Universidad Central "Marta Abreu" de Las Villas

Facultad de Ingeniería Eléctrica

Departamento de Automática y Sistemas Computacionales

TRABAJO DE DIPLOMA

Metodología para la obtención de códigos HDL empotrables en FPGAs.

Autor: Javier Oscar Suárez Aday

Tutores: Alleini Machado Sosa

Consultantes: Juan Pablo Barrios Rodríguez

Lester Daniel Suárez Cabrera

Erisbel Orozco Crespo

Santa Clara

2013

"Año 55 de la Revolución"

Universidad Central "Marta Abreu" de Las Villas

Facultad de Ingeniería Eléctrica

Departamento de Automática y Sistemas Computacionales

TRABAJO DE DIPLOMA

Metodología para la obtención de códigos HDL empotrables en FPGAs.

Autor: Javier Oscar Suárez Aday

E-mail: aday@uclv.edu.cu

Tutores: MSc. Alleini Machado Sosa

Consultantes: Dr. Juan Pablo Barrios Rodriguez

Dr. Lester Daniel Suárez Cabrera

Ing. Erisbel Orozco Crespo

Santa Clara

2013

"Año 55 de la Revolución"

Hago constar que el presente trabajo de diploma fue realizado en la Universidad Central "Marta Abreu" de Las Villas como parte de la culminación de estudios de la especialidad de Ingeniería en Automática, autorizando a que el mismo sea utilizado por la Institución, para los fines que estime conveniente, tanto de forma parcial como total y que además no podrá ser presentado en eventos, ni publicados sin autorización de la Universidad.

_____

Firma del Autor

Los abajo firmantes certificamos que el presente trabajo ha sido realizado según acuerdo de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

_____ _____

Firma del Tutor

Firma del Jefe de Departamento donde se defiende el trabajo

_____

Firma del Responsable de Información Científico-Técnica

PENSAMIENTO

La sabiduría suprema es tener sueños bastante grandes para no perderlos de vista mientras se persiguen.

William Faulkner

DEDICATORIA

A mi familia en especial mis padres, mi hermano, mi tía y mis primos.

A mi novia y todos mis amigos.

AGRADECIMIENTOS

A mis familiares por el sacrificio que representaron mis cinco años de Universidad.

A mi novia y mis amigos.

A mis tutores y a los profesores que contribuyeron a mi formación como profesional.

A todos, gracias.

TAREA TÉCNICA

Con la intención de darle cumplimiento a los objetivos trazados a fin de realizar esta tesis, se tuvo en cuenta una serie de tareas técnicas de suma importancia para la confección del informe, ellas fueron:

1. Caracterizar los FPGA de Altera, en específico la familia Cyclone.
2. Análisis del kit de desarrollo DEO.
3. Analizar las herramientas de software a utilizar.
4. Proponer la metodología de trabajo para la obtención de códigos HDL a partir de Matlab.
5. Desarrollar algún ejemplo que emplee el Kit.

Firma del Autor

Firma del Tutor

RESUMEN

La presente investigación propone una metodología para la obtención de códigos HDL[1] correspondientes a bloques de Simulink y empotrables en FPGA[2]. Con este fin se hace un análisis de la arquitectura y estructura interna de estos dispositivos. Además se realiza una caracterización de los principales lenguajes de descripción de hardware, en particular VHDL. Se realiza un análisis del Quartus II, pues es este el software que se utiliza para diseñar circuitos sintetizables en FPGAs de Altera y se expone el flujo de trabajo para el diseño de aplicaciones con este software. Se describen las principales características del kit DEO, por ser esta la tarjeta empleada para el desarrollo de esta investigación. De esta forma se propone una solución para un problema real, consistente en el control de movimiento de plataformas neumáticas de simuladores de conducción. Para validar dicha solución se implementó en el Kit de desarrollo un Lector de Encoder, siendo este uno de los elementos que componen el sistema propuesto. Este sistema puede ser implementado en una FPGA en futuros trabajos.

[1] Hardware Description Languaje
[2] *Field Programable Gate Array*

TABLA DE CONTENIDOS

INTRODUCCIÓN

El hardware lógico programable es una alternativa interesante para los ingenieros de diversas áreas basadas en electrónica. Una evidencia de ello, está reflejada en el crecimiento y desarrollo de las empresas dedicadas a este campo, entre las que se destacan: Xilinx y Altera. Existe polémica respecto a qué es mejor, si las soluciones basadas en sistemas con microprocesadores o soluciones con hardware lógico programable como CPLD[3] y FPGA. En realidad este cuestionamiento no es tan importante si se tiene en cuenta que cada una de estas vertientes tiene su propósito particular, su área para la cual son idóneos, independientemente a que puedan darse soluciones a problemas con las dos opciones. En el caso de FPGA y CPLD, se puede decir que han tributado de manera especial a la creación de prototipos de hardware, lo cual puede además estar asociado al desarrollo de aplicaciones específicas versátiles de procesamiento digital de señales, comunicaciones, control industrial, entre otros.

Lo mencionado anteriormente, justifica la utilización de esta novedosa tecnología en la implementación de sistemas empotrados a la medida; donde además de microprocesadores se requiera algún tipo de lógica secuencial o combinacional, que acelere alguna función que demande requerimientos críticos de tiempo, imposibles de realizar por el microcontrolador empleado. Además existe la necesidad de interfaces, que de realizarse solo con microcontroladores y lógica discreta, resultaría engorrosa, voluminosa y por consiguiente poco fiable; además que sería siempre una solución costosa.

Tal es el caso de la electrónica asociada a los simuladores de conducción, fabricados por el Centro de Investigación y Desarrollo de Simuladores Profesionales (SIMPRO). Anteriormente se han realizado trabajos relacionados con la implementación de controladores empotrados para plataformas neumáticas, (Rodríguez, 2008) y (Morfa, 2011); además se desarrollaron controladores que utilizaron hardware como Flashlite (Sosa, 2007), donde las soluciones de hardware para el control de movimiento de las plataformas

[3] *Complex Programmable Logical Device*

neumáticas que dan movilidad a dichos simuladores, dadas por el grupo de Automatización Robótica y Percepción (GARP) a cargo del proyecto, han estado basadas en microcontroladores. Estas soluciones necesitan microcontroladores de altas prestaciones, pues el cálculo de los algoritmos de control demanda períodos de muestreo de 1 ms y costo computacional elevado, además de ser algoritmos no convencionales que deben ser implementados en dispositivos programables. El alcance de los trabajos anteriormente mencionados, llegaba hasta la implementación de los algoritmos de control, mientras que el resto de las tareas del simulador relacionadas con hardware, entiéndase adquisición de datos de palancas, timones etc; así como todo el trucaje de los indicadores de velocidad, combustible, entre otros, es realizado por una electrónica basada en componentes discretos con altas tasas de fallos y además costosa de producir y mantener.

La solución a tal disyuntiva está precisamente en la utilización de lógica programable como los FPGAs y CPLDs, los cuales permiten con bajos costos, empotrar en un sistema único y compacto tanto la adquisición y trucaje del simulador, como el control de movimiento. Esto traería como ventaja que para nuevos diseños de simuladores, los cambios del hardware serían mínimos. El peso fundamental estaría en la programación de las diferentes interfaces; así como de los algoritmos de control de movimiento, reduciendo costos, tiempo de desarrollo y puesta en el mercado.

Desafortunadamente, aunque la solución propuesta anteriormente es prometedora, aún quedan algunas lagunas por resolver y es precisamente lo que da pie al problema científico de esta tesis. En este sentido es necesario señalar que la metodología de desarrollo e implementación de algoritmos de control no convencionales, por parte del GARP, para dar solución a los problemas de control de las plataformas de los simuladores producidos por SIMPRO, siempre ha estado marcada por el uso de herramientas de software como Matlab-Simulink. Con el empleo de estas herramientas se editaban, se ajustaban, se simulaban y finalmente se probaban los algoritmos de control, para luego codificarlos en librerías de lenguaje C y ser usadas en la implementación de los controladores empotrados, basados en microcontroladores. En esta nueva variante que se propone, el sistema empotrado final, estaría basado en FPGAs o CPLDs y el código del algoritmo de control tendría que estar en código VHDL o Verilog, compatible con dichos dispositivos programables.

A partir de lo mencionado anteriormente podríamos formular nuestro problema científico: ¿Es posible obtener algoritmos de control desarrollados en Matlab-Simulink codificados en VHDL y listos para ser usados en controladores empotrados basados en FPGAs o CPLDs?

La Hipótesis que responde a esta interrogante afirma que si es posible lograr una metodología que nos permita obtener algoritmos de control no convencionales codificados en VHDL a partir del uso de herramientas como Matlab-Simulink.

Para cumplir con lo postulado en la Hipótesis planteada, se propone el Objetivo General:

- Desarrollar una metodología que nos permita obtener algoritmos de control codificados en HD, listos para ser empotrados en sistemas basados en FPGAs o CPLDs a partir de la poderosa herramienta de desarrollo que constituye el binomio Matlab-Simulink.

De este objetivo, se derivan los siguientes objetivos específicos:

- Realizar una revisión bibliográfica relacionada con el tema de los FPGAs y CPLDs.
- Profundizar en el estudio de lenguajes de descripción de hardware tales como VHDL y Verilog.
- Investigar las potencialidades del HDL Coder de Matlab y sus variantes de generación de código.
- Proponer una solución para el controlador empotrado de las plataformas neumáticas de los simuladores de conducción producidos por SIMPRO basado en FPGAs de Altera.
- Implementar algún elemento del controlador propuesto en el Kit de desarrollo DEO.

Para dar cumplimiento a los objetivos planteados en este trabajo de investigación, se propone la estructura siguiente:

Capítulo 1: Dispositivos basados en FPGA y sus aplicaciones.

Este capítulo brinda una breve introducción a los PLDs[4], particularmente a las FPGAs y aún más preciso al chip Cyclone III de Altera. Expone una revisión bibliográfica de los materiales recopilados que tributan a la realización del trabajo. Aborda características internas de las FPGA. Se realiza un análisis de los principales lenguajes de descripción de hardware en especial VHDL y se desarrolla un estudio de algunos fabricantes de PLD y sus principales características.

Capítulo 2: Metodología de diseño con el HDL Coder de Matlab.

En este capítulo se realiza un análisis de diversos software en los que se pueden diseñar circuitos lógicos empotrables en FPGAs y CPLDs, tal es el caso del Xilinx ISE desarrollado por Xilinx y el Quartus II desarrollado por Altera. Este último software se aborda a grandes rasgos debido a que en el próximo capítulo se realizará un análisis detallado del flujo de diseño con el mismo. Se realiza una metodología que nos permite obtener códigos VHDL y Verilog HDL correspondiente a bloques de Simulink sintetizables en una FPGA.

Capítulo 3: Propuesta de Controlador Empotrado en FPGA.

En este capítulo se presenta una propuesta de solución para el controlador empotrado de las plataformas neumáticas de los simuladores de conducción producidos por SIMPRO basado en FPGAs de Altera. Se implementó en el Kit uno de los elementos del controlador propuesto disponible en el departamento, un lector de encoder. Se verificó su adecuado funcionamiento obteniendo los resultados esperados.

[4]*Programmable Logical Device*

CAPÍTULO 1. DISPOSITIVOS BASADOS EN FPGA.

En este capítulo se abordan las características generales de los FPGA, la estructura interna y principales prestaciones de los mismos. Se realiza una revisión de los principales fabricantes y se expone un análisis de los principales lenguajes de descripción de hardware haciendo énfasis en VHDL. Además se desarrolla un estudio de los principales fabricantes de FPGA y CPLD.

1.1. **Fundamentos de los FPGA**

Los dispositivos lógicos programables o PLD son circuitos integrados digitales que no tienen una función predefinida por el fabricante. Su función puede ser definida o programada por el usuario. Además, los PLDs tienen una estructura interna regular, flexible y pueden ser programados mediante una estructura de interruptores. Permiten reemplazar grandes diseños digitales que antes se realizaban con componentes discretos como compuertas y *flip-flops*, por un solo integrado. Entre los principales tipos de PLD se encuentran los CPLD y los FPGA; cabe añadir que estos dos tipos son también de los más utilizados en la actualidad;(Bozich, 2005),(Charles H. Roth and Kinney., 2010).

Dentro de la clasificación de los PLDs, resulta en ocasiones un tanto difícil ubicar a ciertos ejemplares, no obstante, varios autores lo han logrado muy bien a pesar de la diversidad. No todas las variantes de PLDs son tan usadas en la actualidad y precisamente, las más comunes, no presentan esta problemática en cuanto a su clasificación; tal es el caso de los FPGA (Oliver, 2007).

Un CPLD, extiende el concepto de un PLD a un nivel de integración superior. Se puede decir que tienen un punto de partida en los SPLD[5], dispone de mayor número de compuertas y de puertos de entradas/salidas en un circuito programable. Cada CPLD contiene bloques lógicos que se comunican entre sí mediante una matriz programable de interconexiones; esta estructura interna está diseña con los objetivos de hacer un eficiente uso del silicio, tener buen desempeño y minimizar el costo del dispositivo.

De la propia traducción del término en inglés, se puede definir el FPGA como un arreglo de compuertas programable por campos. El significado de este término está dado a partir de la

[5]*Simple Programmable LogicalDevice*

propia estructura interna, diseñada para el desarrollo de vastos sistemas en un chip o SoC[6]. Debido a la gran capacidad lógica que tienen estos dispositivos modernos, sistemas complejos pueden desarrollarse sobre un solo circuito integrado. Esto da lugar a las denominaciones: sistemas programables en un chip o, SoPC[7] y sistemas reprogramables en un chip o SoRC[8] (Zamora, 2010).

Los CPLDs y FPGAs actuales tienen una capacidad lógica de hasta millones de compuertas, incluyen interfaces programables para varios estándares de interfaz eléctrica y tienen bloques de funciones especiales, embebidos entre la lógica programable tales como memorias, multiplicadores e incluso CPUs.

Es importante mencionar que en el presente, la frontera entre FPGA y CPLD se puede tornar borrosa en algunos casos, como por ejemplo con la familia MAX de Altera. En esta familia de CPLDs, si bien tiene programación no volátil, la misma está almacenada en una memoria flash interna al chip y se carga en una SRAM de configuración al iniciar el dispositivo. Por eso, se hace preciso conocer a fondo la estructura interna del dispositivo, sus características y su funcionamiento para avalar su clasificación. Es un error pensar como desarrollador para FPGA o para cualquier otro PLD, sin antes pasar por una revisión de este tipo a partir de los datos de los fabricantes y de los autores clásicos en el tema (Oliver, 2007).

En (Oliver, 2007) se menciona que el origen de la estructura del FPGA se remonta al año 1984, cuando Xilinx da a conocer un dispositivo lógico programable denominado *Logic Cell Array*, compuesto de una gran cantidad de celdas lógicas programables, interconectadas a su vez por bloques de conexión programables de diversos tipos. Estos dispositivos eran diferentes a cualquier PLD hasta ese momento.

1.2 Estructura interna

Los CPLDs tienen una estructura básica basada en dos niveles AND-OR, con el primer nivel AND programable y el nivel OR fijo; y con un *Flip-Flop* a la salida del OR como se muestra en la Figura 1.1.

---

[6] *System-on Chip*
[7] *System-on-a-Programmable Chip*
[8] *System-on-a-Reprogrammable Chip*

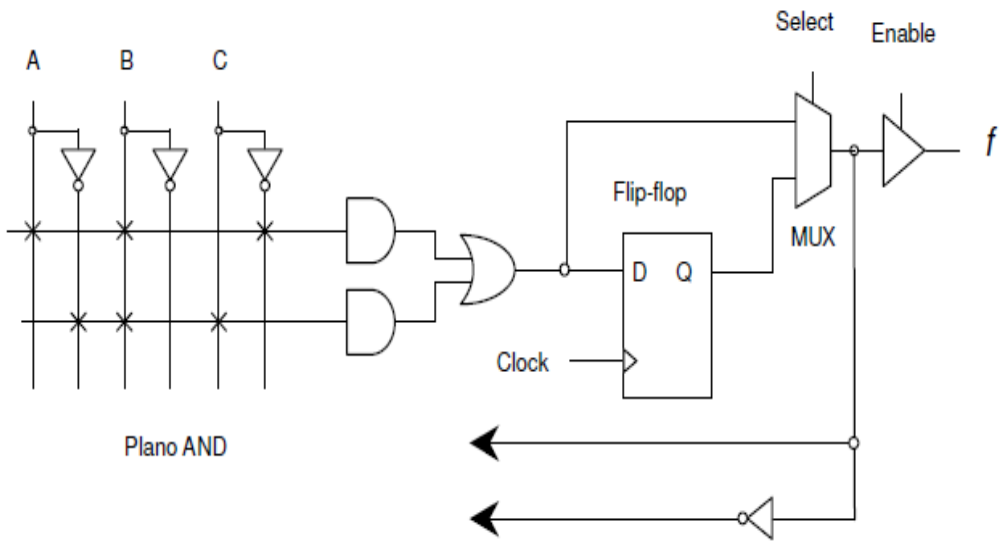


Figura 1.1 Estructura de la *macrocelda*[9] de un CPLD. Extraída de (Oliver, 2007).

Un FPGA consiste en arreglos de varios bloques programables o bloques lógicos, los cuales están interconectados entre sí y con celdas de entrada/salida mediante canales de conexión vertical y horizontal, tal como muestra la Figura 1.2. En general, se puede decir que posee una estructura bastante regular, aunque el bloque lógico y la arquitectura de ruteado varían de un fabricante a otro.

[9] Viene del término *macrocell*, pero es una palabra que no existe en el español.

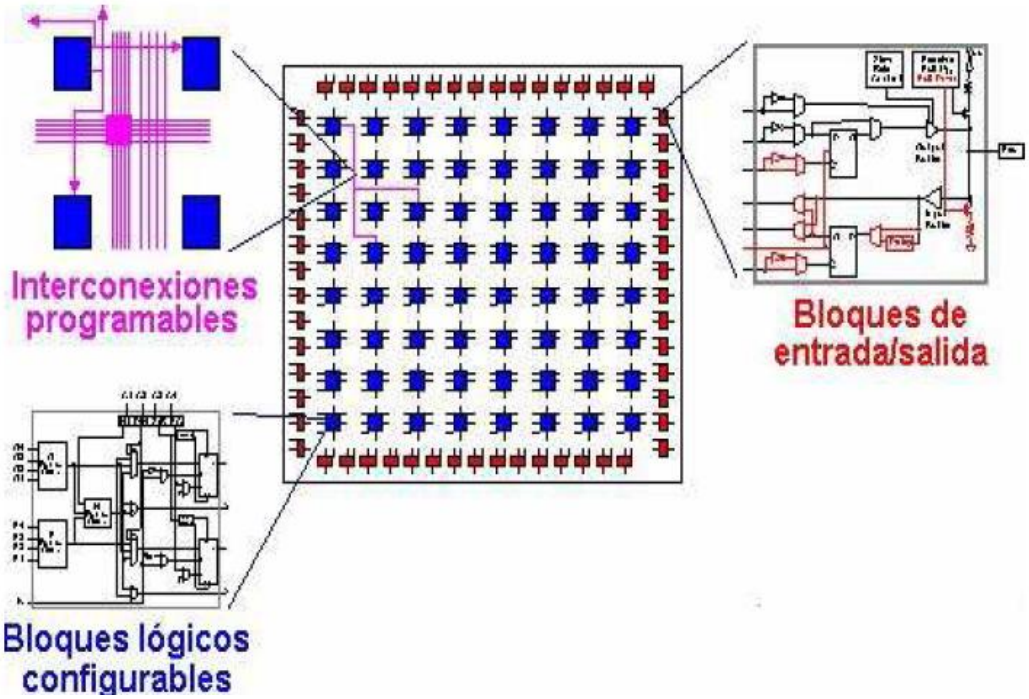


Figura 1.2 Estructura Interna clásica de una FPGA. Imagen tomada de.(ALTERA, 2013)

1.2.1 Bloques Lógicos Configurables

Los Bloques Lógicos Configurables o CLB[10] constituyen una estructura muy importante dentro de un FPGA. Constan de una parte combinacional que permite implementar funciones lógicas booleanas, más una parte secuencial que permite sincronizar la salida con una señal de reloj externa e implementar registros. Estos bloques contienen tablas de consultas LUT[11] las cuales funcionan como generador de funciones. A continuación se presenta la estructura interna de una LUT de cuatro entradas y una salida (ver Figura 1.3).

[10] *Configurable Logic Block*
[11] *Look Up Table*

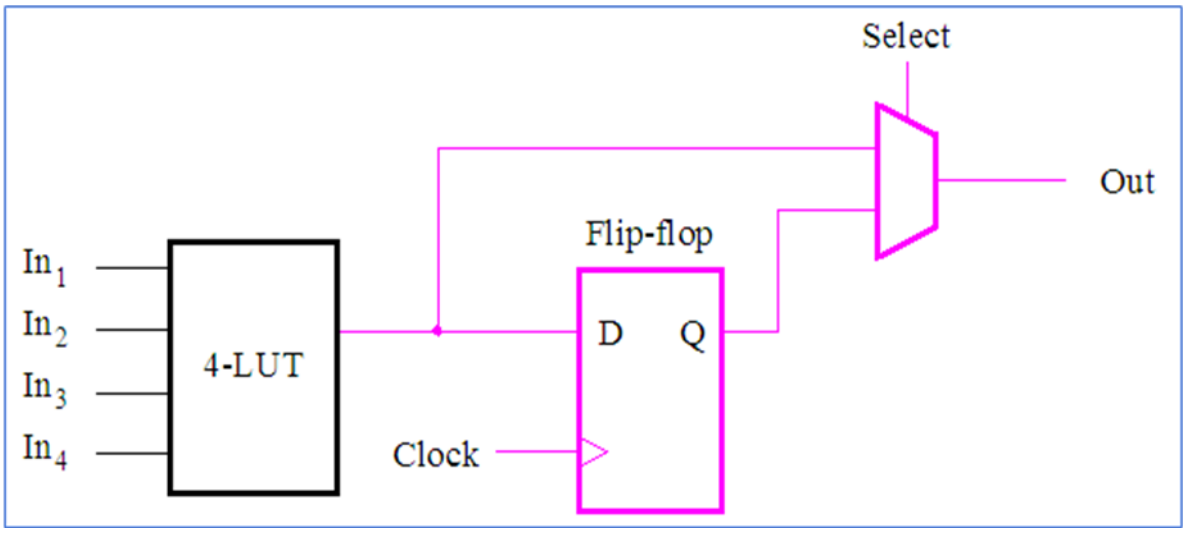


Figura 1. 3 Estructura típica general de un generador de funciones del CLB. Imagen tomada de (Czajkowski, 2010).

Los elementos que permiten implementar lógica secuencial son los elementos de almacenamiento como *flip-flops* tipo D. La salida del CLB es conectada a segmentos cableados usando Puntos de Interconexión Programables o PIPs. Si el bloque no usa todos los PIPs, estos se desconectan y el bloque no genera señal a ningún segmento. (Bozich, 2005).

1.2.2 Bloque de Entrada/Salida

Los Bloques de Entrada/Salida (IOB) son otros de los componentes fundamentales dentro de una FPGA. Para implementar circuitos lógicos se hace necesario configurar adecuadamente los bloques de I/O, la función de estos bloques es permitir el paso de una señal hacia dentro y hacia el exterior del dispositivo. Cada IOB controla un pin del encapsulado, puede ser configurable como entrada, salida o bidireccional. Estos bloques no solo deben configurarse adecuadamente, también deben conectarse correctamente con los CLB mediante interconexiones programables (Bozich, 2005).

1.2.3 Interconexiones programables

Las interconexiones usualmente están organizadas en forma de una malla jerárquica, con caminos rápidos entre bloques contiguos, caminos verticales y horizontales. Así, los elementos de procesamiento forman una isla rodeada de líneas de interconexión. Usualmente existen tres tipos de interconexiones:

- Interconexiones directas entre bloques adyacentes.
- Interconexiones de propósito general, que conectan canales verticales y horizontales hasta llegar al punto final de la conexión. Se realizan a través de una matriz de interconexión para conectar dos bloques no adyacentes.
- Líneas largas, reservadas para distribuir señales críticas fundamentalmente señales de reloj.

La figura 1.4 muestra lo mencionado anteriormente detalladamente.

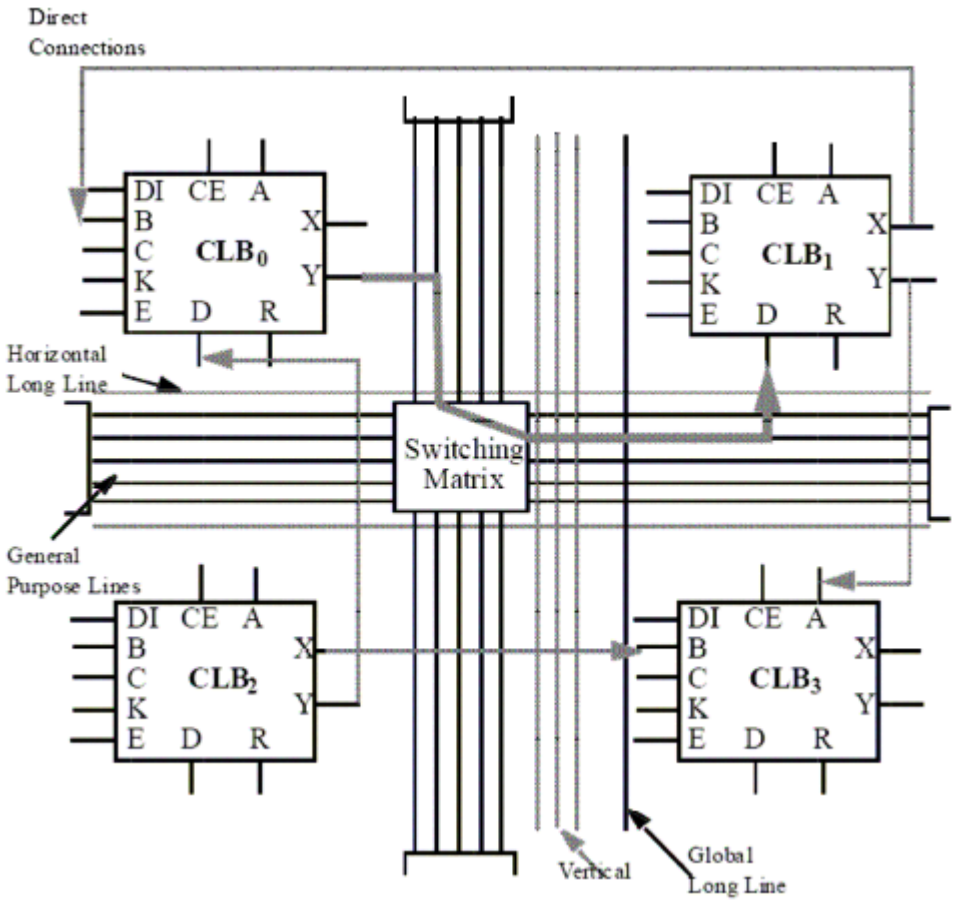


Figura 1.4 Diferentes tipos de interconexiones. Imagen capturada de (Oliver, 2007).

1.3 Flujo de diseño basado en FPGA

Para iniciar el trabajo con una FPGA se necesita crear el diseño; ya sea en códigos o de forma esquemática. Luego de cada paso se debe hacer una simulación. Lo siguiente es la síntesis del modelo seguido de la simulación correspondiente a dicho paso; en este punto se simula basado en el diseño para validar el modelo sintetizado. Posteriormente se

implementa el modelo, en caso de haber errores se hace una revisión en los pasos anteriores, de no obtener los resultados deseados luego de esta revisión se recomienda comenzar con un nuevo diseño. En el caso de Altera el ciclo se desarrolla en el software Quartus II y luego es implementado directamente en una FPGA. La Figura 1.5 muestra el flujo de diseño para una FPGA de Altera con el software Quartus II.

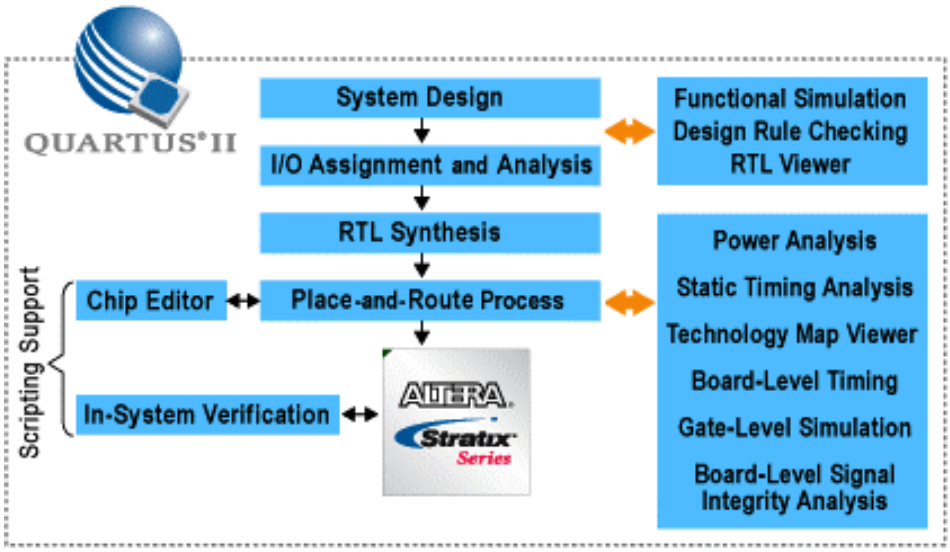


Figura 1.5 Flujo de diseño con productos de Altera. Extraída de (ALTERA, 2013).

La mayoría de los fabricantes de FPGAs entregan su recomendación para la metodología de trabajo en función del software EDA[12] que venden para operar con sus productos. Es importante conocer que más allá de esta particularidad, existe una metodología general para el diseño con FPGAs. Esta metodología está más en correspondencia con el fin del diseño que no con el software o el fabricante. Una buena visión de esto se hace en la Figura 1.6.

[12] *Electronic Design Automation*

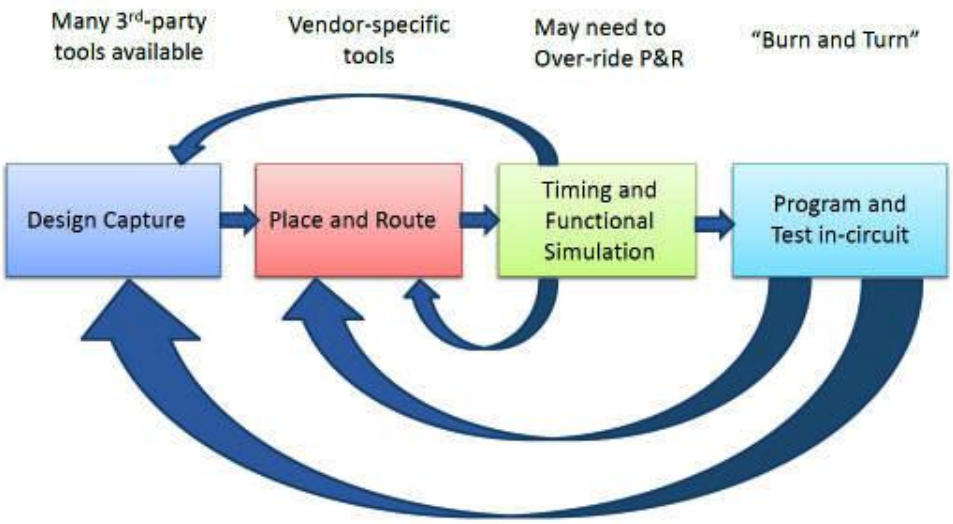


Figura 1.6 Visión general del flujo de diseño para CPLDs y FPGAs. Tomado de (Brown and Rose, 2010)

En la Figura 1.6 se es explícito en que para la captura del diseño, existen diversas herramientas que bien pueden ser de terceros. Ya el procedimiento de ruteo y emplazamiento depende del fabricante, por las razones obvias de que es el único que conoce la forma de emplear su tecnología a fondo. En el propio sitio de (Brown and Rose, 2010), aparecen en detalle los flujos de diseño con Altera y con Xilinx. El primero se expone en el Anexo 1.

1.4 Lenguajes de descripción de hardware

Los lenguajes de descripción de hardware permiten modelar sistemas digitales completos. Mediante herramientas de software EDA, estos modelos pueden luego sintetizarse para realizarlos como circuitos reales. La utilización de HDLs para sintetizar sistemas digitales y la utilización de PLDs, permiten crear prototipos funcionales en plazos relativamente cortos. Esto hace que todo el proceso de desarrollo de un sistema digital sea mucho más simple y rápido en comparación con metodologías clásicas.

Entre los modelos de hardware en HDLs están los estructurales y de comportamiento. A nivel estructural se describe la interconexión y jerarquía entre componentes. A nivel de comportamiento de hardware se describe la respuesta I/O de un componente. El comportamiento de un sistema puede modelarse a distintos niveles de abstracción: algoritmos y comportamiento general, nivel de transferencia de registros, nivel de

compuertas, etc. El tipo de modelo más usado para síntesis es el denominado RTL[13] o nivel de transferencia de registros (CHU, 2006).

Al utilizar un HDL para modelar un sistema, es importante recordar que se está modelando hardware y no escribiendo software. El software se caracteriza por ser secuencial, una CPU ejecutará cada instrucción después de la anterior. Los efectos de una instrucción dependen exclusivamente de los efectos de las instrucciones anteriores. En el hardware, sin embargo, hay muchas tareas que suceden de manera concurrente y el tiempo como variable juega un papel predominante. Un cambio en el tiempo de propagación de una compuerta o el no cumplir con los tiempos de establecimiento de un circuito puede cambiar de manera radical el comportamiento de un sistema electrónico digital.

Aunque hay diversos lenguajes de descripción de hardware, dos predominan actualmente en el mundo del desarrollo de hardware programable: VHDL y Verilog.

1.4.1 VHDL

Las siglas VHDL provienen de *VHSIC Hardware Description Languaje* y a su vez VHSIC quiere decir *Very High Speed Integrated Circuit*. O sea que VHDL significa lenguaje de descripción de hardware para circuitos integrados de alta velocidad. En la actualidad, VHDL se utiliza no solo para modelar circuitos electrónicos; sino también para crear o sintetizar nuevos circuitos. Las múltiples facilidades que ofrece este lenguaje con respecto a otros HDLs, han propiciado que VHDL sea utilizado para el desarrollo de diferentes proyectos sobre todo en países de América Latina y Europa. Permite modelar, diseñar y simular desde un alto nivel de abstracción hasta un nivel más bajo. Es un lenguaje independiente de la tecnología y del fabricante. Admite implementar diseños existentes en nuevas tecnologías. Requiere de un corto tiempo de llevar el diseño al mercado. VHDL facilita la descripción del proceso rápidamente. Dispone de una gran versatilidad para la descripción de sistemas complejos, por lo que posee una sintaxis amplia y flexible. (Maza, 2008)

Entre las características deseables del lenguaje está el hecho de que permite la descripción de sistemas concurrentes, es decir, paralelos en el tiempo. Además es un lenguaje del tipo

[13] *Register Transfer Level*

flujo de datos que, a diferencia de los lenguajes que ejecutan secuencialmente las instrucciones, permite la ejecución da las operaciones en disposición del flujo de datos entre éstas. Quizás la debilidad más sensible del lenguaje es que no incluye explícitamente los recursos para modelar sistemas analógicos.

Otro aspecto relevante del lenguaje es que tiene un grupo de trabajo para la estandarización en la IEEE. El lenguaje es un estándar de la IEEE. Este grupo es el P1076 con nombre VHDL *Analysis and Standardization Group* (VASG). En el año 2008 *Accellera Systems Initiative* VHDL 2008 o 4.0 para que la IEEE lo incluyera en su estándar IEEE 1076-2008 que fue publicado en enero del 2009. De cualquier forma es común que el software actual brinde la opción de especificar qué versión del lenguaje se emplea en los modelos.

1.4.2 Verilog HDL

Es un lenguaje de descripción de hardware usado para modelar sistemas electrónicos. El lenguaje, algunas veces llamado Verilog, soporta el diseño, prueba e implementación de circuitos analógicos y digitales a diferentes niveles de abstracción. Un diseño en Verilog consiste de una jerarquía de módulos. Los módulos son definidos con conjuntos de puertos de entrada, salida y bidireccionales. Internamente un módulo contiene una lista de cables y registros. Las sentencias concurrentes y secuenciales definen el comportamiento del módulo, describiendo las relaciones entre los puertos, cables y registros (Oliver, 2007).

Verilog también está estandarizado como IEEE 1364 y su evolución llega a la actualidad con el estándar IEEE 1800-2009 que representa al System Verilog.

Los diseñadores del lenguaje buscaron una sintaxis similar a la del lenguaje C, esto lo diferencia notablemente de VHDL. Otra diferencia es que es sensible a mayúsculas y minúsculas. Además tiene un preprocesador que, aunque es más básico que el del C, VHDL no lo tiene en su paradigma de funcionamiento.

1.5 Fabricantes de PLDs y kits de desarrollo basados en PLDs

En el caso de Xilinx, las primeras familias que surgieron fueron: XC2000, XC3000 y XC4000; correspondientes respectivamente a la primera, segunda y tercera generación de dispositivos. Se distinguen por el tipo de bloque lógico configurable que contienen. En la

actualidad existen también las Spartan, Spartan II, Spartan XL, Spartan IIE, Spartan III, Virtex, Virtex II y Virtex IIPro (Bozich, 2005).

Son muchas las compañías en la actualidad enfocadas en el desarrollo de hardware lógico programable. En la producción de FPGAs se encuentran además de Xilinx los fabricantes Altera, Actel, Atmel, Achronix, Agere, Quick Logic, Cypress y Lattice.

En el caso de Altera los CPLDs y las FPGAs se diferencian por diversas estructuras de interconexión. La estructura de interconexión segmentada es utilizada por las FPGAs y utilizan líneas múltiples de longitud variable unidas por transistores de paso o antifusibles para conectar las celdas lógicas. En contraste, la estructura de interconexión continua es utilizada por los CPLDs, de acuerdo a esta clasificación se toman a los dispositivos APEX, ACEX y FLEX, Cyclone y Stratix como FPGAs mientras que a los MAX se toman como CPLDs (Bozich, 2005).

Altera es el otro pionero de la lógica programable, desarrolla algunas características que están orientadas hacia capacidad de sistemas en chips programables (SOPC). Algunos de los ejemplos más recientes incluyen memoria embebida, procesadores embebidos, y transceptores de alta velocidad. Los procesadores tipo *soft-core*, Nios II, Nios y los dispositivos Hard Copy II y Hard Copy, han extendido el alcance de Altera en el mercado y coloca a esta empresa en el mundo de los procesadores embebidos y ASICs estructuradas respectivamente. Altera ofrece también el software Quartus II, dirigido al diseño y simulación de circuitos lógicos. Quartus II es una herramienta de software producida por Altera para el análisis y la síntesis de diseños realizados en HDL, le permite al programador compilar sus diseños y realizar análisis de secuenciación temporal (Altera, 2003).

1.5.1 Familia Cyclone

Los dispositivos Cyclone contienen una arquitectura basada en fila y columna de dos dimensiones para implementar lógica. Las conexiones columna y fila de distintas velocidades proporcionan señales de interconexión entre los Arreglos de Bloques Lógicos (LABs) y los bloques integrados de memoria. Cada LAB está constituido por 10 elementos lógicos. Los dispositivos Cyclone tienen entre 2910 a 20060 elementos lógicos. Un elemento lógico es una unidad pequeña de lógica que proporciona una aplicación eficaz de funciones lógicas. Cada pin de entrada-salida del dispositivo Cyclone es alimentado por un

elemento de entrada-salida (IOE) ubicado en los finales de las filas y columnas del LAB, alrededor de la periferia del dispositivo. Cada IOE contiene un buffer de E/S bidireccional y tres registros para registrar señales de entrada, de salida, y señales de habilitación de salida (Bozich, 2005).

1.5.2 Kit DEO de Terasic

La tarjeta DEO con que se cuenta para este trabajo es una plataforma desarrolladora de circuitos basada en la FPGA Cyclone III EP3C16F484 de Altera. Entre las principales características de este Kit se encuentran las siguientes:

Contiene un puerto USB *blaster* para la programación de la FPGA, tiene 8-Mbyte SDRAM, 4-Mbyte de memoria flash, una entrada de tarjeta SD, 3 botones de pulso, 10 interruptores, 10 LEDs activos en verde. Además el Kit posee un puerto PS-2 para mouse y teclado, dos puertos de expansión de 40 pines cada uno. El kit DEO proporciona tecnologías de primera que cualquier persona puede utilizar para ganar experiencia en el diseño digital. En esta tarjeta se puede diseñar incontables sistemas digitales basados en FPGAs y estos pueden crecer fácilmente más allá de la misma. La Figura 1.7 muestra el diagrama en bloques del Kit DEO de Terasic extraído de (Terasic, 2009).

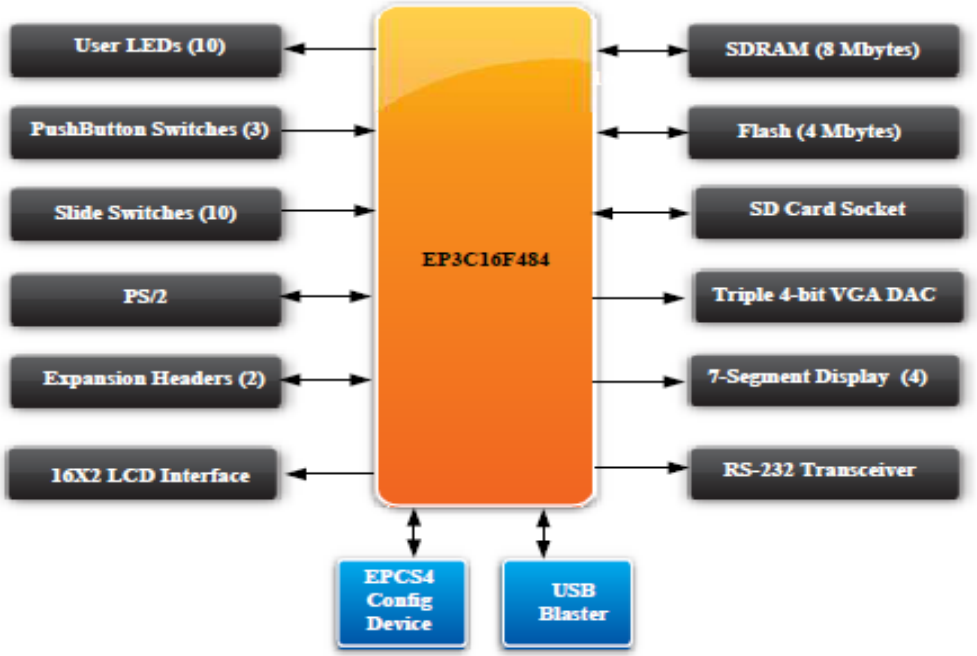


Figura 1.7 Diagrama en Bloques del Kit DEO de Terasic.

Las flechas que señalan hacia adentro indican que esa interfaz se puede usar como entrada, en caso que señalen hacia afuera se podrán usar como salida, si las flechas están en las dos direcciones significa que se pueden usar de ambas formas. Este diagrama posee máxima flexibilidad para el usuario y todas las conexiones están hechas a través de la FPGA Cyclone III.

1.6 Generación de código HDL con Matlab

Antes de analizar en detalles las herramientas generadoras de código HDL, se hace necesario estudiar algunas características propias de Matlab y Simulink para facilitar el trabajo.

1.6.1 Matlab

Matlab es un programa de cálculo numérico y visualización de datos para la resolución de problemas complejos planteados en la realización y aplicación de modelos matemáticos de ingeniería. Este software posee una extraordinaria versatilidad y capacidad para resolver problemas de Matemática aplicada, Física, Ingeniería entra otras aplicaciones. Su base la constituye el cálculo matricial e integra análisis numérico, procesamiento de señales y visualización gráfica. Además se encuentra disponible para un amplio número de plataformas que operan bajo sistemas operativos Unix, Mac OS y Windows. Matlab dispone de herramientas adicionales que expanden sus prestaciones como Simulink.

1.6.2 Simulink

Simulink es una herramienta para modelar, simular y analizar sistemas dinámicos, proporciona una Interfaz Gráfica del Usuario (GUI), para construir modelos como diagramas de bloques, utilizando operaciones con el ratón del tipo pulsar y arrastrar. El comportamiento de dichos sistemas se define mediante funciones de transferencia, operaciones matemáticas, elementos de Matlab y señales predefinidas. Después de definir un modelo, se puede simular e interactuar con dicha simulación a través de los diferentes menú que ofrece el programa. Además, es posible cambiar los parámetros y ver de forma inmediata lo que sucede luego del cambio; pudiendo transferir los resultados de la simulación al espacio de trabajo del programa Matlab, para su posterior procesamiento y visualización. Simulink incluye una amplia biblioteca de fuentes y herramientas de visualización que lo hacen ideal para el diseño de sistemas. Esto supone un cambio radical

respecto a otras herramientas de simulación, que requieren una formulación de las ecuaciones en forma de lenguaje o programa.

1.6.3 HDL Coder

HDL Coder permite la generación de códigos VHDL y Verilog HDL a partir de diagramas en bloques de Simulink.. Lo más significativo de esta posibilidad es el hecho de minimizar la complejidad del proceso de modelación basado en el lenguaje. El alto grado de abstracción favorece a la rapidez con la que se puede generar el modelo. Quizás la mayor deficiencia está en la poca especialización del código generado, que en cierto escenario se puede traducir en ineficiencia e incluso en un resultado no deseado. Esta variante también genera código para excitar el circuito a la hora de simular, este código es lo que los autores denominan como *Testbench*. En el próximo capítulo se abordará con detalles el flujo de diseño con esta herramienta.

1.6.4 System Generator

El System Generator es una herramienta de diseño de alto nivel desarrollada por Xilinx que permite el diseño basado en modelos en el entorno Matlab/Simulink para el desarrollo de sistemas de procesamiento digital en FPGAs.

Como todos los *blocksets* de Simulink, System Generator se integra como una biblioteca de bloques que pueden ser conectados para crear modelos funcionales de un sistema dinámico. De esta forma, permite modelar, simular y analizar sistemas de procesamiento complejos y de alto rendimiento para una plataforma de hardware específica, mediante un entorno flexible, robusto y fácil de utilizar. System Generator no puede considerarse como un sustituto de los lenguajes de descripción de hardware hasta la fecha, pero sí como un potente complemento. Es válido aclarar que los códigos obtenidos con esta herramienta son sintetizables solo en FPGAs de Xilinx (Elhossini, 2013)

1.7 Conclusiones del Capítulo

En este capítulo se realizó un acercamiento a los dispositivos lógicos programables y en específico los FPGAs donde se abordan los elementos principales que describen su arquitectura. Además se realiza un estudio de los principales fabricantes de PLD haciendo énfasis en los FPGA de Xilinx y Altera. El estudio estuvo enfocado fundamentalmente en los

FPGAs de Altera en específico el Cyclone III, dando a conocer las características más significativas del mismo. Se muestran las principales características de los lenguajes de descripción de hardware como VHDL y Verilog, los cuales son usados en el diseño digital. Se analizan algunas herramientas de generación de código HDL como el HDL Coder y el System Generator, ambas herramientas desarrollas en Matlab.

CAPÍTULO 2. METODOLOGÍA DE DISEÑO CON EL SIMULINK HDL CODER.

El diseño de aplicaciones a implementar en los Dispositivos Lógico Programable como ya sabemos se realiza en lenguajes de descripción de hardware. En la actualidad los diseñadores necesitan, una herramienta que permita evaluar la viabilidad y el desempeño de los sistemas implementados. Para cubrir estas necesidades Xilinx creó el software Xilinx ISE (*Integrated Software Environment*) y Altera desarrolló la herramienta de software Quartus II.

2.1 Quartus II

Quartus II es una herramienta de software producida por Altera para el análisis y la síntesis de diseños realizados en HDL. Este software brinda un ambiente de diseño multiplataforma que se adapta con facilidad a las necesidades del diseñador, brindando grandes facilidades en el diseño de SOPC. Quartus II incluye soluciones para cada una de las fases de diseño de FPGAs y CPLDs. Además, Quartus II permite el uso de interfaces gráficas de usuario (GUI) e interfaces de líneas de comando para cada una de las etapas de diseño; las mismas pueden ser utilizadas indistintamente, sin restricción de tipo alguno. Quartus II permite al desarrollador compilar sus diseños, realizar análisis temporales, examinar diagramas RTL y configurar el dispositivo de destino con el programador. Como contraste adicional, se tiene el bajo precio del Quartus II en comparación con otras herramientas de diseño de ASIC. En el capítulo 3 de este trabajo se explica el flujo de diseño en Quartus II para una aplicación en particular.

2.2 Xilinx ISE

El entorno de diseño Xilinx ISE es una herramienta de software que realiza diseño digital basado en lógica programable e incluye todas las etapas del flujo de diseño basado en FPGA. El diseño se puede realizar a través de lenguaje de descripción de hardware o mediante representación gráfica de diagrama de estado. La síntesis es el proceso que se realiza luego de la entrada del diseño y prepara el terreno para realizar la implementación. Este segundo paso se realiza con XST (Xilinx Synthesis Technology), sintetizador nativo de Xilinx. En la implementación se convierte el circuito lógico en un fichero físico que

puede ser cargado en la FPGA. Luego se verifica la funcionalidad del diseño utilizando el software como simulador. El simulador interpreta el código fuente dentro de la lógica circuital y muestra resultados lógicos que determinan si el funcionamiento es correcto. La simulación permite crear y verificar funciones complejas en tiempo relativamente corto, pudiéndose comprobar el funcionamiento del diseño antes de programar el dispositivo. Luego de generar el fichero que se cargará en el dispositivo lógico programable se configura el dispositivo y el fichero es descargado desde la PC hacia la FPGA de Xilinx.

El entorno de desarrollo ISE Xilinx posee un aspecto similar al de los entornos de programación actuales como Visual Basic o Visual C. Consta de varias herramientas que permiten entre otras cosas la generación de bancos de prueba para la verificación del diseño, la especificación de restricciones o indicaciones para realizar una implementación óptima. También contiene herramientas de programación que permiten descargar el diseño de una manera rápida sobre la FPGA. Entre las herramientas más usadas por este software se encuentran:

- *ECS* (*Engineering Capture System*), esta herramienta crea unos ficheros fuente con extensión *.sch*.
- *StateCAD* genera diagramas de estados creando ficheros con extensión *.dia*.
- *Emacs*, edita códigos fuente para diversos lenguajes de programación.
- *HDLBencher* crea un banco de prueba o Testbench para comprobar el funcionamiento del circuito.
- *ISESimulator* nos permite ver los cronogramas que generan nuestros diseños.

2.3 HDL Coder del Matlab 2011a

El HDL Coder le brinda al usuario un flujo de trabajo para la programación en FPGAs. Esta herramienta genera códigos VHDL y Verilog sintetizables e independientes.

Para generar el código HDL se requiere una serie de pasos que se encuentran relacionados entre sí. El primero es la modelación del diseño, usando una combinación de códigos de Matllab o bloques de Simulink. La ayuda sugiere una optimización del modelo para alcanzar los objetivos de diseño en cuanto a velocidad y espacio. Le sigue la generación del código HDL con el Simulink HDL Coder. Luego se desarrolla un proceso de verificación del código usando el verificador (HDL Verifier), en este proceso se genera un *testbench*

para una rápida verificación del código. También archivos de comandos para automatizar el proceso de compilación y simulación de códigos en simuladores HDL. El código HDL obtenido es optimizado para su posterior implementación en el hardware programable.

Con Simulink se pueden desarrollar algoritmos usando una librería de más de 200 bloques. El Simulink HDL Coder brinda una guía para generar códigos a partir de un Demo de Matlab, de manera que esto nos puede servir como punto de partida para generar sistemas más complejos.

2.4 Flujo de diseño según la ayuda de Matlab

El HDL Coder genera códigos VHDL y Verilog para la síntesis en FPGA u otro dispositivo lógico programable con soporte para estos lenguajes. A continuación se deriva la metodología para el trabajo con el HDL Coder, a partir de un ejemplo de la ayuda del Matlab que consiste en la generación del código de un Filtro Simétrico.

2.4.1 Generación del código VHDL usando códigos CLI (Command Line Interface)

Una vez que se tiene identificado el modelo para el cual se generará el código HDL, se debe estudiar a fondo dicho modelo para determinar si el o los subsistemas presentes en el mismo pueden generar código; es decir, si los bloques presentes, generan código HDL. Una buena herramienta para esta tarea, cuando no se es un experto en HDL Coder, está en la función *hdllib*. Dicha función genera una librería partir de la distribución de bloques de Simulink que generan código HDL, pero esto tendría más sentido cuando se trata de diseñar el modelo de forma tal que cumpla con este requisito. Por otro lado, si se tiene el modelo previamente elaborado entonces la herramienta más apropiada es la función *checkhdl*. Esta es una utilidad que comprueba un subsistema o modelo para la compatibilidad con la generación de código HDL. Si se detectan incompatibilidades como el uso de bloques no compatibles o uso ilegal de tipo de datos, *checkhdl* muestra información sobre los bloques y los problemas potenciales en un informe HTML.

Es una buena práctica definir un directorio para trabajar con el modelo y generar el código deseado. Esto puede ser hecho desde el propio Matlab con el empleo de los comandos *mkdir*, *cd*, *copyfile*, *delete*, *dir*, *fileattrib*, *movefile* y *rmdir*. Es evidente que hay que mover el espacio de trabajo hacia este directorio, a partir de entonces se podrá comprobar en la vista del *Workspace* el contenido de dicho directorio.

El modelo debe ser copiado o salvado en el directorio de trabajo. En el caso del ejemplo del FIR simétrico, la ayuda de Matlab propone una serie de pasos muy particulares para este ejemplo, pero realmente se resume a localizar el modelo dentro del espacio de trabajo. El modelo debe abrirse y mantenerse abierto por el tiempo que se ejecute el procedimiento de generación con el Simulink HDL Coder.

El próximo paso en el procedimiento general debe ser la inicialización de parámetros del modelo mediante la función *hdlsetup*. Dicha función, invocada sin argumentos, cambia los parámetros del modelo de Simulink actual (*bdroot*) a los valores que se utilizan comúnmente para la generación de código HDL. Si por el contrario se invoca *hdlsetup ("modelo")*, se cambian los parámetros del modelo especificado por el argumento *"modelo"*, a valores que se utilizan comúnmente. El comando utiliza la función *set_param* con tal de preparar los modelos para la generación de código HDL de forma rápida y consistente. Los ajustes de parámetros del modelo proporcionados por *hdlsetup* están pensados como valores útiles por defecto, pero pueden no ser apropiados para todas las aplicaciones.Para ver los parámetros afectados por el *hdlsetup* y sus particularidades, se puede abrir el archivo *hdlsetup.m* en el editor de MATLAB y se busca en la tabla *Model Parameters* en la sección *Model and Block Parameters* de Simulink.

A este punto el modelo está listo para generar el código con la función *makehdl*. Esta función sin argumentos genera el código para el modelo abierto con los valores por defecto. Si se especifica el argumento *bdroot* el resultado es exactamente el mismo. Si se especifica en los argumentos el camino y el nombre del modelo, pues será para este para el que se genere el código. Además se puede generar el código para un subsistema del modelo en Simulink, para ello hay que invocar en el argumento a dicho subsistema de la forma *nombre_del_modelo/subsistem.* Por último, se pueden especificar los pares *propiedad/valor*, de la forma:

makehdl (gcb, 'Nombre_de_la_propiedad', Valor_de_la_propiedad,...)

Entre estos pares se encuentran el lenguaje del código generado, el camino destino y la opción *checkhdl*. Esto se puede estudiar a fondo con la ayuda del Matlab.

Cuando el código es generado con éxito aparece en Matlab:

HDL Code Generation Complete.

Para abrir los archivos generados se puede hacer doble clic en los enlaces que se muestran en Matlab en letras azules.

Para manejar y verificar el funcionamiento de la entidad a la que se le generó anteriormente el código VHDL, se realiza el *testbench* o nicho de prueba. Generar un *testbench* incluye datos de estímulos, generados por fuentes señaladas conectadas a la entidad bajo la prueba y datos de salida generados por la entidad bajo la prueba. Durante una simulación del *testbench*, estos datos se comparan con las salidas VHDL del modelo, con tal de obtener información que sirva para la comprobación del funcionamiento del circuito.

Para generar el testbench en Matlab se teclea:

makehdltb ('modelo')

Las características de esta función son similares a la anteriormente expuesta. Solo que si se ejecuta antes de haber generado el código con *makehdl*, *makehdltb* genera el modelo antes de generar el nicho de pruebas. Si el lenguaje que se especificó para el *testbench* difiere del que fue utilizado para generar el código para el modelo, entonces *makehdltb* regenerará el modelo en el mismo lenguaje que el *testbench*. Si el proceso termina con éxito, se muestra el mensaje:

HDL TestBench Generation Complete.

Como prueba de la metodología usada anteriormente se puede comprobar el código VHDL en Anexo II La siguiente figura muestra el proceso completo ejecutado para un ejemplo de Matlab de un filtro simétrico. Nótese todo el proceso basado en comandos.

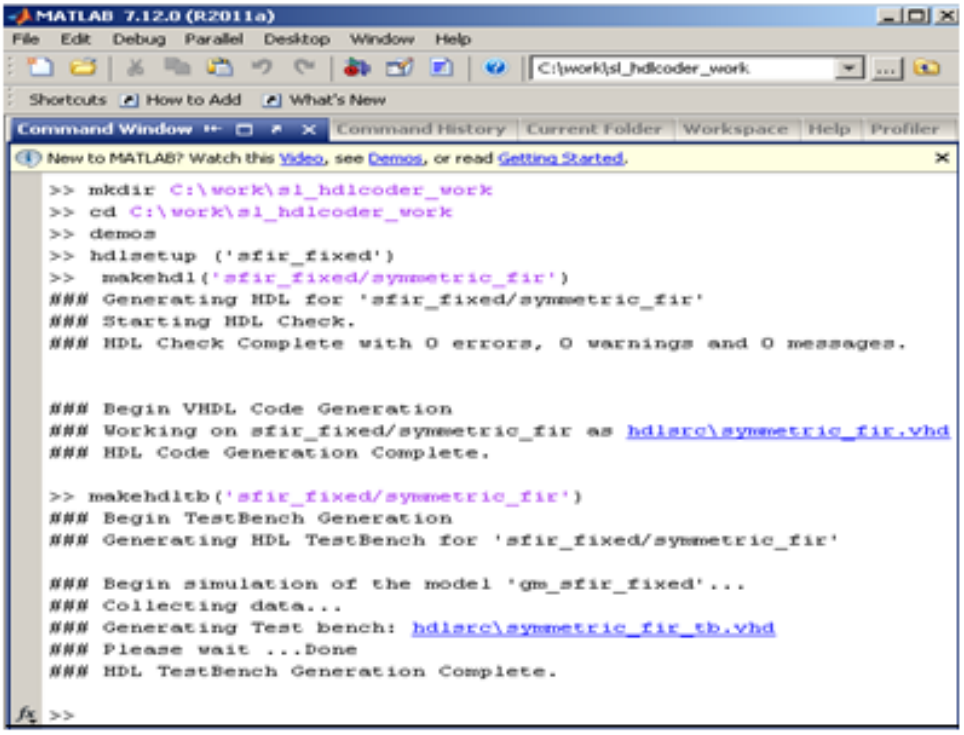


Figura 2.1 Captura de pantalla con la generación del código del modelo y de verificación para el filtro simétrico de la ayuda de Matlab. Imagen de elaboración propia.

2.4.2 Metodología para generar códigos Verilog

El comando para generar el código Verilog es:

```
makehdl ('sfir_fixed/symmetric_fir','TargetLanguage','verilog')
```

Para generar códigos Verilog la metodología es la misma hasta este paso, solo que por defecto *makehdl* genera códigos VHDL, por lo que se necesita realizar cambios en la especificación del lenguaje. Los archivos generados son los siguientes:

- symmetric_fir.v: Contiene el código Verilog.
- symmetric_fir_compile.do: Archivo para compilar el código Verilog.
- symmetric_fir_synplify.tcl: Archivo para la síntesis de la escritura.
- symmetric_fir_map.txt.: Brinda una relación de todas las entidades que forman el subsistema.

Si el código es generado con éxito Matlab muestra el mensaje:

HDL Code Generation Complete.

Se analiza el código de la misma forma que en la generación del VHDL, ya sea abriendo el archivo en la dirección de trabajo o dando clic en el enlace que Matlab propone. Luego se verifica el código generado mediante:

```
makehdltb('sfir_fixed/symmetric_fir','TargetLanguage','verilog')
```

Los archives generados son:

- symmetric_fir_tb.v: Testbench del código Verilog, prueba generada y datos de salida.
- symmetric_fir_tb_compile.do: Esta escritura compila y carga las entidades de (symmetric_fir.v) y (symmetric_fir_tb.v).
- symmetric_fir_tb_sim.do: Para inicializar el simulador y ejecutar una simulación.

2.4.3 Generación del código usando (Interfaz Gráfica del Usuario)

Para generar códigos HDL usando GUI la metodología a seguir es similar a la anterior. El cambio más significativo es que por esta vía la configuración de parámetros se realiza de forma manual. Los pasos a seguir con el mismo ejemplo del filtro simétrico son:

Los cinco primeros pasos se mantienen. La inicialización de parámetros del modelo. Con el archivo *sfir_fixed* abierto, dirigirse a *Simulation, Configuration* Parameters, en la barra de menú. Entonces se despliega la siguiente ventana:

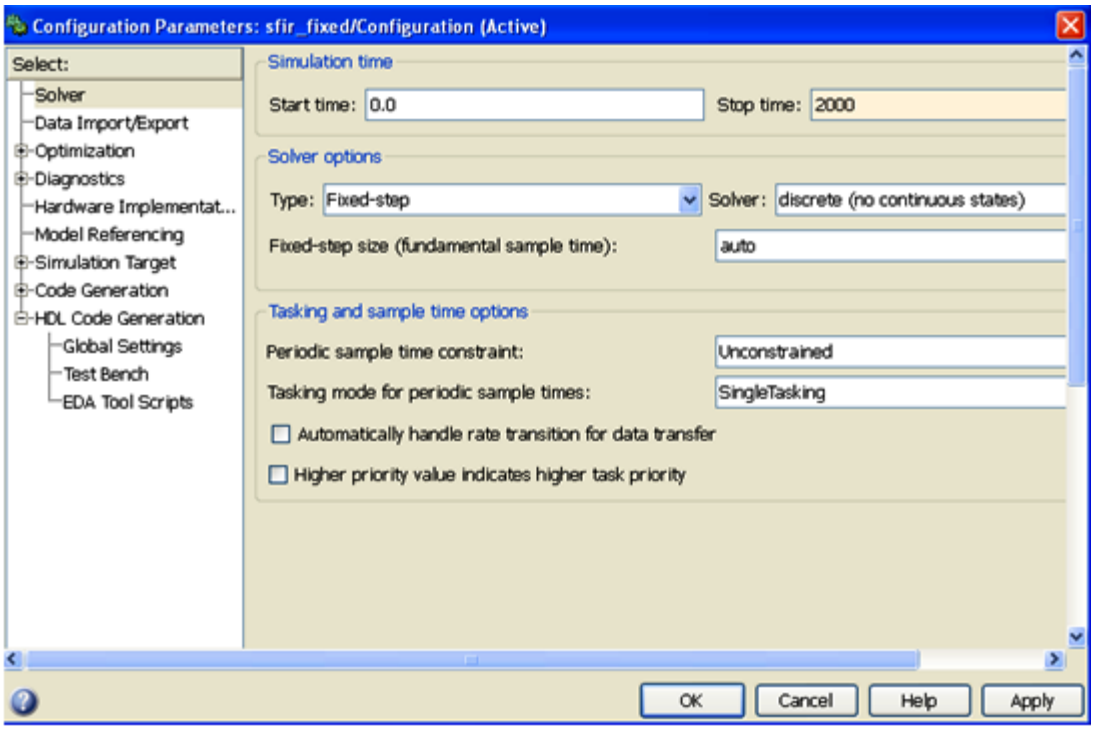


Figura 2.2 Captura de pantalla con la ventana de parámetros del HDL Coder.

Los parámetros que antes eran configurados con *hdlsetup*, ahora se hacen por defecto con la opción *Solver*, dando clic en *Apply*. A la izquierda de esta ventana se encuentra la opción *HDL Code Generation* como se muestra en la Figura 2.3.

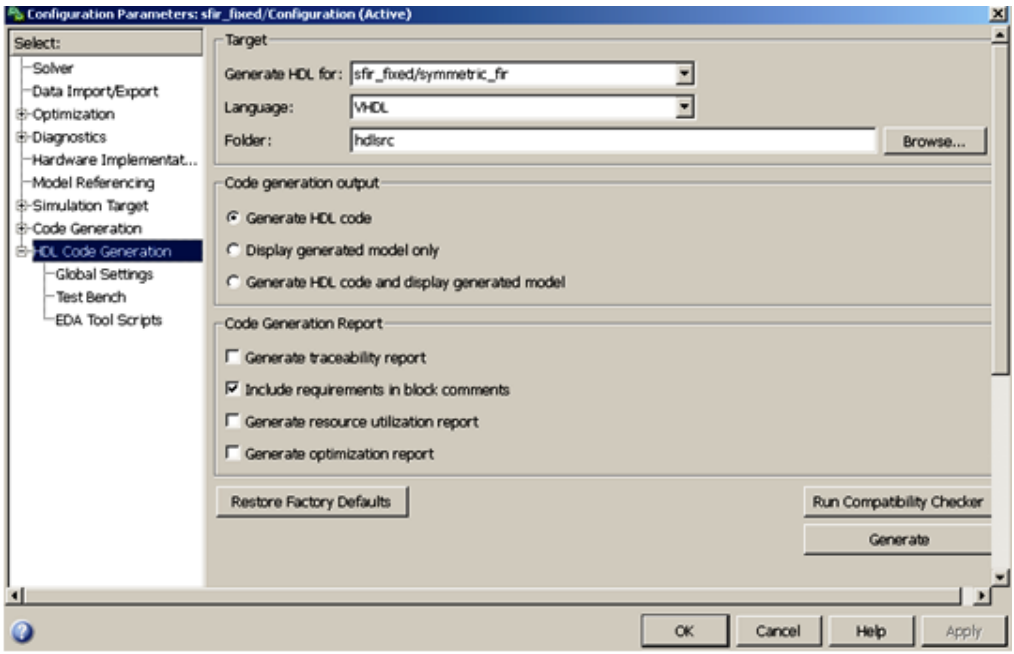


Figura 2.3 Captura para la opción *HDL Code Generation*.

El campo *Genererate HDL for* especifica el subsistema del *sfixed_fir/symmetric_fir* para la generación del código. El campo *Language* especifica el lenguaje que se quiere obtener, por defecto Matlab tiene VHDL. El campo *Folder* especifica una carpeta designada *hdlsrc*, que es una subcarpeta de la carpeta de trabajo que contiene archivos del código.

Estos tres campos están comprendidos dentro de la primera sección (*Target*) ubicada en el centro de la ventana.

Se genera el código haciendo clic en la opción *Generate* de la ventana anterior. Otra vía es seleccionar desde el menú de *sfir_fixed, Tools-HDL CodeGeneration-GenerateHDL*. Cuando el código se genera con éxitoMatlab muestra el siguiente mensaje:

```
### HDL Code Generation Complete.
```

Para verificar la compatibilidad de HDL con el subsistema, se hace clic en el botón *Run Compatibility Checker* de la ventana *Configuration Parameters*. En este caso, el subsistema seleccionado es totalmente compatible y el comprobador de compatibilidad despliega el mensaje siguiente:

```
### Starting HDL Check.

### HDL Check Complete with 0 errors, warnings and messages.
```

El comprobador de compatibilidad también despliega un informe de HTML en un navegador de Web, como es mostrado en la Figura 2.4.

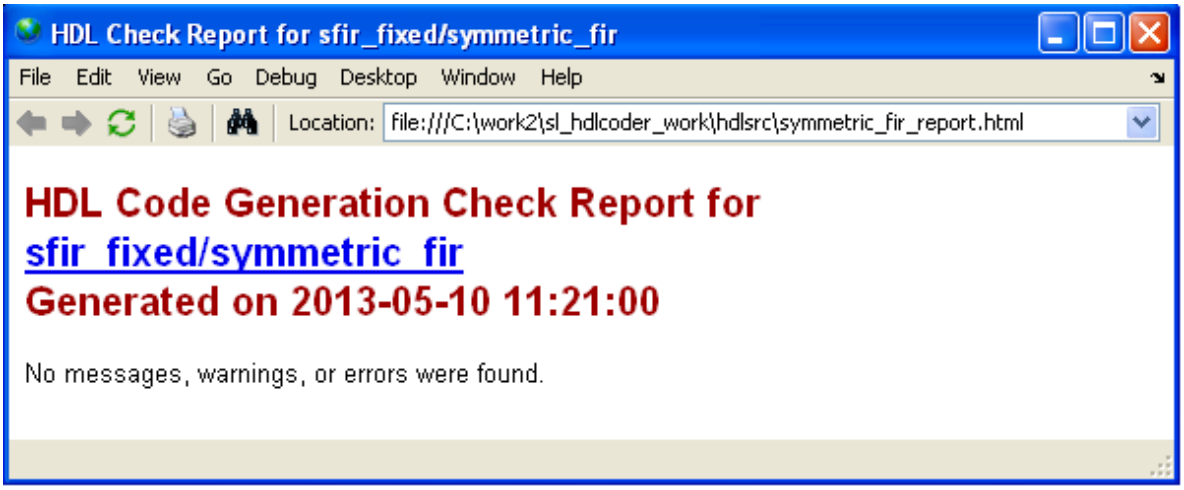


Figura 2.4 Informe HTML con el resultado del la generación del código del filtro SFIR usado en el ensayo de la metodología.

Se procede a cerrar cualquier archivo abierto en el editor para obtener el *TestBench*. Se Selecciona en *Configuration Parameters* la opción *HDL Code Generation-Testbench.* La ventana que se mostrará es similar a las anteriores y los parámetros que se encuentran en ella son cargados por defecto, si se desea hacer algún cambio, se puede hacer manualmente siempre teniendo en cuenta las características de la aplicación. La siguiente figura muestra lo expuesto anteriormente:

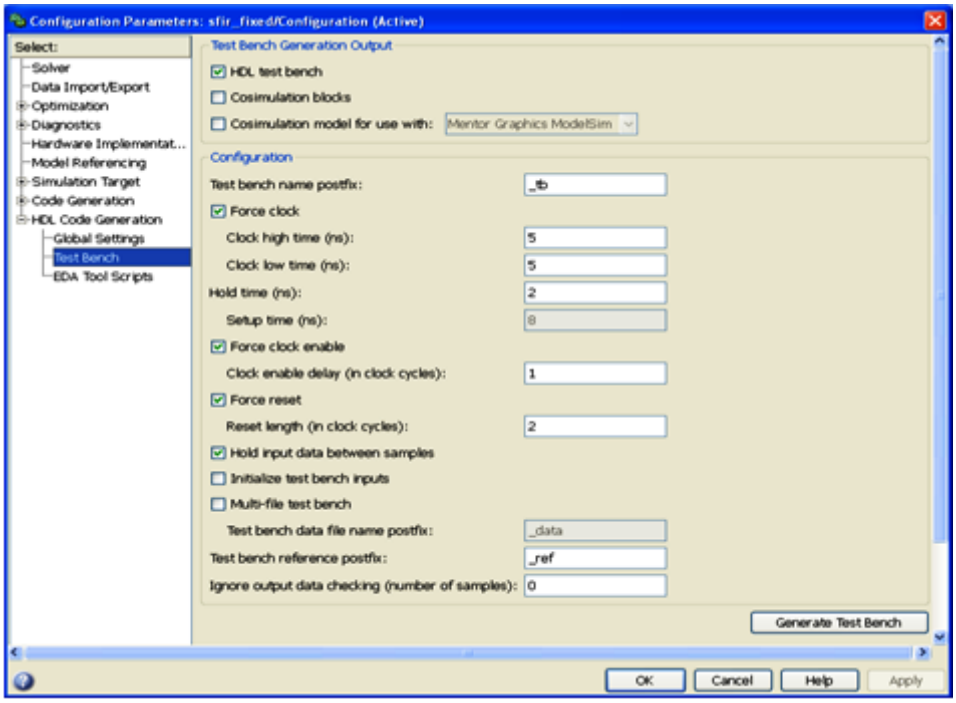


Figura 2.5 Captura de pantalla con el paso de la interfaz gráfica para la generación del *testbench*.

Después de dar clic en *Generate TestBench*, en caso de no existir errores aparece el mensaje:

```
### HDL TestBench Generation Complete.
```

Los archivos generados son almacenados en *hdlsrc*, estos son:

- *symmetric_fir_tb.vhd*: VHDL test bench, prueba generada.
- *symmetric_fir_tb_compile.do*: Esta escritura compila y carga las entidades de (symmetric_fir.vhd) y (symmetric_fir_tb.vhd).
- *symmetric_fir_tb_sim.do*: Inicializa el simulador y ejecuta una simulación.

Ahora se puede llevar el código generado y banco de la prueba a un simulador para este tipo de modelos, ejecutar la simulación y comprobarlos resultados.

2.5 Conclusiones del Capítulo

Se desarrolló una metodología de obtención de códigos VHDL y Verilog empotrables en FPGAs, a partir de diagramas en bloques en Simulink, esta generación de códigos se puede hacer mediante líneas de código o mediante Interfaz Gráfica de Usuario. Además se realiza un análisis de las principales características del Quartus II y Xilinx ISE software en los que se desarrollan los diseños digitales sintetizables en FPGAs de Altera y Xilinx respectivamente.

CAPÍTULO 3. PROPUESTA DE CONTROLADOR EMPOTRADO EN FPGA.

En este tercer y último capítulo se hace una propuesta donde mostramos la utilidad de la metodología obtenida para dar solución a un problema real consistente en el control de movimiento de plataformas neumáticas de simuladores de conducción.

Como bien es sabido en un sistema de control de lazo cerrado, el controlador o algoritmo de control no es el único elemento del sistema necesario para corregir una variable de salida a partir de la medición real y el valor deseado. Como muestra la figura 3.1, el algoritmo de control recibe el error, en este caso de posición, calculado a partir de la diferencia del valor deseado o set point y la medición real. La salida o mando calculado por el controlador, es entonces entregado al elemento encargado de acondicionar la señal, que llegará al elemento actuador y de esta forma corregir el error en posición.

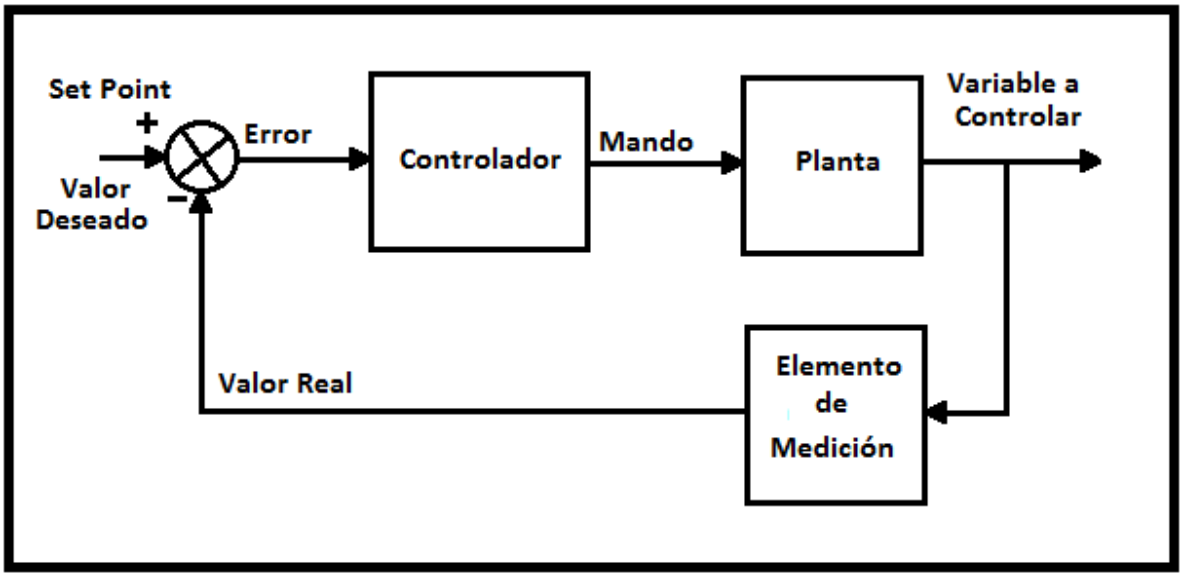


Figura 3.1 Sistema de control de lazo cerrado.

3.1 Propuesta de sistema de control empotrable en una FPGA

En el sistema que se propone hasta el momento, y siguiendo la metodología obtenida, solo se dispone del algoritmo de control y es por tanto necesario crear los restantes elementos, que junto al controlador formen un sistema de lazo cerrado empotrable en un FPGA, listo

para controlar un sistema físico real, en este caso la plataforma neumática de 2 grados de libertad producida por SIMPRO.

Los elementos que faltarían para dar solución al control de ejes de la plataforma serían:

1. Lector de Encoder de cuadratura para obtener la posición real del vástago de los cilindros neumáticos.
2. Puerto de comunicación serie RS-232 para recibir valores de set points provenientes de la PC que ejecuta el mundo virtual del simulador. Aquí solo es necesario la recepción.
3. Elemento detector de error.
4. Puerto SPI para interfaz, con Convertidor Digital-Analógico (DAC) para comando de electroválvulas. Este elemento es opcional y depende del convertidor DAC que se utilice. En nuestro caso se propone un DAC de Microchip MCP4821 disponible y usado en trabajos anteriores.

La figura 3.2 muestra el esquema general de la propuesta, donde se muestran los elementos necesarios a implementar dentro del FPGA; así como los pocos elementos necesarios que habría que conectar externamente al mismo.

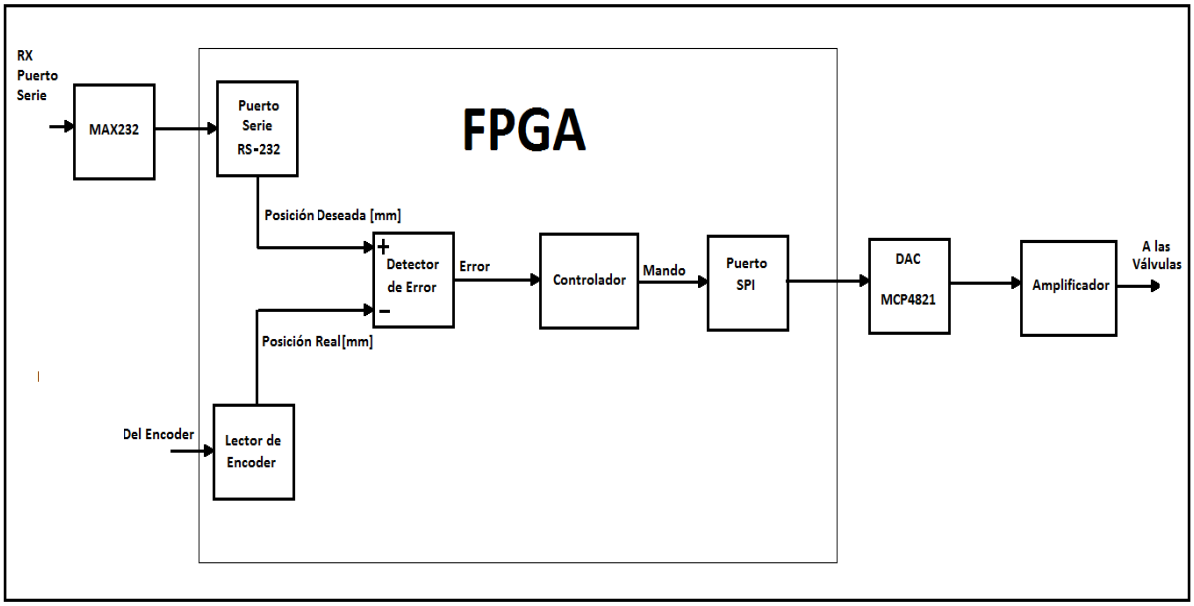


Figura 3.2 Propuesta de un sistema a implementar en una FPGA.

La figura 3.3 muestra el sistema de control desarrollado en Quartus II en el que se puede observar los diferentes bloques necesarios para cerrar el lazo de control de posición de la plataforma neumática.

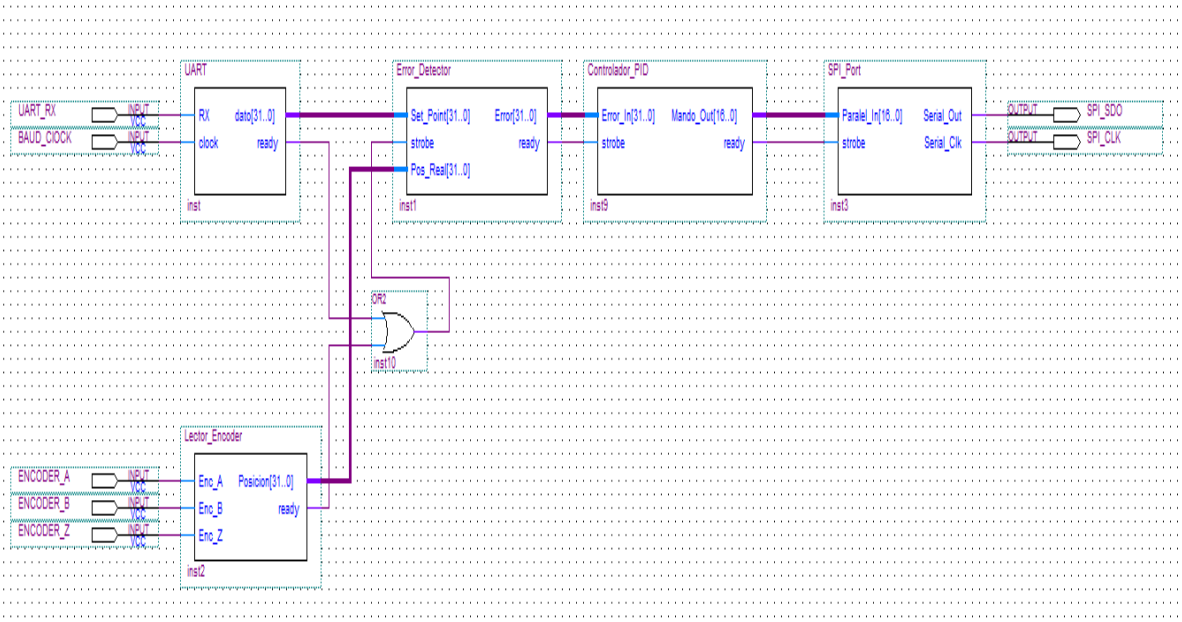


Figura 3.3 Sistema de control diseñado en el Quartus II

Como se puede observar la figura muestra los elementos necesarios, diseñados a la medida de la aplicación, haciendo un uso eficiente de los recursos para una posible expansión del sistema, donde además del control se incluya todo lo relacionado con el simulador (adquisición de datos y trucaje de la instrumentación). El esquema de la figura 3.3 es solo para un eje, pero la metodología es válida para cualquier cantidad de ejes, por ejemplo una plataforma Stewart de 6 grados de libertad. En tal caso solo habría que rediseñar el puerto serie para que de salida a la cantidad de detectores de error necesarios en función de los grados de libertad de la plataforma; así como adicionar bloques de lectura de encoder, controladores con sus ajustes por ejes y puertos SPI. La única limitación en este sentido la impondría la propia capacidad de la FPGA, lo cual hoy en día no es problema, como se comentó en el primer capítulo, son capaces de albergar sistemas realmente complejos con muy bajos costos.

3.2 Implementación del Lector de Encoder

Para dar cumplimiento a uno de los objetivos específicos de este trabajo se implementó un Lector de Encoder, siendo el mismo uno de los elementos del sistema propuesto. Resultaría interesante en trabajos posteriores, implementar todo el sistema antes mencionado dentro de una FPGA.

La figura 3.4 muestra el bloque de lectura de Encoder que fue adaptado para mostrar dicha lectura en un display de 4 dígitos de 7 segmentos presente en el Kit DE0 disponible.

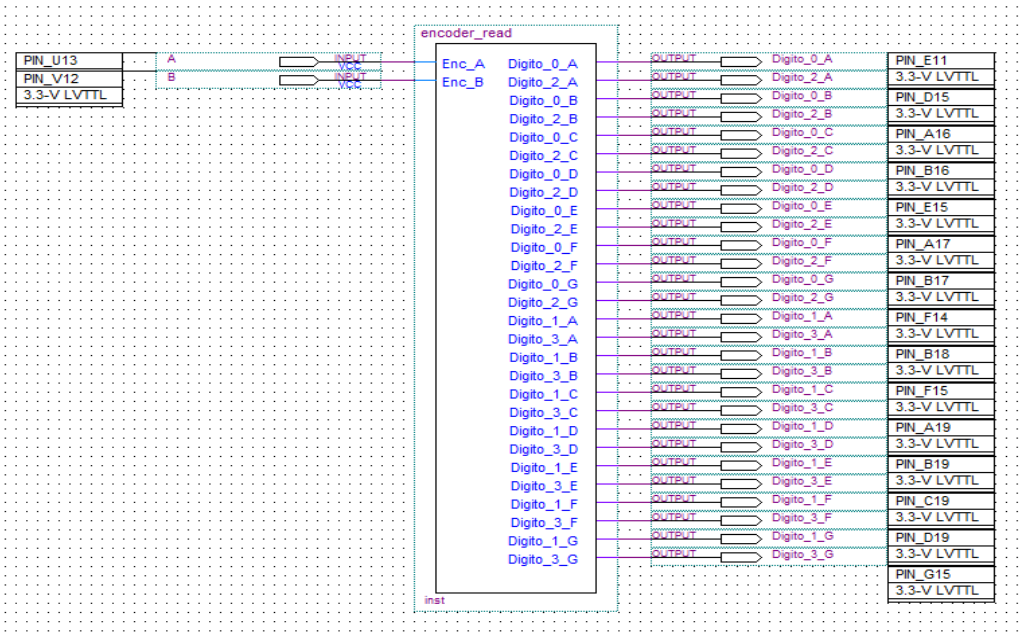


Figura 3.4 Bloque de lectura de Encoder

En el bloque mostrado en la figura 3.4, se fija las entradas A y B que están conectadas internamente a los pines U13 y V12, de uno de los puertos de expansión de la FPGA. A estas entradas se le asignarán los pulsos generados por el encoder. Las salidas de este bloque serán mostradas en los cuatro display 7 segmentos del Kit DE0, por lo que el bloque tendrá asignado 28 pines de salidas, 7 para cada display, también es necesario fijar los voltajes de salida del FPGA que pueden ser desde 1.2V hasta 3.3V, lo que muestra la gran versatilidad en este sentido de estos chips. Ver en Anexo 3 y Anexo 4 la asignación de pines de los puertos de expansión y los display 7 segmentos.

Las figuras 3.5 y 3.6 muestran los elementos internos del bloque de lectura de Encoder, todo esto fue editado, compilado y simulado en Quartus II. La figura 3.6 es una

continuación de la figura 3.5, o sea, una a continuación de la otra mostraría el esquema total.

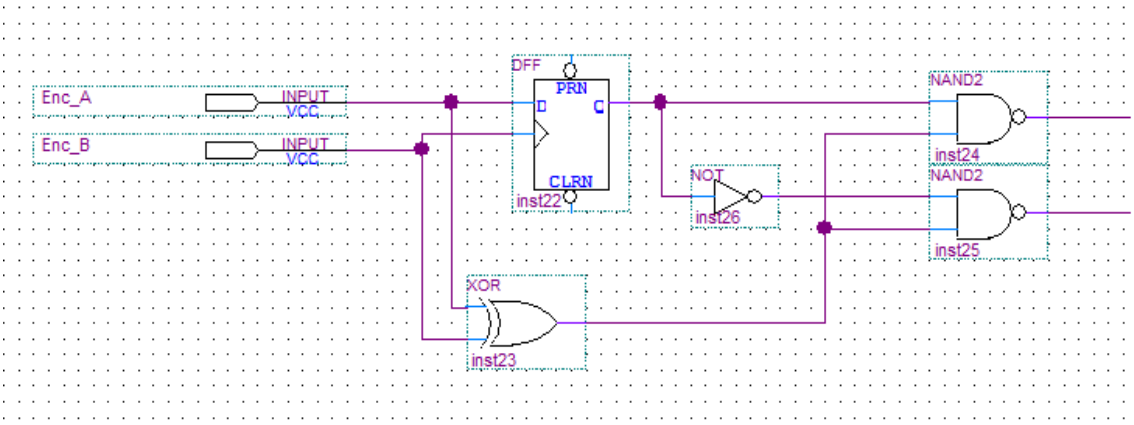


Figura 3.5 Elementos del Lector de Encoder

En la figura anterior, se muestran los elementos necesarios para obtener dos trenes de pulsos al doble de la frecuencia de los generados por el encoder, esto logra mayor resolución. Solo habrá salida en una de las compuertas NAND mientras el eje del encoder se mueve y dependerá del sentido de giro. Esto sirve para incrementar o decrementar un arreglo de contadores (figura 3.6), cuya salida será la medida de la posición referida a pulsos de encoder.

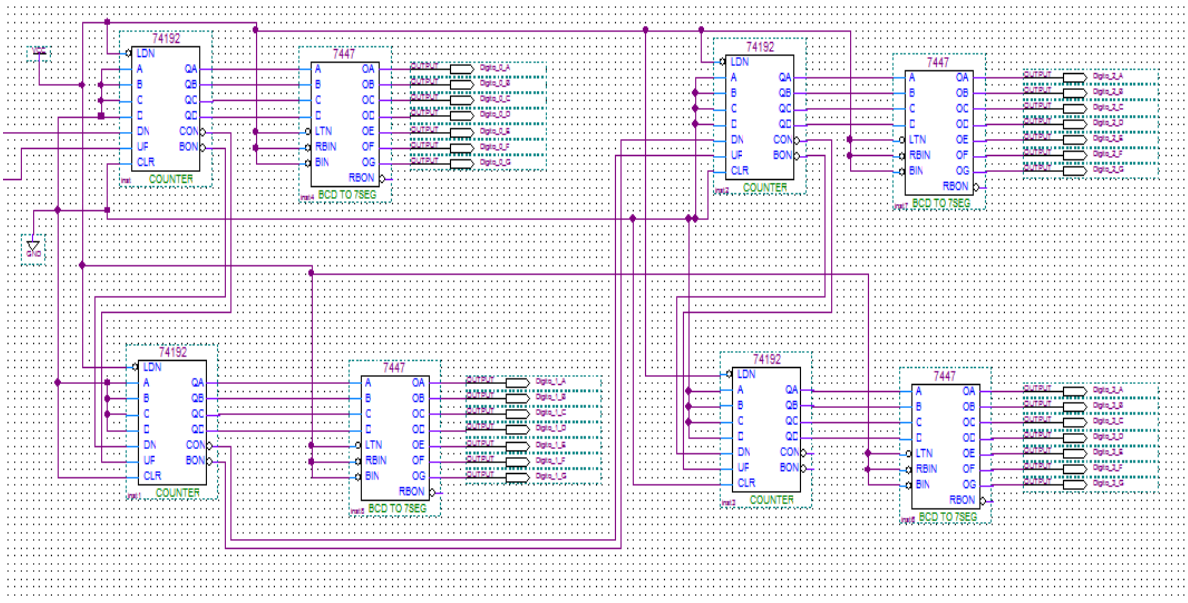


Figura 3.6 Elementos del Lector de Encoder

Este diagrama está compuesto por cuatro contadores BCD 74192 ascendentes y descendentes, predefinidos en bibliotecas de componentes del Quartus. Esta pastilla fue seleccionada para realizar el conteo que será enviado a cada uno de los displays, se utiliza un contador para cada display 7 segmentos. Además se usó cuatro conversores BCD-7 segmentos 7447. Estos son los encargados de convertir el número de cuatro bit que envía el contador a un número de 7segmentos. Cada contador le envía la señal al conversor correspondiente y este a su vez al display que se le asigne.

Para realizar el experimento se utilizó un Encoder BK 16.05A500, con frecuencia de 500 pulsos/vueltas. La figura 3.7 muestra el experimento de la lectura del Encoder totalmente funcional.

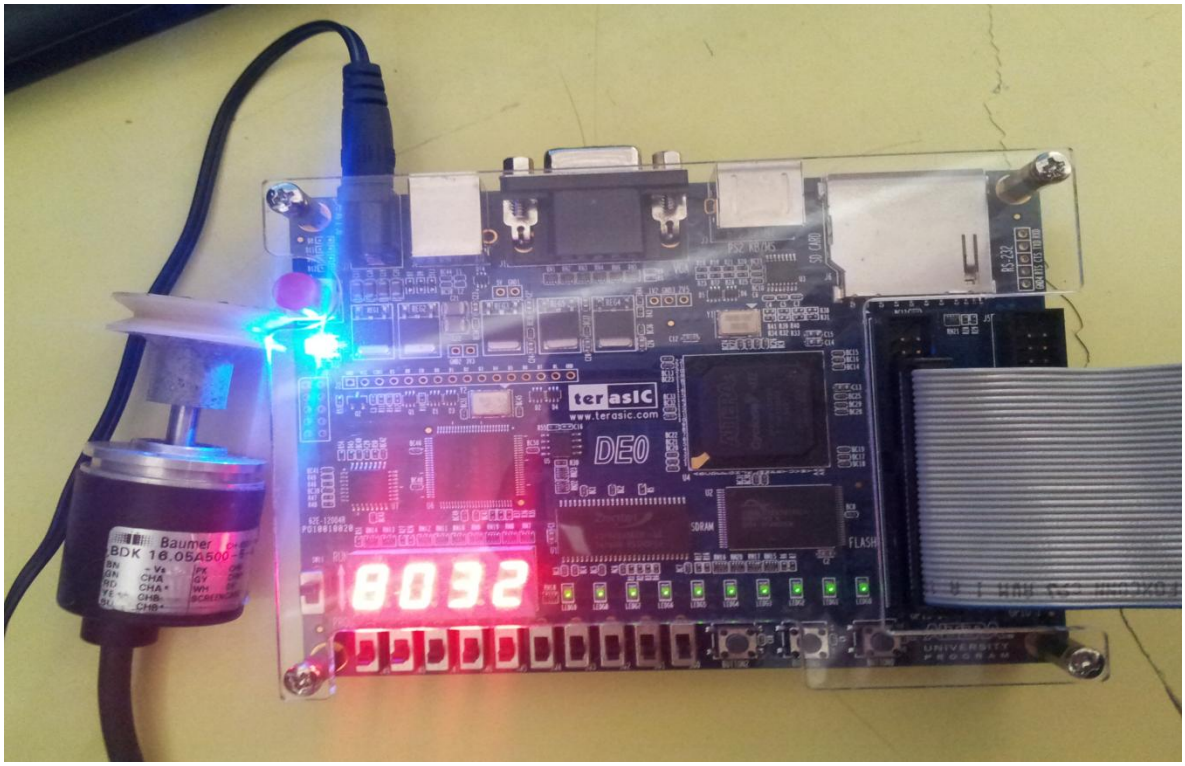


Figura 3.7 Lectura del Encoder en el Kit DEO.

Cuando se diseña con lógicas programables, independiente del método usado para diseñar el circuito (HDLs, esquemáticos, etc.), el proceso desde la definición del circuito por el desarrollador, hasta tenerlo funcionando sobre un FPGA implica varios pasos intermedios y en general utiliza una variedad de herramientas. A este proceso se le denomina ciclo o flujo

de diseño. Para cada uno de estos pasos, se utilizan herramientas de software diferentes que pueden o no estar integradas bajo un ambiente de desarrollo. En muchos casos las herramientas utilizadas en cada paso del diseño son provistas por diferentes empresas. En el caso del fabricante Altera, la herramienta que provee para los usuarios que realizan diseños con este tipo de dispositivos lógicos programables es el software Quartus II.

En la figura 3.8 se muestra una simulación con las entradas A, B y la salida del XOR. Se puede observar como por cada pulso en las entradas A y B se producen dos a la salida del XOR doblando la frecuencia de los pulsos del encoder y por consiguiente aumentado la resolución.

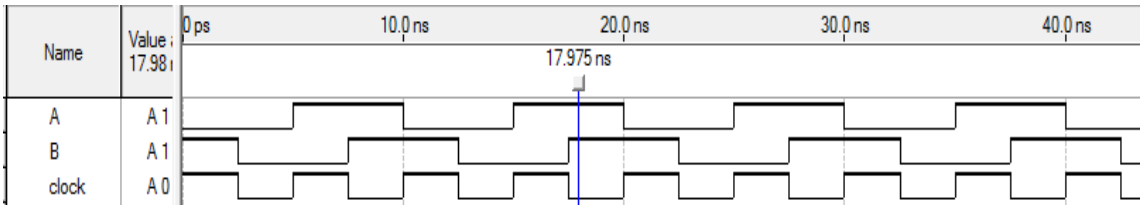


Figura 3.8 Simulación con las entradas A, B y la salida del XOR

Es importante aclarar que en la simulación del sistema que se va a implementar en la FPGA, la compuerta XOR no tiene pin de salida. Solo se le asignó un pin de salida en el Quatus para explicar mejor el funcionamiento del XOR en el circuito de la figura 3.5. La línea azul representa un instante de tiempo seleccionado y a la izquierda el valor que tomará cada entrada y salida en ese momento.

3.3 Diseño en el Software Quartus II.

Al iniciar el software se crea un nuevo proyecto, *New Proyect Wizard,* aquí es necesario especificar el nombre del proyecto y la carpeta de destino en la cual se almacenará la información. Se selecciona la familia del CPLD o la FPGA con la que se realizará el diseño, en nuestro caso usamos una FPGA de Altera de la familia Cyclone III EP3C16F484C6 presente en el Kit de desarrollo DE0, también escogemos parámetros específicos del fabricantes para una selección detallada del dispositivo como la cantidad de pines el tipo de encapsulado entre otros factores.

Quartus II brinda la posibilidad que después de creado el circuito ya sea desde un esquemático o mediante lenguajes de descripción de hardware, crear un símbolo, que

funcionaría como una caja negra con las mismas entradas y salidas del circuito creado y su misma funcionalidad. Esto permite usar dicho circuito en proyectos más grandes o utilizar varios de estos símbolos creados en un mismo proyecto sin necesidad de crearlos nuevamente. Además es una forma más amigable de ver el diseño.

Una vez que esté creado el proyecto con todos los bloques de programación, las entradas y salidas conectadas correctamente se procede a realizar la asignación de pines para definir en el encapsulado los puertos de entrada y salida de la aplicación. Cuando todos los pasos antes mencionados se realicen sin ningún error se procede con la compilación del diseño. En este paso el código VHDL se convierte en un formato que permite una simulación más rápida y eficaz. Al finalizar este paso el Quartus le ofrece al programador una información detallada de algunos resultados del diseño, entre los aspectos más importantes se encuentra la cantidad de elementos lógicos que se utilizan para desarrollar la aplicación, en este caso se usaron solo 84 elementos lógicos lo que representa menos del 1% de los elementos del dispositivo lo que nos da idea de las innumerables aplicaciones que se pueden realizar en este tipo de dispositivos lógicos programables. También se muestran la cantidad de pines que se utilizaron, en el caso del encoder se usaron solo 30 pines, 2 para las entradas del encoder y 28 para las salidas que fueron asignadas a los display 7 segmentos. Si tenemos en cuenta que el Kit DEO se fabricó fundamentalmente con propósitos de aprendizaje vale entonces preguntarnos cuantas aplicaciones de alto nivel de complejidad se pudieran desarrollar en una FPGA diseñada con fines Industriales.

Al concluir la compilación se crea un archivo .sof que es el que se cargará en la RAM interna del FPGA. El Quartus II nos permite además crear un archivo .pof que se almacena en una memoria externa no volátil cuando el sistema está listo. Esto nos permite que cuando el usuario encienda el dispositivo por defecto, este sea el programa que se ejecute sin necesidad de volver a programarlo en la FPGA.

Antes de programar el diseño en la FPGA, es necesario realizar una simulación para comprobar si el comportamiento del sistema es el deseado. La figura 3.9 muestra el resultado de dicha simulación.

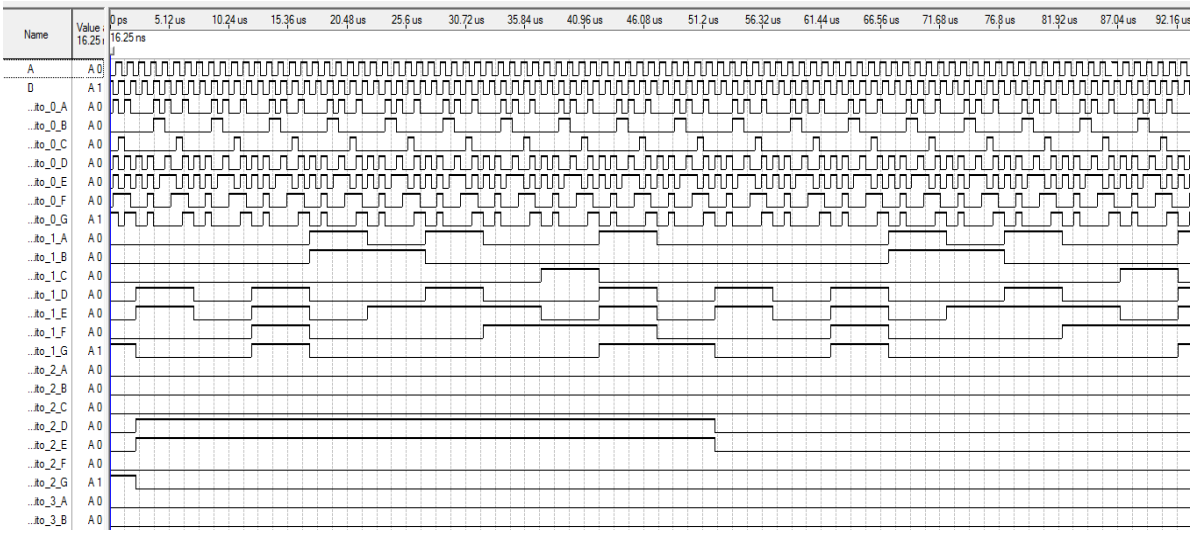


Figura 3.9. Simulación del lector de encoder.

En esta imagen podemos observar en la parte izquierda las dos entradas A y B así como las salidas del bloque. El Quartus también le brinda al usuario la posibilidad de verificar el comportamiento de las señales en distintos instantes de tiempo.

Las entradas A y B se encuentran desfasadas 90 grados una con respecto a la otra (simulando un encoder real). En el software, se representan las señales de salida desde cero hasta tres que se corresponden con los cuatro display del Kit DEO. Cada salida está conformada por 7 señales representadas desde A hasta G, a medida que transcurre el tiempo las salidas comenzado por la cero comienzan a generar pulsos que representan números de cero a nueve. Cuando cada salida llega al nueve la siguiente comienza el conteo y así respectivamente hasta llegar hasta la última. En la figura 3.9 se puede observar como las primeras salidas generan una mayor cantidad de pulsos, algo lógico teniendo en cuenta que representan el dígito menos significativo dentro del número de cuatro cifras del display.

3.4 Conclusiones del Capítulo

Se realizó una propuesta de un sistema de control empotrable en una FPGA de Altera, diseñada en Quartus II. Se implementó en el Kit DEO uno de los elementos de esta propuesta: Un Lector de Encoder, obteniendo los resultados deseados. El diseño del Encoder se realizó en Quartus, se simuló el programa obteniendo los resultados esperados y de esta forma quedó plasmado el flujo de diseño con este software.

CONCLUSIONES

Con la realización de este trabajo se arribó a las siguientes conclusiones:

1. Se arribó a una metodología que permite obtener algoritmos de control codificados en VHDL, donde fue necesario profundizar en el estudio de lenguajes de descripción de hardware, de las potencialidades del HDL Coder de Matlab y sus variantes de generación de código
2. .El algoritmo de control obtenido, unido a los demás elementos del esquema de control propuesto, constituyen un sistema completo totalmente funcional y empotrable en FPGAs o CPLDs.
3. Se implementó en un Kit de desarrollo un elemento de medición, que forma parte del sistema de solución propuesto, se demostró la validez de dicho sistema y se obtuvo los resultados esperados en la medición.

RECOMENDACIONES

Para establecer la necesaria continuidad que debe tener este trabajo se recomienda lo siguiente:

1. Implementar totalmente la solución propuesta, así como aplicar la metodología obtenida a nuevos y novedosos algoritmos de control en desarrollo actualmente.
2. Emplear las tecnologías de FPGAs y CPLDs en la electrónica asociada a la adquisición de datos y trucaje del simulador en conjunto con el sistema de control de movimiento, lo que reduciría la cantidad de componentes y el tiempo de desarrollo y puesta en el mercado del mismo haciéndolo mas compacto, fiable, robusto y económico, mejorando la respuesta del sistema en su conjunto.

REFERENCIAS BIBLIOGRÁFICAS

ALTERA 2003. *Configuration Handbook*.

ALTERA. 2013. *FPGA and CPLD Design Flow. Compile Your First FPGA or CPLD Design Using Quartus II Software* [Online]. ALTERA Corporation. Available: www.altera.com/products/software/flows/fpga/flo-fpga.html [Accessed 13-3-2013.

BOZICH, E. C. 2005. *Introducción a los dispositivos FPGA. Análisis y ejemplos de diseño.* Departamento de Electrónica, Universidad Nacional de La Plata.

BROWN, S. & ROSE, J. 2010. *Unit 1.11 CPLD or FPGA devices. Design flow.* [Online]. Universitat Politècnica de Catalunya. Available: http://digsys.upc.es/ed/CSD/units/Ch1/U1_11/Unit1_11.html [Accessed 13-3-2013.

CZAJKOWSKI, T. 2010. Altera_University_Program_Advanced_Course.

CHARLES H. ROTH, J. & KINNEY., L. L. 2010. *Fundamentals of Logic Design*.

CHU, P. P. 2006. *RTL HARDWARE DESIGN USING VHDL*, Canada, John Wiley & Sons, Inc., Hoboken, New Jersey.

ELHOSSINI, A. 2013. Using Xilinx System Generator.

MAZA, Y. E. A. 2008. *Síntesis de Circuitos Digitales Utilizando VHDL y FPGAs.* Escuela Polítecnica Nacional.

MORFA, E. E. C. 2011. *Controlador Empotrado para Plataformas Neumáticas de Dos Grados de Libertad empleando Arquitectura ARM9 y Sistema Operativo Linux.* UNIVERSIDAD CENTRAL "MARTA ABREU" DE LAS VILLAS.

OLIVER, J. P. 2007. *Diseño Digital Utilizando Lógica Programable: Aplicaciones a la Enseñanza.* Instituto de Ingeniería Eléctrica, Facultad de Ingeniería Universidad de la República, Montevideo, Uruguay

RODRÍGUEZ, A. E. R. 2008. *Modelación, identificación y control de actuadores electro-neumáticos para aplicaciones industriales.* Tesis de Doctorado, Universidad Central "Marta Abreu" de Las Villas.

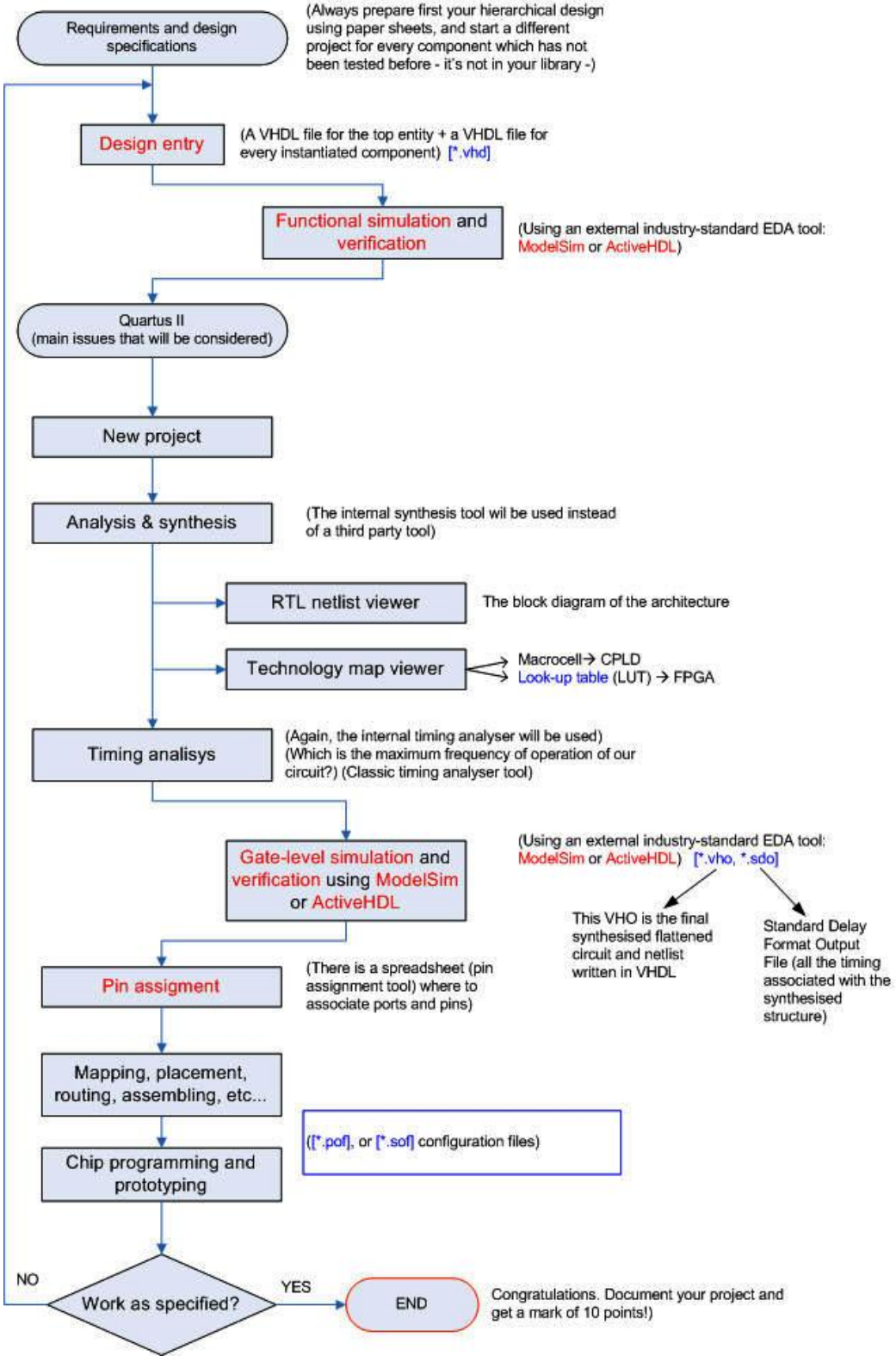
SOSA, A. M. 2007. *CONTROLADOR EMPOTRADO PARA PLATAFORMA NEUMÁTICA DE SIMULADOR DE CONDUCCIÓN.* Tesis de Maestría, UNIVERSIDAD CENTRAL "MARTA ABREU" DE LAS VILLAS.

TERASIC 2009. DE0 User Manual.

ZAMORA, R. O. A. 2010. *Metodología para el diseño de aplicaciones medianas en FPGAs de Xilinx.* Tesis de grado, Departamento de Telecomunicaciones y Electrónica.

ANEXOS

Anexo I Flujo de diseño con Altera (detallado)



Anexo II VHDL de un Filtro Simétrico generado desde Matlab-Simulink

```
File Name: hdlsrc\symmetric_fir.vhd
-- Created: 2013-05-14 11:26:43
--
-- Generated by MATLAB 7.12 and Simulink HDL Coder 2.1
--
--
-- -------------------------------------------------------------
-- Rate and Clocking Details
-- -------------------------------------------------------------
-- Model base rate: 1
-- Target subsystem base rate: 1
--
--
-- Clock Enable Sample Time
-- -------------------------------------------------------------
-- ce_out    1
-- -------------------------------------------------------------
--
--
-- Output Signal         Clock Enable Sample Time
-- -------------------------------------------------------------
-- y_outce_out    1
-- delayed_x_outce_out    1
-- -------------------------------------------------------------
--
```

```
-- -------------------------------------------------------------



-- -------------------------------------------------------------
--
-- Module: symmetric_fir
-- Source Path: sfir_fixed/symmetric_fir
-- Hierarchy Level: 0
--
-- Simulink model description for sfir_fixed:
--
-- Symmetric FIR Filter
-- This model shows how to use Simulink(R) HDL Coder(TM) to check,
generate,
-- and verify HDL for a fixed-point symmetric FIR filter.
--
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;


ENTITY symmetric_fir IS
 PORT( clk                :  IN  std_logic;
    reset              :  IN  std_logic;
clk_enable           :  IN  std_logic;
x_in               :  IN  std_logic_vector(15 DOWNTO 0); --
sfix16_En10
```

```
    h_in1               :  IN  std_logic_vector(15 DOWNTO 0); --
sfix16_En10
    h_In2               :  IN  std_logic_vector(15 DOWNTO 0); --
sfix16_En10
    h_In3               :  IN  std_logic_vector(15 DOWNTO 0); --
sfix16_En10
    h_In4               :  IN  std_logic_vector(15 DOWNTO 0); --
sfix16_En10
ce_out              :  OUT  std_logic;
y_out               :  OUT  std_logic_vector(34 DOWNTO 0); --
sfix35_En20
delayed_x_out           :  OUT  std_logic_vector(15 DOWNTO 0) --
sfix16_En10
    );
END symmetric_fir;


ARCHITECTURE rtl OF symmetric_fir IS

 -- Signals
 SIGNAL enb               : std_logic;
 SIGNAL x_in_signed           : signed(15 DOWNTO 0); --
sfix16_En10
 SIGNAL Unit_Delay1_out1         : signed(15 DOWNTO 0); --
sfix16_En10
 SIGNAL Unit_Delay_out1          : signed(15 DOWNTO 0); --
sfix16_En10
 SIGNAL Unit_Delay2_out1         : signed(15 DOWNTO 0); --
sfix16_En10
```

```
 SIGNAL Unit_Delay3_out1         : signed(15 DOWNTO 0); --
sfix16_En10

 SIGNAL Unit_Delay4_out1         : signed(15 DOWNTO 0); --
sfix16_En10

 SIGNAL Unit_Delay5_out1         : signed(15 DOWNTO 0); --
sfix16_En10

 SIGNAL Unit_Delay6_out1         : signed(15 DOWNTO 0); --
sfix16_En10

 SIGNAL Unit_Delay7_out1         : signed(15 DOWNTO 0); --
sfix16_En10

 SIGNAL Add_add_cast           : signed(16 DOWNTO 0); --
sfix17_En10

 SIGNAL Add_add_cast_1          : signed(16 DOWNTO 0); --
sfix17_En10

 SIGNAL Add_out1             : signed(16 DOWNTO 0); -- sfix17_En10

 SIGNAL Add1_add_cast          : signed(16 DOWNTO 0); --
sfix17_En10

 SIGNAL Add1_add_cast_1         : signed(16 DOWNTO 0); --
sfix17_En10

 SIGNAL Add1_out1            : signed(16 DOWNTO 0); -- sfix17_En10

 SIGNAL Add2_add_cast          : signed(16 DOWNTO 0); --
sfix17_En10

 SIGNAL Add2_add_cast_1         : signed(16 DOWNTO 0); --
sfix17_En10

 SIGNAL Add2_out1            : signed(16 DOWNTO 0); -- sfix17_En10

 SIGNAL Add3_add_cast          : signed(16 DOWNTO 0); --
sfix17_En10

 SIGNAL Add3_add_cast_1         : signed(16 DOWNTO 0); --
sfix17_En10
```

```
 SIGNAL Add3_out1           : signed(16 DOWNTO 0); -- sfix17_En10
 SIGNAL h_in1_signed          : signed(15 DOWNTO 0); --
sfix16_En10
 SIGNAL Product_out1          : signed(32 DOWNTO 0); --
sfix33_En20
 SIGNAL h_In2_signed          : signed(15 DOWNTO 0); --
sfix16_En10
 SIGNAL Product1_out1         : signed(32 DOWNTO 0); --
sfix33_En20
 SIGNAL Add5_add_cast         : signed(33 DOWNTO 0); --
sfix34_En20
 SIGNAL Add5_add_cast_1        : signed(33 DOWNTO 0); --
sfix34_En20
 SIGNAL Add5_out1           : signed(33 DOWNTO 0); -- sfix34_En20
 SIGNAL h_In3_signed          : signed(15 DOWNTO 0); --
sfix16_En10
 SIGNAL Product2_out1         : signed(32 DOWNTO 0); --
sfix33_En20
 SIGNAL h_In4_signed          : signed(15 DOWNTO 0); --
sfix16_En10
 SIGNAL Product3_out1         : signed(32 DOWNTO 0); --
sfix33_En20
 SIGNAL Add6_add_cast         : signed(33 DOWNTO 0); --
sfix34_En20
 SIGNAL Add6_add_cast_1        : signed(33 DOWNTO 0); --
sfix34_En20
 SIGNAL Add6_out1           : signed(33 DOWNTO 0); -- sfix34_En20
 SIGNAL Add4_add_cast         : signed(34 DOWNTO 0); --
sfix35_En20
```

```
 SIGNAL Add4_add_cast_1         : signed(34 DOWNTO 0); --
sfix35_En20
 SIGNAL Add4_out1            : signed(34 DOWNTO 0); -- sfix35_En20


BEGIN
x_in_signed<= signed(x_in);


enb<= clk_enable;


 Unit_Delay1_process : PROCESS (clk, reset)
 BEGIN
  IF reset = '1' THEN
   Unit_Delay1_out1 <= to_signed(0, 16);
  ELSIF clk'EVENT AND clk = '1' THEN
   IF enb = '1' THEN
    Unit_Delay1_out1 <= x_in_signed;
   END IF;
  END IF;
 END PROCESS Unit_Delay1_process;



Unit_Delay_process : PROCESS (clk, reset)
 BEGIN
  IF reset = '1' THEN
   Unit_Delay_out1 <= to_signed(0, 16);
  ELSIF clk'EVENT AND clk = '1' THEN
   IF enb = '1' THEN
```

```
    Unit_Delay_out1 <= Unit_Delay1_out1;
   END IF;
 END IF;
END PROCESS Unit_Delay_process;



Unit_Delay2_process : PROCESS (clk, reset)
BEGIN
 IF reset = '1' THEN
  Unit_Delay2_out1 <= to_signed(0, 16);
 ELSIF clk'EVENT AND clk = '1' THEN
  IF enb = '1' THEN
   Unit_Delay2_out1 <= Unit_Delay_out1;
  END IF;
 END IF;
END PROCESS Unit_Delay2_process;



Unit_Delay3_process : PROCESS (clk, reset)
BEGIN
 IF reset = '1' THEN
  Unit_Delay3_out1 <= to_signed(0, 16);
 ELSIF clk'EVENT AND clk = '1' THEN
  IF enb = '1' THEN
   Unit_Delay3_out1 <= Unit_Delay2_out1;
  END IF;
```

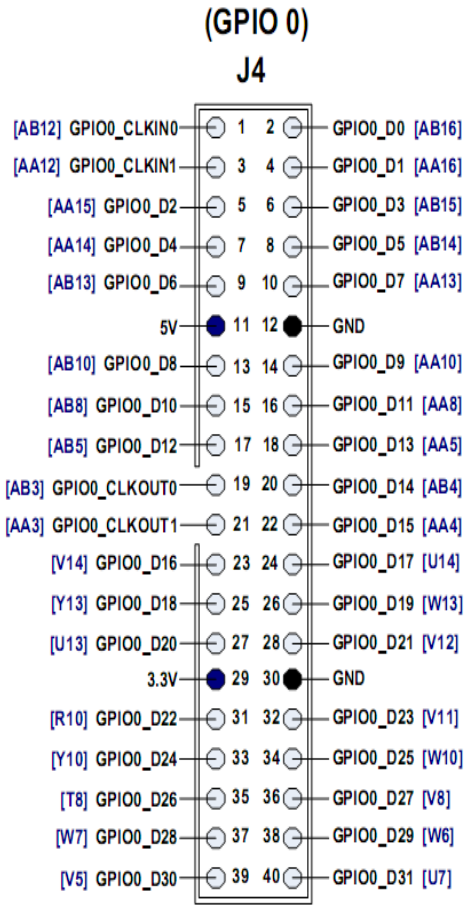
```
 END IF;
END PROCESS Unit_Delay3_process;



Unit_Delay4_process : PROCESS (clk, reset)
BEGIN
 IF reset = '1' THEN
  Unit_Delay4_out1 <= to_signed(0, 16);
 ELSIF clk'EVENT AND clk = '1' THEN
  IF enb = '1' THEN
   Unit_Delay4_out1 <= Unit_Delay3_out1;
  END IF;
 END IF;
END PROCESS Unit_Delay4_process;



Unit_Delay5_process : PROCESS (clk, reset)
BEGIN
 IF reset = '1' THEN
  Unit_Delay5_out1 <= to_signed(0, 16);
 ELSIF clk'EVENT AND clk = '1' THEN
  IF enb = '1' THEN
   Unit_Delay5_out1 <= Unit_Delay4_out1;
  END IF;
 END IF;
END PROCESS Unit_Delay5_process;
```

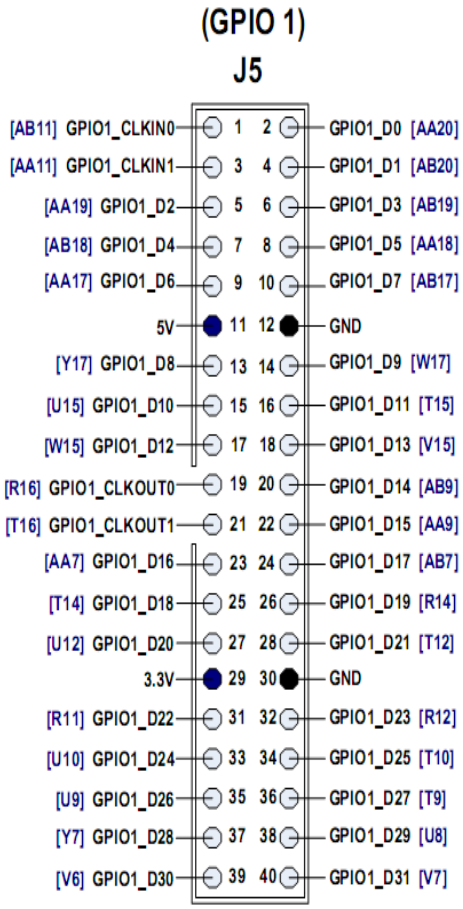
```
Unit_Delay6_process : PROCESS (clk, reset)
BEGIN
 IF reset = '1' THEN
  Unit_Delay6_out1 <= to_signed(0, 16);
 ELSIF clk'EVENT AND clk = '1' THEN
  IF enb = '1' THEN
   Unit_Delay6_out1 <= Unit_Delay5_out1;
  END IF;
 END IF;
END PROCESS Unit_Delay6_process;


Unit_Delay7_process : PROCESS (clk, reset)
BEGIN
 IF reset = '1' THEN
  Unit_Delay7_out1 <= to_signed(0, 16);
 ELSIF clk'EVENT AND clk = '1' THEN
  IF enb = '1' THEN
   Unit_Delay7_out1 <= Unit_Delay6_out1;
  END IF;
 END IF;
END PROCESS Unit_Delay7_process;
```

```
Add_add_cast<= resize(Unit_Delay7_out1, 17);
 Add_add_cast_1 <= resize(Unit_Delay1_out1, 17);
 Add_out1 <= Add_add_cast + Add_add_cast_1;

 Add1_add_cast <= resize(Unit_Delay6_out1, 17);
 Add1_add_cast_1 <= resize(Unit_Delay_out1, 17);
 Add1_out1 <= Add1_add_cast + Add1_add_cast_1;

 Add2_add_cast <= resize(Unit_Delay5_out1, 17);
 Add2_add_cast_1 <= resize(Unit_Delay2_out1, 17);
 Add2_out1 <= Add2_add_cast + Add2_add_cast_1;

 Add3_add_cast <= resize(Unit_Delay4_out1, 17);
 Add3_add_cast_1 <= resize(Unit_Delay3_out1, 17);
 Add3_out1 <= Add3_add_cast + Add3_add_cast_1;

 h_in1_signed <= signed(h_in1);

 Product_out1 <= Add_out1 * h_in1_signed;

 h_In2_signed <= signed(h_In2);

 Product1_out1 <= Add1_out1 * h_In2_signed;

 Add5_add_cast <= resize(Product_out1, 34);
 Add5_add_cast_1 <= resize(Product1_out1, 34);
```

```
 Add5_out1 <= Add5_add_cast + Add5_add_cast_1;


 h_In3_signed <= signed(h_In3);


 Product2_out1 <= Add2_out1 * h_In3_signed;


 h_In4_signed <= signed(h_In4);


 Product3_out1 <= Add3_out1 * h_In4_signed;


 Add6_add_cast <= resize(Product2_out1, 34);
 Add6_add_cast_1 <= resize(Product3_out1, 34);
 Add6_out1 <= Add6_add_cast + Add6_add_cast_1;


 Add4_add_cast <= resize(Add5_out1, 35);
 Add4_add_cast_1 <= resize(Add6_out1, 35);
 Add4_out1 <= Add4_add_cast + Add4_add_cast_1;


y_out<= std_logic_vector(Add4_out1);


delayed_x_out<= std_logic_vector(Unit_Delay7_out1);


ce_out<= clk_enable;


END rtl;
```

Anexo III Asignación de pines de los dos puertos de expansión del Kit DEO de Terasic.





Anexo IV Asignación de pines de los display 7 segmentos del Kit DEO.

Signal Name	FPGA Pin No.	Description
HEX0_D[0]	PIN_E11	Seven Segment Digit 0[0]
HEX0_D[1]	PIN_F11	Seven Segment Digit 0[1]
HEX0_D[2]	PIN_H12	Seven Segment Digit 0[2]
HEX0_D[3]	PIN_H13	Seven Segment Digit 0[3]
HEX0_D[4]	PIN_G12	Seven Segment Digit 0[4]
HEX0_D[5]	PIN_F12	Seven Segment Digit 0[5]
HEX0_D[6]	PIN_F13	Seven Segment Digit 0[6]
HEX0_DP	PIN_D13	Seven Segment Decimal Point 0
HEX1_D[0]	PIN_A13	Seven Segment Digit 1[0]
HEX1_D[1]	PIN_B13	Seven Segment Digit 1[1]
HEX1_D[2]	PIN_C13	Seven Segment Digit 1[2]
HEX1_D[3]	PIN_A14	Seven Segment Digit 1[3]
HEX1_D[4]	PIN_B14	Seven Segment Digit 1[4]
HEX1_D[5]	PIN_E14	Seven Segment Digit 1[5]
HEX1_D[6]	PIN_A15	Seven Segment Digit 1[6]
HEX1_DP	PIN_B15	Seven Segment Decimal Point 1

HEX2_D[0]	PIN_D15	Seven Segment Digit 2[0]
HEX2_D[1]	PIN_A16	Seven Segment Digit 2[1]
HEX2_D[2]	PIN_B16	Seven Segment Digit 2[2]
HEX2_D[3]	PIN_E15	Seven Segment Digit 2[3]
HEX2_D[4]	PIN_A17	Seven Segment Digit 2[4]
HEX2_D[5]	PIN_B17	Seven Segment Digit 2[5]
HEX2_D[6]	PIN_F14	Seven Segment Digit 2[6]
HEX2_DP	PIN_A18	Seven Segment Decimal Point 2
HEX3_D[0]	PIN_B18	Seven Segment Digit 3[0]
HEX3_D[1]	PIN_F15	Seven Segment Digit 3[1]
HEX3_D[2]	PIN_A19	Seven Segment Digit 3[2]
HEX3_D[3]	PIN_B19	Seven Segment Digit 3[3]
HEX3_D[4]	PIN_C19	Seven Segment Digit 3[4]
HEX3_D[5]	PIN_D19	Seven Segment Digit 3[5]
HEX3_D[6]	PIN_G15	Seven Segment Digit 3[6]
HEX3_DP	PIN_G16	Seven Segment Decimal Point 3