INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

MAESTRÍA EN INGENIERÍA DE CÓMPUTO
CON ESPECIALIDAD EN SISTEMAS DIGITALES

IMPLEMENTACIÓN DE UN CONTROLADOR DIFUSO EN UN FPGA, UTILIZANDO LENGUAJES DE DESCRIPCIÓN DE HARDWARE

TESIS

QUE PARA OBTENER EL GRADO DE MAESTRO EN CIENCIAS EN INGENIERÍA DE CÓMPUTO CON ESPECIALIDAD EN SISTEMAS DIGITALES PRESENTA:

JUAN CARLOS HERRERA LOZADA

DIRECTOR DE TESIS:
M. EN C. OSVALDO ESPINOSA SOSA

MÉXICO, D.F. MARZO 2002

Índice

Relación de Tablas y Figuras

Página

Capítulo 1 Realizaciones Hardware de FLCs

Capítulo 2 Generalidades de las Tecnologías Programables

Capítulo 3 Modelado de FLCs, Utilizando HDLs

Capítulo 4 Integración de las Interfaces

Implementación de un Controlador Difuso en un FPGA, Utilizando Lenguajes de Descripción de Hardware

Resumen

Este trabajo de tesis versa sobre el diseño e implementación de controladores digitales en FPGAs, cuya operación está basada en lógica difusa. El estudio y la realización electrónica del algoritmo seleccionado, permitió obtener soluciones arquitecturales de procesamiento serie y otras de procesamiento paralelo, codificadas en HDLs, que se adaptan a cualquier aplicación.

Las aportaciones están dirigidas al área de las Tecnologías Programables, utilizando herramientas de diseño VLSI CAD comerciales para sintetizar códigos en Verilog y VHDL, abarcando una de las líneas más prometedoras de la electrónica digital contemporánea.

Implementation of a Fuzzy Controller on FPGAs, Using Hardware Description Languages

Abstract

This work of thesis deals with design and implementation of digital Fuzzy Logic Controllers on FPGAs. The study and electronic implementation from selected algorithm, allowed to obtain architectures with serie processing and one with parallel processing, encoded in HDLs, adaptables to any application.

The contributions are directed towards Programmable Technologies with comercial VLSI Design Tools to synthesize Verilog and VHDL codes, embracing one promising line of contemporary digital electronics.

Introducción

Cuando un controlador realiza su función incorporando las técnicas de la *Lógica Difusa*, se dice que es un *Controlador de Lógica Difusa (FLC - Fuzzy Logic Controller)*. La lógica difusa es una lógica multivaluada que ejecuta una serie de operaciones propias y determinadas, estimadas como lingüísticas, que se relacionan y procesan de manera similar a la forma en que lo hace el cerebro humano, propiciando una solución a problemas no lineales. Los métodos y principios de este tipo de razonamiento en particular, se encuentran definidos en la teoría matemática provista por los *Conjuntos Difusos (Fuzzy Sets)* [1, 3, 4].

La operatividad que evidencia el funcionamiento de un FLC, constituye una aplicación real y altamente demostrativa en el campo de la *Inteligencia Computacional*, mapeando el algoritmo involucrado dentro de un *FPGA* (*Field Programmable Gate Array - Arreglo de Compuertas Programable en Campo*). La naturaleza de la arquitectura programable de estos dispositivos, así como los métodos y herramientas de diseño e implantación física, favorecen la creación de prototipos en una sola pastilla de silicio[1], a bajo costo, versátiles e inmediatos [2]. Los *Lenguajes de Descripción de Hardware (HDLs – Hardware Description Languages)*, utilizados para modelar la solución, amplían el nivel del estudio al proponer más elementos de exploración en el planteamiento original del controlador, propiciando la *Automatización* de la metodología de diseño [18, 21, 33].

La automatización del diseño sobre *Tecnologías Programables* es una rama del conocimiento aplicado, perteneciente a la electrónica digital contemporánea. Con la estandarización en 1987 de VHDL como el primer lenguaje capaz de describir hardware, el diseño de circuitos digitales de alta complejidad ha sido posible a un coste razonable, permitiendo que el diseñador alcance un gran nivel de abstracción utilizando herramientas comerciales o propias. Con la liberación estandarizada de Verilog en 1995, el diseño con HDLs se nutre aún más debido, entre otros aspectos, a la facilidad sintáctica en comparación a la propuesta por VHDL.

Los FLCs resultan ser materia de estudio constante y actual por parte de investigadores en el ámbito mundial; quienes buscan robustez, factibilidad y el acrecentamiento en el desempeño de los controladores en términos de la velocidad de procesamiento y la exactitud. La prioridad en este tipo de indagaciones recae en

[1] *Para algunos autores, esta técnica es denominada SOC (System On Chip), haciendo alusión a que es posible integrar un sistema completo en un solo circuito integrado.*

encontrar el sentido de una realización electrónica que no esté supeditada a utilizar los dispositivos existentes comercialmente sino a crear prototipos propios, lo que admite el análisis de resultados y la optimización de los mismos [1, 2, 4, 5]. Los alcances no se limitan, entonces, a sistemas de control, sino a toda la gama de ejercicios encaminados hacia los sistemas expertos y a la inteligencia artificial, bajo el rubro del diseño en hardware digital.

Definición del problema

Se han efectuado innumerables desarrollos de aplicación específica que demuestran la factibilidad de la realización electrónica de FLCs en FPGAs, obteniéndose resultados aceptables, los cuales se comentan debidamente en el **Capítulo 1**. La tendencia en este trabajo de tesis es formalizar un estudio detallado de las bases prácticas que definen la implementación de un algoritmo de lógica difusa, de aplicación completamente general. El mapeo de la solución en el FPGA, infiere arquitecturas que no sólo competen al desempeño del controlador mismo, sino a la optimización de resultados con el propósito de reducir la cantidad de *módulos lógicos* en hardware utilizados del total disponible dentro del dispositivo.

La teoría general del control de sistemas es muy basta, por lo que las aportaciones originales sucintas de este documento no se enfocan plenamente a la línea del control como tal, sino a la propia de la electrónica digital. El análisis provisto está orientado a las topologías de procesamiento serie y paralelo, como arquitecturas ambiguas y clásicas del desarrollo digital de controladores, sin desvirtuar la importancia de la correcta programación en HDLs. La descripción comportamental del controlador, codificada en Verilog y VHDL, permite que el dominio de prácticas y conocimientos, sea más general y accesible a cualquier nivel de estudio y bajo cualquier preferencia sintáctica.

Objetivo General

Diseñar un controlador digital basado en *Lógica Difusa*, con arquitectura adaptable a cualquier aplicación. Realizar la descripción de su funcionamiento utilizando *Lenguajes de Descripción de Hardware* e implementarlo en un *FPGA*.

Objetivos Específicos

1. Plantear el marco teórico del diseño de FLCs en FPGAs.

2. Demostrar que con un grupo reducido de topologías similares, es posible resolver la mayoría de los casos aplicables.

3. Detallar las soluciones alternando descripciones en VHDL y Verilog, como HDLs principales.

Justificaciones

No obstante que para la mayor parte de las aplicaciones resulta una solución satisfactoria, un sistema implementado para trabajar vía software con una computadora como lazo primario de control[2], presenta una velocidad de procesamiento lenta en comparación a la de un desarrollo en hardware que cumpla con el mismo propósito. Para sistemas muy complejos que requieren control en tiempo real y precisan un número elevado de reglas de inferencia (refiriéndose al control de lógica difusa) o cuando los sistemas a controlar tienen tiempos de respuesta muy cortos, es obligatoria una realización en hardware [2, 6].

El diseño de un controlador con arquitectura adaptable permite cubrir de manera general cualquier aplicación que infiera técnicas de control difuso. En la mayoría de los casos, se aumenta considerablemente la cantidad de hardware para reducir el tiempo de procesamiento[3], cuestión que no preocupa considerablemente, sin estimar la

[2] *A este tipo de sistema de control se le denomina Control Digital Directo (DDC – Direct Digital Control). Tuvo sus inicios a la par de la proliferación de la PC.*

[3] *Existen algoritmos matemáticos que sirven para realizar la misma función. Por ejemplo, en los sumadores la función primordial es la misma; sin embargo, en la configuración de un sumador de 8 bits conectado en cascada, se tiene un número de compuertas mucho menor que otro igual, pero en configuración Carry Look - Ahead, el cual paraleliza las operaciones, aportando mayor velocidad de procesamiento pero sacrificando espacio físico.*

optimización, dadas las características de un FPGA [18, 20]. Con la automatización del diseño VLSI, la generación de niveles lógicos envuelve alta inmunidad al ruido evitando variaciones importantes en las características de los transistores involucrados, proveyendo un procesamiento de datos exacto y confiable [23].

La *Ingeniería de Control* establece el diseño de sistemas mediante cualquiera de las técnicas ya conocidas. El llamado *Control Clásico*, su evolución hacia el *Control Moderno* y la clara predisposición al llamado *Control Inteligente*, enmarcan la participación de la ciencia teórica en la mayoría de las prácticas en manejo de procesos sin importar el campo de diligencia. Las especificaciones de desempeño[4] determinadas para el sistema en particular sugieren el método a utilizar para crear el diseño [15].

El control clásico y sus procesamientos numéricos basados en la teoría matemática diferencial, han sido una buena solución para la mayoría de los sistemas que proponen un comportamiento lineal, aunque los métodos de diseño resultan muy complejos. Los sistemas de control basados en lógica difusa han exhibido una mayor eficacia en cuanto a su funcionamiento comparándolos con los controladores clásicos[5], especialmente cuando se destinan a procesos no lineales difíciles de modelar, y cuando existe un manifiesto conocimiento heurístico por parte del operador humano. La técnica clásica no realiza actividades cognoscitivas o procesamiento de información abstracta debido a la aplicación pura del conocimiento matemático y plenamente comprobado, siendo la diferencia básica entre el controlador con técnicas de lógica difusa y el controlador clásico [16].

El **Capítulo 1** presenta el marco teórico del diseño de controladores digitales de lógica difusa, describiendo las particularidades del trabajo sobre FPGAs. El **Capítulo 2**, abarca la teoría básica de las Tecnologías Programables en general, proporcionando una idea abstracta de la metodología de diseño sugerida.

El tercer capítulo, **Capítulo 3**, describe la metodología y el desarrollo seguido para el diseño individual de las interfaces difusas. En el **Capítulo 4** se muestra la integración de las interfaces para la implementación completa del controlador, así como las pruebas de funcionamiento. Finalmente, en el **Capítulo 5** se dan a conocer las conclusiones más representativas de este trabajo de tesis, generadas de los logros alcanzados. En este mismo capítulo se exponen los trabajos a futuro.

[4] *Básicamente, en el Control Clásico las especificaciones de desempeño se pueden presentar en términos de las características de respuesta transitoria y/o las medidas de desempeño en el dominio de la frecuencia. En el Control Moderno, se presentan en términos de las variables de estado, a diferencia del Control Inteligente, donde se muestran como variables cognoscitivas.*

[5] *Cuando se menciona a los controladores clásicos, se hace referencia a los controladores diseñados mediante las técnicas del control clásico o del control moderno, que independientemente de la diferencia entre las técnicas, ambas siguen siendo lógicas bivaluadas.*

Capítulo 1

Realizaciones Hardware de FLCs

Contenido del Capítulo

Los FLCs han sido ampliamente utilizados en una gran diversidad de aplicaciones, existiendo una sustentable base teórica que advierte las principales tendencias en su realización electrónica digital.

En la investigación previa concerniente a este documento, se obtuvieron una serie de trabajos similares con respecto a la naturaleza de la línea propuesta. La mayoría de estas aportaciones comprueban la factibilidad del diseño de FLCs sobre FPGAs, limitándose solamente a la parte de la realización electrónica dirigida hacia una aplicación en particular, más no alternan con la optimización de la solución y mucho menos proponen una arquitectura genérica.

En este capítulo se presenta una revisión breve de la lógica difusa aplicada a los sistemas de control, haciendo hincapié en los elementos más importantes relacionados con las realizaciones hardware (FLCs dedicados), y denotando el contexto sobre el cual se encaminaron los esfuerzos de este trabajo de tesis (FLCs digitales). No se pretende cubrir de manera detallada la teoría difusa, por lo que se recomienda auxiliarse de bibliografía especializada para una mayor acotación de términos.

1.1 Soluciones Actuales en la Realización de Sistemas de Control Basados en Lógica Difusa

Actualmente existen excelentes referencias bibliográficas en torno a la *Lógica Difusa* encaminada a resolver problemas de *Sistemas Expertos* e *Inteligencia Artificial*. Desde 1965, con la primera publicación formal del trabajo realizado por *Lofti A. Zadeh*, la lógica difusa comenzó a tomar un auge muy particular no sólo como una nueva técnica de manipulación matemática de eventos no lineales, sino como una alternativa simple para reproducir un razonamiento humano.

No existe una estandarización de conceptos para determinar a ciencia cierta los medios para concretar físicamente un sistema de control de lógica difusa. La mayor parte de los autores, coinciden en exponer una clasificación basada en la estimación estructural del sistema de acuerdo a la participación porcentual dedicada del software y del hardware en la solución provista [9, 10].

Considerando el criterio anterior, los sistemas de control en general que aplican las técnicas de la lógica difusa, se clasifican en cuatro grupos dependientes de los elementos con los que se construye el sistema: *totalmente software, procesador con instrucciones dedicadas al tratamiento de lógica difusa, procesador apoyado por un coprocesador especializado en lógica difusa* y finalmente, *controladores difusos dedicados*; estos últimos (en su naturaleza digital) son el elemento sustancial expuesto en esta tesis y denotan realizaciones totalmente hardware. La **tabla 1.1** muestra una idea comparativa entre soluciones [2].

Tabla 1.1: **Soluciones en la realización de sistemas de control de lógica difusa.**

Realización	Características Principales
Totalmente Software	Fácil y de bajo costo. Velocidad baja
Procesador con conjunto de instrucciones dedicadas al tratamiento de operaciones difusas	Mediana velocidad de operaciones difusas y costo medio considerable
Procesador apoyado por un coprocesador con instrucciones difusos dedicadas	Alta velocidad de operaciones difusas, costo moderado. El protocolo de intercambio de información dificulta el manejo de datos y la generalidad
Controlador dedicado al tratamiento de operaciones difusas	Bajo, medio o alto costo, dependiendo de la experiencia del diseñador y de la complejidad del controlador. Poca generalidad, alta velocidad. Soporte a múltiples arquitecturas

1.1.1 Totalmente Software (Full Software)

Esta tendencia de realización indica que a partir de una computadora personal y más a fondo, de cualquier procesador de propósito general, se utiliza un lenguaje de programación formal para crear el sistema totalmente vía software. Existe en el mercado un respaldo muy respetable de herramientas que permiten crear sistemas completos de manera simple e inmediata, que van desde las aplicaciones clásicas de control hasta los sistemas expertos dedicados al manejo de bases de datos.

Existe en el mercado, software propietario que cumple con el apoyo a todas las fases del desarrollo: *diseño, simulación, optimización, verificación* e *implementación*. Evidentemente los recursos de cómputo necesarios para implementar sistemas mediante este método son más baratos e inmediatos que los de cualquiera de los otros. La gran desventaja que tienen los sistemas creados bajo este concepto es que la velocidad de procesamiento es muy lenta, a causa de que se está utilizando un procesador de propósito general cuyo conjunto de instrucciones no incluye algunas dedicadas para las operaciones difusas, además de que la programación de eventos es necesariamente serial[1]. El número de reglas de inferencia que pueden evaluarse en determinada unidad de tiempo es característica primordial para considerar el desempeño de un sistema difuso. Aspecto por el cual, si se considera la posibilidad de tener una cantidad muy grande de reglas difusas en un sistema full software, es inminente que el tiempo de procesamiento también será muy grande debido a la ejecución secuencial en la evaluación de las reglas.

1.1.2 Procesador con Instrucciones Difusas Dedicadas

Este segundo concepto en las realizaciones, surge como una alternativa para mejorar el desempeño de los sistemas full software. En éste, el conjunto de instrucciones del procesador de propósito general incluye algunas dedicadas para las operaciones difusas. Las instrucciones incluidas en este caso fueron las clásicas *MAX* y *MIN,* con la intención de desahogar un poco el grueso del procesamiento cuando se evalúan las reglas difusas. Aún así, la ejecución sigue siendo prácticamente secuencial.

Es posible optar por la denominada *programación paralela*, que aborda todo un campo de estudio dentro de las aplicaciones software encaminadas a paralelizar

[1] *Con respecto al caso que se presenta de manera más común, recordando que puede haber programación paralela basada en recursos con arquitectura dispuesta para soportar este tipo de procesamiento.*

procesos; sin embargo, las tendencias evolutivas en la programación de la inferencia de las reglas difusas, no permiten un paralelismo vía software[2] que permita evaluar de manera independiente cada una de las reglas [14, 17].

1.1.3 Procesador Apoyado por un Coprocesador de Lógica Difusa

El coprocesador difuso es un dispositivo hardware que realiza operaciones difusas especializadas y que trabaja bajo el mando de un procesador de propósito general. Esta aproximación aumenta la velocidad del sistema, ya que algunas operaciones como la evaluación de las reglas y la comparación entre tablas, son ejecutadas por el coprocesador, quien a su vez entrega los resultados de su ejercicio al procesador. El inconveniente directo en este tipo de realizaciones, es que el sistema se encuentra restringido por el número de entradas y salidas entre ambos dispositivos, requiriéndose protocolos de intercambio de información y un sofisticado control de sincronización; razones suficientes para considerar que no se trata de una solución óptima.

1.1.4 Controlador de Lógica Difusa Dedicado

Se trata de realizaciones hardware (comerciales o no) de muy alta velocidad de procesamiento, pero con el inconveniente de ser casi totalmente de propósito específico para cada aplicación. Los costos de investigación y producción suelen ser relativamente altos, especialmente cuando se decide por una implementación analógica, debido a que conlleva al uso de metodologías distintas y más complicadas de desarrollo.

Como se ha venido mencionando a lo largo de este documento, el total de entradas, salidas y el número de reglas que puede evaluar un dispositivo, son factores que determinan el desempeño de un procesador difuso. Tecnológicamente hablando, se dice que entre más entradas y más salidas se tengan, el procesador se vuelve más general aunque internamente las operaciones aumentan considerablemente, por lo que la arquitectura y sus mecanismos de operación matemática, deben ser

[2] *Como se comentará posteriormente, en hardware es totalmente realizable.*

cuidadosamente diseñados para evitar que el desempeño se vea afectado sacrificando velocidad por versatilidad.

Si se trata de utilizar microcontroladores provenientes de fabricantes, se dispone de software de soporte que crea el ensamblador para programarlos directamente orientados al trabajo con lógica difusa (*control ordinario* o *redes neuronales*), tal y como sucede con *FuzzyTECH* , *NeuroFuzz* y otros programas comerciales [1, 12]. El funcionamiento de los microcontroladores de propietario más actuales (*National Semiconductor*, *SGS – Thompson Microelectronics*, *Mitsubishi Electric Corporation*, *Texas Instruments*, *Togai, Siemens, OmRom*, entre otros), es aceptable en términos de trabajo definido más no en generalidad, puesto que tienen un número restringido de entradas y salidas, así como de reglas de inferencia, además de que el procesamiento en la mayoría de estos dispositivos es secuencial debido a su arquitectura diseñada para competir comercialmente buscando un punto intermedio entre costo y desempeño.

A nivel más especializado es posible encontrar comercialmente software sofisticado que permite modelar arquitecturas de lógica difusa automáticamente sobre FPGAs (*FuzzyPL, FuzzyShell,* entre otras). Estas herramientas generan código en algún HDL (Verilog o VHDL, principalmente) que modela la función de un controlador o de una red neuronal, con los inconvenientes de que no optimizan el diseño, el número de variables y reglas de inferencia está limitado, tienen pocas opciones en la forma de las funciones de pertenencia y su precio es muy alto. Con la evolución de las soluciones en software, estas herramientas serán motivo de análisis consistente en el futuro.

1.2 La Lógica Difusa Aplicada a los Sistemas de Control en Realizaciones Hardware

La lógica en la cual se apoya el control clásico define dos valores aceptados y restringidos que no conciben ninguna consideración intermedia: *falso* o *verdadero* [34]. Estos a su vez, aplicados a diferentes contextos solamente cambian de nombre, pero en esencia tienen el mismo sentido y aplican la misma norma: *frío* o *caliente*, *alto* o *bajo*, *prendido* o *apagado*, *rápido* o *lento*, *0* ó *1.* La *Lógica Booleana*[3] no es suficiente para analizar eventos de la vida real en la que se deben considerar valores intermedios.

[3] *La lógica booleana es una forma de razonamiento propia de los sistemas bivaluados, ya que utiliza sólo dos valores discretos para representar sus operaciones.*

En 1965, en la *Universidad de California*, *Lofti A. Zadeh* publicó el resultado de un trabajo matemático al que tituló *Conjuntos Difusos (Fuzzy Sets),* donde expone una forma de razonamiento novedosa a la que nombró *Lógica Difusa,* que es básicamente una lógica multivaluada que permite a valores intermedios ser definidos entre valores convencionales. Este tipo de lógica está basado tradicionalmente en reglas que se determinan al azar dependiendo de la experiencia del operador humano. En otros casos, se usan sistemas de determinación y optimización de relaciones por medio de *algoritmos genéticos* o bien, *redes neuronales* [7, 8].

La lógica difusa en vez de definir un evento como 100% falso o verdadero, lo concibe como parcialmente falso o verdadero, es decir, otorga a un evento *grados de pertenencia* (membresía), permitiendo así un manejo más concreto de situaciones. El grado de pertenencia de un elemento a un conjunto puede tener infinitos valores dentro del rango *[0, 1]*. Por tanto, si x es el dominio (discreto o continuo) de todos los elementos, un conjunto difuso cualquiera A está completamente definido por la función de pertenencia

$$\mu_A : x \rightarrow [0,1] \tag{1.1}$$

donde $\mu_A(x)$ es el grado de pertenencia del elemento x al conjunto A.

Similarmente a la manera como se hace extensible la teoría clásica de conjuntos hacia el campo de la lógica bivaluada, es posible definir el campo de acción de la lógica difusa y, en vez de hablar de grados de pertenencia de un conjunto, se habla de grados de verdad de una proposición lógica.

El funcionamiento del control se emula por medio de reglas en lenguaje aproximado humano, que no son más que expresiones formuladas cotidianamente que culminan en una acción. Así, el controlador de lógica difusa en capaz de evaluar el grado de verdad de una proposición lógica (*premisa* o *antecedente*) formulada a partir de las variables de estado de un sistema y posteriormente inferir el grado de verdad de otra proposición utilizada como salida (*conclusión* o *consecuente*), cuyo valor será reducido a escalar, provocando una acción sobre el sistema bajo control.

La sistemática de trasladar el conjunto experto de las reglas a una estrategia de control, con conjuntos y lógica difusa, se conoce con el nombre de *Control de Lógica Difusa* o simplemente *Control Difuso (FLC – Fuzzy Logic Controller)*. Esta innovativa tecnología mejora el diseño de sistemas convencionales por medio de *Ingeniería Experta*[4].

[4] *La Ingeniería Experta se basa en la experiencia humana. La forma en la que es operado un sistema cualquiera por un humano, se toma como guía para sistemas independientes.*

En la **tabla 1.2**, se puede observar una comparación simple entre las principales características del *Controlador Clásico* y un FLC.

Tabla 1.2: **Comparación entre controladores.**

Característica Comparativa	Controlador Clásico	FLC
Diseño	A través de modelo matemático y bajo técnicas conocidas. Baja flexibilidad de diseño ante variables múltiples	A través de un modelo heurístico. Alta flexibilidad de diseño ante variables múltiples
Respuesta	Depende de la sintonización: cálculo de las constantes de *Ganancia, Reset* y *Rate*	Depende de la base de conocimiento (base de reglas difusas)
Confiabilidad de respuesta a cambios drásticos en las condiciones de operación	Baja	Alta (siempre y cuando la base de conocimiento esté bien planteada)
Velocidad de procesamiento	Aceptable	En función de la arquitectura del controlador
Control no lineal	Es posible, aunque con problemas tácitos y requiere la ayuda de componentes adicionales	Total
Robustez	En casos limitados	Demasiada, debido a la naturaleza del proceso difuso

1.2.1 Estructura del FLC Dedicado

El diagrama a bloques de un controlador difuso dedicado se muestra en la **figura 1.1**. El sistema consta de cuatro módulos básicos: La interfaz de fuzificación, un mecanismo de inferencia que interactúa con una base de reglas difusas (juntas, forman la interfaz de inferencia) y la generación de valores reales o interfaz de defuzificación. En el **Capítulo 3** de este trabajo de tesis, se comenta detalladamente el diseño de cada interfaz, por lo que la revisión dada en esta sección no revela pormenores más técnicos.

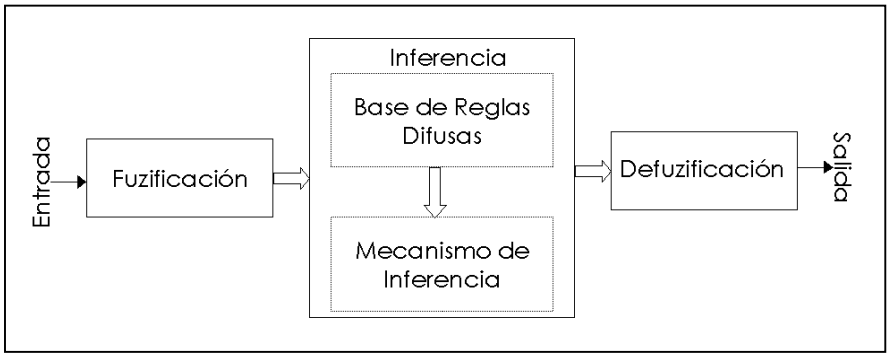


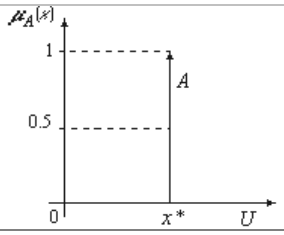
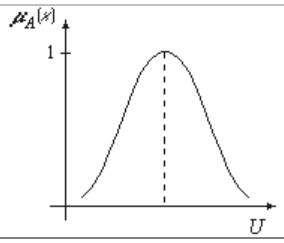
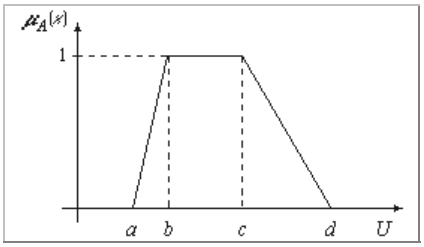
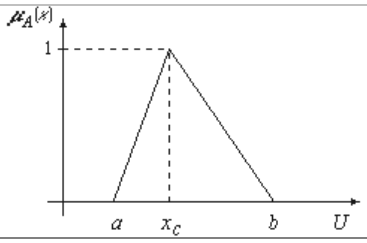
Figura 1.1: **Estructura básica del controlador de lógica difusa dedicado.**

1.2.1.1 Interfaz de Fuzificación

La interfaz de fuzificación, asocia a cada entrada su grado de pertenencia a un conjunto difuso. Este bloque toma los valores numéricos, conocidos como valores *frágiles (crisp)* de las entradas reales, que para el caso del controlador son escalares, y les asigna un grado de pertenencia respecto de una función previamente definida heurísticamente por el experto humano, generando en su salida valores difusos (fuzificados). El universo de los valores se divide en *funciones de pertenencia* a las cuales se identifica mediante etiquetas representativas de la variable. Entre mayor es el número de etiquetas, se tiene un grado mayor de resolución. Las funciones de pertenencia dan significado numérico a cada etiqueta e identifican el rango de valores de entrada que corresponden a una etiqueta.

Las formas más usuales de las funciones de pertenencia son cuatro: *singleton, gaussiano o campana*, *trapezoidal* y *triangular*, los cuales se muestran en la **tabla 1.3**, en donde x^* representa un valor numérico real, A es un conjunto difuso que está contenido en el conjunto x, x_c es el valor de x donde la función es máxima, σ es la desviación estándar y $\mu_A(x)$ es la función de pertenencia o de membresía. En la literatura especializada a estas formas gráficas también se les conoce como fuzificadores.

Tabla 1.3: **Tipos de fuzificadores.**

Tipo de Fuzificador	Forma de la Función de Pertenencia	Cálculo del Grado de Pertenencia
Singleton		$\mu_A(x) = \begin{cases} 1, & x = x^* \\ 0, & x \neq x^* \end{cases}$
Gaussiano		$\mu_A(x) = e^{\frac{-(x^*-x_c)^2}{2\sigma^2}}$
Trapezoidal		$\mu_A(x) = \begin{cases} 1, & \text{si } b \leq x^* \leq c \\ \left(1 - \frac{\lvert b - x^* \rvert}{b - a}\right), & \text{si } a < x^* < b \\ \left(1 - \frac{\lvert x^* - c \rvert}{d - c}\right), & \text{si } c < x^* < d \\ 0, & \text{otro} \end{cases}$
Triangular		$\mu_A(x) = \begin{cases} 1, & \text{si } x^* = x_c \\ 1 - \frac{\lvert x_c - x^* \rvert}{x_c - a}, & \text{si } a < x^* < x_c \\ 1 - \frac{\lvert x_c - x^* \rvert}{b - x_c}, & \text{si } x_c < x^* < b \\ 0, & \text{otro} \end{cases}$

1.2.1.2 Interfaz de Inferencia

La interfaz de inferencia consta de dos módulos que interactúan entre sí: la *base de reglas difusas* y el *mecanismo de inferencia* (referirse a la **figura 1.1**).

1. *Una base de reglas difusas (Fuzzy Rules Base)* consiste en un conjunto de reglas *Si – Entonces (en inglés, If - Then)* de la forma:

$$\textbf{Si } x_1 \textbf{ Es } A_1 \textbf{ Y } x_2 \textbf{ Es } A_2 \textbf{ Y } \ldots x_n \textbf{ Es } A_n \textbf{ Entonces } z \textbf{ Es } B \qquad (1.2)$$

donde A_n y B, representan los valores lingüísticos definidos por los conjuntos difusos en el universo de discurso. Las reglas modelan el funcionamiento del sistema, por lo que el planteamiento correcto de la base de reglas de acuerdo a la experiencia del operador humano, es factor fundamental para el comportamiento satisfactorio del controlador. Por lo mismo, a la base de reglas también se le conoce como *Base del Conocimiento*.

La base de reglas debe cumplir las siguientes propiedades:

a. *Completa*. Para cualquier combinación de valores de entrada se obtiene un valor apropiado de salida. Se produce el total cubrimiento del espacio de entradas.

b. *Consistente*. No se presentan contradicciones. No existen dos reglas con idénticos antecedentes paro distintos consecuentes.

c. *Continua*. Reglas contiguas tienen funciones de pertenencia en los consecuentes con intersección nula.

2. *La máquina o mecanismo de inferencia difusa* emplea la información almacenada en la base de reglas difusas para determinar la función de pertenencia de los antecedentes, el grado de activación de las funciones de pertenencia del dominio de entrada, y de esta forma obtener la función de pertenencia de los consecuentes proporcionando una única salida difusa. Para obtener la salida, la máquina de inferencia combina las diferentes reglas de composición de acuerdo a lo estipulado en la teoría de conjuntos difusos. Así, la expresión **Si** x **Es** A **Entonces** z **Es** B, es abreviada $A \rightarrow B$, describiendo una relación entre dos variables, lo que sugiere que esta regla puede ser definida como una relación R en el espacio entrada-salida AxB. Las dos principales formas de obtención de las funciones de pertenencia de los antecedentes y de interpretación de la sentencia de conexión, y/o grado de activación del dominio de entradas, en la regla difusa $A \rightarrow B$ son:

a. *Método del mínimo* o *de Mamdani.* Se almacena en una relación difusa R donde se elige la función de pertenencia mínima:

$$\mu_R(x,z) = \min\{\mu_A(x), \mu_B(z)\} \tag{1.3}$$

o bien,

$$A \times B = \int_{x \times z} \left(\mu_A(x) \wedge \mu_B(z)\right) / (x,z) \tag{1.4}$$

b. *Método del producto o de Larsen*. Se almacena en una relación difusa R, que realiza el producto de las funciones de pertenencia respectivas:

$$\mu_R(x, z) = \mu_A(x)\mu_B(z) \tag{1.5}$$

Ambas correlaciones aplican posteriormente el método del *Máximo*, para encontrar un valor resultante total para referenciar una sola función de pertenencia del consecuente. El método más comúnmente utilizado para realizar la inferencia en hardware es el de Mamdani, debido a que es más simple construir unidades que realicen el mínimo y sucesivamente el máximo, que multiplicadores dedicados. Por la naturaleza del proceso realizado por el método Mamdani de inferencia, también es conocido simplemente como método *MAX – MIN*.

1.2.1.3 Interfaz de Defuzificación

El bloque defuzificador genera su respuesta a partir de que la variable lingüística difusa resultante sale de la interfaz de inferencia. La interfaz de defuzificación, será la encargada de generar el escalar que es el valor real que se entregará a la salida del controlador.

Son varios los métodos actuales para generar la salida escalar equivalente al conjunto difuso de inferencia. Tanto para un sistema difuso *MISO (Entradas Múltiples, Salida Única)* como para uno *MIMO (Entradas Múltiples, Salidas Múltiples)*, los razonamientos son análogos considerando que la inferencia se realiza mediante el método MAX – MIN. Así, los principales métodos de defuzificación más aplicados son:

1. *Defuzificador del Área o Centro de Gravedad (COG).* La variable Z^* representa la salida escalar y puede ser cualquier punto en la altura B (conjunto difuso de salida, proveniente de la inferencia). En la **ecuación 1.6** se muestra la relación matemática que genera la salida escalar. En esta ecuación $\mu_B(Z)$ es la función de pertenencia, asociada al conjunto difuso de salida, resultante de aplicar el mecanismo de inferencia al conjunto de reglas difusas.

$$Z^* = \frac{\int_v \mu_B(Z) Z dZ}{\int_v \mu_B(Z) dZ} \tag{1.6}$$

2. *Defuzificador del Promedio de Centros.* En este defuzificador, también conocido como *Centro de Sumas*, el valor escalar de la variable de salida Z^* está determinado por el promedio de los centros de los M conjuntos difusos de salida con los pesos ω, siendo igual a la altura de los conjuntos difusos correspondientes. La **ecuación 1.7** exhibe la relación matemática aplicable. A su vez, este método es el más utilizado para la realización en hardware.

$$Z^* = \frac{\sum_{i=1}^{n} \mu_{Bi}(\bar{Z})\omega_1}{\sum_{i=1}^{n} \mu_{Bi}(\bar{Z})} \tag{1.7}$$

3. *Defuzificador de Criterio del Máximo.* La salida escalar es el valor numérico del punto donde la función de pertenencia del conjunto difuso de la salida asume su valor máximo. Si la altura B contiene un solo punto, se define Z^* en forma única; si por el contrario contiene más de un punto, se pueden especificar tres defuzificadores de naturaleza diferente:

a. *Defuzificador del Valor Más Pequeño del Máximo.* Este método indica que se debe seleccionar como escalar Z^* de salida, al menor valor (primero) de la mayor función de pertenencia consecuente inferida.

b. *Defuzificador del Valor Más Grande del Máximo.* Se debe seleccionar como escalar Z^* de salida, al mayor valor (primero) de la mayor función de pertenencia consecuente inferida.

c. *Defuzificador del Valor Promedio del Máximo*. Esta aproximación requiere que se elija como escalar Z^* de salida, el valor promedio de la mayor función de pertenencia consecuente inferida, tal y como lo sugiere la **ecuación 1.8**.

$$Z^* = \frac{\int_{altura(B)} ZdZ}{\int_{altura(B)} dZ} \tag{1.8}$$

1.3 Principales Realizaciones Hardware de FLCs

Los primeros trabajos encaminados a vincular la lógica difusa con los diferentes entornos de aplicación en casos prácticos, no necesariamente sistemas de control, surgieron en el momento en que la base teórica de los *Conjuntos Difusos* fue considerada sustentable por la comunidad tecnológica y científica. En 1969, *W. Marinos* de la *Duke University*, conduce las primeras investigaciones sobre implementaciones electrónicas de la lógica difusa. Las limitaciones comunes de la época no desmerecieron la importancia práctica de la nueva técnica, respetando aún en nuestros días las restricciones en respuesta de los componentes adicionales al sistema; es decir, una situación es crear dispositivos electrónicos rápidos que realicen el procesamiento y otra es que los elementos mecánicos presenten una respuesta inmediata [3].

En 1974, *E. H. Mamdani* presentó la primera aplicación de la lógica difusa en un sistema de control, resultando lo que se denominó *Control Difuso (Fuzzy Control)*. Mamdani planteó la posibilidad de utilizar componentes discretos con la finalidad de realizar un proceso de inferencia manipulando valores de resistencias para variar corrientes, aportando una nueva técnica para el tratamiento electrónico.

En 1985, *Togai y Watanabe* de *AT & T Bell Laboratories,* desarrollaron la primera máquina de inferencia difusa, la cual fue construida sobre un circuito integrado *VLSI* con una entrada y una salida de 4 bits. Este integrado ejecutaba hasta 250, 000 *FRPS*[5] *(Fuzzy Rules Per Second – Reglas Difusas Por Segundo)* sin defuzificación[6]. Los alcances se limitaron a sistemas de control muy básicos ya que resolvían solamente funciones de membresía triangulares clásicas, con un número muy reducido de reglas de inferencia, así como topologías seriales basadas en un modelo de memoria de tabla de datos.

En 1986, los *Hitachi Corporation Laboratories* desarrollaron un controlador de lógica difusa dedicado a ajustar la temperatura de un horno, cuyas características eran más generales que su predecesor, comenzando con un soporte mayor de entradas y reglas de inferencia, pero restringido a una sola salida [2].

T. Yamakawa, en 1987 presentó el primer controlador difuso completamente analógico en un chip VLSI, integrando algunos circuitos dedicados *análogos – bipolares*. Internamente este procesador exhibió por primera vez circuitos simples que realizaban las funciones *MAX y MIN,* logrando velocidades de hasta *un millón de FIPS*

[5] *En el campo de los procesadores difusos, la principal característica que diferencia las arquitecturas, es la velocidad en términos de las reglas difusas (FRPS) o de las inferencias (FIPS), evaluadas por segundo.*
[6] *Si no hay defuzificación, las salidas siguen siendo difusas, por lo que se tenía la necesidad de estimar mediante un acoplamiento independiente, algún método para defuzificar los valores y encontrar las salidas reales.*

(Fuzzy Inferences Per Second - Inferencias Difusas Por Segundo) y una unidad defuzificadora basada en el método del *Centro de Gravedad (COG - Center Of Gravity)*. Se tenía la opción de utilizar el controlador sin la unidad de defuzificación, obteniendo en este caso hasta *10 MFIPS*.

En 1989, *Togai y Watanabe,* complementaron sus investigaciones en el campo del diseño VLSI, presentado el primer *controlador digital basado en lógica difusa* con distribución comercial, que no era totalmente adaptable a cualquier sistema, pero que superaba por mucho la gama de aplicaciones y desempeño propuesto por los antecesores. Este controlador adoptaba la *suma lógica de mínimos* y el *producto lógico de máximos* como operadores de unión y de intersección respectivamente. Este circuito integrado trabajaba a *36 MHz*, y traducía cerca de *580,000 FIPS*, con un conjunto aceptable de aproximadamente 102 reglas de inferencia.

A la par de estos logros electrónicos, las herramientas de automatización de diseños, refiriéndose específicamente a las propias de las tecnologías programables, comenzaron el despunte hasta alcanzar niveles inimaginables. A partir de 1997 comienza una serie de trabajos dirigidos hacia el control digital bajo mecanismos inteligentes utilizando como base para el diseño del hardware, arreglos lógicos programables y lenguajes de descripción de hardware, tal y como lo demuestran los siguientes artículos encontrados.

1. *"Implementing Fuzzy Control Systems Using VHDL y Statecharts"*, es un artículo escrito por *Valentina Salapura* y *Volker Hamann*, de la *Technische Universität Wien* de *Austria* [55]. Este documento aborda la implementación de un sistema de control difuso muy sencillo, pero que detalla una forma muy interesante de atacar la problemática y que es en sí la misma idea de este trabajo de tesis: separa los procesos de fuzificación, inferencia y defuzificación, en tres bloques independientes. El análisis de la base de reglas del controlador se realiza utilizando *Cartas de Estados (Statecharts)*. Todo el desarrollo se concreta mediante *VHDL* (*VHSIC HDL, Very High Speed Integrated Circuit HDL – Circuito Integrado de Muy Alta Velocidad modelado con HDL)*; la parte de la base de reglas, se realiza primeramente en *SPeeDCHART*[7] para posteriormente utilizar el sintetizador de VHDL y convertir todo a código.

2. *"The Programmable Fuzzy Logic Array"*, encontrado en *Internet* en la página electrónica de *Texas Instruments* [56], es un artículo escrito por *Philip Thrift* del *Central Research Laboratories, Texas Instruments Incorporated*. El documento

[7] *Se trata de una herramienta gráfica que permite trazar diagramas de estados en máquinas de estados finitos. Tiene una interfaz con VHDL - based logic Synthesis, que convierte los diagramas dibujados a código VHDL.*

describe un dispositivo programable para lógica difusa, el cual cuenta con su propio ambiente de programación.

El Arreglo de Lógica Difusa Programable (*PFLA - Programmable Fuzzy Logic Array)*, tal y como fue nombrado por su creador, produce mapeos multidimensionales no lineales implementando reglas de lógica difusa sobre una arquitectura de arreglos programable. Estudiando la información suministrada, se encontraron varias desventajas: se trata de un dispositivo con recursos comprometidos, como en el caso de un *PLD*, lo que restringe el diseño; por otra parte, sólo realiza la inferencia difusa manipulando las reglas lógicas, lo que implicaría rediseñar en base a mecanismos de control para que sirviera a propósitos más elaborados y generales. No considera ni la fuzificación ni la defuzificación de las variables, pero exhibe una forma inteligente de resolver la inferencia trazando un arreglo.

1.3.1 Clasificación de los FLCs Dedicados

Por sí mismo, el proceso difuso es de carácter serial. Está demostrado que mínimamente debe operar con una división de tres interfaces o módulos que se ejecutan de manera secuencial y que no cambian de orden: fuzificación, Inferencia (integrando la base de reglas y el mecanismo de inferencia) y defuzificación. Cada etapa depende de su predecesora; por ejemplo, no es posible evaluar las reglas de inferencia si aún no se ha realizado la fuzificación de las variables. La optimización del proceso difuso no se encuentra en la forma de organizar la arquitectura en la interconexión de las tres particiones básicas, que como ya se advirtió es invariable, sino en la forma en que cada módulo de la división funcional procesa los datos respectivos. Lo anterior dirige la discusión a afirmar que no es posible paralelizar el proceso completo (debido a la dependencia entre módulos funcionales), pero sí es posible paralelizar las operaciones que cada uno de los módulos realiza internamente, aspecto que se comenta más a detalle en el **Capítulo 3** de esta tesis.

La realización en hardware de los controladores difusos dedicados se clasifica en dos grandes grupos: controladores *dependientes de la naturaleza de la señal tratada* y controladores *supeditados por la estructura física interna*. A continuación se incluye una breve explicación de los elementos que caracterizan cada grupo.

1.3.1.1 Controladores Dependientes de la Naturaleza de la Señal Tratada

De acuerdo a la naturaleza de la señal que procesan, los controladores pueden construirse mediante estrategias de hardware *analógico, digital,* o un *híbrido (mixto)* de ambos [2, 15].

1. *Analógicos.* Si se parte de la idea de que un controlador (hablando en términos generales) recibe y entrega señales provenientes de un mundo analógico cuyo comportamiento es lineal y continuo, no hay más que reconocer que no hay mejor procesamiento que el realizado por componentes de la misma naturaleza. Las realizaciones analógicas son muy eficientes en cuanto al área de silicio que ocupan, no necesitan interfaces AD/ DA con el mundo real y son capaces de obtener muy buenos resultados cuando se realizan estructuras de procesamiento paralelo[8]. Sin embargo, suelen tener un número reducido de entradas, salidas y reglas difusas. Además, la precisión del cálculo analógico presenta limitaciones tecnológicas como el desapareamiento entre transistores y las variaciones de los parámetros del proceso de fabricación, por lo que son muy sensibles al ruido y a las interferencias.

2. *Digitales.* Las realizaciones de este tipo son más robustas (permiten más variables de entrada, salida y reglas difusas) y fáciles de diseñar, así mismo presentan una programación más simple. En contraparte, ocupan más área de silicio para realizar operaciones dedicadas como la multiplicación o la división y requieren convertidores AD/ DA para comunicarse con el exterior.

3. *Híbridos.* La implementación híbrida, utiliza las mejores propiedades de las dos anteriores: utiliza elementos digitales para controlar la circuitería analógica, así como para el almacenamiento de datos y utiliza elementos analógicos para realizar los procesamientos debidos. Las herramientas de diseño son más costosas y sofisticadas.

2.3.1.2 Controladores Supeditados por la Estructura Interna

Esta clasificación de controladores, está enfocada directamente a las realizaciones digitales e híbridas [2, 4]. En el contexto particular de este trabajo de tesis, se cataloga como propia de la realización digital.

[8] *En la actualidad, el camino que persiguen los investigadores enfocados a las realizaciones analógicas, es mejorar el procesamiento de la información. Existen dos derivaciones: utilizando amplificadores operacionales (modo tensión) y utilizando amplificadores de transconductancia (modo corriente).*

Las implementaciones pertenecientes a este grupo presentan arquitecturas basadas en la forma interna en la que obtienen sus datos para comenzar las operaciones especializadas: si provienen de tablas con valores de fuzificación y defuzificación precalculados, o si se obtienen en el mismo instante con ayuda de unidades aritméticas en hardware. En este sentido se tienen dos tipos de modelos a seguir: *Modelo Orientado a Memoria* y *Modelo Orientado a Cálculos*.

1. *Modelo Orientado a Memoria.* También llamado *Modelo de Obtención Indirecta de Datos Difusos*, presenta la particularidad de utilizar tablas precalculadas de valores, almacenadas en una memoria que puede ser externa o interna al controlador. Los datos pueden ser calculados directamente por el diseñador o como en la mayoría de los casos (incluidas las versiones comerciales) se utiliza un software que realiza los cálculos y crea el ensamblador que configura las características que cumplirá el controlador de acuerdo a los requerimientos propuestos por el diseñador. La **figura 1.2** esquematiza un controlador orientado a memoria.

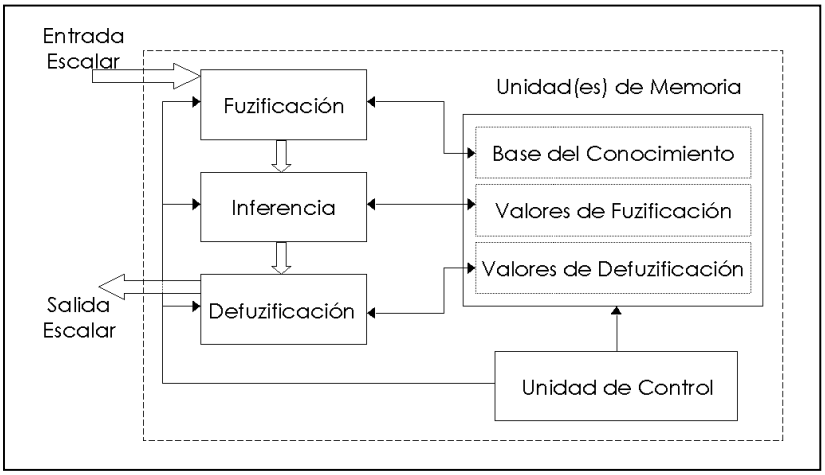


Figura 1.2: **Controlador con modelo orientado a memoria**.

La utilización de un dispositivo de memoria para almacenar las tablas precalculadas implica el diseño de un controlador de memoria y a la vez de un sistema de sincronización por ciclos para la obtención de cada valor. La velocidad de acceso para lectura y escritura, es una limitante física en este tipo de modelos; aunado a que los datos se van obteniendo uno a uno a la vez. Para sistemas mecánicos simples, este tipo de modelo es una buena opción dado que la gran mayoría de los controladores difusos dedicados se diseñan de acuerdo a esta orientación, así el mercado actual contempla un soporte muy importante en lo concerniente al software y dispositivos de propietario.

2. Modelo Orientado a Cálculos. La **figura 1.3** muestra un controlador de este tipo; también se conoce como *Modelo de Obtención Directa de Datos Difusos* y tiene la característica de utilizar unidades aritméticas construidas en software o hardware[9], que realizan el cálculo a medida que se tiene un valor a procesar. La estimación de las arquitecturas orientadas a este modelo necesitan de algoritmos aritméticos óptimos en la implementación de cada una de las unidades de cálculo. Este modelo no requiere de cálculos previos y específicamente es el tipo de orientación que se utilizó en esta tesis.

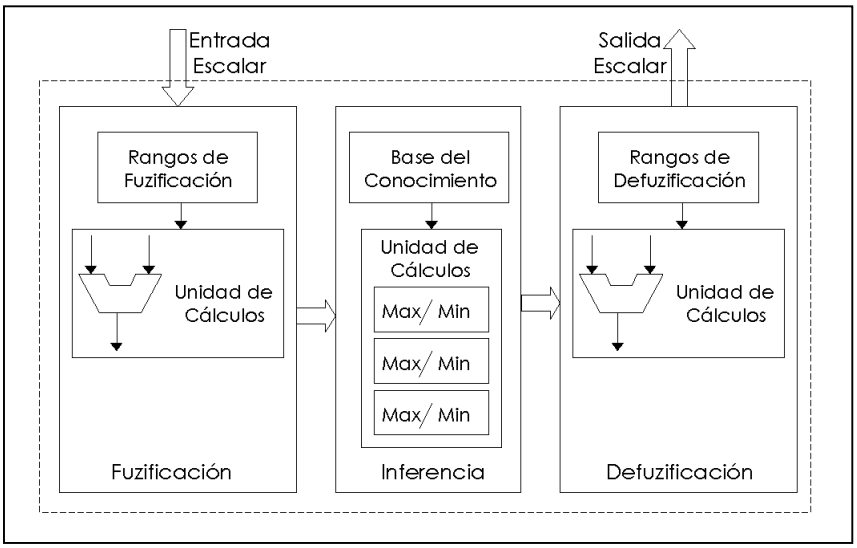


Figura 1.3: **Controlador con modelo orientado a cálculos**.

Resumen del Capítulo

En este primer capítulo, se presentó una revisión sucinta a la realización hardware de los controladores de lógica difusa, planteando un marco teórico sobre la lógica difusa encaminada al control.

En el siguiente capítulo, **Capítulo 2**, se da una breve introducción a las Tecnologías Programables en general, con la intención de analizar la tecnología de implantación sugerida para el controlador.

[9] *La mayoría de las bibliografías refiriéndose al Modelo Orientado a Cálculos, tratan indiferentemente la implementación de las unidades aritméticas vía software o hardware; sin embargo, de manera muy particular y de acuerdo al análisis de acceso a memoria, la implementación de una unidad aritmética en software nos conduce al caso de un modelo orientado a memoria más que a uno orientado a cálculos.*

Capítulo 2

Generalidades de las Tecnologías Programables

Contenido del Capítulo

El contexto de las Tecnologías Programables resulta primordial en soluciones que infieren características configurables y reconfigurables, ya sea para optimizar un diseño o para variar los parámetros del mismo. La realización de sistemas electrónicos vertida sobre FPGAs es todo un campo tecnológico en el diseño digital moderno.

En este apartado sólo se describe un panorama muy general sobre estas tecnologías. Independientemente de su operatividad programable, un dispositivo de esta naturaleza no deja de ser un circuito integrado, por lo que también existen otras características que no se asumen a profundidad en esta sección como son el retardo de propagación, la inmunidad al ruido y la disipación de potencia.

El origen de este resumen es inherente al desarrollo del trabajo de tesis, debido a que para considerar cumplidos los objetivos específicos planteados, es imprescindible el hecho de conocer el marco teórico que envuelve el uso de estas herramientas para aplicar correctamente la metodología de diseño y crear soluciones no sólo funcionales, sino óptimas y genéricas.

2.1 Fundamentos Teóricos sobre ASICs

Un *ASIC* (*Application Specific Integrated Circuit - Circuito Integrado de Aplicación Específica*) es un dispositivo terminal. Un diseño que ha sido probado y optimizado, se manufactura en una pastilla de silicio cuya operación será prefijada. Sólo contiene las interconexiones y elementos necesarios para realizar la función específica deseada.

De manera general, un ASIC contiene dos elementos básicos: M*ódulos Lógicos* e *Interconexiones* (referirse a la **figura 2.1**). Los módulos lógicos[1] (arreglos lógicos fijos o bloques configurables, dependiendo de la tecnología del dispositivo programable) contendrán una configuración elemental para generar una función lógica dada. Por su parte, las interconexiones permiten el enlace entre módulos lógicos básicos para crear módulos más complejos que realicen operaciones más elaboradas, a esta acción se le conoce como *enrutado* [18].

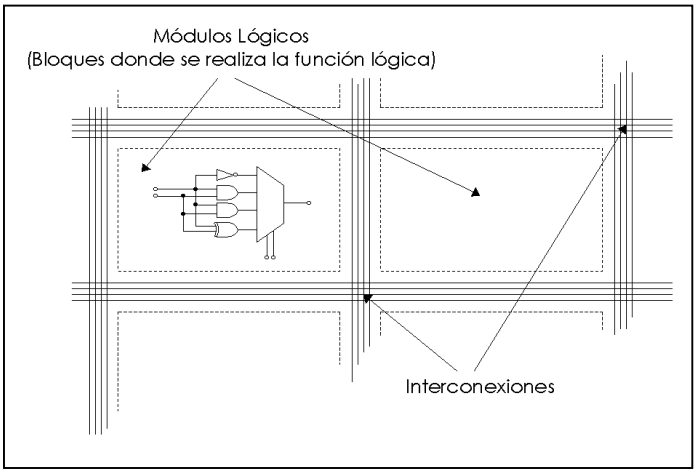


Figura 2.1: **Esquema básico que muestra la arquitectura general, no detallada, de un ASIC.**

Los ASICs se clasifican en tres grupos característicos: *Full – Custom (arquitectura a la medida)*, *Semi - Custom (arquitectura parcialmente a la medida)* y *Programmables (arquitectura programable)*. La programación, tanto para los ASIC Full – Custom como para los Semi – Custom, se realiza directamente en fábrica. Los Programmables ASICs pueden programarse por un usuario final, por lo que se dice que un ASIC de este tipo es un dispositivo *Programable en Campo*. Posteriormente se comentará a detalle esta particularidad.

[1] *Son conocidos como unidades básicas de desarrollo, y son las encargadas de realizar las operaciones lógicas definidas por el diseñador.*

2.2 ASIC Full - Custom

Para los *ASIC Full Custom*, los módulos lógicos y los enrutamientos, se construyen hasta el más mínimo detalle de distribución de acuerdo a las especificaciones que el usuario requiere para su diseño. Es decir, se crean módulos y enrutamientos propios para *personalizar* (*Custom*) o trazar a la medida un dispositivo, por lo que es necesario manufacturar en fábrica.

De acuerdo a la evolución no sólo de tecnologías sino de conceptos, el ASIC Full – Custom, tradicionalmente llamado *MPGA (Mask programmable Gate Array - Arreglo de Compuertas Programable por Máscara*) consiste en una serie de transistores que pueden conectarse entre sí de acuerdo a los requerimientos del diseñador, para crear funciones lógicas simples o complejas. La personalización se realiza durante la fabricación del integrado especificando las interconexiones de metal. Esto significa que para que un usuario emplee un MPGA existe un costo industrial grande y el tiempo de manufacturación es generalmente de 12 semanas.

2.3 ASIC Semi - Custom

Los *ASICs Semi – Custom*, también conocidos como *ASICs Basados en Celdas Estándar (Standard – Cell – Based ASICs)*, utilizan *módulos prediseñados*. Estos módulos se denominan *Celdas Lógicas* debido a la particularidad de su función. El diseñador se limita a utilizar el recurso de las celdas preconstruidas colocándolas en el lugar que sea necesario dentro del dispositivo de acuerdo a las especificaciones de su diseño. En este tipo de ASICs, los fabricantes proporcionan un software que contiene una serie de bibliotecas llamadas *Bibliotecas de Celdas Estándar (Standard – Cell Libraries*) que contienen los módulos prediseñados [19].

La acción de utilizar Celdas Estándar prediseñadas y plenamente probadas, produce un ahorro notorio de tiempo, dinero y sobre todo se evitan riesgos posteriores a la realización electrónica. Las desventajas principales radican en el tiempo de manufactura que tiene que ser realizada en fábrica (por lo general toma de 6 a 8 semanas) y en que en la mayoría de las ocasiones, los fabricantes venden por separado las celdas[2] que realizan funciones lógicas más complejas.

[2] *La mayoría de los fabricantes venden por separado módulos a los que han denominado CORES y que son más aplicados en el diseño con ASICs Programables. Un CORE es un módulo prediseñado que implementa una función específica que va desde un simple multiplicador hasta un microprocesador completo, pasando por filtros e interfaces con bus PCI, entre otros.*

Las interconexiones se realizan de manera personalizada, mediante líneas de unión construidas específicamente para conectar las celdas colocadas por el usuario en un diseño particular, por lo que sus arquitecturas internas son parcialmente a la medida: una parte está definida por el fabricante (las celdas estándar y la manufacturación) y la otra está definida por el diseñador (selección de las celdas, colocación de las mismas en el dispositivo y las Interconexiones entre ellas).

2.4 ASIC Programable

Los *ASICs Programables (Programmables ASICs)* conocidos también como *ASICs Basados en Arreglos de Compuertas (Gate – Array – Based ASICs*, son las arquitecturas más representativas entre otras tantas existentes en el mercado actual y que conservan la característica básica de *programabilidad* [18, 20]. Incluyen a los *PLDs (Programmable Logic Devices – Dispositivos Lógicos Programables)*, *CPLDs (Complex PLDs – PLDs Complejos)* y *FPGAs (Field Programmable Gate Array – Arreglo de Compuertas Programable en Campo).*

Estos ASICs contienen módulos prediseñados por el fabricante (similares a las celdas estándar de los Semi – Custom) que pueden ser enrutados por usuario. A la vez permiten crear nuevos módulos que pueden ser reutilizados. La ventaja inmediata está en que los diseños pueden ser implementados en su totalidad, incluyendo la programación, directamente en un mismo ambiente de desarrollo por el usuario final, reduciendo el costo y el tiempo de diseño, no necesitando de una manufactura terminal en fábrica. Debido a esta versatilidad en el diseño y a su aplicación inmediata, los ASICs Programables también son conocidos en la práctica como *FPDs (Field Programmables Devices – Dispositivos Programables en Campo)* con una serie de denominaciones comunes: *FPLA (Field Programmable Logic Array – Arreglo Lógico Programable en Campo*), *FPA (Field Programmable Array – Arreglo Programable en Campo)*, etcétera.

Para asumir una mejor comprensión de los dispositivos, así como para cubrir el objetivo de esta revisión, nos centraremos en el estudio de un ASIC Programable, tratando con tópicos de importancia general que conforman el contexto del diseño con tecnologías programables: *Tipos de Arquitecturas, Disposición de Programación, Lógica Programable,* y *Tecnologías de Programación*.

2.4.1 Arquitecturas de los ASICs Programables: Simple y Avanzada

Se dice que un dispositivo es *Programable en Campo* cuando puede ser programado directamente por un usuario final. De esta manera, apegándonos a la definición, cualquier ASIC programable tiene una arquitectura programable en campo; sin embargo, la mayoría de los usuarios de estas tecnologías coinciden en distinguir un *Arreglo Programable en Campo* (por ejemplo, un FPGA) de un *Dispositivo Lógico Programable* (por ejemplo, un GAL), siendo que ambos conservan la misma naturaleza de programabilidad.

La apreciación estriba en las diferencias existentes entre las estrategias de programación, que a su vez están determinadas por la complejidad en la arquitectura de cada dispositivo. Así, se dice que una *arquitectura avanzada* como la de un CPLD o un FPGA, es una arquitectura programable en campo, término utilizado para hacer referencia a este tipo de circuitos integrados y a sus aplicaciones más elaboradas en comparación a las concernientes a un PLD [24, 26].

Un PLD tiene recursos comprometidos, es decir, no requiere de un proceso de interconexión entre sus elementos internos, ya que sus arreglos AND – OR, son fijos. Debido a esta característica, sus estructuras internas son *simples* y no pueden ser muy grandes, limitándose a una escala de integración *SSI*. Los dispositivos más avanzados, como los CPLDs y los FPGAs[3], están conformados por recursos no comprometidos, que pueden seleccionarse, configurarse e interconectarse, en función a los requerimientos del diseñador. Este tipo de arquitecturas más versátiles y complejas desembocan en escalas *LSI* y *VLSI*. La **figura 2.2**, esquematiza dos arreglos que denotan la diferencia arquitectural entre dispositivos programables [25, 27, 28].

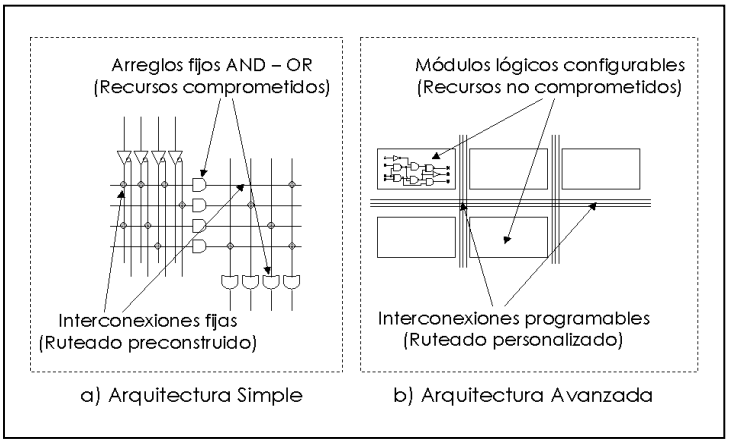


Figura 2.2: **Arquitectura simple y avanzada, de los ASICs Programables.**

[3] *Aún entre ellos, existe una gran diferencia en cuanto a desempeño y capacidad (densidad de compuertas lógicas), tal y como se comentará posteriormente.*

2.4.2 Disposición de Programabilidad

Existen dos derivaciones de acuerdo a la forma en que se descarga el archivo de programación que configurará el dispositivo: *ASICs Programables Sobre el Sistema (IS – In System)* y *ASICs Programables Fuera del Sistema (OS - Out of System)*[4]. La diferencia entre ambos, es que los *IS* permiten su programación dentro del mismo sistema del cual forman parte configurando los pines apropiados para dicha tarea. La programación se logra a través de un cable que sirve de interfaz entre la computadora que genera el archivo de programación y el dispositivo. Los *OS*, se programan únicamente por medio de un *Programador Universal* común. Es necesario colocar el dispositivo en el socket del *Programador*, por lo que se desprende del sistema del cual forma parte. La **figura 2.3**, exhibe las diferentes tendencias en la programabilidad [16].

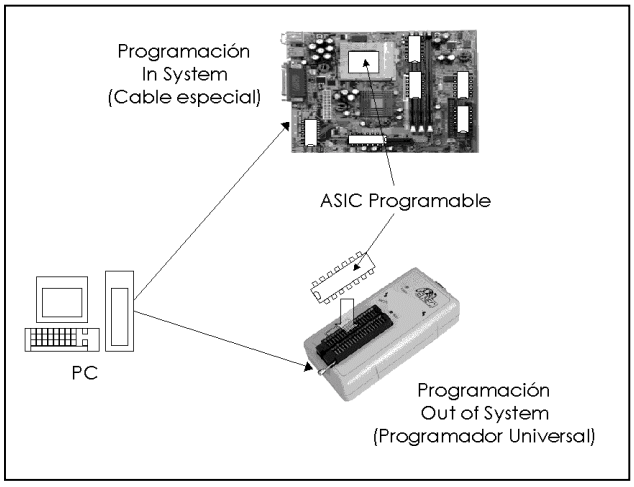


Figura 2.3: **Clasificación de los ASICs Programables de acuerdo a su disposición de programación.**

2.4.3 Lógica Programable

Los ASICs Programables, obtienen dos derivaciones lógicas en su programación: los llamados de *Lógica Reconfigurable* y los de *Lógica No Reconfigurable*. Los primeros se refieren a la particularidad de que el dispositivo pueda ser reprogramado, por lo que también se conocen como dispositivos *MTP (Many Times Programmables – Programables Varias Veces)*. Los de lógica no reconfigurable se conocen como dispositivos *OTP (One Time Programmables – Programables una Sola Vez)*. Ambas derivaciones se muestran en la **figura 2.4**.

[4] *Estas disposiciones de programabilidad, también son conocidas por algunos autores como In Circuit – En Circuito, y Out of Circuit – Fuera del Circuito.*

Un dispositivo MTP es idóneo para realizar prototipos, ya que puede ser reutilizado sin ejercer una inversión mayor. Su principal desventaja radica en que debido a su naturaleza reprogramable, sus líneas de interconexión presentan un aumento en el tiempo de propagación de la señal, por lo que no son dispositivos muy veloces. Los OTP son más útiles en su disposición como prototipos terminales, ya que es posible emigrar una configuración probada en un MTP a un OTP y así mejorar las condiciones de velocidad. Resulta admisible que un OTP no es un buen dispositivo para crear prototipos primarios, debido a que solamente se programan una sola ocasión, lo que implica una alta inversión en caso de que las pruebas iniciales no resulten satisfactorias [30].

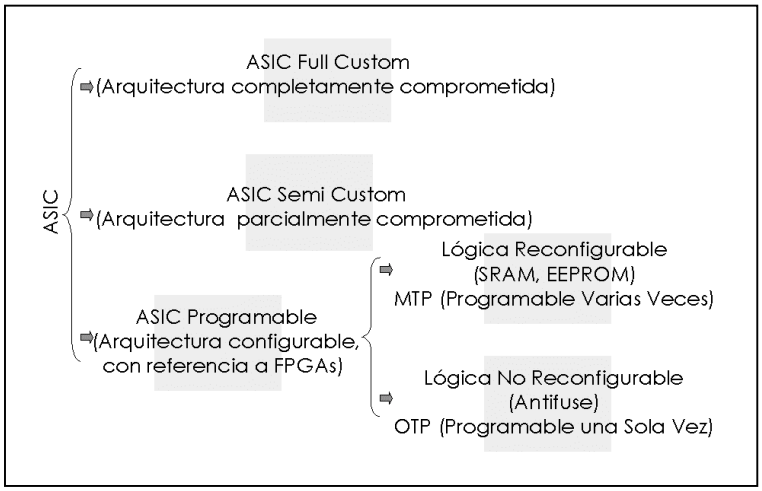


Figura 2.4: **Derivaciones MTP y OTP, de los ASIC Programables.**

2.4.4 Tecnologías Programables

Es evidente que al hablar de *tecnologías,* tácitamente se generalizan todos los aspectos que incluye un ASIC Programable; por ejemplo, cuando se mencionan las tecnologías programables se relaciona el término con un PLD, un CPLD, un FPGA, y demás dispositivos símiles, aspecto que resulta válido en la práctica. Cada tipo de ASIC programable se diferencia inmediatamente en dos aspectos: su *Arquitectura* (simple o compleja) y su *Tecnología de Programación*.

La arquitectura está representada por la estructura interna del dispositivo; es decir, la disposición física de cada elemento que lo conforman, así como la forma en que interactúan entre ellos. La tecnología de programación está representada por las características de fabricación a nivel sustrato con las que los componentes del dispositivo son construidos, involucrando los diferentes mecanismos de la física

electrónica que permiten que sean programados: determinan la forma[5] en las que se *establecen y rompen las interconexiones* para generar el ruteo, se construyen *tablas de búsqueda (LookUp Tables)* o se controlan las *conmutaciones entre transistores*.

La **tabla 2.1** lista de forma ordenada las diferentes tecnologías de programación, con sus respectivas lógicas (*Reconfigurable (MTP) y No Reconfigurable (OTP)*), así como sus *Usos*. Inmediatamente después se proporciona una breve descripción de la teoría básica de cada una de las tecnologías.

Tabla 2.1: **Tecnologías programables, lógicas y usos.**

Tecnología	Lógica Programable	Usos
Fusibles (Fuses)	OTP	Control de interconexiones
Programable por Máscara (Mask Programmable)	OTP	Control de interconexiones
Antifusibles (Antifuses)	OTP	Control de interconexiones
RAM Estática (SRAM - Static RAM)	MTP	Control de interconexiones Tablas de Búsqueda (LUT – Look Up tables)
PROM Borrable (EPROM - Erasable PROM)	MTP	Conmutación de transistores
PROM Eléctricamente Borrable (EEPROM - Electrically Erasable PROM)	MTP	Conmutación de transistores

2.4.4.1 Fusibles

La más antigua de las tecnologías de programación es la de los *Fusibles (Fuses)*. Cada uno de los puntos programables del ASIC consta de una conexión formada por un fusible. Cuando se aplica un voltaje considerablemente mayor que el de suministro normal (alimentación de energía) a través del fusible, la alta corriente rompe la conexión y lo quema. Así, un fusible intacto se encuentra en estado de *CLOSED* (*CERRADO* - Interconexión cerrada o unida), y cuando se quema se dice que está en un estado de *OPEN* (*ABIERTO* – Interconexión abierta o separada).

[5] *Algunos autores llaman usos o aplicaciones, a las tareas realizadas por las tecnologías programables, con la finalidad de llevar a cabo la interconexión entre componentes.*

2.4.4.2 Programable por Máscara

La tecnología *Mask Programming* (*Programable por Máscara*), también controla interconexiones. Este tipo de tecnología es aplicado por el fabricante durante los últimos pasos del proceso de elaboración del circuito integrado. Las conexiones se hacen en las capas metálicas que sirven como conductores dentro del dispositivo. Dependiendo de la función lógica deseada, la estructura de las capas la determina el proceso de fabricación. El procedimiento es costoso debido a que el fabricante hace al usuario un cargo económico por la fabricación a la medida (ASIC Full Custom) del diseño, que contiene las especificaciones requeridas. Este tipo de programación solamente es viable si se ordena una gran cantidad de ASICs con la misma configuración.

2.4.4.3 Antifusibles

La tecnología *Antifusibles (Antifuses)*, es lo apuesto a la *Fuses*. Cada antifusible consiste en un área pequeña en la que dos conductores están separados por un material de alta resistencia, por lo tanto, actúa como una ruta de *OPEN* antes de programarse. Aplicando a través de los dos conductores un voltaje mayor que el de suministro normal, el material que los separa se funde o cambia a una resistencia baja, haciendo las veces de una unión. El material de baja resistencia, se convierte en un conductor, por lo que ahora la conexión será una ruta de *CLOSED* que permitirá el paso del voltaje.

2.4.4.4 RAM Estática

La tecnología *One Bit of Static RAM (SRAM – Un Bit de RAM Estática)*, más conocida simplemente como *SRAM*, al igual que las anteriores, controla conexiones manejando la compuerta de un *transistor MOS* de canal *n* como punto de programación. Si el bit de SRAM almacena un *1*, entonces el transistor se enciende (*on*) y forma una ruta de conexión en CLOSED, uniendo las terminales. Con el bit de SRAM igual a *0*, el transistor está en apagado (*off*), por lo que se encuentra en un estado de OPEN, y no existe una conexión. Puesto que el bit de SRAM puede variar electrónicamente, es fácil reprogramar el dispositivo.

Debido a la necesidad de almacenar el bit de SRAM para establecer las interconexiones, no podemos apagar el suministro de la energía. Por tanto, encontramos que la tecnología basada en SRAM es *volátil*; esto es, la lógica programada (*configuración*) se pierde sin el voltaje de suministro.

Además de controlar conexiones, la tecnología SRAM sirve para realizar funciones lógicas mediante tablas de búsqueda (*LookUp Tables*). En este caso, las entradas lógicas son entradas de dirección para leer la SRAM y las salidas lógicas son los valores almacenados en la palabra direccionada que aparecen en las salidas de datos de la SRAM. Así la lógica se implementa con sólo almacenar la tabla de verdad en la SRAM – de ahí el término *Tabla de Búsqueda* (en inglés, *LUT - LookUp Table*).

2.4.4.5 EPROM Y EEPROM

Mediante el control de conmutación de transistores, también es posible programar los dispositivos. Básicamente se conocen dos tecnologías que usan el principio de la conmutación: la *EPROM (Erasable PROM – PROM Borrable)* y la *EEPROM (Electrically Erasable PROM – PROM Eléctricamente Borrable*). Alternando la carga almacenada en una compuerta flotante de un transistor MOS, es posible borrar y reprogramar un dispositivo construido bajo esta tecnología.

2.5 Arquitecturas Avanzadas de los ASICs Programables

Dependiendo de las características del diseño, es permisible utilizar un *CPLD* o un *FPGA*. Para elegir entre una de las dos opciones, suponiendo que se tuviera la necesidad de hacerlo, es importante familiarizarse con la arquitectura[6] de cada uno de ellos, para seleccionar la que se acople más a los requerimientos propuestos, observando el desempeño y el costo.

Un CPLD consta de una serie de módulos lógicos que se interconectan a través de una disposición arquitectural tipo *Crossbar (Matriz Interruptora Central o Barra de Cruce*), tal y como se aprecia en la **figura 2.5**. La lógica que implementa cada módulo

[6] *El análisis se centra en la arquitectura, ya que hablar de las tecnologías de programación no sería un distintivo general entre ambos. Las diferencias entre tecnologías son más notorias comparando entre diferentes fabricantes de un mismo dispositivo.*

lógico puede ser tan simple o compleja como la de un FPGA; entonces, la diferencia radica en el tipo de ruteo entre las interconexiones. Una crossbar entre módulos del CPLD tiene varias ventajas, aunque la más determinante de éstas es que se conoce de antemano el retardo de propagación entre un módulo lógico y la crossbar, debido a que las líneas de ruteo conservan la misma distancia entre sí, aunque por otra parte implica mayor consumo de potencia por parte del dispositivo. En un FPGA, los módulos lógicos se interconectan libremente a través de líneas programables (*Interconexión tipo Incrementativa*), pudiéndose conectar un bloque de extrema izquierda en la parte de arriba con uno de extrema derecha en la parte de abajo, lo que propicia que existan algunos problemas de retardo, causados por líneas más largas que otras, por la misma razón un CPLD es más rápido en cuestiones de desempeño en comparación a un FPGA.

La conexión crossbar de los CPLDs presenta el inconveniente de que es necesario un protocolo que maneje tantas interconexiones como líneas provenientes de módulos lógicos lleguen a la barra, complicando el manejo de información a medida que se aumenta la cantidad de módulos, razón por la cual, se sostiene una velocidad aceptable en contraparte a una cantidad limitada de módulos disponibles. Un FPGA es ajeno a este tipo de problemas, ya que su arquitectura está totalmente independizada, permitiendo que existan grandes cantidades de módulos lógicos[7] dentro de un solo dispositivo.

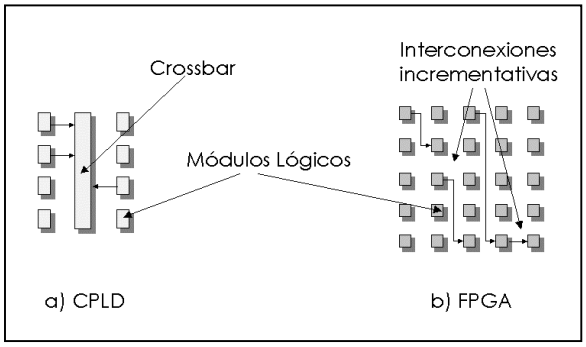


Figura 2.5: **Arquitecturas no detalladas de un CPLD y de un FPGA.**

Por lo expuesto con anterioridad, se observa que un CPLD es un dispositivo económicamente más barato que un FPGA, debido a su densidad de componentes (aunque claro, comparados con un PLD son mucho más grandes); sin embargo, son

[7] *En las hojas de especificaciones de los fabricantes, la característica de capacidad o densidad de componentes internos de un CPLD o de un FPGA, está dada en función del número de compuertas lógicas o en otros casos por la cantidad de módulos lógicos que pueden llegar a construir la función de una simple compuerta o de algún otro circuito más complejo.*

más rápidos y para prototipos de baja densidad son muy útiles. La **tabla 2.2** nos muestra los resultados de las comparaciones[8].

Tabla 2.2: **Comparación entre dispositivos programables de arquitectura avanzada.**

Dispositivo	Densidad	Desempeño en términos de velocidad	Tipo de interconexión	Costo	Aplicaciones Típicas
CPLD	Entre baja y media. De 500 hasta 50,000 compuertas lógicas.	Arriba de los 450 MHz, actualmente.	Crossbar	Bajo	Interfaces, Controladores, Contadores, Memorias Registros de corrimiento, etc.
FPGA	Entre media y alta. De 1,000 hasta 4,000,000 de compuertas lógicas.	Depende del diseño. Arriba de los 300 MHz, actualmente.	Incrementativa	Alto	Microprocesadores, Microcontroladores, Interfaz con Bus PCI, ALUs dedicadas, etc.

3.5.1 Arquitectura FPGA

Un FPGA se compone de elementos con recursos no comprometidos que pueden ser seleccionados, configurados e interconectados por usuario. Se ha mencionado que en un PLD las interconexiones entre los elementos ya están prefijadas, solamente es posible habilitar o deshabilitar la interconexión; en el caso de un FPGA no hay nada interconectado.

Estos dispositivos se componen de cierto número de *Módulos Lógicos*, que determinan la capacidad del dispositivo. Los módulos son independientes entre sí y pueden interconectarse para formar un modulo más complejo. Dependiendo del fabricante, estos módulos pueden ser *Bloques Configurables*, como en los FPGAs de *Xilinx* y de *Altera*, o bien, *Elementos de Función Fija* formados por arreglos de compuertas, como en el caso de los dispositivos de *Actel*.

Los módulos del FPGA, se interconectan por medio de *Canales Configurables* como se aprecia en la **figura 2.6**. Al proceso de interconexión, se le conoce como *enrutamiento* y consiste en determinar la mejor estrategia de interconectar los módulos, utilizando para ello alguna herramienta de *diseño electrónico*.

8 *Datos vigentes hasta el 2do. Semestre del año 2000. Muy posiblemente hayan variado a la fecha, principalmente con respecto a la densidad y al desempeño de los dispositivos.*

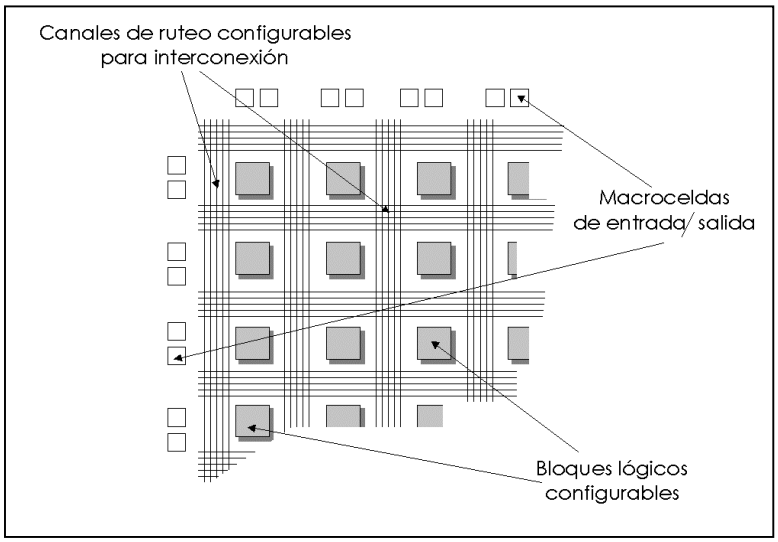


Figura 2.6. - **Arquitectura demostrativa, no detallada, de un FPGA de Xilinx**.

Por la capacidad de un FPGA (el más sencillo contiene 1,200, hasta los más grandes que contienen 4,000,000[9] de compuertas lógicas), se dice que son dispositivos para diseños *LSI* y *VLSI*. La gran ventaja de utilizar estos dispositivos radica en que todo el desarrollo se lleva a cabo en un solo ambiente de trabajo. El diseñador propone la función lógica a realizar y a través de métodos de descripción define los parámetros de su problema. Esto se hace por medio de código programable, que puede ser un *Lenguaje de Descripción de Hardware*, o bien, un *diagrama esquemático* de conexiones.

De acuerdo a varias bibliografías consultadas, las principales ventajas de diseñar sobre FPGAs se listan a continuación:

a. *Minimización del número de componentes en un diseño*. Con esto se reducen los gastos de inventario, inspección y prueba, así como el número de fallas a nivel circuito impreso, propiciando un ahorro de espacio físico. Una medida de la versatilidad de un dispositivo programable se expresa mediante el número de dispositivos de función fija (circuitos Integrados de catálogo) que pueden remplazarse.

b. *Reducción en el tiempo de diseño*. Debido a su naturaleza programable, reducen el tiempo y los costos de desarrollo, no sólo de nuevos productos sino también de aquellos que requieren modificaciones (*reingeniería*). Si se está utilizando un dispositivo programable en el mismo circuito (referirse al tópico

[9] *Dato actualizado hasta el primer semestre del año 2000. El XCV3200 de la familia Virtex de Xilinx, es el FPGA con mayor capacidad en el mercado.*

2.4.2), los cambios en el diseño son realizados mediante una nueva configuración que se prueba de inmediato.

c. Uso de una gran variedad de herramientas de *Diseño Asistido por Computadora (CAD*), disponibles actualmente en el mercado. Estas herramientas promueven y facilitan el diseño sobre este tipo de dispositivos. Así mismo, no se requiere de grandes recursos de cómputo

2.5.1.1 Granularidad de los FPGAs

La complejidad de los elementos contenidos dentro de cada módulo lógico, es factor determinante para medir el desempeño de un FPGA. Independientemente del fabricante, los módulos lógicos realizan las operaciones básicas que en conjunto representan la función que operará el FPGA. La arquitectura de estos dispositivos tiene dos derivaciones estructurales de acuerdo al tipo de módulos lógicos que la conforman: *Granularidad Gruesa* y *Granularidad Fina*.

1. Los módulos lógicos en una arquitectura de *Granularidad Gruesa (Coarse Grained – CG)*, son módulos grandes generalmente consistentes de una o más *Tablas de Búsqueda* (*LUT - LookUp Table*) y dos o más, flip – flops. Si analizamos la configuración física de cada módulo lógico o g*rano*, podemos advertir que particularmente cada uno de ellos, puede ejecutar una función simple o una función compleja, que adicionada a otra función ejecutada por un módulo diferente conforman un sistema más complejo.

La tabla de búsqueda actúa como una memoria donde se encuentra almacenada la tabla de verdad que representa la función lógica del circuito; así en una *LUT* es posible construir cualquier función deseada. Por lo anterior se dice que un módulo que contiene elementos como éstos, es un *Grano Grueso,* propiciando que desde un nivel muy básico se tengan módulos complejos.

Los FPGAs CG, son dispositivos con una gran densidad de compuertas, ya que la acción de utilizar LUTs deja entrever que se pueden realizar diseños más elaborados. Los FPGAs con tecnología SRAM, como los de *Xilinx* o los de *Altera,* tienen arquitecturas de esta índole, y son programables en sistema.

2. Por otra parte, una arquitectura de *Granularidad Fina (Fine Grained – FG)*, está estructurada por una gran cantidad de módulos lógicos pequeños que realizan funciones relativamente simples. Cada grano en este tipo de topología está

compuesto de un circuito de dos entradas que realiza una función lógica determinada, o en algunos otros casos por un multiplexor 4 a 1. Adicionalmente contienen un único flip – flop.

La arquitectura FG, como la de los FPGAs de *Actel*, está relacionada con la tecnología de programación antifusibles por lo que son del tipo OTP y programables fuera del sistema. La simpleza de la constitución de cada módulo, permite implementaciones a nivel más detallado y sobre todo más veloces.

A causa de que para llegar a construir una función compleja se requiere el uso de varios módulos, se trata de dispositivos de alta densidad de módulos (comparándolos con los anteriores); pero cada módulo tiene un número mínimo de compuertas lógicas a diferencia de los FPGAs CG, que pueden tener menor número de módulos pero cada módulo tiene un número grande de compuertas lógicas.

2.5.1.2 Algunas Familias de FPGAs, Utilizadas en el Mercado Actual

Los diseñadores coinciden en afirmar que desde el punto de vista usuario, existen tres fabricantes mayoritarios en la distribución de FPGAs y software de soporte: *Xilinx, Altera* y *Actel*. En el mercado es posible encontrar otros tantos con producciones menores que figuran también como FPGAs útiles: *Lucent, Texas Instruments, Lattice, QuickLogic, Cypress, Atmel*, etc. En este apartado nos enfocaremos a los tres principales mencionados de inicio, dando una breve introducción a las familias lógicas y sus características principales.

1. *FPGAs de Xilinx* [46]. Considerado como uno de los fabricantes más fuertes a nivel mundial, los FPGAs de Xilinx (también fabrica PLDs y CPLDs) están basados en la tecnología SRAM y son dispositivos *MTP*, *programables en sistema (IS*).

   Sus principales familias de FPGAs son: *XC3000, XC4000, XC Virtex*, *y XC Spartan*. La estructura de estos dispositivos está compuesta por módulos lógicos denominados *CLBs (Configurable Logic Blocks – Bloques Lógicos Configurables*), basados en tablas de búsqueda. Cada CLB contiene circuitos que les permiten realizar operaciones aritméticas eficientes. También los usuarios pueden configurar las tablas de búsqueda como celdas *read/write (lectura/escritura)* de RAM. A la vez, a partir de la serie XC4000, se incluye un generador interno de señal de reloj con 5 diferentes frecuencias.

Además de los CLBs, los FPGA de este fabricante incluyen otros bloques complejos que configuran la entrada de los pines físicos que conectan el interior del dispositivo con el exterior, a estos bloques se les conoce como *IOBs (Input/Output Blocks – Bloques de Entrada/Salida)*. Cada IOB contiene una lógica compleja que permite que un pin físico pueda actuar como entrada, salida o un tercer estado.

La **figura 2.7** muestra a detalle, la arquitectura del XC4003E de Xilinx. Nótese la complejidad de un CLB, así como la disposición de los IOBs alrededor del dispositivo, sirviendo como interfaces entre el FPGA y el mundo exterior.

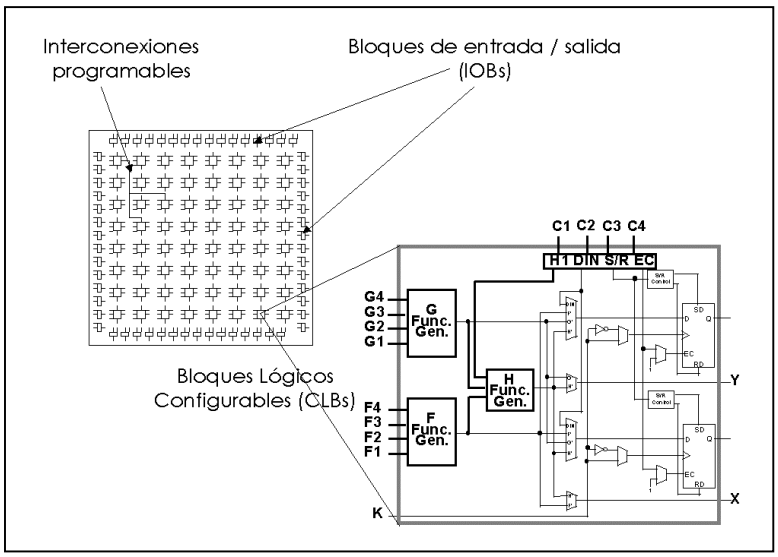


Figura 2.7: **Arquitectura de un FPGA XC4003E de Xilinx.**

La serie *XC Virtex*, o tan solo *Virtex*, es la más nueva de todas las pertenecientes a Xilinx. Los dispositivos de esta familia son los más rápidos (velocidades de trabajo de hasta 250 MHz), densos en compuertas y con menor consumo de potencia, pero por lo mismo son los más costosos. El FPGA *XCV3200E* de la familia Virtex (en específico, de la serie *Virtex E*) es el dispositivo más grande y poderoso de Xilinx, con cerca de 4,047,000 compuertas lógicas.

La serie *Spartan* surgió como una opción para sustituir diseños probados de menos de 15,000 compuertas por dispositivos de bajo costo y alto desempeño (además incluyen el soporte de los *CORES* prediseñados), pero sacrificando algunas características que manejan las series estándar de Xilinx.

Las series estándar *XC3000* y su sucesora, la *4000*, son series que muestran la mayor parte de las características funcionales de los FPGAs de Xilinx, pero con la gran desventaja que son dispositivos de baja densidad de compuertas. Sin embargo, el uso de la serie XC4000 (en especial el XC40003E) es muy socorrido para el diseño e implementación de prototipos de bajo impacto[10]. Para este trabajo de tesis se utilizó un dispositivo XC4010XL.

La **tabla 2.3**, ilustra una breve comparación entre familias de Xilinx, tomando como parámetros de comparación a la densidad de compuertas lógicas y al número de pines configurables como entradas y/o salidas.

Tabla 2.3: **Comparación simple entre las diferentes Familias de FPGAs de Xilinx.**

Familia	Pines de E/S	Número de Compuertas Lógicas
Viirtex E	176 a 804	470,000 a 4,047,000
Virtex	180 a 512	34,000 a 1,124,000
Spartan y Spartan II	77 a 260	2,000 a 150,000
XC 4000	56 a 448	10,000 a 180,000

2. *FPGAs de Altera* [44]. Altera ofrece dos familias de FPGAs con características diferentes, pero conservando algunas básicas que representan las ventajas originales de las primeras familias estándar: *FLEX 6000, 8000*, y *10K*; así como la más novedosa*, APEX 20K*. Las primeras familias estándar, la FLEX 6000 y la 8000, aún se utilizan ampliamente. La serie FLEX (*Flexible Logic Element Matrix – Matriz Flexible de Elementos Lógicos*) estándar, contiene un número considerado de compuertas en tecnología SRAM con tablas de búsqueda.

En la **tabla 2.4**, se listan las diferencias en densidad de compuertas y pines entre las diferentes familias. Nótese que la serie APEX 20K es la que contiene el mayor número de compuertas y es la que se usa para diseños más complejos y dedicados.

[10] *Un diseño de bajo impacto, es un circuito digital que no realiza funciones demasiado complejas.*

Tabla 2.4: **Comparación simple entre las diferentes Familias de FPGAs de Altera.**

Familia	Pines de E/S	Número de Compuertas Lógicas
APEX 20K	250 a 780	263,000 a 2,670,000
FLEX 10K	59 a 470	10,000 a 250,000
FLEX 8000	71 a 218	16,000 a 24,000
FLEX 6000	68 a 208	2,500 a 16,000

La serie estándar FLEX combina la arquitectura de los CPLDs con los FPGAs. El dispositivo consiste de una arquitectura muy parecida a la de un CPLD, en la que el nivel más bajo de la jerarquía es un conjunto de tablas de búsqueda, por lo mismo se considera como una arquitectura FPGA.

Específicamente, un FPGA FLEX 8000 está basado en tecnología de programación SRAM, teniendo una tabla de búsqueda de 4 entradas en su módulo lógico más básico. La **figura 2.8**, ilustra la arquitectura general del FLEX 8000. El módulo lógico básico, nombrado por Altera como *Elemento Lógico (Logic Element*), contiene la LUT de 4 entradas, un flip – flop y un elemento de acarreo (*carry*) de propósito especial para circuitos aritméticos (similar al XC4000 de Xilinx). El Elemento Lógico también incluye circuitos en cascada que permiten una implementación eficiente de funciones AND amplias.

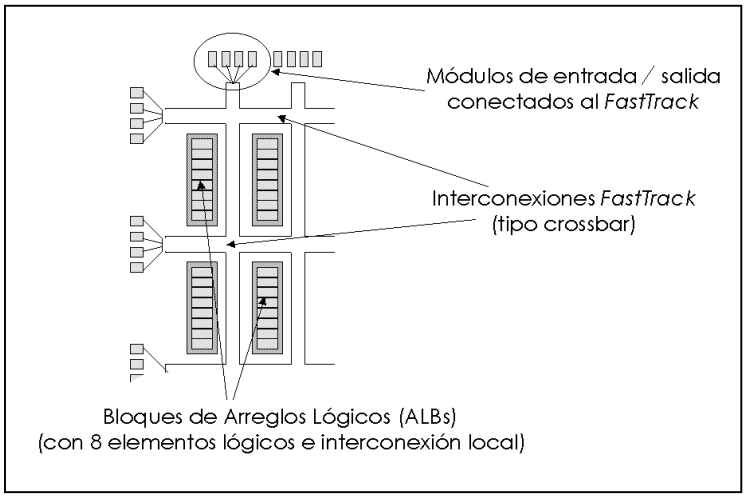


Figura 2.8: **Arquitectura de un FPGA FLEX 8000 de Altera.**

Esta arquitectura agrupa elementos lógicos en grupos de 8, y los llama *Bloques de Arreglos Lógicos (Arrays Logic Blocks – ALBs*). Cada ALB, contiene una interconexión local que le permite conectarse con otro ALB; a la vez, la misma

interconexión sirve para conectarse a la interconexión global de la crossbar[11] (matriz de interconexiones), nombrada por Altera como *FastTrack* (ver **figura 2.8**). Así, las interconexiones se hacen al estilo de los CPLDs, pero la configuración de los ALBs utiliza tecnología SRAM propia de los FPGAs reconfigurable. Este dispositivo de Altera es capaz de trabajar a 200 MHz.

3. *FPGAs de Actel* [45]. Este fabricante ofrece una serie de familias no reconfigurables (OTPs) que resultan ampliamente utilizadas después de haber probado satisfactoriamente un diseño (*emigrar* a otro FPGA). Las principales son: La serie estándar *ACT*, y las más nuevas por orden cronológico de aparición, *sX, sX – A*, *mX* y la más reciente, *eX*. Todas las anteriores son programables fuera del sistema (*OS*). También ofrece una familia reconfigurable a la que llama *Pro ASIC* (es de alta densidad de componentes, y Actel no la considera parte de los FPGAs), basada en una tecnología *Flash EEPROM* programable en sistema (*IS*).

Los FPGAs de Actel, emplean como módulo o elemento básico una estructura tipo *Arreglo Fijo de Compuertas (FA – Fixed Array)*. La **figura 2.9**, muestra un dispositivo *ACT 3*, donde podemos apreciar como la lógica del arreglo está dispuesta en renglones de módulos lógicos interconectables, rodeados hacia fuera por *Módulos de E/S (Input/Output Modules)*. La estructura de interconexiones consiste en pistas o líneas fijas de interconexión horizontales y verticales con los segmentos de alambrado. Existen muchas pistas en cada canal entre los renglones de la lógica. Las pistas verticales son menos y pasan sobre los canales horizontales y los módulos lógicos.

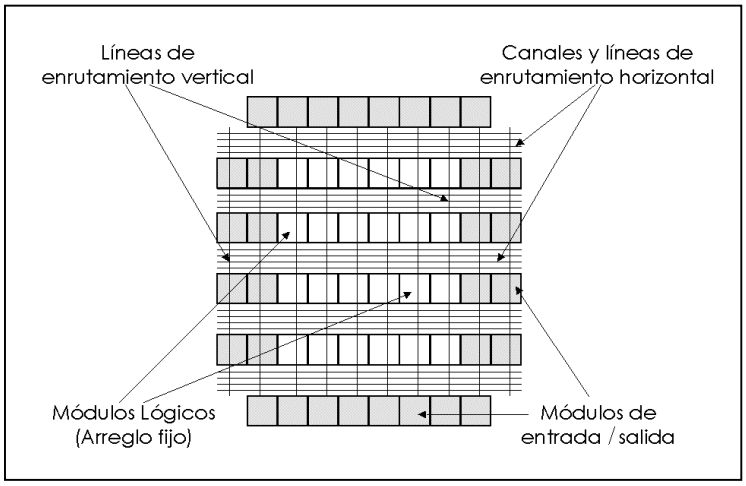


Figura 2.9: **Arquitectura de un FPGA ACT 3 de Actel.**

[11] *La crossbar de un FPGA de Altera está conformada por varias otras, por lo que resulta muy avanzada, incluyendo mayores ventajas que la de un CPLD típico.*

El FPGA de Actel utiliza tecnología *antifusible* que proporciona una programación permanente y no volátil. El dispositivo tiene muchos antifusibles para conectar las entradas y salidas de los módulos de lógica y E/S a los segmentos de alambrado de los canales, además también interconectan los segmentos de alambrado entre las pistas para ofrecer conexiones de diferentes longitudes.

Dentro de las series antifusibles, la familia *eX* es la tiene el mejor desempeño, con las características más novedosas, pero en contraparte es la de menor densidad de compuertas. La serie *mX* es la que entrega el menor retardo de propagación entre todas las familias y fabricantes de FPGAs, combinando la tecnología antifusibles con las interconexiones de alta rapidez para alcanzar velocidades de trabajo de hasta 450 MHz.

La **tabla 2.5** ilustra la comparación entre familias de Actel, en función al número de compuertas.

Tabla 2.5: **Comparación simple entre las diferentes Familias de FPGAs de Actel.**

Familia	Pines de E/S	Número de Compuertas Lógicas
eX	84 a 132	3,000 a 12,000
SX y sX – A	130 a 360	12,000 a 108,000
MX	57 a 202	3,000 a 54,000
Pro ASIC	210 A 446	98,000 a 473,000
ACT 1, 2, y 3	56 a 244	2,000 a 25,000

2.6 Metodología de Diseño VLSI con Herramientas CAD

El diseño sobre dispositivos programables, particularmente digitales[12], tiene sus inicios desde hace más de dos décadas. El conjunto *EDA (Electronic Design Automation – Diseño Electrónico Automatizado*) son todas las herramientas, tanto hardware como software, que se utilizan para el diseño de sistemas electrónicos. Dentro de EDA, las herramientas *VLSI CAD* juegan un importante papel en el diseño de hardware a través

[12] *Actualmente es posible diseñar sobre dispositivos digitales (FPGAs, CPLDs) o sobre analógicos (FPAAs - Field Programmable Analog Array – Arreglo Analógico Programable en Campo); de cualquier forma, el flujo de diseño es muy similar respetando algunas restricciones en cuanto a la aplicación y topología del dispositivo.*

de software. En virtud del inminente incremento en la complejidad de los circuitos con arquitectura avanzada, se hace indispensable un sofisticado aporte por parte de las herramientas CAD para automatizar el proceso de desarrollo, repercutiendo en una *disminución en el tiempo de diseño, aumentando la calidad del producto* y *reduciendo los costos de producción*.

En la actualidad, existen variados ambientes de desarrollo para FPGAs y CPLDs, algunos con herramientas de software completas para aceptar la *captura, simulación, síntesis y configuración física* del dispositivo; a la vez que otros son más modestos en sus capacidades adaptándose a las herramientas de propietarios mayores.

En la **tabla 2.6**, se puede apreciar algunos ejemplos de herramientas de desarrollo y sus respectivos fabricantes, así como algunas familias de FPGAs soportadas por cada software de diseño. Cabe mencionar que no todos son ambientes completos, pero se listan a manera de ejemplificar recursos aportados por propietario y las tecnologías específicas sobre las cuales aplican. Sobresalen los ambientes completos *Foundation Series, Actel DeskTOP* y *MAX+PLUS II*, que representan los más utilizados actualmente para el diseño y realización de aplicaciones con lógica configurable. Lo concerniente a los CPLDs no es materia de discusión en este trabajo, aunque no se desconoce la familiaridad con los FPGAs, por lo que resulta similar el tratamiento en el diseño.

Tabla 2.6: **Ambientes de desarrollo para FPGAs.**

Fabricante	Ambiente de Desarrollo	Algunas Familias Soportadas
Xilinx	Alliance Series Software Foundation Series Software	XC3000, XC4000, SPARTAN, VIRTEX
Actel	Actel Designer Series Development System	ACT 1, ACT2, ACT 3, Integrator 1200XL, 3200 DX
Altera	Max+Plus II	MAX 7000, MAX 9000, FLEX 8000, FLEX 10K
Atmel	Atmel Integrated Development System	AT6000
QuickLogic	The Quick Works	pASIC 1, pASIC 2, pASIC 3
Gatefield	ASICMaster	GF250F, GF260F
Lucent Technologies	ORCA Foundry Development System	ORCA OR2C/OR2T, ORCA OR2CA/OR2TA, ORCA OR3CA/OR3TA
Motorola	MPA Design System	MPA 1000

La **figura 2.10** esquematiza de manera general, la secuencia de fases necesarias para diseñar sobre FPGAs. Obsérvese la prioridad que precede cada uno de los pasos dentro del flujo de diseño VLSI con herramientas CAD, considerando que dependiendo

del fabricante, el nombre de la etapa puede variar manteniendo su operatividad básica de acuerdo al caso de estudio presentado.

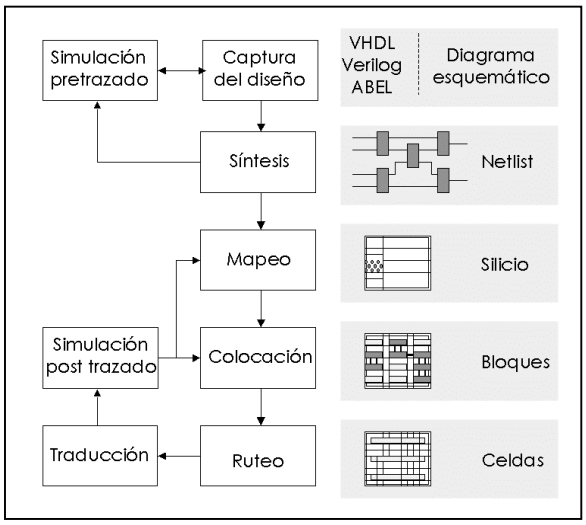


Figura 2.10: **Flujo de desarrollo para ASICs programables, utilizando herramientas VLSI CAD.**

1. *Captura del diseño (Design Entry).* En esta primer etapa se procede a ingresar el modelo de solución dentro del ambiente de desarrollo VLSI CAD seleccionado. La captura puede ser mediante uno (es válido el uso de varios) de los siguientes medios de captura que describen el funcionamiento del circuito: un *diagrama esquemático,* un *Lenguaje de Descripción de Hardware*, o mediante un *Netlist (Archivo de Conexiones)*. Entre los diferentes fabricantes se respetan algunos estándares de diseño con HDL y captura de esquemáticos[13], pero no así de configuración física de dispositivos, debido a que los ambientes y arquitecturas de propietario, son distintas entre sí.

En el caso de la captura de esquemático, el software de diseño cuenta con bibliotecas muy extensas de componentes de uso común, por lo que sólo basta con realizar interconexiones entre las *primitivas*[14] formando el sistema digital (referirse a la **figura 2.11**). El método clásico para la interconexión de los distintos símbolos de una hoja de diagrama son los *cables* o *wires*. Un cable tiene una correspondencia inmediata con el circuito real, se trata de una conexión física

[13] *Mediante un archivo EDIF (Electronic Design Interchange Format – Formato de Intercambio de Diseño Electrónico), es posible intercambiar información de esquemáticos en función a la definición de celdas básicas elementales para la representación de diagramas. Sin embargo, no es totalmente compatible si se utilizan celdas más elaboradas disponibles en las bibliotecas específicas del fabricante. EDIF es un estándar ANSI/EIA catalogado con el número 548 – 1988.*

[14] *Una primitiva es una celda lógica primordial, o bien un circuito muy simple formado con elementos muy básicos; por ejemplo, una compuerta lógica, un buffer, un latch, o un registro. Debido a su naturaleza, la primitiva es el elemento que se encuentra en el nivel más bajo de la jerarquía de diseño VLSI CAD.*

que une una terminal con otra, creando una correspondencia para la transmisión de la señal eléctrica.

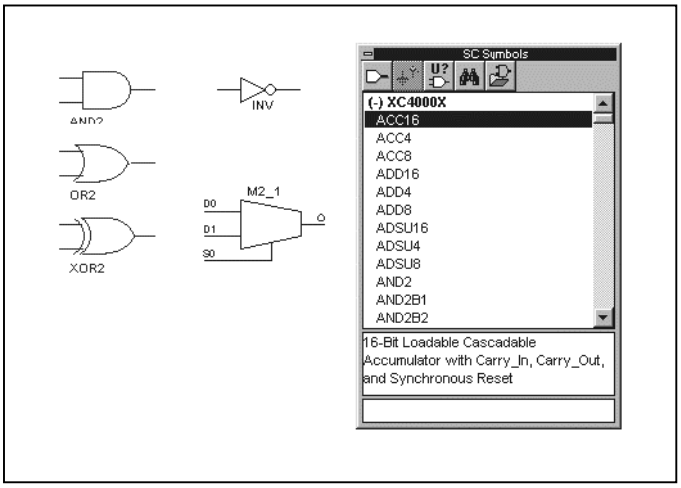


Figura 2.11: **Editor de diagramas esquemáticos.**

La conexión hacia el exterior del circuito integrado se conoce como *puerto o pin*, no obstante la mayoría de los editores hace la referencia conforme a la dirección de su señal: *IPAD (puerto de entrada*), *OPAD (Puerto de Salida*), agregando el pin bidireccional *IOPAD (Puerto de Entrada/Salida*).

Una captura por medio de un HDL también culmina con *símbolos* que se integran es un diagrama esquemático. Algunos ambientes, en especial los más sofisticados, permiten editar código introduciendo la descripción del funcionamiento del circuito por medio de: *Tablas de Estado, Diagramas de Estado* (con herramienta *gráfica* o código), y *Ecuaciones*.

Los editores que incorporan los diferentes ambientes VLSI CAD admiten código HDL y posteriormente crean una *Macro* reutilizable, que es un *módulo esquemático* (simbólico) que representa un componente cuya función es específica. La biblioteca de la macro se adiciona automáticamente a las bibliotecas del software para quedar disponible en el entorno de la captura esquemática, pudiéndose utilizar como elemento independiente (símbolo) o combinado con las primitivas originales.

2. *Síntesis Lógica (Logic Síntesis*). El software de desarrollo incluye la herramienta propia para realizar la síntesis lógica de un código en HDL o de un diagrama esquemático. A través de una compilación es posible producir un *netlist* a partir de cualquiera de los métodos de captura. Un netlist es un archivo que registra la

descripción de las *celdas lógicas* (primitivas) y sus conexiones específicas, que conforman un circuito. El formato de netlist más utilizado es *EDIF (Formato de Intercambio de Diseño Electrónico - Electronic Design Interchange Format)*.

3. *Simulación Pretrazado (Prelayout Simulation*). También conocida como *Preruteado;* hasta el segundo paso aún no se ha comenzado con el *diseño físico*, solamente se ha realizado el *diseño lógico* del circuito modelado. La mayoría de los ambientes de desarrollo incluyen dos tipos de simulación, una llamada *lógica* y la otra *física*. La diferencia es común: la primera es antes del ruteo o trazado (interconexión de líneas) y la otra es posterior. La simulación lógica solamente permite observar el comportamiento del diseño con las celdas lógicas más básicas del dispositivo seleccionado.

4. *Mapeo o Planeación de la Superficie (Floorplanning*). Una vez que se simuló el diseño, se procede a mapear el netlist sobre el FPGA seleccionado. La configuración de las celdas lógicas y sus respectivas conexiones, descritas por el netlist, se distribuyen sobre la superficie del circuito integrado a manera de identificar recursos. Los algoritmos involucrados en este proceso permiten minimizar espacio físico y retardos de señal, en una primera aproximación.

5. *Colocación (Placement*)[15]. Estratégicamente, el software decide la colocación de las primitivas sobre un bloque del dispositivo físico (módulo lógico). Los algoritmos inteligentes con los que trabaja el sistema de sintetización, deciden la mejor ubicación para cada celda lógica, considerando las *redundancias* en el diseño *(componentes o conexiones repetidas)* y una segunda aproximación para eliminar los retardos críticos.

6. *Trazado o Ruteo (Rutting)*. Realiza la conexión física entre los módulos lógicos y determina el tipo de interconexión proponiendo rutas cortas o largas, según sea el mejor caso.

7. *Traducción (translate)*. Es la extracción de características del circuito, determinando la *resistencia* y *capacitancia* eléctrica, entre las interconexiones para verificar un correcto ruteo de las líneas. El software de diseño se encarga automáticamente de generar el ruteo y la extracción de impedancias; sin embargo, algunas herramientas incluyen editores para realizar las conexiones de modo personalizado.

8. *Simulación Post trazado (Postlayout Simulation*). Una vez que se han colocado y ruteado las celdas primitivas y los módulos que las contienen, la configuración

[15] *Las tareas de Place y Rute, propias de los FPGAs, son conocidas como el Fitter (Ajuste) en el diseño sobre CPLDs.*

física que ha adquirido el dispositivo puede simularse. A este tipo de simulación se le conoce como *física*, debido a su proximidad con el comportamiento real que tendrá el diseño.

Es posible considerar los retardos de propagación que sufren las señales. El retardo generado dentro de un módulo lógico será igual a la suma de los retardos de los elementos contenidos por el mismo (flip flops, tablas de búsqueda, multiplexores, y demás). Para determinar la *ruta crítica* , el analizador del software suma los retardos de cada módulo lógico más los retardos de las interconexiones que participan en la ruta, desde un puerto de entrada hasta un puerto de salida. Las bibliotecas de cada dispositivo en particular, contienen información referente a los retardos de cada elemento dentro del módulo lógico, por lo que los resultados entregados por la simulación son certeros y confiables.

2.7 Panorama General de los Lenguajes de Descripción de Hardware

Un HDL, es un lenguaje usado para modelar la operación funcional de una pieza de hardware (circuitos electrónicos y/o sistemas completos), en una forma textual. Al igual que en los lenguajes de programación comunes, se observan ciertas diferencias entre HDLs que van desde la sintaxis de codificación hasta los métodos de simulación y síntesis, pasando por su capacidad de compatibilidad. Por lo mismo, cualquiera de ellos, implica un aprendizaje formal [33, 39].

Para algunos diseñadores, se conocen dos tipos generales de HDLs: los de *bajo modelado* y los de *alto modelado*. Los de bajo modelado son capaces de realizar descripciones simples y no permiten la jerarquización de módulos; los de alto modelado son aptos para diseños más complejos.

En el contexto tecnológico mundial, los HDLs más comunes e importantes para alto modelado son dos:*, Verilog (Lógica Verificable - Verify Logic)* y *VHDL* (*VHSIC Hardware Description Language,* donde el vocablo *VHSIC se refiere a Very High Speed Integrated Circuit – Circuito Integrado de Muy Alta Velocidad*). Ambos permiten diseñar de acuerdo a *ecuaciones, tablas de verdad* y *diagrama de estados*; su sintaxis es particular a cada uno con palabras reservadas para indicar al sintetizador el método

de diseño que se elija. Los métodos de diseño se cometan posteriormente en este mismo capítulo.

VHDL (estándares IEEE 1076 – 1987 y IEEE 1076 – 1993)[16] presenta cierta complejidad debido a la sintaxis propuesta, aunque en otro sentido, está diseñado para trabajar preferentemente en escala VLSI. Este lenguaje de descripción es ampliamente utilizado actualmente al igual que Verilog (estándar IEEE 1364 – 1995)[17] con la excepción de que éste es más reciente y presenta menor complejidad de sintaxis, debido a que sus raíces sintácticas se basan en *Lenguaje C* estándar.

Las características que comparten ambos lenguajes son:

a. El diseño puede descomponerse jerárquicamente, por lo que la modularización puede ser realizada desde un nivel muy bajo de abstracción.

b. Soportan cualquier nivel de modelado y de abstracción. Verilog, contiene todas las funciones cubiertas por el VHDL, e incluye la capacidad de poder diseñar a *Nivel de Descripción Estructural (SDL - Structural Description Level)* dirigiéndose a un nivel transistor, que es aún más bajo que el nivel de las primitivas.

c. Cada elemento de diseño tiene una interfaz bien definida (para conectarse a otros elementos) y una precisa especificación de comportamiento (funcionamiento del circuito) que permite certeras simulaciones.

d. La concurrencia, el retardo y el tiempo de sincronía pueden ser modelados. Ambos lenguajes permiten estructuras síncronas, así como asíncronas.

2.7.1 Elementos Básicos de VHDL y Verilog.

Partiendo de la abstracción de un diseño, la cual se representa por un solo bloque, es posible comparar los elementos básicos que componen cada uno de los lenguajes de descripción utilizados. La intención de esta breve introducción es facilitar la comprensión de los códigos generados por este trabajo de tesis, por lo que se omiten detalles que son plenamente necesarios si se tiene la inquietud de codificar en HDLs.

[16] *Desarrollado en 1984 por el Departamento de Defensa de los Estados Unidos de América con patrocinio del IEEE.*
[17] *Desarrollado originalmente por Gateway Design Autmation en 1984, fue un software de propietario y no poseía estándar IEEE. Posteriormente, la empresa Cadence adquirió los derechos modificando sus alcances para su estandarización en 1995.*

Para declarar un módulo, VHDL recurre a dos instancias: entity y architecture. Entity es la declaración de los pines de entrada y salida del módulo, así como de las señales internas que maneja el diseño; también identifica el nombre que lleva el módulo. Architecture es la arquitectura del módulo referido por el nombre con el que se etiquetó entity, conteniendo la descripción detallada de la estructura interna o del comportamiento que tiene el módulo.

Verilog declara una sola instancia, que es propiamente el módulo y se determina utilizando la palabra reservada module. Dentro del mismo módulo se declaran los puertos de interfaz externa y el funcionamiento que presenta la pieza de hardware diseñada.

A continuación se listan los códigos en VHDL y Verilog, que ejemplifican el modelado de un circuito combinatorio que realiza operaciones lógicas básicas.

```vhdl
-- Codificación en VHDL
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- Declaración de módulo y puertos
entity MOD_OP is
   port (A, B): std_logic;
   C, D: out std_logic);
end MOD_OP;
-- Comienza funcionamiento
architecture FUNCIONAMIENTO of MOD_OP is
begin
   C <= A and B;
   D <= A or B;
end FUNCIONAMIENTO;
```

```verilog
// Codificación en Verilog
// Declaración del módulo
module MOD_OP (D, C, B, A);
// Declaración de puertos
input A, B;
output C, D;
// Comienza funcionamiento
assign C = A & B;
assign D = A || B;
endmodule
```

Nótese que la sintaxis en VHDL es más complicada que la de Verilog; es imprescindible incluir las bibliotecas IEEE del estándar[18] para VHDL, mientras que Verilog no las requiere. En los códigos anteriores, MOD_OP (Módulo Operación) es el nombre asignado al módulo diseñado. En el caso del código en VHDL, FUNCIONAMIENTO, representa el nombre de la arquitectura que describe el comportamiento de MOD_OP.

Para que el diseño sea secuencial en vez de combinatorio, se incluye el lazo de retroalimentación que permite la asincronía o la sincronía, según convenga a los requerimientos propuestos. En VHDL, dentro de architecture se declara un process que iniciará cuando una variable (diseño asíncrono) o cuando el reloj (diseño síncrono)

[18] *Si se deseara trabajar con operaciones aritméticas, sería necesario incluir las bibliotecas que lo permiten. VHDL tiene una cantidad grande de paquetes de bibliotecas, aunque las más comunes están en el paquete del estándar 1164.*

lo permitan. En el caso de Verilog, la palabra reservada equivalente es always @. En el **capítulo 3** se muestran ejemplos de codificación bajo estos rubros.

2.7.2 Niveles de Abstracción

Un diseño mediante HDL, inicia con el planteamiento tradicional de método de descripción. El proceso de diseño VLSI puede verse como una secuencia de transformaciones sobre representaciones físicas del modelo a partir de una descripción del *comportamiento*[19] *(behavioral*) o una descripción *estructural (structural*) del mismo.

La *descripción del comportamiento* representa el funcionamiento de un circuito con respecto a sus entradas y salidas; esto es, se modela la operación integral del circuito desde un panorama muy exterior, por lo que se considera de alto nivel de abstracción. La *representación estructural* describe la conformación del circuito en términos de los componentes y las interconexiones entre ellos, connotando un nivel bajo de abstracción.

Cada una de las descripciones tiene connotaciones más formales, por ejemplo a la descripción de comportamiento se le llama también *RTL (Register Transfer Level – Transferencia a Nivel de Registros*), mientras que a la estructural se le conoce como *GTL (Gate Transfer Level – Transferencia a Nivel de Compuertas Lógicas*).

La mayoría de los diseños complejos (por ejemplo, un procesador) se describen a nivel RTL, en consecuencia a que es más simple describir un comportamiento que realizar un diseño componente a componente. La desventaja es obvia, a nivel registro perdemos de cierta forma el control de las acciones, suponiendo solamente el funcionamiento sin reparar en cómo se está llevando a cabo y por quién. Hace algunos años era un verdadero inconveniente; sin embargo, la evolución y sofisticación de las herramientas VLSI CAD, soporta algoritmos inteligentes que compilan de la mejor manera un diseño, dejando a consideración del diseñador si se desea un circuito con restricciones de *velocidad* o de *espacio físico*.

Independientemente de la descripción abstracta global del diseño (*Comportamiento o Estructural*), es necesario definir el método que se seguirá para jerarquizar los diferentes niveles de concepción con los que se trabajará. Principalmente, los diseñadores definen dos métodos para jerarquizar los niveles de

[19] *Algunas bibliografías, denotan a este tipo de abstracción como Comportamental.*

desarrollo en función de una operatividad por bloques, donde cada bloque realiza de manera independiente una tarea.

1. *Diseño de Abajo hacia Arriba(Bottom –Up).*Se trata de un método *ascendente* mediante el cual se realiza la descripción del circuito a modelar, empezando por describir los componentes más básicos del sistema (primitivas) para posteriormente agruparlos en diferentes módulos, y éstos a su vez en otros módulos hasta llegar a uno solo que representa el sistema completo. Esta metodología comienza con un nivel bajo de abstracción (nivel de compuertas) y se dirige progresivamente hacia un nivel alto de abstracción (nivel de registros). Una aproximación a este método se representa en el diagrama a bloques de la **figura 2.12**. Para diseños grandes, no es factible conectar miles de componentes a bajo nivel y pretender que el diseño completo funcione adecuadamente. Por la misma razón el flujo *Bottom – Up* no es del todo recomendable.

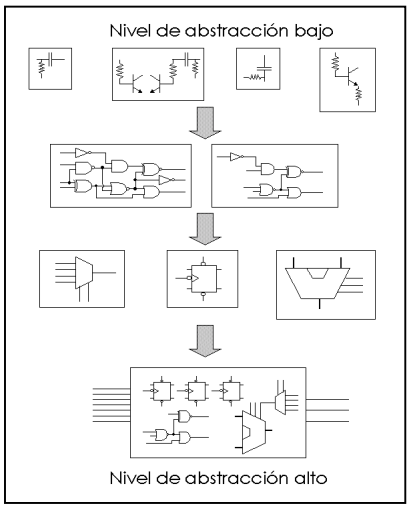


Figura 2.12: **Diseño Bottom - Up.**

2. *Diseño de Arriba hacia Abajo (Top – Down).* Este método parte de una idea en un alto nivel de abstracción y después proseguir la descripción hacia abajo, incrementando el nivel de detalle según sea necesario. El circuito inicial se particiona en diferentes módulos, cada uno de los cuales se encuentra a su vez subdividido hasta llegar a los elementos primarios de la descripción, siguiendo un flujo *descendente* (**figura 2.13**). No necesariamente se debe alcanzar un nivel de primitivas, ya que un planteamiento correcto no lo permite. El método *Top – Down* es ampliamente utilizado en la actualidad, ya que permite dividir un circuito grande en otros circuitos más pequeños derivados del mismo, lo que permite tratar de manera más personal un módulo sin llegar a un nivel de

abstracción muy bajo. Expresamente, este tipo de jerarquización presenta tres ventajas considerables:

a. *Incrementa la productividad del diseño*. Al especificar un diseño en HDL, el software de desarrollo generará automáticamente el nivel correspondiente de compuertas lógicas, por lo que el tiempo utilizado en un diseño disminuye radicalmente.

b. *Incrementa la reutilización del diseño*. En el proceso de diseño VLSI se utilizan *tecnologías genéricas (Circuitos Integrados Genéricos*), esto es que la tecnología a utilizar no se fija sino hasta llegar al *diseño físico (mapeo, colocación y ruteo,* ver **figura 2.10***)*, permitiendo reutilizar los datos del diseño únicamente cambiando la tecnología de implementación.

c. *Rápida detección y predicción de errores*. Como es necesario un claro análisis en la definición de la descripción del diseño, es posible detectar y predecir errores desde el momento de la modularización.

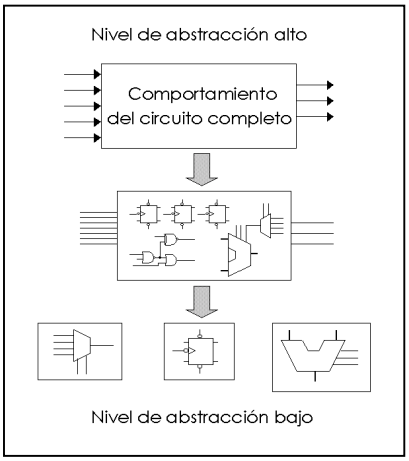


Figura 2.13: **Diseño Top - Down.**

Resumen del Capítulo

En este capítulo, se revisó un breve compendio sobre las Tecnologías Programables, con el propósito de conocer las herramientas y los dispositivos electrónicos que en conjunto permitieron la implementación del controlador. En el **Capítulo 3** se procederá a describir la metodología y el desarrollo seguido para resolver el problema de la realización electrónica del controlador.

Capítulo 3

Modelado de FLCs, Utilizando HDLs

Contenido del Capítulo

Este Capítulo representa el grueso del desarrollo de la tesis. Más que resolver el algoritmo de control bajo mecanismos de lógica difusa, el cual resulta muy intuitivo, se presentan modelados en HDL que son el resultado optimizado de una investigación dirigida a la realización electrónica digital.

Así, la sustentación planteada comienza con el análisis y diseño por separado de cada interfaz, proponiendo soluciones arquitecturales de procesamiento serie y paralelo, con la intención de comparar resultados y comprobar que cualquier otra topología ajena estará considerada dentro de las aportaciones originales generadas por este trabajo. Las comparaciones elementales están en función de la cantidad de Bloques Lógicos Configurables utilizados en la sintetización y en los ciclos de reloj consumidos en el proceso total.

*Todos los códigos generados están contenidos en el **CD** que acompaña esta tesis. Algunos otros que auxilian a la explicación del trabajo, se listan parcialmente con la finalidad de evitar referencias redundantes.*

La descripción principal del funcionamiento de cada interfaz se realizó en Verilog. Opcionalmente, algunos códigos indicados puntualmente también se describieron en VHDL, con la intención de no restringir los diseños hacia un solo HDL, respetando que la mayoría de los desarrollos actuales de la misma índole se realizan en VHDL.

3.1 El Esquema Básico del FLC Digital

La arquitectura FPGA, estructuralmente compuesta por arreglos de lógica digital programable, permite entrever la viabilidad de realizar dentro de él, cualquier derivación que conserve la misma condición digital. La implementación electrónica es completamente sostenible, preponderando flexibilidad, velocidad y exactitud.

La naturaleza del proceso de lógica difusa aplicada al control de sistemas, como ya se ha comentado con anterioridad, es apreciablemente serial debido a que cada una de las interfaces funcionales básicas (fuzificador, inferencia y defuzificador) están supeditadas entre sí. La lógica difusa se analiza como una metodología para la obtención de mapeos no lineales, apoyándose en un conjunto de reglas que modelan el control del sistema; así mismo, la lógica programable proporciona los recursos necesarios para la realización de mapeos booleanos. El concentrado del diseño formal compromete convertir el mapeo de control no lineal en una estructura física de principio digital soportada en el FPGA. Es factible presuponer la vital importancia del estudio de la problemática de cuantificación y truncamiento de datos, al tratarse de un sistema digital con longitud de palabra finita.

En la **figura 3.1**, se observa el diagrama a bloques de un FLC convencional. Se aprecian las interfaces básicas y su integración, así como su interacción con el exterior, conformando el sistema de control. El ajuste, que por lo general es un restador, también está considerado dentro del bloque general correspondiente al controlador; no obstante, se considerará hasta la integración de las interfaces, aspecto que se desarrolla en el **Capítulo 4** de este mismo documento.

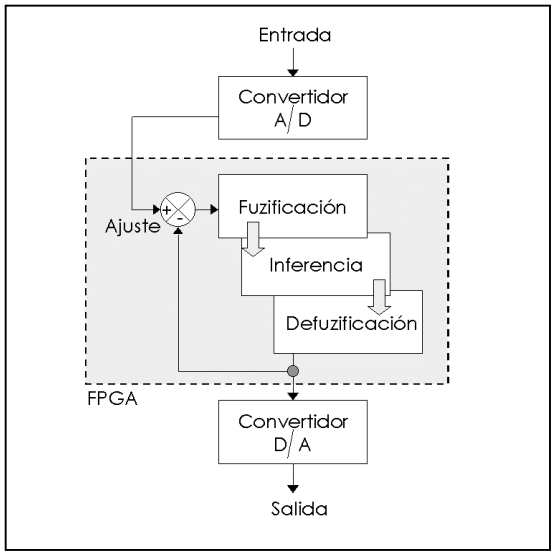


Figura 3.1: **Diagrama a bloques de un controlador convencional basado en lógica difusa.**

La *unidad de adquisición de datos*, correspondiente a la entrada analógica y su conversión a digital, es de suma importancia debido a que el dato convertido no debe acceder hacia el controlador sino hasta que éste lo considere conveniente (una vez terminado el procesamiento previo), habilitando para ello las señales apropiadas. El diseño de esta unidad adquisitiva, es el mismo que se sigue para cualquier otra aplicación, por lo que se considera fuera de los alcances propuestos para los subsecuentes apartados.

3.1.1 Ejercicio de Aplicación: Controlador de Irrigación

El objetivo está encaminado a encontrar soluciones adaptables a cualquier sistema de control sin importar el contexto de diligencia. Para unificar criterios, se premedita el ejercicio de un controlador de irrigación cuya metodología de diseño será la misma sugerida para cualquier otro controlador. Se propone que una *válvula de irrigación* se controle con dos variables de entrada: *Temperatura del Aire* y *Humedad de la Tierra*. El FLC deberá hacer que la combinación de ambas variables determine el *Tiempo de Irrigación*. Las unidades de medida de cada variable se consideran normalizadas en un rango de *0* a *255*, para un controlador de 8 bits, sin signo y con escalares de punto fijo[1].

Tomando como referencia las funciones de pertenencia fijadas por un experto humano [6], las cuales al simularse presentan un comportamiento lineal satisfactorio del controlador, se obtienen los universos de discurso. Las **figuras 3.2** y **3.3**, muestran las funciones de pertenencia respectivas de fuzificación.

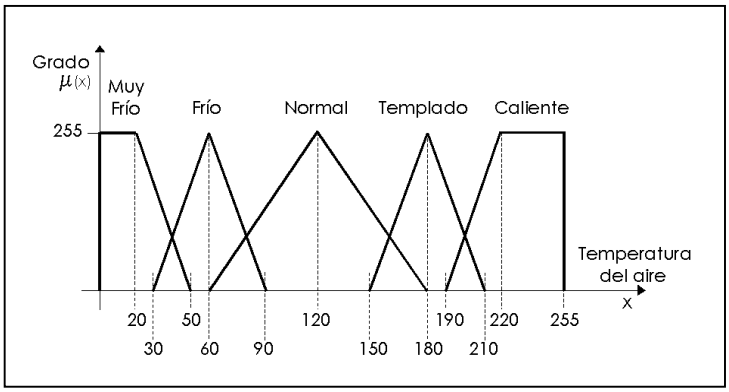


Figura 3.2: **Funciones de pertenencia para la variable de entrada *Temperatura del Aire*.**

[1] *El análisis simple comienza con funciones simétricas y cálculos sin signo. Posteriormente, en este mismo trabajo, se complementa la solución tratando con funciones asimétricas y proponiendo una aproximación a la aritmética signada.*

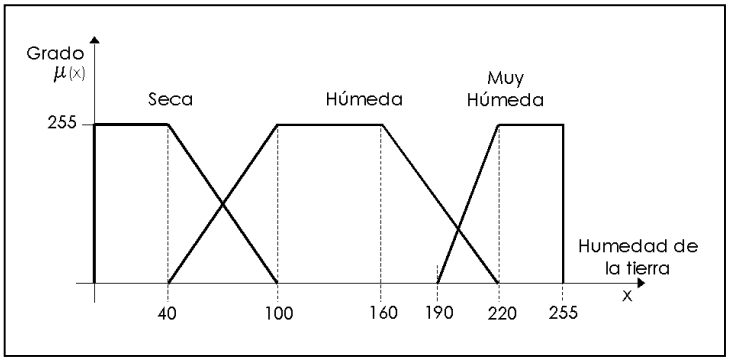


Figura 3.3: **Funciones de pertenencia para la variable de entrada *Humedad de la Tierra*.**

La **figura 3.4**, exhibe las funciones de pertenencia *singletons*, para defuzificar los valores y calcular el valor real de salida.

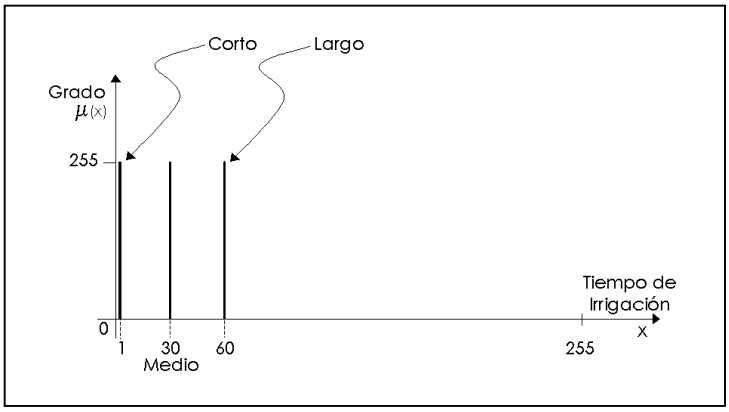


Figura 3.4: **Funciones de pertenencia para defuzificar la variable de salida *Tiempo de Irrigación*.**

Por tratarse de dos variables de entrada con 5 y 3 funciones de pertenencia respectivamente, la base del conocimiento está compuesta por las 15 reglas anotadas en la **tabla 3.1**.

Tabla 3.1: **Reglas de inferencia difusa para el controlador de irrigación propuesto.**

Tiempo de Irrigación		*Temperatura del Aire*				
		Muy Frío	Frío	Normal	Templado	Caliente
Humedad de la Tierra	Muy Húmeda	Corto	Corto	Corto	Corto	Corto
	Húmeda	Corto	Medio	Medio	Medio	Medio
	Seca	Largo	Largo	Largo	Largo	Largo

3.2 Aproximación al Diseño de la Interfaz de Fuzificación

En el **capítulo 1** se dio una breve introducción al algoritmo que se utiliza para realizar el control de algún sistema bajo mecanismos de lógica difusa. En consecuencia, el estudio se centrará en cómo es posible implementar la interfaz de fuzificación vertida en un FPGA proponiendo soluciones de procesamiento serie y paralelo, modeladas a través de HDLs. Recordando lo expuesto, las formas más usuales de las funciones de pertenencia o membresía son cuatro: *singleton, gaussiano*, *trapezoidal* y *triangular*, los cuales se muestran en la **tabla 1.3**, del mismo capítulo. En el análisis aquí expuesto, se omite lo referente a las funciones de tipo gaussiano debido a que su realización está más ligada a soluciones de electrónica analógica a causa de la naturaleza de su comportamiento. Además, de acuerdo al tópico **1.3.1.2**, las soluciones presentadas a continuación están sujetas de manera preferente a *Modelos Orientados a Cálculos*.

Una interfaz fuzificadora con n variables de entrada, estará compuesta por n módulos resolutivos, donde cada módulo corresponde a una única variable de entrada. Cada uno de estos módulos, a su vez, está compuesto por varias unidades básicas fuzificadoras cuya tarea consiste en calcular el grado de pertenencia con respecto a un tipo de fuzificador preestablecido. Los grados calculados representan las salidas de la interfaz fuzificadora que se conectarán a la interfaz de inferencia. La **figura 3.5**, muestra la idea anterior.

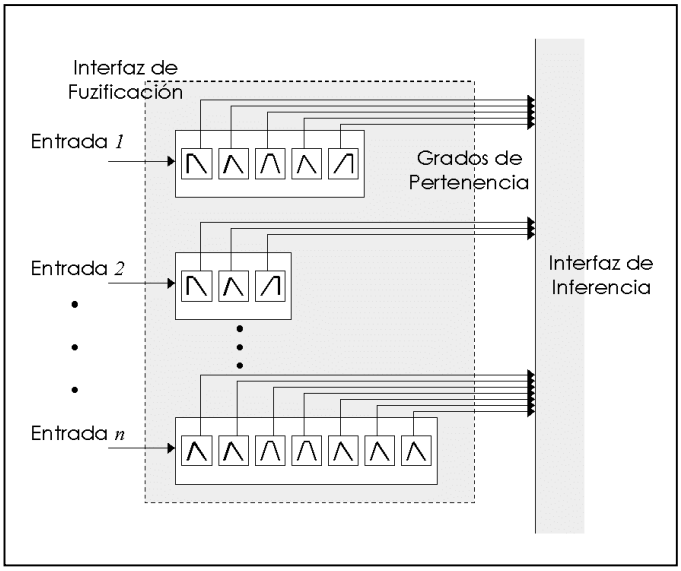


Figura 3.5: **Modularización de la interfaz de fuzificación.**

Sosteniendo la premisa anterior, supóngase el conjunto de funciones de pertenencia mostrado en la **figura 3.6**, dentro del universo de discurso de una unidad fuzificadora convencional, con un rango de valores de entrada desde *0* hasta *255* para un controlador de 8 bits, sin signo y de punto fijo.

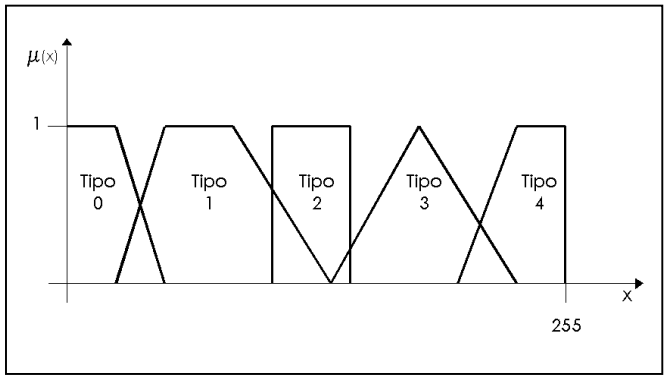


Figura 3.6: **Funciones de pertenencia más comunes.**

Se advierte que cada función de pertenencia se construye por al menos dos *bloques de formación*. Un bloque de formación es la figura geométrica que limita la forma gráfica; en este caso, rectángulos y triángulos.

Es posible simplificar el análisis matemático, al existir similitudes notables en el tratamiento de las funciones. Si se asume con respecto a la **figura 3.6**, que el fuzificador *tipo 0* en suma con el *tipo 4* forman el *tipo 1* (fuzificador *trapezoidal*), se elimina el análisis independiente de éste último. De la misma forma, el fuzificador *tipo 2* se considera implícito a causa de su naturaleza *singleton*. El marco de referencia se resume a únicamente tres fuzificadores base: *tipo 0* (inclusiva derecha), *tipo 1* (triangular, inclusiva simétrica/ asimétrica) y *tipo 2* (inclusiva izquierda). La **figura 3.7**, muestra la simplificación conceptual**.**

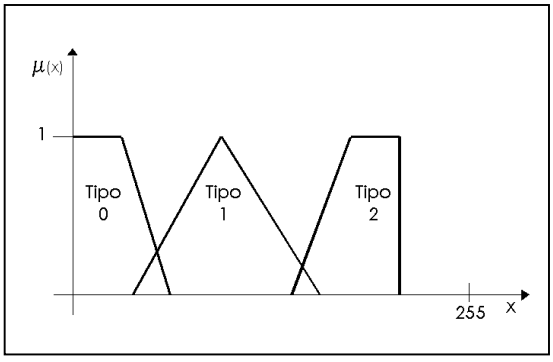


Figura 3.7: **Conjunto reducido de funciones de pertenencia.**

3.2.1 Estrategias para el Cálculo del Grado de Pertenencia

De manera independiente, cada unidad básica fuzificadora calcula su propio grado de pertenencia (**figura 3.5**). El algoritmo que resuelve el trabajo del fuzificador, se presenta como una respuesta al cálculo de un valor y en las ordenadas (grado de pertenencia), en correspondencia a un valor de entrada x en las abscisas.

La **figura 3.8**, muestra una función *triangular* de pertenencia (*tipo 1*, con respecto a la **figura 3.7**), compuesta por dos pendientes ambiguas, lo que sugiere que la solución estará en términos de dos ecuaciones de recta. Nótese que la distancia entre a y b es diferente a la distancia entre b y c, por lo tanto las pendientes son asimétricas.

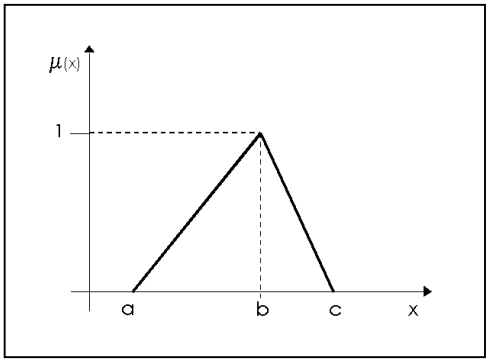


Figura 3.8: **Fuzificador triangular con pendientes diferentes.**

Para calcular el grado de pertenencia μ, ante un valor de entrada x, se tienen dos estrategias de cálculo que difieren en el algoritmo de solución. Ambas están optimizadas[2] y son plenamente aplicables.

3.2.1.1. Estrategia 1: Algoritmo de los Puntos Extremos

Este algoritmo resuelve utilizando los valores a y c, en ambos extremos de la función de pertenencia (ver **figura 3.8**); se hace extensivo para tratamientos simétricos y asimétricos. La función se divide en dos partes: el intervalo [a, b] y el intervalo [b, c], cada uno de éstos permite cálculos individuales de la pendiente y del grado de pertenencia respectivo. Si la entrada x está entre el intervalo [a, b], factiblemente se calcula el grado de pertenencia μ, mediante

[2] *El cálculo matemático ofrece varios algoritmos similares para obtener la solución; sin embargo, no todos están optimizados con respecto al código utilizado para su implementación física.*

$$\mu(x) = (s)(x - a), \tag{3.1}$$

donde s es la pendiente (*slope*, en inglés) y se calcula por medio de la relación clásica

$$s = \frac{\mu_{máximo}}{(b - a)}. \tag{3.2}$$

El valor $\mu_{máximo}$, es una constante determinada por la palabra máxima finita del controlador, igual a 2^{n-1}; en la particularidad del diseñado en este estudio se tiene que $\mu_{máximo} = 255$, por tratarse de un controlador de 8 bits.

La **ecuación 3.1** representa la pendiente con tendencia positiva y caracteriza la solución matemática para gráficas de este tipo. Para el valor de μ con respecto a un valor x dentro del intervalo [b, c], se tiene una pendiente negativa, por lo que el cálculo se realiza mediante

$$\mu(x) = (s)(c - x), \tag{3.3}$$

y para el cálculo de la pendiente negativa, se aplica

$$s = \frac{\mu_{máximo}}{(c - b)}. \tag{3.4}$$

El algoritmo de codificación basado en la *estrategia 1*, se resume como sigue:
"Si x no está entre a y c, x no tiene *grado* de pertenencia μ"
"Si x está entre a y b, calcular el *grado* de pertenencia μ utilizando las **ecuaciones 3.1** y **3.2**"
"Si x está entre b y c, calcular el *grado* de pertenencia μ utilizando las **ecuaciones 3.3** y **3.4**"

La **figura 3.9**, muestra un diagrama a bloques del hardware que implicaría una solución con la *estrategia 1*. Los multiplexores conmutan entradas de acuerdo a un comparador. El *comparador de pertenencia* [a, c],conectado a *Mux3*, verifica que la variable x pertenezca a la función, de no ser así el grado será 0. Si x corresponde al intervalo de la función, entonces tiene un grado de pertenencia que se evalúa con ayuda del *comparador de pertenencia* [a, b], que es el encargado de decidir en qué intervalo se trabajará. Si x está entre [a, b] la salida del comparador es *1*, propiciando que por *Mux1* entre x y por *Mux2* entre a, para así tener $x - a$, como una entrada al *multiplicador* con la pendiente correspondiente s. En caso contrario, las entradas del *restador* serán c y x, para tener $c - x$ como entrada al *multiplicador* con su correspondiente pendiente s. Si las pendientes fueran asimétricas sería necesario adicionar un multiplexor que seleccionara la pendiente en cuestión.

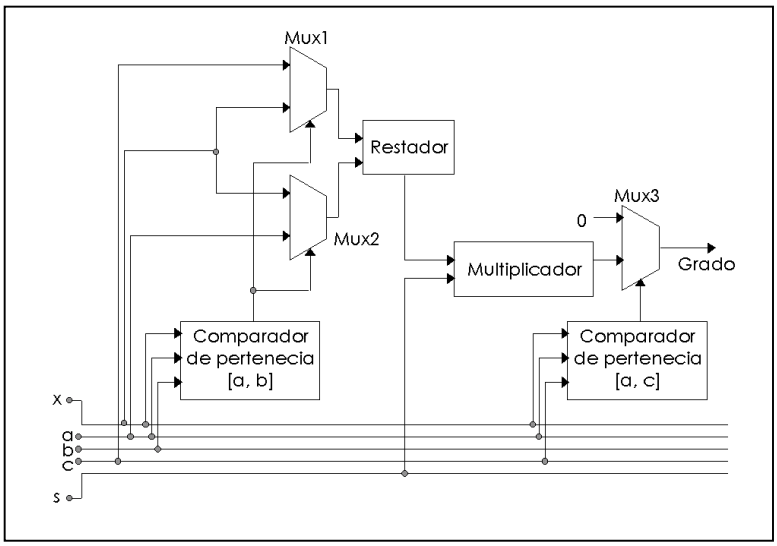


Figura 3.9: **Módulo resolutivo fuzificador, implementado bajo la *estrategia 1*.**

3.2.1.2. Estrategia 2: Algoritmo del Punto Medio

Es viable calcular el grado de pertenencia considerando únicamente el valor del punto medio b, de la función. Para aplicar este algoritmo, se utiliza la **ecuación 3.3**, con ciertas modificaciones:

$$\mu(x) = \mu_{máximo} - (s)(delta), \tag{3.5}$$

donde, si x está entre $[a, b]$

$$delta = b - x \tag{3.6}$$

o si está entre $[b, c]$, hacer

$$delta = x - b. \tag{3.7}$$

Las pendientes se calculan igual que en las **ecuaciones 3.2** y **3.4**; el valor máximo del rango, sigue siendo $\mu_{máximo} = 2^{n-1}$.

El algoritmo de codificación de la función de pertenencia triangular (**figura 3.4**) basado en la *estrategia 2*, se resume así:
"Si x no está entre a y c, x no tiene *grado* de pertenencia μ"
"Si x está entre a y b, hacer $delta = b - x$, y calcular el *grado* de pertenencia μ utilizando las **ecuaciones 3.5 y 3.2**"
"Si x está entre c y b, hacer $delta = x - b$, y calcular el *grado* de pertenencia μ utilizando las **ecuaciones 3.5 y 3.4**"

La **figura 3.10**, muestra un diagrama a bloques del hardware de formación que implicaría una solución con la *estrategia 2*.

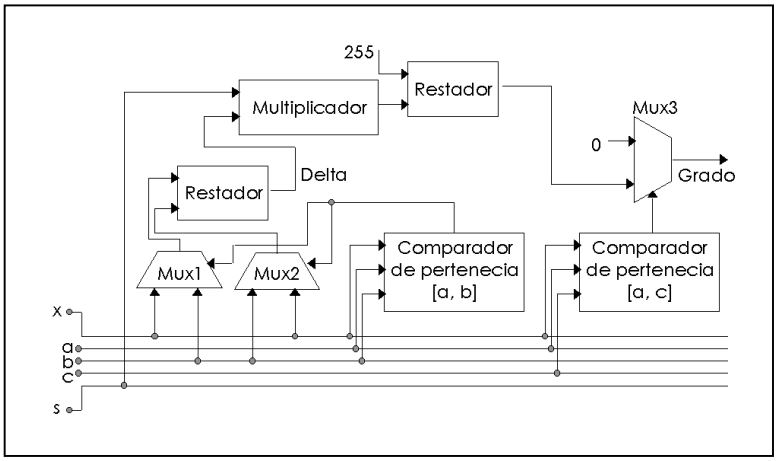


Figura 3.10: **Módulo resolutivo fuzificador, implementado bajo la estrategia 2.**

El *comparador de pertenencia* [a, b] determina con ayuda de *Mux1* y *Mux2*, qué valor tendrá *delta* con respecto al intervalo de pertenencia. La salida del primer *restador* entrará al *multiplicador* junto con el valor de la pendiente s correspondiente. La salida del multiplicador se restará a 255 (rango máximo). Finalmente, el *comparador de pertenencia* [a, c] determinará si x está entre [a, c] para validar la pertenencia. Al igual que se comentó en la *estrategia 1*, si se tuvieran pendientes diferentes sería necesario incluir un multiplexor cuya tarea sería seleccionar la pendiente adecuada a cada intervalo.

3.2.2 Soluciones Fuzificadoras de Procesamiento Serie

Si la aplicación no requiere de grandes velocidades de procesamiento, sino al contrario, lo que requiere es reducir el número de módulos lógicos utilizados y así minimizar la cantidad de hardware, una interfaz fuzificadora con procesamiento de índole serie es adecuada.

Tomando como referencia cualquiera de los módulos resolutivos de la **figura 3.5**, indiscutiblemente es necesario buscar mecanismos para que cada unidad fuzificadora individual actúe en un determinado tiempo sobre un multiplicador común, con la finalidad de que cada una de ellas calcule su correspondiente grado de pertenencia mediante alguna de las estrategias matemáticas comentadas anteriormente. La

problemática visible es la secuenciación de operaciones, debido a que el multiplicador únicamente podrá ser accedido por una sola unidad a la vez. La solución presenta matices síncronas (si se opta por utilizar ciclos de reloj para controlar todas las operaciones) y asíncronas (si se opta por elegir un cambio en la entrada y un ciclo suficientemente grande para garantizar que se lleven a cabo todas las operaciones pertinentes).

Si en la fuzificación de una variable se tienen funciones de pertenencia con traslape, existen varios grados de pertenencia a la vez, como se aprecia en la **figura 3.11**. En consecuencia, en la topología serie todas las unidades fuzificadoras utilizan las mismas líneas de salida, por lo que es inevitable considerar guardar el resultado de todos los grados calculados y presentarlos al final del proceso como salidas individuales.

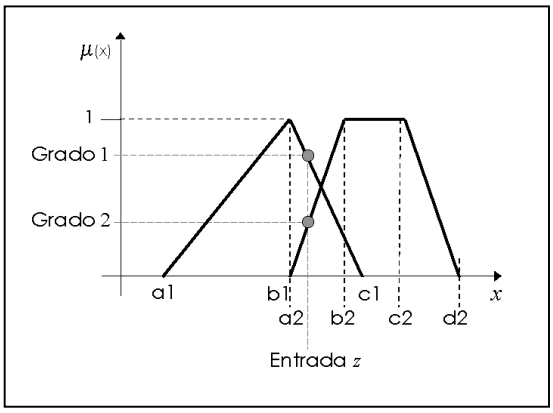


Figura 3.11: **Funciones de pertenencia traslapadas.**

Debido a los bits de resolución, se tiende a acotar el grado de pertenencia máximo ($\mu_{máximo}$) con un valor decimal indicado de *255,* debido a que el controlador propuesto es de 8 bits y $2^8=256$, con un rango de valores desde *0* a *255*, por lo que el marco de referencia del controlador será proporcional en ambos ejes.

En el caso de una función de pertenencia trapezoidal, se procederá a construirla a través de la unión gráfica de una función del *tipo 0* y una del *tipo 2*, de las analizadas en la **figura 3.7**. En este trabajo, se denominarán funciones del *tipo 3*, aunque para fines algorítmicos se considerarán las funciones *tipo 0* y *tipo 2*. Para la nomenclatura referente a este tipo de funciones, se propone el ejercicio marcado en la **figura 3.12**.

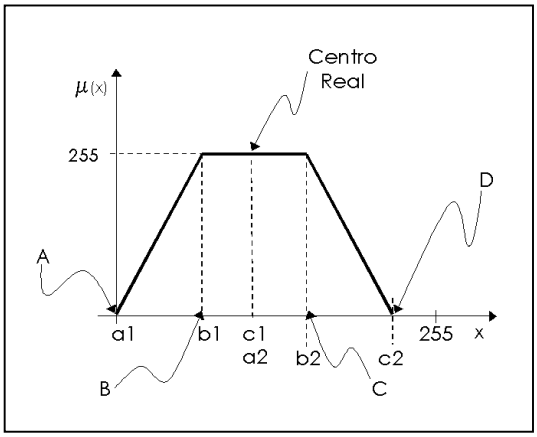


Figura 3.12: **Función de pertenencia trapezoidal, formada por una función del tipo 0 unida con una del tipo 2.**

3.2.2.1 Comparación de las Estrategias Para el Cálculo del Grado de Pertenencia, Bajo el Modelo Serie

Para ejemplificar este apartado, se hará referencia a la **figura 3.2** que muestra el conjunto de funciones de pertenencia para la variable de entrada *Temperatura del Aire*, considerando el diseño del controlador de irrigación propuesto.

Cualquier solución de procesamiento serie, implica la utilización de una memoria de datos. Los datos que debe almacenar la memoria son los límites gráficos y el valor de la(s) pendiente(s)[3]. Una máquina de estados, cuya complejidad es variable, accederá a la memoria para leer los datos, uno a uno, y así colocarlos disponibles en el módulo resolutivo creado para la interfaz de fuzificación. El proceso debe repetirse cada vez que cambie el valor de la entrada. La **figura 3.13**, representa una aproximación abstracta a nivel alto del módulo diseñado.

Existen diferentes variantes, independientemente de la estrategia de cálculo para el grado de pertenencia, apoyadas por el uso de la memoria. Si se opta por un controlador más específico que general, la mejor solución es una sola memoria para todos los datos, otorgando tamaños fijos a todos los valores, especialmente si las funciones de pertenencia se asumen como simétricas. La memoria de datos, puede ser interna o externa al bloque de la interfaz de fuzificación, de cualquier forma dentro del FPGA.

[3] *Si se tratara de funciones de pertenencia asimétricas, invariablemente se tienen pendientes diferentes.*

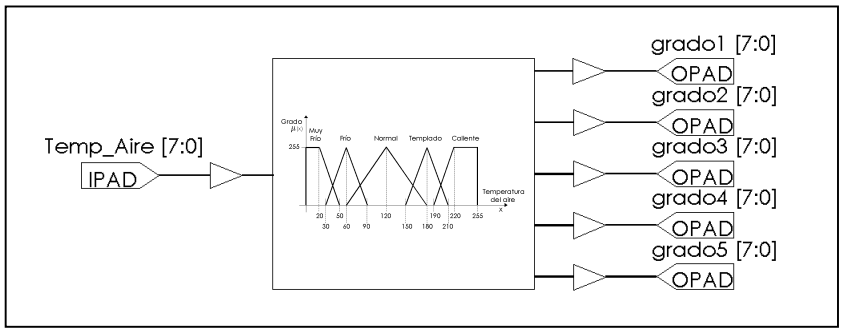


Figura 3.13: **Módulo resolutivo fuzificador, de procesamiento serie, para la variable de entrada *Temperatura del Aire*.**

La codificación para el módulo fuzificador de la variable *Temperatura del Aire*, del controlador de irrigación diseñado, se realiza en Verilog[4]. El código **fuz_TAs1.v**, dentro de la carpeta *fuzificadores Serie* en el **CD**[5], hace referencia a la *estrategia 1* utilizada para resolver la unidad fuzificadora de la *Temperatura del Aire* bajo modelo serie. La memoria está residente dentro del bloque resolutivo, por lo que la máquina de estados forma parte del mismo código. Las formas gráficas y sus respectivos puntos limitantes (**figura 3.2**), así como las pendientes, se definen dentro de la memoria. A través de un ciclo for se accede a la memoria para obtener los valores que definen las diferentes funciones de membresía, evaluando una a una de izquierda a derecha (desde *Muy_Frío* hasta *Caliente*), tal y como se aprecia en el siguiente listado que sólo representa una parte significativa de **fuz_TAs1.v**. La **figura 3.14** muestra los resultados de la simulación.

```
//Fuzificación de la Variable Temperatura del Aire en Verilog,
//utilizando la estrategia 1 para una aproximación serie, Parte significativa fuz_Tas1.v

for (i=0; i<5; i=i+1)          //Realiza 5 iteraciones, una para cada función de
begin                          //Pertenencia

temp=(temp_aire<b[i])?1:0;     //Una variable Temporal que determina si la pendiente es
                               //positiva con 1 ó negativa con 0
  if (tipo[i]==0 & temp==1)    //Identifica Tipo 0 ó 2
  grad_mem[i] =255;
   else if (tipo[i]==2 & temp==0)
   grad_mem[i]=255;
    else if (temp_aire<=a[i] || temp_aire>=c[i])   //Pertenencia a la función
    grad_mem[i]=0;
     else                                          //Cálculo del grado
      grad_mem[i]=(temp_aire<b[i])? pend[i]*(temp_aire-a[i]):pend[i]*(c[i]- temp_aire);
  end
                               //Asigna a cada salida su respectivo grado
grado1=grad_mem[0];
grado2=grad_mem[1];
grado3=grad_mem[2];
grado4=grad_mem[3];
grado5=grad_mem[4];
```

[4] *Todos los códigos generados en esta tesis se realizaron en Verilog; algunos también fueron modelados en VHDL, proponiendo una alternancia híbrida entre módulos esquemáticos.*

[5] *Para conservar la continuidad, de este punto en adelante se asumirá que todos los códigos están listados en el CD anexo a este mismo trabajo, con la salvedad de aquellos que directamente indiquen otra situación.*

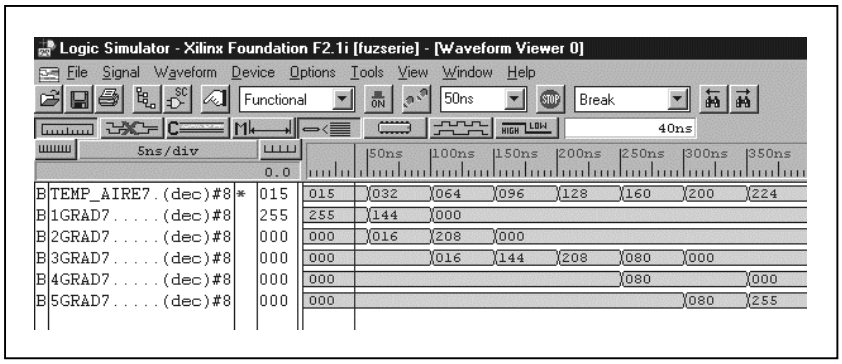


Figura 3.14: **Resultados de simulación para el fuzificador serie de la variable de entrada *Temperatura del Aire, bajo la estrategia 1*.**

En la **figura 3.14** se aprecia que la entrada *x*, denotada para el diseño propuesto como *Temperatura del Aire* (*TEMP_AIRE*), toma diferentes valores colocados de manera horizontal en la gráfica de simulación. Todos los valores están dados en base decimal. Así, cada columna entrega cinco grados de pertenencia para su respectivo valor de entrada; *1GRAD* es la función de pertenencia *Muy_Frío*, así como *5GRAD* es *Caliente*. De acuerdo a lo anterior, para una *Temperatura del Aire = 064*, se tienen los grados *Muy_Frío = 0, Frío = 208, Normal = 16, Templado = 0* y *Caliente = 0*. Para el mismo valor de entrada, la *estrategia 2* (**figura 3.15**) entrega *Muy_Frío = 0, Frío = 223, Normal = 31, Templado = 0* y *Caliente = 0.*

Para la *estrategia 2*, se utilizó el código señalado como **`fuz_TAs2.v`**. Una parte significativa de este código se lista a continuación. En la **figura 3.15** se muestran los resultados obtenidos en la simulación. Se observa que los valores resultantes varían debido a que este segundo algoritmo utiliza una diferencia sobre el rango máximo (*255* en este caso) para el cálculo del grado, como se aprecia en el código siguiente.

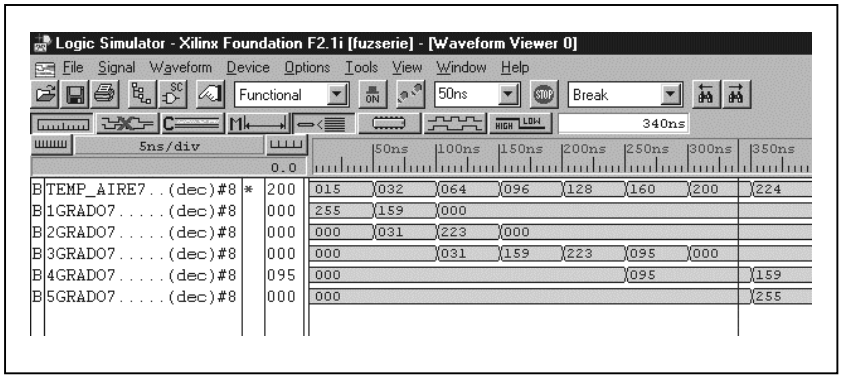


Figura 3.15: **Resultados de simulación para el fuzificador serie de la variable de entrada *Temperatura del Aire, bajo la estrategia 2*.**

```
//Fuzificación de la Variable Temperatura del Aire en Verilog,
//utilizando la estrategia 2 para una aproximación serie
//Parte significativa de fuz_Tas2.v

   for (i=0; i<5; i=i+1)          //Realiza cinco iteraciones, una para cada función de
   begin                          //Pertenencia
     if (Temp_Aire<b[i])          //Determina pendiente positiva con 1 ó negativa con 0
     temp=1;
     else
     temp=0;

     if (Temp_Aire<b[i])          //Cálculo de Delta
     delta=b[i]-Temp_Aire;        //b - x
     else
     delta=Temp_Aire-b[i];        //x - b

     if (tipo[i]==0 & temp==1)    //Identifica tipo 0 ó 2
     grad_mem[i] = 255;
      else if (tipo[i]==2 & temp==0)
      grad_mem[i]=255;
      else
       begin
        if (Temp_Aire <=a[i] || Temp_Aire >=c[i])    //Pertenencia a la función
        grad_mem[i]= 0;
        else
        grad_mem[i]= 255 - (delta*pend[i]);          //Cálculo del grado
      end
     end

   grado1=grad_mem[0];                               //Asigna a cada salida su respectivo
   grado2=grad_mem[1];                               //grado de pertenencia
   grado3=grad_mem[2];
   grado4=grad_mem[3];
   grado5=grad_mem[4];
```

Para comparar los resultados obtenidos en ambas estrategias de manera formal, se analiza particularmente la función de pertenencia *frío* (triangular simétrica, con $a = 30$, $b = 60$ y $c = 90$, ver **figura 3.16**). Asumiendo que su pendiente es $255/30$, cuyo valor truncado es 8, se procede a evaluar una entrada de valor 31, propuesta de manera deliberada para encontrar el *mínimo incremento*. Para la *estrategia 1*, la diferencia entre x y a es 1, por lo tanto el grado calculado es de 8. Para la *estrategia 2*, la diferencia entre b y x es 29, por lo tanto el grado calculado está a razón de resolver $255 - 232$, obteniendo un valor de 23.

El valor exacto real esperado es de 8.5, lo que coincide más hacia la *estrategia 1*; sin embargo, para una entrada igual al valor del punto medio b, la *estrategia 2* alcanza el 255, mientras que la *estrategia 1* el 240. Aún así, la *estrategia 1* conserva una linealidad sin cambios abruptos desde el principio, mejorando la respuesta fuzificadora. En el caso del procesamiento paralelo, como se comentará posteriormente, se tiene un bloque fuzificador para cada función en particular, lo que permite variar el rango máximo a

conveniencia, resultando que si se propone para *frío* un rango máximo de *240* en vez de *255*, los resultados calculados con ambas estrategias serán los mismos. La **figura 3.16** muestra una idea comparativa entre ambos algoritmos y la respuesta real exacta. En el punto **3.2.4** de este mismo documento se hace un análisis de los valores obtenidos por hardware contra los reales esperados, considerando el *error en la exactitud* debido al manejo de unidades de punto fijo.

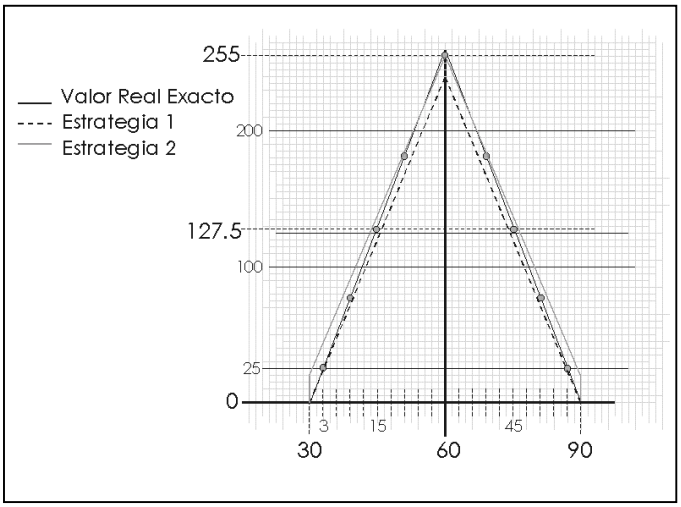


Figura 3.16: **Comparación gráfica de los resultados obtenido en la función de pertenencia *Frío*, bajo las *estrategias 1* y *2*.**

La **tabla 3.2** muestra la cantidad de bloques lógicos utilizados por cada solución sintetizada[6]. Dependiendo del hardware, la versatilidad del fuzificador puede aumentar o disminuir, propiciando un efecto directo sobre la generalidad de la aplicación. El software utilizado para todos los diseños fue *Foundation* de *Xilinx* en su versión *2.1i*, por lo que en cada sintetización de código se proporciona la cantidad de CLBs[7] utilizados, que es un parámetro de espacio físico y granularidad[8].

Tabla 3.2: **Comparación entre resultados de sintetización de los módulos resolutivos fuzificadores *fuz_TAs1 y fuz_TAs2*, bajo ambas estrategias de cálculo.**

Módulo	**Estrategia 1** CLBs utilizados	**Estrategia 2** CLBs utilizados
fuz_TAs	Verilog: 61	Verilog: 68

[6] *Versiones codificadas sólo en Verilog.*
[7] *En el capítulo 3, se determinó que un FPGA está compuesto por módulos lógicos e interconexiones. En el caso de los FPGAs de Xilinx, un módulo lógico es un CLB (Bloque Lógico Configurable – Configurable Logic Blox). La tendencia es comprometer un número mínimo de CLBs bajo una solución óptima.*
[8] *La granularidad, en términos simples puede concebirse como un parámetro que mide la complejidad de las operaciones.*

Si se opta por una memoria de datos externa al módulo resolutivo, es necesario crear por separado una unidad de control de memoria. El código **fuz_TAs3.vhd**, contenido en el **CD** de la tesis, lista una codificación en VHDL que atiende datos provenientes de una memoria ajena al modulo resolutivo. La **figura 3.17**, muestra el bloque fuzificador bajo esta variante.

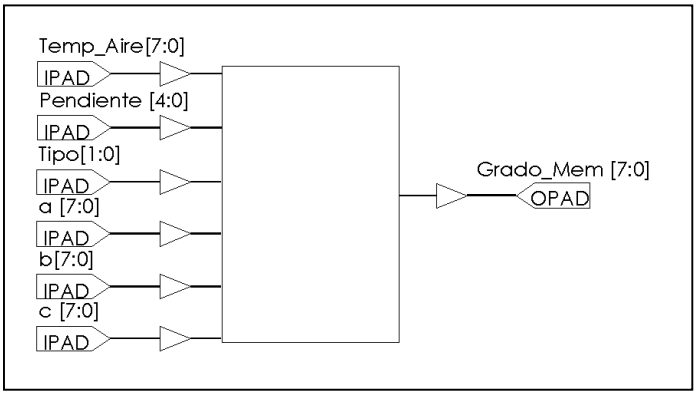


Figura 3.17: **Módulo resolutivo con memoria de datos externa.**

Utilizando la aproximación con memoria externa, se obtuvieron los resultados de síntesis mostrados por la **tabla 3.3**. Los propios de simulación son los mismos expuestos en las **figuras 3.14** y **3.15**.

Tabla 3.3: **Comparación entre resultados de sintetización del módulo resolutivo fuzificador *fuz_TAs3*, bajo la estrategia de cálculo 1.**

Módulo	**Estrategia 1** CLBs utilizados
fuz_TAs2	VHDL: 43

Los códigos **fuz_HTs1.v** y **fuz_HTs2.v,** incluidos en el **CD**, presentan los listados en Verilog de la solución de procesamiento serie para la variable de entrada *Humedad de la Tierra*, bajo ambas estrategias de cálculo. La **tabla 3.4** expone los correspondientes resultados en síntesis. Así mismo, el bloque diseñado está esquematizado de forma abstracta en la **figura 3.18**.

Tabla 3.4: **Comparación entre resultados de síntesis de los módulos resolutivos fuzificadores *fuz_HTs1 y fuz_HTs2*, bajo ambas estrategias de cálculo.**

Módulo	**Estrategia 1** CLBs utilizados	**Estrategia 2** CLBs utilizados
fuz_HTs1	Verilog: 32	Verilog: 42

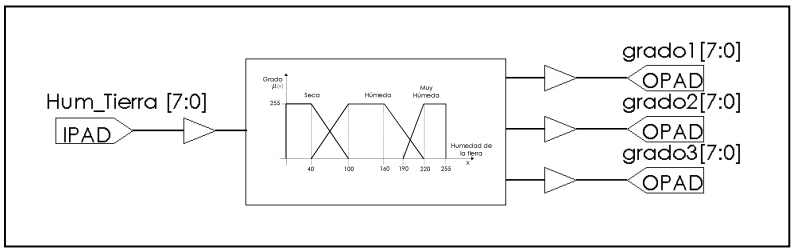


Figura 3.18: **Módulo resolutivo fuzificador, de procesamiento serie, para la variable de entrada *Humedad de la Tierra*.**

En la **figura 3.19**, aparecen los resultados de simulación para el bloque fuzificador sintetizado, utilizando el algoritmo de la *estrategia 1*. La interpretación gráfica es la misma que en la simulación para *Temperatura del Aire* (ver **figura 3.14**). Cuando la *Humedad de la Tierra* (*HUM_TIERRA*) tiene un valor de *48*, los grados calculados para cada función son *Seca =208*, *Húmeda = 32*, *Muy_Húmeda = 0*. La **figura 3.20** muestra los resultados calculados con la *estrategia 2*.

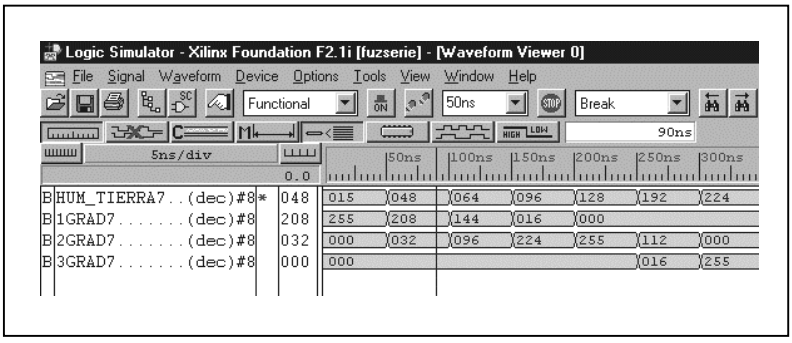


Figura 3.19: **Resultados de simulación para el fuzificador serie de la variable de entrada *Humedad de la Tierra*, bajo la estrategia 1.**

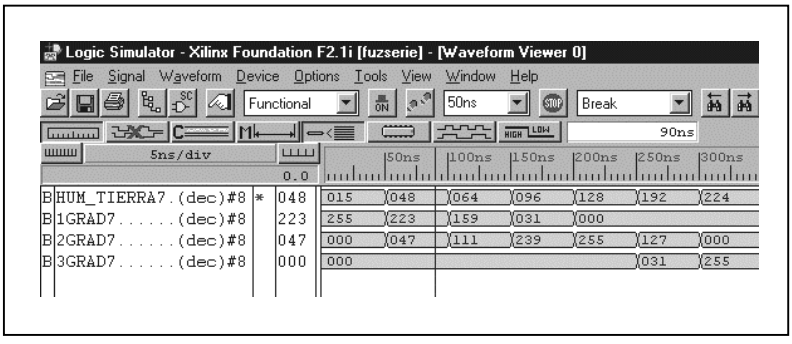


Figura 3.20: **Resultados de simulación para el fuzificador serie de la variable de entrada *Humedad de la Tierra*, bajo la estrategia 2.**

Para que la interfaz fuzificadora serie del control de irrigación quede finalizada, únicamente falta integrar ambos módulos resolutivos en un solo bloque. En el **capítulo 4** se profundizará con mayor precisión en el análisis de la integración modular para la implementación completa del controlador de irrigación.

3.2.3 Soluciones Fuzificadoras de Procesamiento Paralelo

Es totalmente realizable una arquitectura paralela de fuzificación en un FPGA. No obstante que para controlar un motor resulta sobrada[9], se prevé la aplicación hacia áreas relativas de conocimiento donde exista una manifiesta necesidad de rapidez en el procesamiento digital de datos.

Considerando cualquiera de los módulos resolutivos de la **figura 3.2**, se asume la particularidad de que la entrada será aplicada al mismo tiempo a todas las unidades fuzificadoras individuales contenidas en él, por lo que de forma independiente cada una calculará su grado de pertenencia con ayuda de un multiplicador por hardware, favoreciendo automáticamente que el proceso se ejecute de forma paralela. La **figura 3.21** exhibe la topología planteada para el módulo resolutivo de la variable *Temperatura del Aire* del controlador analizado.

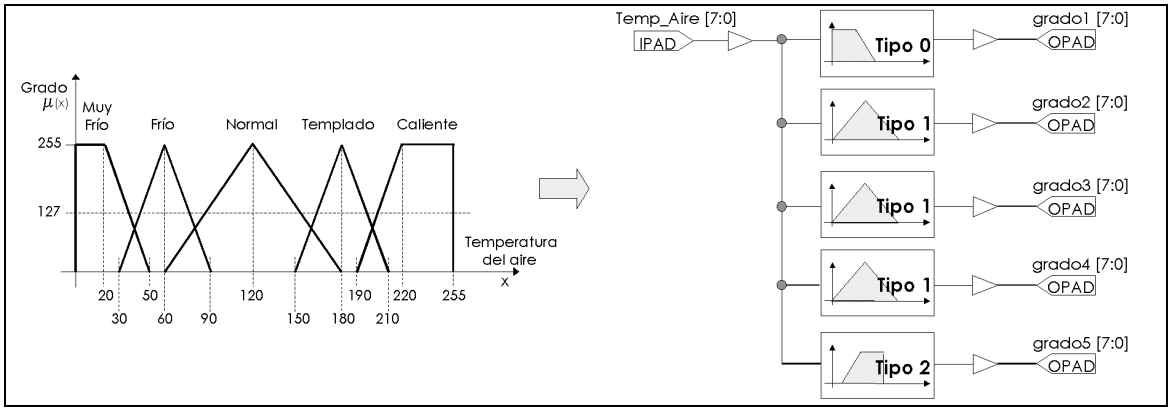


Figura 3.21: **Topología paralela del módulo resolutivo correspondiente a la variable de entrada *Temperatura de Aire*.**

Obsérvese que el esquemático resultante es adaptable a cualquier sistema de control. Se advierte que mantiene un único nivel de paralelismo sin importar el tamaño de la interfaz, ni el orden de las unidades fuzificadoras; la granularidad realizada en

[9] *Por lo general, el oscilador utilizado para respuestas mecánicas es del orden de 16 KHz.*

cada unidad individual resulta aproximada en todos los casos. Para una interfaz fuzificadora más elaborada (con más variables de entrada y más funciones de pertenencia por variable) el sistema es consistente y proporcional a tantas líneas de salida (valor del grado de pertenencia calculado) como bloques básicos fuzificadores existan.

El proceso logra paralelizarse aún más, si se descompone cada una de las unidades fuzificadoras en módulos más pequeños formados por los bloques de formación de la figura (triángulos y rectángulos). Así, la entrada se aplicaría directamente a cada bloque de formación al mismo tiempo. Cada bloque realizará su propio cálculo del grado de pertenencia, por lo que cada uno contiene de manera individual un multiplicador por hardware[10], propiciando el aumento del número de CLBs requeridos. Esta solución perdería por demás generalidad, adicionando una complejidad excesiva en el diseño a medida que existan más funciones de pertenencia. La **figura 3.22** muestra una aproximación al proceso mencionado. Esta segunda paralelización es la máxima permisible por la estructura de la interfaz fuzificadora.

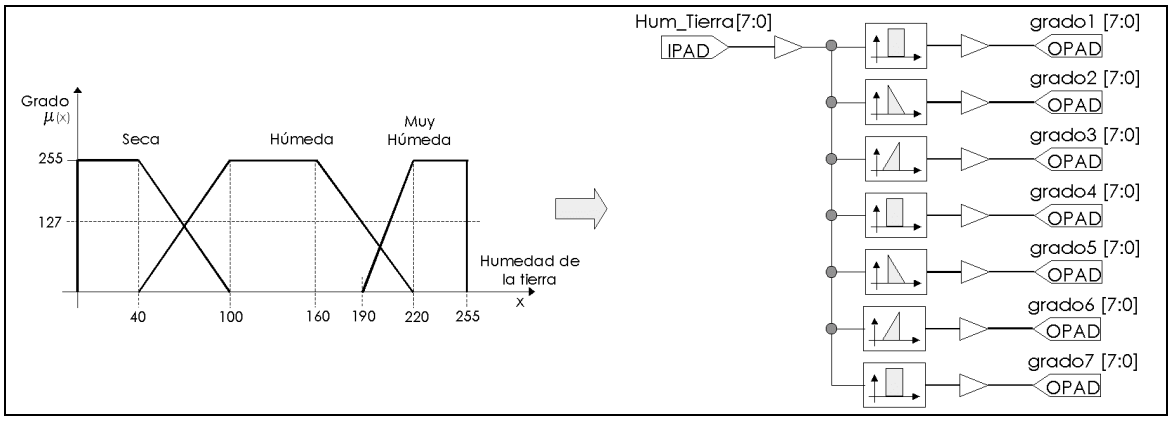


Figura 3.22: **Topología paralela máxima correspondiente a la variable de entrada *Humedad de la Tierra*.**

3.2.3.1 Diseño Individual de las Unidades Básicas Fuzificadoras

Planteado el esquema paralelo simple (no máximo), se procede a diseñar previo examen, cada una de las unidades fuzificadoras individuales comenzando con el fuzificador de tipo triangular (*Tipo 1*, según la **figura 3.7**) que resulta muy evidente en el muestreo gráfico de las pendientes. Para la función fuzificadora básica *Frío*, de la **figura**

[10] *Para mejorar la exactitud en los cálculos, idealmente resulta más efectivo implementar las unidades de multiplicación y de división, dentro de cada unidad fuzificadora. Cuestión que amerita una explicación posterior más formal.*

3.21, considérese $a = 30$, $b = 60$ y $c = 90$. De manera independiente, la función de pertenencia *Frío* se muestra en la **figura 3.23**.

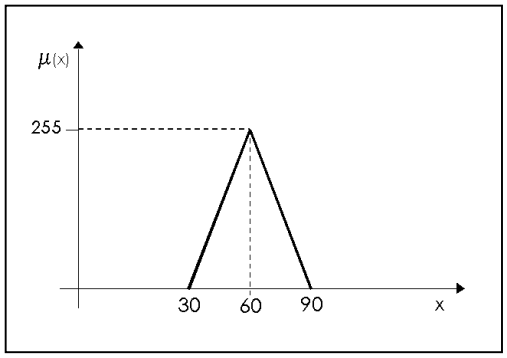


Figura 3.23: **Función de pertenencia *Frío* de la variable de entrada *Temperatura del Aire.***

El algoritmo de solución permite cualquiera de las estrategias de cálculo analizadas, no requiriendo el parámetro *Tipo* de la función como sucede con las topologías serie, debido al carácter individual que adquiere la unidad fuzificadora.

Los códigos siguientes, son aproximaciones que listan parcialmente la solución estimada utilizando la *estrategia 1*. Se proveen las versiones completas, tanto en Verilog (**A_Frío1.v**) como en VHDL (**A_Frío1.vhd**), dentro de la carpeta *Fuzificadores Paralelos* en el **CD** anexo. El acrónimo A_Frío, connota una unidad básica fuzificadora de la variable de entrada *Temperatura del Aire*, con procesamiento paralelo, en su función de pertenencia *Frío*. Los correspondientes a la *estrategia 2* están referidos como **A_Frío2.v** y **A_Frío2.vhd** respectivamente, y consideran el rango máximo de *240*, para que los resultados sean los mismos obtenidos a través de la estrategia *1*, tal y como lo exhibe la **figura 3.24**.

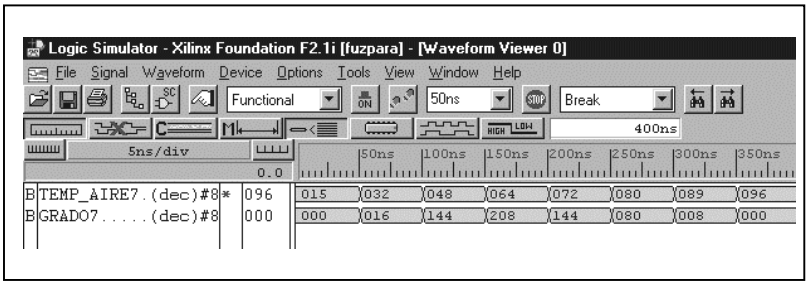


Figura 3.24: **Resultados de simulación para el fuzificador *Frío* bajo las *estrategias1 y 2*. Con un rango máximo de *240* para la *estrategia 2*, los resultados son los mismos.**

La pieza de hardware modelada es de propósito evidentemente particular para los valores dados en la gráfica de la función provista en la **figura 3.23**; sin embargo, el código es reutilizable por completo (aunque VHDL representa una mayor dificultad de sintaxis), únicamente bastaría con cambiar los rangos para adaptar la solución.

```
// Fuzificador de la función de pertenecia Frío, para la variable Temperatura del
// Aire, utilizando la estrategia 1 para una aproximación paralela, en Verilog
// Parte significativa de A_Frío1.v

always @(Temp_Aire)                          // Diseño Asíncrono
begin
 pendiente = 255/30;                         // Definición de la pendiente simétrica
 if (Temp_Aire<=30 || Temp_Aire>=90)         // Límites gráficos
grado = 0;
 else if (Temp_Aire<=60)
grado = pendiente*(Temp_Aire - 30);          // Pendiente positiva
 else
grado= pendiente*(90 - Temp_Aire);           // Pendiente negativa
end
```

```
-- Fuzificador de la función de pertenecia Frío, para la variable Temperatura del
-- Aire, utilizando la estrategia 1 para una aproximación paralela, en VHDL
-- Parte significativa de A_Frío1.vhd

process (Temp_Aire)                                  -- Diseño Asíncrono
begin
 if (Temp_Aire<=30 or Temp_Aire>=90) then            -- Límites gráficos
grado <= "00000000";            --  Previamente se define a=30, b=60, c=90
 elsif (Temp_Aire<=60) then     --  y pend.=255/30
grado <= Conv_std_logic_vector (unsigned(pendiente)*(unsigned(Temp_Aire)-unsigned(a)),8);
 else
grado <= Conv_std_logic_vector (unsigned(pendiente)*(unsigned(c)-
unsigned(Temp_Aire))),8);
 end if;
end process;
```

Cuando las pendientes difieren en una función del tipo anterior, será necesario adecuar la codificación para que cada ecuación del grado de pertenencia, aplique su correspondiente pendiente. Por ejemplo, en la **figura 3.25** se aprecia una función de pertenencia triangular que no tiene pendientes iguales. La solución codificada **`fuz_asim.v`** incluida en el **CD**, no transige mayores modificaciones comparándola con la solución simétrica. A continuación, se lista una parte significativa del mismo código.

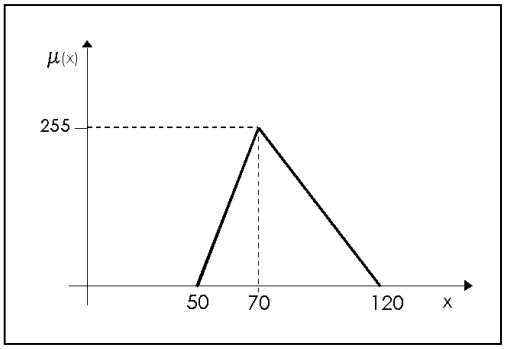


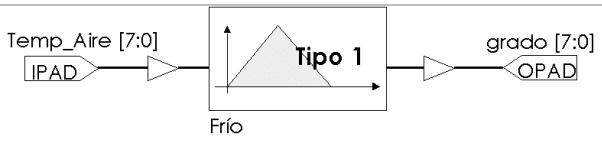
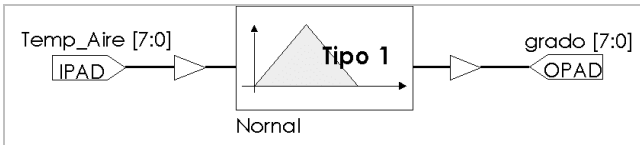
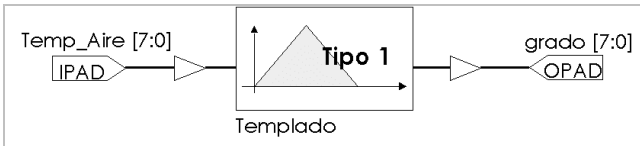
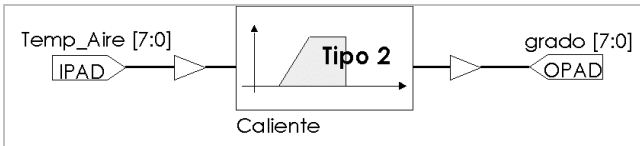
Figura 3.25: **Función de pertenencia triangular con pendientes asimétricas.**

```
// Esta es una aproximación al algoritmo resolutivo, para pendientes asimétricas
// empleando la estrategia 1, codificada en Verilog
// Parte significativa de fuz_asim.v

always @(entrada)                          // Diseño Asíncrono
begin: fuz_triangular_asimétrico
  pend_1 = 255/20;                         // Pendientes asimétricas
  pend_2 = 255/50;
  if (entrada<=50 || entrada>=90)          // Límites gráficos de la función
grado = 0;
  else if (entrada<=70)
grado = pend_1*(entrada - 50);             // Pendiente positiva
  else
grado= pend_2*(90 - entrada);              // Pendiente negativa
end
```

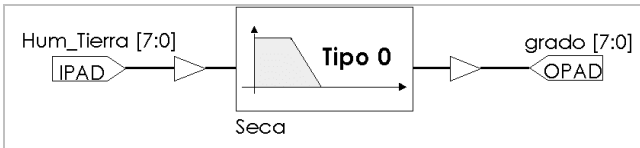
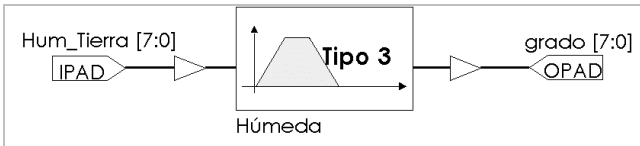
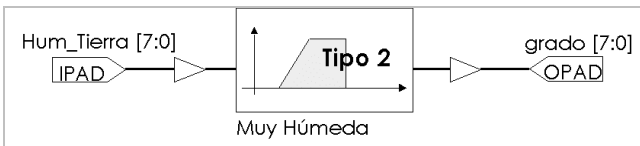
El método se hace extensivo para cualquiera de los fuzificadores lineales inicialmente propuestos. La **tabla 3.5**, resume los bloques construidos para la unidad resolutiva de la variable de entrada *Temp_Aire*. Los códigos completos se encuentran dentro de la carpeta *fuzificadores paralelos* en el **CD** anexo a este trabajo de tesis.

Tabla 3.5: **Comparación entre resultados de síntesis para la unidad resolutiva de *Temp_Aire*.**

Módulo	**Estrategia 1** CLBs utilizados	**Estrategia 2** CLBs utilizados
	Verilog: 11 VHDL: 12	Verilog: 16 VHDL: 16
	Verilog: 13 VHDL: 14	Verilog: 14 VHDL: 16
	Verilog: 18 VHDL: 18	Verilog: 23 VHDL: 24
	Verilog: 14 VHDL: 14	Verilog: 14 VHDL: 14
	Verilog: 5 VHDL: 8	Verilog: 8 VHDL: 12

De manera similar, la **tabla 3.6** muestra los resultados de síntesis causados por la unidad resolutiva para la variable de entrada *Hum_Tierra*. Las soluciones codificadas se aprecian listadas en la misma carpeta del **CD**.

Tabla 3.6: **Comparación entre resultados de síntesis para la unidad resolutiva de *Hum_Tierra*.**

Módulo	**Estrategia 1** CLBs utilizados	**Estrategia 2** CLBs utilizados
	Verilog: 10 VHDL: 12	Verilog: 10 VHDL: 12
	Verilog: 14 VHDL: 17	Verilog: 12 VHDL: 16
	Verilog: 12 VHDL: 14	Verilog: 10 VHDL: 13

3.2.4 Error en la Exactitud, Causado por el Truncamiento

Debido a que la pendiente está previamente calculada mediante un factor específico dado por la relación matemática de la **ecuación 3.2**, donde *pendiente = 255/30* (como se aprecia en el código resolutivo de la **figura 3.23**, de la función Frío), se ocasiona un truncamiento en la exactitud del controlador al aplicar un *módulo 30* para la fracción entera, por lo que el factor para los fines de la unidad individual en hardware no es *8.5*, sino sólo *8*. De este modo, para un valor de entrada *Temp_Aire = 45*, no se obtiene un grado de pertenencia igual a *127.5*, sino de *120* para la *estrategia 1* y *135* para la *estrategia 2*. Para obtener un resultado más exacto, es necesario eliminar la utilización del factor que calcula la pendiente.

Tanto Verilog como VHDL, manejan datos constantes para generar una multiplicación o una división, no siendo admisible la utilización de los operadores * y /, en cálculos que no mantengan un tamaño de datos fijo, lo que trasciende en la necesidad de crear unidades dedicadas mediante código personalizado.

La realización en hardware de multiplicadores y divisores, es motivo de análisis constante debido a las repercusiones que tienen sobre un sistema aritmético. Al igual que en los algoritmos convencionales, tanto multiplicadores como divisores pueden implementarse con tendencias paralela (registros de corrimiento) o serie (sumas o restas sucesivas, con predicción de acarreo).

Si se desea calcular el grado de pertenencia μ, ante un valor de entrada x en el intervalo [a, b] de una función triangular de pertenencia, se aplica la **ecuación 3.1**.

$$\mu(x) = s(x-a),$$

donde s es la pendiente y se calcula por medio de la relación mostrada por la **ecuación 3.2**:

$$s = \frac{\mu_{máximo}}{(b-a)}$$

lo anterior, explica que para forzar mayor exactitud en los cálculos es necesario encontrar la pendiente y resolver de la siguiente forma:

$$(\frac{\mu_{máximo}}{(b-a)})(x-a), \tag{3.8}$$

así, para que el truncamiento sea mínimo se propone realizar

$$\frac{(\mu_{máximo})(x-a)}{(b-a)}, \tag{3.9}$$

lo que sugiere la implementación en hardware de un multiplicador y de un divisor. Para corroborar lo anterior se diseñó una unidad fuzificadora como la de la **figura 3.23**, bajo la *estrategia 1*. El divisor en hardware está construido con un algoritmo de optimización que a su vez se utiliza nuevamente en la interfaz de defuzificación[11].

El código que modela esta solución se encuentra incluido en el **CD** de la tesis bajo el nombre **`A_Fríoex.v`**.

La **figura 3.26** muestra los resultados de simulación, que comparados con los obtenidos en la **figura 3.24** trabajando sobre la misma unidad individual, son más cercanos al valor real reduciendo de 5.88% a 0,39% el error; sin embargo el número de CLBs comprometidos crece de *13* hasta *258*.

Como se mencionó con anterioridad, es posible elegir un rango máximo que sea un número que dividido entre pendientes comunes resulte un entero, lo que evitaría errores en la exactitud, manteniendo la linealidad de las unidades fuzificadoras diseñadas.

[11] *Los algoritmos que modelan divisores se comentan más detalladamente en la parte concerniente al diseño de la interfaz de defuzificación, donde su utilización no es optativa, sino obligatoria.*

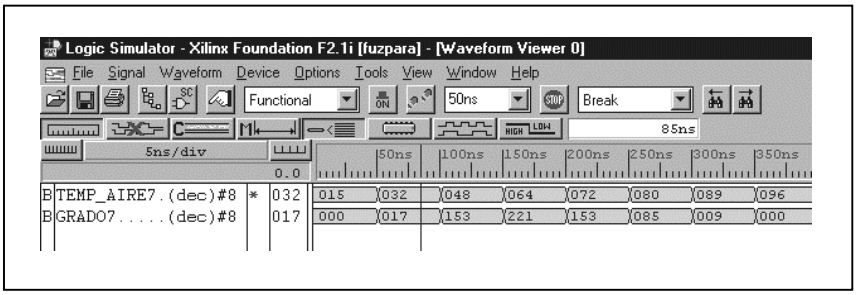


Figura 3.26: **Resultados de simulación para el fuzificador *Frío* bajo la *estrategia1*, mejorando la exactitud.**

3.2.5 Valores Signados

Para adicionar mayor generalidad a los diseños propuestos, es importante considerar la utilización de la aritmética signada. No es necesario cambiar rotundamente lo anteriormente expuesto, sólo basta con realizar el análisis comparativo sobre un bit que representará el signo. Un *0* para valores positivos y *1*, para los negativos.

Suponiendo un controlador como el diseñado en esta tesis, se tienen 8 bits del valor más 1 del signo, creando un formato de 9 bits. Considérese el bit más significativo como signo: $110110010_2 = -178_{10}$.

Las funciones de pertenencia pueden tener rangos positivos y/o negativos; sin embargo, el cálculo del grado de pertenencia siempre será positivo, por lo que es posible afirmar que la estrategia de cálculo se aplica de manera indistinta. El signo únicamente se utiliza como un distintivo que puede trabajarse por separado.

El algoritmo de solución más simple, consiste en cuestionar la existencia del signo:

If (signo==0) resolver parte positiva
else
resolver parte negativa

3.3 Aproximación al Diseño de la Interfaz de Inferencia

La interfaz de inferencia, es el bloque que realiza la evaluación de las reglas difusas en relación a los grados de pertenencia provenientes de la unidad de fuzificación. El marco teórico se expuso en el **Capítulo 1**, por lo que en esta sección, el análisis está dirigido a la realización hardware de las reglas.

Esencialmente una regla de inferencia difusa asume la forma referida por la **ecuación 1.2**:

Si x_1 **Es** A_1 **Y** x_2 **Es** A_2 **Y** . . . x_n **Es** A_n **Entonces** z **Es** B.

Para el controlador de irrigación modelado, se asumieron *15* reglas definidas en la **tabla 3.1**, las cuales adaptándolas a la forma anterior, se caracterizan de la siguiente forma:

"Si *Temp_Aire* **Es** *Frío* **Y** *Hum_Tierra* **Es** *Seca* **Entonces** *Tiempo* **Es** *Largo*"

Con la finalidad de simplificar la evaluación y verificar la dependencia de los consecuentes, es posible plantear una solución de referencia matricial como la expuesta en la **tabla 3.7**. Así, se suponen las variables *Temp_Aire* y *Hum_Tierra* en un arreglo de tal forma que las reglas posean una coordenada de ubicación. Por ejemplo, para la regla de inferencia anterior se escribe:

Regla (1,2), Largo.

Tabla 3.7: **Reglas de inferencia difusa para el controlador de irrigación, en esquema matricial.**

Tiempo de Irrigación		*Temperatura del Aire*				
		Muy Frío(0)	Frío(1)	Normal(2)	Templado(3)	Caliente(4)
Humedad de la Tierra	Muy Húmeda(0)	Corto	Corto	Corto	Corto	Corto
	Húmeda(1)	Corto	Medio	Medio	Medio	Medio
	Seca(2)	Largo	Largo	Largo	Largo	Largo

De manera natural, una regla de inferencia **If** - **Then**, se resuelve también bajo mecanismos del mismo tipo; es decir, utilizando **If** y **Then** en el HDL elegido. Modelando el método de *Mamdani* (MAX – MIN), expuesto en el **Capítulo 1**, resulta más simple realizar la evaluación.

Las reglas MIN que repiten el consecuente deberán aplicar un operador máximo (MAX) para entregar un único valor hacia la etapa de defuzificación. Lo anterior indica

que en el diseño existen 3 bloques MAX debido a que se tienen 3 consecuentes de evaluar las premisas (*Corto, Medio, Largo*). Se puede generalizar con n bloques MIN y m bloques MAX, donde n es el número de reglas y m es el número de consecuentes.

3.3.1 Interfaz de Inferencia con Topología Serie

De manera optativa, se plantea una interfaz de inferencia con topología serie. La característica primordial consiste en respetar la utilización de un solo bloque MIN y otro MAX, para evaluar todas las reglas.

Para este tipo de aproximaciones se requiere el uso de módulos de memoria auxiliares para retener los datos debido a que el resultado de la evaluación de mínimos culmina con otra evaluación de máximos, y de esta forma entregar los consecuentes que deberá procesar la interfaz de defuzificación.

A continuación se lista una parte significativa del código, en Verilog, que modela la interfaz de inferencia serie para el controlador propuesto. Nótese que los bloques MIN y MAX han sido implementados como funciones; la variable A representa la *Temperatura del Aire* con índice i; de la misma manera, B representa la *Humedad de la Tierra* con un índice j. El código completo está incluido en el **CD**, dentro de la carpeta *Inferencia Serie* bajo el nombre **`Inf_Serie.v`**.

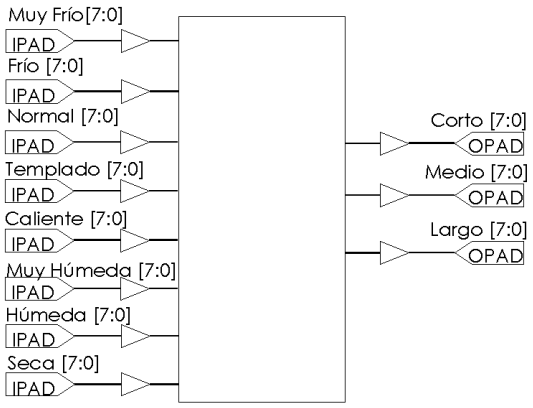
```
// Parte del código para implementar una interfaz de inferencia serie.
// Codificada en Verilog, 15 reglas hacia tres consecuentes, Inf_Serie.v

cont= 0;
for (i=0; i<5; i=i+1)                       // 5 funciones de pertenencia TEMP_AIRE
begin
  for (j=0; j<3; j=j+1)                     // 3 funciones de pertenencia HUM_TIERRA
  begin
  MEM [cont] = min (A[i], B[j]);            // Función MIN
  cont=cont+1;                              // Incrementa índice
  end
end

corto=maxA (MEM[0], MEM[1], MEM[3], MEM[6], MEM[9], MEM[12]);      // Consecuentes hacia
medio=maxB (MEM[4], MEM[7], MEM[10], MEM[13]);                     // la interfaz de
largo=maxC (MEM[2], MEM[5], MEM[8], MEM[11], MEM[14];              // Defuzificación
```

Los resultados comparativos de la síntesis en Verilog para esta interfaz serie están anotados en la **tabla 3.8**. Los resultados de la simulación son los mismos obtenidos mediante el tratamiento paralelo, los cuales son tratados a detalle posteriormente.

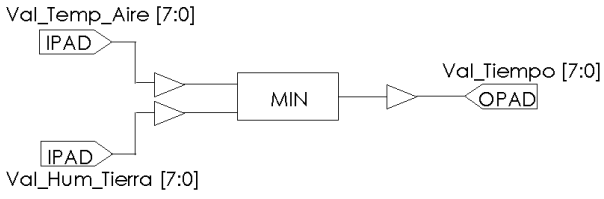
Tabla 3.8: **Resultados de síntesis de la interfaz de inferencia serie.**

Módulo	CLBs utilizados
	Verilog: 78 VHDL: 82

3.3.2 Interfaz de Inferencia con Topología Paralela

Los bloques MIN siempre serán del tamaño del número de variables de entrada al controlador, en el caso particular de este trabajo se tienen dos entradas. La aproximación abstracta al bloque MIN se muestra en la **tabla 3.9**.

Tabla 3.9: **Comparación entre resultados de síntesis del bloque básico MIN.**

Módulo	**Con Latches** CLBs utilizados	**Sin Latches** CLBs utilizados
	Verilog: 5 VHDL: 5	Verilog: 5 VHDL: 5

El código resolutivo es muy simple. Tanto para Verilog como para VHDL, se distinguen dos formas distintas de codificar: La primera codificación, en ambos HDLs, se basa en asincronía proponiendo un modelo secuencial con latches. La segunda opción, no genera latches, por lo que es puramente combinatorio. Los códigos parciales se listan a continuación; los completos están disponibles en la carpeta *Inferencia Paralela* del **CD**, con los nombres **`MINL.v`**, **`MINL.vhd`** para los bloques con latches y **`MIN.v`**, **`MIN.vhd`**, para los que no incluyen latches. La variable A representa la *Temperatura del Aire*, la variable B la *Humedad de la Tierra* y el *Consecuente* caracteriza al *Tiempo de Irrigación*.

```
// Bloque MIN para la Inferencia Difusa
// con Latches, Verilog, MINL.v

always @(A or B)
 begin:Fuzzy_Rule
 if(A<B)        //Si Temp_Aire < Hum_Tierra
  consecuente=A;
 else
  consecuente=B;
 end
```

```
// Bloque MIN para la Inferencia Difusa
// sin Latches, Verilog, MIN.v

assign consecuente = (A<B) ? A : B;
```

```
-- Bloque MIN para la Inferencia Difusa
-- con Latches, VHDL, MINL.vhd

process (A,B)
begin           --Si Temp_Aire < Hum_Tierra
 if (A<B) then
  consecuente <= A;
 else
  consecuente <= B;
 end if;
end process;
```

```
-- Bloque MIN para la Inferencia Difusa
-- con sin Latches, VHDL, MIN.vhd

consecuente <= A when (A<B) else B;
```

El número de entradas de los bloques MAX depende del número de reglas que compartan el mismo consecuente; por ejemplo, el consecuente *Tiempo Corto* (ver **tabla 3.7**)es aplicado por seis reglas, por lo que será el número de entradas al bloque. El análisis de codificación sugerido para los bloques MIN se utiliza para los MAX con la obviedad de la función contraria, por lo que sólo es necesario modificar la comparación con `if (A > B)` en la línea correspondiente.

Aunque el número de entradas varíe en cualquier tipo de bloque operador, se resuelve conectando bloques de dos entradas. La **figura 3.27**, es una abstracción de la idea anterior. Las referencias entre paréntesis son las coordenadas de la regla aplicada, de acuerdo a la **tabla 3.7**.

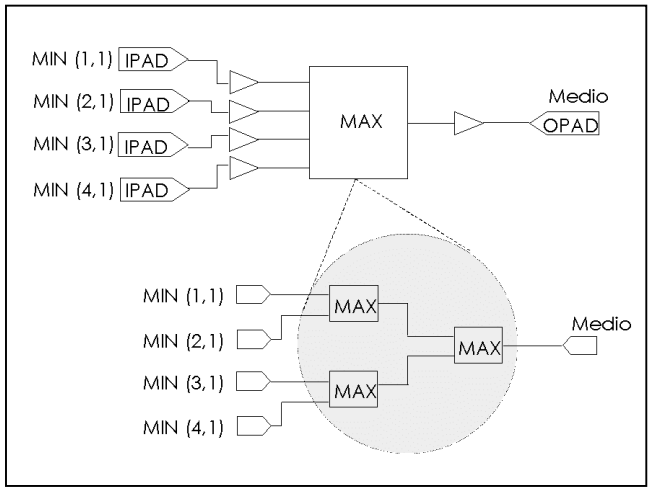
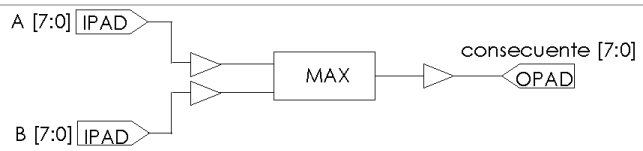


Figura 3.27: **Bloque MAX de cuatro entradas, implementado con bloques básicos MAX de dos entradas.**

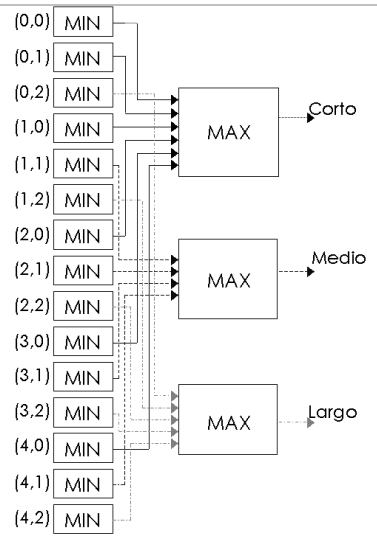
Los resultados de sintetización para bloques individuales del operador MAX básico, están anotados en la **tabla 3.10**. Los códigos **MAXL.v**, **MAXL.vhd**, **MAX.v** y **MAX.vhd**, se encuentran disponibles en la carpeta *Inferencia Paralela* del **CD**. Los dos primeros modelan aproximaciones con latches.

Tabla 3.10: **Comparación entre resultados de síntesis del bloque MAX de dos entradas.**

Módulo	**Con Latches** CLBs utilizados	**Sin Latches** CLBs utilizados
	Verilog: 5 VHDL: 5	Verilog: 5 VHDL: 5

La red de inferencia para la interfaz queda esquematizada abstractamente como lo muestra el bloque en la **tabla 3.11**. En la misma tabla se incluyen los resultados comparativos de sintetización. Nótese que los resultados de la síntesis no son necesariamente proporcionales a los obtenidos en los bloques individuales, debido a que cuando se eliminan redundancias se optimiza la arquitectura.

Tabla 3.11: **Síntesis comparativa de la inferencia con procesamiento paralelo.**

Módulo	**Con Latches** CLBs utilizados	**Sin Latches** CLBs utilizados
	Verilog: 108 VHDL: 112	Verilog: 106 VHDL: 110

Para simular la interfaz diseñada, los valores de entrada están listados en la **tabla 3.12**. Para *Temperatura del Aire = 32*, se obtienen las entradas a la interfaz: *Muy Frío = 144, Frío = 16, Normal = 0, Templado = 0* y *Caliente = 0*; y para *Humedad de la Tierra = 64*, se obtienen: *Muy Húmeda = 0, Húmeda = 96* y *Seca = 144;* se evalúan las reglas *(0, 1), (0, 2), (1, 1)*

y (1, 2), cuyos resultados sobre el *Tiempo* de *Irrigación* son: *corto = 96, Medio =16* y *Largo =144*. La **figura 3.28**, muestra los resultados.

Tabla 3.12: **Valores asumidos para la simulación de la interfaz de inferencia, del controlador de irrigación.**

Temp_Aire	Hum_Tierra	Pertenencia	Reglas Evaluadas y Consecuentes
32	64	Muy Frío = 144, Frió = 16 Húmeda = 96, Seca = 144	(0, 1), (0, 2), (1, 1), (1, 2) Corto = 96 Medio =16 Largo =144
64	48	Frío = 208, Normal = 16 Húmeda = 32, Seca = 208	(1, 1), (1, 2), (2, 1), (2, 2) Corto = 0 Medio = 32 Largo = 208
128	192	Normal = 208 Muy Húmeda = 16, Húmeda = 112	(2, 0), (2, 1) Corto = 16 Medio = 112 Largo = 0
160	96	Normal = 80, Templado = 80 Húmeda = 224, Seca = 16	(2, 1), (2, 2), (3, 1), (3, 2) Corto = 0 Medio = 80 Largo = 16
15	192	Muy Frío = 255 Muy Húmeda = 16, Húmeda = 112	(0, 0), (0,1) Corto = 112 Medio = 0 Largo = 0
224	7	Caliente = 255 Seca = 255	(4, 2) Corto = 0 Medio = 0 Largo = 255

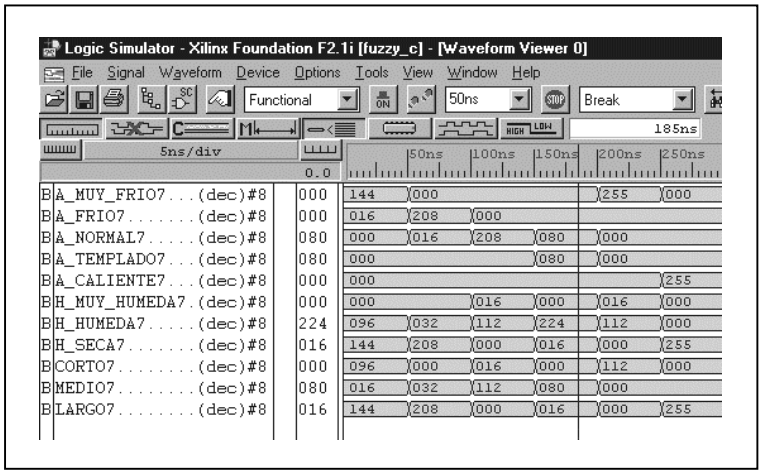


Figura 3.28: **Resultados de simulación para la interfaz de inferencia. Obsérvese que coinciden con los propuestos en la tabla 3.12**

3.4 Aproximación al Diseño de la Interfaz de Defuzificación

La interfaz de defuzificación calcula el valor(es) real(es) que afectarán al sistema bajo control. El trabajo de esta última unidad está supeditado por los datos provenientes de la interfaz de inferencia.

Es necesario considerar el total de conjuntos de salida. En el caso del controlador de irrigación diseñado como ejemplo de la metodología, las funciones de pertenencia se aprecian en la **figura 3.4**. La estrategia que se sigue para el diseño en hardware de esta interfaz es el *Defuzificador del Promedio de Centros o Centro de Sumas*, comentado formalmente en el **Capítulo 1**. La **ecuación 1.7** del mismo capítulo representa la relación matemática:

$$Z^* = \frac{\sum_{i=1}^{n} \mu_{Bi}(\bar{Z})\omega_1}{\sum_{i=1}^{n} \mu_{Bi}(\bar{Z})},$$

donde Z* es el valor escalar de la variable de salida, $\mu_B(Z^*)$ es el grado concluido de la evaluación de reglas y ω es el peso de la función de salida.

Para fines particulares, el número de funciones de pertenencia de salida de la variable *Tiempo*, es tres: *Corto, Medio* y *Largo*. Los pesos referidos son *1, 30* y *60*, respectivamente, en relación al diseño propuesto.

El algoritmo resolutivo implica que primeramente se multiplican los grados concluidos (provenientes de la interfaz de inferencia) por su respectivo peso. Los resultados de las multiplicaciones se suman, obteniéndose el dividendo de la relación matemática. Posteriormente se suman los grados concluidos para integrar el divisor.

3.4.1 Aproximación Serie

La topología serie del defuzificador se realiza utilizando como elementos básicos, una unidad de adición, una multiplicadora y una divisora, las cuales serán accedidas en forma secuencial por los datos entrantes provenientes de la inferencia. La abstracción del modelado del hardware se muestra en la **figura 3.29**. Debido a que las

entradas acceden a la única unidad multiplicativa, es imprescindible almacenar en memorias auxiliares los datos previos y actuales, con la intención de reutilizarlos en las fases subsecuentes. Un multiplexor colocado a la entrada del sumador permite seleccionar la suma de los productos(dividendo) o la suma de las entradas (divisor). Finalmente los datos almacenados de forma temporal, se dividen para entregar el valor real respetando escalares sin signo y de punto fijo.

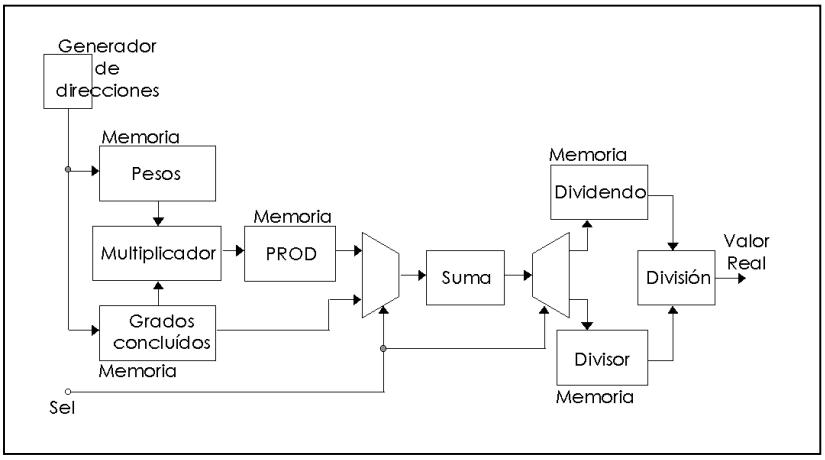


Figura 3.29: **Abstracción hardware de la interfaz defuzificadora serie.**

Una aproximación de la codificación en Verilog, se lista a continuación.

```
// Parte significativa del defuzificador serie.
// Codificación en Verilog, def_Serie.v
w[0] = 1;                                   // Declaración de Pesos
w[1] = 30;
w[2] = 60;

for (i=0; i<3; i=i+1)                       // El índice es el número de singletons
begin
  PROD [i] = por (A[i], w[i]);              // Arreglo para el producto
end

dividendo = suma (PROD[0], PROD[1], PROD[2]);
divisor = suma (B[0], B[1], B[2];
defuz = entre (dividendo, divisor)          // Obtención del valor real
```

Los pesos (*w*), específicos para la variable de salida *Tiempo de Irrigación* del controlador diseñado, se definen en una memoria de datos denominada en el código como w. Los grados concluidos entrantes se almacenan en otra memoria llamada A. Posteriormente se ejecuta una multiplicación secuencial, guardando los resultados de cada operación en otra memoria auxiliar llamada PROD. Finalmente se ejercen las sumas[12] y se ejecuta la división. Nótese que todas las operaciones aritméticas están

[12] *Se trata de la misma función suma, la cual es accedida en diferentes tiempos. No es necesario incluir el multiplexor en el código.*

implementadas como funciones (por, suma, entre), las cuales se definen en los códigos completos listados dentro de la carpeta *Defuzificadores Serie* en el **CD**, bajo los nombres **def_Serie1.v** y **def_Serie2.v**. Estas versiones se proporcionan codificadas únicamente en Verilog.

En definitiva, la función que realiza la división (entre) es la operación aritmética que consume la mayor cantidad de CLBs. La construcción de unidades divisoras en hardware que pueden tomar connotaciones síncronas (*restas sucesivas*) o asíncronas (*corrimientos constantes*), determinando en gran medida el desempeño y el espacio físico comprometido, tal y como se explica en el siguiente apartado. Para el controlador de irrigación, se modela un dividendo de *16* bits y un divisor de *9*, para obtener un cociente de *8* bits. **def_Serie1.v** es una interfaz defuzificadora de procesamiento serie que implementa una división mediante restas sucesivas, **def_Serie2.v**, es la misma interfaz pero con una división de corrimientos constantes.

Si los requerimientos de un diseño consideraran más de una salida real, únicamente sería necesario colocar tantas unidades defuzificadoras como salidas se requiera. La adaptación de código recae en que la multiplicación del grado concluido de la inferencia por el peso respectivo, son de carácter específico para cada unidad.

Los resultados de sintetización se anotan en la **tabla 3.13**. Los resultados de simulación son los mismos que se obtienen en la topología paralela, por lo mismo se analiza en el apartado correspondiente.

Tabla 3.13: **Comparación entre resultados de síntesis de la interfaz defuzificadora serie con ambos tipos de módulos de división.**

Módulo	**División con Restas Sucesivas** CLBs utilizados	**División con Corrimientos Constantes** CLBs Utilizados
def_Serie	Verilog: 118	Verilog: 376

3.4.2 Aproximación Paralela

Esta topología permite un análisis más conciso del trabajo de la interfaz de defuzificación, debido a que es posible dividir el proceso en bloques individuales. La abstracción del modelo en hardware se exhibe en la **figura 3.30**. Las entradas a la unidad de división son la suma de los grados provenientes de la interfaz de inferencia

(*Suma1*) y la suma de los productos de los mismos grados con sus respectivos pesos (*Suma2*).

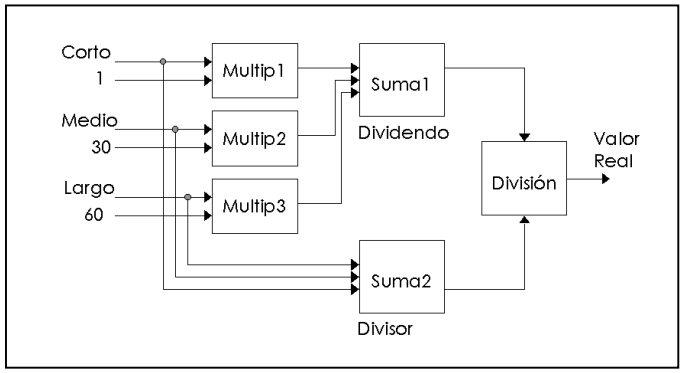


Figura 3.30: **Interfaz defuzificadora con topología paralela.**

El código que realiza un multiplicador por hardware con latches se lista a continuación. La referencia completa está dentro de la carpeta *Defuzificadores Paralelos* en el **CD** anexo, con los nombres: **`Mult_L.v`** y **`Mult_L.vhd`** para módulos con latches y **`Mult.v`** y **`Mult.vhd`**, para diseños sin latches. La variable A es el grado (*Corto, Medio* o *Largo*) proveniente de la etapa de inferencia. La variable B es el peso respectivo (*1, 30* ó *60*).

```
// Código que modela un multiplicador de 8*8, con latches, Verilog, Mult_L.v

module mult_L (dividendo, A, B);
input [7:0] A, B;                    // Entradas de 8 bits
output [15:0] prod;                  // Resultado en 16 bits
reg [15:0] prod;
 always @ (A or B)                   // Espera por un cambio en A o B (Asincronía)
 begin
 prod = A*B;                         // Operador de multiplicación
 end
end module
```

```
-- Código que modela un multiplicador de 8*8, con latches, VHDL, Mult_L.vhd
Library IEEE;
Use IEEE.Std_Logic_1164.All;
Use IEEE.Std_Logic_Arith.All;

Entity Mult_L Is
Port(A : In._aire Std_Logic_Vector(7 downto 0);
B : In  Std_Logic_Vector(7 downto 0);                -- Entradas de 8 bits
prod : Out Std_Logic_Vector(15 downto 0));           -- Resultado de 16 bits
End Entity Mult_L;

Architecture Arch_multp1 Of Mult_L Is
Begin
Process(A, B)                                        -- Espera por un cambio en A o B
  prod <= Conv_Std_Logic_Vector(A*B)),16);           -- Operador de Multiplicación
End Process;
End Arch_multp1
```

El código que realiza un sumador por hardware, es similar al del multiplicador, con la salvedad del operador. En el caso del defuzificador diseñado (**figura 3.30**), *Dividendo* realiza una suma de tres números de tamaño diferente, entregando *16* bits que se integran a la entrada correspondiente del módulo *división*. *Divisor* por su parte resuelve sobre una salida de *9* bits que también se integrará al mismo módulo que dividirá para obtener el *valor real*. **Sum_L.v** y **Sum_L.vhd**, con las mismas variantes que el multiplicador anterior, se listan enseguida adaptados al módulo *Divisor*. Opcionalmente los mismos códigos, más las alternativas **Sum.v** y **Sum.vhd** (sumadores sin latches), están disponibles en el apartado pertinente dentro del **CD** suplementario. *A* representa el valor del grado calculado para *Corto*, *B* el correspondiente para *Medio* y *C* para *Largo*.

```
// Sumador con Latches, Sum_L.v, Verilog

module Sum_L (divisor, A, B, C);
input [7:0] A, B, C;                          // Tres entradas de 8 bits cada una
output [8:0] divisor;                         // Resultado en 9 bits
reg [8:0] divisor;
 always @ (A or B or C)                       // Espera por un cambio en A, B o C
 begin
 divisor = A + B + C;                         // Operador Suma
 end
end module
```

```
-- Sumador con Latches, Sum_L.vhd, VHDL

Library IEEE;
Use IEEE.Std_Logic_1164.All;
Use IEEE.Std_Logic_Arith.All;

Entity Sum_L Is
Port(A, B, C : In._aire Std_Logic_Vector(7 downto 0);    -- Entradas de 8 bits
divisor : Out Std_Logic_Vector(8 downto 0));             -- Salida de 9 bits
End Entity Sum_L;

Architecture Arch_suma1 Of Sum_L Is
Begin
Process(A, B, C)                                         -- Espera por cambio en A, b o C
  dividendo <= Conv_Std_Logic_Vector(A + B + C)),9);     -- Operador Suma
End Process;
End Arch_auma1
```

Para el bloque que realiza la división, se propone una unidad de división con restas sucesivas (síncrono) y otra con corrimientos constantes (asíncrono). La primera repercute en un ahorro sustancial de hardware sacrificando desempeño. La instrucción While en Verilog (similar al Case –When, state en VHDL) utilizada, se controla obligatoriamente por una señal de reloj, causando que la unidad consuma tantos ciclos de reloj como restas sucesivas realice, procurando además el uso de una señal de *handshake* que indique la finalización del proceso de división. A continuación, se

lista una parte significativa del módulo **Divsión1.v**. El código completo está contenido en el **CD** dentro de la carpeta *Defuzificadores Paralelos*.

```
// Parte significativa de un Divisor de restas sucesivas, Divsión1.v
// Codificación en Verilog
always
begin : divide @(posedge reloj)                  // Diseño Síncrono
cont = 0;
hand=0;
minuendo = dividendo;
sustraendo = divisor;
  if (!minuendo || !sustraendo)
  begin
  resul = 0;                                     // Resultado = 0
  hand=1;
  disable divide;
  end
   else if (minuendo == sustraendo)
   begin
   resul = 1;                                    // Resultado = 1
   hand=1;
   disable divide;
   end
    else
    begin
    while (minuendo >= sustraendo)               // Realiza restas sucesivas
       begin: repite @(posedge reloj)
       resta = minuendo - sustraendo;
       cont = cont + 1;
       minuendo = resta;
       end
       resul = cont;
    hand=1;
    disable divide;
end
end
```

Los resultados de simulación, para este algoritmo se muestran en la **figura 3.31**. Se requieren tantos pulsos de reloj como restas sean necesarias.

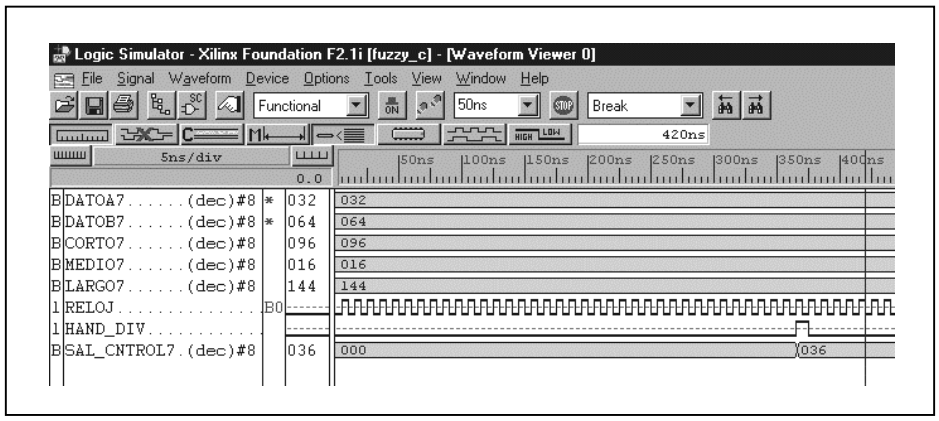


Figura 3.31: **Resultados de simulación para la interfaz defuzificadora paralela con un módulo de división implementado mediante un algoritmo de restas sucesivas.**

Los valores simulados son los consecuentes ejemplificados como salidas de la interfaz de inferencia en la **figura 3.28**. Observando el diagrama de tiempos, se aprecia que cuando *Corto = 96*, *Medio = 16*, y *Largo = 64* (valores resultantes de haber introducido DATO_A *Temp_Aire= 32* y DATO_ B *Hum_Tierra = 64*, según la **tabla 3.12**), tendrán que transcurrir *36* ciclos de reloj.

La segunda opción permite una ejecución asíncrona, aumentando considerablemente el hardware y el desempeño de la unidad de división. Este modelado ejecuta únicamente 16 iteraciones reflejadas es 16 corrimientos, que son las necesarios para encontrar el cociente del mismo número de bits. La gran ventaja es que las 16 iteraciones pueden realizarse dentro de un solo ciclo de reloj, lo suficientemente grande para garantizar que la operación fue resuelta en su totalidad, por lo que no es necesario el handshake dentro del módulo, pero sí lo será para indicar que el proceso difuso total ha sido completado. El código completo se referencia con el nombre de **`Divsión2.v`**, localizado en la carpeta pertinente.

```
// Parte significativa del algoritmo que realiza una división con corrimientos
// constantes, Divsión2.v
// Codificación en Verilog
always @(dividendo or divisor)                          // Diseño asíncrono
begin  :divide
       integer i;
       reg_cor={17'b0, dividendo};
       comp_divisor={9'b0, divisor};                    // Divisor de 9 bits
       for (i=0; i<16; i=i+1)                           // 16 Corrimientos
              begin
              reg_cor= reg_cor<<1;
              reg_cor[32:16]= reg_cor[32:16]-comp_divisor;
              guarda[15-i]= !reg_cor[32];
              if (reg_cor[32]) reg_cor[32:16]=reg_cor[32:16]+comp_divisor;
              resul=guarda[7:0];                        // Resultado en 8 bits
              end
disable divide;
end
```

La **figura 3.32** muestra los resultados de simulación, considerando los valores entregados por la interfaz de inferencia en la **figura 3.28**. Los valores introducidos son los mismos que muestra la **tabla 3.12**.

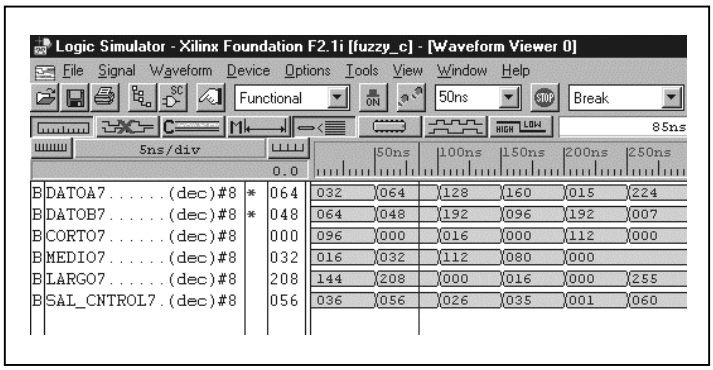


Figura 3.31: **Resultados de simulación para la interfaz defuzificadora con un módulo de división implementado mediante un algoritmo de corrimientos constantes.**

Finalmente, la **tabla 3.14** muestra los resultados de sintetización, considerando versiones paralelas del defuzificador con la unidad de división de restas sucesivas y su contraparte de corrimientos constantes. Para los resultados de síntesis del código en VHDL, se consideraron modelos híbridos con las unidades de división codificadas en Verilog.

Tabla 3.14: **Comparación entre resultados de síntesis de la interfaz defuzificadora paralela con ambos tipos de módulos de división.**

Módulo	**División Restas Sucesivas** CLBs utilizados	**División Corrimientos Constantes** CLBs Utilizados
Def_pa	Verilog: 130 VHDL: 138	Verilog: 410 VHDL: 438

Resumen del Capítulo

En este capítulo se explicó la metodología seguida para resolver el problema de diseño e implementación del controlador. Se propusieron arquitecturas de procesamiento serie y otras de procesamiento paralelo, que trabajan sobre una división modular del controlador en interfaces individuales (Fuzificador, Inferencia y Defuzificador).

En el capítulo subsiguiente se integrarán las interfaces para conformar el controlador completo y realizar las pruebas que validen los logros estimados.

Capítulo 4

Integración de las Interfaces

Contenido del Capítulo

A partir del análisis y diseño individual de cada interfaz, así como de su gradual optimización, fue posible estimar la integración de los tres módulos para conformar el controlador de irrigación propuesto como ejemplo de aplicación.

Este capítulo presenta los resultados de simulación y síntesis lógica, obtenidos sobre el controlador terminal, que produjeron las conclusiones más representativas de esta experiencia.

Se procedió a la implementación de una de las arquitecturas descritas, configurando una tarjeta de desarrollo para un dispositivo con recursos limitados. El proyecto está incluido en su totalidad, en el ***CD*** *suplementario a este trabajo escrito, dentro de la carpeta FLC.*

4.1 Integración del FLC Digital

Para contribuir a las conclusiones de este trabajo, se realizaron cuatro derivaciones en consecuencia a la naturaleza del procesamiento. La primera integración está constituida por las tres interfaces serie con la división por restas sucesivas en la defuzificación; la segunda está conformada por el mismo modelo serie, con la excepción de que la unidad de división está descrita a través de corrimientos constantes. Las otras dos integraciones son las contrapartes paralelas, respetando el cambio en las características de la división defuzificadora. Estas últimas fueron las contribuciones principales en este trabajo, debido a que se buscó la generalidad del algoritmo implementado, cuestión poco analizable en las interfaces de procesamiento serie.

Para realizar una comparación entre las soluciones originadas, se consideraron cinco aspectos básicos: cantidad de CLBs (espacio físico y granularidad)[1], retardos de propagación, ciclos de reloj consumidos, desempeño y versatilidad. Los parámetros se reducen cuando se comprueba que cualquier solución aporta un desempeño aceptable en términos de exactitud y precisión. Los retardos de propagación son predecibles y estimados desde la etapa de síntesis lógica en dependencia al FPGA elegido y que por lo general son del orden de los nanosegundos, lo cual se traduce en una alta respuesta hacia medios mecánicos.

Por lo anteriormente expuesto, el marco comparativo se delimita por el espacio físico, los ciclos de reloj y la facilidad de adaptar diseños diferentes tomando como referencia un modelo extensivo.

La **tabla 4.1**, muestra los valores introducidos para simular la respuesta de los controladores diseñados. Los valores fueron seleccionados de manera intencional para obligar respuestas simples y complejas. Las primeras dos columnas representan los valores reales introducidos al controlador, en numeración decimal. La tercera columna hace mención a la fuzificación de las variables, considerando la función de pertenencia y el grado calculado que se entrega a la interfaz de inferencia. La columna siguiente, resuelve la evaluación de reglas en su etapa MIN que se conectará hacia los respectivos bloques MAX. Se decidió hacerlo de esta manera con la intención de hacer visible la cantidad de reglas que se evalúan, en consecuencia a diferentes valores de entrada. Las dos últimas columnas comparan el valor real esperado y el valor real obtenido a través del FLC diseñado. Las cuatro integraciones del controlador, mantienen la misma exactitud y precisión, por lo que los resultados no varían.

[1] *La correlación entre los retardos de propagación y el desempeño, depende de la topología planteada y de la cantidad de CLBs.*

Tabla 4.1: **Valores asumidos para la simulación del FLC de irrigación diseñado.**

Temp Aire	Hum Tierra	Pertenencia	Reglas Evaluadas y consecuente	Valor Real	Valor Real por HW
15	205	Muy Frío = 255 Muy Húmeda = 120, Húmeda = 60	(0, 1) Corto = 60 (0, 0) Corto = 120 Corto = 120	1.0	1
32	205	Muy Frió = 144, Frío = 16 Muy Húmeda = 120, Húmeda = 60	(0, 1) Corto = 60 (0, 0) Corto = 120 (1, 1) Medio = 16 (1, 0) Corto = 16 Corto = 120 Medio = 16	4.41	4
64	205	Frío = 208, Normal = 16 Muy Húmeda = 120, Húmeda = 60	(1, 1) Medio = 60 (1, 0) Corto = 120 (2, 1) Medio = 16 (2, 0) Corto = 16 Corto = 120 Medio = 60	10.66	10
80	205	Frío = 80, Normal = 80 Muy Húmeda = 120, Húmeda = 60	(1, 1) Medio = 60 (1, 0) Corto = 80 (2, 1) Medio = 60 (2, 0) Corto = 80 Corto = 80 Medio = 60	13.42	13
128	192	Normal = 208 Muy Húmeda = 16, Húmeda = 112	(2, 0) Corto = 16 (2, 1) Medio = 112	26.37	26
60	128	Frío = 255 Húmeda = 255	(1, 1) Medio = 255	30.0	30
32	64	Muy Frío = 144, Frío = 16 Húmeda = 144, Seca = 96	(0, 1) Corto = 96 (0, 2) Largo = 144 (1, 1) Medio = 16 (1, 2) Largo = 16 Corto = 96 Medio = 16 Largo = 144	36.0	36
64	48	Frío = 208, Normal = 16 Húmeda = 32, Seca = 208	(1, 1) Medio = 32 (1, 2) Largo = 208 (2, 1) Medio = 16 (2, 2) Largo = 16 Medio = 32 Largo = 208	56.0	56
224	7	Caliente =255 Seca = 255	(4, 2) Largo = 255	60	60

Es importante mencionar que la adaptación del FLC a cualquier sistema de diligencia, requiere incluir señales de comunicación con las demás etapas ajenas al proceso difuso, como la adquisición de datos y el indicador de término de procesamiento. Estas señales no están contempladas en las aproximaciones analizadas, aunque de manera implícita, la unidad de división por restas sucesivas incluye un handshake que indica la finalización del proceso.

4.1.1 Integración del FLC con División Síncrona

La **figura 4.1**, representa la pantalla de simulación para el FLC paralelo con la unidad de división con algoritmo de restas sucesivas. En el **Anexo C** de este mismo documento, se imprime una aproximación al diagrama esquemático de esta arquitectura con el nombre **FUZZY_C1.SCH** (**figura C.1**).

Dadas las características de la defuzificación mediante este mecanismo (comentadas en el **capítulo 3**), únicamente se introdujeron algunos valores. El número de ciclos de reloj consumidos para que el *Handshake* sea validado, es el resultado de la división y por ende, el valor real que entrega el FLC. Tanto las interfaces de fuzificación como de inferencia están implementadas de manera asíncrona, por lo que en la figura sólo aparecen los ciclos de reloj consumidos por la división.

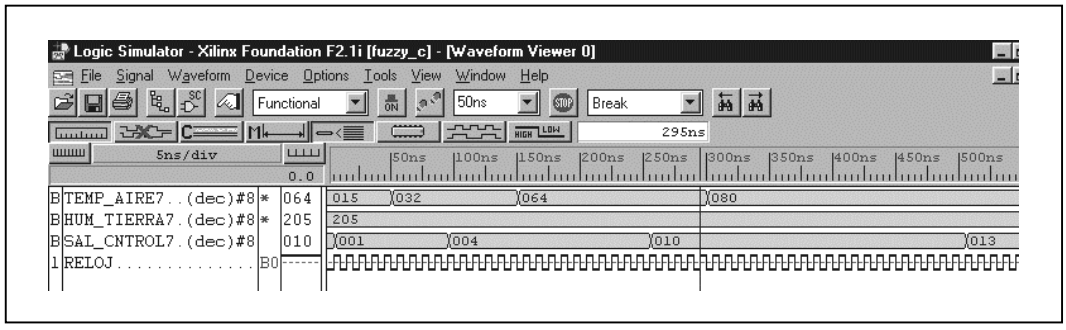


Figura 4.1: **Resultados de simulación para el FLC diseñado, con topología paralela y división mediante restas sucesivas en la defuzificación.**

Cuando la *Temperatura del Aire* tiene un valor de *32* y la *Humedad de la Tierra* es igual a *205*, la salida del controlador es *004*, después de *4* pulsos de reloj, tal y como se aprecia en la **figura 4.1**. Manteniendo constante el valor de la *Humedad de la Tierra* y con *Temperatura del Aire* igual a *80*, transcurren *13* ciclos de reloj para entregar el resultado *013*. Lo anterior es un indicativo que para este diseño en particular, los ciclos de reloj

consumidos pueden ir desde *1* hasta *60*, que es el valor máximo propuesto para la defuzificación en su *singleton* *Tiempo Largo*.

En el caso de las interfaces en una arquitectura con interfaces de procesamiento serie, es necesario sincronizarlas mediante el reloj del sistema, que como ya se comentó oportunamente, es del orden de los *16 KHz* en el caso de un control mecánico. En particular, se condicionan los ciclos totales de la interfaz fuzificadora, más los ciclos de la propia de inferencia, más los ciclos de la interfaz defuzificadora con su respectiva etapa de división.

4.1.2 Integración del FLC con División Asíncrona

Por naturaleza, de acuerdo a los diseños planteados, el procesamiento paralelo de las interfaces se realiza de manera asíncrona; sin embargo es factible sincronizar individualmente o resumir el grueso de todo el proceso, respetando que debe seleccionarse un pulso de reloj lo suficientemente grande que garantice la finalización de una etapa antes de pasar a la siguiente. En la **figura 4.2** se muestran los resultados de la simulación, indicados en la **tabla 4.1**, para un FLC con topología paralela y con una unidad de división con algoritmo de corrimientos constantes que se ejecutan de forma asíncrona.

En el **Anexo C**, se incluye la aproximación al diagrama esquemático de esta descripción bajo el nombre **FUZZY_C2.SCH** (**figura C.2**).

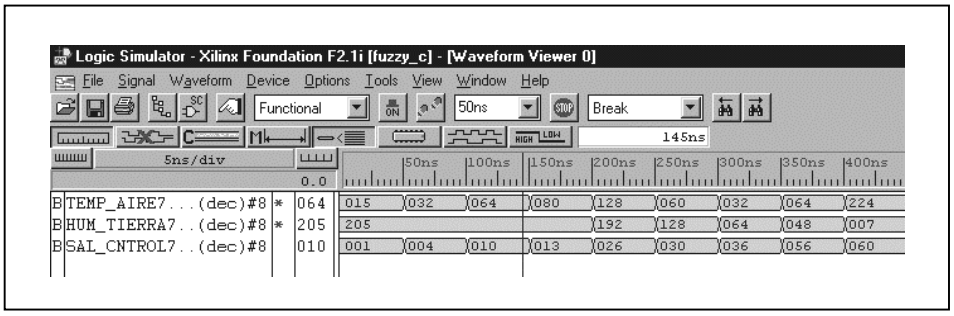


Figura 4.2: **Resultados de simulación para el FLC diseñado, con topología paralela y división mediante corrimientos constantes en la defuzificación.**

Respecto a la **figura 4.2** y comparando con lo expuesto en el ejemplo de la **figura 4.1**, para una *Temperatura del Aire* = *32* y *Humedad de la Tierra* = *205*, se obtiene una salida de *004*, sin necesidad de esperar por pulsos de sincronía.

La **figura 4.3**, muestra los mismos resultados de simulación de la **figura 4.2**, con la variante de que se incluye un desglose de las señales intermedias, donde se advierten los grados de pertenencia calculados por el fuzificador, así como los consecuentes entregados por la interfaz de inferencia. Los resultados coinciden con los mostrados por la **tabla 4.1**.

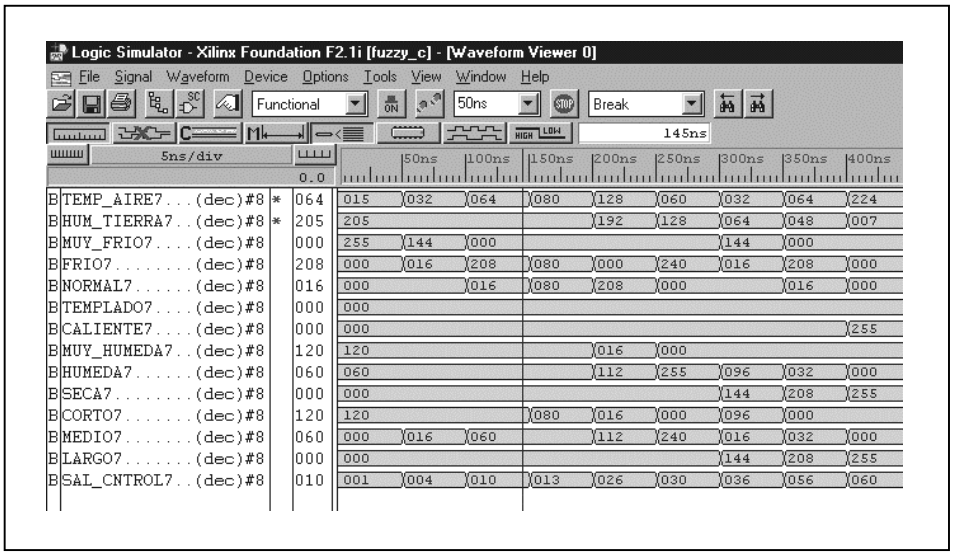


Figura 4.3: **Resultados de simulación para el FLC diseñado, considerando señales intermedias, Para la topología paralela con división mediante corrimientos constantes en el defuzificador.**

4.1.3 Resultados de Sintetización

En la **tabla 4.2**, se listan los resultados de sintetización de cada uno de los diseños realizados.

Tabla 4.2: **Resultados de sintetización para el FLC diseñado.**

Características del FLC	Codificación en HDL	Cantidad de CLBs	FPGA
Arquitectura Serie con División R.S.	Verilog	342	XC4010XL
	VHDL	356	XC4010XL
Arquitectura Serie con División C.C.	Verilog	446	XC4013XL
	VHDL	456	XC4013XL
Arquitectura Paralela con División R.S.	Verilog	331	XC4010XL
	VHDL	336	XC4010XL
Arquitectura Paralela con División C.C.	Verilog	535	XC4013XL
	VHDL	530	XC4013XL

R.S. – Restas Sucesivas, C.C .- Corrimientos Constantes.

Un dispositivo XC4010XL dispone de 400 CLBs, y su costo es de aproximadamente *90* dlls[2]. El XC4013XL contiene 576 CLBs y su costo aproximado es de *110* dlls. Ambos tipos de FPGAs mantienen una frecuencia de trabajo máxima de hasta *6 MHz* con retardos de *20 ns*, según las especificaciones del fabricante. La simulación *Post Ruteo* indica los siguientes valores que pueden asumirse como reales:

```
Minimum period: 168.440ns (Maximum frequency:   5.937MHz)
Maximum net delay:  19.469ns
```

Considerando el número de CLBs obtenidos y la aplicación del FLC, es posible optar por una solución más viable. La serie estándar *Spartan* de Xilinx, aporta menor frecuencia de trabajo a un coste más razonable; por ejemplo, en el caso de los FLC con unidad de división paralela, se opta por un FPGA XCS30 con 576 CLBs y cuyo costo aproximado es de *18* dlls. Los rangos de frecuencia Post Ruteo obtenidos son:

```
Minimum period: 242.989ns (Maximum frequency:   4.115MHz)
Maximum net delay:  23.193ns
```

4.1.3.1 Implementación

La implementación se realizó utilizando una tarjeta de desarrollo de distribución comercial (ver **figura 4.4**). La documentación de la tarjeta se incluye en el **CD**, dentro de la carpeta *Xess*.

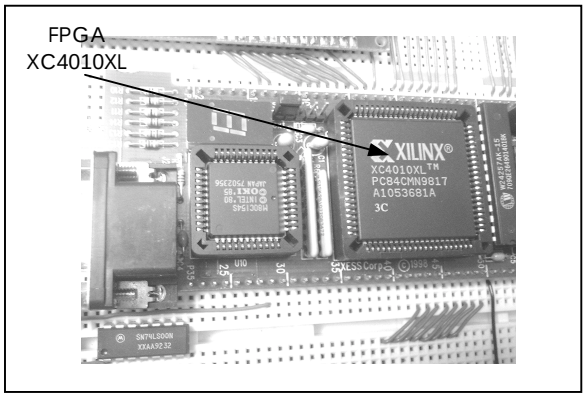


Figura 4.4: **Tarjeta de Desarrollo XESS XC4010XL.**

El FPGA que contiene la tarjeta es un XC4010XL de Xilinx con 400 CLBs, por lo tanto no fue posible comprobar físicamente todos los controladores simulados. Se resolvió implementar el FLC paralelo con división de restas sucesivas en el defuzificador,

[2] *Hasta el segundo semestre del año 2001.*

justificando que los modelos paralelos diseñados presentan mayor generalidad que los de procesamiento serie. Para simular las entradas del controlador en la parte física, se utilizaron primeramente dos *ADCs 0804* (convertidores Analógico – Digital) en configuración básica; posteriormente, dos dipswitches con 8 polos útiles cada uno.

Para desplegar los resultados entregados por el controlador y validar el diseño, se adecuó una salida binaria directa al exterior de 8 bits. Internamente en el FPGA, la misma salida binaria entra a un controlador para desplegar el mismo resultado en base decimal y en una pantalla de LCD. Debido a que la síntesis del código en Verilog que describe el funcionamiento del controlador utilizó 331 CLBs, únicamente se disponía de 69 para crear la interfaz con la pantalla de LCD. Después de optimizar el código inicial se logró una descripción que consumió 44 CLBs. El diseño completo del controlador con pantalla, utilizó 375 CLBs y se incluye en el **CD** de la tesis, dentro de la carpeta *FLC*.

La **figura 4.5** muestra la tarjeta y sus respectivas interfaces de entrada y salida, adaptadas al diseño del controlador. Los valores considerados en la **tabla 4.1** se comprobaron experimentalmente. El **Anexo B** incluye los resultados obtenidos en pantalla.

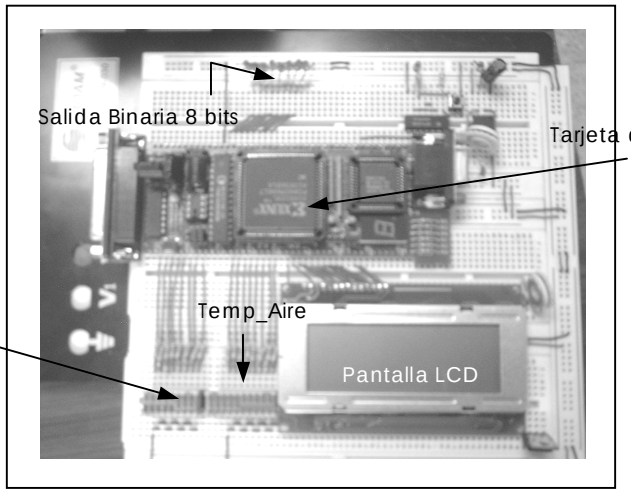


Figura 4.5: **Tarjeta con interfaces de entrada y de salida.**

Resumen del Capítulo

Este capítulo mostró la integración de las interfaces diseñadas en el **Capítulo 3**, para la implementación completa del controlador en el FPGA. Se realizaron pruebas experimentales con valores predefinidos y simulados, que resultaron satisfactorias, cumpliendo con los objetivos planteados de inicio en este trabajo. El **Capítulo 5**, presenta las conclusiones más importantes derivadas de este desarrollo.

Capítulo 5

Conclusiones y Trabajos a Futuro

Contenido del Capítulo

En este último capítulo, se incluyen las conclusiones finales generadas, así como una serie de propuestas que adjudican la continuación de los estudios siguiendo la misma línea de investigación, con el propósito de mejorar y aplicar los resultados aportados.

Los marcos teóricos de los FLCs y de las tecnologías programables, fueron parte primordial para llevar a buen término este desarrollo, transigiendo en que se considerará cumplido el objetivo general del trabajo de tesis.

Implícitamente, la experiencia realizada permitió generar material docente en la enseñanza de las tecnologías programables a nivel licenciatura y diplomados; también dará pie a artículos especializados y a la vez sirve de perfil para definir una posible línea para estudios de Doctorado en el área del diseño VLSI.

5.1 Conclusiones

En esta tesis se ha presentado una metodología de desarrollo para controladores digitales basados en lógica difusa, que se hace extensiva a cualquier aplicación práctica de sistemas de control no lineales. La utilización de herramientas VLSI CAD comerciales contribuye a la práctica más detallada de los alcances de la electrónica digital contemporánea al proponer la automatización del diseño.

El estudio realizado, tomando como base los algoritmos de cuantificación, fundamenta la viabilidad de la realización en hardware, debido a que permite una minimización en la utilización de recursos lógicos disponibles en el FPGA, sin sacrificar el desempeño del controlador (a razón de la velocidad de procesamiento), recordando que la solución digital aporta mayor robustez ante el ruido eléctrico, así como una mayor exactitud en los cálculos realizados.

Citando el objetivo general de este trabajo, es posible afirmar que los códigos que describen el funcionamiento del FLC digital aquí diseñado, se optimizaron con respecto a los resultados iniciales, siendo adaptables a cualquier diligencia sin cambios drásticos. Además, el interponer modelos en Verilog y VHDL, adiciona mayor generalidad de acuerdo a la preferencia sintáctica del diseñador. La modularización e independencia de las interfaces del proceso difuso aporta un análisis más exhaustivo del procesamiento individual propiciando que existan códigos que no necesariamente se apliquen a los controladores difusos, sino a cualquier otro ejercicio digital, como en el caso de los módulos aritméticos diseñados.

5.1.2 Conclusiones Específicas

Una arquitectura con interfaces de procesamiento serie, omitiendo las características de la unidad de división, consume varios ciclos de reloj para asumir como finalizado un proceso. La arquitectura con interfaces de procesamiento paralelo no sólo es más adaptable a cualquier aplicación, sino que posee una velocidad de respuesta final mucho mayor que la serie. Las cantidades de CLBs consumidas por ambas entidades (sin división) son muy similares, por lo que se concluye que es mejor plantear de inicio una solución paralela.

La unidad de división adoptada por la interfaz de defuzificación, se debe utilizar invariablemente dependiendo de la velocidad de respuesta del sistema a controlar. Si se trata de un sistema mecánico, por ejemplo un motor, un divisor serie es una solución

aceptable. Si se tratará de controlar sistemas electrónicos, como en el caso de una red *Neurofuzzy*, un divisor implementado bajo la técnica de los corrimientos constantes aportaría la mayor velocidad de respuesta total. La frecuencia de trabajo sustentada por el controlador varía en función de los algoritmos de modelado y de las características del FPGA utilizado.

Por último, las características que cumple el FLC de irrigación diseñado como ejemplo de aplicación, se resumen así:

a. Número de bits del controlador: *8*.
b. Número de entradas: *2*.
c. Número de salidas: *1*.
d. Número de funciones de pertenencia: *5 para la primer variable de entrada, 3 para la segunda variable de entrada y 3 para la variable de salida*.
e. Tipo de fuzificadores: *Triangular, trapezoidal, inclusivas izquierda y derecha. Genéricamente soporta cualquiera, a excepción del gaussiano. Permiten asimetría de pendientes*.
f. Número de Reglas de Inferencia: *15*.
g. Método de Inferencia: *Mamdani (MIN – MAX)*.
h. Método de Defuzificación: *Promedio de pesos*.
i. Ciclos de reloj: *Variables, dependiendo de la arquitectura. En el caso de una topología de procesamiento paralelo con división mediante corrimientos constantes en el defuzificador, se puede ejecutar todo el proceso en un solo ciclo de reloj.*

5.2 Trabajos a Futuro

Los trabajos a futuro están dirigidos hacia las aplicaciones. Por una parte continuar las investigaciones en un campo más ligado a la inteligencia artificial en derivaciones como lo son las redes neuronales y el reconocimiento. La otra variante es aplicar los resultados en FLCs específicos aumentando las generalidades de los diseños aportados, así como implementado controladores híbridos, donde el almacenamiento sea digital y el procesamiento analógico, comparando con realizaciones heterogéneas.

Se propone realizar comparaciones elementales entre Verilog y VHDL, para explotar al máximo las opciones de codificación. Los HDLs se pueden observar como lenguajes de programación, razón de peso para suponer que existen varias formas de codificar que conllevan a la misma solución, pero no necesariamente con la mínima cantidad de recursos lógicos dentro del FPGA. Este apartado es de vital importancia, especialmente porque los resultados pueden tomar connotaciones ajenas como sería la aportación de material docente y la tendencia a diseñar correctamente.

Bibliografía

[1] “Fuzzy Logic: Implementation and Applications”.
Patyra, Mylnek.
Wiley, 1996.

[2] “Fuzzy Hardware: Architectures and Applications”.
Kendel, Abraham.
Kluwer Academic, 1998.

[3] “Fuzzy Logic: Intelligence, Control and Information”.
Yen, John.
Prentice Hall, 1999.

[4] “New Approaches to Fuzzy Modeling and Control: Design and Analysis”.
Margaliot, Michael.
World Scientific, 2000.

[5] “Fuzzy Modeling for Control”.
Babuska, Robert.
Kluwer Academic, 1998.

[6] “Fuzzy Algorithms for Control”.
Verbruggen, H.
Kluwer Academic, 1999.

[7] “Fuzzy Logic and Neurofuzzy Applications”.
Von Altrock, Constantin.
Addison – Wesley, 1995.

[8] “ Neural Fuzzy Systems”.
Ling, Chin – Teng.
Prentice Hall, 1997.

[9] “ Fuzzy Learning and Applications”.
Russo, Marco.
CRC Press, 2001.

[10] “ Fuzzy Logic with Engineering Applications”.
Ross, Timothy.
McGraw – Hill, 1998.

[11] “Fusion of Neural Networks, Fuzzy Systems and Genetic Algorithms”.
Jain, Lakhmi.
CRC Press, 1999.

[12] “Fuzzy Expert System Tools”.
Schneider, Moti.
Wiley, 1997.

[13] “ Parallel VLSI Neural System Design”.
Zhang, David.
Springer, 1999.

[14] “Learning on Silicon : Adaptive VLSI Neural Systems”.
Cauwembergs, Gert.
Kluwer Academic, 1999.

[15] “ Digital Control of Dinamic Systems”.
Franklin, Gene.
Addison – Wesley, 1998.

[16] “ Digital Control Systems, Theory, Hardware, Software”.
Houpis, Constantine.
McGraw – Hill, 1996.

[17] “Parallel Processing in Digital Control”.
Fleming, Peter.
Springer – Verlag, 1996.

[18] “Applications Specific Integrated Circuits”.
Smith, Michael.
Addison – Wesley, 1998.

[19] “Advanced ASIC Chip Synthesis: Using Sinopsis Design Compiler and PrimeTime”.
Bhatnagar, Himanshu.
Kluwer Academic, 1999.

[20] “ Designing with High Performance ASICs”.
Di Giacomo, Joseph.
Prentice Hall, 1997.

[21] “ The VLSI Handbook”.
Chen, Wai - Kai.
CRC Press, 1999.

[22] “Fundamentals of Modern VLSI Devices”.
Taur, Yuan.
Cambridge University Press, 1998.

[23] “Algorithms for VLSI Physical Design Automation”.
Sherwani, N.
Kluwer Academic, 1999.

[24] “Practical Design Using Programmable Logic”.
Pallerin, David.
Prentice Hall, 1996.

[25] “Digital Systems Design and Prototyping Using Field Programmable Logic”.
Zoran, Salcic.
Kluwer Academic, 1998.

[26] “Programmable Logic Handbook”.
Sharma, Ashok.
McGraw – Hill, 1999.

[27] “Programmable Logic PLDs and FPGAs”.
Seals, Richard.
McGraw – Hill, 1997.

[28] “Logic and Computer Design Fundamentals”.
Mano, Morris.
Prentice Hall, 2000.

[29] “Digital Design: Principles and Practices”.
Wakerly, John.
Prentice Hall, 2000.

[30] “Architecture and CAD for Deep – Submicron FPGAs”.
Vaughn, Betz.
Kluwer Academic, 1999.

[31] “The Practical Xilinx Designer Lab Book”.
Van Den Bout, Dave.
Prentice Hall, 1997.

[32] “Digital Designing with Programmable Logic Devices”.
Carter, John.
Prentice Hall, 1997.

[33] “Hardware Description Languages”.
Ghosh, Sumit.
IEEE Press, 2000.

[34] “VHDL Starter’s Guide”.
Yalamanchili, Sudhakar.
Prentice Hall, 1999.

[35] “VHDL: Coding and Logic Synthesis with Synopsis”.
Weng Fook, Lee.
Academic Press, 2000.

[36] “VHDL: Análisis and Modeling of Digital Systems”.
Zainalabedin, Navabi.
McGraw – Hill, 1999.

[37] “VHDL Primer”.
Bhasker, J.
Prentice Hall, 1999.

[38] “The Desgner’s Guide to VHDL”.
Ashenden, Peter.
Morgan Kaufmann Publishers, 2000.

[39] “Real World FPGA Design with Verilog”.
Coffman, Ken.
Prentice Hall, 2000.

[40] “Verilog Quickstart”.
James, Lee.
Kluwer Academic, 1997.

[41] “Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL”.
Ciletti, M.
Prentice Hall, 1999.

[42] “Verilog HDL: A Guide to Digital Design and Synthesis”.
Palnitkar, S.
Prentice Hall, 1996.

[43] “The Verilog hardware Description Language”.
Thomas, D.
Kluwer Academic, 1998.

Referencias en Internet

[44] Sitio Altera.
www.altera.com

[45] Sitio Actel.
www.actel.co

[46] Sitio Xilinx
www.xilinx.com

[47] Tutoriales herramientas CAD y síntesis.
www.ee.yale.edu/~strumpen/ee348/xilinx/foundation.html
www.digital5.ece.tntech.edu/Max/
www.actel.com/docs/29105_0.pdf

[48] Tutoriales VHDL.
www.eej.ulst.ac.uk/tutor.html
www.seas.upenn.edu:8080/~ee201/vhdl/vhdl_primer.html

[49] Tutoriales Verilog.
www.eg.bucknell.edu/~cs320/1995-fall/verilog-manual.html
www.ee.ed.ac.uk/~gerard/Teach/Verilog/manual/index.html

[50] Sitio VHDL.
www.vhdl.org

[51] Sitio Verilog.
www.verilog.com

[52] Cores en VHDL y Verilog.
www.opencores.org

[53] Sitio FPGA Central.
http://cseg.inaoep.mx/fpgacentral/

[54] Capítulo ITAM –Altera.
www.itam.mx/~altera/

[55] Artículo de Referencia.
www.vlsivie.tuwien.ac.at/vanja/eurodac/eurodac.html

[56] Artículo de Referencia.
http://alds.stts.edu/APPNOTE/Fuzzy/SPRA028.PDF

Glosario de Términos

Controlador de Lógica Difusa.
Controlador que utiliza las técnicas de la lógica difusa para realizar su función. En la literatura especializada se les conoce como FLCs (Fuzzy Logic Controllers), cuya implementación puede ser a través de hardware (analógico, digital o híbrido) y/o software.
Básicamente, consta de tres interfaces que conservan la precedencia en la ejecución del prceso: Fuzificador, Inferencia y Defuzificador.

Defuzificador.
El bloque defuzificador genera su respuesta a partir de que la variable lingüística difusa resultante sale de la interfaz de inferencia. La interfaz de defuzificación, será la encargada de generar el escalar que es el valor real que se entregará a la salida del controlador.

FPGA.
Field Programmable Gate Array – Arreglo de Compuertas Programable en Campo. Es una arquitectura avanzada perteneciente a los Circuitos Integrados de Aplicación Específica (ASICs) programables. Consta de grandes cantidades de recursos no comprometidos (módulos lógicos e interconexiones programables de tipo incremental) que pueden ser configurados por un usuario final (programación en campo). Solamente los FPGAs con tecnología SRAM son reconfigurables.

Fuzificador.
La interfaz de fuzificación, asocia a cada entrada su grado de pertenencia a un conjunto difuso. Este bloque toma los valores numéricos, conocidos como valores frágiles (crisp) de las entradas reales, que para el caso del controlador son escalares, y les asigna un grado de pertenencia respecto de una función previamente definida heurísticamente por el experto humano, generando en su salida valores difusos (fuzificados). El universo de los valores se divide en funciones de pertenencia a las cuales se identifica mediante etiquetas representativas de la variable.

HDL.
Hardware Description Language – Lenguaje de Descripción de Hardware. Es un lenguaje utilizado para modelar la operación funcional de una pieza de hardware (circuitos electrónicos y/o sistemas completos), en una forma textual.
En el contexto tecnológico mundial, los HDLs más comunes e importantes para alto modelado son dos:, Verilog (Lógica Verificable - Verify Logic) y VHDL (VHSIC Hardware Description Language, donde el vocablo VHSIC se refiere a Very High Speed Integrated Circuit – Circuito Integrado de Muy Alta Velocidad). Ambos permiten diseñar de acuerdo a ecuaciones, tablas de verdad y diagrama de estados; su sintaxis es particular a cada uno con palabras reservadas para indicar al sintetizador el método de diseño que se elija.

Inferencia.
La interfaz de inferencia consta de dos módulos que interactúan entre sí: la base de reglas difusas y el mecanismo de inferencia. Una base de reglas difusas (Fuzzy Rules Base) consiste en un conjunto de reglas Si – Entonces (en inglés, If - Then) que modelan el funcionamiento del sistema, por lo que el planteamiento correcto de la base de reglas de acuerdo a la experiencia del operador humano, es factor fundamental para el comportamiento satisfactorio del controlador. Por lo mismo, a la base de reglas también se le conoce como Base del Conocimiento.
La máquina o mecanismo de inferencia difusa emplea la información almacenada en la base de reglas difusas para determinar la función de pertenencia de los antecedentes, el grado de activación de las funciones de pertenencia del dominio de entrada, y de esta forma obtener la función de pertenencia de los consecuentes proporcionando una única salida difusa.

Lógica Difusa.
Forma de razonamiento multivaluada que permite a valores intermedios ser definidos entre valores convencionales (0 ó 1). Este tipo de lógica está basado tradicionalmente en reglas que se determinan al azar dependiendo de la experiencia de un operador humano; en vez de definir un evento como 100% falso o verdadero, lo concibe como parcialmente falso o verdadero otorgando a un evento grados de pertenencia, permitiendo así un manejo más concreto y más lineal de situaciones.

Síntesis Lógica.
Es el equivalente a compilar un código escrito en algún HDL, trabajando en herramientas de diseño VLSI. El proceso genera un netlist (archivo de conexiones) que registra la descripción de las celdas lógicas (primitivas) y sus conexiones específicas, que conforman un circuito modelado. El formato de netlist más utilizado es EDIF (Formato de Intercambio de Diseño Electrónico - Electronic Design Interchange Format).

Tecnologías Programables.
Características que cumple la construcción de dispositivos de lógica programable, para permitir la configuración o programación. En el caso de los FPGAs se consideran dos tecnologías principales: SRAM (Bit de RAM estática) para dispositivos reconfigurables y Antifusibles para los no reconfigurables. Para algunos autores, tecnologías programables es todo el contexto del diseño digital sobre FPGAs y CPLDs.

VLSI.
Very Large Scale Integrated – Muy Alta Escala de Integración. Circuitos integrados que contienen entre mil y hasta cien millones de compuertas lógicas. Para algunos autores, VLSI alcanza solamente cien mil compuertas, posteriormente consideran una escala ULSI (Ultra) para los millones.

Anexo A

Documentación de Códigos

Los códigos completos están disponibles en el **CD** suplementario a este trabajo de tesis. Los siguientes se imprimen con la intención de mostrar la forma sintáctica en que aparecen documentados.

fuz_TAs1.v

```
// Unidad Fuzificadora de procesamiento serie para la variable de entrada Temperatura del
// Aire. 5 funciones de pertenencia, bajo la estrategia de cálculo 1. Código en Verilog

module fuz_TAs1 (grado1, grado2, grado3, grado4, grado5, temp_aire);
input [7:0] temp_aire;
output [7:0] grado1, grado2, grado3, grado4, grado5;
reg [7:0] a [0:4];
reg [7:0] b [0:4];
reg [7:0] c [0:4];
reg [3:0] pend [0:4];
reg [1:0] tipo [0:4];
reg [7:0] grado1, grado2, grado3, grado4, grado5;
reg temp;
reg [7:0] grad_mem[0:4];

always @(temp_aire)      // Diseño asíncrono, no tiene reloj, cambia con la entrada.
Begin:fuzi_serie
integer i;
a[0]=0;                  // Datos en memoria.
a[1]=30;
a[2]=60;
a[3]=150;                //       *                     *****
a[4]=190;                //      *|*                    |   |*
b[0]=20;                 //     * | *                   |   | *
b[1]=60;                 //    *  |  *                  |   |  *
b[2]=120;                //   *   |   *                 |   |   *
b[3]=180;                //   a   b   c                 a   b   c
b[4]=210;
c[0]=50;
c[1]=90;
c[2]=180;
c[3]=210;
c[4]=255;
pend[0]=255/30;          // Declaración de las pendientes positiva: M_máximo/(b-a)
pend[1]=255/30;          //                                 negativa: M_máximo/(c-b)
pend[2]=255/60;
pend[3]=255/30;
pend[4]=255/30;
tipo[0]=0;               // Declaración del tipo de fuzificador:
tipo[1]=1;               // Tipo 0: Inclusiva derecha.
tipo[2]=1;               // Tipo 1: Triangular.
tipo[3]=1;               // Tipo 2: Inclusiva Izquierda.
tipo[4]=2;

for (i=0; i<5; i=i+1)    // Comienza ciclo de obtención de datos, uno a uno de memoria.
begin
if (temp_aire<b[i])      // Declaración de una variable Temp., para diferenciar entre las
temp=1;                  // inclusivas de los tipos 0 y 2.
else
temp=0;
```

```
if (tipo[i]==0 & temp==1)   // Eliminación del cálculo de pendiente fuera de la
grad_mem[i] = 255;                                // Inclusiva de los tipos 0 y 2
else if (tipo[i]==2 & temp==0)
grad_mem[i]=255;
else if (temp_aire<=a[i] || temp_aire>c[i])      // Verifica pertenencia a la función
grad_mem[i]=0;                                    // analizada, sino, grado=0.
else
begin
if (temp_aire<b[i])
grad_mem[i]=pend[i]*(temp_aire - a[i]);         // Pendiente positiva.
else
grad_mem[i]= pend[i]*(c[i] - temp_aire);        // Pendiente negativa.
end
end
grado1=grad_mem[0];                               // El muestro final de datos será
grado2=grad_mem[1];                               // paralelo.
grado3=grad_mem[2];
grado4=grad_mem[3];
grado5=grad_mem[4];

end
endmodule
```

fuz_TAs3.vhd

```
-- Unidad Fuzificadora de procesamiento serie para la variable de entrada Temperatura del
-- Aire. Memoria de datos externa al bloque, 5 funciones de pertenencia, bajo la
-- estrategia de cálculo 2. Codificada en VHDL.

Library IEEE;
Use IEEE.Std_Logic_1164.All;
Use IEEE.Std_Logic_Arith.All;

Entity Fuzser1 Is
Port(temp._aire : In._aire Std_Logic_Vector(7 downto 0);
     b : In  Std_Logic_Vector(7 downto 0);
     tipo   : In  Std_Logic_Vector(1 downto 0);
     pend  : In  Std_Logic_Vector(3 downto 0);
     a  : In  Std_Logic_Vector(7 downto 0);
     grado : Out Std_Logic_Vector(7 downto 0));
End Entity FUZZIFIER;

Architecture Arch_Fuzzifier Of FUZZIFIER Is
Begin
Process(Input,Centre,Tipe,Pend,Width)
   Variable Delta : Unsigned(7 downto 0);
   Variable Temp : Std_Logic;
   Constant FF    : Unsigned(7 downto 0) := (Others => '1');
   Begin
   IF (temp._aire<b) Then
   temp := '1';                                        -- Verifica si se trata de una
   ELSE                                                -- pendiente positiva o negativa
   temp := '0';
   END IF;

   IF (tipo="00" And temp='1')  Then Output <= (Others => '1');
   ELSIF (tipo="10" And temp='0')  Then Output <= (Others => '1');
   ELSE
   delta := Conv_Unsigned(Abs(Signed(b)-Signed(temp_aire)),8);     -- Determina delta
   IF (Unsigned(delta)<Unsigned(b-a)) Then
   Output <= Conv_Std_Logic_Vector(FF-(Unsigned(Delta)*Unsigned(Pend)),8);  -- Grado
   ELSE
   Output <= (Others => '0');
   END IF;
   END IF;
   End Process;
End Arch_Fuzzifier;
```

Anexo B

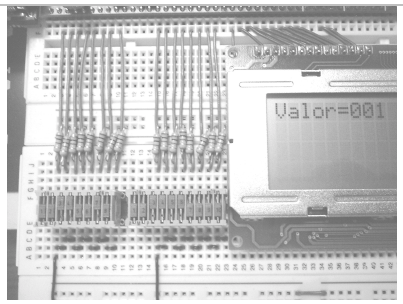
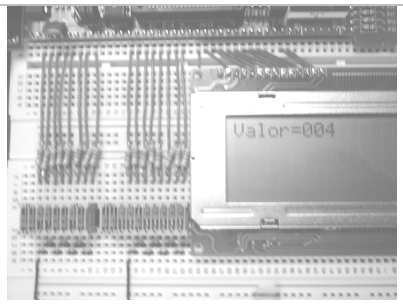
Comprobación Experimental

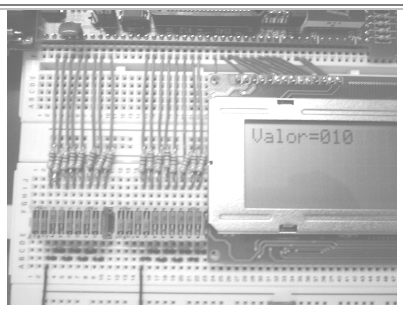
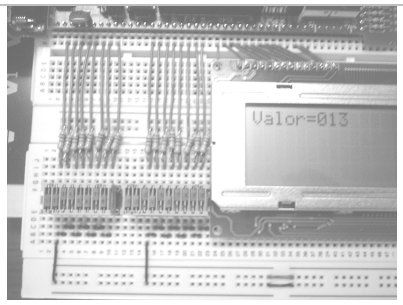
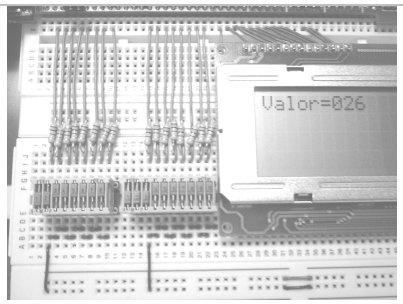
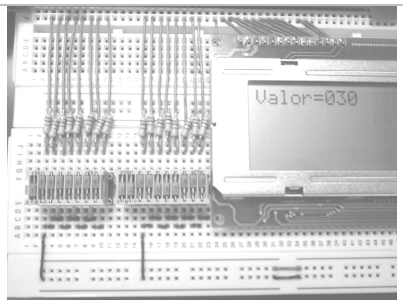
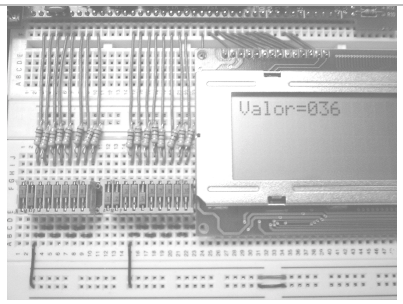
En la tarjeta de desarrollo con las interfaces de entrada y salida conectadas (**figura 4.5, Capítulo 4**), el dipswitch de la izquierda se utiliza para introducir los datos de la variable *Humedad de la Tierra*. El polo extremo derecho es el bit menos significativo.

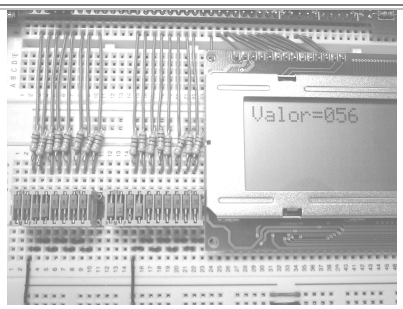
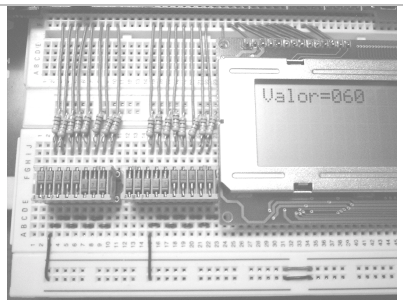
El dipswitch de la derecha es el propio para la variable *Temperatura del Aire*. Éste tiene 10 polos, de los cuales los dos de la extrema izquierda no se utilizan, por lo tanto en la extrema izquierda comienza el dato binario con el bit menos significativo. Cuando un interruptor está en ON se introduce un *0*, caso contrario un *1*.

Para comprobar los resultados, se utilizaron los mismos datos anotados en la **tabla 4.1** del **Capítulo 4**, los cuales se muestran nuevamente en la **tabla B.1**.

Tabla B.1: **Valores asumidos para la simulación del FLC de irrigación diseñado.**

Temp Aire	Hum Tierra	Valor Real por HW
15	205	
32	205	

64	205	
80	205	
128	192	
60	128	
32	64	

64	48	
224	7	

Anexo C

Diagramas Esquemáticos

Las diferentes arquitecturas diseñadas conservan la modularidad de sus componentes. A través de los HDLs se sintetizó una Macro que se interconectó como módulo independiente para conformar la estructura completa.

A continuación se adicionan los diagramas esquemáticos de las versiones paralelas, los cuales al tamaño impreso no son completamente legibles; sin embargo, están contenidos en la carpeta *FLC* del **CD** como parte del proyecto completo. Todos los bloques visibles fueron realizados mediante un HDL a excepción del oscilador interno del FPGA (*OSC4*) y de algunos contadores y lógica de compuertas, que interactúan de manera híbrida en la interfaz de la pantalla de LCD.

El esquemático **FUZZY_C1.SCH** (**figura C.1**) muestra la arquitectura paralela con división de restas sucesivas en el defuzificador, éste a su vez fue el diseño que se implementó para validar el trabajo de tesis. **FUZZY_C2.SCH** (**figura C.2**), es el diagrama de la topología paralela con división de corrimientos constantes en el defuzificador. El bus *SAL_CNTROL* entrega los datos binarios resultantes del proceso difuso.

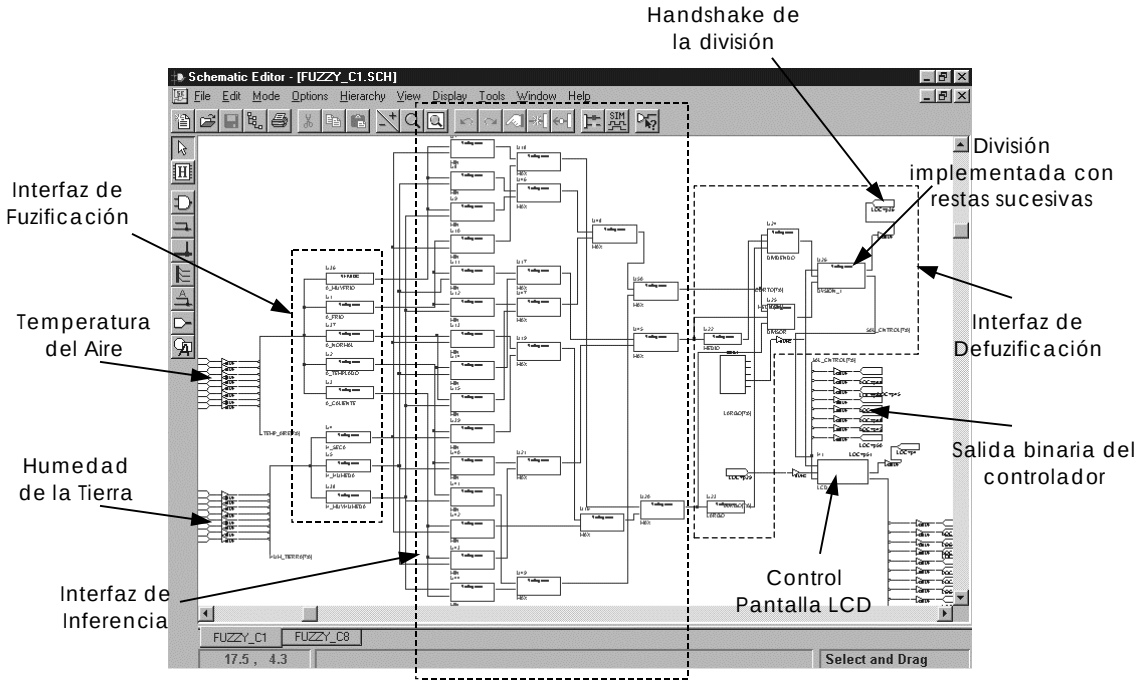


Figura C.1: **Diagrama esquemático del FLC con división de restas sucesivas en el defuzificador.**

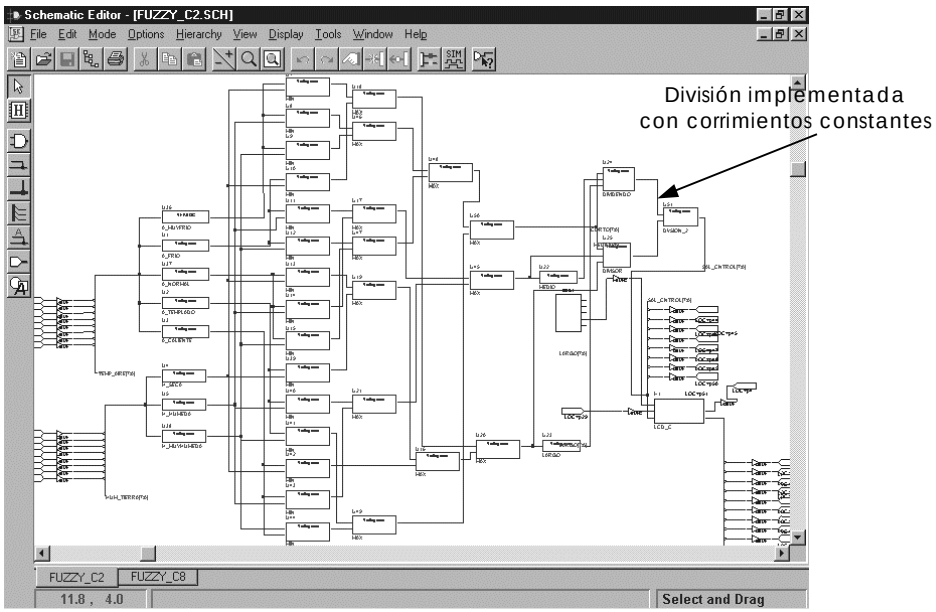


Figura C.2: **Diagrama esquemático del FLC con división de corrimientos constantes en el defuzificador.**