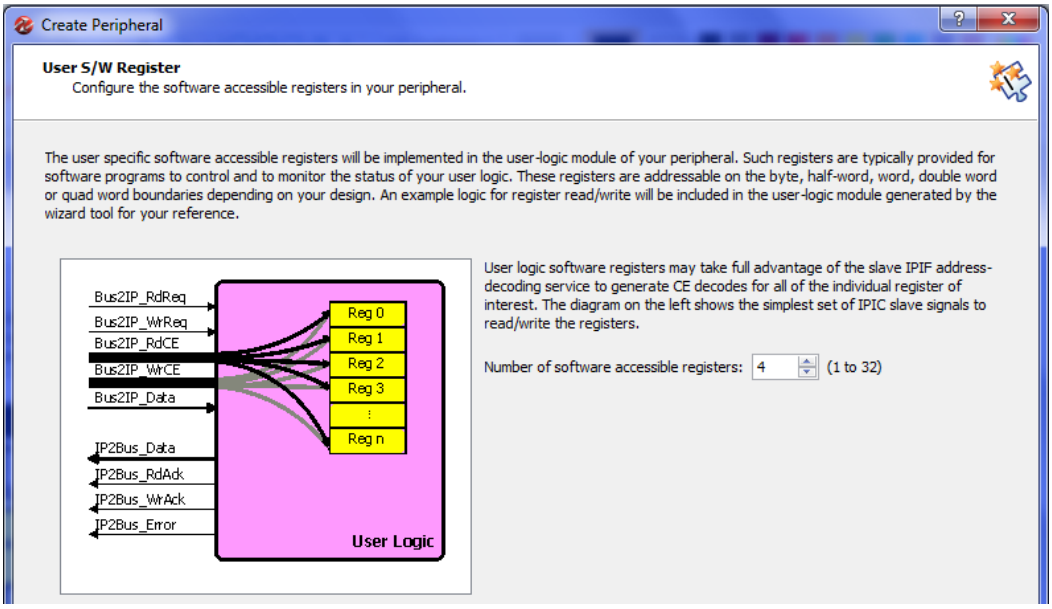


FIGURA 73. CREACIÓN DE UN PERIFÉRICO EN XPS - SELECCIÓN DEL NÚMERO DE REGISTROS

A continuación, a través de la herramienta IPIC (*AXI IP Interconnect*) se definen las señales que comunican al periférico con su módulo IPIF que lo conecta al bus anteriormente definido. Para este diseño se dejarán las señales que aparecen marcadas por defecto, pero el usuario puede seleccionarlas según la funcionalidad del periférico.

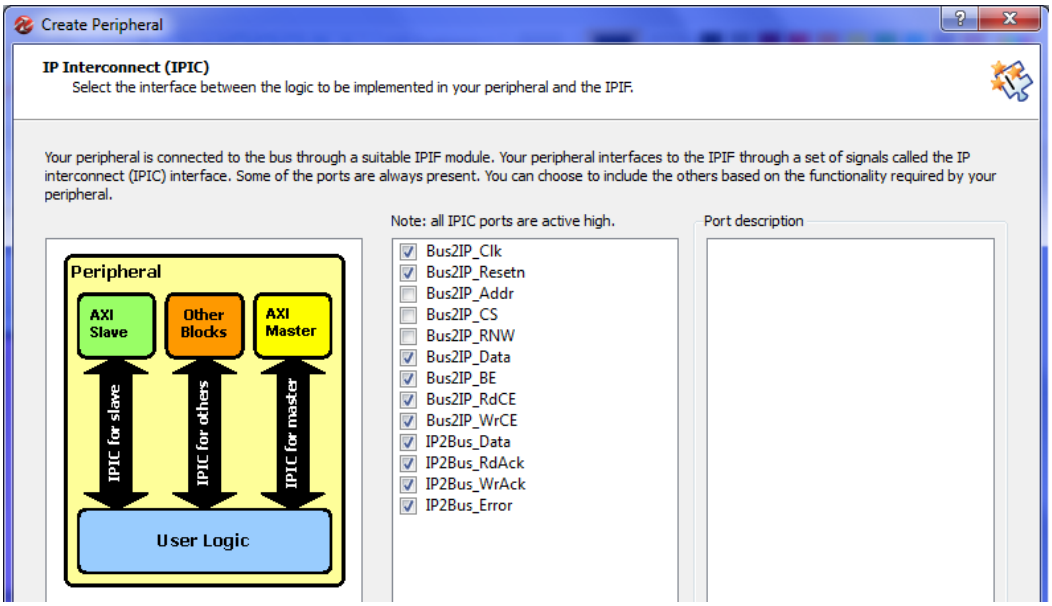


FIGURA 74. CREACIÓN DE UN PERIFÉRICO EN XPS - SEÑALES PARA LA COMUNICACIÓN CON EL MÓDULO IPIF

Estos puertos IPIC vienen definidos en la Tabla 3:

Bus → User Logic	
`Bus2IP_Clk`	IPIC clock, para sincronización de las señales.
`Bus2IP_Resetn`	IPIC reset a nivel bajo
`Bus2IP_CS`	IPIC chip select signal
`Bus2IP_RdCE`	IPIC read transaction, chip enable a nivel alto para lecturas
`Bus2IP_WrCE`	IPIC write transaction, chip enable a nivel alto para escrituras
`Bus2IP_Addr`	IPIC address
`Bus2IP_RNW`	IPIC read/write indication
`Bus2IP_BE`	IPIC byte enable, en una petición indica con bit '1' las líneas con datos válidos.
`Bus2IP_Data`	Read data from IPIC interface to IP
User Logic → Bus	
`IP2Bus_Data`	Read data from IP to IPIC interface
`IP2Bus_WrAck`	Write Data acknowledgment from IP to IPIC interface
`IP2Bus_RdAck`	Read Data acknowledgment from IP to IPIC interface
`IP2Bus_Error`	Error indication from IP to IPIC interface

TABLA 3. PUERTOS IPIC

Finalmente, aparecen un par de ventanas opcionales para la creación del periférico. La primera de ellas hace referencia a la posibilidad de generar archivos adicionales para simulación. Hay que decir que EDK dispone de la herramienta de simulación BFM (*Bus Functional Models*) mediante la

cual puede simularse el comportamiento del periférico, por lo que marcando la opción *Generate BFM simulation platform* se crean diversos ficheros que pueden emplearse en esta plataforma. Para este ejemplo no se va a utilizar pero puede marcarse la opción por si resulta útil en un futuro.

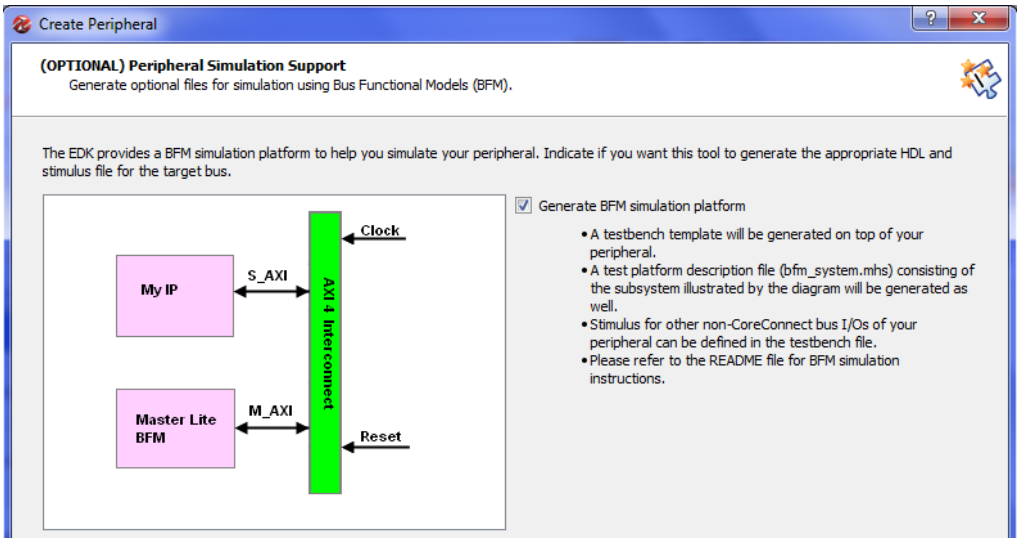


FIGURA 75. CREACIÓN DE UN PERIFÉRICO EN XPS - PLATAFORMA DE SIMULACIÓN (OPT)

La segunda ventana opcional ofrece al usuario la posibilidad de generar plantillas HDL sobre las que diseñar el periférico. Marcando la opción *Generate template driver files to help you implement software interface* se genera una carpeta que se identifica por el nombre del periférico seguido de su versión, situada en el directorio C:\...\project_1\project_1.srcs\sources_1\edk\module_1\pcores\and4_v1_00_a.

Esta carpeta contiene, por un lado, en and4_v1_00_a\data los ficheros generados por defecto propios de la interfaz EDK, *.mpd y *.pao, que se explicarán posteriormente, y por otro lado en and4_v1_00_a\hdl\vhdl los ficheros HDL, and4.vhd y user_logic.vhd. Será sobre el fichero user_logic donde se implementará la funcionalidad del periférico.

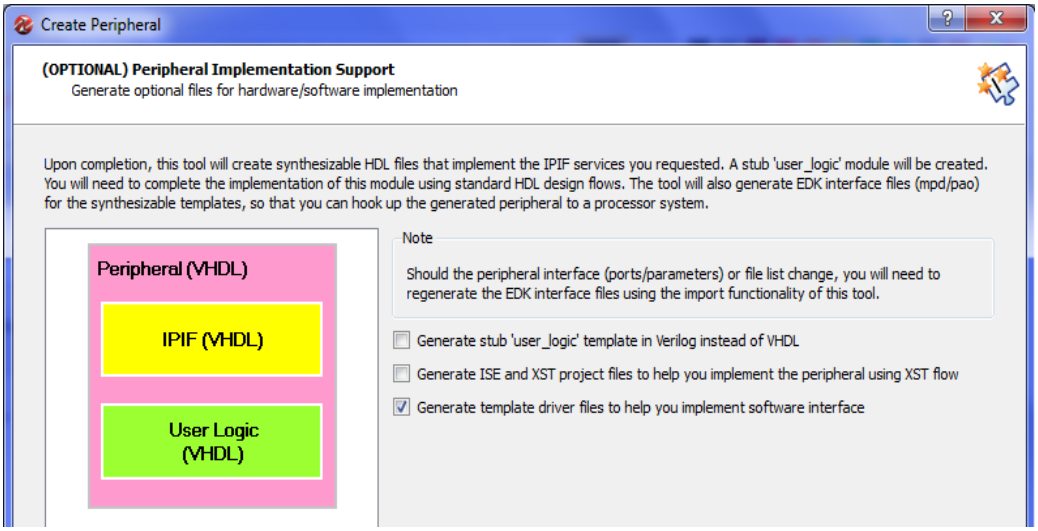


FIGURA 76. CREACIÓN DE UN PERIFÉRICO EN XPS - PLANTILLAS PARA LA INTERFAZ SOFTWARE (OPT)

Con esto finaliza el asistente para la creación de periféricos, mostrándose un resumen con los datos del periférico como nombre, versión y protocolo, los bloques de direcciones asignados y los ficheros generados (*.vhd,*.mpd,*.pao, plataforma BFM, drivers…). Esto puede verse de forma detallada en las siguientes imágenes.

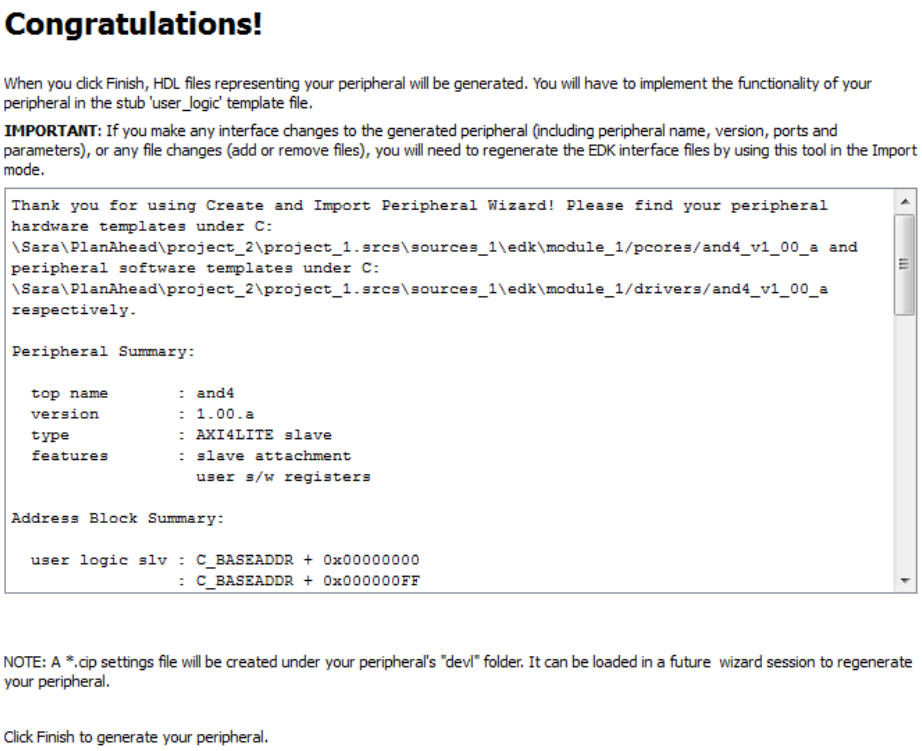


FIGURA 77. VENTANA DE FINALIZACIÓN DE LA HERRAMIENTA PARA LA CREACIÓN DE PERIFÉRICOS

En esta ventana de finalización de la herramienta *Create or Import Peripheral* aparece información referente al periférico y los ficheros generados, cuyo carácter se explicará en el siguiente apartado de *Funcionalidad en XPS*.

- Nombre, versión, interfaz, características y direcciones de memoria.

```
Peripheral Summary:

  top name       : and4
  version        : 1.00.a
  type           : AXI4LITE slave
  features       : slave attachment
                   user s/w registers

Address Block Summary:

  user logic slv : C_BASEADDR + 0x00000000
                 : C_BASEADDR + 0x000000FF
```

FIGURA 78. CARACTERÍSTICAS DEL PERIFÉRICO CREADO (I)

- Directorio de los ficheros HDL: entidad *and4.vhd* y lógica de usuario *user_logic.vhd*, y de los ficheros de datos del periférico: *and4_v2_1_0.mpd* y *and4_v2_1_0.pao.*

```
File Summary

  - HDL source -
  C:
\Sara\PlanAhead\project_2\project_1.srcs\sources_1\edk\module_1/pcores/and4_v1_00_a/hdl
  top entity      : vhdl/and4.vhd
  user logic      : vhdl/user_logic.vhd

  - XPS interface -
  C:
\Sara\PlanAhead\project_2\project_1.srcs\sources_1\edk\module_1/pcores/and4_v1_00_a/data
  mpd             : and4_v2_1_0.mpd
  pao             : and4_v2_1_0.pao
```

FIGURA 79. CARACTERÍSTICAS DEL PERIFÉRICO CREADO (II)

- Directorio y ficheros generados para la herramienta de simulación BFM, ya que se marcó su opción en el asistente para la creación de periféricos.

```
  - BFM simulation project -
  C:
\Sara\PlanAhead\project_2\project_1.srcs\sources_1\edk\module_1/pcores/and4_v1_00_a/devl/b
fmsim
  bfm sim help    : README.txt
  bfm platform    : bfm_system.mhs
  xps project     : bfm_system.xmp
  bfm testbench template: scripts/bfm_system_tb.v

  - Misc file -
  C:
\Sara\PlanAhead\project_2\project_1.srcs\sources_1\edk\module_1/pcores/and4_v1_00_a/devl
  help            : README.txt
  option          : ipwiz.opt
  log             : ipwiz.log
```

FIGURA 80. CARACTERÍSTICAS DEL PERIFÉRICO CREADO (III)

- Directorio de los drivers para el periférico y para la interfaz.

```
  - Driver source -
  C:
\Sara\PlanAhead\project_2\project_1.srcs\sources_1\edk\module_1/drivers/and4_v1_00_a/src
  makefile        : Makefile
  header          : and4.h
  source          : and4.c
  selftest        : and4_selftest.c

  - Driver interface -
  C:
\Sara\PlanAhead\project_2\project_1.srcs\sources_1\edk\module_1/drivers/and4_v1_00_a/data
  mdd             : and4_v2_1_0.mdd
  tcl             : and4_v2_1_0.tcl
```

FIGURA 81. CARACTERÍSTICAS DEL PERIFÉRICO CREADO (IV)

Funcionalidad en XPS

Llegados a este punto se ha creado y añadido el periférico a nuestro diseño, y a continuación se le debe dar la funcionalidad requerida. Para ello, como ya se ha comentado, deben modificarse los ficheros generados al finalizar la herramienta empleada en XPS para la creación del periférico. Estos ficheros se encuentran bajo la carpeta and4_v1_00_a con el nombre de *and4_v2_1_0.mpd*, *and4_v2_1_0.pao*, *and4.vhd* y *user_logic.vhd*. Primeramente, es conveniente especificar el carácter de estos ficheros.

- El fichero *.mpd (*Microprocessor Peripheral Definition*) define opciones y parámetros configurables con sus valores por defecto, la interfaz de bus seleccionada, y los puertos accesibles para el periférico, tanto señales del bus como datos de usuario y señales de control.
- El fichero *.pao (*Peripheral Analysis Order*) especifica los ficheros HDL necesarios para la síntesis, así como el orden a seguir durante la compilación.
- Los ficheros *.vhd se crean como plantillas de ayuda para implementar la lógica de usuario. Como se podrá ver, estas plantillas contienen indicaciones que facilitan al usuario la labor de localizar dónde debe incluir la lógica para una correcta interpretación. En *and4.vhd* pueden verse las librerías, la entidad y la arquitectura del periférico, y en *user_logic.vhd* es donde debe añadírsele funcionalidad.

A continuación se realizarán las modificaciones oportunas en estos ficheros para caracterizar al periférico creado con la funcionalidad que interesa.

El fichero *and4_v2_1_0.mpd* debe actualizarse para incluir los puertos del periférico. En este diseño se tienen tres puertos de entrada provenientes de tres dips, y un puerto de salida que se corresponderá con un led. Se indica que son de un único bit con " ", y sentido I/O para entrada/salida. Por tanto, debe añadirse en *## Ports*:

```
PORT DIP0 = "", DIR = I
PORT DIP1 = "", DIR = I
PORT DIP2 = "", DIR = I
PORT LED0 = "", DIR = O
```

El fichero *and4.vhd* también debe tener especificados los puertos. Éstos deben añadirse dentro de *Entity section*, en el apartado denominado *port*:

```
------------------------------------------------------------------------------
-- Entity section
------------------------------------------------------------------------------
(…)
port
  (
    -- ADD USER PORTS BELOW THIS LINE ------------------
    --USER ports added here
    DIP0 : in std_logic;
    DIP1 : in std_logic;
    DIP2 : in std_logic;
```

```
    LED0 : out std_logic;
    -- ADD USER PORTS ABOVE THIS LINE ------------------
```

Además, estos puertos deben mapearse dentro de *Architecture section*, en el apartado denominado *instantiate User Logic*:

```
------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------
(…)
-----------------------------------------
  -- instantiate User Logic
  -----------------------------------------
(…)
port map
    (
      -- MAP USER PORTS BELOW THIS LINE ------------------
      --USER ports mapped here
      DIP0 => DIP0,
      DIP1 => DIP1,
      DIP2 => DIP2,
      LED0 => LED0,
     -- MAP USER PORTS ABOVE THIS LINE ------------------
```

Finalmente, se considera oportuno mostrar el contenido completo del fichero *user_logic.vhd* una vez modificado para añadir la funcionalidad del periférico. De igual forma, se han especificado los puertos a emplear dentro de *Entity section*, en el apartado denominado *port*, y la parte correspondiente a la arquitectura se ha incluido en *Architecture section*, añadiéndose la señal *in4* para la cuarta entrada a la puerta and, y posteriormente el proceso correspondiente para dar la funcionalidad requerida. A continuación se muestra el fichero *user_logic.vhd* completo:

```
------------------------------------------------------------------------------
-- Entity section
------------------------------------------------------------------------------
-- Definition of Generics:
--   C_NUM_REG                   -- Number of software accessible registers
--   C_SLV_DWIDTH                -- Slave interface data bus width
--
-- Definition of Ports:
--   Bus2IP_Clk                  -- Bus to IP clock
--   Bus2IP_Resetn               -- Bus to IP reset
--   Bus2IP_Data                 -- Bus to IP data bus
--   Bus2IP_BE                   -- Bus to IP byte enables
--   Bus2IP_RdCE                 -- Bus to IP read chip enable
--   Bus2IP_WrCE                 -- Bus to IP write chip enable
--   IP2Bus_Data                 -- IP to Bus data bus
--   IP2Bus_RdAck                -- IP to Bus read transfer acknowledgement
--   IP2Bus_WrAck                -- IP to Bus write transfer acknowledgement
--   IP2Bus_Error                -- IP to Bus error response
------------------------------------------------------------------------------

entity user_logic is
  generic
  (
    -- ADD USER GENERICS BELOW THIS LINE ---------------
```

```
    --USER generics added here
    -- ADD USER GENERICS ABOVE THIS LINE ---------------

    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol parameters, do not add to or delete
    C_NUM_REG                      : integer              := 4;
    C_SLV_DWIDTH                   : integer              := 32
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );
  port
  (
    -- ADD USER PORTS BELOW THIS LINE ------------------
    --USER ports added here
      DIP0 : in  std_logic;
      DIP1 : in  std_logic;
      DIP2 : in  std_logic;
      LED0 : out std_logic;
    -- ADD USER PORTS ABOVE THIS LINE ------------------

    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol ports, do not add to or delete
    Bus2IP_Clk                  : in  std_logic;
    Bus2IP_Resetn               : in  std_logic;
    Bus2IP_Data                 : in  std_logic_vector(C_SLV_DWIDTH-1 downto 0);
    Bus2IP_BE                   : in  std_logic_vector(C_SLV_DWIDTH/8-1 downto 0);
    Bus2IP_RdCE                 : in  std_logic_vector(C_NUM_REG-1 downto 0);
    Bus2IP_WrCE                 : in  std_logic_vector(C_NUM_REG-1 downto 0);
    IP2Bus_Data                 : out std_logic_vector(C_SLV_DWIDTH-1 downto 0);
    IP2Bus_RdAck                : out std_logic;
    IP2Bus_WrAck                : out std_logic;
    IP2Bus_Error                : out std_logic
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );

  attribute MAX_FANOUT : string;
  attribute SIGIS : string;

  attribute SIGIS of Bus2IP_Clk    : signal is "CLK";
  attribute SIGIS of Bus2IP_Resetn : signal is "RST";

end entity user_logic;

------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture IMP of user_logic is

  --USER signal declarations added here, as needed for user logic
  signal in4 : std_logic;
  ------------------------------------------
  -- Signals for user logic slave model s/w accessible register example
  ------------------------------------------
  signal slv_reg0                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
  signal slv_reg1                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
  signal slv_reg2                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
  signal slv_reg3                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
  signal slv_reg_write_sel            : std_logic_vector(3 downto 0);
  signal slv_reg_read_sel             : std_logic_vector(3 downto 0);
  signal slv_ip2bus_data              : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
  signal slv_read_ack                 : std_logic;
  signal slv_write_ack                : std_logic;
```

```
begin

  --USER logic implementation added here
  process (DIP0, DIP1, DIP2) is
  begin

  slv_reg1(0) <= DIP0;
  slv_reg2(0) <= DIP1;
  slv_reg3(0) <= DIP2;
  slv_reg1(31 downto 1) <= (others => '0');
  slv_reg2(31 downto 1) <= (others => '0');
  slv_reg3(31 downto 1) <= (others => '0');

  in4 <= slv_reg0(0);
  LED0 <= ((DIP0 and DIP1) and DIP2) and in4;

  end process;
```

Después de estre proceso principal se encuentran las asociaciones entre las señales del bus y los registros definidos inicialmente como accesibles por software, además de procesos de lectura y escritura en estos registros. Estas operaciones en el bus se han implementado de forma automática en la plantilla generada, lo único que hay que detenerse a analizar en este punto es a qué registros se les puede dejar permiso de lectura o escritura por software de forma que no generen un conflicto con el periférico. Para este diseño, los registros del uno al tres se emplean para almacenar el valor de los tres dips definidos como entrada, de forma que el usuario pueda comprobar si el valor que está tomando es el correcto sin más que leer de estos registros. Por otro lado, el registro cero se emplea para dar valor a la cuarta entrada del periférico. Este valor es escrito por software, por lo tanto debe permitirse la escritura en este registro.

En definitiva, el registro cero puede ser escrito desde el proceso de escritura que se muestra, pero debe deshabilitarse la escritura para los registros uno, dos y tres, para lo cual basta con comentar la sentencia correspondiente. En cuanto al proceso de lectura, éste puede permanecer habilitado para los cuatro registros.

```
  slv_reg_write_sel <= Bus2IP_WrCE(3 downto 0);
  slv_reg_read_sel  <= Bus2IP_RdCE(3 downto 0);
  slv_write_ack     <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2) or Bus2IP_WrCE(3);
  slv_read_ack      <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2) or Bus2IP_RdCE(3);

  -- implement slave model software accessible register(s)
  SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
  begin
```

Proceso de escritura

```
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
      if Bus2IP_Resetn = '0' then
        slv_reg0 <= (others => '0');
        --slv_reg1 <= (others => '0');
        --slv_reg2 <= (others => '0');
        --slv_reg3 <= (others => '0');
      else
        case slv_reg_write_sel is
          when "1000" =>
            for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
```

Deshabilitar el reset en los registros que contienen el valor de los dips

```
              if ( Bus2IP_BE(byte_index) = '1' ) then
                slv_reg0(byte_index*8+7  downto  byte_index*8)  <=  Bus2IP_Data(byte_index*8+7
downto byte_index*8);
              end if;
            end loop;
          when "0100" =>
            for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                --slv_reg1(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7
downto byte_index*8);
              end if;
            end loop;
          when "0010" =>
            for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                --slv_reg2(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7
downto byte_index*8);
              end if;
            end loop;
          when "0001" =>
            for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                --slv_reg3(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7
downto byte_index*8);
              end if;
            end loop;
          when others => null;
        end case;
      end if;
    end if;

  end process SLAVE_REG_WRITE_PROC;

  -- implement slave model software accessible register(s) read mux
  SLAVE_REG_READ_PROC : process( slv_reg_read_sel, slv_reg0, slv_reg1, slv_reg2, slv_reg3 )
is
  begin

    case slv_reg_read_sel is
      when "1000" => slv_ip2bus_data <= slv_reg0;
      when "0100" => slv_ip2bus_data <= slv_reg1;
      when "0010" => slv_ip2bus_data <= slv_reg2;
      when "0001" => slv_ip2bus_data <= slv_reg3;
      when others => slv_ip2bus_data <= (others => '0');
    end case;

  end process SLAVE_REG_READ_PROC;

  ----------------------------------------
  -- Example code to drive IP to Bus signals
  ----------------------------------------
  IP2Bus_Data  <= slv_ip2bus_data when slv_read_ack = '1' else
                  (others => '0');

  IP2Bus_WrAck <= slv_write_ack;
  IP2Bus_RdAck <= slv_read_ack;
  IP2Bus_Error <= '0';

end IMP;
```

Habilitar escritura en el registro 0 y deshabilitar escritura en los registros 1, 2 y 3 pertenecientes a los switches

Proceso de lectura

Permitir la lectura en todos los registros accesibles

Con esto queda caracterizado al completo el periférico diseñado.

A continuación, es conveniente seleccionar *Rescar User Repositories* en la pestaña *Project* para guardar los cambios realizados en los ficheros anteriores. Una vez realizado, puede localizarse el periférico en *IP Catalog>Project Local Pcores>USER>AND4* y añadirlo al diseño hardware. Al finalizar la instanciación aparece en la consola el mensaje indicando que se ha conectado correctamente al bus, además puede comprobarse en la pestaña *Bus Interfaces* que el periférico está conectado a axi_interconnect_1, y en *Ports>and4_0* seleccionar *DIP0,DIP1,DIP2* y *LED0* como *External Ports*. Finalmente, en la pestaña *Addresses,* puede verse la dirección base asignada al periférico.

Tras estas comprobaciones, puede terminarse seleccionando *Design Rule Check* en la pestaña Project, y si todo está correcto puede cerrarse la herramienta XPS.

Retornando a PlanAhead, como se ha explicado en capítulos previos, sobre *module_1.xmp* (que contiene la configuración realizada en XPS para este periférico) debe seleccionarse *Create Top HDL* para generar el fichero *module_1_stub.vhd*.

El último paso es modificar el fichero de Constraints que se tenía anteriormente, para adaptarlo a los nuevos puertos externos. En el diseño anterior se tenía un puerto vector de 8 bits para los switches y un puerto vector de 5 bits para los botones (ver Figura 46), por lo que el fichero *.ucf quedaba de la forma de la Figura 26, pero para este nuevo diseño se tienen 4 puertos independientes, uno por cada bit de los switches (se usarán los 3 LSB) y uno para el led, por lo que deben definirse de la siguiente forma:

```
############################
# On-board Slide Switches  #
############################
net and4_0_DIP0_pin    LOC = F22 | IOSTANDARD = LVCMOS33;  # SW0
net and4_0_DIP1_pin    LOC = G22 | IOSTANDARD = LVCMOS33;  # SW1
net and4_0_DIP2_pin    LOC = H22 | IOSTANDARD = LVCMOS33;  # SW2

############################
# On-board LED             #
############################
NET and4_0_LED0_pin    LOC = T22 | IOSTANDARD = VCMOS33;   # LD0
```

FIGURA 82. MODIFICACIÓN DEL FICHERO “CONSTRAINTS” PARA LOS NUEVOS PUERTOS

A continuación, sería el momento de realizar la síntesis, implementación y generación del bitstream, y exportar el diseño a SDK, donde se incluirá la parte software para la comunicación entre el procesador y el periférico.

Aplicación en SDK

Una vez abierta la herramienta SDK, si se está empleando un proyecto nuevo en el explorador de proyecto se encontrará únicamente el módulo hardware exportado y sería necesario repetir los pasos indicados en capítulos anteriores para la creación de una plataforma de diseño software (Board Support Package) sobre ese módulo hardware. Si por el contrario el proyecto ya contiene una plataforma hardware, puede continuarse empleando la misma.

Para comenzar, es necesario añadir una aplicación a dicha plataforma. Para ello, como ya se comentó, debe hacerse desde *File>New>Application Project* e indicar nombre, BSP existente y escoger *Empty Application* para comenzar desde un fichero vacío. Esta aplicación aparecerá en el explorador de proyecto, y con la opción *Import* se buscará el fichero *.c que contenga el programa software, en este caso se le ha dado el nombre de puerta_and.c, el cual contiene el siguiente código de programa.

```
#include "xparameters.h"
#include "xutil.h"
#include "and4.h"

//====================================================

int main (void)
{
        int i;
        int dip0_value=0, dip1_value=0, dip2_value=0, in4;
        int value;

        value = 1;
        xil_printf("Start of the program \r\n");
   while (1)
   {
        //Write register 0 which contains the value of the 4th in
        AND4_mWriteSlaveReg0(XPAR_AND4_0_BASEADDR, AND4_SLV_REG0_OFFSET,value);

        //Read register 1 which contains the value of the dip 0
        dip0_value = AND4_mReadSlaveReg1(XPAR_AND4_0_BASEADDR, AND4_SLV_REG0_OFFSET);
        //Read register 2 which contains the value of the dip 1
        dip1_value = AND4_mReadSlaveReg2(XPAR_AND4_0_BASEADDR, AND4_SLV_REG0_OFFSET);
        //Read register 3 which contains the value of the dip 2
        dip2_value = AND4_mReadSlaveReg3(XPAR_AND4_0_BASEADDR, AND4_SLV_REG0_OFFSET);
        //Read register 0 which contains the value of the 4th in
        in4 = AND4_mReadSlaveReg0(XPAR_AND4_0_BASEADDR, AND4_SLV_REG0_OFFSET);

        xil_printf("DIPS Switches IN 1-3 Value %x %x %x\r\n", dip0_value, dip1_value,
dip2_value);
        xil_printf("LED0 = and( %x %x %x %x )\r\n", dip0_value, dip1_value, dip2_value,
in4);

        for (i=0; i<9999999; i++);
   }
}
```

Puede verse que se mantienen las cabeceras *xparameters.h* y *xutil.h* como se tenían en el anterior diseño con botones y switches en SDK, sin embargo la cabecera *xgpio.h* ya no se encuentra puesto

que no se tienen periféricos GPIO, sino que se utiliza la cabecera generada para el periférico creado, *and4.h*, cuyo contenido se comenta en la Tabla 4.

En cuanto al programa se observa que realiza en primer lugar una escritura en el registro cero, que es el registro que tiene asignado la cuarta entrada a la puerta AND, y posteriormente lee el valor de los cuatro registros disponibles para conocer los cuatro valores de entrada a la puerta, incluido el que se ha escrito en la sentencia anterior vía software. Para terminar, se imprime por consola el valor de las entradas provenientes de los tres dips, y la operación que se va a realizar con las cuatro entradas correspondientes, siendo en LED0 donde se podrá ver el resultado de la operación, encontrándose el led encendido en caso de respuesta verdadera o apagado en caso de respuesta falsa.

CABECERA AND4.H - COMUNICACIÓN CON EL PERIFÉRICO	
AND4_mWriteReg(BaseAddress, RegOffset, Data)	Escribe Data en la dirección BaseAddress+RegOffset
AND4_mReadReg(BaseAddress, RegOffset)	Lee el dato de la dirección BaseAddress+RegOffset
AND4_mWriteSlaveReg0(BaseAddress, RegOffset, Value) AND4_mWriteSlaveReg1(BaseAddress, RegOffset, Value) AND4_mWriteSlaveReg2(BaseAddress, RegOffset, Value) AND4_mWriteSlaveReg3(BaseAddress, RegOffset, Value)	Función de escritura ya particularizada para los registros definidos como accesibles por software. No retorna nada.
AND4_mReadSlaveReg0(BaseAddress, RegOffset) AND4_mReadSlaveReg1(BaseAddress, RegOffset) AND4_mReadSlaveReg2(BaseAddress, RegOffset) AND4_mReadSlaveReg3(BaseAddress, RegOffset)	Función de lectura ya particularizada para los registros definidos como accesibles por software. Retorna el dato leído.
AND4_SelfTest(void * baseaddr_p)	Self-test en el periférico. Retorna XST_SUCCESS o XST_FAILURE.

TABLA 4. DEFINICIONES CONTENIDAS EN LA CABECERA AND4.H

A continuación se deben añadir ciertas sentencias en el driver del periférico. Para ello, desde el directorio C:\...\project_1.srcs\sources_1\edk\module_1\drivers\and4_v1_00_a\src, hay que abrir *and4_selftest.c* y comentar **#include** "xio.h", ya que este fichero contiene la interfaz para los componentes XIo, los cuales contienen funciones de E/S para los procesadores y no interesa acceder a registros del procesador sino del esclavo. También hay que escribir en *Constant Definitions* **#define** AND4_USER_NUM_REG 4, que indica el número de registros donde el usuario puede realizar operaciones de lectura o escritura, quedando el fichero de esta forma:

```
#include "and4.h"
#include "stdio.h"
//#include "xio.h"
#include "xparameters.h"

/************************** Constant Definitions ***************************/
```

```
#define READ_WRITE_MUL_FACTOR 0x10
#define AND4_USER_NUM_REG 4
```

Finalmente, antes de descargar el diseño en la tarjeta, en *Xilinx Tool Repositories* debe especificarse el directorio local del módulo hardware C:\...\project_1.srcs\sources_1\edk\module_1 para evitar conflictos de no reconocimiento desde la aplicación. También hay que incorporar el driver al periférico en el BSP del diseño abriendo *Standalone_bsp_0>Board support package settings>drivers* identificando el nombre del periférico and4_0 y modificando el driver definido por defecto de *generic* a *and4*.

Con estos cambios realizados, puede compilarse la aplicación desde el menú de la aplicación en *Clean Project* y observar si aparece algún error en la consola que se deba revisar.

Llegados a este punto, la interfaz de usuario tiene el aspecto de la Figura 83.

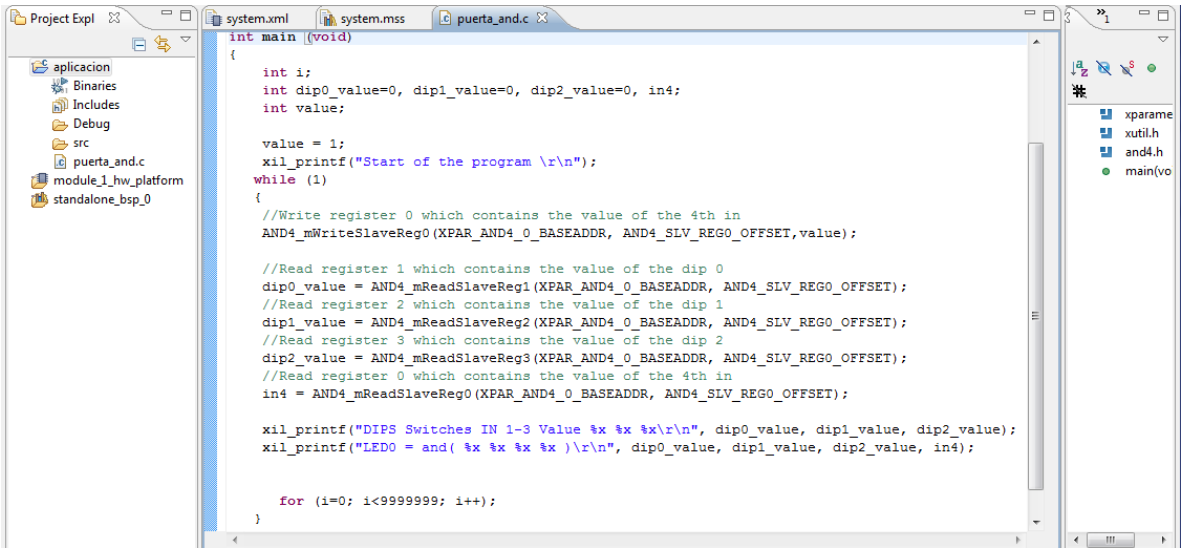


FIGURA 83. INTERFAZ DE USUARIO EN SDK

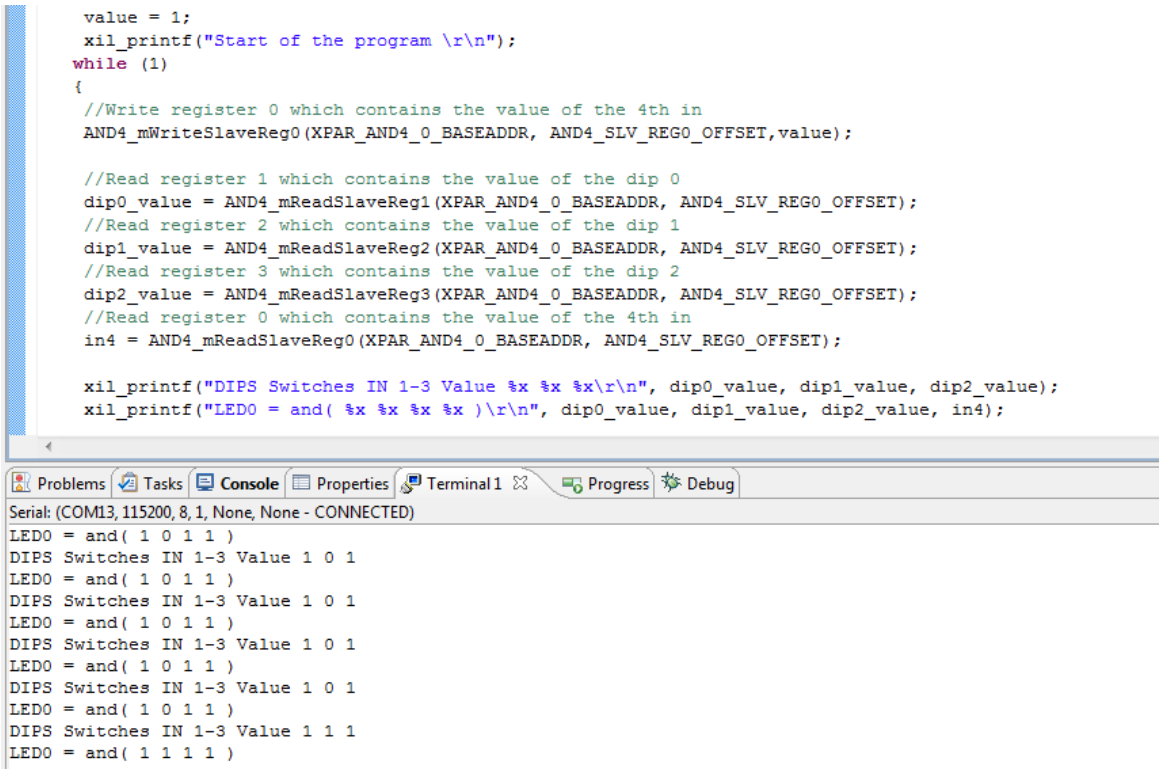
Prueba en hardware

Para realizar la prueba en hardware debe conectarse la tarjeta apropiadamente al ordenador [8], programar la FPGA desde *Xilinx Tools>Program FPGA* cargando el fichero *.elf correspondiente (se enciende el led LD12 azul), localizar el puerto COM que se está empleando, conectar la tarjeta al terminal, y lanzar la aplicación desde el menú de aplicación en *Run As>Launch on hardware*.

Gracias a las sentencias introducidas en el código de programa para impresión de resultados por el terminal, se puede observar si lo que se imprime en la consola corresponde con las entradas que se tienen en los primeros tres dips de la tarjeta, DIP0, DIP1 y DIP2, y si la cuarta entrada tiene el valor 1 que se le ha asignado en el código del programa. Además, en la tarjeta se puede jugar con

diferentes combinaciones de los dips y observar si el primer led, LED0, se enciende únicamente cuando los tres dips toman el valor 1 (ya que la cuarta entrada se ha fijado a 1).

La captura siguiente muestra uno de los instantes de ejecución en el cual se tienen los dips DIP0 y DIP2 activos a 1, y el dip DIP1 se encuentra a 0, por lo que la operación and realizada resulta en valor cero y el led se mantiene apagado. En el momento en que DIP1 se activa también a 1 la operación and se realiza con las cuatro entradas activas y el led se enciende.



```
 value = 1;
 xil_printf("Start of the program \r\n");
while (1)
{
 //Write register 0 which contains the value of the 4th in
 AND4_mWriteSlaveReg0(XPAR_AND4_0_BASEADDR, AND4_SLV_REG0_OFFSET,value);

 //Read register 1 which contains the value of the dip 0
 dip0_value = AND4_mReadSlaveReg1(XPAR_AND4_0_BASEADDR, AND4_SLV_REG0_OFFSET);
 //Read register 2 which contains the value of the dip 1
 dip1_value = AND4_mReadSlaveReg2(XPAR_AND4_0_BASEADDR, AND4_SLV_REG0_OFFSET);
 //Read register 3 which contains the value of the dip 2
 dip2_value = AND4_mReadSlaveReg3(XPAR_AND4_0_BASEADDR, AND4_SLV_REG0_OFFSET);
 //Read register 0 which contains the value of the 4th in
 in4 = AND4_mReadSlaveReg0(XPAR_AND4_0_BASEADDR, AND4_SLV_REG0_OFFSET);

 xil_printf("DIPS Switches IN 1-3 Value %x %x %x\r\n", dip0_value, dip1_value, dip2_value);
 xil_printf("LED0 = and( %x %x %x %x )\r\n", dip0_value, dip1_value, dip2_value, in4);
```

Problems | Tasks | Console | Properties | Terminal 1 | Progress | Debug

Serial: (COM13, 115200, 8, 1, None, None - CONNECTED)

```
LED0 = and( 1 0 1 1 )
DIPS Switches IN 1-3 Value 1 0 1
LED0 = and( 1 0 1 1 )
DIPS Switches IN 1-3 Value 1 0 1
LED0 = and( 1 0 1 1 )
DIPS Switches IN 1-3 Value 1 0 1
LED0 = and( 1 0 1 1 )
DIPS Switches IN 1-3 Value 1 0 1
LED0 = and( 1 0 1 1 )
DIPS Switches IN 1-3 Value 1 1 1
LED0 = and( 1 1 1 1 )
```

FIGURA 84. PRUEBA EN HARDWARE DE LA OPERACIÓN AND

Para terminar la ejecución de forma controlada se selecciona el botón *Terminate* en la pestaña *Console*, se desconecta del terminal y se apaga la tarjeta.

Como se ha podido ver, XPS presenta una herramienta para creación e importación de periféricos que resulta muy útil a la hora de diseñar cierto periférico, dando la posibilidad además de generar plantillas con comentarios que resultan muy útiles para hacer de guía al usuario.

No obstante, a continuación se tratará el tema de creación de periféricos desde un nuevo punto de vista, empleando la herramienta Vivado HLS de Xilinx para trabajar en un nivel de abstracción superior utilizando lenguajes de alto nivel.

3.2. VIVADO HIGH-LEVEL SYNTHESIS

3.2.1. **Introducción al diseño basado en FPGA**

¿Qué es Vivado HLS?

Vivado HLS es una herramienta de síntesis de alto nivel para FPGAs de Xilinx, que deja a un lado los lenguajes de descripción de hardware (HDL) para evolucionar a lenguajes de alto nivel (HLL) [28].

Esta metodología de diseño ofrece mejoras en términos de productividad y de rendimiento ya que, por un lado, permite a los diseñadores hardware beneficiarse de las posibilidades que ofrece trabajar a un alto nivel de abstracción y, por otro lado, facilita a los desarrolladores software optimizar sus algoritmos en cuanto a rendimiento, coste y potencia.

Su principal ventaja es que permite trabajar a nivel de lenguaje C, transformando descripciones C/C++ en implementaciones RTL para sintetizarlas en una FPGA. Esto supone para los desarrolladores una mayor rapidez de diseño ya que evita entrar en detalles de implementación [21]. Además, el proceso de síntesis se controla mediante directivas que permiten crear diferentes implementaciones hardware del código fuente, de forma que resulte más fácil encontrar la implementación más óptima del diseño sin necesidad de modificar el código fuente original.

No obstante, la herramienta Vivado HLS lleva consigo algunas limitaciones que pueden consultarse a lo largo de la guía de usuario [28]. Entre ellas se encuentra que permite hacer uso de los lenguajes C, C++ y SystemC, pero no para todos ellos están soportadas todas las funcionalidades, entre ellas emplear precisión arbitraria de bits en lenguaje C sólo es posible para datos enteros, la cabecera que soporta datos float se encuentra únicamente para lenguaje C++, además trabajando con datos de precisión arbitraria no es posible emplear el modo de depuración. Otra de las limitaciones es que permite exportar el diseño RTL en los estándares VHDL, Verilog o SystemC, pero no se conoce ninguna herramienta de Xilinx que pueda sintetizar SystemC en un fichero bitstream una vez exportado.

Al margen de las limitaciones propias de la herramienta se encuentran también limitaciones dadas por el uso del lenguaje C. Trabajando con la herramienta EDK se empleaba lenguaje de descripción hardware VHDL para la descripción de los periféricos creados por el usuario, sin embargo con esta herramienta se va a diseñar el periférico en lenguaje software como es el lenguaje C. Esto supone una ventaja en cuanto a que si no se está familiarizado con el lenguaje VHDL resulta menos laborioso programar en C, pero C es un lenguaje secuencial y el hardware tiene naturaleza concurrente.

Además, en los lenguajes de descripción hardware se tiene un mayor control de lo que realmente sucede en el diseño, lo que permite tener diseños más rápidos y optimizados que los que empleen un lenguaje software, aunque conlleve un tiempo de desarrollo mayor.

Otro hecho importante es que con VHDL la concurrencia viene dada por los procesos (*process*) que se ejecutan de forma concurrente entre ellos y contienen sentencias secuenciales en su interior. Vivado HLS está diseñado para aportar concurrencia a una descripción C si lo considera oportuno mediante asignación de recursos dedicados a cada operación del código, e incluye directivas de optimización que ayudan en la segmentación de tareas, pero esto supone un mayor consumo de recursos hardware y en ciertos diseños con muchas operaciones pueden no existir recursos suficientes.

A continuación se va a hacer un estudio previo de la herramienta para conocer cómo trabajar con ella y entender la forma en que se realizan las transformaciones a descripciones RTL, finalizando con un ejemplo de aplicación de directivas en un código para multiplicación de matrices. Finalmente se va a diseñar en lenguaje C un periférico que consiste en un filtro FIR paso bajo de orden 20 empleando el método de la ventana y utilizando una ventana tipo Hamming, donde los coeficientes y las muestras de entrada que se apliquen a este filtro serán obtenidos desde Matlab. Una vez se haya hecho el diseño en Vivado HLS y encontrado la solución más óptima se exportará a Vivado para continuar con el diseño y posteriormente a SDK para añadir la aplicación software que comunique al procesador con el filtro (ver Figura 6).

Interfaz gráfica de usuario (GUI)

Como punto inicial, antes de profundizar en cómo funciona la herramienta en términos de operaciones, ficheros y diseño, conviene mostrar cómo es la interfaz gráfica de usuario Vivado HLS, en qué regiones se divide y la función de cada una de ellas, de forma que el usuario tenga una primera visión del programa y de las opciones que ofrece.

Para ello, se indica en la Figura 85, las regiones en que está dividido.

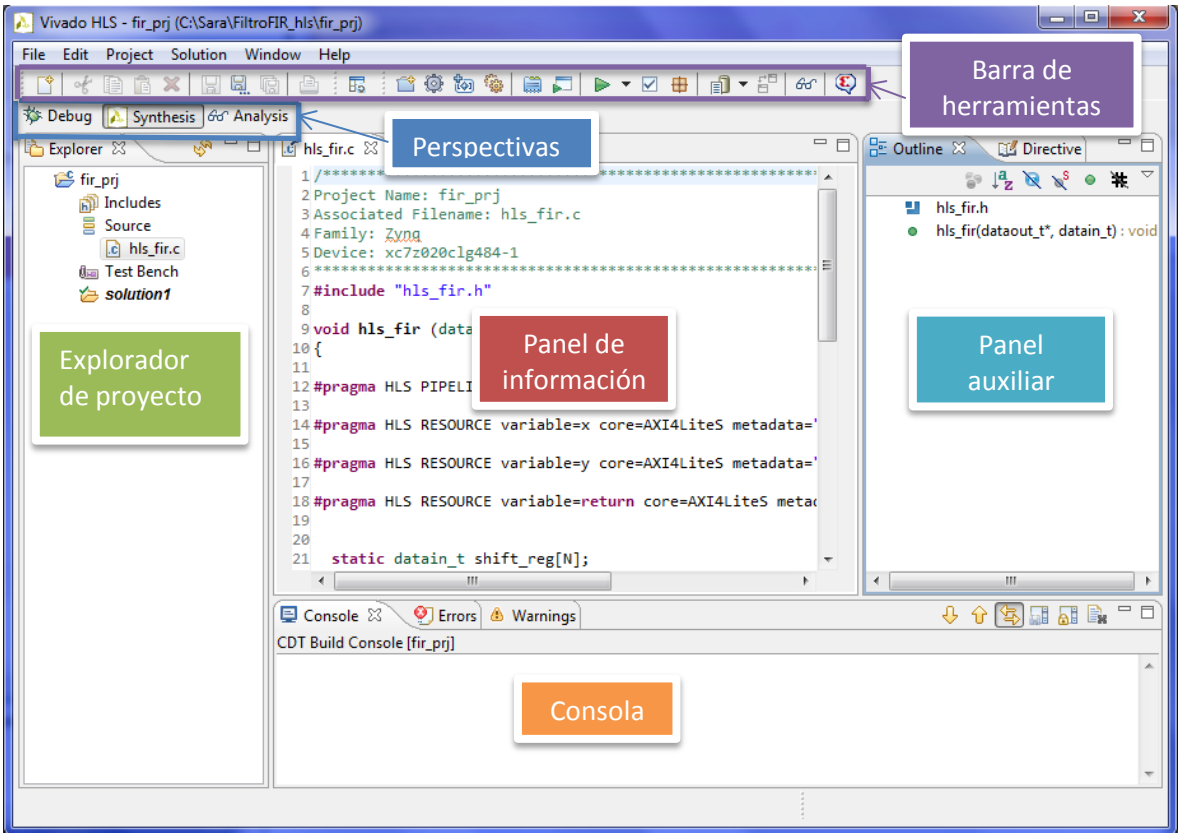


FIGURA 85. INTERFAZ GRÁFICA DE USUARIO (GUI) EN VIVADO HLS

En el panel Explorador de proyecto se muestra la jerarquía del proyecto sobre el que se esté trabajando, los ficheros que contiene y en qué directorio se encuentran. Según se van sucediendo las opciones de simulación, síntesis, verificación y empaquetado IP, se van creando subdirectorios en la solución sobre la que se estén llevando a cabo, denominados *csim*, *syn*, *sim* e *impl*, respectivamente.

El panel de información muestra el contenido de los ficheros que se abren desde el explorador de proyecto, así como los informes generados tras la síntesis y la verificación RTL.

El panel auxiliar muestra en la pestaña *Outline* los ficheros asociados al fichero que se muestra en el panel de información, y en la pestaña *Directive* las variables del código junto con las directivas que tiene aplicadas.

La consola muestra mensajes informativos, warning y errores que se generan cuando el programa está corriendo.

Los botones de la barra de herramientas sirven para llevar a cabo operaciones comunes. Cabe destacar la siguiente fila de botones  en la que, de izquierda a derecha, pueden identificarse los botones de *Project Settings*, *New Solution*, *Solution*

Settings, *Index C Source*, *Run C Simulation*, *Run C Synthesis*, *Run C/RTL Cosimulation*, *Export RTL*, *Open Report* y *Compare Reports*.

Finalmente las perspectivas que ofrece Vivado HLS para el análisis del diseño son:

- Síntesis: Es la perspectiva principal y se muestra por defecto al abrir un proyecto. En ella pueden realizarse las operaciones principales desde la barra de herramientas.

- Depuración: Se puede acceder a ella tras la compilación y permite ejecutar el código por pasos observando el contenido de las variables del diseño.

- Análisis: Permite tener una mejor visión de los resultados obtenidos tras la síntesis. La ventana superior izquierda *Module Hierarchy* muestra la jerarquía del diseño y los recursos que se consumen, así como la latencia y el intervalo. La ventana inferior izquierda contiene la pestaña *Performance Profile*, que muestra además la latencia por iteración y el número de iteraciones, y la pestaña *Resource Profile* donde se ve la cantidad recursos consumidos por cada tipo de operación. Y finalmente, la ventana derecha muestra los ciclos de operación para cada sentencia de código (pestaña *Performance*) o para cada recurso (pestaña *Resource*).

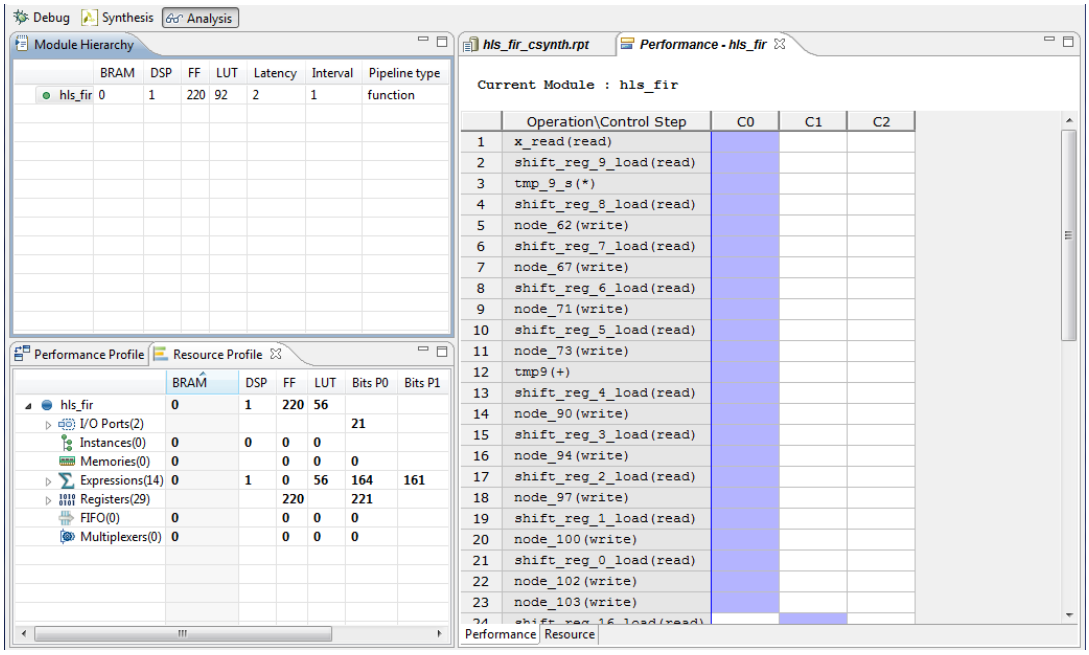


FIGURA 86. VENTANA DE ANÁLISIS EN VIVADO HLS

Entradas y salidas

Este software está diseñado para aceptar como entrada principal una función escrita en lenguaje C, C++ o System C junto con las restricciones propias de la tarjeta seleccionada y las directivas incorporadas por el usuario, y además puede incluirse un *test bench* (y ficheros asociados) empleado para simular dicha función principal como paso previo a la síntesis. La función principal del *test bench* es la función `main()` la cual llama a la función principal C a sintetizar entregándole una entrada de prueba y recogiendo una salida.

Es útil que este *test bench* tenga la propiedad de ser self-checking, de forma que compruebe si los resultados devueltos por el algoritmo a implementar son correctos. Esto puede hacerse almacenando en un fichero la salida de la función C que se va a sintetizar y compararla con un fichero que contenga los resultados correctos (y que se encuentre en el proyecto como fichero asociado al *test bench*), indicando al usuario con `return 0` si éstos coinciden. Así, puede comprobarse la validez del algoritmo ante una modificación del código fuente de forma más rápida y previa a la síntesis.

La importancia de un buen *test bench* además puede verse en que no sólo se emplea para validar la funcionalidad de la función principal C en simulación sino que Vivado HLS también reutiliza este mismo *test bench* para verificar el diseño RTL, no es necesario un nuevo *test bench*. Esta verificación se lleva a cabo mediante C/RTL Cosimulation y puede hacerse para los estándares VHDL, Verilog o SystemC.

Una vez llevada a cabo la síntesis y verificación, se dispone de la implementación RTL de la función C de entrada en formato HDL (Hardware Description Language), la cual está preparada para exportarse como bloque IP a otra herramienta de diseño de Xilinx para ser sintetizada en un fichero *bitstream* necesario para programar la FPGA. En este caso será la herramienta Vivado Design Suite la empleada para crear este fichero *bitstream* a partir de los ficheros Verilog o VHDL (System C no puede ser sintetizado en un fichero *bitstream* por ninguna herramienta de Xilinx y no se incluye en el bloque IP).

Metodología de trabajo

Para entender de qué forma Vivado HLS lleva a cabo este proceso de transformación de una descripción C en un diseño RTL conviene saber que el término High-Level Synthesis se basa en dos procesos: planificación (*scheduling*) y enlace (*binding*).

- Mediante el proceso de planificación, a partir de la frecuencia de reloj, de las características temporales del dispositivo empleado y de directivas de optimización que haya incorporado el usuario a su diseño, se decide en qué ciclo de reloj se realiza cada operación. El tiempo que tarda una operación en completarse es función de la FPGA que

se esté utilizando, de forma que, con un periodo de reloj mayor o una FPGA más rápida, se planificará mayor número de operaciones dentro de un ciclo de reloj.

- Mediante el proceso de enlace se determinan los recursos hardware que implementarán cada operación de las que componen el proceso de planificación. No sólo decide los recursos necesarios sino que, conociendo cierta información del dispositivo empleado, busca la solución más óptima entre mayor rendimiento y menor área ocupada.

La forma que tiene HLS de crear la implementación más óptima está basada en su comportamiento fijado por defecto, en las restricciones, y en las directivas que el usuario haya especificado.

Este comportamiento por defecto se caracteriza por:

- Implementar los argumentos de la función principal como puertos del diseño RTL final.
- Trasladar las funciones C a bloques RTL manteniendo la jerarquía, de forma que, si el código C tiene una jerarquía de sub-funciones, el diseño RTL presentará una jerarquía de módulos o entidades.
- Los bucles que contenga el código se mantienen *"enrollados"* (roll), lo que quiere decir, que en el momento de síntesis el proceso de enlace determinará la lógica necesaria para realizar una iteración del bucle y esa lógica será reutilizada para el resto de iteraciones de forma secuencial.
- Los arrays se sintetizarán como bloques RAM en el diseño RTL. Además, si el array se encuentra como argumento de la función principal, crea puertos para acceder a dicho bloque RAM: puerto de datos, puerto de direcciones y señales chip-enable y write-enable.

No obstante, este comportamiento puede modificarse por el usuario incluyendo directivas de optimización al diseño para así conseguir la implementación deseada.

Como ya se ha dicho, para un mismo diseño pueden realizarse distintas implementaciones o soluciones en función de las directivas que se fijen y comparar los resultados de síntesis de cada una de ellas para encontrar la implementación óptima. Para ello, tras completarse el proceso de síntesis se devuelve un informe de resultados. Este informe incluye:

- Información general del diseño: fecha, hora, versión, nombre, solución sobre la cual se realiza la síntesis, e identificadores de la tarjeta.

- Estimaciones de rendimiento: reloj, latencia e intervalo de iniciación (de la función o de un bucle). Tanto la latencia como el intervalo de iniciación (II) pueden definirse para una función o para un bucle del código. Cabe distinguir, por tanto, estos conceptos.

  La latencia de la función se define como el número de ciclos de reloj necesarios para proporcionar unos datos de salida. Por otro lado, la latencia de iteración de un bucle es el

número de ciclos necesarios para completar una iteración de dicho bucle, y la latencia del bucle es el número de ciclos necesarios para completar todas las iteraciones.

El intervalo de iniciación de una función se define como el número de ciclos de reloj que deben sucederse antes de que la función pueda aceptar una nueva entrada de datos que procesar. Por otro lado, el intervalo de iniciación de un bucle es el número de ciclos que deben sucederse antes de que la siguiente iteración del bucle comience a procesar los datos.

- Estimaciones de área: recursos hardware necesarios para implementar el diseño. Pueden verse de forma resumida en una tabla donde se recoge el número de bloques RAM, de DSP48s, de FFs y de LUTs que emplea el diseño frente a los disponibles para la tarjeta seleccionada, y de manera detallada posteriormente para cada operación, agrupándose en *Instances* (recursos empleados por sub-bloques del diseño), *Memory*, *FIFO*, *Shift Registers*, *Expressions* (multiplicadores, sumadores y comparadores), *Multiplexors* y *Registers*.

- Interfaz: muestra el conjunto de puertos que contendrá el diseño RTL y que han sido definidos a partir de la descripción C, junto con su dirección, tamaño en bits, protocolo que emplean y tipo de dato.

De esta forma resulta más sencillo para el desarrollador comprobar los efectos que tienen las directivas incluidas en el diseño y si se cumple los requisitos que espera obtener en vistas a la implementación RTL.

Librerías

Vivado HLS dispone de distintas librerías para el estándar de lenguaje C las cuales han sido optimizadas para asegurar un diseño de alto rendimiento que haga un uso óptimo de los recursos.

Entre estas librerías cabe destacar la librería que incorpora Vivado HLS para manejar tipos de datos de precisión arbitraria. Esta librería surge como necesidad de asegurar un diseño con alta calidad de resultados basándose en el hecho de que emplear los tipos de datos comunes en C, con un ancho de bit fijado a 8, 16, 32 o 64 bits, supone un hardware ineficiente. Esta ineficiencia viene dada porque si el diseño a implementar opera con datos de, por ejemplo, 12-bit, 22-bit o 34-bits de ancho, usando datos estándar de C estaría obligado a emplear datos de tamaño 16-bit, 32-bit y 64-bit respectivamente, los cuales consumirían más recursos de forma innecesaria para obtener la misma precisión de salida que si se emplearan datos con ancho arbitrario.

Con esta librería es posible emplear variables con un ancho de bit escogido por el usuario. La diferencia entre forzar un diseño a emplear los tipos de datos C estándar y usar precisión en los tipos de datos se traduce en operadores hardware más pequeños y más rápidos, menor área de recursos consumida en la FPGA, y menor tiempo de ejecución.

Para el caso de lenguaje C, Vivado HLS permite dotar de precisión arbitraria a únicamente tipos enteros de datos. Para ello proporciona la cabecera *"ap_cint.h"*, la cual debe incluirse en el código fuente, definiéndose los tipos de datos, con o sin signo, como [u]int<W> siendo <W> el ancho de bit, que puede ir desde 1 a 1024 bits. Nuevamente, es útil disponer de un *test bench* para probar y verificar cuál es el tamaño mínimo adecuado para los datos. Para los diseños en C++ y System C, Vivado HLS proporciona cabeceras para dotar de precisión arbitraria tanto a tipos enteros de datos como a tipos de coma flotante. Puede consultarse en [28] *Chapter 1. HLS UltraFast Design Methodology. Data Types for Efficient Hardware*.

Modo de operación

Para un mayor rendimiento, las operaciones del código C son implementadas en el diseño RTL usando hardware único y ejecutándose de forma concurrente. Para ello se asignan recursos dedicados a cada operación del código permitiendo que éstas puedan ejecutarse de forma paralela. En ciertos diseños esta medida supone un beneficio en latencia e intervalo de iniciación, pero si la síntesis determina que no existe dicho beneficio automáticamente optará por compartir los recursos para beneficiarse, al menos, en menor área ocupada.

No obstante, la concurrencia en el hardware aún puede ser mejorada mediante segmentación, consiguiendo un menor tiempo de ejecución usando todos los recursos a emplear al mismo tiempo, permitiendo que posteriores transacciones del código comiencen a operar tan pronto como los recursos hardware estén disponibles.

Para aplicar segmentación al diseño existe una directiva de optimización denominada **Pipeline**, la cual se verá más en detalle posteriormente, y que puede aplicarse tanto a bucles como a funciones. Con ella, se indica al diseño que asigne tantos recursos como hagan falta para poder ejecutar en paralelo tantas operaciones como sea posible, y tan pronto como éstas hayan liberado recursos, se empleen para las transacciones posteriores. Un resultado ideal sería obtener intervalo de iniciación 1, lo que indica que se puede procesar una nueva entrada de datos por cada ciclo de reloj, brindando al diseño máximo rendimiento.

Por otro lado, existe también una directiva denominada **Dataflow**, con la cual se indica a la síntesis que debe garantizar, en la medida de lo posible, mayor rapidez en el paso de muestras de unas tareas a otras, de forma que en los casos en que sea posible, una *función consumidora* pueda empezar a operar antes de que la *función productora* de la que depende haya finalizado. Así se reduce en la medida de lo posible el intervalo de iniciación a costa de crear hardware que permita estas operaciones en paralelo.

Síntesis, directivas y configuraciones

Ya se ha comentado en apartados anteriores el comportamiento que tiene Vivado HLS de sintetizar el código fuente y las decisiones que toma en el momento de la implementación, pero como bien se dijo, este comportamiento puede modificarse por medio de directivas que el usuario incluye en el código fuente y que permiten personalizar distintas soluciones o alternativas de implementación para un mismo diseño.

Por supuesto, Vivado HLS cuenta con una estrategia propia de optimización, creando la implementación que considera más óptima basándose en su comportamiento por defecto y atendiendo a las características propias de la tarjeta, como la frecuencia de reloj. El orden de prioridad que tiene en cuenta en el momento de optimizar es:

Primero - Minimizar el intervalo de iniciación.

Segundo - Minimizar la latencia.

Tercero - Minimizar el área.

En ausencia de directivas incluidas por el usuario Vivado HLS busca cumplir estos objetivos, siguiendo el orden de prioridad y actuando en función de su comportamiento por defecto, para crear una solución inicial de la que partir.

En el apartado previo de *Metodología de trabajo* se detallaba de forma breve las claves del comportamiento por defecto que presenta Vivado HLS. A continuación se describirá de una forma más detallada el comportamiento de la síntesis ante distintas partes del diseño.

- Los argumentos de la función principal son sintetizados en puertos de datos controlados por un protocolo, *IO protocol*, el cual añade una o más señales asociadas al puerto de datos para sincronizar la comunicación entre el puerto y otros bloques hardware del sistema. A su vez, la función principal también implementa un protocolo IO para controlar el comienzo y fin de operación. Estos protocolos pueden definirse por el usuario mediante directivas, o pueden asumirse como Vivado HLS define por defecto a cada argumento en función de su naturaleza.

- Cada función en la descripción C se transforma en un único bloque en el diseño RTL. Sin embargo, en ocasiones es conveniente usar funciones *inline* para mejorar la optimización de la lógica. Si una función se declara *inline* el compilador realiza una copia del código de dicha función cada vez que se llame a ésta.

  Vivado HLS realiza automáticamente esta acción con funciones pequeñas para mejorar la calidad de la implementación, ya que se ejecutan más rápidamente evitando usar la pila y ahorrándose los pasos de salto y retorno en una llamada.

Puede especificarse a la síntesis de forma manual mediante la directiva Inline, pero debe tenerse en cuenta que si se declara *inline* una función grande, gran cantidad de código se copiará allí donde se llame a la función, y no se tendrá el beneficio de velocidad buscado.

- Los bucles de código, como ya se comentó, se mantienen enrollados (*roll*) utilizándose la misma lógica para cada iteración de manera secuencial. Pero, en ocasiones, para reducir la latencia, es necesario que estos bucles sean desenrollados (*unroll*) para completar las iteraciones en paralelo. Esto puede especificarse manualmente mediante la directiva Unroll.

- Los arrays son sintetizados en bloques RAM, decidiendo de forma automática si debe usarse un bloque RAM de un solo puerto o de doble puerto en función de las repercusiones que vaya a tener sobre la implementación en cuanto a intervalo y latencia. No obstante, la directiva Resource permite escoger el tipo de bloque RAM.

  Otra posibilidad consiste en la partición del array mediante la directiva Array_Partition, imponiendo que dicho array sea implementado en arrays más pequeños o mediante registros de desplazamiento en vez de emplear bloques RAM evitando así cuellos de botella.

Para dar una visión más clara de las acciones que se pueden llevar a cabo para redirigir el proceso de síntesis, se recogen en la siguiente Tabla 5 las principales directivas. La columna de aplicación indica la naturaleza del argumento del diseño donde puede ser aplicada.

Para aplicarla se abre el fichero principal, se busca en la pestaña *Directive* del panel auxiliar (derecha de la pantalla) el argumento sobre el que se quiere aplicar la directiva (array, función, bucle, escalar...) y se despliega su menú con el botón derecho escogiendo *Insert Directive*, pestaña en la cual se escoge la directiva y se fijan sus parámetros en caso de que sea necesario.

También hay que decir que, en el momento de inclusión, cualquier directiva puede situarse en el código como pragmas o en un fichero del proyecto que recoja las directivas aplicadas, *directives.tcl*. Si se escoge como destino *Directive File* aparecerá reflejada en el fichero .tcl y en el panel auxiliar bajo el argumento donde se ha aplicado, y si se escoge *Source File* aparecerá una sentencia en el código definida como pragma bajo el argumento donde se aplica. La desventaja de esta opción es que una vez se haya aplicado al código ésta se irá copiando de una solución a otra, mientras que si se encuentra en el panel auxiliar existe la opción de copiar o no las directivas al crear una nueva solución.

Directiva	Aplicación	Objetivo
INTERFACE	Argumentos de una función	Define cómo deben implementarse los puertos para el diseño RTL.
PIPELINE	Funciones y bucles	Favorece la concurrencia desenrollando los bucles o funciones a los que se aplica.
DATAFLOW	Funciones	Favorece el flujo de tareas permitiendo a funciones dependientes de otras comenzar a operar aún cuando éstas no hubieran finalizado.
ARRAY_PARTITION	Arrays	Particionado en arrays más pequeños (*type=block*) para tener así más puertos de acceso a los datos, o en registros individuales (*type=complete*).
INLINE	Funciones	Sugiere al compilador mejores oportunidades de optimización, mejorando la latencia al evitar el salto y retorno en una llamada a una función a costa de realizar una "copia" del código de la función llamada. Puede provocar el aumento de área.
UNROLL	Bucles	Desenrolla un bucle separando un conjunto de operaciones en múltiples operaciones independientes a ejecutar en paralelo, con el consiguiente incremento de la lógica a emplear.
ALLOCATION	Funciones y bucles	Limita el número de operaciones, recursos o funciones a usar. Puede forzar la compartición de recursos. Si se trata de recursos es equivalente a Resource.
RESOURCE	Arrays	Especificar el tipo de recurso hardware a usar con cierto elemento del código (arrays, operaciones).
LATENCY	Funciones y bucles	Permite especificar una latencia máxima y mínima. Fuerza compartición de recursos o introduce retardos.
LOOP_TRIPCOUNT	Bucles	En bucles con un número de iteraciones variable (que dependan de una condición, p.e contener un *if*) muestra una estimación del número de iteraciones.
LOOP_FLATTEN	Bucles	Combina bucles anidados en un único bucle para reducir la latencia del diseño.

TABLA 5. DIRECTIVAS DE OPTIMIZACIÓN MÁS COMUNES EN VIVADO HLS

Podría decirse que la clave para encontrar una implementación final óptima de un diseño está basada en las directivas Pipeline y Dataflow (si es posible). De hecho, un buen punto de partida es comenzar aplicando segmentación a los bucles antes que segmentar la propia función, ya que es posible que esta segunda opción resulte en más recursos hardware consumidos. Además, en caso de existir bucles anidados resulta más óptimo empezar por segmentar el bucle más bajo en la jerarquía.

En cuanto a configuraciones para la síntesis, se muestran las principales en la Tabla 6. Estas configuraciones se seleccionan desde los ajustes de la solución (*Solution Settings>Add*) donde se quieran especificar, no obstante, por lo general tienen menos impacto en la mejora de implementación que las directivas.

Configuraciones	Descripción
Config Interface	Controla los puertos E/S no asociados a los argumentos de la función principal permitiendo que puertos que no vayan a ser usados no se añadan diseño RTL final.
Config Array Partition	Controla los parámetros en la partición de arrays (partition_threshold, include_extern_globals, include_ports).
Config Compile	Controla ciertas optimizaciones automáticas de la síntesis, como el segmentado de bucles automático al aplicar segmentación a la función principal.
Config Bind	Controla el nivel de esfuerzo en el proceso de elección de recursos. Es útil para minimizar el número de operadores a emplear.
Config Schedule	Controla la cantidad de tiempo empleada para buscar una implementación óptima del diseño. Puede ser útil fijarlo a Low cuando hay muchas combinaciones posibles para así reducir el tiempo de ejecución o escoger High si se quiere emplear más tiempo en buscar un diseño menor y más rápido a pesar de haber satisfecho ya los requisitos marcados.

TABLA 6. PRINCIPALES CONFIGURACIONES EN VIVADO HLS

La manera en que se aplican las directivas y el impacto que tienen en el diseño podrá verse posteriormente en el apartado *Ejemplo de aplicación de directivas*. El propósito de este ejemplo es familiarizar al usuario con el entorno Vivado HLS para que sitúe cada uno de los conocimientos teóricos que se han expuesto a lo largo de esta base teórica, de esta forma esto no supondrá una distracción a la hora de comprender el diseño final que trata sobre la implementación de un filtro FIR.

Por tanto, antes de dar paso a dicho ejemplo es conveniente continuar conociendo cómo funciona la herramienta Vivado HLS para posteriormente afianzar los conceptos expuestos mediante el diseño que se vaya a mostrar.

Interfaces

En apartados previos se ha hablado de la sección denominada Interfaz que se muestra al final del informe devuelvo tras la síntesis sobre el diseño RTL implementado. En esta sección, como ya se dijo, se muestran los puertos RTL, a los que ha dado lugar la síntesis, asociados a los argumentos de la función principal. Estos puertos van controlados por protocolos I/O de diferentes tipos. Cada protocolo asignado a un puerto del diseño especifica el conjunto de señales que irán asociadas a ese puerto.

El proceso en el cual Vivado HLS, tras la síntesis, crea los puertos del bloque RTL y escoge el protocolo (conjunto de señales) asociado que más se ajuste a la naturaleza de cada puerto, es conocido como *Interface Synthesis*.

Vivado HLS crea tres tipos de puertos:

- Puertos de reloj y reset (*ap_clk* y *ap_rst*).
  Se incluyen cuando al diseño le va a llevar más de un ciclo ejecutarse.

- Puertos de control (*ap_start, ap_ready, ap_idle* y *ap_done*).
  Se añaden por defecto para controlar cuándo el bloque RTL puede empezar a operar (ap_start), cuándo puede procesar nuevas entradas (ap_ready), cuándo está desocupado (ap_idle) y cuándo ha terminado de operar (ap_done), consultando el valor del bit asociado a la señal para saber si está activa o no.

- Puertos de datos.
  Puertos correspondientes a cada parámetro de la función principal y al parámetro *return* (aunque la función no retorne valor), para manejar los datos de entrada y salida al bloque RTL.

Vivado HLS tiene un modo por defecto de asignar estos puertos y sus protocolos:

- ✓ Para argumentos de entrada pasados por valor y para punteros se asigna un puerto simple sin protocolo asociado (protocolo *ap_none*).
- ✓ Para punteros de salida se asigna un puerto acompañado de una señal de salida válida (protocolo *ap_vld*).
- ✓ El parámetro *return* se implementa en un puerto de salida acompañado de las señales de control arriba citadas, *ap_start, ap_ready, ap_idle* y *ap_done* (protocolo *ap_ctrl_hs*).
- ✓ Los argumentos que son tanto de lectura como de escritura son separados en puertos diferentes de entrada y de salida, cada uno con su protocolo correspondiente.

Como puede verse, las interfaces que tengan los puertos dependen de la naturaleza del argumento y de si existe o no una directiva de optimización especificada por el usuario.

Brevemente se van a exponer los tipos de protocolo que existen y por qué se caracterizan:

- Protocolos a nivel de bloque (*ap_ctrl_none, ap_ctrl_hs* y *ap_ctrl_chain*): son protocolos para el argumento *return* de la función principal que contienen señales de control del bloque RTL. El modo *ap_ctrl_hs* es el escogido por defecto, cuyas señales son *ap_start, ap_ready, ap_idle* y *ap_done*. El modo *ap_ctrl_chain* contiene además de ésas una señal de entrada *ap_continue* para dar paso o inhabilitar nuevas transacciones de código. Por otro lado puede especificarse un puerto sin protocolo de bloque con *ap_ctrl_none*.

- Protocolos a nivel de puerto - Sin asignación de protocolo (*ap_none* y *ap_stable*): se implementa el argumento como un puerto de datos sin señales asociadas.

- Protocolos a nivel de puerto – Señales de asentimiento (*ap_ack, ap_vld, ap_ovld* y *ap_hs*): El modo más completo es *ap_hs* pues incluye dos formas de supervisión: una señal de datos válidos (*ap_vld*) y otra señal de reconocimiento (*ap_ack*). Los modos *ap_vld* y *ap_ack* contienen únicamente la señal de validez o de reconocimiento, respectivamente. El modo *ap_ovld* para argumentos IO se divide en un puerto de entrada con protocolo *ap_none* (sin señales), y un puerto de salida con protocolo *ap_vld* (con señal *ap_vld*).

- Protocolos a nivel de puerto – Interfaces de memoria (*ap_memory, bram, ap_fifo* y *ap_bus*): Por defecto los arrays se implementan en el modo *ap_memory*, el cual incluye puertos para datos, dirección, chip-enable y write-enable.
  El modo *bram* es exactamente igual que *ap_memory* salvo que *ap_memory* se implementa como múltiples e independientes puertos, y *bram* como un único grupo de puertos que sólo requieren de una conexión para conectarlos al bloque RAM.
  Para arrays donde el acceso será secuencial puede especificarse el modo *ap_fifo*. Finalmente la interfaz *ap_bus* no se asocia con ningún bus estándar y puede comunicarse con un puente de buses.

- Protocolos a nivel de puerto - Interfaces AXI4: Vivado HLS soporta las interfaces AXI4-Stream (*axis*), AXI4-Lite (*s_axilite*) y AXI4 master (*m_axi*). Éstas son equivalentes a las ya vistas con las herramientas EDK.
  La interfaz AXI4-Stream sirve para transferir datos en modo ráfaga. Con AXI4-Lite pueden agruparse varios puertos en esta misma interfaz y crean unos drivers para la comunicación entre el periférico y el microcontrolador. Y finalmente con AXI4 master pueden hacerse transferencias de datos individuales o en modo ráfaga.

  La Tabla 7 muestra las posibles asociaciones que pueden tener lugar entre argumentos y protocolos, indicándose con un asterisco (*) la interfaz asignada por defecto.

	ARGUMENTO							
PROTOCOLO	*ESCALAR / Por valor*		*ARRAY/ Por referencia*			*PUNTERO/Por referencia*		
	Entrada	**Return**	**I**	**IO**	**O**	**I**	**IO**	**O**
ap_ctrl_none	⊘	☺	⊘	⊘	⊘	⊘	⊘	⊘
ap_ctrl_hs	⊘	☺*	⊘	⊘	⊘	⊘	⊘	⊘
ap_ctrl_chain	⊘	☺	⊘	⊘	⊘	⊘	⊘	⊘
ap_none	☺*	⊘	⊘	⊘	⊘	☺*	☺	☺
ap_stable	☺	⊘	⊘	⊘	⊘	☺	⊘	⊘
ap_ack	☺	⊘	⊘	⊘	⊘	☺	☺	☺
ap_vld	☺	⊘	⊘	⊘	⊘	☺	☺	☺*
ap_ovld	⊘	⊘	⊘	⊘	⊘	⊘	☺*	☺
ap_hs	☺	⊘	☺	⊘	☺	☺	☺	☺
ap_memory	⊘	⊘	☺*	☺*	☺*	⊘	⊘	⊘
bram	⊘	⊘	☺	☺	☺	⊘	⊘	⊘
ap_fifo	⊘	⊘	☺	⊘	☺	☺	⊘	☺
ap_bus	⊘	⊘	☺	☺	☺	☺	☺	☺
axis	☺	⊘	☺	⊘	☺	☺	⊘	☺
s_axilite	☺	☺	⊘	⊘	⊘	☺	☺	☺
m_axi	⊘	⊘	☺	☺	☺	☺	☺	☺

TABLA 7. ASOCIACIONES DE PROTOCOLOS PERMITIDAS O DENEGADAS SEGÚN EL TIPO DE ARGUMENTO EN VIVADO HLS

Si el usuario quisiera emplear distintas interfaces de las que Vivado HLS asigna por defecto, puede hacerlo aplicando sobre el puerto la directiva Interface (en la pestaña Directive del panel auxiliar de la Figura 85 donde se mostrarán los puertos del diseño, clic con el botón derecho sobre el

nombre del puerto y escoger *Insert Directive>Interface*), donde podrá escoger el modo de interfaz de entre las posibles para ese puerto.

Antes de finalizar este apartado, conviene entrar en detalles de cómo funciona la interfaz AXI4-Lite, ya que será ésta la empleada en el diseño a realizar.

Esta interfaz se aplica sobre dispositivos esclavos y permite que sean controlados por el microcontrolador. Se caracteriza por agrupar en una única interfaz todos los puertos que llevan esta interfaz especificada, y por crear drivers en el momento de exportación para la comunicación con el microcontrolador.

Es conveniente, en el momento de asignar la directiva Interface a cada puerto, introducir el nombre que se desea que tenga dicha interfaz en la opción *bundle* siendo éste igual para todos los puertos para así recoger estos puertos bajo ese nombre en el diseño RTL final. Además, el argumento *return* de la función debe llevar también esta interfaz especificada si se quiere que el diseño final incluya un puerto de salida de interrupción.

En cuanto a los drivers creados en el momento de empaquetado del diseño como IP, éstos se encuentran en el directorio de implementación y están compuestos por un conjunto de APIs cuyo nombre viene dado por la función principal y que se emplearán en el software para comunicarse con el dispositivo esclavo mediante la interfaz AXI4-Lite.

La Tabla 8 recoge los drivers que se crean tras la síntesis para un diseño de nombre “example”.

Directorio principal: *impl>ip>drivers>top_function_v1_0>*	**Descripción**
data/example.mdd	Fichero de definición. Contiene las opciones definidas para los driver del periférico.
data/example.tcl	Necesaria para constituir el proyecto en SDK.
src/xexample_hw.h	Fichero de cabecera hardware que contiene las direcciones y offset de los puertos en memoria.
src/xexample.h	Contiene Includes, definiciones de tipos y definiciones de las funciones para la comunicación.
src/xexample.c *src/xexample_sinit.c* *src/xexample_linux.c*	Contienen las funciones para inicialización, control, estado, lectura/escritura de registros y manejo de interrupciones.
src/Makefile	Fichero makefile.

TABLA 8. DIRECTORIO DE LOS DRIVERS CREADOS TRAS LA SÍNTESIS EN VIVADO HLS

En el fichero *xexample.c* que contiene el cuerpo de funciones para comunicación con el periférico mediante la interfaz AXI4-Lite pueden encontrarse las siguientes funciones [28]. Su uso se verá posteriormente cuando se empleen para manejar el periférico desde el procesador una vez haya sido exportado el diseño de Vivado HLS a SDK.

- *XExample_Initialize*: Inicializa el dispositivo escribiendo información necesaria para su configuración, que posteriormente será empleada por el resto de funciones.
- *XExample_CfgInitialize*: Inicializa la configuración del dispositivo con la información retornada por la función de inicialización.
- *XExample_LookupConfig*: Sirve para obtener la información de configuración del dispositivo a partir de su ID devolviendo un puntero a dicha información (Id + Dirección base en memoria).
- *XExample_Start*: A partir del puntero devuelto por la función anterior inicia el dispositivo para que comience a operar. Pone la señal ap_start a nivel alto.
- *XExample_IsDone*: Comprueba si la función ha terminado de ejecutarse devolviendo el valor del puerto ap_done.
- *XExample_IsIdle*: Comprueba si el dispositivo se encuentra desocupado devolviendo el valor del puerto ap_idle.
- *XExample_IsReady*: Comprueba si el dispositivo está listo para el siguiente conjunto de entradas a procesar devolviendo el valor del puerto ap_ready.
- *XExample_Continue*: Pone el puerto ap_continue a nivel alto en caso de que éste pertenezca al diseño (modo ap_ctrl_chain).
- *XExample_EnableAutoRestart*: Habilita que el dispositivo continúe ejecutando una transacción tras otra de manera automática.
- *XExample_DisableAutoRestart*: Deshabilita la función anterior.
- *XExample_Set_ARG*: Escribe el valor pasado por argumento en el puerto ARG de entrada.
- *XExample_Set_ARG_vld*: Pone a nivel alto la señal ARG_vld indicando entrada válida si el puerto ARG es un puerto de entrada implementado con la interfaz ap_hs o ap_vld (protocolos que contienen la señal ARG_vld).
- *XExample_Set_ARG_ack*: Asiente ARG_ack reconociendo la salida si el puerto ARG es un puerto de salida implementado con la interfaz ap_hs o ap_ack (protocolos que contienen la señal ARG_ack).
- *XExample_Get_ARG*: Obtiene el valor del puerto ARG de salida.
- *XExample_Get_ARG_vld*: Obtiene el valor de ARG_vld si ARG es un puerto de salida implementado con la interfaz ap_hs o ap_vld.
- *XExample_Get_ARG_ack*: Obtiene el valor de ARG_ack si ARG es un puerto de entrada implementado con la interfaz ap_hs o ap_ack.
- *XExample_InterruptGlobalEnable*: Habilita la salida de interrupción del periférico. Para ello debe existir la señal ap_start.
- *XExample_InterruptGlobalDisable*: Deshabilita la función anterior.
- *XExample_InterruptEnable*: Habilita la fuente de interrupción. Para ello tiene que haber al menos dos fuentes de interrupción (ap_done y ap_ready).
- *XExample_InterruptDisable*: Deshabilita la fuente de interrupción.
- *XExample_InterruptClear*: Borra el flag de interrupción.
- *XExample_InterruptGetEnabled*: Comprueba qué fuentes de interrupción están habilitadas.
- *XExample_InterruptGetStatus*: Comprueba el estado de las fuentes de interrupción.

La localización en memoria de todos estos puertos se encuentra en el fichero *xexample_hw.h*, donde viene indicada la dirección base para los conjuntos *Control signals*, *Global Interrupt Enable Register*, *IP Interrupt Enable Register*, *IP Interrupt Status Register*, *Control signal of ARG1*, *Data signal of ARG1*, *Data signal of ARG2*, etc, y el offset a cada señal dentro de cada conjunto. Por ejemplo, para las señales de control la dirección base es 0x00 donde el bit 0 corresponde a *ap_start*, bit 1 a *ap_done*, bit 2 a *ap_idle*, bit 3 a *ap_ready* y bit 7 a *auto_restart*.

Finalmente, el modo más común de programar esta interfaz y de controlar el bloque hardware desde el procesador sigue la siguiente secuencia:

1.- Crear la instancia hardware.

2.- Obtener la configuración del periférico.

3.- Inicializar el dispositivo.

4.- Cargar los datos de entrada al bloque.

5.- Iniciar el dispositivo.

6.- Leer los resultados de los registros de salida.

7.- Repetir estos pasos para la siguiente ejecución.

Además, puede configurarse para que el periférico interrumpa al micro cuando existan resultados a su salida.

Este procedimiento para controlar el bloque hardware podrá verse de forma más clara posteriormente cuando se lleve a cabo la creación de un periférico y se realice la aplicación software para la comunicación periférico-microprocesador.

Reloj

Vivado HLS permite fijar una única frecuencia de reloj determinada aplicable a todas las funciones del diseño para diseños en C y C++. Este periodo de reloj se especifica al crear un nuevo proyecto, y además puede fijarse en los ajustes de la propia solución del diseño.

A partir de la frecuencia de reloj y de la información referente a la tarjeta a utilizar, Vivado HLS estima los ciclos que conlleva cada operación pero no puede determinar con exactitud el tiempo total del diseño ya que no conoce la disposición final de los componentes ni su enrutamiento, por ello utiliza el concepto de "*clock uncertainty*" para añadir un margen de incertidumbre a las estimaciones que realiza de forma que la síntesis del diseño RTL y las operaciones de *place & route* tengan suficiente margen temporal para completar sus operaciones con éxito.

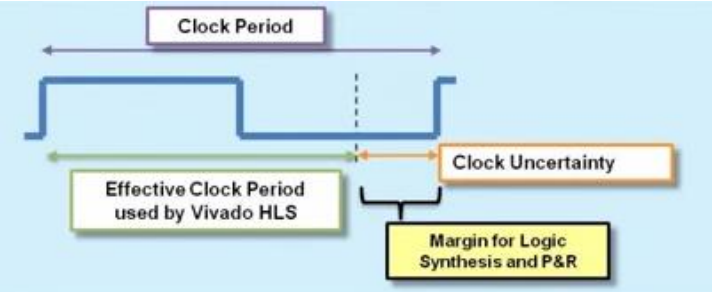


FIGURA 87. CONCEPTO DE CLOCK UNCERTAINTY EN VIVADO HLS

Por defecto este margen temporal es el 12.5% del ciclo de reloj, pero puede darse el caso de que el dispositivo requiera un alto grado de emplazamiento y enrutamiento necesitando un retardo mayor del esperado. En ese caso se debe aumentar el margen temporal de forma que no se cree un diseño con mucha lógica por ciclo de reloj para que así la síntesis RTL pueda completarse. Si esta solución supone usar un tiempo demasiado grande, puede optimizarse el diseño empleando directivas para fijar los ciclos máximos y mínimos de latencia, o los recursos a utilizar.

Optimización en rendimiento

Hasta ahora, en lo referente a optimizaciones de código, se ha hablado de optimización en rendimiento mediante las directivas Pipeline y Dataflow. Se sabe que con estas directivas puede conseguirse que las operaciones se realicen de forma concurrente, segmentando funciones y bucles. Esto supone una mejora en términos de intervalo de iniciación y de latencia.

La directiva Pipeline se aplica sobre una función provocando también un desenrollado de los bucles que contiene tal como hace la directiva Unroll, para permitir la ejecución concurrente, pero esta segmentación no afecta a sub-funciones que se encuentren dentro de ella, éstas deben segmentarse individualmente o supondrán un factor limitante en la eficacia de esta optimización.

En el caso de segmentación aplicada a bucles, la segmentación tiene lugar entre iteraciones del bucle y, por defecto, un bucle debe terminar de operar antes de comenzar a ejecutarse el siguiente (también segmentado), dando lugar a que exista cierto momento entre bucles en que no se leen entradas (porque se está escribiendo la última salida del *"bucle i"*) o no se escriben salidas (porque se está leyendo la primera entrada del *"bucle i+1"*). Esto puede entenderse gráficamente como muestra la Figura 88.

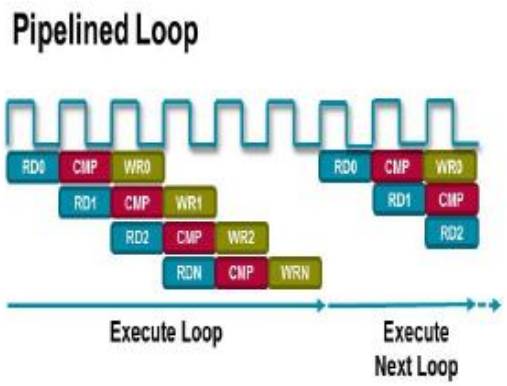


FIGURA 88. EJECUCIÓN POR DEFECTO DE BUCLES SEGMENTADOS EN VIVADO HLS

Esta medida provoca el consumo de ciclos de reloj poco productivos pero puede solucionarse empleando la directiva Pipeline, con la opción *rewind* marcada indicando que al finalizar el bucle finalizaría la función principal del diseño (se verá en el apartado *Ejemplo de aplicación de directivas*), consiguiendo una ejecución continua tal como si se tratase de iteraciones dentro de un mismo bucle.

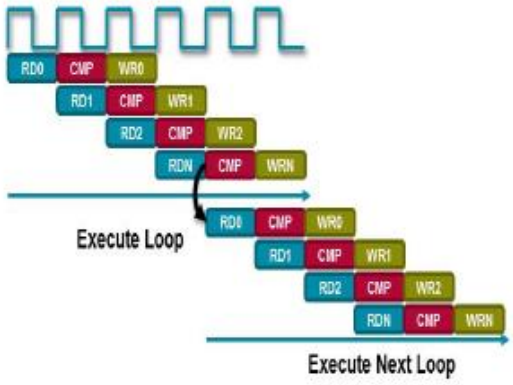


FIGURA 89. EJECUCIÓN DE BUCLES SEGMENTADOS CON OPCIÓN REWIND EN VIVADO HLS

Además, existe la posibilidad de especificar que los bucles sean segmentados de forma automática desde la configuración *config_compile*, que se puede encontrar en los ajustes de la propia solución, donde se puede fijar el límite de iteraciones que debe tener un bucle (factor *pipeline_loops*) para que se lleve a cabo la segmentación automática, y, si no se quiere que esto suceda para algún bucle específico del diseño, puede deshabilitarse fijando la directiva Pipeline con la opción *off* marcada.

Otro hecho que limita la eficacia de estas directivas es el uso de arrays en el diseño. Vivado HLS implementa los arrays por defecto como bloques de memoria RAM. Si se aplica segmentación a ese diseño, la síntesis devolverá un warning informando de que no ha sido posible conseguir el objetivo marcado (el objetivo se tratará de conseguir intervalo de iniciación 1, por defecto) debido a que los bloques RAM tienen un máximo de dos puertos de datos, limitando en gran medida el rendimiento si el diseño tiene que realizar numerosas lecturas y escrituras. Para solucionarlo, como ya se comentó, se dispone de la directiva Array_partition, con la cual puede dividirse el array original en múltiples arrays de menor tamaño (y por consiguiente, múltiples bloques RAM),

incrementándose el número de puertos de acceso, o realizarse la partición completa del array original en elementos individuales, implementándose como registros de desplazamiento.

Esta medida puede ser necesaria en escenarios de desenrollado parcial o completo de bucles, ya que esta directiva fuerza a que la lógica empleada para una iteración se copie cierto número de veces para operar en paralelo. Así, puede darse el caso de que al aplicar la directiva Unroll a un bucle donde se suceden operaciones de lectura/escritura, resulte en tantas operaciones de lectura/escritura a ejecutar de forma paralela en un mismo ciclo como iteraciones tenga el bucle o como factor de desenrollado parcial se haya especificado. Si deben sucederse 2 operaciones R/W pueden emplearse RAMs de doble puerto para soportarlas, pero si el factor es mayor o si el desenrollado es completo no existen bloques RAM que puedan realizar tantas operaciones R/W por ciclo, por lo que es necesario aplicar la directiva Array_partition.

De igual forma, puede automatizarse el particionado de arrays desde la configuración *config_array_partition*, que se puede encontrar en los ajustes de la propia solución, donde se puede fijar el número de elementos que debe tener un array (factor *throughput_driven*) para que se lleve a cabo la partición automática.

Finalmente, la directiva Dataflow consigue que las diferentes tareas que se suceden en el código se ejecuten de forma paralela, sin necesidad de que una tarea tenga que esperar a que una tarea previa termine de completar sus operaciones sino que si ya tiene la información necesaria dada por la tarea que la precede pueda comenzar a ejecutarse. Esta superposición de tareas supone un beneficio en rendimiento al reducir la latencia del diseño, pero puede significar un aumento de la lógica a emplear si es necesario salvar la dependencia de unas tareas con otras. No obstante, esta optimización no puede aplicarse en tareas cuya ejecución dependa de una condición, como por ejemplo un bucle precedido por la sentencia if, en cuyo caso el código debería ser modificado para trasladar la sentencia if dentro del bucle.

Optimización en latencia

La directiva principal para optimizar el número de ciclos que requiere el diseño para producir una salida es la directiva Latency, con la cual puede especificarse un máximo y/o un mínimo de ciclos indicando a Vivado HLS que debe asegurar que todas las operaciones se completan dentro de ese rango.

Si esta directiva se aplica en el interior de un bucle el rango de ciclos especificados es referido a la latencia de una única iteración. Sin embargo, si se aplica en el exterior del bucle, el número de ciclos especificados en la directiva se referirá a la latencia total de todas las iteraciones, incluso cuando el bucle fuese desenrollado.

Otra opción que ofrece Vivado HLS para reducir la latencia del diseño es la directiva Loop_merge mediante la cual la lógica perteneciente a varios bucles consecutivos puede ser optimizada de

forma conjunta. Esta medida tiene como objetivo evitar el ciclo de más que suponen la entrada y salida de cada bucle, agrupando la lógica como si de un único bucle se tratara.

En la misma línea se encuentra la directiva Loop_flatten, la cual tiene un objetivo similar para reducir la latencia, y es evitar el ciclo de más que suponen los movimientos entre bucles anidados, la entrada y salida de un bucle externo a uno interno, aplicándose esta directiva al bucle de menor jerarquía. De esta forma se evita al usuario la necesidad de modificar el código para mejorar la optimización.

Optimización en área

Existen diversas formas de optimizar el diseño para emplear el menor número de recursos hardware posible.

La forma más eficaz de lograr este objetivo es emplear la precisión de bits adecuada para cada puerto del diseño. Esta medida ya fue comentada en el apartado denominado *Librerías*, en el cual se habla de las opciones que ofrece Vivado HLS para trabajar con tipos de datos con un ancho de bit definido por el usuario y que se corresponda con el mínimo ancho necesario. Es importante analizar el diseño para determinar el tipo de dato que debe asignarse a cada variable de forma que se ocupe únicamente el hardware que realmente se necesita, aumentando el número de operaciones que pueden llevarse a cabo en un ciclo de reloj y reduciendo considerablemente la latencia y el intervalo de iniciación.

Otro aspecto que se debe analizar en el diseño es si éste contiene varios arrays de menor tamaño que puedan mapearse en un array mayor reduciendo el número de bloques RAM a utilizar y aprovechando al completo los que se utilizan (puede hacerse un seguimiento de los bloques RAM que utiliza el diseño en el apartado de *Memoria* del informe devuelto por la síntesis)

Para ello la directiva a emplear se denomina Array_map y debe aplicarse a cada uno de los arrays que se quieran mapear en un array mayor. Soporta mapeado horizontal y vertical.

En el mapeado horizontal los arrays se concatenan uno tras otro en el orden en que se ha aplicado la directiva. Tiene como ventaja un menor consumo de bloques RAM, mejorando el área, pero peor rendimiento al existir menor número de puertos de datos para operaciones en memoria.

Para el caso de mapeado horizontal de los arrays: array1[M]={0,1,…,M-2,M-1} y array2[N]={0,1,…,N-2,N-1}, la implementación en memoria del array3 resultante quedaría de la forma que se observa en la Figura 90.

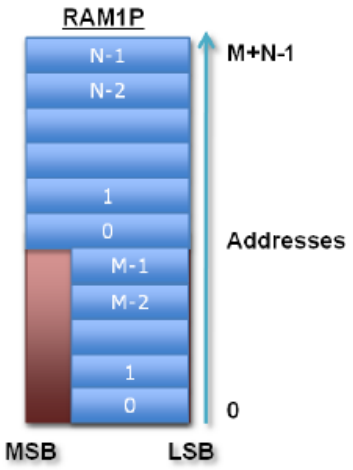


FIGURA 90. MAPEADO HORIZONTAL DE ARRAYS EN VIVADO HLS

En el mapeado vertical se concatenan los elementos de los arrays menores dando lugar a un array con mayor ancho de bit. Siguen el orden en que se ha aplicado la directiva, situando para cada elemento el primer array en el LSB (*Lest Significant Bit*) y el último array en el MSB (*Most Significant Bit*) de cada posición. Supone una mejora en el rendimiento con respecto al mapeado horizontal al poder acceder a mayor número de elementos para operaciones en memoria.

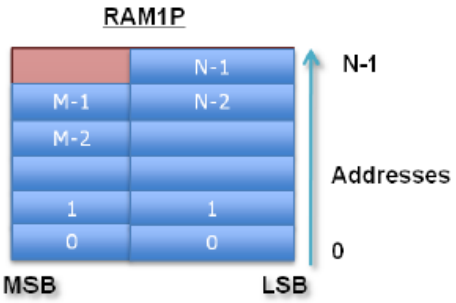


FIGURA 91. MAPEADO VERTICAL DE ARRAYS EN VIVADO HLS

La Figura 91 muestra la implementación en memoria del array mayor de la Figura 90 para mapeado horizontal en mapeado vertical.

Finalmente, otra opción que ofrece Vivado HLS para reducir el área consumida, es llevar un control de los recursos hardware para la implementación.

La directiva Allocation toma el papel de optimizar recursos, pues permite limitar el número de operadores, recursos o funciones que se utilizarán en el diseño. De esta manera se fuerza, en algunos casos, la compartición de recursos, pues puede suceder que un diseño requiera un número de operaciones con cierto operador que supere el número de recursos de la FPGA disponibles para esa operación. En ese caso podría resultar útil fijar como límite de la directiva Allocation el número de operadores disponibles escogiendo en la opción *type*: *operation*, y seleccionando de la lista si se trata de multiplicación (*mul*), suma (*add*), división con o sin signo (*sdiv/udiv*), resta (*sub*), registro de desplazamiento (*shl*), comparación (*icmp*), etc…

También es posible controlar los recursos mediante la configuración *Config_bind* para minimizar en todo el diseño el número de operadores mediante la opción *min_op* y escogiendo la operación a minimizar de entre las anteriormente citadas. A su vez, puede seleccionarse la estrategia a seguir entre *Low Effort* (dedica poco tiempo a buscar compartición de recursos para no malgastar ciclos de ejecución), *Medium Effort* (por defecto) y *High Effort* (maximiza la compartición de recursos sin importar el tiempo de ejecución, explorando todas las posibles combinaciones para un mejor resultado).

La otra directiva principal para especificación de recursos a emplear en el diseño es la directiva Resource, con la cual es posible indicar de forma directa a la síntesis qué recurso debe asociar a cierta variable. De entre los recursos disponibles se muestran tanto las operaciones lógicas comunes (*AddSub, Cmp, Div, Mul, MulnS* (n-stage pipelined),..) como registros o memorias (*FIFO, RAM_1P, RAM_1P_BRAM, RAM_1P_LUTRAM, RAM_2P, ROM_1P, ROM_2P,…*).

Diseño RTL

Tras la síntesis, una vez encontrada la implementación más óptima para el diseño a realizar, es necesaria una verificación del diseño RTL. Esta verificación se realiza mediante la opción *C/RTL Cosimulation*, la cual automáticamente reutiliza, como ya se comentó, el *test bench* empleado para la simulación de la descripción C previa a la síntesis.

Este proceso de verificación llevado a cabo por la opción *C/RTL Cosimulation* se compone de tres fases que realiza Vivado HLS al aplicarse esta acción:

- Simulación del diseño C y generación de vectores de test de entrada.
- Simulación del diseño RTL con los vectores de entrada generados en la fase anterior, y generación de vectores de salida.
- Verificación de los vectores de salida generados en la fase anterior introduciéndolos en el *test bench* y esperando que éste retorne un 0 (*PASS*).

Para poder verificar que el diseño RTL produce los mismos resultados que el código C original, es necesario que se disponga de un *test bench* con la propiedad de ser self-checking que permita conocer si la verificación ha resultado exitosa o no, retornando un valor 0 o distinto de 0, respectivamente. Esto aumenta también la productividad evitando que sea el propio desarrollador quien realice la comprobación manualmente.

Una vez se hayan realizado simulación, síntesis y verificación RTL, el último paso es exportar el diseño RTL final como un bloque IP (*Intellectual Property*) a otras herramientas de diseño de Xilinx para ser sintetizada en un fichero *bitstream* necesario para programar la FPGA.

Al seleccionar la opción *Export RTL* el usuario puede escoger entre los siguientes formatos de salida, disponibles para diseños en 7-series o dispositivos Zynq:

- IP Catalog: Fichero ZIP para añadir al Catálogo IP de Vivado Design Suite.
- System Generator for DSP para emplear en la edición de Vivado.
- System Generator for DSP (ISE) para emplear en la edición de ISE.
- Pcore for EDK para emplear en Xilinx Platform Studio.
- Synthesized Checkpoint (.dcp) crea *"Vivado checkpoint files"* para añadirlos a Vivado Design Suite.

Y además de estos formatos de empaquetado del diseño, los ficheros independientes pueden encontrarse en los directorios *vhdl* o *verilog* dentro del directorio de implementación, como el proyecto para abrir en Vivado para cada lenguaje.

Otra opción, disponible para el usuario durante el proceso de exportación del diseño, consiste en ejecutar la síntesis RTL para comprobar la precisión de las estimaciones indicadas en el informe de síntesis inicial y confirmar si la frecuencia de reloj, la latencia y los recursos empleados se corresponden con los estimados. Para activar esta opción debe marcarse la casilla *Evaluate* en la opción Export RTL.

Ejemplo de aplicación de directivas

Como se ha dicho, el diseño en Vivado HLS que aquí se va a realizar no tiene un propósito final de optimización concreto, sino que únicamente pretende mostrar el impacto que tienen en el código diferentes directivas según el momento y lugar de aplicación, guiándolo a conseguir baja latencia e intervalo de iniciación, así como el menor número de recursos posible.

El código creado consiste en una multiplicación de matrices de dimensiones 7x7. La función principal se denomina matrix_mul y está contenida en el fichero *matrix_mul.c*. También se han generado los ficheros *matrix_mul.h* y *matrix_mul_test.c* con las definiciones del programa y el testbench, respectivamente.

El código de estos ficheros y su descripción puede verse a continuación:

```
/*************************************************************************
Project Name: MatrixMul_hls
Associated Filename: matrix_mul.c
Family: Zynq
Device: xc7z020clg484-1
*************************************************************************/
#include "matrix_mul.h"
```

```
void matrix_mul(matrix_a a[DIM][DIM], matrix_b b[DIM][DIM], matrix_c
c[DIM][DIM])
{

	int i,j,k;

	Row_Loop:
	for (i=0; i<DIM;i++)
	{
	  Col_Loop:
	  for(j=0;j<DIM;j++)
	  {
		c[i][j] = 0;

		Multiplication_Loop:
		for(k=0;k<DIM;k++)
		{
			c[i][j] += a[i][k] * b[k][j];
		}

	  }
	}

}
```

Éste es el código para la función top-level, matrix_mul, el cual contiene la sentencia include para asociarlo al fichero de cabecera y el cuerpo de dicha función que realiza la multiplicación de las matrices a y b pasadas por argumento a la función y devuelve el resultado en la matriz c que también se encuentra como argumento para posteriormente ser recogida desde el testbench.

Como puede verse la multiplicación se lleva a cabo mediante bucles anidados. El primer bucle, Row_Loop, recorre las filas de la matriz a, el segundo bucle, Col_Loop, recorre las columnas de la matriz b (tal como se haría en una multiplicación de matrices manual) y finalmente el tercer bucle recoge en la matriz c de salida el resultado de la suma de las multiplicaciones de cada factor fila-columna. Además, la sentencia `c[i][j] = 0;` resetea el resultado acumulado cada vez que se finaliza una combinación fila*columna.

```
/*********************************************************************
Project Name: MatrixMul_hls
Associated Filename: matrix_mul.h
Family: Zynq
Device: xc7z020clg484-1
*********************************************************************/
#define DIM  7

typedef int matrix_a;
typedef int matrix_b;
typedef int matrix_c;

void matrix_mul (matrix_a[DIM][DIM], matrix_b[DIM][DIM], matrix_c[DIM][DIM]);
```

Este código pertenece al fichero de cabecera *matrix_mul.h* generado, el cual contiene la definición de la variable *DIM* para la dimensión de las matrices, que en este caso son matrices cuadradas de dimensión 7x7, la definición de los tipos de las matrices denominados *matrix_a, matrix_b* y *matrix_c* que por el momento equivalen a tipo int, y la definición de la función principal o top-level, matrix_mul.

```
/*************************************************************************
Project Name: MatrixMul_hls
Associated Filename: matrix_mul_test.c
Family: Zynq
Device: xc7z020clg484-1
*************************************************************************/
#include <stdio.h>
#include "matrix_mul.h"

int main ()
{

      int i,j;
      // Define input arrays in1 and in2
      matrix_a   in1[DIM][DIM]={{1,2,3,4,5,6,7},{1,2,1,2,1,2,1},{3,4,3,4,3,4,3},
{4,5,4,5,4,5,4},{5,5,5,6,6,6,7},{7,6,7,6,7,6,8},{8,9,8,9,8,9,8}};

      matrix_b   in2[DIM][DIM]={{-1,3,5,-4,0,1,2},{6,2,-3,7,-4,-5,3},{1,2,1,2,1,
2,1},{7,6,5,4,3,2,1},{-2,-3,-4,-5,-8,-1,0},{9,8,7,6,5,4,3},{-2,-1,3,0,4,-3,2} };

      matrix_c result[DIM][DIM];

      // Execute the function.
      matrix_mul(in1,in2,result);

      //Print results.
      for (i=0; i<DIM;i++)
      {
        for(j=0;j<DIM;j++)
        {
              printf("%d ",result[i][j]);

        }
              printf("\n");
      }
    return 0;
}
```

En el testbench inicializa las matrices de entrada *in1* e *in2* y se define la matriz *result* para recoger el resultado de la multiplicación. Posteriormente se pasan como argumentos en la llamada a la función *matrix_mul(in1,in2,result)* y finalmente se imprimen por la consola los valores recogidos en la matriz *result*. Además contiene también la sentencia include con el fichero de cabecera.

Una vez descrito el código del ejemplo, puede profundarse en las operaciones que realiza la función principal. Se van a realizar DIMxDIM operaciones de lectura en la matriz in1, DIMxDIM operaciones de lectura en la matriz in2 y DIMxDIM operaciones de escritura en la matriz result.

Este es el código que se tendrá para la primera solución del diseño en Vivado HLS, *solution1*. Simulando esta solución con Run C Simulation de la barra de herramientas (ver Figura 85) se comprueba que el código es correcto y retorna un resultado válido, como se muestra en la Figura 92.

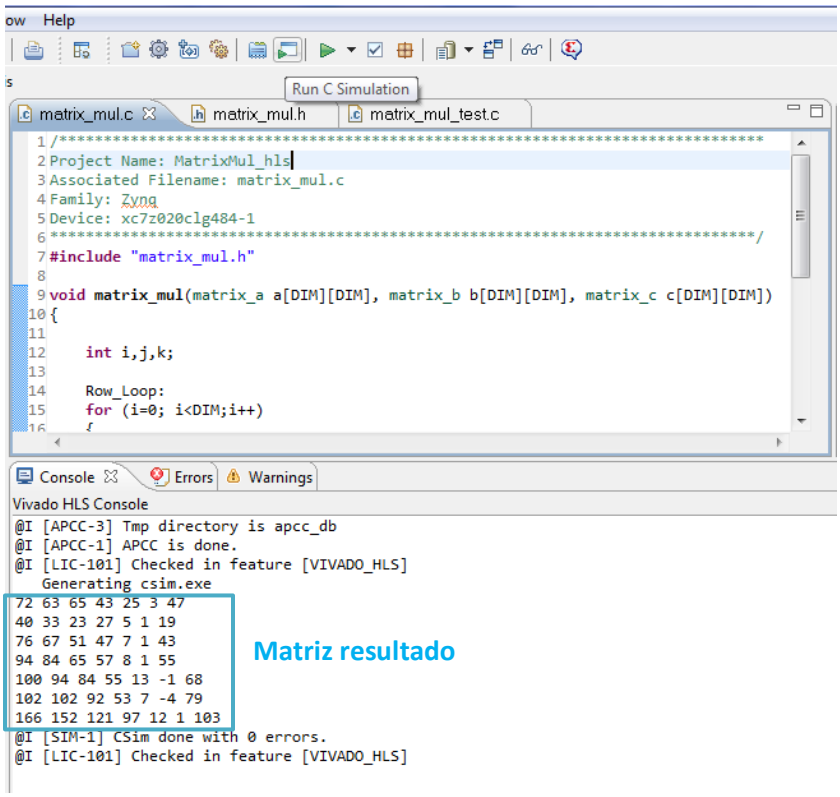


FIGURA 92. SIMULACIÓN DEL EJEMPLO MATRIXMUL_HLS

El siguiente paso es realizar la síntesis del diseño para observar la implementación que Vivado HLS genera con su comportamiento por defecto (sin directivas). El panel Directive correspondiente a esta solución tiene el aspecto de la Figura 93.

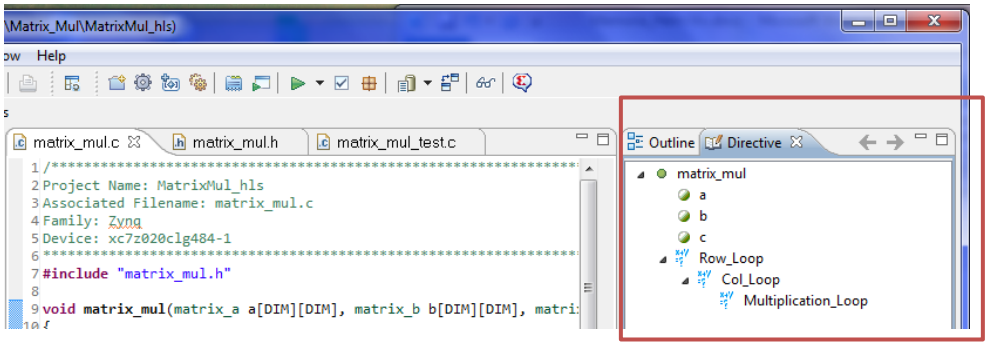
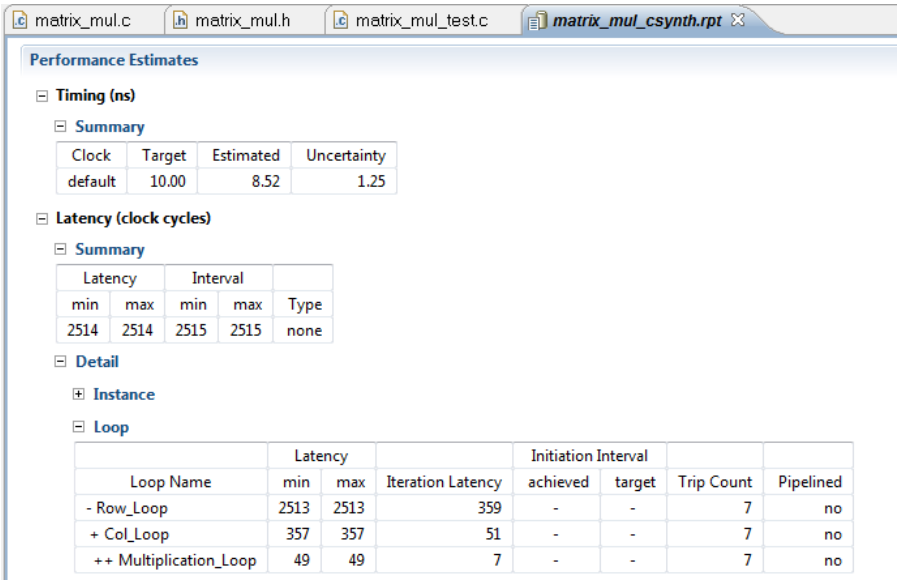


FIGURA 93. PANEL DIRECTIVES DE LA SOLUCIÓN I

Para sintetizar el diseño se selecciona en la barra de herramientas Run C Synthesis, situado al lado derecho de Run C Simulation. Al acabar la síntesis se abre automáticamente el informe *Synthesis Report* donde observar resultados de implementación.


matrix_mul.c | matrix_mul.h | matrix_mul_test.c | matrix_mul_csynth.rpt

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.52	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2514	2514	2515	2515	none

Detail

Instance

Loop

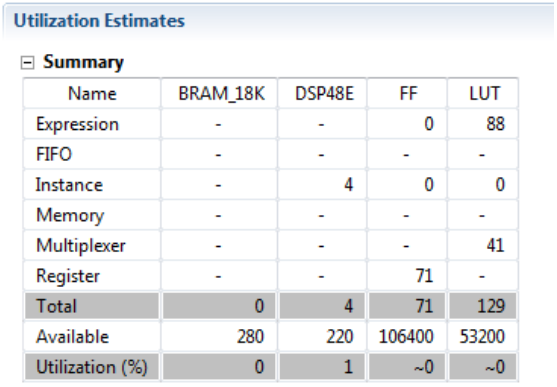
	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Loop	2513	2513	359	-	-	7	no
+ Col_Loop	357	357	51	-	-	7	no
++ Multiplication_Loop	49	49	7	-	-	7	no

FIGURA 94. RENDIMIENTO DE LA SOLUCIÓN I

En cuanto al rendimiento se observa que el periodo de reloj estimado es 8.52ns, que tiene una latencia de 2514 ciclos de reloj (consumidos a lo largo de los bucles anidados como se ve en el apartado Loop) y un intervalo de iniciación de 2515 ciclos. Esto quiere decir que la implementación realizada devuelve resultados tras 2514 ciclos y acepta nuevas entradas cada 2515 ciclos, lo cual son valores bastante altos. Además, el salir de los bucles anidados consume 1 ciclo más de reloj (finaliza Row_Loop con 2513 ciclos y se da la salida a los 2514 ciclos).

En cuanto a recursos empleados observando la Figura 95 se ve que utiliza 4 DSP48, 71 FFs y 129 LUTs, por lo que tiene un bajo área consumida.

Vivado HLS ha optado en este caso por una solución con bajo consumo de recursos a costa de una mayor latencia.


Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	88
FIFO	-	-	-	-
Instance	-	4	0	0
Memory	-	-	-	-
Multiplexer	-	-	-	41
Register	-	-	71	-
Total	0	4	71	129
Available	280	220	106400	53200
Utilization (%)	0	1	~0	~0

FIGURA 95. RECURSOS EN LA SOLUCIÓN I

En cuanto a los puertos del diseño RTL puede verse en el apartado Interface que ha incluido en el diseño las señales de reloj y reset (ap_clk y ap_rst), las señales ap_start, ap_done, ap_idle y ap_ready pertenecientes al protocolo ap_ctrl_hs que ha asignado por defecto al puerto return (ver Tabla 7) y para cada array a, b y c ha asignado por defecto el protocolo ap_memory que asigna a los arrays de entrada a y b señales de dirección (a_address0), chip-enable (a_ce0) y datos (a_q0) y al array de salida c señales de dirección (c_address0), chip-enable (c_ce0), write-enable (c_we0) y datos (c_d0). Además, los puertos de datos son de 32 bits como se había indicado en *matrix_mul.h*.

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matrix_mul	return value
ap_rst	in	1	ap_ctrl_hs	matrix_mul	return value
ap_start	in	1	ap_ctrl_hs	matrix_mul	return value
ap_done	out	1	ap_ctrl_hs	matrix_mul	return value
ap_idle	out	1	ap_ctrl_hs	matrix_mul	return value
ap_ready	out	1	ap_ctrl_hs	matrix_mul	return value
a_address0	out	6	ap_memory	a	array
a_ce0	out	1	ap_memory	a	array
a_q0	in	32	ap_memory	a	array
b_address0	out	6	ap_memory	b	array
b_ce0	out	1	ap_memory	b	array
b_q0	in	32	ap_memory	b	array
c_address0	out	6	ap_memory	c	array
c_ce0	out	1	ap_memory	c	array
c_we0	out	1	ap_memory	c	array
c_d0	out	32	ap_memory	c	array

FIGURA 96. PUERTOS DE LA SOLUCIÓN I

Tras estos resultados de síntesis se va a proceder a aplicar diferentes optimizaciones para buscar mejores soluciones de implementación. Todas las nuevas soluciones que se creen se harán escogiendo New Solution... en la barra de herramientas y aceptando que se copie la configuración de la solución 1.

Se va a comenzar aplicando la **directiva Unroll** para desenrollar los bucles y permitir así la compartición de los recursos hardware de forma que se acelere el procesado de muestras y se reduzca la latencia. Esta directiva puede ser aplicada a cada uno de los tres bucles o a todos ellos a la vez, pero ya de antemano puede decirse que el resultado es mejor segmentando el bucle más interno. Se verá por qué situando en diferentes soluciones la directiva Unroll en lugares diferentes (creando diferentes soluciones) y posteriormente comparando resultados mediante la opción *Compare Reports* de Vivado HLS.

En la solución 2 se sitúa la directiva Unroll al bucle más bajo en la jerarquía, Multiplication_Loop, por lo tanto en el panel de directivas se escoge *Insert Directive>Unroll* sobre este bucle como se ve en la Figura 97. Al tener marcado *Destination: Directive File* aparecerá incluida en este panel de directivas y en el fichero directives.tcl.

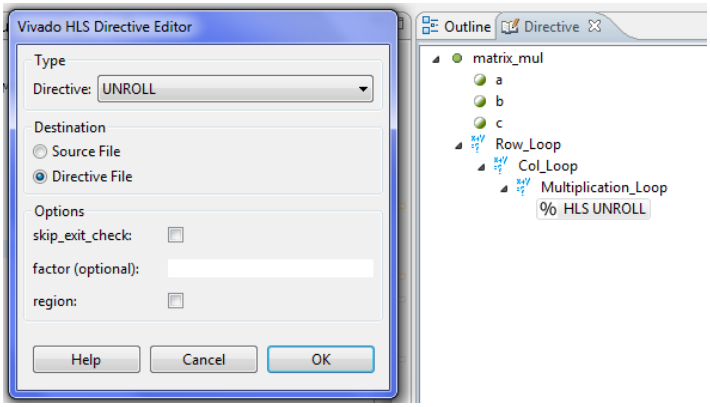


FIGURA 97. UNROLL MULTIPLICATION_LOOP EN LA SOLUCIÓN II

En la solución 3 se sitúa de la misma forma la directiva Unroll al bucle intermedio en la jerarquía, Col _Loop en vez de en el bucle más interno.

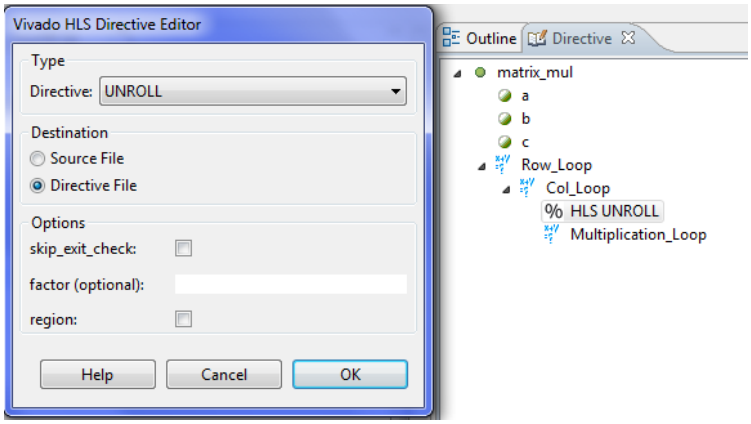


FIGURA 98. UNROLL COL_LOOP EN LA SOLUCIÓN III

En la solución 4 se sitúa la directiva Unroll al bucle superior en la jerarquía, Row _Loop.

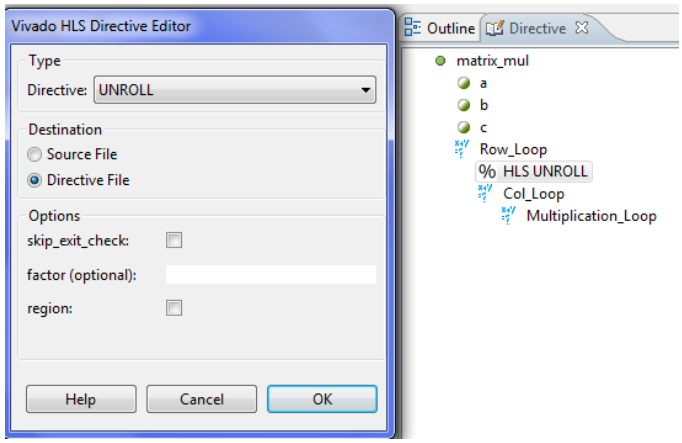


FIGURA 99. UNROLL ROW_LOOP EN LA SOLUCIÓN IV

Y finalmente, en la solución 5 se sitúa la directiva Unroll en los 3 bucles a la vez.

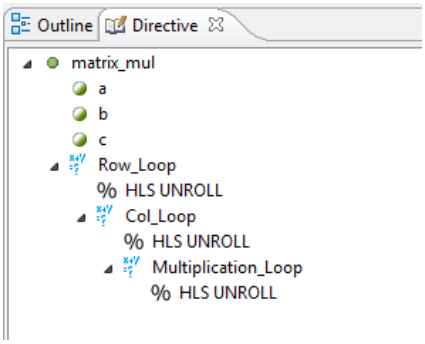


FIGURA 100. UNROLL DE TODOS LOS BUCLES EN LA SOLUCIÓN V

Ahora es momento de comparar el efecto que ha tenido en cada una de las posiciones. Para ello se encuentra la opción *Compare Reports* donde se eligen las 4 soluciones a comparar y muestra un informe de síntesis conjunto con los datos de cada una. Se va a desglosar ese informe:

Performance Estimates

Timing (ns)

Clock		solution2	solution3	solution4	solution5
default	Target	10.00	10.00	10.00	10.00
	Estimated	8.52	8.52	8.52	8.52

Latency (clock cycles)

		solution2	solution3	solution4	solution5
Latency	min	554	2458	2506	40
	max	554	2458	2506	40
Interval	min	555	2459	2507	41
	max	555	2459	2507	41

Utilization Estimates

	solution2	solution3	solution4	solution5
BRAM_18K	0	0	0	0
DSP48E	8	4	4	1356
FF	240	328	370	11302
LUT	265	828	843	6940

FIGURA 101. COMPARACIÓN DE SOLUCIONES

A partir de este informe se obtiene el siguiente gráfico de la Figura 102 que sirve de comparativa entre las soluciones.

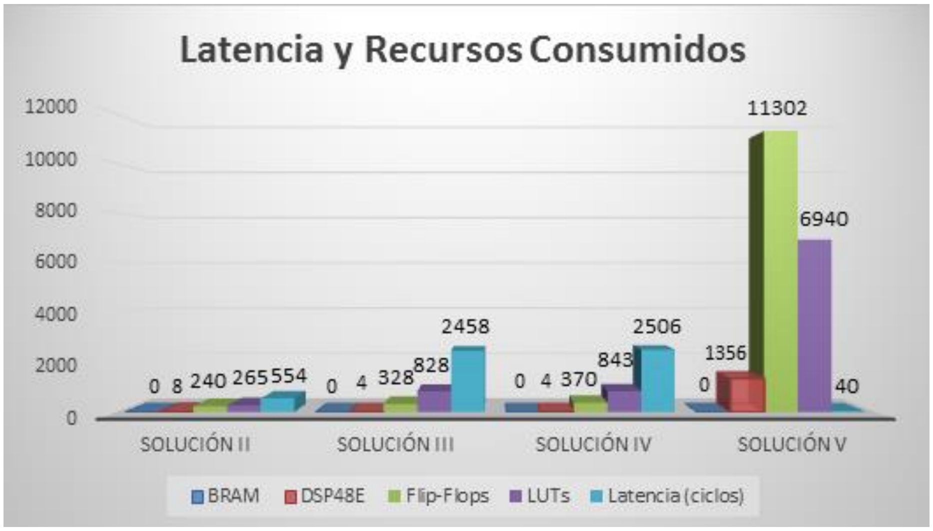


FIGURA 102. GRÁFICO COMPARATIVA LATENCIA Y RECURSOS CONSUMIDOS EN LAS SOLUCIONES II,III, IV Y V

A la vista del gráfico puede verse que las soluciones 2, 3 y 4 tienen un consumo similar de recursos, aunque la latencia es mayor en las soluciones 3 y 4 en que se ha aplicado Unroll al bucle Row_Loop o al bucle Col_Loop, y comparando con la solución 1 inicial (ver Figura 94) no suponen ninguna mejora de latencia. Esto último se debe a que al aplicar Unroll a un bucle que contiene dentro otro bucle lo que hace es crear tantos bucles internos como iteraciones tenga el bucle desenrollado. Entrando al informe de síntesis de la solución 4 (Unroll en Row_Loop) se comprueba este hecho:

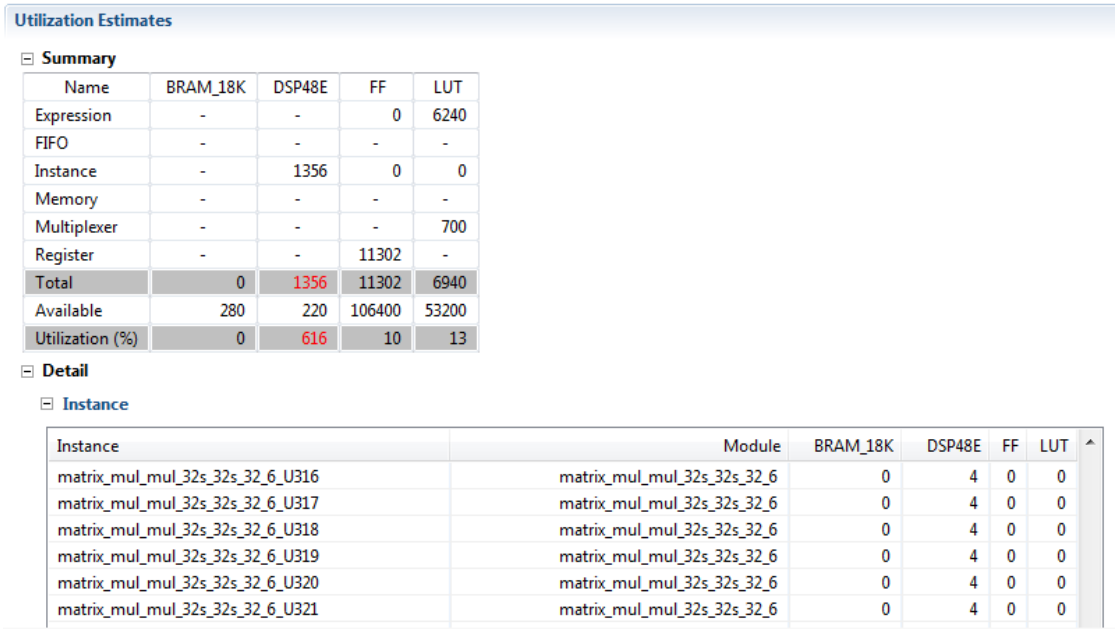
⊟ **Loop**

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Col_Loop	357	357	51	-	-	7	no
+ Multiplication_Loop	49	49	7	-	-	7	no
- Col_Loop	357	357	51	-	-	7	no
+ Multiplication_Loop	49	49	7	-	-	7	no
- Col_Loop	357	357	51	-	-	7	no
+ Multiplication_Loop	49	49	7	-	-	7	no
- Col_Loop	357	357	51	-	-	7	no
+ Multiplication_Loop	49	49	7	-	-	7	no
- Col_Loop	357	357	51	-	-	7	no
+ Multiplication_Loop	49	49	7	-	-	7	no
- Col_Loop	357	357	51	-	-	7	no
+ Multiplication_Loop	49	49	7	-	-	7	no
- Col_Loop	357	357	51	-	-	7	no
+ Multiplication_Loop	49	49	7	-	-	7	no

FIGURA 103. RESULTADO UNROLL BUCLE ROW_LOOP

Se han multiplicado por 7 los bucles Col_Loop y Multiplication_Loop que contenía, y de la misma forma sucede aplicando Unroll a Col_Loop, se multiplica por 7 el bucle Multiplication_Loop, por lo tanto no suponen ningún beneficio la solución 3 ni la solución 4.

Ahora en la solución 2 que aplica Unroll al bucle interno se tiene mayor latencia que en la solución 5 que aplica Unroll en todos los bucles, sin embargo se dijo que lo más óptimo era hacer lo que se ha hecho en la solución 2. Esto es así porque desenrollando todos los bucles como en la solución 5 dejan de existir bucles en el diseño y pueden realizarse todas las operaciones en paralelo, lo que implica un consumo de, en este caso, multiplicadores muy alto (7 multiplicadores por cada una de las 49 combinaciones fila-columna) y se consigue una latencia baja a costa de este uso elevado de recursos. Si se mira entonces el informe de síntesis de la solución 5 se ve que se superan los recursos disponibles en la tarjeta.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	6240
FIFO	-	-	-	-
Instance	-	1356	0	0
Memory	-	-	-	-
Multiplexer	-	-	-	700
Register	-	-	11302	-
Total	0	1356	11302	6940
Available	280	220	106400	53200
Utilization (%)	0	616	10	13

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
matrix_mul_mul_32s_32s_32_6_U316	matrix_mul_mul_32s_32s_32_6	0	4	0	0
matrix_mul_mul_32s_32s_32_6_U317	matrix_mul_mul_32s_32s_32_6	0	4	0	0
matrix_mul_mul_32s_32s_32_6_U318	matrix_mul_mul_32s_32s_32_6	0	4	0	0
matrix_mul_mul_32s_32s_32_6_U319	matrix_mul_mul_32s_32s_32_6	0	4	0	0
matrix_mul_mul_32s_32s_32_6_U320	matrix_mul_mul_32s_32s_32_6	0	4	0	0
matrix_mul_mul_32s_32s_32_6_U321	matrix_mul_mul_32s_32s_32_6	0	4	0	0

FIGURA 104. EXCESO DE RECURSOS EN LA SOLUCIÓN V

En la Figura 104 se observa que se tiene una utilización del 616% de las unidades DSP48E, ya que como se ve en Instances se tiene un multiplicador de 32x32bits (para variables tipo int) por cada multiplicación, consumiendo cada multiplicador 4 DSPs.

Por tanto, se confirma finalmente que la mejor implementación es desenrollar el bucle más interno de la jerarquía de bucles anidados, en este caso Multiplication_Loop, consiguiendo una mejora de latencia e intervalo y empleando más recursos de los que se empleaban para la solución 1 inicial pero siguen siendo un número bajo.

A partir de ahora las nuevas soluciones que se creen se harán partiendo de la implementación lograda en la solución 2, es decir, partiendo del diseño inicial con la directiva Unroll en Multiplication_Loop.

Con vistas a seguir mejorando la implementación óptima del diseño se va a ver la **directiva Array_Partition**. Vivado HLS ha implementado los arrays a, b y c por defecto como bloques de memoria RAM, estos bloquen limitan el rendimiento del diseño al tenerse que realizan lecturas y escrituras en ellos (las cuales consumen 2 ciclos cada una, uno para direccionar y otro para

leer/escribir) por lo que relantizan el diseño. Además, si se aplica la directiva Unroll a un bucle donde se suceden operaciones de lectura/escritura, se fuerza a realizar tantas operaciones de lectura/escritura de forma paralela en un mismo ciclo como iteraciones tenga el bucle, por lo que si el número de operaciones R/W es de 2 puede solventarse con RAMs de doble puerto pero con un factor mayor sería un hecho bloqueante y debe aplicarse la directiva Array_partition.

Así, se crea una nueva solución 6 donde ver el impacto de esta directiva.

La directiva Array_partition se aplica sobre arrays, especificando la dimensión del array sobre la que se quiere aplicar. En este ejemplo se tienen arrays de 2 dimensiones porque se trata de matrices, por lo que puede aplicarse sobre una de las dos dimensiones o sobre ambas. Se verán las ventajas que se obtienen de cada una de las dos formas.

Se aplica primero sobre ambas dimensiones de los 3 arrays del diseño, partiéndolos en registros que puedan operar de forma independiente.

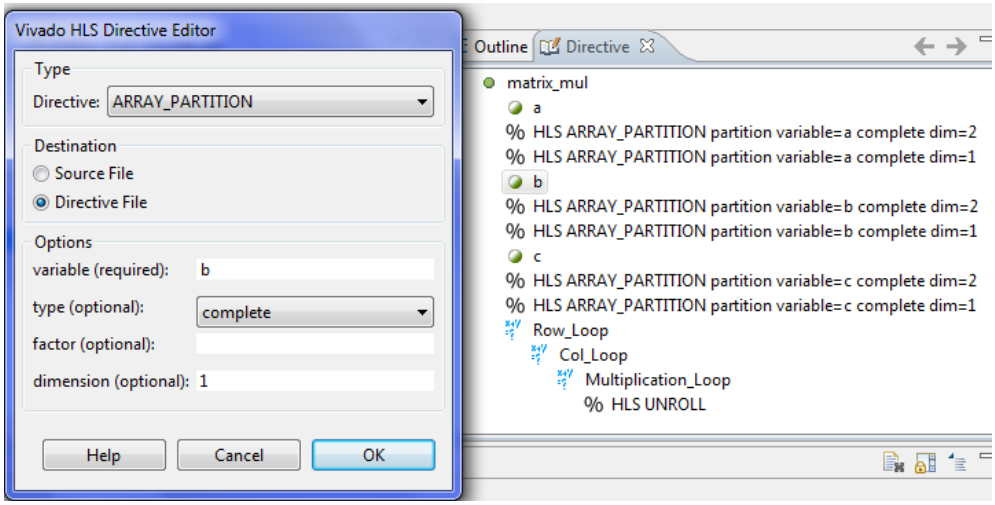


FIGURA 105. PARTICIÓN DE LAS 2 DIMENSIONES DE LOS ARRAYS EN LA SOLUCIÓN VI

Como resultado se observa que la latencia ha aumentado al aparecer multitud de registros y romper el orden que se tenía para operar con arrays, ya que esta medida de partición total de arrays sólo tiene beneficio si se va a aplicar segmentación (directiva Pipeline como se verá posteriormente).

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.52	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2122	2122	2123	2123	none

FIGURA 106. RENDIMIENTO DE LA SOLUCIÓN VI

El motivo por el que se ha querido ver este impacto es para observar la forma en que se implementan los puertos RTL como punteros ahora que no se tienen arrays, y los nuevos protocolos que asigna Vivado HLS a esos puertos.

Observando el apartado Interface puede verse que el puerto return de la función se mantiene implementado como se tenía por defecto, pero ahora el resto de puertos en los que se ha partido cada array se implementan como punteros de entrada o de salida.

Existen 49 puertos de entrada a, 49 puertos de entrada b y 49 puertos de salida c, uno por cada celda de las que componen las matrices.

En la Figura 107 se ve que el puerto return de la función se mantiene implementado como se tenía por defecto con el protocolo ap_ctrl_hs y las señales que conlleva, pero cada uno de los punteros de entrada de 32 bits a los que ha dado lugar el array a tienen ahora el protocolo ap_none como les corresponde por defecto a los punteros de entrada (ver Tabla 7), sin ninguna señal asociada al puerto.

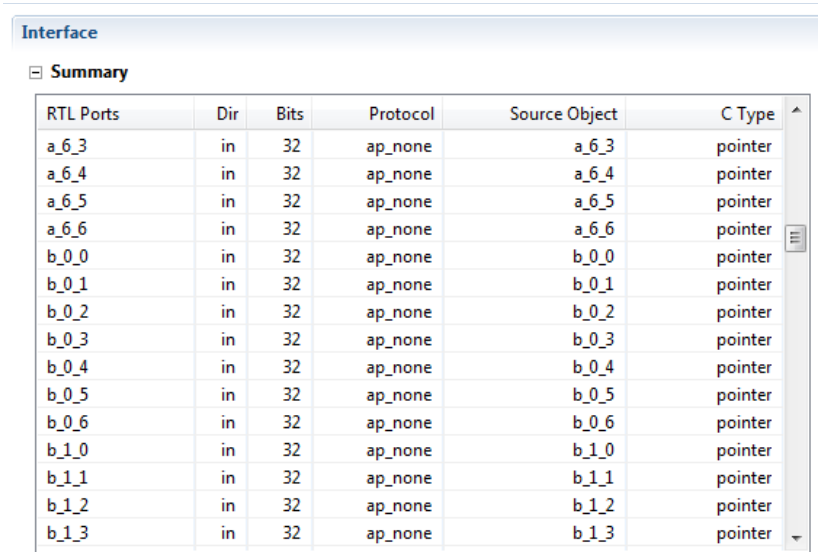
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matrix_mul	return value
ap_rst	in	1	ap_ctrl_hs	matrix_mul	return value
ap_start	in	1	ap_ctrl_hs	matrix_mul	return value
ap_done	out	1	ap_ctrl_hs	matrix_mul	return value
ap_idle	out	1	ap_ctrl_hs	matrix_mul	return value
ap_ready	out	1	ap_ctrl_hs	matrix_mul	return value
a_0_0	in	32	ap_none	a_0_0	pointer
a_0_1	in	32	ap_none	a_0_1	pointer
a_0_2	in	32	ap_none	a_0_2	pointer
a_0_3	in	32	ap_none	a_0_3	pointer
a_0_4	in	32	ap_none	a_0_4	pointer
a_0_5	in	32	ap_none	a_0_5	pointer
a_0_6	in	32	ap_none	a_0_6	pointer
a_1_0	in	32	ap_none	a_1_0	pointer
a_1_1	in	32	ap_none	a_1_1	pointer

FIGURA 107. PUERTOS RETURN Y A EN LA SOLUCIÓN VI

De igual forma, en la Figura 108 se ve que cada uno de los punteros de entrada de 32 bits a los que ha dado lugar el array b tienen ahora el protocolo ap_none, sin ninguna señal asociada al puerto.



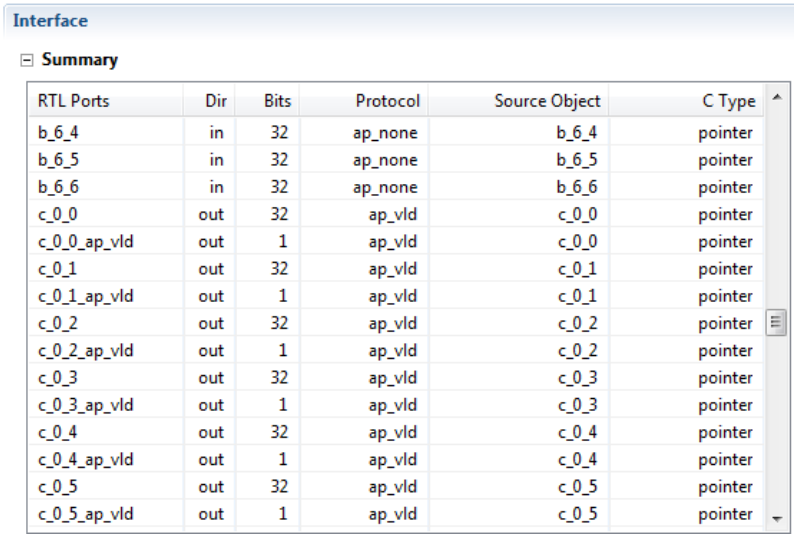
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
a_6_3	in	32	ap_none	a_6_3	pointer
a_6_4	in	32	ap_none	a_6_4	pointer
a_6_5	in	32	ap_none	a_6_5	pointer
a_6_6	in	32	ap_none	a_6_6	pointer
b_0_0	in	32	ap_none	b_0_0	pointer
b_0_1	in	32	ap_none	b_0_1	pointer
b_0_2	in	32	ap_none	b_0_2	pointer
b_0_3	in	32	ap_none	b_0_3	pointer
b_0_4	in	32	ap_none	b_0_4	pointer
b_0_5	in	32	ap_none	b_0_5	pointer
b_0_6	in	32	ap_none	b_0_6	pointer
b_1_0	in	32	ap_none	b_1_0	pointer
b_1_1	in	32	ap_none	b_1_1	pointer
b_1_2	in	32	ap_none	b_1_2	pointer
b_1_3	in	32	ap_none	b_1_3	pointer

FIGURA 108. PUERTOS A Y B EN LA SOLUCIÓN VI

Y por último, en la Figura 109 se ve que cada uno de los punteros de salida de 32 bits a los que ha dado lugar el array c tienen ahora el protocolo ap_vld con la señal de validez (c_fila_col_ap_vld) como les corresponde por defecto (ver Tabla 7).



Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
b_6_4	in	32	ap_none	b_6_4	pointer
b_6_5	in	32	ap_none	b_6_5	pointer
b_6_6	in	32	ap_none	b_6_6	pointer
c_0_0	out	32	ap_vld	c_0_0	pointer
c_0_0_ap_vld	out	1	ap_vld	c_0_0	pointer
c_0_1	out	32	ap_vld	c_0_1	pointer
c_0_1_ap_vld	out	1	ap_vld	c_0_1	pointer
c_0_2	out	32	ap_vld	c_0_2	pointer
c_0_2_ap_vld	out	1	ap_vld	c_0_2	pointer
c_0_3	out	32	ap_vld	c_0_3	pointer
c_0_3_ap_vld	out	1	ap_vld	c_0_3	pointer
c_0_4	out	32	ap_vld	c_0_4	pointer
c_0_4_ap_vld	out	1	ap_vld	c_0_4	pointer
c_0_5	out	32	ap_vld	c_0_5	pointer
c_0_5_ap_vld	out	1	ap_vld	c_0_5	pointer

FIGURA 109. PUERTOS B Y C EN LA SOLUCIÓN VI

A continuación se va a aprovechar que se está tratando el tema de protocolos asociados a puertos RTL para probar el uso de la **directiva Interface** mediante la cual se puede cambiar el protocolo de los puertos a, b y c. Para este ejemplo se crea una nueva solución donde lo que se va a hacer es

asociar el protocolo ap_ack a los puertos de entrada a, el protocolo ap_vld a los puertos de entrada b y el protocolo ap_hs a los puertos de salida c.

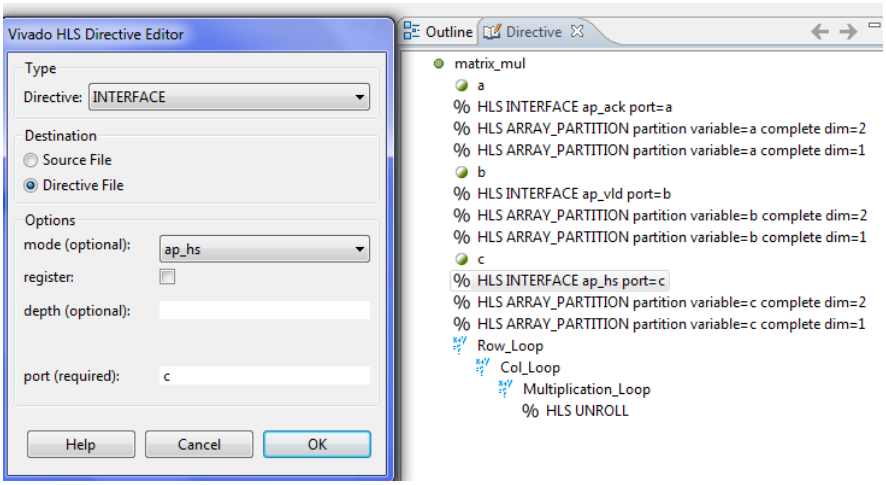


FIGURA 110. MODIFICACIÓN DE PUERTOS CON INTERFACE EN LA SOLUCIÓN VII

Sintetizando esta nueva solución se observa que los protocolos asociados a los puertos han cambiado, y que ahora incluyen para cada uno las señales asociadas a dichos protocolos.

Los puertos del array a incluyen además del puerto de datos la señal de salida ap_ack (a_fila_col_ap_ack) para contestar ante una petición poniendo a nivel alto la señal de asentimiento.

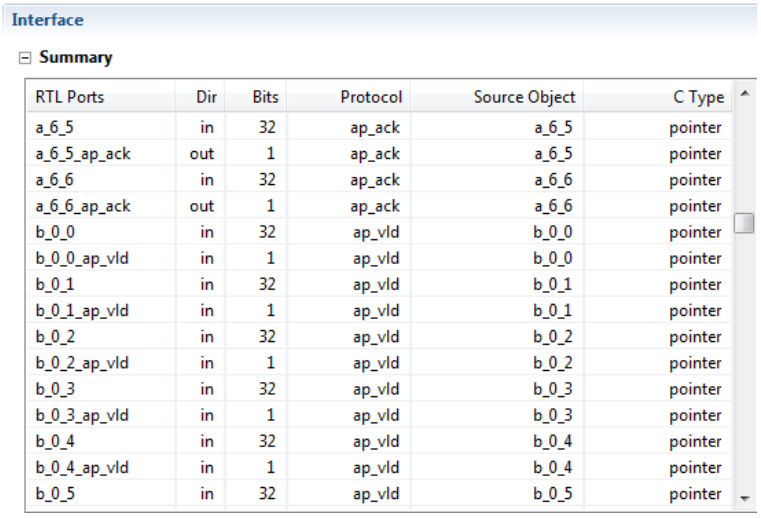
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matrix_mul	return value
ap_rst	in	1	ap_ctrl_hs	matrix_mul	return value
ap_start	in	1	ap_ctrl_hs	matrix_mul	return value
ap_done	out	1	ap_ctrl_hs	matrix_mul	return value
ap_idle	out	1	ap_ctrl_hs	matrix_mul	return value
ap_ready	out	1	ap_ctrl_hs	matrix_mul	return value
a_0_0	in	32	ap_ack	a_0_0	pointer
a_0_0_ap_ack	out	1	ap_ack	a_0_0	pointer
a_0_1	in	32	ap_ack	a_0_1	pointer
a_0_1_ap_ack	out	1	ap_ack	a_0_1	pointer
a_0_2	in	32	ap_ack	a_0_2	pointer
a_0_2_ap_ack	out	1	ap_ack	a_0_2	pointer
a_0_3	in	32	ap_ack	a_0_3	pointer
a_0_3_ap_ack	out	1	ap_ack	a_0_3	pointer
a_0_4	in	32	ap_ack	a_0_4	pointer

FIGURA 111. CAMBIO A PROTOCOLO AP_ACK EN LOS PUERTOS DADOS POR EL ARRAY A

Los puertos del array b incluyen además la señal de entrada ap_vld (b_fila_col_ap_vld) para indicar entrada válida poniendo a nivel alto la señal de validez.



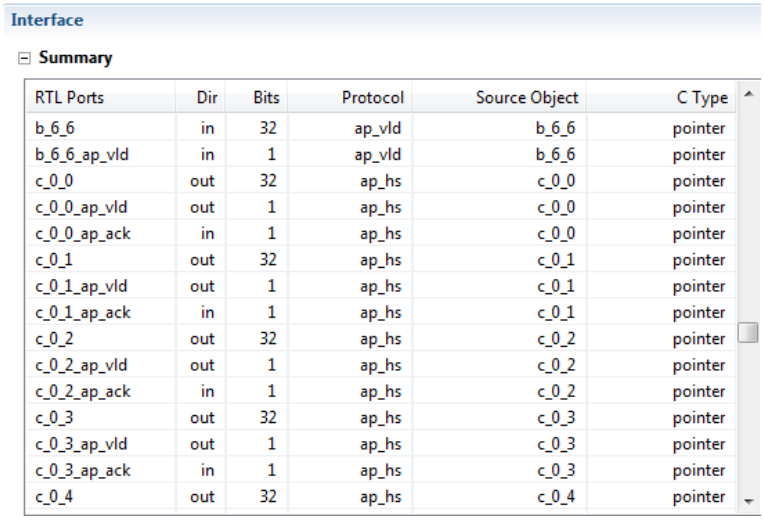
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
a_6_5	in	32	ap_ack	a_6_5	pointer
a_6_5_ap_ack	out	1	ap_ack	a_6_5	pointer
a_6_6	in	32	ap_ack	a_6_6	pointer
a_6_6_ap_ack	out	1	ap_ack	a_6_6	pointer
b_0_0	in	32	ap_vld	b_0_0	pointer
b_0_0_ap_vld	in	1	ap_vld	b_0_0	pointer
b_0_1	in	32	ap_vld	b_0_1	pointer
b_0_1_ap_vld	in	1	ap_vld	b_0_1	pointer
b_0_2	in	32	ap_vld	b_0_2	pointer
b_0_2_ap_vld	in	1	ap_vld	b_0_2	pointer
b_0_3	in	32	ap_vld	b_0_3	pointer
b_0_3_ap_vld	in	1	ap_vld	b_0_3	pointer
b_0_4	in	32	ap_vld	b_0_4	pointer
b_0_4_ap_vld	in	1	ap_vld	b_0_4	pointer
b_0_5	in	32	ap_vld	b_0_5	pointer

FIGURA 112. CAMBIO A PROTOCOLO AP_VLD EN LOS PUERTOS DADOS POR EL ARRAY B

Los puertos del array c incluyen ambas señales de supervisión, tanto la señal ap_ack (c_fila_col_ap_ack) de entrada para recibir asentimientos como la señal ap_vld (c_fila_col_ap_vld) de salida para indicar salida válida.



Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
b_6_6	in	32	ap_vld	b_6_6	pointer
b_6_6_ap_vld	in	1	ap_vld	b_6_6	pointer
c_0_0	out	32	ap_hs	c_0_0	pointer
c_0_0_ap_vld	out	1	ap_hs	c_0_0	pointer
c_0_0_ap_ack	in	1	ap_hs	c_0_0	pointer
c_0_1	out	32	ap_hs	c_0_1	pointer
c_0_1_ap_vld	out	1	ap_hs	c_0_1	pointer
c_0_1_ap_ack	in	1	ap_hs	c_0_1	pointer
c_0_2	out	32	ap_hs	c_0_2	pointer
c_0_2_ap_vld	out	1	ap_hs	c_0_2	pointer
c_0_2_ap_ack	in	1	ap_hs	c_0_2	pointer
c_0_3	out	32	ap_hs	c_0_3	pointer
c_0_3_ap_vld	out	1	ap_hs	c_0_3	pointer
c_0_3_ap_ack	in	1	ap_hs	c_0_3	pointer
c_0_4	out	32	ap_hs	c_0_4	pointer

FIGURA 113. CAMBIO A PROTOCOLO AP_HS EN LOS PUERTOS DADOS POR EL ARRAY C

Ahora, continuando con la directiva Array_partition, se va a ver el otro caso posible de partición de arrays en una única dimensión. El motivo por el que se quiere ver esta aplicación es para demostrar que también puede ser útil separar arrays bidimensionales en arrays unidimensionales, lo que en este caso ayuda a tener una menor latencia ya que se sigue manteniendo el orden en las operaciones y no provoca un consumo elevado de recursos como sucedía al particionar al completo el array.

En una nueva solución se va a hacer partición de la dimensión 2 para el array a y de la dimensión 1 para el array b, de forma que pueda realizarse un mayor número de operaciones en paralelo entre vectores fila de a y vectores columna de b.

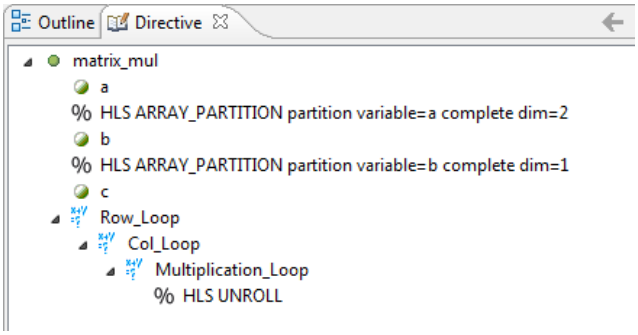


FIGURA 114. PARTICIÓN DE ARRAYS BIDIMENSIONALES EN UNIDIMENSIONALES EN LA SOLUCIÓN VIII

Como se esperaba se observa un beneficio en la latencia como se ve en la Figura 115 debido a que se facilita el orden de operación manteniendo las operaciones por vectores y a su vez permite realizar multiplicaciones entre vector fila y vector columna más rápidamente. Esta solución sería más óptima si no se piensa aplicar segmentación que la solución que realiza el particionado completo de arrays.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.52	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
407	407	408	408	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Loop	406	406	58	-	-	7	no
+ Col_Loop	56	56	8	-	-	7	no

FIGURA 115. RENDIMIENTO EN LA SOLUCIÓN VIII

Lo que se quiere ver también en esta Figura 115 es que el bucle Col_Loop tiene una latencia de iteración que se encuentra ya al mínimo número de ciclos posible, un ciclo por cada una de las 7 iteraciones y otro ciclo que se consume en la entrada al bucle. Esto viene bien para explicar ahora el funcionamiento de la **directiva Latency**.

Se crea una nueva solución donde probar esta directiva. Con ella se comentaba que se podía fijar la latencia de un bucle o de una función a un mínimo o a un máximo. En este caso ya se encuentra a lo mínimo que se puede conseguir con las directivas que están incluidas, por lo que si se fijara un

mínimo menor de 8 ciclos no tendría efecto. De igual forma, si se fijase un máximo igual o superior a 8 ciclos, Vivado HLS seguiría mostrando una latencia de iteración de 8 ciclos porque pone su esfuerzo en conseguir lo mejor que pueda. Por tanto, para ver su efecto se va a fijar una latencia de iteración en el bucle Col_Loop de mínimo 20, es decir, aumentar la latencia por iteración.

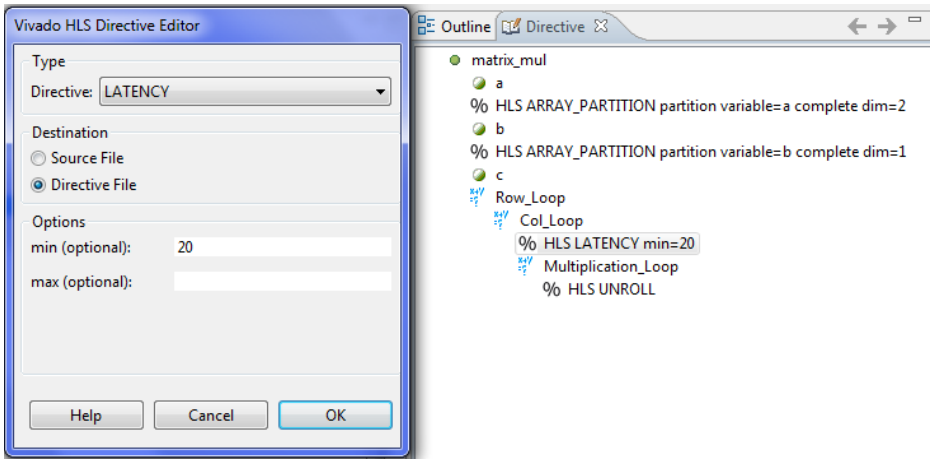


FIGURA 116. MODIFICAR LATENCY DE UN BUCLE EN LA SOLUCIÓN IX

El resultado del rendimiento que se obtiene es el siguiente:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.52	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1044	1044	1045	1045	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Loop	1043	1043	149	-	-	7	no
+ Col_Loop	147	147	21	-	-	7	no

FIGURA 117. RENDIMIENTO EN LA SOLUCIÓN IX

Se ha forzado a que las iteraciones consuman un mayor número de ciclos y tanto las latencias de bucles como la latencia del diseño han aumentado.

Otra directiva que resulta útil al trabajar con bucles anidados es la **directiva Loop_Flatten**. Se había dicho que lleva un ciclo de reloj entrar a un bucle y otro ciclo salir. Si se tienen bucles anidados sin ninguna operación entre ellos estos ciclos relentizan el diseño. Para eso sirve esta directiva, elimina esos ciclos de transición conjuntando bucles entre los que no existen operaciones en un único bucle.

Aplicando esta directiva al bucle Col_Loop se fusiona con el bucle Row_Loop.

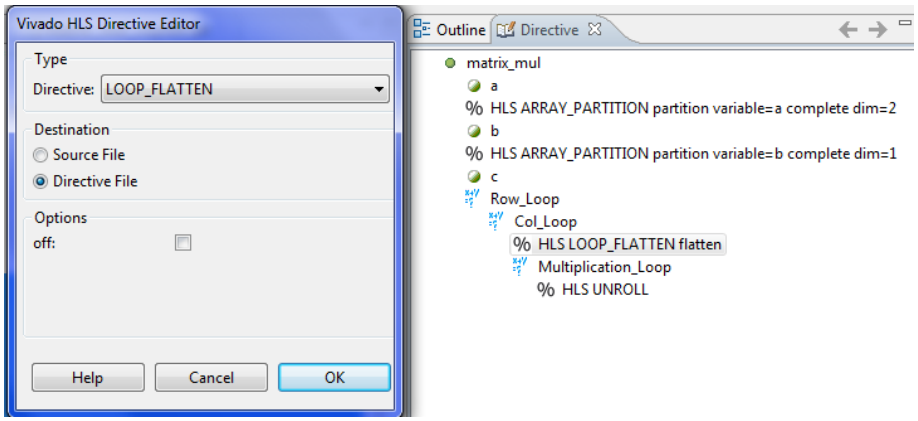


FIGURA 118. FUSIÓN DE BUCLES EN LA SOLUCIÓN X

Puede verse el resultado desde el informe de síntesis. Se han fusionado ambos bucles en uno único llamado Row_Loop_Col_Loop y así se ha conseguido reducir la latencia con respecto a la que se tenía sin incluir esta directiva (solución 8, ver Figura 115). No obstante es una reducción pequeña, de 14 ciclos correspondientes a los dos ciclos que constaba entrar y salir del bucle Col_Loop interno para cada una de las 7 iteraciones.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.52	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
393	393	394	394	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Loop_Col_Loop	392	392	8	-	-	49	no

FIGURA 119. RENDIMIENTO EN LA SOLUCIÓN X

Finalmente, es el momento de hablar de la **directiva Pipeline** que realiza la segmentación de tareas forzando a que se asigne tantos recursos como sea necesario para poder ejecutar en paralelo tantas operaciones como sea posible, y tan pronto como éstas hayan liberado recursos, se reutilicen para las transacciones posteriores. Lo ideal sería poder conseguir un intervalo de iniciación II=1 indicando que se procesa una muestra nueva por cada ciclo de reloj.

En una nueva solución se va a aplicar esta directiva al bucle fusionado Row_Loop_Col_Loop como se tenía en la solución anterior.

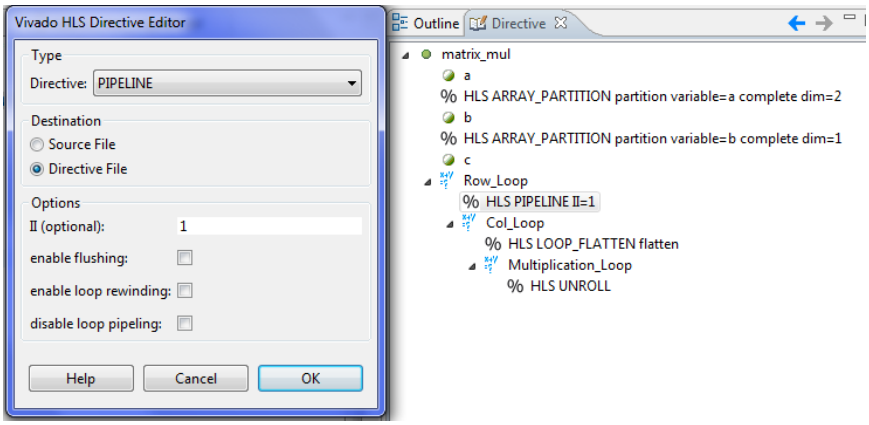


FIGURA 120. SEGMENTACIÓN DE BUCLES EN LA SOLUCIÓN XI

El resultado que se obtiene al sintetizar (ver Figura 121) es que no es posible lograr un intervalo de iniciación 1 (target, solicitado) en el bucle conjunto, sino que se tiene un intervalo de 4 (achieved, logrado), aunque sí se ha reducido considerablemente la latencia y el intervalo del diseño, y además la consola muestra warnings indicando que en ocasiones no es posible planificar algunas operaciones de lectura o escritura porque el número de puertos con que acceder a los arrays unidimensionales es limitado. Es decir, si no se parten los arrays al completo para manejarlos como puertos independientes la directiva Pipeline se ve limitada por operaciones R/W.

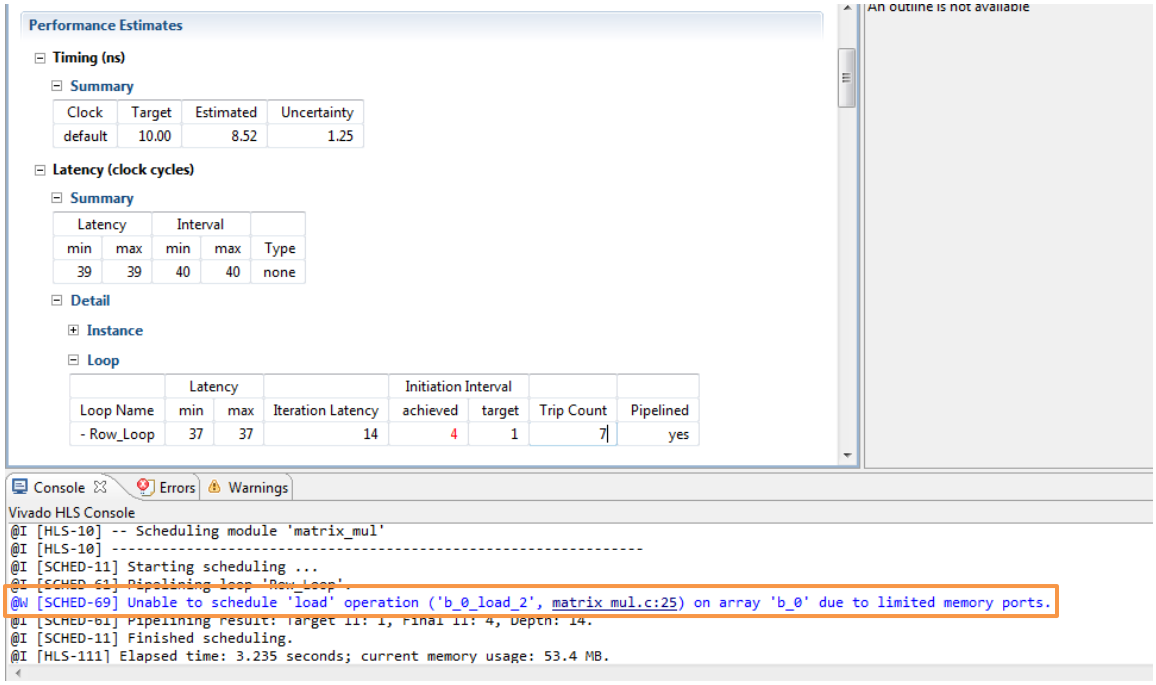


FIGURA 121. RENDIMIENTO EN LA SOLUCIÓN XI

Otra opción que se quiere ver de la directiva Pipeline es la opción rewind. Como se ha visto hasta ahora la latencia del diseño, o de la función top-level, siempre tiene un ciclo de reloj más que la latencia del bucle que contiene, a pesar de contener únicamente este bucle. Esto es así porque cuando la función principal se empieza a ejecutar, como no hay operaciones lógicas antes del

bucle Row_Loop_Col_Loop empieza éste a ejecutarse inmediatamente, pero cuando termina por defecto conlleva un ciclo de reloj la acción de salir del bucle.

En este ciclo de reloj no sucede ninguna operación, y en diseños en los que se realizan consecutivas llamadas a la función principal supone un ciclo desaprovechado. Para ello, la opción rewind indica a la síntesis que el bucle en sí es la función top-level, que al terminar él habrá terminado la función principal y evitar así este ciclo vacío (ver Figura 89). Se va a aplicar esta opción a la solución actual marcando *enable loop rewinding.*

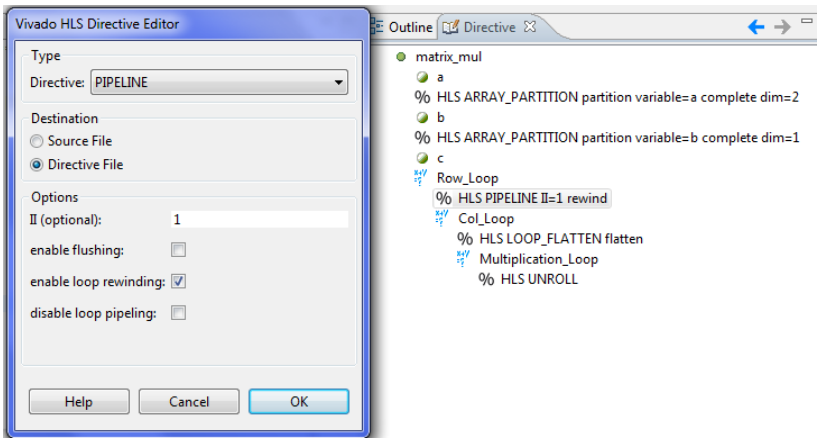


FIGURA 122. APLICAR OPCIÓN REWIND EN LA SOLUCIÓN XI

El resultado que se obtiene con esta opción es el siguiente:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.52	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
37	38	28	28	function

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Loop	37	37	14	4	1	7	yes

FIGURA 123. RENDIMIENTO CON LA OPCIÓN REWIND EN LA SOLUCIÓN XI

Con esta opción finalmente se ve que la latencia del bucle fusionado y con segmentación coincide con la latencia del diseño completo, 37 ciclos de reloj.

Ahora, dada la limitación que suponía para la directiva Pipeline el tener en el diseño arrays, en una nueva solución se va a usar la directiva Array_partition para partir ambas dimensiones en los tres arrays y se aplicará Pipeline directamente a la función principal, lo que conlleva que automáticamente sea aplicada la función Unroll a los bucles que componen el diseño.

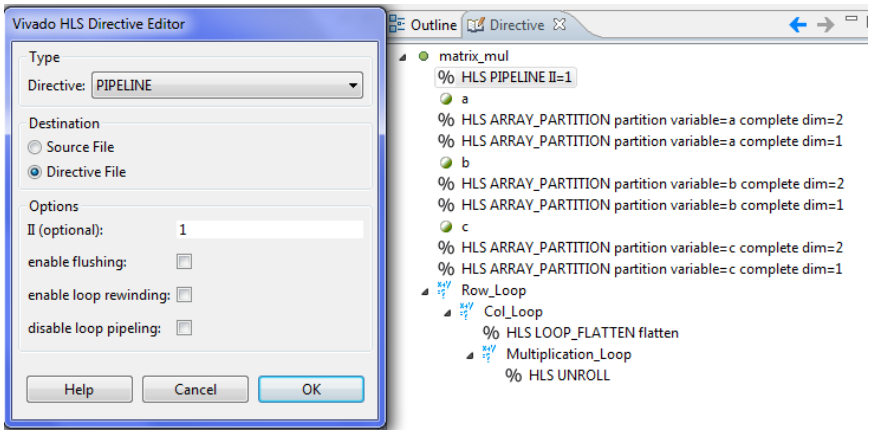


FIGURA 124. PARTICIONADO DE ARRAYS Y SEGMENTACIÓN EN LA SOLUCIÓN XII

Si se observa el rendimiento en el informe de síntesis se ve que el intervalo de iniciación se ha conseguido que sea de 1 ciclo de reloj y que se tenga una latencia de 6 ciclos. Son valores bastante buenos y hasta ahora es la mejor implementación en rendimiento.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.52	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
6	6	1	1	function

Detail

Instance

Loop

N/A

FIGURA 125. RENDIMIENTO EN LA SOLUCIÓN XII

Sin embargo, si se observan los recursos utilizados se ve que sobrepasan los recursos hardware que dispone la tarjeta debido a la gran cantidad de multiplicaciones que se suceden, por lo que no es una implementación realizable.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	6272
FIFO	-	-	-	-
Instance	-	1372	0	0
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	7847	-
Total	0	1372	7847	6272
Available	280	220	106400	53200
Utilization (%)	0	623	7	11

FIGURA 126. RECURSOS HARDWARE EN LA SOLUCIÓN XII

De forma gráfica se puede ver este aumento de recursos que supone la solución 12 frente a la solución 11 al desenrollar todos los bucles y segmentar la función principal.

FIGURA 127. GRÁFICO COMPARATIVA DE LATENCIA Y RECURSOS CONSUMIDOS EN LAS SOLUCIONES XI Y XII

Se requieren multiplicadores grandes, de 32 bits ya que se está trabajando con variables de tipo entero, que utilizarán cada 4 DSP48E (ver Figura 104) y además gran cantidad de ellos para multiplicar cada factor fila-columna como ya se comentaba al aplicar la directiva Unroll a todos los bucles en la solución 5.

Vista la limitación que ha supuesto trabajar con variables de tipo int se comprende la ventaja que ofrece Vivado HLS al permitir trabajar con tipos de datos enteros con **precisión de bit arbitraria**. Esto permite que las variables del diseño tengan el ancho de bit que necesitan y no se desaprovechen recursos que en realidad no son necesarios.

Para ello, hay que modificar el fichero de cabecera, *matrix_mul.h*, para incluir la cabecera que soporta este tipo de datos, *ap_cint.h*, que ya la trae Vivado HLS incluida, y cambiar el tipo de dato de las matrices de int a int<w>.

A la vista de la inicialización de matrices que se tiene en el testbench se ve que para las matrices a y b es suficiente con 5 bits ya que los valores que contienen se encuentran en el rango $\{-2^4,+2^4\}$, y

para el array c que contiene valores en el rango {-4, +166} se necesitan 9 bits, por lo que el nuevo fichero *matrix_mul.h* tiene el siguiente aspecto:

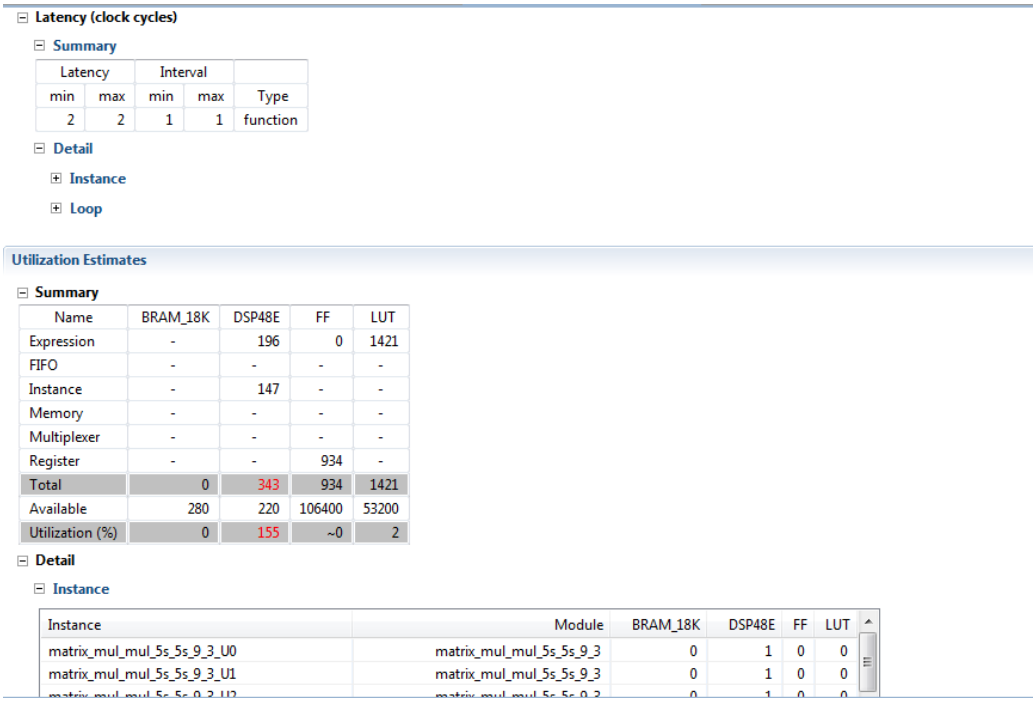
```
/*************************************************************************
Project Name: MatrixMul_hls
Associated Filename: matrix_mul.h
Family: Zynq
Device: xc7z020clg484-1
*************************************************************************/

#define DIM  7
#include "ap_cint.h"

typedef int5 matrix_a;
typedef int5 matrix_b;
typedef int9 matrix_c;


void matrix_mul (matrix_a[DIM][DIM], matrix_b[DIM][DIM], matrix_c[DIM][DIM]);
```

Con ello lo que se espera es tener una reducción en el área consumida. Realizando la síntesis con el panel de directivas tal cual se tenía en la solución 12 se observa que efectivamente se requieren menos DSPs (1 DSP48E por cada multiplicador) ya que ahora con uno de ellos basta para realizar las operaciones y además de una latencia del diseño de apenas 2 ciclos, pero sigue suponiendo mayor cantidad de recursos de los que la tarjeta puede soportar.



Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2	2	1	1	function

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	196	0	1421
FIFO	-	-	-	-
Instance	-	147	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	934	-
Total	0	343	934	1421
Available	280	220	106400	53200
Utilization (%)	0	155	~0	2

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
matrix_mul_mul_5s_5s_9_3_U0	matrix_mul_mul_5s_5s_9_3	0	1	0	0
matrix_mul_mul_5s_5s_9_3_U1	matrix_mul_mul_5s_5s_9_3	0	1	0	0

FIGURA 128. RESULTADOS CON ANCHO DE BIT ARBITRARIO EN LA SOLUCIÓN XIII

Hasta aquí se ha analizado el impacto en el diseño que tiene aplicar diferentes directivas y el lugar y momento en que aplicarlas. Otras directivas que aparecen en la Tabla 5 como son la directiva

Dataflow y la directiva Inline no han podido mostrarse con este ejemplo de código ya que su aplicación esta condicionada a que existan funciones dependientes de otras o llamadas a funciones en el código.

Ahora que se ha visto cómo funcionan se va a realizar el diseño que era objetivo de este apartado, diseñar un periférico en Vivado HLS.

3.2.2. **Periférico diseñado por el usuario con Vivado HLS**

El objetivo de este apartado es mostrar al usuario cómo puede sacar partido a las ventajas que ofrece Vivado HLS para la creación de un periférico, que en este caso se tratará de un filtro FIR, y la forma de comunicarlo con el microprocesador de la tarjeta Zynq-7000 [29].

Filtro FIR

Un filtro FIR es un tipo de filtro digital de respuesta al impulso finita (*Finite Impulse Response*), cuya respuesta ante una señal impulso de entrada es un número finito de términos no nulos a su salida. Esta salida es combinación lineal de las entradas actuales y pasadas [35].

El procesado interno del filtro consiste en la multiplicación de la muestra de entrada actual y de muestras anteriores por unos coeficientes que definen el tipo de respuesta del filtro, y finalmente la suma de todas estas multiplicaciones constituye la salida en el instante actual [35]. Puede expresarse esta relación con la Ecuación 1:

$$y[n] = a_0 \cdot x[n] + a_1 \cdot x[n - 1] + a_2 \cdot x[n - 2] + \dots + a_{N-1} \cdot x[n - (N-1)] + a_N \cdot x[n - N]; \text{ [38]}$$

ECUACIÓN 1. ECUACIÓN DE SALIDA DE UN FILTRO FIR

siendo y[n] la muestra de salida actual, x[n] la muestra de entrada actual, $x[n - i]$ $i=1,\dots,N$ las muestras en instantes anteriores y los factores $a_0,a_1,\dots,a_N$ los coeficientes del filtro.

Este tipo de filtro presenta la ventaja de que pueden diseñarse para tener una respuesta de fase lineal (distorsión de fase nula), de que son estables dado a que no poseen realimentación, y de su facilidad de implementación. Por el contrario, son filtros con mayor complejidad computacional, ya que requieren un mayor número de coeficientes para satisfacer ciertas especificaciones (como bandas de transición estrechas), lo que se traduce en mayor consumo de memoria, mayor número de operaciones y mayor tiempo de procesamiento [39].

Existen distintos métodos para diseñar este tipo de filtros, como el método de las ventanas, muestreo en frecuencia o rizado constante [38].

Mediante el método de las ventanas se obtienen los coeficientes de la respuesta al impulso enventanados como la multiplicación de la respuesta al impulso ideal por una ventana, expresado por la Ecuación 2:

$$h(n) = h_D(n) \cdot w(n); \qquad 0 \leq n \leq N \quad [1]$$

ECUACIÓN 2. RESPUESTA AL IMPULSO DE UN FILTRO FIR

Las ventanas más habituales son la ventana rectangular, ventana de Barlett, ventana de Hanning, ventana de Hamming, ventana de Blackman y ventana de Kaiser [35].

En el diseño que se va a llevar a cabo se ha escogido realizar un filtro FIR paso bajo de orden 20 empleando el método de la ventana y utilizando una ventana Hamming. El motivo por el que se ha escogido esta ventana es por comodidad a la hora de emplear las funciones de filtrado de las que dispone Matlab para realizar las comprobaciones necesarias, ya que todas ellas trabajan con este tipo de ventana por defecto. A su vez, el número de coeficientes se ha escogido bajo para no consumir demasiados recursos hardware, pues éste ha sido uno de los inconvenientes que han surgido a lo largo del diseño, como se explicará en detalle posteriormente en el apartado referente a otras alternativas de diseño. No obstante, la elección de la ventana, y por tanto los coeficientes resultantes, condiciona la respuesta del filtro pero no afecta al objetivo principal de este ejercicio que es diseñar el funcionamiento del filtro, por tanto se deja libertad al usuario para repetir el experimento con los parámetros de diseño que crea convenientes.

Estas especificaciones de diseño se situarán en la herramienta fdatool de Matlab para diseño de filtros. Para este diseño se han seleccionado, como se comentaba, las siguientes opciones de la Figura 129.

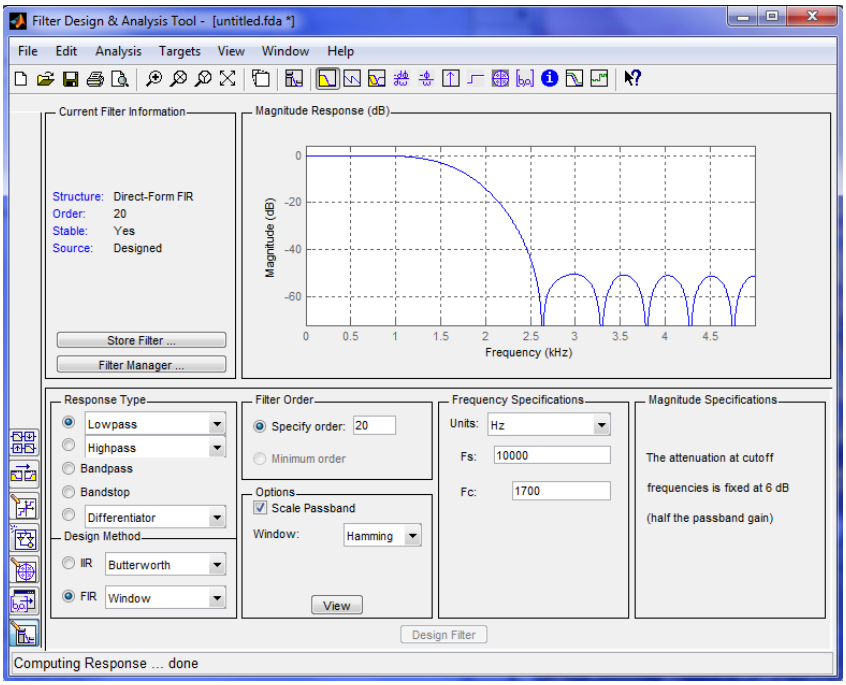


FIGURA 129. REALIZACIÓN DEL FILTRO CON LA HERRAMIENTA FDATOOL DE MATLAB

En la Figura 129 puede verse la respuesta del filtro diseñado. Dicha herramienta permite también ver otras características como la respuesta al impulso, el diagrama de polos y ceros, el coste de su implementación en recursos, o los coeficientes generados, los cuales se exportarán al espacio de trabajo desde *File>Export* recogiéndose en un array de 1x(N+1) valores de tipo double, siendo N el orden del filtro.

A partir de este array se obtendrán los coeficientes a introducir en el diseño del filtro en Vivado HLS. Un hecho a tener en cuenta es que estos coeficientes son de tipo double y en el diseño se trabajará con datos de tipo entero, por lo que previamente es necesario analizar el tamaño de los datos a emplear en el diseño C y convertir el rango {-1,+1} de los coeficientes de Matlab en el rango de la variable que recoja los coeficientes en el diseño.

Diseño del filtro – Vivado HLS

ESTRUCTURA INTERNA

Antes de comenzar a diseñar el filtro es necesario comprender cuál es su funcionamiento interno, es decir, cómo realiza el procesado de datos de entrada para proporcionar una salida en cada instante de tiempo.

La Figura 130 muestra un diagrama de bloques de este proceso [35], el cual el diseñador debe traducir a código para Vivado HLS.

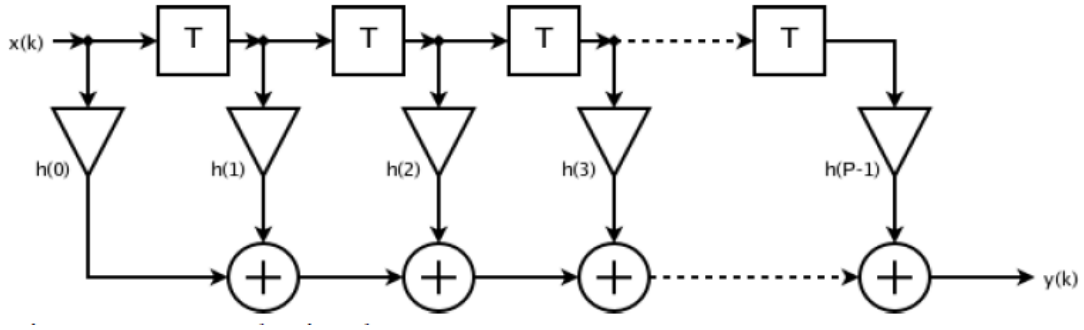


FIGURA 130. DIAGRAMA DE BLOQUES DE UN FILTRO FIR

Para implementarlo se tiene un array con los coeficientes que caracterizan el filtro y un registro de desplazamiento, donde se irán almacenando las muestras presentes y pasadas de la señal de entrada, ambos de tamaño *"P"* (según la Figura 130), siendo P el número de coeficientes del filtro. Así, en cada ejecución del bucle acumulador, se desplazan las muestras pasadas del registro paralelo una posición, entrando a dicho registro la muestra de señal actual (*x(k)*), y se multiplica

cada posición del registro por el coeficiente correspondiente a esa posición, acumulándose estas *"P"* multiplicaciones en una variable que al final bucle es asignada al valor de salida (*y(k)*).

PARÁMETROS DE DISEÑO

La función principal que se creará en Vivado HLS para constituir la descripción C del filtro va a tener como argumentos la entrada y la salida del filtro en el instante actual de tiempo.

Por otro lado, en el cuerpo de la función, se define un registro de desplazamiento, el array de coeficientes, una variable para acumulación y las variables internas que sean necesarias.

Para este diseño será necesario conocer de antemano:

- La señal de entrada que se quiere tener, o en su defecto las muestras que se quieren entregar al filtro. Para este ejemplo de diseño se ha escogido una señal sencilla que puede representarse con muestras enteras como es una señal rampa ascendente. Esta señal, en el contexto de Vivado HLS, será proporcionada en la llamada a la función principal desde el *test bench*.

- Los coeficientes del filtro, ya que éstos irán prefijados en el cuerpo de la función principal. Como se ha comentado anteriormente, estos coeficientes no son exactamente los exportados por la función fdatool de Matlab, sino que han sido convertidos al rango de la variable que recoge los coeficientes en el diseño, y redondeados a un valor entero, ya que Vivado HLS trabaja principalmente con tipos enteros y además proporciona cabeceras para escoger el tamaño adecuado y optimizar así el diseño.

OBTENCIÓN DE COEFICIENTES

Como ya se comentó, mediante la herramienta fdatool de Matlab se fijan las especificaciones del diseño, en este caso se quiere realizar un filtro FIR paso bajo de orden 20 empleando el método de la ventana y utilizando una ventana Hamming. Una vez diseñado se exportan los coeficientes al *workspace* de Matlab en un array, por defecto denominado *"Num"*, de 21 coeficientes de tipo double con valores en el rango {-0.0492, 0.3407}.

En la función del filtro se definirá un array de coeficientes de tipo int7, es decir, datos enteros con un ancho de 7 bits. El porqué de esta elección se contará más en detalle una vez diseñado el filtro en Vivado HLS, pero está basado en una solución de compromiso entre el error que existe en la salida real obtenida mediante Matlab y la salida ofrecida por el filtro diseñado, y la cantidad de recursos hardware empleados en el diseño.

Por tanto, para un array de coeficientes con 7 bits de ancho, los coeficientes de Matlab deben convertirse del rango {-1, +1} al rango {-64, +63}, y esto puede hacerse directamente desde la

ventana de comandos de Matlab multiplicando por 2^6. El nuevo array quedará multiplicado muestra a muestra, alcanzando ahora un rango {-3.1499, + 21.8040} con datos tipo double como se tenía originalmente.

El último paso es aplicar redondeo a estos coeficientes con la función *round* para tener finalmente el array buscado de enteros en el rango de trabajo de Vivado HLS.

La ventana de comandos y el espacio de trabajo de Matlab puede verse en la Figura 131.

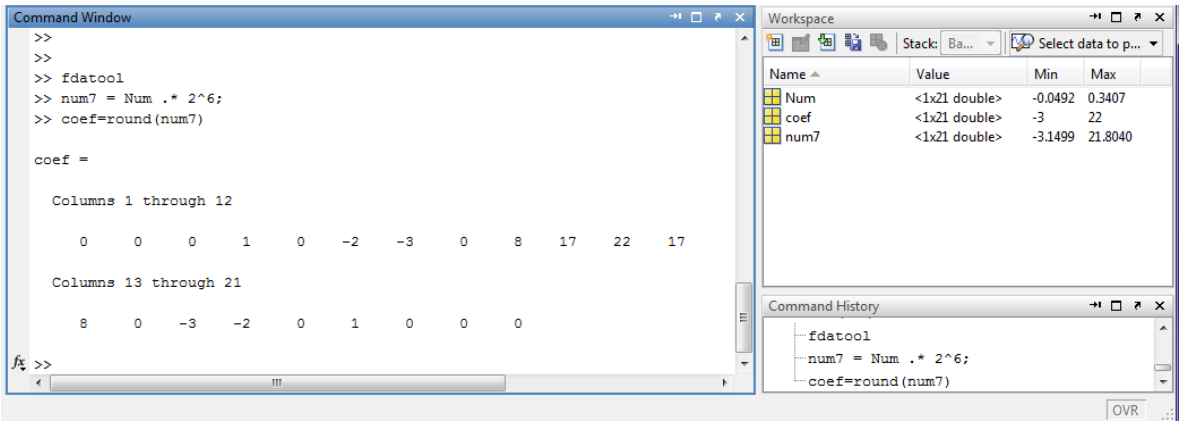


FIGURA 131. COMANDOS APLICADOS EN MATLAB PARA LA OBTENCIÓN DE COEFICIENTES

CÓDIGO C DE LOS FICHEROS A EMPLEAR EN VIVADO HLS

Al hablar de Vivado HLS se comentó que esta herramienta tiene como entrada principal una función (*top-function*) que puede estar escrita en C, C++ o System C, y además incluir un *test bench* y sus ficheros asociados.

Para el periférico a diseñar se va a trabajar en lenguaje C, y en cuanto a la función para el filtro, su cabecera y el *test bench* asociado, existen dos opciones de incluirlas en un proyecto en Vivado HLS: disponer de los ficheros *.c y *.h ya creados para importarlos a un nuevo proyecto, o comenzar a crearlos dentro del nuevo proyecto.

En este caso, se ha optado por la primera opción, ya que puede resultar más útil al usuario conocer en qué lugar deben importarse los ficheros y cuáles de ellos son necesarios para componer el diseño. No obstante, si el usuario lo prefiere, puede crearse un nuevo proyecto y desde el explorador de proyecto en las secciones *Source* y *Test Bench* abrir el menú con el botón derecho y escoger *New File* para abrir un fichero en blanco sobre el que escribir el código correspondiente.

Los ficheros a emplear son los siguientes:

- Función principal, *hls_fir.c*: Contiene el código necesario para el procesado de cada muestra de entrada, realizando las operaciones correspondientes al diagrama de bloques de la Figura 130 y proporcionando una salida.

- Cabecera, *hls_fir.h*: Contiene la definición de tipos para cada variable a emplear en la función principal, así como la definición de dicha función.

- Test bench, *hls_fir_test.c*: Contiene el código correspondiente a un *test bench* con propiedad de self checking. Este *test bench*, recogido bajo la función main, llama a la función principal pasándole como argumento cada muestra de la señal de entrada, almacena en un fichero externo *"out.dat"* los resultados devueltos por el filtro, y una vez finalizado el procesado de todas las muestras, verifica su funcionamiento comparando este fichero con otro fichero externo creado por el usuario que contenga las salidas correctas.

- Fichero asociado al *test bench*, *out.valid.dat*: Contiene las muestras de salida correctas para el filtro que se ha diseñado. Su función es permitir que el *test bench* tenga la propiedad de ser self checking, comparando out.dat y out.valid.dat y retornando un valor 0 si las muestras coinciden.

Con el propósito de ofrecer una visión más clara al usuario del contenido de los ficheros anteriormente nombrados y de cómo están relacionados entre ellos, se ha considerado útil incluir el código de dichos ficheros.

Nota.: Este código es el que se emplea en el diseño final a falta de las directivas de optimización, cuya aplicación se detallará posteriormente.

```
/*********************************************************************
Project Name: fir_prj
Associated Filename: hls_fir.c
Family: Zynq
Device: xc7z020clg484-1
*********************************************************************/

#include "hls_fir.h"


void hls_fir (dataout_t *y, datain_t x)
{
  static datain_t shift_reg[N];
  coef_t c[N]={0,0,0,1,0,-2,-3,0,8,17,22,17,8,0,-3,-2,0,1,0,0,0};
  acc_t acc;
  datain_t data;
  int i;
```

```
  acc=0;


Shift_Accum_Loop:

  for (i=N-1;i>=0;i--)
  {

       if (i==0){
              shift_reg[0]=x;
              data = x;
       }
       else{
              shift_reg[i]=shift_reg[i-1];
              data=shift_reg[i];
       }

       acc+=data*c[i];
  }
  *y=acc;
}
```

Este código corresponde al fichero *hls_fir.c* que contiene la función top-level, *hls_fir*, en la cual se definen las variables a emplear: el registro de desplazamiento *shift_reg*, el array de coeficientes *c* de dimensión N, la variable de acumulación *acc* y la variable de recogida de datos *data*.

Además se puede ver que a esta función principal se le pasará como argumento desde el testbench una entrada *x* que es una variable escalar, y el testbench recogerá el el contenido del puntero de salida *y* una vez haya finalizado la función.

```
/************************************************************************
Project Name: fir_prj
Associated Filename: hls_fir.h
Family: Zynq
Device: xc7z020clg484-1
************************************************************************/

#include "ap_cint.h"
#define N    21

typedef int7       coef_t;
typedef int8       datain_t;
typedef int13      dataout_t;
typedef int13      acc_t;

void hls_fir (dataout_t *y, datain_t x);
```

Este código corresponde al fichero de cabecera hls_fir.h que contiene la definición de la variable N para el número de coeficientes, el include con la cabecera que permite hacer uso de un ancho de bit arbitrario en variables de tipo entero, ap_cint.h, y la definición de tipos int<w> con el ancho justo necesario para cada variable del diseño.

También se encuentra situada la definición de la función top-level, *hls_fir*.

```
/************************************************************************
Project Name: fir_prj
Associated Filename: hls_fir_test.c
Family: Zynq
Device: xc7z020clg484-1
************************************************************************/

#include <stdio.h>
#include <math.h>
#include "hls_fir.h"

int main ()
{
  const int SAMPLES=70;

  datain_t signal;
  dataout_t output;

  FILE *fp;
  int i;
  signal = 0;

  fp=fopen("out.dat","w");

  for (i=0;i<SAMPLES;i++)
  {
       signal = signal + 1; // Ramp signal.

    // Execute the function with latest input.
    hls_fir(&output,signal);
    // Save results.
    fprintf(fp,"%d ",output);
  }

  fclose(fp);


 // Self-ckecking property.
  printf ("Comparing against output data...\n");
```

```
  if (system("diff -w out.dat out.valid.dat"))
  {
       fprintf(stdout, "*********************************************\n");
       fprintf(stdout, "FAIL: Output DOES NOT match the correct output :( \n");
       fprintf(stdout, "*********************************************\n");
        return 1;
  }
  else
  {
       fprintf(stdout, "*******************************************\n");
       fprintf(stdout, "PASS: Output MATCHES the correct output! :) \n");
       fprintf(stdout, "*******************************************\n");
        return 0;
  }


}
```

Este código corresponde al fichero de testbench, *hls_fir_test.h*, donde se inicializan las variables del diseño tales como el número de muestras de entrada, *SAMPLES*, las variables de entrada y de salida, *signal* y *output*, y un puntero a ficheros para almacenar la salida en un fichero externo, *fp*.

A continuación se abre ese fichero para escribir con la sentencia `fp=fopen("out.dat","w")` y se comienza a mandar muestras de señal en cada llamada a la función, recogiendo el contenido del puntero de salida *y* en la variable output que se almacena en el fichero con la sentencia `fprintf(fp,"%d ",output)` y finalmente se cierra el fichero con `fclose(fp).`

Finalmente tiene lugar la propiedad del testbench de ser self-cheking. Gracias al uso de ficheros se ha podido almacenar la salida en el archivo out.dat, y su vez se tiene la salida válida en el archivo out.valid.dat, por lo que el testbench comprueba con la sentencia `if (system("diff -w out.dat out.valid.dat"))` si los archivos son diferentes y si no es así devuelve un mensaje de éxito por la consola a través de la función *fprintf*.

CREACIÓN DEL PROYECTO, SIMULACIÓN Y ANÁLISIS DE RESULTADOS TRAS LA SÍNTESIS

Una vez se tienen estos ficheros generados, es momento de crear un nuevo proyecto en Vivado HLS que permita analizar el comportamiento del filtro diseñado, validar su funcionamiento, y añadir directivas para controlar el proceso de síntesis en busca de la implementación más óptima del diseño.

Para ello, se abre Vivado HLS 2014.1 y se escoge en la pantalla de bienvenida *Create New Project*.

Aparecerá un asistente que guiará al usuario durante la creación, comenzando por especificar nombre y directorio donde se crearán todos los archivos del proyecto, que deberá coincidir con el directorio donde se encuentren los ficheros a importar.

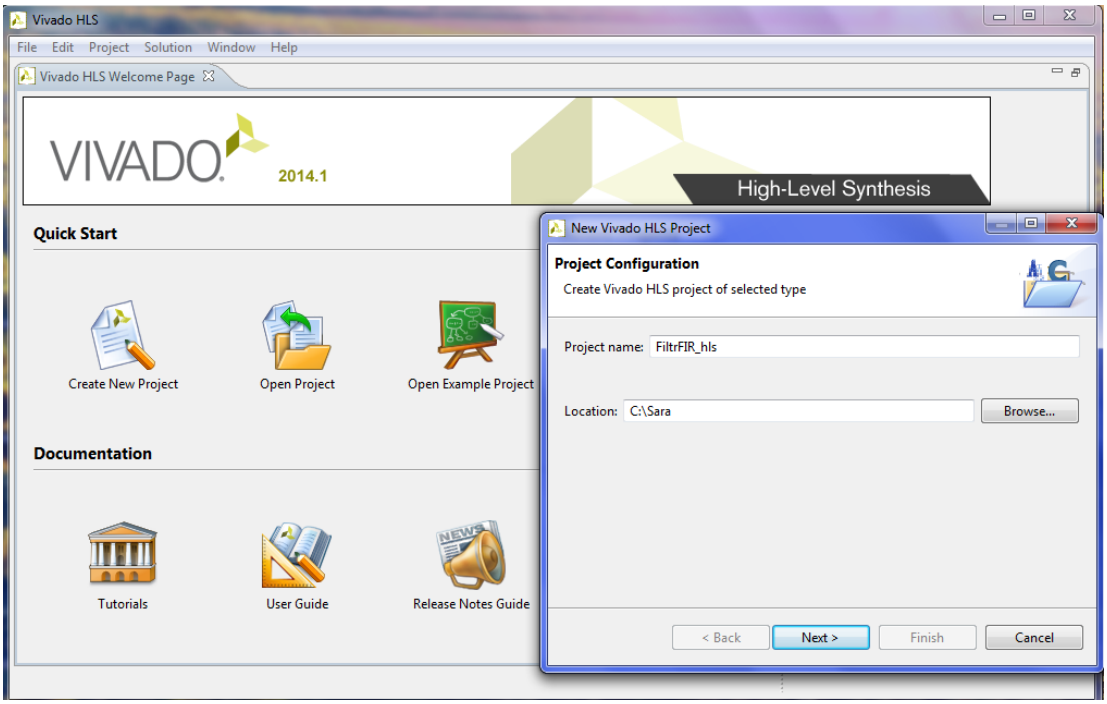


FIGURA 132. NUEVO PROYECTO EN VIVADO HLS - NOMBRE Y DIRECTORIO

A partir de aquí, el asistente pedirá los ficheros que van a constituir el proyecto. Primero pregunta por el nombre de la función principal, hls_fir, y su fichero fuente asociado, momento en el cual debe localizarse con *Add Files…* el fichero *hls_fir.c* e incluirlo en el proyecto. Por su parte, el fichero de cabecera asociado, *hls_fir.h*, se incluirá automáticamente siempre que se encuentre contenido en el directorio del proyecto.

A su vez, si existieran más ficheros fuente para el diseño deberían ser incluidos en este momento.

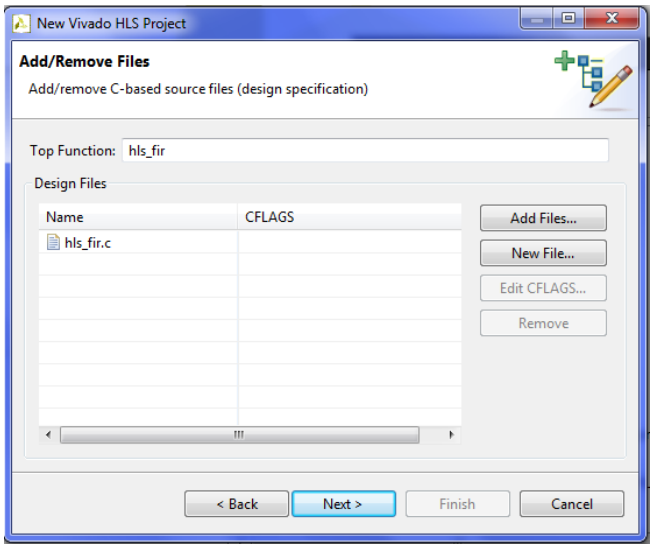


FIGURA 133. NUEVO PROYECTO EN VIVADO HLS - AÑADIR FICHERO FUENTE

Al hacer clic en *Next*, el asistente continúa preguntando por los ficheros C del proyecto, pero esta vez referidos al *test bench*, por lo que debe localizarse con *Add Files…* el fichero *hls_fir_test.c* e incluirlo en el proyecto. Además, es necesario incluir el archivo *out.valid.dat*, leído por el *test bench* para la validación de la salida del filtro.

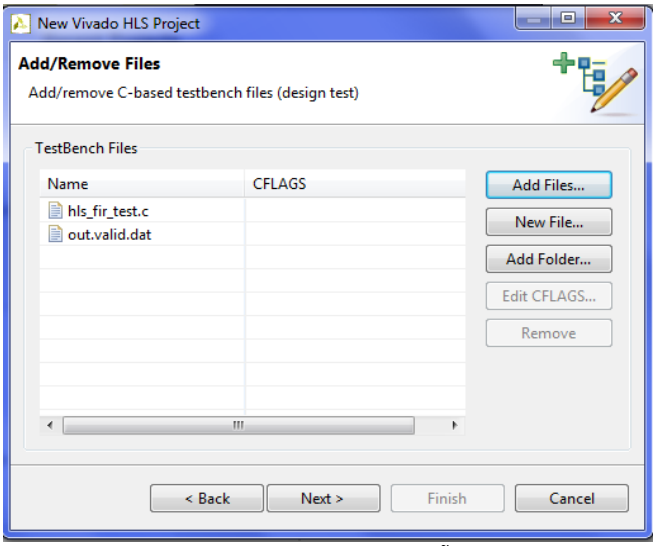


FIGURA 134. NUEVO PROYECTO EN VIVADO HLS - AÑADIR FICHEROS PARA EL TESTBENCH

Finalmente, se debe seleccionar el nombre de la solución/implementación a analizar, el periodo de reloj, por defecto de 10ns, el margen de incertidumbre (que si no se especifica es, por defecto, el 12.5% del periodo de reloj, 1.25ns) y la tarjeta donde se va a implementar, Zynq ZC702 Evaluation Platform.

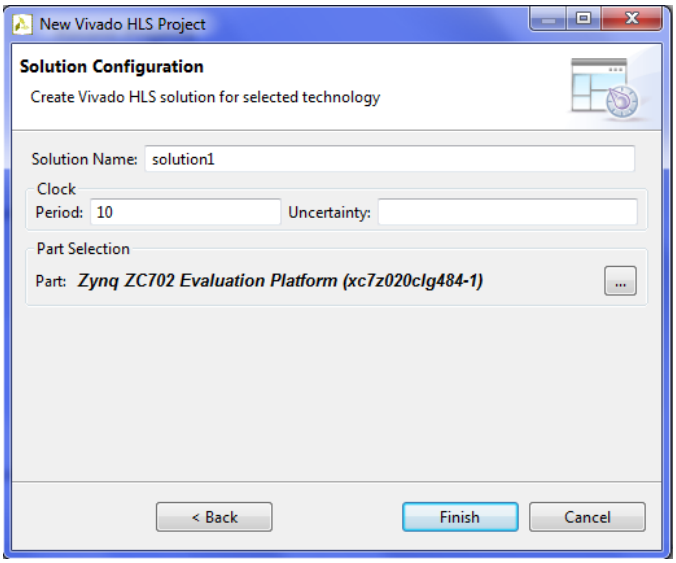


FIGURA 135. NUEVO PROYECTO EN VIVADO HLS - PARÁMETROS DE LA PRIMERA SOLUCIÓN

Al finalizar se muestra la pantalla de trabajo principal bajo la perspectiva Síntesis, y puede comprobarse en el explorador de proyecto cómo los ficheros importados aparecen en sus correspondientes secciones.

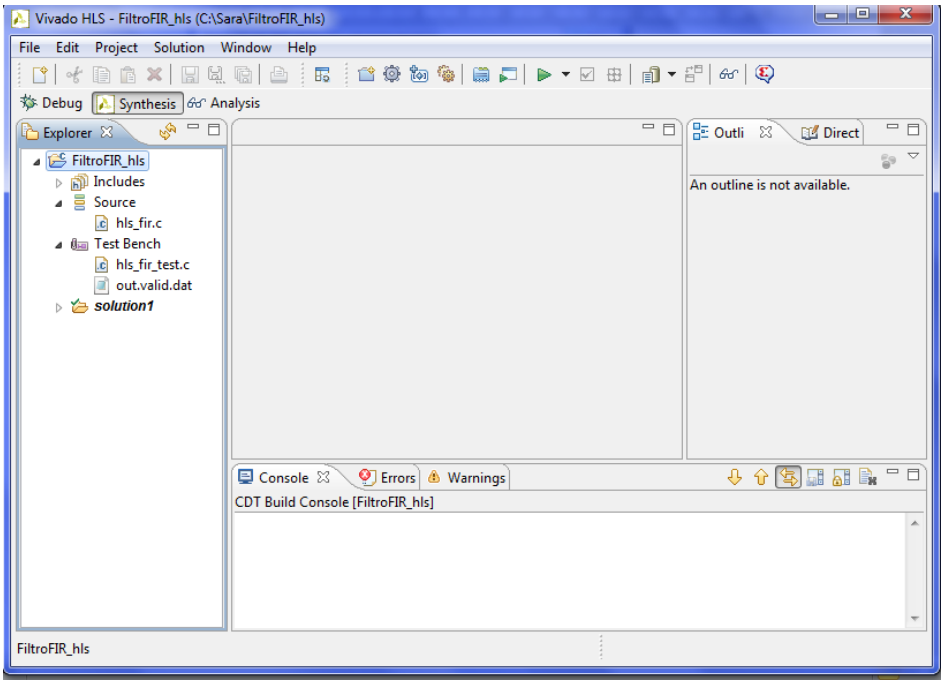


FIGURA 136. INTERFAZ DE USUARIO PARA SÍNTESIS EN VIVADO HLS

El siguiente paso es realizar la simulación del diseño, como paso previo a la síntesis, para verificar si el código C realizado para el filtro responde de forma adecuada a las muestras de entrada que se van introduciendo.

Puede simularse el diseño desde el botón situado en la barra de herramientas denominado *Run C Simulation*. Aparecerá la ventana de la Figura 137, donde por el momento se dejan todas las opciones sin marcar y se selecciona *OK* para compilar y ejecutar el diseño.

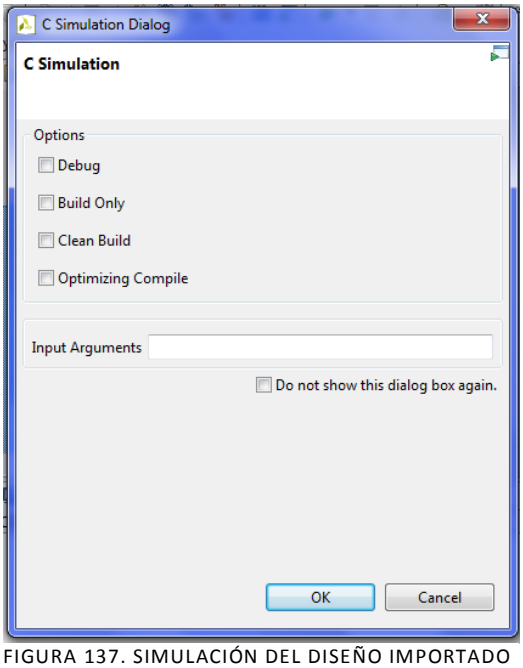


FIGURA 137. SIMULACIÓN DEL DISEÑO IMPORTADO

En la consola pueden verse los mensajes producidos durante la simulación, terminando con el mensaje devuelto por el *test bench* indicando si los resultados coinciden o no con los esperados. Además, puede observarse que, en el panel de explorador de proyecto, se ha añadido la carpeta *csim* con información referente a la simulación, donde puede encontrarse el fichero *out.dat* creado por el *test bench*.

Si la simulación falla, es conveniente revisar el código y marcar la opción *Debug* en el diálogo de *Run C Simulation* para cambiar a la perspectiva de depuración, donde resultará más fácil al usuario corregir el problema. Si, por el contrario, finaliza con el siguiente mensaje de éxito, todo estará en orden y se puede continuar con la síntesis.

```
Comparing against output data...
*******************************************
PASS: Output MATCHES the correct output! :)
*******************************************
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

FIGURA 138. MENSAJE DE ÉXITO RETORNADO POR EL TESTBENCH TRAS LA SIMULACIÓN

Una vez simulado el código y verificado que los resultados coinciden con los esperados, se continúa realizando la síntesis del diseño C en diseño RTL. Para ello, en la barra de herramientas se encuentra el botón *Run C Synthesis*, el cual comienza directamente el proceso mostrando mensajes en la consola. De igual forma que para la simulación, la síntesis también crea su propia carpeta de ficheros en el explorador de proyecto bajo el nombre *syn*.

Lo importante de la síntesis es el informe de resultados que muestra al finalizar, bajo el nombre de *hls_fir_csynth.rpt*. Las secciones contenidas en este informe son ya familiares para el lector puesto que se comentaron en el apartado de *Metodología de trabajo*. Para esta solución pueden extraerse entonces los siguientes datos:

- Tiempo
  - Periodo de reloj fijado para la tarjeta: 10ns.
  - Periodo estimado: 6.38ns.
  - Margen temporal: 1.25ns.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	6.38	1.25

FIGURA 139. SYNTHESIS REPORT (I)

- Ciclos de reloj

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
85	85	86	86	none

Detail

Instance

N/A

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	84	84	4	-	-	21	no

FIGURA 140. SYNTHESIS REPORT (II)

  - Latencia de la función: 85 ciclos. Ofrece una salida cada 85 ciclos. Corresponde a la latencia del bucle más el ciclo que conlleva salir del bucle. No cuenta también un ciclo de entrada porque al ejecutarse la función comienza inmediatamente el bucle.
    >>Latencia del bucle: 84 ciclos. (Correspondiente a las 21 iteraciones).
    >>>>Latencia por iteración: 4 ciclos.

  - Intervalo de iniciación: 86 ciclos. Puede aceptar una nueva entrada cada 86 ciclos de reloj. Necesita un ciclo más después de que los resultados sean escritos (latencia)

debido a que el diseño no está segmentado (*pipelined type=none*) y debe esperar a que termine completamente la transacción actual.

- Recursos hardware

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	1	0	17
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	1	-	16	28
Multiplexer	-	-	-	40
Register	-	-	75	-
Total	1	1	91	85
Available	280	220	106400	53200
Utilization (%)	~0	~0	~0	~0

FIGURA 141. SYNTHESIS REPORT (III)

- Bloques RAM, BRAM_18K: 1 de 280.

  Memoria: El array *c* de coeficientes y el registro de desplazamiento *shift_reg* implementados mediante un bloque RAM con dos bancos de memoria.

Memory

Memory	Module	BRAM_18K	Words	Bits	Banks	W*Bits*Banks
c_U	hls_fir_c	1	21	6	1	126
shift_reg_U	hls_fir_shift_reg	0	21	8	1	168
Total	2	1	42	14	2	294

FIGURA 142. SYNTHESIS REPORT (IV)

- Procesador digital de señal, DSP48E: 1 de 220.

  Expresiones: La operación de multiplicación (*) se ha implementado utilizando un DSP.

Expression

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
tmp_8_fu_181_p2	*	1	0	0	8	6
i_1_fu_168_p2	+	0	0	6	6	2
tmp_2_fu_149_p2	+	0	0	5	5	2
tmp_1_fu_139_p2	icmp	0	0	6	6	1
Total	4	1	0	17	25	11

FIGURA 143. SYNTHESIS REPORT (V)

- Flip-flops, FF: 91 de 106400.
  Memoria: 16 FFs empleados.
  Registros: 75 FFs empleados.

Register

Name	FF	LUT	Bits	Const Bits
acc_reg_91	13	0	13	0
ap_CS_fsm	3	0	3	0
c_load_reg_232	6	0	6	0
data1_reg_116	8	0	8	0
i_1_reg_227	6	0	6	0
i_cast_reg_199	32	0	32	0
i_reg_104	6	0	6	0
tmp_1_reg_208	1	0	1	0
Total	75	0	75	0

FIGURA 144. SYNTHESIS REPORT (VI)

- Logic Under Test, LUT: 85 de 53200.

  >Expresiones: Se han empleado 17 LUTs para las operaciones de suma (*+*) y comparación (*icmp*).

Expression

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
tmp_8_fu_181_p2	*	1	0	0	8	6
i_1_fu_168_p2	+	0	0	6	6	2
tmp_2_fu_149_p2	+	0	0	5	5	2
tmp_1_fu_139_p2	icmp	0	0	6	6	1
Total	4	1	0	17	25	11

FIGURA 145. SYNTHESIS REPORT (VII)

>Memoria: 28 LUTs empleados.

>Multiplexores: 40 LUTs empleados.

Multiplexer

Name	LUT	Input Size	Bits	Total Bits
acc_reg_91	13	2	13	26
data1_reg_116	8	2	8	16
i_reg_104	6	2	6	12
shift_reg_address0	5	4	5	20
shift_reg_d0	8	3	8	24
Total	40	13	40	98

FIGURA 146. SYNTHESIS REPORT (VIII)

- Interfaz (ver Tabla 7)

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	hls_fir	return value
ap_rst	in	1	ap_ctrl_hs	hls_fir	return value
ap_start	in	1	ap_ctrl_hs	hls_fir	return value
ap_done	out	1	ap_ctrl_hs	hls_fir	return value
ap_idle	out	1	ap_ctrl_hs	hls_fir	return value
ap_ready	out	1	ap_ctrl_hs	hls_fir	return value
y	out	13	ap_vld	y	pointer
y_ap_vld	out	1	ap_vld	y	pointer
x	in	8	ap_none	x	scalar

FIGURA 147. SYNTHESIS REPORT (IX)

- Señales de reloj y de reset, *ap_clk* y *ap_rst,* incorporadas al diseño RTL, dado que al diseño le lleva más de un ciclo de reloj ejecutarse y es un diseño síncrono.
- Puerto *return* implementado por defecto con el protocolo *ap_ctrl_hs*, incluyendo así las señales *ap_start*, *ap_done, ap_idle* y *ap_ready* al diseño RTL.
- Puerto de salida, *y*, implementado como un puerto de datos de 13 bits, por defecto con protocolo *ap_vld* el cual incluye una señal *y_ap_vld* que permite al usuario conocer cuándo la salida se encuentra disponible (es válida).
- Puerto de entrada, *x*, implementado como un puerto de datos de 8 bits, por defecto sin protocolo y sin añadir nuevas señales (protocolo *ap_none*).

A la vista de los resultados obtenidos, esta primera implementación no es la mejor que se puede conseguir para el diseño. Esto es, en cierta medida, lógico puesto que no se ha aplicado todavía ninguna directiva que especifique a la síntesis cómo debe actuar, por lo que Vivado HLS ha implementado el diseño siguiendo su comportamiento fijado por defecto, el cual fue descrito en el apartado de *Metodología de trabajo*.

El siguiente paso debe ser, por tanto, analizar el informe de resultados y pensar en posibles opciones de mejora para el diseño. Es posible que alguna optimización que a primera vista parece que pueda mejorar algún aspecto del resultado, resulte no ser eficaz o lo mejore a costa de empeorar otro, por eso Vivado HLS permite crear diferentes soluciones sobre el mismo código para probar distintas posibilidades sin necesidad de modificar el código original, y posteriormente comparar unos resultados con otros.

OPTIMIZACIONES

Para empezar, es conveniente crear una nueva solución en el proyecto donde realizar los cambios que se consideren para así no afectar a la solución inicial, *solution1*. Esto puede hacerse desde el botón *New Solution* situado en la barra de herramientas, el cual abrirá una ventana donde especificar el nombre de la nueva solución (por defecto, *solution2*), reloj y tarjeta (ya seleccionados y comunes a la anterior solución), y en opciones permite copiar directivas y restricciones, si existen, de la solución anterior, que para este caso puede dejarse marcado por defecto. Al finalizar se observará en el explorador de proyecto una nueva carpeta llamada *solution2* que es la solución que en este momento se encuentra activa, aunque permite retornar a la *solución 1* cuando se necesite seleccionando ésta.

Una vez se encuentra activa la solución 2 es momento de comenzar a añadir directivas de optimización al diseño. Los objetivos son:

- Buscar el máximo rendimiento en el diseño.
- Buscar, en la medida de lo posible, el mínimo área consumida.
- Lograr un intervalo de iniciación 1 para poder leer nuevas entradas en cada ciclo de reloj.

El primer y último objetivo están relacionados entre sí, puesto que un buen rendimiento implica tener un intervalo lo más bajo posible.

En el análisis del informe de síntesis de la solución 1 podía verse que la latencia del diseño viene dada por la latencia del bucle *Shift_Accum_Loop*. Vivado HLS mantiene este bucle enrollado por defecto, es decir, los recursos hardware estimados para una iteración del bucle son reutilizados para cada iteración, lo cual obliga a que el bucle se ejecute de forma secuencial, teniendo una latencia de bucle mayor pero optimizando en recursos.

Por otro lado, el array *shift_reg* ha sido implementado mediante un bloque RAM, puesto que así lo hace Vivado HLS por defecto, por lo que el diseño consumiría muchos ciclos con operaciones de lectura/escritura, ya que una operación de lectura requiere dos ciclos, un ciclo para obtener la dirección y otro para leer, y cada iteración del bucle requiere una lectura y una escritura en este bloque RAM.

Puede verse, por tanto, que los dos factores que limitan el rendimiento del diseño son el bucle for, y el array *shift_reg*. Comenzando con el bucle for, *Shift_Accum_Loop*, puede aplicarse la directiva UNROLL para permitir que todas las operaciones se ejecuten en paralelo (si las dependencias entre lecturas/escrituras a registros o memorias lo permiten) y reducir así la latencia del bucle.

Esto se hace desde el panel auxiliar en la pestaña *Directive*, que puede verse cuando en el panel de información se muestra la función principal *hls_fir.c*.

Hasta el momento no se ha incluido ninguna directiva, por lo que esta pestaña únicamente mostrará la jerarquía de variables, arrays y bucles que contiene la función principal.

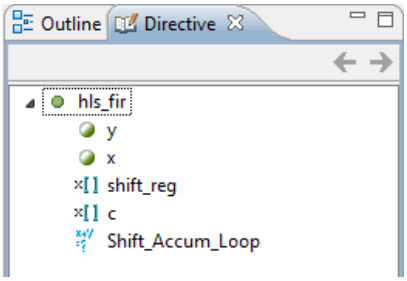


FIGURA 148. DIRECTIVAS APLICADAS AL DISEÑO ACTUAL

Seleccionando la etiqueta *Shift_Accum_Loop* y con el botón derecho *Insert Directive*, aparecerá una ventana donde seleccionar la directiva UNROLL. Las opciones pueden dejarse por defecto, sin marcar, y como destino se deja por defecto *Directive File*.

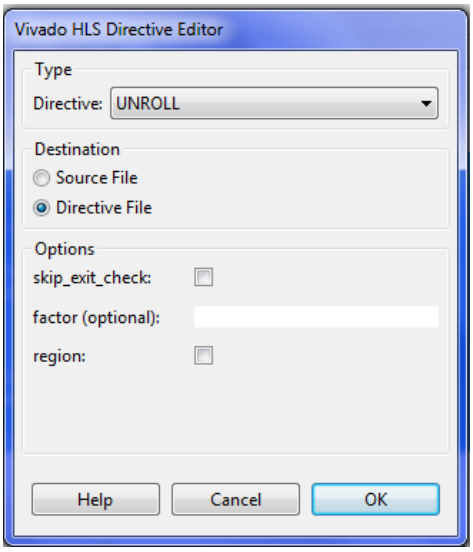


FIGURA 149. INSERTAR DIRECTIVA UNROLL AL DISEÑO ACTUAL

La directiva únicamente aparecerá reflejada en la pestaña Directive bajo el bucle puesto que, por defecto, se ha seleccionado como destino el fichero de directivas (*directives.tcl*) en vez de situarla como pragma en el código, lo que asegura que en futuras soluciones dicha directiva no sea copiada automáticamente de esta solución a la siguiente y pueda realizarse una combinación diferente de optimizaciones.

A pesar de que ahora el bucle haya sido desenrollado y permita realizar las operaciones en paralelo para reducir la latencia, al estar el array *shift_reg* implementado como bloque RAM, en cada iteración del bucle se va a tener que realizar una lectura y una escritura a esta RAM, limitándose la ventaja de la ejecución en paralelo. Por ello, si es posible, es mejor no emplear bloques RAM en el diseño.

En este caso, el array *shift_reg* puede ser separado en elementos individuales que se implementarán como un registro, de forma que se tiene un registro paralelo en lugar de un bloque RAM. El procedimiento es el mismo, seleccionando el array *shift_reg* y con el botón derecho *Insert Directive*, aparecerá una ventana donde seleccionar la directiva ARRAY_PARTITION. El resto puede dejarse como aparece por defecto.

La directiva únicamente aparecerá reflejada en la pestaña *Directive* bajo *shift_reg* puesto que, por defecto, se ha seleccionado como destino el fichero de directivas (*directives.tcl*) en vez de situarla como pragma en el código, lo que asegura que en futuras soluciones dicha directiva no sea copiada automáticamente de esta solución a la siguiente y puedan tener diferentes optimizaciones.

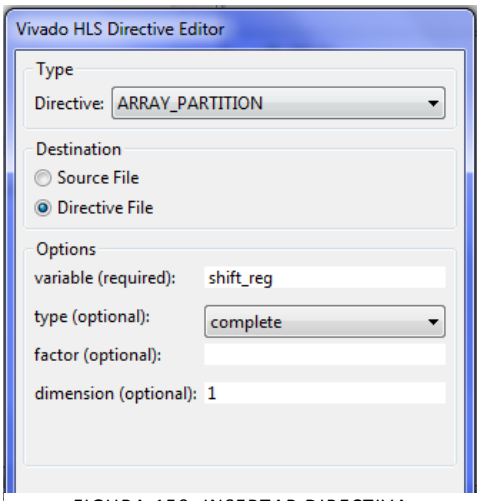


FIGURA 150. INSERTAR DIRECTIVA ARRAY_PARTITION EN EL DISEÑO

En este punto, el panel auxiliar de directivas debe tener el aspecto de la Figura 151.

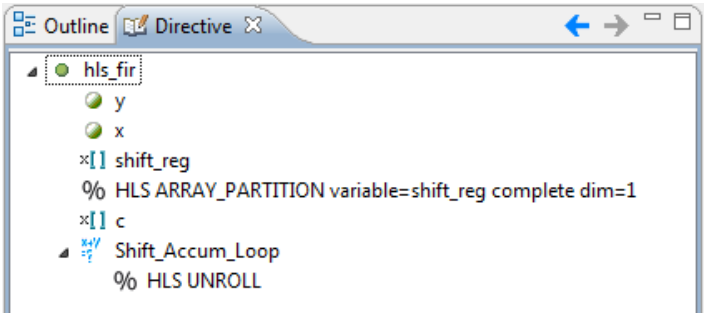


FIGURA 151. RESUMEN DE DIRECTIVAS APLICADAS A LA SOLUCIÓN ACTUAL

En el fichero que almacena las directivas que han sido aplicadas por el usuario y están contenidas en el diseño puede abrirse desde el explorador de proyecto en *solution2>constraints>directives.tcl*. En la Figura 152 pueden verse aplicadas las directivas Array_partition al array *shift_reg* y la directiva Unroll al bucle *Shift_Accum_Loop*.

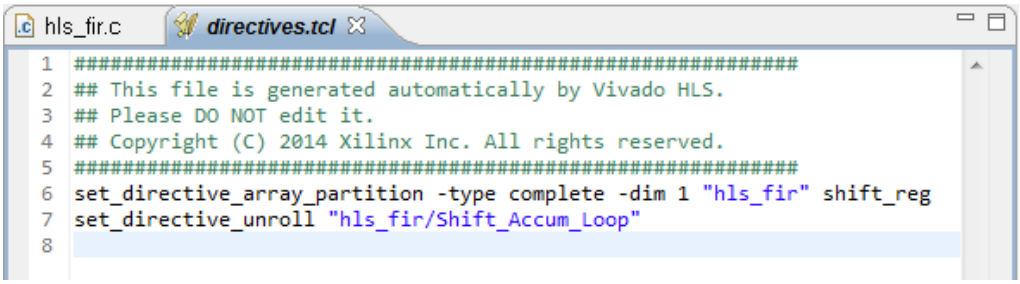


FIGURA 152. FICHERO DE DIRECTIVAS DE LA SOLUCIÓN ACTUAL, DIRECTIVES.TCL

Para comprobar el impacto de estas directivas sobre el diseño, se realiza sobre esta nueva solución, *solution2*, la simulación para comprobar el funcionamiento, y posteriormente la síntesis para observar el informe de resultados. Además, durante el proceso de síntesis, es importante estar atento a los mensajes mostrados por la consola, porque si una directiva no es posible aplicarla o no tiene beneficio Vivado HLS así lo indicará explicando el porqué.

Una vez realizada la síntesis, lo más conveniente es hacer uso de la opción *Compare Report* que ofrece Vivado HLS, facilitando al usuario comparar los informes de resultados de varias soluciones para comprobar si se han logrado los objetivos esperados. Comparando estas dos soluciones, se concluye, a partir de la Figura 153 lo siguiente:

En cuanto a rendimiento, el periodo de reloj estimado ha aumentado, dado que la lógica ha crecido al desenrollar el bucle for para realizar operaciones en paralelo. No obstante, se observa que la latencia se ha reducido considerablemente gracias a este desenrollado del bucle y la eliminación de las operaciones de lectura y escritura a la RAM.

El intervalo de iniciación sigue estando un ciclo por encima que la latencia dado que no existe segmentación en el diseño y debe esperar a finalizar la transacción completa para volver a procesar nuevas entradas.

En cuanto a los recursos hardware a emplear, puede verse que se ha eliminado el bloque RAM del diseño a cambio de consumir FF's para implementar cada registro del registro de desplazamiento, y un aumento de los LUTs debido al incremento de operaciones por ciclo al no reutilizar la misma lógica para todas las iteraciones del bucle for sino operar de forma paralela.

Vivado HLS Report Comparison

All Compared Solutions

solution1: xc7z020clg484-1
solution2: xc7z020clg484-1

Performance Estimates

Timing (ns)

Clock		solution1	solution2
default	Target	10.00	10.00
	Estimated	6.38	7.84

Latency (clock cycles)

		solution1	solution2
Latency	min	85	2
	max	85	2
Interval	min	86	3
	max	86	3

Utilization Estimates

	solution1	solution2
BRAM_18K	1	0
DSP48E	1	1
FF	91	163
LUT	85	92

Export the report(.html) using the Export Wizard

FIGURA 153. COMPARACIÓN DE INFORMES TRAS LA SÍNTESIS PARA LA SOLUCIÓN INICIAL Y LA SOLUCIÓN CON DIRECTIVAS UNROLL Y ARRAY_PARTITION

Aplicando las directivas UNROLL y ARRAY_PARTITION puede verse que se ha conseguido un mejor rendimiento, a pesar de haber aumentado los recursos. Se considera una buena implementación puesto que, aunque se esté consumiendo más área, sigue siendo un área prácticamente despreciable teniendo en cuenta todos los recursos hardware disponibles en la tarjeta. Se usa 1 DSP48E de 220 disponibles, 163 FFs de 106.400 disponibles y 92 LUTs de 53.200 disponibles.

Se entiende entonces que se va por buen camino hacia la implementación más óptima del diseño. Ahora que se han realizado las optimizaciones necesarias para la ejecución concurrente, puede hacerse uso de la directiva PIPELINE para segmentar la función principal de forma que todas las transacciones de la función principal puedan hacerse sin necesidad de esperar a que los resultados sean escritos, reutilizando los recursos hardware tan pronto como queden libres.

Para mantener la solución encontrada hasta ahora, se crea una nueva solución, *solution3*, marcando la opción de copiar las directivas de la solución 2, ya que se quiere añadir una nueva entre ellas.

En esta nueva solución se selecciona en la pestaña *Directive* la función principal hls_fir y se inserta la directiva PIPELINE, en la cual puede especificarse que se quiere un II (*Initiation Interval*) igual a 1, tal como puede verse en la Figura 154.

Esta indicación permitirá ver el objetivo de aplicación de la directiva (II=1) pero no es necesaria puesto que Vivado HLS intenta por defecto conseguir ese intervalo unidad.

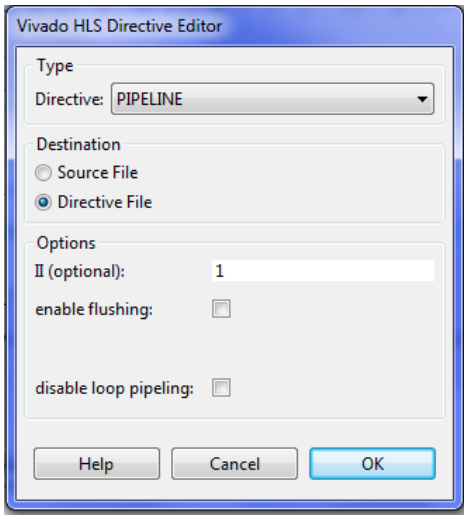


FIGURA 154. INSERTAR DIRECTIVA PARA LA SEGMENTACIÓN EN LA SOLUCIÓN ACTUAL

El panel auxiliar de directivas tendrá ahora el aspecto de la Figura 155.

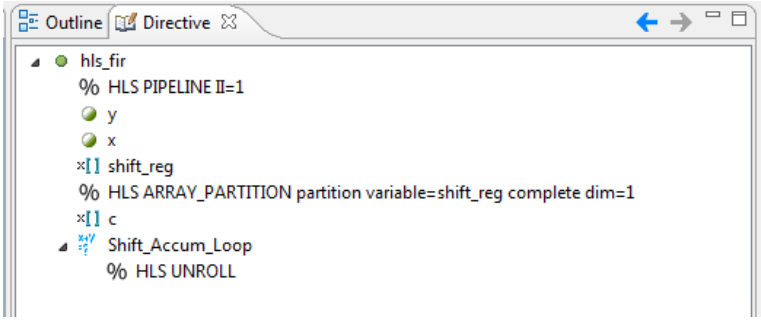


FIGURA 155. RESUMEN DE DIRECTIVAS EN ESTA NUEVA SOLUCIÓN

No obstante, una vez incluida en el diseño la directiva PIPELINE puede omitirse la directiva UNROLL, ya que la segmentación de la función lleva consigo un desenrollado de bucles, y se obtiene el mismo resultado de implementación.

De igual forma se habrá incorporado esta directiva al fichero *directives.tcl* de la solución.

Realizando la síntesis de esta nueva solución y comparándola con las dos soluciones anteriores (solución original sin directivas y solución con directivas UNROLL y ARRAY_PARTITION) se obtiene el siguiente informe de resultados de la Figura 156:

Puede verse que el periodo de reloj estimado se mantiene con respecto a la solución 2, pero ahora el intervalo de iniciación se ha conseguido que sea la unidad y un ciclo inferior a la latencia gracias a la directiva PIPELINE.

Esta implementación aceptará nuevas entradas en cada ciclo de reloj, siendo ésta la solución más óptima que se puede conseguir para este tipo de diseño.

En cuanto a los recursos hardware utilizados se observa que esta última solución emplea algunos registros más, pero sigue suponiendo un consumo despreciable. La solución final emplea por tanto 1 DSP48E, 220 FFs y 92 LUTs.

Vivado HLS Report Comparison

All Compared Solutions

solution1: xc7z020clg484-1
solution2: xc7z020clg484-1
solution3: xc7z020clg484-1

Performance Estimates

Timing (ns)

Clock		solution1	solution2	solution3
default	Target	10.00	10.00	10.00
	Estimated	6.38	7.84	7.84

Latency (clock cycles)

		solution1	solution2	solution3
Latency	min	85	2	2
	max	85	2	2
Interval	min	86	3	1
	max	86	3	1

Utilization Estimates

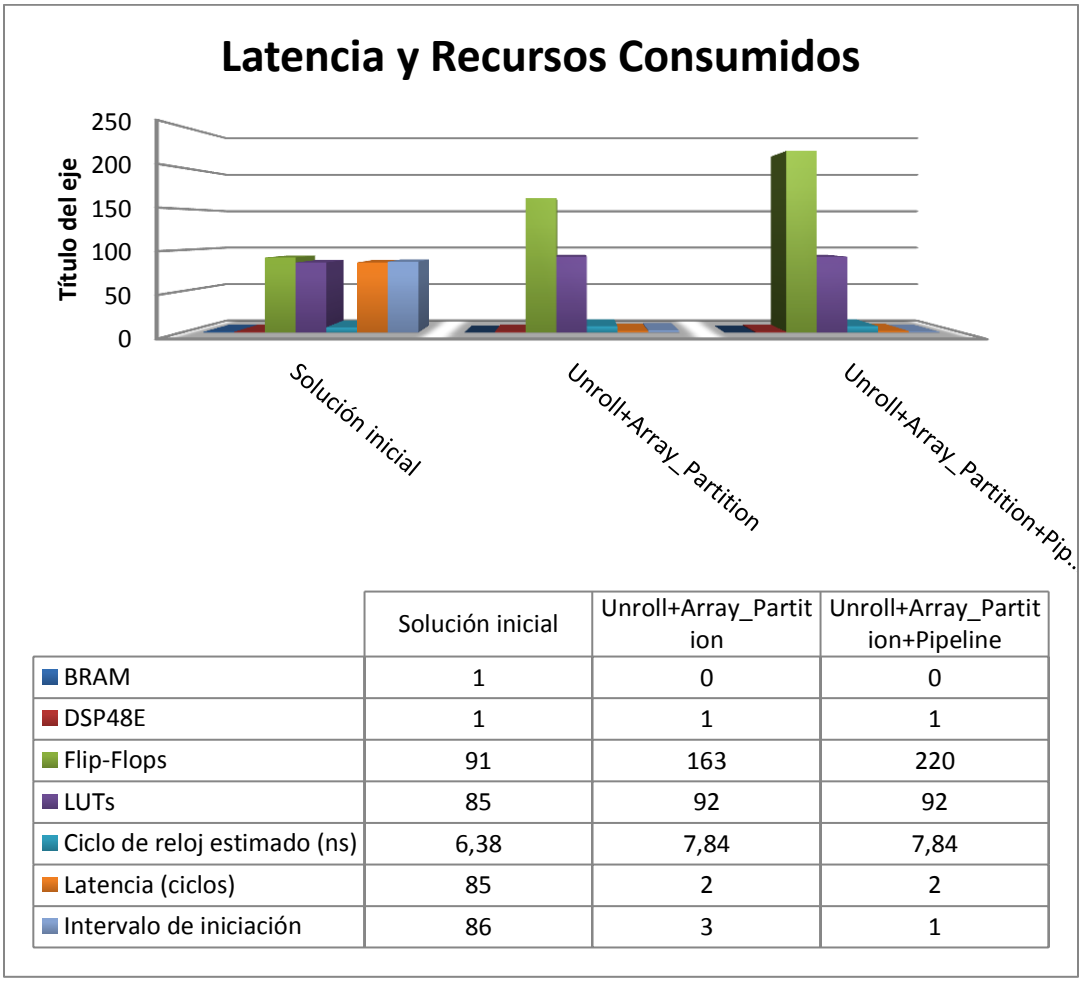
	solution1	solution2	solution3
BRAM_18K	1	0	0
DSP48E	1	1	1
FF	91	163	220
LUT	85	92	92

Export the report(.html) using the Export Wizard

FIGURA 156. COMPARACIÓN DE INFORMES TRAS LA SÍNTESIS PARA LAS DOS SOLUCIONES ANTERIORES Y LA NUEVA SOLUCIÓN CON DIRECTIVA PIPELINE

A la vista del gráfico de la Figura 157 con esta tercera solución se ha llegado entonces a una implementación que puede considerarse la más óptima pues aunque tiene un consumo de área mayor que en las anteriores implementaciones, no supera los recursos disponibles en la tarjeta, en

cambio sí consigue el mejor rendimiento que se puede esperar al tener II=1 y procesar una muestra nueva por ciclo de reloj.



	Solución inicial	Unroll+Array_Partition	Unroll+Array_Partition+Pipeline
BRAM	1	0	0
DSP48E	1	1	1
Flip-Flops	91	163	220
LUTs	85	92	92
Ciclo de reloj estimado (ns)	6,38	7,84	7,84
Latencia (ciclos)	85	2	2
Intervalo de iniciación	86	3	1

FIGURA 157. GRÁFICO COMPARATIVA DE LATENCIA Y RECURSOS EN EL FILTRO FIR

Por otro lado, no se descarta que exista alguna otra combinación de directivas que consiga un resultado similar, pero para el caso de este diseño este último conjunto de directivas es el que ofrece mejores resultados.

Llegados a este punto, es importante comentar otro aspecto que hace que el diseño tenga una implementación óptima, y es el uso de ancho de bit no estándar que se ha empleado para las variables del diseño desde el inicio del proyecto. A continuación se comentará por qué se ha escogido ese ancho de bit y qué ventajas conlleva utilizarlo.

USO DE TIPOS DE PRECISIÓN ARBITRARIA EN ANSI-C

Inicialmente, se comenzó realizando el diseño con variables que empleaban tipos de datos estándar de C, pero tratándose de un diseño que da lugar a hardware es importante no utilizar más recursos de los necesarios, y en eso se basa el uso de ancho de bit arbitrario que permite Vivado HLS.

Escoger el ancho de bit más preciso para cada variable del diseño resultará en menor número de operadores necesarios, mínimo área consumida y menor número de ciclos de ejecución. Como desventaja puede comentarse que Vivado HLS no permite emplear la opción de depuración *Debug* al usar tipos de precisión arbitraria si se está trabajando con lenguaje ANSI C, sino que deben emplearse las funciones *printf* o *fprintf* para revisar puntos del código.

Para escoger el ancho de bit que más se ajusta a las variables del diseño, se ha buscado una solución de compromiso entre el número de recursos hardware consumidos y el error medio a la salida, como muestra la Tabla 9.

Trabajando en paralelo con Matlab y Vivado HLS, se tiene en el espacio de trabajo de Matlab el array de coeficientes exportados por la herramienta fdatool, *Num*. Éste se convertirá al rango del tipo que se asigne a la variable de coeficientes y se redondeará para tener los coeficientes de parte entera que introducir en Vivado HLS.

Por otro lado, el array de coeficientes previo al redondeo se introducirá en la función *filter* de Matlab para obtener la salida real del filtro que se comparará con la salida de Vivado HLS para obtener el error medio que existe entre ambas.

La salida de Vivado HLS puede recogerse abriendo un puerto serie desde Matlab o utilizando también la función *filter* con los coeficientes de parte entera.

Este análisis se ha hecho variando el ancho del tipo de *coef_t* para el array *c* entre 4 y 10 bits, ajustando los anchos de bit de las variables de salida (*dataout_t* y *acc_t*) al mínimo necesario que cubra el valor máximo de salida obtenido, es decir, si la salida del filtro tiene un valor máximo de 1.920, el ancho de datos será el inmediato superior, 2^{11}=2.048 > 1.920, datos de 12 bits.

En cuanto al ancho de bit de la señal de entrada al filtro, para el diseño que se está realizando con una señal rampa de 1 a 70, se necesita un ancho fijo de 8 bits puesto que $2^6 < Vin_{máx} < 2^7$.

La Tabla 9 muestra el análisis que se ha seguido. Los comandos de Matlab convierten el array de coeficientes Num al rango de la variable coef_t, utilizan ese array para calcular la salida ideal con la función filter, redondean los coeficientes a números enteros, se obtiene la salida de Vivado HLS con esos coeficientes y finalmente se calcula el error medio de ambas salidas, Matlab vs Vivado.

Elección del ancho de bit para tipos con precisión arbitraria		
Ancho de bits	Comandos Matlab	Resultados
4 bits	>> num4=Num.*2^3; >> yreal4=filter(num4,1,x); >> coef=round(num4); >> yvivadohls4=filter(coef,1,x); >>error4=(abs(yreal4)-abs(yvivadohls4)); >>errormedio4=abs(mean(error4));	Error medio: 26.1858
int4 coef_t int8 datain_t int11 dataout_t int11 acc_t		Hardware: -DSP48E: 0 -FF: 97 -LUT: 41
6 bits	>> num6=Num.*2^5; >> yreal6=filter(num6,1,x); >> coef=round(num6); >> yvivadohls6=filter(coef,1,x); >>error6=(abs(yreal6)-abs(yvivadohls6)); >>errormedio6=abs(mean(error6));	Error medio: 26.2567
int6 coef_t int8 datain_t int12 dataout_t int12 acc_t		Hardware: -DSP48E: 1 -FF: 181 -LUT: 101
7 bits	>> num7=Num.*2^6; >> yreal7=filter(num7,1,x); >> coef=round(num7); >> yvivadohls7=filter(coef,1,x); >>error7=(abs(yreal7)-abs(yvivadohls7)); >>errormedio7=abs(mean(error7));	Error medio: 0.0294
int7 coef_t int8 datain_t int13 dataout_t int13 acc_t		Hardware: -DSP48E: 1 -FF: 220 -LUT: 92
8 bits	>> num8=Num.*2^7; >> yreal8=filter(num8,1,x); >> coef=round(num8); >> yvivadohls8=filter(coef,1,x); >>error8=(abs(yreal8)-abs(yvivadohls8)); >>errormedio8=abs(mean(error8));	Error medio: 0.1589
int8 coef_t int8 datain_t int14 dataout_t int14 acc_t		Hardware: -DSP48E: 3 -FF: 298 -LUT: 127
10 bits	>> num10=Num.*2^9; >> yreal10=filter(num10,1,x); >> coef=round(num10); >> yvivadohls10=filter(coef,1,x); >>error10=(abs(yreal10)-abs(yvivadohls10)); >> errormedio10=abs(mean(error10));	Error medio: 52.735
int10 coef_t int8 datain_t int16 dataout_t int16 acc_t		Hardware: -DSP48E: 5 -FF: 353 -LUT: 257

TABLA 9. ANÁLISIS DEL ANCHO DE BIT DE LA VARIABLE DE COEFICIENTES PARA TIPOS DE PRECISIÓN ARBITRARIA

Si el diseño se hubiese realizado con tipos estándares de C con entradas de tipo *short* (ancho de 16 bits) y salidas de tipo *int* (ancho de 32 bits) se hubiera obtenido un error medio de 0.1518 y dado lugar a un consumo de 19 DSP48E, 1.074 FF y 269 LUTS.

Por tanto, a la vista de los resultados obtenidos para los diferentes anchos de bit, queda justificado que la tercera opción con un ancho de 7 bits para coeficientes y de 13 bits para salidas, es la opción más adecuada a utilizar en el diseño, con la que se consigue reducir el error medio en un

80.6% respecto a la solución que emplea anchos de bit estándares de C, y emplear una cantidad de recursos hardware notablemente baja, suponiendo un 0.454% de DSPs, un 0.207% de FFs y un 0.173% de LUTs.

IMPLEMENTACIÓN DE LAS INTERFACES AXI4-LITE

Este es el momento de agrupar los puertos de entrada y salida del diseño en una única interfaz AXI4 Lite conjunta que permita que el periférico sea controlado por el procesador. Esto se hará definiendo la interfaz AXI4-Lite en los puertos I/O para que en el momento de exportación se cree un conjunto de drivers para la comunicación. Además, el argumento *return* de la función debe llevar también esta interfaz especificada si se quiere que el diseño final incluya un puerto de salida de interrupción.

Existen dos opciones para definir estas interfaces.

Una de ellas es insertando la directiva INTERFACE sobre los puertos *"x", "y"* y sobre la función *hls_fir* (para especificarla en el puerto *return*), escogiendo el modo *s_axilite* y colocando una etiqueta, por ejemplo, *"HLS_FIR_PERIPH_BUS"* en la opción *bundle*.

Esta etiqueta será el nombre que tenga dicha interfaz, debiendo ser éste igual para todos los puertos para así agruparlos bajo ese nombre en el diseño RTL final.

La Figura 158 muestra los parámetros dados para insertar esta directiva.

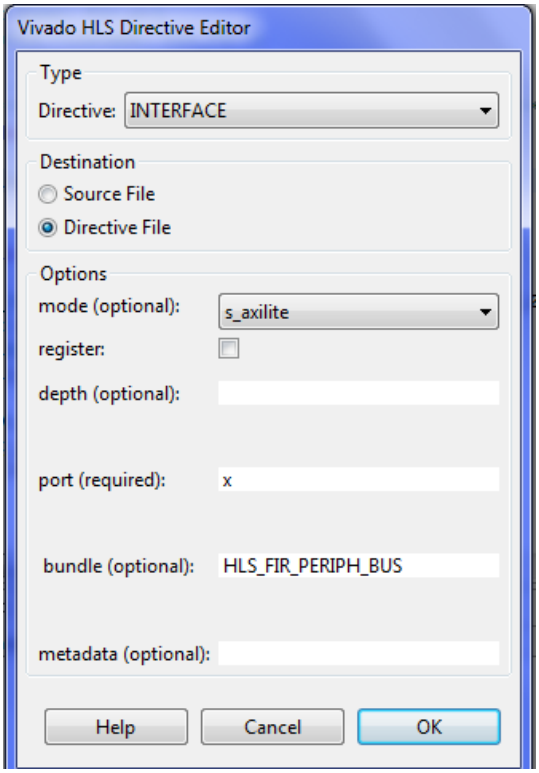


FIGURA 158. INSERTAR DIRECTIVA INTERFACE PARA DEFINIR LAS INTERFACES DEL DISEÑO

La otra opción para definir estas interfaces es especificar con la directiva RESOURCE en cada puerto I/O del diseño, que se emplee el recurso *AXI4LiteS* (*AXI4-Lite slave adapter with interrupt*) y en la opción *metadata* definir la etiqueta tal como se hacía en la opción *bundle* de la directiva INTERFACE, escribiendo *"-bus_bundle HLS_FIR_PERIPH_BUS"*.

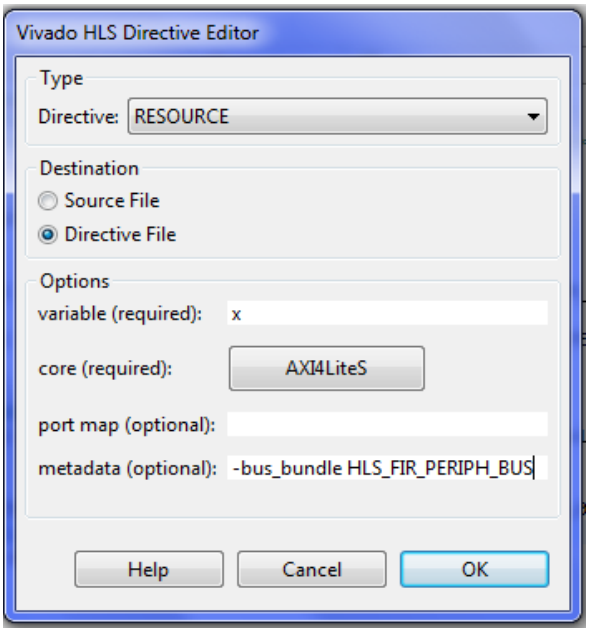


FIGURA 159. INSERTAR DIRECTIVA RESOURCE PARA DEFINIR LAS INTERFACES DEL DISEÑO

El usuario es libre de escoger la opción que prefiera. Lo que se recomienda al usuario es, una vez finalizado el proceso de inclusión de directivas, modificar cada una de ellas para situarlas en el código en vez de en el fichero de directivas *directives.tcl*. De esa forma, queda visible en el diseño las opciones que se han escogido para la síntesis, evitando tener que localizar el fichero *.tcl para consultarlo.

Finalmente, el código de la función principal *hls_fir.c* queda de la siguiente manera, incluyendo las directivas en forma de pragmas.

```
/************************************************************************
Project Name: fir_prj
Associated Filename: hls_fir.c
Family: Zynq
Device: xc7z020clg484-1
************************************************************************/
#include "hls_fir.h"

void hls_fir (dataout_t *y, datain_t x)
{
#pragma HLS PIPELINE II=1
#pragma HLS RESOURCE variable=x core=AXI4LiteS metadata="-bus_bundle
HLS_FIR_PERIPH_BUS"
#pragma HLS RESOURCE variable=y core=AXI4LiteS metadata=""-bus_bundle
HLS_FIR_PERIPH_BUS"
```

```
#pragma HLS RESOURCE variable=return core=AXI4LiteS metadata="-bus_bundle
HLS_FIR_PERIPH_BUS"

  static datain_t shift_reg[N];
  #pragma HLS ARRAY_PARTITION variable=shift_reg complete dim=1 partition
  coef_t c[N]={0,0,0,1,0,-2,-3,0,8,17,22,17,8,0,-3,-2,0,1,0,0,0};
  acc_t acc;
  datain_t data;
  int i;

  acc=0;
  Shift_Accum_Loop:
  for (i=N-1;i>=0;i--)
  {
       #pragma HLS UNROLL
              if (i==0){
                     shift_reg[0]=x;
                     data = x;}
              else{
                     shift_reg[i]=shift_reg[i-1];
                     data = shift_reg[i];}

    acc+=data*c[i];
  }
  *y=acc;
}
```

Se ven a lo largo del código correspondiente a la función top-level los pragmas que hacen referencia a las directivas que han sido fijadas. En primer lugar aparece el pragma que indica que la función principal ha sido segmentada, en segundo lugar aparecen los pragmas para la interfaz AXI4-Lite aplicada a cada argumento de la función, en tercer lugar aparece el pragma que indica que el array unidimensional shift_reg ha sido partido en registros independientes para poder aprovechar los beneficios de la segmentación, y en último lugar aparece el pragma que indica que el bucle Shif_Accum_Loop ha sido desenrollado para poder realizar sus operaciones en paralelo, aunque esta última directiva no habría sido necesaria fijarla ya que automáticamente la directiva Pipeline aplicada a la función top-level desenrolla los bucles contenidos en el código.

EXPORTAR EL DISEÑO RTL

Una vez seleccionada la implementación más óptima para el diseño a realizar, es necesario llevar a cabo una verificación del diseño RTL.

Esta verificación se realiza mediante la opción *Run C/RTL Cosimulation* del botón situado en la barra de herramientas, pudiendo elegirse entre los distintos lenguajes de descripción hardware, HDL.

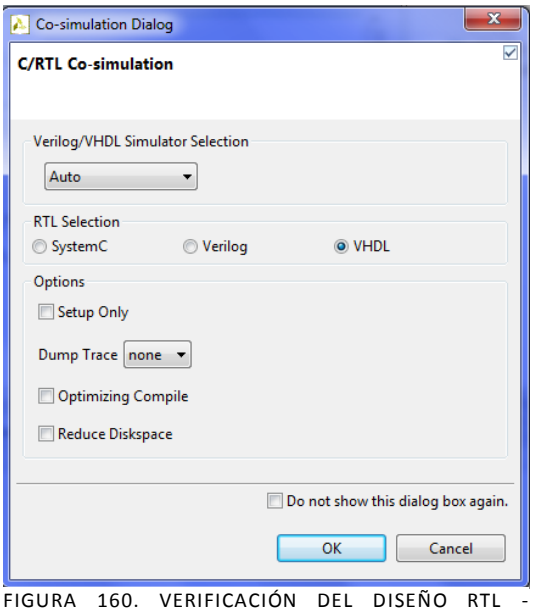


FIGURA 160. VERIFICACIÓN DEL DISEÑO RTL - COSIMULACIÓN

De igual forma que en la síntesis, al realizarse la verificación RTL se devuelve un informe de resultados informando de si ésta ha resultado ser exitosa (*PASS*) o dio error (*FAILED*), y además puede verse el mensaje del *test bench* indicando si los resultados coinciden.

Si la verificación fue correcta, el último paso en Vivado HLS es exportar el diseño RTL final como un bloque IP (*Intellectual Property*) a otras herramientas de diseño de Xilinx para ser sintetizada en un fichero *bitstream* necesario para programar la FPGA. En este caso será exportado a Vivado Design Suite para continuar con el diseño.

Para exportarlo existe un botón en la barra de herramientas denominado *Export RTL* que abre un diálogo donde se puede escoger entre los formatos de salida *IP Catalog, System Generator for DSP, System Generator for DSP (ISE), Pcore for EDK* y *Synthesized Checkpoint (.dcp)*. En este caso se escogerá *IP Catalog* para añadirlo posteriormente al Catálogo IP de Vivado.

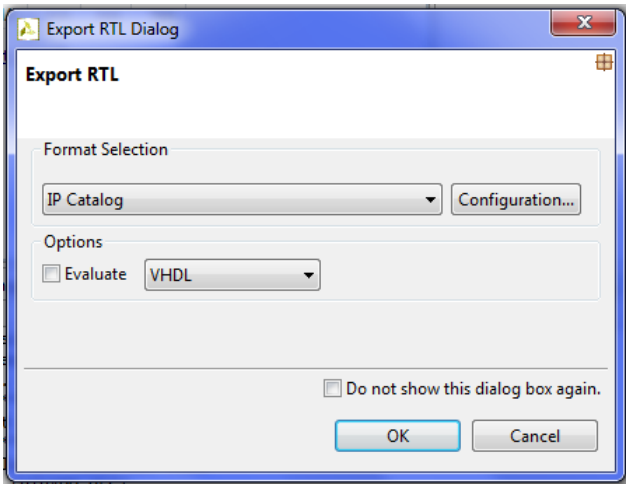


FIGURA 161. EXPORTAR EL DISEÑO RTL COMO BLOQUE IP

Puede verse, como ya se comentó, la opción *Evaluate* para comprobar la precisión de las estimaciones indicadas en el informe de síntesis mediante la síntesis RTL durante el proceso de exportación del diseño, aunque supone emplear un mayor tiempo de verificación.

Al finalizar el proceso de exportación se añadirá la carpeta *impl* al explorador de proyecto, y en ella el subdirectorio *ip*, con información referente al periférico que será importado desde Vivado recogido en un fichero *.zip, y los subdirectorios *vhdl* y *verilog*, con los proyectos para Vivado en cada uno de los lenguajes.

GENERACIÓN DEL BITSTREAM EN VIVADO

A partir de este punto se trabaja con la herramienta Vivado Design Suite 2014.1 de Xilinx. Para continuar con el diseño del periférico, debe crearse un nuevo proyecto en Vivado en el que importar el bloque IP que ha sido exportado desde Vivado HLS.

Para crear un nuevo proyecto, en la ventana de bienvenida de Vivado aparece la opción *Create New Project*, que lanza un asistente para comenzar la creación.

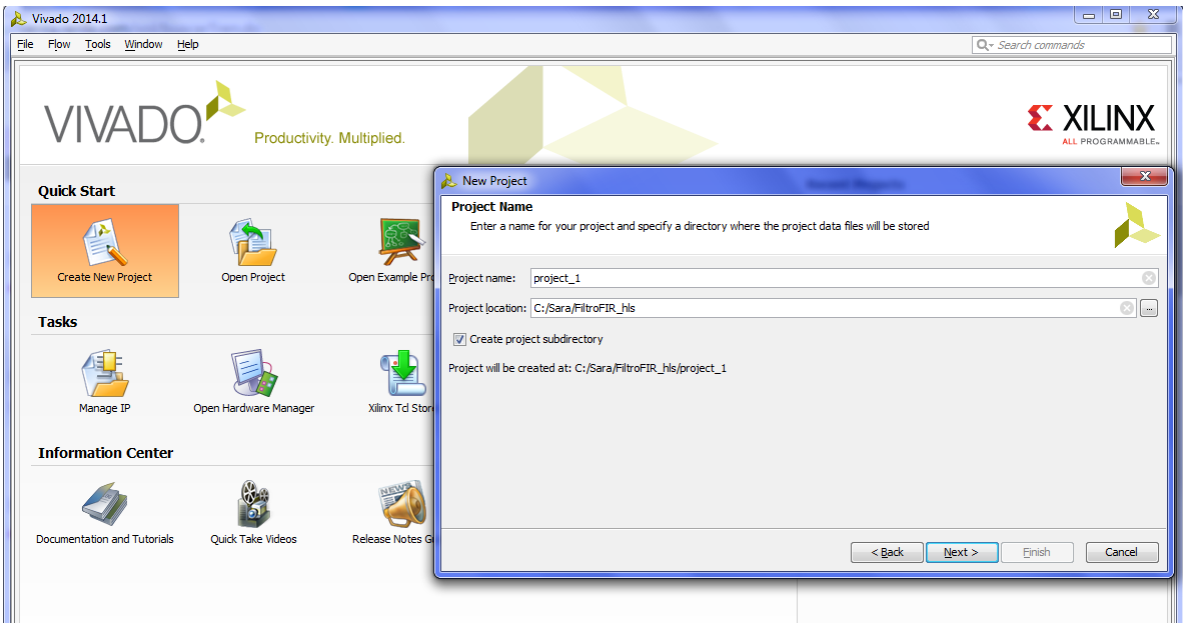


FIGURA 162. NUEVO PROYECTO EN VIVADO - NOMBRE Y DIRECTORIO

Una vez escogidos el nombre y el directorio del proyecto (que coincide con el directorio del proyecto en Vivado HLS), y con la opción *Create Project subdirectory* marcada, se hace clic en *Next*, donde se especifica el tipo de proyecto que se quiere crear, en este caso *RTL Project*.

Puede verse que esta opción indica al usuario que es posible añadir fuentes, crear bloques de diseño con IP Integrator, generar IP, realizar análisis, síntesis e implementación RTL,… que son las opciones que interesan para nuestro diseño.

Además, conviene marcar la opción *Do not specify sources at this time* para que el asistente no pregunte por ficheros HDL para añadir al proyecto, ya que se añadirá el bloque IP en conjunto.

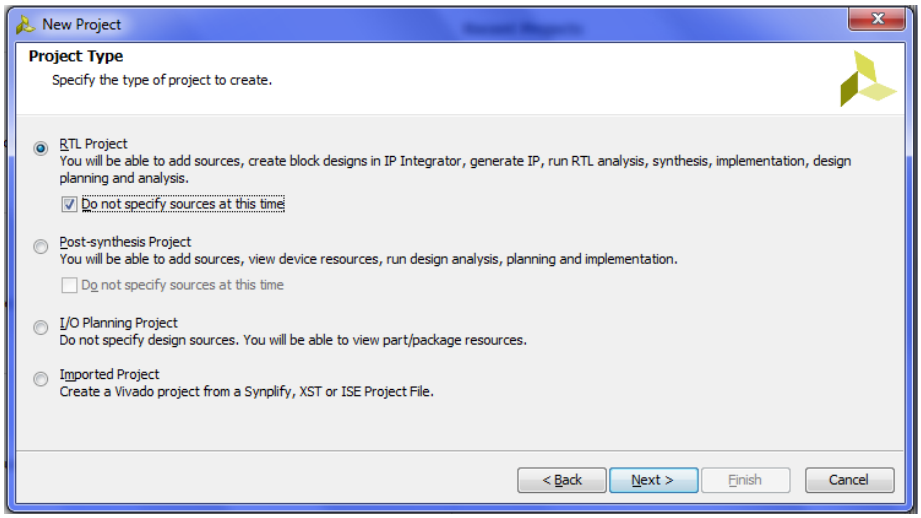


FIGURA 163. NUEVO PROYECTO EN VIVADO - RTL PROJECT

A continuación se selecciona la tarjeta sobre la que se va a trabajar, ZYNQ-7 ZC702 Evaluation Board.

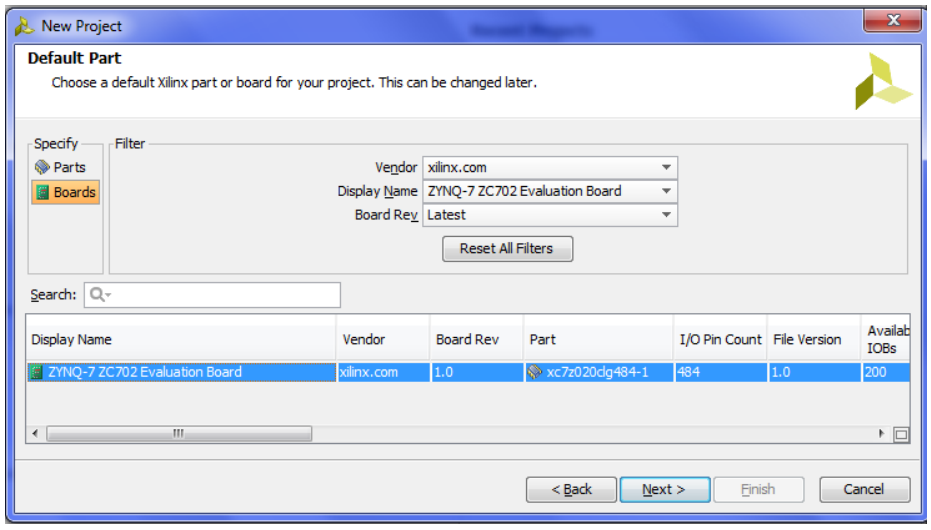


FIGURA 164. NUEVO PROYECTO EN VIVADO - TARJETA ZYNQ

Finalmente aparece un resumen de las opciones especificadas para el proyecto durante su creación. Al seleccionar *Finish* se muestra la interfaz gráfica de usuario de Vivado.

Al abrirse Vivado, puede distinguirse entre distintas ventanas que aparecen en la pantalla. A la izquierda se encuentra el explorador de proyecto, con distintas acciones para realizar sobre él, las cuales se verán más adelante al continuar con el diseño. A la derecha se encuentra el panel de información donde se mostrará el contenido tanto de ficheros como de bloques de diseño, y el catálogo IP. En la parte central se observan los directorios de proyecto de forma jerárquica. Y en la parte inferior se encuentra la consola, donde se mostrarán mensajes de warning, de error o de procesos de síntesis, implementación y generación del *bitstream* cuando se ejecuten.

El primer paso a dar en Vivado es abrir el catálogo IP desde el explorador de proyecto, para incluir en él el bloque IP del periférico.

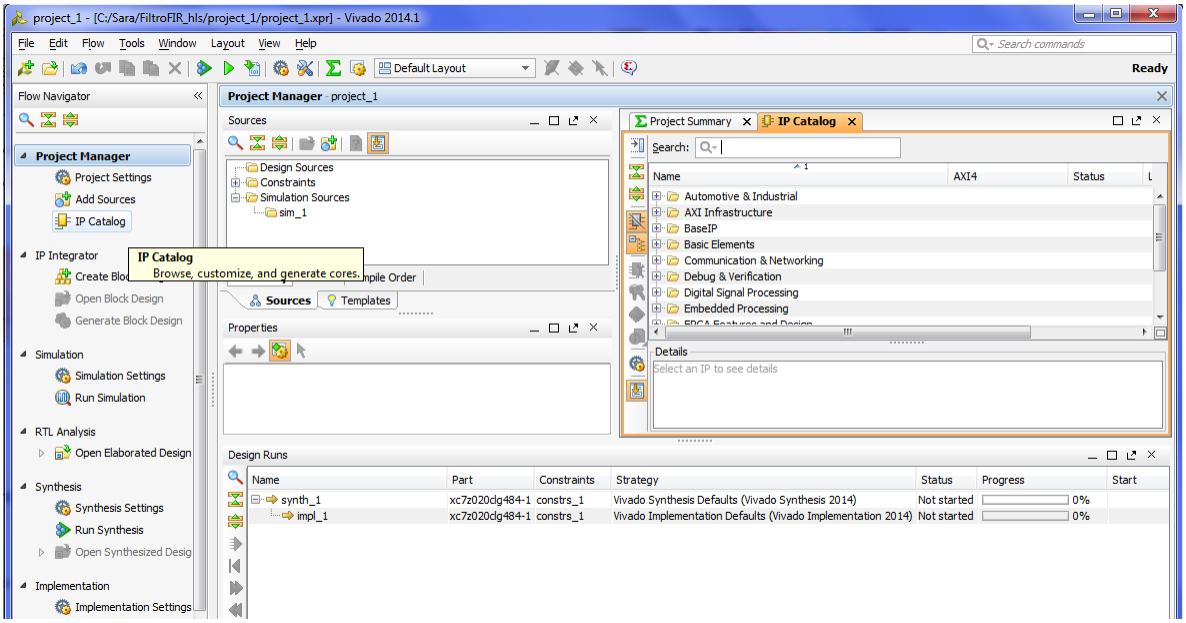


FIGURA 165. INTERFAZ DE USUARIO DE VIVADO - ABRIR CATÁLOGO IP

Al abrirse el catálogo en el panel de información, se selecciona el icono *IP Settings* situado entre los últimos iconos de la columna izquierda del catálogo.

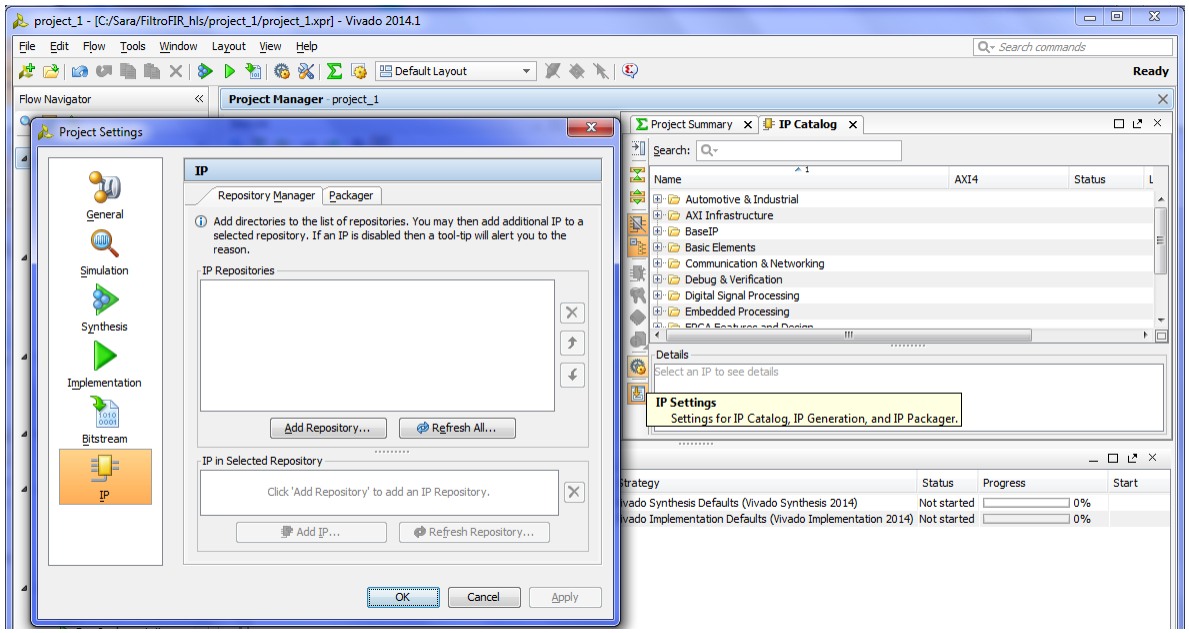


FIGURA 166. VENTANA DE CONFIGURACIÓN DE IP'S

Para añadir el bloque IP, debe crearse primero un repositorio donde almacenarlo. Esto se hace en la opción *Add Repository* que se muestra, localizando el directorio del proyecto, seleccionando *Create a new folder* y escogiendo el nombre para el repositorio, en este caso *vivado_ip_repo*.

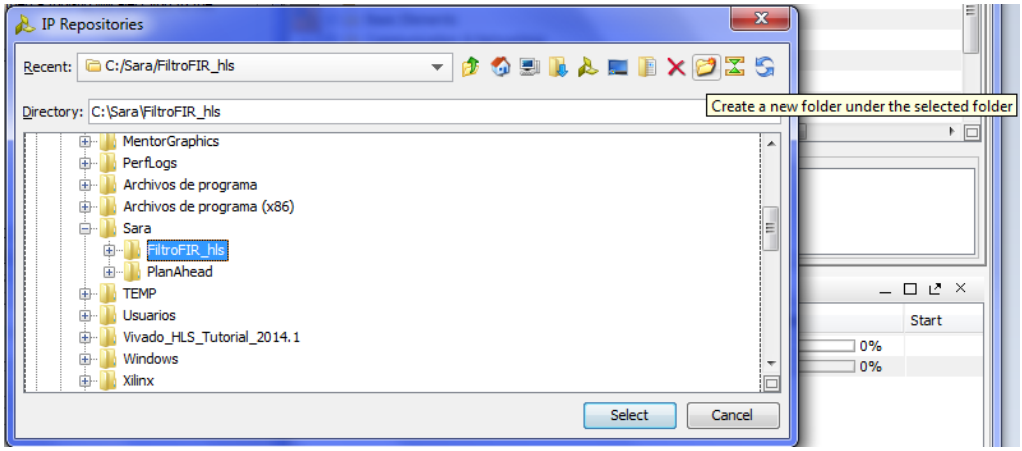


FIGURA 167. CREACIÓN DE UN REPOSITORIO DONDE AÑADIR EL BLOQUE IP EXPORTADO

El siguiente paso es incluir el bloque IP en el repositorio creado. Desde la ventana *IP Settings* debe aparecer en *IP Repositories* el directorio del repositorio creado, y abajo *IP in Selected Repository* se encontrará vacío puesto que aún no se ha añadido. Se selecciona *Add IP* y se localiza en el proyecto de Vivado HLS dentro de la solución final implementada, en los ficheros de

implementación (carpeta *impl*) y en la carpeta *ip*, el fichero .zip que contiene el bloque IP exportado.

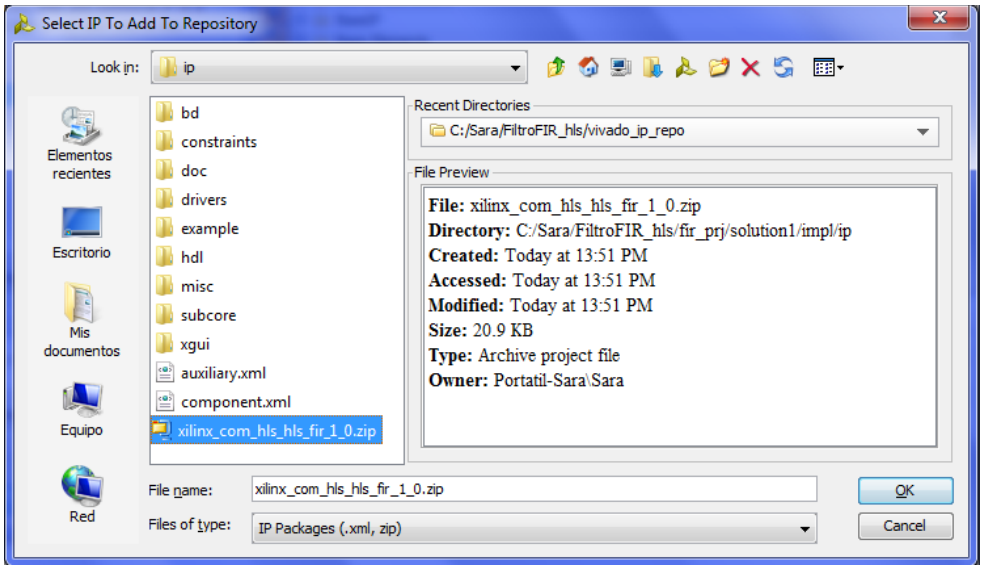


FIGURA 168. LOCALIZACIÓN DEL FICHERO CON EL BLOQUE IP EXPORTADO

Finalmente, con el repositorio y el bloque IP añadido, la ventana de ajustes tendrá el siguiente aspecto.

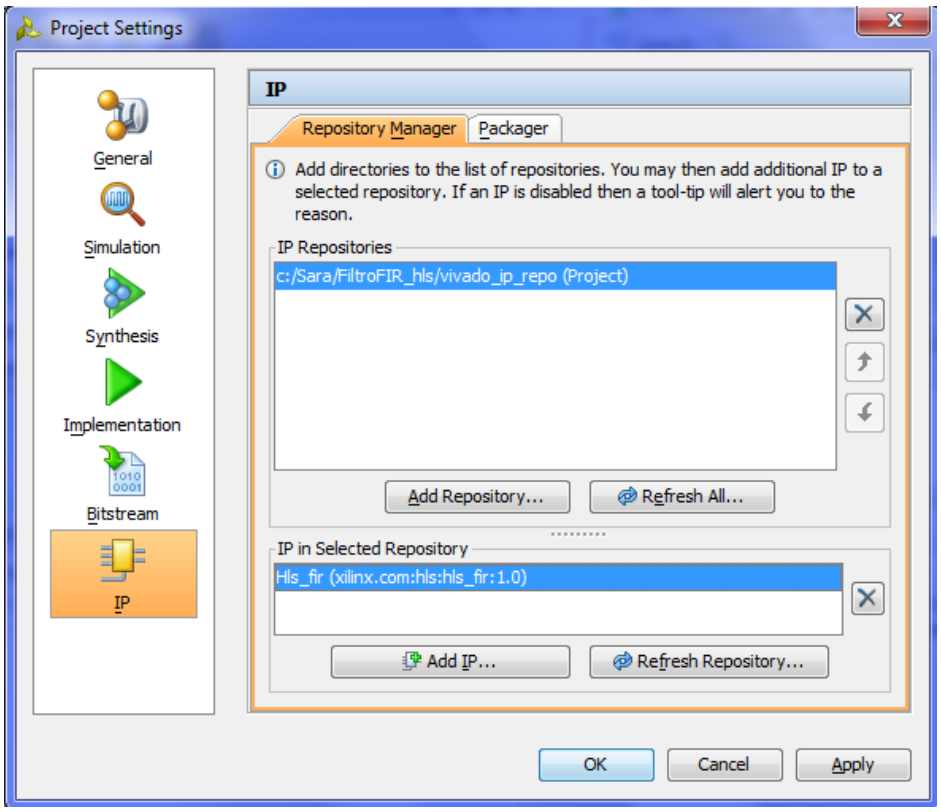


FIGURA 169. VENTANA IP SETTINGS CON EL BLOQUE IP INCORPORADO

Y desde *IP Catalog* de Vivado se tendrá acceso ahora al bloque IP importado, *Hls_fir*.

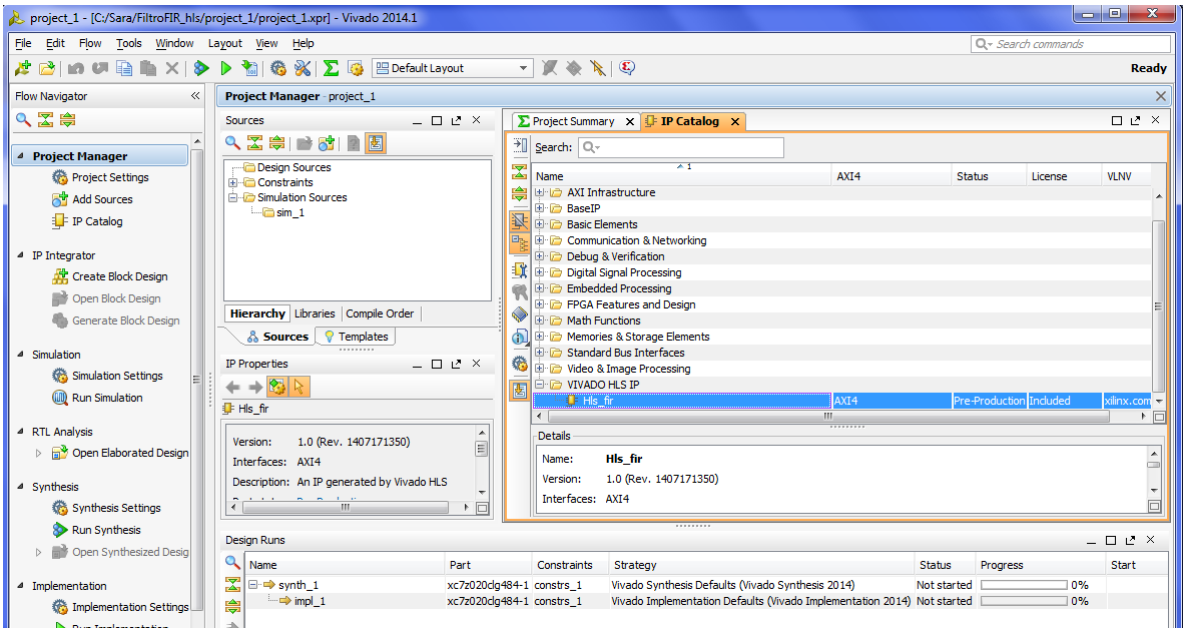


FIGURA 170. BLOQUE IP CREADO CONTENIDO DENTRO DE IP CATALOG DE VIVADO

Ya es posible crear el bloque de diseño donde comunicar periférico y procesador. Para crear un nuevo bloque de diseño, desde el explorador de proyecto se selecciona *Create Block Design* en la sección *IP Integrator*, se escoge un nombre para el diseño, en este caso *Zynq_design*, y aparecerá un espacio en blanco en el panel de información, bajo la pestaña *Diagram*.

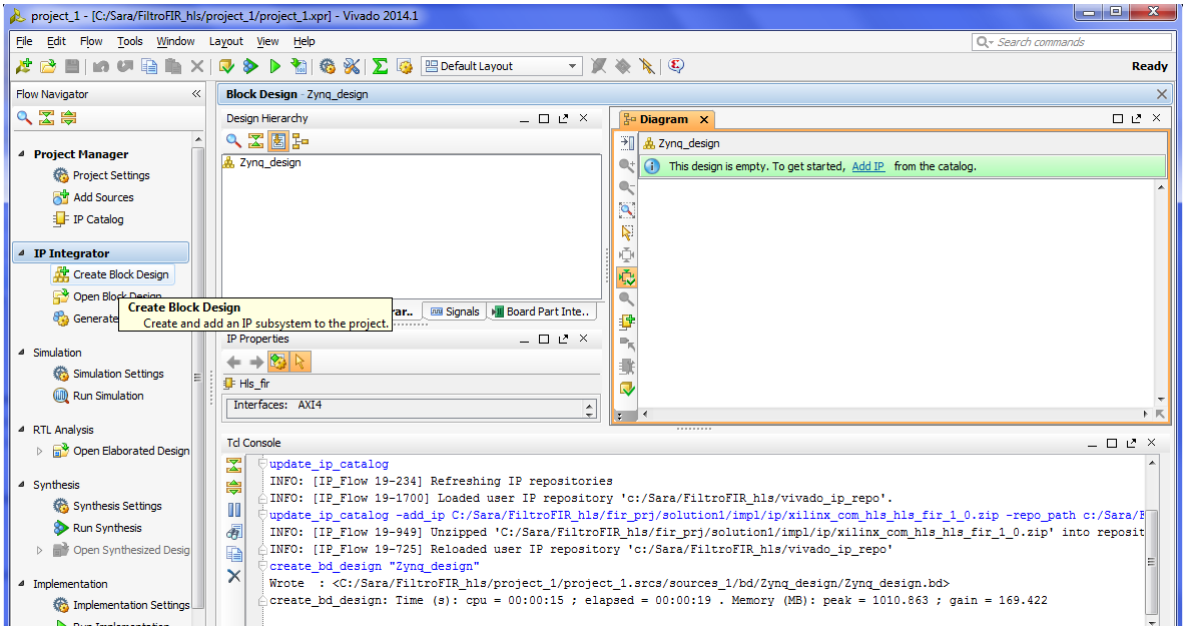


FIGURA 171. CREAR UN NUEVO BLOQUE DE DISEÑO EN VIVADO

Puede leerse el aviso en el margen superior "*This design is empty. To get started, Add IP from the catalog.*" Desde el vínculo *Add IP* se abre el buscador de IP's y debe insertarse primero el bloque ZYNQ7 Processing System que representa al dispositivo maestro.

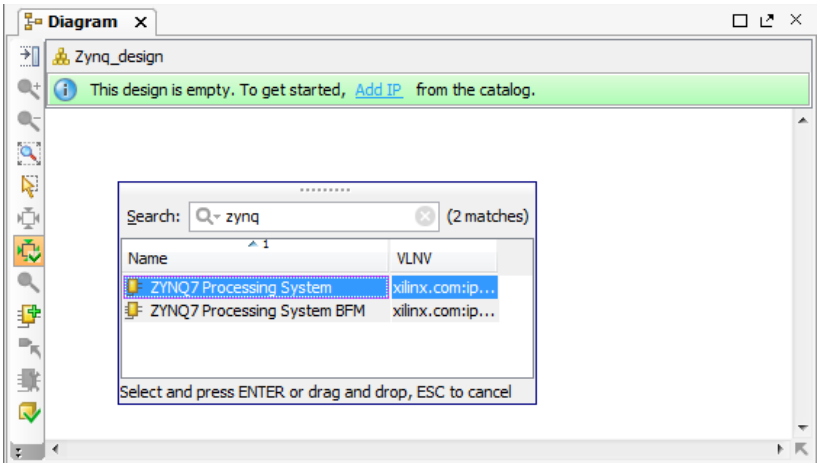


FIGURA 172. INSERTAR EL BLOQUE CORRESPONDIENTE A LA PARTE PS EN EL BLOQUE DE DISEÑO

En el diseño aparece entonces el bloque *processing_system7_0*, sobre el cual deben realizarse unos ajustes, abriendo con doble clic su configuración.

En la ventana *Re-customize IP* debe seleccionarse en *Presets* la tarjeta ZC702.

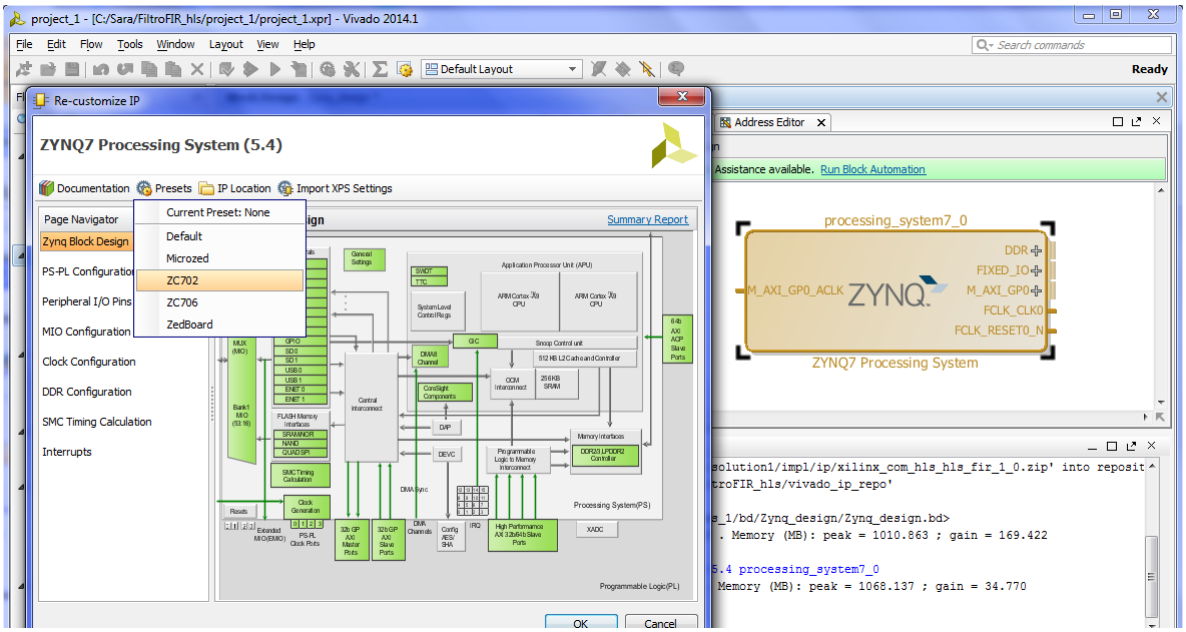


FIGURA 173. AJUSTES DEL BLOQUE PS - TARJETA ZC702

En la opción *MIO Configuration* > *Application Proccesor Unit* desactivar los timers que aparezcan activos.

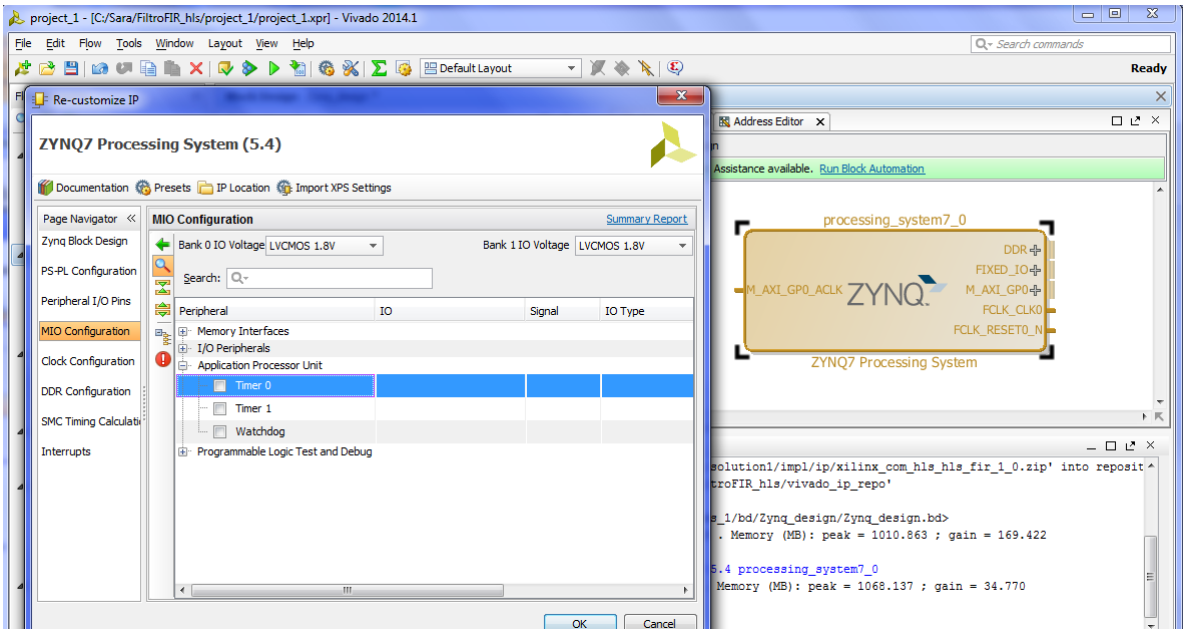


FIGURA 174. AJUSTES DEL BLOQUE PS - DESACTIVAR TIMERS

Y por último, en la opción *Interrupts*, seleccionar *Fabric interrupts* y activar, en *PL-PS Interrupt Ports*, la opción *IRQ_F2P[15:0]* para habilitar las interrupciones del periférico al micro.

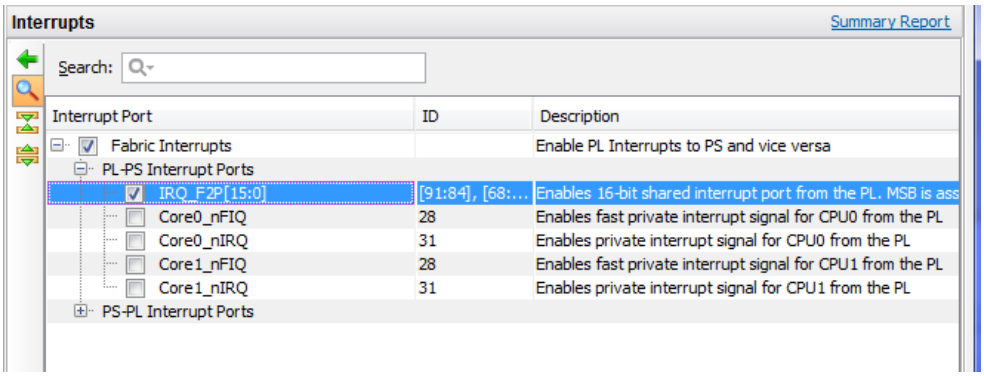


FIGURA 175. AJUSTES DEL BLOQUE PS - HABILITAR INTERRUPCIONES DEL PERIFÉRICO (PL A PS)

Al aplicarse esta configuración sobre el bloque ZYNQ7 Processing System aparece un nuevo aviso en el margen superior: *"Designer Assistance available. Run Block Automation"*.

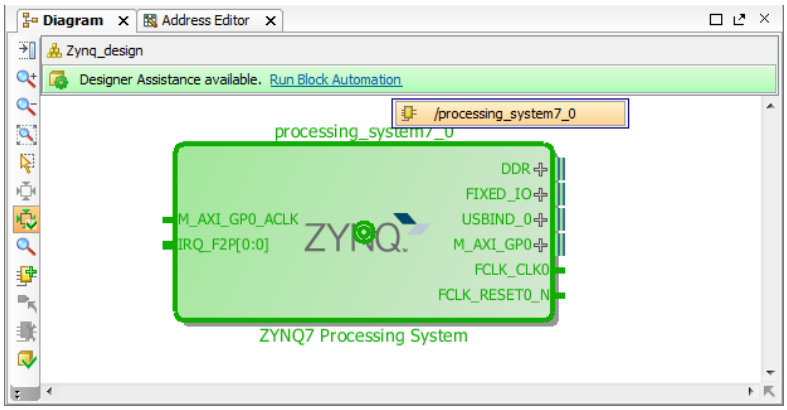


FIGURA 176. AJUSTES DEL BLOQUE PS - RUN BLOCK AUTOMATION

Desde el vínculo *Run Block Automation*, asociado a /processing_system7_0 se abre un asistente para generar las conexiones externas para *Fixed_IO* y memoria *DDR*. Es importante desmarcar la opción *Apply Board Preset* para que la configuración realizada previamente en *Re-customize IP* para temporizadores e interrupciones se mantenga. El resto de opciones se dejan por defecto.

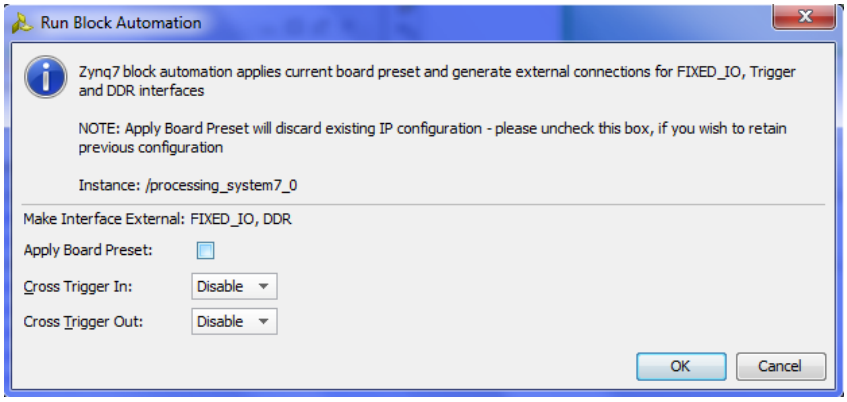


FIGURA 177. AJUSTES DEL BLOQUE PS - GENERAR CONEXIONES EXTERNAS PARA FIXED_IO Y MEMORIA DDR

Tras este procedimiento, el bloque de procesador del diagrama debe tener el aspecto de la Figura 178.

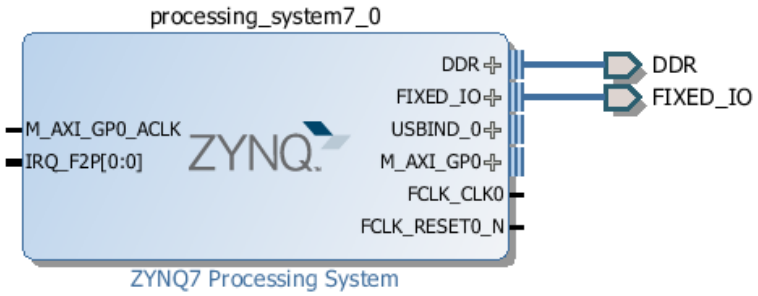


FIGURA 178. ASPECTO FINAL DEL BLOQUE DE PROCESADOR EN EL BLOQUE DE DISEÑO CREADO

A continuación, se añade el bloque IP del periférico haciendo clic con el botón derecho sobre un hueco vacío en el diseño y escogiendo la opción *Add IP*. Nuevamente aparecerá el buscador y localizando *Hls_fir* se inserta en el diseño, apareciendo un bloque *hls_fir_0* junto al bloque anterior *processing_system7_0*.

En este momento aparece el aviso *"Designer Assistance available. Run Connection Automation"*. Este aviso aparece porque Vivado es consciente de la interfaz AXI4-Lite que fue incluida en los puertos E/S del filtro, es decir los puertos "*x*" e "*y*", y es capaz de realizar de forma automática la conexión de esta interfaz del dispositivo esclavo, de nombre *S_AXI_HLS_FIR_PERIPH_BUS*, con la interfaz del dispositivo maestro *M_AXI_GP0*. Ambas han sido recuadradas en la Figura 179.

Nótese que el nombre de la interfaz AXI4 del bloque *hls_fir_0* se corresponde con el nombre que se dio a dicha interfaz al insertar las directivas en Vivado HLS.

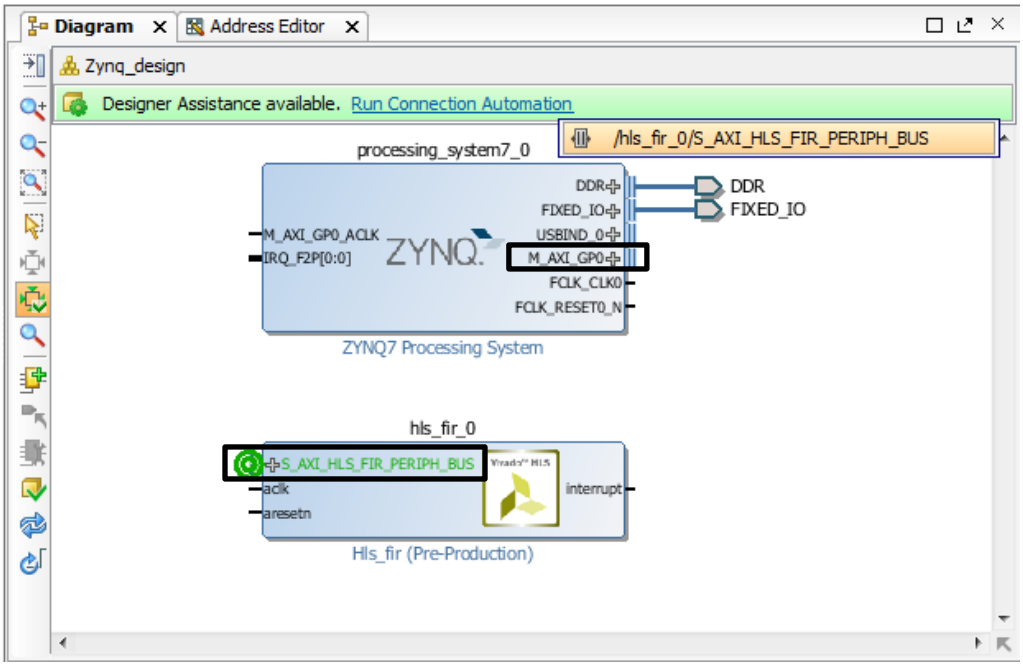


FIGURA 179. CONEXIÓN PARA LA INTERFAZ DE COMUNICACIÓN AXI4-LITE

A partir del vínculo *Run Connection Automation* referido a la interfaz del periférico, se abre un asistente para la conexión, en el cual debe dejarse la opción *Auto* por defecto.

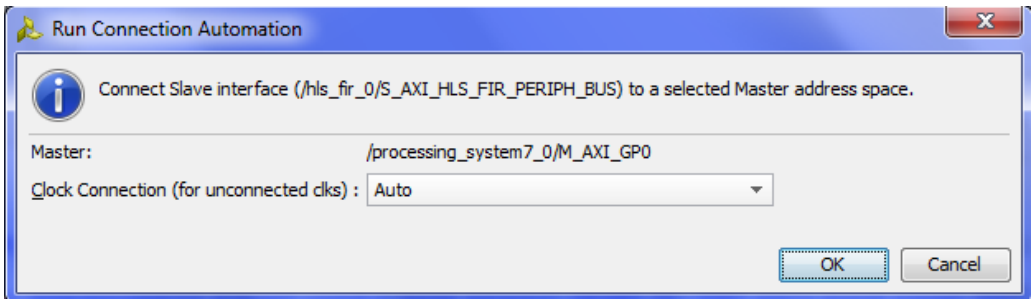


FIGURA 180. REALIZAR LAS CONEXIONES AUTOMÁTICAMENTE - RUN CONNECTION AUTOMATION

Este proceso realiza de forma automática las conexiones necesarias, incluyendo en el diseño los bloques AXI Interconnect (*processing_system7_0_axi_periph*) y Processor System Reset (*rst_processing_system7_0_50M*). La única conexión que debe realizar el usuario es aquella que conecta la salida de interrupción del bloque *hls_fir_0* a la entrada *IRQ_F2P[0:0]* del bloque *processing_system7_0*, como puede verse en la Figura 181.

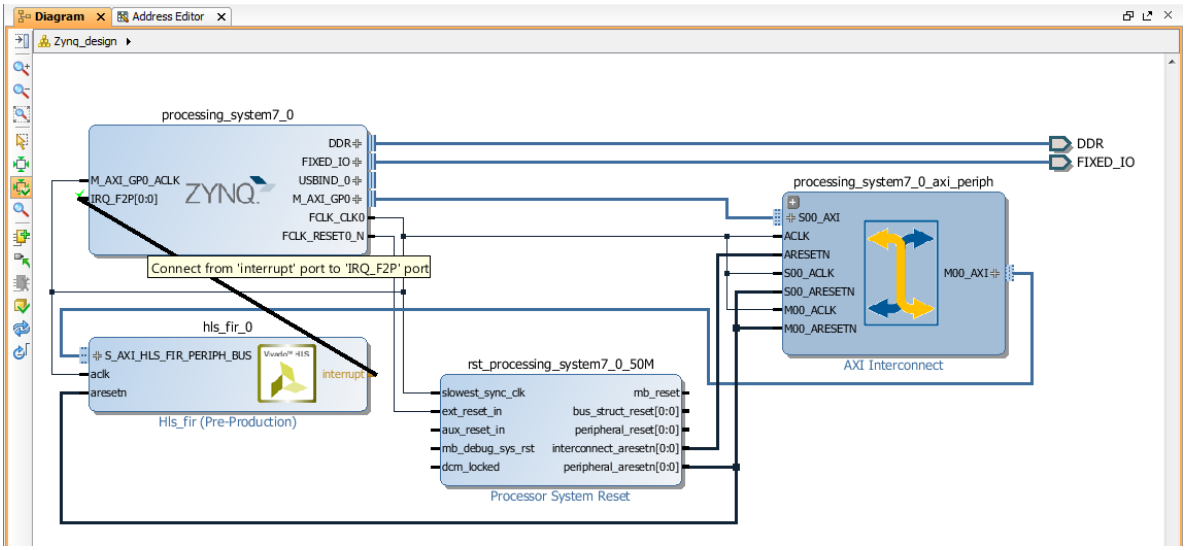


FIGURA 181. CONEXIÓN DE LA SALIDA DE INTERRUPCIÓN DEL PERIFÉRICO AL PROCESADOR

En este punto ya está conectado el periférico que se ha diseñado al bloque del procesador. Debe comprobarse que se ha asignado un rango de direcciones a *hls_fir_0* en la pestaña *Address Editor*, situada al lado derecho de *Diagram*, y si no es así generarlo mediante el botón *Auto Assign Address*.

Para finalizar, se valida el diseño con el botón *Validate Design* de la barra de herramientas, y si no existen errores se guarda el diseño con el botón *Save Block Design*.

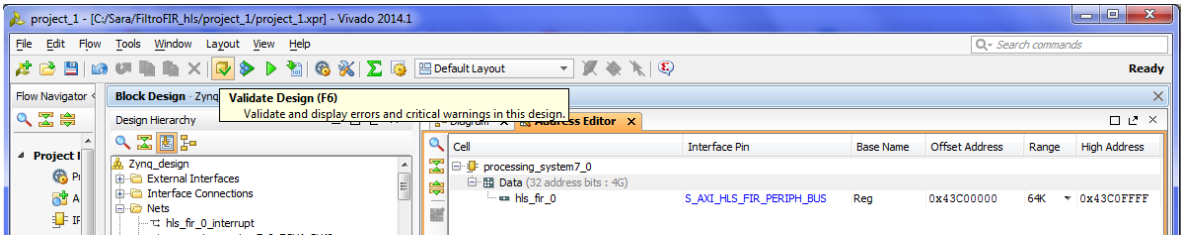


FIGURA 182. VALIDACIÓN DEL DISEÑO CREADO

Antes de proceder con la generación del *bitstream*, es necesario generar los ficheros de implementación del diseño que se acaba de crear, en el panel de fuentes del proyecto, mediante el botón derecho seleccionando *Generate Output Products*.

De esta forma se generan los ficheros para la síntesis, implementación y simulación del diseño realizado.

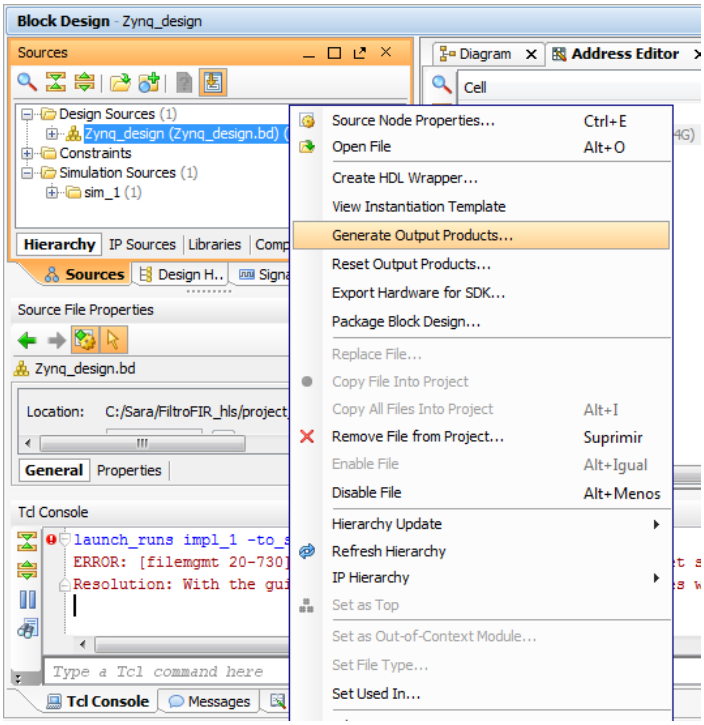


FIGURA 183. GENERAR FICHEROS DE SALIDA DEL BLOQUE CREADO

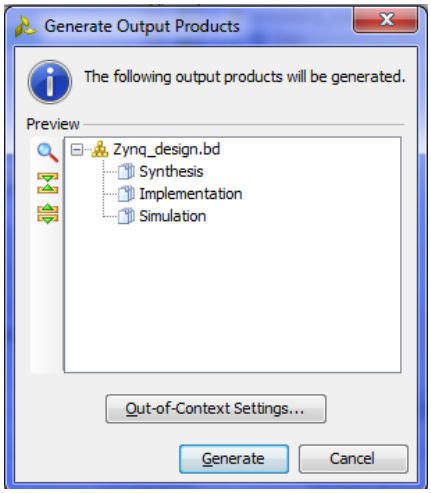


FIGURA 184. GENERACIÓN DE FICHEROS PARA SÍNTESIS, IMPLEMENTACIÓN Y SIMULACIÓN

También es necesario crear una envolvente HDL como módulo top_level para la síntesis e implementación. Para ello se selecciona *Create HDL Wrapper* y en la ventana que se abre se deja la opción por defecto, *Let Vivado manage wrapper and auto-update*, para que Vivado genere el fichero *Zynq_design_wrapper.v* sobre el diseño *Zynq_design.bd* en el panel de fuentes.

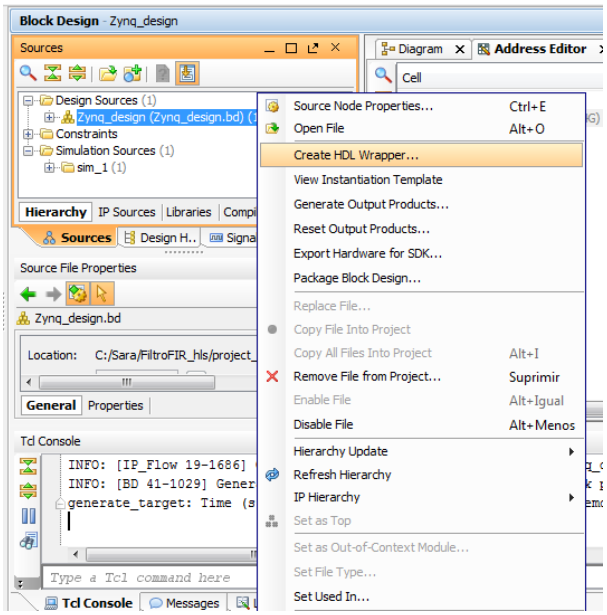


FIGURA 185. GENERACIÓN DEL FICHERO HDL CORRESPONDIENTE AL DISEÑO CREADO

Ya es posible generar el *bitstream* del diseño para posteriormente exportarlo a la herramienta de desarrollo software SDK.

En el explorador de proyecto aparece la opción *Generate Bitstream* bajo la pestaña *Program and Debug*. Al seleccionarlo aparecerá una ventana informando de que aún no existen resultados de implementación y que es necesario realizar la síntesis y la implementación del diseño como paso previo a la generación del *bitstream*. Aceptando comenzará automáticamente el proceso de síntesis, que continuará con la implementación y finalmente generará un fichero *bitstream*.

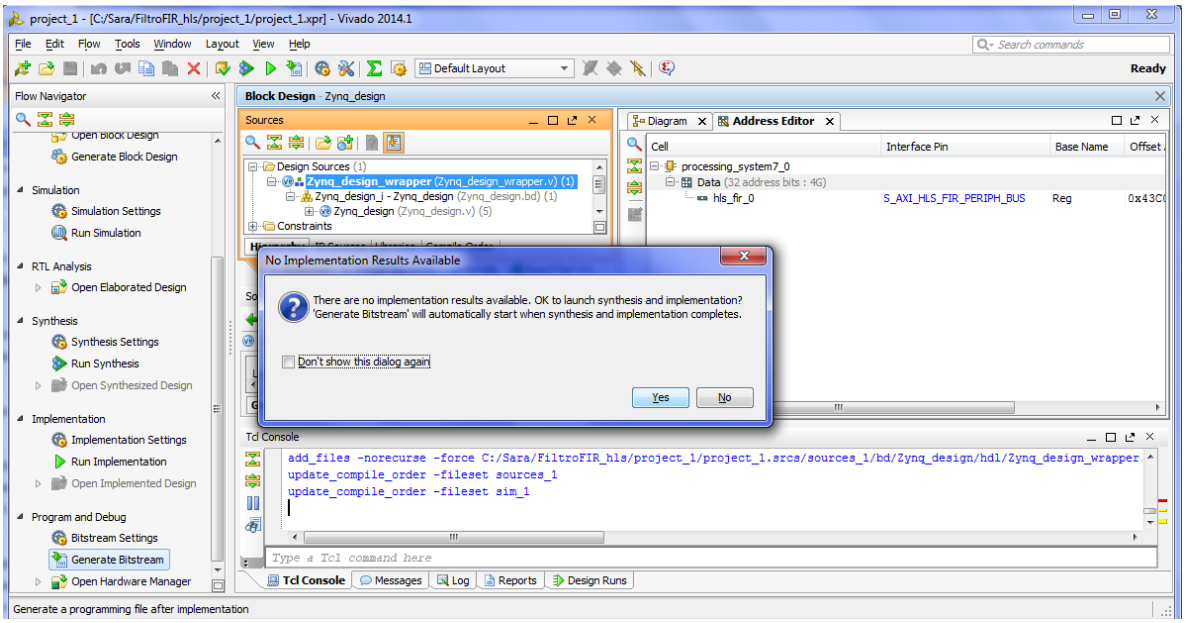


FIGURA 186. LANZAR LOS PROCESOS DE SÍNTESIS, IMPLEMENTACIÓN Y GENERACIÓN DEL BITSTREAM

Finalizados todos los procesos, el proceso de generación del *bitstream* devuelve un mensaje preguntando al usuario si desea realizar alguna acción de las que especifica. Entre esas acciones se encuentra abrir el diseño implementado (*Open Implemented Design*), la cual se recomienda escoger.

Esta recomendación se debe a que, en el momento de exportar el diseño a SDK, es necesario que se encuentren abiertos en Vivado el diseño implementado y el diagrama del diseño, que puede abrirse desde la pestaña *Sources* en el panel de fuentes. Si estos diseños no se encuentran abiertos aparecerá un mensaje de error.

Para exportar el diseño a SDK debe seleccionarse *File>Export>Export Hardware for SDK* y en la ventana que aparece asegurar que estén marcadas todas las opciones: *Export Hardware, Include bitstream* y *Launch SDK.*

DISEÑO SOFTWARE EN SDK

Como ya se ha visto, Software Development Kit es una herramienta de Xilinx que permite añadir funcionalidad software a la parte PS de la tarjeta. A partir de una plataforma hardware, que en este caso ha sido exportada desde Vivado, se creará sobre ella una aplicación software para comunicarse con el periférico desde el procesador.

Al abrirse SDK únicamente se encuentra en el proyecto el módulo hardware del diseño, *hw_platform_0*, que puede verse en el explorador de proyecto y muestra información del diseño, mapas de direcciones y bloques IP incluidos en el diseño.

Para añadir la aplicación sobre esa plataforma hardware puede hacerse desde *File>New>Application Project*.

Se abrirá una ventana donde indicar el nombre que se quiere dar a la aplicación, en este caso *Filter*, lenguaje C, y en *Board Support Package* dejar marcada la opción *Create New* para que además se cree previamente una plataforma software sobre la que se encuentre la aplicación *Filter*.

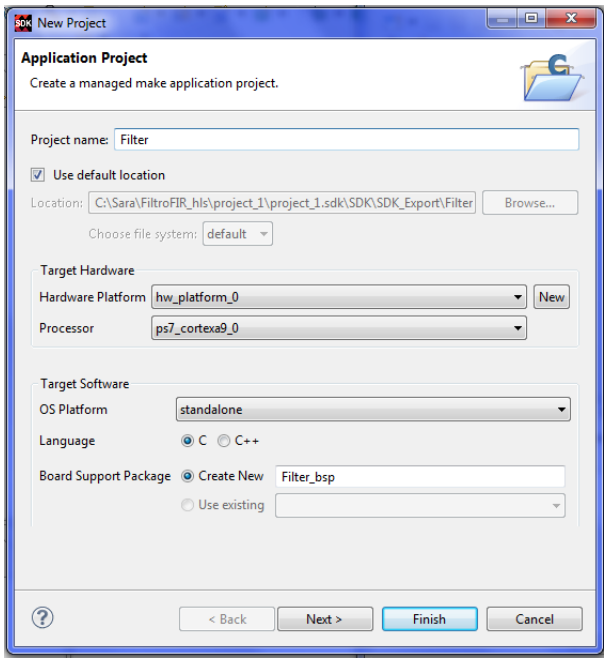


FIGURA 187. NUEVA APLICACIÓN SOBRE EL MÓDULO HARDWARE EXPORTADO DE VIVADO

Al hacer clic en *Next* se abre la ventana para seleccionar alguna plantilla existente. Como paso previo se puede escoger la plantilla *Hello World* para probarla en la FPGA y comprobar que funciona correctamente, como se ha explicado en capítulos anteriores, y posteriormente escribir sobre ella el código software de la aplicación que se desea realizar.

Una vez que aparezca en la consola el mensaje *"Finished building libraries"* se observará la siguiente jerarquía en el explorador de proyecto:

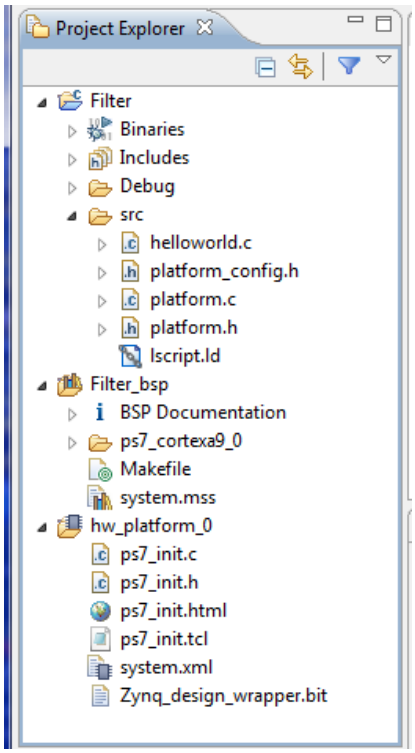


FIGURA 188. EXPLORADOR DE PROYECTO CON EL MÓDULO HARDWARE (HW_PLATFORM_0), LA PLATAFORMA SOFTWARE (FILTER_BSP) Y LA APLICACIÓN SOFTWARE (FILTER)

El siguiente paso es renombrar el fichero *helloworld.c* con la opción *Rename* a *filter.c* y sobre ese fichero escribir el código correspondiente.

Para empezar, hay que volver la vista atrás al apartado de *Interfaces* en el apartado 3.2.1, donde se habla de la interfaz AXI4-Lite. Se comentó que en el momento de empaquetado del diseño en un bloque IP para su exportación en Vivado HLS, eran creados unos drivers en el directorio de implementación, y que se emplean desde SDK para la comunicación con el dispositivo esclavo mediante la interfaz AXI4-Lite.

Para un diseño de nombre *Hls_fir*, los drivers que se van a emplear tendrán los nombres *XHls_fir.h*, *XHls_fir.c* y *XHls_fir_hw.h* (ver Tabla 8).

El fichero *XHls_fir.h* contiene la definición de las funciones que permiten la comunicación con el periférico, pudiendo verse el cuerpo de las funciones en *XHls_fir.c* para comprender su funcionamiento, no obstante, estas funciones ya fueron aclaradas de modo genérico en dicho apartado de *Interfaces* y puede retrocederse ahí para su consulta.

En cuanto al fichero *XHls_fir_hw.h* sirve de consulta tanto para el usuario como para el resto de ficheros para saber a qué dirección de memoria acceder cuando se quiera leer o escribir una señal de control, un registro de interrupción o una señal de datos.
Para que resulte más claro al usuario, se muestra su contenido y cómo interpretarlo:

```
// HLS_FIR_PERIPH_BUS
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x10 : Control signal of y
//        bit 0  - y_ap_vld (Read/COR)
//        others - reserved
// 0x14 : Data signal of y
//        bit 12~0 - y[12:0] (Read)
//        others   - reserved
// 0x18 : reserved
// 0x1c : Data signal of x
//        bit 7~0 - x[7:0] (Read/Write)
//        others  - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on
Handshake)

#define XHLS_FIR_HLS_FIR_PERIPH_BUS_ADDR_AP_CTRL 0x00
#define XHLS_FIR_HLS_FIR_PERIPH_BUS_ADDR_GIE     0x04
#define XHLS_FIR_HLS_FIR_PERIPH_BUS_ADDR_IER     0x08
#define XHLS_FIR_HLS_FIR_PERIPH_BUS_ADDR_ISR     0x0c
#define XHLS_FIR_HLS_FIR_PERIPH_BUS_ADDR_Y_CTRL  0x10
#define XHLS_FIR_HLS_FIR_PERIPH_BUS_ADDR_Y_DATA  0x14
#define XHLS_FIR_HLS_FIR_PERIPH_BUS_BITS_Y_DATA  13
#define XHLS_FIR_HLS_FIR_PERIPH_BUS_ADDR_X_DATA  0x1c
#define XHLS_FIR_HLS_FIR_PERIPH_BUS_BITS_X_DATA  8
```

La utilidad que tiene consultar este fichero es que aquí pueden localizarse las definiciones que se emplean en el cuerpo de las funciones de comunicación que se encuentran en el fichero de drivers *XHls_fir.c*, por lo que para entender lo que realiza una de dichas funciones basta con consultar a qué señal está accediendo en este fichero de hardware.

Por ejemplo, para la función que permite conocer cuándo el periférico ha terminado de operar mediante la señal ap_done se tiene la siguiente función en *XHls_fir.c*:

```
u32 XHls_fir_IsDone(XHls_fir *InstancePtr) {
    u32 Data;
    Xil_AssertNonvoid(InstancePtr != NULL);
    Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
    Data=XHls_fir_ReadReg(InstancePtr->Hls_fir_periph_bus_BaseAddress,
XHLS_FIR_HLS_FIR_PERIPH_BUS_ADDR_AP_CTRL);
    return (Data >> 1) & 0x1;
}
```

Lo que se tiene que interpretar de esta función es que en la sentencia en que hace uso de la función XHls_fir_ReadReg va a leer de una señal, para saber de cuál hay que fijarse en los argumentos pasados a la función, el primero referido a la dirección base del periférico y el segundo al offset de donde se va a leer. Este offset se consulta en el fichero XHls_fir_hw.h que corresponde al 0x00 (sentencia `#define XHLS_FIR_HLS_FIR_PERIPH_BUS_ADDR_AP_CTRL 0x00`), por lo que se trata de las señales de control, almacenando el valor de todas ellas en la variable Data.

Al retornar valor es cuando escoge el bit de la señal que interesa en ese momento, que es la señal ap_done, por ello desplaza Data una posición a la derecha y emplea una operación and con la unidad para quedarse únicamente con esa posición correspondiente a la señal de finalización, ap_done (`bit 1  - ap_done (Read/COR)`).

Por otro lado se encuentra el fichero *xparameters.h*, que incluye definiciones de parámetros del sistema así como los identificadores y las direcciones base de memoria de los periféricos asociados al procesador (ver Tabla 1).

Desde el fichero de aplicación *filter.c* se incluirán las cabeceras *xparameters.h* y *XHls_fir.h*, la cual está incluida también en *XHls_fir.c* e incluye a *XHls_fir_hw.h*. Observando estas cabeceras, el desarrollador software debe realizar una aplicación que se comunique con el periférico creado empleando las funciones que se encuentran disponibles.

Para este diseño la comunicación entre procesador y periférico está basada en funciones para la inicialización del periférico, comprobación de su estado, paso de parámetros de entrada, activación y recogida de resultados. La Figura 189 muestra el intercambio de comunicación que seguirá la aplicación.

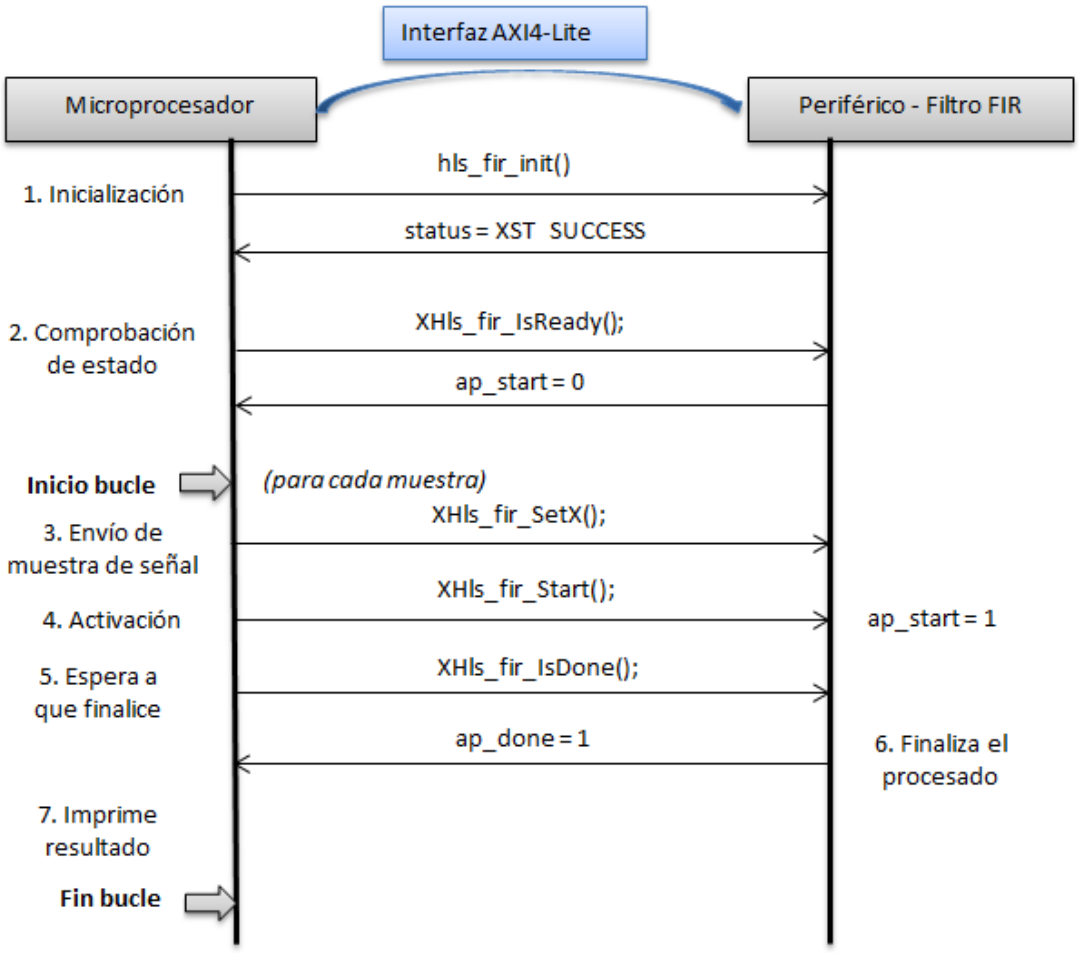


FIGURA 189. INTERCAMBIO DE COMUNICACIÓN QUE SE DARÁ ENTRE EL MICROPROCESADOR Y EL PERIFÉRICO

El software para la comunicación con el bloque HLS, descrito en *filter.c*, comienza con las cabeceras que recogen las funciones estándar de C como *<stdio.h>* y *<stdlib.h>*, *<stdbool.h>* para operaciones booleanas, la cabecera *platform.h* para habilitar o deshabilitar la caché e inicializar la uart, y las cabeceras *xparameters.h*, con las definiciones de los parámetros para los periféricos del procesador, y *XHls_fir.h*, con las definiciones de las funciones para la comunicación con el bloque HLS.

```
#include <stdio.h>
#include "platform.h"
#include <stdlib.h>
#include <stdbool.h>
#include "xparameters.h"
#include "XHls_fir.h"
```

Además, se define una variable que hace referencia al periférico para utilizarla en las llamadas a las funciones de los drivers. Esta variable es de tipo estructura XHls_fir, y contiene la dirección base del periférico y el flag *IsReady*.

```
XHls_fir HlsFir;
```

Y se añade, previo a la función *main()*, la definición de la función de configuración del periférico, cuyo cuerpo se verá posteriormente.

```
int hls_fir_init(XHls_fir *hls_firPtr);
```

A continuación se desarrolla la función *main()* haciendo uso de diferentes funciones para probar el hardware. Estas funciones han sido descritas en

```
int main()
{
     print("........FIR FILTER........ \n\r");
     const int  SAMPLES=70;
     int signal=0;
     int i;
     int status;
     int result;

     init_platform();

     //Initialization
     status = hls_fir_init(&HlsFir);
     if(status != XST_SUCCESS)
     {
         print("HLS peripheral setup failed\n\r");
         exit(-1);
     }
     else
       print("Initialization was completed without errors.\n\r");


     if (XHls_fir_IsReady(&HlsFir))
           print("HLS peripheral is ready. Starting... \n\r");
     else
     {
           print("!!! HLS peripheral is not ready! Exiting...\n\r");
           exit(-1);
     }


     for (i=0;i<SAMPLES;i++)
     {
         signal = signal+1;
```

```
        XHls_fir_SetX(&HlsFir,signal);

        // Start the device and read the results
        XHls_fir_Start(&HlsFir);

        do
        {
             result = XHls_fir_GetY(&HlsFir);
        }while(XHls_fir_IsDone(&HlsFir)==0);


        if((i==6)|(i==7)|(i==8))
           printf("%i %d -%d\n",i,signal,((~result)+1)&(0x07));
        else
           printf("%i %d %d\n",i,signal,result);
     }

     print("........END........ \n\r");
     cleanup_platform();
     return 0;
}
```

Puede verse que lo primero que se realiza, tras definir las variables que se van a emplear en el programa, es la llamada a la función *hls_fir_init*. Esta función para la configuración e inicialización del periférico se encuentra externa a la función *main()* y tiene el siguiente cuerpo:

```
int hls_fir_init(XHls_fir *hls_firPtr)
{

   XHls_fir_Config *cfgPtr;
   int status;

   cfgPtr = XHls_fir_LookupConfig(XPAR_XHLS_FIR_0_DEVICE_ID);

   if (!cfgPtr)
   {
      print("ERROR: Lookup of accelerator configuration failed.\n\r");
      return XST_FAILURE;
   }
   else
       print("SUCCESS: Configuration for your device was found.\n\r");

   status = XHls_fir_CfgInitialize(hls_firPtr, cfgPtr);

   if (status != XST_SUCCESS) {
      print("ERROR: Could not initialize accelerator.\n\r");
      return XST_FAILURE;
   }
```

```
    else
        print("SUCCESS: Your device was initialized.\n\r");

    return status;
}
```

Básicamente, hace uso de las funciones *XHls_fir_LookupConfig* y *XHls_fir_CfgInitialize*, la primera de ellas para obtener la información necesaria para la configuración del periférico a partir de su identificador (el cual se encuentra definido en *xparameters.h*), almacenándola en la estructura XHls_fir_Config, que contiene la dirección base física del dispositivo, y la segunda utilizando esa información para configurar el dispositivo.

Una vez realizada la configuración, si todo está correcto, se continúa comprobando si el dispositivo está listo para aceptar una nueva entrada. Eso se hace mediante la función *XHls_fir_IsReady*, que devuelve el valor *!(ap_start & 0x1)*, por lo que si la señal ap_start se encuentra a nivel bajo, el periférico se encuentra disponible para comenzar a operar, ya que esta señal se activa cuando el periférico está en uso.

Hasta este punto se ha utilizado la comunicación entre procesador y periférico para configuración, inicialización y comprobaciones previas. Es en este momento cuando se comienza a probar la funcionalidad hardware.

Para ello, siguiendo con la función *main()*, se entra en un bucle *for* con tantas iteraciones como muestras de la señal de entrada se quieran pasar al filtro, y en dicho bucle *for* se comienza pasando el valor escalar de la muestra de la señal al periférico a través de la función *XHls_fir_SetX*, se da la orden al periférico para que comience a operar mediante la función *XHls_fir_Start*, que pone a nivel alto la señal *ap_start* indicando que a partir de este momento se encuentra ocupado y no puede recibir nuevas entradas, y se espera a que el periférico indique al procesador que ha finalizado de procesar esa entrada, comprobando el valor de la señal *ap_done* retornada por la función *XHls_fir_IsDone* mediante un bucle do while.
Una vez haya finalizado, puede recogerse el resultado del filtro mediante la función *XHls_fir_GetY* e imprimirlo por el terminal, donde aparecerán todas las muestras de entrada filtradas, de igual forma que se veía la simulación de Vivado HLS.

Otro aspecto a tener en cuenta, es que las funciones que sirven para la comunicación, definidas en *XHls_fir.h*, retornan valores de tipo unsigned int de 32 bits, lo que implica que el usuario debe ser quien escoja los bits oportunos para visualizar el resultado correctamente.

En este caso, la función que recoge desde el procesador los valores de salida del filtro viene definida como:

```
u32 XHls_fir_GetY(XHls_fir *InstancePtr);
```

Por este motivo, al imprimir resultados, se distinguen las iteraciones *6, 7* y *8*, cuyo resultado se conoce por Vivado HLS que es un número negativo, ya que para obtener el valor correcto desde esos 32 bits ha sido necesario emplear la herramienta de depuración y observar el número en binario que retornaba la función *XHls_fir_GetY*, concluyendo que es necesario hacer el complemento a dos del valor retornado y escoger los tres últimos bits, mediante la sentencia `((~result)+1)&(0x07)`, para visualizar el resultado correctamente.

Llegados a este punto se ha visto cómo diseñar el software necesario para probar el hardware y se considera completo el proyecto en SDK, por lo que se puede proceder a verificar su funcionamiento en la FPGA [8].

En el panel inferior debe aparecer la pestaña Terminal (si esto no es así, puede abrirse desde *Window>Show View>Terminal*) y en ella se encuentra un icono de ajustes, *Settings*, que permite conectarse al puerto serie donde esté conectada físicamente la tarjeta.

Puede verse a qué puerto corresponde desde *Inicio* > *Equipo* > (botón derecho) *Administrar* > *Administrador de dispositivos* > *Puertos (COM y LPT)* > *USB Serial Port (COM X)*.

En los ajustes del *Terminal*, como puede verse en la Figura 190, se escoge conexión serie (*Serial*), el puerto que haya asignado el ordenador a la tarjeta (*COM13* en este caso), la tasa máxima disponible, 115200 baudios, y resto de las opciones por defecto.

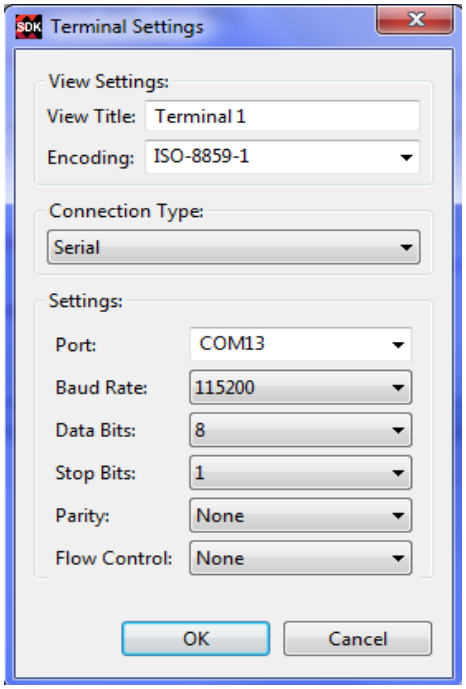


FIGURA 190. AJUSTES DEL TERMINAL

Al conectarse aparecerá indicado en la ventana *Serial: (COM13, 115200, 8, 1, None, None – CONNECTED)*.

Es el momento de programar la FPGA con el fichero *bitstream* que fue generado en Vivado. Para ello se encuentra el icono *Program FPGA* en la barra de herramientas de SDK (si esto no es así, puede hacerse desde *Xilinx Tools>Program FPGA*), y en la ventana que se abre para su configuración asegurarse de que la plataforma hardware es la perteneciente al diseño y que en *Bitstream* se encuentra el *.bit generado, el cual puede localizarse desde el directorio del proyecto *project_1\ project_1.sdk\ SDK\ SDK_Export\ hw\ Zynq_design_wrapper.bit*.

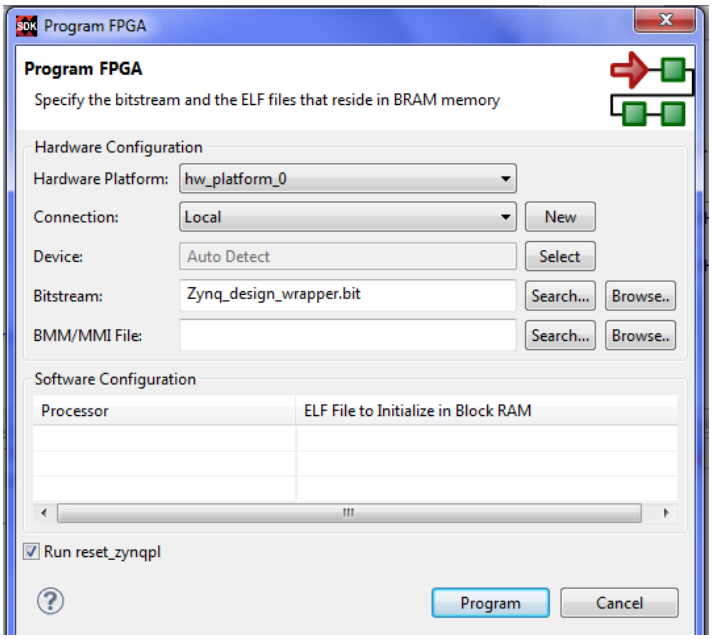


FIGURA 191. PROGRAMACIÓN DE LA FPGA CON EL FICHERO BITSTREAM

Al programarla deberá encenderse el LED LD12 Done de color azul indicando que ha sido programada correctamente.

Ahora para lanzar el programa en el procesador se escoge *Filter>Run As>Launch on Hardware*, y se espera a que se cargue el fichero *.elf que contiene a la aplicación software. Puede verse en la esquina inferior derecha de la interfaz de SDK el aviso *Launching Filter.elf (%)*.

Cuando se haya completado, en la pestaña *Terminal* aparecerán los mensajes que se suceden a lo largo de la aplicación, entre ellos los resultados recogidos del periférico. En la Figura 192 puede verse una parte de dichos resultados, mostrados como iteración-señal-resultado.

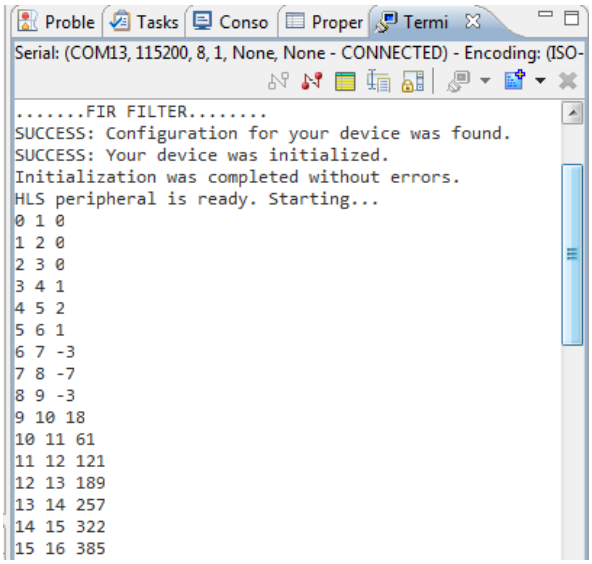


FIGURA 192. RESULTADOS DE LAS MUESTRAS FILTRADAS RETORNADAS POR EL PERIFÉRICO

3.2.3. **Otras alternativas de diseño**

Diseño en SDK con interrupción

Otra opción a la hora de diseñar el software en SDK es hacer uso de las funciones disponibles para manejar la interrupción del periférico al procesador. Esto es posible porque en el diseño de Vivado HLS se incorporó el puerto *return* a la interfaz AXI4Lite, permitiendo una salida de interrupción que posteriormente se conectó al puerto de interrupciones del procesador.

Existe un driver para el controlador de interrupciones del procesador, recogido en la cabecera *xscugic.h*, que debe ser incluida en el diseño.

```
#include "xsugic.h"
```

Además, de igual forma que para el bloque HLS, debe definirse una variable referida al controlador de interrupciones para utilizarla en las llamadas a las funciones de los drivers. Esta variable es de tipo estructura XScuGic, y contiene un puntero a la configuración del controlador, el flag *IsReady* y las interrupciones que no son soportadas.

```
XScuGic ScuGic;
```

También son necesarias un par de variables globales para la rutina de atención a la interrupción, cuya función se verá dentro de la misma.

```
volatile static int RunHlsFir = 0;
volatile static int ResultAvailHlsFir = 0;
```

Y se añaden, previas a la función *main()* y junto a la definición de la función de configuración del periférico, las definiciones de las funciones necesarias para configurar y atender a las interrupciones.

```
int setup_interrupt();
void hls_fir_start(void *InstancePtr);
void hls_fir_isr(void *InstancePtr);
```

A continuación, dentro de la función *main()*, la modificación de código necesaria para obtener los resultados mediante interrupción consiste únicamente en configurar las interrupciones a través de una llamada a la función *setup_interrupt*, y posteriormente, dentro del bucle *for*, llamar a la función *hls_fir_start* para habilitar las interrupciones y activar el bloque HLS, y recoger los resultados con *XHls_fir_GetY* cuando el flag *ResultAvailHlsFir* se encuentra a nivel alto, lo cual sucede en la rutina de atención a la interrupción, *hls_fir_isr*.

Para ofrecer una visión más clara, se muestra el código de la función *main()* completo modificado para trabajar con interrupciones.

```
int main()
{
     print("........FIR FILTER........ \n\r");
     const int  SAMPLES=70;
     int signal=0;
     int i;
     int status;
     int result;

     init_platform();

     //Initialization
     status = hls_fir_init(&HlsFir);
     if(status != XST_SUCCESS)
     {
         print("HLS peripheral setup failed\n\r");
         exit(-1);
     }
     else
          print("Initialization was completed without errors.\n\r");

     //Setup the interrupt
     status = setup_interrupt();
     if(status != XST_SUCCESS)
     {
         print("Interrupt setup failed\n\r");
         exit(-1);
     }

     if (XHls_fir_IsReady(&HlsFir))
           print("HLS peripheral is ready. Starting... \n\r");
     else
     {
           print("!!! HLS peripheral is not ready! Exiting...\n\r");
           exit(-1);
     }

     for (i=0;i<=SAMPLES;i++)
     {
           signal = signal+1;
           XHls_fir_SetX(&HlsFir,signal);

           hls_fir_start(&HlsFir);
           while(!ResultAvailHlsFir);
           result = XHls_fir_GetY(&HlsFir);

           if((i==6)|(i==7)|(i==8))
             printf("%i %d -%d\n",i,signal,((~result)+1)&(0x07));
```

```
            else
              printf("%i %d %d\n",i,signal,result);
     }

     print("........END........ \n\r");
     cleanup_platform();
     return 0;
}
```

La función *main()* llama a las nuevas funciones empleadas para configuración y atención de interrupciones, estando éstas declaradas fuera de la función *main()* al igual que sucedía con *hsl_fir_init* para la configuración e inicialización del periférico. Se tienen por tanto, además de *hls_fir_init*, las funciones *setup_interrupt, hls_fir_start* y *hls_fir_isr,* cuyo cuerpo es el siguiente:

```
int setup_interrupt()
{
   //This functions sets up the interrupt on the ARM
   int result;

   XScuGic_Config *pCfg = XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
   if (pCfg == NULL){
      print("Interrupt Configuration Lookup Failed\n\r");
      return XST_FAILURE;
   }

   result = XScuGic_CfgInitialize(&ScuGic,pCfg,pCfg->CpuBaseAddress);
   if(result != XST_SUCCESS){
      return result;
   }

   //Self-test
   result = XScuGic_SelfTest(&ScuGic);
   if(result != XST_SUCCESS){
      return result;
   }

   //Initialize the exception handler
   Xil_ExceptionInit();

   //Register the exception handler
   Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
(Xil_ExceptionHandler)XScuGic_InterruptHandler, &ScuGic);

   //Enable the exception handler
   Xil_ExceptionEnable();

   //Connect the Adder ISR to the exception table
```

```
   result = XScuGic_Connect(&ScuGic, XPAR_FABRIC_HLS_FIR_0_INTERRUPT_INTR,
(Xil_InterruptHandler)hls_fir_isr,&HlsFir);
   if(result != XST_SUCCESS){
      return result;
   }

   //Enable the Adder ISR
   XScuGic_Enable(&ScuGic,XPAR_FABRIC_HLS_FIR_0_INTERRUPT_INTR);

   return XST_SUCCESS;
}
```

Esta función *setup_interrupt* configura el manejador de interrupciones del procesador mediante las funciones XScuGic_LookupConfig y XScuGic_CfgInitialize, realiza una prueba de funcionamiento con la función XScuGic_SelfTest, registra en él la rutina de atención a la interrupción del periférico para permitir que sea atendida y reconocida por el manejador con las funciones Xil_ExceptionInit, Xil_ExceptionRegisterHandler,Xil_ExceptionEnable, XScuGic_Connect, y finalmente habilita el manejador de interrupciones con XScuGic_Enable.

```
void hls_fir_start(void *InstancePtr)
{
   XHls_fir *pAccelerator = (XHls_fir *)InstancePtr;
   XHls_fir_InterruptEnable(pAccelerator,1);
   XHls_fir_InterruptGlobalEnable(pAccelerator);
   XHls_fir_Start(pAccelerator);
}
```

La función *hls_fir_start* habilita las interrupciones con XHls_fir_InterruptEnable y XHls_fir_InterruptGlobalEnable, y activa el bloque HLS para que comience a operar, poniendo a nivel alto la señal *ap_start* con la función XHls_fir_Start.

```
void hls_fir_isr(void *InstancePtr){
   XHls_fir *pAccelerator = (XHls_fir *)InstancePtr;

   //Disable the global interrupt
   XHls_fir_InterruptGlobalDisable(pAccelerator);
   //Disable the local interrupt
   XHls_fir_InterruptDisable(pAccelerator,0xffffffff);
   //Clear the local interrupt
   XHls_fir_InterruptClear(pAccelerator,1);

   ResultAvailHlsFir = 1;
   //Restart the core if it should run again
   if(RunHlsFir){
      hls_fir_start(pAccelerator);
   }
}
```

Esta función corresponde a la rutina de atención a la interrupción necesaria para que el procesador responda ante una interrupción del periférico. Lo que hace es deshabilitar y limpiar la interrupción que se ha producido con XHls_fir_InterruptGlobalDisable, XHls_fir_InterruptDisable y XHls_fir_InterruptClear, y activar un flag, en este caso *ResultAvailHlsFir*, indicando que el resultado está disponible para ser retornado por el periférico, tarea que se hará posteriormente desde *main()* con la función *XHls_fir_GetY*.
Por otro lado, la otra variable global que fue definida al comienzo de la aplicación, *RunHlsFir*, sirve para avisar a la rutina de que debe habilitar interrupciones y reiniciar el periférico llamando de nuevo a la función *hls_fir_start*.

Diseño en Vivado HLS empleando tipos de coma flotante

Una vez realizado el diseño con Vivado HLS, Vivado y SDK y comprendida de qué forma puede comunicarse un periférico creado por el usuario con el micro, surge la idea de hacer un diseño más real realizando tres nuevos cambios:

1.- Aumentar el orden del filtro a 50, teniéndose 51 coeficientes.
2.- Utilizar el tipo de datos *float* para entrada y salida del filtro, coeficientes y variables internas.
3.- Emplear una señal de entrada al filtro formada por un seno al que se le añade ruido blanco uniformemente distribuido.

Empleando Matlab, se abre la herramienta fdatool y se diseña un filtro paso bajo mediante el método de la ventana usando ventana Hamming, de orden 50, frecuencia de muestro 2000Hz y frecuencia de corte 300Hz (estas frecuencias han sido seleccionadas por proporcionar el mejor filtrado de la señal senoidal con ruido que se introduce al filtro).

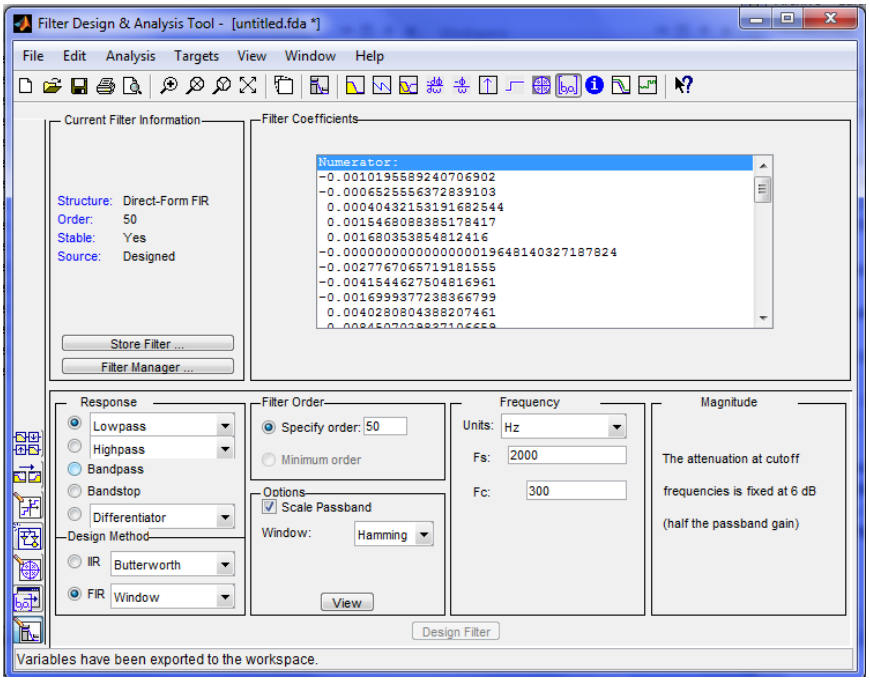


FIGURA 193. DISEÑO DEL FILTRO PARA ORDEN 50

Una vez diseñado se exportan los coeficientes en un array *Num* al espacio de trabajo de Matlab, de igual forma que para el diseño inicial. Este array de coeficientes ya se encuentra en coma flotante, por lo que se va a introducir al diseño en Vivado HLS tal cual es exportado de fdatool, sin necesidad de convertir al rango de enteros de la variable de coeficientes, ya que ésta será definida como *float*, ni, por tanto, redondear para obtener únicamente la parte entera, como se hacía con el diseño inicial.

En cuanto a la señal de entrada, se genera en Matlab una señal senoidal a 10Hz, a la cual se añade ruido blanco uniforme para observar el filtrado realizado por el periférico. Los comandos empleados en Matlab son los siguientes:

```
>> t=0:0.01:1;
>> x=sin(2*pi*10*t);
>> n=0.5*randn(size(t));
>> z=x+n;
>> figure, plot(t,x)
>> figure, plot(t,z)
>> y=filter(Num,1,z);
>> figure, plot(t,y)
```

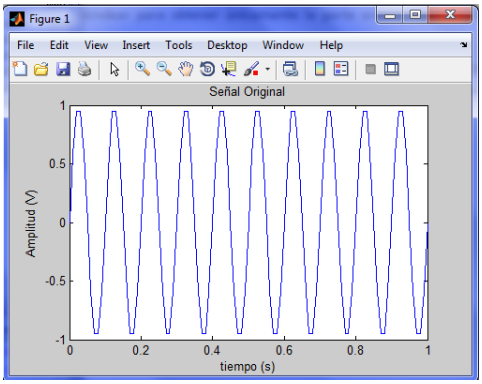


FIGURA 194. SEÑAL ORIGINAL SENOIDAL

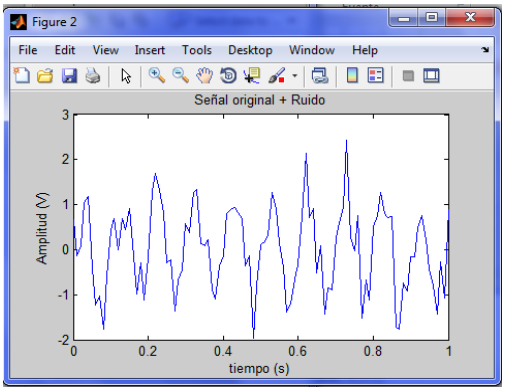


FIGURA 195. SEÑAL SENO CONTAMINADA CON RUIDO

La señal senoidal con ruido añadido será la que se introduzca en el filtro diseñado. Probando la salida que se tendrá con Vivado HLS en Matlab con la función *filter*, se observa que con los coeficientes recogidos en el array *Num*, se consigue filtrar el ruido en su mayoría, resultando la señal de la Figura 196.

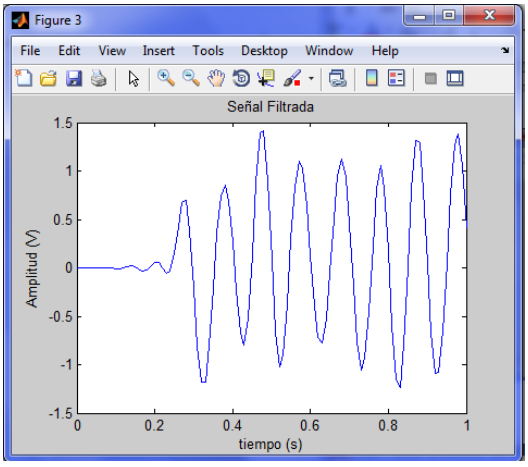


FIGURA 196. SEÑAL FILTRADA

La función principal *hls_fir* en Vivado HLS tiene entonces el siguiente código. Nótese que ahora los tipos de datos empleados son todos datos *float*, y que el array de coeficientes se corresponde con los coeficientes exportados de fdatool de Matlab. Además, se siguen empleando las mismas directivas de optimización que se tenían para el programa original con tipos de datos enteros arbitrarios.

```
/*************************************************************************
Project Name: fir_prj
Associated Filename: hls_fir.c
Family: Zynq
Device: xc7z020clg484-1
*************************************************************************/

#include "hls_fir.h"

void hls_fir (float *y, float x)
{
#pragma HLS PIPELINE II=1
#pragma HLS RESOURCE variable=x core=AXI4LiteS metadata="-bus_bundle
HLS_FIR_PERIPH_BUS"
#pragma HLS RESOURCE variable=y core=AXI4LiteS metadata="-bus_bundle
HLS_FIR_PERIPH_BUS"
#pragma HLS RESOURCE variable=return core=AXI4LiteS metadata="-bus_bundle
HLS_FIR_PERIPH_BUS"

  float c[N] = {-0.00101955892407069,-0.000652555637283910,
0.000404321531916825,0.00154680883851784,0.00168035385481242,-1.96481403271878e-
18,-0.00277670657191816,-0.00415446275048170,-0.00169993772383668,
0.00402808043882075,0.00845070298371067,0.00607046404166686,-
0.00387098195557225,-0.0143651734835567,-0.0146736172848365,7.98496381387025e-
18,0.0210752037857275,0.0297897290739465,0.0117193567328012,-0.0273211767863346,
-0.0581242779571558,-0.0441549385653823, 0.0317587162010157, 0.149319516435409,
0.256827688764472,0.300284889915224,0.256827688764472,0.149319516435409,0.031758
7162010157,-0.0441549385653823,-0.0581242779571558,-0.0273211767863346,
0.0117193567328012,0.0297897290739465,0.0210752037857275,7.98496381387025e-18,-
0.0146736172848365,-0.0143651734835567,-0.00387098195557225,
0.00607046404166686,0.00845070298371067,0.00402808043882075,-
0.00169993772383668,-0.00415446275048170,-0.00277670657191816,-
1.96481403271878e-18, 0.00168035385481242, 0.00154680883851784,
0.000404321531916825,-0.000652555637283910,-0.00101955892407069};

  static float shift_reg[N];
#pragma HLS ARRAY_PARTITION variable=shift_reg complete dim=1
  float acc;
  float data;
  int i;
  acc=0;
```

```
Shift_Accum_Loop:

  for (i=N-1;i>=0;i--)
  {
   #pragma HLS UNROLL
      if (i==0){
            shift_reg[0]=x;
            data = x;}
      else{
            shift_reg[i]=shift_reg[i-1];
            data = shift_reg[i];}
      acc+=data*c[i];
  }
  *y=acc;
}
```

En el fichero de cabecera de dicha función se ha definido únicamente el factor *N* que representa el número de coeficientes, y la propia función principal.

```
/*************************************************************************
Project Name: fir_prj
Associated Filename: hls_fir.h
Family: Zynq
Device: xc7z020clg484-1
*************************************************************************/
#define N    51
void hls_fir (float *y, float x);
```

Y finalmente en el fichero *test bench* será donde se incluya una señal seno equivalente a la señal original que fue generada en Matlab y se le añada a cada muestra de señal la muestra de ruido correspondiente del array de ruido.

```
/*************************************************************************
Project Name: fir_prj
Associated Filename: hls_fir_test.c
Family: Zynq
Device: xc7z020clg484-1
*************************************************************************/

#include <stdio.h>
#include <math.h>
#include "hls_fir.h"

int main ()
{
```

```
  float output;
  float signal;

  float ruido[101] = {0.657077404387922,-0.727533291241120,-0.871174610298806,
0.102652341646096,0.596465199262861,-0.401411549152969,-0.632818211118985,-
0.0746656765917804,-0.818223339126947, 0.00867217302182911, 0.414193632995546,
0.108869174190739,-0.954622450631363,-0.268410910643749,-0.151016149223183,
0.906791065452167,0.457425876420537,-0.0285403579572386, 0.654681038914710,-
0.522367924204814,-0.174133405167633, 0.706280580364647, 0.751191464291398,
0.365187997268644,0.245376135951685,-0.293063069788201, 0.372449825212139,-
0.414077484902828, 0.287260365824715, 0.140920691322631, 0.569653134515253,-
0.212933922785855,0.318069945701484,0.396589040694787,-0.449188557050374,
0.0781224221387747,0.798626955638785,0.0562198552968705,-0.154312467273514,
0.228329804557575,-0.137550415283835,0.221571804391893,-0.0673825633614250,-
0.00916411486934316, 0.230394712503606, 0.681157732171151, 0.225937283690354,
0.824191828523085,-1.01418098042638,-0.224628405722455, 0.117996600855975,-
0.417586481891641,-0.637977581704109, 0.308517540640294, 0.306350751853637,
0.144690577647440,0.197658061184232,-0.435281421028505,-0.248844189102113,-
0.0533359085280763,-0.343914567713930, 0.165940438490711, 1.18261237129946,-
0.241115359234879,0.323724103350383,-0.517212358617083, 0.669777298952058,-
0.484570177914415,0.104357800636099,-0.309296677680274, 0.256007821868227,
0.00567708555041003,-0.0219943131333504,1.47454627065779,-0.315023096763674,-
0.0234396997690965,1.34151275538587,-0.573345343089522, 0.276499372733133,-
0.538229204546250, 0.515319792043396, 0.163764906238105, 0.326062405634966,-
0.139430558193911,0.122595796862011,0.736256739410789,-1.13755077117370,-
0.816645362460621,0.207734468489379,-0.327384425929007,-0.148174157703299,-
0.748459438223777,-0.452417223663851,-0.202090772515601,-0.362899083024479,-
0.433242512679806,-0.210923390220986,-0.471333162140308, 0.670941858628974,-
0.494217356147432,0.908971343821737};

  int i,j;
  j=0;
  for (i=0;i<101;i++)
  {
      signal = sin(2*M_PI*10*i*0.01) + ruido[j];
      j=j+1;
       // Execute the function with latest input
      hls_fir(&output,signal);
      // Print results.
      printf("%i %f %f\n",i,signal,output);
  }
  return 0;
}
```

Realizando la simulación del código con *Run C Simulation* se observa en la consola la salida que tiene el filtro para cada muestra de entrada. Esta salida corresponde con la salida de la función *filter* en Matlab, por lo que se comprueba que el filtro está funcionando correctamente.

El siguiente paso es realizar la síntesis del diseño con *Run C Synthesis* y observar el informe de resultados. En éste, lo primero que llama la atención es una serie de cifras en rojo dentro de la tabla de estimaciones de recursos. Estas cifras corresponden al número de recursos DSP48E empleados y al porcentaje de utilización de estos recursos en la tarjeta.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	255	17748	36261
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	6691	7232
Total	0	255	24439	43493
Available	280	220	106400	53200
Utilization (%)	0	115	22	81

FIGURA 197. INFORME DE SÍNTESIS - ESTIMACIÓN DE RECURSOS A UTILIZAR

Como puede verse en la Figura 197, se requieren 255 DSP48E para implementar el diseño, por lo que existe un problema de recursos al tener la tarjeta disponibles 220 DSP y superarse el 100% de la utilización de dicho recurso.

Analizando el problema, esto es debido a que los DSP48E se emplean para operaciones tales como multiplicaciones o sumas, y consultando en la sección de detalles del informe el apartado *Instance*, puede verse que se necesitan 2 DSP48E para cada una de las 51 operaciones *fadd* (suma entre float) y 3 DSP48E para cada una de las 51 operaciones *fmul* (multiplicación entre float).

En la Figura 198 que muestra la sección de detalles *Detail* puede verse un extracto de los dispositivos necesarios tanto para sumas como para multiplicaciones. Además, como puede verse, supone un coste hardware alto también para FFs y LUTs, aunque éstos no superan el total disponible.

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
hls_fir_fadd_32ns_32ns_32_5_full_dsp_U42	hls_fir_fadd_32ns_32ns_32_5_full_dsp	0	2	205	390
hls_fir_fadd_32ns_32ns_32_5_full_dsp_U43	hls_fir_fadd_32ns_32ns_32_5_full_dsp	0	2	205	390
hls_fir_fadd_32ns_32ns_32_5_full_dsp_U44	hls_fir_fadd_32ns_32ns_32_5_full_dsp	0	2	205	390
hls_fir_fadd_32ns_32ns_32_5_full_dsp_U45	hls_fir_fadd_32ns_32ns_32_5_full_dsp	0	2	205	390
hls_fir_fadd_32ns_32ns_32_5_full_dsp_U46	hls_fir_fadd_32ns_32ns_32_5_full_dsp	0	2	205	390
hls_fir_fadd_32ns_32ns_32_5_full_dsp_U47	hls_fir_fadd_32ns_32ns_32_5_full_dsp	0	2	205	390
hls_fir_fadd_32ns_32ns_32_5_full_dsp_U48	hls_fir_fadd_32ns_32ns_32_5_full_dsp	0	2	205	390
hls_fir_fadd_32ns_32ns_32_5_full_dsp_U49	hls_fir_fadd_32ns_32ns_32_5_full_dsp	0	2	205	390
hls_fir_fadd_32ns_32ns_32_5_full_dsp_U50	hls_fir_fadd_32ns_32ns_32_5_full_dsp	0	2	205	390
hls_fir_fmul_32ns_32ns_32_4_max_dsp_U51	hls_fir_fmul_32ns_32ns_32_4_max_dsp	0	3	143	321
hls_fir_fmul_32ns_32ns_32_4_max_dsp_U52	hls_fir_fmul_32ns_32ns_32_4_max_dsp	0	3	143	321
hls_fir_fmul_32ns_32ns_32_4_max_dsp_U53	hls_fir_fmul_32ns_32ns_32_4_max_dsp	0	3	143	321
hls_fir_fmul_32ns_32ns_32_4_max_dsp_U54	hls_fir_fmul_32ns_32ns_32_4_max_dsp	0	3	143	321

FIGURA 198. INFORME DE SÍNTESIS - DETALLE DE LOS RECURSOS A EMPLEAR

El motivo de que se estén empleando varios DSPs para realizar las operaciones de suma y multiplicación es porque un DSP48 contiene un multiplicador de ancho 18*18-bits y el diseño está compuesto por datos float de 32 bit de ancho, por lo que son necesarios 3 DSP48 para implementar un multiplicador.

En el diseño inicial en el que se trabaja con tipos de datos enteros de precisión arbitraria se tiene mucha más eficiencia en hardware porque el ancho de los datos se ajusta al necesario y por tanto los recursos para esas operaciones serán los mínimos requeridos. Por este motivo se comentaba, al hablar de las ventajas de Vivado HLS, que para optimizar en hardware la forma más útil es usar variables con el ancho de bit más pequeño posible, soportados en ANSI-C gracias a la cabecera *ap_cint.h*. Sin embargo, para datos de tipo *float* no existe en ANSI-C una cabecera que permita escoger el ancho de bit, teniéndose que trabajar con el ancho estándar de 32 bits.

Ante este inconveniente se piensa en utilizar directivas que modifiquen el comportamiento por defecto del compilador a la hora de asignar recursos.

Con la directiva ALLOCATION puede limitarse el número de operadores, recursos o funciones que se utilizarán en el diseño. Se prueba insertando ésta en el bucle *Shift_Accum_Loop* escogiendo limitar los recursos para suma y multiplicación (*instances=DAddSub_fulldsp* y *DMul_maxdsp*) a 88 core para las sumas y 132 core para las multiplicaciones, manteniendo la misma proporción pero con un máximo de 220 DSPs.

El panel de directivas *Directive* queda de la siguiente manera:

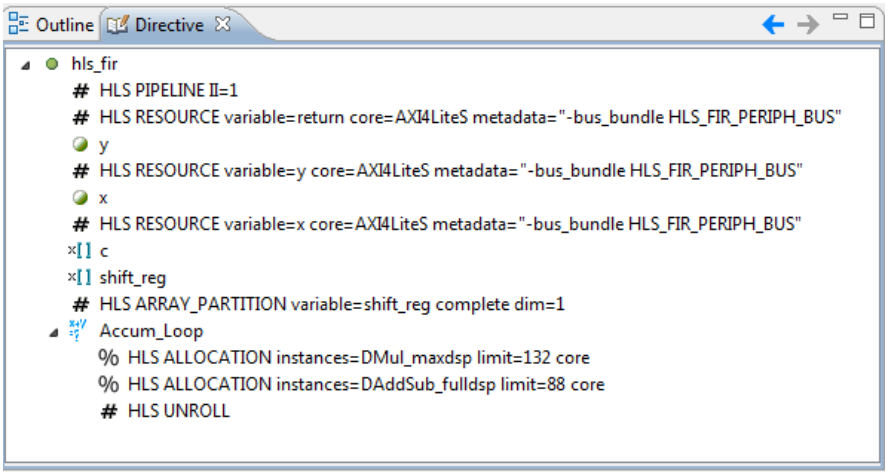


FIGURA 199. INCLUSIÓN DE DIRECTIVAS PARA LIMITAR LOS RECURSOS A UTILIZAR

Otra opción que puede resultar útil es usar la configuración *Config_bind* de la solución buscando minimizar en el diseño el número de operadores de multiplicación mediante la opción *min_op* y escogiendo la estrategia *High Effort* para forzar al compilador a que busque la mejor combinación posible sin importar el tiempo que eso conlleve.

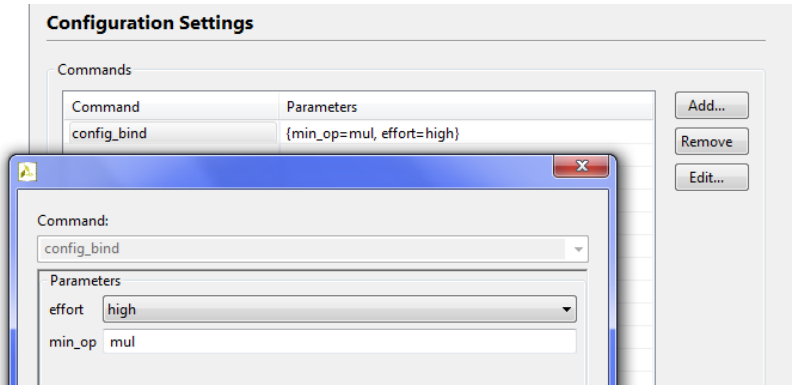


FIGURA 200. CONFIGURACIÓN CONFIG_BIND PARA MINIMIZAR EL NUMERO DE OPERADORES

A pesar de estas dos modificaciones en el diseño, el informe de síntesis sigue mostrando una superación de recursos, 255 de 220 para los DSP48 de la tarjeta. No ha sido posible reducir el área consumida mediante directivas debido al uso de datos en coma flotante, ya que las operaciones *fadd* y *fmul* requieren bastantes más recursos y desde un primer momento Vivado HLS ya mostraba en el informe de síntesis la implementación más óptima que se puede conseguir. La única solución a la vista es emplear un nuevo filtro con un menor número de coeficientes (menor orden) para solventar este problema.

Se diseña de igual forma con Matlab un filtro de orden 20 (N=21) y se copian los coeficientes exportados de fdatool al diseño en Vivado HLS modificándose el array *c[N]*. Por otro lado, se

mantienen los tipos de datos *float*, las directivas empleadas en el diseño inicial, y la misma señal senoidal con muestras de ruido en el *test bench*.

Tras comprobar mediante simulación que se obtiene la salida adecuada, se realiza la síntesis del diseño y se obtiene el siguiente resumen de recursos:

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	103	7103	14541
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	2109	224
Total	0	103	9212	14765
Available	280	220	106400	53200
Utilization (%)	0	46	8	27

FIGURA 201. INFORME DE SÍNTESIS PARA UN FILTRO DE ORDEN 20

Al reducir el número de coeficientes a la mitad se ve solventado el problema de superación de recursos, ya que ahora en *Instances* se ven implementados 2 DSP48E para cada una de las 20 operaciones *fadd* y 3 DSP48E para cada una de las 21 operaciones *fmul*, resultando un total de 103 DSP que suponen un 46% de los disponibles.

Este diseño es posible implementarlo en un bloque IP para su exportación a Vivado, generar el *bitstream* y continuar el diseño software para la comunicación en SDK. No obstante, no es común trabajar en coma flotante y los drives para controlar el bloque HLS están fijados para trabajar con datos enteros, y como ya se vio retornan valores en *unsigned int*, suponiendo una dificultad a la hora de querer observar los resultados en el terminal en formato *float* y comprobar que la salida es correcta.

3.2.4. **Comparativa y análisis de resultados en Matlab**

Tras este apartado donde se hace mención a otras alternativas y pruebas realizadas en el diseño, se vuelve al punto anterior en *Diseño software en SDK*, donde el diseño se encuentra en la plataforma de desarrollo software con la aplicación para la comunicación entre procesador y periférico creada y funcionando correctamente.

Hasta ahora, la salida del filtro podía verse conectando el Terminal de Vivado HLS o de SDK al puerto serie asignado a la tarjeta, de esta forma se imprimían en pantalla, a través de la función *printf*, las muestras de salida en cada instante. Además, la propiedad de self-checking del *test*

bench permitía escribir la salida en un fichero externo y saber, sin necesidad de comprobarlo visualmente, si la salida del filtro era correcta al coincidir este fichero con un fichero asociado al *test bench* que contenía la salida de la función *filter* de Matlab.

Otra posibilidad, más cómoda para el usuario, consiste en recoger la salida del filtro desde Matlab en un array, permitiendo tanto ver las muestras que corresponden a cada instante, como representarla. Para ello, puede abrirse desde Matlab un puerto serie indicando el puerto asignado a la tarjeta de igual forma que se hacía para el Terminal de SDK, y la tasa de baudios.

Una vez abierto el puerto, se lanza la aplicación en la FPGA y se escriben una serie de comandos en la consola de Matlab para recoger la salida por el puerto serie.

Para un correcto funcionamiento es necesario que en SDK se encuentre desconectado dicho puerto del Terminal (*CLOSED*) y así poder hacer la conexión, y para evitar más complejidad en la serie de comandos de Matlab es mejor asegurar que en el código no se imprime por printf más información que las muestras de salida que serán recogidas.

Una vez cumplidos estos requisitos, el procedimiento es el siguiente:

1.- En SDK, programar la FPGA mediante el icono *Program FPGA*. Estará listo tras completarse la barra de progreso y encenderse el LED LD12 de la tarjeta.

2.- Abrir un puerto serie en Matlab. Para eso se deben escribir los siguientes comandos:

```
>> s1=serial('COM13','Baudrate',115200);
>> fopen(s1);
```

3.- Desde SDK lanzar la aplicación en *Filter* > *Run As* > *Launch on Hardware*.

4.- Recoger la salida desde Matlab mediante el siguiente bucle for:

```
>> for i=1:70
salida=fscanf(s1,'%d');
ysdk(i)=salida;
end
```

En este momento se ha creado un array denominado "*ysdk"* en el espacio de trabajo de Matlab, de dimensiones 1x70, que contiene exactamente las mismas muestras que se veían desde el *Terminal*.

Siguiendo los pasos de la Tabla 9, puede generarse también en el espacio de trabajo la señal de entrada, el array de coeficientes convertidos al rango int7, y la salida real del filtro tras la función *filter*.

```
>> fdatool
>> num7=Num.*2^6;
>> x=1:70;
>> yreal7=filter(num7,1,x);
```

Ahora pueden representarse, mediante la función *plot()*, la señal de entrada (Figura 202), las muestras de salida real del filtro (Figura 203), y las muestras de salida obtenidas por el puerto serie desde SDK (Figura 204). Puede verse que ambas salidas son prácticamente la misma porque el error entre ellas es despreciable, con un error medio de 0.0294, tal como se calculó en la Tabla 9.

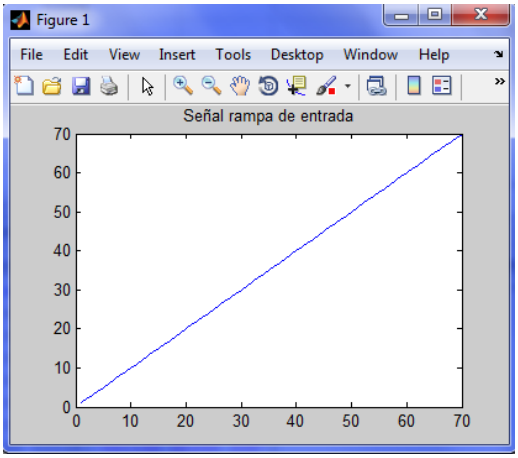


FIGURA 202. SEÑAL RAMPA DE ENTRADA

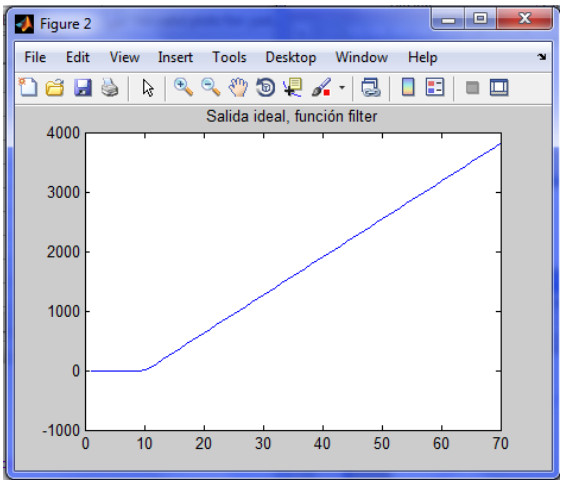


FIGURA 203. SEÑAL DE SALIDA IDEAL - FUNCIÓN FILTER DE MATLAB

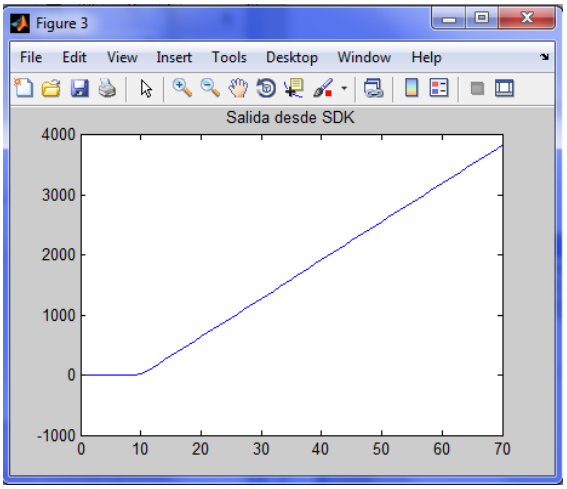


FIGURA 204. SEÑAL DE SALIDA EN SDK DEL FILTRO CREADO

4. CONCLUSIONES

El objetivo principal de este Trabajo de Fin de Grado consistía en hacer uso de las ventajas que ofrece la novedosa familia de dispositivos Zynq-7000 estudiando la creación de un periférico que pudiera ser controlado desde el microprocesador.

Con este propósito, en un primer lugar, se realizó un estudio del diseño de la tarjeta ZedBoard (Zynq Evaluation and Development Board), analizando las partes en las que se divide y la forma en que se comunican entre ellas. Además, se inició una búsqueda desde la propia página web de Xilinx para conocer qué clase de herramientas y cuáles versiones permitían trabajar con esta tarjeta, llevándose a cabo la instalación y prueba de varias de ellas hasta dar con la versión, de entre las más actuales, que mejor funcionamiento mostraba en el equipo.

Una vez se disponía de las herramientas adecuadas, para comenzar a familiarizarse con el entorno de trabajo y con la tarjeta de desarrollo se fueron dando pasos de creciente complejidad.

Inicialmente, con la intención de ofrecer una visión general al usuario de la interacción entre herramientas de diseño, se comienza por crear un diseño al que se le añade algún periférico de los ya contenidos en el Catálogo IP de XPS, se habilita la comunicación entre ambas partes, se asignan los puertos GPIO a los pines de la FPGA y se realiza la prueba en hardware creando una aplicación en SDK que emplea las funciones proporcionadas para el manejo de los periféricos.

Posteriormente, se lleva a cabo la creación de un periférico propio no contenido en el Catálogo IP, desde las herramientas IPIF e IPIC de XPS, accediendo además a los ficheros que describen el periférico y su funcionalidad.

Hasta ese punto se había trabajado con las herramientas de EDK proporcionadas por Xilinx, pero resultaba interesante y conveniente estudiar el modo de trabajo con una herramienta diferente también proporcionada por Xilinx para una mayor productividad, Vivado HLS, por lo que la segunda mitad de este trabajo se centra en este punto de vista desde un nivel mayor de abstracción, empleando lenguajes de alto nivel.

Para trabajar con Vivado HLS se comenzó un nuevo estudio acerca del funcionamiento de esta herramienta, consultándose su manual de usuario [28] para analizar su comportamiento y sacar beneficio de las ventajas que ofrecía.

Una vez analizada su metodología de trabajo, con sus directivas de optimización y sus fases de diseño, se continuó con un ejemplo de aplicación de directivas consistente en una multiplicación de matrices sobre la cual se van implementando diferentes soluciones observando el impacto que conlleva la aplicación de una directiva a la síntesis en función del momento y el lugar.

A continuación se continuó con el diseño de un periférico, concretamente un filtro FIR, haciendo uso de la herramienta Matlab para el diseño del filtro y la obtención de coeficientes, así como para comprobar la validez de los resultados obtenidos.

A partir de ahí, se realizó el código en lenguaje C para el diseño del filtro en Vivado HLS, se estudió la solución más óptima del diseño haciendo uso de directivas y de tipos de precisión arbitraria, se exportó éste a Vivado Design Suite para la generación del bitstream que programe la FPGA y finalmente se exportó a SDK para programar la aplicación que comunique al procesador con el filtro, sucediéndose las siguientes fases en la comunicación: inicialización, comprobación de estado, envío de muestra de señal, activación, espera y recogida de muestra filtrada.

Para finalizar, se proponen varias alternativas de diseño, como es emplear interrupciones del periférico al microprocesador a la hora de recoger las muestras filtradas, y los inconvenientes encontrados en cuanto a consumo de recursos en el uso de tipos de datos de coma flotante.

Vistas las diferentes herramientas que propone Xilinx para la creación de diseños en dispositivos Zynq, cabe resaltar Vivado HLS por las ventajas que ofrece en cuanto a productividad y rendimiento para diseñadores tanto de hardware como de software al hacer uso de lenguajes de alto nivel, destacando además la opción que presenta la herramienta de inclusión de directivas para la optimización del diseño sin necesidad de modificar el código fuente.

En definitiva, la nueva serie de dispositivos Zynq-7000 apuesta por una arquitectura novedosa que ha conseguido el éxito que se ha propuesto en el mercado gracias a su carácter *All Programmable* que la convierte en la mejor opción para dotar a sistemas de inteligencia de la forma más rápida y eficaz a través de la programación del software, del hardware y de las interfaces del sistema.

MANUAL DE USUARIO

Como se ha podido ver, este Trabajo de Fin de Grado consta de dos partes diferencias, una primera parte donde se trabaja con las herramientas Embedded Development Kit de Xilinx, y una segunda parte más desarrollada donde se da a conocer la forma de trabajo de Vivado High-Level Synthesis, también de Xilinx.

Cada una de estas herramientas se encuentra además asociada a otras herramientas de Xilinx para completar el diseño de los dispositivos Zynq. En el caso de EDK, las herramientas XPS y SDK emplean también PlanAhead para coordinar las fases del diseño, y en el caso de Vivado HLS, emplea Vivado Design Suite y SDK para continuar el diseño a partir de la implementación RTL.

Esta variedad de herramientas conlleva a la creación de distintos proyectos de trabajo para cada una de ellas, por ello en el CD que se adjunta se pueden encontrar 6 carpetas, correspondiéndose cada directorio a cada uno de los ejercicios comentados durante el trabajo.

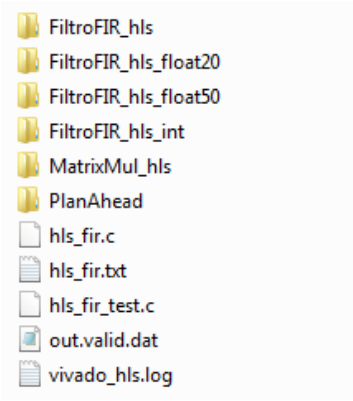


FIGURA 205. PROYECTOS CONTENIDOS EN EL TRABAJO Y FICHEROS ASOCIADOS

Por un lado, los proyectos correspondientes a la primera parte de la memoria se localizan en la carpeta *PlanAhead*, donde se encuentran en su interior las carpetas *project_1* y *project_2*.
La primera de ellas contiene el proyecto realizado en el apartado *"3.1.3. PlanAhead"* para la creación de un diseño que contenga un periférico existente en el Catálogo IP de XPS, y la segunda de ellas contiene el proyecto realizado en el apartado *"3.1.6. Periférico creado por el usuario mediante IPIF e IPIC"*, el cual se trata del proyecto anterior modificado para incluir un periférico creado por el usuario. Estas carpetas que dan nombre al proyecto serán las que se deban localizar en el directorio de búsqueda de PlanAhead para indicar el proyecto que se quiere abrir. Una vez abierto, es desde la propia herramienta desde donde se llama a XPS y a SDK llegado el momento.

Por otro lado, los proyectos correspondientes a la segunda parte de la memoria llevan en su nombre la indicación *"hls"* para indicar que pertenecen a proyectos realizados con Vivado HLS.
La carpeta *FiltroFIR_hls* es el proyecto realizado en el apartado *"3.2.2. Periférico diseñado por el usuario con Vivado HLS"*, dando lugar a las carpetas *.apc*, *.reference* y *.settings* y a los archivos

.cproject, *.project, vivado_hls_log_all.xml* y *vivado_hls.app* generados en el momento de creación del proyecto y necesarios para su correcto funcionamiento en la herramienta, y a las carpetas *solution1*, *solution2* y *solution3* creadas a lo largo del diseño con las tres diferentes implementaciones que el usuario ha querido analizar.

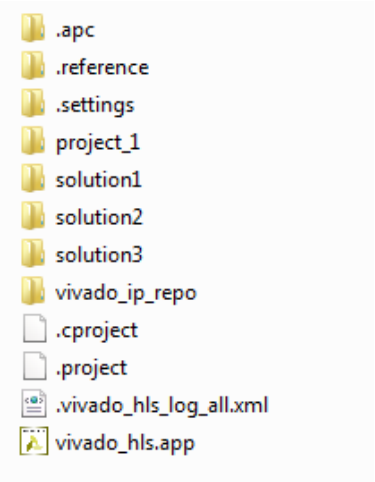


FIGURA 206. ARCHIVOS PERTENECIENTES AL PROYECTO DEL FILTRO CREADO CON VIVADO HLS

Dentro de cada carpeta de solución del proyecto se encuentran carpetas con los ficheros generados durante la simulación (*csim*), síntesis (*syn*), verificación del diseño RTL (*sim*) y empaquetado IP (*impl*).

En el interior de la carpeta *csim* resulta interesante acceder a la carpeta *build*, donde puede encontrarse un fichero .dat con la salida del programa si se ha diseñado un testbench con propiedad de self-checking, además se encuentra una carpeta *report* con los ficheros generados para el informe mostrado tras la simulación.

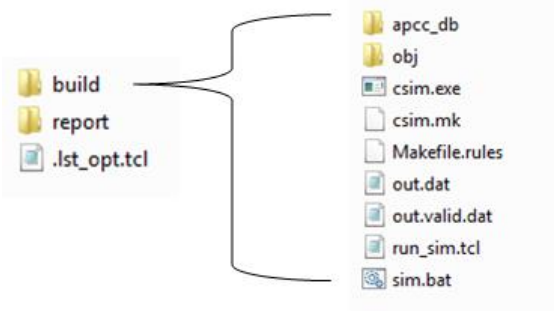


FIGURA 207. CONTENIDO DE LA CARPETA DE SIMULACIÓN

En la carpeta *syn* se encuentran carpetas con los ficheros generados en la síntesis en cada uno de los tres lenguajes de descripción hardware: *systemc, verilog* y *vhdl*, además de la carpeta *report* con los ficheros generados para el informe mostrado tras la síntesis.

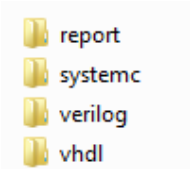


FIGURA 208. CONTENIDO DE LA CARPETA DE SÍNTESIS

La carpeta *sim* es generada tras hacer la verificación del diseño RTL final, por ello generalmente se encuentra en el interior de la última solución implementada dentro de un proyecto. En ella se puede encontrar una carpeta *report* con los ficheros generados para el informe mostrado tras la co-simulación, y otras carpetas con ficheros referentes a los programas y al testbench empleados en la verificación.

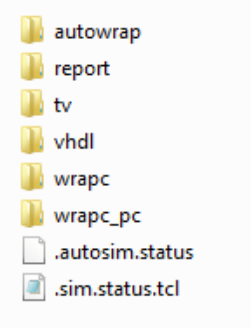


FIGURA 209. CONTENIDO DE LA CARPETA DE COSIMULACIÓN

Finalmente, la carpeta *impl* se genera tras haberse validado los procesos anteriores y en el momento de exportación del diseño RTL, por lo que únicamente se encuentra dentro de la solución final escogida por el usuario como implementación óptima para su diseño. Contiene una carpeta de exportación con el formato de salida escogido, para el caso de IP Catalog la carpeta toma el nombre *ip* y contiene un fichero *.zip con información referente al periférico que será importado desde Vivado, y las carpetas *vhdl* y *verilog*, con los proyectos para Vivado en cada uno de los lenguajes.

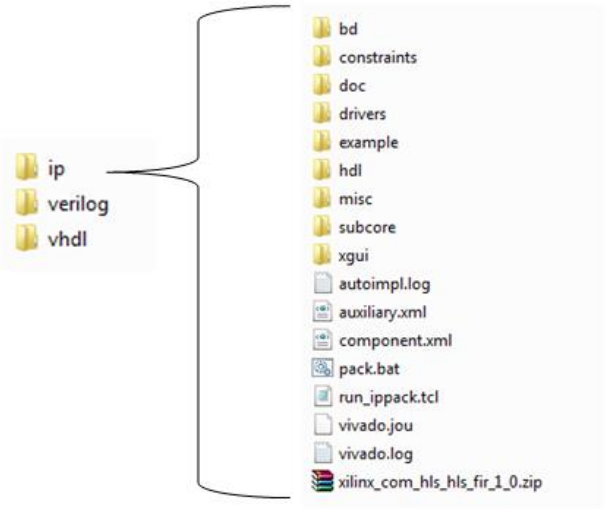


FIGURA 210. CONTENIDO DE LA CARPETA DE IMPLEMENTACIÓN

Además, en el interior de *FiltroFIR_hls*, ver Figura 211, se encuentran las carpetas *project_1*, que es el proyecto creado en Vivado Design Suite para la generación del bitstream, y *vivado_ip_repo*, que es el repositorio creado en Vivado para contener el fichero .zip del periférico exportado. Además, dentro de *project_1* se encuentra una carpeta *project_1.sdk* con los ficheros generados en el momento de exportar el bitstream de Vivado a SDK y carpetas para la plataforma hardware exportada (*hw* y *hw_platform_0*), la plataforma software donde crear la aplicación, *Filter_bsp*, y la aplicación creada en SDK, *Filter*.

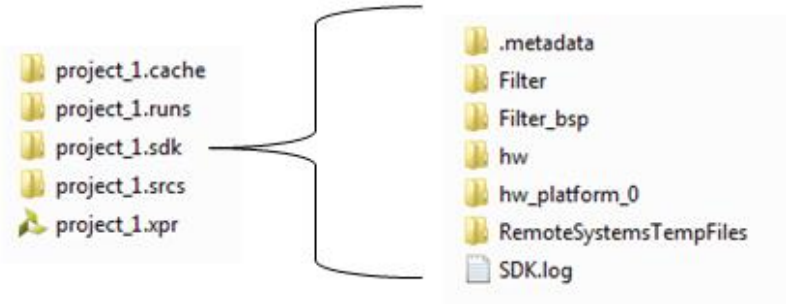


FIGURA 211. INTERIOR DEL PROYECTO DEL FILTRO CREADO EN VIVADO HLS

Con contenidos similares se encuentra la carpeta *MatrixMul_hls* perteneciente al apartado *Ejemplo de aplicación de directivas* realizado al finalizar la parte teórica de esta herramienta, y las carpetas creadas en el apartado *"3.2.3 Otras Alternativas",* como son *FiltroFIR_hls_int*, que contiene un proyecto *fir_prj* igual al que se tiene en *FiltroFIR_hls* pero con la alternativa de que la aplicación creada en SDK se ha realizado mediante interrupciones del periférico al microprocesador, y las carpetas *FiltroFIR_hls_float50* y *FiltroFIR_hls_float20*, que contienen la misma jerarquía de carpetas con la alternativa de realizar los proyectos empleando tipos de datos float con un orden de filtro de 50 y 20, respectivamente.

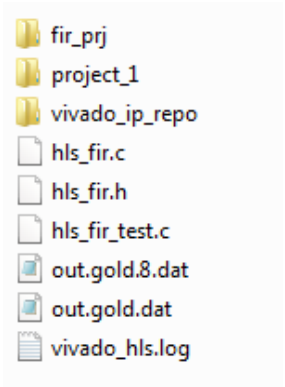


FIGURA 212. PROYECTOS GENERADOS EN OTRAS ALTERNATIVAS

PRESUPUESTO

La realización de este Trabajo Fin de Grado ha supuesto unos costes cuyo presupuesto se desglosa a continuación.

- **Coste de los materiales.**

En este apartado se engloba el precio del uso de los diversos equipos empleados para desarrollar el presente trabajo, describiendo tanto el precio de la parte hardware como el de la parte software. Por último, se hará un pequeño resumen del conjunto de material de oficina utilizado durante la realización del proyecto.

- Recursos hardware

EQUIPO	**Total en €**
PC Acer Aspire 6930G	599 €
*Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit **	669,74 €
TOTAL RECURSOS HARDWARE	**1.268,74 €**

*Nota: La tarjeta ZedBoard ha sido proporcionada por el Director del proyecto.

- Recursos software

SOFTWARE	PRECIO	AMORTIZACIÓN	**Tiempo utilizado**	**Total en €**
Xilinx Vivado+SDK for Windows v2014.1	0 €	-	7 meses	0 €
Xilinx ISE Design Suite for Windows v14.3	0 €	-	7 meses	0 €
Matlab Student	69 €	1 año	4 meses	23 €
Microsoft Office 2010	110 €	3 años	7 meses	21,38 €
*Licencia ISE Design Suite **	0 €	1 año	7 meses	0 €
*Licencia Vivado Design Suite System Edition Node-Locked ***	3.588,43 €	4 años	7 meses	523,31 €
TOTAL RECURSOS SOFTWARE				**567,69 €**

*Nota 1: El pack Zynq-7000 All Programmable SoC ZC702 Evaluation Kit contiene un código para obtener la licencia de la herramienta de diseño ISE Design Suite, por lo que no requiere gasto adicional.

**Nota 2: El precio estimado para la licencia de Vivado Design Suite, incluyendo Vivado HLS, es de 3.588,43 €, pero para la realización del TFG se ha empleado una licencia gratuita permanente para ISE+Vivado y aparte una licencia de prueba de 30 días gratuita para Vivado HLS que se ha ido renovando varias veces según fuera conveniente.

- Material de oficina

MATERIAL	**Total en €**
Papel	30 €
Impresión	144.15 €
Encuadernación	36 €
TOTAL MATERIAL DE OFICINA	**210.15 €**

Llegados a este punto, se puede realizar el cálculo final del coste que supone el conjunto de recursos utilizados:

TOTAL DE LOS RECURSOS HARDWARE	**1.268,74 €**
TOTAL DE LOS RECURSOS SOFTWARE	**567,69 €**
TOTAL DE LOS MATERIALES DE OFICINA	**210,15 €**
TOTAL MATERIALES	**2.046,58 €**

El coste total de los materiales empleados asciende a **dos mil cuarenta y seis euros con cincuenta y ocho céntimos**.

- **Coste de mano de obra.**

Los costes que se incluyen en este apartado, se derivan del pago por la mano de obra utilizada para realizar el presente proyecto. En este tipo de costes aparecen los derivados del diseño de Ingeniería, redacción y realización del libro (mecanografía).

Los costes de mano de obra, se calculan por precio por hora trabajada, estos precios por hora, están establecidos por el Colegio Oficial de Ingenieros Técnicos de Telecomunicación.

REALIZACIÓN	COSTE POR HORA	**Total horas**	**Total en €**
Documentación e instalación de herramientas	20 €	120 horas	2.400 €
Diseño de Ingeniería y realización del Proyecto	58 €	290 horas	16.820 €
Redacción del libro	10 €	200 horas	2.000 €
TOTAL MANO DE OBRA			**21.220 €**

El coste total de la mano de obra asciende a **veintiún mil doscientos veinte euros**.

- **Presupuesto de ejecución material.**

Se calcula como la suma del coste total material y el coste total debido a la mano de obra.

COSTE TOTAL DEL MATERIAL EMPLEADO	**2.046,58 €**
COSTE TOTAL DE LA MANO DE OBRA	**21.220 €**
PRESUPUESTO DE EJECUCIÓN MATERIAL	**23.266,58 €**

El presupuesto de ejecución material asciende a **veintitrés mil doscientos sesenta y seis con cincuenta y ocho céntimos.**

- **Importe de ejecución por contrata.**

Se incluyen en este apartado los gastos derivados del uso de las instalaciones donde se ha llevado a cabo el proyecto, cargas fiscales, gastos financieros, tasas administrativas y derivados de las obligaciones de control del proyecto. De igual forma se incluye el beneficio industrial. Para cubrir estos gastos se establece un recargo del 22 % sobre el importe del presupuesto de ejecución material.

COSTE TOTAL DE EJECUCIÓN MATERIAL	**23.266,58 €**
22% DE GASTOS FINANCIEROS, BENEFICIOS, ETC.	**5.118.64 €**
PRESUPUESTO DE EJECUCIÓN POR CONTRATA	**28.385,22 €**

El importe de ejecución por contrata asciende a **veintiocho mil trescientos ochenta y cinco con veintidós céntimos**.

- **Honorarios facultativos.**

Los honorarios facultativos por la realización del proyecto se calculan de acuerdo a lo estipulado por el Colegio Oficial de Ingenieros Técnicos de Telecomunicación en el apartado de tarifas de trabajos particulares y se calculan teniendo en cuenta la siguiente expresión:

$$Honorarios = 0{,}07 \times P \times C$$

Siendo P el presupuesto de ejecución por contrata y C es el coeficiente de reducción.
El coeficiente de reducción es distinto dependiendo de la cantidad de P:

- Para $P < 30.050{,}61$ € $\Rightarrow C = 1$.
- Para $60.101{,}21$ € $> P > 30.050{,}61$ € $\Rightarrow C = 0{,}9$
- Para $90.151{,}82$ € $> P > 60.101{,}21$ € $\Rightarrow C = 0{,}8$

Así en el presente proyecto se tienen los siguientes importes de honorarios facultativos:

0,07 X 28.385, 22€ X 1	**1.986,96 €**
TOTAL HONORARIOS FACULTATIVOS	**1.986,96 €**

El importe total de los Honorarios Facultativos es de **mil novecientos ochenta y seis euros con noventa y seis céntimos**.

- **Presupuesto total.**

El importe del presupuesto final de este proyecto es la suma del presupuesto por contrata y los honorarios facultativos:

PRESUPUESTO DE EJECUCIÓN POR CONTRATA	**28.385,22 €**
HONORARIOS FACULTATIVOS	**1.986,96 €**
PRESUPUESTO TOTAL	**30.372,18 €**
21 % IVA	**6.378,15 €**
PRESUPUESTO FINAL	**36.750,33 €**

El importe final del proyecto asciende a la cantidad de **TREINTA Y SEIS MIL SETECIENTOS CINCUENTA EUROS CON TREINTA Y TRES CÉNTIMOS.**

En Alcalá de Henares, a 25 de septiembre del 2014.

Sara Ortega Lázaro Graduada en Ingeniería en Tecnologías de la Telecomunicación

BIBLIOGRAFÍA

[1] Oppenheim, Alan V.; Schafer, Ronald W. *Tratamiento de señales en tiempo discreto.* 3ª ed. Madrid: Pearson Educación, 2011.

[2] Proakis, Jhon G.; Manolakis, Dimitris G. *Tratamiento digital de señales*. 4ª ed. Madrid: Pearson Prentice-Hall, 2007.

[3] Xilinx. "Why Zynq?". http://www.xilinx.com/training/zynq/why-zynq.htm

[4] Xilinx. "Zynq Processing System Highlights". http://www.xilinx.com/training/zynq/zynq-processing-system-highlights.htm

[5] Xilinx. "Hardware Architecture Highlights". http://www.xilinx.com/training/zynq/zynq-hardware-architecture-highlights.htm

[6] ZedBoard. "Getting Started Guide". 30 de junio de 2014. http://www.zedboard.org/sites/default/files/documentations/GS-AES-Z7EV-7Z020-G-V7.pdf

[7] ZedBoard. "Avnet Product Brief ZedBoard". http://www.zedboard.org/sites/default/files/product_briefs/ZedBoard%20Brochure%20%28English%29.pdf

[8] ZedBoard. "Getting Started Instructions". http://www.zedboard.org/sites/default/files/documentations/GSC-AES-Z7EV-7Z020-G-v1f-press.pdf

[9] ZedBoard. "ZedBoard Hardware User's Guide". 27 de junio de 2014. http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[10] ZedBoard. "Create a PlanAhead Project with Embedded Processor". http://zedboard.org/content/zedboard-create-planahead-project-embedded-processor

[11] Zynq Geek. "Zedboard – SDK HelloWorld Example". 23 de Agosto de 2012. http://zedboard.org/sites/default/files/blogger_importer/08/zedboard-sdk-helloworld-example.html

[12] Getman, Lawrence. "Creating the Xilinx Zynq-7000 Extensile Processing Platform". *EE Times*. Designlines, Programmable Logic. 17 de octubre de 2011. http://www.eetimes.com/document.asp?doc_id=1279151

[13] Xilinx. "Zynq Architecture". http://www.ioe.nchu.edu.tw/Pic/CourseItem/4468_20_Zynq_Architecture.pdf

[14] Xilinx. "Zynq-7000 All Programmable SoC". http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm

[15] Xilinx. "Vivado Design Suite". http://www.xilinx.com/products/design-tools/vivado/index.htm

[16] Xilinx. "ISE Design Suite". http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm

[17] Xilinx. "EDK Concepts, Tools, and Techniques. A Hands-On Guide to Effective Embedded System Design". 16 de octubre de 2012. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/edk_ctt.pdf

[18] Xilinx. "PlanAhead User Guide". 16 de octubre de 2012. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/PlanAhead_UserGuide.pdf

[19] Xilinx. "Quick Front-to-Back Overview Tutorial. PlanAhead Design Tool". 8 de mayo de 2012. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/PlanAhead_Tutorial_Quick_Front-to-Back_Overview.pdf

[20] Xilinx. "Vivado Design Suite Tutorial. Embedded Processor Hardware Design".23 de abril de 2014. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug940-vivado-tutorial-embedded-design.pdf

[21] Bailey, Brian. "High Level Synthesis: Significant Differences Remain". *Semiconductor Engineering.* Low power – High performance. 9 de abril de 2014. http://semiengineering.com/high-level-synthesis-significant-differences-remain/

[22] Maxfield, Clive. "Xilinx Zynq-7000 All Programmable SoC wins Microprocessor Report Analyst Choice Award". *EE Times*. Designlines, Automotive. 21de enero de 2013. http://www.eetimes.com/document.asp?doc_id=1262996

[23] Bailey, Brian. "Xilinx Zynq-7000 receives product of the year ACE award". *EE Times*. Designlines, Automotive. 30 de marzo de 2012. http://www.eetimes.com/document.asp?doc_id=1261456

[24] Xilinx. "ZedBoard: Zynq-7000 AP SoC Concepts, Tools and Techniques. A Hands-On Guide to Effective Embedded System Design". 8 de Julio de 2013. http://forums.xilinx.com/xlnx/attachments/xlnx/EMBEDDED/12233/1/zedboard_CTT_v2013_2_130807.pdf

[25] Bagni, Daniele; Noguera, Juanjo; Martinez Vallina, Fernando. "Zynq-7000 All Programmable SoC Accelerator for Floating-Point Matrix Multiplication using Vivado HLS". 29 de junio de 2013. http://www.xilinx.com/support/documentation/application_notes/xapp1170-zynq-hls.pdf

[26] Xilinx. "Introduction to FPGA Design with Vivado High-Level Synthesis". 2 de julio de 2013. http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

[27] Xilinx. "Vivado Design Suite User Guide. Design Flows Overview". 2 de abril de 2014. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug892-vivado-design-flows-overview.pdf

[28] Xilinx. "Vivado Design Suite User Guide. High-Level Synthesis". 30 de mayo de 2014. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug902-vivado-high-level-synthesis.pdf

[29] Xilinx. "Vivado Design Suite Tutorial. High-Level Synthesis". [Ch.10 Using HLS IP in a Zynq Processor Design]. 6 de mayo de 2014. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug871-vivado-high-level-synthesis-tutorial.pdf

[30] Xilinx. "Enabling Smarter Systems". http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/smarter-system.html

[31] Ouellette, Matthew. "Using HLS and Programmable SoCs to Drive Real-Time Digital Signal Processing". *RTC magazine*. Technology in context. Noviembre de 2012. http://www.rtcmagazine.com/articles/view/102828

[32] Filter Online. "TFilter". http://t-filter.appspot.com/fir/index.html

[33] Verner, Eric. "Digital Filtering in Matlab". Matlab Geeks. 19 de marzo de 2011. http://matlabgeeks.com/tips-tutorials/digital-filtering-in-matlab/

[34] Cervetto, Marcos; Marchi Edgardo. "Filtros FIR en FPGAs". Facultad de Ingeniería de la Universidad de Buenos Aires. SASE 2012. http://www.sase.com.ar/2012/files/2012/09/SASE_2012_track_DSP_04.pdf

[35] Álvarez Cedillo, Jesús Antonio; Lindig Bos, Klauss Michael; Martínez Romero, Gustavo. "Implementación de Filtros Digitales Tipo FIR en FPGA". http://polibits.gelbukh.com/37_11.pdf

[36] Bravo, Ignacio; Rivera, Raúl; Hernández, Álvaro; Mateos, Raúl; Gardel, Alfredo; Meca, Francisco Javier. "Implementación de filtros FIR en FPGA's". Departamento de Electrónica de la Universidad de Alcalá, Madrid. http://e-spacio.uned.es/fez/eserv.php?pid=taee:congreso-2004-1137&dsID=SP106.pdf

[37] Documentación Online. “Filtros Digitales”. ftp://ftp.udistrital.edu.co/Documentacion/Electronica/Dsp/capitulo5.PDF

[38] “Diseño de filtros FIR”. Departamento de Ingeniería Electrónica. Universidad de Valencia. http://ocw.uv.es/ingenieria-y-arquitectura/filtros-digitales/tema_3._diseno_de_filtros_fir.pdf

[39] Lara, Amanda; Tapia, Víctor; Gamboa, Darwin; Narváez, Javier; Parreño, Darwin. “Filtros IIR y FIR”. Escuela de Telecomunicaciones y Redes. Facultad de Informática y Electrónica. Escuela superior politécnica de Chimborazo. http://es.scribd.com/doc/46470741/Los-Filtros-FIR

[40] Xilinx. “Xilinx Glossary”. http://www.xilinx.com/company/terms.htm