UNIVERSIDAD CATÓLICA DE SANTIAGO DE GUAYAQUIL

TESIS DE GRADO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO(A) EN TELECOMUNICACIONES CON MENCIÓN EN GESTIÓN EMPRESARIAL

TEMA:

“Diseño e implementación de prácticas con FPGA para la materia de Digitales II mediante la herramienta de trabajo Cyclone II de Altera”

ALUMNOS:

Carlos Benalcazar
Iván Andrade

DIRECTOR

MSC. LUIS CORDOVA RIVADENEIRA

TESIS DE GRADO

DISEÑO E IMPLEMENTACIÓN DE PRÁCTICAS EN FPGA PARA DIGITALES II MEDIANTE LA HERRAMIENTA DE TRABAJO CYCLONE II DE ALTERA.

Presentada a la Facultad de Educación Técnica para el Desarrollo, Carrera de Ingeniería en Telecomunicaciones de la Universidad Católica de Santiago de Guayaquil

REALIZADO POR:

Carlos Benalcazar

Iván Andrade

Para dar cumplimiento con uno de los requisitos para optar por el titulo de:
Ingeniero en Telecomunicaciones con Mención en Gestión Empresarial

MsC. Luis Córdova Rivadeneira
Director de Tesis

Vocal
MsC. Manuel Romero Paz
Decano de la Facultad

Vocal
MsC. Luis Córdova Rivadeneira
Director de Carrera

Vocal
MsC. Edwin Palacios Meléndez

Vocal
MsC. Luzmila Ruilova

CERTIFICACIÓN

Certifico que el proyecto de grado titulado **DISEÑO E IMPLEMENTACIÓN DE PRÁCTICAS EN FPGA PARA DIGITALES II MEDIANTE LA HERRAMIENTA DE TRABAJO CYCLONE II DE ALTERA** desarrollado por Carlos Benalcázar e Iván Andrade fue realizado, corregido y terminado, razón por la cual está apto para su presentación y sustentación.

Carlos Benálcazar
Iván Andrade

MsC. Luis Córdova Rivadeneira
DIRECTOR DE TESIS

Índice General

Índice de Figuras

Capítulo 2

Capítulo 3

Capítulo 4

Agradecimientos

A nuestras familias por su apoyo incondicional y comprensión que sin ellos no hubiera sido posible este trabajo.

A nuestro director de tesis MsC. Luis Córdova Rivadeneira por su constante apoyo, confianza e innumerables contribuciones.

Al profesor MsC. Fernando Palacios por sus valiosos aportes.

A los revisores metodológicos por su colaboración, observaciones y aportes.

Al MsC. Manuel Romero Paz, Decano de la Facultad de Educación para el Desarrollo, por su aportes en las materias impartidas en la carrera de Ingeniería en Telecomunicaciones.

Dedicatoria

A nuestros padres por su entrega abnegada y sublime siendo pilares fundamentales para culminar nuestra vida universitaria, a nuestros hermanos y amigos por su apoyo incondicional ayudándonos con sus experiencias y consejos para así cumplir nuestras metas, convirtiéndonos en profesionales de éxito y mejores personas.

Resumen

La presente tesis describe las ventajas de utilizar la programación VHDL a través de la tarjeta Cyclone II de Altera, para lo cual se evidencio el potencial del sistema de entrenamiento a nivel universitario para el cual fue construido por la empresa ALTERA (http://www.altera.com/). Asimismo la plataforma de programación Quartus II o MaxPlus II pertenecientes a Altera, que nos permitieron trabajar y a la vez investigar en una herramienta nunca antes estudiada en la Carrera de Ingeniería en Telecomunicaciones y Electrónica en Control y Automatismo.

Para el diseño de cada una de las prácticas experimentales basto en analizar el programa de estudios de Sistemas Digitales II en la cual se basa en el diseño secuencial lo cual permite a los alumnos adquirir las competencias relativas a electrónica dentro de los estudios de ofrece la Facultad Técnica para el Desarrollo.

Se utiliza la metodología Project Based Learning (PBL) para proponer a los alumnos de pregrado pequeños proyectos donde experimenten la problemática habitual del desarrollo de máquinas secuenciales. De hecho, como se verá la plataforma de programación QUARTUS II proporciona un entorno de trabajo muy rico con el que además de trabajar las competencias específicas relativas a Electrónica Digital y programación resultando valiosas para trabajar competencias de otras materias como Sistemas Digitales I, Fundamentos de Digitales, Laboratorio de Digitales, Microcontroladores, Microprocesadores, Diseño Electrónico Digital.

Abstract

This thesis describes the advantages of using VHDL programming through theAltera Cyclone II card, for which evidenced the potential of the university-level training for which it was built by the company ALTERA (http://www.altera.com/). Likewise, the Quartus II software platform or MaxPlus II belonging to Altera, which allowed us to work while a research tool ever studied in the Engineering Degree in Telecommunications and Electronics in Control and Automation.

For the design of each of the rough experimental practices in analyzing the curriculum in Digital Systems II which is based on the sequential design which allows students to acquire skills related to electronics within the studies offered by the School technical Development.

Methodology is used Project Based Learning (PBL) to propose to undergraduates small projects where experience the usual problems of the development of sequential machines. In fact, as will the Quartus II software platform provides a rich working environment also with specific skills to work on proving digital electronics and valuable programming skills to work in other subjects as Digital Systems I, Fundamentals of Digital, Laboratory of Digital, Microcontrollers, Microprocessors, Digital Electronics Design.

CAPÍTULO 1: GENERALIDADES

1.1. INTRODUCCIÓN

La tecnología de arreglos de compuertas programables en campo (FPGA) continúa siendo impulsada. Se esperaba que el mercado de FPGAs en todo el mundo aumentará de $1,900 millones en el 2005 a $2,750 millones en el 2010[1] Desde que Xilinx los inventó en 1984, los FPGAs han pasado de ser sencillos chips de lógica de acoplamiento a reemplazar a los circuitos integrados de aplicación específica (ASICs) y procesadores para procesamiento de señales y aplicaciones de control. ¿Por qué ha tenido tanto éxito esta tecnología? Este artículo ofrece una introducción a los FPGAs y destaca algunos de los beneficios que hacen a los FPGAs únicos [NI, 2011].

En las instituciones de educación superior extranjeras como por ejemplo España emplean los FPGAs y lenguaje de descripción circuital (VHDL) en carrera como Ingeniería en Electrónica, Telecomunicaciones, Telemática e Industrial; en el Ecuador la tecnología comienza a despertar el interés que realmente merece por su gran capacidad de despliegue para futuras investigaciones, por este motivo la Facultad de Educación Técnica para el Desarrollo (FETD) de la Universidad Católica de Santiago de Guayaquil empleará los FPGAs y VHDL en el Laboratorio de Electrónica para su uso en asignaturas como Sistemas Digitales I, II y Laboratorio de Digitales.

1.2. ANTECEDENTES

Desde la creación de la carrera de Ingeniería en Telecomunicaciones en mayo de 1998, misma que ha venido desarrollándose hasta la presente con 3 mallas de estudio (malla 1, 1998; malla 2, 2002; malla actual, 2004), obviamente esta actualización se debe a los avances tecnológicos en materias del área de telecomunicaciones y electrónica, esta última debe considerar los laboratorios de las asignatura impartidas en la mencionada área.

[1] *The Field-Programmable Gate Array (FPGA): Expanding Its Boundaries,* InStat Market Research, April 2006

El laboratorio de la Facultad Técnica para el desarrollo cuenta equipos que permiten a los alumnos realizar experimentos en electrónica analógica, sistemas digitales, microcontroladores y microprocesadores, aunque estos no han sido actualizados en estos últimos 2 años, faltan dispositivos electrónicos como el caso de FPGA a través de la tarjeta Cyclone II de Altera. En cuanto al aprendizaje de sistemas digitales resulta difícil llegar a fabricar un circuito de alta escala de integración, que involucra riesgos y costos de diseño muy altos e imposibles de asumir por las empresas. A partir de esa gran dificultad, diversos grupos de investigadores empiezan a crear y desarrollar los llamados “lenguajes de descripción de hardware”, en los que no fuera necesario caracterizar eléctricamente cada componente del circuito al nivel de transistor sino enfocándose solamente en el funcionamiento lógico del sistema.

1.3. JUSTIFICACIÓN DEL PROBLEMA

En la actualidad el laboratorio de electrónica de la FETD en la UCSG el aprendizaje de la práctica se reduce normalmente a simulaciones por ordenador, protoboard dado que se suele disponer en general de unas pocas maquetas de procesos, y muy limitadas, debido sobre todo al alto precio de éstas. Asimismo se debe considerar un lenguaje de descripción circuital para la asignatura de Sistemas Digitales II, dado que cuando se aborda el diseño de un sistema electrónico y surge la necesidad de implementar una parte con hardware dedicado son varias las posibilidades que hay.

1.4. DEFINICIÓN DEL PROBLEMA

Necesidad de equipar con tecnología actualizada el Laboratorio de Electrónica específicamente en la asignatura de Sistemas Digitales II a través de Cyclone II FPGA de ALTERA para mejorar el proceso de aprendizaje de los alumnos y promover la investigación para diferentes aplicaciones que se pueden realizar con el mismo.

1.5. OBJETIVOS DE LA INVESTIGACIÓN

1.5.1. OBJETIVO GENERAL

Diseñar prácticas con FPGA para la materia de Digitales II mediante la implementación de Cyclone II FPGA Starter Development Board de Altera de manera que los estudiantes de las carreras de Ingeniería en Telecomunicaciones y Electrónica accedan a otro tipo de tecnología que emplean Universidades locales como extranjeras.

1.5.2. OBJETIVOS ESPECÍFICOS

- Describir la fundamentación teórica de FPGA para los sistemas digitales a nivel de diseño y programación.
- Elaborar las prácticas para la asignatura de Sistemas Digitales II e incluirlas en el programa de estudio.
- Evaluar las prácticas desarrolladas en el Cyclone II FPGA Starter Development Board de Altera.

1.6. IDEA A DEFENDER

Con la elaboración de prácticas de sistema digitales e implementarlas en la placa Cyclone II FPGA de Altera permitirá a los estudiantes de las carrera mencionadas con anterioridad darse una idea de cómo medir un proceso simple, o decidir si cierta estructura es mejor que otra, también contribuirá a los docentes a explicar ciertos algoritmos sin necesidad de pasa mucho tiempo en el laboratorio. Resumiendo todo esto es que contribuirá a que los docentes y estudiantes elaboren sus propios diseños, con funciones más específicas y más apegadas a la realidad.

1.7. METODOLOGÍA

Para alcanzar los objetivos propuestos se desarrolla una metodología, que se presenta a continuación:

1. Revisar las investigaciones ya realizadas por expertos.

2. Diseñar las prácticas de sistemas digitales II para ser programadas en VHDL.
3. Representar los comportamientos básicos elegidos a nivel de software, en módulos programados en diagramas de bloques.
4. En base a esto y a los resultados esperados en las programaciones realizar las modificaciones necesarias para implementar en la tarjeta Cyclone II.

CAPÍTULO 2: Lenguaje de descripción circuital (VHDL)

En el presente capítulo se describirá el estado del arte del lenguaje de descripción circuital (VHDL) que hacen referencia al proyecto de grado previo a la obtención del grado de Ingeniero en Telecomunicaciones.

2.1. LÓGICA PROGRAMABLE Y LENGUAJE DE DESCRIPCIÓN EN HARDWARE (HDL).

Como consecuencia de la creciente necesidad de integrar un mayor número de dispositivos en un solo circuito integrado, se desarrollaron nuevas herramientas de diseño que auxilian al ingeniero a integrar sistemas de mayor complejidad. Esto permitió que en la década de los cincuenta aparecieran los lenguajes de descripción en hardware (HDL) como una opción de diseño para el desarrollo de sistemas electrónicos elaborados. Estos lenguajes alcanzaron mayor desarrollo durante los años setenta, lapso en que se desarrollaron varios de ellos como IDL de IBM, TI-HDL de Texas Instruments, ZEUS de General Electric, etc., todos orientados al área industrial, así como los lenguajes en el ámbito universitario (AHPL, DDL, CDL, ISPS, etc.) [IEEE, 97].

Los primeros no estaban disponibles fuera de la empresa que los manejaba, mientras que los segundos carecían de soporte y mantenimiento adecuados que permitieran su utilización industrial. El desarrollo continuó y en la década de los ochenta surgieron lenguajes como VHDL, Verilog, ABEL 5.0, AHDL, etc., considerados lenguajes de descripción en hardware porque permitieron abordar un problema lógico a nivel funcional (descripción de un problema sólo conociendo las entradas y salidas), lo cual facilita la evaluación de soluciones alternativas antes de iniciar un diseño detallado.

Una de las principales características de estos lenguajes radica en su capacidad para describir en distintos niveles de abstracción (funcional, transferencia de registros RTL y lógico o nivel de compuertas) cierto diseño. Los niveles de abstracción se emplean para clasificar modelos HDL según el grado de detalle y precisión de sus descripciones [Teres, 98]. Los niveles de

abstracción descritos desde el punto de vista de simulación y síntesis del circuito pueden definirse como sigue:

- ✓ Algorítmico: se refiere a la relación funcional entre las entradas y salidas del circuito o sistema, sin hacer referencia a la realización final.
- ✓ Transferencia de registros (RT): Consiste en la partición del sistema en bloques funcionales sin considerar a detalle la realización final de cada bloque.
- ✓ Lógico o de compuertas: el circuito se expresa en términos de ecuaciones lógicas o de compuertas.

2.1.1. VHDL, LENGUAJE DE DESCRIPCIÓN EN HARDWARE.

En la actualidad, el lenguaje de descripción en hardware más utilizado a nivel industrial es VHDL[2] (*Hardware Description Language*), que apareció en la década de los ochenta como un lenguaje estándar, capaz de soportar el proceso de diseño de sistemas electrónicos complejos, con propiedades para reducir el tiempo de diseño y los recursos tecnológicos requeridos.

El Departamento de Defensa de Estados Unidos creó el lenguaje VHDL como parte del programa "*Very High Speed Integrated Circuits*" (VHSIC), a partir del cual se detectó la necesidad de contar con un medio estándar de comunicación y la documentación para analizar la gran cantidad de datos asociados para el diseño de dispositivos de escala y complejidad deseados [Delgado, 1993]; es decir, VHSIC debe entenderse como la rapidez en el diseño de circuitos integrados.

Después de varias versiones revisadas por el gobierno de los Estados Unidos, industrias y universidades, el IEEE (Instituto de Ingenieros Eléctricos y Electrónicos) publicó en diciembre de 1987 el estándar IEEEstd 1076-1987. Un año más tarde, surgió la necesidad de describir en VHDL todos los ASIC creados por el Departamento de Defensa, por lo que en 1993 se adoptó el estándar adicional de VHDL IEEE1164. Hoy en día VHDL se considera como un estándar para la descripción, modelado y síntesis de circuitos digitales y

[2] http://www.altera.com/products/fpga.html

sistemas complejos. Este lenguaje presenta diversas características que lo hacen uno de los HDL más utilizados en la actualidad.

2.1.2. VENTAJAS DE EMPLEAR VHDL EN EL DESARROLLO DE CIRCUITOS INTEGRADOS.

A continuación se exponen algunas de las ventajas que representan los circuitos integrados con VHDL:

- Notación formal. Los circuitos integrados VHDL cuentan con una notación que permite su uso en cualquier diseño electrónico.
- Disponibilidad pública. VHDL es un estándar no sometido a patente o marca registrada alguna, por lo que cualquier empresa o institución puede utilizarla sin restricciones. Además, dado que el IEEE lo mantiene y documenta, existe la garantía de estabilidad y soporte.
- Independencia tecnológica de diseño. VHDL se diseñó para soportar diversas tecnologías de diseño (PLD, FPGA, ASIC, etc.) con distinta funcionalidad (circuitos combinacionales, secuenciales, síncronos y asincronos), a fin de satisfacer las distintas necesidades de diseño.
- Independencia de la tecnología y proceso de fabricación. VHDL se creó para que fuera independiente de la tecnología y del proceso de fabricación del circuito o del sistema electrónico. El lenguaje funciona de igual manera en circuitos diseñados con tecnología MOS, bipolares, BICMOS, etc., sin necesidad de incluir en el diseño información

2.2. VHDL como lenguaje para describir, simular, validar y diseñar.

En un principio, la captura de esquemas fue la forma habitual de diseño CAD (apoyado y almacenado en un computador). Pero, hoy día, ha sido sustituida (casi por completo) por su descripción funcional en texto (programa que detalla el funcionamiento de las diversas partes del circuito y la conexión entre ellas), utilizando para ello un lenguaje de descripción de hardware (HDL).

La forma textual presenta numerosas ventajas: suele requerir menor tiempo y esfuerzo para comprender lo que el circuito hace (en el caso de sistemas complejos); es independiente de la implementación a bajo nivel (en puertas, bloques, biestables y registros); es directamente trasladable a los

diversos dispositivos programables y a las diversas librerías de ASICs; y, sobre todo, resulta mucho más sencillo revisar e introducir modificaciones en el texto descriptor que en el esquema gráfico del mismo.

Actualmente, son dos los lenguajes de descripción circuital que se han impuesto como estándares para el diseño digital: VHDL y Verilog; y, de entre ellos, en el contexto europeo predomina el VHDL (si bien Verilog resulta, en buena medida, más cercano al hardware y a los esquemas gráficos que se utilizaban anteriormente). El lenguaje de descripción de sistemas digitales VHDL nació como herramienta de documentación y de comunicación en relación con los circuitos integrados digitales.

Un circuito integrado complejo, a lo largo del tiempo pasa por muchas manos (por muchas mentes): quien lo define y quien lo utiliza no suele ser el mismo que quien lo diseña; quien repara o quien modifica los sistemas en que participa el circuito no suele ser el propio diseñador; e, incluso, quien lo ha diseñado, al volver a su circuito cuando ha transcurrido un cierto tiempo, difícilmente recordará los detalles de las diversas funciones y recursos con los que lo ha configurado.

De ahí la necesidad de un lenguaje que ofrezca una descripción funcional precisa, carente de ambigüedades, estructurada y de fácil lectura e independiente de la implementación concreta a bajo nivel. Que quien lea esa descripción sea capaz de comprender, con poco esfuerzo y absoluta claridad, las funciones que hace el circuito y cómo las hace; sin tener que descender al nivel booleano de puertas y biestables que, por su mayor amplitud en componentes, resultaría más difícil de analizar.

Además, una misma descripción funcional puede configurarse circuitalmente deformas muy variadas y tal configuración dependerá de la librería de celdas estándar disponibles (para el diseño de un ASIC) o, en su caso, del dispositivo programable en que se inserte; en principio no es necesario conocer su configuración a nivel booleano para utilizar

eficientemente un circuito digital. Lo que sí es necesario es disponer de una adecuada y detallada descripción funcional.

La escritura es la herramienta de las ideas; el vehículo hacía la claridad, la precisión, la estructura (disposición, orden y enlace de las partes que configuran el todo) y la comunicación (transmisión de las ideas en el espacio y en el tiempo); la escritura es el mejor medio para poder trasladar las ideas a otras personas (comunicación espacial) y poder contrastarlas y debatirlas y, también, para recordar las ideas (comunicación temporal) y poder recuperarlas y revisarlas pasado el tiempo.

La documentación es un requisito inexcusable para considerar que un diseño se ha completado y hecho efectivo y, cuando el diseño se refiere a un sistema complejo, requiere un lenguaje preciso y estructurado (estructurado tanto en su forma de expresar el diseño como en el sentido de que confiera estructura al propio diseño).

Por otra parte, antes de construir un diseño complejo, con el coste económico y de tiempo que ello supone, conviene saber si el diseño es correcto; para ello es de gran ayuda efectuar una simulación virtual de su funcionamiento. Un lenguaje que describa con precisión el comportamiento de un sistema servirá, sin duda, para simular su comportamiento: bastará con una aplicación informática que ejecute la descripción del sistema en relación con el transcurso del tiempo.

Aún más, un circuito integrado no tiene sentido funcional por sí mismo, sino que formará parte de un sistema más amplio; generalmente será una pieza de control de un sistema con partes eléctricas y mecánicas. El objetivo final no es que el circuito integrado funcione (individualmente) bien, sino que el sistema global actúe correctamente: en definitiva, el objetivo que se persigue no es el funcionamiento del propio circuito integrado, sino el del sistema de qué forma parte.

De manera que el propósito de un lenguaje eficiente se refiere a la capacidad de describir y simular, además de los circuitos integrados digitales, los mecanismos y sistemas controlados por ellos. Es decir, validar al circuito integrado en el entorno funcional para el que ha sido diseñado. Estamos, pues, planteando un objetivo cada vez más ambicioso: de la documentación se pasa a la simulación y se pretende, también, la validación de los circuitos integrados complejos.

Con esa intencionalidad ha sido desarrollado el *Very high speed integrated circuit Hardware Description Language*, que conocemos como VHDL, siglas que podemos referir mejor a *Versatile Hardware Description Language* (pues la referencia a la alta velocidad de los circuitos es superflua; de igual forma puede ser utilizado para documentar, describir, simular y validar circuitos integrados lentos).Un lenguaje con capacidad para describir con precisión y simular con eficiencia puede ser fácilmente utilizado para diseñar: una vez descrito un sistema digital, basta compilar la descripción sobre una librería de recursos.

Se entiende por compilación el paso de la descripción funcional a la configuración circuital (del algoritmo al circuito), utilizando como componentes de dicho circuito los contenidos en una librería de recursos:

- ✓ En el caso de CPLDs dicha librería se refiere a las funciones booleanas en forma de suma de productos (configuración PAL) y biestables tipo D;
- ✓ Para las FPGAs los recursos son las funciones booleanas, con un limitado número de variables (dividiendo, en su caso, las funciones más amplias), expresadas en forma de tabla de verdad (configuración LUT) y los biestables D;
- ✓ y en el diseño de ASICs, la librería de celdas básicas prediseñadas, propia del diseño con librería estándar (*standard cell*).

El compilador (la herramienta informática de compilación) traslada la descripción funcional al circuito que la materializa, conformado por componentes disponibles en la librería de recursos sobre la que se compila. VHDL es una excelente herramienta de documentación, simulación, validación

y diseño de sistemas digitales, estandarizada y en permanente proceso de actualización bajo los auspicios de la IEEE (*Institute of Electrical and Electronics Engineers*).

El único lenguaje alternativo que goza, también, de amplia aceptación es Verilog, pero su difusión en el contexto americano es mucho menor (aunque, en buena medida, Verilog es un lenguaje más directo y más cercano al propio circuito digital) versus el continente europeo.

2.3. VHDL elemental para diseño de circuitos combinacionales.

El lenguaje VHDL no se refiere solamente a sistemas digitales sino que está abierto a la descripción de sistemas de cualquier tipo; por otra parte, VHDL es un lenguaje tremendamente versátil y potente. Su descripción requiere todo un amplio volumen y su estudio precisa de, al menos, un curso específico dedicado solamente a este lenguaje.

Interesa aquí la utilización de VHDL para diseñar sistemas digitales, es decir, aquella parte del lenguaje que es compilable para dar como resultado un circuito digital. A continuación, se expone un breve resumen, a la vez parcial y útil, como primera aproximación a este lenguaje y, a fin de ser lo más directa y práctica posible, esta introducción se realiza a través de ejemplos concretos de descripción circuital.

2.3.1. Nociones básicas.

VHDL no distingue entre MAYÚSCULAS y minúsculas (salvo unas pocas excepciones referidas a valores de las señales). Pueden incluirse comentarios y, para identificarlos, se inician con el símbolo repetido "--" que indican al compilador que ignore todo lo que sigue hasta final de línea. Cada «módulo» descriptivo y cada asignación se cierran con el símbolo ";". Los elementos básicos de la descripción digital son las señales (*signal*); para ellas suele utilizarse el tipo std_logic (*standard logic*) que admite los siguientes nueve valores:

'0' -- cero '1' – uno	valores booleanos típicos
'X' – desconocido	no se conoce el valor

'Z' -- alta impedancia		propio de tri-estado
'U' -- sin inicializar		biestables en su situación previa
'-' -- no importa (*don't care*)		indiferente (para simplificación)
'L' -- 0 débil	'H', -- 1débil	'W', -- desconocido débil

Los valores de una señal se expresan siempre entre comillas simples: '0', '1' y los valores X y L no admiten la minúscula (x y l no son válidas). Los tres primeros valores (0, 1, X) son de tipo fuerte, si se encuentran dos de ellos aplicados sobre un nudo el resultado es X (desconocido). Los valores débiles corresponden a determinadas situaciones circuitales que, si confluyen con algún valor fuerte, dan como resultado dicho valor fuerte; en cambio, si se encuentran dos valores débiles sobre un nudo el resultado es W (desconocido débil).

Un conjunto de señales constituye un vector, std_logic_vector, que puede ser declarado en forma ascendente std_logic_vector (0 to 7) o descendente std_logic_vector (7downto 0), siendo más frecuente esta segunda declaración porque corresponde a la forma típica en la que el dígito más significativo es el de mayor subíndice; el conjunto de valores que adopta un vector se expresa entre comillas dobles: por ejemplo, "11010001".

Para conjuntos de señales (vectores), se utiliza también el tipo *integer* (entero) que debe ser declarado para un rango determinado: *integer range* 0 to 15 (señal de 4 bits). Los tipos de señales, std_logic y std_logic_vector, aunque son los más habituales en diseño digital (ya que describen bien las señales electrónicas en todas sus posibilidades)

No se encuentran definidos en el propio VHDL básico (están definidos los tipos bit y bit_vector, que admiten sólo los dos valores booleanos 0 y 1). Los tipos standard logic han sido introducidos en la normalización hecha por IEEE y requieren la declaración de la librería (y de los paquetes) que los definen al principio de la descripción:

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

```
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

Con el paquete *ieee.std_logic_unsigned* las operaciones se realizan en binario natural. Si se desea efectuarlas en complemento a 2 debe utilizarse el paquete *ieee.std_logic_signed*. En estos paquetes se dispone de dos funciones muy útiles:

CONV_INTEGER(a) que convierte el *std_logic_vector* **a** en *integer*

CONV_STD_LOGIC_VECTOR (b,n) que convierte el *integer* **b** en vector de longitud **n**.

Las operaciones básicas entre señales son:

asignación:	<=					
operaciones booleanas	and	or	not	xor		
comparaciones	=	/=	>	<	>=	<=
aritméticas	+	-	*			
concatenación	&					

La concatenación se refiere a poner señales o vectores juntos, formando un vector más largo, cuyo número de bits es la suma de los números de ambas señales. La base de la descripción VHDL es la asignación de valores a una señal, la cual puede hacerse directamente o por medio de operaciones entre señales.

Ejemplos de asignaciones directas:

```
signal a, b, c, Y: std_logic_vector(3 downto 0);
signal m, n: integer range 0 to 15;          -- 4 bits
y <= "1001";
Y <= (3 => '1', 0 => '1', others => '0');    -- equivale a la anterior ("1001")
m <= 9;
Y <= ((not a) and b) or (a and not b);       -- equivale a y <= a xor b;
y <= a + b;                                  -- suma aritmética
```

El lenguaje VHDL es muy disciplinado: una señal de un tipo no admite asignación de valores o de señales de otro tipo. Ejemplos de asignaciones incorrectas:

```
y <= 9;                 -- y no es de tipo integer
m <= "1001";            -- m no es del tipo std_logic
Y <= m;                 -- tipos de señal diferentes
M <= a;                 -- tipos de señal diferentes
Y <= "001";             -- faltan componentes
m <= 18;                -- fuera de rango
y <= '1001';            -- faltan comillas dobles
M <= 4                  -- falta;
```

2.3.2. Estructura de librerías, entidades y arquitecturas.

En VHDL se describe por un lado la caja del circuito con sus entradas y salidas, ósea, los terminales de conexión hacia el exterior, y eso se hace en un módulo denominado *entity*, y en otro módulo posterior, denominado *architecture*, se describe lo que hace el circuito, es decir, su funcionamiento interno. Además, es preciso declarar previamente las librerías necesarias para compilar el circuito. En consecuencia, la descripción VHDL tiene la siguiente estructura:

- declaración de librerías
- módulo de terminales

```
entity nombre_de_la_entidad is
port(
        declaración de entradas y salidas
                                              );
        end nombre_de_la_entidad;
```

- módulo de funciones

```
architecture nombre_de_la_arquitectura of nombre_de_la_entidad is
signal          declaración de señales internas
    begin
                    descripción del funcionamiento (asignaciones)
    end nombre_de_la_arquitectura ;
```

Para comprobar lo descrito anteriormente consideramos una compuerta lógica como el CI 7400 que contiene 4 puertas ***NAND***. La representación gráfica de la entidad y la arquitectura se puede observar en el circuito de la figura 2.5 es *entity* cuatro_puertas *is*, y *architecture* puertas_nand *of* cuatro_puertas *is* respectivamente.

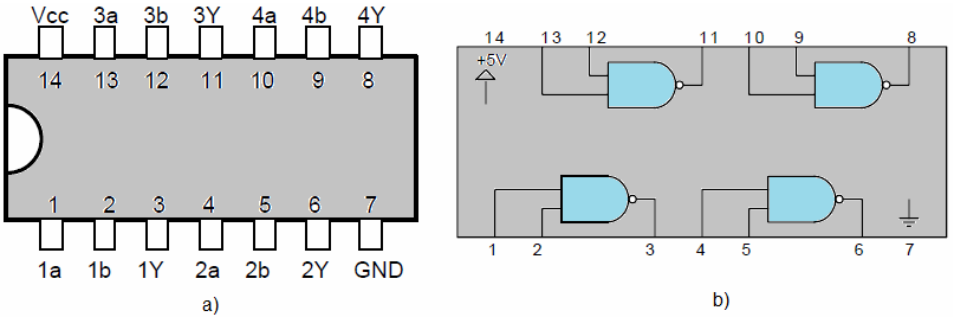


Figura 2. 1: Compuerta lógica NAND: (a) Da Vinci (b) AESOP y (c) Zeus
Fuente: http://digitale.galeon.com/graficas/7400.png

En la entidad (*entity*) se describen los terminales del circuito dentro del epígrafe de puertos (*ports*); hay cuatro tipos de puertos: entrada (*in*), salida (*out*), bidireccionales (*inout*) y adaptados (*buffer*). Los puertos de salida no se pueden leer dentro del circuito, es decir, no pueden figurar como entradas en ninguna de las asignaciones de su arquitectura; en cambio, los puertos adaptados son salidas que sí se pueden leer dentro del circuito (sin embargo, suele utilizarse poco este tipo de puertos).

2.3.3. Asignaciones concurrentes

Son asignaciones concurrentes aquellas que se ejecutan siempre y directamente sobre una señal; de forma que una señal no puede recibir dos asignaciones ocurrentes (daría lugar a error al intentar imponer dos valores a la misma señal). En lo que sigue, los valores se representan genéricamente con el grafismo """""""" y las condiciones (principalmente, comparaciones) con.....; denominaremos expresión a cualquier conjunto de operaciones entre señales, entre valores y entre ambos y utilizaremos ------- para representarlas; una expresión puede ser un valor, una señal, una operación (o una serie de operaciones) entre señales o entre valores o entre ambos. Las asignaciones concurrentes pueden ser:

- ✓ Fijas: señal <= -------;
  es decir, señal <= valor; señal <= señal;
  señal <= operaciones entre señales, entre valores y entre ambos;

- ✓ Condicionales: señal <= ------- when ..... else
  ------- when ..... else
  ------- when ..... else
  -------;

- ✓ múltiples: with ------- select
  señal <= ------- when "",
  ------- when "",
  ------- when ""
  ------- when "",
  ------- when others;

Para aplicar correctamente la programación describiremos las asignaciones concurrentes de un multiplexor de cuatro líneas de entrada como el mostrado por la figura 2.6.

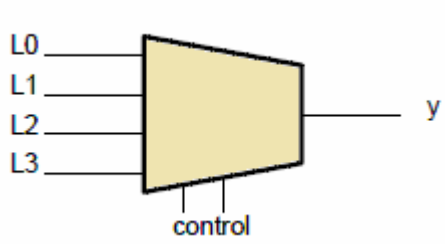


Figura 2. 2: Diagrama de bloques del multiplexor (4 a 1)
Fuente: http://digitalsystemsmei.blogspot.com/2010/11/multiplexor.htm

- ➢ *La descripción de un multiplexor de 4 a 1 se puede realizar en dos métodos:*

```
signal control : integer range 0 to 3;
y <= L0 when control = 0     else
L1 when control = 1 else
L2 when control = 2 else
L3;
```

- *Descripción de un decodificador (ver figura 2.7) de ocho líneas*

```
entrada <= c & b & a;
with entrada select
salida <= "10000000" when "000",
"01000000" when "001",
"00100000" when "010",
"00010000" when "011",
"00001000" when "100",
"00000100" when "101",
"00000010" when "110",
"00000001" when others;
```

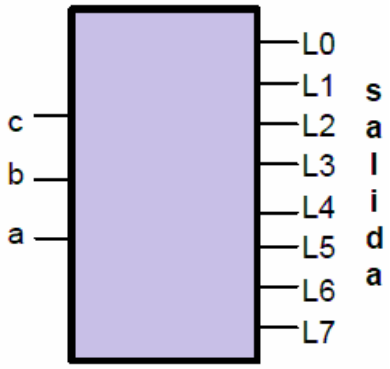


Figura 2. 3: Decodificador de ocho líneas
Elaborado por: Los autores

- *Descripción de un codificador de prioridad (ver figura 2.8) de ocho líneas*

```
salida <=   "1001" when L9 = '1' else
            "1000" when L8 = '1' else
            "0111" when L7 = '1' else
            "0110" when L6 = '1' else
            "0101" when L5 = '1' else
            "0100" when L4 = '1' else
            "0011" when L3 = '1' else
            "0010" when L2 = '1' else
            "0001" when L1 = '1' else "0000";
```

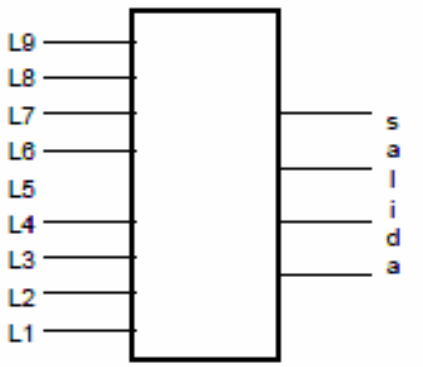


Figura 2. 4: Codificador de prioridad de ocho líneas
Elaborado por: Los autores

Comparación de números:

```
igual <= '1' when A = B else '0';
mayor <= '1' when (A > B) else '0';
menor <= '1' when (A < B) else '0';
```

Suma de dos números:

```
R <= A + B;
```

También los procesos (*process*), que se describen en el próximo acápite, son concurrentes (cada uno de ellos considerado globalmente es una asignación concurrente) y no puede asignarse valores a una señal en dos procesos diferentes.

2.3.4. Asignaciones secuenciales

Las asignaciones secuenciales se encuentran dentro de un módulo denominado proceso (process) y no se ejecutan hasta que se ha terminado de leer todo el módulo. Dentro de un proceso puede haber dos o más asignaciones referidas a la misma señal y es válida la última de ellas; en el caso de asignaciones condicionales (que será el caso general), es válida la última de ellas que resulta efectiva (es decir, cuyas condiciones se cumplen).

Las asignaciones secuenciales no corren peligro de imponer doble valor a una misma señal, pues son consideradas en el orden en que están escritas (igual que un programa de computador) y solamente se aplica la última efectiva de ellas. Los procesos se declaran y se concluyen de la siguiente forma:

```
nombre_del proceso (opcional):   process (lista de sensibilidad)
                                 begin
                                 asignaciones
                                 end process;
```

La lista de sensibilidad se refiere a las señales que «despiertan» el proceso (que lo hacen operativo) y, en el caso de descripción circuital, debe contener todas las señales que actúan como entradas sobre el proceso. En principio, los compiladores no tienen en cuenta la lista de sensibilidad pero suelen avisar si ésta es incompleta.

Cada proceso, considerado globalmente, es una asignación concurrente (o, si asigna valor a varias señales, un conjunto de asignaciones concurrentes sobre ellas): no se puede efectuar asignación a una misma señal en dos procesos diferentes. Las asignaciones secuenciales pueden ser fijas, condicionales o múltiples. Las fijas utilizan la misma sintaxis que las asignaciones concurrentes fijas, pero las condicionales y las múltiples utilizan sintaxis diferentes:

- ✓ condicionales:

```
if .....        then señal <= -------; señal <= -------; señal <= -------;
  elsif .....   then señal <= -------; señal <= -------; señal <= -------;
  elsif .....   then señal <= -------; señal <= -------; señal <= -------;
  else                 señal <= -------; señal <= -------; señal <= -------;
end if;
```

- ✓ múltiples:

```
case ..... is
    when """""""" => señal <= -------; señal <= -------; señal <= -------;
    when """""""" => señal <= -------; señal <= -------; señal <= -------;
    when """""""" => señal <= -------; señal <= -------; señal <= -------;
    when others => señal <= -------; señal <= -------; señal <= -------;
end case;
```

Dentro de un proceso no pueden utilizarse asignaciones con when o con with y, de igual forma, las estructuras if y case no pueden utilizarse fuera de procesos. Ahora mostraremos algunas aplicaciones prácticas de programación en VHDL de asignaciones secuenciales:

- *Descripción de un multiplexor de cuatro líneas de entrada*

```
process (control,L3,L2,L1,L0)
begin
if control = 0 then y <= L0; end if;
if control = 1 then y <= L1; end if;
if control = 2 then y <= L2; end if;
if control = 3 then y <= L3; end if;
end process;
end case;
end process;
```

- *Descripción de un codificador de prioridad de nueve líneas*

```
process (L9,L8,L7,L6,L5,L4,L3,L2,L1)
begin
if L9 = '1' then                salida <= "1001";
elsif L8 = '1' then             salida <= "1000";
elsif L7 = '1' then             salida <= "0111";
elsif L6 = '1' then             salida <= "0110";
elsif L5 = '1' then             salida <= "0101";
elsif L4 = '1' then             salida <= "0100";
elsif L3 = '1' then             salida <= "0011";
elsif L2 = '1' then             salida <= "0010";
elsif L1 = '1' then             salida <= "0001";
else                            salida <= "0000";
end if;
end process
```

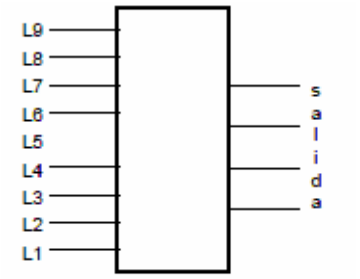


Figura 2. 5: Codificador de prioridad de nueve líneas
Elaborado por: Los autores

- *Descripción de un demultiplexor de ocho líneas.* (ver figura 2.10)

```
control : integer range 0 to 7
process (control,entrada)
begin
-- asignaciones por defecto
L0 <= '0'; L1 <= '0'; L2 <= '0'; L3 <= '0';
L4 <= '0'; L5 <= '0'; L6 <= '0'; L7 <= '0';
case control is
                when 0 => L0 <= entrada;
                when 1 => L1 <= entrada;
                when 2 => L2 <= entrada;
                when 3 => L3 <= entrada;
                when 4 => L4 <= entrada;
                when 5 => L5 <= entrada;
                when 6 => L6 <= entrada;
                when 7 => L7 <= entrada;
end case;
end process
```

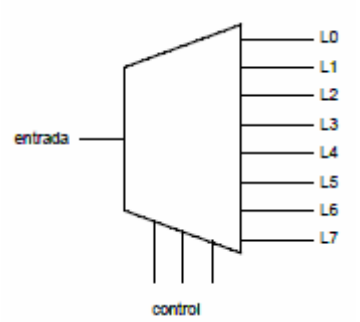


Figura 2. 6: Codificador de prioridad de nueve líneas
Elaborado por: Los autores

2.3.5. Conversor BCD a 7 segmentos de ánodo común

El conversor mostrado en la figura 2.11 recibe las 4 entradas BCD y proporciona las 7 salidas (g f e d c b a) correspondientes a la activación de los 7 segmentos (que se activarán con valor 0, ya que son de ánodo común); además, dispone de una entrada LT para test de lámparas y otra entrada RBI, con su correspondiente salida RBO, para apagado de ceros no significativos.

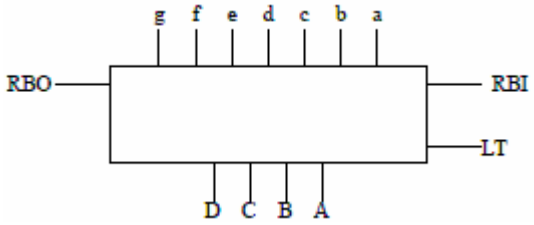


Figura 2. 7: Codificador de prioridad de nueve líneas
Elaborado por: Los autores

```
library ieee;    use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity BCD_7SEG is
     port ( D,C,B,A,LT,RBI        : in std_logic;
     SALIDA                       : out std_logic_vector(1 to 7); -- a b c d e f g
     RBO                           : out std_logic );
end BCD_7SEG;
-- con asignaciones concurrentes
architecture CODIFICADOR of BCD_7SEG is
signal bcd:     std_logic_vector(3 downto 0);
signal ENTRADA: integer range 0 to 9;
signal AUX:      std_logic_vector(1 to 7); -- señal auxiliar;
begin
bcd <= D & C & B & A;      ENTRADA <= conv_integer (bcd);
RBO <= '1' when (RBI = '1') and (entrada = 0) else '0';
SALIDA <=      "0000000" when LT = '1' else
                    "1111111" when (RBI = '1') and (entrada = 0) else AUX;
with ENTRADA select                  -- ánodo común: activo el 0
AUX <=          "0000001" when 0, "1001111" when 1,
                    "0010010" when 2, "0000110" when 3,
                    "1001100" when 4, "0100100" when 5,
```

```
                "0100000" when 6, "0001111" when 7,
                "0000000" when 8, "0000100" when 9,
                "1111111" when others;
end CODIFICADOR;
```

2.3.6. Decodificador de mapa de memoria

Se trata de situar, en un mapa de memoria cuyo bus de direcciones tiene 16 líneas, un circuito integrado RAM de 8 K al comienzo del mapa, de 0000(H a 1FFF(H, un adaptador de puertos PIA que tiene 4 registros a partir de la posición A000(H, y dos circuitos integrados ROM al final de memoria, uno de 2K de F000(H a F7FF(H y el otro de 4K de F800(H a FFFF(H; el decodificador de direcciones utiliza, para ello, solamente las 6 líneas superiores del bus de direcciones.

```
library ieee; use ieee.std_logic_1164.all; use ieee.std_logic_unsigned.all;
entity MAPA is
    port ( A15, A14, A13, A12, A11, A10  : in std_logic;
           RAM, IO,ROM1,ROM2             : out std_logic);
 end MAPA;
architecture HABILITACIONES of MAPA is
signal DIR      :std_logic_vector (15 downto 0);
-- se introduce la señal auxiliar DIR para referir las direcciones a las 16
-- líneas del bus de direcciones (y por tanto 16 bits del mapa de memoria)
begin
DIR       <= A15 & A14 & A13 & A12 & A11 & A10 & "0000000000";
RAM       <= '1' when DIR <= 16#1FFF# else '0';
-- la notación 16#…….# indica que el número es hexadecimal
IO        <= '1' when DIR = 16#A000#;
ROM1    <= '1' when (DIR >= 16#F000#) and (DIR <= 16#F7FF#) else '0';
ROM2    <= '1' when DIR >= 16#F800# else '0';
end HABILITACIONES;
```

2.4. Descripción de circuitos secuenciales y sistemas síncronos

2.4.1. Biestables (asíncronos)

En los biestables, la salida actúa también como entrada (realimentación) como se muestra en la figura 2.12, y habida cuenta que las salidas VHDL (*port out*) no pueden ser leídas desde dentro del circuito (es decir, no pueden actuar como entradas de ninguna asignación), es necesario utilizar para la realimentación una señal interior, del mismo valor que la salida.

```
.......
port( q :out std_logic;
.......
architecture nombre_de_la_arquitectura of nombre_de_la_entidad is
signal q_interna: std_logic;
begin
q <= q_interna;
```

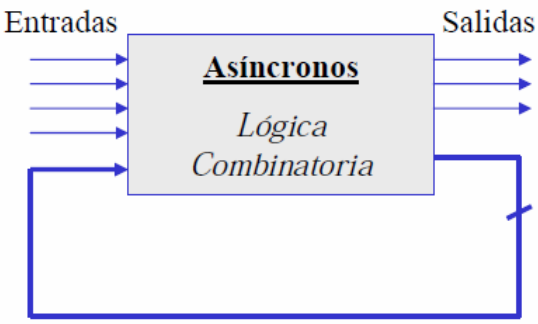


Figura 2. 8: Diagrama de bloques de un sistema asíncrono
Elaborado por: Los autores

Por otra parte, los procesos tienen memoria implícita: si una señal recibe una asignación condicional dentro de un proceso y el conjunto de asignaciones no es completo (es decir, existe alguna condición en que la asignación a dicha señal no está especificada), el proceso asigna por defecto la conservación del valor de dicha señal. Es como si, al comienzo del proceso existiera la asignación señal <= señal; referida, por defecto, a cada una de las señales que reciben alguna asignación dentro del proceso.

Ejemplo:
```
process(a,b)
begin
if a = '1' then p <= b; end if;
end process;
```

En este caso, cuando a = 1, p adopta el valor de b y, cuando a = 0, como no se especifica nada dentro del proceso, p conserva el valor que tenía anteriormente; es equivalente a cualquiera de las dos descripciones siguientes:

```
process(a,b)                          process(a,b)
begin                                 begin
if a = '1' then p <= b;               p <= p;
else p <= p;                          if a = '1' then p <= b; end if;
end if;                               end if;
end process;                          end process;
```

- *Diversas formas de describir un biestable RS*

```
q_interna <= '0' when R = '1' else '1' when S = '1' else q_interna;
process (R,S)
begin
if S = '1' then q_interna <= '1'; end if;
if R = '1' then q_interna <= '0'; end if;
end process; -- borrado prioritario
process (R,S)
begin
if R = '1' then q_interna <= '0'; elsif S = '1' then q_interna <= '1'; end if;
end process;  -- también borrado prioritario
```

- *Diversas formas de describir un biestable D*

```
q_interna <= D when E = '1' else q_interna;
with E select q_interna <= D when '1', q_interna when others;
process (D,E)
begin      if E = '1' then q_interna <= D; end if;     end process;
```

2.4.2. Circuitos síncronos.

El diagrama de bloques de un circuito síncrono se muestra en la figura 2.13, asimismo la descripción CK (señal de reloj) en VHDL ha de hacerse dentro de un proceso, de las siguientes formas:

- ✓ Si todo el proceso es síncrono

  process – sin lista de sensibilidad

```
begin
wait on CK until CK = '1';          -- flanco ascendente
```

- ✓ Si hay una parte asíncrona (por ejemplo, un borrado asíncrono con R)

```
process(R,CK)
begin
if R = '1' then …….
elsif CK'event and CK = '1' then  -- flanco ascendente
-- o, también,
elsif rising-edge(CK) then          -- flanco ascendente
```

- ✓ Biestable D con habilitación y con borrado asíncrono

```
process (R,D,E,CK)
begin
if R = '1' then q_interna <= '0';
elsif CK'event and CK = '1' then
if E = '1' then q_interna <= D; end if;
end if;
end process;
```

- ✓ Biestable JK con marcado y borrado asíncronos

```
process (R,S,J,K,CK)
begin
if R = '1' then q_interna <= '0';
elsif S = '1' then q_interna <= '1';
elsif CK'event and CK = '1' then
if J = '1' and K = '1'  then q_interna <= not q_interna;
elsif J = '1'                  then q_interna <= '1';
elsif K = '1'              then q_interna <= '0';
end if;
end if;
end process;
```

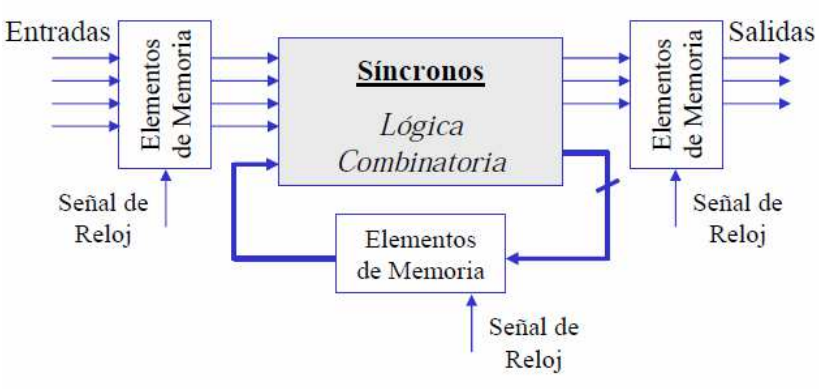


Figura 2. 9: Diagrama de bloques de un sistema asíncrono
Elaborado por: Los autores

2.4.3. Contador

En la figura 2.14 se puede ilustra a un contador módulo 10, bidireccional con habilitación y con borrado y carga paralela síncronos se muestra en la figura

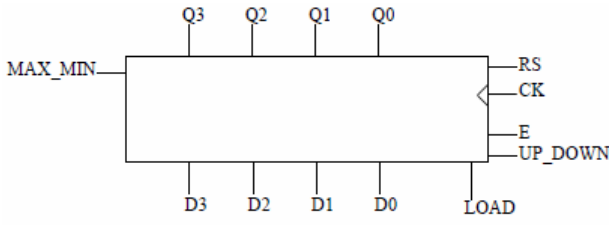


Figura 2. 10: Contador módulo 10
Elaborado por: Los autores

```
library ieee; use ieee.std_logic_1164.all; use ieee.std_logic_unsigned.all;
entity DECADA is
port ( CK,RS,E,UP_DOWN,LOAD : in std_logic;
        D                   : in std_logic_vector(3 downto 0);
        MAX_MIN             : out std_logic;
       Q                    : out std_logic_vector(3 downto 0));
end DECADA;
architecture CONTADOR of DECADA is
signal Q_interior :          std_logic_vector(3 downto 0);
begin
-- COMBINACIONAL:
Q <= Q_interior;
MAX_MIN <= '1' when ((UP_DOWN = '1') and (Q_interior = "1011"))
```

```
or ((UP_DOWN = '0') and (Q_interior = "0000")) else '0';
SINCRONO: process
begin
wait until CK ='1';
if ( RS = '1' ) then                    Q_interior <= "0000";
elsif ( LOAD = '1' ) then               Q_interior <= D;
  elsif ( E = '1' ) then
        if ( UP_DOWN = '1') then
        if Q_interior = "1001" then Q_interior <= "0000";
              else                      Q_interior <= Q_interior + 1; end if;
        else if     Q_interior <= "0000" then     Q_interior <= "1001";
              else                      Q_interior <= Q_interior - 1; end if;
        end if;
end if;
end process;
end CONTADOR;
```

2.4.4. Temporizaciones sucesivas

Sea un cruce de peatones que cuenta con un semáforo para detener a los automóviles, con un pulsador P que debe ser activado por los peatones cuando desean cruzar; la activación de P da lugar al siguiente ciclo (ver figura 2.15): 10" en amarillo para detener a los automóviles, 20" en rojo (verde para peatones), 10" en amarillo para peatones, pasando finalmente al estado de circulación de automóviles (rojo para peatones); cuando en dicho estado de circulación se recibe una nueva demanda de paso, es atendida pero asegurando siempre que el intervalo mínimo de paso de automóviles sea de 40".

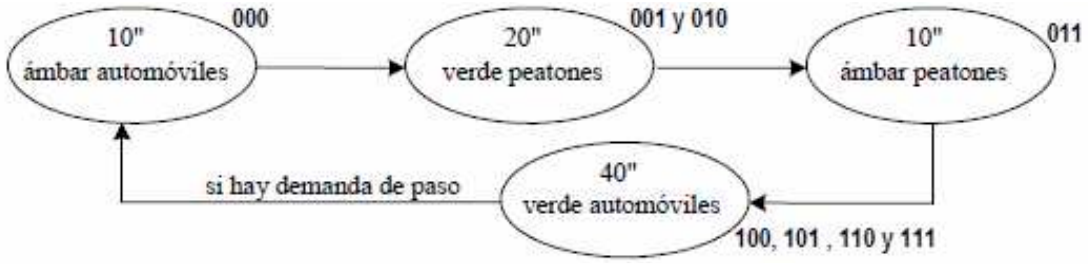


Figura 2. 11: Diagrama de estados del semáforo
Elaborado por: Los autores

Se utiliza un biestable RS para recoger la demanda de paso por parte de los peatones; dicho biestable se borra en el intervalo de ámbar para peatones (que es cuando se completa el paso de peatones, en respuesta a una demanda anterior). El reloj del sistema es de 1 Hz (1 segundo de período).

El ciclo comienza por el servicio a la demanda de paso (ámbar para automóviles) y dura un total de 80"; mientras hay solicitudes de paso se ejecuta normalmente el ciclo completo, pero, si no hay demanda, el ciclo se detiene al final del mismo (cuarto intervalo de 10" de paso de automóviles) y permanece en dicha situación (paso de automóviles) hasta que se produce una petición de paso por parte de peatones.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity PEATONES is
    port (CK, RS,DEMANDA                                    : in std_logic;
AMBAR,ROJA,VERDE,PAMBAR,PROJA ,PVERDE    : out std_logic);
end PEATONES;
architecture TEMPORIZADOR of PEATONES is
signal contador_1            : std_logic_vector(3 downto 0);
signal contador_2            : std_logic_vector(2 downto 0);
signal activo                : std_logic;
begin
-- biestable para guardar la solicitud de paso
SOLICITUD: process(activo, RS, contador_2, DEMANDA)
begin
if RS = '1' or contador_2 = "011" then       activo <= '0';
elsif DEMANDA = '1' then             activo <= '1';
end if;
end process;
TEMPORIZACION: process
-- el contador_1 divide por 10: pasa del reloj de 1" a 10"
begin
wait until CK = '1';
```

```
if RS = '1'        then contador_2 <= "000";          contador_1 <= "0000";
elsif contador_1 = "1001" then                        -- cada 10"
if contador_2 = "111" then                            -- fin de ciclo
if activo ='1' then                                   contador_2 <= "000";
 contador_1 <= "0000"; end if;
else                                                  contador_2 <= contador_2 + 1;
 contador_1 <= "0000";
end if;
      else                                            contador_1 <= contador_1 + 1;
end if;
end process;
SALIDAS: process(contador_2)
begin
VERDE <= '0';           AMBAR <= '0';          ROJA<= '0';
PVERDE <= '0';          PAMBAR <= '0';         PROJA<= '0';
case contador_2 is    when "000" =>       AMBAR<= '1';      PROJA<='1';
                      when "001" =>       ROJA<= '1';       PVERDE<='1';
                       when "010" =>      ROJA<= '1';       PVERDE<= '1';
                      when "011" =>       ROJA<= '1';       PAMBAR<= '1';
                      when others =>      VERDE<= '1';      PROJA<= '1';
-- paso de automóviles :others ≡ contador_2 de 100 a 111: 40"
end case;
end process;
end TEMPORIZADOR;
```

2.5. Descripción de grafos de estados

La evolución del estado de un sistema secuencial se describe muy bien con la asignación múltiple case para referirse a cada uno de los estados y, dentro de ella, utilizando adecuadamente la asignación condicional if para las transiciones. Existen diversas posibilidades para asignar nombres y códigos binarios a los estados; si se prefiere puede dejarse al compilador la tarea de codificar los estados. A continuación se detallan las descripciones VHDL de varios sistemas secuenciales, a partir de sus grafos de estados.

2.5.1. Autómata de Moore

Para la aplicación correcta del autómata de Moore, necesitaremos de un ejemplo básico, se trata de un semáforo de aviso de paso de tren en un cruce de vía única bidireccional con un camino; la vía posee, a una distancia adecuadamente grande, sendos detectores de paso de tren a y b; los trenes circulan por ella en ambas direcciones y se desea que el semáforo señale presencia de tren desde que éste alcanza el primer sensor en su dirección de marcha hasta que pasa por el segundo sensor tras abandonar el cruce. En la figura 2.16 se muestra la solución mediante el diagrama de estados.

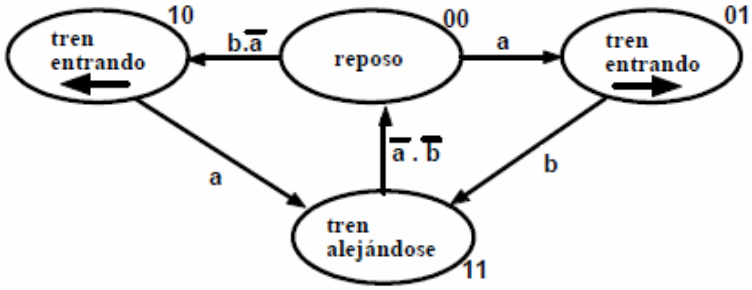


Figura 2. 12: Diagrama de estados del semáforo bidireccional.
Elaborado por: Los autores

```
library ieee;
use ieee.std_logic_1164.all;
entity SEMAFORO is
port (              CK,RS,A,B     : in std_logic;
SEMF: out std_logic);
end SEMAFORO;
architecture GRAFO of SEMAFORO is
-- definición de los estados
subtype mis_estados is std_logic_vector(1 downto 0);
constant reposo                 : mis_estados :="00";
constant entra_por_a            : mis_estados :="01";
constant entra_por_b            : mis_estados :="10";
constant saliendo               : mis_estados :="11";
signal estado    : mis_estados;
begin
-- evolución del estado:
SECUENCIAL: process
```

```
Begin
wait until CK = '1';
if ( RS = '1' ) then                                    estado <= reposo;
else      case estado is
          when reposo=>      if (A='1') then       estado<= entra_por_a;
                             elsif (B='1') then estado<=entra_por_b; end if;
          when entra_por_a=>if (B='1') then    estado <= saliendo; end if;
          when entra_por_b=>if (A='1') then    estado <= saliendo; end if;
          when saliendo=>if (A='0') and (B='0') then estado<=reposo; end if;
when others =>
      end case;
end if;
end process;
-- funciones de activación de las salidas:
SEMF <= '0' when estado = reposo else '1';
end GRAFO;
```

2.5.2. Autómata de Mealy

Sean dos carritos motorizados que se mueven linealmente, entre sendos detectores ‘a’ y ‘b’ el primero; y, entre ‘c’ y ‘d’ el segundo, de forma que, al activas un pulsador P, ambos carritos inician el movimiento desde ‘a’ y ‘c’ y el primero en alcanzar el otro extremo ‘b’ o ‘d’, espera a que el otro alcance el suyo, para iniciar juntos el movimiento de vuelta. Un diagrama de transición de estados detallado del sistema de dos carritos puede incluir siete estados (como autómata de Moore) pero puede ser simplificado dando como resultado el diagrama mostrado por la figura 2.17 (autómata de Mealy: la necesidad de memoria se limita a distinguir entre dos situaciones: el movimiento de ida hacia b y d y el de vuelta hacia a y c):

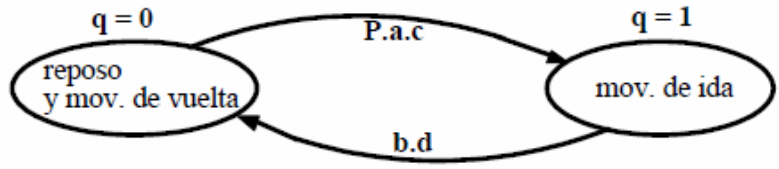


Figura 2. 13: Diagrama de estados para la máquina de Mealy.
Elaborado por: Los autores

```
library ieee;
use ieee.std_logic_1164.all;
entity CARRITOS is
port (      CK,RS,A,B,C,D,P                    : in std_logic;
  iz_1,der_1,iz_2,der_2              : out std_logic);
end CARRITOS;
architecture GRAFO of CARRITOS is
type mis_estados is (vuelta_y_reposo, ida);
signal estado: mis_estados;
begin
-- evolución del estado:
SECUENCIAL: process
begin
wait until CK = '1';
if ( RS = '1' ) then              estado <= vuelta_y_reposo;
else case estado is
when vuelta_y_reposo => if (P and A and C)='1' then estado<= ida; end if;
when ida => if (B and D) = '1' then estado <= vuelta_y_reposo;
end if;
end case;
end if;
end process;
-- funciones de activación de las salidas:
der_1          <= '1' when (estado = ida) and (B='0') else '0';
iz_1           <= '1' when (estado = vuelta_y_reposo) and (A='0') else '0';
der_2          <= '1' when (estado = ida) and (D='0') else '0';
iz_2           <= '1' when (estado = vuelta_y_reposo) and (C='0') else '0';
end GRAFO;
```

2.5.3. Empleo conjunto de Moore y Mealy

Para emplear conjuntamente el diseño secuencial por Moore y Mealy (ver figura 2.18) realizaremos una aplicación práctica como el de un depósito con mezcla de 3 líquidos, el depósito se llena con una mezcla de tres líquidos diferentes, para lo cual dispone de tres electroválvulas A, B, C que controlan la

salida de dichos líquidos y de cuatro detectores de nivel n1, n2, n3, n4, siendo n1 el inferior y n5 el de llenado máximo. Solamente cuando el nivel del depósito desciende por debajo del mínimo n1 se produce un ciclo de llenado: primero con el líquido A hasta el nivel n2, luego el líquido B hasta el nivel n3 y, finalmente, el líquido C hasta completar el depósito n4. El diagrama de transición de estado correspondiente al autómata de Moore se muestra en la figura 2.19.

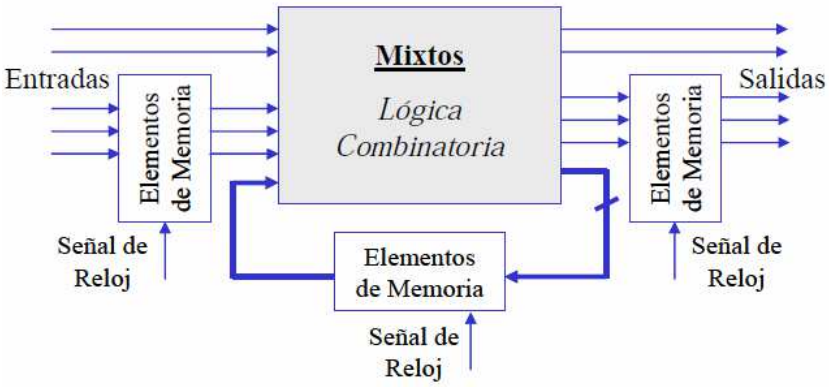


Figura 2. 14: Diagrama de estados del semáforo bidireccional.
Elaborado por: Los autores

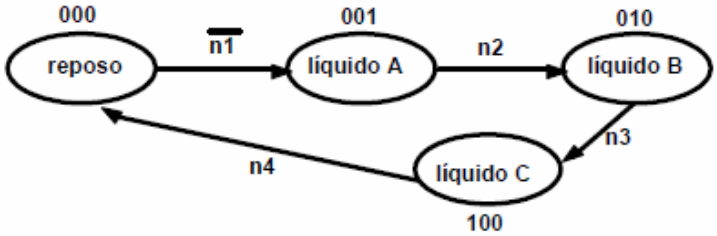


Figura 2. 15: Diagrama de estados del semáforo bidireccional.
Elaborado por: Los autores

```
library ieee;
use ieee.std_logic_1164.all;
entity DEPOSITO is
  port (          CK,RS,n1,n2,n3,n4          : in std_logic;
A,B,C                                        : out std_logic);
end DEPOSITO;
architecture MOORE of DEPOSITO is
subtype mis_estados is std_logic_vector(3 downto 1);
-- código de un solo uno
constant reposo                 : mis_estados :="000";
```

```
constant liquido_A              : mis_estados :="001";
constant liquido_B              : mis_estados :="010";
constant liquido_C              : mis_estados :="100";
signal estado                   : mis_estados;
begin
-- funciones de activación de las salidas:
A <= estado(1);         B <= estado(2);      C <= estado(3);
-- evolución del estado:
SECUENCIAL: process
begin
wait until CK = '1';
if ( RS = '1' ) then                                    estado <= reposo;
else              case  estado is
    when reposo => if (n1 = '0') then estado <= liquido_A; end if;
            when liquido_A => if (n2 = '1') then estado <= liquido_B; end if;
    when liquido_B => if (n3 = '1') then estado <= liquido_C; end if;
    when liquido_C => if (n4 = '1') then estado <= reposo; end if;
    when others =>
end case;
end if;
end process;
end MOORE;
```

Este mismo sistema secuencial puede configurarse con un número más reducido de estados, según el diagrama de estados mostrado por la figura 2.20 que corresponde a un autómata de Mealy (en este caso se necesita solamente una variable de estado q, pero las funciones de activación de las salidas resultan más complejas, pues dependen de las entradas, de la información que aportan los detectores de nivel):

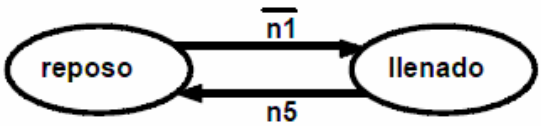


Figura 2. 16: Diagrama de estados del semáforo bidireccional.
Elaborado por: Los autores

```
architecture MEALY of DEPOSITO is
type mis_estados is (reposo, llenado);
signal estado: mis_estados;
begin
-- evolución del estado
SECUENCIAL: process
begin
wait until CK = '1';
if ( RS = '1' ) then                              estado <= reposo;
else case estado is
when reposo =>        if (n1 = '0') then      estado <= llenado ;   end if;
when llenado =>     if (n4 = '1') then        estado <= reposo ;    end if;
   end case;
end if;
end process;
-- funciones de activación de las salidas:
A         <= '1' when (estado = llenado) and (n2='0') else '0';
B         <= '1' when (estado = llenado) and (n2='1') and (n3 ='0') else '0';
C         <= '1' when (estado = llenado) and (n3='1') else '0';
end MEALY;
```

CAPÍTULO 3: FPGA Y QUARTUS II DE ALTERA

En el presente capítulo se describirá la tecnología FPGA y la herramienta de programación Quartus II que permite fundamentar la programación en VHDL en la asignatura de Sistemas Digitales II en la carrera de Ingeniería en Telecomunicaciones.

3.1. INTRODUCCIÓN A LOS DISPOSITIVOS FPGA

El continuo avance de la Microelectrónica a finales de los años 70 dio como resultado circuitos de muy gran escala de integración VLSI (*Very Large Scale of Integration*). Esto permitió el desarrollo de diferentes tipos de circuitos y sistemas electrónicos digitales entre los que cabe citar:

a) Los circuitos combinacionales programables, en sus dos versiones:
   - Matrices lógicas programables (PLA, *Programmable Logic Array*) [PHILIPS, 1987a].
   - Matrices lógicas Y-programables (PAL, *Programmable Array Logic*) [MM, 1981] [MM, 1986].

   La combinación de estas matrices con un registro síncrono de entrada y salida en paralelo dio lugar a los secuenciadores lógicos programables (PLS, *Programmable Logic Sequencer*) [MM, 1981] [PHILIPS, 1987a], cuyo esquema básico se representa en la figura 3.1.

b) Los circuitos integrados semimedida [CHAM, 1988] [GEIGER, 1990] [RANDELL, 1982] que utilizan celdas predefinidas cuya interconexión se programa durante el proceso de fabricación. Las celdas pueden ser de dos grandes tipos:
   - Básicas, como por ejemplo puertas NO-AND, dando lugar a los conjuntos programables de puertas lógicas (PGA, *Programmable Gate Array*) [PHILIPS, 1987b] [TEXAS, 1990].
   - Bloques funcionales que dan lugar a las celdas estándar ("standardcells") [PHILIPS, 1987b] [TEXAS, 1990].

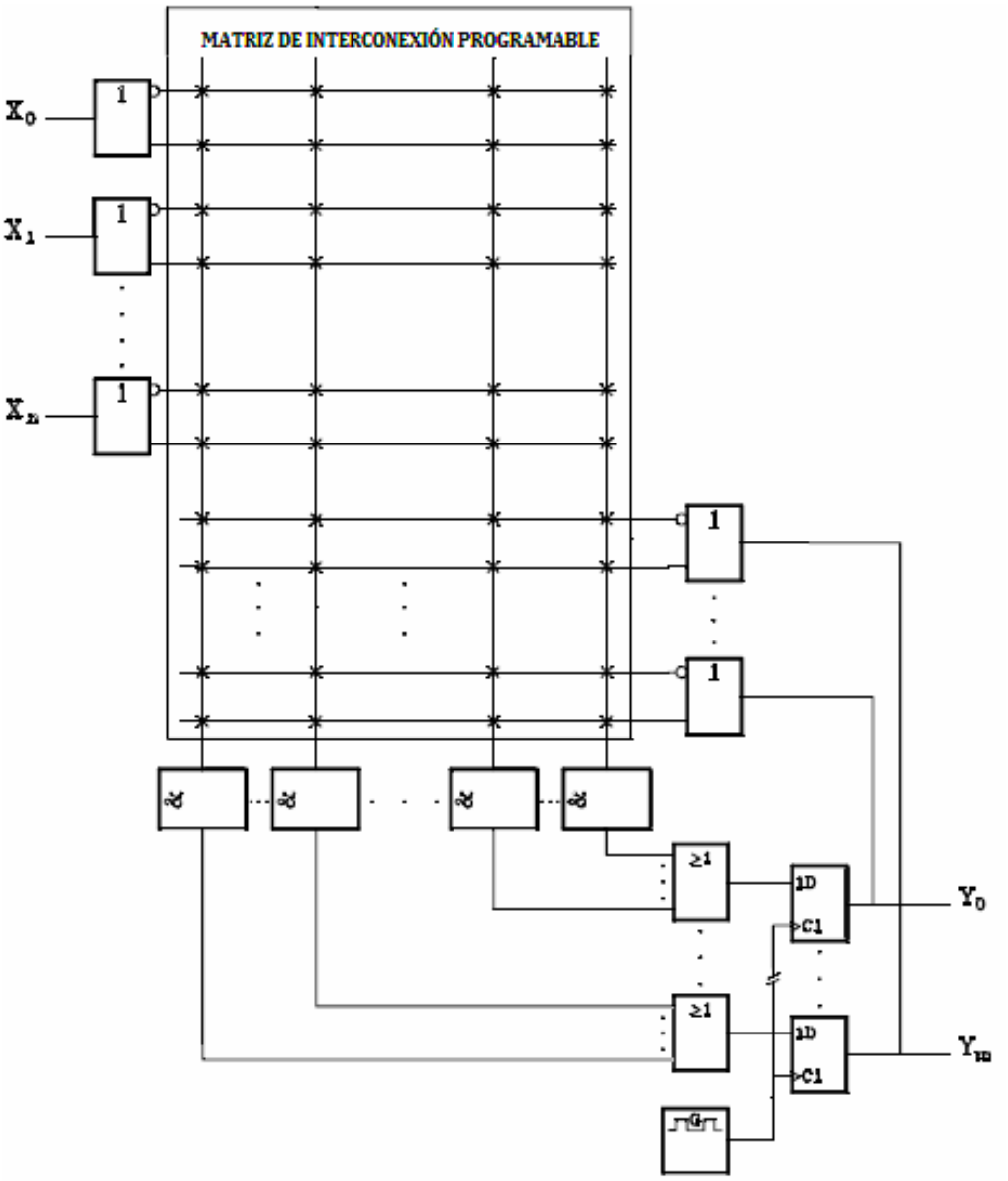


Figura 3. 1: Estructura básica de un secuenciador lógico programables

Fuente: www.cypress.com

A pesar de todos los avances indicados, los circuitos integrados descritos no proporcionan una respuesta totalmente satisfactoria a la resolución de algunos problemas prácticos. Esto se debe a que los circuitos que presentan suficiente versatilidad, por ejemplo, modularidad y velocidad para ciertas aplicaciones, resultan muy caros y los circuitos de menor coste no presentan las prestaciones necesarias [AGRAWAL, 1986] [LUPON, 1988].

Desde su creación, en la década de los 1970s, los PLDs se han caracterizado por ofrecer la mayor flexibilidad de diseño, ya que son dispositivos cuya arquitectura interna es predeterminada por el fabricante, pero que son creados de tal forma que puedan ser configurados por los ingenieros en campo, para ejecutar una variedad de funciones [XilinxDK, 2003]. Después de las primeras versiones de PLDs, donde se ofrecieron arquitecturas con planos AND/OR programables y diversos niveles de interconexión de suma de productos y arquitecturas complejas (CPLDs), que extendieron la densidad de los primeros componentes.

Este dispositivo, concebido por Ross Freeman, fundador de Xilinx, revolucionó la tecnología al abandonar la restricción de suma de productos y utilizar módulos combinacionales configurables a los que llamó LUT (Look-Up-Table), cada uno acompañado por un Flip-Flop (FF) e interconectado por conexiones programables [Xcell, 2001].

Los dispositivos FPGA son circuitos integrados digitales que contienen bloques de lógica programable e interconexiones configurables entre estos bloques. Dependiendo de su realización, algunas FPGA pueden ser programadas una sola vez y otras pueden ser reprogramadas un sinnúmero de veces. El concepto de “programables en campo”, se refiere al hecho de que su configuración puede tomar lugar en el laboratorio o en el sistema electrónico que lo incluye [Maxfield04] y el concepto “matriz de puertas”, hace alusión a la estructura interna que posibilita su reprogramación.

En la figura 3.2 se muestra las principales aproximaciones ordenándolas en función de los parámetros coste, flexibilidad, prestaciones y complejidad. Asimismo podemos observar, que las mejores prestaciones las proporciona un diseño *full-custom*, consiguiéndose a costa de elevados costes y enorme complejidad de diseño. En el otro extremo del abanico de posibilidades se encuentra la implementación software, que es muy barata y flexible, pero que en determinados casos no es válida para alcanzar un nivel de prestaciones relativamente alto.

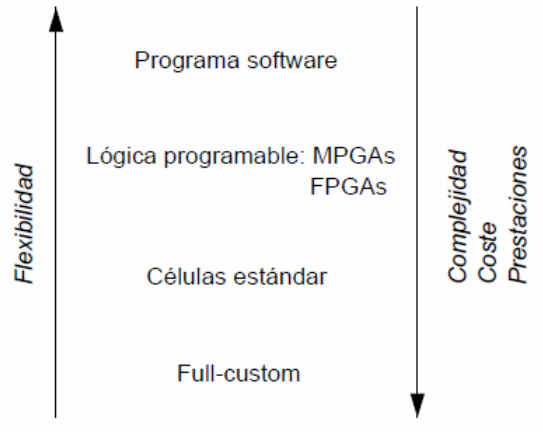


Figura 3. 2: Variedad de soluciones para diseño de circuito digitales.
Fuente: Maxinez G. David

3.2. EVOLUCIÓN DE LOS DISPOSITIVOS PROGRAMABLES.

Se entiende por dispositivo programable aquel circuito de propósito general que posee una estructura interna que puede ser modificada por el usuario final (o a petición suya, por el fabricante) para implementar una amplia gama de aplicaciones. El primer dispositivo que cumplió estas características era una memoria PROM, que puede realizar un comportamiento de circuito utilizando las líneas de direcciones como entradas y las de datos como salidas (implementa una tabla de verdad). Hay dos tipos básicos de PROM:

a. Programables por máscara (en la fábrica), proporcionan mejores prestaciones. Son las denominadas de conexiones hardwired.
b. Programables en el campo (field) por el usuario final. Son las EPROM y EEPROM. Proporcionan peores prestaciones, pero son menos costosas para volúmenes pequeños de producción y se pueden programar de manera inmediata.

El PLD, Programmable Logic Device, es una matriz de puertas AND conectada a otra matriz de puertas OR más biestables. Cualquier circuito lógico se puede implementar, por tanto, como suma de productos. La versión más básica del mismo es una PAL, con un plano de puertas AND y otro fijo de puertas OR. Las salidas de estas últimas se pueden pasar por un biestable en la mayoría de los circuitos del mercado.

Ventaja: son bastante eficientes si implementan circuitos no superiores a unos centenares de puertas.

Inconvenientes: arquitectura rígida, y está limitado por un número fijo de biestables y entradas/salidas.

La PLA, Programmable Logic Array, es más flexible que la PAL: se pueden programar las conexiones entre los dos planos. Estos dispositivos son muy simples y producen buenos resultados con funcionalidades sencillas (sólo combinacional). Hace falta algo un poco más sofisticada y general: una matriz de elementos variados que se puedan interconectar libremente. Este es el caso de una MPGA (Mask-Programmable Gate Array), cuyo principal representante está constituido por un conjunto de transistores más circuitería de E/S. Se unen mediante pistas de metal que hay que trazar de forma óptima, siendo ésta la máscara que hay que enviar al fabricante.

Las FPGA (Field Programmable Gate Array), introducidas por Xilinx en 1985, son el dispositivo programable por el usuario de más general espectro. También se denominan LCA (Logic Cell Array). Consisten en una matriz bidimensional de bloques configurables que se pueden conectar mediante recursos generales de interconexión. Estos recursos incluyen segmentos de pista de diferentes longitudes, más unos conmutadores programables para enlazar bloques a pistas o pistas entre sí. En realidad, lo que se programa en una FPGA son los conmutadores que sirven para realizar las conexiones entre los diferentes bloques, más la configuración de los bloques.

3.3. Estructura general de las FPGAs

Las FPGA se caracterizan principalmente por su organización lógica (ver figura 3.3) tamaño, recursos especiales de procesamiento, almacenamiento y control y por su velocidad y consumo de energía. Las FPGA están basadas en una matriz de bloques lógicos programables (también llamados bloques lógicos configurables CLBs o bloques matriciales lógicos LABs) acoplados por líneas de interconexión y cajas de conmutadores.

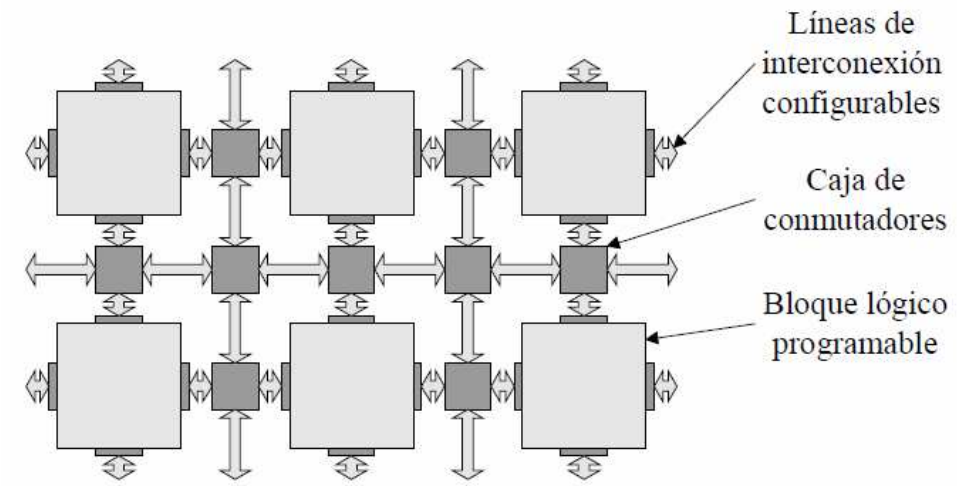


Figura 3. 3: Organización lógica de un FPGA
Fuente: http://www.annapmicro.com

Las funciones digitales que el usuario define son mapeadas a uno o más de los bloques lógicos. Las líneas de interconexión configurables forman parte de los recursos de rutado del dispositivo, los cuales son la clave de la flexibilidad de la FPGA, pero que también representan un compromiso entre flexibilidad de programación y eficiencia de área. El rutado incluye típicamente una jerarquía de canales que van desde las líneas de alta velocidad, hasta las dedicadas para la difusión de la señal de reloj y reset. Los conmutadores programables, que pueden ser basados en RAM, borrables eléctricamente o programables una sola vez, habilitan la conexión de las líneas de rutado y de los recursos internos y elementos externos, reduciendo al mínimo el retardo de red.

El proceso de diseño de un circuito digital utilizando una matriz lógica programable puede descomponerse en dos etapas básicas:

a. Dividir el circuito en bloques básicos, asignándolos a los bloque configurables del dispositivo.
b. Conectar los bloques de lógica mediante los conmutadores necesarios.

Para ello el fabricante proporciona las herramientas de diseño adecuadas. Los elementos básicos constituyentes de una FPGA como las de Xilinx se aprecian en la figura 3.4 y son los siguientes:

a. Bloques lógicos, cuya estructura y contenido se denomina arquitectura.
b. Recursos de interconexión, cuya estructura y contenido se denomina arquitectura de rutado.
c. Memoria RAM, que se carga durante el RESET para configurar bloques y conectarlos.

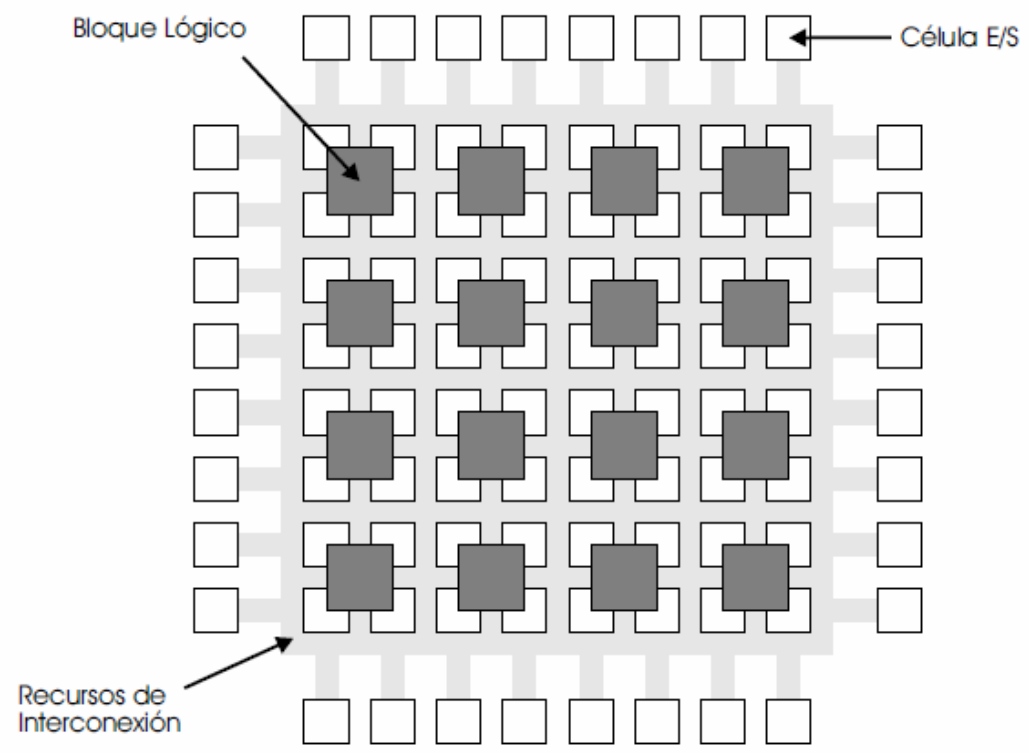


Figura 3. 4: Estructura general de un FPGA
Fuente: http://www.annapmicro.com

Entre las numerosas ventajas que proporciona el uso de FPGAs dos destacan principalmente: el bajo coste de prototipado y el corto tiempo de producción. No todo son ventajas. Entre los inconvenientes de su utilización están su baja velocidad de operación y baja densidad lógica (poca lógica implementable en un solo chip). Su baja velocidad se debe a los retardos introducidos por los conmutadores y las largas pistas de conexión.

Por supuesto, no todas las FPGA son iguales. Dependiendo del fabricante nos podemos encontrar con diferentes soluciones. Las FPGAs que existen en la actualidad en el mercado se pueden clasificar como pertenecientes a cuatro grandes familias, dependiendo de la estructura que adoptan los bloques lógicos que tengan definidos.

Las cuatro estructuras mostradas en la figura 3.5, sin necesidad de que aparezcan en la misma los bloques de entrada/salida.

- Matriz simétrica, como son las de XILINX
- Basada en canales, ACTEL
- Mar de puertas, ORCA
- PLD jerárquica, ALTERA o CPLD's de XILINX.

En concreto, para explicar el funcionamiento y la estructura básica de este tipo de dispositivos programables sólo se considerarán las distintas familias de ALTERA.

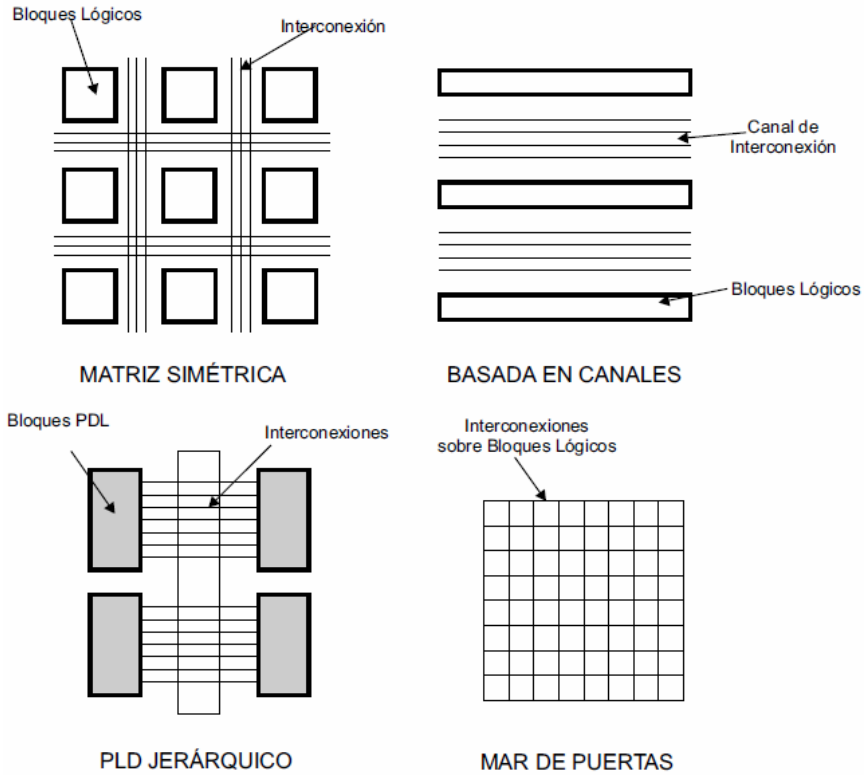


Figura 3. 5: Tipos de FPGAs
Fuente: http://www.annapmicro.com

3.4. METODOLOGÍA DEL DISEÑO FPGA.

La metodología de diseño inicia con la lógica para atender el proceso de descripción y representación de un producto final o un componente para un sistema mayor. En términos generales, existen dos métodos básicos, el diseño de abajo hacia arriba (bottom-up) y el diseño de arriba hacia abajo (top-down) [Pardo00].

Para un diseño hardware, una vez concebida la idea, el flujo de diseño se apoya en las herramientas de diseño asistido por computadora (CAD). En un nivel superior, que puede incluir también el diseño software, las herramientas se engloban en los sistemas de automatización del diseño electrónico (EDA), las cuales integran en el mismo marco de trabajo, tanto las herramientas de descripción como las de simulación, síntesis, realización y verificación.

3.4.1. Metodologías de descripción.

En la metodología botton-up, la descripción del sistema inicia con los componentes más pequeños, para posteriormente agruparlos en diferentes módulos y estos en otros módulos hasta llegar a la representación del sistema completo. El proceso no implica una estructuración jerárquica de los elementos del sistema, y esta solo se puede definir al término de la descripción del módulo superior. La descripción parte de los componentes de más bajo nivel, que representan unidades funcionales con significado propio dentro del diseño (primitivas). En un sistema electrónico, las primitivas pueden ser chips, transistores, resistencias y condensadores, entre otros elementos [Pardo, 2000].

La metodología top-down consiste en capturar una idea en un alto nivel de abstracción y realizarla, dividiéndola en módulos y sub-módulos cada vez con mayor grado de detalle y con una funcionalidad determinada, hasta llegar a las primitivas del sistema. Este es el principio de que un problema muy complejo puede ser dividido en varios sub-problemas y estos en otros problemas mucho más sencillos de resolver [Pardo, 2000].

Los sistemas EDA han evolucionado hacia la metodología top-down, ya que presenta las ventajas de incrementar la productividad de los diseñadores, aumentar la reutilización de los módulos intermedios del diseño y detectar los errores en etapas tempranas del ciclo de desarrollo.

3.4.2. Flujograma de diseño FPGA.

La disponibilidad de las herramientas EDA han facilitado el desarrollo de sistemas digitales con lógica programable. En el caso de las FPGA, se tiene un

flujo de diseño que incluye las etapas de especificación, verificación, realización y depuración, en un proceso totalmente automatizado (ver figura 3.6) [XilinxDK03].

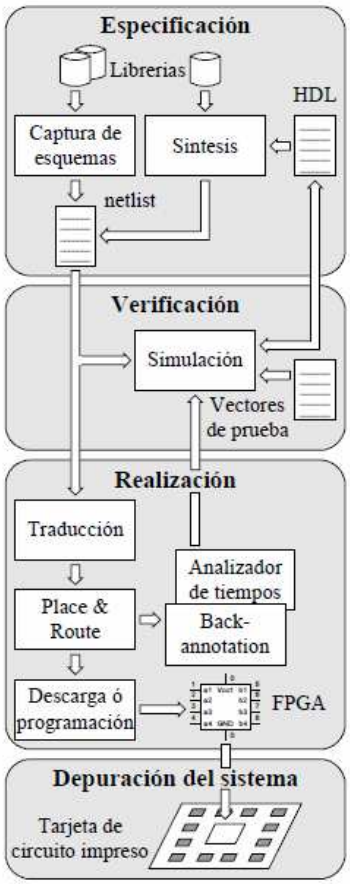


Figura 3. 6: Flujograma del diseño.
Fuentes: Los Autores

Anteriormente, los diseños se especificaban completamente por medio de esquemas, pero dada la complejidad de los sistemas actuales, se prefiere utilizar lenguajes de descripción de hardware (HDL) como el VHDL y el Verilog, dejando los esquemas para la representación de los módulos de jerarquías superiores. Los HDLs permiten describir en forma de texto la función o comportamiento del circuito, en vez de hacerlo gráficamente y a bajo nivel.

La descripción HDL es procesada por la herramienta de síntesis para obtener la representación del comportamiento al nivel de componentes electrónicos. La síntesis es el proceso de adaptación de la lógica de diseño a los recursos lógicos disponibles en el chip, partiendo de la especificación de entrada con alto grado de abstracción y de las restricciones de área y tiempo

definidas por el diseñador y llegando a una descripción menos abstracta, mas enfocada al dispositivo.

En este punto del flujo de diseño, solo se puede realizar una simulación funcional, en la cual se verifica la operación correcta de la descripción HDL. Si los resultados de salida no son correctos, se debe modificar la descripción del diseño y repetir el ciclo hasta que la funcionalidad sea satisfactoria. La etapa de realización tiene como propósito obtener un dispositivo FPGA programado con la operación lógica que describe el *netlist*, por medio de la secuencia de procesos de traducción (*translate*), mapeo (*map*), emplazamiento (*place*), rutado (*route*) y programación (*programming*).

En la traducción se integran los programas para importar el netlist y fusionarlo con las restricciones del diseño definidas por el usuario, para obtener un archivo que describe el diseño lógico a partir de primitivas. En el mapeo, el diseño se ajusta a los recursos disponibles del dispositivo meta, relacionando los elementos lógicos del diseño con los recursos físicos del chip. El emplazamiento es el proceso de asignar los módulos o bloques lógicos del diseño a elementos específicos de la FPGA.

En el rutado se interconectan los elementos lógicos seleccionados para que cumplan con la función lógica del diseño. La programación consiste en descargar a la FPGA la información de configuración que define la funcionalidad diseñada. En este punto del ciclo el dispositivo puede estar ya trabajando, pero todavía es necesario depurar su funcionamiento dentro del sistema que lo contiene y bajo las condiciones del entorno donde trabajará.

Este es el momento de resolver situaciones como funcionalidades no consideradas, operaciones incorrectas o requerimientos de E/S fuera de norma. La ventaja de la tecnología FPGA es que se puede regresar a las etapas iniciales del ciclo de desarrollo para resolver este tipo de anomalías, sin perjudicar significativamente los planes de tiempos y costes del proyecto.

3.5. Herramientas de diseño para FPGA.

En el mercado se encuentran gran cantidad de herramientas EDA para el diseño con FPGA y VHDL. Algunas de ellas son de uso público (*open-source*), otras cubren casi todos los dispositivos del mercado, como las de Mentor *Graphics*, *Exemplar*, *Synopsys* y *Synplicity*. Las más son especificas de las compañías de dispositivos, como las de *Actel*, **Altera**, *Cypress* y *Xilinx*. La mayoría de las herramientas funcionan sobre PC y algunas están disponibles para *workstation*.

Ya que las herramientas específicas de cada compañía están optimizadas para las arquitecturas de sus dispositivos, además de cubrir toda la gama de posibilidades de diseño que sus FPGA ofrecen, desde la síntesis hasta el desarrollo de sistemas embebidos, esta sección se orienta a las herramientas de la empresa Xilinx, al ser esta el fabricante de los dispositivos utilizados en la realización de las arquitecturas hardware que se proponen en este trabajo.

3.5.1. Herramientas de síntesis.

La síntesis es uno de los pasos más esenciales en la metodología de diseño con FPGA, por lo que se necesitan utilizar las técnicas del estado del arte para generar la mejor representación lógica para el dispositivo seleccionado, a partir de la definición conceptual del diseño.

3.5.1.1. Síntesis HDL con enfoque físico.

Las herramientas de síntesis que salieron al mercado a mediados de los 1980s, referidas como tecnología de síntesis HDL, toman una descripción RTL de un diseño ASIC junto con un conjunto de restricciones de tiempo y generan un netlist al nivel de puertas, en un proceso de minimización y optimización. A mediados de los 1990s, estas herramientas fueron extendidas para incluir las arquitecturas de las FPGA, para producir un netlist al nivel de LUTs y bloques lógicos configurables (LUT/CLB). En esa época, los chips eran diseñados con tecnología del estado sólido de baja resolución, por lo que el retardo de rutado tenia poco peso en el retardo de red. Por lo mismo, las herramientas de síntesis utilizaban modelos simples para evaluar los efectos de los retardos de rutado.

En los chips actuales con tecnología de sub-micrones, donde el retardo de rutado representa hasta el 80% del retardo de red, estas herramientas no estimarían adecuadamente los tiempos del diseño. A partir de 1996 se consideraron herramientas de síntesis con enfoque físico en el flujo de diseño ASIC. El flujo de diseño de las FPGA también las adoptó a principios del 2000. Estas herramientas de síntesis utilizan información de emplazamiento de los elementos lógicos del diseño asociada al dispositivo objetivo, para estimar los retardos de rutado lo más pronto posible en el proceso de síntesis.

Actualmente, las herramientas EDA inician el proceso de síntesis con un paso de síntesis HDL (para obtener la información de emplazamiento de los elementos lógicos) y continúan con un paso de síntesis con enfoque físico (Figura 3.7) [Maxfield04].

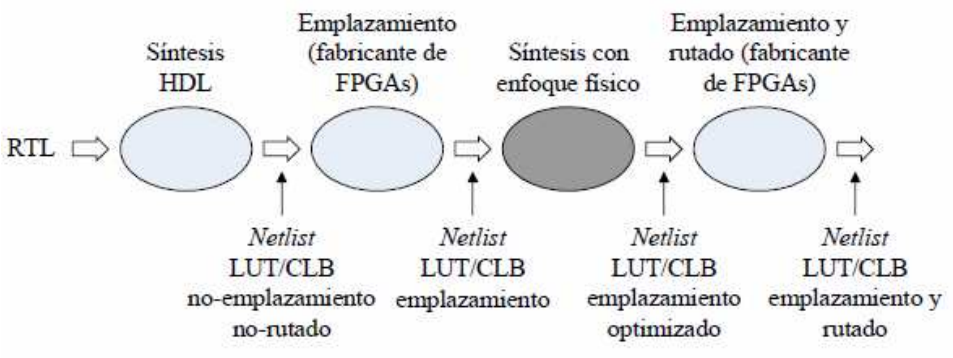


Figura 3. 7: Flujograma del diseño.
Fuentes: Los Autores

3.6. Arquitectura de los Dispositivos de Altera

Inicialmente debemos diferenciar la clasificación en CPLDs y FPGAs utilizadas por Altera con la descrita en nuestro trabajo en puntos anteriores. Nosotros describimos la arquitectura de una CPLD como una agrupación de PALs o GALs, interconectadas entre sí, donde cada bloque lógico tiene una parte combinacional compuesta por matrices de compuertas AND y OR, más un registro asociado al pin de entrada/salida. En cambio la arquitectura de la FPGA la describimos también como un bloque lógico con una parte combinacional y una parte secuencial, en el cual la parte combinacional es mucho más simple que la de una de las SPLD interna de una CPLD; ya que pude ser basado en LUTs o en multiplexores.

Altera, por su parte, diferencia las CPLDs y las FPGAs por diferentes 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 las CPLDs para proveer conexiones de celda lógica a celda lógica que lleva finalmente a velocidades más altas comparable con las FPGAs. De esta forma Altera ofrece las siguientes familias de CPLDs:

- APEX 20K
- ACEX 1K
- FLEX 10K
- FLEX 8000
- FLEX 6000
- MAX 9000
- MAX 7000
- MAX 5000
- Classic

Con características como que se muestran en la tabla 3.1:

Familia	**Estructura del Bloque Lógico**	**Estructura de Interconexión**	**Tecnología de programación**
Stratix	LUT	Continua	SRAM
Cyclone	LUT	Continua	SRAM
APEX 20K	LUT y termino producto	Continua	SRAM
ACEX 1K	LUT	Continua	SRAM
FLEX 10K	LUT	Continua	SRAM
FLEX 8000	LUT	Continua	SRAM
FLEX 6000	LUT	Continua	SRAM
MAX 9000	termino producto	Continua	EEPROM
MAX 7000	termino producto	Continua	EEPROM
MAX 5000	termino producto	Continua	EPROM
Classic	termino producto	Continua	EPROM

Tabla 3.1: Dispositivos de Altera.
Fuentes: Los Autores

De acuerdo a nuestra clasificación tomaremos a los dispositivos APEX, ACEX y FLEX como FPGAs mientras que los MAX como CPLDs y los Classic como SPLDs. Cabe aclarar que también, como vemos en la tabla 3.1 existen dispositivos nuevos clasificados como FPGAs por Altera, los cuales son Cyclone y Stratix.

3.7. FLEX 10K

Los dispositivos de la familia de dispositivos FLEX 10K de Altera integran elementos de memoria SRAM dispuestos en una matriz reconfigurable de elementos lógicos (FLEX: Flexible Logic Element matriX). Los dispositivos FLEX 10K, con hasta 250000 puertas, proporcionan la densidad, velocidad y prestaciones necesarias para integrar sistemas completos en un único dispositivo. Los dispositivos de 2.5V FLEX 10KE, fabricados en una tecnología de 0.22µm, son entre un 20% y un 30% más rápidos que los de la familia de 0.3µm FLEX10 KA que operan a 3.3V.

De forma similar, los dispositivos FLEX 10KA son, en promedio, entre un 20% y un 30% más rápidos que los de los dispositivos FLEX 10K que operan a 5V y se fabrican en una tecnología de 0.42µm. La arquitectura de los dispositivos FLEX 10K se inspira en el rápido crecimiento que han experimentado en la industria los dispositivos programables basados en matrices de puertas. A su vez, estos dispositivos disponen de zonas designadas para la integración de elementos de memoria que se pueden utilizar para construir funciones de mayor complejidad.

3.7.1. Arquitectura de los dispositivos FLEX 10K

Cada dispositivo de la familia FLEX 10K está integrado por bloques de memoria y una matriz de elementos lógicos. Los bloques de memoria son conocidos como EABs (Embedded Array Blocks) y pueden utilizarse para definir pequeñas memorias o funciones lógicas especiales. Cuando se utiliza como un elemento de memoria, cada EAB proporciona 2048 bits y se puede utilizar para definir memorias RAM, ROM, RAM de doble puerto o funciones *first-in first-out* (FIFO).

Si un EAB se utiliza para definir funciones lógicas complejas tales como multiplicadores, controladores, máquinas de estados o funciones para DSP, contribuye con entre 100 y 600 puertas. Los EABs se pueden utilizar de forma independiente o también pueden utilizarse en modo combinado para realizar funciones más complejas o elementos de memoria de mayor capacidad. La matriz de elementos lógicos (Logic Array) está constituida por bloques lógicos conocidos como LABs (Logic Array Blocks).

Cada LAB agrupa 8 elementos lógicos o LEs (*Logic Elements*) e interconexiones locales. Un LE incluye una tabla de look-up (LUT, Look Up Table) de 4 entradas, un flip-flop programable y lógica para la rápida generación y propagación de acarreo (*carry*) y la conexión en cascada (*cascade*). Los ocho LEs de un LAB se pueden utilizar para definir pequeñas funciones lógicas como contadores y sumadores de hasta 8 bits; además, se pueden agrupar varios LABs para definir bloques lógicos mayores.

Cada LAB representa aproximadamente 96 puertas lógicas. Las conexiones entre elementos de memoria y elementos lógicos se realizan mediante el llamado *FastTrack Interconnect*, una serie de rápidos canales de fila y columna continuos que recorren todo el ancho y el alto del dispositivo. Cada pin de Entrada/Salida se alimenta por un elemento de Entrada/Salida (IOE, *Input/Output element*) localizado al final de cada fila y columna del FastTrack Interconnect. Cada IOE contiene un buffer de Entrada/Salida bidireccional y un flip-flop que pueden usarse tanto como registro de la salida o entrada para alimentar la entrada, la salida, o las señales bidireccionales.

En la figura 3.8 se muestra un diagrama del bloque de la arquitectura de la FLEX 10K. Cada grupo de LEs se combina en un LAB; los LAB's son colocados en las filas y columnas. Cada fila también contiene un solo EAB. Los LAB's y EABs son interconectados por el FastTrack Interconnect. Los IOEs son localizados al final de cada fila y columna del FastTrack Interconnect.

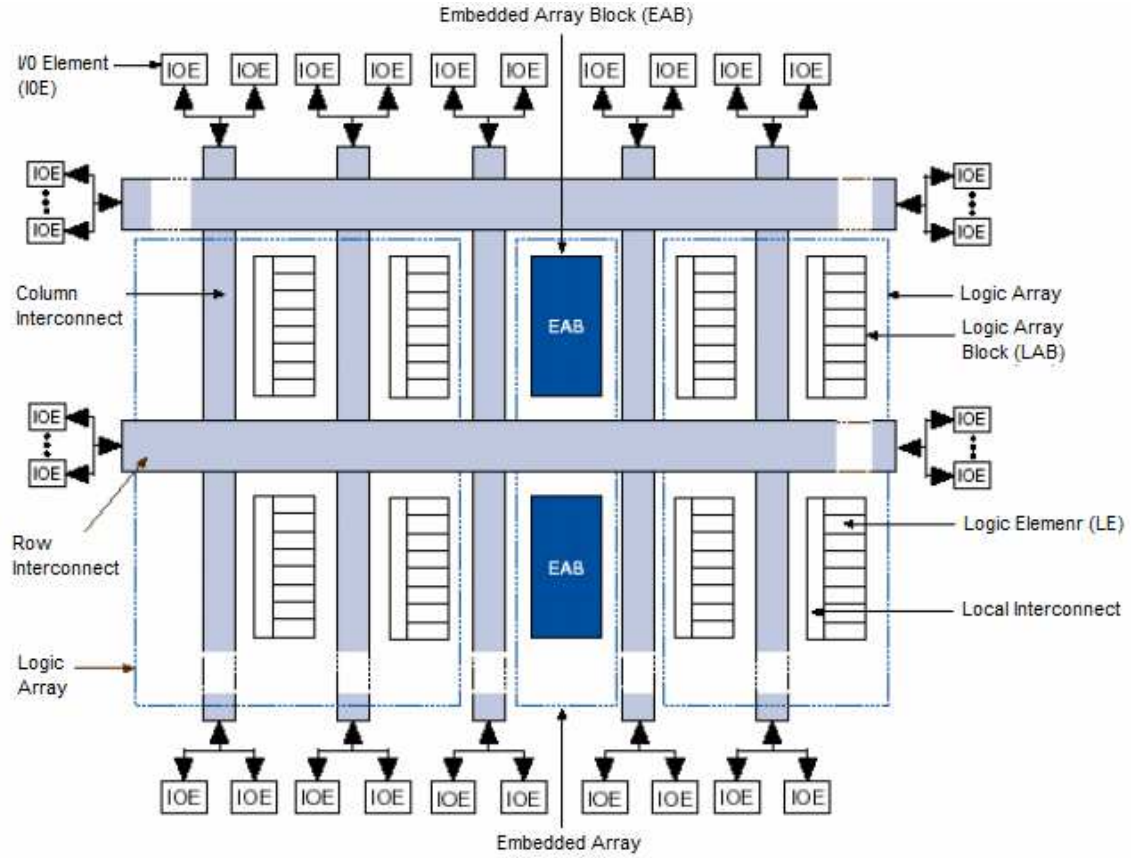


Figura 3. 8: Arquitectura del dispositivo FLEX 10K.
Fuente: http://www.annapmicro.com

La tabla 3.2 muestra los recursos que integran algunos de los miembros de la familia de dispositivos FLEX 10K.

Dispositivo	LEs	EABs	Bits de memoria	Entradas Salidas
EPF10K10	576	3	6K	150
EPF10K20	1152	6	12K	189
EPF10K30	1728	6	12K	246
EPF10K40	2304	8	16K	189
EPF10K50	2880	10	20K	310
EPF10K70	3744	9	18K	358
EPF10K100	4992	12	24K	406
EPF10K130	6656	16	32K	470
EPF10K250	12160	20	40K	470

Tabla 3.2: Dispositivos de FLEX 10K.
Fuente**:** Altera
Elaborado por: Autores

Los dispositivos FLEX 10K proporcionan seis entradas especializadas que manejan los *flip-flops* que controlan las entradas para asegurar la distribución eficaz de alta velocidad, baja distorsión (*skew*) de las señales de control. Estas señales usan canales dedicados al ruteo que proporcionan retrasos más cortos y más bajas distorsiones que el FastTrack Interconnect.

Cuatro de las entradas especializadas manejan cuatro señales globales. Estas cuatro señales globales también pueden ser manejadas por la lógica interior, proporcionando una solución ideal para el divisor del reloj o una señal asincrónica internamente generada que limpia muchos registros en el dispositivo.

3.7.2. Bloques de arreglos integrados (EAB, Embedded Array Block)

La figura 3.9 muestra un esquema interno del elemento de memoria. Cada EAB es una memoria RAM con registros en los puertos de E/S que se puede utilizar de diversas formas. Si se desea construir una función lógica compleja, el EAB se configura con un patrón de sólo lectura y se utiliza como una LUT. De esta manera, la función lógica deseada se almacena en la tabla y no es necesario calcularla para cada valor de entrada.

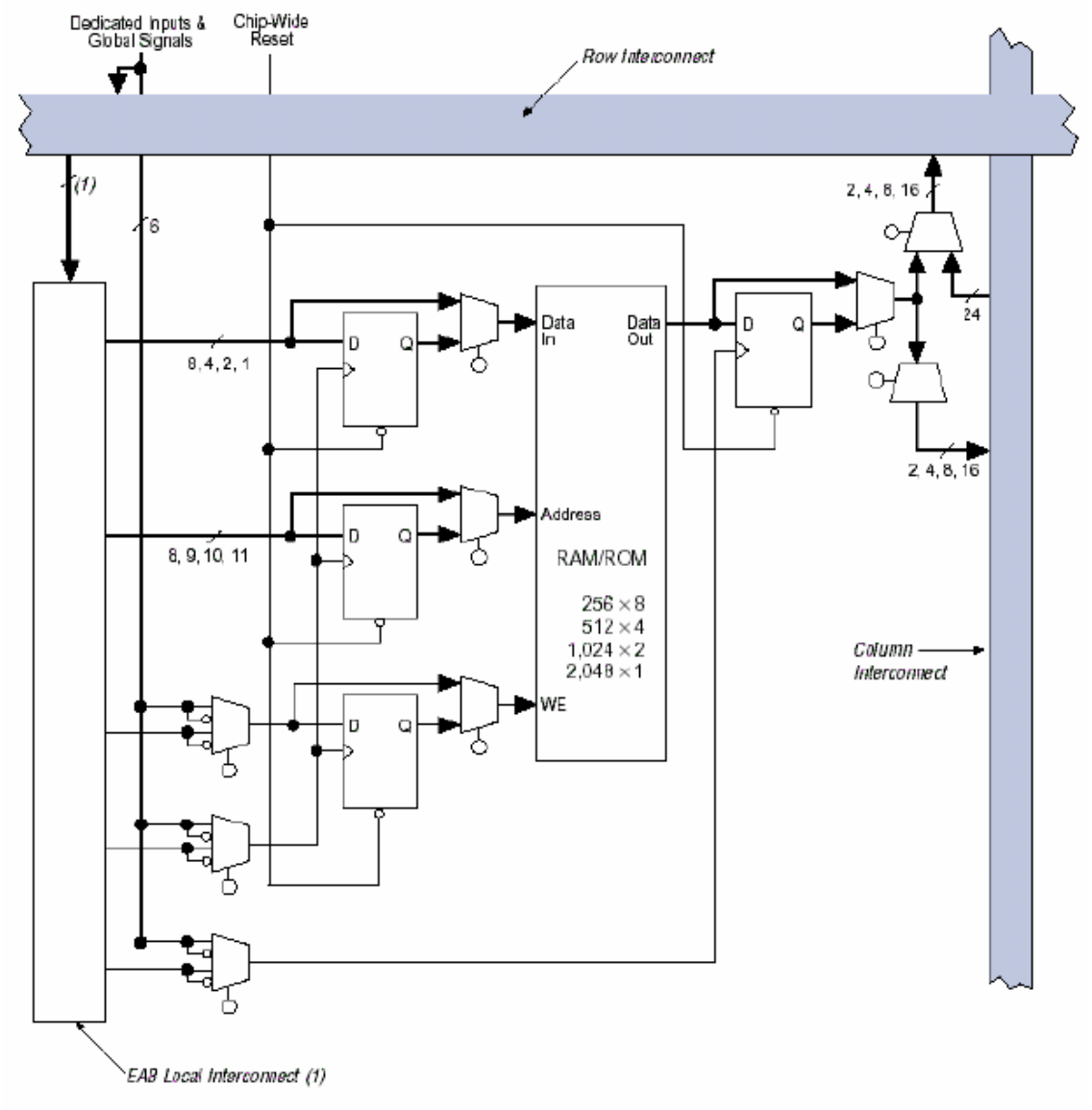


Figura 3. 9: Bloques de arreglos integrado.
Fuente: http://www.annapmicro.com

Las funciones lógicas de elevada complejidad así construidas pueden tener menor retardo que si se construyen haciendo uso de elementos lógicos. Cuando el EAB se utiliza como una memoria RAM, se puede configurar como un bloque de tamaño 256 x 8, 512 x 4, 1024 x 2 o 2048 x 1. Se pueden crear bloques de memoria mayores combinando varios bloques de memoria. Por ejemplo, dos bloques de memoria RAM de 256 x 8 bits se pueden combinar para crear un solo bloque de 256 x 16; del mismo modo, dos bloques de 512 x 4 permiten crear un bloque de 512 x 8 y así sucesivamente.

Los EABs proveen opciones flexibles para manejar y controlar las señales del reloj. Pueden usarse diferentes relojes para las entradas y salidas del EAB. Pueden insertarse registros independientemente en la entrada de datos, salida del EAB, o las entradas de dirección y WE (Write Enable de la RAM). Las señales globales y las interconexiones locales (Local Interconnect) del EAB pueden manejar la señal WE. Las señales globales, pines dedicados a reloj, e interconexiones locales del EAB puede manejar las señales de reloj del EAB. Debido a que los LEs manejan las interconexiones locales del EAB, los LEs pueden controlar la señal WE o las señales de reloj del EAB.

Cada EAB se alimenta por una interconexión fila (row interconnect) y puede desembocar en interconexiones fila y columna. Cada salida del EAB puede conducir a dos canales de fila y a dos canales de columna; el canal de fila sin usar puede ser manejado por otros LEs. Este rasgo aumenta los recursos de la asignación de ruta disponible para las salidas de EAB.

3.7.3. Bloques de arreglos lógicos (LAB, Logic Array Block)

Cada bloque lógico LAB está formado por ocho LEs, las cadenas de acarreo y conexión en cascada asociadas a estos LEs, señales de control, e interconexiones locales. Cada LAB tiene cuatro señales de control de inversión programable que se pueden utilizar en los ocho elementos lógicos (LEs). Dos de estas señales se pueden utilizar como señales de reloj mientras las otras dos se pueden utilizar como señales de preset y clear de los registros.

Las señales de reloj del LAB se pueden controlar por medio de los pines de entrada de reloj, señales globales, señales de E/S (I/O) o señales generadas internamente y conectadas por medio de la interconexión local de los LABs. Las señales de control de preset y clear del LAB pueden ser manejadas por las señales globales, señales I/O, o señales interiores vía interconexión local del LAB. Las señales de control globales se usan típicamente para señales del reloj global, clear, o preset porque ellas proporcionan control sincrónico con muy baja distorsión a través del dispositivo.

Si se requiere lógica en una señal de control, puede ser generada en uno o más LEs en cualquier LAB y manejada en la interconexión local de LAB designado. Además, las señales de control globales pueden generarse de las salidas de LE. En la figura 3.10 se muestra el diagrama interno de un LAB.

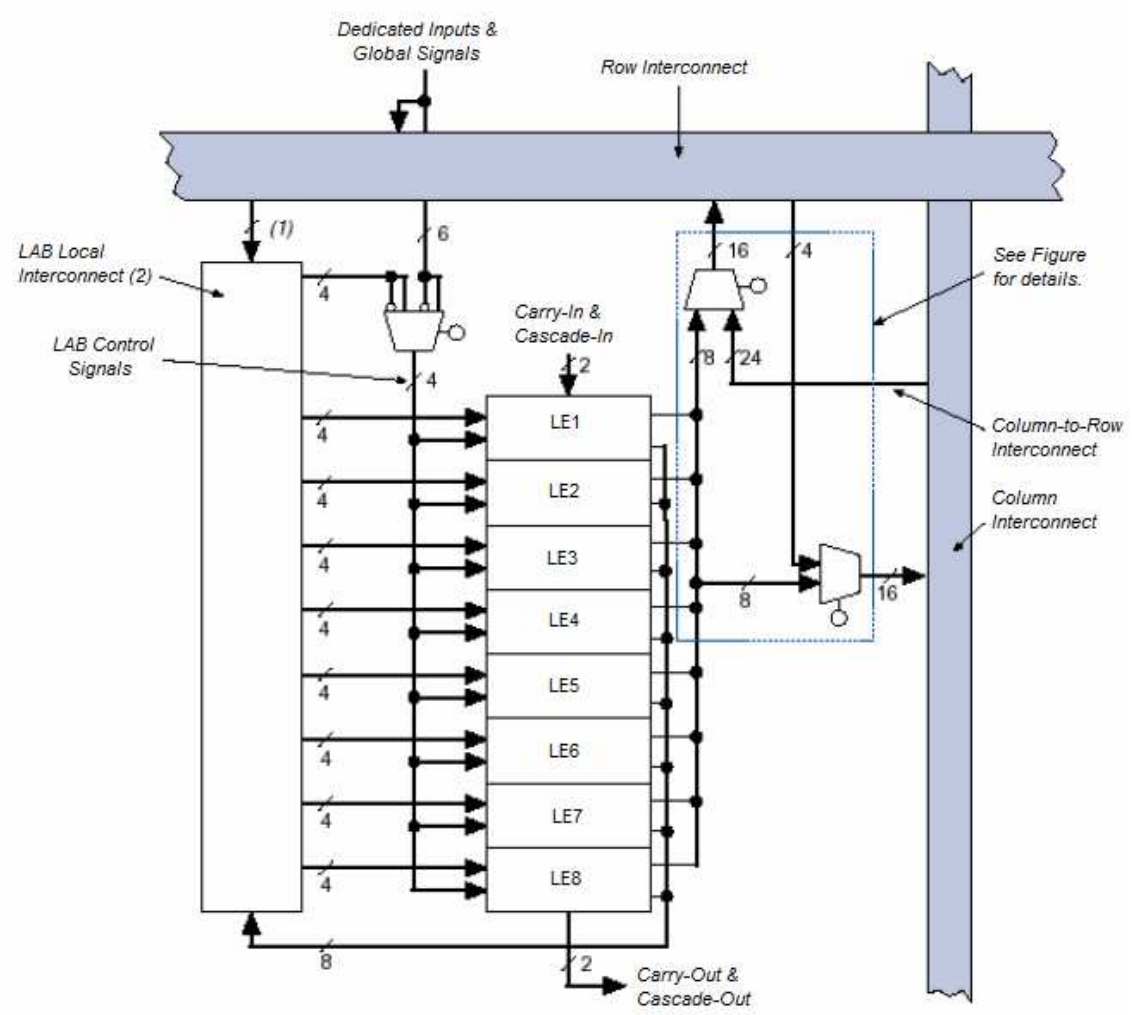


Figura 3. 10: Bloques de arreglos lógicos.
Fuente: http://www.annapmicro.com

3.7.3.1. Elemento Lógico

Un LE es la entidad lógica básica de un dispositivo de la familia FLEX10K. Cada LE se compone de una LUT de cuatro entradas, la cual es un generador de funciones que permite computar cualquier función de cuatro variables rápidamente. A su vez, cada LE incluye un flip-flop programable con una entrada de habilitación sincrónica *enable*, lógica para la rápida propagación de acarreo entre LEs en un mismo LAB y lógica para la construcción de cadenas de conexión en cascada. La salida de cada LE se puede conectar local o

globalmente en el dispositivo con otros elementos lógicos o de memoria. La figura 3.11 muestra el diagrama interno de un LE.,

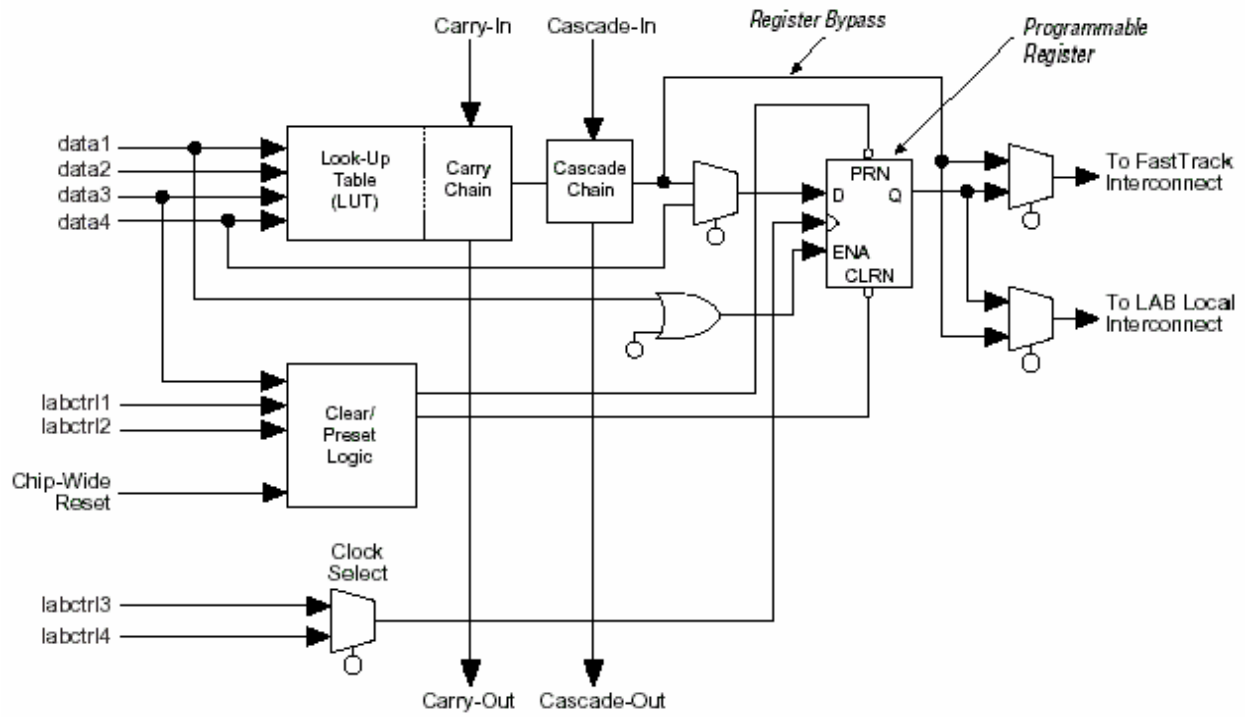


Figura 3. 11: Bloques de arreglos lógicos.
Fuente: http://www.annapmicro.com

El flip-flop del LE se puede configurar para que opere como un biestable D, T, JK o SR. Las señales de control *clock*, clear y preset en el *flip-flop* pueden manejarse por las señales globales, pines I/O de uso general, o cualquier lógica interna. Para las funciones combinatorias, el flip-flop es "*bypaseado*" (esquivado) y la salida del LUT maneja la salida del LE.

Cada LE tiene dos conexiones de salida, una con la interconexión local y otra con la interconexión global que se encuentran organizadas por filas y columnas (*FastTrack Interconect*). Las dos salidas se pueden controlar de forma independiente. Por ejemplo, la LUT puede ser una salida mientras que el registro puede ser otra. Esta posibilidad, llamada embalaje del registro (*register packing*), puede mejorar la utilización de LE porque pueden usarse el registro y la LUT para las funciones no relacionadas.

3.7.3.1.1. Cadenas de acarreo

Las cadenas de acarreo realizan la función de la rápida propagación de acarreo (2ns) entre LEs. La cadena de acarreo genera el acarreo de entrada del bit de orden superior a partir del acarreo de salida del bit que le precede y las entradas a la tabla. Estas cadenas de acarreo permiten a los dispositivos FLEX10K realizar eficientemente sumadores, contadores y comparadores de gran velocidad y de longitud arbitraria. Las cadenas de acarreo de más de ocho LEs se implementan de forma automática uniendo LABs. Para una mejor distribución de la cadena de acarreo en el dispositivo, se evita la utilización de dos LABs consecutivos en una misma fila. Al mismo tiempo, no se generan cadenas de acarreo conectando dos LABs separados por el EAB situado en el centro de la fila.

3.7.3.1.2. Cadenas de conexión en cascada

Las cadenas de conexión en cascada se utilizan en los dispositivos FLEX10K para realizar funciones lógicas de un amplio número de variables. De esta manera, se utilizan tablas adyacentes para calcular porciones de la función en paralelo y la cadena de conexión en cascada conecta en serie las salidas intermedias. La cadena de conexión en cascada puede utilizar una puerta AND o una puerta OR para conectar las salidas de dos LEs adyacentes.

Cada LE puede operar sobre cuatro entradas de la función que se calcula y el retardo de la conexión en cascada es de tan solo 0.7ns por LE. Se pueden implementar cadenas de conexión en cascada de más de ocho LEs uniendo varios LABs. Para una mejor distribución en el dispositivo, se evita la utilización de dos LABs consecutivos en una misma fila. Al mismo tiempo no se generan cadenas de conexión en cascada conectando dos LABs separados por el EAB situado en el centro de la fila.

3.7.3.1.3. Modos de operación

Un elemento lógico puede configurarse de acuerdo con alguno de los cuatro modos de operación siguientes:

- ✓ modo normal *(normal mode)*.
- ✓ modo aritmético *(arithmetic mode).*
- ✓ modo contador ascendente/descendente *(up/down counter mode)*.

✓ modo contador con puesta a cero *(clearable counter mode).*

Cada uno de estos modos usa de forma diferente los recursos que proporciona cada LE. En cada modo, las siete señales de entrada (las cuatro señales de entrada, la realimentación del registro y las de acarreo y conexión en cascada) se dirigen a distintos destinos para implementar la función lógica deseada. Las entradas de reloj, preset y clear de cada LE se utilizan para el control del registro.

✓ **Modo normal:**

Este modo está especialmente orientado para la generación de funciones lógicas y por tanto, se puede aprovechar la cadena de conexión en cascada. En el modo normal, las cuatro entradas de datos y la entrada de acarreo son entradas para la LUT. El compilador se encarga de seleccionar una de las señales carry-in o data3 como una de las entradas de la tabla. La salida de la LUT se puede combinar con la señal cascade-in para generar una cadena de conexión en cascada a través de la señal cascade-out. Tanto la salida de la tabla como la del registro se pueden conectar con la interconexión local o global al mismo tiempo.

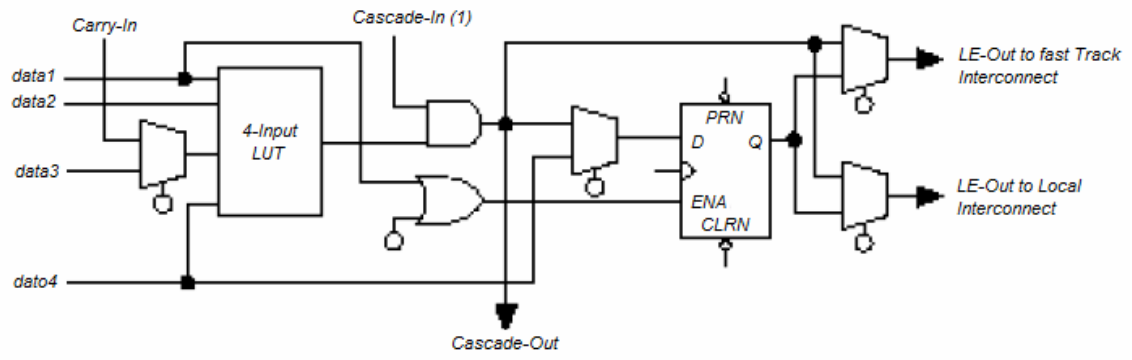


Figura 3. 12: Bloques de arreglos lógicos modo normal.
Fuente: http://www.annapmicro.com

La LUT y el registro se pueden utilizar de forma independiente. Para ello él LE tiene dos salidas, una conectada localmente y la otra globalmente. Existen diversas formas de utilizar la LUT y el registro de forma separada. La señal *data4* puede ser utilizada como entrada del registro y así permite que la tabla calcule una función que es independiente del valor registrado. La LUT

puede también calcular una función de tres entradas al mismo tiempo que cuatro señales independientes se almacenan en el registro. Por último, la LUT puede calcular una función de cuatro entradas y una de estas entradas se puede utilizar para controlar el registro.

✓ **Modo aritmético:**

El modo aritmético ofrece dos LUTs de tres entradas y es ideal para la implementación de sumadores, acumuladores y comparadores. Una LUT calcula una función de tres entradas y la otra genera una salida de acarreo. Tal y como se muestra en la figura 3.13, la primera LUT usa la señal carry-in y dos entradas de datos para generar una salida combinacional o registrada. La segunda LUT utiliza las mismas entradas y genera el acarreo de salida carry-out y, por tanto, crea una cadena de acarreo. El modo aritmético también soporta el uso simultáneo de la cadena de conexión en cascada.

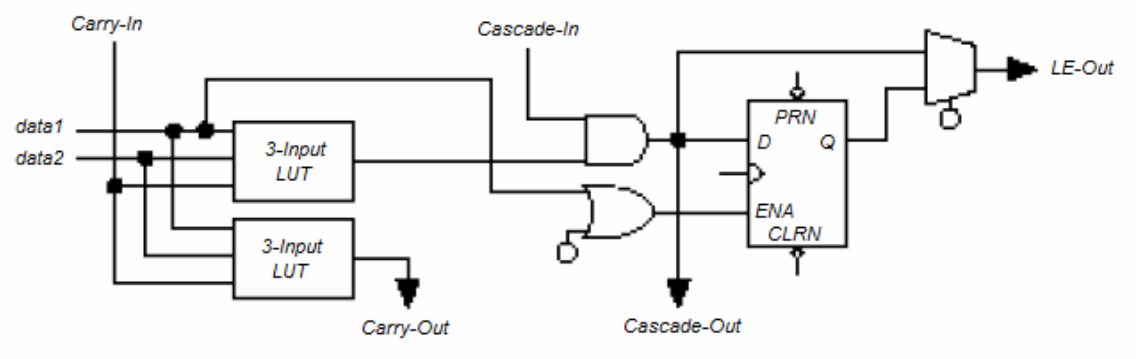


Figura 3. 13: Bloques de arreglos lógicos modo normal.
Fuente: http://www.annapmicro.com

✓ **Modo contador ascendente/descendente:**

El modo de contador ascendente/descendente (ver figura 3.14) ofrece control de habilitación, incremento/decremento sincrónico y carga de datos. Las señales de control se toman de la interconexión local, de la entrada de acarreo carry-in y de la realimentación del registro. Este modo utiliza dos LUTs de tres entradas, una genera el dato del contador y la otra genera un bit de acarreo. Un multiplexor 2:1 permite la carga sincrónica del contador. También se pueden cargar datos de forma asincrónica y sin utilizar los recursos de la LUT utilizado las señales de control clear y preset.

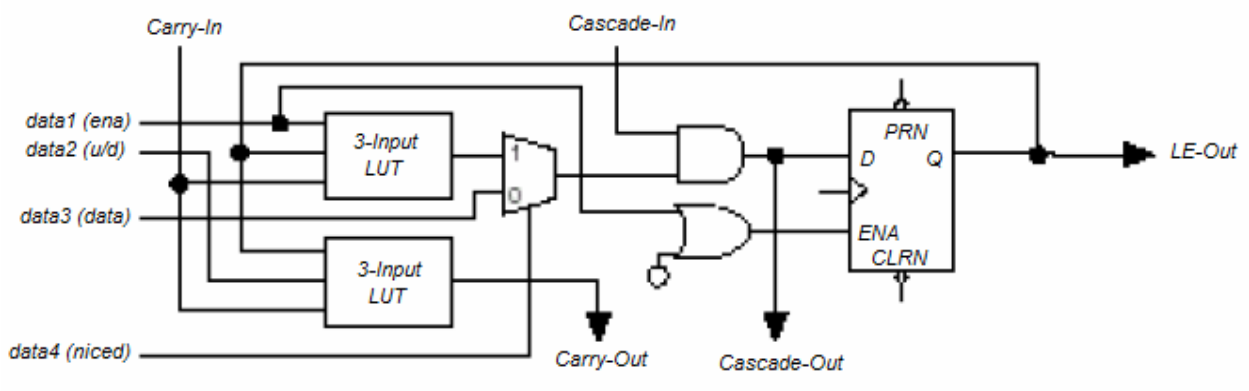


Figura 3. 14: Bloques de arreglos lógicos modo contador ascendente/descendente.
Fuente: http://www.annapmicro.com

✓ **Modo contador con puesta a cero:**

El modo de contador con puesta a cero (ver figura 3.15) es similar al modo de contador ascendente/descendente. La diferencia está en que éste soporta puesta a cero sincrónica en lugar de control del incremento ascendente o descendente. La función de puesta a cero se canaliza a través de la entrada carry-in. En este modo se utilizan dos LUTs de tres entradas, una genera los datos del contador y la otra genera el bit de acarreo. La carga sincrónica del contador se consigue mediante un multiplexor de 2:1.

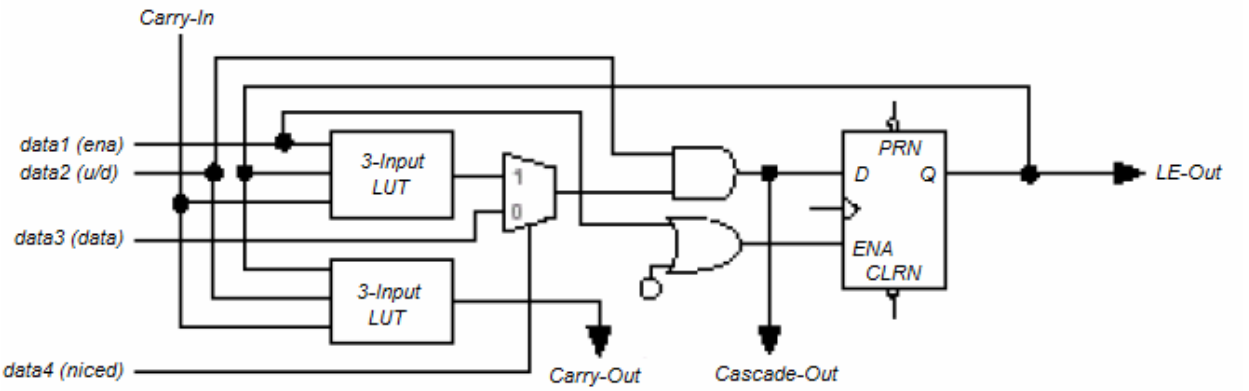


Figura 3. 15: Bloques de arreglos modo contador con puesta a cero.
Fuente: http://www.annapmicro.com

3.7.4. Pista rápida de interconexión.

Las conexiones entre elementos lógicos (*LEs*) y pines de entrada/salida del dispositivo FLEX 10K se realiza por medio del FasrTrack Interconect, el cual es una serie de canales continuos de enrutamiento dispuestos en forma

horizontal y vertical y extendiéndose sobre todo el dispositivo. Esta estructura de enrutamiento global es muy eficiente incluso en diseños complejos.

Con este esquema de enrutamiento cada fila de *LABs* dispone de un canal de interconexión horizontal. Este canal permite la conexión tanto con otros *LABs* del dispositivo como con pines de entrada/salida. La figura 3.16 detalla la organización de estas conexiones y la forma en que se comunican con los *LEs* del *LAB*. A cada fila se puede conectar un *LE* u otro de entre tres canales columna. Una de estas cuatro señales se conecta a través de un multiplexor de 4:1 con dos canales fila específicos.

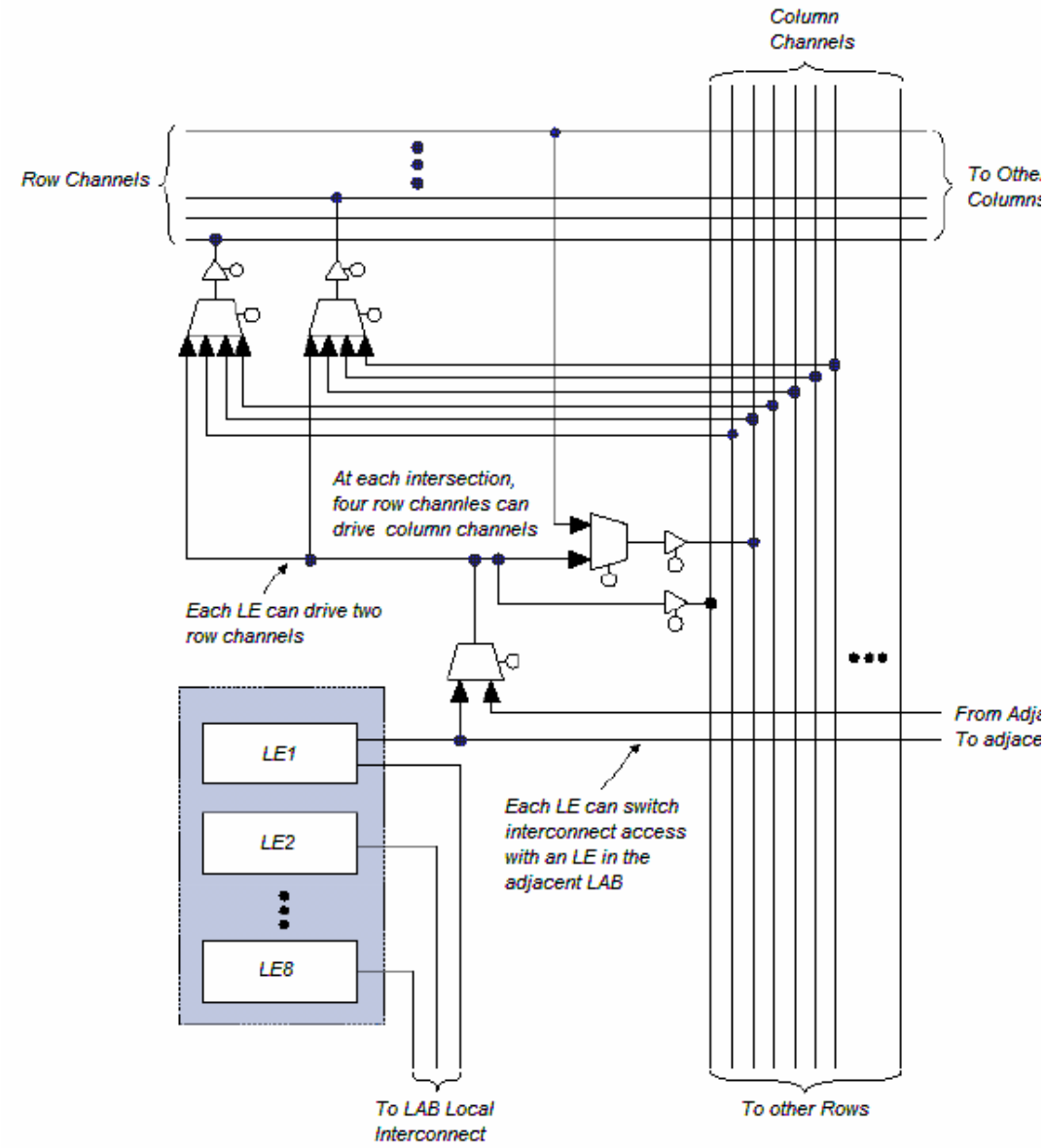


Figura 3. 16: Bloques de arreglos modo contador con puesta a cero.
Fuente: http://www.annapmicro.com

Estos multiplexores permiten conectar los canales columna con canales filas incluso cuando los ocho *LEs* de un *LAB* se encuentran conectados con la fila. Del mismo modo, cada columna de *LABs* se conecta con una interconexión columna. La interconexión columna puede manejar entonces pines de I/O u otra interconexión de fila para dirigir las señales a otros *LABs* en el dispositivo. Una señal de la interconexión columna, que tanto puede ser la salida de un *LE* o una entrada de un pin de *I/O*, debe ser ruteada a la interconexión fila antes de que pueda entrar un *LAB* o *EAB*. Cada canal de fila que es manejado por un *IOE* o *EAB* puede manejar un canal de columna específico.

3.8. APEX 20K

La familia de dispositivos APEX 20K de Altera representa una evolución de la arquitectura de los dispositivos FLEX 10K. Esta nueva familia incorpora nuevas características, mayor densidad y mejores prestaciones en velocidad. Los dispositivos APEX 20K, con hasta 1.500.000 puertas, se fabrican en tecnologías de 0.22µm, 0.18µm y 0.15µm. Los dispositivos APEX 20K incorporan lógica basada en LUT, lógica basada en término producto y memoria, en un único dispositivo. Las interconexiones de señales dentro del dispositivo APEX 20K son proporcionadas por el FastTrack Interconnect.

La matriz de elementos lógicos agrupa los elementos lógicos en LABs de 10 LEs. Los bloques de memoria, ahora llamados ESBs (*Embedded System Blocks*), incorporan nuevas funcionalidades como por ejemplo la capacidad de definir memorias direccionable por el contenido (CAM, *Content Addressable Memory*). Cada ESB proporciona 2048 bits de memoria y se puede configurar como un elemento de memoria de 128 x 16, 256 x 8, 512 x 4, 1024 x 2 o 2048 x 1. Del mismo modo que para un dispositivo de la familia FLEX 10K, se pueden definir bloques de memoria mayores agrupando dos o más ESBs. El tamaño de estos ESBs es programable. La figura 3.17 muestra un diagrama en bloques de la Arquitectura de los dispositivos APEX 20K.

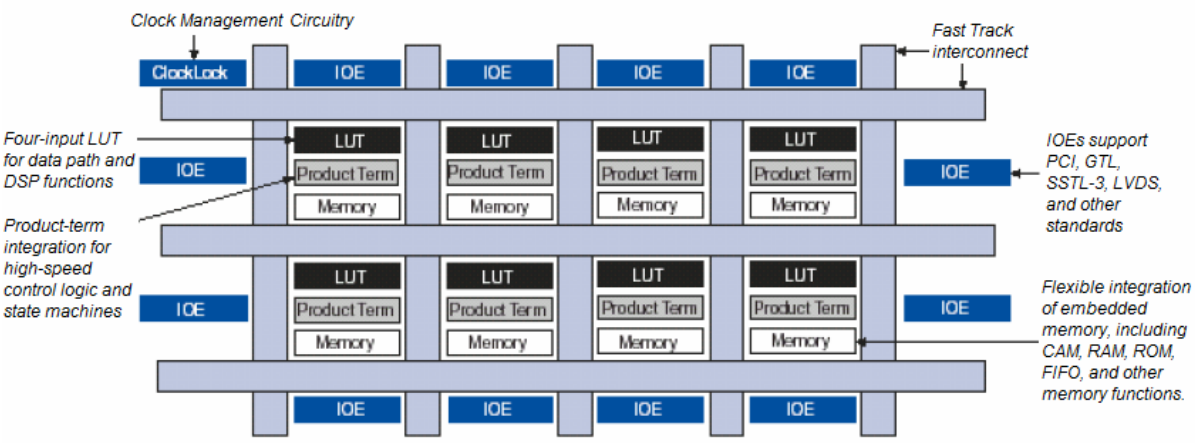


Figura 3. 17: Arquitectura del APEX 20K.
Fuente: http://www.annapmicro.com

La tabla 3.3 muestra algunos miembros de la familia APEX 20K y sus principales características.

Dispositivo	LEs	ESBs	Bits de memoria	Entradas Salidas
EP20K60E	2260	16	32768	504
EP20K100(E)	4160	26	53248	252
EP20K160E	6400	40	81920	316
EP20K200(E)	8320	52	106496	382
EP20K300E	11520	72	147456	408
EP20K400(E)	16640	104	212992	502
EP20K600E	24320	152	311296	624
EP20K1000E	38400	160	327680	708
EP20K1500E	51840	216	442368	808

Tabla 3.3: Dispositivos de APEX 20K.
Fuente**:** Altera

3.9. LA FAMILIA ACEX 1K

La arquitectura de los dispositivos ACEX 1K es similar a la de los dispositivos FLEX 10K. Están integrados por bloques de memoria llamados EABs y una matriz de elementos lógicos constituida por bloques lógicos (LABs). Cada LAB agrupa 8 elementos lógicos (LEs) e interconexiones locales.

Un LE incluye una tabla de look-up de 4 entradas, un flip-flop programable y lógica para la rápida generación y propagación de acarreo y la conexión en cascada. Las conexiones entre elementos de memoria y elementos lógicos se realizan mediante el llamado FastTrack Interconnect.

Cada pin de Entrada/Salida se alimenta por un elemento de Entrada/Salida (IOE) localizado al final de cada fila y columna del FastTrack Interconnect. Cada IOE contiene un buffer de Entrada/Salida bidireccional y un flip-flop que pueden usarse tanto como registro de la salida o entrada para alimentar la entrada, la salida, o las señales bidireccionales. La tabla 1.8 muestra los recursos que integran algunos de los miembros de la familia de dispositivos ACEX 1K.

Dispositivo	LEs	EABs	Bits de memoria	Entradas Salidas
EP1K10	576	3	12288	136
EP1K30	1728	6	24576	171
EP1K50	2880	10	40960	249
EP1K100	4992	12	49152	333

Tabla 3.4: Dispositivos de ACEX 1K.
Fuente**:** Altera

La principal diferencia con los dispositivos FLEX 10K, es que en los dispositivos ACEX 1K el EAB también puede usarse para aplicaciones de memoria bidireccional de doble puerto, donde dos puertos leen o escriben simultáneamente o sea se permite el acceso simultáneo de dos procesadores al mismo bloque de memoria. Para llevar a cabo este tipo de memoria, se usan dos EABs para apoyar dos lecturas o escrituras simultáneas.

3.10. Cyclone

Las FPGAs Cyclone utilizan tecnologías de 0.13µm con densidades de hasta 20,060 elementos lógicos (LEs, Logic Elements) y hasta 288 Kbits de RAM. Los dispositivos Cyclone contienen una arquitectura basada en fila y

columna de dos dimensiones para implementar lógica. Las interconexiones columna y fila de distintas velocidades proporcionan señales de interconexión entre los LABs y los bloques integrados de memoria.

El arreglo lógico esta constituido por LABs, con 10 LEs en cada LAB. Un LE es una unidad pequeña de lógica que proporciona una aplicación eficaz de funciones lógicas. Los LABs se agrupan en filas y columnas a través del dispositivo. Los dispositivos Cyclone tienen entre 2910 a 20060 LEs. Los bloques M4K RAM son los verdaderos bloques de memoria de doble puerto con 4K bits de memoria más la paridad (4608 bits).

Estos bloques proporcionan memorias de doble puerto especializado, doble puerto simple, o de un puerto de hasta 36 bits de ancho llegando a los 200 MHz. Estos bloques se agrupan en columnas a través del dispositivo entre ciertos LABs. Los dispositivos Cyclone ofrecen entre 60 y 288 Kbits de RAM integrada. Cada pin E/S del dispositivo Cyclone es alimentado por un elemento de E/S (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.

Los dispositivos Cyclone proporcionan una red de reloj global y hasta dos PLLs. La red de reloj global consiste en ocho líneas de relojes globales que recorren el dispositivo entero. La red de reloj global puede proveer a relojes a todos los recursos dentro del dispositivo, como IOEs, LEs, y bloques de memoria. Las líneas de relojes globales también pueden usarse para señales del control. La figura 3.18 muestra un diagrama arquitectura de los dispositivos EP1C12 de la familia Cyclone.

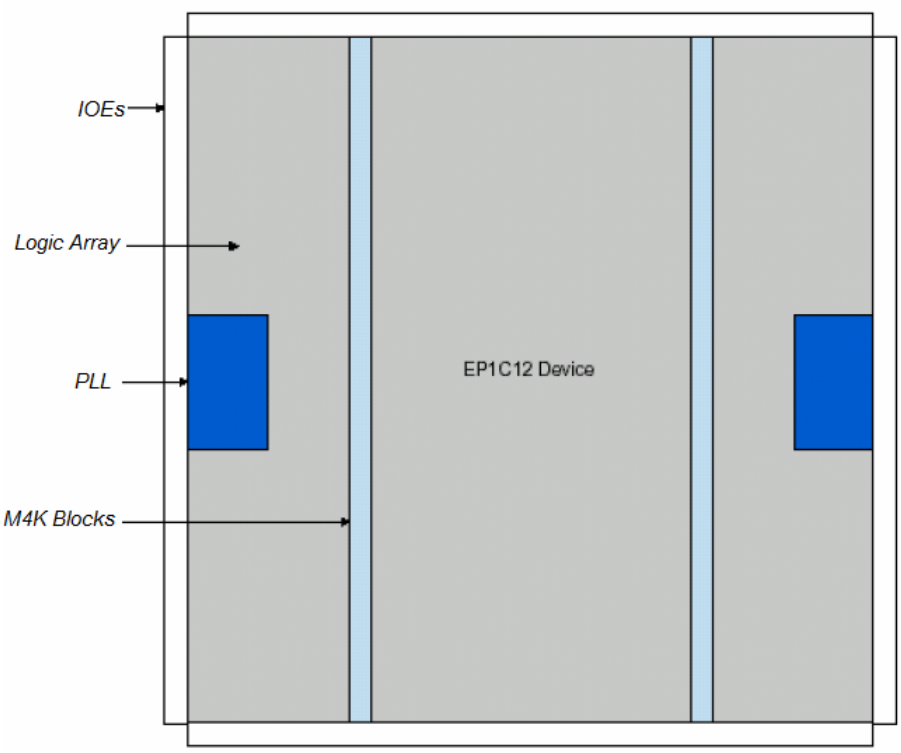


Figura 3. 18: Arquitectura del Cyclone II.
Fuente: http://www.annapmicro.com

En la tabla 3.5 se muestra los recursos que integran algunos de los miembros de la familia de dispositivos Cyclone.

Dispositivo	LEs	Bloques M4K RAM(128x36)bits	Bits de memoria	PLLs	Entradas Salidas
EP1C3	2910	13	59904	1	104
EP1C4	4000	17	78336	2	301
EP1C6	5380	20	92160	2	185
EP1C12	12060	52	239616	2	249
EP1C20	20060	64	294912	2	301

Tabla 3.5: Dispositivos de Cyclone II.
Fuente**:** Altera

3.11. QUARTUS II

QUARTUS, es una herramienta avanzada de diseño ayudado por computador (CAD) que la compañía ALTERA ha creado para diseñar sistemas

digitales e implementarlos en dispositivos de lógica programable. Durante el curso la utilizaremos en todas las etapas involucradas en la realización de un proyecto de diseño digital: edición, simulación, programación y verificación. En este acápite se explica de manera breve el funcionamiento básico de esta herramienta y las opciones que existen a la hora de definir y simular el sistema digital.

3.11.1. Procedimiento de diseño.

Usando la herramienta QUARTUS, estas son las etapas que deben seguirse para diseñar un sistema lógico:

a) Definición del sistema lógico, que en función de su complejidad se subdividirá en subsistemas formando una estructura jerárquica. Cada uno de estos subsistemas se definirá de alguna de las siguientes maneras:
   - Editando gráficamente su esquema, esto es, dibujando su esquema a partir de bibliotecas de elementos básicos y símbolos que representan otros subdiseños.
   - Editando en forma de texto o escribiendo su descripción en un lenguaje de descripción de hardware como VHDL.

b) Compilación: Después de definir el circuito, éste debe compilarse. Se detectan errores introducidos en la definición. Tras corregirlos se crea un fichero que contiene la lógica del sistema y que permitirá simular su funcionamiento. Este es el momento de elegir una PLD concreta en la que programar el sistema completo. El compilador necesitará esta información para realizar simplificaciones y minimizar la lógica del sistema adecuándola de la mejor manera a la PLD.

c) Definición de entradas Todo sistema lógico procesa señales externas y genera resultados. Por ello, antes de simular el funcionamiento del mismo, hay que definir sus entradas, es decir los valores aplicados a todas las entradas del sistema durante el tiempo que dure la simulación. Para definir las entradas se utiliza un editor de formas de onda

d) Simulación lógica: A continuación, ya puede realizarse la simulación lógica del sistema, verificando su comportamiento. Se pueden realizar dos tipos de simulación: simulación funcional o lógica y simulación

temporal o física. En la primera no se tienen en cuenta los tiempos de respuesta del circuito ni los retardos. En la segunda, en cambio, sí.

e) Implementación en PLD: Una vez comprobado que el sistema funciona como se desea, puede programarse en un dispositivo PLD, si ese es el objetivo. Después, en el laboratorio, se realizarán las fases de montaje y prueba.

El programa funciona mediante menús. A continuación se va a describir de manera muy breve las pautas a seguir para diseñar con esta herramienta.

3.11.2. Pasos en el diseño de sistemas digitales.

Para proceder al diseño de un sistema digital, seguiremos los siguientes pasos:

i. Directorio de diseño

Siempre que vayamos a definir un módulo, crearemos en el directorio de trabajo (por ejemplo: Digitales II VHDL) una carpeta con el nombre del módulo, que deberá ser el mismo que el proyecto de Quartus que creemos, por ejemplo ***practica1***. A la hora de definir mediante QUARTUS uno de nuestros diseños, mantendremos la estructura jerárquica utilizada al realizarlo, pero en la definición (gráfica o textual) seguiremos la filosofía *Bottomup <<esto es, se comienza diseñando los submódulos de nivel más bajo hasta completar el diseño total>>.*

En el punto más alto de la jerarquía, tendremos un módulo que representará el diseño completo y que incluirá símbolos de los submódulos que lo componen. Supongamos que el directorio en el que vamos a guardar nuestros diseños se llama Digitales II VHDL y que nuestro diseño de llama practica1. Éste, a su vez, consta de los submódulos practica11 y practica12. La estructura de directorios a crear sería el que se muestra en la figura 3.19:

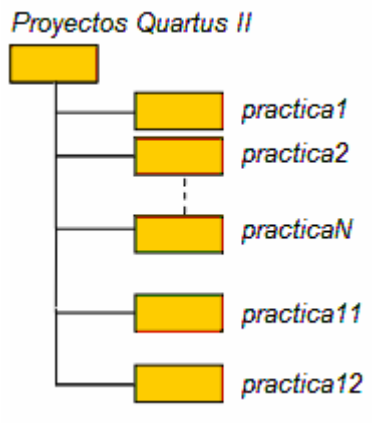


Figura 3. 19: Estructura de directorios Quartus II.
Fuente: Los autores

ii. Abrir Quartus II y crear nuevo proyecto de diseño.

Lo primero de todo hay que arrancar el programa que se va a utilizar. Para ello hay que elegir la opción Quartus II 7.2 Web Edition que está en la ruta: Botón de Inicio → Programas → D.E.A. → Altera → Quartus II 7.2 Web Edition.

Posteriormente se va a declarar **el proyecto** de trabajo en el programa *Quartus II*. Con este proyecto se le va a indicar al programa cuál es el directorio de trabajo y cuál es el archivo principal de la jerarquía del esquema de la Figura 3.20.

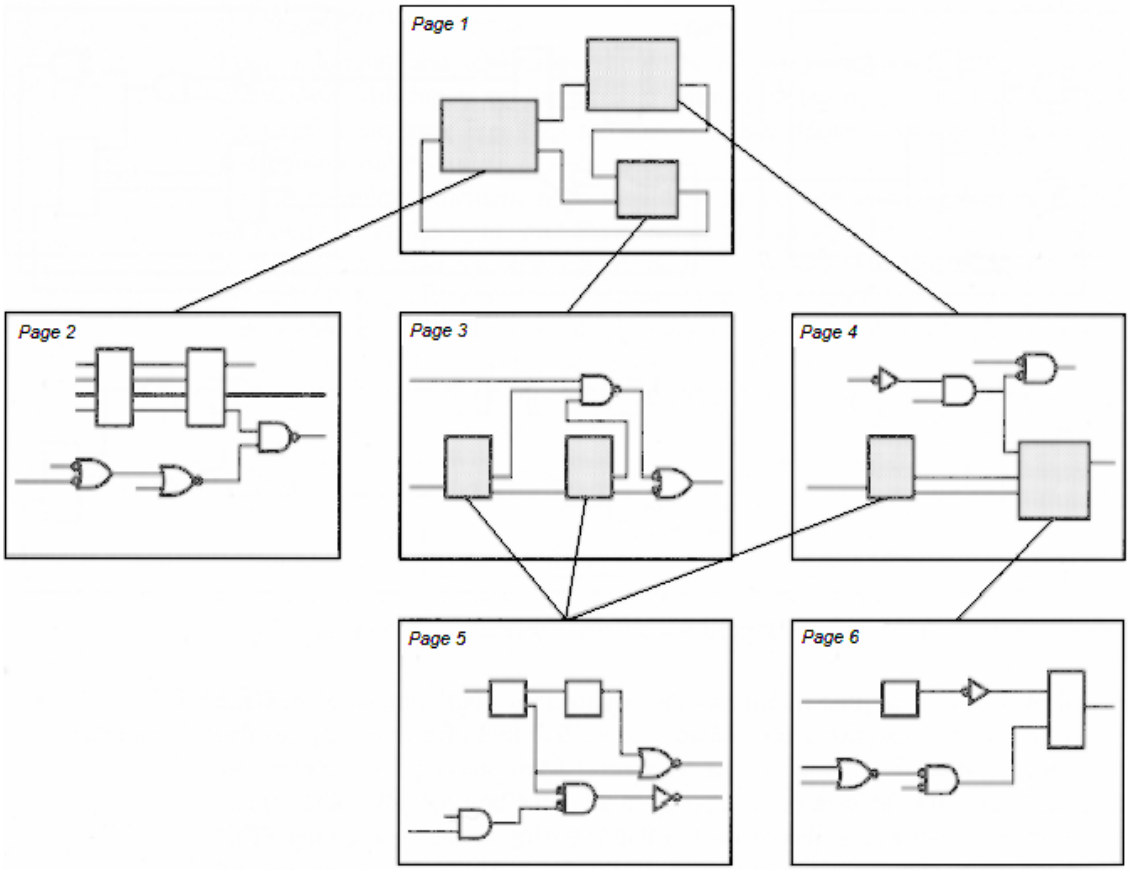


Figura 3. 20: Arquitectura del Cyclone II.
Fuente: Maxinez G. David

Ya arrancado *Quartus II* hay que seleccionar la opción New Project Wizard… en el menú File. Aparece una ventana de inicio, pulsar Next. A continuación, después de pulsar Next, aparecerá la ventana en la que hay que indicar el directorio (C:\Users\pc\Desktop\Escritorio\DigitalesII\practica1) en el que se va a trabajar; el nombre del proyecto (practica1); y el nombre del archivo principal de la jerarquía (practica1), tal y como se muestra en la figura 3.21. Para evitar confusiones se da el mismo nombre en los dos campos. A continuación pulsar Next.

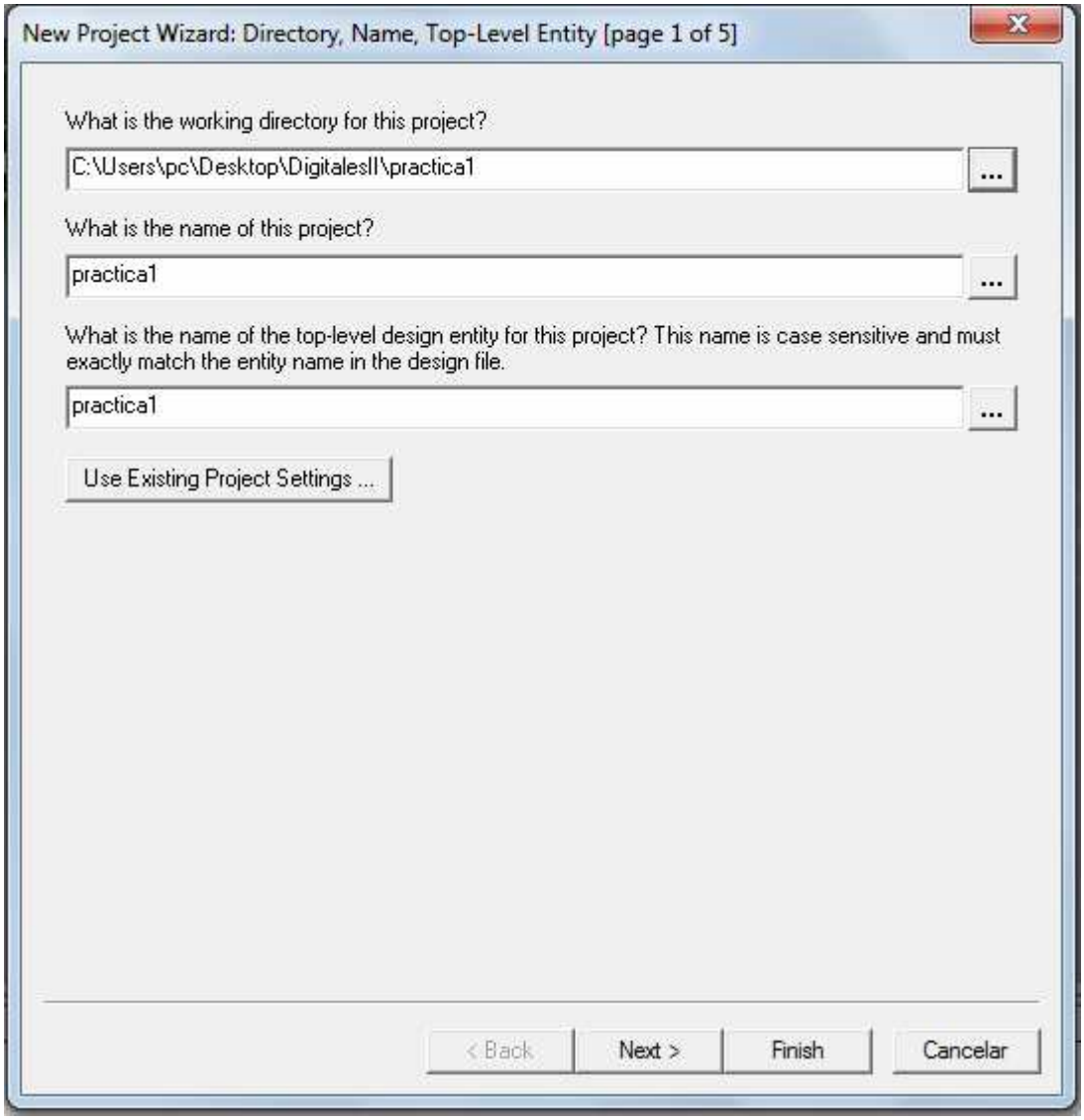


Figura 3. 21: Ventana para la creación de un nuevo proyecto.
Fuente: Los autores

En la siguiente ventana no hay que hacer nada, simplemente pulsar en Next. Después aparecerá otra ventana que se muestra en la figura 3.22 donde hay que indicarle al programa el dispositivo (familia y modelo) con el que se va a trabajar más adelante. Para ello primero hay que seleccionar la familia ***Cyclone II***, y después, en la lista de dispositivos disponibles de la parte de abajo, seleccionar el dispositivo ***EP2C20F484C7***, que es el que hay montado en la tarjeta de desarrollo de lógica programable del laboratorio de electrónica de la Facultad de Educación Técnica para el desarrollo de la UCSG.

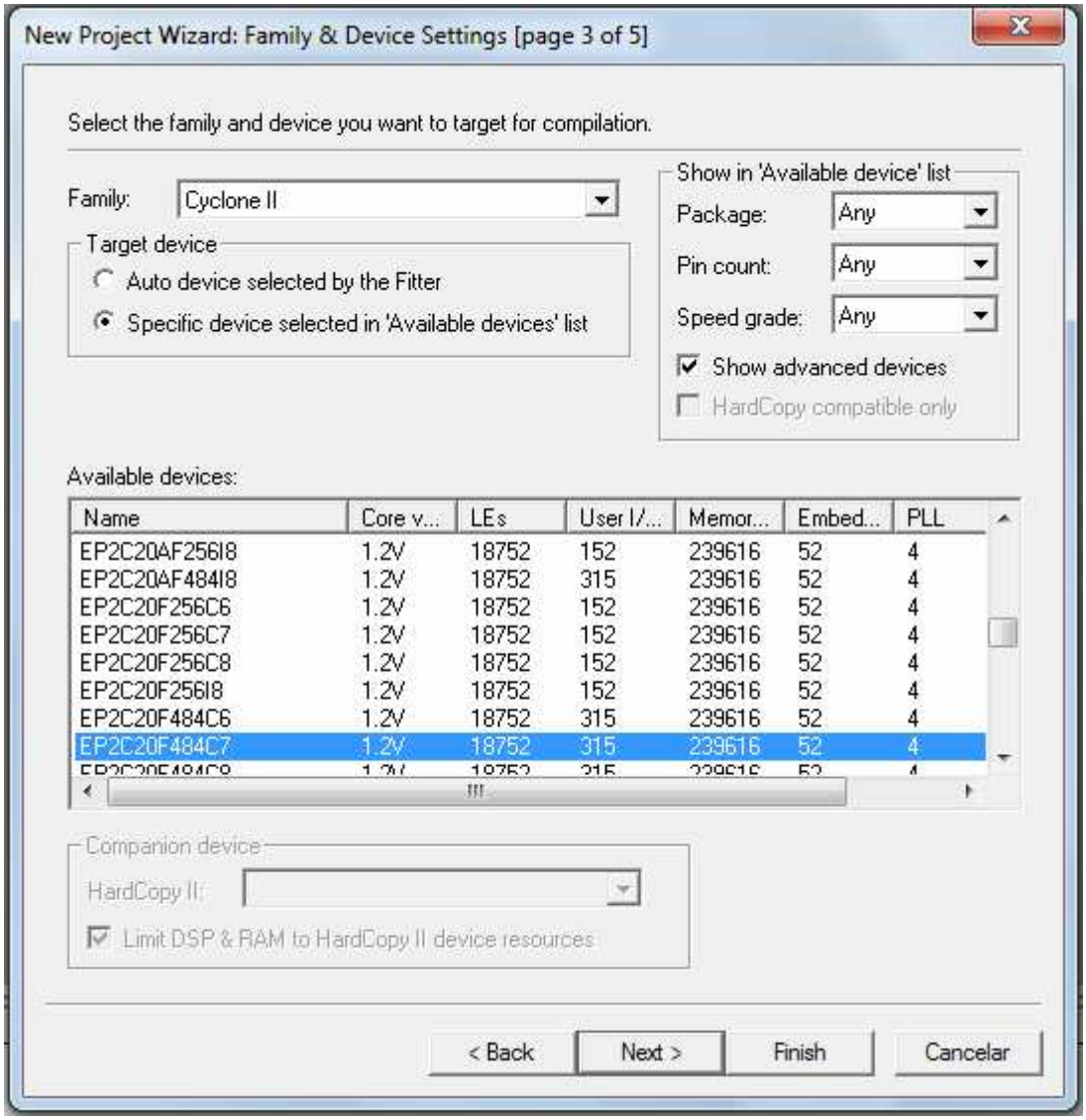


Figura 3. 22: Asignación del dispositivo Cyclone II.
Fuente: Los autores

Con estos pasos ya se tiene todo lo necesario para declarar el proyecto, falta por elegir la herramienta que nos permite simular el circuito digital. Para ello damos al botón de Next. En la siguiente ventana (ver figura 3.22), hay que indicarle al programa la herramienta de Simulación que va a usarse para trabajar más adelante. Para ello primero hay que seleccionar en la fila Simulation la opción de ModelSim-Altera.

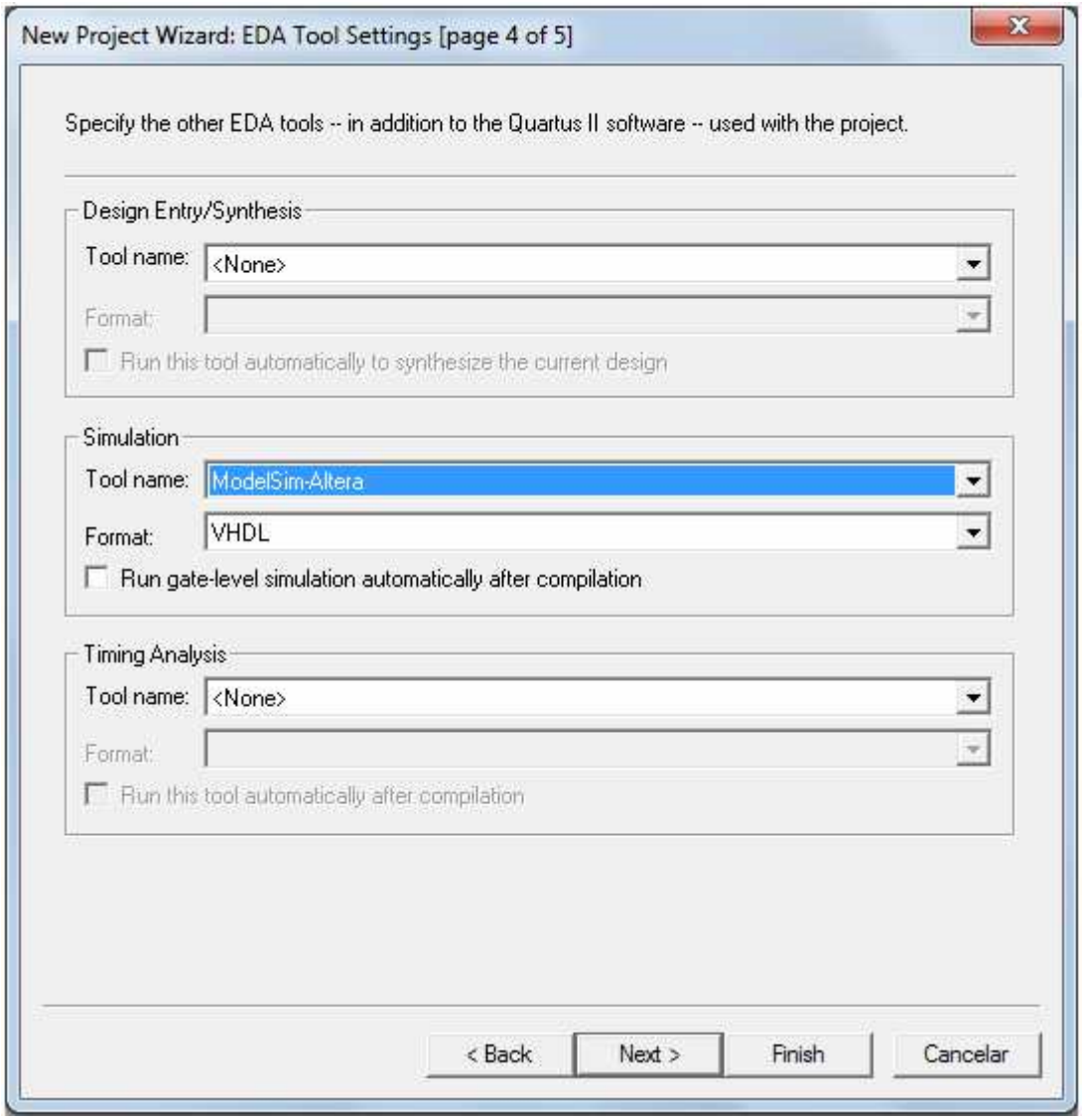


Figura 3. 23: Asignación del tipo de simulación.
Fuente: Los autores

Con estos pasos ya se tiene todo listo para empezar a trabajar y simular, por lo que, para terminar, hay que pulsar Next para poder ver un resumen de los datos del proyecto y comprobar que todo esté bien y luego pulsar el botón Finish.

iii. Abrir un fichero tipo esquemático o texto.

En este fichero se a diseñar nuestro módulo, en este caso tendrá de nombre comparador. Se puede escoger entre un fichero VHDL o un fichero tipo schematic entre otros. Empezaremos por un fichero VHDL. Se ejecuta: File → New

iv. Descripción del circuito.

Al haber elegido un fichero VHDL, aparece una hoja en la que vamos a poder describir nuestro diseño a nivel de programación, ya habíamos elegido el nombre del ejemplo << comparador >>, en la figura 3.24 se muestra la declaración de la entidad comparador.

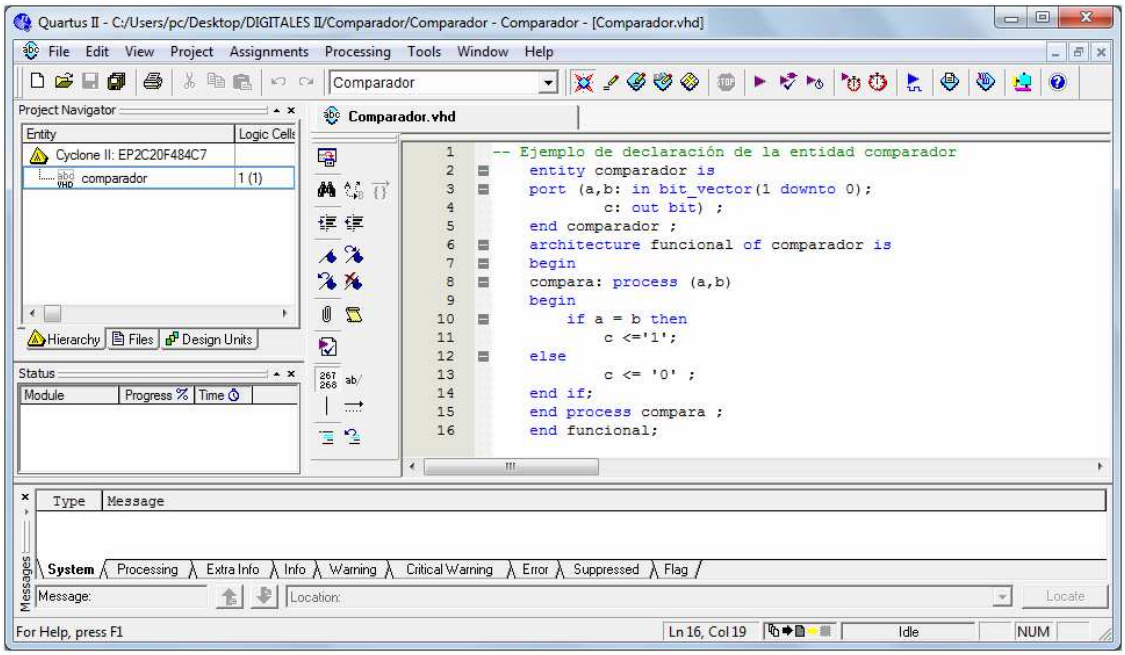


Figura 3. 24: Declaración de la entidad comparador.
Fuente: Los autores

v. Compilar el diseño.

Una vez que se ha programado en VHDL y configurando los pines de entrada/salida, el siguiente paso consiste en compilar el diseño. Para lo cual se ha de ir al menú Processing → Start Compilation (ver figura 3.25). Este proceso supone que el dispositivo donde se va a implementar el diseño es el seleccionado en el punto 1.

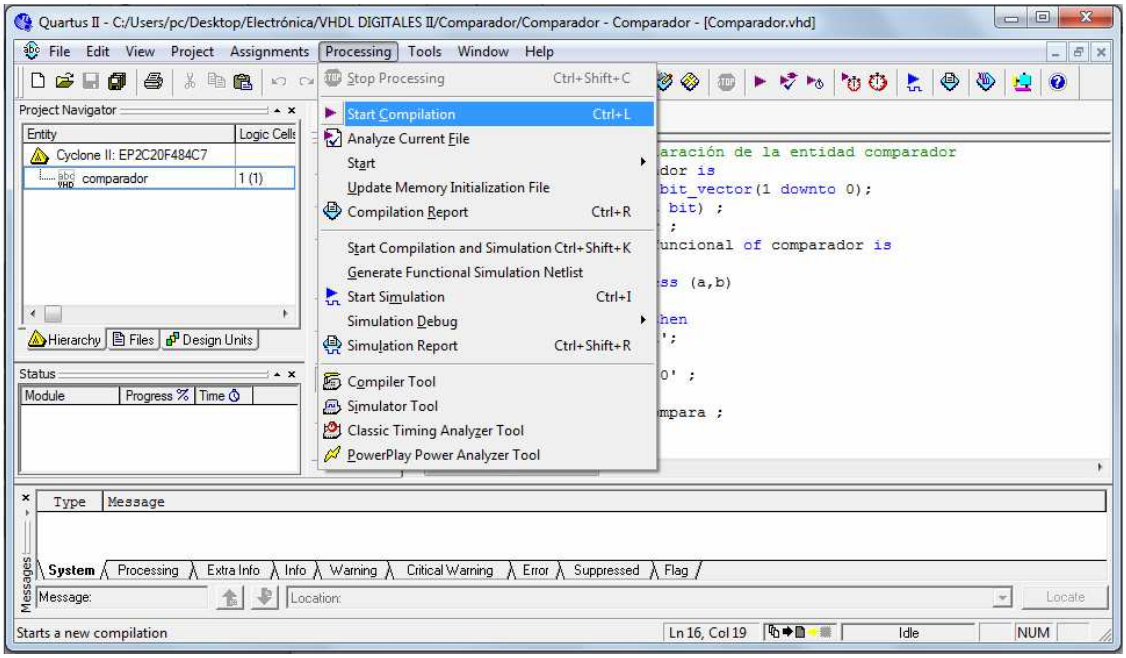


Figura 3. 25: Compilación del programa en VHDL.
Fuente: Los autores

vi. Simulación.

Una vez que la compilación es correcta el siguiente paso es simular el funcionamiento de nuestro diseño. Si funciona correctamente tendremos ya un modulo que podrá servir para configurar una PLD. Los pasos a seguir son:

1. Generar las entradas al sistema (su evolución en el tiempo)

   Para generar las señales de entrada, se usa un editor de ondas:

   File → new (other files) → vector waveform file

   Asignarle el mismo nombre que el módulo y proyecto a simular (con la extensión vwf)
2. Elegir las señales a analizar.
3. Simular y comprobar el resultado. Si no es correcto modificar el diseño y volver a simular.

CAPÍTULO 4: DESARROLLO EXPERIMENTAL DE SISTEMAS DIGITALES

En el presente capítulo de desarrollarán cada una de las prácticas a través del simulador Quartus II, ya en el anterior capítulo se describió una breve introducción del manejo del mismo; para lo cual empezaremos con la programación en VHDL de flip-flops, después con el diseño de sistemas secuenciales. También se incluirá en el programa de estudio de Digitales II para su ejecución en el semestre A-2011.

4.1. PRÁCTICA 1: Diseño Lógico Secuencial en VHDL

Introducción a los biestables y a las máquinas de estados.

OBJETIVOS

El propósito de esta práctica es:

- Comprender el funcionamiento de un biestable.
- Implementar y simular máquinas de estado en VHDL.

MATERIALES

- Computadora o portátil.
- Software Quartus II.

DURACIÓN

2 horas

TRABAJO PREVIO

En esta práctica o sesión se programarán y simularan ejercicios previamente desarrollados:

- Escribir un programa en VHDL que describa el funcionamiento de un flip-flop tipo ‘SR’ de acuerdo a la figura 4.1 y en base a la tabla 4.1 de verdad.
- Revisar las hojas de características de los biestables usados, anotando los tiempos de propagación de cada uno de los biestables.
- Diseñar una máquina de estados.

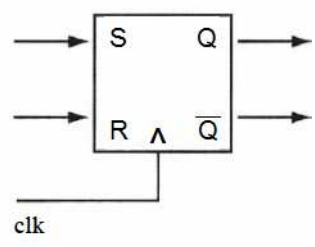


Figura 4. 1: Diagrama de bloque del Flip-Flipo SR
Fuente: Los autores

En la tabla 4.1 de verdad del flip-flop SR muestra que cuando la entrada S es igual a 1 y la entrada R es igual a 0, la salida Q_{t+i} toma valores lógicos de 1. Por otro lado, cuando S = 0 y R = 1, la salida Q_{t+i}= 0; en el caso de que S y R sean ambas igual a 1 lógico, la salida Qt+j queda indeterminada (don´t care); es decir, no es posible precisar su valor y éste puede adoptar el 0 o 1 lógico.

s	R	Q	Qt+i
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

Tabla 4.1: Tabla de verdad del flip-flop SR
Fuente: Los autores

Finalmente, al no existir cambios de las entradas S y R, es decir, su resultado igual a 0; hace que el valor de Qt+i se mantenga en su estado actual Q. De acuerdo al anterior análisis, en la figura 4.2 se muestra el listado del programa realizado en VHDL, utilizando instrucciones condicionales y un nuevo tipo de datos: valores no importa ('-'), los cuales permiten adoptar un valor de ‘0’ o ‘1’ lógico de manera indistinta.

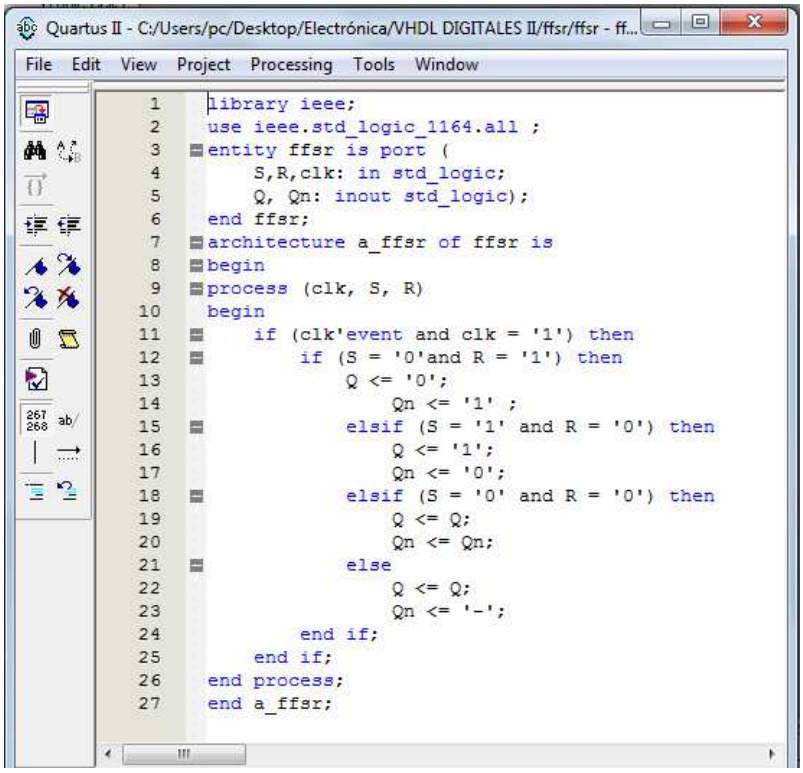


Figura 4. 2: Listado de programación VHDL de la práctica 1.
Fuente: Los autores

En la figura 4.3 se muestra la ejecución y reporte de compilación del programa mostrado en la figura 4.2 de la práctica 1.

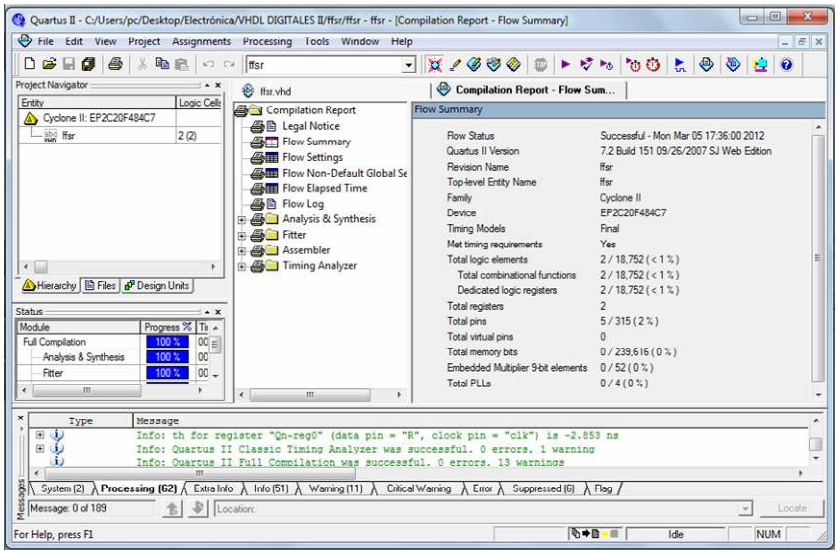


Figura 4. 3: Resultado de la compilación de programación VHDL de la práctica 1.
Fuente: Los autores

Una vez realizada la compilación sin ningún tipo de error de sintaxis en la programación, procederemos a ejecutar el diseño RTL realizado por el software Quartus II, para eso nos dirigimos *Tools* <herramientas> + *Netlist Viewers* y se

escoge *RTL Viewer*, en la figura 4.4 se muestra el resultado obtenido del diseño una vez que se escogió *RTL*.

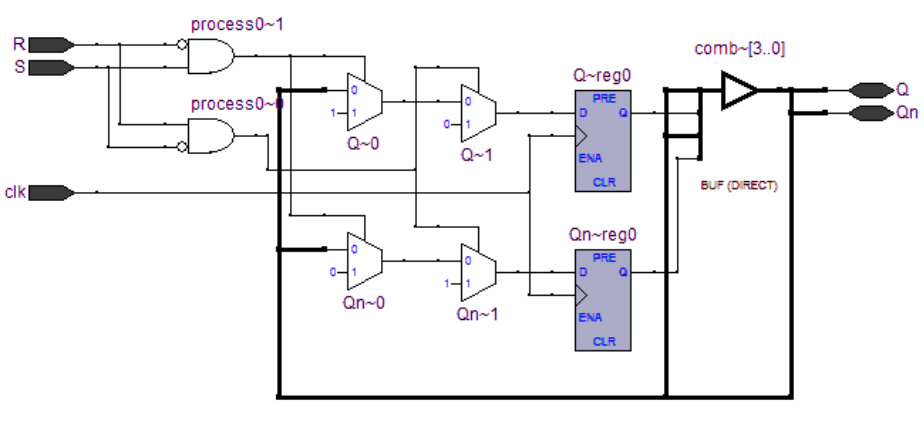


Figura 4. 4: Diseño RTL de la práctica 1.
Fuente: Los autores

4.2. PRÁCTICA 2. Diseño de un registro de 4 bits.

Realizar la programación en VHDL de un registro de 4 bits, como el mostrado en la figura 4.5, en el diseño deben utilizarse instrucciones *if - then - else* y procesos.

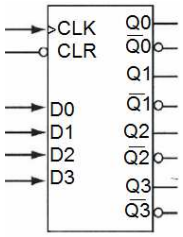


Figura 4. 5: Registro de 4 bits de la práctica 2.
Fuente: Los autores

Como puede observarse en la tabla 4.2, donde se describe el comportamiento del registro de 4 bits, si CLR = 0, las salidas Q son ‘0’; pero si CLR = 1, toman el valor de las entradas DO, D1, D2 y D3.

C

CLR	D	Q	QN
0		0	1
1	Dn	Dn	Dn

Tabla 4.2: Tabla de estados del registro de 4 bits.

Fuente: Los autores

De acuerdo a lo establecido en la tabla 4.2 y de las condiciones dadas, procedemos a realizar el código del programa para un registro de 4 bits en VHDL, el mismo que se muestra en la figura 4.6.

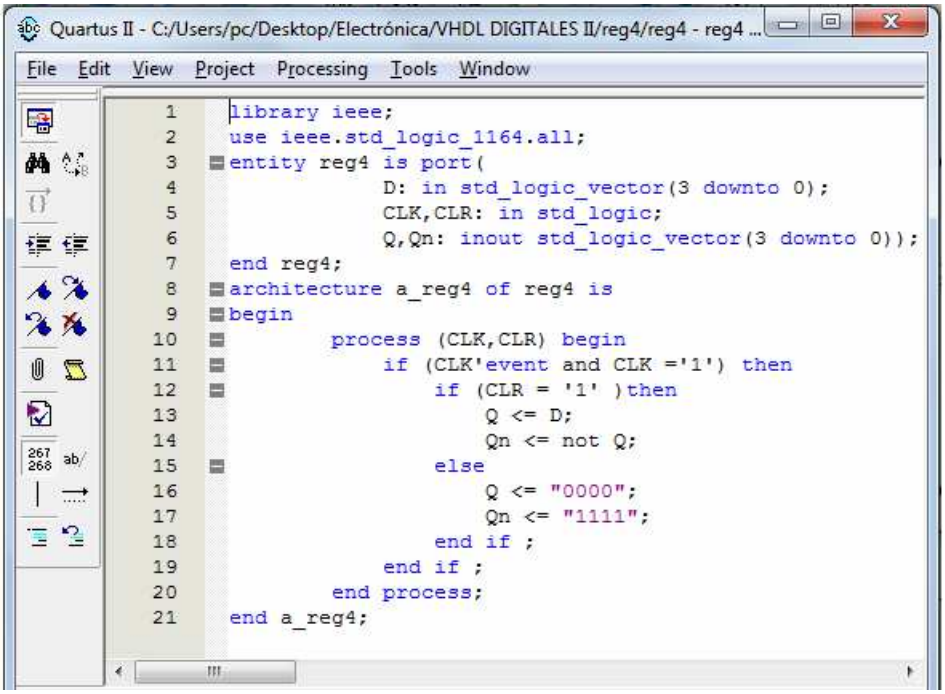


Figura 4. 6: Programa VHDL del registro de 4 bits de la práctica 2.
Fuente: Los autores

Una vez creado el programa en VHDL se realiza la compilación del mismo, en la figura 4.7 se muestra el reporte sin ningún error de sintaxis para la práctica 2.

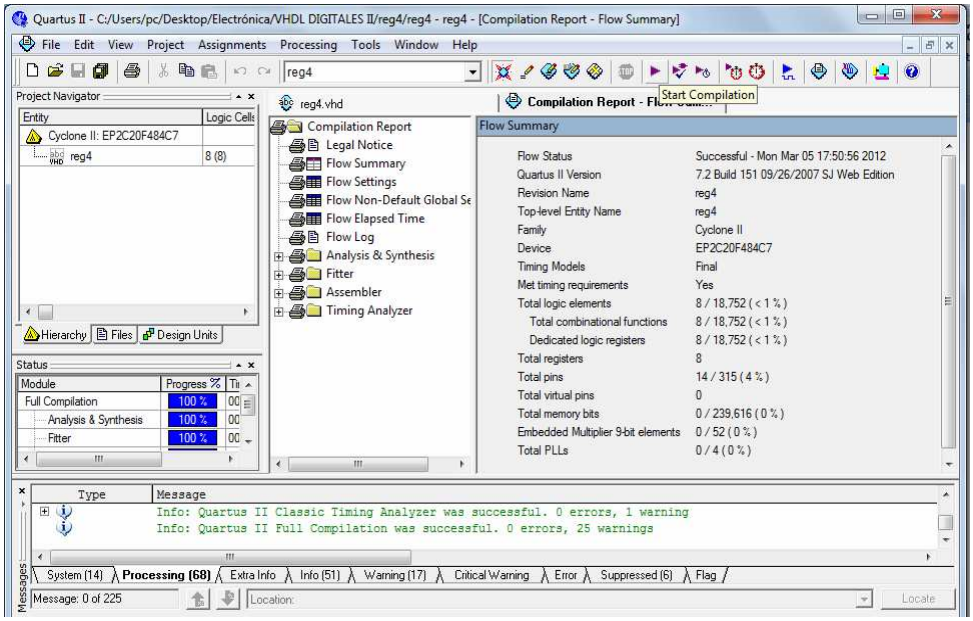


Figura 4. 7: Resultado de la compilación de programación VHDL de la práctica 2.
Fuente: Los autores

En la figura 4.8 se muestra el diseño RTL de acuerdo a la programación del registro de 4 bits en VHDL, el procedimiento ya se lo explicó en la práctica anterior.

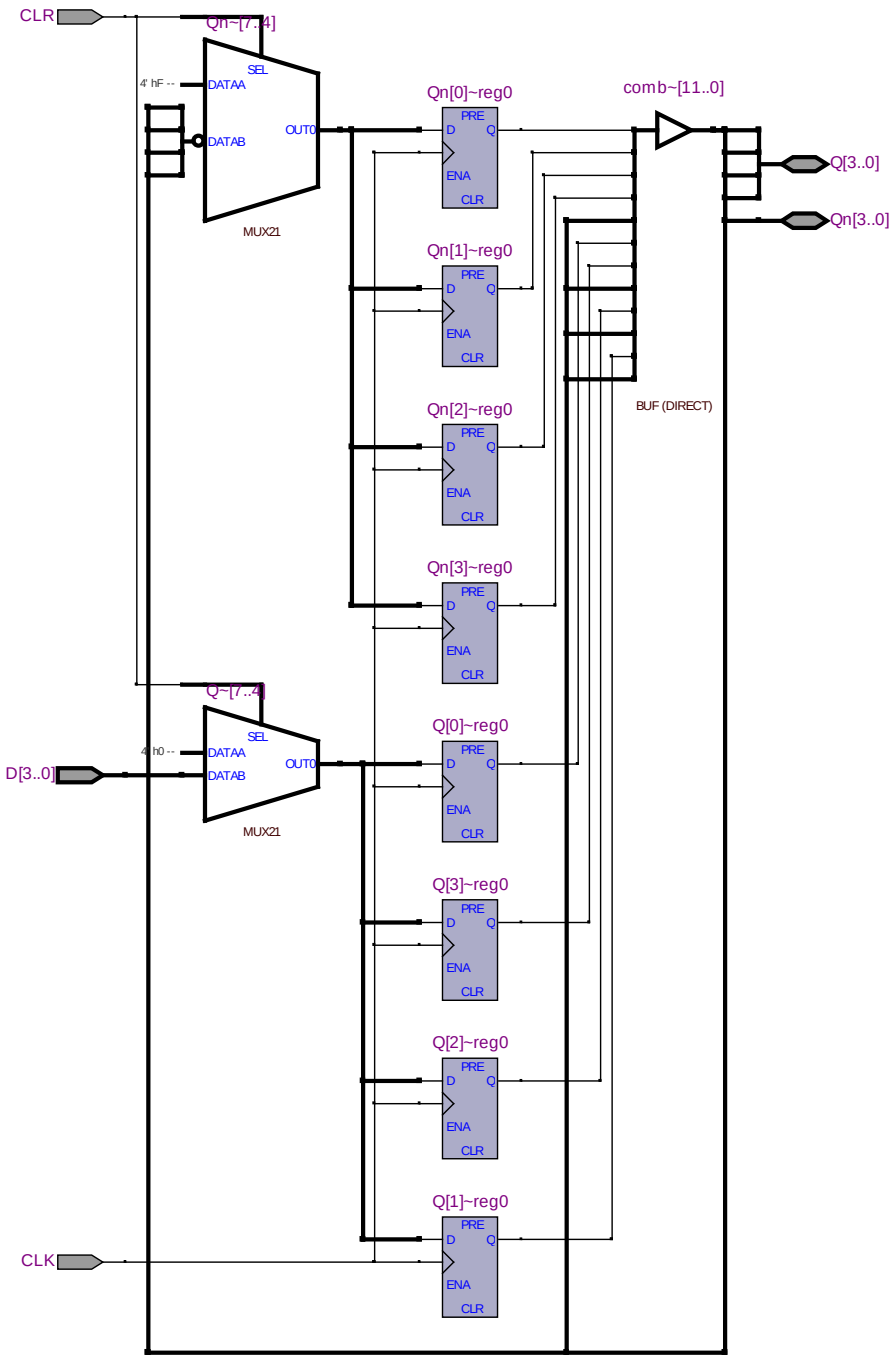


Figura 4. 8: Diseño RTL de la práctica 2.
Fuente: Los autores

4.3. PRÁCTICA 3. Diseño a partir de diagrama de estados.

Se requiere programar el diagrama de estados de la figura 4.9.

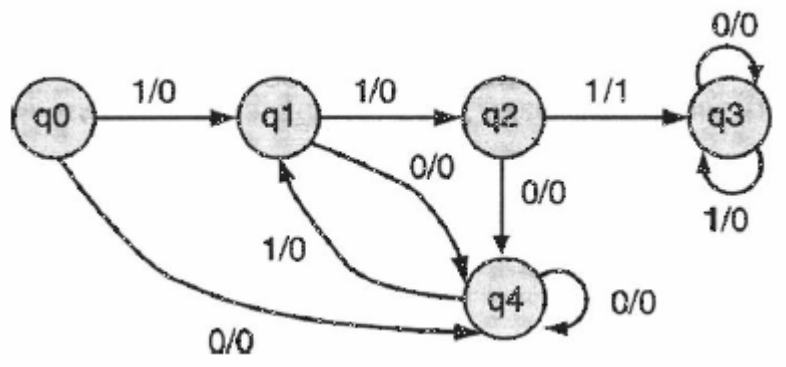


Figura 4. 9: Diagrama de estados para la práctica 3.
Fuente: Los autores

Por ejemplo, en la figura 4.9 notamos que el circuito pasa al estado q1 con el primer ‘1’ de entrada y al estado q2 con el segundo. Las salidas asociadas con estas dos entradas es ‘0’. La tercera entrada consecutiva de ‘1’ genera un ‘1’ en la salida y hace que el circuito pase al estado q3. Una vez que se encuentra en q3, el circuito permanecerá en el mismo estado, emitiendo salidas 0. En lo que respecta a la programación en VHDL, para la práctica 3 seguiremos con la misma metodología que las prácticas anteriores, la cual consiste en usar estructuras *case - when* y tipo de datos enumerado, que en este caso contiene los cinco estados (q0, q1, q2, q3 y q4) que componen el diagrama de estados mostrado en la figura 4.9.

En base al análisis realizado del diagrama de estados mostrado en la figura 4.9, procedemos a realizar la respectiva programación en VHDL. En la figura 4.10 se muestra el listado correspondiente al programa de la práctica 3.

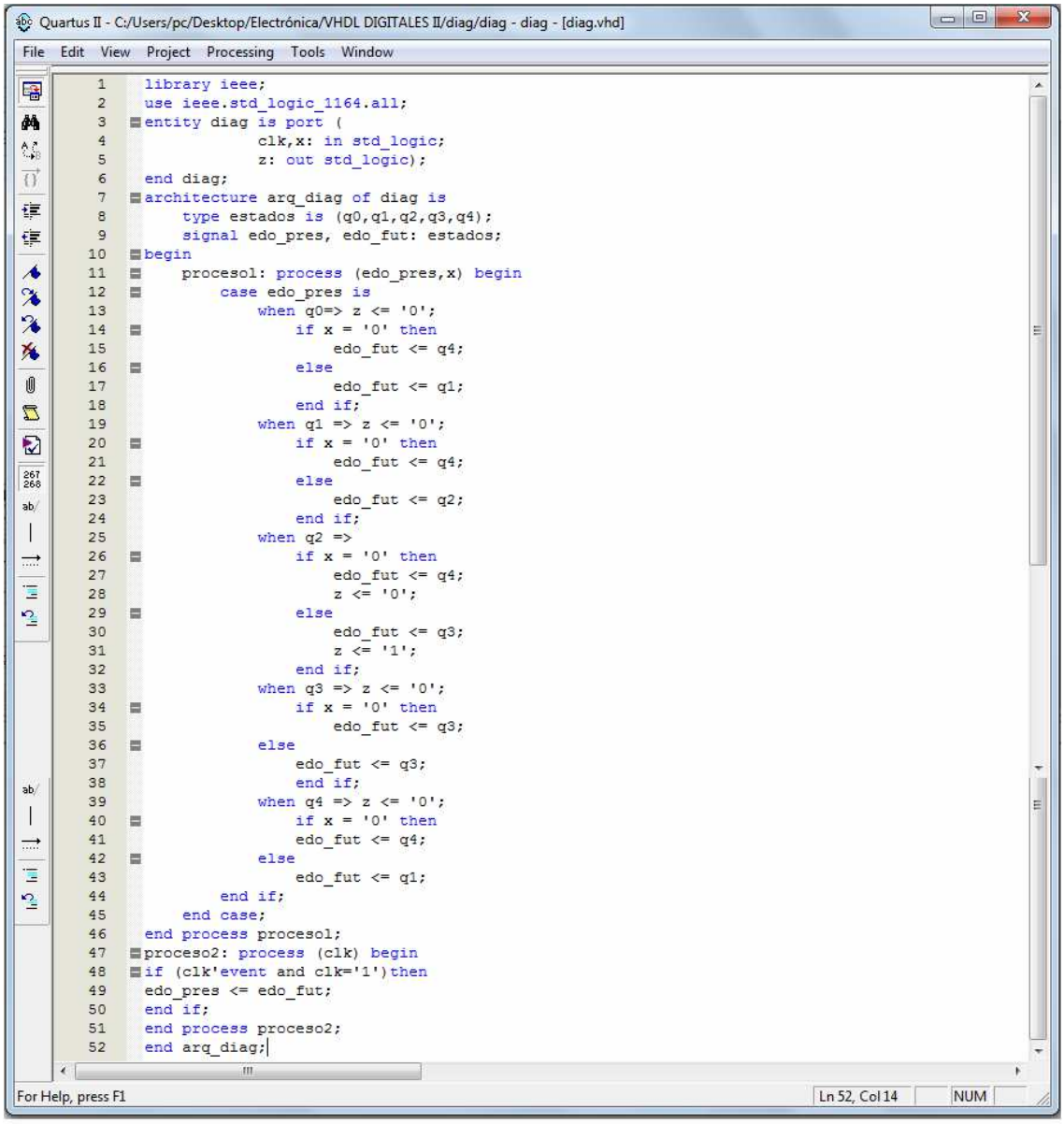


Figura 4. 10: Programa VHDL del registro de 4 bits de la práctica 3.
Fuente: Los autores

En la figura 4.11 se muestra el resultado obtenido de la máquina de estado generada a partir de la programación en VHDL del registro de 4 bits, donde se pudo comprobar la igualdad del diagrama de estados de la figura 4.9.

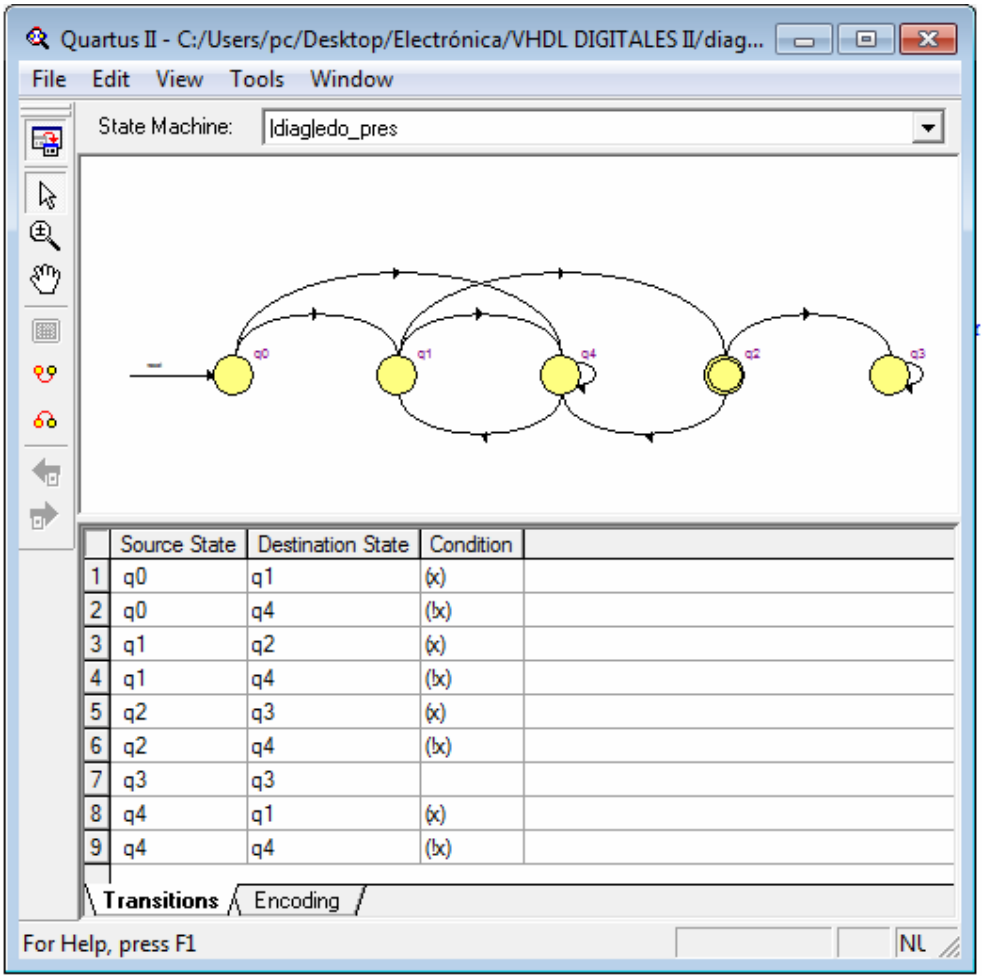


Figura 4. 11: Máquinas de estados del registro de 4 bits de la práctica 3.

Fuente: Los autores

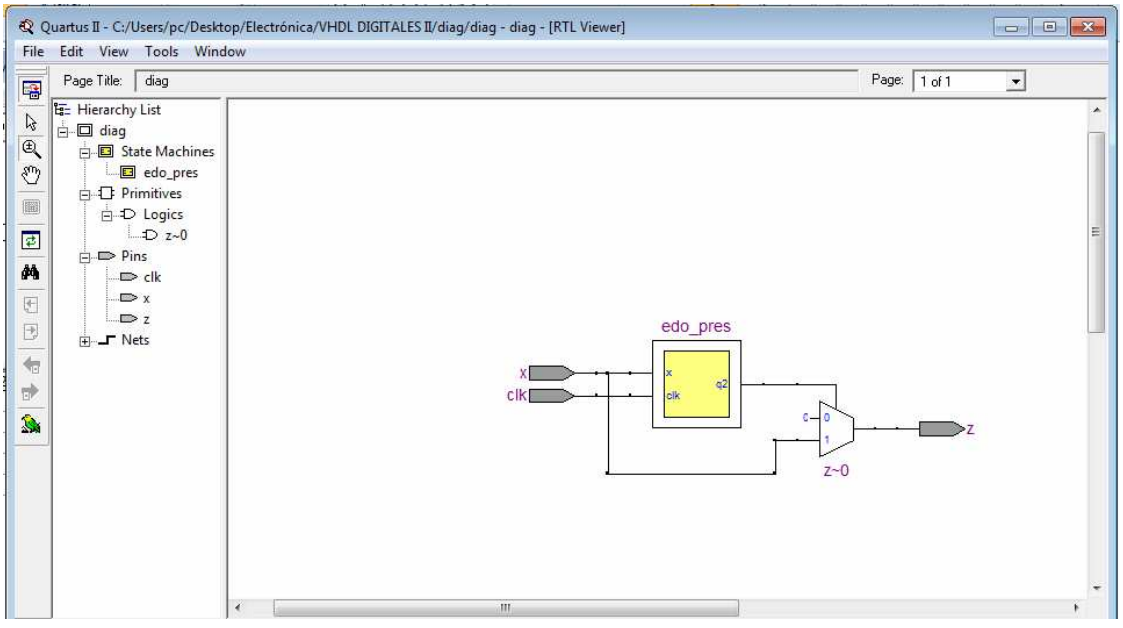


Figura 4. 12: Diseño RTL del registro de 4 bits de la práctica 3.

Fuente: Los autores

4.4. PRÁCTICA 4. Integración de entidades en VHDL.

Hasta el momento se ha empleado la programación en VHDL para diseñar entidades individuales (bloques lógicos mínimos), con el único objeto de familiarizarlo con los diversos estilos de diseño y programación o ambos, así como con el uso y aplicación de las palabras reservadas en VHDL. Sin embargo, es obvia que esta herramienta de diseño no fue creada para este fin.

Existen diversas razones para su empleo; pero tal vez su verdadera fortaleza radica en que permite integrar "sistemas digitales" que contienen una gran cantidad de subsistemas electrónicos con el fin de minimizar el tamaño de la aplicación. En primera instancia, en un solo circuito integrado y si el problema es muy complejo, a través de una serie sucesiva de circuitos programables, sea que se llamen CPLD (dispositivo lógico programable complejo) o FPGA (arreglos de compuertas programables en campo).

Esquema básico de integración de entidades

La integración de entidades puede realizarse mediante el diseño individual de cada bloque lógico a través de varios procesos internos que posteriormente pueden unirse mediante un programa común. Otra posibilidad es observar y analizar de manera global todo el sistema evaluando su comportamiento sólo a través de sus entradas y salidas. En ambos casos el resultado es satisfactorio; más bien, nuestra tarea consiste en analizar las ventajas y desventajas que existen en ambas alternativas de solución. En el primer caso, el inconveniente principal es el número excesivo de terminales utilizadas en el dispositivo, debido a que al diseñar entidades individuales, se tendrían que declarar las terminales de entrada-salida de cada entidad (ver figura 4.13a).

En el segundo caso, cuando observamos el sistema como un todo (ver figura 4.13b) el número de terminales de entrada y salida disminuye, por lo que nuestro trabajo es desarrollar no sólo un algoritmo interno capaz de interpretar el funcionamiento de cada bloque, sino también cómo conectar cada uno de ellos.

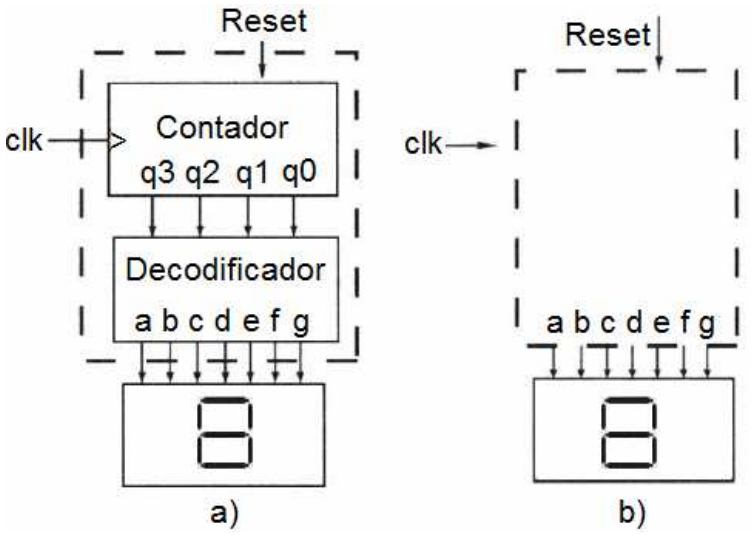


Figura 4. 13: a) Diseño mediante entidades individuales, b) Diseño mediante la relación de entradas/salidas.
Fuente: Los autores

Programación de entidades individuales

Procederemos a la programación del contador y decodificador de la figura 4.13a), mediante la programación de entidades individuales, como el mostrado en la figura 4.14.

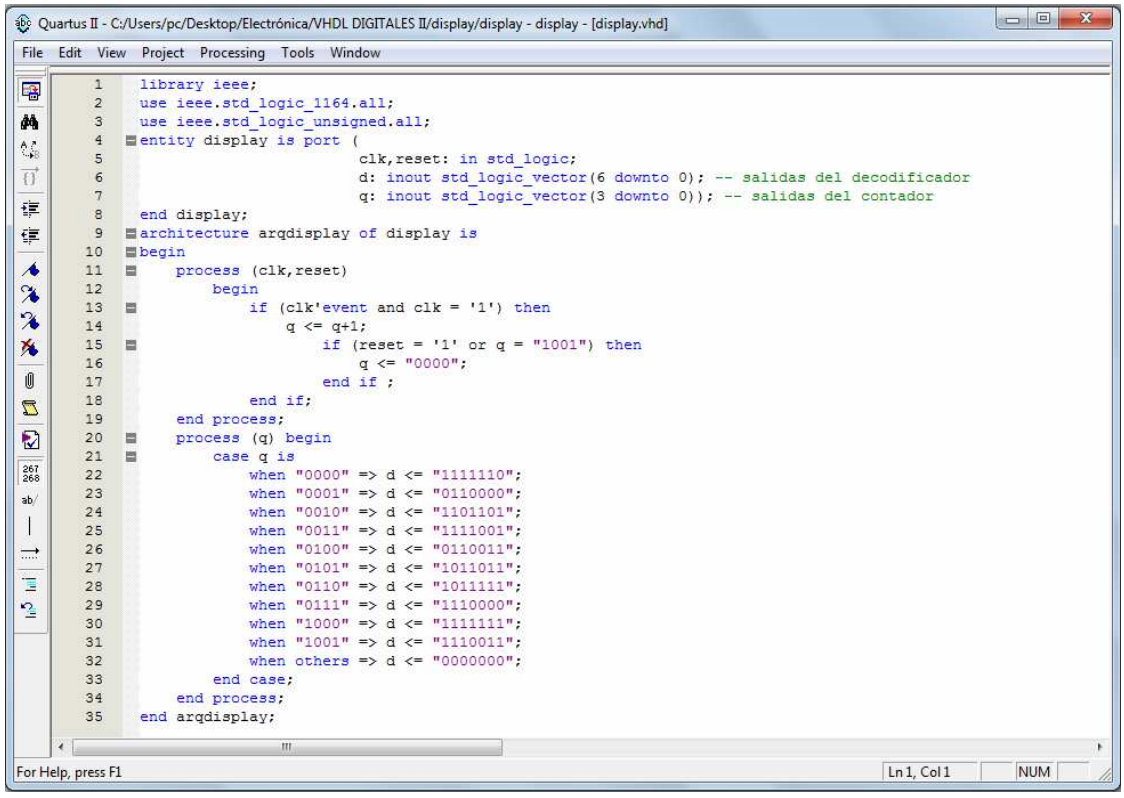


Figura 4. 14: Programación en VHDL mediante entidades individuales.
Fuente: Los autores

Como podemos observar, en la declaración de la entidad (entity) se han asignado las terminales de entrada y salida correspondientes a cada bloque lógico de manera individual; así por ejemplo, el contador tiene las salidas q [q3, q2, q1, q0] y el decodificador las salidas d (da, db, de, dd, de, df, dg). Note que la programación requiere dos procesos: el primero (líneas 11 a 19) describe el funcionamiento del contador; el segundo (líneas 20 a 35), el funcionamiento del decodificador.

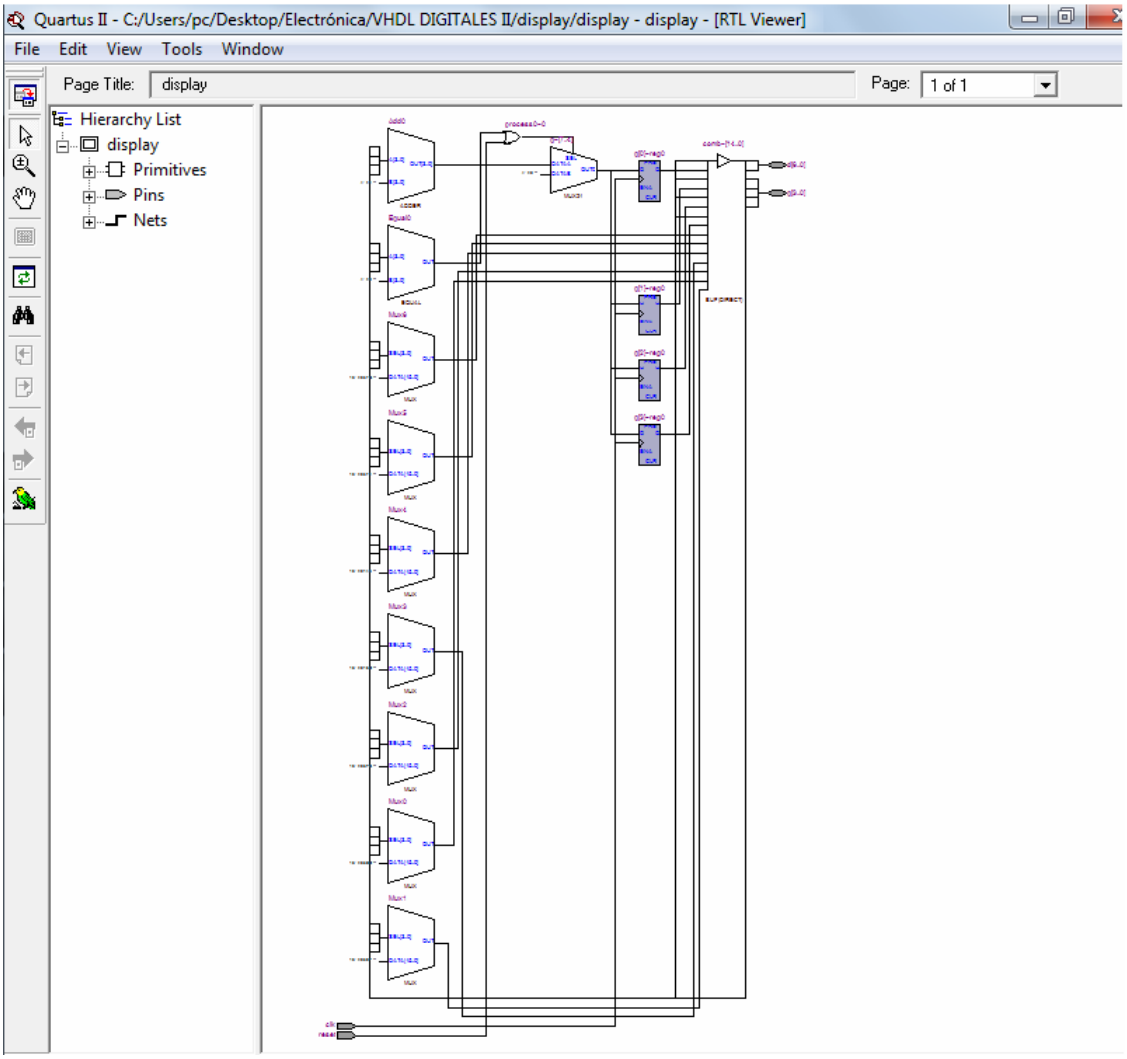


Figura 4. 15: Diseño RTL mediante entidades individuales.
Fuente: Los autores

Programación mediante relación entradas/salidas

Por otro lado, en la figura 4.16 se muestra el programa en VHDL del sistema de la figura 4.13b. Observemos cómo nuevamente el programa supone dos procesos: el primero (líneas 11 a 19) programa el contador y el segundo (líneas 20 a 34) describe el decodificador.

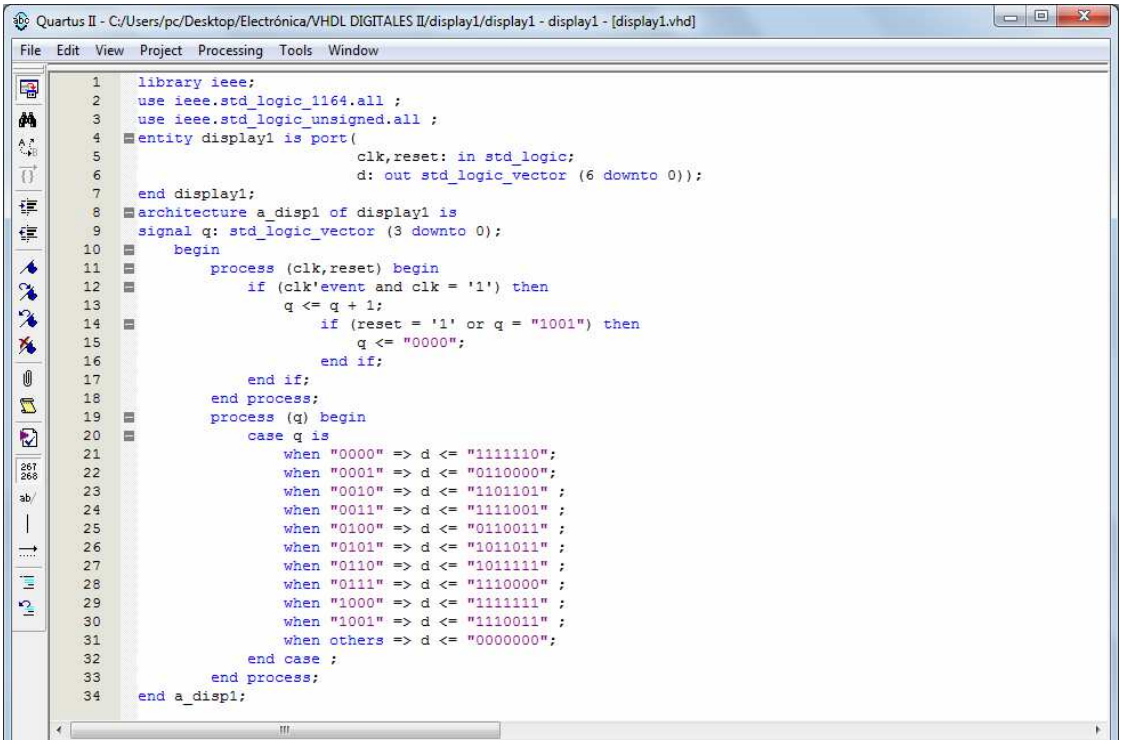


Figura 4. 16: Programación en VHDL mediante relación de entradas / salidas.
Fuente: Los autores

Sin embargo, a diferencia del programa anterior en éste sólo se asignan las terminales de salida correspondientes al decodificador (línea 6), el enlace interno entre el contador y este último se realiza mediante señales (signal) línea 9. De nuevo, es importante mencionar que los dos programas son correctos y que la disponibilidad de terminales y capacidad de integración del dispositivo empleado determinan la utilización de uno u otro.

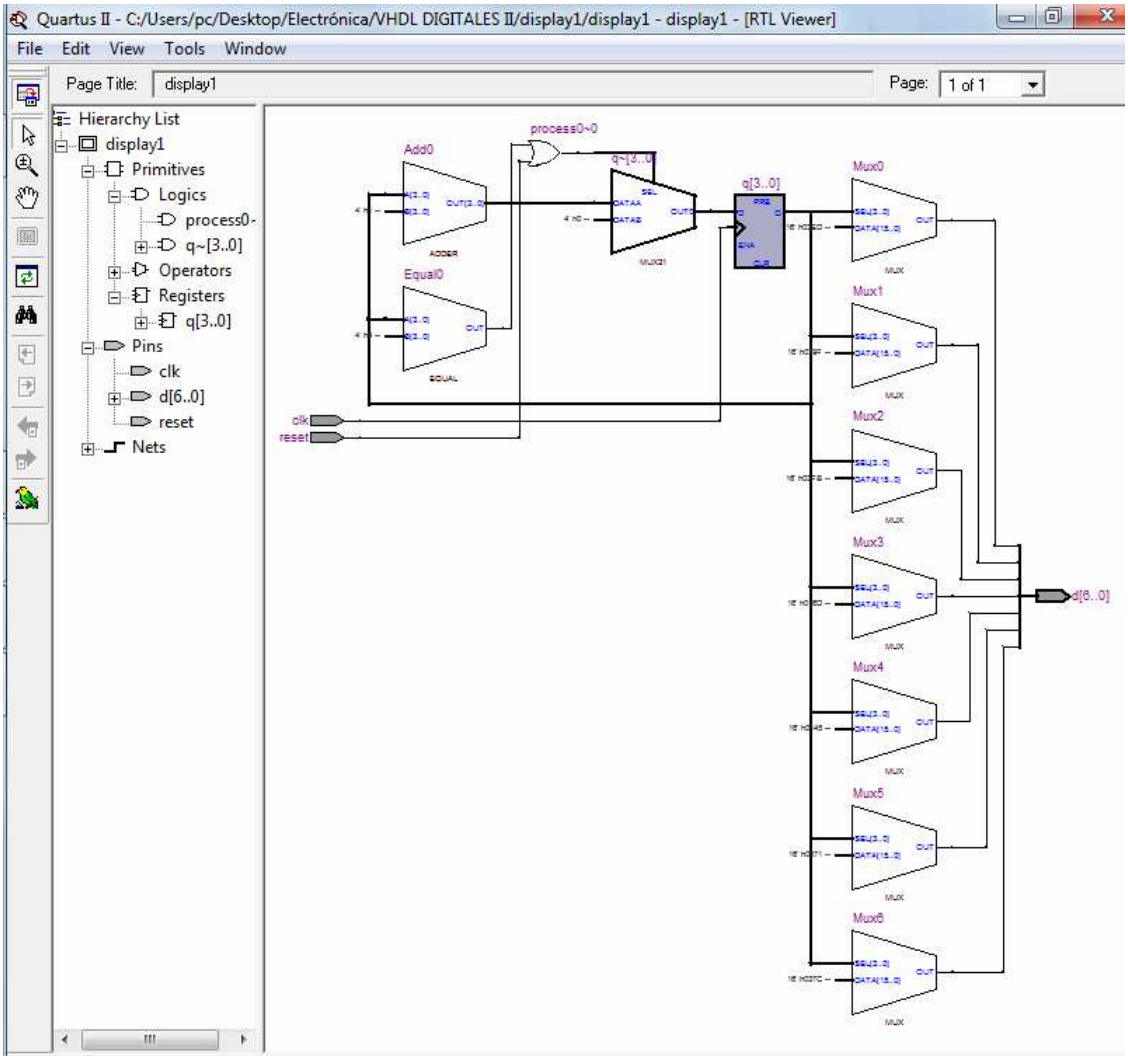


Figura 4. 17: Diseño RTL mediante relación de entradas / salidas.
Fuente: Los autores

4.5. PRÁCTICA 5. Integración de bloques lógicos en VHDL.

Para la siguiente práctica realizaremos la programación en VHDL para integrar diferentes bloques lógicos, como el mostrado por la figura 4.18. El circuito de la figura 4.18 consiste en automatizar procesos de empaquetamiento de muñecas. Considerar que las muñecas se deben empaquetar en forma individual o con un máximo de nueve unidades por paquete. Al inicio el operador selecciona mediante el teclado decimal la cantidad de piezas a empaquetar. Como sabemos, este número decimal se convierte a BCD mediante el circuito codificador y luego pasa mediante el registro hacia el decodificador de 7 segmentos para mostrar en el display el valor del número seleccionado.

Observe que este valor binario es a su vez la entrada A (A3, A2, A1 y A0) del circuito comparador de cuatro bits.

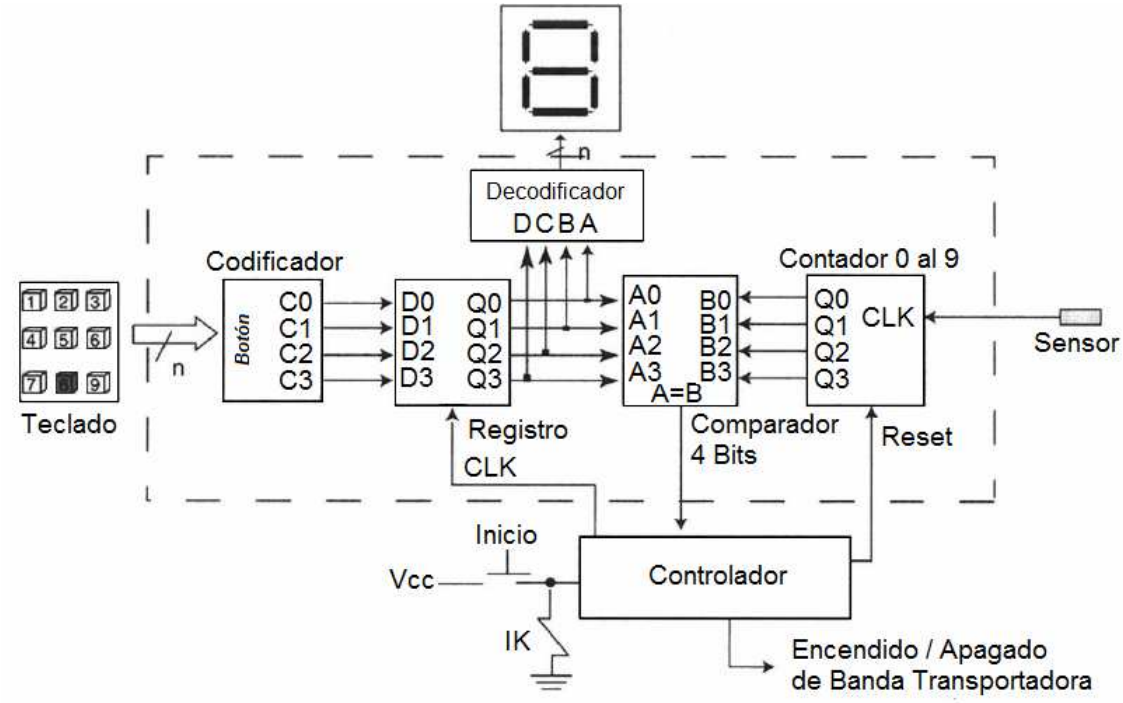


Figura 4. 18: Circuito para automatizar proceso de empaquetamiento de muñecas.
Fuente: Los autores

Una vez explicado el funcionamiento del circuito descrito en la figura 4.18, y de seleccionar la cantidad de muñecas por empaquetar, el operador presiona el botón de inicio, el cual desencadena una serie de acciones controladas por el bloque denominado controlador. El mismo comienza con una señal de salida llamada Reset, que coloca al contador binario en el estado de cero; enseguida envía la señal de arranque (Encendido) al motor que controla el avance de la banda transportadora. Ahora bien, cada vez que una de las muñecas colocadas sobre la banda transportadora pasa por el "sensor" (ver figura 4.19), donde se origina un impulso eléctrico (pulso) que hace que el contador aumente en uno su conteo.

Este procedimiento continúa incrementando al contador en una unidad, hasta el momento en que la cifra actual del contador (número B) es igual al "número A" dentro del comparador (A = B), con lo cual este último envía una señal al bloque controlador que detiene la banda transportadora (Apagado) y marca el fin del proceso.

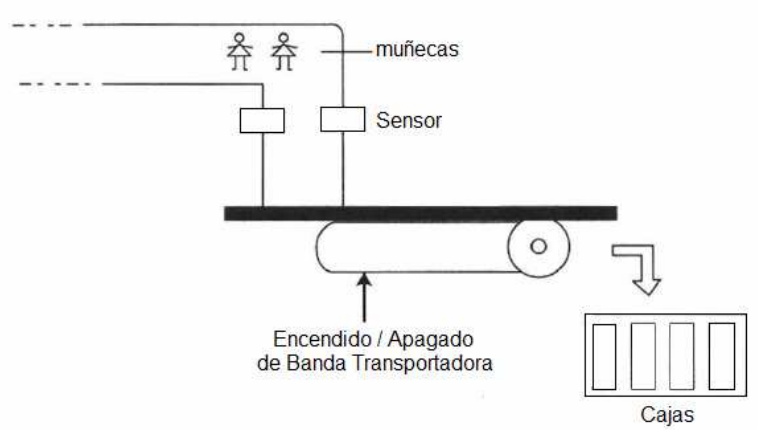


Figura 4. 19: Banda transportadora para empaquetamiento de muñecas.
Fuente: Los autores

En la figura 4.20 se observa el intercambio de señales que se realiza entre el controlador y cada uno de los bloques lógicos del sistema.

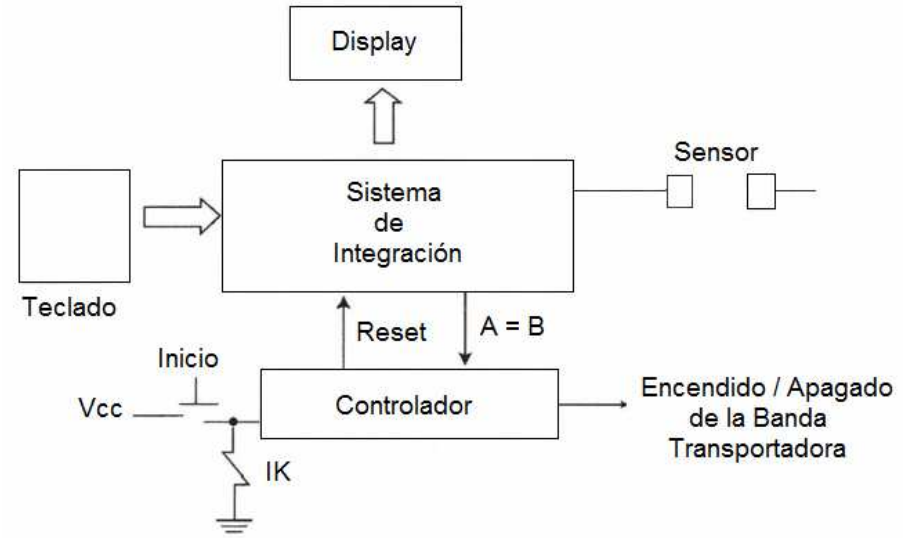
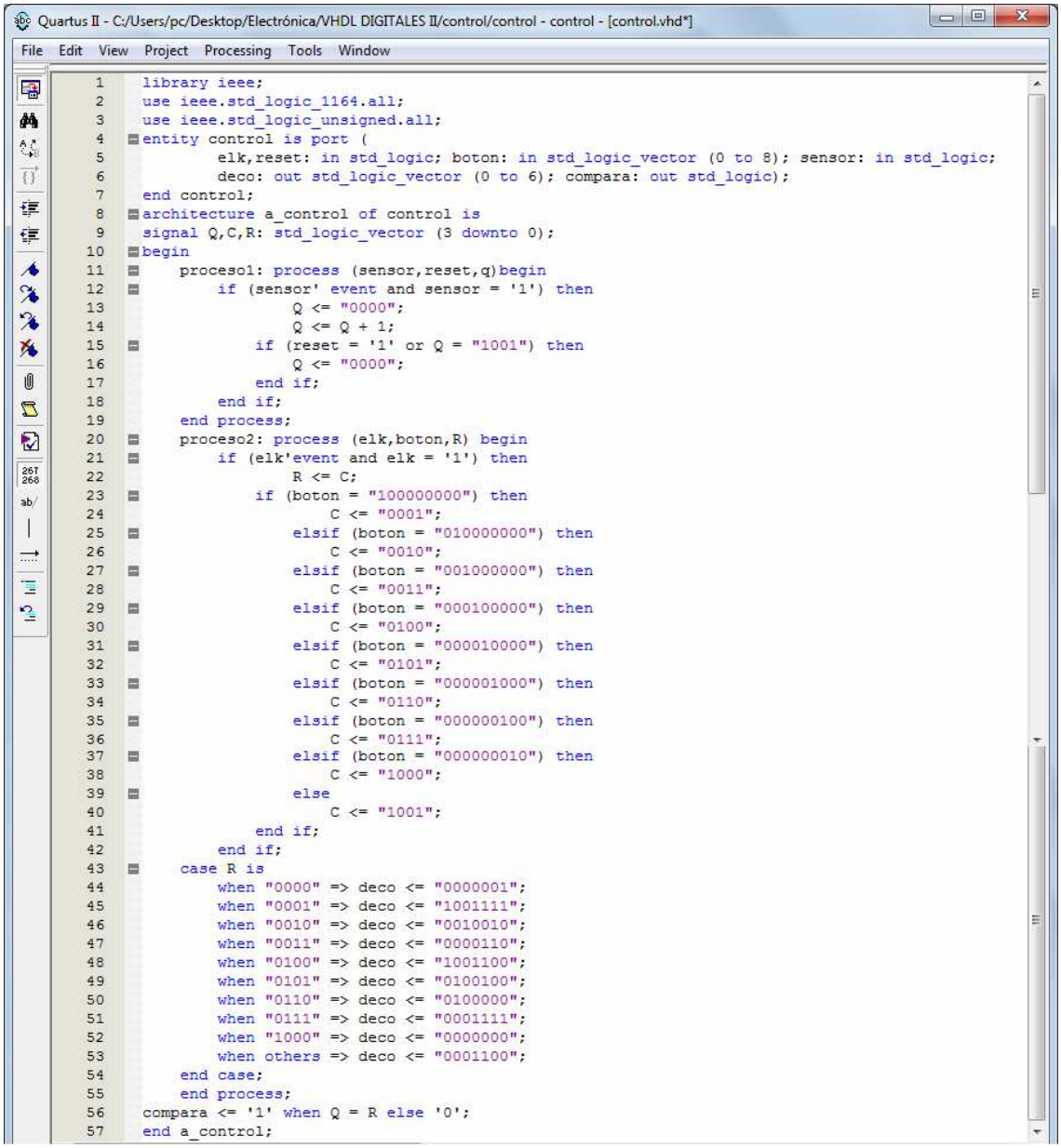


Figura 4. 20: Diagrama de bloques para automatizar proceso de empaquetamiento de muñecas.
Fuente: Los autores

Una vez entendido lo que se requiere el proceso de automatizar el empaquetamiento de muñecas procedemos a realizar la programación correspondiente a esta integración que se muestra en la figura 4.21.




```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3  use ieee.std_logic_unsigned.all;
 4  entity control is port (
 5          elk,reset: in std_logic; boton: in std_logic_vector (0 to 8); sensor: in std_logic;
 6          deco: out std_logic_vector (0 to 6); compara: out std_logic);
 7  end control;
 8  architecture a_control of control is
 9  signal Q,C,R: std_logic_vector (3 downto 0);
10  begin
11      proceso1: process (sensor,reset,q)begin
12          if (sensor' event and sensor = '1') then
13                  Q <= "0000";
14                  Q <= Q + 1;
15              if (reset = '1' or Q = "1001") then
16                  Q <= "0000";
17              end if;
18          end if;
19      end process;
20      proceso2: process (elk,boton,R) begin
21          if (elk'event and elk = '1') then
22                  R <= C;
23              if (boton = "100000000") then
24                      C <= "0001";
25                  elsif (boton = "010000000") then
26                      C <= "0010";
27                  elsif (boton = "001000000") then
28                      C <= "0011";
29                  elsif (boton = "000100000") then
30                      C <= "0100";
31                  elsif (boton = "000010000") then
32                      C <= "0101";
33                  elsif (boton = "000001000") then
34                      C <= "0110";
35                  elsif (boton = "000000100") then
36                      C <= "0111";
37                  elsif (boton = "000000010") then
38                      C <= "1000";
39                  else
40                      C <= "1001";
41              end if;
42          end if;
43      case R is
44          when "0000" => deco <= "0000001";
45          when "0001" => deco <= "1001111";
46          when "0010" => deco <= "0010010";
47          when "0011" => deco <= "0000110";
48          when "0100" => deco <= "1001100";
49          when "0101" => deco <= "0100100";
50          when "0110" => deco <= "0100000";
51          when "0111" => deco <= "0001111";
52          when "1000" => deco <= "0000000";
53          when others => deco <= "0001100";
54      end case;
55      end process;
56  compara <= '1' when Q = R else '0';
57  end a_control;
```

Figura 4. 21: Programación en VHDL proceso de empaquetamiento de muñecas.
Fuente: Los autores

Asimismo como en anteriores prácticas procedemos siempre a realizar la compilación, la misma que no debe tener ningún error para así obtener el diseño RTL que se muestra en la figura 4.22, del circuito que permite automatizar el proceso de empaquetamiento de muñecas.

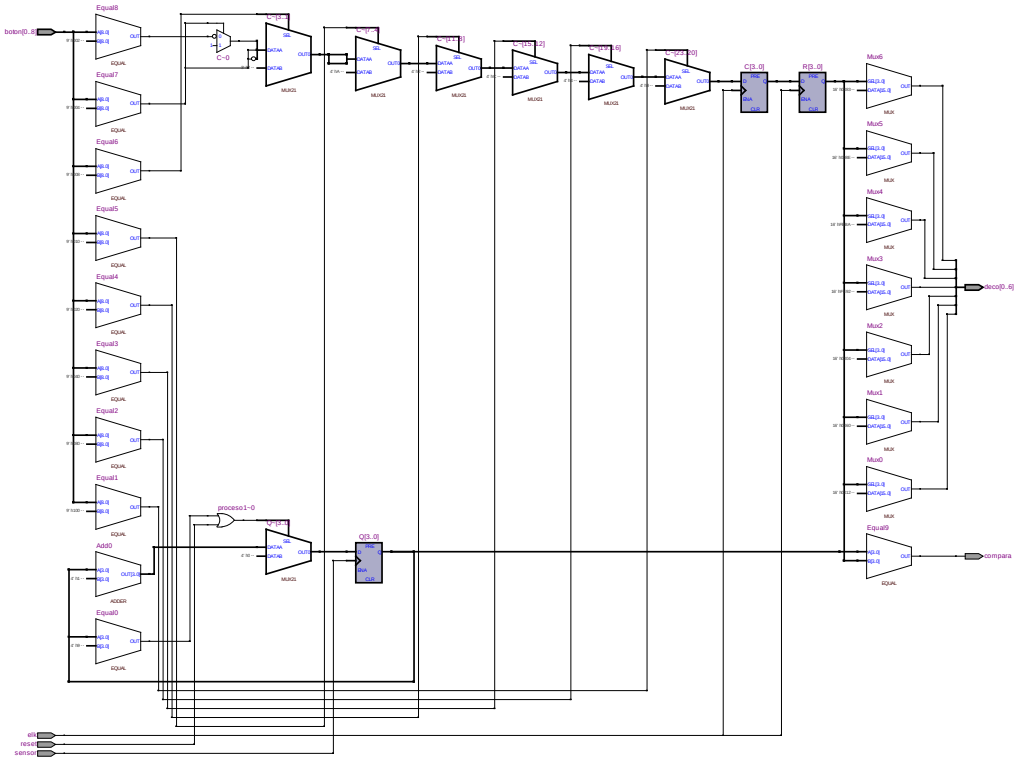


Figura 4. 22: Circuito RTL del proceso de empaquetamiento de muñecas.
Fuente: Los autores

CAPÍTULO 5: CONCLUSIONES Y RECOMENDACIONES.

5.1. CONCLUSIONES

- A través del estado del arte o fundamentos teóricos de la programación en VHDL y de FPGA se pudo determinar que dichas herramientas son empleadas en instituciones de educación superior para la enseñanza de sistemas digitales en cuanto al diseño de controladores.

- Se determinó la eficiencia de la herramienta a utilizar en el Laboratorio de Electrónica específicamente en la asignatura de Sistemas Digitales II en la Carrera de Ingeniería en Telecomunicaciones, para contribuir a su aprendizaje significativo a los estudiantes de la Facultad de Educación Técnica para el Desarrollo en la Universidad Católica de Santiago de Guayaquil.

- Con el desarrollo de la programación y posterior implementación de estas prácticas de sistemas digitales II, el estudiante de VI Ciclo de la Carrera de Ingeniería en Telecomunicaciones estará en capacidad de realizar diseños electrónicos que a futuro servirán como parte de su vida profesional, así adquirir nuevos conocimientos y generar nuevas competencias.

5.2. RECOMENDACIONES

- Realizar la compra de la licencia profesional del software Quartus II de Altera para aplicaciones de mayor envergadura que requieren de librerías que no cuentan la versión gratuita para estudiantes, así como también las gestiones apropiadas para la adquisición de nuevas tarjetas de entrenamiento en FPGA como Cyclone II o mayores, para la materia de Sistemas Digitales II, tomando como referencia el valor de estas tarjetas (ver Anexo 1).

- Contribuir al desarrollo investigativo con aplicaciones y adaptaciones orientados en la carrera de telecomunicaciones, tales como la biomedicina.

- Capacitar al personal involucrado en el cuidado y mantenimiento del laboratorio de electrónica específicamente en las actualizaciones de licencia e instalación de los drivers para compatibilidad de la tarjeta Cyclone II de Altera para así aprovechar en forma eficiente los recursos ya existentes.

REFERENCIAS BIBLIOGRÁFICAS

[AGRAWAL, 1986] AGRAWAL, Om, SHANKAR, Kapil, "PAL device buries registers, brings state machines to life", Electronic Design, 24-7-1986, pp. 101-106.

[CHAM, 1988] CHAM, K.M., SOO-YOUNG, CHIN, D., MOLL, J.L., LEE, K., VOORDE, P.V., *"Computer-aided design and VLSI device development"*, Kluwer academic publishers, 1988.

[Delgado, 1993] Delgado C., Lecha E., Moré M., Teres Ll., Sánchez L., *Introducción a los lenguajes VHDL, Verilog y UDL/I.* Novática No. 112, España, 1993.

[IEEE, 1997] IEEE, Instituto de Ingeniería Eléctrica y Electrónica, *Revista Computer IEEE*, 1977.

[LUPON, 1988] LUPON, Emili, RUBIO, José Antonio, *"Improving performance of PLDs"*, Departamento de Ingeniería Electrónica, Universidad de Cataluña, Barcelona, 1988.

[MM, 1981] "Programmable array logic handbook", Monolithic Memories, 1981.

[MM, 1986] "LSI data book", Monolithic Memories, 1986.

[NI, 2011] NI, National Instruments, “Introducción a la Tecnología FPGA: Los Cinco Beneficios Principales” Tutorial NI, 2011.

[PHILIPS, 1987a] "Semi-custom programmable logic devices data handbook", Philips, 1987.

[PHILIPS, 1987b] "System Cell 1987. Design manual. Cell library", Philips, 1987.

[RANDELL, 1982] RANDELL, B., TRELEAVEN, P.C., "VLSI architecture", Prentice Hall, New Jersey, 1982.

[Teres, 1998] Teres Ll., Torroja Y., Olcoz S., Villar E., *VHDL Lenguaje Estándar de Diseño Electrónico*. McGraw-Hill, 1998.

[TEXAS, 1990] "Semiconductor products, circuits design tools and support", Texas Instruments, 1990.

[XilinxDK, 2003] Xilinx Inc., “CoolRunner-II RealDigital CPLD,” Xilinx Design Kit, USA, Nov. 2003.

[Pardo, 2000] F. Pardo y J. A. Boluda, “VHDL Lenguaje para Síntesis y Modelado de Circuitos,” Alfaomega ra-ma, España, 2000.

Bibliografía complementaria

T.L. Floyd: *Fundamentos de* sistemas *digitales.* Prentice Hall, 1998.

Ll. Teres; Y. Torroja; S. Olcoz; E. Villar: *VHDL Lenguaje estándar de diseño electrónico.* McGraw-Hill, 1998.

David G. Maxinez, Jessica Alcalá: *Diseño de Sistemas Embebidos a través del Lenguaje de Descripción en Hardware VHDL.* XIX Congreso Internacional Académico de Ingeniería Electrónica. México, 1997.

C. Kloos, E. Cerny: *Hardware Description Language and their Aplications. Specification, modelling, verification and synthesis of microelectronic systems.* Chapman &Hall, 1997.

IEEE: *The IEEE standard VHDL Language Reference Manual.* IEEE-Std-1076-1987, 1988

Zainalabedin Navabi: *Analysis and Modeling of Digital Systems.* McGraw-Hill, 1988.

E J. Ashenden: *The Designer's guide to VHDL*. Morgan Kauffman Publishers, Inc., 1995.

R. Lipsett, C. Schaefer: *VHDL Hardware Description and Design.* Kluwer Academic Publishers, 1989.

S. Mazor, P Langstraat: A *guide to VHDL.* Kluwer Academic Publishers, 1993.

J.R. Armstrong y F. Gail Gray: *Structured Design with VHDL.* Prentice Hall, 1997.

K. Skahill: *VHDL for Programmable Logic.* Addison Wesley, 1996.

J. A. Bhasker: A *VHDL Primer.* Prentice Hall, 1992.

H. Randolph: *Applications of VHDL to Circuit Design.* Kluwer Academic Publisher.

Anexo 1

COMPROBANTE DE PEDIDO Y COMPRA DE 8 TARJETAS FPGA

Digi-Key CORPORATION

Orders 1-800-344-4539
Fax 218-681-3380
www.digikey.com

Invoice # 36487894
U.S. $

701 Brooks Ave. South, P.O. Box 677, Thief River Falls, MN 56701-0677 USA

Sold To:
CUSTOMER 6296633
JANET PARRA ORTEGA
CDLA KENNEDY NORTE Y AV
A ANDRADE ED CLINICA SAN FRANC
GUAYAQUIL, GUAYA 090150
ECUADOR

Bill To:
JANET PARRA O.
SOLNET
CDLA KENNEDY NORTE Y AV
A ANDRADE ED CLINICA SAN FRANC
GUAYAQUIL GUAYA EC090150
ECUADOR

Terms	Invoice Date	Page
Mastercard	26-OCT-2011	1
Customer Purchase Order		**Sales Order** 31295561
Back Orders: Accepts to 16-NOV-2011		Account 1826353
Entered By / Date: A0FX/17-OCT-2011	Shipped Via: UGT	Ship Date: 26-OCT-2011

Easy to Remember: 1-800-DIGI-KEY

For Office Use Only	Received	Prev Sales Order	Prev Invoice	Billing	Pack List No.	Printing Date	Currency Type	MSC #
	INTERNET	0	0	BILL SHIP	1	26-OCT-2011	U.S. $	0

Idx	Box	Ordered	Cancelled	Shipped	Item Number/Description	Back Order	Unit Price US $	Amount US $
1	1	8	0	8	P0528 BOARD DEV DE1 ALTERA CUST REF #: CARLOS ANTONIO BENALCAZAR PARRA (TERASIC) HTSUS: 8471.41.0150 ECCN: 3A001A7A LIC: EXC ENC LEAD: LEAD FREE ROHS: ROHS COMP COUNTRY/ORIGIN: TAIWAN BOX 1 SHIPPED UGT WEIGHT 18 LBS 0 OZS BOX ID 1Z5674320337712178 ATTN RECIPIENT: EXPORT OF THESE OR ANY COMMODITIES MAY CONSTITUTE A ROUTED TRANSACTION. THIS INVOICE CONTAINS THE NECESSARY INFORMATION TO FILE THE EXPORT UNDER THE U. S. DEPARTMENT OF CENSUS AUTOMATED EXPORT SYSTEM (AES), WHEN REQUIRED BY 15 CFR PART 30, FOREIGN TRADE REGULATIONS.		156.25000	1250.00
					TOTAL INVOICED SHIPPING CHARGES APPLIED ** CHARGES SUBTOTAL ** TOTAL CHARGED TO CREDIT CARD U.S. $$			1250.00 21.21 1271.21 1271.21
					YOUR CREDIT CARD HAS BEEN CHARGED THE ABOVE INDICATED AMOUNT THE ORDER IS COMPLETE			
				Ship To:	HERBERT ALVAREZ ABOARD CARGO SERVICE INC 8560 NW 72 STREET MIAMI FL 33166-0000			
				Ship From:	DIGI-KEY CORPORATION 701 BROOKS AVE. SOUTH P.O. BOX 677 THIEF RIVER FALLS MN 56701-0677			
				General -	WEB ORDER ID: 37692495 17-OCT-2011 CALLED JANET TO VERIFY PERSONAL OR COMPANY ORDER. SHE IS ORDERING FOR HER SON CARLOS TO USE AT UNIVERSITY. A2P9/2236			

Claims for pricing errors, shortages, and defective product must be reported within 30 days of invoice date.

Contact Customer Service at 1-800-858-3616

Anexo 2