UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

CAMPUS ARAGON

VHDL
EL ARTE DE LA PROGRAMACION DE SISTEMAS DIGITALES

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A :

JESSICA ALCALA JARA

ASESOR: M. en I. DAVID G. MAXINEZ

MEXICO, D. F. SEPTIEMBRE, 1999.

Índice

CAPÍTULO 3. SÍNTESIS DE DISEÑO E IMPLEMENTACIÓN

CAPÍTULO 4. SÍNTESIS DE LÓGICA COMBINACIONAL Y SECUENCIAL

CAPÍTULO 5. DISEÑO JERÁRQUICO EN VHDL

Introducción

Hoy en día, en nuestro ambiente familiar o de trabajo nos encontramos rodeados de sistemas electrónicos de alta sofisticación; teléfonos celulares, computadoras personales, televisores portátiles, equipos de sonido, dispositivos de comunicaciones, estaciones de juego interactivo, etc., no son mas que algunos ejemplos del desarrollo tecnológico que ha cambiado nuestro estilo de vida haciéndolo cada vez más confortable. Todos los sistemas anteriores tienen una misma similitud, su tamaño, de dimensiones tan pequeñas que parece increíble que sean igual o más potentes que los sistemas grandes que existieron hace algunos años. Estos avances son posibles gracias al desarrollo de la *microelectrónica,* la cual ha permitido la miniaturización de los componentes obteniéndose mayores prestaciones de los chips (circuitos integrados) y ampliando el rango de posibilidades de los ingenieros de aplicación.

La evolución en el desarrollo de los circuitos integrados se ha venido perfeccionando a través de los años, iniciando con los circuitos de baja escala de integración ***SSI*** **(Small Scale Integration)**, siguiendo con los de mediana escala de integración ***MSI*** **(Medium Scale Integration)**, para continuar con los de muy alta escala de integración, ***VLSI*** **(Very Large Scale Integrated)** hasta llegar a los circuitos integrados de propósito específico **(ASICs)**.

Actualmente, los creadores de tecnología perfeccionan el diseño de los circuitos integrados orientados a una aplicación y/o solución especifica (ASIC´s), logrando dispositivos muy potentes y que ocupan un mínimo de espacio. Estos chips optimizan su diseño siguiendo dos vertientes en su conceptualización; primero mediante la técnica de ***Full custom desing*** *(Diseño totalmente a la medida)* que consiste en desarrollar mediante la integración de transistor por transistor un circuito para una aplicación específica, siguiendo en su fabricación los pasos tradicionales de diseño: preparación de la oblea o base, el crecimiento epitaxial, la difusión de impurezas, la implantación de iones, la oxidación, la fotolitografía, la metalización y la limpieza química [1].

En el segundo caso el diseño de ASICs proviene de una innovadora propuesta, que sugiere la utilización de celdas programables preestablecidas e insertadas dentro de un circuito integrado. Esta idea dio surgimiento a la familia de dispositivos lógicos programables (PLDs), cuyo nivel de densidad de integración ha venido evolucionando a través del tiempo, iniciando con los arreglos lógicos programables (PAL), hasta el uso de los CPLDs (Dispositivos Lógicos Programables Complejos) y FPGAs (Arreglos de Compuertas Programables en Campo) los cuales y dada su conectividad interna sobre cada una de sus celdas ha hecho posible el desarrollo de circuitos integrados de aplicación especifica de una forma mucho mas fácil y también económica, en beneficio de los ingenieros integradores de sistemas.

El desarrollo de este trabajo se encuentra orientado hacia este tipo de diseño, sin embargo y como parte fundamental de esta investigación se me brinda la oportunidad de presentar, manejar y aplicar de una forma accesible el lenguaje de programación mas poderoso para este tipo de aplicaciones... **VHDL.**

VHDL "El Lenguaje de Descripción en Hardaware" considerado como la máxima herramienta de diseño por las industrias y universidades de primer mundo, brinda y proporciona el detalle de elegancia en la planeación y diseño de los sistemas electrónicos digitales.

Teniendo presente las observaciones arriba señaladas, el trabajo realizado tiene varios objetivos: primero, presentar el estado del arte actual de los dispositivos lógicos programables y su campo de aplicación. Segundo, proporcionar al ingeniero en computación una forma de programar las aplicaciones digitales utilizando el lenguaje de descripción en hardware VHDL; finalmente brindar la oportunidad al estudiante de crecer como microempresario, comprometido con el desarrollo de su entorno social, generando los empleos que brinda esta área de desarrollo en beneficio de nuestro País.

Bajo esta idea, la investigación se encuentra estructurada en cinco capítulos y cuatro apéndices. En síntesis, el capítulo uno ***"El Estado del Arte de la Lógica Programable"***, describe de forma cualitativa el estado actual de la lógica programable, sus antecedentes y perspectivas, además proporciona la información referente a las compañías que brindan el soporte en software y hardware para adentrarse en este campo de investigación. El capítulo dos ***" VHDL Su Organización y Arquitectura"***, presenta la estructura básica del lenguaje de descripción en hardware VHDL, así como los diferentes arquitecturas empleadas en la programación introduciendo al lector en el desarrollo de sus primeros programas. En el capítulo tres ***"Síntesis de Diseño e Implementación"*** se describe el funcionamiento de las herramientas de software empleadas a lo largo de este trabajo, describiéndolas de una forma amigable e interactiva, haciendo una descripción detallada de cómo manejar la programación, la simulación e implementación dentro de un circuito CPLD. El capítulo cuatro ***"Síntesis de Lógica Combinacional y Secuencial"***, realiza una síntesis de diseño de los principales circuitos combinacionales y secuenciales resueltos a través de diferentes estilos de diseño, proporcionando al lector un panorama completo y general de la potencialidad de este lenguaje. Finalmente el capitulo cinco ***"Diseño Jerárquico en VHDL"***, detalla el funcionamiento del diseño jerárquico utilizado en la solución de problemas extensos, dividiéndolo en pequeños subsistemas que pueden particularmente analizarse y simularse para después integrarse en un todo mediante el diseño de un nuevo paquete.

El apéndice A proporciona los principales identificadores, tipos y atributos que maneja VHDL así como la sintaxis utilizada en cada declaración. El apéndice B contiene todas las palabras reservadas por VHDL así como también los diferentes tipos de operadores definidos en su estructura. EL apéndice C contiene toda la información referente a la instalación del software de soporte, en este caso **WARPR4** de Cypress semiconductor. Finalmente el Apéndice D maneja las hojas de datos técnicos utilizados en el desarrollo de este trabajo.

Capítulo 1

El Estado del Arte de la Lógica Programable

En la actualidad el nivel de integración alcanzado con el desarrollo de la microelectrónica ha hecho posible desarrollar sistemas completos dentro de un solo circuito integrado SOC (System On Chip), mejorando de manera relevante características tales como velocidad, confiabilidad, consumo de potencia y sobre todo el área de diseño. Esta última característica es la que nos ha permitido observar día a día cómo los sistemas de uso industrial, militar y de consumo han minimizado el tamaño de sus desarrollos; por ejemplo los teléfonos celulares, computadoras personales, calculadoras de bolsillo, agendas electrónicas, relojes digitales, sistemas de audio, sistemas de telecomunicaciones, etc., no son mas que aplicaciones típicas que muestran la evolución de los circuitos integrados también conocidos como chips.

El proceso de miniaturización de los sistemas electrónicos comenzó con la interconexión de elementos discretos como resistencias, capacitores, resistores, bobinas, etc., todos ellos colocados entre sí en un chasis reducido con una escasa separación. Posteriormente se diseñaron y construyeron los primeros circuitos impresos, que actualmente siguen vigentes, los cuales relacionan e interconectan los elementos antes mencionados a través de cintas delgadas de cobre adheridas a un soporte aislante (generalmente baquelita) que permite el montaje de estos elementos.

Posteriormente, el desarrollo del transistor de difusión planar construido durante 1947 y 1948, permitió en 1960 la fabricación del primer circuito integrado monolítico el cual integra cientos de transistores, resistencias, diodos y capacitores, todos ellos fabricados sobre una sola pastilla de silicio. El término monolítico dentro de la tecnología de los circuitos integrados se deriva de las raíces griegas "mono" y "lithos" que significan uno y piedra, respectivamente. Por lo tanto, un circuito monolítico está construido sobre una piedra única o cristal de silicio. Dicho cristal tiene generalmente un grosor de 0.25mm y una cubierta de $1mm^2$ a $10mm^2$ que contiene tanto elementos activos (transistores, diodos) como elementos pasivos (resistencias, capacitores), y las conexiones entre ellos.

La fabricación de los circuitos monolíticos se basa en los mismos principios de materiales, procesos y diseño que constituyen la tecnología altamente desarrollada de los transistores y diodos individuales. Dicha fabricación incluye la preparación de la oblea o base, el crecimiento epitaxial, la difusión de impurezas, la implantación de iones, la oxidación, la fotolitografía, la metalización y la limpieza química.

La integración de sistemas se ha superado a medida que surgen nuevas tecnologías de fabricación, obteniéndose componentes estándar de mayor complejidad y que nos brindan mayores beneficios. Sin embargo, el desarrollo de nuevos productos requiere de bastante tiempo, por lo cual en la actualidad solo se emplea cuando se necesita un alto volumen de producción.

Una forma mas rápida y directa de integrar aplicaciones es mediante el uso de la lógica programable, la cual permite independizar el proceso de fabricación con el proceso de diseño fuera de la fábrica de semiconductores. Esta idea en sus inicios fue conceptualizada por Hon & Sequin y Conway &Mead a finales de los años sesenta.

1.1 Dispositivos Lógicos Programables

Los dispositivos lógicos programables (también llamados por sus siglas en inglés *PLDs*) favorecen la integración de aplicaciones y desarrollos lógicos, mediante el empaquetamiento de soluciones en un solo circuito integrado obteniéndose como consecuencia la reducción de espacio físico dentro de la aplicación, es decir, se trata de dispositivos totalmente fabricados y verificados que se pueden personalizar desde el exterior mediante diversas técnicas de programación. El diseño se basa en bibliotecas y mecanismos específicos de mapeado de funciones, mientras que su implementación tan sólo requiere de una fase de programación del dispositivo que habitualmente realiza el propio diseñador en unos pocos segundos.

Actualmente la tendencia en el desarrollo de aplicaciones a nivel microelectrónica, es mediante el diseño de ASICs (Circuitos Integrados desarrollados para Aplicaciones Especificas), tendencia que presenta varias alternativas de desarrollo como se observa en la tabla 1.

A nivel de ASICs los desarrollos full y semi custom ofrecen grandes ventajas en sistemas que emplean circuitos diseñados para una aplicación en particular, sin embargo, su diseño ahora es solo relevante en aplicaciones que requieren un alto volumen de producción, por ejemplo los sistemas de telefonía celular, computadoras portátiles, cámaras de vídeo, etc.

Los FPGAs y CPLDs ofrecen las mismas ventajas de un ASIC, solo que a un menor costo, es decir, el costo por desarrollar un ASIC es mucho mas alto que el de un FPGA (Arreglos de Compuertas Programables en Campo) y CPLD (Dispositivos Lógicos Programables Complejos), con la ventaja de que son circuitos reprogramables en los cuales una función ya programada puede ser modificada o borrada sin alterar el funcionamiento del circuito.

CATEGORÍA	CARACTERÍSTICAS
Diseño totalmente a la media (full-Custom)	• Total libertad de diseño pero el desarrollo requiere de todas las etapas del proceso de fabricación: preparación de la oblea o base, el crecimiento epitaxial, la difusión de impurezas, la implantación de iones, la oxidación, la fotolitografía, la metalización y la limpieza química. • Los riesgos y los costos son muy elevados, sólo justificables para grandes volúmenes o para proyectos con restricciones (área, velocidad, consumo, etc.)
Matrices de puertas predifundidas (semi-custom/gate arrays)	• Existe una estructura regular de dispositivos básicos (transistores) prefabricada que se puede personalizar mediante un conexionado específico que sólo requiere de las últimas etapas del proceso tecnológico. • El diseño está limitado a las posibilidades de la estructura prefabricada y se realiza en base a una biblioteca de celdas precaracterizadas para cada familia de dispositivos.
Celdas estándar precaracterizadas (semi-custom/standar cells)	• No se trabaja con ninguna estructura fija prefabricada, pero sí con bibliotecas de celdas y módulos precaracterizados y específicos para cada tecnología. • Libertad de diseño (en función de las facilidades de la biblioteca), pero el desarrollo exige un proceso de fabricación completo.
Lógica Programable (FPGA, CPLD).	• Se trata de dispositivos totalmente fabricados y verificados que se pueden personalizar desde el exterior mediante diversas técnicas de programación. • El diseño se basa en bibliotecas y mecanismos específicos de mapeado de funciones, mientras que su implementación tan sólo requiere de una fase de programación del dispositivo que habitualmente realiza el propio diseñador en unos pocos segundos.

Tabla 1. Tecnologías de fabricación de circuitos integrados

En la actualidad existe una gran variedad de dispositivos lógicos programables, los cuales son utilizados para reemplazar circuitos SSI, MSI e incluso circuitos VLSI, ya que ahorran espacio y reducen de manera significativa el número y el costo de los diseños. Estos dispositivos llamados PLDs (tabla 2), se clasifican de acuerdo a su arquitectura, es decir, la forma funcional en que se encuentran ordenados los elementos internos que proporcionan al dispositivo sus características específicas.

DISPOSITIVO	DESCRIPCIÓN
PROM	Programmable Read-Only Memory, Memoria Programable de Sólo Lectura
PLA	Programmable Logic Array, Arreglo Lógico Programable
PAL	Programmable Array Logic, Arreglo Lógico Programable
GAL	Generic Logic Array, Arreglo Lógico Genérico
CPLD	Complex PLD, Dispositivo Lógico Programable Complejo
FPGA	Field Program Gate Array, Arreglos de Compuertas Programables en Campo

Tabla 2. Dispositivos Lógicos Programables

1.1.1. Estructura Interna de un PLD

Los dispositivos PROM, PAL, PLA y GAL están formados por arreglos o matrices que pueden ser fijas o programables, mientras que los CPLDs y FPGAs se encuentran estructurados mediante bloques lógicos configurables y celdas lógicas de alta densidad, respectivamente.

La arquitectura básica de un PLD se forma por un arreglo de compuertas AND y OR conectadas a las entradas y salidas del dispositivo. La finalidad de cada una de ellas se describe a continuación.

Arreglo de compuertas AND. Está formado por compuertas AND conectadas a un arreglo programable que contiene fusibles en cada punto de intersección, figura 1.1.a. La programación del arreglo consiste básicamente en fundir los fusibles para eliminar las variables seleccionadas, tal y como se ilustra en la figura 1.1 (b). Obsérvese como en cada una de las entradas a las compuertas AND, se queda intacto el fusible que conecta la variable seleccionada con la entrada a la compuerta. Una vez que los fusibles han sido fundidos no pueden volver a programarse .

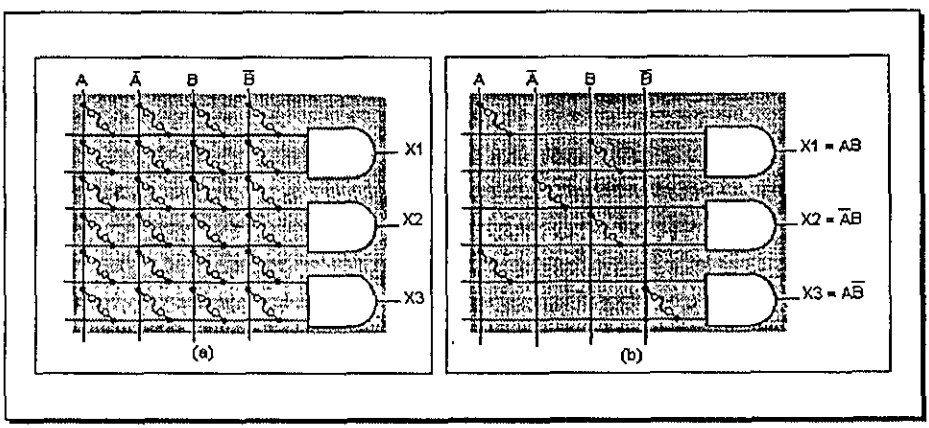


Figura 1.1. (a) Arreglo AND no programado
(b) Arreglo AND programado

Arreglo OR. Está formado por un conjunto de compuertas OR conectadas a un arreglo programable, el cual contiene un fusible en cada punto de intersección, este tipo de arreglo es muy similar al de compuertas AND, ya que de igual manera se programa fundiendo los fusibles para eliminar las variables de la función de salida. En la figura 1.2 se observa el arreglo OR programado y sin programar.

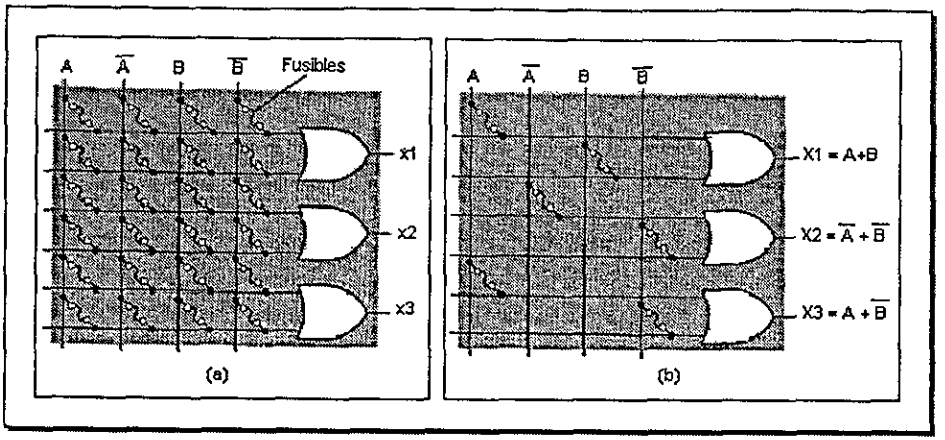


Figura 1.2 (a) Arreglo OR no programado
(b) Arreglo OR programado

En base a lo anterior, observemos en la tabla 3 la estructura de los dispositivos lógicos programables básicos:

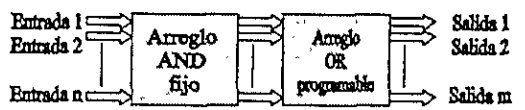
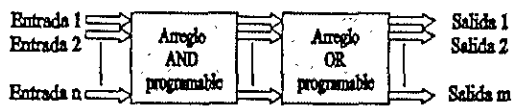
Dispositivo	Esquema básico
La Memoria Programable de Sólo Lectura (PROM), está formada por un arreglo no programable de compuertas AND conectadas como decodificador y un arreglo programable OR	
El Arreglo Lógico Programable (PLA), es un PLD formado por un arreglo AND y un arreglo OR programables.	
El Arreglo Lógico Programable (PAL), este dispositivo está formado por dos arreglos, el arreglo AND programable y un arreglo OR fijo con lógica de salida.	

Tabla 3. Estructura básica de PLDs

- La PROM no es utilizada como un dispositivo lógico, sino como una memoria direccionable, debido a las limitaciones que presenta con las compuertas AND fijas.

- Básicamente el PLA fue desarrollado para superar las limitaciones de la memoria PROM. Este dispositivo fue llamado también FPLA (Arreglo Lógico Programable en Campo) ya que es el usuario y no el fabricante quien lo programa.

El PAL fue desarrollado para superar algunas limitaciones presentadas por el PLA, tales como retardos provocados por la implementación de fusibles adicionales que resultan de la utilización de dos arreglos programables y de la complejidad del circuito. Un ejemplo típico de estos dispositivos es la familia PAL16R8, la cual fue desarrollada por la compañía AMD (Advanced Micro Devices) e incluye los dispositivos PAL16R4, PAL16R6, PAL16L8, PAL16R8, dispositivos programables por el usuario para reemplazar circuitos combinacionales y secuenciales SSI y MSI por un solo circuito.

1.1.2. Arreglo Lógico Genérico GAL

El Arreglo Lógico Genérico (GAL), es similar al PAL, ya que se forma con arreglos AND programable y OR fijo, con una salida lógica programable. Las dos principales diferencias entre los dispositivos GAL y PAL radica en que la GAL es reprogramable y contiene configuraciones de salida programables. Los dispositivos GAL pueden ser programados una y otra vez, ya que usan la tecnología E^2 CMOS (Ellectrically Erasable CMOS, CMOS Borrable Eléctricamente), en lugar de tecnología bipolar y fusibles, figura 1.3.

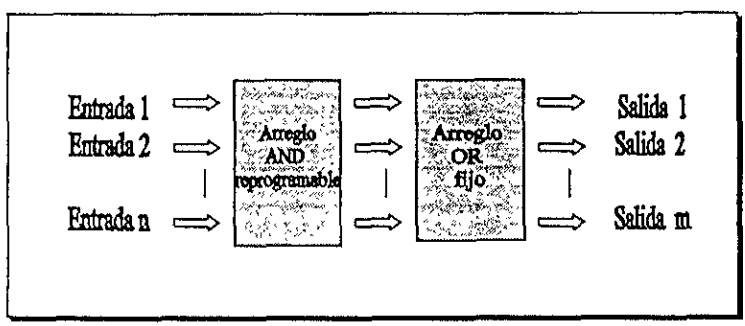


Figura 1.3. Diagrama a bloques del arreglo GAL

1.1.2.1. Programación de un arreglo GAL.

A diferencia de un PAL, el GAL está formado por celdas programables, las cuales pueden ser reprogramadas las veces que sea necesario. Como se observa en el diagrama 1.4, cada fila se conecta a una entrada de la compuerta AND y cada columna a una variable de entrada y sus complementos. Cuando una celda es programada, esta se activa aplicándose cualquier combinación de las variables de entrada o sus complementos, a la compuerta

AND, permitiendo con esto la implementación de cualquier función (producto de términos) requerida.

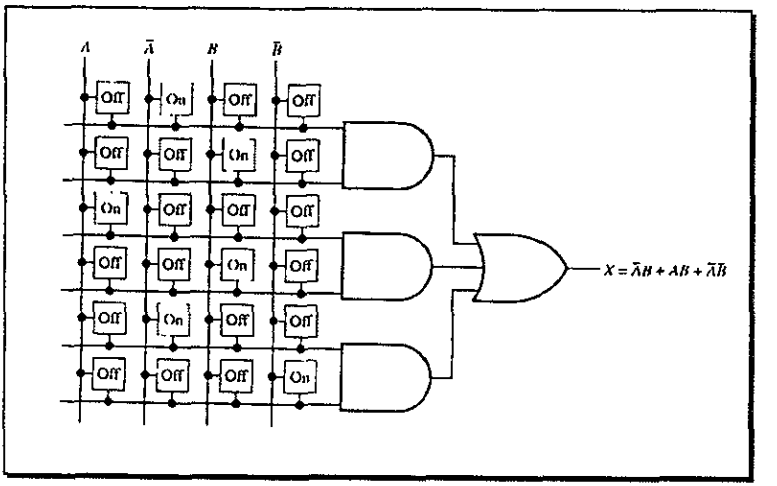


Figura 1.4. Programación del dispositivo GAL

1.1.2.2. Arquitectura de un dispositivo GAL

Con el fin de esquematizar una arquitectura GAL, se toma como ejemplo el dispositivo GAL22V10, figura 1.5a. Este circuito cuenta con 22 líneas de entrada y sus complementos, sumando un total de 44 líneas de entrada a cada una de las compuertas AND (estas entradas se encuentran representadas por las líneas verticales en el diagrama). La intersección que forman las líneas de entrada con los términos producto (líneas horizontales), representan a cada una de las celdas que pueden ser programadas para conectar una variable de entrada (o su complemento) a una línea de término producto, figura 1.5b, donde es observable la forma de obtener la suma de productos.

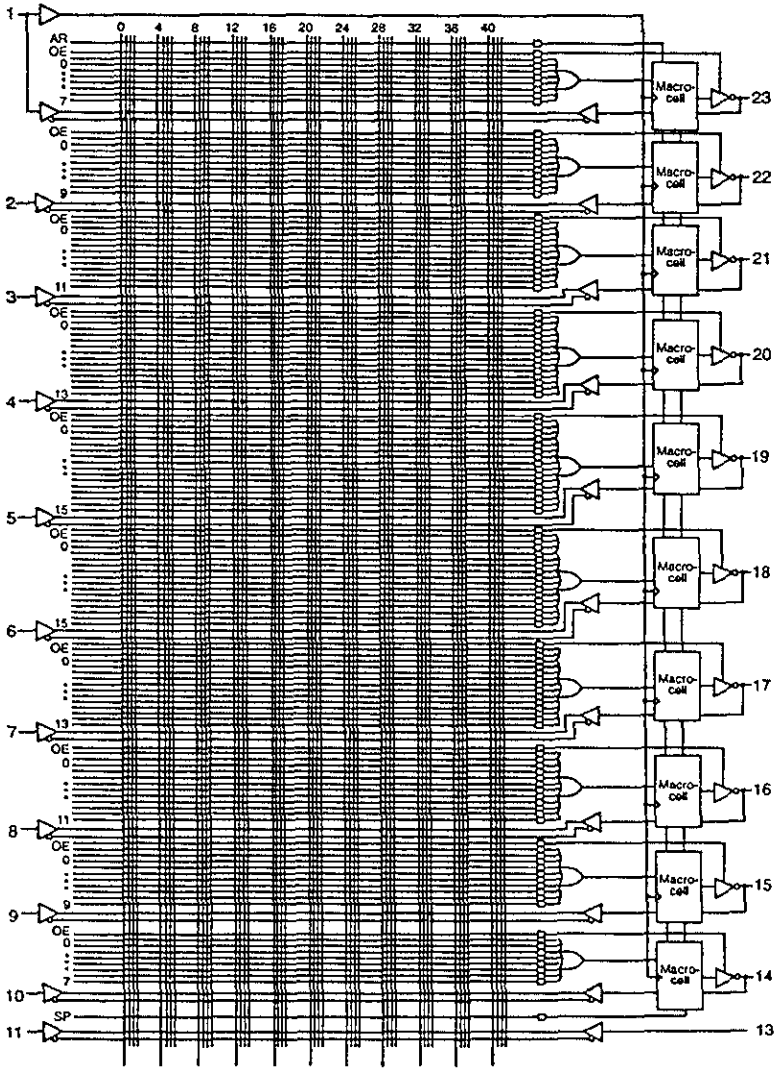


Figura 1.5a. Arquitectura del GAL22V10

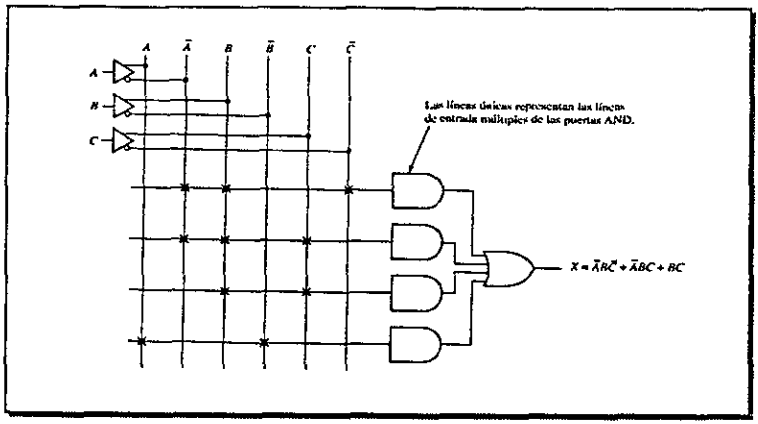


Figura 1. 5b. Implementación de una suma de productos dentro de un GAL

Macroceldas lógicas de salida (OLMCs). Una macrocelda lógica de salida OLMC *(output logic macrocell)* está formada por circuitos lógicos que se pueden programar como *lógica combinacional* o como *lógica secuencial.*. Las configuraciones combinacionales son implementadas por medio de programación, mientras que en las secuenciales la salida resulta de un flip-flop. En la figura 1.6, se observa la arquitectura de una macrocelda del dispositivo GAL22V10, la cual de manera general está formada por un flip-flop y dos multiplexores.

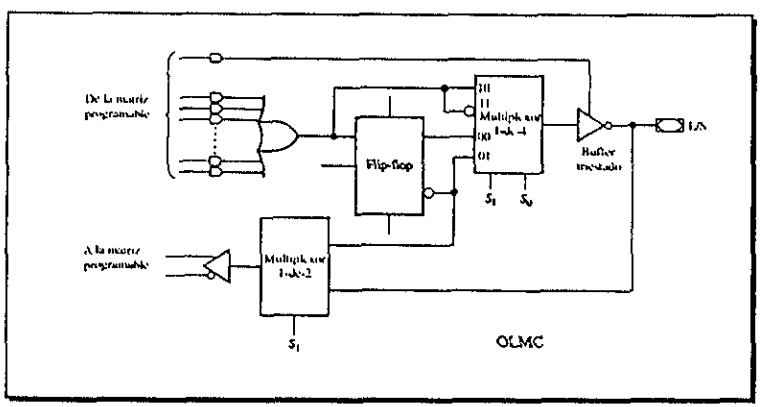


Figura 1.6. Arquitectura de una macrocelda OLMC 22V10

Las entradas de las compuertas AND a la compuerta OR, varían desde ocho hasta dieciséis, esto indica las operaciones producto que pueden efectuarse en de cada macrocelda. El área sombreada está formada por dos multiplexores y un flip-flop; el multiplexor 1 de 4 conecta una de sus cuatro líneas de entrada al buffer triestado de salida, en función de las líneas de

selección S0 y S1. Por otro lado, el multiplexor de 1 de 2 conecta por medio del buffer, la salida del flip-flop o la salida del buffer triestado al arreglo AND; esto se determina por medio de S1. Cada una de las líneas de selección es programada mediante un grupo de celdas especiales que se encuentran dentro del arreglo AND.

1.3. Dispositivos Lógicos Programables de Alto Nivel de Integración

Los PLDs de alto nivel de integración fueron creados con el objeto de integrar una mayor cantidad de dispositivos dentro de un solo circuito (sistema en un chip SOC), caracterizándose por la reducción de espacio y costo, aunado a la mejora sustancial en la habilidad para diseñar sistemas complejos, incrementándose la velocidad y las frecuencias de operación. Además, brinda a los diseñadores la oportunidad de enviar productos al mercado de una forma más rápida, adicionando la facilidad de hacer cambios en el diseño sin afectar la lógica, agregando periféricos de entrada/salida sin consumir una gran cantidad de tiempo, dado que los circuitos son reprogramables en el campo de trabajo

1.3.1. Dispositivos Lógicos Programables Complejos CPLDs.

Un circuito CPLD consiste en un arreglo de múltiples PLDs agrupados como bloques dentro de un solo chip. En algunas ocasiones estos dispositivos son nombrados también como EPLD (Enhanced PLD, PLD mejorado), Super PAL, Mega PAL, etc. Estos dispositivos son considerados de alto nivel de integración, ya que tienen una gran capacidad equivalente a aproximadamente 50 PLDs sencillos.

En su estructura básica cada CPLD contiene múltiples bloques lógicos (similares al GAL22V10), conectados por medio de señales canalizadas desde la **Interconexión Programable (PI).** Esta unidad (PI) se encarga de interconectar a los bloques lógicos, los bloques de entrada y salida del dispositivo, sobre las redes apropiadas, figura 1.7.

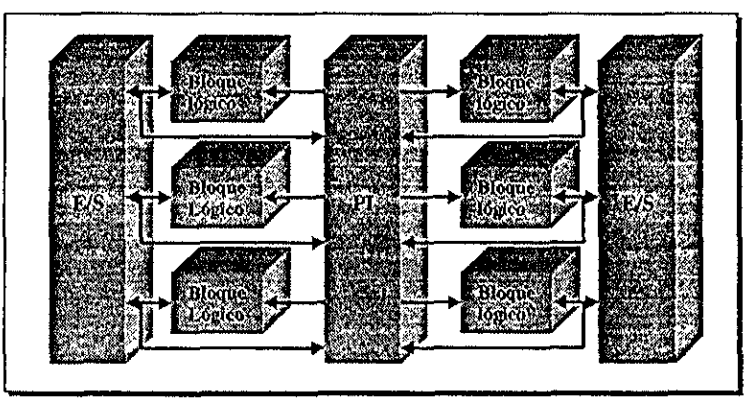


Figura 1.7. Arquitectura básica de un CPLD

Los bloques lógicos, también conocidos como celdas generadoras de funciones, están formados por un *arreglo de productos de términos* que se encarga de implementar los productos efectuados en las compuertas AND; un *esquema de distribución de términos* que permite crear las sumas de los productos provenientes del arreglo AND y por macroceldas (similares a las incorporadas en la GAL22V10), figura 1.8. En ocasiones las celdas de entrada/salida son consideradas parte del bloque lógico, aunque la mayoría de los fabricantes coinciden en que son externas a él. Cabe mencionar que el tamaño de los bloques lógicos es importante, ya que de este depende la capacidad del dispositivo, debido a que determina cuanta lógica puede ser implementada dentro del CPLD.

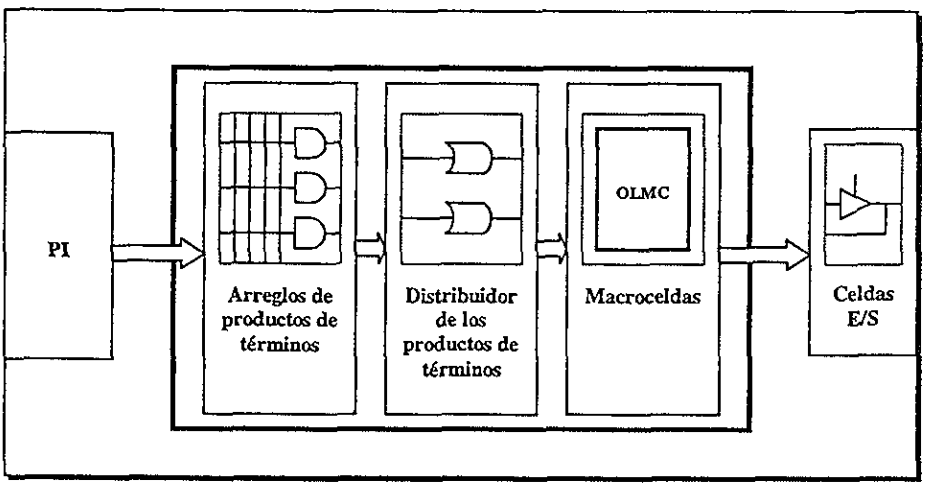


Figura 1.8. Bloque lógico programable

Arreglos de Productos de Términos. Es la parte del CPLD que identifica el porcentaje de número de términos implementados por cada macrocelda, y el número máximo de productos de términos por bloque lógico. Esto es similar al arreglo de compuertas AND programable de un dispositivo GAL22V10.

Esquema de Distribución de Términos. Es el mecanismo utilizado para distribuir los productos de términos a las macroceldas; esto es realizado mediante el arreglo programable de compuertas OR de un PLD. Los diferentes fabricantes de CPLDs tienen implementada la distribución de productos de términos con diferentes esquemas. Mientras dispositivos como la GAL22V10 usan un esquema de distribución variable (los cuales pueden variar en 8,10,12,14 o 16 productos por macrocelda), los CPLDs como la familia MAX de Altera Corporation y Cypress Semiconductor, distribuyen cuatro productos de términos por macrocelda, además de utilizar "productos de términos expandidos" para ser asignados de forma adicional a una o varias macroceldas.

Macroceldas. Una macrocelda de un CPLD está configurada internamente por flip-flops y un control de polaridad que habilita cada afirmación o negación de una expresión. Los CPLDs suelen tener macroceldas de entrada/salida, macroceldas de entrada y macroceldas ocultas, mientras que los PLDs tienen únicamente macroceldas de entrada/salida.

El número de macroceldas que contiene un CPLD es importante, debido a que cada uno de los bloques lógicos que conforman el dispositivo, son típicamente expresados en términos del número de macroceldas que contiene. Generalmente, cada bloque lógico puede variar en número de macroceldas (desde 4 hasta 60), permitiendo con esto la implementación de funciones mas complejas en dispositivos con mayor número de ellas.

1.3.2 Arreglos de Compuertas Programables en Campo (FPGA)

Los dispositivos FPGA están basados en lo que se conoce como **arreglos de compuertas**, los cuales consisten en una parte de la arquitectura que contiene tres elementos configurables: bloques lógicos configurables (CLBs), Bloques de entrada y de salida (IOBs) y canales de comunicación. A diferencia de los CPLDs, en los FPGAs se establece su densidad en cantidades equivalentes a cierto número de compuertas.

Internamente un FPGA está formado por arreglos de bloques lógicos configurables (CLBs) las cuales se comunican entre sí y con las terminales de entrada/salida (E/S), por medio de alambrados llamados *canales de comunicación*. Cada FPGA contiene una matríz de bloques lógicos idénticos, usualmente de forma cuadrada. conectados por medio de líneas metálicas que corren vertical y horizontalmente entre cada bloque. Figura 1.9.

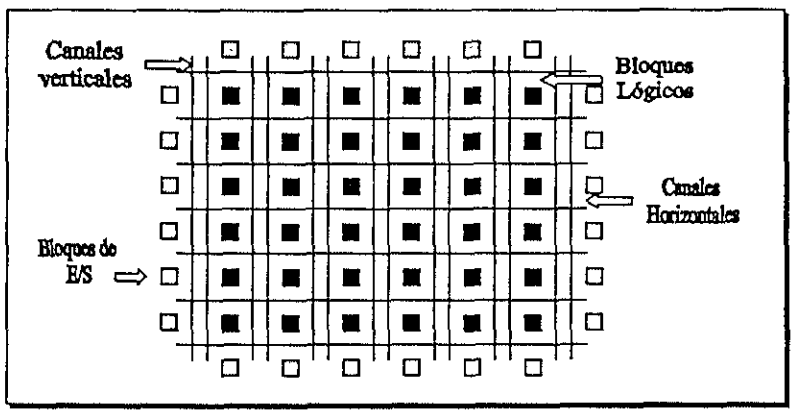


Figura 1.9. Arquitectura básica de un FPGA

En la figura 1.10 se puede observar una arquitectura FPGA de la familia XC4000 de la compañía Xilinx. Este circuito muestra a detalle la configuración interna de cada uno de los componentes principales que conforman el dispositivo.

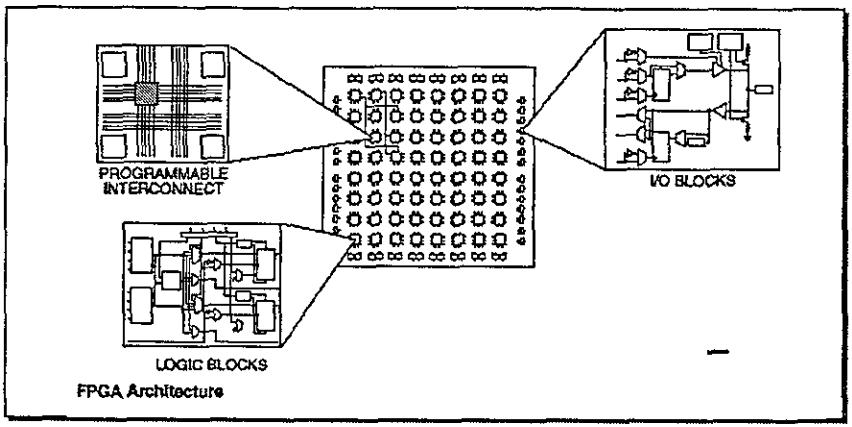


Figura 1.10. Arquitectura del FPGA XC4000 de Xilinx

Los bloques lógicos (llamados también celdas generadoras de funciones), están configurados para procesar cualquier aplicación lógica. Estos bloques presentan la característica de ser funcionalmente completos, es decir permiten la implementación de cualquier función booleana representada en la forma de suma de productos. La forma de implementar el diseño lógico es mediante bloques conocidos como *generadores de funciones o LUTs* (Look Up Table, Tabla de búsqueda), los cuales permiten almacenar la lógica requerida, ya que cuentan con una pequeña memoria interna generalmente de 16 bits.

Cuando se aplica alguna combinación en las entradas de la LUT, el circuito la traduce en una dirección de memoria y envía fuera del bloque el dato almacenado en esa dirección. En la figura 1.11, se observan los tres LUTs que contiene esta arquitectura, los cuales se encuentran etiquetados con las letras G,F y H.

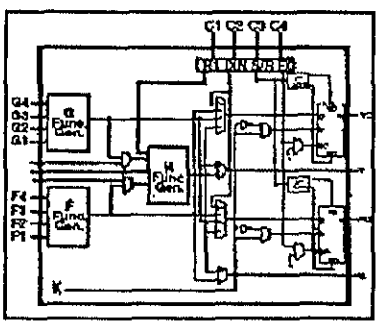


Figura 1.11. Arquitectura de un bloque lógico configurable FPGA

Dentro de un dispositivo FPGA, los CLBs están ordenados en arreglos de matrices programables (Programmable Switch Matrix PSM), la cual se encarga de dirigir las salidas de un bloque a otro. Las terminales de entrada y salida del FPGA pueden estar conectadas directamente al PSM o CLBs, o pueden ser conectadas por medio de vías o canales de comunicación.

En algunas ocasiones se pueden llegar a confundir los conceptos de FPGA y CPLD ya que ambos utilizan bloques lógicos en su fabricación. La diferencia radica en el número de flip-flops, ya que la arquitectura FPGA es rica en registros. En la tabla 4 se presentan algunas otras diferencias entre una y otra arquitectura.

CARACTERISTICAS	CPLD	FPGA
Arquitectura	▪ Similar a un PLD ▪ Más combinacional	▪ Similar a los arreglos de compuertas ▪ Más registros + RAM
Densidad	▪ Baja a media	▪ Media a alta
Funcionalidad	▪ Trabajan a frecuencias superiores de 200Mhz	▪ Depende de la aplicación (arriba de los 135Mhz)
Aplicaciones	▪ Contadores rápidos ▪ Máquinas de estado ▪ Lógica combinacional	▪ Excelentes en aplicaciones para arquitecturas de computadoras ▪ Procesadores Digitales de Señales (DSP) ▪ Diseños con registros

Tabla 4. Diferencias entre CPLDs y FPGAs

1.4. Ambiente de desarrollo de la Lógica Programable

Una de las grandes ventajas al diseñar sistemas digitales mediante dispositivos lógicos programables radica en el bajo costo de los recursos requeridos para el desarrollo de estas aplicaciones. De manera generalizada el soporte básico se encuentra formado por una computadora personal, un grabador de Dispositivos Lógicos Programables y el software de aplicación que soporta a las diferentes familias de circuitos integrados PLDs. Figura 1.12

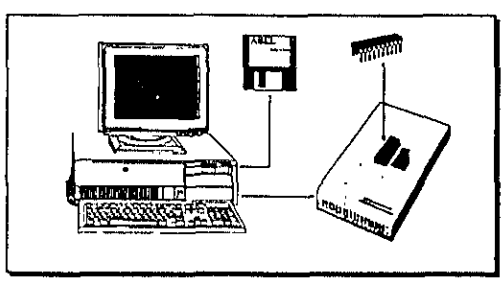


Figura 1.12 Herramientas necesarias en la programación de PLDs

Actualmente existen diversos programas para computadora denominados CAD (Diseño Asistido por Computadora), que facilitan el diseño de sistemas digitales. Algunos de estos programas se muestran en la tabla 5.

COMPILADOR LÓGICO	CARACTERÍSTICAS
PALASM (PAL Assembler, Ensamblador de PAL)	Creado por la compañía Advanced Micro Devices (AMD) Desarrollado únicamente para aplicaciones con dispositivos PAL Acepta el formato de ecuaciones booleanas Utiliza cualquier editor que grabe en formato ASCII
OPAL (Optimal PAL lenguage, Lenguaje de Optimización para Arreglos Programables)	Desarrollado por National Semiconductors Se aplica en dispositivos PAL y GAL Formato para usar lenguaje de máquinas de estado, ecuaciones booleanas de distintos niveles, tablas de verdad, o cualquier combinación entre ellas. Disponible en versión estudiantil y profesional (OPAL Jr y OPAL Pro) Genera ecuaciones de diseño partiendo de una tabla de verdad
PLPL (Programable Logic Programming Language, Lenguaje de Programación de Lógica Programable)	Creado por AMD Introduce el concepto de jerarquías en sus diseños Formatos múltiples (ecuaciones boolenas, tablas de verdad, diagramas de estado y las combinaciones entre estos) Aplicaciones en dispositivos PAL y GAL
ABEL (Advanced Boolean Expression Language, Lenguaje Avanzado de Expresiones Booleanas)	Creado por Data I/O Corporation Programa cualquier tipo de PLD (Versión 5.0) Proporciona tres diferentes formatos de entrada: ecuaciones booleanas, tablas de verdad y diagramas de estados. Es catalogado como un lenguaje avanzado HDL (Lenguaje de Descripción en Hardware)
CUPL (Compiler Universal Programmable Logic, Compilador Universal de Lógica Programable)	Creado por AMD para desarrollo de diseños complejos Presenta una total independencia del dispositivo Programa cualquier tipo de PLD Facilita la generación de descripciones lógicas de alto nivel Al igual que ABEL, también es catalogado como HDL

Tabla 5. Descripción de Compiladores Lógicos para PLDs

Estos programas conocidos también como **compiladores lógicos**, tienen una función en común: procesar y sintetizar el diseño lógico para ser introducido mediante un método específico (ecuaciones booleanas, diagramas de estado, tablas de verdad), dentro de un dispositivo lógico programable específico.

1.4.1. Método tradicional de diseño con lógica programable

La manera tradicional de diseño con lógica programable, parte de la representación esquemática del circuito que se requiere implementar, para de forma posterior definir por el método adecuado (ecuaciones booleanas, tablas de verdad, diagramas de estado) la solución del sistema. Por ejemplo, en la figura 1.13 se observa un diagrama que representa a un circuito implementado con compuertas lógicas AND y OR. En este caso se eligió el método de ecuaciones booleanas para representar su funcionamiento, aunque pudo haberse utilizado también una tabla de verdad.

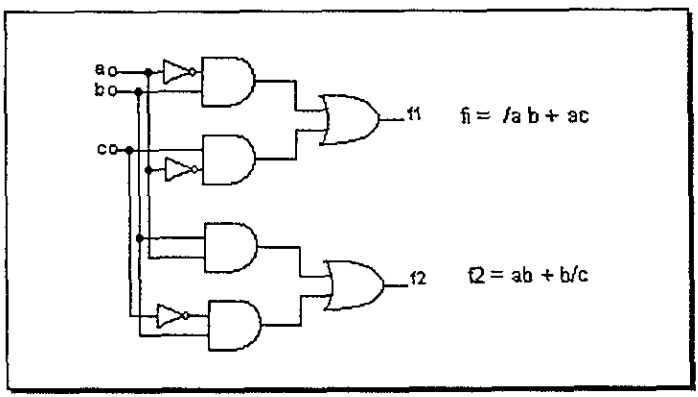


Figura 1.13. Obtención de las ecuaciones booleanas

Como se puede observar, las ecuaciones que rigen el comportamiento del sistema se encuentran derivadas en función de las salidas f1 y f2 del circuito. Una vez que se obtienen estas ecuaciones, el siguiente paso consiste en introducir a la computadora el archivo fuente o de entrada, es decir el programa que contiene los datos que le permitirán al compilador sintetizar la lógica requerida. Típicamente se introduce alguna información preliminar que indique datos tales como el nombre del diseñador, la empresa, fecha, nombre del diseño, etc. Posteriormente se especifica el tipo de dispositivo PLD que se va a utilizar, la numeración de los pines de entrada y salida, así como las variables que serán utilizadas en el diseño. Por último, se define la función lógica en forma de ecuaciones booleanas o cualquier formato que sea aceptado por el compilador.

En la figura 1.14 se observa la pantalla principal del software **PALASM**, en el cual se compilará, con el fin de ejemplificar la metodología a seguir, el diseño de la figura 1.13.

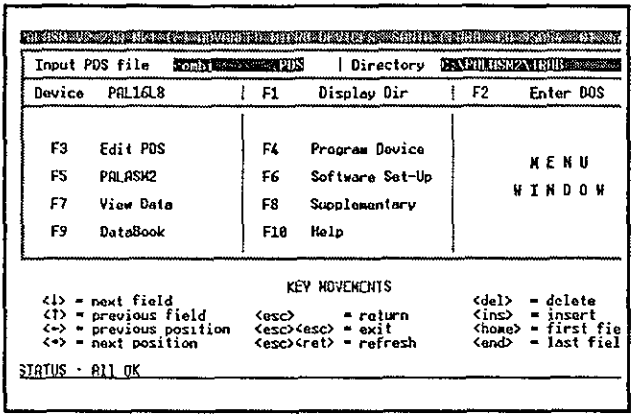


Figura 1.14. Pantalla principal de PALASM

La forma de introducir el diseño se muestra en el listado 1. Nótese que las palabras reservadas por el compilador se encuentran representadas por letras negritas.

```
TITLE          EJEMPLO
PATTERN        EJEMPLO.PDS
REVISION       1.0
AUTHOR         JESSICA
COMPANY        UNAM
DATE           00-00-00
CHIP           XX PAL16L8

; 1  2  3  4  5  6  7  8  9  10
 NC NC NC NC  A  B  NC NC NC GND

:11  12  13  14  15  16  17  18  19  20
 NC   C  NC  NC  NC  NC  NC  NC  NC  VCC

EQUATIONS

F1 = /a*b + /a*c
F2 = a*b + b + /c

SIMULATION

TRACE_ON A B

SETF /A /B
CHECK C
SETF /A B
CHECK /C
SETF A /B
CHECK C

TRACE OFF
```

ENCABEZADO

DECLARACION DE PINES DE ENTRADA/SALIDA

ECUACIONES DEL CIRCUITO

SIMULACIÓN (CONDICIONES E/S)

Listado 1. Archivo Fuente compilado en formato PALASM

El siguiente paso consiste en la compilación del diseño, el cual radica básicamente en localizar los errores de sintaxis[1] o de otro tipo, cometidos durante la introducción de los datos en el archivo fuente. El compilador procesa y traduce el archivo fuente y minimiza las ecuaciones. En este paso, el diseño es simulado utilizando un conjunto de entradas y sus correspondientes valores de salida conocidos como *vectores de prueba*. Es durante este proceso cuando se "verifica" que el diseño funcione de manera correcta antes de ser introducido al PLD. Si se detecta algún error en la simulación entonces el diseño debe ser depurado para corregir este defecto.

Una vez que el diseño no tiene errores, el compilador genera un archivo conocido como **JEDEC** (Joint Electronic Device Engineering Council)[2] o mapa de fusibles. Este archivo se encarga de indicar al grabador cuales fusibles serán fundidos y cuales se activan, para que de forma posterior se grabe el PLD de acuerdo al mapa de fusibles. Figura 1.15

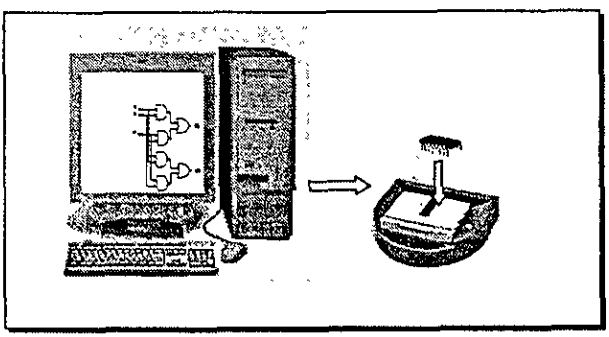


Figura 1.15. Implementación final del diseño en un PLD

Como se puede observar, ciertos PLDs (PROM, PAL, GAL) se programan empleando un grabador de dispositivos lógicos. Algunos otros PLDs, como los CPLDs y FPGAs, presentan la característica de ser programables dentro del sistema (ISP, In-System Programmable), es decir no tienen que ser introducidos dentro del grabador, ya que por medio de elementos auxiliares pueden ser programados dentro de la tarjeta de circuito integrado.

Como se observa el método tradicional de diseño con lógica programable simplifica de manera considerable el tiempo de diseño y permite al diseñador un mayor control de los errores que se pudieran presentar, ya que la corrección se realiza en el software y no en el diseño físico.

[1] La sintaxis se refiere al formato establecido y la simbología utilizada para describir una categoría de funciones.
[2] Los archivos JEDEC están estandarizados para todos los compiladores lógicos existentes

1.5 Campos de aplicación de la lógica programable

La lógica programable es una herramienta de diseño muy poderosa, aplicada a nivel industrial y en proyectos efectuados en diversas universidades del mundo. Actualmente son utilizados desde los PLDs mas sencillos (como el GAL, PAL, PLA.) que se usan en aplicaciones para reemplazar circuitos LSI y MSI, hasta los potentes CPLDs y FPGAs que tienen aplicaciones en áreas como telecomunicaciones, computación, redes, medicina, procesamiento digital de señales, Multiprocesamiento de datos, Microondas digitales de radio, Telefonía celular, Filtros digitales programables, entre otros.

De manera general, los CPLDs son recomendables en aplicaciones donde se requieran muchos ciclos de sumas de productos, ya que pueden introducirse en el dispositivo para ejecutarse a un mismo tiempo, lo que conduce a pocos retrasos. En la actualidad, los CPLDs son muy utilizados a nivel industrial, debido a que se pueden convertir fácilmente diseños compuestos por múltiples PLDs sencillos a un solo circuito CPLD.

Por otro lado, los FPGAs son recomendables en aplicaciones secuenciales que no combinen grandes cantidades de términos producto. Por ejemplo los FPGA desarrollados por la compañía ATMEL son utilizados para proveer una alta velocidad en cómputo intensivo, aplicaciones en procesadores digitales de señales (DSPs) y en otras fases del diseño lógico, debido al número tan grande de registros con los cuentan sus dispositivos (de 1024 a 6400), esto los hace ideales para su uso en dichas áreas.

1.5.1. Desarrollos recientes

Existen desarrollos realizados por diversas compañías donde su funcionamiento se basa en un PLD, por ejemplo en la figura 1-16, se observa una tarjeta basada en un FPGA de la familia XC4000 de Xilinx Corporation. Este desarrollo permite el procesamiento de datos en paralelo a alta velocidad, reduciendo con esto los problemas de procesamiento de datos intensivo[3].

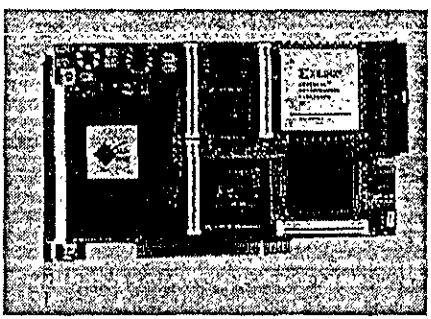


Figura 1.16. Sistema basado en un FPGA

[3] fuente de información: http://www.annapmicro.com

En la figura 1.17 observamos otra aplicación implementada en un dispositivo CPLD de la Familia Flex10K de Altera Corporation (nivel de integración de 7000 compuertas). La función de esta tarjeta es permitir diversas aplicaciones en tiempo real, como por ejemplo el procesamiento digital de señales[4].

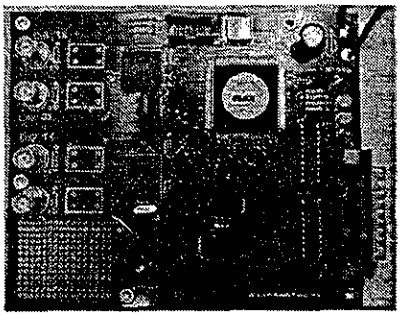


Figura 1.17. Ejemplo de un diseño lógico programable completo.

Como se mencionó anteriormente, el campo de la lógica programable se ha extendido en la industria en los últimos años, ya que compañías de nivel internacional integran o desarrollan lógica programable en sus diseños. Tabla 6.

COMPAÑÍA	PRODUCTOS DESARROLLADOS CON LOGICA PROGRAMABLE
Andraka Consulting Group http://users.ids.net/~randraka/Inc	Procesadores digitales de señales (DSP) Comunicaciones digitales Procesadores de audio y video
Code Logic http://home.intekom.com/codelogic/	Lógica configurable Control embebido
Boton Line http://www.bltinc.com/	Módems de alta velocidad Audio, video, adquisición de datos y procesamiento de señales en general Aplicaciones militares: Criptografía, seguridad en comunicaciones, proyectos espaciales.
Comit's Services http://www.comit.com/	Redes: aplicaciones en protocolos TCP/IP Multimedia : Compresión de Audio/Video Aplicaciones en tiempo real
New Horizonts GU http://www.netcomuk.co.uk/~newhoriz/index.html	Digitalizadores, Cámaras de Video (120Mbytes/sec) Video en tiempo real Puertos paralelos de comunicaciones para PC
Design Service Segments http://www.smartech.fi/	Diseño de microprocesadores complejos Dispositivos para telecomunicaciones, DSP Aplicaciones en diseños para control industrial

Tabla 6. Compañías que incorporan lógica programable en sus diseños

[4] fuente de información. http://www fgpa com

1.6. La lógica programable y los Lenguajes de Descripción en Hardware (HDLs).

Como consecuencia de la creciente necesidad de integrar cada vez mayor número de dispositivos dentro de un circuito integrado, se ve la necesidad de desarrollar nuevas herramientas de diseño que auxiliaran al ingeniero integrador de sistemas en su difícil tarea. Este paradigma hace que en la década de los años 50 los lenguajes de descripción en hardware (HDLs), aparezcan como una opción de diseño para el desarrollo de sistemas electrónicos complejos. Estos lenguajes alcanzan un mayor desarrollo durante los años 70, donde es posible encontrar varios de ellos como IDL de IBM, TI-HDL de Texas Instruments, ZEUS de General Electric, etc., todos ellos a nivel industrial, así como los lenguajes en el ámbito universitario (AHPL, DDL, CDL, ISPS, etc.). 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. Algunos sólo tenían aplicaciones como herramientas de simulación o de síntesis, otros por el contrario estaban enfocados a una tecnología a nivel de diseño o metodología de diseño dados.

Continuando con su desarrollo en la década de los ochenta surgen lenguajes como VHDL, Verilog, ABEL 5.0, AHDL, etc., que son considerados Lenguajes de Descripción en Hardware, debido a que permiten abordar un problema lógico a nivel funcional facilitando con esto la evaluación de soluciones alternativas antes de abordar 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) un diseño específico. Los niveles de abstracción se emplean para poder clasificar un modelo HDL según el grado de detalle y precisión de sus descripciones.

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 implementación final.

- *Transferencia de registros (RT):* se desarrolla una partición del sistema en bloques funcionales sin considerar a detalle la realización final de cada bloque.

- *Lógico o de compuertas*: el circuito se encuentra expresado en términos de ecuaciones lógicas o de compuertas.

1.6.1. El Lenguaje de Descripción en Hardware VHDL

Actualmente el lenguaje de descripción en hardware más utilizado a nivel industrial es VHDL[5] (VHSIC, Hardware Description Language), el cual nace en la década de los 80 como un lenguaje estándar, capaz de soportar el proceso de diseño de sistemas electrónicos complejos, con las características de reducir el tiempo de diseño y los recursos tecnológicos requeridos. VHDL fue creado por el Departamento de Defensa de los Estados Unidos, como parte del programa "Very High Speed Integrated Circuits" (VHSIC), el cual detectó la necesidad de un medio estándar de comunicación y documentación para la gran cantidad de datos asociados al diseño de dispositivos de escala y complejidad deseados, es decir VHSIC debe de conceptualizarse como la rapidez en el diseño de circuitos integrados.

Después de varias versiones revisadas por el gobierno de los Estados Unidos, industrias y universidades, finalmente el IEEE (Instituto de Ingenieros Eléctricos y Electrónicos) publicó en diciembre de 1987 el estándar IEEEstd 1076-1987. Un año más tarde, surge la necesidad de que todos los ASICs creados por el Departamento de Defensa sean descritos en VHDL, por lo que en 1993 el estándar adicional de VHDL IEEE1164 era adoptado.

Hoy en día VHDL es considerado como un estándar para la descripción, modelado y síntesis tanto de circuitos digitales, como de sistemas complejos. Este lenguaje presenta diversas características que lo hacen uno de los HDL's más utilizados en la actualidad:

1.6.2. Ventajas del desarrollo de circuitos integrados con VHDL

- VHDL presenta una notación formal con la intención de ser usada en cualquier diseño electrónico.

- **Disponibilidad pública.** VHDL es un estándar no sometido a ninguna patente o marca registrada, por lo que puede ser utilizado sin restricción alguna por cualquier empresa o institución. Además, al estar mantenido y documentado por IEEE, existe la garantía de estabilidad y soporte.

- **Independencia tecnología de diseño.** VHDL fue diseñado para soportar diversas tecnologías de diseño (PLDs, FPGAs, ASICs, etc.) con distinta funcionalidad (circuitos combinacionales, secuenciales, síncronos y asíncronos), permitiendo de esta forma satisfacer las distintas necesidades de diseño.

- **Independencia de la tecnología y proceso de fabricación.** VHDL ha sido creado para ser independiente de la tecnología y del proceso de fabricación del circuito o del sistema electrónico. El lenguaje puede ser utilizado de igual manera en circuitos MOS,

[5] La letra V que antecede a las siglas HDL en el término VHDL, tiene su origen en el proyecto del Departamento de Defensa de los Estados Unidos VHSIC (Very High Speed Integrated Circuit).
fuente: http://www.vhdl.org

bipolares, BICMOS, etc., sin necesidad de incluir dentro del diseño información específica de la tecnología utilizada o de sus características (retardos, consumos, temperatura, etc.), aunque esto puede hacerse de manera opcional.

- **Capacidad descriptiva en distintos niveles de abstracción.** El proceso de diseño consiste de varios niveles de detalle, desde la especificación hasta la implementación final (niveles de abstracción). VHDL presenta la ventaja de poder diseñar en cualquiera de estos niveles y combinarlos, creando con esto lo que se conoce como simulación **multinivel.**

- **Uso como formato de intercambio de información.** VHDL permite el intercambio de información a lo largo de todas las etapas del proceso de diseño, favoreciendo con esto el trabajo en equipo.

- **Independencia de los proveedores.** Debido a que VHDL es un lenguaje estándar, permite que las descripciones o modelos generados en un sitio sean accesibles desde cualquier otro, independientemente de las herramientas de diseño que sean utilizadas. Este hecho independiza al usuario de los proveedores: por un lado de los distribuidores de componentes comerciales, ya que los modelos generados por VHDL van a ser compatibles; por otro lado, de los centros de diseño, debido a que las especificaciones requeridas van a ser aceptadas por cualquiera de ellos; por último, de los vendedores de herramientas, porque estas siempre van a utilizar VHDL como medio de entrada.

- **Reutilización del código.** El uso de VHDL como un lenguaje estándar permite reutilizar los códigos en diversos diseños, independientemente de que hayan sido generados para una tecnología (CMOS, bipolar, etc.) e implementación (FPGA, ASIC, etc) en particular.

- **Facilitación en participación en proyectos internacionales.** Actualmente VHDL constituye el lenguaje estándar de referencia a nivel internacional. Impulsado originalmente por el Departamento de Defensa de los Estados Unidos, cualquier programa lanzado por alguna de las agencias oficiales de ese país, obliga al uso de VHDL para el modelado de los sistemas y en la documentación del proceso de diseño. Este hecho ha motivado que diversas empresas y universidades adopten a VHDL como su lenguaje de diseño.

  En Europa la situación es similar, ya que actualmente la mayoría de las grandes empresas del ramo han definido a VHDL como el lenguaje de referencia en todas las tareas de diseño, modelado, documentación y mantenimiento de los sistemas electrónicos. De hecho, el número de usuarios de VHDL en Europa es mayor que en Estados Unidos, debido en gran parte a que VHDL resulta el lenguaje más común en la mayoría de los consorcios.

1.6.3. Desventajas del desarrollo de circuitos integrados con VHDL

Como se puede observar, VHDL presenta grandes ventajas, sin embargo, es necesario mencionar también algunas desventajas que muchos diseñadores consideran importantes:

- Debido a que VHDL es un lenguaje diseñado por un comité, presenta una complejidad extensa, ya que se debe dar gusto a las diversas opiniones de los miembros de éste. En base a esto resulta un lenguaje difícil de aprender para un novato.

- En algunas ocasiones el uso de una herramienta provista por alguna compañía en especial, tiene características adicionales al propio lenguaje, donde se pierde un poco la libertad de diseño. Como método alternativo, se pretende que entre diseñadores que utilizan distintas herramientas exista una compatibilidad en sus diseños, sin que esto requiera un esfuerzo importante en la traducción del código.

1.6.4. VHDL en la actualidad

La actividad que se ha generado en torno a VHDL es muy intensa. En muchos países, como España, se han creado grupos de trabajo en torno a VHDL, realizándose periódicamente conferencias, reuniones, etc., donde se presentan trabajos tanto en Estados Unidos (en el VIUF, VHDL International User's Forum) como en Europa (VHDL Forum for CAD in Europe), así como en el congreso EuroVHDL celebrado desde 1992.

La participación europea en el esfuerzo de estandarizar el lenguaje se canaliza a través del proyecto ESPRIT, lidereado por SIEMENS-NIXDORF, en el proyecto participan prácticamente todas las grandes compañías europeas del sector electrónico, como ANACAD, ICL, PHILLIPS, TGI y THOMSON-CSF, además de diversas universidades y centros de investigación. Otras empresas dedicadas a la microelectrónica han ido paulatinamente adaptándose a VHDL; incluso en Japón está teniendo una gran aceptación, a pesar que ellos cuentan con su propio lenguaje estándar llamado UDL/I.

El proceso de estandarización del VHDL no se detuvo con la primera versión del lenguaje (VHDL'87) , por el contrario ha continuado con la nueva versión (VHDL'93) así como en constantes adiciones, mejoras y metodologías de uso. Entre estas adiciones se encuentra una muy importante: *la extensión analógica (1076.1)* que permite la utilización de un único lenguaje en todas las tareas de especificación, simulación y síntesis tanto de sistemas electrónicos digitales como en analógicos o mixtos.

1.7. Compañías de Soporte en Hardware y Software

Existen diversas compañías internacionales que fabrican o distribuyen dispositivos lógicos programables, algunas ofrecen productos con características generales y otras por el contrario introducen innovaciones a sus dispositivos. A continuación se mencionan algunas de las más importantes:

ALTERA CORPORATION

Altera es una de las compañías mas importantes de producción de dispositivos lógicos programables y también es la que mas familias ofrece, ya que actualmente tiene en el mercado ocho familias: APEX™20K, FLEX®10K, FLEX 8000, FLEX 6000, MAX® 9000, MAX7000, MAX5000, y Classic™ . La capacidad de integración en cada familia varía desde 300 hasta 1,000,000 de compuertas utilizables por dispositivo, además de que todas tienen la capacidad de integrar sistemas complejos.

Las características generales de los dispositivos Altera radican en los siguientes aspectos:

- Frecuencia de operación del circuito superior a los 175Mhz y retardos pin a pin de menos de 5ns.

- La implantación de bloques de arreglos embebidos (EABs), los cuales son utilizados para realizar funciones aritméticas como multiplicadores, ALUs, y secuenciadores, de forma mas eficiente que con arreglos lógicos tradicionales. Microprocesadores, microcontroladores y funciones complejas con DSPs (Procesadores Digitales de Señales), son mas fáciles de implementar por medio de arreglos embebidos [12].

- La programación en sistema (ISP), que permite programar los dispositivos montados en la tarjeta. Figura.1.18.

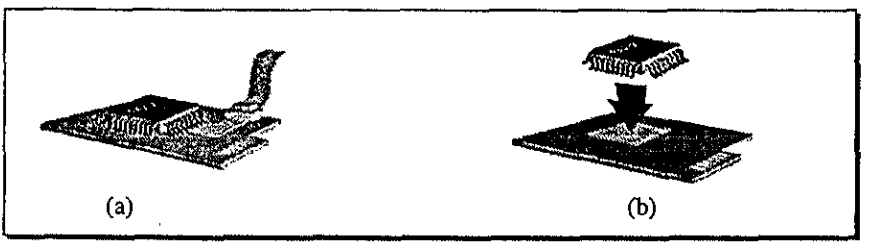


Figura 1.18 (a)Programación en sistema. (b) Programación en montaje
(Cortesía de Altera Corporation)

En la figura 1.18a, observamos la programación en sistema, es decir, el circuito no necesita ser retirado de la tarjeta para programarse. En la figura 1.18b, se muestra lo contrario, donde este tipo de programación es similar a la grabación cotidiana que realizamos, donde el dispositivo tiene que ser colocado y retirado todas las veces que se quiera programar.

- Más de cuarenta tipos y tamaños de encapsulados, incluyendo el novedoso TQFP (thin quad flat pack), el cual es un dispositivo delgado, de forma cuadrangular y plano, que permite ahorrar un espacio considerable en la tarjeta.

- Operación multivoltaje, entre los 5 y 3.3 Volts, para máximo funcionamiento y 2.5V en sistemas híbridos.

- Potentes herramientas de software como MAX+PLUS II, que soporta todas las familias de dispositivos de Altera, así como el software estándar compatible con VHDL.

CYPRESS SEMICONDUCTOR

La compañía Cypress Semiconductor ofrece una amplia variedad de dispositivos lógicos programables complejos (CPLDs), que se encuentran ubicados dentro de las familias Ultra37000™ y FLASH370i™. Cada una de estas familias ofrece la reprogramación en sistema (ISR), la cual permite reprogramar los dispositivos las veces que se quiera dentro de la tarjeta.

Todos los dispositivos de ambas familias trabajan con voltajes de operación de 5 ó de 3.3V y contienen internamente desde 32 hasta 128 macroceldas.

En lo que respecta a software de soporte, Cypress ofrece su poderoso programa *Warp*, el cual está basado en VHDL. Este programa nos permite de manera gráfica simular el circuito programado, generando un archivo de mapa de fusibles (jedec) que puede ser implantado directamente sobre cualquier PLD, CPLD o FPGA de Cypress o de alguna otra compañía que sea compatible con ella.

CLEAR LOGIC

La compañía Clear Logic introdujo en noviembre de 1998, los Dispositivos Lógicos Procesados por Láser (LPDL), tecnología que provee reemplazos con los dispositivos de la Compañía Altera, pero con un menor costo y tamaño. La tecnología LPLD puede cortar arriba de un millón de transistores para construir aproximadamente 512 macroceldas, reemplazando al dispositivo MAX 7512A de Altera, reduciendo en tamaño más del 60% con respecto al chip original. Las primeras familias introducidas por Clear Logic son CL7000 y CL7000E, las cuales tienden a crecer en un futuro.

MOTOROLA

Motorola empresa líder en comunicaciones y sistemas electrónicos ofrece también dispositivos FPGA y FPAA (*Field Programmable Array Analog*, Campos Programables de Arreglos Analógicos). Los FPAA fueron los primeros campos programables para aplicaciones analógicas, utilizados en aplicaciones dentro de las áreas de Transporte, Redes, Computación y telecomunicaciones.

XILINX

Xilinx es una de las compañías líder en soluciones de lógica programable, incluyendo circuitos integrados avanzados, herramientas en software para diseño, funciones predefinidas y soporte de ingeniería. Xilinx fue la compañía que inventó los FPGA y actualmente sus dispositivos ocupan mas de la mitad del mercado mundial de los dispositivos lógicos programables.

Los dispositivos de Xilinx reducen significativamente el tiempo requerido para desarrollar aplicaciones en las áreas de computación, telecomunicaciones, redes, control industrial, instrumentación, aplicaciones militares y para el consumo general.

Las familias de CPLDs XC9500 y XC9500XL, proveen una larga variedad de dispositivos programables con características que van desde los 5 a 3.3 Volts de operación, 36 a 288 macroceldas, 34 a 192 terminales de entrada y salida, así como la programación en sistema. Los dispositivos de las familias XC4000 y XC1700 de FPGAs, manejan voltajes de operación entre los 5 y 3.3V, una capacidad de integración arriba de las 40,000 compuertas y programación en sistema.

En lo que se refiere a software, Xilinx desarrolló una importante herramienta llamada *Foundation Series*, el cual soporta diseños estándares basados en ABEL-HDL y en VHDL. Esta herramienta es ofrecida en versión estudiantil y profesional.

De manera general, existe una amplia y variada gama de dispositivos lógicos programables disponibles actualmente en el mercado. La elección de uno u otro depende directamente de los recursos con los que cuenta el diseñador, así como los requerimientos que exige el diseño. En la tabla 7 se muestra de forma simplificada a algunas de las compañías que actualmente ofrecen soluciones de lógica programable, mientras que en la figura 1.19 se muestran los productos que éstas ofrecen

1.7.1. El futuro de la lógica programable

Debido al auge que tiene la lógica programable en la actualidad, no es difícil suponer que se pretende mejorar a futuro las herramientas existentes, con el fin de extender su campo de aplicación en más áreas. Algunas compañías buscan mejorar la funcionalidad de sus circuitos, así como aumentar la integración en ellos, esto permitiría competir con el mercado de los ASICs, ya que se mejoraría el costo por volumen, el ciclo de diseño y se disminuiría el voltaje de consumo (actualmente de 5 a 3.3 volts) al mínimo.

Otra característica que se pretende mejorar es la reprogramación de los circuitos, debido a que su implementación requiere muchos recursos físicos y tecnológicos. Por esta razón se busca cambiar las metodologías existentes de diseño para incluir sistemas totalmente reprogramables.

Algunos desarrollos cuentan con memoria RAM o microprocesadores integrados en la tarjeta de programación, la tendencia de algunos fabricantes es integrar estos recursos dentro de un solo circuito.

COMPAÑÍA	PRODUCTOS DE HARDWARE	HERRAMIENTAS SOFTWARE
Altera	FPGAs: Familias APEX 20K, FLEX 10K, FLEX 6000, MAX 9000, MAX 7000, MAX 5000 y CLASSIC	MAX + PLUS II: Soporta VHDL, Verilog y entrada esquemática.
Chip Express	LPGA (Laser Program Gate Array): CX3000, CX2000 y QYH500	QuICk Place&route: Diseños en base a Vectores de prueba (CTV , ChipExpress Test Vector) y VHDL
Clear Logic	LPLDs (Laser-processed Logic Decice): CL7000, CL7000E, CL7000S	Desarrollos basados con herramientas de Altera.
Cypress Semiconductors	PLDs : GAL22V10 CPLDs: Ultra37000, FLASH370i	WARP: Soporta VHDL, Verilog y esquemáticos.
Motorola	FPAA (Field Programmable Analog Array): MPAA020	Easy Analog: herramienta de diseño interactiva exclusiva para diseño con FPAAa.
Vantis	FPGAs: Familias MACH4 y MACH5A	MACHXL: VHDL y Verilog
Quick Logic	PAsic (Asic Programable) y la Familia QL de FPGAs.	Quick Works: herramienta de soporte para VHDL, Verilog y captura esquemática.
Xilinx	CPLDs: Familia XC9500 y XC9500XL FPGAs: Familia XC400 y XC1700	Xilinx Foundation Series: soporta ABEL-HDL, VHDL y esquemáticos.

Tabla 7. Compañías de Soporte de lógica programable

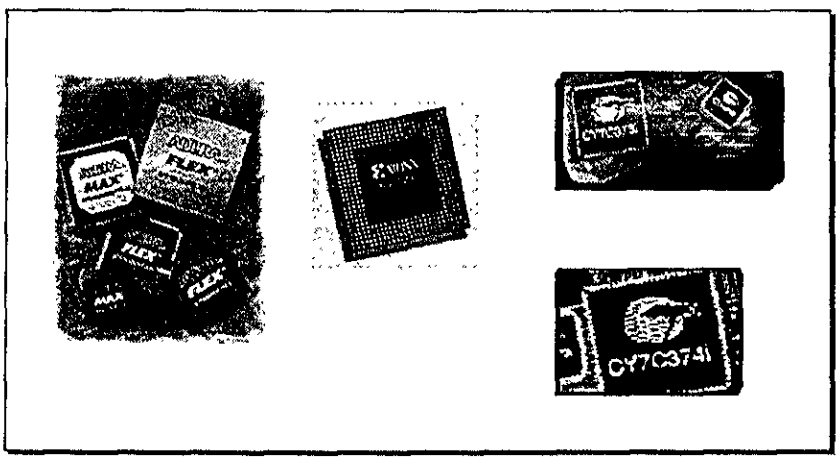


Figura 1.19. Dispositivos lógicos programables
(cortesía de Altera Corp., Xilinx y Cypress)

Capítulo 2

VHDL
Su Organización y Arquitectura

VHDL (Hardware Description Language) tal como lo indican las siglas de su nombre, es un lenguaje orientado a la descripción o modelado de sistemas digitales, es decir, se trata de un lenguaje mediante el cual se puede describir, analizar y evaluar el comportamiento de un sistema electrónico.

VHDL es un lenguaje poderoso que permite la integración de sistemas digitales sencillos y/o complejos dentro de un dispositivo lógico programable, sea este de baja capacidad de integración como un GAL, o de mayor capacidad como los CPLDs y FPGAs.

2.1. Unidades básicas de diseño

La estructura general de un programa en VHDL está formado por **módulos o unidades** de diseño. Cada uno de estos módulos está compuesto por un conjunto de declaraciones e instrucciones que definen, describen, estructuran, analizan y evalúan el comportamiento de un sistema digital.

Existen cinco tipos de unidades de diseño en VHDL: la **declaración de entidad** (entity declaration), la **arquitectura** (architecture), la **configuración** (configuration), la **declaración del paquete** (package declaration) y el **cuerpo del paquete** (package body). En el desarrollo de programas en VHDL pueden o no utilizarse tres de los cinco módulos anteriores de diseño, pero dos de ellos (**la entidad** y **la arquitectura**) son indispensables en la estructuración de un programa.

Las declaraciones de entidad, paquete y configuración son consideradas unidades de diseño **primarias**, mientras que la arquitectura y el cuerpo del paquete son unidades de diseño **secundarias**, debido a que dependen de una entidad primaria que debe ser analizada antes que ellas.

2.1.1 Entidad (entity)

Una entidad es la representación en hardware de un sistema digital, es decir, entidades son todos aquellos elementos electrónicos (compuertas, flip-flops, memorias, multiplexores, etc.,) que forman de manera individual o en conjunto un sistema digital. La entidad puede representarse de muy diversas maneras, por ejemplo, la figura 2.1a muestra la arquitectura de un sumador a nivel de compuertas, esta misma entidad puede ser representada a nivel de sistema indicándose únicamente las entradas (Cin, A y B) y salidas (Suma y Cout) del circuito, figura 2.1b

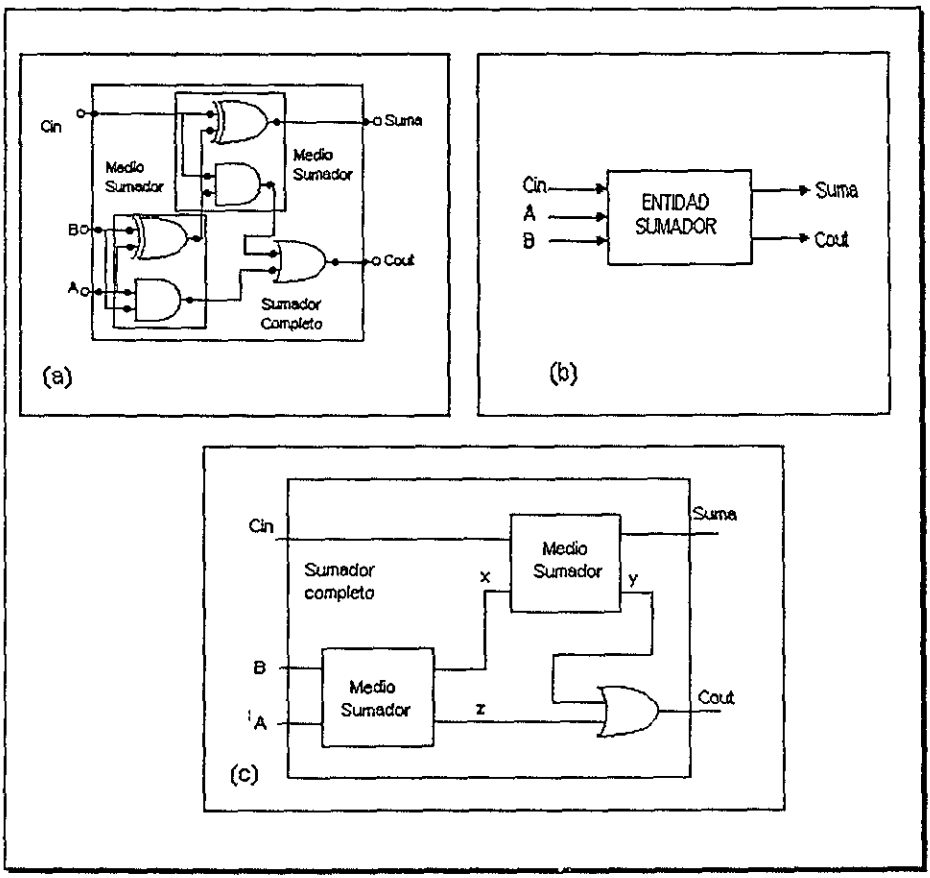


Fig. 2.1(a). Símbolo esquemático representativo de una entidad de diseño

b) Símbolo funcional de la entidad; (c) Diagrama a bloques representativo de la entidad

De igual forma, la integración de varios subsistemas también puede representarse mediante una sola entidad, tal y como se muestra en la figura 2.1c. Los subsistemas pueden conectarse internamente entre sí, pero la entidad sigue identificando de manera clara sus entradas y salidas generales.

2.1.1.1. Puertos de entrada - salida

Cada una de las señales de entrada y salida en una entidad son referidas como un **puerto**, el cual es similar a una terminal (pin) de un símbolo esquemático. Todos los puertos que son declarados deben tener un **nombre, un modo** y un **tipo de dato**. El nombre es utilizado como una forma de llamar al puerto; el modo permite definir la dirección que tomará la información, mientras que el tipo define que clase de información se transmitirá por el puerto. Por ejemplo, en el comparador de igualdad de la figura 2.2, los puertos de entrada están representados por **a** y **b**, y el puerto de salida por la variable **c**.

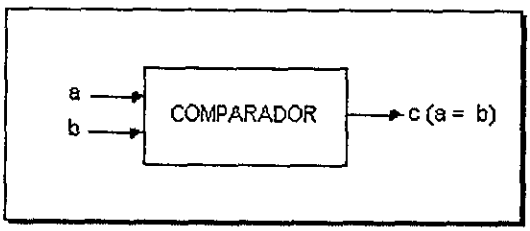


Figura 2.2 Comparador de igualdad

2.1.1.2. Modos

Como ya se mencionó, un modo permite definir la dirección en la cual el dato es transferido a través de un puerto. Un modo puede tener uno de cuatro valores: **in** (entrada), **out** (salida), **inout** (entrada/salida) y **buffer**. Figura 2.3

- **Modo in**. Se refiere a las señales de entrada a la entidad. El modo **in** es solo unidireccional y permite solo el flujo de datos hacia dentro de la entidad.
- **Modo out**. Indica las señales de salida de la entidad.
- **Modo inout**. Este modo permite declarar a un puerto de forma bidireccional, es decir como de entrada/salida, además permite la retroalimentación de señales dentro o fuera de la entidad.
- **Modo buffer**. El modo buffer permite hacer retroalimentaciones internas dentro de la entidad, pero a diferencia del modo **inout**, el puerto declarado se comporta como una terminal de salida.

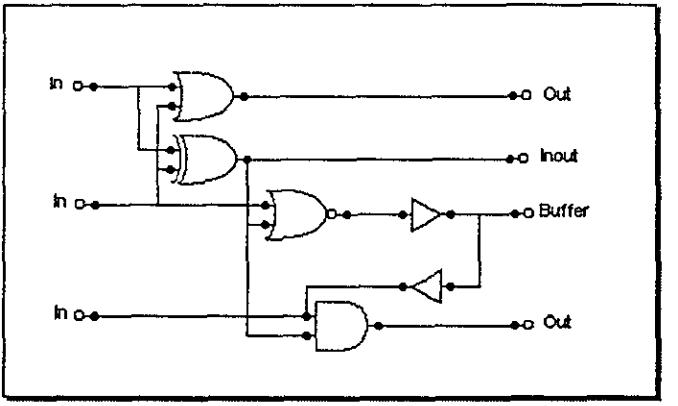


Figura 2.3. Modos y el curso de sus señales

2.1.1.3. Tipos de datos

Los tipos son los valores (datos) que el diseñador establece para los puertos de entrada y salida dentro de una entidad, y son asignados de acuerdo a las características de un diseño en particular. Los tipos más utilizados en VHDL son el tipo **bit** el cual tiene valores de 0 y 1 lógico, el tipo **boolean** (booleano) que define valores de verdadero o falso en una expresión, **bit_vector** (vectores de bits) que representa un conjunto de bits para cada variable de entrada y/o salida, el tipo **integer** (entero). Estos son solo algunos de los tipos que maneja VHDL, pero no son los únicos, en el apéndice A se explican con mas detalle todos los tipos existentes.

2.1.2. Declaración de entidades

Como se mencionó en la sección 2.1, los módulos elementales en el desarrollo de un programa dentro del Lenguaje de Descripción en Hardware (**VHDL**) son la entidad y la arquitectura.

Una declaración de una entidad consiste en la descripción de las entradas y salidas de un circuito de diseño identificado como **entity** (entidad), es decir, la declaración señala las terminales o pines de entrada y salida con los que cuenta la entidad de diseño.

La forma de declarar la entidad correspondiente al circuito de la figura 2.1(b), se muestra a continuación en el listado 1.

```
1--Declaración de la entidad de un circuito sumador
2     entity sumador is
3     port (a,b, Cin:   in bit;
4           suma, Cout: out bit);
5     end sumador;
```

Listado 1. Declaración de una entidad de diseño

Los números de las líneas no son parte del código, éstos son usadas como referencia para explicar alguna sección en particular. Las palabras en negritas están reservadas para el lenguaje de programación VHDL, es decir tienen un significado especial para el programa; las otras palabras son asignadas por el diseñador.

Ahora comencemos a analizar el código línea por línea. Observemos que la línea 1 inicia con dos guiones (--), los cuales indican que el texto que esta a la derecha es un comentario, usado únicamente con el fin de documentar el programa, ya que todos los comentarios son ignorados por el compilador. En la línea 2, se inicia la declaración de la entidad utilizando la palabra reservada **entity**, seguida del **identificador** o nombre de la entidad (sumador) y la palabra reservada **is**. Los puertos de entrada y salida (**port**) son declarados en las líneas 3 y 4 respectivamente; en este caso los puertos de entrada son *a*, *b* y *Cin*, mientras que los puertos de salida están representados por *suma* y *Cout*. El tipo de dato que cada puerto maneja es del tipo **bit**, lo cual indica que solo pueden manejarse valores de '0' y '1' lógicos.

Debemos notar que como cualquier lenguaje de programación, VHDL sigue una sintaxis y una semántica dentro del código, la cual hay que respetar. En esta entidad notemos el uso de punto y coma (;) al finalizar una declaración, así como los dos puntos (:) al asignar nombres a las entradas y salidas.

2.1.2.1. Identificadores

Los identificadores son simplemente los nombres o etiquetas que se usan para referenciar variables, constantes, señales, procesos, etc. Estos identificadores pueden ser números, letras del alfabeto y guiones bajos (_) que separen caracteres. Todos los identificadores deben seguir ciertas especificaciones o reglas para que puedan ser compilados sin errores:

REGLA	INCORRECTO	CORRECTO
El primer caracter siempre es una letra mayúscula o minúscula.	4suma	Suma4 SUMA4
El segundo carácter no puede ser un guión bajo	S_4bits	S4_bits
Dos guiones juntos no son permitidos	Resta__4	Resta_4_
Un identificador no puede utilizar símbolos	Clear#8	Clear_8

VHDL cuenta con una lista de palabras reservadas que no pueden ser utilizadas como identificadores (ver Apéndice B).

2.1.3. Diseño de entidades utilizando vectores

La operación lógica realizada en el circuito del listado 1 usa bits individuales, los cuales solo pueden representar dos valores (0 o 1). De manera general, en la práctica se utilizan conjuntos (palabras) de varios bits; en VHDL las palabras binarias son conocidas como **vectores de bits,** los cuales son considerados un grupo y no como bits individuales. Como ejemplo considérese los vectores de 4 bits que se muestran a continuación:

```
vector_a     = [a3,a2, a1,a0]
vector_b     = [b3,b2,b1,b0]
vector_suma  = [s3, s2, s1,s0]
```

En la figura 2.4 se observa la entidad del sumador analizado anteriormente, solo que ahora incorpora vectores de 4 bits en sus puertos:

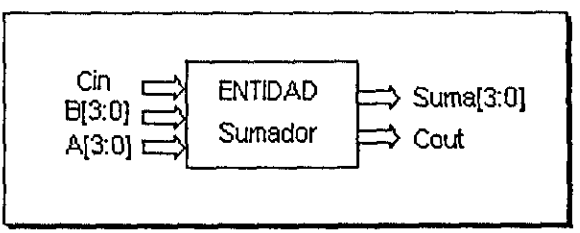


Figura 2.4. Entidad representada con vectores.

La manera de describir en VHDL una configuración que utilice vectores, radica en la utilización de la sentencia **bit_vector**, mediante la cual se especifican los componentes de cada uno de los vectores utilizados. La parte del código que se usa para declarar un vector dentro de los puertos es el siguiente:

```
port ( vector_a, vector_b: in bit_vector  (3 downto 0);
                vector_suma: out bit_vector (3 downto 0));
```

esta declaración define a los vectores (a, b y suma) con cuatro componentes distribuidos en orden descendente por medio del comando 3 **downto** 0 (3 **hacia** 0):

vector_a(3) = a3	vector_b(3) = b3	vector_suma(3) = s3
vector_a(2) = a2	vector_b(2) = b2	vector_suma(2) = s2
vector_a(1) = a1	vector_b(1) = b1	vector_suma(1) = s1
vector_a(0) = a0	vector_1(0) = b0	vector_suma(0) = s0

Una vez que se ha establecido el orden en que aparecerán los bits enunciados en cada vector, este no puede ser modificado, a menos que se utilice el comando **to** (0 **to** 3) que indica el orden de aparición de forma *ascendente*.

El código en VHDL que describe la utilización de vectores quedaría de la siguiente manera, obsérvese cómo la entrada Cin (Carry in) y la salida Cout (Carry out) están expresadas de forma individual, listado 2:

```
entity sumador is
port (a,b:   in bit_vector (3 downto 0);
      Cin:   in bit;
      Cout:  out bit;
      Suma:  out bit_vector(3 downto 0));
end sumador;
```

Listado 2. Entidad de un sumador de 4 bits

2.1.3.1. Declaración de entidades utilizando librerías y paquetes.

Una parte importante en la programación con VHDL radica en el uso de **librerías** y **paquetes** que permiten declarar y almacenar estructuras lógicas, seccionadas o completas, que facilitan el diseño.

Una librería (**biblioteca**) es un lugar al que se puede acceder para utilizar las unidades de diseño predeterminadas por el fabricante de la herramienta (**paquete**), y que se utilizan para agilizar el diseño. En VHDL se encuentran definidas dos librerías llamadas **ieee** y **work**, (figura 2.5). Como puede observarse dentro de la librería **ieee**, se encuentra el paquete

`std_logic_1164`, mientras que en la librería **work** se encuentran: `numeric_std`, `std_arith` y `gatespkg`.

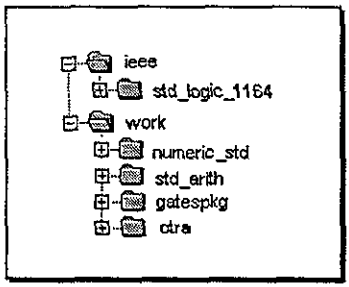


Figura 2.5. Contenido de las librerías ieee y work

Dentro de una librería también se permite almacenar el resultado obtenido de la compilación de un diseño, con el fin de que este pueda ser utilizado dentro de uno o varios programas. La librería **work** es el lugar establecido donde se almacenan los programas que el usuario va realizando. Esta librería se encuentra siempre presente en la compilación de un diseño, y mientras no sea especificada otra librería, los diseños serán guardados dentro de ella. La carpeta (otra) mostrada en la figura 2.5, representa esta situación.

Un paquete es una unidad de diseño que permite desarrollar un programa en VHDL de una manera ágil, debido a que en este se encuentran algoritmos prestablecidos (sumadores, restadores, contadores, etc.) que ya tienen optimizado un comportamiento; por lo cual, el diseñador no necesita caracterizar paso a paso una nueva unidad de diseño si esta ya se encuentra almacenada en algún paquete, en este caso solo basta llamarlo y especificarlo dentro del programa. Es decir, un paquete no es mas que una unidad de diseño formada por declaraciones, programas, componentes y subprogramas, que incluyen los diversos tipos de datos (`bit`, `booleano`, `std_logic`), empleados en la programación en VHDL y que generalmente forman parte de las herramientas en software.

Finalmente, cuando en el diseño se utiliza algún paquete, es necesario llamar a la librería que lo contiene, esto se lleva a cabo por medio de la siguiente declaración:

library ieee;

la cual permite el uso de todos los componentes incluidos en la librería **ieee**. En el caso de la librería de trabajo (**work**) su uso no requiere de la declaración **library**, dado que la carpeta work siempre está presente al desarrollar un diseño.

2.1.3.1.1. Paquetes

- El paquete **std_logic_1164** (estándar lógico_1164) que se encuentra dentro de la librería ieee, contiene a todos los tipos de datos comunmente utilizados en VHDL (`std_logic_vector, std_logic`, entre otros).

La forma en que se accede a la información contenida dentro de un paquete, es por medio de la sentencia **use**, seguida del nombre de la librería y del paquete respectivamente:

use nombre_librería.nombre_paquete.**all**;

por ejemplo

use ieee.std_logic_1164.**all**;

en este caso ieee es la librería, std_logic_1164 es el paquete y la palabra reservada **all**, indica que todos los componentes almacenados dentro del paquete pueden ser utilizados.

- El paquete **numeric_std**, define funciones para realizar operaciones entre diferentes tipos de datos (sobrecargado), además los tipos puede representarse con y sin signo. Apéndice A.
- El paquete **numeric_bit**, define tipos de datos binarios signados y no signados.
- El paquete **std_arith**, define funciones y operadores aritméticos, como igual (=), mayor que (>), menor que (<), entre otros. Apéndice A.

En lo sucesivo, se hará uso extensivo de las librerías y paquetes dentro de los programas desarrollados en el texto.

2.1.4. Arquitecturas (architecture)

Una **arquitectura** se define como la estructura que describe el funcionamiento de una entidad, de tal forma que permita el desarrollo de los procedimientos que se llevarán a cabo con el fin de que la entidad cumpla las condiciones de funcionamiento deseadas.

La gran ventaja que presenta VHDL para definir una arquitectura, radica en la manera en como pueden describirse los diseños. Es decir, en el algoritmo de programación puede describirse desde el nivel de compuertas hasta el nivel de sistema.

De manera general, los estilos de programación utilizados en el diseño de arquitecturas se clasifican como:

- Estilo estructural
- Estilo funcional
- Estilo por flujo de datos

El nombre asignado a estos estilos no es importante, ya que es tarea del diseñador escribir el comportamiento de un circuito utilizando uno u otro estilo, que a su juicio le parezca el más acertado.

2.1.4.1. Descripción Funcional

En la figura 2.6. se muestra un ejemplo de un circuito representado como una descripción funcional. Se llama así porque se describe la forma en como funciona el circuito, es decir las descripciones son hechas considerando la relación que se presenta entre las entradas y las salidas del circuito, sin importar cómo este organizado internamente.

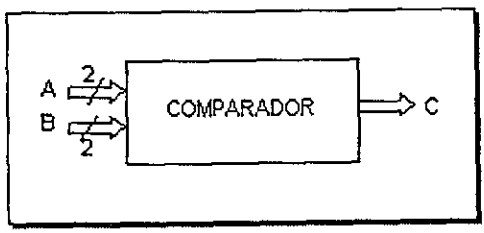


Figura 2.6. Descripción funcional de un comparador de igualdad de dos bits

El código que representa el circuito de la figura 2.6 se muestra en el listado 3.

```
1   -- Ejemplo de una descripción funcional
2   library ieee;
3   use ieee.std_logic_1164.all;
4   entity comp is
5   port (a,b: in bit_vector( 1 downto 0);
6           c: out bit);
7   end comp;
8   architecture funcional of comp is
9   begin
10  compara: process (a,b)
11  begin
12        if a = b then
13            c <='1';
14        else
15            c<='0';
16  end if;
17  end process compara;
18  end funcional;
```

Listado 3. Arquitectura funcional de un comparador de igualdad de 2 bits

Ahora obsérvese el código desde la línea 8 a la 18, donde se desarrolla el algoritmo (**architecture**) que describe el funcionamiento del comparador. Para iniciar la declaración de la arquitectura (línea 8) es necesario definir un nombre arbitrario con la que pueda ser referenciada, en nuestro caso el nombre asignado es **funcional**, además de incluir a la entidad con la que está relacionada (**comp**). En la línea 9, se puede observar el inicio (**begin**) de la sección donde se comienza a declarar los procesos que rigen el comportamiento del sistema. La declaración del **proceso** (línea 10) es utilizada para la definición de algoritmos, e inicia con una etiqueta opcional (en este caso *compara*), seguida de dos puntos (:), la palabra reservada **process** y una la *lista sensitiva* (a y b), la cual hace referencia a las señales que determinan el funcionamiento del proceso.

Siguiendo con el análisis del código, nótese que de la línea 11 a la 17 el proceso es ejecutado mediante **declaraciones secuenciales** del tipo **if-then-else (si-entonces-sino)**. Esto se interpreta como sigue: si el valor de la señal *a* es igual al valor de la señal *b*, entonces '1' es asignado a *c*, sino se asigna un '0' (el símbolo <= se lee como "es asignado a"). Una vez que se ha definido el proceso, se termina con la palabra reservada **end process** y de manera opcional el nombre del proceso (compara); de forma similar la etiqueta (funcional) al terminar la arquitectura en la línea 18.

Como se puede observar, la descripción funcional radica principalmente en el uso de procesos y de declaraciones secuenciales, las cuales permiten de forma rápida el modelado de la función.

2.1.4.2. Descripción por flujo de datos

La descripción por flujo datos indica la forma en que los datos pueden ser transferidos de una señal a otra sin el uso de declaraciones secuenciales (**if-then-else**). Este tipo de descripciones permiten definir el flujo que tomarán los datos entre módulos encargados de implementar operaciones. En este tipo de descripción se pueden utilizar dos formatos: primero mediante instrucciones **when-else** (cuando-sino) o segundo, mediante ecuaciones booleanas.

2.1.4.2.1. Descripción por flujo de datos utilizando when-else

A continuación se muestra el código del comparador de igualdad de dos bits descrito anteriormente.

Notemos que la diferencia entre los listados 3 y 4, radica en la eliminación del proceso y en la descripción sin el uso de declaraciones secuenciales (if-then-else). En VHDL se manejan dos tipos de declaraciones: **secuenciales** y **concurrentes**.

```
--Ejemplo de declaración de la entidad de un comparador
library ieee;
use ieee.std_logic_1164.all;
entity comp is
port (a,b:  in bit_vector (1 downto 0);
        c:  out bit);
end comp;

architecture f_datos of comp is
begin
c <= '1' when (a = b)  else '0';
end f_datos;
```

Listado 4. Arquitectura de flujo de datos

Una declaración secuencial de la forma **if-then-else,** la observamos en el listado 3 dentro del proceso, en donde su ejecución debe seguir un orden para evitar la pérdida de la lógica descrita, mientras que en una declaración concurrente esto no es necesario, ya que el orden en que son ejecutadas no es importante, tal es el caso del listado 4.

2.1.4.2.2. Descripción por flujo de datos utilizando ecuaciones booleanas

Otra forma de describir el circuito comparador de dos bits que se muestra en la figura 2.7, es mediante la obtención de sus ecuaciones booleanas. En el listado 5 se observa este desarrollo:

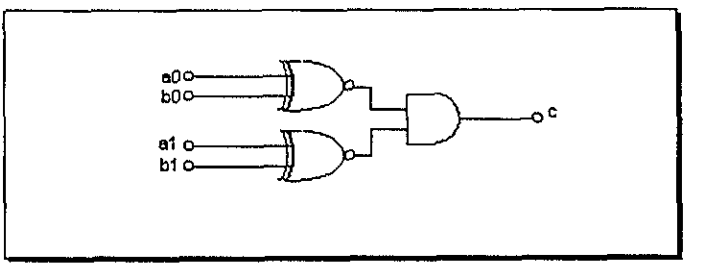


Figura 2.7. Comparador de dos bits implementado con compuertas

```
-- Ejemplo de declaración de la entidad de un comparador
library ieee;
use ieee.std_logic_1164.all;
entity comp is
port    ( a,b:   in bit_vector (1 downto 0) ;
           c:    out bit) ;
end comp;

architecture booleana of comp is
begin
c <= (a(1)  xnor b(1)
    and   a(0) xnor b(0));
end booleana;
```

Listado 5. Arquitectura de forma flujo de datos implementada por medio de ecuaciones booleanas.

La forma de flujo de datos en cualquiera de sus representaciones describe el camino que los datos siguen al ser transferidos de las operaciones efectuadas entre las entradas a y b a la señal de salida c.

2.1.4.3 Descripción estructural

Como su nombre lo indica, una descripción estructural basa su comportamiento en modelos lógicos ya establecidos, (compuertas, sumadores, contadores, etc.) como veremos mas adelante, estas estructuras pueden ser diseñadas por el usuario y guardadas para su posterior utilización o extraídas de los paquetes contenidos en las librerías de diseño del software que se esté utilizando.

En la figura 2.8, se observa una representación esquemática del circuito comparador de igualdad de 2 bits, el cual está formado por compuertas nor-exclusivas y una compuerta AND.

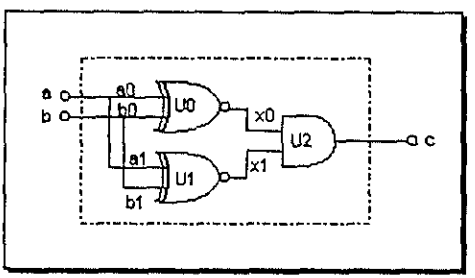


Figura 2.8. Representación esquemática de un comparador de 2 bits

En nuestro caso, cada una de estas compuertas (modelo lógico) se encuentra dentro del paquete **gatespkg**[6] del cual son extraídas para estructurar el diseño. A su vez este tipo de arquitecturas estándar son conocidas como **componentes**, las cuales al ser conectadas entre sí por medio de señales internas (x0, x1), permiten integrar una solución. En VHDL esta conectividad se conoce como **netlist**[7] o listado de componentes.

Para iniciar la programación de una entidad de manera estructural, es necesario la descomposición lógica del diseño en pequeños submódulos (jerarquizar), los cuales permiten analizar de manera práctica el circuito, ya que la función de entrada/salida es conocida. En nuestro ejemplo se conoce la función de salida de las dos compuertas **xnor**, por lo que al unirlas a la compuerta **and**, la salida *c* es solo el resultado de la operación **and** efectuada internamente a través de las señales x0 y x1, figura 2.8.

Es importante resaltar que una jerarquía en VHDL se refiere a dividir en bloques y no el que un bloque tenga mayor peso que otro. Esta forma de dividir el problema hace de la descripción estructural una forma sencilla de programar. Dentro del contexto del diseño lógico esto es observable cuando se analiza por separado alguna sección de un sistema integral

En el listado 6 se muestra el código del programa que representa al diagrama esquemático de la figura 2.8.

```
library ieee;
use ieee.std_logic_1164.all;
entity comp is port(
        a,b: in bit_vector (0 to 1);
     c: out bit);
end comp;

use work.gatespkg.all;
architecture estructural of comp is
signal x: bit_vector (0 to 1);
begin

   U0:    xnor2 port map (a(0), b(0), x(0));
   U1:    xnor2 port map (a(1), b(1), x(1));
   U2:    and2       port map (x(0), x(1), c);

end estructural;
```

Listado 5. Descripción estructural de un comparador de igualdad de 2 bits

[6] El paquete gatespkg no se encuentra disponible en todas las herramientas de VHDL

[7] Un netlist se refiere a la forma en como se encuentran conectados los componentes dentro de una estructura y las señales que propician esta interconexión.

En el código se puede ver que en la entidad únicamente son descritas las entradas y salidas del circuito (a, b y c respectivamente), tal como se ha venido haciendo. Los componentes **xnor** y **and** no se declaran, debido a que se encuentran incluidos en el paquete de compuertas (**gatespkg**), el cual a su vez está contenido dentro de la *librería de trabajo* (**work**).

En la arquitectura nombrada *estructural*, se describe la estructura de la siguiente forma: cada compuerta se maneja como un bloque lógico independiente (componente) del diseño original, al cual se le asigna una variable temporal (U0,U1 y U2); la salida de cada uno de estos bloques se maneja como una señal x (x0 y x1), las cuales se declaran dentro de la arquitectura y no en la entidad. De forma posterior, la compuerta **and** recibe las dos señales de x, ejecuta la operación y asigna el resultado a la salida *c* del circuito.

2.1.4.4. Comparación entre los estilos de diseño

El estilo de diseño utilizado en la programación del circuito, depende exclusivamente del diseñador y de la complejidad del proyecto. Por ejemplo, un diseño puede describirse por medio de ecuaciones booleanas, pero si este es muy extenso, quizá sea más apropiado describirlo mediante estructuras jerárquicas para dividirlo; ahora bien, si se requiere diseñar un sistema cuyo funcionamiento dependa solo de sus entradas y salidas es conveniente utilizar la descripción por flujo de datos, la cual presenta la ventaja de requerir un número menor de instrucciones y de que el diseñador no necesita un conocimiento previo de cada componente que forma el circuito.

Capítulo 3

Síntesis de Diseño e Implementación

La tendencia actual en la fabricación y producción de casi cualquier producto es la utilización de programas computacionales (software de aplicación) que permiten al diseñador moldear y modelar detalladamente las características del producto a elaborar. El uso de estos programas (que actualmente se consideran como una herramienta poderosa de diseño) representa para la industria que los emplea, la optimización de los recursos con los que cuenta, así como el aprovechamiento de los avances tecnológicos de vanguardia, elevando su capacidad competitiva y la calidad de los productos que desarrolla.

En el terreno de la electrónica las ventajas que presenta la utilización de estas herramientas en el diseño de circuitos analógicos y/o digitales es muy importante, ya que permiten conocer de forma anticipada el comportamiento del circuito antes de programarse y/o fabricarse, reduciendo notablemente los tiempos de diseño, así como también los costos de producción, pues en lugar de emplear equipo de medición, instalaciones adecuadas, material y personal capacitado para efectuar pruebas preliminares y modificaciones al diseño del circuito, sólo se tendrá que adquirir la herramienta de diseño adecuada a su necesidad.

Ya con anterioridad se han mencionado las ventajas que presenta el diseño de circuitos ASIC´S mediante dispositivos lógicos programables, una de las más importantes radica en el bajo costo de los elementos requeridos para el desarrollo de estas aplicaciones, dado que el soporte básico lo forman tan solo una computadora personal, un grabador de dispositivos lógicos programables y el software de aplicación. Sobre esto último existen varias compañías que brindan el servicio para este tipo de aplicaciones (capitulo 1 sección 1.6). En nuestro caso utilizaremos el software WarpR4 de la compañía Cypress Semiconductor [1].

La finalidad de introducir en este capítulo el uso de software es con la idea de que el lector pueda de manera inmediata realizar y programar sus primeros diseños y/o circuitos. La practica obtenida le permitirá enfrentar con relativa facilidad los ejemplos de los siguientes capítulos.

3.1 WarpR4 la herramienta de soporte

WarpR4 es una herramienta para el diseño con lógica programable creada por Cypress Semiconductor, la cual procesa varios tipos de entrada de datos (captura esquemática, compilador estándar de VHDL y la combinación de ambos), haciéndola muy flexible y funcional.

Actualmente es uno de los estándares mas usados en la industria ya que presenta la característica de optimizar los diseños de forma rápida y precisa utilizando solamente una pequeña área del circuito, además ofrece una interface gráfica (Galaxy) amigable con el usuario.

En la parte correspondiente al hardware WarpR4 permite la grabación en diferentes familias de dispositivos lógicos programables por ejemplo: PLD's (22V10,20V8 y 16V8), CPLD (de la serie Cypress FLASH370™), CPLD (de la familia MAX340™) y FPGA (de la familia FPGA-pASIC380).

3.1.1 Iniciando WarpR4

Para utilizar la herramienta de trabajo WarpR4, esta debe de encontrarse previamente instalada. En la figura 3.1 se muestra esta condición y como se observa, dentro de este software se encuentra la interface gráfica (Galaxy), el simulador (Nova), las notas técnicas (Release Notes)y la barra de herramientas (Warp Toolbar).

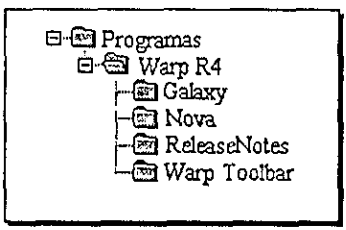


Figura 3.1 El software de soporte WARPR4

3.1.2 Galaxy (Interface Gráfica del Usuario)

Galaxy es la interface que permite la interacción entre el usuario y la herramienta de trabajo, en esta se realiza la edición, compilación y síntesis de los archivos escritos en código VHDL

Para iniciar Galaxy es necesario entrar al menú de WarpR4 y seleccionar Galaxy, otra forma de ejecutarlo es desde la barra de herramientas, la cual puede estar de manera fija en la

pantalla, figura 3.2a. La selección de Galaxy da origen al menú gráfico mostrado en la figura 3.2b

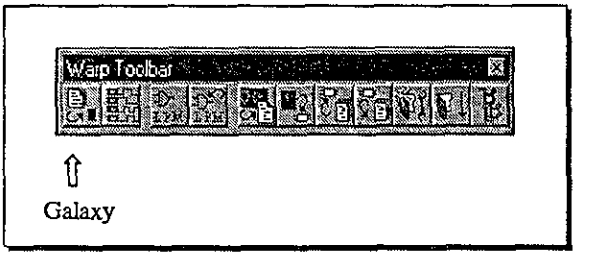


Figura 3.2a. Barra de herramientas de Warp

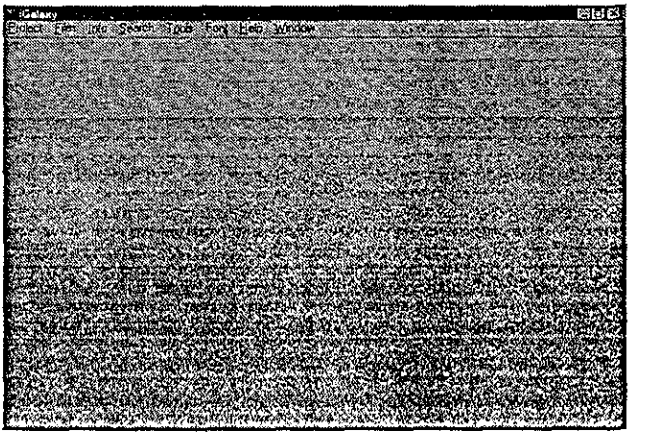


Figura 3.2b. Menú Gráfico de Galaxy

3.1.2.1 Creación inicial de un Proyecto (Project)

Para iniciar un proyecto es necesario seleccionar **Project**. Aquí se presentan dos casos, primero: el de inicio, al seleccionar **Project** por primera vez aparecerá una pantalla similar a la mostrada en la figura 3.3, en esta se pide que declaremos la ruta donde se guardarán los proyectos (en nuestro caso ejemplo), así como también el nombre que se asignará al archivo (diseños) en el que se almacenaran todos los diseños creados por el usuario.

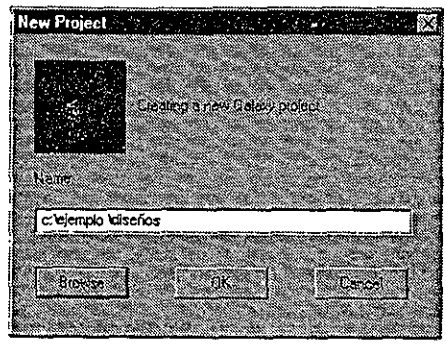


Figura 3.3. Creación de un nuevo Proyecto

Segundo, si ya con anterioridad existe un proyecto, al seleccionar Galaxy aparecerá la pantalla de la figura 3.4.

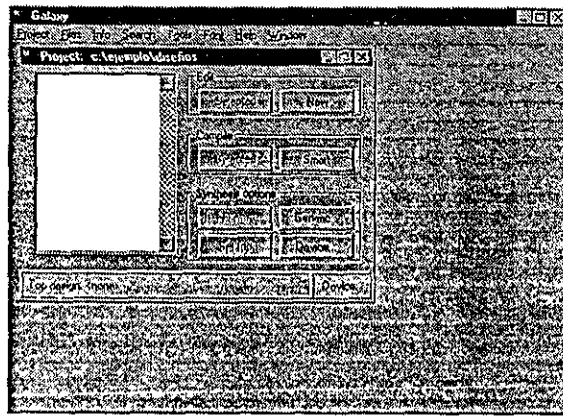


Figura 3.4. Proyecto creado en Galaxy

En esta se observa la presencia de un menú para Galaxy y uno para **Proyect.**

Dentro del menú **Proyect** existe el bloque **Edit,** en él se tiene la posibilidad de seleccionar un diseño ya almacenado (Select) o bien editar uno nuevo a través de la opción **New**, con esta aparecerá la pantalla de edición que se mu
muestra en la figura 3.5

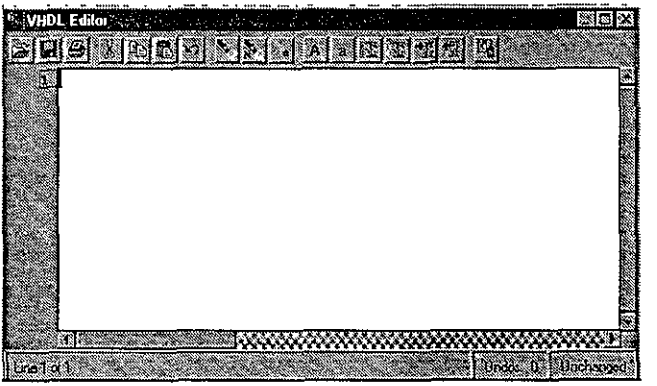


Figura 3.5 Editor de Galaxy

Para proseguir con nuestro análisis, considere editar el comparador de igualdad de dos bits mostrado en la figura 2.6 y por comodidad desplegado nuevamente en la figura 3.6

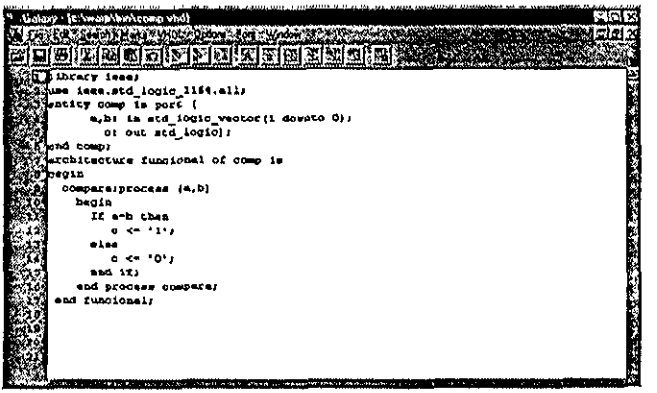


Figura 3.6. Edición de un Programa

Una vez que el programa ha sido editado, es necesario almacenarlo a través de la opción **File ⇨ Save as,** la cual se encuentra dentro del menú de Galaxy (figura 3.4). Nótese como de forma automática el programa agrega la extensión **.vhd** (compara .vhd). El diseño así generado debe de almacenarse para su posterior uso, esta operación se realiza a través del

menú de Galaxy por medio del comando **File** ⟹ **Add**, de esta manera el nuevo diseño aparecerá listado en la ventana de **Proyect**, figura 3.7.

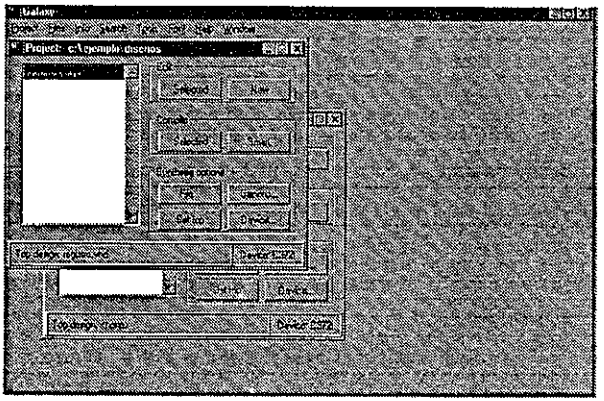


Figura 3.7 Agregando un nuevo diseño

3.1.3 Compilación del diseño

Un compilador lógico tiene como función procesar y sintetizar el diseño, generando el archivo JEDEC (conocido como mapa de fusibles); este archivo es el que reconoce el grabador de dispositivos lógicos programables. Como una ventaja adicional este tipo de compiladores detectan los errores de sintaxis y semántica, los cuales en el caso de existir impiden la generación del archivo JEDEC.

3.1.3.1 Selección del dispositivo.

Antes de compilar un diseño es necesario seleccionar al dispositivo en el que se grabará la aplicación. La selección se realiza dentro del bloque **synthesis options** del menú **Project** (figura 3.7), en este se elige la opción dispositivo (**device**), desplegándose la pantalla mostrada en la figura 3.8.

En la parte superior izquierda de la pantalla se observan las opciones **Device y Package**; dentro de **Device** se elige de entre varias familias, la correspondiente al dispositivo que se va a utilizar (en este caso seleccionamos la familia **C372I**). En la opción **Package** se despliega una lista de los dispositivos disponibles dentro de la familia seleccionada, en ella se encuentra el tipo de encapsulado del dispositivo elegido, en este caso es el **CY7C372I-66JC.**

Otra de las opciones a escoger, es la referente al voltaje de funcionamiento, el cual puede ser de 3.3 ó 5 Volts. El dispositivo CY7C372I funciona con 5 volts (consultar las hojas técnicas en el Apéndice D)

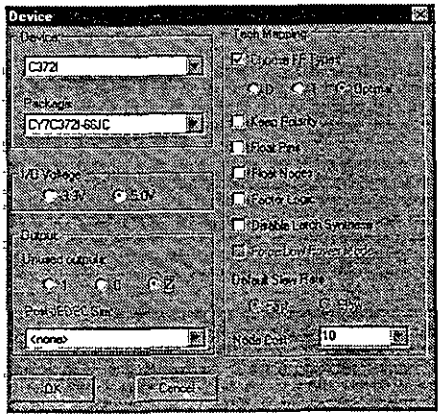


Figura 3.8 Elección del dispositivo

3.1.3.2 Tecnología del dispositivo (Tech Mapping)

La tecnología del dispositivo se refiere a las opciones de caracterización disponibles para el circuito seleccionado, que a su vez depende de la estructura de su arquitectura interna.

- *Elegir tipos de Flip-Flops (Choose FF Types)*

Esta opción permite seleccionar los diferentes tipos de flip-flops que se utilizan en la optimización de un diseño. La opción **Opt** es la recomendada, ya que permite al programa elegir de forma automática el flip-flop que mejor se adecue a cada diseño.

- *Mantener polaridad (Keep Polarity)*

Cuando esta opción es habilitada el usuario tiene la posibilidad de elegir la polaridad deseada para cada salida. Cuando no está seleccionada el programa asigna de manera automática la polaridad que el considera mas adecuada.

- *Terminales y nodos variables (Float pins, Float nodes)*

Se refiere a las terminales y a los nodos internos del dispositivo, que serán ignorados al momento de implementar la lógica producida por los procesos de síntesis y optimización. Generalmente esta opción no se activa, dejando al criterio del programa estos cambios.

- *Factor Lógico (Factor Logic)*

Es una opción aplicable únicamente a la familia MAX340, este módulo permite la implementación lógica en tres niveles en lugar del modo normal de dos niveles (suma de productos).

- *Nodos (Node cost)*

Permite sintetizar el diseño de tal manera que sea compatible con otro. El valor recomendado es "10".

- *Opciones de Simulación (PostJEDEC Sim)*

Esta opción es solamente aplicable a PLDs y CPLDs, consiste en un modelo de simulación que permite al usuario simular el diseño con información basada en periodos de tiempo. El menú PostJEDEC Sim despliega una lista de simuladores disponibles. En nuestro ejemplo la opción no está seleccionada (<none>).

3.1.3.3. Opciones Genéricas (Generics)

Diversas características relativas a la compilación son consideradas dentro de la opción **Generics**, figura 3.9.

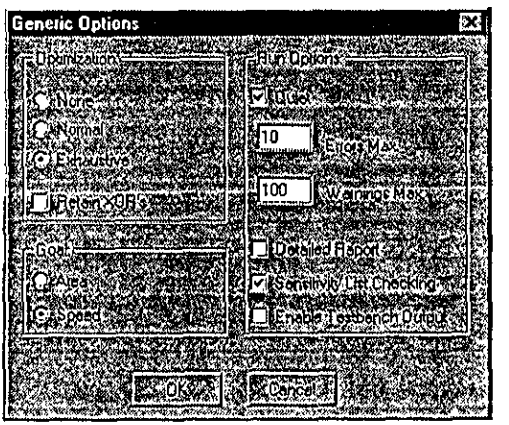


Figura 3.9. Opción de Genéricos

- *Nivel de optimización (Optimization)*

Esta propiedad se utiliza para indicar el nivel en el que se quiere optimizar el diseño. El nivel exhaustivo permite una optimización que abarca diagramas de estado y condiciones no importa.

- *Retención de XORs (Retain XOR's)*

Esta opción provoca que Warp retenga operadores XOR encontrados en el diseño, con el fin de lograr una implementación más adecuada en el dispositivo.

- *Síntesis de Campos (Goals)*

Con esta opción el usuario puede seleccionar el camino que seguirá la síntesis del diseño. (Sólo recomendado para usuarios experimentados)

- *Número Máximo de errores y condiciones de riesgo (Max errors, Max Warnings)*

Ambas opciones se utilizan principalmente cuando los diseños son largos. Su función es que el usuario especifique el número máximo de errores y condiciones de riesgo (warning) que se desplegarán en la pantalla.

- *Reporte detallado (Detailed Report)*

Esta opción despliega un reporte detallado de la compilación de un archivo. No es recomendable habilitarla en diseños largos, debido a que el reporte que produce sería también muy extenso.

- *Verificar la lista sensitiva (Sensitivy List Checking)*

Como su nombre lo indica, esta opción verifica la lista sensitiva de los procesos que se encuentran en los diseños. Si el diseño no presenta procesos, esta opción no es recomendable.

Iniciando la compilación del diseño.

Antes de iniciar la compilación del diseño, es necesario que dentro del menú del proyecto se seleccione el archivo y posteriormente la opción **Set top**, ya que esta permite la compilación de dicho diseño (una vez realizado esto el archivo debe aparecer iluminado en la pantalla).

Ahora bien, consideremos que existen dos caminos para iniciar la compilación:

1) Seleccionar de la ventana de proyecto el archivo que se desea compilar. Presionar Selected para iniciar la compilación.

2) Seleccionar el archivo y presionar **Smart**. Cuando un archivo se modifica, es necesario volver a compilarlo.

Una vez compilado el diseño, se genera una pantalla que despliega, entre otras cosas, el resultado de los diversos procesos por los que atraviesa el diseño para ser compilado exitosamente (Figura 3.10).

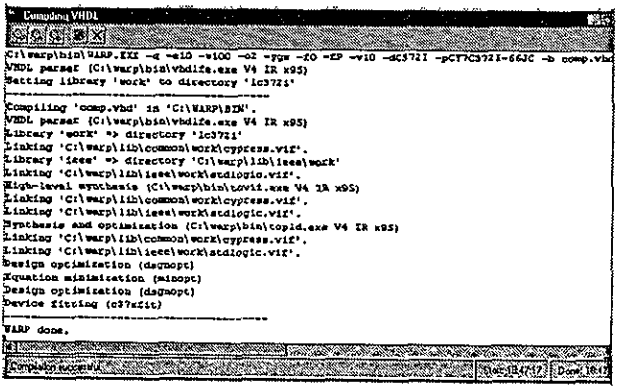


Figura 3.10. Resultados de la compilación

Si el diseño tiene errores, estos aparecen dentro de la ventana de compilación. La manera más sencilla de localizar un error, es presionando el botón **Locate error**, ya que esta opción indica donde se encuentra cada uno de los errores del código.

Una vez terminada la compilación se pueden guardar los mensajes en un archivo, esto se hace únicamente seleccionando la opción **Save as** ⇨ **nombre**.

3.1.4 Nova (el simulador)

Nova es la herramienta que nos permite simular el comportamiento de un diseño basado en el trazado de formas de onda. La manera de entrar al simulador Nova es por medio de la barra de herramientas o dentro del menú de Warp. La pantalla principal se muestra en la figura 3.11

La pantalla principal contiene un menú con las opciones **File, Edit, Simulate, Views** y **Options.** Observe como en la sección izquierda de la pantalla se tiene una columna de botones en los cuales se indica el número de terminal (pin) y el número de nodos[8] asociados a las señales que serán simuladas, figura 3.11

[8] Un nodo es un área del circuito que contiene uno o más puntos donde el usuario puede trazar una señal

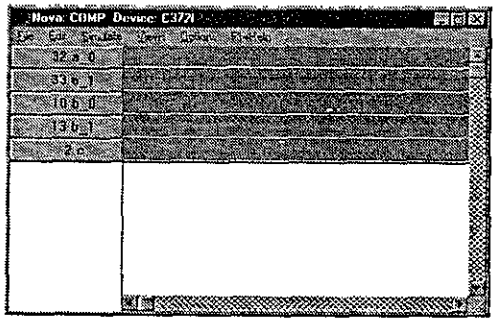


Figura 3.11. Simulador Nova

3.1.4.1 Área de Trazado

El área de trazado despliega los valores de las señales y nodos listados en la columna izquierda de la pantalla.

Para iniciar la simulación se debe asignar un valor a cada señal de entrada. Esto se realiza a través de la opción **Edit**, ya que en ella se encuentran contenidos los diversos valores que puede tomar una señal (valor alto, bajo, señal de reloj y un pulso de reloj), cada valor se asigna seleccionando primero el botón correspondiente a la señal y posteriormente indicando un valor opcional. Figura 3.12.

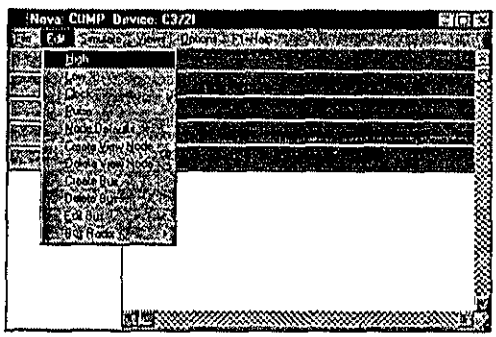


Figura 3.12. Asignación de valores a señales

En nuestro ejemplo asignaremos el valor de '1' a las señales de entrada (a_0,a_1,b_0 y b_1). Una vez realizado lo anterior, se procede a simular el diseño ejecutando el menú **Simulate** ⇨ **Ejecute**, el cual despliega los siguientes resultados. Figura 3.13.

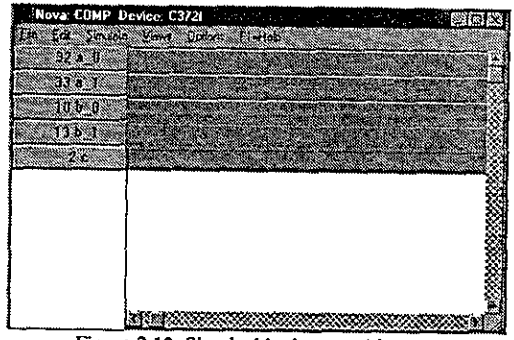


Figura 3.13. Simulación de un archivo

Notemos que la señal de salida *c* toma el valor de '1', debido a que los bits de entrada son ambos iguales.

3.1.5 Archivo de Reporte

La compilación del diseño trae consigo la generación de un archivo de reporte (**.rpt**), el cual presenta información importante del diseño y de la forma en que se implementará en el dispositivo. Para analizar el archivo de reporte, se regresa a la pantalla principal y se elige el menú **Info** ⟹ **Report file**, el cual despliega una pantalla como la que se muestra a continuación, figura 3.14:

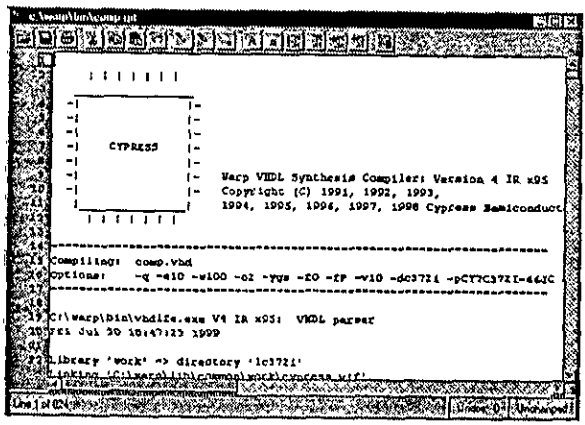


Figura 3.14. Archivo de Reporte

Debido a que el archivo de reporte es muy extenso, se indican en la figura 3.15 cada una de las partes.

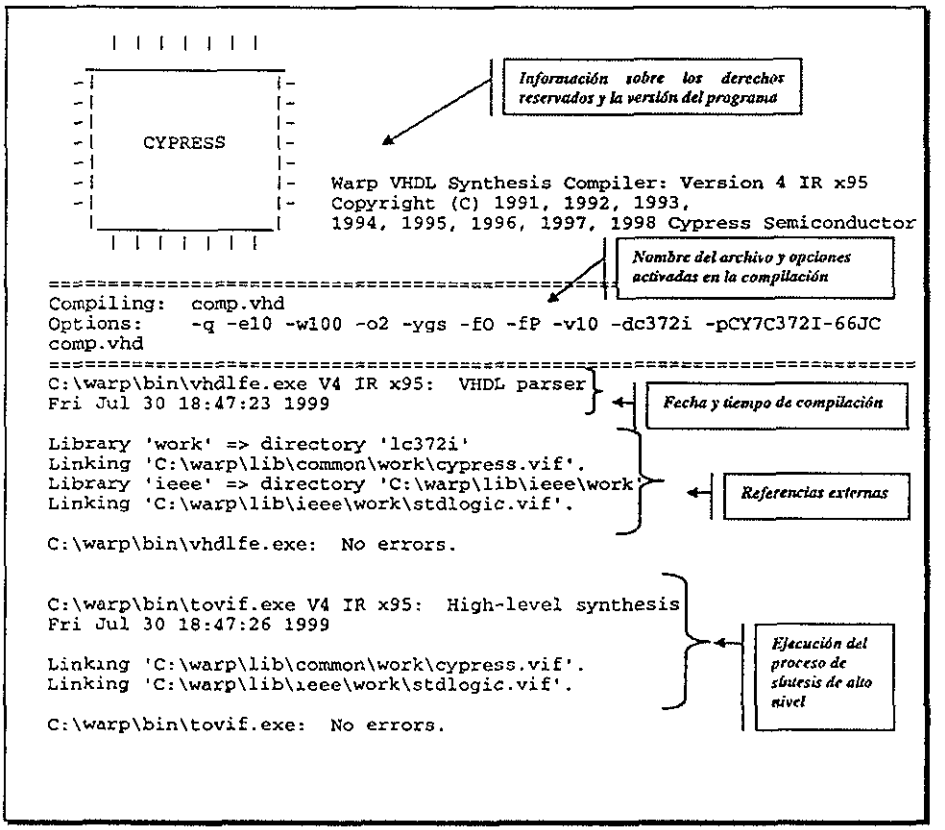


Figura 3.15. Archivo de Reporte

En esta sección del reporte conocida como **front end**, se muestran los datos generados por la herramienta VHDLFE, la cual entre otras tareas, se encarga de verificar los errores de sintaxis y semántica que se pudieran presentar en el diseño. Otra de las funciones que realiza esta herramienta, radica en el traslado del archivo VHDL (por medio de simples ecuaciones y registros), a la siguiente fase conocida como **síntesis y optimización.**

En la figura 3.16 se muestra la parte del reporte generada en la sección conocida como **optimización**, en donde de manera general, se presentan las ecuaciones que rigen el comportamiento del diseño. La notación utilizada para representar dichas ecuaciones es del tipo suma de productos (la / representa una inversión, el * una función AND, el + una función OR). Cada ecuación impresa en este archivo espera ser introducida en una macrocelda de un bloque lógico en particular.

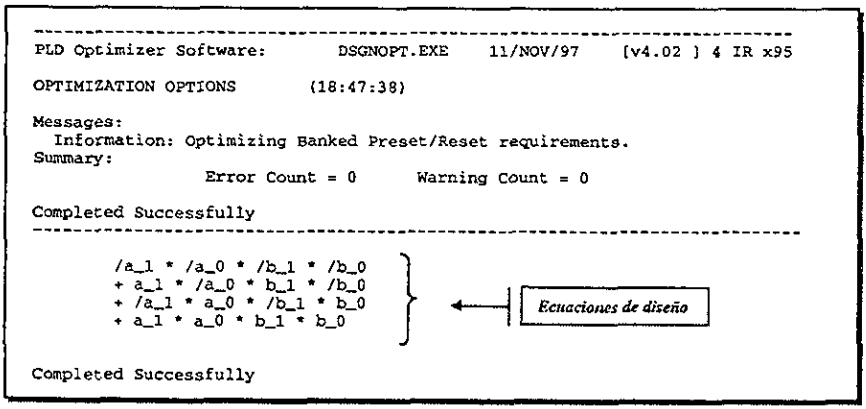


Figura 3.16. Proceso de optimización del diseño

Una vez obtenidas las ecuaciones, el paso siguiente se conoce como **Fitting**, o sea, la forma de implementar dentro del dispositivo la lógica producida por los procesos de síntesis y optimización. Generalmente este término es típicamente utilizado para describir el proceso de distribuir recursos en arquitecturas del tipo CPLD.

En la figura 3.17 se puede observar la implementación de las ecuaciones producidas en la etapa anterior, las cuales serán introducidas en las macroceldas correspondientes a uno de los bloques lógicos con los que cuenta el dispositivo (es este caso el CPLD CY372I).

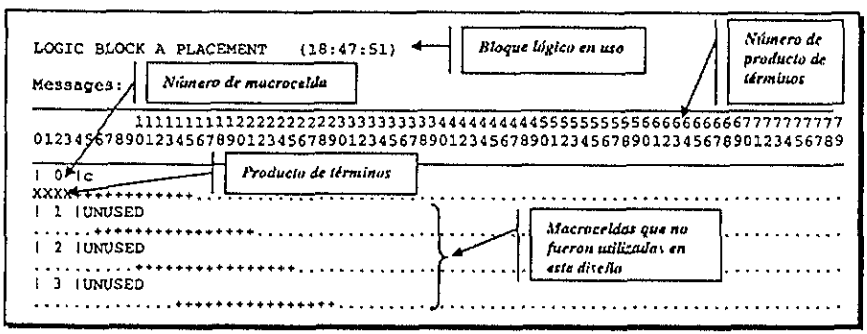


Figura 3.17. Implementación de las ecuaciones dentro del bloque lógico

En este bloque, podemos observar que la primera columna indica el número de la macrocelda dentro del bloque lógico, en donde la "X" representa un producto de términos (PT) y el signo + un espacio dentro del arreglo de PT que no está ocupado en esta macrocelda pero puede ser ocupado en otra.

Otra de las partes incluidas en esta sección es la que se refiere a las estadísticas, las cuales son incorporadas debido a que presentan diversa información relativa al bloque lógico, tal como el número de productos de términos implementados, las señales de control que fueron usadas, el porcentaje de utilización del bloque, etc. (Figura 3.18).

```
Total count of outputs placed        =  1
Total count of unique Product Terms  =  4
Total Product Terms to be assigned   =  4
Max Product Terms used / available   =   4 /  80   = 5.1  %


Control Signals for Logic Block A
---------------------------------
CLK pin  13 : <not used>
CLK pin  35 : <not used>
PRESET      : <not used>
RESET       : <not used>
OE 0        : <not used>
OE 1        : <not used>
OE 2        : <not used>
OE 3        : <not used>
```

Figura 3.18. Estadísticas presentadas por bloque lógico

Otra de las partes que incluye el archivo de reporte, es la correspondiente a la distribución de terminales del dispositivo, la cual se muestra en la figura 3.19.

```
Device:  c372i
Package: CY7C372I-66JC

                  1  :  GND                    23  :  GND
                  2  :  c                      24  :  Not Used
                  3  :  Not Used               25  :  Not Used
                  4  :  Not Used               26  :  Not Used
                  5  :  Not Used               27  :  Not Used
                  6  :  Not Used               28  :  Not Used
                  7  :  Not Used               29  :  Not Used
                  8  :  Not Used               30  :  Not Used
                  9  :  Not Used               31  :  Not Used
                 10  :  b_0                    32  :  a_0
                 11  :  VPP                    33  :  a_1
                 12  :  GND                    34  :  GND
                 13  :  b_1                    35  :  Not Used
                 14  :  Not Used               36  :  Not Used
                 15  :  Not Used               37  :  Not Used
                 16  :  Not Used               38  :  Not Used
                 17  :  Not Used               39  :  Not Used
                 18  :  Not Used               40  :  Not Used
                 19  :  Not Used               41  :  Not Used
                 20  :  Not Used               42  :  Not Used
                 21  :  Not Used               43  :  Not Used
                 22  :  VCC                    44  :  VCC
```

Figura 3.19. Asignación de terminales para el CPLD C372i

En el apéndice D se muestra la hoja técnica correspondiente a este dispositivo. Recordemos que la asignación de terminales (pines) la hace el programa automáticamente, basándose en la distribución de productos de términos dentro de cada macrocelda.

Por último se despliega la información correspondiente al mapa de fusibles (JEDEC) el cual se genera primero para cada bloque lógico y posteriormente para el archivo de salida JEDEC (comp.jed), el cual finalmente se implementará en el dispositivo elegido, figura 3.20.

```
JEDEC ASSEMBLE                (18:47:52)

Messages:
   Information: Processing JEDEC for Logic Block 1.
   Information: Processing JEDEC for Logic Block 2.
   Information: Processing JEDEC for Logic Block 3.
   Information: Processing JEDEC for Logic Block 4.
   Information: JEDEC output file 'comp.jed' created.


Summary:
                    Error Count = 0       Warning Count = 0

Completed Successfully at 18:47:53
```

Figura 3.20. Generación del archivo JEDEC

3.1.6. Uso del programador ISR

Uno de los aspectos más importantes que hacen de un PLD un dispositivo dinámico es la habilidad de poderlo borrar y reprogramar sin necesidad de borradores ultravioleta. Los CPLDs se pueden borrar tanto electrónicamente como mediante la opción ISR *Reprogramabilidad En Sistema* (In System Reprogrammable), en donde la función es implementada desde la interface paralelo de la computadora hacia el conector tipo serial JTAG. Este programador nos permite borrar, verificar y/o leer uno varios dispositivos que empleen el estándar JTAG.

3.1.6.1. Características del programador ISR

El archivo que se utiliza para programar CPLDs se llama **isr.exe,** el cual entre otras funciones, convierte el archivo con extensión **.jed** generado por Warp, al archivo con extensión **.bit** que finalmente se grabará en el CPLD. Para programar una aplicación se recomiendan seguir los siguientes pasos:

1) Copiar el archivo **isr.exe** al directorio de nuestra preferencia, por ejemplo:

C: >md isr

C: > cd isr
C: \ISR> copy a: \isr.exe

2) Una vez que tengamos nuestro archivo **.jed** generado en Warp, necesitamos usar el editor de MS-DOS para generar un archivo con extensión **.dat.** Por ejemplo:

C: \ISR>edit config.dat

3) Dentro de este archivo se deben de llenar los siguientes campos:

nombre_dispositivo[opción][archivo.jed];

- El *nombre_ dispositivo* se refiere al circuito integrado en el que se quiere realizar la grabación.

- En el campo de opción se pueden declarar los siguientes parámetros:

p – para programar el CPLD con el archivo .JED
v – para verificar si lo que hay en el CPLD es el archivo .JED
r – para leer lo que hay en el CPLD y colocarlo en el archivo .JED
e - para borrar lo que hay en el CPLD

existen otras variantes que pueden ser consultadas en el manual de ISR [3].

- El campo *archivo.jed* es el lugar donde se declara el nombre del archivo (con extensión .jed) que se va a grabar.

4) Al terminar la edición, este archivo debe guardarse con el fin de que pueda continuarse con la siguiente secuencia:

- Conectar el programador al puerto paralelo de la PC
- Conectar el programador al dispositivo
- Polarizar el circuito CPLD

5) Finalmente en el símbolo del sistema se ejecuta el siguiente comando:

C:/ISR/ISR/d config.dat

Si no existen problemas en la grabación, el programa entregará un reporte en el que se indica que no existieron errores y que la aplicación ya se encuentra instalada dentro del dispositivo.

Capítulo 4

Síntesis de Lógica Combinacional y Secuencial

En este capítulo se diseña a través del lenguaje de descripción en hardware, los circuitos combinacionales y secuenciales más utilizados en el diseño lógico. Permitiéndonos en estas soluciones introducir nuevos conceptos, palabras reservadas, reglas, algoritmos, etc., que nos muestren la potencialidad y profundidad del lenguaje VHDL.

El desarrollo de cada una de las entidades de diseño descritas en este capítulo, se puede optimizar mediante el uso adecuado de las declaraciones secuenciales y/o concurrentes, utilizando en esta descripción cualesquiera de los tres diferentes tipos de arquitectura (funcional, estructural y por flujo de datos). Sin embargo y dada la filosofía teórico-práctica que queremos manejar en este texto, nos parece conveniente presentar soluciones que incluyan nuevas declaraciones, nuevos tipos de datos y nuevos algoritmos de análisis, es decir, no se pretende presentar la mejor opción de diseño para un problema, por el contrario, se propone brindar el mayor número de soluciones (modelos) que permitan al lector deducir y construir sus propias estrategias de diseño para optimizar sus resultados.

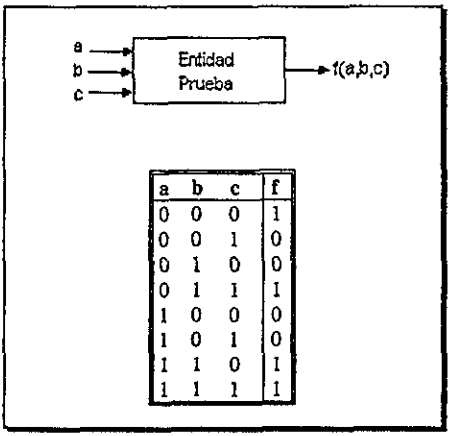
4.1. Programación de estructuras básicas mediante declaraciones concurrentes en VHDL

Como se mencionó anteriormente las declaraciones concurrentes se encuentran fuera de la declaración de un proceso, estas son generalmente utilizadas en las descripciones de flujo de datos y estructural, debido a que una declaración concurrente no sigue un orden de ejecución, es decir, no importa el orden en el que escriban las señales ya que el resultado para una determinada función sería el mismo. Existen tres tipos de declaraciones concurrentes:

- Declaraciones condicionales asignadas a una señal (**when-else**)
- Declaraciones concurrentes asignadas a señales
- Selección de una señal (**with-select-when**)

4.1.1. Declaraciones condicionales asignadas a una señal (when-else)

La declaración **when-else** se utiliza para asignar valores a una señal, determinando así la ejecución de una condición propia del diseño. Para ejemplificar, consideremos la entidad mostrada en la figura 4.1, cuyo funcionamiento se define en la tabla de verdad.



a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figura 4.1. Definición del funcionamiento de una entidad

La entidad se puede programar mediante el uso de *declaraciones condicionales* (**when-else**), debido a que este modelo permite definir paso a paso el comportamiento del sistema, listado 4.1.

```
1 -- Ejemplo combinacional básico
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity tabla is port(
5     a,b,c: in std_logic;
6         f: out std_logic);
7 end tabla;
8 architecture ejemplo of tabla is
9 begin
10    f <= '1' when (a='0' and b='0' and c='0') else
11         '1' when (a='0' and b='1' and c='1') else
12         '1' when (a='1' and b='1' and c='0') else
13         '1' when (a='1' and b='1' and c='1') else
14         '0';
15 end ejemplo;
```

Listado 1. Descripción de la entidad tabla

Observemos que la función de salida **f**, depende directamente de las condiciones que presentan las variables de entrada, además y dado que la ejecución inicial de una u otra condición no afecta la lógica del programa el resultado es el mismo. Es decir, la condición de entrada "111" puede ejecutarse antes que la condición "000".

La ventaja de la programación en VHDL en comparación con el diseño lógico puede intuirse considerando que la función de salida **f** mediante álgebra booleana se representa por:

$$f = /a\ /b\ /c + /a\ b\ c + a\ b\ /c + a\ b\ c$$

en el diseño convencional se utilizarían inversores, compuertas OR y compuertas AND; en VHDL la solución se realiza de manera directa utilizando la función lógica (AND). Por ejemplo las líneas 10 y 11 se interpretan como sigue:

asigna a "**f**" el valor de 1 cuando a = 0 y b = 0 y c = 0 sino
asigna a "**f**" el valor de 1 cuando a = 0 y b = 1 y c = 1 sino
.
.
.

Operadores Lógicos. Los operadores lógicos utilizados en la descripción con ecuaciones boolenas y definidos dentro de los diferentes tipos de datos `bit`, `boolean` o `std_logic` son los operadores: `and`, `or`, `nand`, `xor`, `xnor` y `not`. Las operaciones que se efectuen mediante el uso ellos (excepto `not`) se deben realizar con datos que tengan la misma longitud o palabra de bits.

Los operadores lógicos presentan el siguiente orden y prioridad al momento de ser compilados:

1) Expresiones entre paréntesis
2) Complementos
3) Función AND
4) Función OR

Las operaciones XOR y XNOR son transparentes al compilador y las interpreta mediante la suma de productos correspondiente a su función

Ejemplo:

Ecuación	En VHDL
q = a + x · y	q = a *or* (x *and* y)
$y = \overline{a + b \cdot /c + d}$	y = *not* (a *or* (b *and not* c) *or* d)

4.1.2. Declaraciones concurrentes asignadas a señales

En este tipo de declaración encontraremos las funciones de salida mediante la ecuación booleana que describe el comportamiento de cada una de las compuertas, figura 4.2.

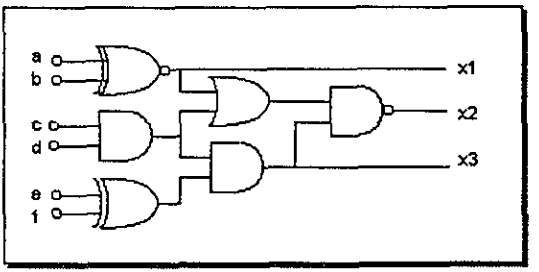


Figura 4.2. Circuito lógico implementado con compuertas

El programa correspondiente al circuito de la figura 4.2 se muestra a continuación, listado 2:

```
library ieee;
use ieee.std_logic_1164.all;
entity logic is port(
      a,b,c,d,e,f: in std_logic;
      x1,x2,x3:    out std_logic);
end logic;
architecture booleana of logic is
begin

     x1 <= a xnor b;
     x2 <= (((c and d)or(a xnor b)) nand
           ((e xor f)and(c and d)));
     x3 <= (e xnor f) and (c and d);

end booleana;
```

Listado 2. Declaraciones concurrentes asignadas a señales

4.1.3. Selección de una señal (with-select-when)

La declaración **with-select-when** es utilizada para asignar un valor a una señal, basado en el valor de otra señal que ha sido previamente seleccionada. Por ejemplo en la figura 4.3 se muestra el listado que presenta a este tipo de declaración. Como puede observarse el valor de la salida *c* depende de las señales de entrada seleccionadas (a(0) y a(1)), de acuerdo con la tabla de verdad correspondiente.

a(0)	a(1)	c
0	0	1
0	1	0
1	0	1
1	1	0

```
library ieee;
use ieee.std_logic_1164.all;
entity circuito is port(
     a: in std_logic_vector (1 downto 0);
     c: out std_logic);
end circuito;
architecture arq_cir of circuito is
begin
     with a select
       c <= '1' when "00",
            '0' when "01",
            '1' when "10",
            '0' when others;
end arq_cir;
```

Figura 4.3. Tabla de verdad y su listado correspondiente

4.2. Declaraciones Secuenciales

Como se mencionó anteriormente (sección 2.1.5.2), las declaraciones secuenciales son aquellas asignadas a señales en las que el orden que llevan puede tener un efecto significativo en la lógica descrita. Una declaración secuencial, a diferencia de una concurrente, debe ejecutarse en el orden en el cual aparece y además debe formar parte de un proceso (**process**).

Declaración if-then-else. Son declaraciones utilizadas para seleccionar una condición o condiciones basadas en el resultado de evaluaciones booleanas (falso o verdadero). Por ejemplo en la siguiente construcción,

if (condición) **then**
    haz algo;
else
    haz algo diferente;
end if;

si la condición especificada es evaluada como verdadera, entonces la declaración *haz algo* seguida de la palabra reservada **then** es ejecutada. Si la condición se evalúa como falsa, entonces la declaración *haz algo diferente* se ejecuta. La construcción se termina con **end if**.

Un ejemplo que ilustra este tipo de declaración es el mostrado en la figura 2.6. del capítulo anterior, el cual por comodidad se representa nuevamente en la figura 4.4.

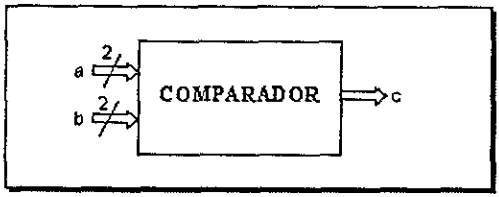


Figura 4.4. Comparador de igualdad de dos bits

El código correspondiente a esta entidad de diseño se muestra en el listado 3.

```
1  --Ejemplo de declaración de la entidad comparador
2entity comp is
3port (a,b: in bit_vector( 1 downto 0);
4        c: out bit);
5end comp;
6architecture funcional of comp is
7begin
8compara: process (a,b)
9begin
10          if a = b then
11              c <='1';
12          else
13              c<='0';
14      end if;
15      end process compara;
16      end funcional;
```

Listado 3. Declaración secuencial de un comparador de igualdad de dos bits

Cuando se requiere tener más condiciones de control se utiliza la estructura llamada **elsif (sino-si)**, la cual permite expandir y especificar prioridades dentro del proceso. La sintaxis para esta operación es:

if (condición1) **then**
  haz algo;
elsif (condición2) **then**
  haz algo diferente;
else
  haz algo completamente diferente;
end if;

la cual se interpreta como sigue: Si (**if**) la condición 1 es verdadera entonces (**then**) se ejecuta *haz algo*, sino-si (**elsif**) la condición 2 es verdadera entonces (**then**) se *ejecuta haz algo diferente*, sino (**else**) se ejecuta *haz algo completamente diferente.*

La forma de utilizar esta nueva declaración se ilustra en el diseño de un comparador de dos números de 4 bits, figura 4.5a. En este caso el sistema tiene tres salidas que indican cuando uno de los números es mayor, igual o menor que el otro. La figura 4.5b representa el mismo comparador de manera simplificada utilizando la notación vectorial:

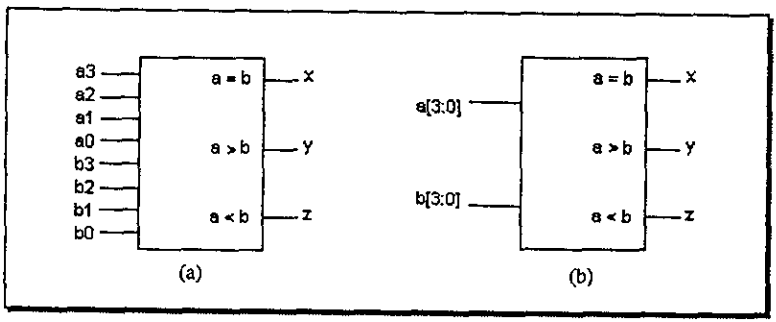


Figura 4.5. Comparador de 4 bits

En el listado 4 se observa el algoritmo en VHDL que describe el funcionamiento del comparador. En la línea 9 se muestra la lista sensitiva (a y b) del proceso (**process**). De la línea 10 a la 17 el proceso se desenvuelve mediante el análisis de las variables de la lista sensitiva. Sin mucho esfuerzo puede analizarse que si **a = b, entonces x toma el valor de 1**, de forma similar se intuye el comportamiento para a > b y a < b, incluyendo la declaración **elsif**.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity comp4 is port(
4 a,b:   in std_logic_vector(3 downto 0);
5 x,y,z: out std_logic);
6 end comp4;
7 architecture arq_comp4 of comp4 is
8 begin
9    process (a,b) begin
10         if (a = b) then
11           x <= '1';
12         elsif (a > b) then
13           y <= '1';
14         else
15         z <= '1';
16       end if;
17       end process;
18 end arq_comp4;
```

Listado 4. Descripción del comparador de 4 bits utilizando el estilo funcional

Operadores Relacionales. Los operadores relacionales son utilizados para evaluar la igualdad, desigualdad o la magnitud en una expresión. Los operadores de igualdad y desigualdad (= y /=) son definidos en todos los tipos de datos. Los operadores de magnitud (<,<=,> y >=) están definidos solo dentro del tipo escalar.

4.2.1. Buffers tri-estado

Los buffers tri-estado tienen diversas aplicaciones, ya sea como salidas de sistemas (**modo buffer**) o como parte integral de un circuito. En VHDL estos dispositivos son definidos a través de los valores que manejan (0,1 y alta impedancia 'Z'). En la figura 4.6 se observa el diagrama correspondiente a este circuito, y en el listado 5 el código que describe su funcionamiento.

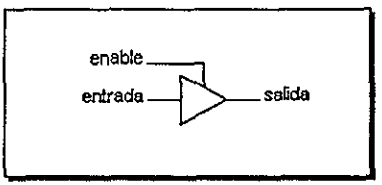


Figura 4.6. Buffer tri-estado

```
library ieee;
use ieee.std_logic_1164.all;
entity tri_est is port(
     enable, entrada: in std_logic;
                salida: out std_logic);
end tri_est;
architecture arq_buffer of tri_est is
begin
   process (enable, entrada) begin
      if enable = '0' then
         salida <= 'Z';
      else
         salida <= entrada;
      end if;
   end process;
end arq_buffer;
```

Listado 5. Descripción utilizando valores de alta impedancia

El listado anterior se basa en un proceso, el cual es utilizado para describir los valores que tomará la salida del buffer. En este proceso se indica que cuando el habilitador del circuito (*enable*) es afirmado, el valor que se encuentra a la entrada del circuito, es asignado a la

salida; si por el contrario *enable* no es afirmado, la salida del buffer tomará un valor de alta impedancia (Z). Este valor, al igual que 0 y 1, es soportado por el tipo `std_logic`, a esto se debe que en el diseño se prefiera utilizar el estándar `std_logic_1164` y no el tipo `bit`, ya que el primero es más versátil al proveer valores de alta impedancia y no importa (-)

4.2.2. Multiplexores

El diseño de multiplexores se puede realizar describiendo su comportamiento mediante la declaración **with-select-when** o a través de ecuaciones booleanas.

En la figura 4.7a. se observa que el multiplexor tiene como entrada de datos las variables a,b,c y d cada una de ellas representadas por dos bits (a0,a1), b(b0,b1), etc., las líneas de selección (s) de dos bits (s0 y s1) y la línea de salida z (z0 y z1).

En la figura 4.7b se muestra un diagrama simplificado que resalta la representación mediante vectores de bits.

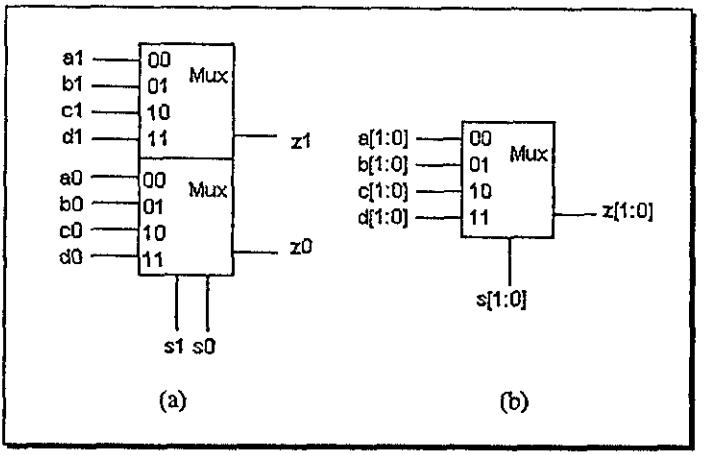


Figura 4.7. Diseño de multiplexores

En el listado 6 se muestra la descripción mediante **with-select-when** de un multiplexor dual de 2 x 4. En este caso la señal *s* determina cual de las cuatro señales es asignada a la salida *z*. Los valores de *s* se encuentran asignados como "00", "01" y "10", el término **others** (otros) especiffica cualquier combinación adicional que pudiera presentarse (que incluye el "11"), la cual puede adoptar nueve posibles valores, que son interpretados por la herramienta de síntesis como **tipos lógicos estándar.**

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is port(
     a,b,c,d: in std_logic_vector(1 downto 0);
           s: in std_logic_vector(1 downto 0);
           z: out std_logic_vector(1downto 0);
end mux;

architecture arqmux4 of mux is
begin
with s select
     z <=  a when "00",
           b when "01",
           c when "10",
           d when others;
end arqmux4;
```

Listado 6. Multiplexor descrito con declaraciones with-select-when

Tipos lógicos estándar. Con el objeto de estandarizar las herramientas de síntesis para VHDL, fue creado el paquete estándar IEEE 1076.3 "*Standard VHDL Synthesis Packages, IEEE 1076.3*". Dentro de él están considerados los tipos lógicos estándar que se muestran a continuación:

type std_ulogic **is** (

'U' – Valor no inicializado
'X' – Valor fuerte desconocido
'0' – 0 Fuerte
'1' – 1 Fuerte
'Z' – Alta impedancia
'W' – Valor débil desconocido
'L' – 0 débil
'H' – 1 débil
'-' – No importa (don't care));

Existen valores como por ejemplo L y H en donde el estándar no especifica ninguna interpretación para ellos, debido a que la mayoría de las herramientas no los soportan. Los valores metalógicos ('U', 'W', 'X', '-') carecen de sentido en la síntesis, pero son utilizados por la herramienta en la simulación del código. Como se mencionó en la sección anterior, el uso de 'Z' implica un valor de alta impedancia.

4.2.3. Descripción de multiplexores utilizando ecuaciones booleanas.

En el listado 7 se muestra una solución mediante ecuaciones booleanas para el multiplexor dual de la figura 4.7.

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is port(
      a,b,c,d: in std_logic_vector(1 downto 0);
      s:       in std_logic_vector(1 downto 0);
      z:       out std_logic_vector(1 downto 0));
end mux;

architecture arqmux of mux is
begin


      z(1) <=(a(1) and not(s(1)) and not(s(0))) or
             (b(1) and not(s(1)) and s(0)) or
             (c(1) and s(1) and not(s(0))) or
             (d(1) and s(1) and s(0));

       z(0)<=(a(0) and not(s(1)) and not(s(0))) or
             (b(0) and not(s(1)) and s(0)) or
             (c(0) and s(1) and not(s(0))) or
             (d(0) and s(1) and s(0));

end arqmux;
```

Listado 7. Multiplexor descrito con ecuaciones booleanas

4.2.4. Sumadores

El desarrollo de un sumador de cuatro bits utilizando compuertas lógicas es un buen ejemplo para reafirmar el manejo de señales (signal). En la figura 4.8 se observa como los acarreos de salida (C0,C1 y C2) se encuentran retroalimentados dentro del circuito, por lo que no tienen un pin externo asignado. En el listado 8 se muestra la programación en VHDL.

Como puede observarse el rango utilizado dentro de signal es (2 to 0) debido a que solo se retroalimentan los acarreos C0, C1 y C2. Por otro lado con un poco de esfuerzo puede intuirse que cada uno de los diferentes bloques que forma el sumador se encuentra caracterizado utilizando compuertas *xor* (or exclusiva).

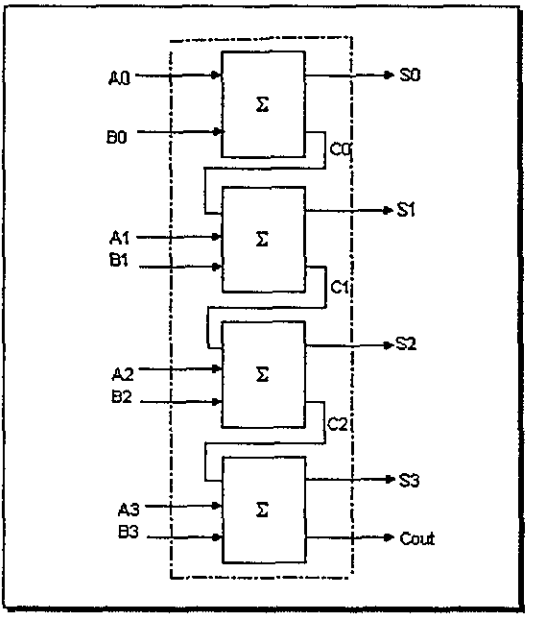


Figura 4.8. Sumador de 4 bits

```
library ieee;
use ieee.std_logic_1164.all;
entity suma is port(
    a,b: in std_logic_vector (0 to 3);
      s: out std_logic_vector (0 to 3);
     Cout: out std_logic);
end suma;
architecture arqsuma of suma is
 signal c: std_logic_vector(0 to 2);
  begin
       s(0) <= a(0) xor b(0);
       c(0) <= a(0) and b(0);
       s(1) <= (a(1) xor b(1)) xor c(0);
       c(1) <= (a(1) and b(1)) or (c(0)and(a(1)xor b(1)));
       s(2) <= (a(2) xor b(2)) xor c(1);
       c(2) <= (a(2) and b(2)) or (c(1)and(a(2)xor b(2)));
       s(3) <= (a(3) xor b(3)) xor c(2);
       Cout <= (a(3) and b(3)) or (c(2)and(a(3)xor b(3)));

end arqsuma;
```

Listado 8. Descripción de un sumador de 4 bits

Operadores Aritméticos. Como su nombre lo indica, los operadores aritméticos permiten realizar operaciones del tipo aritmético, tales como suma, resta, multiplicación, división, cambios de signo, valor absoluto y concatenación. Estos operadores son generalmente utilizados en el diseño lógico para la descripción de sumadores y restadores o para las operaciones de incremento y decremento de datos. En la tabla 1, se muestran los operadores aritméticos predefinidos en VHDL:

Operador	Descripción
+	Suma
-	Resta
/	División
*	Multiplicación
**	Potencia

Tabla 1. Operadores Aritméticos utilizados en VHDL

Como un ejemplo, analicemos el diseño de un circuito sumador de 4 bits que no considera el acarreo de salida, listado 9.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity sum is port(
    a,b:  in std_logic_vector(3 downto 0);
    suma: out std_logic_vector(3 downto 0));
end sum;
architecture arqsum of sum is
begin
    suma <= a + b;
end arqsum;
```

Listado 9. Descripción de un sumador utilizando std_logic_vector y el paquete std_arith

En el listado 9, se introdujo el paquete `std_arith`, el cual como ya se ha mencionado, se encuentra dentro de la librería de trabajo (**work**). Este paquete permite el uso de los operadores aritméticos con operaciones realizadas entre arreglos del tipo `std_logic_vector`. Es decir, debido a que dentro del paquete estándar (`std_logic_1164`) no se encuentran definidos los operadores aritméticos es necesario hacer uso del paquete std_arith.

El uso de los operadores existentes en VHDL, así como los tipos de datos para los cuales se encuentran definidos, se incluyen dentro del apéndice B.

4.2.5. Decodificadores

La programación de circuitos decodificadores, se basa en el uso de declaraciones que permitan establecer la relación que existe entre un código binario aplicado a las entradas del dispositivo y el nivel de salida obtenido.

En esta sección se presentan dos tipos de decodificadores, el decodificador BCD-decimal y el decodificador de BCD a siete segmentos, ya que consideramos que son dos de los más utilizados dentro diseño lógico combinacional.

4.2.5.1. Decodificador BCD a decimal

En la figura 4.9 podemos observar la entidad de diseño correspondiente a un circuito decodificador, el cual convierte código BCD (código de binario a decimal) en uno de los diez dígitos decimales. Generalmente estos dispositivos son llamados decodificadores de 4 a 10 líneas debido a que contienen 4 líneas de entrada y 10 de salida.

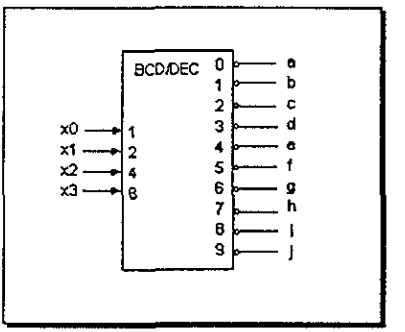


Figura 4.9. Decodificador de BCD a decimal

El programa que describe el comportamiento de la entidad de la figura 4.9 es el que se muestra en el listado 10.

Como se puede observar, el código correspondiente a este circuito se basa en la ejecución de un proceso, dentro del cual se establecen las condiciones que serán evaluadas para que cada salida sea activada de acuerdo con al valor binario correspondiente. Para fines prácticos, se asignó a cada salida un nombre con el cual pudiera ser identificada fácilmente.

Como ejemplo consideremos el valor de la entrada $x = 0010$, la cual corresponde al dígito decimal 2; veamos como la condición que determina la asignación del valor, evalúa primero la condición de x y si ésta es afirmada, entonces asigna a la salida c el valor correspondiente del dígito decimal 2.

Por otro lado, se puede ver que al inicio del proceso se declararon todas las salidas con un valor inicial de '1', esto se realizó con el fin de asegurar que permanecieran desactivadas cuando no estén siendo evaluadas.

```
--Decodificador de BCD a decimal
library ieee;
use ieee.std_logic_1164.all;
entity deco is port (
 x: in std_logic_vector(3 downto 0);
      a,b,c,d,e,f,g: out std_logic);
end deco;
architecture arqdeco of deco is
begin

   process (x) begin
     a <= '1';
     b <= '1';
     c <= '1';
     d <= '1';
     e <= '1';
     f <= '1';
     g <= '1';
     h <= '1';
     i <= '1';
     j <= '1';

   if  (x = "0000") then
        a <= '0';
   elsif x = ("0001") then
        b <= '0';
   elsif x = ("0010") then
        c <= '0';
   elsif x = ("0011") then
        d <= '0';
   elsif x = ("0100") then
        e <= '0';
   elsif x = ("0101") then
        f <= '0';
   elsif x = ("0101") then
        g <= '0';
   elsif x = ("0111") then
        h <= '0';
   elsif x = ("1000") then
        i <= '0';
   else
        j <= '0';
   end if;
  end process;
end arqdeco;
```

Listado 10. Descripción de un decodificador de BCD a decimal

4.2.5.1. Decodificador de BCD a display de 7 segmentos

En la figura 4.10a se muestra un circuito decodificador, el cual acepta código BCD en sus entradas y proporciona salidas capaces de excitar un display de siete segmentos que indica el dígito decimal seleccionado. En la figura 4.10b se observa la distribución de los segmentos dentro del display.

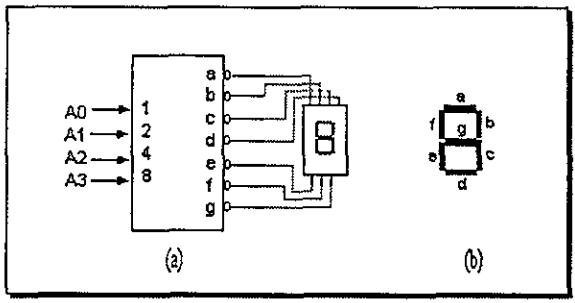


Figura 4.10. Decodificador BCD a 7 segmentos

Como se puede apreciar, la entidad del decodificador cuenta con una entrada llamada *A* formada por cuatro bits (A0,A1,A2,A3) y siete salidas (a,b,c,d,e,f,g) activas en nivel bajo que corresponden a los segmentos del display. En la siguiente tabla (tabla 2) se indican los valores lógicos de salida correspondientes a cada segmento.

Código BCD			Segmento del display						
			a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	1
0	0	1	1	0	0	1	1	1	1
0	1	0	0	0	1	0	0	1	0
0	1	1	0	0	0	0	1	1	0
1	0	0	1	0	0	1	1	0	0
1	0	1	0	1	0	0	1	0	0
1	1	0	0	1	0	0	0	0	0
1	1	1	0	0	0	1	1	1	0

Tabla 2. Valores lógicos correspondientes a cada segmento del display

La función del programa cuyo código se despliega en el listado 11, se basa en declaraciones del tipo **case-when**, las cuales describen de que manera el decodificador es manejado de acuerdo al valor que toma la entrada A. Nótese que para fines prácticos, se declararon todas las salidas como un solo vector de bits (identificado como d), de esta forma se entiende que la salida *a* corresponde al valor *d0*, la *b* al valor d1 y así sucesivamente. Por otro lado, la palabra reservada **others** como se indicó anteriormente, define el comportamiento de la salida *d* para todos los posibles valores de *A*.

```
library ieee;
use ieee.std_logic_1164.all;
entity deco is port (
     A: in std_logic_vector(2 downto 0);
     d: out std_logic_vector(6 downto 0));
end deco;
architecture arqdeco of deco is
begin
    process (a) begin
     case A is
      when "000" => d <= "0000001";
      when "001" => d <= "1001111";
      when "010" => d <= "0010010";
      when "011" => d <= "0000110";
      when "100" => d <= "1001100";
      when "101" => d <= "0100100";
      when "110" => d <= "0100000";
      when "111" => d <= "0001110";
      when others =>  d <= "1111111";
    end case;
  end process;
end arqdeco;
```

Listado 11. Uso de declaraciones case-when

4.2.6. Codificadores

En esta sección se muestra la forma de programar un circuito codificador, el cual como se observa en la figura 4.11, posee 10 entradas (cada una correspondiente a un dígito decimal) y cuatro salidas para el código binario de 4 bits BCD.

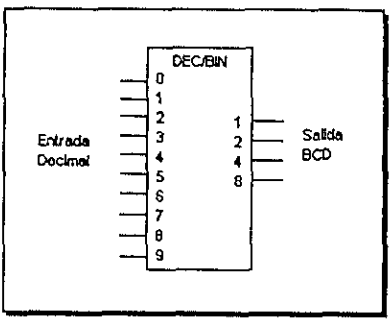


Figura 4.11. Codificador de decimal a BCD

El programa que describe al circuito de la figura 4.11, se muestra en el listado 12. Como se puede ver, el puerto de entrada *a* se declara como de tipo entero (**integer**) que tiene un

valor en el rango (**range**) del 0 al 9 inclusive (debido a que son diez los dígitos que se usarán), permitiéndonos con esto manejar los datos directamente de la forma decimal. Por otro lado, el puerto de salida es un arreglo del tipo `std_logic_vector`, lo cual nos permite seleccionar un dato en forma decimal y obtener a la salida su equivalente binario. Al igual que en el listado anterior, el funcionamiento del circuito está basado en un proceso, el cual se ejecuta a través de declaraciones secuenciales.

```
-- Codificador de decimal a BCD
library ieee;
use ieee.std_logic_1164.all;
entity codif is port (
     a: in integer range 0 to 9;
    d: out std_logic_vector(3 downto 0));
end codif;
architecture arqcodif of codif is
begin
   process (a) begin

     If a = 0 then
        d <= "0000";
     elsif a = 1 then
        d <= "0001";
     elsif a = 2 then
        d <= "0010";
     elsif a = 3 then
        d <= "0011";
     elsif a = 4 then
        d <= "0100";
     elsif a = 5 then
        d <= "0101";
     elsif a = 6 then
        d <= "0110";
     elsif a = 7 then
        d <= "0111";
     elsif a = 8 then
        d <= "1000";
     else
        d <= "1001";
     end if;
  end process;
end arqcodif;
```

Listado 12. Descripción de un codificador usando enteros y vectores

Enteros (integer). El tipo **integer** es utilizado en VHDL para representar números enteros con o sin signo. Los valores soportados por el lenguaje se encuentran definidos en el rango de –2,147-483,647 (-2^{31}-1) hasta 2,147,483,647 (2^{31}-1).

Un rango (**range**) es una palabra reservada por VHDL utilizada para definir el conjunto de valores que el entero tomará. Con las palabras reservadas **to** y **downto** se puede especificar el rango creciente o decreciente respectivamente, de tal forma que pueda establecerse el orden en que serán ejecutados los datos.

4.3. Diseño lógico secuencial con VHDL

Como se sabe un sistema secuencial esta compuesto por un sistema combinacional y un elemento de memoria conocido como flip-flop. Este elemento es fundamental en el diseño y funcionamiento de los registros y contadores por lo cual iniciaremos el diseño y desarrollo de este tema ejemplificando la programación de un flip-flop.

4.3.1. Flip-Flops

Para iniciar el análisis de lógica secuencial, consideremos el flip-flop tipo D mostrado en la figura 4.12. Como se observa este circuito cuenta con dos pines de entrada (la entrada d y la entrada de reloj *clk*), y una salida representada por q.

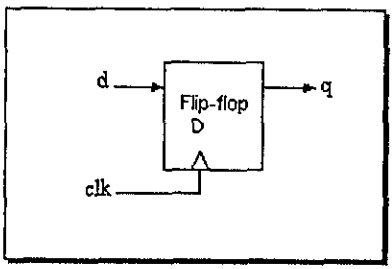


Figura 4.12. Entidad de un flip-flop tipo D

Debido a la facilidad para implementar funciones secuenciales, las declaraciones **if-then-else** son las más utilizadas. En el programa mostrado en el listado 13, se aprecia el uso estas declaraciones dentro del proceso, así como también de la variable sensitiva **clk**, la cual es la encargada de sincronizar los cambios de estado en el circuito.

La ejecución del proceso es sensible a los cambios en *clk* (pulso de reloj). Esto es, cuando *clk* cambia de valor de una transición de 0 a 1 (clk='1') el valor de *d* es asignado a *q*, y se mantiene hasta que se genere un nuevo pulso. De manera inversa, si *clk* no presenta dicha transición, entonces el valor de *q* se mantiene igual. Esto puede observarse de manera clara en la figura 4.13.

```
library ieee;
use ieee.std_logic_1164.all;
entity ffd is port(
 d,clk: in std_logic;
 q:     out std_logic);
end ffd;
architecture arq_ffd of ffd is
begin
   process (clk) begin
         if (clk'event and clk='1') then
             q <= d;
         end if;
   end process;
end arq_ffd;
```

Listado 13. Descripción de un flip-flop disparado por flanco positivo

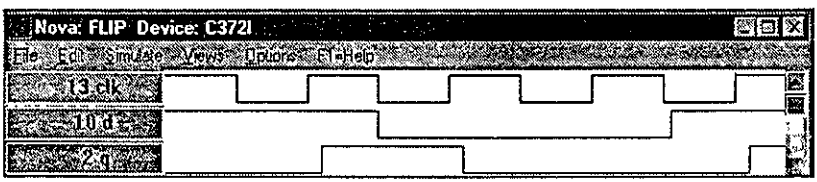


Figura 4.12. Simulación del flip-flop D

Notemos que la salida *q* toma el valor de la entrada *d*, solo cuando la transición del pulso de reloj es de '0' a '1' y se mantiene hasta que se ejecuta nuevamente el cambio de valor de *clk*

Atributo 'event. Los atributos son utilizados dentro del lenguaje VHDL para definir características que pueden ser asociadas a cualquier tipo de datos, objeto o entidades. El atributo **'event**[9] (evento) es utilizado para describir un evento u ocurrencia de una señal en particular.

Retomando el código mostrado en el listado 13, observemos que la condición **if clk'event** es cierta solo cuando ocurre un cambio de valor, es decir un **evento (event)** de la señal *clk*. Como se puede apreciar, la declaración **if-then** no maneja la condición **else**, debido a que el compilador mantiene el valor de *q* hasta que no exista un cambio de valor en la señal clk.

Para mayor información de los atributos predefinidos en VHDL consúltese el apéndice A.

[9] El apóstrofe ' indica que se trata de un atributo

4.3.2. Registros

En la figura 4.14 se presenta la estructura de un registro de 8 bits con entrada y salida de datos en paralelo. El diseño es muy similar al flip-flop anterior, la diferencia radica en la utilización de vectores de bits en lugar del bit individual.

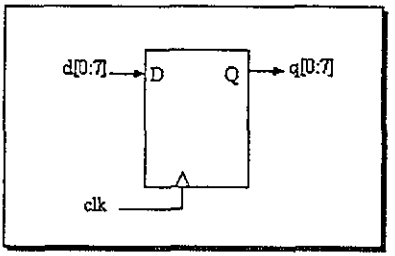


Figura 4.14. Registro paralelo de 8 bits

El código correspondiente a esta entidad de diseño se muestra en el listado 14.

```
library ieee;
use ieee.std_logic_1164.all;
entity reg is port(
      d:   in std_logic_vector(0 to 7);
      clk: in std_logic;
      q:   out std_logic_vector(0 to 7));
end reg;
architecture arqreg of reg is
begin
   process (clk) begin
   if (clk'event and clk='1') then
       q <= d;
   end if;
 end process;
end arqreg;
```

Listado 14. Código VHDL de un registro de 8 bits

Al igual que el listado 13, la descripción presentada en el código del registro maneja el atributo **'event** encargado de la ejecución del proceso.

4.3.3. Contadores

Los contadores son entidades muy utilizadas en el diseño lógico. La forma usual para describirlos en VHDL es mediante operaciones de incremento y/o decremento de datos.

Como ejemplo veamos la figura 4.15 que representa un contador ascendente de 4 bits, así como el diagrama de tiempos que indica su funcionamiento.

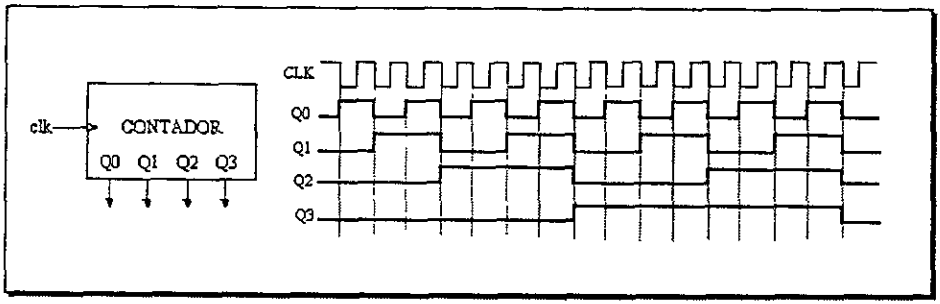


Figura 4.15. Contador binario de cuatro bits

Cabe mencionar que la presentación del diagrama de tiempos de este circuito, tiene la finalidad de ilustrar el procedimiento seguido en la programación, ya que puede observarse claramente el incremento que presentan las salidas cuando un pulso de reloj es aplicado a la entrada, listado 15.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity cont4 is port(
 clk: in std_logic;
   q: inout std_logic_vector(3 downto 0));
end cont4;
architecture arqcont of cont4 is
begin
    process (clk)
       begin
         If (clk'event and clk = '1') then
             q <= q + 1;
         end if;
     end process;
end arqcont;
```

Listado 15. Código que describe un contador de 4 bits

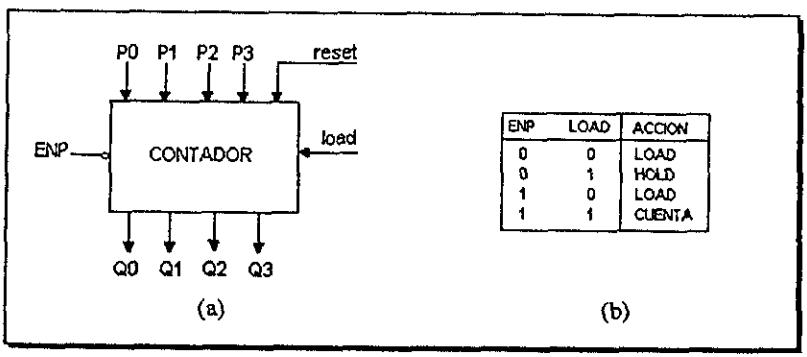
Cuando requerimos que una señal sea retroalimentada (`q <= q+1`) ya sea dentro o fuera de la entidad, utilizamos el modo **inout** (sección 2.1.1.2). En nuestro caso el puerto correspondiente a *q* se maneja como tal, debido a que la señal es retroalimentada en cada pulso de reloj. Notemos también el uso del paquete `std_arith`, el cual como se mencionó anteriormente permite utilizar el operador "+" con el tipo `std_logic_vector`.

El funcionamiento del contador se define básicamente en un proceso, en el cual se llevan a cabo los eventos que determinan el comportamiento del circuito. Al igual que en los otros programas, una transición de 0 a 1 efectuada por el pulso de reloj, provoca que el proceso sea ejecutado, incrementando en 1 el valor asignado a la variable *q*. Cuando esta salida tiene el valor de 15 ("1111") y si el pulso de reloj sigue siendo aplicado, el programa empieza a contar nuevamente de 0.

4.3.3.1. Contador con reset y carga en paralelo (load)

La entidad de diseño mostrada en la figura 4.16a es un ejemplo de un circuito contador síncrono de 4 bits. Este contador tiene varias características adicionales con respecto al anterior.

Su funcionamiento se encuentra predeterminado en la tabla mostrada en la figura 4.16b. como se puede observar este contador tiene las entradas de control *enable* y *load* y dependiendo del valor lógico que tengan en sus terminales realizarán cualesquiera de las operaciones mostradas en la tabla. Véase que la señal de *reset* se activa en nivel alto de forma asíncrona; finalmente el circuito tiene cuatro entradas en paralelo declaradas como p0, p1, p2,p3.



ENP	LOAD	ACCION
0	0	LOAD
0	1	HOLD
1	0	LOAD
1	1	CUENTA

Figura 4.16. (a) Contador binario de cuatro bits; (b) Tabla de funcionamiento

El propósito de describir este ejemplo radica en el uso de varias condiciones, de tal manera que una instrucción no puede ser evaluada si no se cumple la o las condiciones predeterminadas. Por ejemplo, observemos el listado 14, en donde la línea 11 muestra la lista

sensitiva de las variables involucradas en el proceso (*clk, enp, load, reset*) nótese que no importa el orden en el cual están declaradas. En la línea 12 se indica que si *reset = 1* entonces las salidas *q* toman el valor de 0, pero si *reset* no es igual a 1 entonces pueden evaluarse las siguientes condiciones (es importante resaltar que la primera condición debe de ser *reset*). En la línea 15 se observa el proceso de habilitar una carga en paralelo por lo que si *load = 0* y *enp* es una condición de no importa ('-') que puede tomar el valor de 0 o 1 según la tabla 4.16b, las salidas *q* adoptarán el valor que este presente en las entradas *p* (p3,p2,p1,p0). Este ejemplo muestra de forma didáctica el uso de la declaración **elsif** (sino-si).

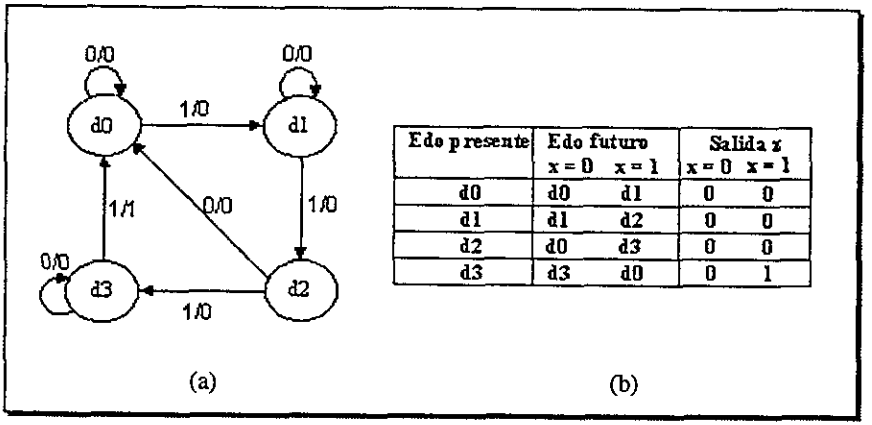
```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use work.std_arith.all;
4 entity cont is port(
5    p: in std_logic_vector(3 downto 0);
6    clk,load,enp,reset: in std_logic;
7    q: inout std_logic_vector(3 downto 0));
8 end cont;
9  architecture arq_cont of cont is
10 begin
11   process (clk, reset,load, enp) begin
12     if (reset = '1') then
13        q <= "0000";
14     elsif (clk'event and clk = '1') then
15              if  (load = '0' and enp = '-') then
16                 q <= p;
17              elsif (load = '1' and enp = '0') then
18                  q <= q;
19              elsif (load = '1' and enp = '1') then
20                 q <= q + 1;
21              end if;
22     end if;
23   end process;
24 end arq_cont;
```

Listado 16. Contador con reset, enable y carga en paralelo

4.3.4. Diagramas de estado

El uso de diagramas de estados en la lógica programable facilita de manera significativa la descripción de un diseño secuencial, ya que no es necesario seguir la metodología tradicional de diseño. En VHDL se puede utilizar un modelo funcional, en el cual únicamente se indica la transición que siguen los estados y las condiciones que controlarán el proceso.

Para ilustrar lo anterior, consideremos el diagrama de estados mostrado en la figura 4.17(a) en donde se observa que el sistema tienen una señal de entrada denominada x y una señal de salida z. En la figura 4.17b se muestra la tabla de estados que describe el comportamiento del circuito.



Edo presente	Edo futuro x = 0	Edo futuro x = 1	Salida z x = 0	Salida z x = 1
d0	d0	d1	0	0
d1	d1	d2	0	0
d2	d0	d3	0	0
d3	d3	d0	0	1

(b)

Figura 4.16. Diagrama de Estados

Este diagrama puede ser codificado fácilmente utilizando una descripción de alto nivel en VHDL. Esta descripción consiste en el uso de declaraciones **case-when**, las cuales determinan en un caso particular, el valor que tomará el siguiente estado. Por otro lado, la transición entre estados se realiza por medio de declaraciones **if-then-else**, de tal forma que ellas son las encargadas de establecer la lógica que seguirá el programa para poder realizar la asignación del estado.

Como primer paso en nuestro diseño consideremos los estados d0,d1,d2 y d3, los cuales para poder ser representados en código VHDL, deben definirse dentro de un tipo de datos enumerado[10] (apéndice A) utilizando la declaración **type**. Observemos como son listados los identificadores de los estados, así como las señales utilizadas para el estado actual (`edo_presente`) y el siguiente (`edo_futuro`):

```
type estados is (d0, d1, d2,d3) ;
signal edo_presente, edo_futuro : estados;
```

[10] Se llaman tipos enumerados porque en ellos se listan o enumeran todos y cada uno de los valores que forman el tipo.

El siguiente paso consiste en la declaración del proceso que definirá el comportamiento del sistema. En él debe considerarse que el `edo_futuro` depende del valor del `edo_ presente` y de la entrada x. De esta manera la lista sensitiva del proceso, quedaría de la siguiente forma:

proceso1: **process** (edo_presente, x)

Es precisamente dentro del proceso donde se describe la transición del `edo_presente` al `edo_futuro`. Primero se inicia con la declaración **case** que especifica el primer estado a evaluar, en nuestro caso, consideremos que el análisis inicia en el estado d0 (**when** d0), en donde la salida z siempre es cero sin importar el valor de x. Si la entrada x es igual a 1 entonces el estado futuro es d1, y en caso contrario el estado futuro es d0.

De esta forma la declaración del proceso quedaría de la siguiente manera:

```
proceso1: process (edo_presente, x) begin
   case edo_presente is
     when d0 => z <= '0';
      if x ='1' then
        edo_futuro <= d1;
      else
        edo_futuro <= d0;
      end if;
```

Nótese que en cada estado debe indicarse el valor de la salida (z <= '0') después de la condición **when**, siempre y cuando la variable z no cambie de valor.

En el listado 17 se muestra la definición completa del código explicado anteriormente. Como podemos observar en el programa se utilizan dos procesos, en el primero, proceso1 (línea 11) se describe la transición que sufren los estados y las condiciones necesarias que determinan dicha transición. Por otro lado, en el segundo proceso2 (línea 42) se lleva a cabo de manera síncrona la asignación del estado futuro al estado presente, de tal forma que cuando un pulso de reloj es aplicado, el proceso se ejecuta.

En la línea 31 se describe la forma de programar la salida z en el estado d3 cuando esta obtiene el valor de 0 o 1 dependiendo del valor de la entrada x.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity diagrama is port(
4  clk,x:  in std_logic;
5       z:  out std_logic);
6 end diagrama;
7 architecture arq_diagrama of diagrama is
8   type estados is (d0, d1, d2, d3);
9   signal edo_presente, edo_futuro: estados;
10   begin
11    proceso1: process (edo_presente, x) begin
12     case edo_presente is
13        when d0 =>  z <= '0';
14          if x ='1' then
15             edo_futuro <= d1;
16          else
17             edo_futuro <= d0;
18          end if;
19        when d1 =>  z <='0';
20          if x='1' then
21             edo_futuro <= d2;
22          else
23             edo_futuro <= d1;
24          end if;
25        when d2 =>  z <='0';
26          if x='1' then
27             edo_futuro <= d3;
28          else
29             edo_futuro <= d0;
30          end if;
31        when d3 =>
32          if x='1' then
33             edo_futuro <= d0;
34               z <='1';
35        else
36             edo_futuro <= d3;
37               z <= '0';
38         end if;
39    end case;
40 end process proceso1;
41
42  proceso2: process(clk) begin
43   if (clk'event and clk='1') then
44       edo_presente <= edo_futuro;
45   end if;
46  end process proceso2;
47 end arq_diagrama;
```

Listado 17. Diseño de un diagrama de estados

4.3.5. Diseño de algoritmos de controladores digitales

En la figura 4.17a, se muestra la carta ASM (algoritmo de la máquina de estado) que representa al algoritmo de control de una máquina despachadora de refrescos. Como puede observarse el algoritmo está formado por siete estados designados mediante las letras A,B,C,D,E,F,G,H. Los rombos utilizados en el diagrama especifican las variables de entrada, mientras que las salidas se encuentran declaradas dentro del bloque o rectángulo de estado.

En la figura 4.17b se tiene esta misma representación mediante el formato de Mealy o diagrama de estados, en donde los rectángulos se han sustituido por los estados, los rombos por las líneas de interconexión que unen a cada estado y las salidas se encuentran indicadas mediante la simbología ↑↓ En el listado 18 se muestra el código de programación en VHDL para este diseño.

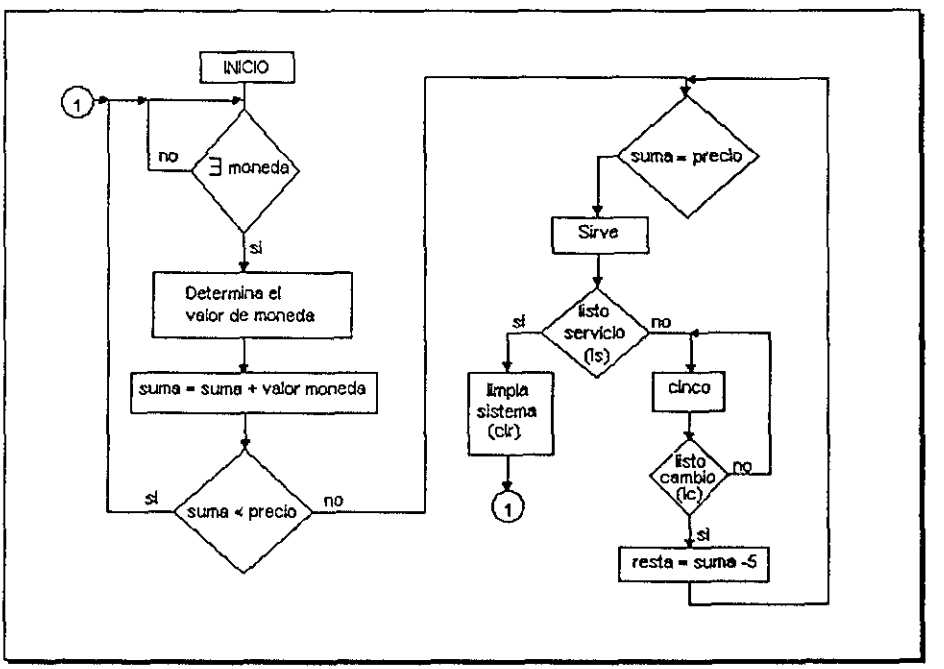


Figura 4.17a. Carta ASM de una máquina despachadora de refrescos

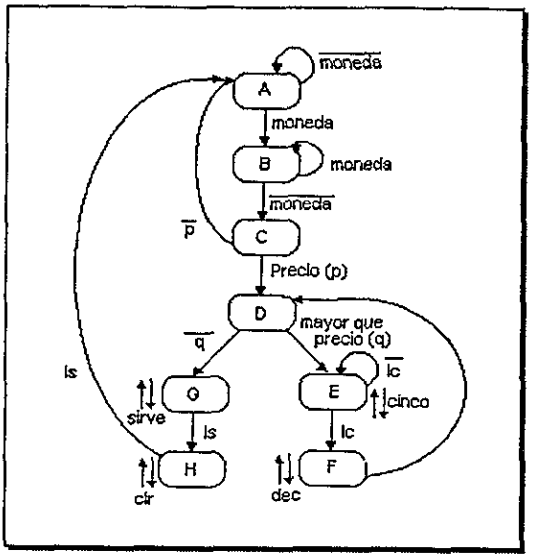


Figura 4.17b. Diagrama de estados de la máquina de refrescos

4.3.6. Integración de entidades

El siguiente ejemplo tiene como finalidad mostrar de manera sintetizada el diseño la interconexión de dos unidades de diseño dentro de una entidad. En la figura 4.18, se muestra un contador que permite visualizar mediante un display de siete segmentos el conteo del 0 al 9. Nótese que en esta parte se integran el contador y un decodificador de BCD a siete segmentos.

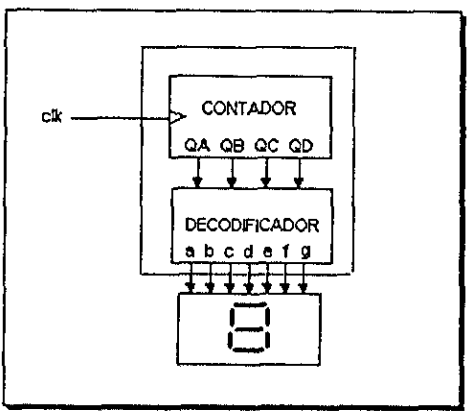


Figura 4.18. Ejemplo de integración de entidades

```
library ieee;
use ieee.std_logic_1164.all;
entity maquina is port(
  clk, moneda, P, Q, LC, LS:  in std_logic;
  clr, sirve, cinco, dec:  out std_logic);
end maquina;

architecture arq_maq of maquina is
   type estados is (A,B,C,D,E,F,G,H);
   signal edo_pres, edo_fut: estados;
 begin
   p_estados: process (edo_pres,moneda,P,Q,LC,LS) begin
     case edo_pres is
        when A => clr <= '0'; sirve <= '0'; cinco <= '0'; dec <= '0';
          if moneda ='1' then
             edo_fut <= B;
          else
             edo_fut <= A;
          end if;
        when B => clr <= '0'; sirve <= '0'; cinco <= '0'; dec <= '0';
          if moneda ='0' then
             edo_fut <= C;
          else
             edo_fut <= B;
          end if;
        when C => clr <= '0'; sirve <= '0'; cinco <= '0'; dec <= '0';
          if P ='0' then
             edo_fut <= D;
          else
             edo_fut <= A;
          end if;
        when D => clr <= '0'; sirve <= '0'; cinco <= '0'; dec <= '0';
          if Q ='0' then
             edo_fut <= G;
          else
             edo_fut <= E;
          end if;
        when G => clr <= '0'; sirve <= '1'; cinco <= '0'; dec <= '0';
             edo_fut <= H;
        when H => clr <= '1'; sirve <= '0'; cinco <= '0'; dec <= '0';
             edo_fut <= A;
        when E => clr <= '0'; sirve <= '0'; cinco <= '1'; dec <= '0';
          if LC ='1' then
             edo_fut <= F;
          else
             edo_fut <= E;
          end if;
        when F => clr <= '0'; sirve <= '0'; cinco <= '0'; dec <= '1';
             edo_fut <= D;
     end case;
  end process p_estados;

   p_reloj: process(clk) begin
    if (clk'event and clk='1') then
        edo_pres <= edo_fut;
    end if;
   end process p_reloj;
end arq_maq;
```

Listado 18. Programa que describe el funcionamiento de la máquina de refrescos

Ahora observemos el programa correspondiente a la entidad de diseño de la figura 4.18. Cabe mencionar, que los circuitos contador y decodificador son programados como una sola entidad, cuyo funcionamiento se basa en el comportamiento general del sistema. Esto es, cuando el contador realice el conteo de 0, el display automáticamente muestra el número 0, cuando sea 1, el display encenderá sus segmentos b y c correspondientes al dígito 1, y así sucesivamente hasta contar al 9. Una vez que sea desplegado este número, se aplica un reset al contador y comienza a funcionar nuevamente de cero. Listado 19.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity display is port(
 clk,reset: in std_logic;
 d: inout std_logic_vector(6 downto 0);
 q: inout std_logic_vector(3 downto 0));
end display;
architecture arqdisplay of display is
begin
    process (clk,reset)
       begin
         If (clk'event and clk = '1') then
              q <= q + 1;
                 if (reset = '1' or q = "1001") then
                    q <= "0000";
                 end if;
         end if;
     end process;

    process (q) begin
    case q is
       when "0000" => d <= "0000001";
       when "0001" => d <= "1001111";
       when "0010" => d <= "0010010";
       when "0011" => d <= "0000110";
       when "0100" => d <= "1001100";
       when "0101" => d <= "0100100";
       when "0110" => d <= "0100000";
       when "0111" => d <= "0001110";
       when "1000" => d <= "1111111";
       when "1001" => d <= "1111011";
       when others => d <= "1111111";
    end case;
  end process;
end arqdisplay;
```

Listado 19. Integración de entidades de diseño

Capítulo 5

Diseño Jerárquico En VHDL

El diseño jerárquico es una herramienta de apoyo, que permite la programación de extensos diseños mediante la integración de pequeños bloques (un diseño jerárquico agrupa a varias entidades electrónicas), los cuales pueden ser fácilmente detallados y simulados de forma individual.

Abordado desde otro punto de vista, una estructura jerárquica (figura 5.1a) relaciona a varias entidades electrónicas combinacionales y/o secuenciales a través de un algoritmo de integración (top level), con el fin de resolver una determinada aplicación. En su concepto, lo anterior es diferente al diseño de subsistemas dentro de una entidad, figura 5.1b.

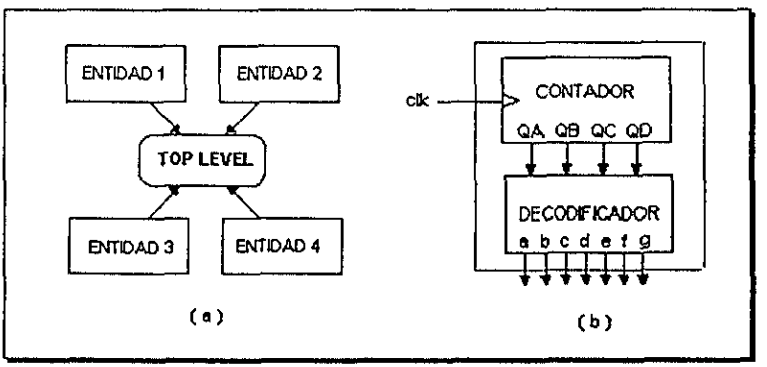


Figura 5.1. (a) Estructuras jerárquicas; (b) Integración de entidades

Otra ventaja importante del diseño jerárquico en la programación de grandes diseños, es la referente a la facilidad para trabajar de forma simultánea con otros diseñadores (paralelismo), ya que mientras uno puede estar diseñando una parte del sistema, otro puede desarrollar otro bloque distinto para de forma posterior conjuntarlos en un solo proyecto.

En este capitulo se describe la metodología utilizada para desarrollar el algoritmo que permite la integración de las diversas entidades electrónicas (contadores, decodificadores, sumadores, registros, máquinas de estado, etc.), además se hace uso extensivo de las librerías, paquetes y componentes, detallados anteriormente.

5.1. Metodología de Diseño de Estructuras Jerárquicas

Con el fin de describir y detallar la metodología empleada en el diseño de estructuras jerárquicas, programaremos como ejemplo la arquitectura interna del secuenciador bit slice AMD2909. Este dispositivo es un secuenciador de microprogramas de 4 bits desarrollado por Advanced Micro Devices, cuya función consiste en transferir a su bus de salida, una de entre 4 fuentes internas y externas de datos; estas señales de salida se conectan a la memoria de microprograma y/o de control. En la figura 5.2 se muestra la estructura externa del circuito.

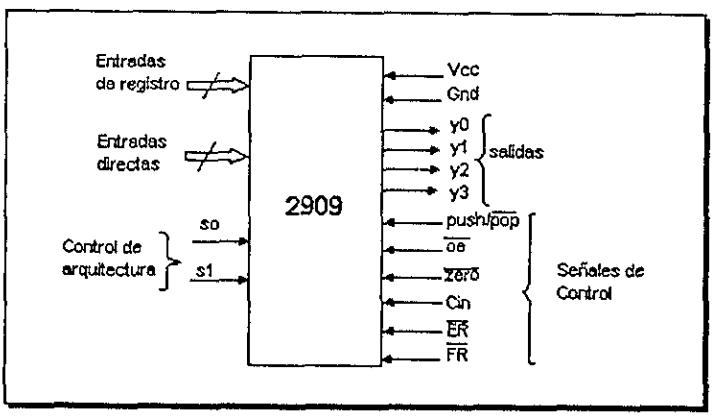


Figura 5.2. Secuenciador 2909

5.1.1 Descripción del circuito AMD2909

De forma interna, el secuenciador de 4 bits está formado por un bus de datos externo (D), un registro (R), un contador de microprograma (program counter (PC)) y un stack pointer de una palabra (ST), figura 5.3.

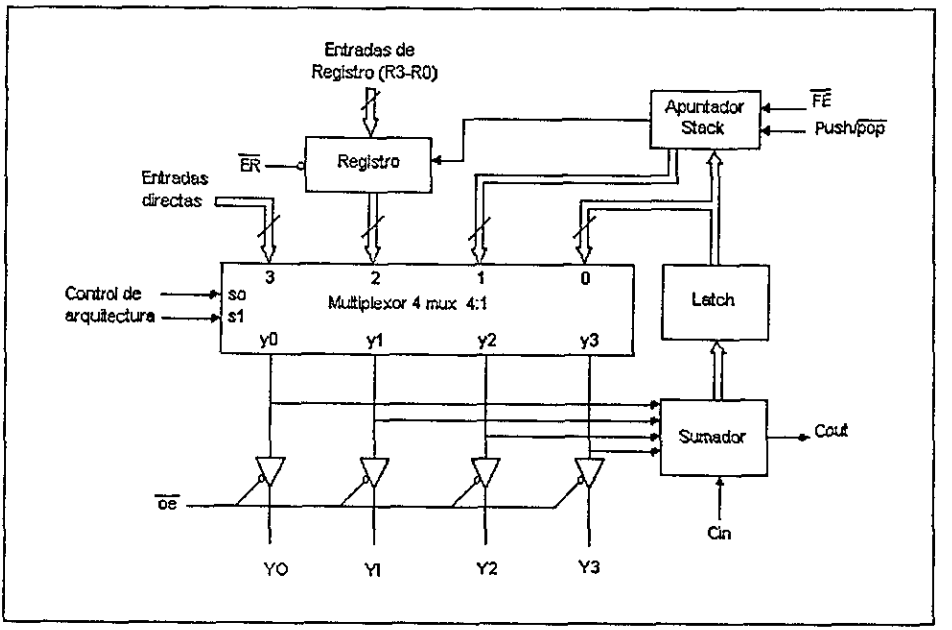


Figura 5.3. Descripción interna del circuito secuenciador AMD2909[11]

Las entradas directas (D), se utilizan para canalizar las direcciones de ramificación desde la memoria de programa y/o control hacia el secuenciador de microprogramas. Las entradas (R) en alguna de las arquitecturas de aplicación sirven para almacenar el estado presente en una función de retén o hold. El contador de microprograma, incrementa la salida en PC+1, siempre y cuando el acarreo de entrada **Cin** sea igual a uno, esto nos permite primero realizar una instrucción de cuenta, o segundo almacenar en el **stack** la siguiente dirección cuando se hace un llamado a subrutina, por lo que cuando se hace el retorno de subrutina, la dirección se obtiene del **stack** y se canaliza al bus de salida (Y).

El circuito contiene 4 multiplexores de 4:1 que seleccionan a una de sus cuatro entradas R, D, PC o ST a través de sus líneas de selección S0 y S1.

5.2 Descomposición de módulos

Como se mencionó anteriormente, el diseño jerárquico basa su fortaleza en la descomposición o división de un diseño, con el fin de poder analizar los diferentes subsistemas que lo conforman, para de forma posterior integrarlos a través de un programa denominado **top level.**

[11] Hayes P. John. Diseño de Sistemas Digitales y Microprocesadores. Mc. Graw Hill, 1988

En la figura 5.4 se muestra la interconexión interna de cada una de las señales de control que interactuán con cada uno de los subsistemas (Multiplexor de selección, Contador de microprograma, Registro y Stack Pointer) del circuito 2909.

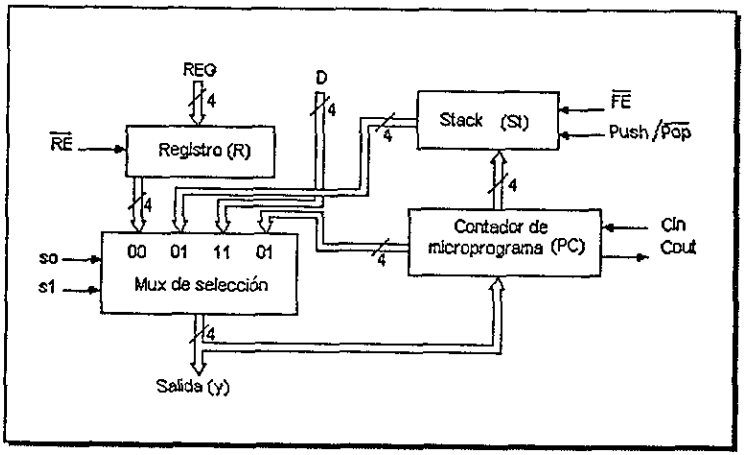


Figura. 5.4. Diagrama General de Diseño

El primer paso de diseño consiste en programar de manera individual cada uno de los **componentes** y/o unidades del circuito. Un componente es la parte dentro de programa que define un elemento físico, el cual puede ser utilizado en otros diseños o entidades. Por ejemplo, cada subsistema del secuenciador 2909 puede ser declarado en forma de ***componente*** dentro de un paquete, permitiendo con esto su fácil acceso y utilización.

5.2.1. Diseño del registro (R)

En la figura 5.5 se muestra el registro R y las señales de control asociadas a él, mientras que en el listado 1 se puede observar el código VHDL correspondiente a esta entidad.

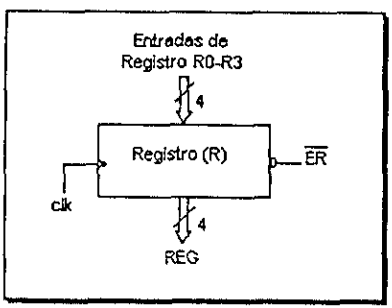


Figura 5.5. Registro de cuatro bits

```
-- Diseño del registro R
library ieee;
use ieee.std_logic_1164.all;
entity registro is port(
        r:    in std_logic_vector(3 downto 0);
 er, clk:     in std_logic;
      reg:    inout std_logic_vector(3 downto 0));
end registro;

architecture arq_reg of registro is
begin
 process (clk,re,reg,r) begin
   if (clk'event and clk ='1')then
       if er = '0' then
          reg <= r;
   else
          reg <= reg;
       end if;
   end if;
 end process;
end arq_reg;
```

Listado 1. Programación de un registro de 4 bits

5.2.2. Diseño del multiplexor

Este componente es simplemente un multiplexor cuadruple de 4:1, el cual tiene dos líneas de selección (S0 y S1), cuatro entradas (R, ST, D,PC) y una salida (Y) de 4 bits. (Figura 5.6).

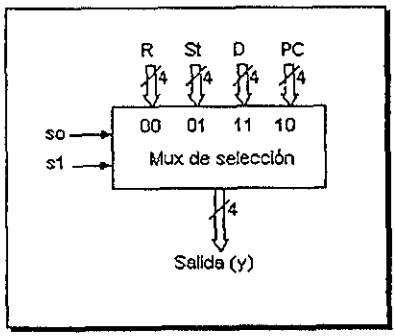


Figura 5.6. Multiplexor de selección

El código correspondiente a este componente se muestra en el listado 2.

```
-- Diseño del mux que selecciona una operación
library ieee;
use ieee.std_logic_1164.all;
   entity mux_4 is port(
      d,r,st,pc: in std_logic_vector(3 downto 0);
              s: in std_logic_vector(1 downto 0);
              Y: out std_logic_vector(3 downto 0));
   end mux_4;

architecture arq_mux of mux_4 is
begin
    with s select
         Y <=   r when "00",
               st when "01",
               pc when "10",
                d when others;
end arq_mux;
```

Listado 2. Código del multiplexor de selección

5.2.3. Contador de Microprograma (PC)

La función de este bloque es básicamente incrementar la dirección de entrada cuando *Cin* sea igual a uno. El diagrama correspondiente se muestra a continuación:

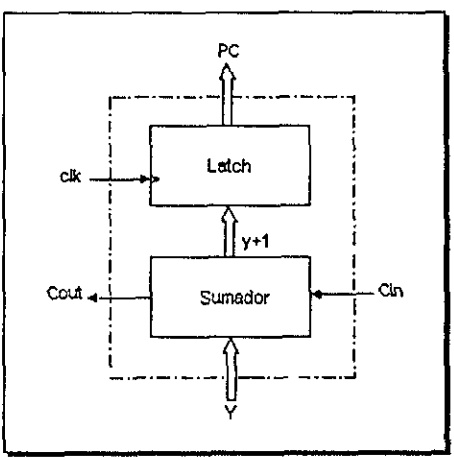


Figura 5.7. Contador de microprograma

En el listado 3 se aprecia el código que describe el funcionamiento del contador de microprograma.

Como puede observarse la manera más sencilla de programarlo es a través de la arquitectura funcional, considerando al circuito sumador y el contador como una sola entidad de diseño, con entradas y salidas generales. El funcionamiento es muy simple, cuando el acarreo de entrada (Cin) tiene el valor de '1' y el reloj del sistema está en la transición de '0' a '1', la dirección Y se incrementa en uno, y su valor es transferido al bus de salida pc. En caso contrario, cuando Cin = '0', se asigna a pc el valor de Y sin cambios.

```
-- Diseño del bloque de cuenta (Contador de microprograma)
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
    entity mpc is port(
   Cin, clk:   in std_logic;
         Y:    in std_logic_vector(3 downto 0);
      Cout:    inout std_logic;
        pc:    inout std_logic_vector(3 downto 0));
    end mpc;

architecture arq_mpc of mpc is
begin
  process (clk, Y, Cin) begin
           if (clk'event and clk = '1') then
               if (Cin = '1') then
                  pc <= Y + 1;
               else
                  pc <= Y;
               end if;
           end if;
  end process;

   Cout <= (Cin and Y(0)and Y(1) and Y(2) and Y(3));

end arq_mpc;
```

Listado 3. Diseño del contador de microprograma

5.2.4. Stack Pointer (St)

La pila ó stack de la figura 5.8, esta diseñada para almacenar un dato de 4 bits, de tal manera que cuando dentro de un programa se hace un llamado a subrutina, se almacena en la pila la siguiente dirección (PC + 1), por lo que al ocurrir el retorno de subrutina, la dirección es obtenida de la pila y se envía al multiplexor de selección, el cual lo canaliza al bus de salida (Y).

Para que un dato pueda ser introducido y almacenado dentro de la pila, se debe habilitar primero la señal "FE" (fifo enable) así como la señal push (fe='0' y push = '1'). De esta forma, cuando la transición del reloj (clk) sea de '0' a '1', el dato que se encuentra en pc se introduce y almacena dentro del stack (pila) hasta que la señal push se deshabilite (push =

'0'). Para sacar el dato se habilita la señal pop (pop ='0'), con lo que el dato es obtenido y canalizado al bus de salida st.

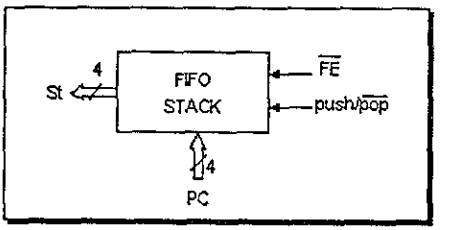


Figura 5.8. Pila o Stack de una palabra de 4 bits

A continuación se muestra el programa que indica el funcionamiento de la entidad correspondiente al stack, listado 4.

```
-- Diseño de un stack de una palabra de 4 bits
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

  entity stack is port(
   clk,fe,push,pop: in std_logic;
      pc:      in std_logic_vector (3 downto 0);
       st:     inout std_logic_vector (3 downto 0));
    end stack;

architecture arq_stack of stack is
  signal var: std_logic_vector  (3 downto 0);
 begin
   process (fe, clk, push, pop, pc)
     variable x: std_logic_vector (3 downto 0);
       begin
          if (clk'event and clk = '1') then
               if (fe = '0') then
          if (push = '1') then
                      x := pc; -- almacena dato
                      var <= x;
                elsif (pop = '0') then
                      st <= var; -- saca dato
                 else
                      st <= st;
                end if;
              end if;
           end if;
   end process;
end arq_stack;
```

Listado 4. Diseño de una pila de una palabra de 4 bits

5.3. Creación de un paquete de componentes

Una vez que han sido diseñados cada uno de los módulos que forman el diseño principal, se debe crear un programa que contenga los componentes de cada una de las entidades de diseño descritas anteriormente. Para esto es necesario identificar primero el paquete en el que se almacenarán los diseños (el nombre asignado al paquete debe ser elegido por el usuario). De manera posterior se declara cada uno de los componentes que integran el diseño, en este caso se trata del registro, multiplexor, pc y stack.

Como se puede observar, en el listado 5 se aprecia la manera de declarar cada uno de los componentes correspondientes a un paquete identificado como comps_sec. Nótese como cada componente es declarado de manera similar a una entidad de diseño, con la omisión de la palabra reservada **is** y agregando la clausula **component**.

```
--Creación del paquete de componentes del secuenciador 2909
library ieee;
use ieee.std_logic_1164.all;
package comps_sec is
   component registro port(
        r:       in std_logic_vector(3 downto 0);
        er,clk: in std_logic;
        reg:     inout std_logic_vector(3 downto 0));
   end component;

 component mpc port(
  Cin,clk: in std_logic;
        Y: in std_logic_vector(3 downto 0);
     Cout: inout std_logic;
       pc: inout std_logic_vector(3 downto 0));
   end component;

   component stack port(
      clk, fe,push,pop:  in std_logic;
       pc: inout std_logic_vector(3 downto 0);
       st: inout std_logic_vector(3 downto 0));
   end component;

    component mux_4 port(
      d,r,st,pc: in std_logic_vector(3 downto 0);
              s: in std_logic_vector(1 downto 0);
              Y: buffer std_logic_vector (3 downto 0));
   end component;
end comps_sec;
```

Listado 5. Creación del paquete que contiene los componentes del secuenciador

5.3.1. Diseño del programa de alto nivel (Top Level).

Como ya se mencionó, en VHDL se puede diseñar de forma estructural, es decir, uniendo componente por componente utilizando señales (listado 5). Esta metodología es la base del diseño jerárquico, ya que cada bloque o componente del diseño se interconecta entre sí a través de señales o buses internos, los cuales son declarados y asociados por medio de claúsulas propias del lenguaje. El programa de alto nivel que realiza esta función es mostrado a continuación en el listado 6.

```
-- Diseño de los componentes del secuenciador 2909

library ieee, amd;
use ieee.std_logic_1164.all;
use work.std_arith.all;
use amd.comps_sec.all; --paquete creado dentro de la librería amd
      entity amd2909 is port(
     r:  in std_logic_vector (3 downto 0);
     d:  in std_logic_vector (3 downto 0);
    er:  in std_logic;
   clk:  in std_logic;
     s:  in std_logic_vector (1 downto 0);
    fe:  in std_logic;
  push:  in std_logic;
   pop:  in std_logic;
   Cin:  in std_logic;
  Cout: inout std_logic;
     Y: buffer std_logic_vector (3 downto 0));
   end amd2909;

architecture arq_amd of amd2909 is
   signal reg: std_logic_vector (3 downto 0);
   signal st:  std_logic_vector (3 downto 0);
   signal pc: std_logic_vector (3 downto 0);

  begin

   -- inicia interconexión de los componentes

 u1: registro port map (clk => clk, er => er, reg => reg, r => r);
 u2: mpc      port map (Cin=>Cin, Cout=>Cout, clk=>clk, Y=>Y, pc=>pc);
 u3: stack    port map (clk => clk, se => se, push => push, pop =>
                        pop, mpc => mpc, st => st);
 u4: mux_4    port map (d=>d, r=>r, st=>st, pc=>pc, s=>s, Y=>Y);

 end arq_sec;
```

Listado 6. Creación del programa principal

En la parte inicial del programa, se llama a la librería amd, la cual contiene el paquete comps_sec, que cuenta con los componentes que se utilizarán en el diseño.

La declaración de la entidad, consiste en todas las terminales de entrada y salida del secuenciador, las cuales son nombradas de la misma forma en que se encuentran referenciadas en su módulo. Las señales encargadas de interconectar cada uno de los módulos se declaran dentro de la arquitectura (recordemos que estas señales no tienen asignada ninguna terminal del dispositivo).

La segunda parte del código hace referencia a la conexión de los distintos componentes utilizando la clausula **port map**. U1,u2,u3 y u4 son llamadas etiquetas de **asignación inmediata**. En cada una de las **asociaciones**, el símbolo => es usado para **asociar (map)** las señales **actuales** (es decir las que conforman la entidad amd2909) con las **locales** (los puertos que componen cada módulo de diseño). Una vez que se conecta cada módulo, el diseño es compilado, generando con esto un archivo **.jed**, el cual finalmente contiene todos los componentes creados.

5.4. Creación de una librería en Warp

Como se puede apreciar, en el listado anterior (listado 6) se utilizó una librería llamada `amd`, la cual fue creada con el propósito de almacenar el paquete `comps_sec` que contiene los componentes de nuestro diseño. Ahora, con el fin de que el lector pueda crear de manera fácil una librería de trabajo, se introduce este apartado, en el cual se indican los pasos a seguir dentro de la herramienta Galaxy, para crear dicha librería.

1. Crear un proyecto llamado c:\ejemplo\librería

2. Seleccionar todos los archivos que fueron creados y adicionarlos al proyecto (multiplexor, registro, pc y stack).

3. Desde el menú principal, seleccionar la opción **Libraries** del menú **File**. Esto genera que una pantalla llamada **Manejador de Librerías en Galaxy** sea desplegada.

4. Seleccionar **Create library** desde la opción **File** (del manejador de librerías). En la pantalla que se muestra, debe escribirse el nombre con que se identificará dicha librería, en nuestro caso tecleamos `amd`. Presionar OK para aceptar los cambios.

5. Presionar el botón **Done** para que el nombre de la librería sea almacenado dentro del proyecto.

6. Ya dentro de la pantalla principal, elegimos la opción **Select all** desde el menú **Files**. Aquí se presiona el botón **File** en el panel correspondiente a **Synthesis options**. Una ventana como la mostrada en la figura 5.9, será desplegada. El nombre en la parte superior de esta ventana debe ser el del archivo de la lista de proyectos. Aquí es necesario seleccionar la opción **other** y la librería de trabajo que ha sido creada (amd). Una vez realizado lo anterior se presiona *Ok*. Este paso debe realizarse con cada uno de los diseños existentes dentro del proyecto.

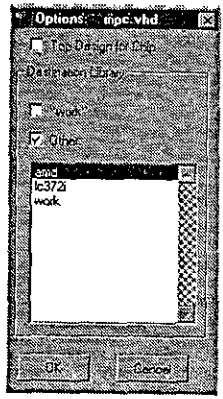


Figura 5.9. Archivos compilados dentro del librarfa amd

7. Una vez que se encuentran todos los diseños dentro de la librería, debe seleccionarse **Save** del menú **File** para asegurar que los cambios realizados han sido guardados.

8. El siguiente paso consiste en la compilación de los diseños dentro de la librería. Esto se lleva a cabo presionando el botón **Smart** que se encuentra dentro de la caja de proyectos (**proyect**). Con esta opción todas las unidades de diseño y su paquete correspondiente, son compiladas dentro de la librería.

9. A manera de verificar si los diseños han sido compilados dentro de la librería, seleccionemos **libraries** del menú **files**. Esta opción abre el manejador de librerías de Galaxy. Aquí notaremos como varios diseños existen ahora dentro de la librería amd.

Bibliografía

Bibliografía Básica

G. Maxinez David: *Amplificación de Señales*. ITESM-CEM, 1993
Hon R.W y Sequin C.H.: *A guide to LSI implentation*. Xerox Parc, 1980
Mead C. y Conway L.: *Introduction to VLSI Systems*. Addison Wesley, VLSI series 1980.
Teres LL; Torroja Y.; Olcoz S.; Villar E.: *VHDL Lenguaje Estándar de Diseño Electrónico*. Mc. Graw Hill, 1998
Floyd T.L.: *Fundamentos de Sistemas Digitales*. Prentice Hall, 1998
Van den Bout Dave: *The practical Xilinx Designer Lab Book*. Prentice Hall, 1998
Instituto de Ingeniería Eléctrica y Electrónica, IEEE. *Revista Computer*. IEEE,1977
Delgado C.; Lecha E.; Moré M.; Terés Ll.; Sánchez L.: *Introducción a los lenguajes VHDL, Verilog y UDL/I*. Novática No. 112, España 1993
Ecker W.: *The Design Cube*. Euro VHDL Forum, 1995
Novatica (varios autores): *Monografía sobre los lenguajes de diseño de hardware*. Revista *Novatica* núms. 112-113, nov-94 a feb-95.
Maxinez David G., Alcalá Jessica: ***Diseño de Sistemas Embebidos a través del Lenguaje de Descripción en Hardware VHDL***. XIX Congreso Internacional Académico de Ingeniería Electrónica. México, 1997.
Kloos C., Cerny E.: *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.
Advanced Micro Devices: *Programmable Logic Handbook/Data book*. Advanced Micro Devices, 1986.
Navabi Zainalabedin: *Analysis and Modeling of Digital Systems*. Mc. Graw Hill, ,1988.
Altera Corporation: *User Configurable Logic Data Book*. Altera Corp., 1988
Altera Corporation: *The Maximalist Handbook*. Altera Corp., 1990
Ismail M., Fiez T.: *Analog VLSI*. Mc. Graw Hill, 1994.
Hayes John P.: *Computer Architecture and Organization*. Mc. Graw Hill, 1979.
Naish P., Bishop P.: *Designing ASICs. Ellis Horwood Limited*. Chichester, 1988.
Barbacci M.R.: *The ISP Computer Description Language*. Carnegie-Mellon University,1981.
Wakerly J. F.: *Digital Desing Principles and practices*. Prentice Hall, 1990.
Skahill Kevin.: *VHDL for programmable logic*. Addison Wesley, 1996.

Bibliografía Complementaria

Mazor S., Laangstraar P.: ***A Guide to VHDL***. Kluwer Academic Publisher, 1993.
Kloos C., Cerny E.: **Hardware Description Language and their aplications. Specification, modelling, verification and synthesis of microelectronic systems**. Chapman&Hall, 1997
Armstrong, James R.; F. G. Gray: ***Structured Logic Desing with VHDL***. Prentice Hall, 1993.
Bhasker J. A: ***A VHDL Primer***. Prentice Hall, 1992.
Randolph H.: ***Applications of VHDL to Circuit Design***, Kluwer Academic Publisher
Altera Corporation: www.altera.com
Cypress Corporarion: www.cypress.com
Xilinx Corporation: www.xilinx.com
Organización Mundial de VHDL: www.vhdl.org
Campos de Lógica Programable: www.fpga.com

Apéndice A. Identificadores, Tipos y Atributos

Al igual que otros lenguajes de alto nivel, VHDL utiliza diversos conceptos que son importantes al momento de programar. Algunos de ellos son explicados en este apartado.

Identificadores

Los identificadores son simplemente los nombres o etiquetas que se usan para referenciar variables, constantes, señales, procesos, etc. Estos identificadores pueden ser números, letras del alfabeto y guiones bajos (_) que separen caracteres. Todos los identificadores deben seguir ciertas especificaciones o reglas para que puedan ser compilados sin errores, por ejemplo, el primer carácter siempre es una letra, la cual puede ser minúscula o mayúscula (solo en el uso de identificadores).

VHDL cuenta con una lista de palabras reservadas (las cuales se muestran en el apéndice B), que no pueden ser utilizadas como identificadores por ser de uso exclusivo del compilador.

Objetos de datos

Un objeto en VHDL es un elemento del lenguaje que tiene un valor especifico, por ejemplo un valor del tipo bit. En este lenguaje, existen cuatro clases distintas de objetos: constantes, señales, archivos y variables.

- *Constantes.*

Son objetos que mantienen siempre un valor fijo durante la ejecución del programa. De manera general, las constantes son utilizadas para mejorar la legibilidad del código, debido a que permiten identificar de manera sencilla el valor que les ha sido asignado.

La sintaxis para declarar una constante es la siguiente:

constant identificador: tipo := expresión;

Ejemplos.

```
constant Vcc: real := 5.0;
constant cinco: integer := 3 + 2;
constant tiempo: time := 100 ps;
constant valores: bit_vector := "10100011";
```

como se puede observar, las constantes requieren de la asignación de un nombre, un tipo de datos y de la expresión que indique el valor especifico que tomarán.

En el momento de programar, **las constantes son válidas únicamente en la unidad de diseño donde han sido declaradas**, por ejemplo, una constante definida en una declaración de entidad, es visible solo dentro de la entidad; cuando la constante es declarada en la arquitectura es visible únicamente dentro de la arquitectura y cuando se encuentra definida en la región de declaraciones de un proceso, es visible solo para ese proceso.

- *Señales.*

Son objetos utilizados como alambrados que permiten interconectar componentes dentro de la entidad. Estas señales permiten representar entradas o salidas de compuertas lógicas que no tienen una terminal externa al dispositivo. Por ejemplo observemos la figura 1.

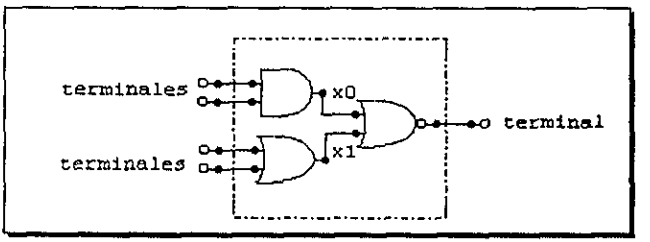


Figura 1. Ubicación de señales dentro de un diseño

Se puede apreciar que las señales etiquetadas como **x0** y **x1**, no tienen asignada una terminal del dispositivo, ya que solo son utilizadas como un medio para interconectar las los componentes.

La forma de declarar una señal es la siguiente:

signal identificador: tipo[:= rango];

Ejemplos.
signal vcc: bit: '1';
signal suma: bit_vector (3 **downto** 0);

- *Variables.*

Una variable tiene asignado un valor que cambia continuamente dentro del programa. Estos objetos se utilizan para manejar datos aleatorios, o que no tienen un valor especifico.

La forma en que se declara una variable es la siguiente:

variable identificador(s): tipo[rango][:= expresión];

Ejemplos.

variable contador: bit_vector (0 **to** 7);
variable x, y: integer;

Las variables no pueden proyectar formas de onda a su salida, debido a que su valor cambia constantemente y por lo tanto no se permite establecer un valor en cierto instante de tiempo.

- ***Archivos.***

Un archivo es un objeto que permite la comunicación del diseño con su entorno exterior, ya por medio de ellos se pueden leer y escribir datos cuando se hacen evaluaciones del circuito. Un archivo es de un tipo de datos determinado y solo puede almacenar datos de ese tipo.

La sintaxis para declarar un archivo es la siguiente:

file identificador: tipo_archivo **is** [dirección "nombre";]

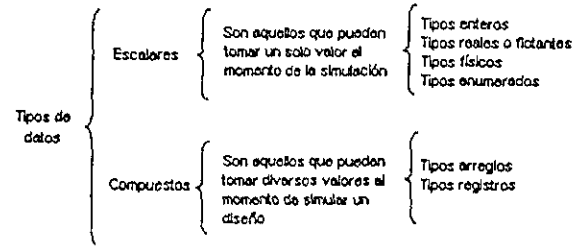
Ejemplos.

file operaciones : Archivo_Enteros **is in** "datos.in";
file salidas : Archivo_Enteros **is out** "datos.out";

Tipos de datos

Un tipo de datos se utiliza para definir el valor que un objeto puede tomar, así como las operaciones que se realizan con ese objeto. Dentro de VHDL existen dos tipos básicos: el tipo **compuesto** y el tipo **escalar**, los cuales agrupan a su vez a varios tipos dentro de ellos.

A continuación se muestran estos tipos de datos, así como la forma en que se encuentran clasificados:



Tipos escalares

En esta sección, se describe a detalle los cuatro tipos escalares que existen en VHDL, también se muestran algunos ejemplos que permiten comprender de mejor manera su uso:

- *Tipos enumerados*

Este tipo es utilizado para listar los diversos valores que puede contener un objeto. Se llaman enumerados debido a que listan todos y cada uno de los valores que forman el tipo.

La sintaxis utilizada para declararlo es la siguiente:

type identificador **is** definición_tipo;

como se puede apreciar, la declaración del tipo contiene un nombre y la definición del tipo. El nombre permite referenciarlo posteriormente, mientras que la definición corresponde a los valores que tomará el tipo.

Algunos ejemplos de declaraciones de tipos enumerados se muestran a continuación:

type nombres **is** (Ana, Mario, Julio, Cecilia);
type máquina **is** (edo_presente, edo_futuro, estado);
type letras **is** ('a', 'b', 'x', 'y', 'z');

los tipos `bit y booleano` están considerados dentro de los tipos enumerados, debido a que pueden tomar mas de un valor:

type booleano **is** (verdadero, falso);
type bit **is** ('0', '1');

- *Tipos enteros y tipos reales*

Los tipos enteros y reales, tal como su nombre lo indica sirven para representar números enteros y reales (fraccionarios) respectivamente. VHDL soporta valores enteros en el rango de –2,147-483,647 (-2^{31}-1) hasta 2,147,483,647 (2^{31}-1), y números reales en el rango de -1.0E38 a 1.0E38.

Ambos tipos, tanto enteros como reales siguen la misma sintaxis:

objeto identificador : **type range** [valores];

Un rango (**range**) es una palabra reservada por VHDL, utilizada para definir un conjunto de valores. Cabe mencionar, que no todas las herramientas en VHDL manejan valores con signo; en nuestro caso el compilador utilizado (**Warp**), solo maneja valores sin signo.

- *Tipos Físicos*

Se refiere a los valores que son utilizados como unidades de medida. En VHDL el único tipo físico que se encuentra predefinido es **time** (tiempo), el cual contiene como unidad primaria el *femtosegundo* (fs). La manera de definir un tipo físico es la siguiente:

Type time **is range** 0 **to** 1E20
units
fs;
ps = 1000 fs;
ns = 1000 ps;
us = 1000 ns;
ms = 1000 us;
sec = 1000 ms;
min = 60 sec;
hr = 60 min;
end units;

se pueden crear otros tipos físicos como metros, gramos, etc., solo que en el diseño digital es difícil utilizar estos parámetros. Por esta razón solo se muestran los tipos predefinidos, sin profundizar en el tema.

Tipos Compuestos

Como se mencionó anteriormente, los tipos compuestos pueden tener valores múltiples en un mismo tiempo. Este tipo está formado por los **arreglos** y **registros**.

- *Tipo arreglo.*

El tipo arreglo está formado por múltiples elementos de un tipo en común. Estos arreglos pueden ser considerados también como vectores, ya que agrupan varios elementos de un mismo tipo.

La sintaxis utilizada para declarar un arreglo es la siguiente:

type identificador **is array** (rango) **of** tipo_objetos;

Como se puede observar, en ambas declaraciones es necesario utilizar un valor (rango), el cual indica el conjunto de valores que va a tomar el tipo. En este caso el rango no se encuentra especificado, pero debe considerarse que al momento de ser asignado, este debe ser un número entero positivo (número natural).

En el estándar IEEE 1076 y 1164 se encuentran definidos dos arreglos importantes, llamados **bit_vector** y **std_logic_vector**, los cuales forman parte de los tipos bit y

std_logic respectivamente. A continuación se puede observar la forma cómo se declaran estos arreglos;

type bit_vector **is array** (rango) **of** bit;
type std_logic_vector **is array** (rango) **of** std_logic;

el tipo std_logic, es mas versátil que el tipo bit, debido a que contiene los valores de *alta impedancia* ('Z') y *no importa* (-).

A continuación se muestran algunos ejemplos de declaraciones de arreglos:

type dígitos **is array** (9 **downto** 0) **of** integer;
type byte **is array** (7 **downto** 0) **of** bit;
type dirección **is array** (10 **to** 62) **of** bit;

otra opción al utilizar arreglos radica en la facilidad que presentan para crear tablas de verdad:

```
type tabla is array (0 to 3,  0 to 2) of bit;
constant comp_and: tabla := (
                            "00_0",
                            "01_0",
                            "10_0",
                            "11_1");
```

el arreglo declarado en este ejemplo es de dos dimensiones, ya que tiene un valor para el número binario que toman las entradas (de 0 a 3) y otro valor para el número de bits de entrada y salida (dos bits de entrada y uno de salida). Los guiones colocados entre los bits, separan las entradas de las salidas.

- *Tipo Archivo (record).*

A diferencia de los arreglos, los tipos archivo están formados por elementos de diferentes tipos, los cuales reciben el nombre de *campos*. Cada uno de estos campos debe tener un nombre que permita identificarlos fácilmente dentro del registro.

Es importante destacar que el nombre de **registro**, no tiene nada que ver con un registro en hardware utilizado para almacenar valores, ya que aunque los nombres son similares, en VHDL son tomados como conceptos totalmente distintos.

La forma de declarar un tipo archivo es la siguiente:

type identificador **is record**
Identificador : tipo;
end **record**;

Apéndice B Palabras Reservadas

A continuación se muestra una lista de las palabras reservadas en VHDL. Ninguna palabra reservada puede ser utilizada como identificador de señales.

Abs
Access
After
Alias
All
And
Architecture
Array
Assert
Atribute
Begin
Block
Body
Buffer
Bus
Case
Compoennt
Configutation
Constant
Disconnect
Downto
Else
Elsif
End
Entity
Exit
File
For
Function
Generate
Generic
Group
Guarded
If
Impure
In
Inertial
Inout
Is
Label
Library
Lindage
Literal
Loop
Map
Mod
Nand
New
Next
Nor
Not
Null
Of
On
Open
Or
Others
Out
Package
Port
Postponed
Procedure
Process
Pure
Range
Record
Register
Reject
Rem
Report
Return
Rol
Ror
Select
Severity
Signal
Shared
Sla
Sll
Sra
Srl
Subtype
Then
To
Transport
Type
Unaffected
Units
Until
Use
Variable
Wait
When
While
With
Xnor
Xor

Esta lista está tomada del estándar IEEE Std1076 –1993 del Manual de Referencia del Lenguaje VHDL, impreso por el Instituto de Ingenieros en Eléctricos y Electrónicos en 1994.

Sección 2. Operadores definidos en VHDL según su orden de precedencia

Operador	Descripción	Tipos de operandos	Resultado
**	potencia	Entero operador entero Real operador entero	Entero Real
Abs	Valor absoluto	Numérico	Ídem operando
not	negación	Bit, booleano, vectores de bits	Ídem operando
*	multiplicación	Entero operador entero Real op real Físico op real Físico op entero Entero op físico Real op físico	Entero Real Físico Físico Físico Físico
/	División	Entero op entero Real op real Físico op entero Físico op real Físico op físico	Entero Real Físico Físico Físico
Mod	módulo	Entero op entero	Entero
+	suma	Numérico op numérico	Ídem operandos
-	resta	Numérico op numérico	Ídem operandos
&	concatenación	Vector op vector Vector op elemento Elemento op vector Elemento op elemento	Vector Vector Vector vector
sll	Desp. Logico izquierdo	Vectores de bits op entero	Vector de bits
srl	Despl. Lógico derecho	Vector de bits op entero	Vector de bits
sla	Despl. Arit. izquierdo	Vector de bits op entero	Vector de bits
rol	Rotación izquierda	Vector de bits op entero	Vector de bits
ror	Rotación derecha	Vector de bits op entero	Vector de bits
=	Igual que	No archivo op no archivo	Booleano
/=	Diferente que	No archivo op no archivo	Booleano
<	Menor que	No archivo op no archivo	Booleano
>	Mayor que	No archivo op no archivo	Booleano
<=	Menor o igual que	No archivo op no archivo	Booleano
>=	Mayor o igual que	No archivo op no archivo	Booleano
and	y lógica	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos
or	o lógica	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos
nand	y lógica negada	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos
nor	o lógica negada	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos
xor	or exclusiva	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos
xnor	or exlusiva negada	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos

Fuente de información: The IEEE Estándar VHDL Language Reference Manual, IEEE Std 1076-1987, 1988.

Instalación de Warp

Warp puede ser instalado tanto en PCs como en plataforma Sun. Los requerimientos para cada uno se muestran a continuación:

REQUERIMIENTOS	WINDOWS PC	ESTACIONES DE TRABAJO SUN
Procesador	80486 mínimo	CPU SPARC
RAM	16Mb	16 Mb
Espacio en Disco Duro	60Mb	60Mb
Sistema operativo	Windows 3.1 en adelante	Sun Os 4.1.1 y porteriores

Una vez que se han verificado los requerimientos anteriores, se procede a instalar Warp en una PC, siguiendo los siguientes pasos:

1) Cerrar todos las aplicaciones antes de correr el programa de instalación
2) Insertar el CD y correr el archivo pc\setup.exe

Si se requiere tener acceso a la documentación en línea, es necesario instalar el Acrobat Reader, el cual se encuentra dentro del CD-ROM. Para instalar este programa existen dos formas: 1) Durante la instalación de Warp, aparece una pantalla desplegando un mensaje donde pregunta si se desea instalar Adobe Reader, seleccione la opción Yes y el programa por si solo lo instalará; 2) ejecutar el archivo pc\acroread\ar32e30.exe y hacer doble click con el botón izquierdo del mouse en este archivo.

Para instalar Warp en Plataforma Sun (con sistema operativo SunOs 4.1.x/ Solaris 2.5) o HP 9000 (serie 7000) se necesitan seguir los siguientes pasos:

En SunOs 4.1.x
Una vez introducido el CD-ROM se ejecutan los siguientes comandos para crear el directorio */cdrom*
mkdir /cdrom
mount -rt hsfs /dev/sr0 /cdrom

En Solaris 2.5
Se ejecutan los siguientes comandos para crear el directorio /cdrom
mkdir /cdrom
mount -F ufs -r /dev/dsk/c0t6d0s2 /cdrom

En HP-UX 10.10
De igual forma que los anteriores, se ejecutan los siguientes comandos par crear el directorio /cdrom
mkdir /cdrom
mount -o ro /dev/dsk/c0t2d0 /cdrom

Apéndice D. Hojas Técnicas del CPLD CY7C372i

CYPRESS

ADVANCED INFORMATION **CY7C372i**

UltraLogic™ 64-Macrocell Flash CPLD

Features

- 64 macrocells in four logic blocks
- 32 I/O pins
- 6 dedicated inputs including 2 clock pins
- In-System Reprogrammable (ISR™) Flash technology
  - JTAG interface
- No hidden delays
- High speed
  - f_{MAX} = 125 MHz
  - t_{PD} = 10 ns
  - t_S = 5.5 ns
  - t_{CO} = 6.5 ns
- Fully PCI compliant
- Available in 44-pin PLCC and CLCC packages
- Pin compatible with the CY7C371i

Functional Description

The CY7C372i is an In-System Reprogrammable Complex Programmable Logic Device (CPLD) and is part of the FLASH370i™ family of high-density, high-speed CPLDs. Like all members of the FLASH370i family, the CY7C372i is designed to bring the ease of use and high performance of the 22V10, as well as PCI Local Bus Specification support, to high-density CPLDs.

Like all of the UltraLogic FLASH370i devices, the CY7C372i is electrically erasable and In-System Reprogrammable (ISR), which simplifies both design and manufacturing flows, thereby reducing costs. The Cypress ISR function is implemented through a 4-pin serial interface. Data is shifted in and out through the SDI and SDO pins, respectively, using the programming voltage pin (V_{PP}). These pins are dual function, providing a pin-compatible upgrade to earlier versions of FLASH370™ devices. Additionally, because of the superior routability of the FLASH370i devices, ISR often allows users to change existing logic designs while simultaneously fixing pinout assignments.

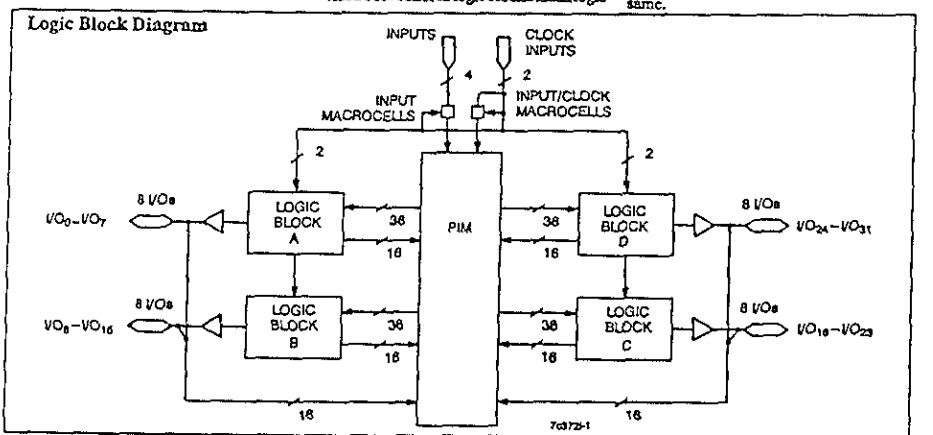
The 64 macrocells in the CY7C372i are divided between four logic blocks. Each logic block includes 16 macrocells, a 72 x 86 product term array, and an intelligent product term allocator.

The logic blocks in the FLASH3 70i architecture are connected with an extremely fast and predictable routing resource—the Programmable Interconnect Matrix (PIM). The PIM brings flexibility, routability, speed, and a uniform delay to the interconnect.

Like all members of the FLASH370i family, the CY7C372i is rich in I/O resources. Every two macrocells in the device feature an associated I/O pin, resulting in 32 I/O pins on the CY7C372i. In addition, there are four dedicated inputs and two input/clock pins.

Finally, the CY7C372i features a very simple timing model. Unlike other high-density CPLD architectures, there are no hidden speed delays such as fanout effects, interconnect delays, or expander delays. Regardless of the number of resources used, or the type of application, the timing parameters on the CY7C372i remain the same.

Logic Block Diagram



Selection Guide

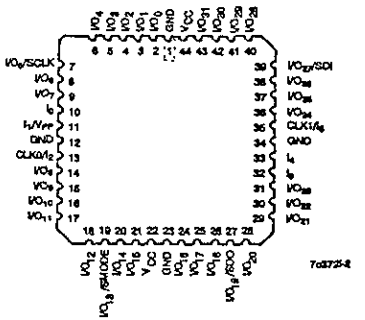
		7C372i–125	7C372i–100	7C372i–83	7C372i–66	7C372iL–66
Maximum Propagation Delay, t_{PD} (ns)		10	12	15	20	20
Minimum Set-up, t_S (ns)		5.5	6.0	8	10	10
Maximum Clock to Output, t_{CO} (ns)		6.5	6.5	8	10	10
Maximum Supply Current, I_{CC} (mA)	Commercial	250	250	250	250	125
	Military/Industrial			300	300	

Cypress Semiconductor Corporation • 3901 North First Street • San Jose • CA 95134 • 408–943–2600

Revision: October 14, 1995

CYPRESS ADVANCED INFORMATION CY7C372i

Pin Configuration



Functional Description (continued)

Logic Block

The number of logic blocks distinguishes the members of the FLASH370i family. The CY7C372i includes four logic blocks. Each logic block is constructed of a product term array, a product term allocator, and 16 macrocells.

Product Term Array

The product term array in the FLASH370i logic block includes 36 inputs from the PIM and outputs 86 product terms to the product term allocator. The 36 inputs from the PIM are available in both positive and negative polarity, making the overall array size 72 x 86. This large array in each logic block allows for very complex functions to be implemented in a single pass through the device.

Product Term Allocator

The product term allocator is a dynamic, configurable resource that shifts product terms to macrocells that require them. Any number of product terms between 0 and 16 inclusive can be assigned to any of the logic block macrocells (this is called product term steering). Furthermore, product terms can be shared among multiple macrocells. This means that product terms that are common to more than one output can be implemented in a single product term. Product term steering and product term sharing help to increase the effective density of the FLASH370 PLDs. Note that product term allocation is handled by software and is invisible to the user.

I/O Macrocell

Half of the macrocells on the CY7C372i have separate I/O pins associated with them. In other words, each I/O pin is shared by two macrocells. The input to the macrocell is the sum of between 0 and 16 product terms from the product term allocator. The macrocell includes a register that can be optionally bypassed. It also has polarity control, and two global clocks to trigger the register. The I/O macrocell also features a separate feedback path to the PIM so that the register can be buried if the I/O pin is used as an input.

Buried Macrocell

The buried macrocell is very similar to the I/O macrocell. Again, it includes a register that can be configured as combinatorial, as a D flip-flop, a T flip-flop, or a latch. The clock for this register has the same options as described for the I/O macrocell. One difference on the buried macrocell is the addition of input register capability. The user can program the buried macrocell to act as an input register (D-type or latch) whose input comes from the I/O pin associated with the neighboring macrocell. The output of all buried macrocells is sent directly to the PIM regardless of its configuration.

Programmable Interconnect Matrix

The Programmable Interconnect Matrix (PIM) connects the four logic blocks on the CY7C372i to the inputs and to each other. All inputs (including feedbacks) travel through the PIM. There is no speed penalty incurred by signals traversing the PIM.

Development Tools

Development software for the CY7C372i is available from Cypress's *Warp2™*, *Warp2+™* and *Warp3™* software packages. Both of these products are based on the IEEE standard VHDL language. Cypress also supports third-party vendors such as ABEL™, CUPL™, and LOGiC™. Please contact your local Cypress representative for further information.

Maximum Ratings

(Above which the useful life may be impaired. For user guidelines, not tested.)

Storage Temperature ..... −65°C to +150°C

Ambient Temperature with Power Applied ..... −55°C to +125°C

Supply Voltage to Ground Potential ..... −0.5V to +7.0V

DC Voltage Applied to Outputs in High Z State ..... −0.5V to +7.0V

DC Input Voltage ..... −0.5V to +7.0V

DC Program Voltage 12.5V

Output Current into Outputs ..... 16 mA

Static Discharge Voltage ..... >2001V (per MIL-STD-883, Method 3015)

Latch-Up Current ..... >200 mA

Operating Range

Range	Ambient Temperature	V_{CC}
Commercial	0°C to +70°C	5V ± 5%
Industrial	−40°C to +85°C	5V ± 10%
Military[1]	−55°C to +125°C	5V ± 10%

Note:

1. T_A is the "instant on" case temperature.

Revision: October 14, 1995

CYPRESS *ADVANCED INFORMATION* CY7C372i

Electrical Characteristics Over the Operating Range[2]

Parameter	Description	Test Conditions		Min.	Max.	Unit
V_{OH}	Output HIGH Voltage	V_{CC} = Min.	I_{OH} = −3.2 mA (Com'l/Ind)	2.4		V
			I_{OH} = −2.0 mA (Mil)			V
V_{OL}	Output LOW Voltage	V_{CC} = Min.	I_{OL} = 16 mA (Com'l/Ind)		0.5	V
			I_{OL} = 12 mA (Mil)			V
V_{IH}	Input HIGH Voltage	Guaranteed Input Logical HIGH Voltage for all Inputs[3]		2.0	7.0	V
V_{IL}	Input LOW Voltage	Guaranteed Input Logical LOW Voltage for all Inputs[3]		−0.5	0.8	V
I_{IX}	Input Load Current	GND ≤ V_I ≤ V_{CC}		−10	+10	μA
I_{OZ}	Output Leakage Current	GND ≤ V_O ≤ V_{CC}, Output Disabled		−50	+50	μA
I_{OS}	Output Short Circuit Current[4, 5]	V_{CC} = Max., V_{OUT} = 0.5V		−30	−160	mA
I_{CC}	Power Supply Current [6]	V_{CC} = Max., I_{OUT} = 0 mA, f = 1 mHz, V_{IN} = GND, V_{CC}	Com'l		250	mA
			Com'l "L" −66		125	mA
			Com'l −125		280	mA
			Mil/Industrial		300	mA

Capacitance[5]

Parameter	Description	Test Conditions	Max.	Unit
C_{IN}	Input Capacitance	V_{IN} = 5.0V at f=1 MHz	10	pF
C_{OUT}	Output Capacitance	V_{OUT} = 5.0V at f = 1 MHz	12	pF

Endurance Characteristics[5]

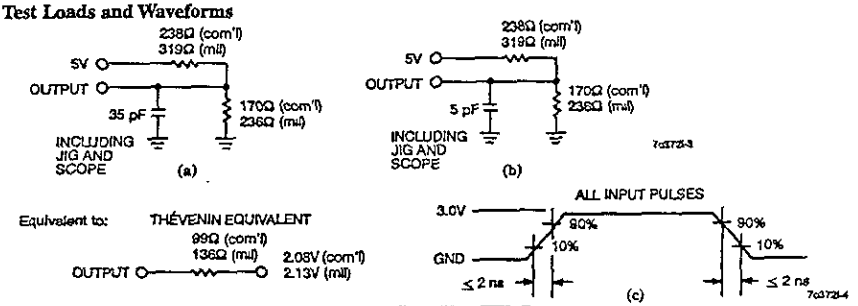
Parameter	Description	Test Conditions	Min.	Max.	Unit
N	Minimum Reprogramming Cycles	Normal Programming Conditions	100		Cycles

Notes:
2. See the last page of this specification for Group A subgroup testing information.
3. These are absolute values with respect to device ground. All overshoots due to system or tester noise are included.
4. Not more than one output should be tested at a time. Duration of the short circuit should not exceed 1 second. V_{OUT} = 0.5V has been chosen to avoid test problems caused by tester ground degradation.
5. Tested initially and after any design or process changes that may affect these parameters.
6. Measured with 16-bit counter programmed into each logic block.

Revision: October 14, 1995

CYPRESS — ADVANCED INFORMATION — CY7C372i

AC Test Loads and Waveforms



Parameter	V_X	Output Waveform—Measurement Level
$t_{ER(-)}$	1.5V	V_{OH}, 0.5V, V_X
$t_{ER(+)}$	2.6V	V_{OL}, 0.5V, V_X
$t_{EA(+)}$	1.5V	V_X, 0.5V, V_{OH}
$t_{EA(-)}$	V_{thc}	V_X, 0.5V, V_{OL}

(d) Test Waveforms

Switching Characteristics Over the Operating Range[7]

Parameter	Description	7C3721–125 Min.	7C3721–125 Max.	7C3721–100 Min.	7C3721–100 Max.	7C3721–83 Min.	7C3721–83 Max.	7C3721–66 7C3721L–66 Min.	7C3721–66 7C3721L–66 Max.	Unit
Combinatorial Mode Parameters										
t_{PD}	Input to Combinatorial Output		10		12		15		20	ns
t_{PDL}	Input to Output Through Transparent Input or Output Latch		13		15		18		22	ns
t_{PDLL}	Input to Output Through Transparent Input and Output Latches		15		16		19		24	ns
t_{EA}	Input to Output Enable		14		16		19		24	ns
t_{ER}	Input to Output Disable		14		16		19		24	ns
Input Registered/Latched Mode Parameters										
t_{WL}	Clock or Latch Enable Input LOW Time[5]	3		3		4		5		ns
t_{WH}	Clock or Latch Enable Input HIGH Time[5]	3		3		4		5		ns
t_{IS}	Input Register or Latch Set-Up Time	2		2		3		4		ns
t_{IH}	Input Register or Latch Hold Time	2		2		3		4		ns
t_{ICO}	Input Register Clock or Latch Enable to Combinatorial Output		14		16		19		24	ns
t_{ICOL}	Input Register Clock or Latch Enable to Output Through Transparent Output Latch		16		18		21		26	ns

Note:
7. All AC parameters are measured with 16 outputs switching.

Revision: October 14, 1995

Switching Characteristics Over the Operating Range[7] (continued)

Parameter	Description	7C372i–125		7C372i–100		7C372i–83		7C372i–66 7C372iL–66		Unit
		Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	
Output Registered/Latched Mode Parameters										
t_{CO}	Clock or Latch Enable to Output		6.0		6.5		8		10	ns
t_S	Set-Up Time from Input to Clock or Latch Enable	5.5		6		8		10		ns
t_H	Register or Latch Data Hold Time	0		0		0		0		ns
t_{CO2}	Output Clock or Latch Enable to Output Delay (Through Memory Array)		14		16		19		24	ns
t_{SCS}	Output Clock or Latch Enable to Output Clock or Latch Enable (Through Memory Array)	8		10		12		15		ns
t_{SL}	Set-Up Time from Input Through Transparent Latch to Output Register Clock or Latch Enable	10		12		15		20		ns
t_{HL}	Hold Time for Input Through Transparent Latch from Output Register Clock or Latch Enable	0		0		0		0		ns
f_{MAX1}	Maximum Frequency with Internal Feedback in Output Registered Mode (Least of $1/t_{SCS}$, $1/(t_S + t_H)$, or $1/t_{CO}$)[5]	125		100		83		66		MHz
f_{MAX2}	Maximum Frequency Data Path in Output Registered/Latched Mode (Lesser of $1/(t_{WL} + t_{WH})$, $1/(t_S + t_H)$, or $1/t_{CO}$)[5]	153.8		153.8		125		100		MHz
f_{MAX3}	Maximum Frequency with External Feedback (Lesser of $1/(t_{CO} + t_S)$ and $1/(t_{WL} + t_{WH})$)[5]	83.3		80		62.5		50		MHz
$t_{OH}-t_{IH}$ 37x	Output Data Stable from Output clock Minus Input Register Hold Time for 7C37x[5, 8]	0		0		0		0		ns
Pipelined Mode Parameters										
t_{ICS}	Input Register Clock to Output Register Clock	8		10		12		15		ns
f_{MAX4}	Maximum Frequency in Pipelined Mode (Least of $1/(t_{CO} + t_{IS})$, $1/t_{ICS}$, $1/(t_{WL} + t_{WH})$, $1/(t_{IS} + t_{IH})$, or $1/t_{SCS}$)[5]	125		100		83.3		66.6		MHz
Reset/Preset Parameters										
t_{RW}	Asynchronous Reset Width[5]	10		12		15		20		ns
t_{RR}	Asynchronous Reset Recovery Time[5]	12		14		17		22		ns
t_{RO}	Asynchronous Reset to Output		16		18		21		26	ns
t_{PW}	Asynchronous Preset Width[5]	10		12		15		20		ns
t_{PR}	Asynchronous Preset Recovery Time[5]	12		14		17		22		ns
t_{PO}	Asynchronous Preset to Output		16		18		21		26	ns
t_{POR}	Power-On Reset[5]		1		1		1		1	µs

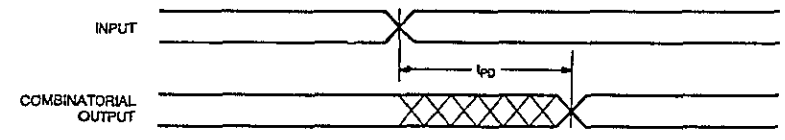
Note:
8. This specification is intended to guarantee interface compatibility of the other members of the CY7C370i family with the CY7C372i. This specification is met for the devices operating at the same ambient temperature and at the same power supply voltage.

Revision: October 14, 1995

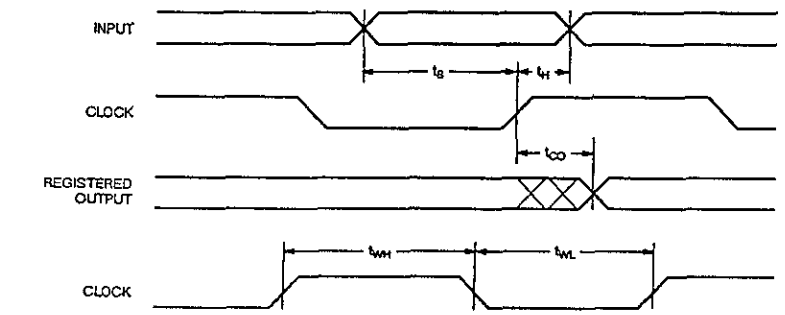
CYPRESS *ADVANCED INFORMATION* **CY7C372i**

Switching Waveforms

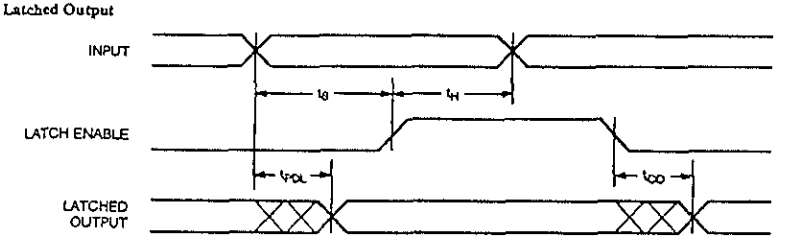
Combinatorial Output



Registered Output



Latched Output

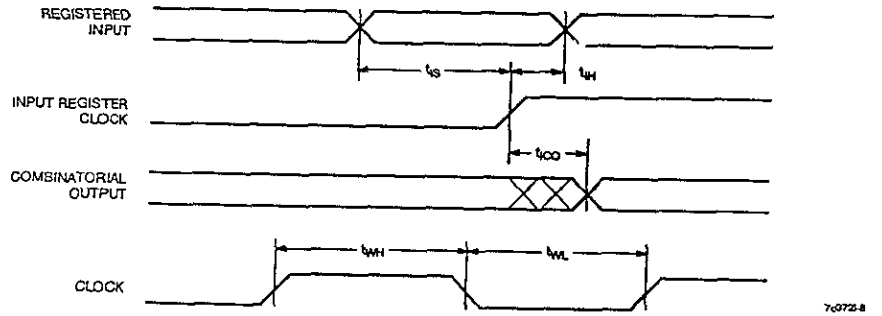


Revision, October 14, 1995

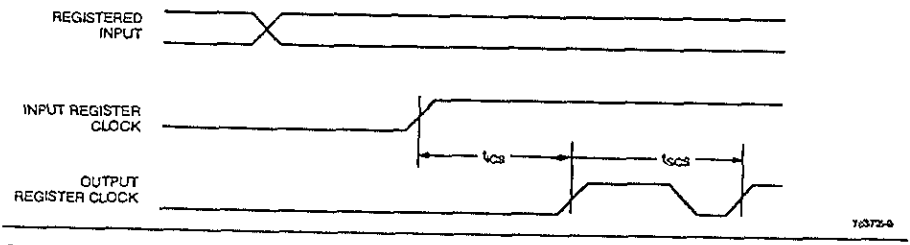
CYPRESS ADVANCED INFORMATION CY7C372i

Switching Waveforms (continued)

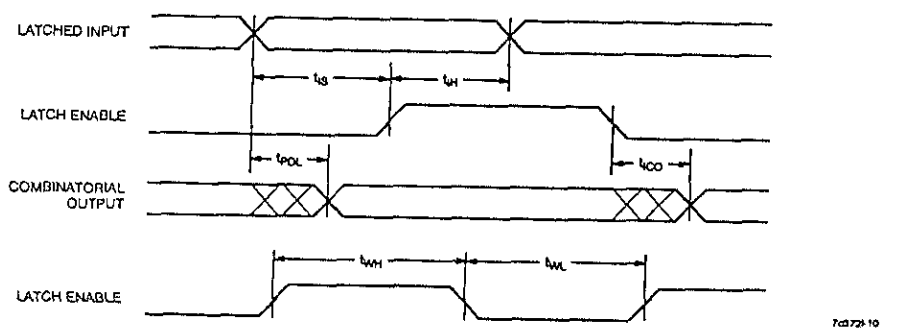
Registered Input



Clock to Clock



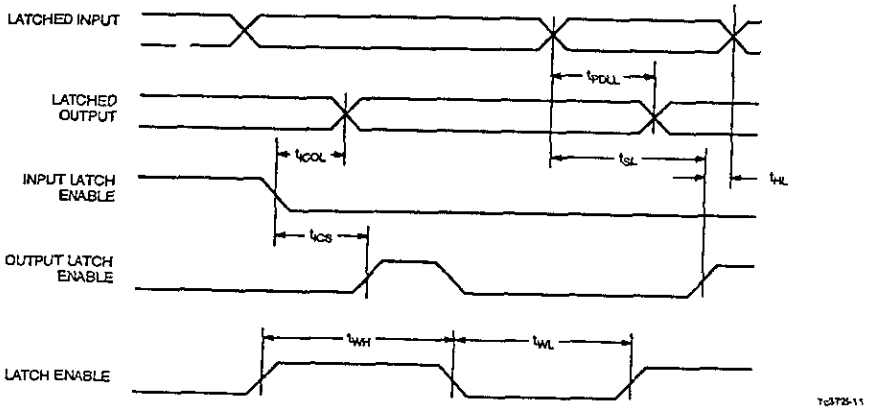
Latched Input



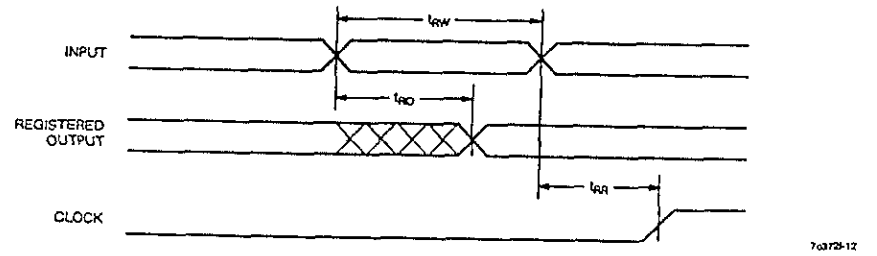
Revision: October 14, 1995

Switching Waveforms (continued)

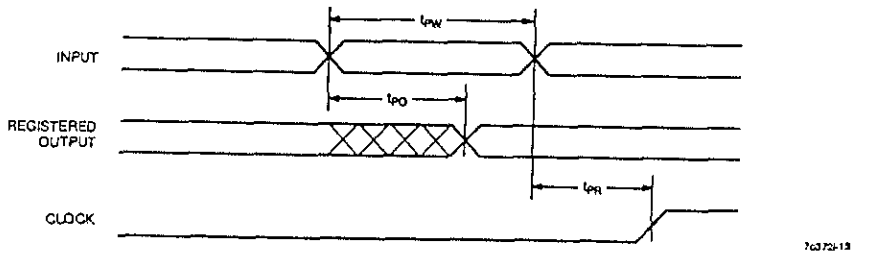
Latched Input and Output



Asynchronous Reset



Asynchronous Preset

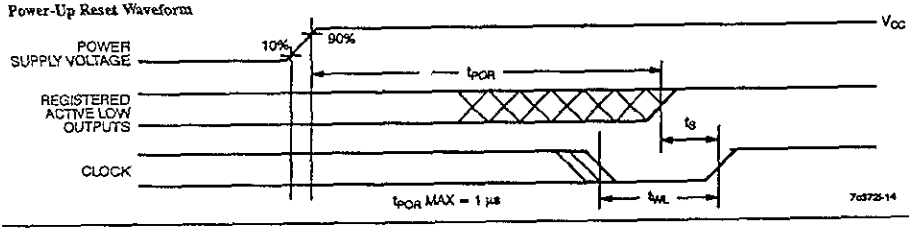


Revision: October 14, 1995

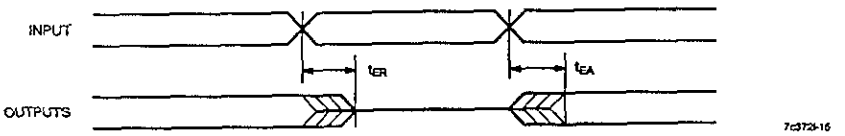
CYPRESS — ADVANCED INFORMATION — CY7C372i

Switching Waveforms (continued)

Power-Up Reset Waveform



Output Enable/Disable



Ordering Information

Speed (MHz)	Ordering Code	Package Name	Package Type	Operating Range
125	CY7C372i–125JC	J67	44-Lead Plastic Leaded Chip Carrier	Commercial
100	CY7C372i–100JC	J67	44-Lead Plastic Leaded Chip Carrier	Commercial
83	CY7C372i–83JC	J67	44-Lead Plastic Leaded Chip Carrier	Commercial
	CY7C372i–83JI	J67	44-Lead Plastic Leaded Chip Carrier	Industrial
	CY7C372i–83YMB	Y67	44-Lead Ceramic Leaded Chip Carrier	Military
66	CY7C372i–66JC	J67	44-Lead Plastic Leaded Chip Carrier	Commercial
	CY7C372i–66YMB	Y67	44-Lead Ceramic Leaded Chip Carrier	Military
	CY7C372i–66JI	J67	44-Lead Ceramic Leaded Chip Carrier	Industrial
66	CY7C372iL–66JC	J67	44-Lead Ceramic Leaded Chip Carrier	Commercial

MILITARY SPECIFICATIONS

Group A Subgroup Testing

DC Characteristics

Parameter	Subgroups
V_{OH}	1, 2, 3
V_{OL}	1, 2, 3
V_{IH}	1, 2, 3
V_{IL}	1, 2, 3
I_{IX}	1, 2, 3
I_{OZ}	1, 2, 3
I_{CC}	1, 2, 3

Switching Characteristics

Parameter	Subgroups
t_{PD}	9, 10, 11
t_{CO}	9, 10, 11
t_{ICO}	9, 10, 11
t_S	9, 10, 11
t_H	9, 10, 11
t_{IS}	9, 10, 11
t_{IH}	9, 10, 11
t_{ICS}	9, 10, 11

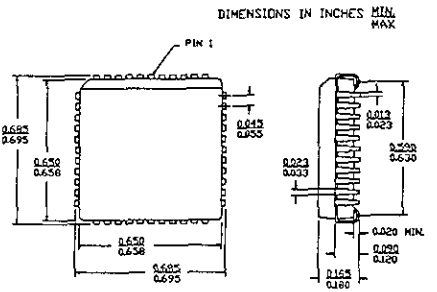
Document #: 38–00498

Revision: October 14, 1995

CYPRESS *ADVANCED INFORMATION* **CY7C372i**

Package Diagrams

44-Lead Plastic Leaded Chip Carrier J67



44-Pin Ceramic Leaded Chip Carrier Y67