Generación automática de VHDL mediante Simulink HDL Coder y aplicación al procesado de señales

Proyecto Fin de Carrera

7 de julio de 2015

Josu López Fernández

Director:
Andoni Arruti Illarramendi

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Agradecimientos

A todos los que me han apoyado durante este proyecto, especialmente a los que gracias a su insistencia me han animado a acabarlo de una vez ;)

Eskerrik asko!

II

Índice general

Índice de figuras

Índice de tablas

Resumen

Gordon E. Moore, co-fundador de Intel, predijo en una publicación del año 1965[1] que aproximadamente cada dos años se duplicaría el número de transistores presentes en un circuito integrado, debido a las cada vez mejores tecnologías presentes en el proceso de elaboración. A esta ley se la conoce como *Ley de Moore* y su cumplimiento se ha podido constatar hasta hoy en día.

Gracias a ello, con el paso del tiempo cada vez se presentan en el mercado circuitos integrados más potentes, con mayores prestaciones para realizar tareas cada vez más complejas. Un tipo de circuitos integrados que han podido evolucionar de forma importante por dicho motivo, son los dispositivos de *lógica programable*, circuitos integrados que permiten implementar sobre ellos las funciones lógicas que desee implementar el usuario.

Hasta hace no muchos años, dichos dispositivos eran capaces de implementar circuitos compuestos por unas pocas funciones lógicas, pero gracias al proceso de miniaturización predicho por la *Ley de Moore*, hoy en día son capaces de implementar circuitos tan complejos como puede ser un microprocesador; dichos dispositivos reciben el nombre de FPGA, siglas de *Field Programmable Gate Array*.

Debido a la mayor capacidad y por lo tanto a diseños más complejos implementados sobre las FPGA, en los últimos años han aparecido herramientas cuyo objetivo es hacer más fácil el proceso de ingeniería dentro de un desarrollo en este tipo de dispositivos, como es la herramienta **HDL Coder** de la compañía *MathWorks*, creadores también **Matlab** y **Simulink**, unas potentes herramientas usadas ampliamente en diferentes ramas de la ingeniería.

El presente proyecto tiene como objetivo evaluar el uso de dicha herramienta para el procesado digital de señales, usando para ello una FPGA **Cyclone II** de la casa *Altera*. Para ello, se empezará analizando la herramienta escogida comparándola con herramientas de la misma índole, para a continuación seleccionar una aplicación de procesado digital de señal a implementar. Tras diseñar e implementar la aplicación escogida, se deberá simular

en PC para finalmente integrarla en la placa de evaluación seleccionada y comprobar su correcto funcionamiento.

Tras analizar los resultados de la aplicación de implementada, concretamente un analizador de la frecuencia fundamental de una señal de audio, se ha comprobado que la herramienta **HDL Coder**, es adecuada para este tipo de desarrollos, facilitando enormemente los procesos tanto de implementación como de validación gracias al mayor nivel de abstracción que aporta.

Capítulo 1

Introducción

Contents

El hardware que podemos usar para el procesado digital de señales (*Digital Signal Processing*) ha ido evolucionando a lo largo del tiempo desde la aparición de los primeros sistemas digitales. Hoy en día disponemos desde procesadores digitales de señal (DSP por sus siglas en inglés – *Digital Signal Processor*, procesadores con una arquitectura específica para cumplir con las exigencias de procesar digitalmente una señal), hasta dispositivos programables como son las FPGA (*Field Programmable Gate Array*, dispositivos electrónicos que pueden ser reconfigurados en campo para acometer una función específica). Con el paso del tiempo, las prestaciones de dichos dispositivos han ido en aumento debido a la cada vez mayor capacidad de integración en un solo chip que presentan los nuevos procesos de fabricación en silicio, pudiendo llegar incluso a implementar en FPGAs de gama alta más de un microprocesador entero en su interior.

El hecho de que estos dispositivos sean programables los hace muy idóneos para tareas de cálculo intensivo como son las aplicaciones de procesado digital de señal, donde los procesadores convencionales no se ajustan muy bien, debido a que están pensados más para sistemas reactivos que para los únicamente centrados en el *data-flow*. Además, hay que tener en cuenta que en un sistema empotrado el uso de recursos es limitado, por lo tanto necesitamos que la unidad de proceso de nuestro sistema sea lo más ajustada posible, exprimiendo al máximo sus posibilidades sin tener que recurrir a soluciones más complejas y por lo tanto más costosas, por ejemplo en consumo de energía.

Como se ha dicho, gracias a los nuevos procesos de fabricación, las FPGAs tienen cada vez mayor capacidad, aceptando así diseños más complejos, pero en contrapartida, esto hace que su manejo sea más complejo a la hora de desarrollar. Es por este motivo que cada vez podemos encontrar en el mercado más herramientas cuyo objetivo es el de facilitar el proceso de desarrollo, reduciendo así el tiempo de desarrollo y por lo tanto costes en el producto final. Los mayores fabricantes de FPGAs, tanto *Altera* como *Xilinx*, ofrecen sendas soluciones para abordar este problema, *DSP Builder* y *System Generator* respectivamente, siendo ambas ofrecidas como un *blockset* para el entorno de desarrollo gráfico *Simulink* de *MathWorks*.

Dichas soluciones funcionan de manera similar: mediante una serie de bloques de unas librerías que proporciona el fabricante diseñaremos nuestro sistema de procesado de señal digital. Podemos simularlo funcionalmente desde el entorno *Simulink*, para después exportarlo a algún lenguaje de descripción de hardware (HDL – *Hardware Description Language*) y así poder sintetizarlo para configurar nuestra FPGA. El problema viene a la hora de que por el motivo que sea cambiemos de fabricante, ya que los diseños hechos para una plataforma no funcionan en la otra. Es por ello que desde las últimas versiones de Simulink disponemos de la herramienta *HDL Coder*, la cual

nos permite hacer diseños con la librería estándar de Simulink (con algunas limitaciones, no están disponibles ni todos los bloques ni todas las funcionalidades de Simulink) y luego exportar el resultado a código HDL sintetizable en cualquier FPGA, aunque por contrapartida tenemos que el código resultante estará menos optimizado que si hubiéramos usado las herramientas propias del fabricante de nuestra plataforma, puesto que el fabricante conoce perfectamente el diseño de sus plataformas y puede hacer una mejor optimización de recursos que una herramienta genérica.

Ésta herramienta, *MathWorks Simulink HDL Coder*, será el objeto de análisis de este Proyecto de Fin de Carrera, analizando su uso específicamente para el procesado digital de señal. Para ello escogeremos una aplicación y la implementaremos con la ayuda de la herramienta. Esto nos permitirá conocer de una mejor manera que ofrece dicha herramienta, sus ventajas y desventajas frente a otras herramientas parecidas (o frente al proceso estándar de desarrollo), y finalmente su idoneidad para este tipo de aplicaciones.

1.1. Aplicación a desarrollar

La aplicación a desarrollar que se ha decidido consiste en un estimador de frecuencia fundamental de una señal, en este caso concreto una señal de audio. A grandes rasgos, conectaremos una fuente de audio externa a la tarjeta de evaluación que vamos a usar (*Altera DE2*), y se nos deberá mostrar en algún tipo de *display* cual es su frecuencia fundamental.

La parte que se pretende desarrollar con la ayuda de Simulink HDL Coder es la referente al procesado de señal, es decir, la encargada de determinar cual es la frecuencia fundamental de la señal. Las demás partes que componen la aplicación serán desarrolladas *a priori* mediante las técnicas previamente conocidas para el desarrollo de aplicaciones para FPGAs, aunque se aprovechará para analizar si la herramienta puede ayudarnos en dichas tareas externas al procesado digital de señal. En el capítulo 4 se explicará en detalle el diseño referente a la aplicación implementada mediante la herramienta a analizar.

1.2. Objetivos

A continuación se detallan los objetivos que se plantean durante el desarrollo de este Proyecto de Fin de Carrera:

- Conocer en profundidad la herramienta *Simulink HDL Coder* de *MathWorks*.

- Aprender a usar dicha herramienta en las diferentes fases del desarrollo de una aplicación de procesado digital de señal, desde el diseño hasta la implementación.
- Comprender en que fases del desarrollo nos puede aportar ventajas frente a otras soluciones y por lo tanto su idoneidad frente a este tipo de aplicaciones.

1.3. Organización del documento

Ésta es la organización de la presente memoria que documenta el Proyecto de Fin de Carrera:

- *Capítulo 1 – Introducción*: introducción al presente proyecto, en el que se define que es lo que se pretende hacer y los objetivos planteados.
- *Capítulo 2 – Estado del arte*: introducción a los dispositivos programables como son las FPGAs, analizando concretamente la plataforma sobre la que se va a implementar la solución.
- *Capítulo 3 – Herramientas de diseño*: análisis de las herramientas disponibles para el desarrollo visual de HDL, con especial hincapié en la solución escogida.
- *Capítulo 4 – Diseño realizado: estimador de frecuencia fundamental*: explicación del diseño realizado mediante la herramienta Simulink HDL Coder.
- *Capítulo 5 – Conclusiones*: conclusiones arrojadas tras las anteriores fases del Proyecto Fin de Carrera. Se centra en la idoneidad de la herramienta Simulink HDL Coder para desarrollos de aplicaciones de procesado digital de señal.
- *Anexo I – Código del módulo de control*: se adjunta el código HDL desarrollado de forma manual para el control del sistema.
- *Anexo II – Código del módulo MAXFFT*: se adjunta el código Matlab que implementa la funcionalidad de calcular el máximo de la salida de la transformada de Fourier.

Capítulo 2

Estado del arte

Contents

En este capítulo se pretende explicar brevemente que son los dispositivos programables, centrándonos en los conceptos que más nos interesan para el presente proyecto, como son las *FPGAs* y más en concreto la tarjeta de evaluación *Altera DE2* y su FPGA *Altera Cyclone II*. No sé pretende profundizar debido a que las FPGAs hoy en día son ampliamente usadas y se conoce bien como funcionan; en caso de querer una buena explicación de su historia, su funcionamiento y demás se recomienda consultar un libro específico en la materia, como puede ser el escríto por *Clive 'Max' Maxfield*[2].

2.1. Dispositivos programables

La inmensa mayoría de los circuitos integrados ofrecen una funcionalidad específica que es determinada a la hora de fabricar el dispositivo; dichos dispositivos son conocidos como ASIC (*Application Specific Integrated Circuit*). En contraposición a esto, existen los dispositivos programables, dispositivos que debido a su diseño no implementan una función concreta, si no que pueden ser programados (o más bien configurados) para implementar una función concreta.

Alrededor de la década de 1970 aparecieron los primeros circuitos integrados programables, los llamados PLD (*Programmable Logic Device*), como son las PROM (*Programmable Read Only Memory*), las PLA (*Programmable Logic Array*) o las PAL (*Programmable Array Logic*), dispositivos que pueden ser programados para realizar una función lógica en base a una suma de productos de las entradas.

Más adelante aparecieron los CPLDs (*Complex Programmable Logic Device*), circuitos compuestos por una serie de PLDs con la opción de poder interconectarlos entre sí para poder implementar funciones más complejas, aunque sin llegar a los niveles de integración que se podían obtener en aquella época con una ASIC.

En 1984, la empresa *Xilinx* fue la pionera al introducir la primera FPGA en el mercado, un dispositivo parecido a los CPLDs pero con la diferencia de estar compuesta con más bloques, que aún siendo más simples que los bloques de los CPLDs, con un flujo de desarrollo más fácil estaban más cerca de la capacidad de integración de un ASIC que de un CPLD, ocupando así el basto vacío existente entre ambas plataformas.

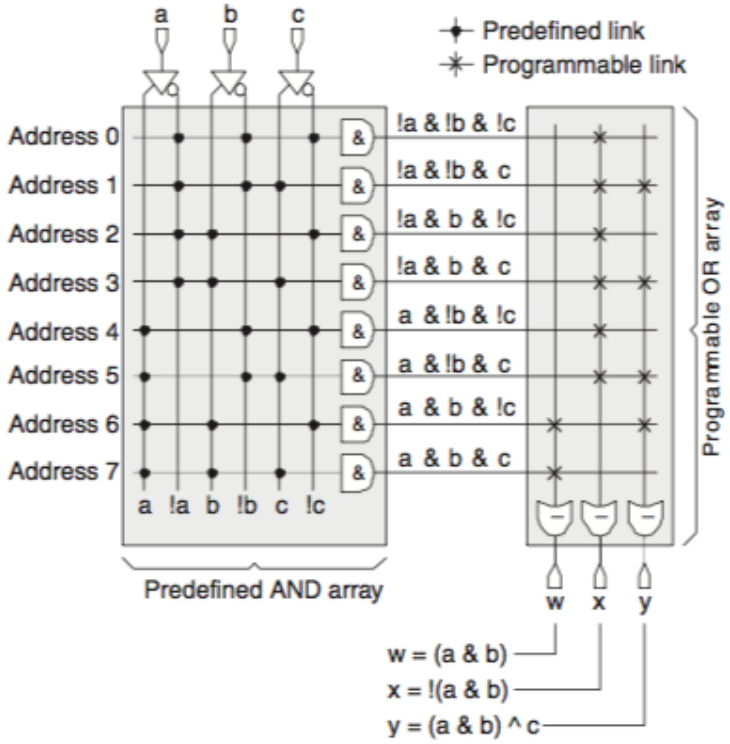

Figura 2.1: Programmable Read Only Memory

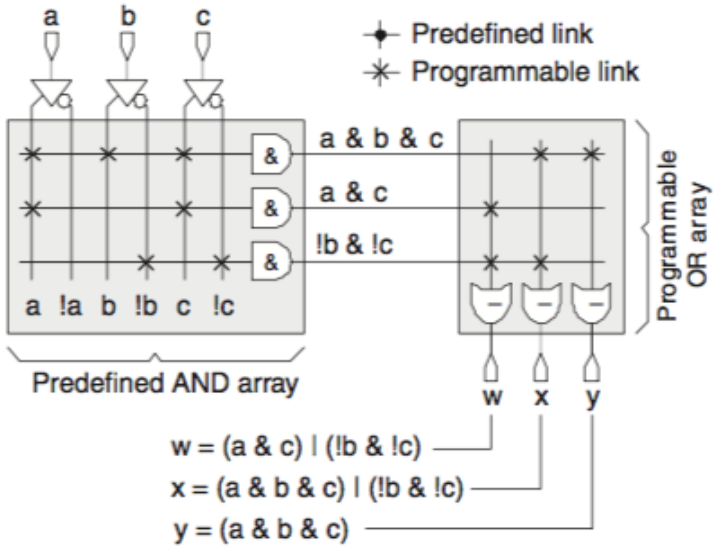

Figura 2.2: Programmable Logic Array

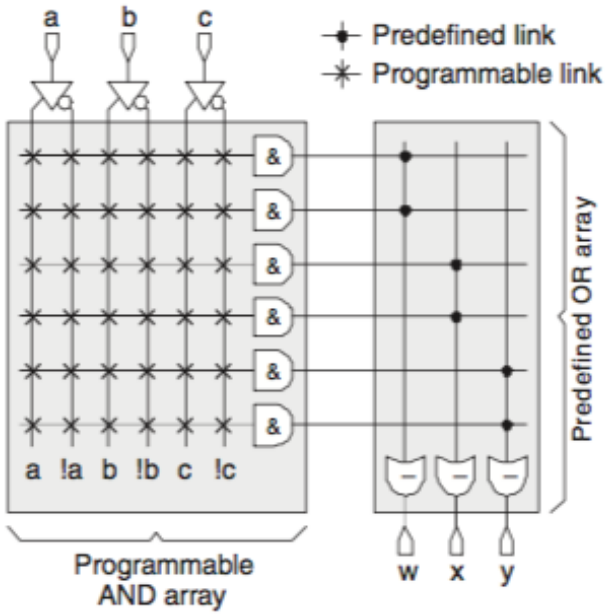

Figura 2.3: Programmable Array Logic

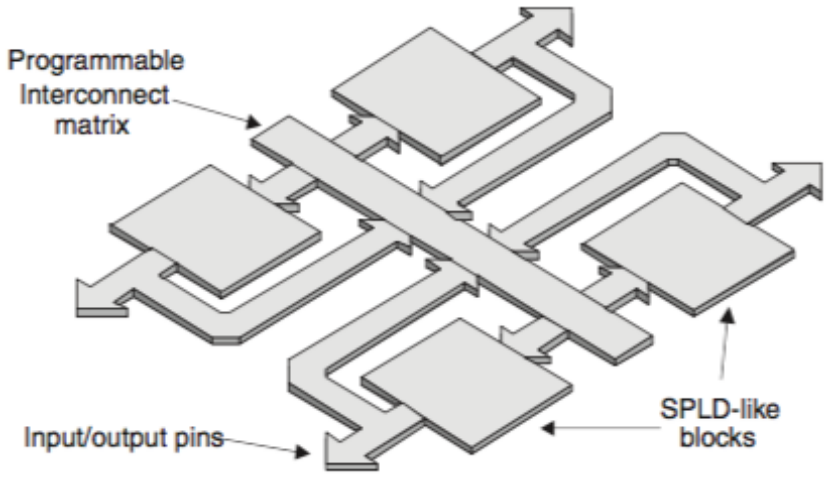

Figura 2.4: Complex Programmable Logic Device

2.2. FPGAs

Las FPGAs están formadas por una serie de bloques configurables, los cuales son capaces de implementar una función lógica y tienen una pequeña memoria, y mediante una red de interconexión permite combinar más de un bloque básico para implementar funciones complejas que no pueden ser implementadas en un solo bloque básico. Dichos bloques básicos son diferentes dependiendo tanto del fabricante como del modelo concreto de la FPGA, siendo un parámetro determinante el número de bits máximo de la función que son capaces de implementar.

La forma más común de implementar una función en un bloque básico se hace mediante una tabla de valores o LUT (*Look-up Table*), guardando los valores de salida que tendría la función a implementar respecto a las entradas en una tabla. La salida de la función puede ir directamente a otro bloque básico mediante la red de interconexión o puede ser almacenada en un biestable, pudiendo implementar de esta forma circuitos síncronos con memoria. En la figura 2.5 puede verse un ejemplo de la estructura interna de una FPGA genérica, en la que los bloques tienen una LUT de 3 bits de entrada y se ve como se implementa en ella una función lógica.

El hecho de usar LUTs para implementar funciones es más costoso en hardware que usar directamente el circuito original que se desea implementar, pero de esta forma se consigue que el dispositivo pueda ser configurado tras su manufacturación (de ahí viene la palabra *Field* de su nombre). La tecnología usada para la configuración del dispositivo ofrece distintas características, siendo la más destacada si la configuración el volátil o no, o pudiendo ser de un solo uso o de más. Si se usan fusibles o antifusibles, el dispositivo no ha de ser configurado cada vez, pero solo se podrá programar una única vez. En el otro extremo están los dispositivos basados en memorias SRAM, los cuales son reprogramables tantas veces como sea necesario, pero la configuración es volátil puesto que depende de la alimentación de dicha memoria SRAM, por lo que cada vez que se inicie el sistema la reprogramación será necesaria, teniendo que añadir para ello más hardware al sistema. En un término medio están las basadas en memorias EEPROM o FLASH, pudiendo ser reprogramadas un número limitado de veces pero sin ser volátiles, por lo que la reconfiguración solo es necesaria en el caso de hacer algún cambio en la aplicación. A día de hoy el uso de las basadas en fusibles o antifusibles es anecdótico, siendo las más usadas las basadas en memorias SRAM y en menor medida las basadas en memorias no volátiles como las EEPROM o FLASH.

Respecto a los fabricantes, existen diversas empresas que comercializan FPGAs, siendo dos las que copan el mercado y por lo tanto grandes compe-

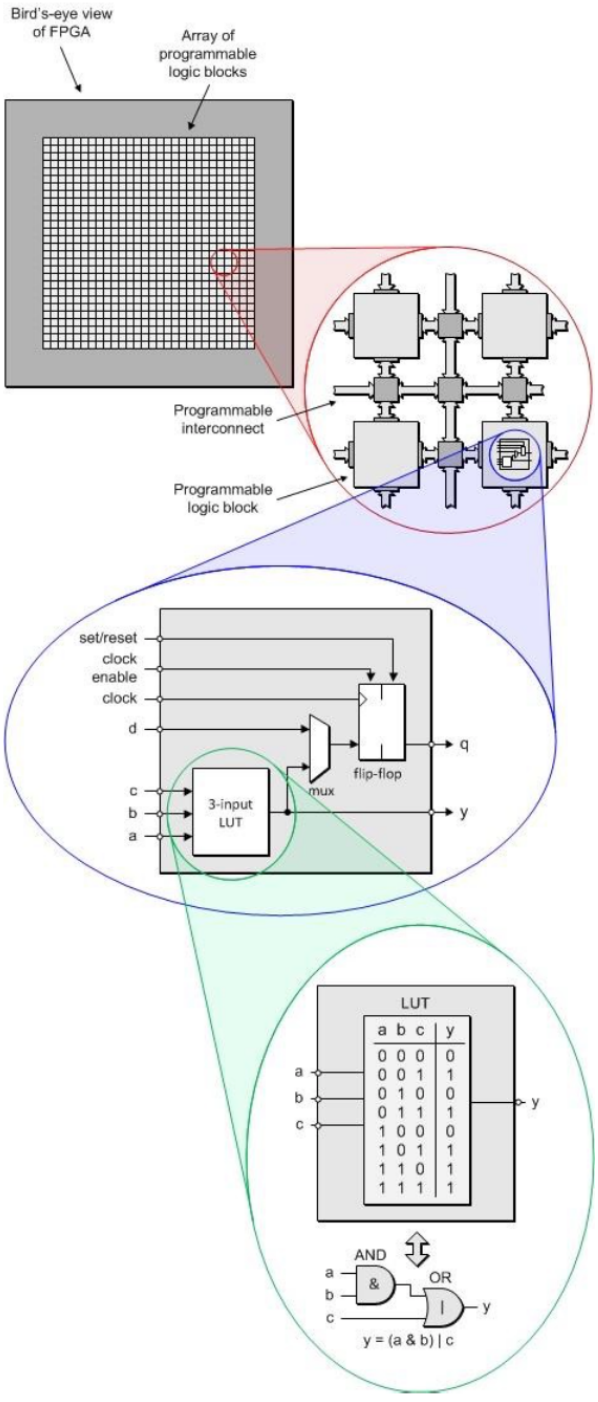

Figura 2.5: Arquitectura genérica de una FPGA

tidoras entre ellas: *Xilinx*, y *Altera*. Además de estas dos hay más empresas que desarrollan sus propias soluciones, aunque con un nivel de ventas mucho más bajo que las dos mencionadas anteriormente, como son *Lattice Semiconductor* y *Actel*, las cuales ofrecen productos más enfocados a las memorias no volátiles en comparación a las dos grandes, que solo ofrecen soluciones basadas en memorias SRAM.

En cuanto a la arquitectura, además de los bloques básicos, hoy en día una FPGA incluye otros tipos de bloque en su interior, con el objetivo de implementar funciones que son muy habituales de forma más eficaz, utilizando menos silicio y por lo tanto con un consumo menor y pudiendo obtener una frecuencia de funcionamiento mayor. Por ejemplo pueden tener bloques multiplicadores, de memoria RAM o para cálculos matemáticos intensivos como los usados en el procesado digital de señal, están todos ellos conectados entre si y con los bloques básicos mediante la misma red de interconexión.

Pese a que al principio la configuración y conexión de los bloques se hacía de forma manual, debido a la cada vez mayor complejidad y capacidad de los dispositivos, hoy en día se usan lenguajes de descripción de hardware como pueden ser *Verilog* o *VHDL*, lenguajes mediante los que se describen los bloques hardware que se quieren implementar en la FPGA, y que tras un proceso de sintetizado generan el fichero de configuración de la FPGA, conocido comúnmente como *bitstream*), encargado tanto de configurar cada uno de los bloques del dispositivo como de la interconexión entre ellos. Estos lenguajes no pueden considerarse lenguajes de programación, ya que lo que hacen no es describir las instrucciones de un algoritmo secuencial, si no que describen bloques de hardware que se van a ejecutar de forma concurrente dentro de la FPGA.

Cuando aparecieron en el mercado su uso se limitaba a implementar sencillas máquinas de estado, tareas de procesamiento limitadas y sobre todo de intermediario entre otros dispositivos. Con el paso del tiempo sus características han mejorado mucho, pasando de tener pocos y simples bloques a contar con muchos y más complejos, por lo que su uso se ha incrementado en áreas en las que se hace un tratamiento de datos intensivo, como pueden ser las telecomunicaciones o el que a nosotros concierne, el procesado digital de señal. Es por todo ello que hoy en día suponen una alternativa a los ASICs, puesto que pueden implementar diseños con el equivalente a millones de puertas lógicas, con un menor *time to market* y con un coste económico más bajo para tiradas menores a las 100.000 unidades.

Como vemos, son muy buenas para tareas de *data-flow*, pero flaquean en comparación a los procesadores convencionales en cuanto a tareas secuenciales, dado que aunque son capaces de implementarlas, el proceso de desarrollo es más costoso. Es por ello, que debido a la capacidad con las que cuentan,

hoy en día son capaces de implementar procesadores completos en su interior, los cuales luego ejecutaran código compilado a su juego de instrucciones como si de un microprocesador al uso se trataran. Además, debido a las cada vez mayores capacidades de integración, hoy en día nos podemos encontrar con plataformas mixtas, que pueden llegar a incluir procesadores multinúcleo ARM que operan a una frecuencia mayor que 1 GHz, junto a FPGAs de alta gama en la que implementar funciones específicas para descargar el procesador, todo ello implementando en el mismo dado de silicio. Ejemplo de esto son las arquitecturas *Zynq* de *Xilinx* o *Altera SoC* de *Altera*.

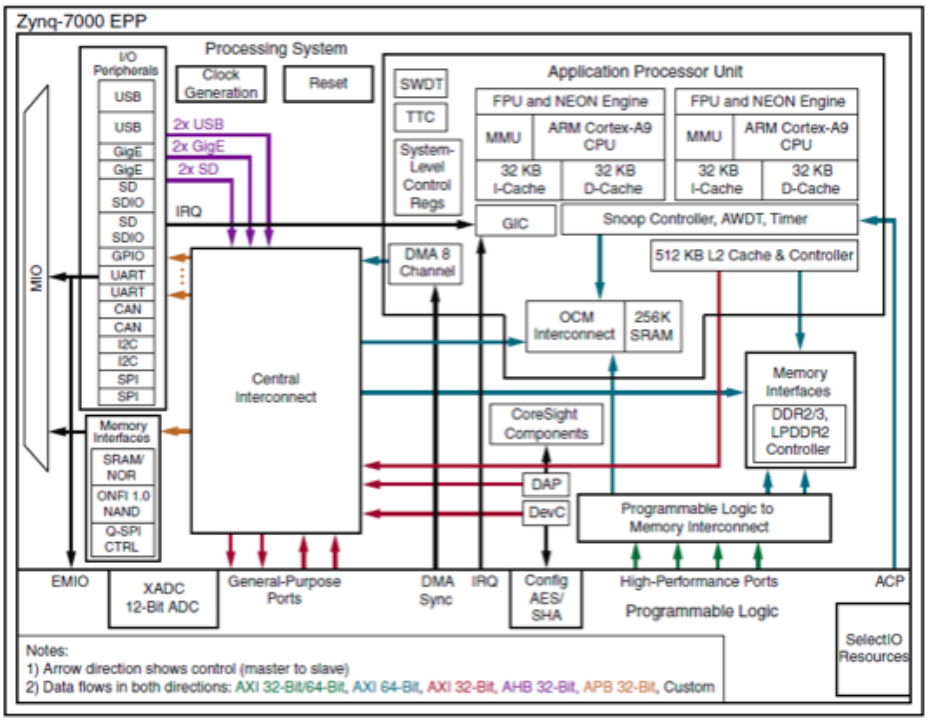

Figura 2.6: Arquitectura del SoC Zynq de Xilinx

2.2.1. Altera Cyclone II

La familia de FPGAs *Cyclone II* de *Altera*, es la segunda revisión de la exitosa familia *Cyclone*, catalogada dentro de las familias de bajo coste de la compañía, a diferencia de las familias *Arria* y *Stratix*, las cuales ofrecen mayores prestaciones. A día de hoy la última revisión de la familia *Cyclone* es la *Cyclone V*, por lo que estamos ante una arquitectura que se ha quedado

un poco obsoleta, puesto que data del año 2004. Pese a ello, Altera sigue soportándolas hoy en día recomendado su uso para nuevos proyectos.

Las principales características de esta familia de FPGAs es que están fabricadas con una tecnología de 90nm de la compañía *TMSC* y que están orientadas a ofrecer un alto rendimiento manteniendo un bajo consumo, con el objetivo de ofrecer una alternativa a los ASIC de la época.

En cuanto a su arquitectura, el bloque principal de estas FPGAs es lo que Altera denomina como LAB (*Logic Array Block*). Estos bloques son en los que se implementarán las funciones de una manera parecida a la que se ha explicado en la sección anterior. Concretamente, cada LAB está compuesto por 16 LEs (*Logic Element*), siendo cada uno de ellos capaz de implementar una función definida por el usuario. Los LABs están agrupados en una matriz bidimensional a lo largo de todo el circuito integrado, pudiendo contener desde 4.608 LEs en los dispositivos más modestos hasta 68.416 en los más avanzados.

Cada uno de los LE está compuesto por una LUT de 4 entradas capaz de implementar funciones de hasta 4 variables, un registro capaz de operar como un biestable D, T, JK o SR, una conexión para el *carry* y la habilidad para interconectarse con otros LEs, ya sean del mismo LAB o de fuera de él. Además, pueden operar en dos modos, los denominados como *Normal*, para implementar funciones lógicas genéricas, y el modo *Arithmetic*, especialmente útil para implementar funciones como sumadores, contadores, acumuladores o comparadores. En las siguientes figuras pueden encontrarse el diagrama de un LE con los elementos que lo componen y la estructura de interconexionado de los LAB.

Disponen de una red global para transmitir las señales de reloj, consistente en hasta 16 canales globales que llegan a todos los bloques. Esta es una de las partes más importantes en una FPGA, puesto que es muy importante que la señal del reloj llegue al mismo tiempo a todos los bloques, para evitar problemas de sincronismo entre las diferentes partes que integren la aplicación que se desea desarrollar. Además, incluye hasta 4 PLL (*Phase-Locked Loops*) capaces de generar señales de reloj a diferentes frecuencias mediante multiplicadores y divisores.

En cuanto a la memoria de la que dispone, además de la ofrecida por cada uno de los LEs, dispone de bloques de memoria M4K, bloques *dual-port* de 4 Kb de capacidad con soporte para paridad, teniendo un total de 4.608 bits. Dichos bloques permiten accesos *dual-port* o *single-port* a palabras de hasta 36 bits a una velocidad de hasta 260 MHz. La cantidad de memoria disponible va desde los 119 hasta los 1.152 Kb de memoria.

Para aplicaciones de procesado digital de señal, la arquitectura *Cyclone II* dispone de bloques multiplicadores dedicados a ello, capaces de implemen-

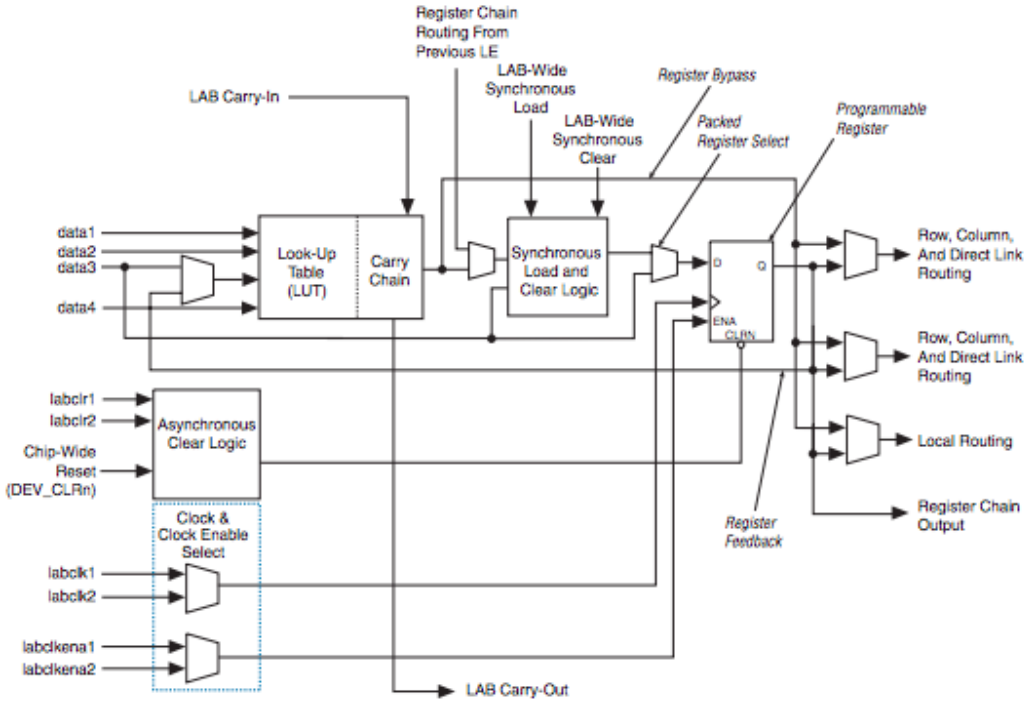

Figura 2.7: Diagrama de un bloque LE

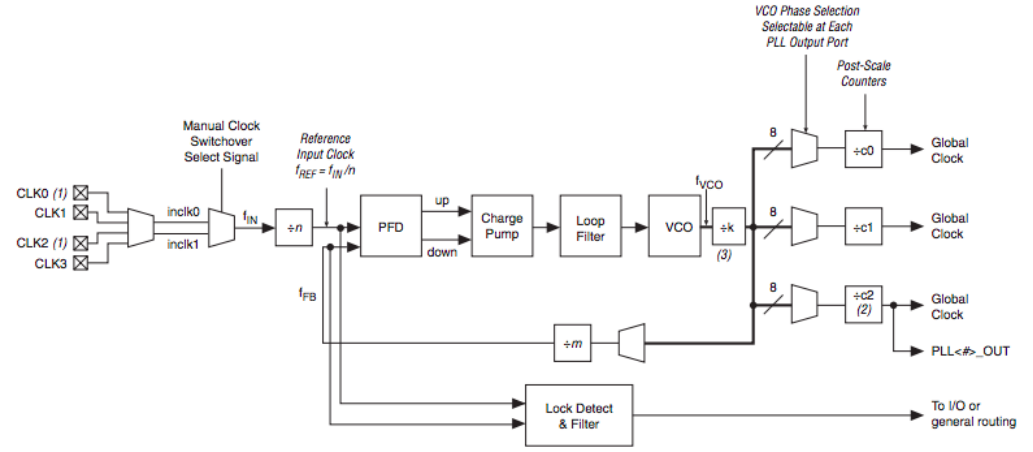

Figura 2.8: Diagrama de un bloque PLL

tar dos operaciones de 9x9 bits o una de 18x18 bits a frecuencias de hasta 250MHz. Dichas operaciones también se puede implementar con el uso de la lógica programable, pero al ser operaciones costosas, y que por lo tanto requieren de bastantes recursos, es bastante común encontrarlas implementadas directamente en hardware para poder hacer este tipo de operaciones de forma más eficiente.

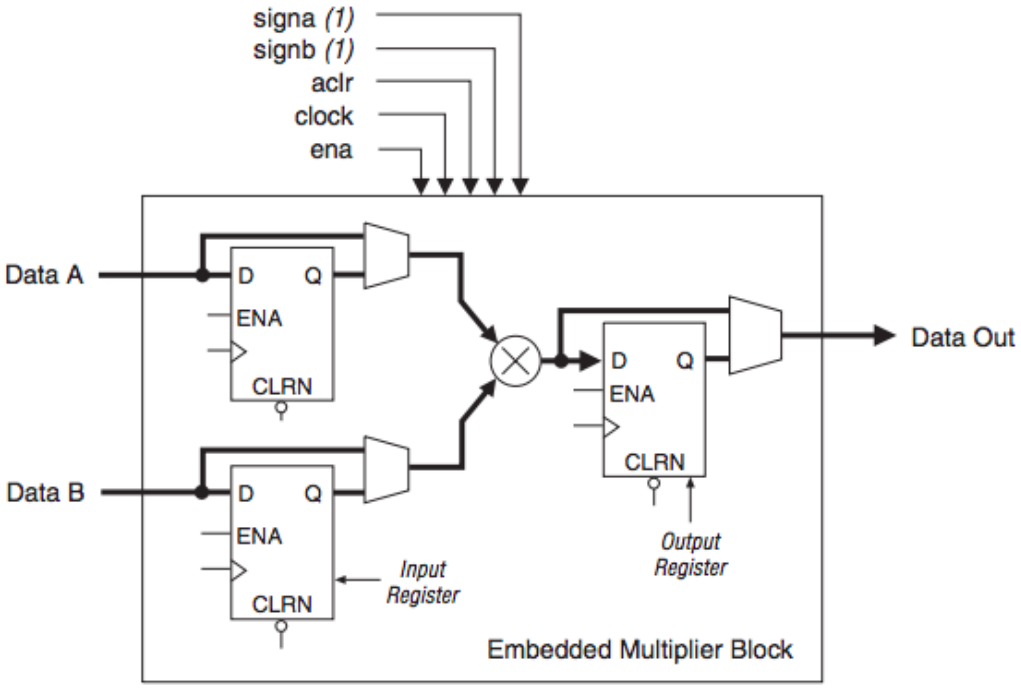

Figura 2.9: Diagrama de un bloque multiplicador

Finalmente, el último tipo de bloque que podemos encontrar dentro de la arquitectura son los IOE (*Input/Output Element*), que son una interfaz entre los pines de entrada salida y la lógica programable. Los pines de entrada y salida soportan diferentes estándares, tanto diferenciales como simples, tal y como pueden ser PCI (33/66 MHz, 32/64 bits), PCI-X, o LVDS a una velocidad máxima de 805 Mbps para entradas y 640 Mbps para salidas.

Como resumen de los diferentes bloques y su disposición dentro de la FPGA, en la siguiente ilustración puede verse como se distribuyen a lo largo y ancho del dado de silicio.

Por último, las herramientas usadas para el desarrollo de aplicaciones para esta familia de FPGAs es el *Quartus II* de la propia Altera, el cual se puede obtener de manera gratuita a través de su página web, previo registro en

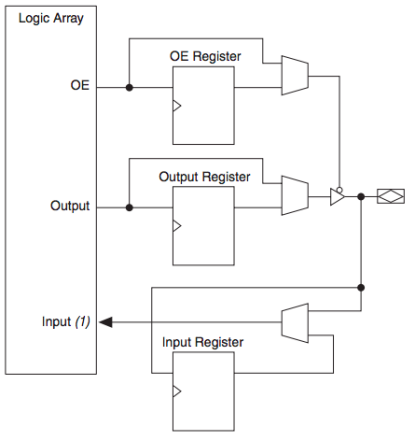

Figura 2.10: Diagrama de un bloque IOE

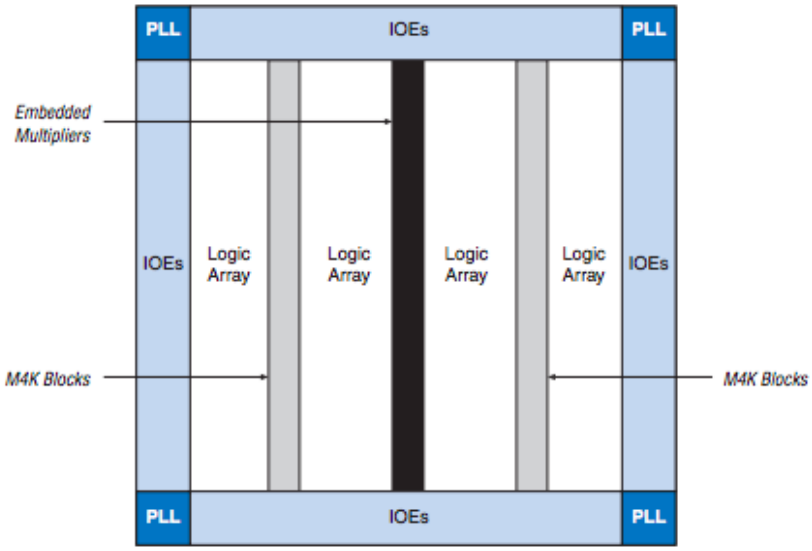

Figura 2.11: Diagrama genera de una FPGA de la familia Cyclone II de Altera

ella. Además de todas las herramientas para diseñar, sintetizar, depurar y demás tareas concernientes al desarrollo de aplicaciones, también se ofrece el procesador *NIOS II*, un *soft-core* que puede ser implementado dentro de la propia FPGA y que luego puede ser programado como si de un procesador al uso se tratara. El procesador embebido *NIOS II* es altamente configurable desde la herramienta *SOPC Builder*, el cual mediante un asistente irá guiando al usuario para que como resultado obtenga el procesador que mejor se adecúe a sus necesidades, pudiendo configurar desde las propias características de la arquitectura del procesador a los periféricos que se incluyen o no. Una vez sintetizado y metido en la FPGA, solo o junto a periféricos desarrollados en algún HDL, este procesador se programará desde un entorno de desarrollo de software basado en Eclipse, pudiendo desarrollar para el como si de un procesador al uso se tratara.

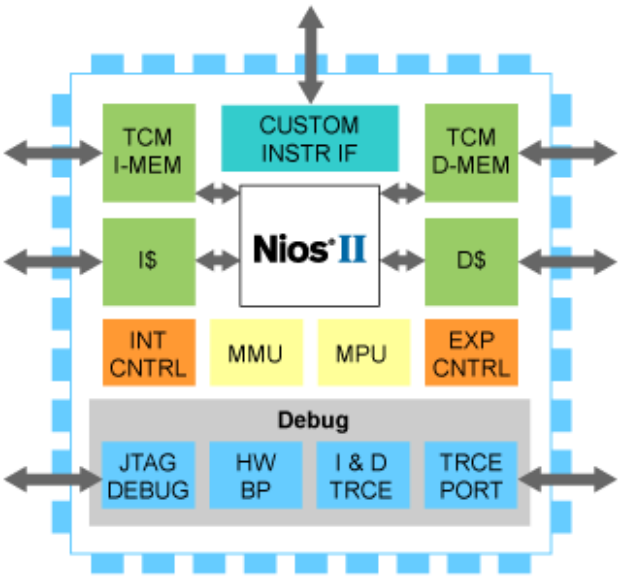

Figura 2.12: Diagrama del procesador embebido NIOS II

Altera Cyclone II 2C35

Dentro de la familia de FPGAs *Cyclone II* de *Altera*, el modelo concreto que usa la tarjeta de evaluación que veremos a continuación es el modelo *2C35*, que se encuentra en la gama media dentro de la familia y que dispone principalmente de las siguientes características técnicas:

- Número de PLLs: 4.

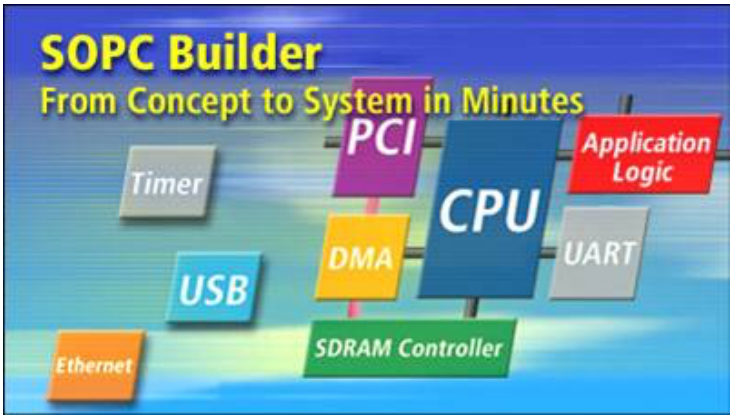

Figura 2.13: Herramienta SOPC Builder

- Número de pines para señales de reloj: 16.
- Número de bloques M4K: 105.
- Capacidad de la memoria ofrecida por los bloques M4K: 483.840 bits.
- Número de bloques multiplicadores: 35.
- Número de pines de entrada/salida para el usuario: 475.
- Número de canales LVDS: 201.
- Número de LEs: 33.216.
- Encapsulado BGA de 672 pines.

2.2.2. Altera DE2

La tarjeta de evaluación que vamos a usar para este proyecto es la placa *DE2* (*Development and Education Board*) de Altera. Tan solo con una FPGA no podemos hacer gran cosa, por lo que esta tarjeta de evaluación tiene todos los elementos que vamos a necesitar para implementar la aplicación de prueba con la que vamos a validar el uso de la herramienta Simulink HDL Coder.

En la siguiente lista podemos encontrar todos los elementos que incluye la tarjeta y que están conectados a la FPGA, por lo que podemos interactuar con ellos:

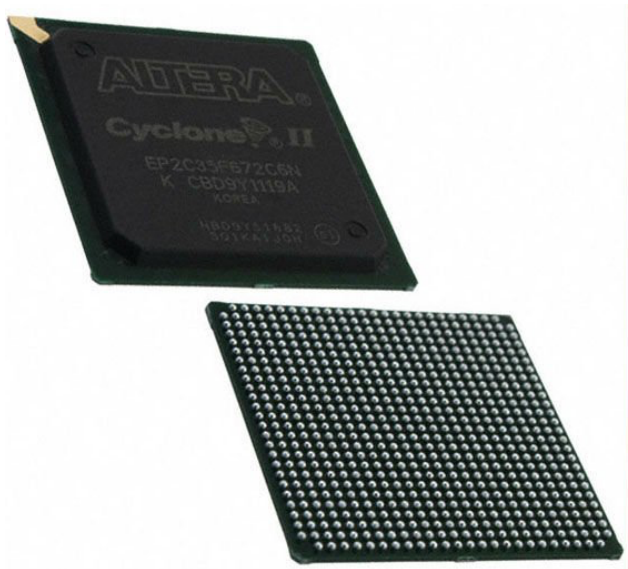

Figura 2.14: Encapsulado de la FPGA Altera Cyclone II 2C35

- FPGA Altera de la familia Cyclone II 2C35, concretamente el modelo *EP2C35F672C6N*.
- Altera Serial Configuration device (EPCS16).
- USB Blaster para programar y depurar.
- Memoria SRAM externa de 512 KB.
- Memoria SDRAM externa de 8 MB.
- Memoria Flash externa de 4 MB.
- Socket para tarjetas SD.
- 4 pulsadores.
- 18 switches.
- 18 LEDs rojos.
- 9 LEDs verdes.
- Oscilador de 50 y 27 MHz para usar como fuentes de reloj.

- CODEC de audio de 24 bits, con conectores mini-jack de entrada de línea, salida de línea y entrada para micrófono.
- DAC de 10 bits para señales VGA con conector hembra.
- Decodificador de señales NTSC y PAL de televisión con conector coaxial de entrada.
- Controlador Ethernet 10/100 Mbps con conector RJ45.
- Controlador USB Host/Slave con conectores de tipo A y B.
- Transceptor RS232 con conector DB9.
- Conector PS/2 para teclado o ratón.
- Transceptor IrDA para transmisión de datos infraroja.
- 2 slots de 40 pines cada uno para uso del usuario con protecciones mediante diodos.
- 8 displays de 7 segmentos con punto.
- Display LCD de 2x16 caracteres retroiluminado.

Como podemos observar, se trata de una tarjeta de evaluación muy completa debido a que dispone de multitud de periféricos tanto de entrada como de salida. Además, en el CD que acompaña a la tarjeta se pueden encontrar multitud de ejemplos que hacen uso de ellos, junto a una gran cantidad de documentación con la que el usuario podrá aprender los entresijos de la tarjeta. Para nuestro caso, la tarjeta cumple con los requisitos mínimos que necesitamos para la implementación.

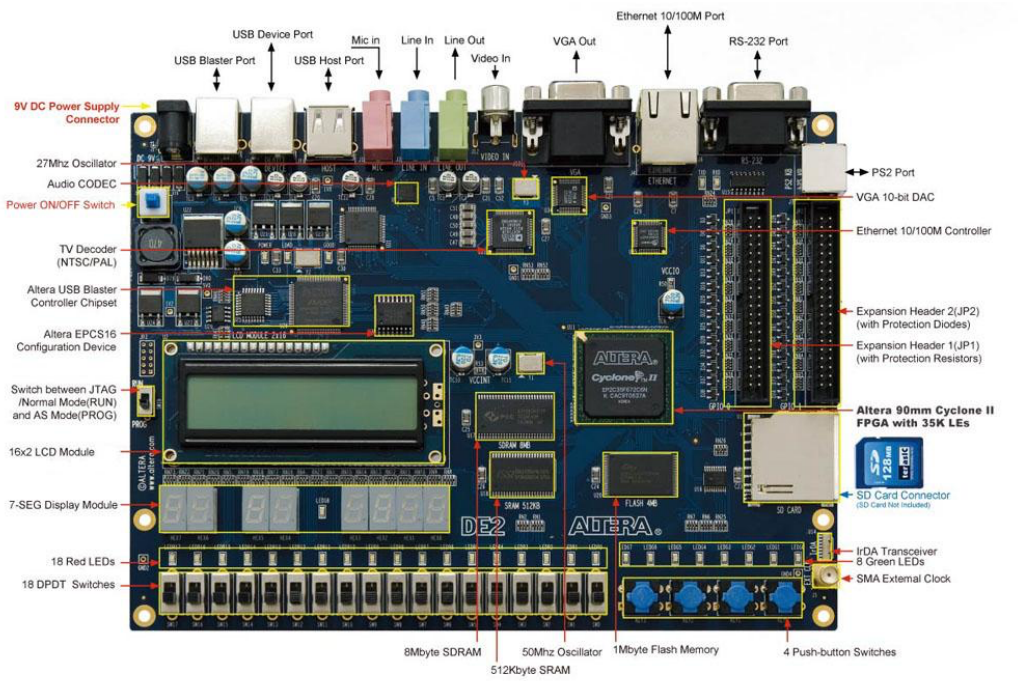

Figura 2.15: Vista general de la tarjeta de evaluación DE2 de Altera

Capítulo 3

Herramientas de diseño

Contents

Tal y como se ha mencionado en la sección 2, la creciente complejidad en los dispositivos programables ha ocasionado que cada vez se dispongan de más herramientas que facilitan el desarrollo de aplicaciones para este tipo de plataforma.

Con las primeras FPGAs, al igual que con los primeros dispositivos programables, la configuración del dispositivo, la cual permite definir el comportamiento que éste tendrá, es decir, la función que implementará, era hecha mediante un proceso manual, en el que a *grosso modo* se definían los contenidos de las LUT y las conexiones entre bloques básicos. La capacidad de los dispositivos de hoy en día hace que esto sea una tarea hercúlea, es por eso que existen los lenguajes de descripción de hardware como pueden ser VHDL o Verilog. Mediante dichos lenguajes, se consigue subir el nivel de abstracción hasta el nivel de que con ellos se describe el funcionamiento del hardware que se quiere que luego se implemente en la FPGA. Es decir, si se quiere implementar un contador, mediante el HDL que se escoja se describirán su interfaz (entradas y salidas) y su comportamiento (a tales entradas, tales salidas). Será tarea del entorno de desarrollo de la arquitectura que se escoja convertir dicha descripción en conexiones y contenido de las LUT que componen la FPGA, en un proceso que toma varios pasos como son el sintetizado o la colocación del hardware dentro del dispositivo. Este proceso evidentemente es muy dependiente del hardware, por lo que pese a que los ficheros de entrada sean los mismos, los ficheros de salida para FPGAs de distinto fabricante son completamente diferentes. Además, los fabricantes no facilitan demasiada información sobre este proceso, por lo que no se dispone de herramientas de terceros para este proceso.

Antes de realizar una implementación en una FPGA mediante un lenguaje de descripción de hardware, se han de realizar una serie de pasos previos, como pueden ser definir lo que se quiere implementar mediante una captura de requisitos, diseñar de la arquitectura general, diseñar cada uno de los módulos a implementar, definir *test* que ayuden a validar la implementación, etc. Pese a que todas estas operaciones pueden hacerse de forma manual, existen ciertas herramientas muy usadas en el mundo de la ingeniería a todos los niveles, que facilitan dichas tareas. A continuación se presentará una herramienta muy usada en el campo del procesado digital de señal que además es una herramienta clave para este proyecto.

3.1. Matlab

Matlab (*Matrix Laboratory*) es un entorno de computación numérica multiparadigma, desarrollado por la empresa MathWorks. Las principales tareas

para las que está pensado Matlab es para manipulación de datos numéricos en matrices, *ploteo* de datos y funciones y la implementación y validación de algoritmos. Además de ello, hoy en día cuenta con interfaces para interactuar con programas escritos en C/C++, Java, Fortran o Python entre otros, además de una serie de paquetes opcionales, comercializados como *toolbox*es que hacen que esta herramienta sea usada hoy en día por gente de los campos de la ingeniería, la ciencia o la economía, tanto en el ámbito de la investigación académica como en el de la industria.

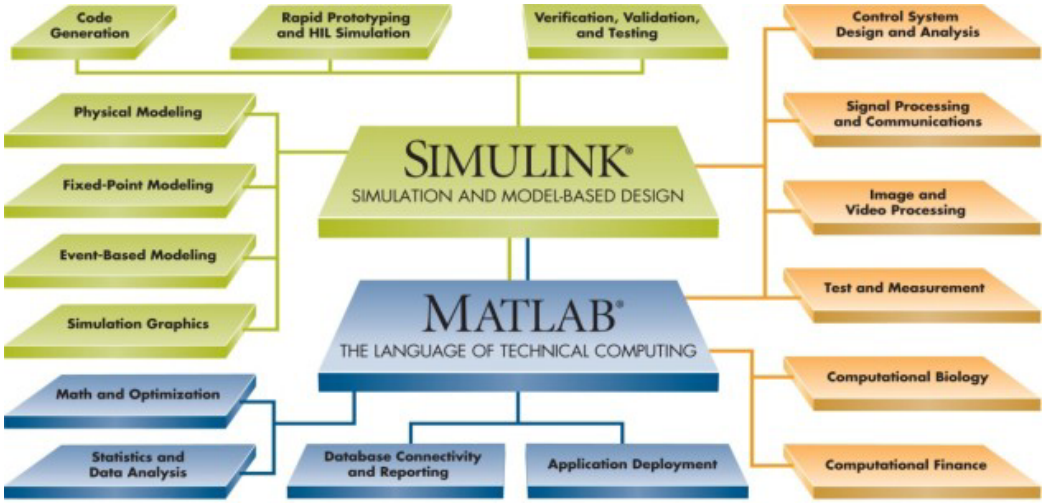

Figura 3.1: Ecosistema de las herramientas Matlab y Simulink

El origen de Matlab se remonta al año 1970, en el que *Cleve Moler*, jefe del departamento de computación de la universidad de Nuevo México, creo un pequeño pack de software para sus estudiantes. A partir de ahí, y visto su potencial, decidieron crear la compañía *MathWorks* en el año 1984 para comercializar Matlab, el cual fue reescrito en C.

A día de hoy, dispone de un potente lenguaje de programación propio, el cual también recibe el nombre de la herramienta, capacidad de computación simbólica, creación de interfaces gráficas y muchas más opciones lo cual lo convierten en una herramienta básica en muchos procesos de ingeniería.

3.1.1. Simulink

Simulink es un añadido a Matlab, desarrollado también por MathWorks, que proporciona un entorno de programación gráfico para el modelado, la simulación y el análisis de sistemas dinámicos multidominio. Para ello, dispone de

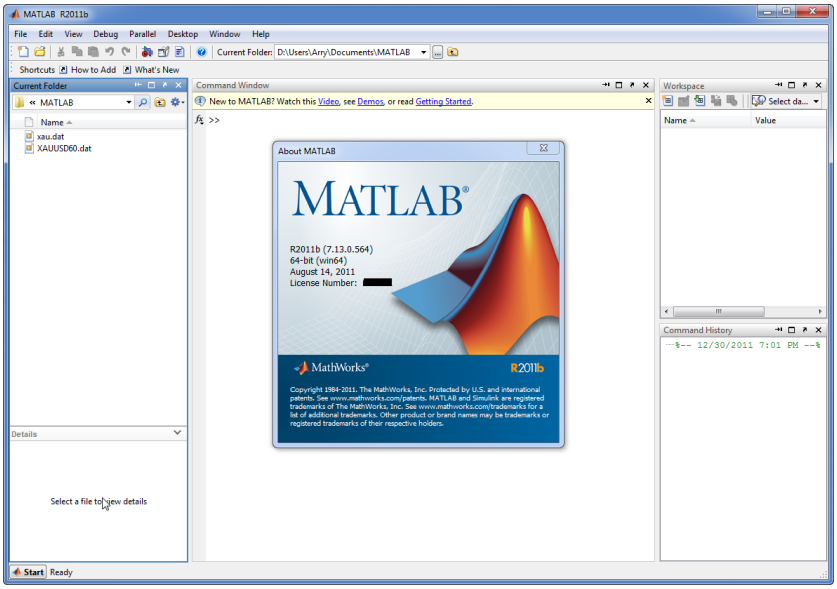

Figura 3.2: Entorno de computación numérica Matlab

una herramienta gráfica con la que diseñar diagramas y de librerías en las que se encuentran bloques básicos con los que programar. Por sus características, Simulink es ampliamente usado en ámbitos como la ingeniería automática, el procesado digital de señales o el diseño basado en modelos.

Además de los bloques estándar ofrecidos por la propia herramienta, existen en el mercado infinidad de colecciones de bloques tanto de la propia Simulink como de empresas externas que permiten extender la funcionalidad de Simulink hacia nuevos dominios, como puede ser por ejemplo la electrónica analógica, el modelado de máquinas eléctricas, etc. También existen diferentes herramientas que permiten comunicar Simulink con herramientas de simulación externas, como puede ser el caso de Plex, para simulaciones de electrónica de potencia; de esta forma, mediante un modelo de Plex se simulará la parte de la electrónica de potencia, mientras desde Simulink se podrá modelar el control de dicha electrónica.

Otra de las características a destacar de Simulink, es una herramienta para generar automáticamente código C para implementaciones en tiempo real. El origen de esta herramienta proviene de que para poder simular los modelos de forma más rápida, Simulink permite convertir el modelo o parte de él a un binario ejecutable sobre un sistema operativo propio de MathWorks, conocido como xPC Target. Debido a la evolución de dicha transformación de Simulink a código C, hoy en día dicho código autogenerado puede ser

usado en plataformas embebidas debido a su calidad, pudiendo incluso crear código certificable para diversos estándares como puede ser el IEC 61508[3], una estándar internacional sobre confiabilidad en dispositivos electrónicos.

Además de ello, Simulink dispone de diferentes herramientas que ayudarán en la validación y verificación del modelo que se esté desarrollando, con herramientas que ayudan a generar diferentes entradas para así poder validar el modelo. De esta forma, Simulink es capaz de detectar problemas de división por cero, *overflow* de números enteros, análisis de números en coma fija, etc.

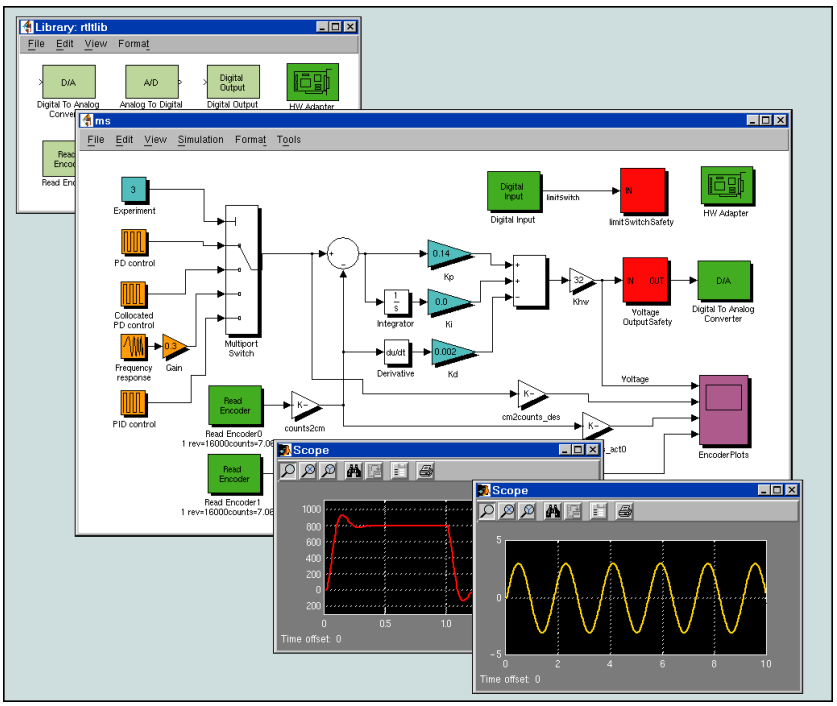

Figura 3.3: Entorno de modelado gráfico Simulink

3.2. Análisis de antecedentes

Tal y como se ha visto, debido a sus características Simulink es una herramienta ampliamente utilizada en el ámbito del procesado digital de señal, siendo su uso típico el diseño y la verificación y validación de los algoritmos que más adelante se implementarán en un dispositivo programable, como puede ser una FPGA.

Debido a ello, y teniendo en cuenta que Simulink es fácilmente ampliable mediante *toolbox*es, los principales fabricantes de FPGAs como son Altera y

Xilinx ofrecen sendas soluciones que permiten exportar modelos de Simulink a sus dispositivos programables, evitando así el paso manual del modelo desarrollado en Simulink a un lenguaje de descripción de hardware. Además de ello, dichas herramientas también permiten verificar y validar el diseño en la propia FPGA, haciendo lo que comúnmente se conoce como *Hardware in the Loop*, donde se sustituye la parte que va en la FPGA del modelo por la propia FPGA, conectándola al PC para nutrirla de entradas y analizar las salidas.

3.2.1. DSP Builder

DSP Builder[4] es la solución que ofrece Altera para desarrollar mediante el entorno Simulink modelos para sus FPGAs, mejorando de esta forma el tiempo de desarrollo de las aplicaciones que se quieran desarrollar.

Está centrado en el desarrollo de algoritmos de procesado digital de señal, proveyendo para ello una librería de bloques específica, con bloques tanto generales como específicos para procesado digital de señal que son sintetizables en las FPGAs de Altera. También existe la opción de usar los bloques básicos de Simulink, que aunque no pueden ser sintetizados sirven para la preparación de los testbench que ayudaran en los procesos de validación y verificación del modelo a sintetizar.

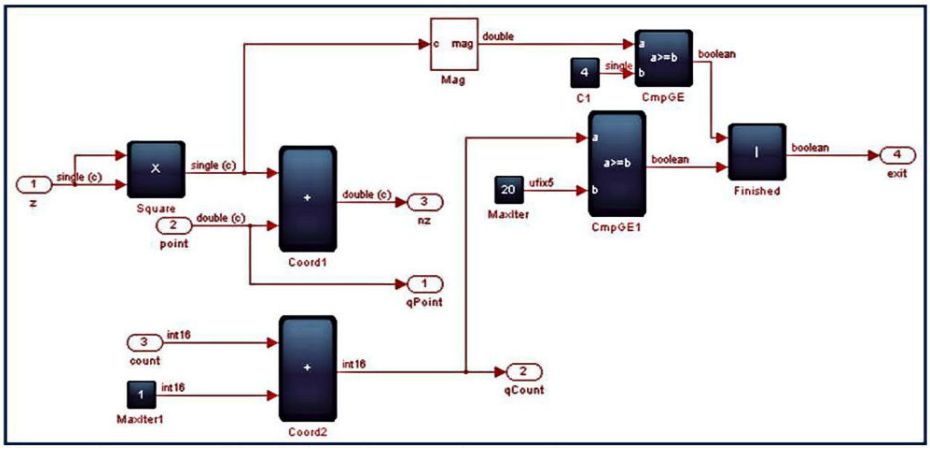

Figura 3.4: Modelo de ejemplo realizado con DSP Builder

Debido a la menor utilización de hardware, y dado que al desarrollar una aplicación concreta se puede analizar con que precisión se necesita desarrollar la aplicación, en las FPGAs se opta por el uso de números en coma fija. Esta herramienta ofrece ayuda para dichos desarrollos, analizando si la precisión

escogida es suficiente para que cumpla con los requisitos establecidos. Además de ello, en caso de que trabajar con números en coma fija no sea suficiente, DSP Builder ofrece la posibilidad de trabajar con números en coma flotante siguiendo el estándar IEEE 754; dicha funcionalidad tan solo está disponible para las FPGAs de ama alta como pueden ser las Arria 10 y las Stratix 10.

En cuanto a los requisitos para poder utilizar dicha herramienta, se deberá tener disponible Matlab, Simulink y el añadido *Fixed-Point Designer* de la propia MathWorks.

Respecto a la librería de bloques que ofrece, están divididos en tres categorías:

- Filtros: de tipo FIR, IIR y CIC.
- FFTs: multiplicaciones de números complejos, generadores de *twiddle*, tablas *butterfly* y retardos.
- Primitivas: operaciones matemáticas (suma/resta, valor absoluto, ...), operaciones lógicas (and, or, ...), comparadores, convertidores entre diferentes precisiones, retardos, contadores y CORDIC (bloque que permite calcular todo lo relacionado con las operaciones trigonométricas.

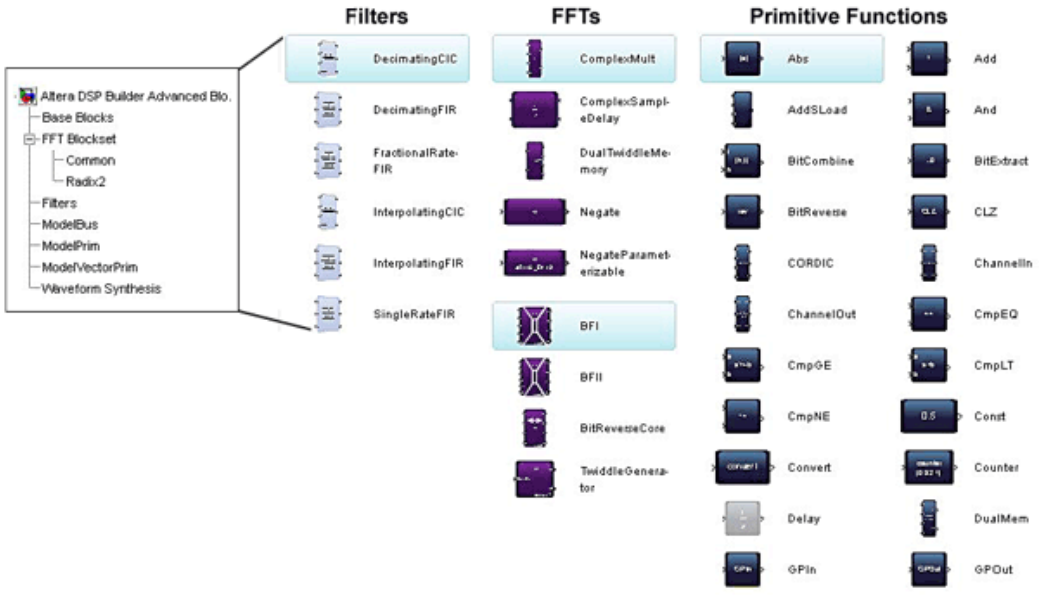

Figura 3.5: Librería de bloques de DSP Builder

Además, se incluyen diversos ejemplos que implementan aplicaciones típicas de procesado digital de señal, como pueden ser transformadas de Fourier

de diferente precisión, filtros CIC, FIR e IIR, controles de motor y un largo etcetera.

Por último, DSP builder se integra con otras herramientas usadas durante el proceso de desarrollo, como pueden ser el propio Quartus II (entorno de desarrollo de Altera) o ModelSim (entorno de simulación). El flujo de trabajo con estas herramientas está perfectamente integrado desde el propio DSP Builder, no es para nada un proceso manual para el usuario. También existe la posibilidad de verificar la implementación en la tarjeta física mediante la herramienta HDL Verifier en un entorno de *Hardware in the Loop*.

3.2.2. System Generator

System Generator[5] es la solución que ofrece Xilinx para el desarrollo en base a modelos para sus dispositivos programables. Al igual que DSP Builder, éste también está compuesto por una serie de herramientas que ayudarán en el desarrollo de la aplicación, junto a una librería de bloques que son sintetizables.

Al igual que su homólogo de Altera, System Generator está focalizado en el desarrollo de algoritmos de procesado digital de señal, preparando el entorno de desarrollo gráfico Simulink para poder implementar directamente las aplicaciones modeladas en él en los dispositivos programables de la propia casa.

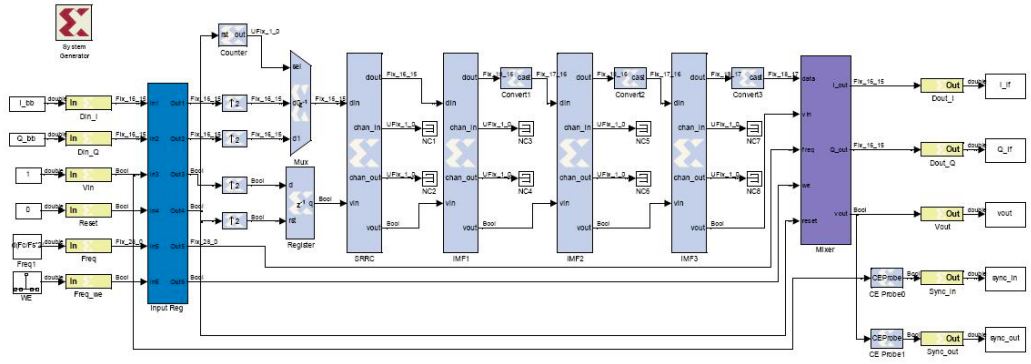

Figura 3.6: Modelo de ejemplo realizado con System Generator

El software necesario para poder usar esta herramienta son Matlab y Simulink de MathWorks, y la suite ISE de Xilinx, en la que se incluye la herramienta, además de ciertas herramientas opcionales como por ejemplo ModelSim para realizar co-simulaciones de los modelos.

Al igual que DSP Builder, System Generator ofrece su propia librería de bloques sintetizables, más amplia que la de su competidor. Además de los bloques avanzados para DSP, en los que podemos encontrar bloques como transformadas de Fourier o implementaciones del algoritmo CORDIC, System Generator ofrece una serie de bloques estándares de hardware, tales como colas FIFO, memorias RAM, registros, sumadores, acumuladores, etc. Finalmente, al igual que DSP Builder, al no poder sintetizar más que sus propios bloques, ofrece remplazos para los bloques básicos de la librería Simulink, tales como constantes, multiplexores, codificadores, etc. Una de las características importantes es que esta herramienta permite trabajar a un alto nivel de abstracción, pero también da la posibilidad de trabajar a más bajo nivel, pudiendo usar primitivas que se encuentran en las FPGA de Xilinx directamente, para así hacer un mejor uso de los recursos disponibles. De esta forma, por ejemplo, en un modelo puede instanciarse un bloque de DSP (DSP48A) de los que incluye la FPGA para la que se esté desarrollando y configurarlo de la manera que mejor convenga para el desarrollo; gracias a ello se obtendrá un mejor resultado en cuanto a recursos y rapidez de la solución, pero en cambio se habrá de invertir un mayor tiempo en el desarrollo.

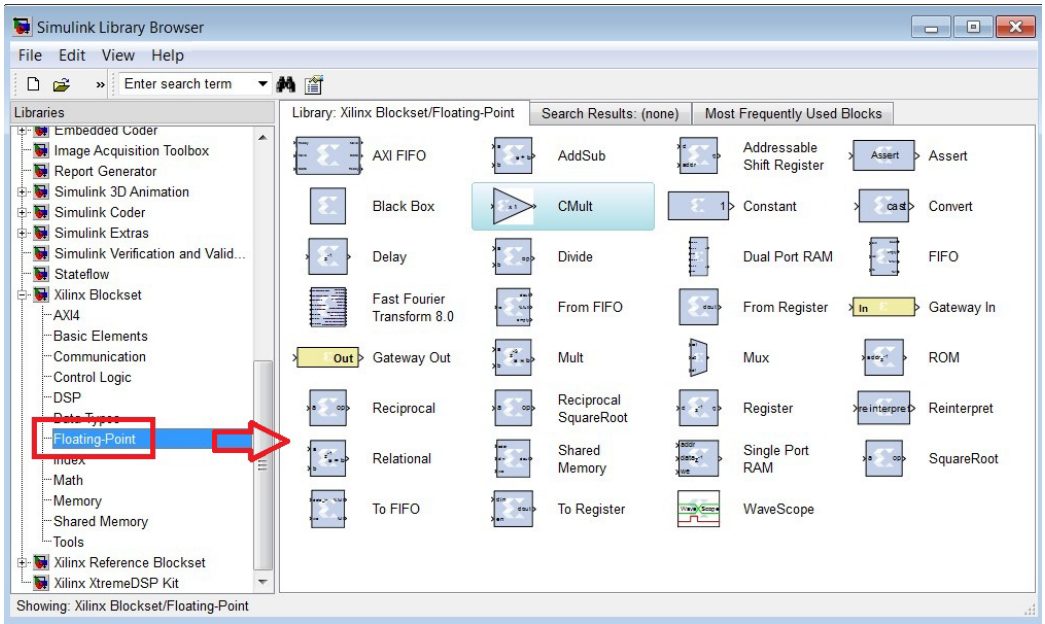

Figura 3.7: Librería de bloques de la herramienta System Generator

Toda la librería de bloques viene perfectamente documentada, teniendo el usuario plena información sobre los bloques, incluyendo los recursos consumi-

dos por cada bloque en cada arquitectura de FPGAs soportada. Al igual que en DSP Builder, esto es debido a que la herramienta está muy optimizada para sus propias FPGAs, por lo que dependiendo de que familia de FPGA se este usando la implementación de un bloque puede variar de forma sustancial. Además de la documentación de los bloques, también se incluyen una serie de ejemplos con implementaciones típicas que se pueden realizar con la ayuda de la herramienta.

Por último, el flujo de trabajo de la herramienta es prácticamente el mismo que en la herramienta de Altera, incluida la utilización de asistentes que guiarán en el proceso de desarrollo y la conexión con herramientas externas como ModelSim. Además, también existe la opción de simular mediante Hardware in the Loop la implementación en un tarjeta física de evaluación.

3.3. HDL Coder

Tal y como se ha visto en la sección anterior, los dos mayores fabricantes de FPGAs ofrecen sendas soluciones que permiten a los desarrolladores crear aplicaciones principalmente focalizadas en el procesado digital de señal, usando para ello como herramienta de modelado Simulink, para posteriormente, tras validar funcionalmente dicha aplicación sobre la propia plataforma Simulink, trasladar dicho modelo directamente a una de sus FPGAs.

Pese a que ambas herramientas trabajan sobre Simulink, ambas disponen de librerías de bloques independientes que son capaces de sintetizar. Debido a ello, como ventaja cabe destacar la buena optimización de las soluciones; como la herramienta es del propio fabricante de FPGAs, conoce bien su arquitectura interna y el proceso de traducción está muy optimizado. Por contra, como desventaja, los modelos realizados tanto con DSP Builder como con System Generator solo podrán ser sintetizados para las FPGAs de cada uno de los fabricantes; de esta forma, al cambiar de fabricante de FPGAs los modelos previamente desarrollados con una de las herramientas deberán ser debidamente transformados a la otra herramienta, con el coste que ello conlleva, puesto que prácticamente habrá que volver a validar todo el diseño.

La herramienta HDL Coder surgió en el mercado de la mano de MathWorks con el mismo principio que DSP Builder de Altera y System Generator de Xilinx, pero con diferencias bastante notables que hacen que sea una herramienta muy interesante.

La principal diferencia es que la herramienta no está ligada a ningún fabricante de FPGAs. Esto lo consigue mediante la creación directa de código HDL, tanto Verilog como VHDL, del modelo que se quiera exportar. De esta forma, un modelo desarrollado en Simulink y que sea sintetizable puede ex-

portarse a código HDL y luego sintetizarlo en cualquier fabricante de FPGAs, tanto Altera como Xilinx o cualquier otra fabricante. De esta forma desaparece la principal desventaja que se ha analizado anteriormente: en caso de cambiar de fabricante de dispositivos programables el modelo a sintetizar podrá ser el mismo. Esto se consigue mediante la utilización de la librería de bloques de Simulink, es decir, si se desea implementar un multiplexor, en el modelo se colocará el multiplexor que ofrece Simulink en su librería de elementos básicos.

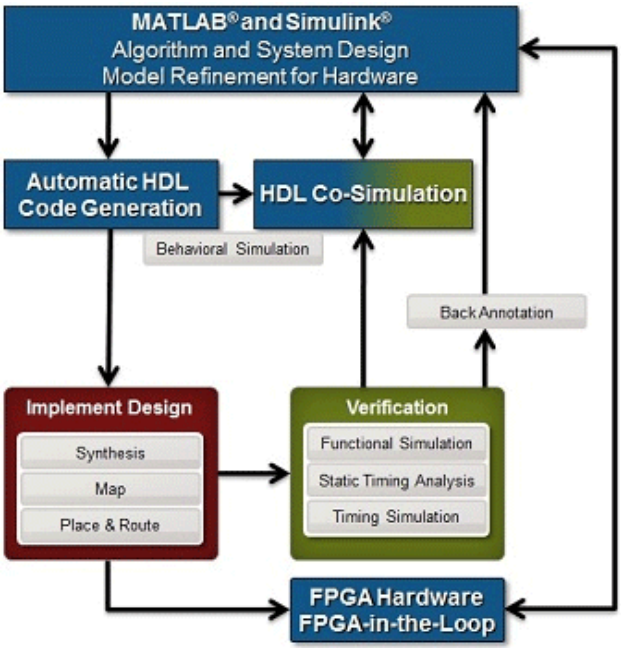

Figura 3.8: Flujo de trabajo con la herramienta HDL Coder

A priori este hecho puede llevar a pensar que todos los bloques disponibles en la librería de Simulink son sintetizables, pero lamentablemente esto no es así. Hay bloques que directamente no se pueden sintetizar, bien porque no es físicamente posible o bien porque la traducción a HDL no está implementada para ese bloque, además de que ciertos bloques solo pueden ser sintetizados si se cumplen ciertas condiciones en su configuración. Para facilitar el desarrollo de modelos que luego puedan ser exportados a HDL para poder ser sintetizados, HDL Coder ofrece una vista de la librería con los bloques que son sintetizables, por lo que si el desarrollo se limita a dichos bloques no habrá ningún problema. Cabe destacar que con cada nueva versión de HDL Coder cada vez son más los bloques soportados por la herramienta.

Como se ha visto las dos herramientas de Xilinx y Altera ofrecen la capacidad de trabajar con números en como flotante. En caso de que el desarrollo requiera de dicho formato de representación numérica, HDL Coder no es una solución válida, puesto que no permite trabajar con ellos. Esto es debido a que las operaciones con números en coma flotante son relativamente complejas, por lo que la forma correcta de implementarlas es mediante una UAL (Unidad Aritmético Lógica). La implementación de dicha UAL en un dispositivo programable es costosa y está muy ligada a la arquitectura específica con el objetivo de no consumir muchos recursos. Altera y Xilinx conocen bien sus dispositivos y pueden ofrecer soluciones para trabajar con números en formato IEEE 754, pero desgraciadamente HDL Coder no.

En cambio, trabajar con números en formato de coma fija, algo a lo que siempre se recurren en dispositivos programables cuando se quiere tratar con números decimales, es algo que está relativamente facilitado por HDL Coder, puesto que en los bloques en los que esto afecta se puede configurar con que precisión se desea trabajar. Por ejemplo en el bloque sintetizable que permite calcular la transformada rápida de Fourier, mediante sus propiedades pueden configurarse las precisiones internas de las multiplicaciones, de las sumas, los valores intermedios, etc.

En cuanto a la forma de trabajar con HDL Coder, lo recomendable es tener un submodelo con el diseño que se quiere exportar a HDL. Las entradas y salidas de dicho submodelo serán luego las que tendrá el simbolo resultante en lenguaje HDL, el cual deberá ser sintetizado en el entorno de desarrollo del fabricante de FPGAs escogido. En el modelo, además del submodelo a exportar también se encontrarán otro tipo de módulos que facilitarán la validación del modelo, como pueden ser generadores, visores de datos, comparadores, bloques de diseño alternativos, etc.

Una vez se tenga el diseño, el primer paso es validarlo desde la propia herramienta Simulink. Los bloques que se hayan usado para el diseño, además de ser sintetizables también serán ejecutables desde la propia herramienta Simulink, por lo que se puede ejecutar dicho modelo y comprobar que la salida obtenida es la esperada respecto a las entradas. Este paso servirá para comprobar la validez de la implementación, por ejemplo en la elección de la precisión de los números escogida. Una vez esto esté realizado, tocará exportar dicho submodelo a un HDL. Para ello, lo primero será ejecutar un asistente que viene con HDL Coder, que verificará que se cumplen todos los requisitos que ha de tener el modelo en cuestión para ser exportado. En caso de que haya algo mal la herramienta lo notificará, especificando claramente cual es el problema y sus posibles soluciones.

Una vez el diseño es apto para ser exportado, la herramienta HDL Coder será capaz de crear los ficheros HDL sintetizables en el formato escogido

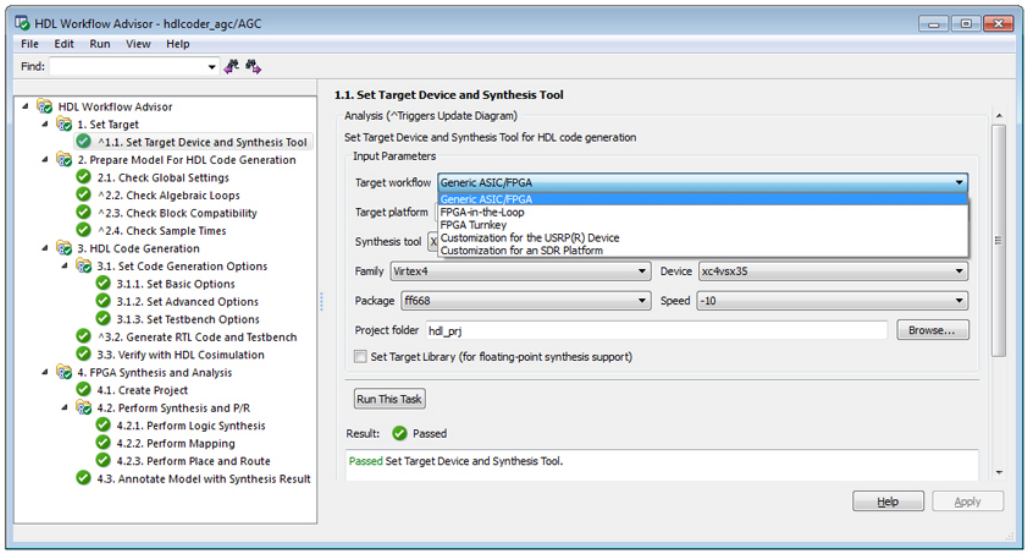

Figura 3.9: HDL Workflow Advisor, herramienta que guiará el correcto desarrollo de una aplicación para HDL Coder

(Verilog o VHDL). Además de ello, también existe la opción de generar automáticamente un testbench en HDL no sintetizable; dicho testbench funciona de una manera muy básica: crea una instancia del submodelo que se ha exportado a HDL sintetizable y le va suministrando todas las entradas que se han usado en la validación previa en Simulink. Posteriormente, va comprobando que la salida del módulo sintetizable es la misma que la obtenida en la validación en Simulink. De esta forma se asegura que el funcionamiento del código HDL generado automáticamente es el mismo que el validado en Simulink.

Tras la realización de estos pasos el uso de la herramienta HDL Coder finaliza. Los siguiente pasos deberán ser realizados mediante los entornos de desarrollo proveídos por el fabricante de FPGAs escogido. En ellos, se deberán incluir los ficheros generados por HDL Coder, juntarlos con otros ficheros que bien hayamos hecho de forma manual o con otros modelos exportados por HDL Coder y sintetizar el modelo en la FPGA escogido.

Una funcionalidad destacable, que se analizará en profundidad en el siguiente capítulo, es que HDL Coder permite convertir código desarrollado en lenguaje Matlab (lenguaje propio de la herramienta) a HDL. Con el lenguaje Matlab es relativamente sencillo trabajar con números en coma fija, por lo que a veces será más sencillo realizar un script en dicho lenguaje, incrustarlo dentro de un bloque en Simulink y luego exportarlo a HDL. La herramienta

también puede integrarse con Stateflow para la realización de máquinas de estado.

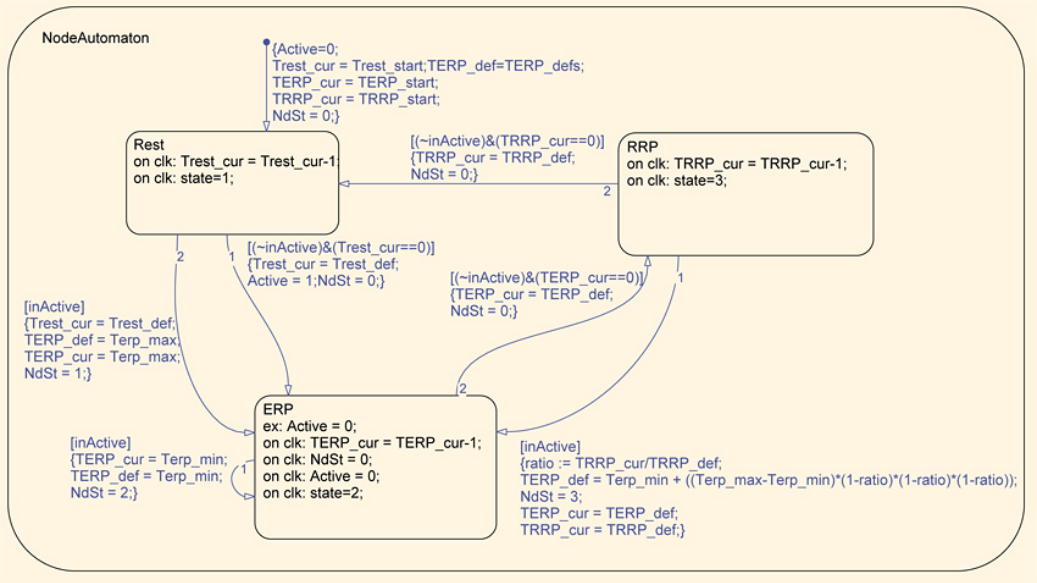

Figura 3.10: Máquina de estados modelada con Stateflow, también puede ser exportada mediante HDL Coder

3.3.1. Comparativa

En la tabla 3.1 puede verse resumida la comparativa de las tres herramientas analizadas en este capitulo, DSP Builder de Altera, System Generator de Xilinx y HDL Coder de MathWorks. Las primeras dos ofrecen prácticamente las mismas características, es en la tercera donde se pueden encontrar las mayores diferencias.

	DSP Builder	System Generator	HDL Coder
Compañía	Altera	Xilinx	MathWorks
Bloques disponibles			
Bloques básicos	Si	Si	Si
Bloques FFT	Si	Si	Si
Bloques Filtros	Si	Si	Si
Código Matlab	No	No	Si
Código HDL	Si	Si	No
Ejemplos	Si	Si	Si

Tabla 3.1: Comparativa entre herramientas de diseño

	DSP Builder	System Generator	HDL Coder
Documentación	Si	Extensa	Extensa
Bloques estándar	No	No	Si
Formatos de codificación numérica			
Enteros	Si	Si	Si
Coma fija	Si	Si	Si, ayudas
Coma flotante	Si	Si	No
Arquitecturas soportadas			
Altera	Si	No	Si
Xilinx	No	Si	Si
Otras	No	No	Si
Integración con herramientas			
Entorno propio	Si	Si	No
Generación testbench	Si	Si	Si
Generación HDL	No	No	Si

Tabla 3.1: Comparativa entre herramientas de diseño

A forma de resumen, DSP Builder y System Generator de Altera ofrecen muy buenas soluciones para sus propias FPGAs: generan código muy optimizado y se integran muy bien en el proceso de desarrollo, a costa de que solo sirven para sus propios dispositivos. Por contra, HDL Coder de MathWorks es una herramienta más universal, puesto que genera directamente código HDL que luego será implementado en dispositivos de cualquier fabricante, a costa de obtener un código menos optimizado y tener un proceso de desarrollo más tradicional.

Capítulo 4

Diseño realizado: estimador de frecuencia fundamental

Contents

Con el objetivo de comprobar la idoneidad de la herramienta HDL Coder de MathWorks para su aplicación en el procesado digital de señal, se ha decidido desarrollar un sistema que sea capaz de determinar la frecuencia fundamental de una señal digital. La parte que se ha desarrollado mediante Simulink para luego ser exportada mediante HDL Coder es única y exclusivamente la relacionado con el cálculo de la frecuencia fundamental.

Cabe recordar que el objetivo del presente proyecto no es la realización de un estimador de frecuencia fundamental, si no el analizar el uso de la herramienta HDL Coder de Simulink para su uso en aplicaciones de procesado digital de señal. Es por ello, que pese a estimar la frecuencia fundamental mediante una transformada de Fourier sin ser esta la mejor opción se ha optado por ella, puesto que es una herramienta fundamental y muy usada en el procesado digital de señal.

La placa de desarrollo usada es una Altera DE2, la cual ha sido previamente analizada en la sección 2.2.2. Está plataforma es idónea para el desarrollo de está aplicación, puesto que dispone de una FPGA Altera Cyclone II[6] (analizada en la sección 2.2.1) junto a diversos elementos necesarios, como son la entrada de audio junto a un CODEC WM8731 de Wolfson[7] para digitalizar la señal de entrada, una memoria SRAM con una capacidad de 256Kx16 donde almacenar temporalmente las muestras capturadas por el CODEC, y una serie de displays de 7 segmentos en los que se mostrará la frecuencia fundamental de la señal de entrada.

La realización de los módulos no relativos al procesado digital de señal no han sido realizados con Simulink. El módulo de control ha sido realizado mediante el proceso tradicional de desarrollo de hardware digital; en cambio, el módulo de control del CODEC es uno previamente desarrollado, el cual había sido previamente utilizado en las asignatura optativas de *Procesado Digital de Señal* o *Laboratorio de Diseño Digital* impartidas en **UPV-EHU** en la presente titulación.

En el presente capítulo se detalla el diseño de cada uno de los módulos que componen la aplicación, centrándose de forma más extensa en el realizado mediante Simulink.

4.1. Arquitectura general del sistema

El presente diseño estará compuesto por tres módulos, tal y como pude verse en la figura 4.1, cada uno de ellos encargado de diferentes cuestiones. En las siguientes secciones se analizarán dichos módulos en profundidad.

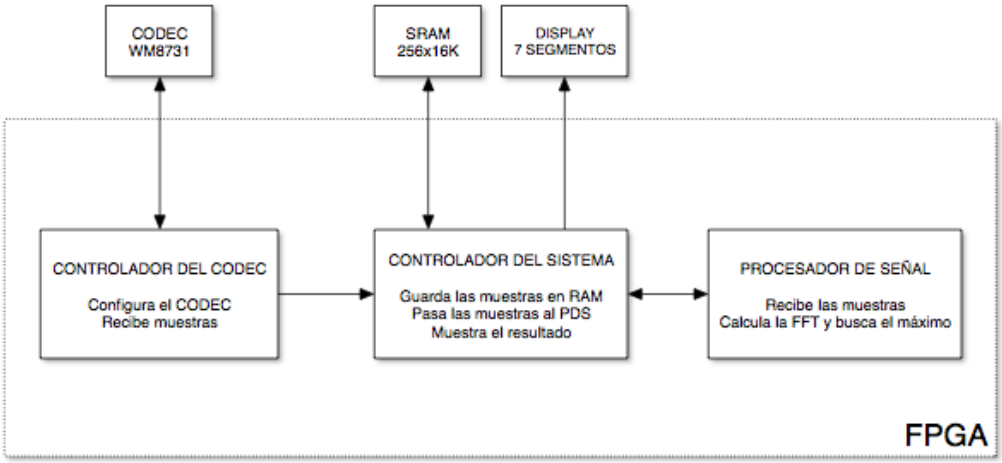

Figura 4.1: Arquitectura general del sistema

- Módulo del CODEC: es el módulo encargado de interactuar con el CODEC de audio presente en la tarjeta DE2 de Altera. Tienes dos funciones: configurar el CODEC y capturar las muestras que éste proporciona.

- Módulo de PDS: es el módulo encargado de determinar la frecuencia fundamental de la señal digitalizada. Para ello realizará una transformada de Fourier y buscará en el espectro la frecuencia fundamental.

- Módulo de control: es el módulo central de la aplicación. Se encarga de guardar en una memoria RAM las muestras digitalizadas por el CODEC, de pasar dichas muestras al módulo de PDS y finalmente de mostrar en un display de 7 segmentos la frecuencia fundamental de la señal.

4.1.1. Módulo del CODEC

La placa Altera DE2 dispone de un CODEC WM8731 de la empresa Wolfson. La configuración de dicho CODEC se realiza mediante el protocolo I^2C, mediante la cual se ha configurado el CODEC de la siguiente manera:

- Frecuencia de muestreo, fs = 8000 Hz (concretamente 50 MHz / 6144 = 8138,02 Hz).

- Resolución de cada muestra, 16 bits en complemento a dos.

- Transmisión de datos serie, mono, con formato justificado a la izquierda.

El bloque encargado de configurar el CODEC se llama `AU_SETUP`. Este bloque ha sido obtenido de asignaturas relacionadas con el presente proyecto de fin de carrera, por lo que no se entrará en detalle con su diseño. En la figura 4.2 puede verse el símbolo correspondiente a dicho módulo.

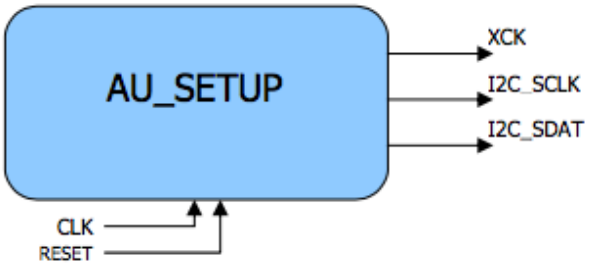

Figura 4.2: Símbolo del módulo `AU_SETUP`

Tal y como puede verse, su interfaz es relativamente sencilla. Además de las típicas señales de `reset` y `clock`, presentes en prácticamente todos los diseños de sistemas digitales, dispone de las tres señales necesarias para la comunicación I^2C. En cuanto se inicie el sistema, este módulo configurará el CODEC con los parámetros previamente establecidos y ahí acabará su función.

A partir de ese momento, el CODEC empezará a enviar mediante un protocolo serie las muestras que digitalice con los parámetros establecidos. El módulo encargado de capturar dichas muestras se llama `AU_IN`, y su símbolo puede verse en la figura 4.3. Este módulo ha sido previamente realizado en la asignatura de *Laboratorio de Sistemas Digitales* de la presente titulación.

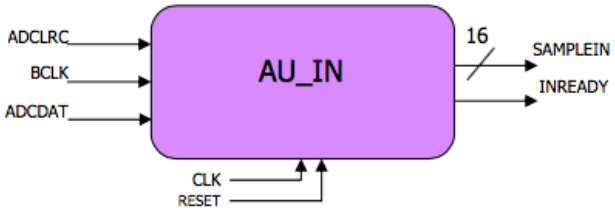

Figura 4.3: Símbolo del módulo `AU_IN`

Además de las típicas señales `clock` y `reset`, el módulo dispone de las siguientes entradas y salidas:

- `ADCLRC`: señal que indica el comienzo de la recepción de una nueva muestra por parte del CODEC. Un flanco ascendente representa una muestra del canal izquierdo y un flanco descencente representa una muestra del canal derecho.

- `BCLK`: señal que indica la mitad de cada uno de los bits enviados por el CODEC en su flanco ascendente.

- `ADCDAT`: señal por la que el CODEC va enviando los datos capturados.

- `SAMPLEIN`: salida que contendrá una muestra de 16 bits capturada por el CODEC.

- `INREADY`: salida que indicará que la salida SAMPLEIN contiene un dato válido.

El funcionamiento de dicho módulo, a *grosso modo* es el siguiente: teniendo en cuenta las señales `ADCLRC` y `BCLK`, el módulo va capturando uno a uno los bits de cada una de las muestras provenientes de `ADCDAT`. Cuando se han capturado las 16 muestras correspondientes al canal izquierdo, dicho dato está disponible en la salida `SAMPLEIN` y la activación de la señal `INREADY` indica que dicho dato puede ser leído.

Por lo tanto, con la utilización de estos dos módulos por un lado se configura el CODEC y por otro lado se obtienen los datos que envía para ofrecerlos de una forma más amigable a los demás subsistemas que forman la presente aplicación.

4.1.2. Módulo de PDS

El módulo de PDS (procesado digital de señal) es el encargado de determinar cual es la frecuencia fundamental de la señal de entrada. En la figura 4.4 puede verse cual es su interfaz de entradas y salidas.

Además de las típicas `clock` y `reset`, como entradas dispone de las siguientes señales:

- `din_re`: entrada de datos con la parte real de la muestra que se desea analizar.

- `din_im`: entrada de datos con la parte imaginaria de la muestra que se desea analizar. Como la señal a analizar es una señal de audio, dicha entrada será siempre 0.

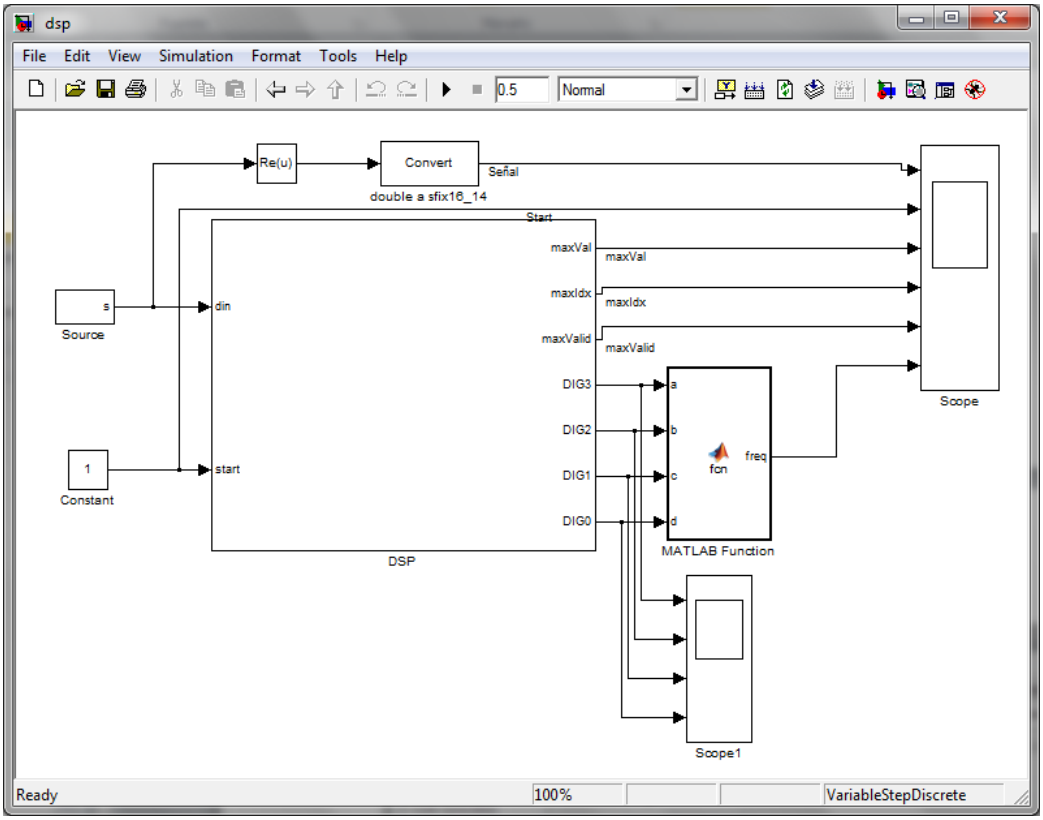

Figura 4.4: Símbolo del módulo de procesado digital de señal

- `start`: señal que indica que se desea empezar un nuevo cálculo. Tras pasar esta señal a 1, cada ciclo de reloj deberá haber una nueva muestra en la entrada `din_re`.

En cuanto a las salidas del módulo PDS, son la siguientes:

- `maxValid`: señal que indica que se ha terminado el cálculo y que hay datos válidos en las demás salidas.
- `maxVal`: valor máximo encontrado en la salida de la transformada.
- `maxIdx`: índice del valor máximo encontrado en la salida de la transformada.
- `DIG4, DIG3, DIG2, DIG1`: dígitos que representan la frecuencia fundamental de la señal en hercios.

Para determinar cual es la frecuencia fundamental de la señal de entrada, se ha decidido hacer este cálculo mediante la transformada de Fourier[8]. Esta herramienta es ampliamente conocida y utilizada en el ámbito del procesado digital de señal, y básicamente su tarea es convertir una señal del dominio del tiempo al dominio de la frecuencia. De esta forma, tras realizar la transformada de Fourier se obtendrá por cada frecuencia cual es su energía dentro de la señal analizada. Por lo tanto, para la presente aplicación, tras calcular la transformada de Fourier, se calcula cual es el punto máximo de dicha transformada y se saca como salida la frecuencia que representa dicho punto.

Este módulo ha sido desarrollado mediante un modelo de Simulink para luego ser exportado a HDL con la herramienta HDL Coder. En la figura 4.5 puede verse cual es su estructura general.

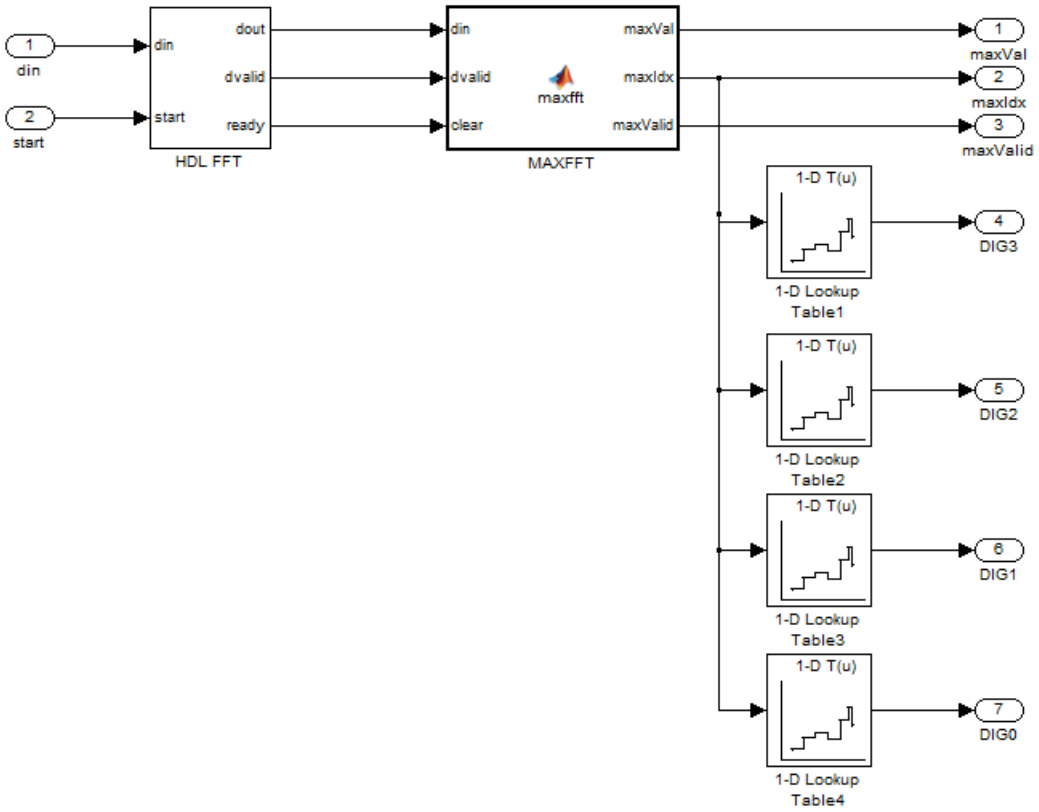

Figura 4.5: Estructura en base a bloques del módulo de procesado digital de señal

Tal y como puede observarse, el diseño está compuesto por varios bloques que se analizarán a continuación, uno para realizar la transformada de Fou-

rier, otro para calcular el valor y el índice máximo de dicha transformada y finalmente unas *Look-up Table* para obtener los dígitos correspondientes a la frecuencia calculada.

Módulo `HDL_FFT`

La primera decisión a la hora de desarrollar el presente módulo radica a la hora de escoger en cuantos puntos se va a calcular la transformada de Fourier, puesto que de ello dependerán una serie de parámetros, siendo el más importante de ellos la precisión de la salida.

Teniendo en cuenta que la frecuencia de muestreo es de 8138,02 Hz, y ajustando el número de puntos de la transformada de Fourier a 256, se obtendrá una precisión de 8138,02 Hz / 256 = 31,79 Hz. Dicha precisión no es muy alta, pero es más que suficiente para la realización del presente proyecto.

Dentro de la librería de bloques sintetizable que ofrece Simulink para HDL Coder, debido a que la transformada de Fourier es una operación muy frecuente en los sistemas de procesado de señal digital, puede encontrarse un bloque que realiza dicha operación denominado `HDL_FFT`. En la figura 4.6 puede verse su símbolo, con las entradas y salidas detalladas a continuación.

Figura 4.6: Símbolo del módulo HDL_FFT que ofrece Simulink

- `din`: señal de entrada, compleja.
- `start`: señal que indica que el bloque empezará a recibir entradas.
- `dout`: señal de salida, compleja.
- `dvalid`: señal que indica la presencia de datos válidos en la salida.
- `ready`: señal que indica que el proceso ha terminado y por tanto puede iniciarse un nuevo proceso.

El módulo usa el algoritmo Radix-2 DIT (*Decimation in Time*) y usa el modo *Burst I/O* para la entrada y salida de datos, es decir, cuando se le indique con la señal `start` que se le empiezan a pasa datos, el módulo espera que se le pase una nueva muestra cada ciclo de reloj. De la misma forma, cuando el módulo indique mediante la señal `dvalid` la presencia de datos validos en la salida, sacará un dato válido cada ciclo de reloj.

Son varios los aspectos del módulo que pueden configurarse, los cuales pueden verse en la figura 4.7, como por ejemplo el número de muestras a analizar (o lo que es lo mismo, el número de puntos en los que se va a calcular la transformada), el cual debe ser potencia de 2 por el algoritmo Radix-2 que emplea, o la precisión de las operaciones internas que realiza, puesto que el módulo trabaja con números en coma fija. Trabajar con números en punto flotante requiere de mucha potencia de cálculo (por ejemplo los microprocesadores cuentan con coprocesadores específicos para este tipo de operaciones), por lo tanto en las FPGAs se suele optar por hacer las operaciones en punto fijo, dado que de esta manera los cálculos se realizan igual que con números enteros, simplemente sabiendo en cada momento en que posición se encuentra la coma a la hora de interpretar u operar con dichos números.

Las muestras capturadas son de 16 bits, de los cuales el primero representa el signo, el siguiente representa la parte entera y los 14 restantes representan la parte fraccionaria, siendo así el rango representable de $[-2, 2)$. En cuanto al formato de la salida, después de hacer unos simples cálculos estos están en formato de 16 bits, de los cuales el primero representa el signo, los 8 siguiente representan la parte entera y los 7 finales representan la parte fraccionaria, pudiendo así representar valores del rango $[-256, 256)$.

Con esta información, la documentación del módulo proporciona cuantos ciclos se necesitarán para realizar el cálculo mediante la siguiente formula:

$$T_{cycle} = 3N/2 - 2 + log_2(N) * (N/2 + 3)$$

Teniendo en cuenta que N representa el número de puntos o muestras, y que en este caso es de 256, se obtiene que:

$$T_{cycle} = 1430 ciclos = 28,6 \mu s$$

Este dato será importante más adelante a la hora de desarrollar el módulo de control.

Módulo `MAXFFT`

El módulo del apartado anterior, `HDL_FFT`, será el encargado de calcular la transformado de Fourier de la señal de entrada, pero lo que interesa para la

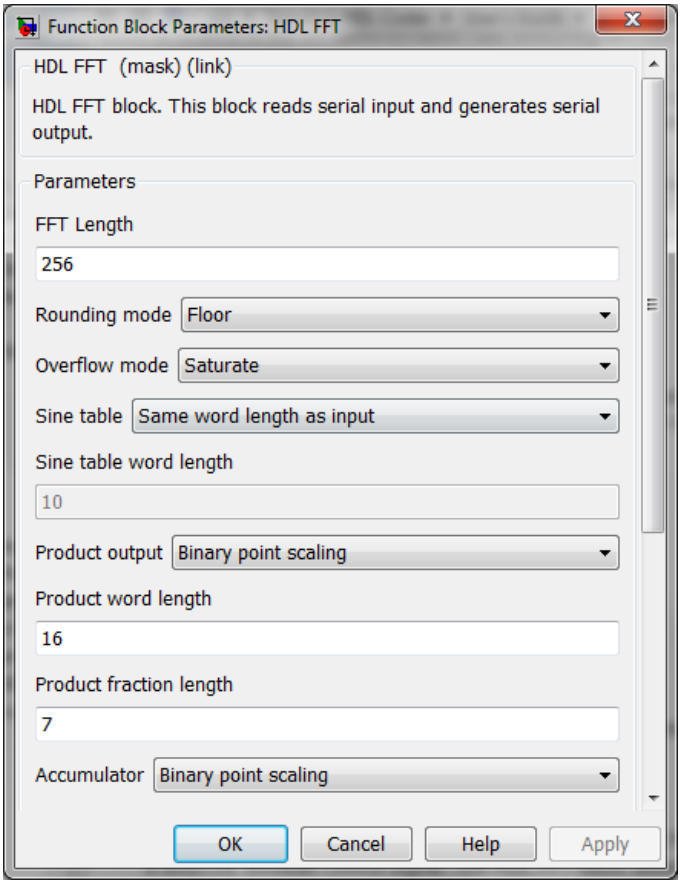

Figura 4.7: Parámetros de configuración del módulo HDL FFT

aplicación es encontrar el índice del valor máximo de la salida, puesto que este corresponderá a la frecuencia dominante de la señal de entrada. Por lo tanto, el objetivo del módulo `MAXFFT` será encontrar dicho valor máximo.

Antes de nada se ha de tener en cuenta el formato de salida del módulo `HDL FFT`. La salida de dicho módulo, al igual que la entrada, es un número complejo, compuesto por una parte real y una parte imaginaria. Lo que realmente interesa para calcular el máximo es el módulo de dicho número complejo, es decir:

$$|C_{i,j}| = \sqrt{C_i^2 + C_j^2}$$

Por lo tanto, el objetivo de dicho módulo es recibir una a una los datos de salida del módulo `HDL_FTT` según vayan estando listos, y calcular como salida cual es el valor máximo y su índice. En la figura 4.8 puede verse su interfaz de entrada y salida:

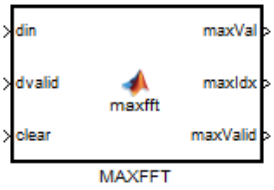

Figura 4.8: Símbolo del módulo MAXFFT

Tal y como puede verse, sus entradas serán las salidas del módulo `HDL_FFT` y sus salidas serán el valor máximo, el índice del valor máximo y una señal cuya activación determinará si en la salida hay datos validos.

Para diseñar este módulo se ha decidido usar el bloque `MATLAB Function` de Simulink, el cual permite crear un bloque de Simulink describiendo su contenido con el lenguaje de programación M de Matlab, tal y como puede verse en la figura 4.9.

El bloque está formado por una función principal, en la cual se definen las entradas y salidas del bloque junto al comportamiento interno, y opcionalmente por más funciones internas que serán llamadas desde la función principal.

Por defecto Matlab trabaja con números en coma flotante, pero para está aplicación se está trabajando con números en coma fija, por lo que para desarrollar este bloque se utilizan ciertas funciones que proporciona Matlab y que facilitan el desarrollo, sobre todo si se compara a la forma de trabajar con números en coma fija de VHDL, el cual es un proceso completamente manual y engorroso. Simplemente se ha de definir una función en la cual se define el tipo de datos a usar, siendo en este caso el analizado en la sección anterior (coma fija, 1 bit signo, 8 bits parte entera, 7 bits parte fraccionaria), y como se realizan las operaciones entre los datos (tipo de redondeo, que hacer en caso de overflow/underflow, precisión interna de la suma, precisión interna de la multiplicación, etc.). Con esta función, cada vez que se vaya a procesar un dato numérico en dicho formato, se obtendrá su representación en coma fija para poder usarlo sin ningún tipo de problema.

El funcionamiento del módulo será el siguiente, y se ejecutará en cada flanco de reloj:

- Si la señal `clear` está activada, se resetean las variables internas del módulo para iniciar un nuevo cálculo.
- Si la señal `dvalid` está activada y si el contador interno no ha llegado hasta el valor 128 ($N/2$), se calcula el valor absoluto de la entrada y se

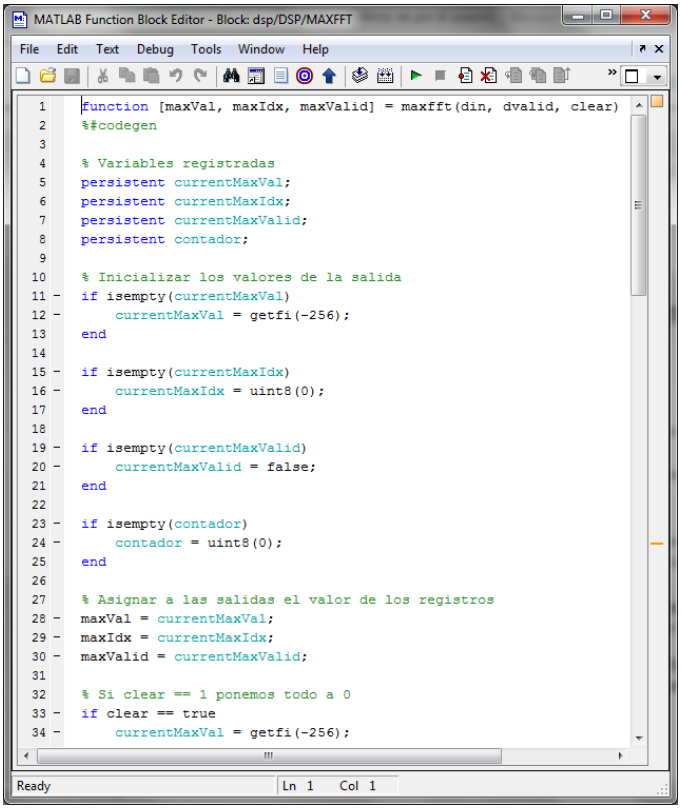

Figura 4.9: Editor de código de Matlab con la función que implementa el módulo MAXFFT

compara con el máximo encontrado hasta ese momento. En caso de que sea mayor, se actualiza el valor máximo y el índice que le corresponde. En cualquiera de los dos caso se incrementa el contador.

- Si el valor del contador es mayor de 128, se activa la señal `maxValid` para indicar que el proceso ha terminado.

Hay que puntualizar que solo se analizan los primeros 128 puntos de entrada, porque al ser la señal analiza una señal con solo parte real (señal de audio), el valor absoluto de la salida de la transformada de Fourier será simétrico.

El mayor problema a la hora de implementar este módulo reside en el cálculo del valor absoluto de la señal compleja de salida. En las siguientes secciones se analizan las opciones que se han tenido en cuenta para el desarrollo del mismo.

Función `ABS` La primera opción planteada es hacer uso de la función `abs` de Matlab. Dicha función devuelve el módulo de un número complejo, ya sea éste en coma flotante o coma fija.

En simulación funciona perfectamente, pero el problema viene a la hora de sintetizar. HDL Coder devuelve un error diciendo que la función `abs` no es sintetizable para números complejos, por lo tanto está opción es desechada.

```
HDL code generation for complex data type is not
supported for abs function.
```

Teorema de Pitagoras Otra opción es calcular manualmente el valor absoluto del número complejo, teniendo el teorema de Pitagoras:

$$|C_{i,j}| = \sqrt{C_i^2 + C_j^2}$$

Tras implementarlo dentro del bloque de Matlab, nuevamente funciona bien en la simulación, pero a la hora de exportarlo a HDL se observa que hace uso de mucho hardware para realizar la operación de la raiz cuadrada y que encima en su implementación se generan *latches*, lo cual acarreará problemas en su implementación final sobre la plataforma hardware. Por lo tanto, está opción también queda desechada.

Aproximación `CORDIC` CORDIC[9] (*COordinate Rotation DIgital Computer*) es un método matemático muy utilizado en la electrónica digital que sirve para calcular funciones trigonométricas.

Su funcionamiento se fundamenta en la rotación de los números complejos. Por ejemplo, para el uso en esta aplicación, el cálculo del valor absoluto de un número imaginario, lo que hace el algoritmo es ir aplicando rotaciones hasta que la parte imaginaria del número complejo es prácticamente 0. Una vez realizado esto, la parte real del número complejo equivaldrá a su magnitud. Las rotaciones que se hacen en el algoritmo CORDIC son muy amigables para sistemas digitales, puesto que trabaja siempre con números que son potencias de 2, por lo que es un método rápido y eficaz.

Tras implementarlo dentro de la función de Matlab se obtienen buenos resultados tras la simulación desde el propio Simulink, pero al exportarlo a HDL aparecen los mismos problemas que en el caso anterior: se generan *latches* a la hora de implementarlo en la FPGA. Esto es debido a que el algoritmo es iterativo, por lo que al querer hacer varias iteraciones del algoritmo en un mismo ciclo de reloj aparecen dichos *latches*, que a la hora de la puesta en marcha del sistema harán que su funcionamiento sea impredecible. Una posible solución para esto será hacer el cálculo mediante un *pipeline*, pero esto haría que la función se volviera realmente compleja.

Aproximación `Alpha-Beta` La aproximación `Alpha-Beta`[10] es una aproximación para el cálculo de la magnitud de un número complejo. Dicha aproximación no permite con exactitud conocer cual es el valor absoludo de la entrada, pero si que sirve para determinar cual es el valor absoluto máximo. Ésta es la aproximación:

$$Magnitud(C_{i,j}) = alpha * max(C_i, C_j) + beta * min(C_i, C_j)$$

Se han de determinar los parámetros `alpha` y `beta` de la aproximación, puesto que dependiendo de cuales sean los escogidos la aproximación obtendrá mejores o peores resultados. Para esta aplicación se ha decidido usar `alpha` = 1 y `beta` = 0.5, puesto que de esta forma el cálculo es mucho más sencillo, teniendo que calcular simplemente el máximo y el mínimo dividiendo este último por 2, es decir, desplazando la coma hacia la izquierda una vez. Los resultados obtenidos son muy bueno tal y como puede observarse en la figura 4.10, siendo el error máximo entre el valor absoluto real y el aproximado de 0,5. Dicho error no afecta mucho en esta aplicación, puesto que en la salida de la transformada de Fourier habrá un valor mucho más dominante que los demás.

Por lo tanto, tras obtener buenos valores en la experimentación ésta es la forma en la que la aplicación determinará el valor absoluto de los datos de entrada, puesto que es una operación sencilla y que se puede implementar en un solo ciclo de reloj.

Representación de dígitos

Por último, además de calcular la transformada de Fourier y encontrar el índice en el que se encuentra el máximo del valor absoluto, se ha de obtener a que frecuencia corresponde dicho índice mediante la siguiente formula:

$$f_{fundamental} = \frac{indice * fs}{256}$$

Este cálculo puede ser realizado cada vez que se desee obtener el índice, o pueden almacenarse los 128 valores posibles (el valor del índice va de 0 a 127) en una tabla y así obtener directamente la frecuencia de la señal. La primera opción requiere más hardware puesto que implementar una división no es algo trivial en una FPGA, por lo que al no necesitar una gran cantidad de espacio para almacenar la información de los 128 índices se opta por esta segunda opción.

Hay varias formas de implementar esta opción. Por un lado se puede almacenar el valor de la frecuencia de corresponiente a cada índice, y luego

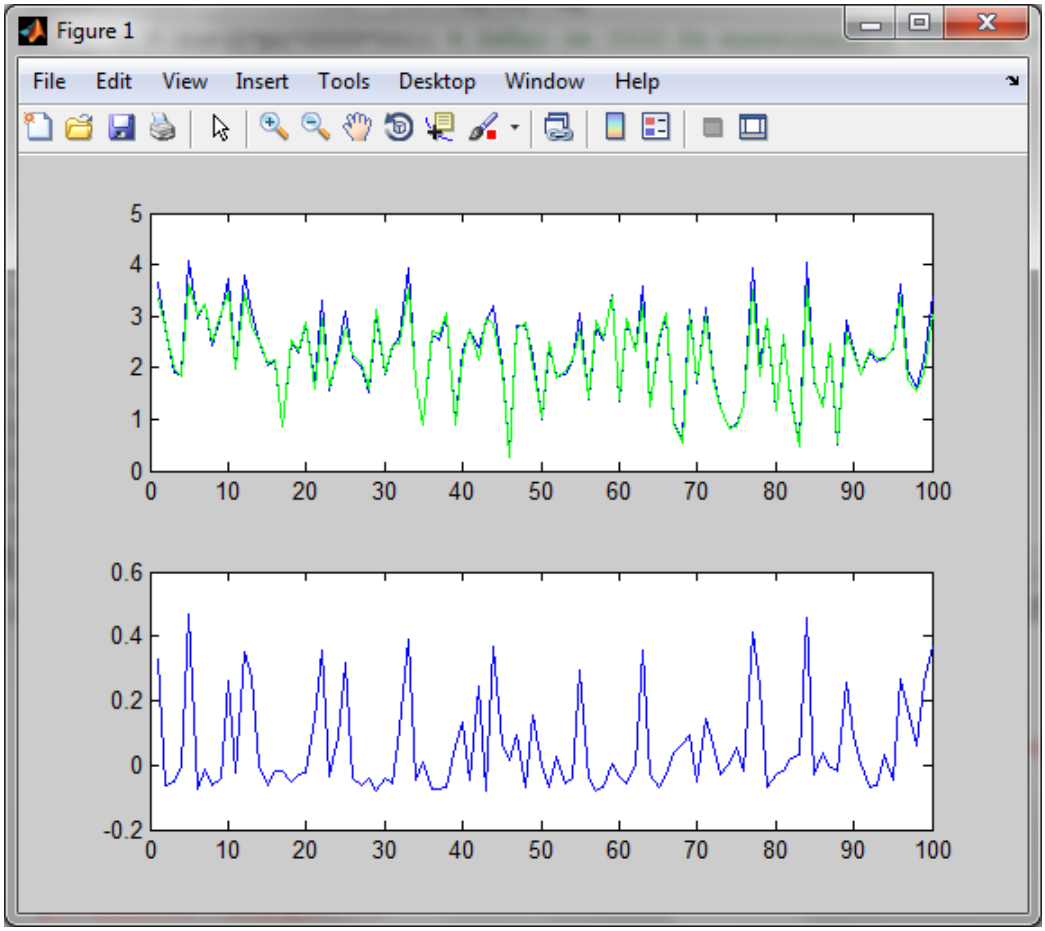

Figura 4.10: Error de la aproximación Alpha-Beta respecto al cálculo real

obtener los dígitos que la componen para mostrarla en los displays de 7 segmentos; por otro lado, se pueden tener 4 tablas y que cada una guarde cada uno de los digitos que componen la frecuencia.

Se han probado las tres opciones explicadas anteriormente, y finalmente analizando los recursos de hardware que se emplean en cada una de ellas y la frecuencia máxima a la que permiten trabajar, se ha optado por la última opción, teniendo en cuenta que solo han de almacenarse 128 valores. En caso de que la FFT se decida hacer en más puntos y por lo tanto haga falta almacenar más valores, puede que esta solución no sea la más idónea.

Para la implementación en el modelo, se han usado 4 bloques *Look-up Table*, siendo la entrada de ellos un valor entre 0 y 127, y la salida un número entero del 0 al 10 que representa cada uno de los dígitos relativos a la frecuen-

cia que representa el índice de entrada. El contenido de las *Look-up Table* se ha generado mediante un script de Matlab.

4.1.3. Módulo de control

El módulo de control es el módulo central de la aplicación, el cual es un intermediario entre los otros dos módulos (módulo del CODEC y módulo del procesado digital de señal). Además, también interactúa con hardware externo a la FPGA como puede ser la memoria SRAM y los displays de 7 segmentos.

Las tareas a realizar por dicho módulo son las siguientes:

- Captura de 256 muestras del módulo `AU_IN` y almacenamiento en la memoria RAM.
- Activar la señal `start` y poner en `din` el primer dato capturado que está almacenado en la memoria RAM.
- Ir cambiando el dato de `din` con nuevas muestras, hasta enviar 256 muestras.
- Esperar a que termine el cálculo del módulo PDS, obtener su salida y mostrarla en el display de 7 segmentos.

El tiempo entre muestra y muestra del módulo `AU_IN` es de 122,9 μs y para procesar las 256 muestras se requieren algo menos de 30 μs, por lo tanto, cada vez que se reciban 256 muestras del CODEC se procederá a calcular la frecuencia fundamental.

En la figura 4.11 puede verse el símbolo del presente módulo.

Este módulo ha sido desarrollado con la metodología típica aprendida en asignaturas como *Laboratorio de Sistemas Digitales*, mediante la cual se divide el diseño en dos partes, una unidad de control (UC) compuesta por una máquina de estados y una unidad de proceso (UP) compuesta por hardware como pueden ser contadores, registros, etc. Dicho diseño se traduce a VHDL y se simula mediante un *testbench* para validar su correcto funcionamiento.

Unidad de control

En la figura 4.12 se encuentra la máquina de estados que compone la unidad de control.

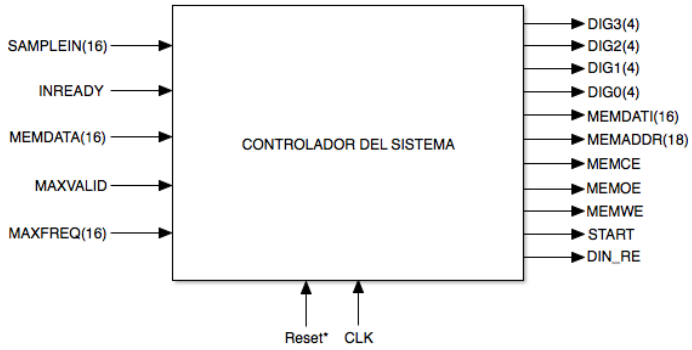

Figura 4.11: Símbolo del módulo de control del sistema

Unidad de proceso

En la figura 4.13 se encuentran los módulos que componen la unidad de proceso. Básicamente es un contador para ir contando las muestras recibidas y enviadas y tres registros: uno para guardar la muestra convertida por el CODEC, otro para guardar la muestra proveniente de la memoria RAM y el último para guardar la frecuencia calculada por el módulo de procesado de señal.

4.2. Flujo de trabajo

Tras analizar en las secciones anteriores el diseño de los diferentes elementos que conforman esta aplicación, en la presente sección se pretende explicar cual ha sido el proceso mediante el cual se ha llegado a los resultados finales.

La primera parte del proyecto ha sido definir que es lo que se ha implementado y su arquitectura general. Como se ha visto anteriormente, se ha decidido dividir la aplicación en tres partes independientes, siendo la implementación de cada una totalmente diferente.

Para el módulo del CODEC, compuesto por los módulos `AU_SETUP` y `AU_IN`, se ha optado por reutilizar unos previamente creados. Dichos módulos ya habian sido validados previamente, por lo que se han podido utilizar directamente sin realizar ningún cambio. Ésta es una de las ventajas de dividir el diseño de una aplicación grande en entidades más pequeñas, que pueden ser reutilizadas más adelante para otros fines.

El módulo de PDS es el que más trabajo ha llevado, puesto que es con el que se ha probado la herramienta HDL Coder de MathWorks. Lo primero

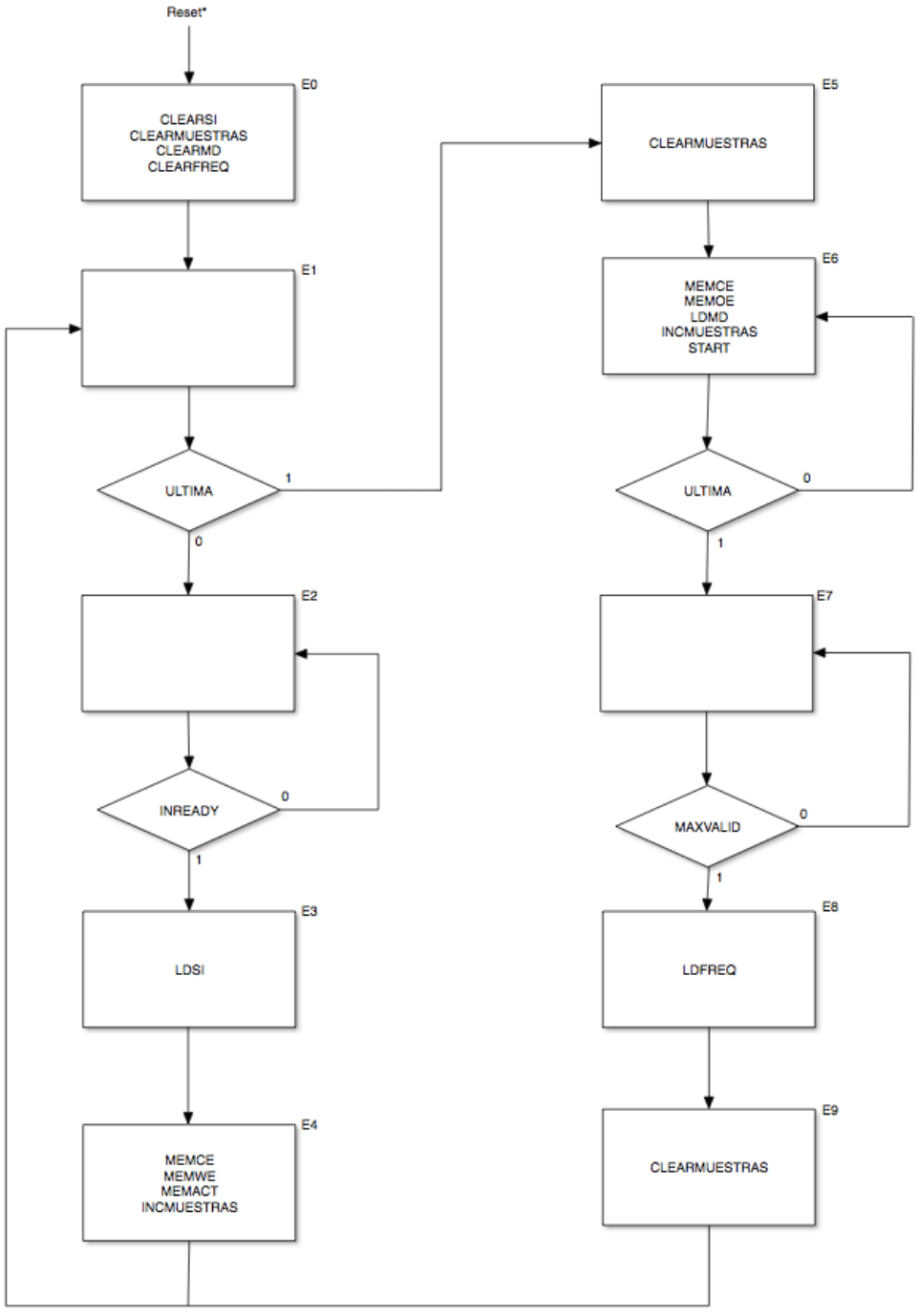

Figura 4.12: Unidad de control del módulo de control del sistema

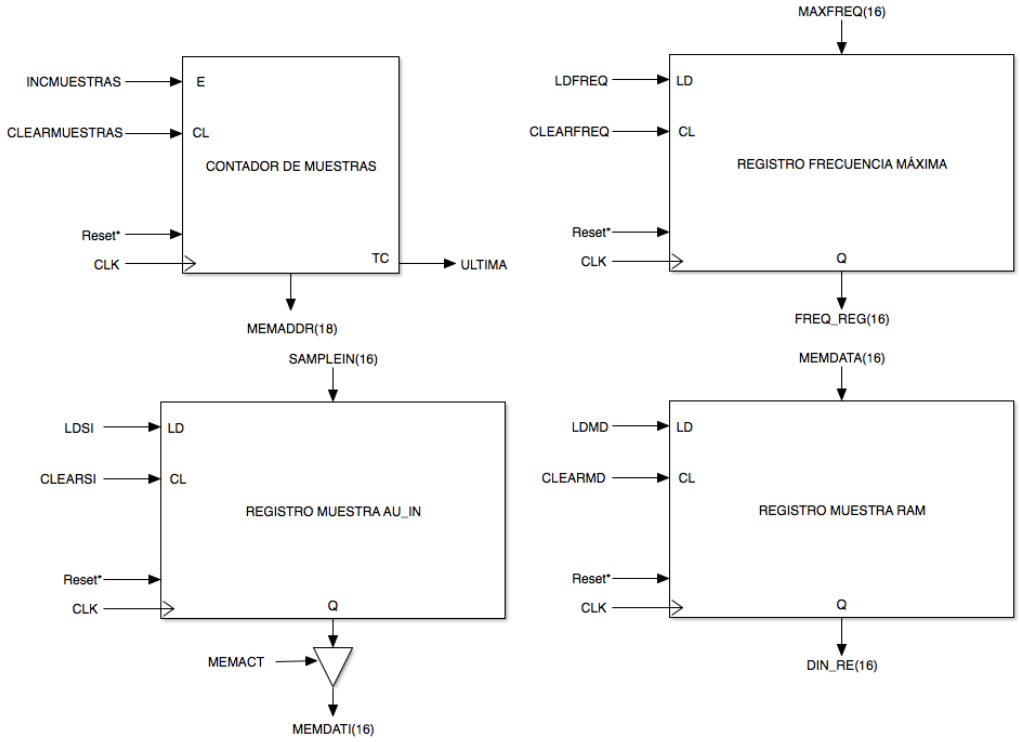

Figura 4.13: Unidad de proceso del módulo de control del sistema

que se ha tenido que hacer es aprender a manejar la herramienta, comprender sus capacidades y limitaciones, y su modo de empleo. Una vez adquiridas las bases se ha ido implementando el módulo, muchas veces con una metodología de ensayo-error. Gracias a esto, durante la implementación se ha ido conociendo en más profundidad el funcionamiento de la herramienta, por lo que poco a poco se han ido corrigiendo errores realizados en los primeros pasos.

El flujo de trabajo a la hora de diseñar el módulo PDS ha sido el siguiente:

- Crear un modelo de Simulink desde Matlab, y con la librería de módulos sintetizables crear un submodelo que es el que ha de exportarse a VHDL.

- Acompañar al submodelo de generadores y visualizadores para comprobar que el submodelo a sintetizar se comporta correctamente desde el punto de vista funcional.

- Comprobar que el submodelo puede ser exportado, exportarlo y generar sus testbench correspondiente, mediante los comandos `checkhdl`, `makehdl` y `makehdltb` respectivamente.
- Desde ModelSim crear un nuevo proyecto para realizar el análisis funcional del submodelo exportado. De esta forma se comprobará que el VHDL generado es funcionalmente equivalente al submodelo de Simulink. HDL Coder genera una serie de scripts para facilitar este proceso de simulación.
- Crear un proyecto en Quartus II de Altera, en el que se incluyen los ficheros VHDL creados por HDL Coder y sintetizarlo. Tras este proceso se ha de comprobar que el uso de hardware y la frecuencia máxima se encuentran dentro de los parámetros establecidos.
- Desde ModelSim crear un nuevo proyecto para realizar el análisis temporal del proyecto, añadiendo para ellos los ficheros .vho que se han generado en el paso anterior. Esta simulación será mucho más costosa, puesto que se está simulando el comportamiento exacto que tendrá la FPGA, pero asegurará que el diseño exportado es correcto.

Finalmente, para el módulo de control se ha seguido el típico proceso de desarrollo de este tipo de proyectos de hardware digital. Tras definir los requisitos, se ha dividido el diseño en una unidad de control y en una unidad de proceso que implementa dichos requisitos. También se ha definido un testbench para poder comprobar que el diseño implementado es correcto.

Una vez los tres módulos que componen la aplicación han sido desarrollados, se ha de crear un proyecto en Quartus II de Altera en el que se instancien y conecten todos los componentes entre ellos. Además de ello, se han de enrutar las señales de entrada y salida hacia las patas correspondientes al hardware que se quiera usar.

Tras esto, se ha de sintetizar el proyecto final, comprobar que no hay ningún error y programar la placa DE2 de Altera mediante el cable USB Blaster. La información respecto al uso de hardware de la implementación final del módulo de procesado digital de señal puede encontrarse en la figura 4.15 a continuación.

Como cada uno de los módulos había sido verificado por separado, para probar que la solución final funciona se ha probado directamente en la plataforma. Mediante un generador de tonos se han ido metiendo a la entrada de audio de la tarjeta señales de diferentes frecuencias puras, y se ha comprobado que el valor resultante en el display de 7 segmentos es el correcto. Una vez validado esto, se ha probado con señales con más de una frecuencia, siendo una de ellas la dominante, obteniendo también unos buenos resultados.

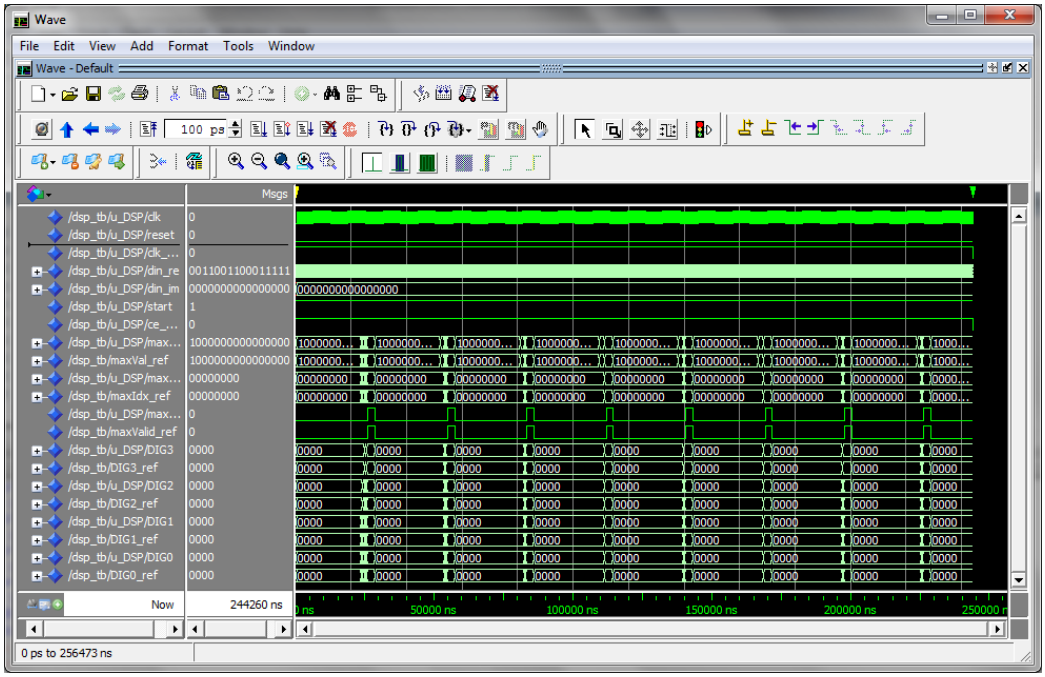

Figura 4.14: Simulación del módulo de procesado digital de señal

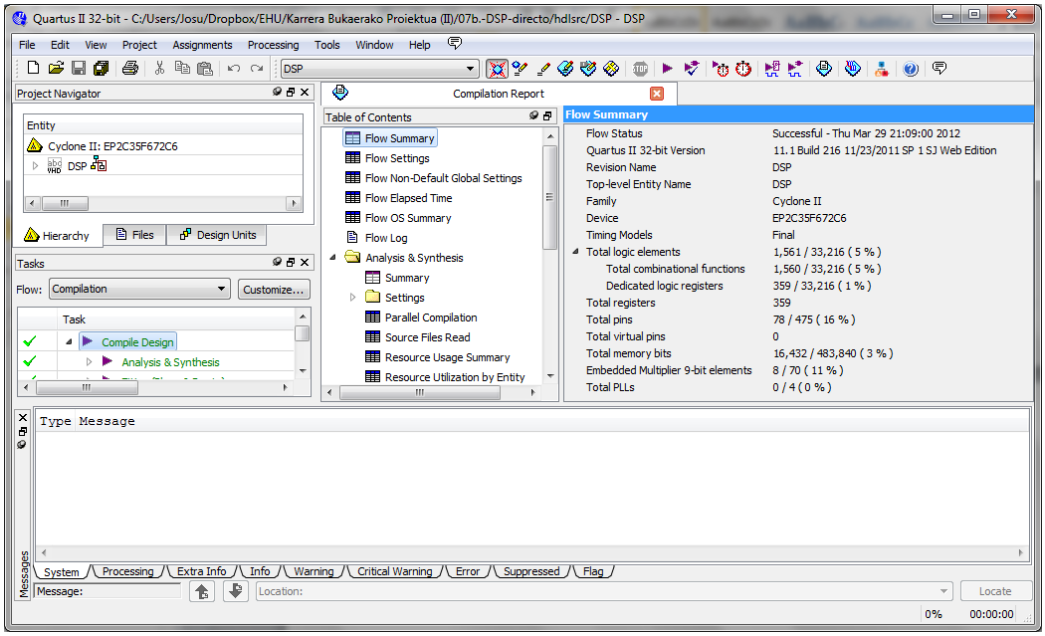

Figura 4.15: Uso de hardware del módulo de procesado digital de señal

4.3. Resultados

Tras diseñar, desarrollar y probar cada uno de los módulos que componen la aplicación a desarrollar, se han juntado todos dentro de un proyecto de Quartus II y se ha sintetizado la solución final, la cual puede verse en al figura 4.16.

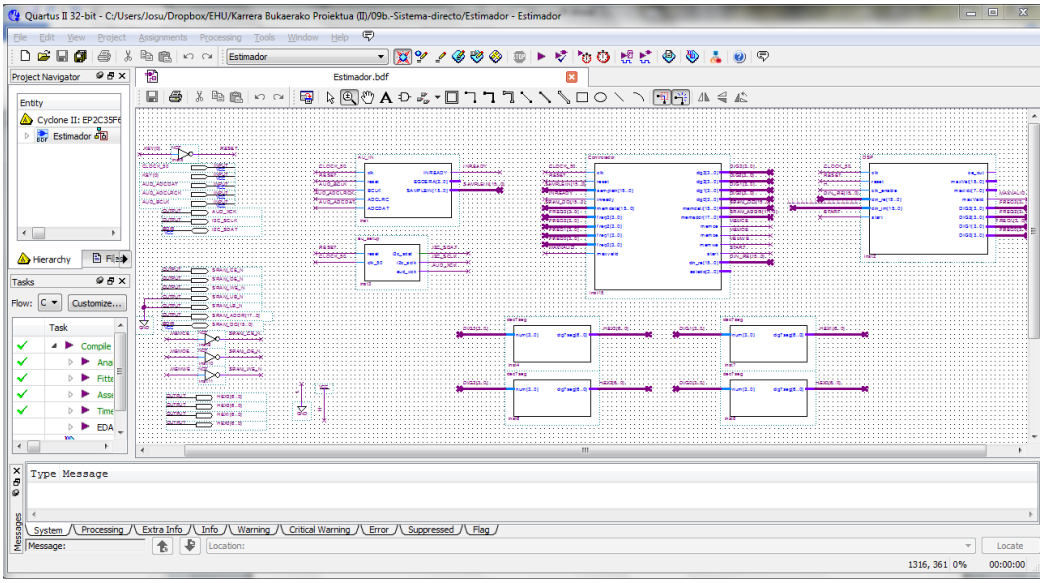

Figura 4.16: Sistema completo en Quartus II de Altera

Una vez programado el *bitstream* resultante de la síntesis del proyecto, tras grabarlo en la FPGA de la placa DE2, se ha procedido a realizar una serie de pruebas para comprobar si el funcionamiento real corresponde con el esperado. Estas son las pruebas realizadas:

- Señales sinusoidales puras de diferentes frecuencias.
- Señal *chirp* en el rango 1-4000 Hz.
- Combinación de dos señales sinusoidales puras.
- Señales triangulares y de diente de sierra de diferentes frecuencias.
- Sonidos reales capturadas con un micrófono provenientes de un instrumento.

Tras la realización de todas las pruebas, se ha verificado que el funcionamiento del sistema es el esperado, puesto que todas ellas han arrojado un resultado satisfactorio.

Por último, en la figura 4.17 puede observarse la aplicación funcionando en la placa DE2, siendo este el entorno usado para las pruebas.

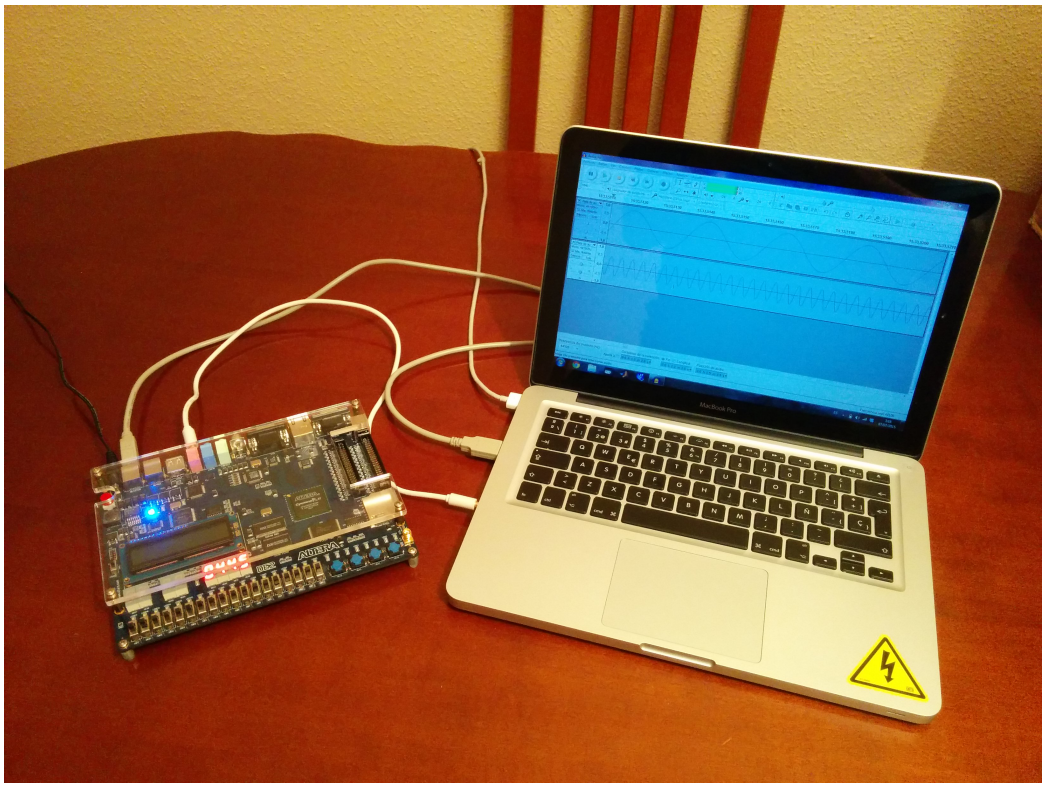

Figura 4.17: Entorno de pruebas de la aplicación desarrollada

Capítulo 5

Conclusiones

Contents

Tras la realización del presente proyecto, son varias las conclusiones que merece la pena enumerar en cuanto a diversos aspectos.

5.1. Aplicación realizada

En cuanto a la aplicación realizada, se ha visto como desarrollar una aplicación de procesado digital de señal para una FPGA. Las conclusiones sacadas al respecto son las siguientes:

- El dividir la aplicación en módulos más pequeños e independientes ofrecen una serie de ventajas, como son la reutilización de módulos en diferentes proyectos, la validación más sencilla de cada una de las partes o la posibilidad de crear cada una de las partes mediante diferentes metodologías.

- Aplicar la transformada de Fourier y buscar el punto más alto es un mecanismo que da buenos resultados para determinar la frecuencia fundamental de una señal.

- Las limitaciones en ciertos cálculos dentro de las FPGAs hacen que por ejemplo calcular una raíz cuadrada o una simple división sean tareas que se han de evitar. Es por ello que existen aproximaciones para realizar cálculos complejos como CORDIC o la aproximación Alpha-Beta, idóneas para este tipo de aplicaciones.

- Debido a la escasez de recursos, optar por números en coma fija es una solución viable pese a ser menos flexible que los números en coma flotante. Es por ello que se ha de analizar con precisión cuales son las necesidades de la aplicación para determinar el formato de los números con los que se va a trabajar.

5.2. Simulink y HDL Coder

Teniendo en cuenta que la realización del módulo de procesado digital de señal ha sido realizado con estas herramientas, estas son las conclusiones sacadas acerca de dichas herramientas:

- Simulink es una muy buena herramienta cuando se quiera trabajar en aplicaciones de procesado digital de señal, debido a que está pensada para diseños de tipo *data-flow*.

- Pese a que el diseño pueda realizarse con los módulos disponibles en la biblioteca estándar de Simulink, no todos ellos son sintetizables por HDL Coder, incluso los que son sintetizables pueden tener diversas limitaciones.

- El código VHDL que se genera al exportar mediante HDL Coder, es un código estándar que puede ser implementado en cualquier FPGA del mercado. No está tan optimizado como si se hubiera hecho con DSP Builder de Altera o con System Generator de Xilinx, pero en caso de no estar atados a una compañia en concreto, la solución de MathWorks es la ideal.

- Simulink, y en concreto el uso del bloque que permite embeber código M de Matlab permite manejar de una forma increiblemente sencilla con números en coma fija, una tarea de lo más engorrosa en caso de tener que hacerse mediante VHDL.

- El hecho de trabajar a un nivel de abstracción superior, ofrece un desarrollo más cómodo. Por ejemplo, solventar un error de diseño es tan simple como cambiar el modelo, validarlo y exportarlo a HDL. Un fallo de la misma magnitud trabajando con VHDL lleva un esfuerzo mayor a la hora de solventarlo.

5.3. Conclusiones generales

Tras la satisfactoria conclusión del presente proyecto, en el cual se han cumplido los objetivos planteados al principio, estas son las conclusiones generales que se pueden arrojar:

- Las plataformas programables como las FPGAs son idóneas para tareas de procesado digital de señal debido a su flexibilidad, y pese a sus limitaciones como pueden ser la ausencia de números en coma flotante.

- Herramientas como Simulink y HDL Coder facilitan enormemente el desarrollo de aplicaciones de procesado digital de señal, gracias a la generación automática de código a partir de un modelo y a la facilidad que presentan a la hora de trabajar con números en coma fija.

5.4. Tareas futuras

Una vez terminado el proyecto, y visto el grado de satisfacción con la herramienta, son varias las tareas futuras que podrían realizarse sobre esta base, o que convendría explorar para adentrarse más en la herramienta.

- Simulink ofrece la posibilidad de realizar la simulación con ModelSim (co-simulación), e incluso ofrece la opción de ejecutar el modelo diseñado directamente en la FPGA mediante el uso deHardware-in-the-Loop (HIL), aunque en este caso no se ha podido probar debido a la incompatibilidad con la versión de ModelSim usada y con la placa de desarrollo usada. Pudiendo hacer uso de estas características, el flujo de trabajo es mucho más simple dado que practicamente todo el trabajo se realiza desde la plataforma Simulink. Estaría bien contar con una placa de desarrollo compatible y con versiones de ModelSim también compatibles para poder analizar estas características.

- Creación de una aplicación más compleja que haga uso del estimador de frecuencia fundamental, como podría ser por ejemplo un afinador de guitarras.

- Creación de un periférico con el analizador de frecuencia fundamental para poder conectarlo a un procesador NIOS II mediante el bus de comunicaciones AVALON. De esta forma, el periférico actuaría como un acelerador hardware de dicho cálculo.

- Analizar la compatibilidad de la herramienta HDL Coder con las herramientas DSP Builder de Altera y System Generator de Xilinx que se anuncian en las versiones más recientes de ambas.

Apéndice I

Código del módulo de control

A continuación se adjunta como anexo el código VHDL que implementa el diseño del módulo de control de la aplicación, detallado en la sección 4.1.3. Dicho código implementa tanto la unidad de control como la unidad de proceso.

Controlador.vhd

```
1  ------------------------------
2  -- Controlador.vhd
3  ------------------------------
4  -- Modulo Controlador
5  --
6  --  Es el modulo intermediario entre la interfaz del CODEC,
7  --  el procesador de senal (DSP), la memoria RAM externa y
8  --  los 4 displays de 7 segmentos.
9  --
10 --  Su funcionamiento es el siguiente:
11 --    - Captura 256 muestras provenientes del CODEC y las
12 --      almacena en la memoria RAM externa.
13 --    - Pasa las 256 muestras al modulo DSP una tras otra
14 --      en cada ciclo de reloj.
15 --    - Espera a que el modulo DSP acabe de realizar el
16 --      calculo y guarda la frecuencia calculada para poder
17 --      mostrarla en el display.
18 --
19 ------------------------------------------------------------
20 -- Autor:
21 --    Josu Lopez Fernandez
22 ------------------------------------------------------------
23 -- Fecha:
24 --      30/03/2012
25 ------------------------------------------------------------
26
27 LIBRARY ieee;
28 USE ieee.std_logic_1164.ALL;
29 USE ieee.numeric_std.ALL;
30
31 ENTITY Controlador IS
32 port(
33   -- Reloj interno de todo el sistema
34   clk : in std_logic;
35   -- Reset interno de todo el sistema
```

```
36    reset : in std_logic;
37
38    -- Entrada de muestras
39    samplein : in std_logic_vector (15 downto 0);
40    -- Dato valido en samplein
41    inready : in std_logic;
42    -- Salida de datos de la memoria RAM
43    memdata : in std_logic_vector (15 downto 0);
44    -- Frecuencia maxima calculada por el modulo DSP
45    freq3, freq2, freq1, freq0 : in std_logic_vector (3 downto 0);
46    -- Dato valido en maxfreq
47    maxvalid : in std_logic;
48
49    -- Digitos a mostrar en el display
50    dig3, dig2, dig1, dig0 : out std_logic_vector (3 downto 0);
51    -- Entrada de datos de la memoria RAM
52    memdati : out std_logic_vector (15 downto 0);
53    -- Direccion de la memoria RAM
54    memaddr : out unsigned (17 downto 0);
55    -- Senales de control de la memoria RAM
56    memce, memoe, memwe : out std_logic;
57    -- Senal de comienzo para el modulo DSP
58    start : out std_logic;
59    -- Entrada de datos del modulo DSP
60    din_re : out std_logic_vector (15 downto 0);
61
62    -- Estado de la maquina de estados
63    estado : out integer range 0 to 9
64 );
65 END Controlador;
66
67
68 ARCHITECTURE SYN OF Controlador IS
69    -- Senales para el estado presente y el estado siguiente
70    signal EP, ES : integer range 0 to 9;
71
72    -- Senales generadas por la UP
73    signal ultima : std_logic;
74
75    -- Senales enviadas a la UP
76    signal memact : std_logic;
```

```
77    signal clearsi, ldsi : std_logic;
78    signal clearmd, ldmd : std_logic;
79    signal clearfreq, ldfreq : std_logic;
80    signal clearmuestras, incmuestras : std_logic;
81
82    -- Registros internos
83    signal memaddr_reg : unsigned (17 downto 0);
84    signal si_reg : std_logic_vector (15 downto 0);
85
86  BEGIN
87    -- UNIDAD DE CONTROL
88    -- Proceso que calcula el estado siguiente
89    process (EP, ultima, inready, maxvalid)
90    begin
91      case EP is
92        when 0 => ES <= 1;
93        when 1 => if (ultima = '0') then
94                    ES <= 2;
95                  else
96                    ES <= 5;
97                  end if;
98        when 2 => if (inready = '0') then
99                    ES <= 2;
100                 else
101                   ES <= 3;
102                 end if;
103       when 3 => ES <= 4;
104       when 4 => ES <= 1;
105       when 5 => ES <= 6;
106       when 6 => if (ultima = '0') then
107                   ES <= 6;
108                 else
109                   ES <= 7;
110                 end if;
111       when 7 => if (maxvalid = '0') then
112                   ES <= 7;
113                 else
114                   ES <= 8;
115                 end if;
116       when 8 => ES <= 9;
117       when 9 => ES <= 1;
```

```
118     end case;
119   end process;
120 
121   -- Proceso que registra el estado de la UC
122   process (clk, reset)
123   begin
124     if (reset = '1') then
125       EP <= 0;
126     elsif (clk'event and clk = '1') then
127       EP <= ES;
128     end if;
129   end process;
130   estado <= EP;
131 
132   -- Definicion de las senales de control
133   clearsi <= '1' when (EP=0) else '0';
134   clearmd <= '1' when (EP=0) else '0';
135   clearfreq <= '1' when (EP=0) else '0';
136   clearmuestras <= '1' when (EP=0 or EP=5 or EP=9) else '0';
137   ldsi <= '1' when (EP=3) else '0';
138   memce <= '1' when (EP=4 or EP=6) else '0';
139   memwe <= '1' when (EP=4) else '0';
140   memact <= '1' when (EP=4) else '0';
141   incmuestras <= '1' when (EP=4 or EP=6) else '0';
142   memoe <= '1' when (EP=6) else '0';
143   ldmd <= '1' when (EP=6) else '0';
144   start <= '1' when (EP=6) else '0';
145   ldfreq <= '1' when (EP=8) else '0';
146 
147   -- UNIDAD DE PROCESO --
148   -- Contador de las muestras
149   process (clk, reset)
150   begin
151     if (reset = '1') then
152       memaddr_reg <= (others => '0');
153     elsif (clk'event and clk = '1') then
154       if (clearmuestras = '1') then
155         memaddr_reg <= (others => '0');
156       elsif (incmuestras = '1') then
157         memaddr_reg <= memaddr_reg + 1;
158       end if;
```

```
159     end if;
160   end process;
161   ultima <= '1' when (memaddr_reg = 256) else '0';
162   memaddr <= memaddr_reg;
163
164   -- Registro para guardar la muestra del AU_IN
165   process (clk, reset)
166   begin
167     if (reset = '1') then
168       si_reg <= (others => '0');
169     elsif (clk'event and clk = '1') then
170       if (clearsi = '1') then
171         si_reg <= (others => '0');
172       elsif (ldsi = '1') then
173         si_reg <= samplein;
174       end if;
175     end if;
176   end process;
177   memdati <= si_reg when (memact = '1') else (others => 'Z');
178
179   -- Registro para guardar la muestra de la RAM
180   process (clk, reset)
181   begin
182     if (reset = '1') then
183       din_re <= (others => '0');
184     elsif (clk'event and clk = '1') then
185       if (clearmd = '1') then
186         din_re <= (others => '0');
187       elsif (ldmd = '1') then
188         din_re <= memdata;
189       end if;
190     end if;
191   end process;
192
193   -- Registros para guardar el valor de la frecuencia maxima
194   process (clk, reset)
195   begin
196     if (reset = '1') then
197       dig3 <= (others => '0');
198       dig2 <= (others => '0');
199       dig1 <= (others => '0');
```

```
200       dig0 <= (others => '0');
201     elsif (clk'event and clk = '1') then
202       if (clearfreq = '1') then
203         dig3 <= (others => '0');
204         dig2 <= (others => '0');
205         dig1 <= (others => '0');
206         dig0 <= (others => '0');
207       elsif (ldfreq = '1') then
208         dig3 <= freq3;
209         dig2 <= freq2;
210         dig1 <= freq1;
211         dig0 <= freq0;
212       end if;
213     end if;
214   end process;
215 END SYN;
```

Apéndice II

Código del módulo MAXFFT

En el presente anexo se encuentra el código del bloque `MAXFFT` realizado en lenguaje de programación M de Matlab. Este código se ha embebido dentro de un bloque de Simulink y es perfectamente exportable a VHDL y sintetizable por la herramienta Quartus II de Altera.

MAXFFT.m

```
function [maxVal, maxIdx, maxValid] =
                        maxfft(din, dvalid, clear)
%#codegen\

% Variables registradas
persistent currentMaxVal;
persistent currentMaxIdx;
persistent currentMaxValid;
persistent contador;

% Inicializar los valores de la salida
if isempty(currentMaxVal)
    currentMaxVal = getfi(-256);
end

if isempty(currentMaxIdx)
    currentMaxIdx = uint8(0);
end

if isempty(currentMaxValid)
    currentMaxValid = false;
end

if isempty(contador)
    contador = uint8(0);
end

% Asignar a las salidas el valor de los registros
maxVal = currentMaxVal;
maxIdx = currentMaxIdx;
maxValid = currentMaxValid;

% Si clear == 1 ponemos todo a 0
if clear == true
    currentMaxVal = getfi(-256);
```

```
36      currentMaxIdx = uint8(0);
37      currentMaxValid = false;
38      contador = uint8(0);
39  end
40
41  if contador < uint8(128)
42      % Si dvalid == 1 comprobamos si din es mayor que
43      % el maximo hasta el momento
44      if dvalid == true
45          % Si el dato de entrada es mayor que el que
46          % tenemos guardado lo actualizamos el valor
47          % y guardamos el indice
48          din_abs = ab_abs(din);
49          if din_abs > currentMaxVal
50              currentMaxVal = din_abs;
51              currentMaxIdx = uint8(contador);
52          end
53          % Siempre incrementamos el contador
54          contador = contador + uint8(1);
55      end
56  else
57      currentMaxValid = true;
58  end
59
60  % Funcion para convertir a Fixed Point
61  function hdl_fi = getfi(val)
62  % Definicion del tipo de datos y operaciones
63  % Tipo de datos: Signed, 16 bits, 7 bits para
64  % la fraccion [-256,256)
65  nt = numerictype(1, 16, 7);
66  % Operaciones entre los datos
67  fm = fimath('RoundMode', 'Floor', ...
68      'OverflowMode', 'Saturate', ...
69      'ProductMode', 'FullPrecision', ...
70      'MaxProductWordLength', 128, ...
71      'SumMode', 'FullPrecision', ...
72      'MaxSumWordLength', 128, ...
73      'CastBeforeSum', true);
74  % Conversion del valor usando el tipo de datos
75  % y las operaciones
76  hdl_fi = fi(val, nt, fm);
```

```
77
78 function mag = ab_abs(val)
79 % Calcula la magnitud del numero complejo x
80 % El numero de entrada y la salida tienen que
81 % estar en fixdt(1,16,7)
82
83 mag = getfi(abs(val));
84
85 x = abs(real(val));
86 y = abs(imag(val));
87
88 mag = getfi(max(x,y) + bitsra(min(x,y), 2));
89
90 function abs = cordic_abs(val)
91
92 % Calcula la magnitud del numero complejo x
93 % El numero de entrada y la salida tienen que
94 % estar en fixdt(1,16,7)
95
96 n = uint8(3); % Numero de iteraciones
97
98 x = getfi(real(val)); % Parte real
99 y = getfi(imag(val)); % Parte imaginaria
100
101 % Primera rotacion para cambiar de cuadrante
102 if y <= 0
103     xo = getfi(-y);
104     yo = getfi(x);
105 else
106     xo = getfi(y);
107     yo = getfi(-x);
108 end
109
110 % Bucle de iteraciones para la aproximacion
111 for k = 1:n
112     ty = bitsra(yo,k-1);
113     tx = bitsra(xo,k-1);
114     if yo < 0
115         x=getfi(xo-ty);
116         y=getfi(yo+tx);
117     else
```

```
118         x=getfi(xo+ty);
119         y=getfi(yo-tx);
120     end
121     xo = x;
122     yo = y;
123 end
124 abs = x
```

Bibliografía

[1] Moore, G. E., *Electronics*, April 1965, *38*(8).

[2] Maxfield, C. M. *The Design Warrior's Guide to FPGAs*. Academic Press, Inc., 2004.

[3] Bell, R. Introduction to iec 61508. in *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*, SCS '05, pp 3–12, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[4] Altera dsp builder. `https://www.altera.com/products/design-software/model---simulation/dsp-builder.html`.

[5] Xilinx system generator. `http://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html`.

[6] Altera cyclone ii device handbook. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyc2/cyc2_cii5v1.pdf`.

[7] Wolfson wm8731 audio codec. `http://www.cs.columbia.edu/~sedwards/classes/2008/4840/Wolfson-WM8731-audio-CODEC.pdf`.

[8] Meyer-Baese, U. *Digital Signal Processing with Field Programmable Gate Arrays*. Springer Publishing Company, Incorporated, 3rd edition, 2007.

[9] Cordic faq. `http://www.dspguru.com/dsp/faqs/cordic`.

[10] Dsp trick: Magnitude estimator. `http://www.dspguru.com/dsp/tricks/magnitude-estimator`.